TR-I-0059

DyNet, a Fast Program for Learning in Neural Networks

DyNet, ニューラルネットワークにおける高速学習プログラム

P. Haffner

パトリック ハフナー

1988.11

内容梗概

本テクニカルレポートは、別報TR-I-0058"Fast Back-Propagation Learning Methods for Neural Networks in Speech"に記述されているプログラムのマニュアルである。

ATR Interpreting Telephony Research Laboratories
ATR 自動翻訳電話研究所

DyNet, a Fast Program for Learning in Neural Networks

Patrick Haffner

October 1988, updated November 25, 1988

Contents

1 Tutorial 3

1.1 Introduction 3

1.2 Preparing data 3

1.2.1 Sample data 3

1.2.2 Network architecture 3

1.2.3 Sigmoid function 4

1.2.4 Parameters 4

1.3 Fundamental controls 4

1.3.1 When to stop 4

1.3.2 Controlling learning rate 4

1.3.3 Convergence vs. generalization 5

1.4 Saving more computing time 5

1.5 Tracing the program 5

1.6 Recognition performance 6

2 Reference Manual 7

2.1 Introduction 7

2.2 The learning Algorithm: an overview 7

2.3 File organization 8

2.4 Network Architecture 8

2.4.1 The Network as a list of connections 9

2.4.2 The Network as a list of units 9

2.4.3 The Network as a list of masks 10

2.4.4 Sequencing the network 11

2.4.5 The *initnet* program 11

2.5 The sample Data 13

2.5.1 The ASCII property file 13

2.5.2 The BINARY sample file 14

2.5.3 The iof states 14

2.5.4 Samples as Words 15

2.5.5 Preparing samples 15

2.5.6 An example: the **XOR** training samples 15

2.6 The Parameters 16

2.6.1 Generalities 16

2.6.2 Flags 17
2.6.3 Dimension numbers 17
2.6.4 Parameters concerning dynamic tables 18
2.7 Control and trace of dynamic values 18
2.7.1 Weight 19
2.7.2 DeltaW 19
2.7.3 dEdw 19
2.7.4 Epsilon 20
2.7.5 Alpha 20
2.7.6 cos 20
2.7.7 Update 21
2.7.8 Error 21
2.7.9 Recognition rate 21

3 Parallelism and Connectionism **23**
3.1 A brief history 23
3.1.1 Concurrency on samples 23
3.1.2 Concurrency on units or sets of units 24
3.1.3 Parallel computation over the set of connections 24
3.2 Our choices 24
3.3 References 24

4 Technical Documentation **25**
4.1 Code 25
4.2 Data 25
4.3 Description of each part of the program 26
4.3.1 dynet.c 26
4.3.2 net.c 26
4.3.3 sample.c 26
4.3.4 pass.c 27
4.3.5 optimize.c 27
4.3.6 weight.c 28
4.3.7 display.c 28
4.3.8 param.c 28

Chapter 1

Tutorial

1.1 Introduction

DyNet is mostly aimed at background tasks; the user needs only to type:

DyNet task

DyNet first build the Network from a netfile, then it loads a set of training samples. It then performs any number of learning pass over the whole sample set (Epoch) . The user may trace the (expected) reduction of the output error or the evolution of many other dynamic parameters. Learning may be stopped and started again with previous connection weights. A special mode is available for recognition only.

1.2 Preparing data

1.2.1 Sample data

Sample data is in the sample directory. There are an ASCII description file and binary sample files. If samples are coded under Waibel's ".sample" format, one needs only to create the sample directory and the *initsample* program will convert the Waibel format into *DyNet* format.

initsample *(net file) (Waibel sample file) (* DyNet *sample directory)*

The program will ask how to code the binary samples through the standard input. With TDNNs, adding to the command line "¡ spara" gives the good coding parameters. After that, the sample directory should include 3 files: ioframes, sampin and sampout.

1.2.2 Network architecture

initnet converts the Waibel ASCII network format (*task*/net) into the precompiled and binary *DyNet* network format (*task*/bnet):

initnet task

1.2.3 Sigmoid function

sigmoid creates a binary description of the sigmoid function in *task*/sigmoid:

sigmoid task

One modifies the sigmoid by changing one of the three functions heading the sigmoid.c program.

- *sigmoid* gives the standard.
- *sigmoid1* is used to compute the derivative.
- *sigmoido* is special for the output units.

You should modify *sigmoido* with great care, to keep real and desired output matching correctly.

1.2.4 Parameters

The file *task*/para gives all the parameters. Each is identified by two letters we shall write in brackets. The important parameters should be at the head of this file.

- The step size (ei)
- The parameters concerning the weight updating period (u...)
- The number of epochs (ne)

1.3 Fundamental controls

1.3.1 When to stop

The user has three ways to stop learning:

1. Impose a maximum number of epochs (ne).
2. Impose a minimum Mean Square Error (on). When the error goes below this value, the programs stops.
3. Use a system interruption. If the catch-error flag(vc) is set, the program will save everything in the result directory before stopping.

1.3.2 Controlling learning rate

It is mainly controlled through the step size *epsilon*. Its initial value (ei) has to be carefully chosen, in a trade-off between speed and overshooting. The dEdw overshooting control (ge) automatically resizes epsilon when it is too large. ge =1 is recommended. (This program also provides an algorithm that scales epsilon automatically, but its use is not completely safe) .

1.3.3 Convergence vs. generalization

It is very difficult to achieve both good convergence on training samples and good generalization on test samples. The choice of many parameterers leads to a kind of trade-off.

Choosing an appropriate weight updating period is perhaps the one way to improve both.

- Set the initial value (ui) to the number of classes N in this problem. (we assume that N consecutive training samples represent the N different classes in our task)
- The evolution parameter (ue) corresponds to the increment in the weight updating period from one epoch to the next one. Set it to a small value.
- This period must not grow forever. Set a maximal value (um) which is a multiple of N.

How to improve convergence at the expense of poorer generalization:

- Add a small linear function (+ 0.01*x) to the sigmoid.
- Use the McClelland new error (vl =1 or 2) .

How to improve generalization at the expense of worse convergence:

- Use the Rumelhart weight decay (we) .
- Increase the momentum control parameter (ae) .

1.4 Saving more computing time

We present here two "tricks" to improve speed. First, it is useless the process the samples which are already learnt. If their output error is below some minimum value, they may be skipped. See the subsection of the reference manual describing the error to do that.

On Parallel computers, when there are only a few units in a later, some processors may remain idle as they have no unit to process. To avoid this, it is possible to replicate the network n times, using the "-g n" option in the initnet program. The network is now presented n samples at a time, and process them concurrently. The weight updating frequency and the sample file are rescaled automatically, so you do not have to worry. However, as the size of the network is multiplied by n, so are the memory requirements to code for the network.

1.5 Tracing the program

DyNet prints important information on the standard output, in 5 columns:

1. Number of epochs.

2. Number of samples actually learned.
3. Mean square output error.
4. Recognition rate (if it possible to compute it) .
5. Elapsed CPU time (in seconds).

Other useful outputs are found in *task*:

display This directory keeps a statistical record of the dynamic tables of values:

1. Number of epochs.
2. Average value.
3. List of Value (Index) . Generally only the maximum and the minimum values are displayed.

result In this directory, *DyNet* saves the important tables of dynamic values such as the weights.

1.6 Recognition performance

During the learning phase, weights are regularly saved into the weightXX files, where XX is the number of epochs. Because of potential overlearning, the last saved weights do not necessarily yield the best performance on testing data.

It is very simple to use *DyNet* for recognition performance only. First some testing data is needed. This has to be prepared in the *task*/test directory (for instance), following the same procedure as for sample data. Then run *DyNet*:

DyNet task *-vr 1 -fs test* < *weightXX*

"-vr 1" means you set recognition mode and "-vs test" means your samples should be obtained from *task*/test instead of *task*/sample . WeightXX are the weights that should be used.

The file *task*/result.r/reco1 may be used to get the confusion matrix.

Chapter 2

Reference Manual

2.1 Introduction

DyNet is an non interactive simulation program running on UNIX computers and super-computers supporting C. When given a neural network architecture and a set of learning samples, it iteratively optimizes the connection weights for good classification of the training samples. This program enjoys its full speed on parallel and vectorized architectures, but performance is still fair on sequential computers. The network architecture must be non-recursive and the learning rule must include a forward and a backward pass. Many different options are available to the user. Parameter tuning has been reduced as much as possible, and the program provides good default values and optimized scaling procedures.

2.2 The learning Algorithm: an overview

We give here a brief description of the learning algorithm. What we call a "Neural Network" is merely a set of very simple computing units. Each unit performs a weighted sum of the activations of its input units and computes its own activation by passing this sum through a non linear function. Data is entered in the network through the activations of the input units. After a propagation through the network, the activations of the output units represent the result. From a set of training samples, the network has to learn to associate, with minimum error, some input to its corresponding desired output.

The network is non recursive: a unit does not receive any input from itself, even through some other units. It is then possible to define a strict sequential order in which the activations are computed.

After setting the activations of the input units and the output unit to their desired values, learning includes 4 phases:

1. Forward propagation The propagation through the network takes place as described above. The user may choose the non linear function.

2. **Error computation** There are many ways to compute the error, among them, the Mean Square Error between actual and desired output activation is widely used.

3. **Backward propagation** It involves a sum over the output units, whose implementation details are given by the learning rule.

4. **Weight Updating** This is the most user-dependent part of the algorithm. The program will try its best to tune the parameters to their optimal laerning speed value. of learning speed.

This algorithm has been implemented in a program that needs 3 input files:

1. One that describes the Architecture of the network.

2. One that contains the input and corresponding output samples for training.

3. Onr that gives gives the useful parameters.

The output is mostly a file containing the weights (real numbers) associated with each connection in the network.

2.3 File organization

DyNet finds everything it needs in the *task* directory:

bnet The binary file describing the connectionist architecture used for *task* (generated by *initnet*) .

sample The directory containing the training samples (generated by *initsample*) .

sigmoid Binary description of the sigmoid function and its derivative (generated by *sigmoid*) .

para Text file describing the default parameters.

result0 Directory generated by a previous run of *DyNet* and containing values to be used again (for instance weights) .

It is also possible to modify parameters in the command line of *DyNet.*

2.4 Network Architecture

We describe in more details the network architecture and how it is coded.

2.4.1 The Network as a list of connections

As each unit has been given a number, a very simple way to represent the network is to list of connections with the 3 following entries:

1. Input unit.
2. Output unit.
3. Connection Weight.

An ASCII file format has been created to code such an architecture and has been widely used in *ATR Interpreting Telephony Research Laboratories* . The files appear as task.net. For instance, the **XOR** network description file is:

```
NETWORK FILE:
6 units
3 inputs
0 1 2
1 outputs
5
9 conns
0 3
1 3
2 3
0 4
1 4
2 4
0 5
3 5
4 5
```

Learning with this architecture is very easy. In the forward pass, for each connection:

- Take the input unit activation.
- Multiply it by the connection weight.
- Add the result to the output unit sum.

However, it is very difficult to work on variables that are proper to a unit. In our program, as we wanted to be able to define learning as a property local to a unit, we had to work on a more suitable architecture.

2.4.2 The Network as a list of units

The description is here more complex: We call mask a vector of floating point numbers (sometimes a vector of pointers to floating point numbers). Each unit needs 4 masks:

- *Forward Propagation*

  - *Input Unit Mask*
  - *Input Weight Mask*
- *Backward Propagation*
  - *Output Unit Mask*
  - *Output Weight Mask*

Learning formalizes nicely if we create a procedure that works for both forward and backward passes.

1. For each unit, it computes

   $x =$ dot-product *(UnitMask, WeightMask)*

2. Then it applies to x a function that depends on:

   The pass: Forward or Backward In the case of the forward pass, it is a non-linear sigmoid function.

   The learning procedure In the case of the *Back-Propagation* procedure, it computes the Error derivative dE/dy =x.

   The unit It is possible to imagine units with different sigmoid functions.

This architecture is well suited for hardware implementation. Moreover, it makes weight updating algorithms which are local to each unit very easy to implement. The masks may use a large amount of memory, but it is possible to reduce the memory requirements having the units share the same masks when possible.

2.4.3 The Network as a list of masks

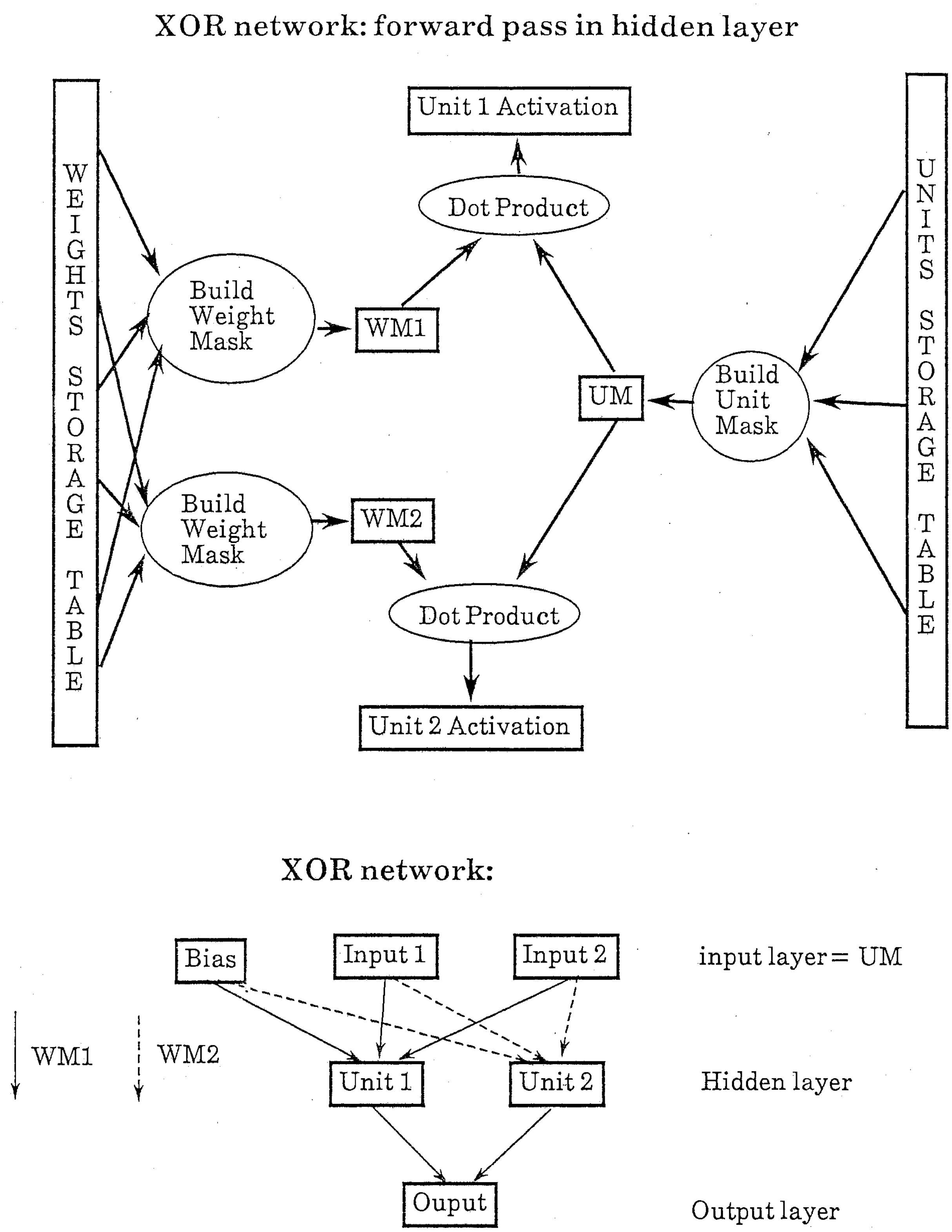
In most architectures we use here, a mask may be shared by several units.

- Generally, the network may be structured in *layers.* In this case, many units of the layer n may share the same *input unit mask*: a subset of the units in layer n-1. The same is true for *output unit masks.*
- In T.D.N.N. , 2 units may have exactly the same *input weight mask* or *output weight mask.*

To build our new architecture, we introduce the notion of *group-mask.* There are four sets of those, one for each mask category. For instance, an input unit *group-mask* includes:

- The input unit mask itself.
- The list of units sharing this mask.

This way to code the network is not only much more memory efficient, it also allows substantial reductions in computation time:



XOR network: forward pass in hidden layer

XOR network:

Fig 1: Preparing vectors for the dot-product:
An example with the XOR network

- Here, as the same number belongs to several masks, a mask must be a vector of *pointers* to the actual number. For the sake of computing speed,a vector of numbers would be much better, as some computers have a very fast vectorized routine to perform the *dot-product* on such vectors. To achieve this, we would have to rebuild all the masks before each iteration. With one mask per unit, the cost would be prohibitive, but here we only have to apply this routine to each *group-mask*.
- Input weight *group-mask* are very useful to update weights in **T.D.N.N.** . The weight updating procedure, which can be time-consuming, has only to be applied once for each input weight mask.

2.4.4 Sequencing the network

To compute the propagation of the activations through the network, we have to sort the units into layers. We define recursively layer number n as the largest unit subset verifying the condition:

A unit in layer n only depends on units in layers 0 to n-1.

It is generally assumed that units in a same layer have the same properties. It is important to set unit parameters according to their layers.

2.4.5 The *initnet* program

The initnet program initializes the file that describes the network architecture.

initnet task

1. The input *task/net* is an ASCII file as described in Section 2.4.1 . *initnet* ignores the input and output units, for these will be described with the sample file. The program first builds the list of units with their associated masks, then groups these masks in *group-mask* .
2. The output *task/bnet* is an binary file as described in Section 2.4.3 . It contains:
    - A list of units.
    - A list of input unit *group-masks*.
    - A list of output unit *group-masks*.
    - A list of input weight *group-masks*.
    - A list of output weight *group-masks*.
    - A list of layers, with their corresponding sequence of units

   initnet gives some information about the way it compiles the network on the standard output. Here, for instance, is what we get with the **XOR** network. An example of the forward pass in the hidden layer using this architecture is given in Fig.1.

```
6 units, 3 inputs:

1 outputs:

9 conns
9 real weights

LAYER 1: # description of the hidden layer
  input unit groups
  0 list(2): 3 4 # 2 units in the first layer
    mask(3): 0 1 2 # they share the same input unit mask:
# 0(bias), 1 and 2.

  output unit groups
  0 list(2): 3 4
    mask(1): 5 # and the same output units mask

  input weights groups
  0 list(1): 3 # but the weight mask are different
    mask(3): 0 1 2

  1 list(1): 4
    mask(3): 3 4 5

  output weights groups
  0 list(1): 3
    mask(1): 7

  1 list(1): 4
    mask(1): 8



LAYER 2: # description of the ouptut layer
  input unit groups
  1 list(1): 5
    mask(3): 0 3 4

  output unit groups  # no output

  input weights groups
  2 list(1): 5
    mask(3): 6 7 8
    output weights groups
```

After the bnet file has been created, the *DyNet* program may use it without having to recompile each time the network. Note that you may not be able to transfer the task/bnet file between computers with different architectures.

2.5 The sample Data

As the network has to pass through it many times, the whole learning set must be stored in Random Access Memory from the beginning. On big learning tasks, we have to be very careful about the way we code our sample data.

- The code should be as compact as possible to avoid memory page faults.
- The initial loading phase has to be very fast to make it possible for the user to have his first results within a very short time. (It is always very useful to trace the first iterations of a learning run) .
- In staged learning, we have to save and load units belonging to hidden layers. For instance, to freeze the connections between the input layer and the first hidden layer amounts to considering this first hidden layer an input layer.

To make all these manipulations easy, we have created the general concept of input-output frame or **iof**. An **iof** includes an ASCII property file and a BINARY sample file.

2.5.1 The ASCII property file

This files contains the formal description of our **iof** with the following entries:

Name (string) : gives the path to the corresponding BINARY file in the *task* directory.

Way (character) : two ways: (i)nput and (o)utput.

State (character) : three states: (A)ctive, (I)nhibited and (L)oading. See Section 2.5.3

.

Type (integer) : three types: Float(0), Char(1) and Word(2) . See Section 2.5.2 .

Number of words (integer) : if the type is word. See Section 2.5.4 .

Word size (integer) : number of units to code for a word.

Num (integer) : number of units used by the **iof**. Num = Number of words x Word size.

Ratio (Float) : resizing ratio to convert the characters back to floating point numbers.

List List of the units used by this **iof**.

All these descriptions are gathered in the task/ioframes file. This file is headed by two numbers:

- The total number of samples.
- The number of iofs.

2.5.2 The BINARY sample file

We have created 3 formats for sample data, depending on the task. They are all stored in binary sample files rather than ASCII files.

- Floating point numbers representing the unit activations.
- Characters representing these activations sampled at 8 bits. The maximum activation is rescaled to 127.
- Integers representing words. Here input or output units have only 0 or 1 as activations. A word is a set of n units. Its value is k when the kth activation is 1 and all the others are 0.

2.5.3 The iof states

The initial state of each **iof** is given by the **iof** description file, but during the learning run, the user may change it through the **iof** message file. When an **iof** is inhibited, it is considered by the program to be non-existent. But the role of an active **iof** varies whether its way is input or output.

Input When an input **iof** is Active, the activation of its units is loaded from outside. Even if the network architecture specifies that a unit belonging to this **iof** has some input units, their activation are not taken into account and the weights linking this unit to its input units are frozen.

Output Units belonging to an active output **iof** file are quite ordinary and have only 2 characteristics: the difference between their desired and their actual activation is added to their local error, and they intervene in the computation of the M.S.E. output Error.

Here are the operations the user may perform:

activate The message 'a' activates an inhibited **iof**. The **iof** storage table should not be empty.

inhibit The message 'i' inhibits an active **iof**. The **iof** storage table is kept in memory.

load The message 'l' brings an inhibited **iof** to the (L)oad state. During one epoch, the program loads the activation from the network to the **iof** storage table. Afterwards, the **iof** automatically comes to the active state. For an input **iof** describing some hidden layer, loading this **iof** leads to freezing the input connections of this layer. Loading is impossible when the **iof** is already active.

save The message 's' creates a saving buffer in only one epoch. Through this buffer, activations for each sample are saved in the corresponding binary sample file.

2.5.4 Samples as Words

Unlike floats and characters, words in output **iof** may be considered recognised classes.

2.5.5 Preparing samples

We will assume that all the files concerning the training data are in the task/sample directory and those concerning test data are in the task/test directory. To prepare task/sample, there are 2 operations:

1. Prepare the **iof** description file task/sample/ioframes.
2. Create each corresponding **iof** binary sample file. **Warning: for each subset of samples to be as representative of the problem as possible, samples are to be mixed carefully.**

These operations are automatically performed when an ASCII file corresponding to the T.D.N.N. format lies in task/sample/ascii. The program to run it is:

initsample *net (Waibel sample file) (*DyNet *sample file)*

Through the standard input, the program asks the coding types for the input **iof** (sampin) and output **iof** (sampout).

2.5.6 An example: the XOR training samples

Here is the **iof** description file for the **XOR** training samples

```
4
3

 name: sampin
way: i
state: A
type: 0
number_of_words: 1
word_size: 3
ratio: 0.007874
list:  0 1 2

 name: hidden
way: i
state: I
type: 0
number_of_words: 1
```

```
word_size: 2
ratio: 0.007874
list:  3 4

 name: sampout
way: o
state: A
type: 2
number_of_words: 1
word_size: 1
ratio: 0.007874
list:  5
```

The **XOR** description file contains three **iofs**:

Input samples This **iof** gives the activations of the input units of the network (number 0, 1 and 2). They are coded with floating numbers. The state is active.

Hidden layer This **iof** allows us to work on the activations of the hidden layer. It is normally Inhibited, but may be activated if we want to freeze the input weights. The type must be floating point.

Output samples This **iof** gives the desired output activations. As **XOR** is a classification problem, we may use the word type for output units.

2.6 The Parameters

It is possible to change many different parameters in this program. Most of them have default values in the parameter file.

2.6.1 Generalities

Each modifiable parameter is identified by two letters and there are two ways to control them. For instance, we set the number of epochs (ne) to 100, and the name of the sample directory (fs) to task/test:

1. Through the parameter file, with the lines

   ne 100
   fs task/test

2. through the command line.

   DyNet *task -ne 100 -fs task/test*

The second method, which should be only used for very important parameters, has priority over the first. There are three main categories of parameters: flags, dimension numbers and dynamic table control parameters.

2.6.2 Flags

Here is a list of the most important flags. For each one, we describe what happens when they are ON (They are ON when equal to 1 and OFF when equal to 0) .

word input (vi) Input units must code for words and have only the activations 0 and 1. This mode saves a lot of computation time when only a small number of units are turned to 1 at the same time, for it only adds the weights corresponding to activated input units, and do not compute anything about non activated input unit.

linear output (vo) When the output has to be real numbers, it is recommended that the sigmoid function not be used for the output units.

catch error (vc) The program catches interruptions and errors, saves all the dynamic tables and stops.

synchronous updates (vy) All the units have to update their weights at the same time.

random seed (vs) May take any integer value, which correspond to the random seed used to set initial weights.

recognition mode (vr) The program only computes the recognition rate on the weights that are given as standard input. No learning is done.

New Error (vl) McClelland's new error is used instead of Mean Square Error. This new error prevents zero learning but may result in overlearning the training samples.

renormalize (vh) Input activations are resized to have -1 as minimum value and +1 as maximum value.

2.6.3 Dimension numbers

Here are the most important dimensions used in the network. A few of them are modifiable by the user (taht marked with an asterisk).

number of connections (nc) Total number of connections

number of real weights (nw) Number of independent weights.

number of units (nu)

***maximum number of samples (nm)**

***number of epochs (ne)**

***number of the unit to trace (nn)**

iteration grain (ng)

2.6.4 Parameters concerning dynamic tables

Each dynamic table comes with a set of control parameters and display parameters. The dynamic table is referenced by a character we shall call here \$. We give here the names of its control parameters. Their exact role depend on the table.

Initial value (\$i)

Evolution parameter (\$e)

Maximum value (\$m)

Minimum value (\$n)

The display parameters command the nature of the trace we get in the task/result/"table" and task/display/"table" files.

result saving period (\$r) The number of epochs between two complete saves of the whole table in task/result/"table". 0 is the default value and inhibits saves.

display period (\$d) The file task/display/"table" traces important information about the values of the table as a function of the epoch. This parameter gives the epoch granularity.

Display mode (\$o) This determines the nature of the information you want to display. Many modes are available.

0 (Epoch) (Average)

1 (Epoch) (Average) (maximum) (minimum)

2 (Epoch) (Average) (list of values)

11 (Epoch) (Average) (maximum class-average) (minimum)

12 (Epoch) (Average) (list of class-average)

When a floating value is followed by an integer number between brackets, the latter gives the index of this value in the table.

Which Average? (\$n) **0** Root Mean Square (RMS) .

1 Normal Mean.

2.7 Control and trace of dynamic values

Each category of dynamic values is stored in a specific table. This table goes with several parameters, which have been described in the previous section. Each value in the table is associated with a integer number we call class. This number may have many different meanings. We call class-average the average value of all the numbers belonging to one class. We give here a detailed description of each class of parameters. The reference character correpond to the first of the two characters used to index parameters.

2.7.1 Weight

What : weights.

Indexed over : physical connections.

Reference character : w

Initial value : Correspond to the standard deviation of the random gaussian function used to initialize weights.

Evolution parameter : Weight decay for Rumelhart algorithm 1 or 2. In both cases, we recommand 1.0. A smaller value speeds up learning and a larger one retards it.

2.7.2 DeltaW

What : weight variations.

Indexed over : physical connections.

Initial value : 0

Reference character : d

Maximum value : no.

Minimum value : no.

Class : Layer

2.7.3 dEdw

What : Error/weights derivatives.

Indexed over : physical connections.

Reference character : g

Initial value : 0

Evolution Parameter : Overshooting control parameter (1.0) . A larger value provides better control of error overshooting at the expense of learning speed.

Maximum value : no.

Minimum value : no.

Class : Layer.

2.7.4 Epsilon

The step size stays constant during learning.

What : step size.

Indexed over : input weights groups.

Reference character : e

Initial value : initial step size.

Evolution Parameter : no.

Maximum value : no.

Minimum value : no.

Class : layer.

2.7.5 Alpha

The recommended values are placed in brackets. The maximum value should be less than 1. The minimum value should be less than 0.9.

What : momentum.

Indexed over : input weight groups.

Reference character : a

Initial value : no..

Evolution Parameter : Momentum control parameter (1.0)

Maximum value : yes (0.99)

Minimum value : yes (0.5)

Class : leyer

2.7.6 cos

What : Cosine of the angle between dEdw vector and deltaW vector.

Indexed over : input weight groups.

Reference character : c

Initial value : 0

Evolution Parameter : Cos momentum.

Maximum value : 1.0

Minimum value : -1.0

Class : layer.

2.7.7 Update

What : Number of iterations between two weights update.

Indexed over : input weights groups.

Reference character : u

Initial value : value during the first epoch.

Evolution Parameter : Increment from one epoch to the next one.

Maximum value : yes.

Minimum value : yes.

Class : layer.

2.7.8 Error

What : Mean Square Error.

Indexed over : Training samples.

Reference character : e

Initial value : 1.0

Maximum value : Learn a sample only if its output error is over this maximum.

Minimum value : Stop learning if average Error goes below this minimum.

Evolution Parameter : Suppose a sample has yielded an error of 0.002 and that the maximum value is 0.01. *DyNet* is going to skip the backward phase for this sample. During the next epochs, *DyNet* is going to skip both forward and backward phases for this sample until its error reaches the maximum value. In the absence of a forward phase, the error is interpolated from the error during the previous epoch by adding the evolution parameter. If the latter is 0.001, it will take 9 epochs for the 0.002 error to reach 0.01 again. This means that the sample is considered sufficiently well learned so that you can forget it during 9 epochs.

Class : no.

2.7.9 Recognition rate

There may be several recognition rate tables.

What : recognition rate for each output **iof** whose type is word.

Indexed over : Training samples.

Reference character : r

Initial value : no.

Evolution **Parameter** : no.

Maximum value : no.

Minimum value : no.

Class : Sample class.

Chapter 3

Parallelism and Connectionism

This chapter will describes how to implement a Neural Network simulator on a parallel computer. We define two kind of parallelism:

- Concurrency is a coarse grain parallelism that allows different processes to run with different instructions. In Neural Networks we can concurrently process different units or different training samples.

- Vectorized computing is a fine grain parallelism. It may be used to compute at the unit level to compute the *dot-product* or, if we consider the network as a list of connections, it is possible to vectorize the propagation through the set weights between two layers.

3.1 A brief history

A lot of different solutions have been tried on super computers with very different architectures. We will go through every possibility and give some examples.

3.1.1 Concurrency on samples

In theory, with the *Back-Propagation* learning algorithm, weights are updated after presentation of many samples. A very efficient way to implement concurrency is to give to each Computing Element (CE) one or several samples to learn. Weights are only updated much later. In this way, Pomerlau et al.(1) improved learning speed up to 17 MCPS (Million Connections Per Second). on a WARP computer. No information has to be exchanged between the CE, and speed increases linearly with the number of CE. We are using this method on an **Alliant** computer at *ATR Interpreting Telephony Research Laboratories*, and have found several limitations:

1. This algorithm imposes a tight synchronisation on weight updating over all the units. Learning algorithms that take advantage of the noise brought by a more frequent or irregular weight updating frequency are here forbidden.

2. Each CE should share a minumum number of writing memory locations with the other CE s. To do so, the program has to replicate all the variables of the network this CE needs to write in.
3. If, for a given sample, the output error is below some value, we may want to skip the backward pass and save much computing time. This seems very difficult with highly synchronized CE such as those on th **Warp**.

3.1.2 Concurrency on units or sets of units

This solution seems the most "natural", but the transfer of activations between units raises problems of synchronisation. Kajihare(2) introduces a slave process which waits for the activations sent by the other units. Another solution is to carefully sort the units to process, so that if unit B depends on unit A, unit B be will be processed only after full completion of the computations needed by unit A. As each unit has its own parameters and its own input (or output) weights, we do not have to bother about shared memory. Because of the very coarse granularity of most parallel computers, clusters of units are often used rather than units.

3.1.3 Parallel computation over the set of connections

This method is well adapted for supercomputers with a lot of R.I.S.C. CE such as the Connection Machine. It may be simulated on pipeline computers, too. The vector is here the whole set of connections between two layers, its size is much larger than in the preceding section and allows a better optimization. However, memory access may take longer as the weights, input activation and output summations are scattered among many units. It is not obvious when to apply the non-linear thresholding.

3.2 Our choices

We have chosen the concurrency on units. Even though it is slightly less efficient than concurrency on samples, it brings much less constraints, especially to update the weights.

3.3 References

(1) D.A.Pomerleau, G.L.Gusciora, D.S.Touretzky, H.T.Kung (1988) Neural Networks Simulation at Warp Speed: How we got 17 Million Connections Per Second. *Proc. of the 1988 IEEE Int. Conf. on Neural Networks.*

(2) N.Kajihara, S.Matsushita, T.Nakata and N.Koike. Parallel Neural Network Simulation Machine : NeuMan. *Proc of the INNS First Annual Meeting, September 1988.*

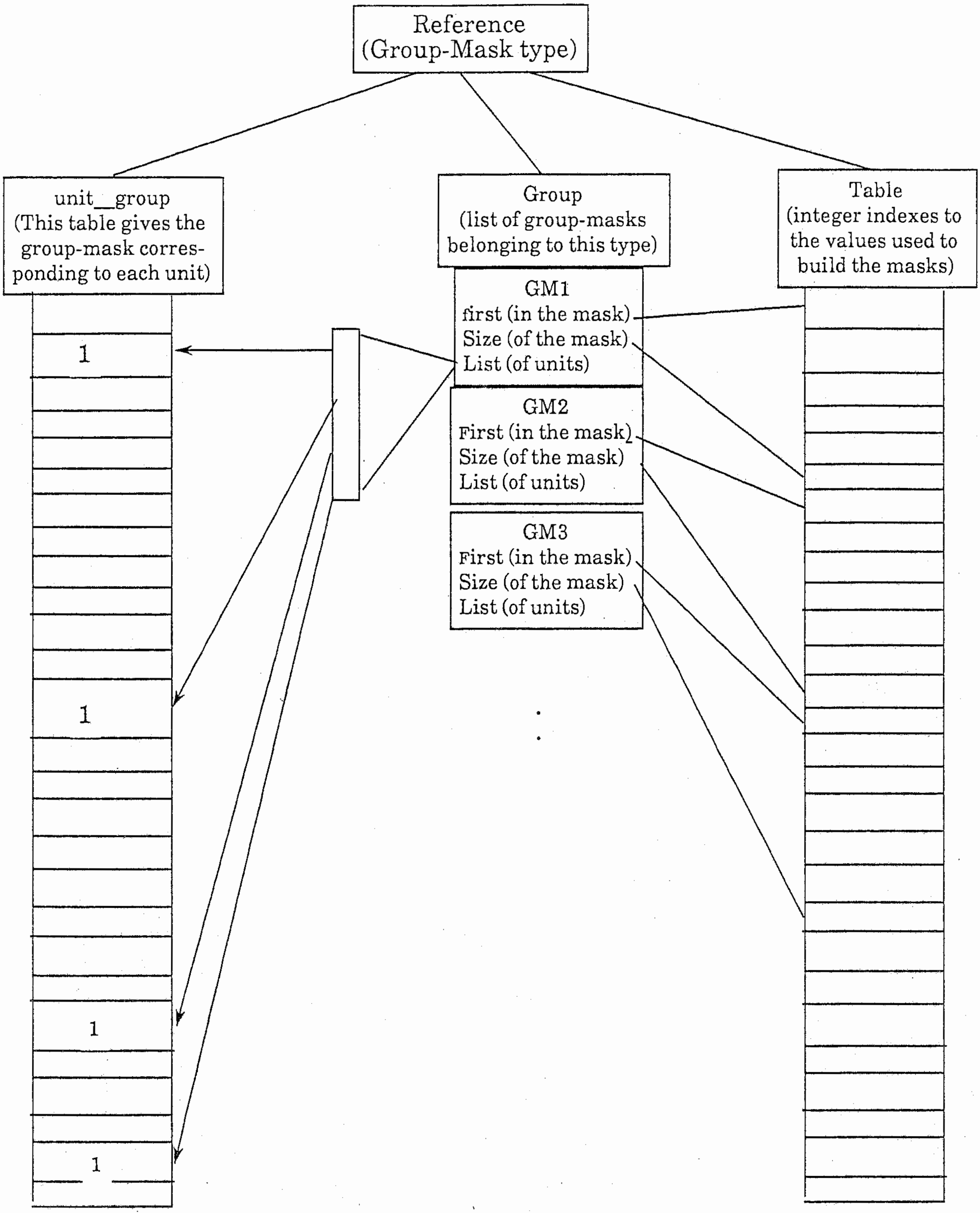


Fig 3: Dynet structure to describe a group-mask type.
(input unit, input weight, output unit, output weight)

Chapter 4

Technical Documentation

This section is of interest only to maintainers of the *DyNet* C code, or those who want to modify the program for some special purpose.

4.1 Code

This program uses standard UNIX C. It has been implemented on a Vax ULTRIX system and on an ALLIANT CONCENTRIX system. Portability to other UNIX systems should be easy, as only a few system calls are made to get the CPU time. The run the program on the Alliant, one only has to add the following to the make file.

- VEC = vec.o : vec.o must be linked with *DyNet*.
- The C compiler needs the options -lcvec -lcommon.
- -DCONCUR optimizes *DyNet* for concurrency on the Alliant. Be sure to remove it on other machines.
- -DVECTOR optimizes *DyNet* for vectorization on the Alliant. Be sure to remove it on other machines.

The default type of the functions is void rather than integer.

4.2 Data

Most global variables are structures. Tables are dynamically allocated through malloc. The complex data structure used by *DyNet* is mostly built and allocated in net.c. The most fundamantal galobal variables are the four group-mask type desriptors, we call here reference: in_u (input unit), in_w (input weight), out_u (output unit), out_w (output weight). The reference structure is very roughly shown in fig.3.

4.3 Description of each part of the program

4.3.1 dynet.c

This is the main program. This program first loads everything it needs:

1. Parameters (see param.c)
2. Sigmoid (see sigmoid.c)
3. Net (see net.c)
4. Samples (see sample.c)

Through *init_statistic* , the statiscal structure used to trace each dynamic table of values are initialized. Then we have the main learning procedure, 2 solutions:

- Recognition only (reco_only) There is only one epoch and the backward phase is inhibited.
- Training samples For each epoch, *bakprop* is called. The latter calls the *learn* procedure only for the samples that need it (their error was still large during the previous epochs) .

4.3.2 net.c

Here are the functions that build the network. They are all called through *initnet* :

net_read reads the network architecture through fname.net.

init_ref initializes the reference structures.

malloc The dynamic tables are allocated memory (dl.table.v) .

init_dyna When necessary, they are allocated additional memory to store the masks (dl.table.m) or to compute the norms (dl.table.n) .

init_unit links each unit to its 4 *group-masks*.

4.3.3 sample.c

All the functions that work on the **iof** are gathered here:

read_sample Read all the **iof** by calling read_io.

read_io Read the **iof** ASCII description. If needed, calls read_data to read the binary sample file.

reset_allio Call reset_io for each **iof**.

reset_io First ends the saving or loading phase. Then checks if there is a new message and modifies **iof** accordingly.

activate_io

inhibit_io

Forward Or Backward Pass in layer S

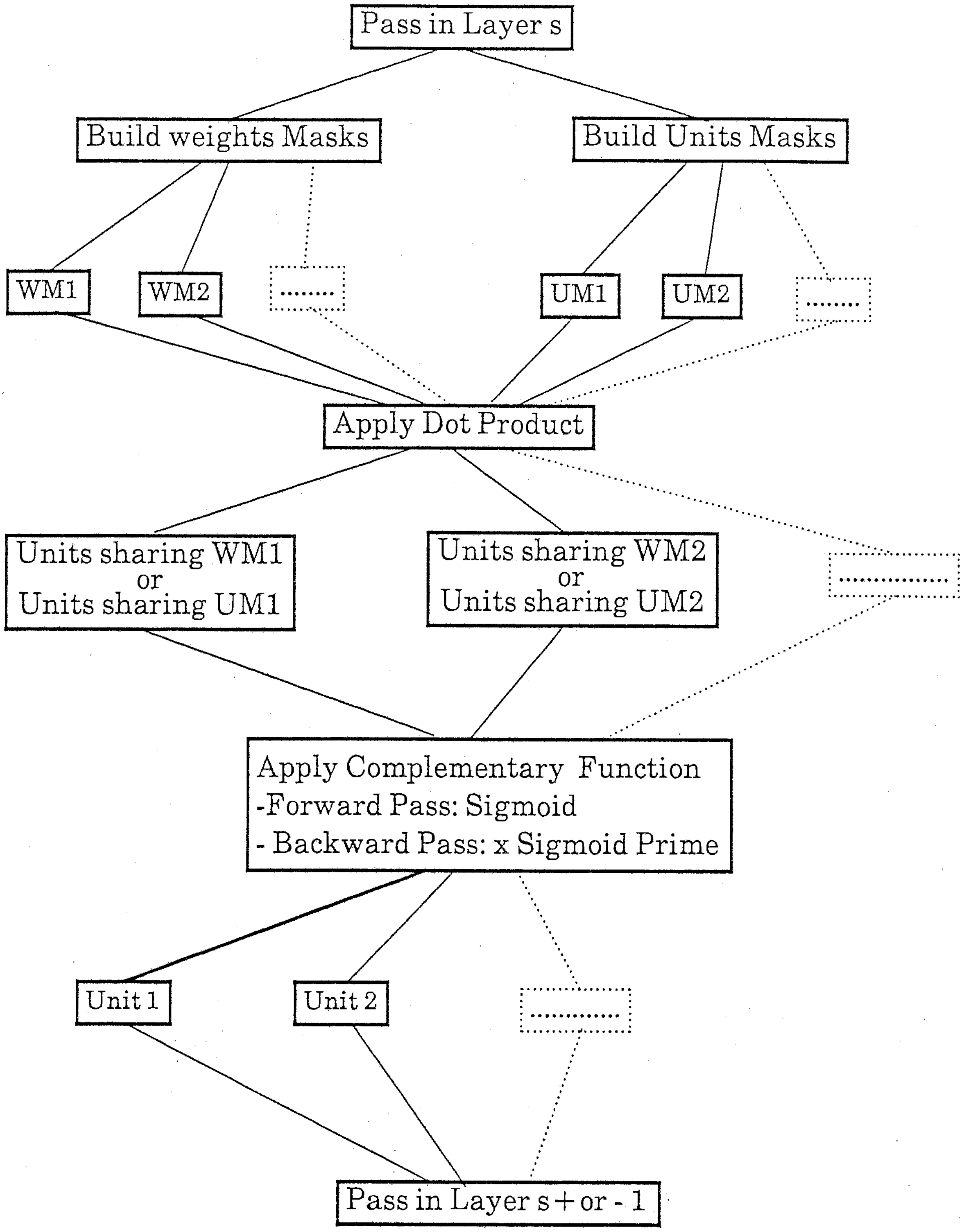


Fig 2 : The concurrent dynet algorithm.

4.3.4 pass.c

This is the "kernel" of the learning algorithm. One finds the description of the pass algorithm in Fig.2. The performance of the program mostly depends on how the functions in pass.c are optimized. Several methods are possible, each corresponding to one value of X:

Vector On a pipelined processor, vectorized routines are called. (X = vec)

Scalar Standard sequential C code is kept. This may be faster when the computer is not vectorized or when the vectors have a small size. (X = scal)

Unique Optimized for vectors of size 1. (X = 1)

List of functions:

pass_layer The pass_layer function is central. It is able to perform the forward OR the backward pass through any layer.

build_mask_X Prepares the *group-mask*: build all the masks in the g-¿dmask list. If needed, computes their norm.

build_dot_X Computes the dot-product for all the units belonging to the group.

mask_dot_sel Calls vector, scalars or unique build_mask and subsequent build_dot routines according to the size of the vectors.

sigmoid Computes the sigmoid and its derivative.

sigmoido Special for ouptut units

backward During the backward phase, multiply the back-propagated Error by the sigmoid derivative.

backward_output Special backward functions for output units that also computes the Error. McClelland new error is available.

4.3.5 optimize.c

These routines, called when building the network, optimize each forward and backward pass for each layer. This is done on two levels:

vectorization CPU times for scalar and vector routines are compared and the better is kept.

concurrency In a given layer, concurrency may be applied over weight masks or unit masks. The fastest solution is chosen.

4.3.6 weight.c

These routines update dEdw and weights.

update_dEdw is called at each iteration. As it is very time consuming, it is carefully optimized and the choice of a scalar or a vector routine is made while initializing the net.

update_weight is called at each weight updating period. On concurrent machines, if the weight mask is shared, weights are locked while beeing updated, to avoid simultaneous acces of shared data, which would cause unpredicable results.

4.3.7 display.c

At the end of each epoch, some traces of the dynamic values are diplayed.

4.3.8 param.c

Parameters are initialized from task/para and displayed on para.out.