TR-I-0058

Fast Back-Propagation Learning Methods for Neural Networks in Speech

音声ニューラルネットワークのための
バックプロパゲーションアルゴリズムの高速化

P. Haffner, A. Waibel, H. Sawai and K. Shikano
パトリック ハフナー、アレックス ワイベル、
沢井 秀文、鹿野 清宏

1988.11

Abstract

Several improvements to the Back-Propagation learning algorithm are proposed to achieve fast optimization of speech tasks in Time Delay Neural Networks. A steep error surface is used, weights are updated more frequently and both the step size and the momentum are scaled to the largest values that do not result in overshooting. Training for the speaker-dependent recognition of the phonemes /b/, /d/ and /g/ takes less than 1 minute on an Alliant parallel computer. The same algorithm needs one hour and 5000 training tokens to recognize all the Japanese consonants with 96.7% correct on test data. Moreover, these fast methods make it possible to study generalization performance on large Neural Network tasks.

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

CONTENTS

List of figures

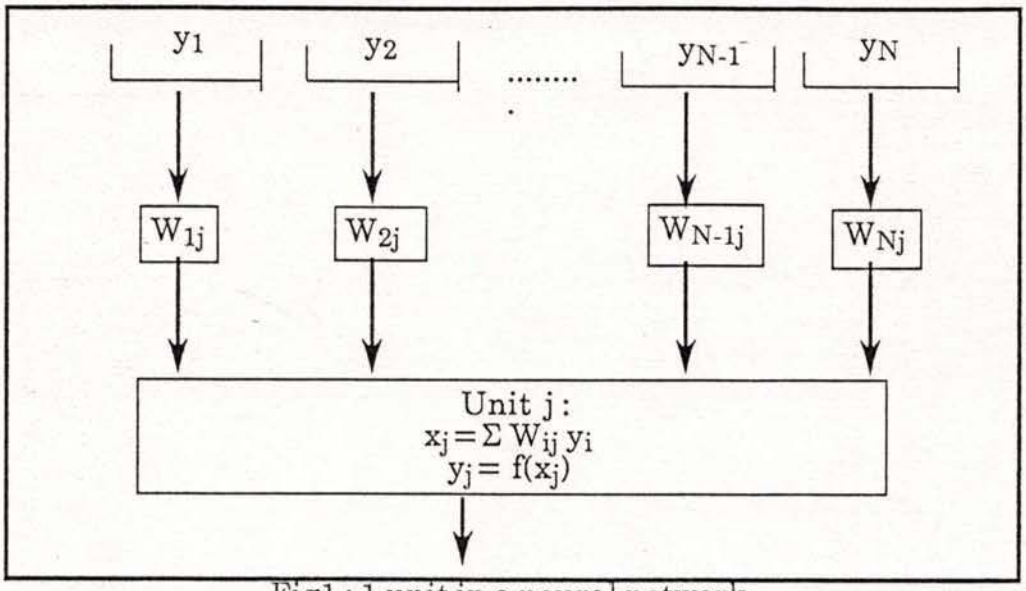


Fig1 : 1 unit in a neural network

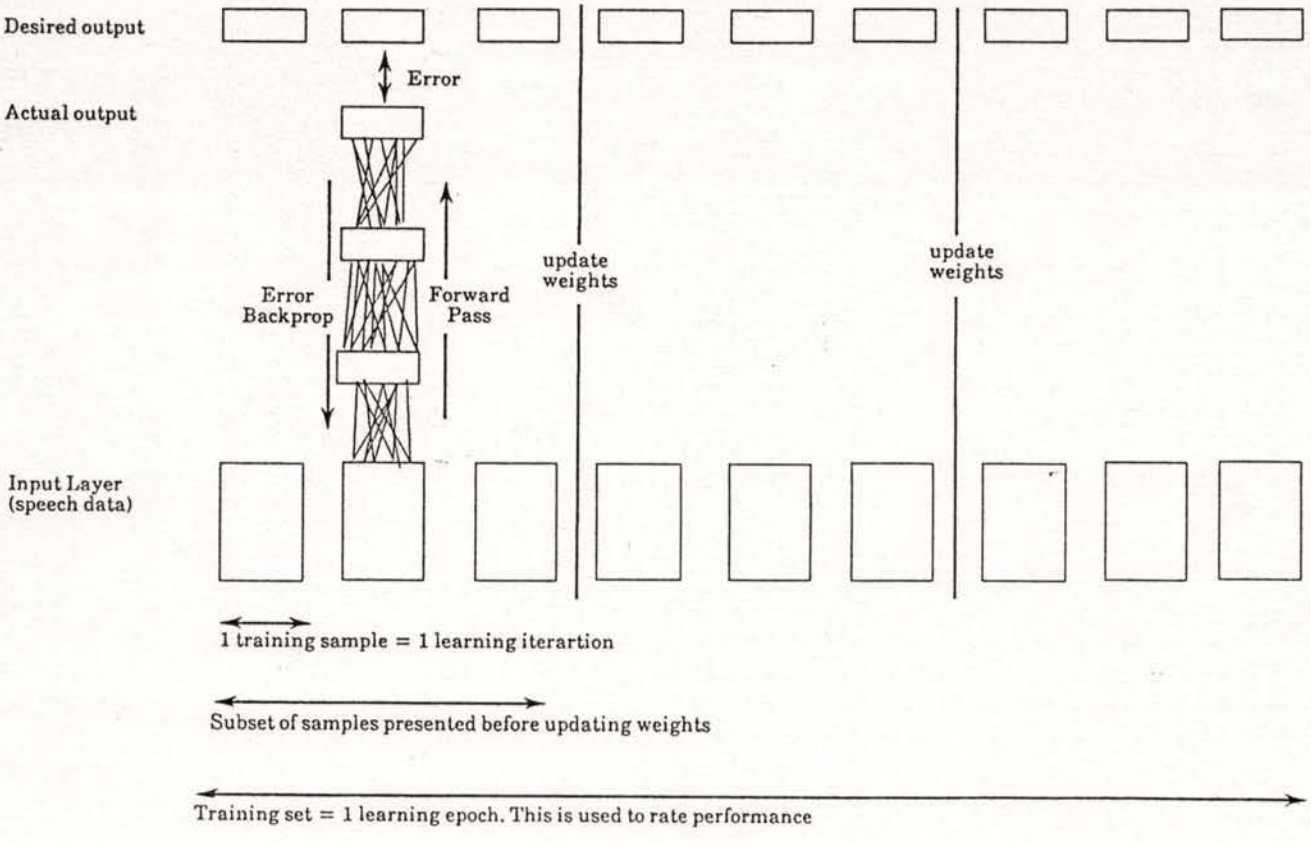


Figure 2 : Learning the training set.

2. Learning with Back-Propagation.

This section presents an overview of learning in Neural Networks, and concludes with a discussion of the improvements we attempted.

2.1. Neural Networks

The basic unit in our network computes a weighted sum of the activations of its incoming units, and then applies a sigmoid function *f* to this sum to compute its own activation. Our networks are not recursive: a unit never feeds back on itself, even through another unit.

2.2. The Back-Propagation learning procedure

We present here the Back-Propagation learning procedure, also known as the generalized Delta Rule. This learning procedure involves the presentation of a set of training samples, which are pairs of input and output patterns. The input pattern is propagated through the network and produces the *actual* output vector, which is then compared with the target or *desired* output vector. The connection weights are changed to reduce the difference between these actual and desired output patterns. We define the error function to be the mean square difference between actual and desired output.

For a training sample p, this error function may be written as:

$$E_p = \Sigma_j (y_{pj} - d_{pj})^2$$

The overall measure of the error is then $E = \Sigma_p E_p$.
The Back-Propagation learning procedure implements a gradient descent on E, and weights are updated according to the relation:

$$\Delta W_k = -\varepsilon \Sigma_p \partial E_p / \partial W_k$$

$\partial E_p / \partial W_k$ is computed for each weight by back-propagating the error signal from the ouput units to the input units. Details may be found in [1]. We describe the algorithm at the unit level in the next section.

We see here that weights are updated after presentation of the whole training set, but it is also possible toupdate them after presentation of a smaller set of samples. We introduce the following definitions:

- Iteration : presentation of one sample.
- Epoch : presentation of the whole training set.
- Updating period : number of iterations between 2 weight updatings.

Fig 2 shows the network learning over one epoch.

2.3. Learning in one unit

The Back-Propagation learning procedure is **local:** the modifying parameters for the weight W_{ij} depend only on unit j and on input activation y_i

$$\partial E/\partial W_{ij}(t) = \Sigma_{samples}\, \partial E/\partial x_j\; y_i + \delta_j W_{ij}(t)$$

$$\Delta W_{ij}(t) = \alpha_j \Delta W_{ij}(t\text{-}1) - \varepsilon_j \partial E/\partial W_{ij}(t)$$

$$W_{ij}(t) = W_{ij}(t\text{-}1) + \Delta W_{ij}(t)$$

$\partial E/\partial x_j$ is the **back-propagated signal:**

$$\partial E/\partial x_j = f'(x_j)\,(\Sigma_k W_{jk}\, \partial E/\partial x_k)$$ k ranges over the output units.

ε_j is the gradient step size.

α_j is the momentum (used to avoid oscillations during the learning phase).

δ_j is the weight decay.

We here consider these parameters as local to each unit. This is different from standard Backprop, where they are shared by all the units; this is also different from making these parameters local to each connection, as adopted by a few researchers. This intermediate solution, of considering the parameters local to each unit, formalizes well. If we write W_j the vector $(W_{1j}, W_{2j}, \ldots\ldots W_{Nj})$, learning is the trajectory of W_j in the weight space, determined by unit j.

2.4. Possible improvements

The goal is to reduce the error measure to zero within the smallest learning time. As a function of the connection weights, this error function defines a complex surface and learning may be seen as a trajectory on this surface, moving down along the steepest slope, preferably toward a global minimum. There are several ways to accelerate convergence:

a) Model a steeper error surface without flat spots, which may be done by an appropriate choice of the sigmoid function or the output error measure.

b) Define an appropriate learning strategy: in which order to present the patterns, when to update the connection weights.

c) Carefully choose the parameters presented in section 2.3.
α_j , ε_j and δ_j will be scaled to:

Make the weight trajectory as straight as possible.

Minimize oscillations.

Have a fast but controlled learning speed.

Attain good generalization on test data

3. Our experimental tasks

We present here our benchmark tasks, in order of increasing size. Due to scaling problems, it is generally very difficult to generalize from a small task to a larger one. The only assumption we make is that something that does not work for a small task is not likely to work for a larger one.

The small tasks we study here are:

XOR: learning exclusive or in a neural network.

838: the network has to learn to encode 8 inputs with just 3 hidden units.

The large tasks we study here deal with phoneme recognition in Time Delay Neural Networks (T.D.N.N.). The data used and the network architecture are the same as those discussed in [2] and [3]. We used a large vocabulary database of 5240 common japanese words [8]. These words are uttered in isolation by one Japanese native male speaker. All utterances were digitized at a 12 kHz sampling rate. The database was then split into a training and a testing set, from which the actual phonetic tokens were extracted.

The training tokens were randomized within each phoneme class. The training set was then built by alternating each class to be learnt. After training, the network is evaluated on the testing set.

We have mostly focused on the TDNN shown in fig.3, whose architecture is discussed in [2]. Two different pairs of training-testing data have been tried:

BDG: Learning the three stop consonants /b/, /d/ and /g/. The total numbers of training samples and testing samples are nearly the same: around 780 (260 of each class).

PTK: Learning the three stop consonants /p/, /t/ and /k/.

Table 1 gives the important properties of this network. Many different units may stand for the same physical unit, but at different times. The two important numbers in this table are the number of connections, which determines the

computation time for one iteration, and the number of physical connections, which corresponds to the number of free parameters we have in our system.

	Input	Layer 1	Layer 2	Output	Total
Units	241	104	27	3	375
Physical units	16	8	3	3	
Fan-in	0	49	41	10	
Connections	0	5096	1107	30	6233
Physical connections	0	392	123	6	521

Table 1: Network numbers for BDG task

All the tasks we study here are classification tasks. For one sample with N output activations, the desired output is always one for one unit and zero for all other units. We say that a pattern is recognized when the output unit with the maximum actual activation corresponds to the unit with the desired activation equal to 1. The recognition rate is defined as the percentage of samples correctly classified. The error rate is therefore the complement of the recognition rate with respect to 100.

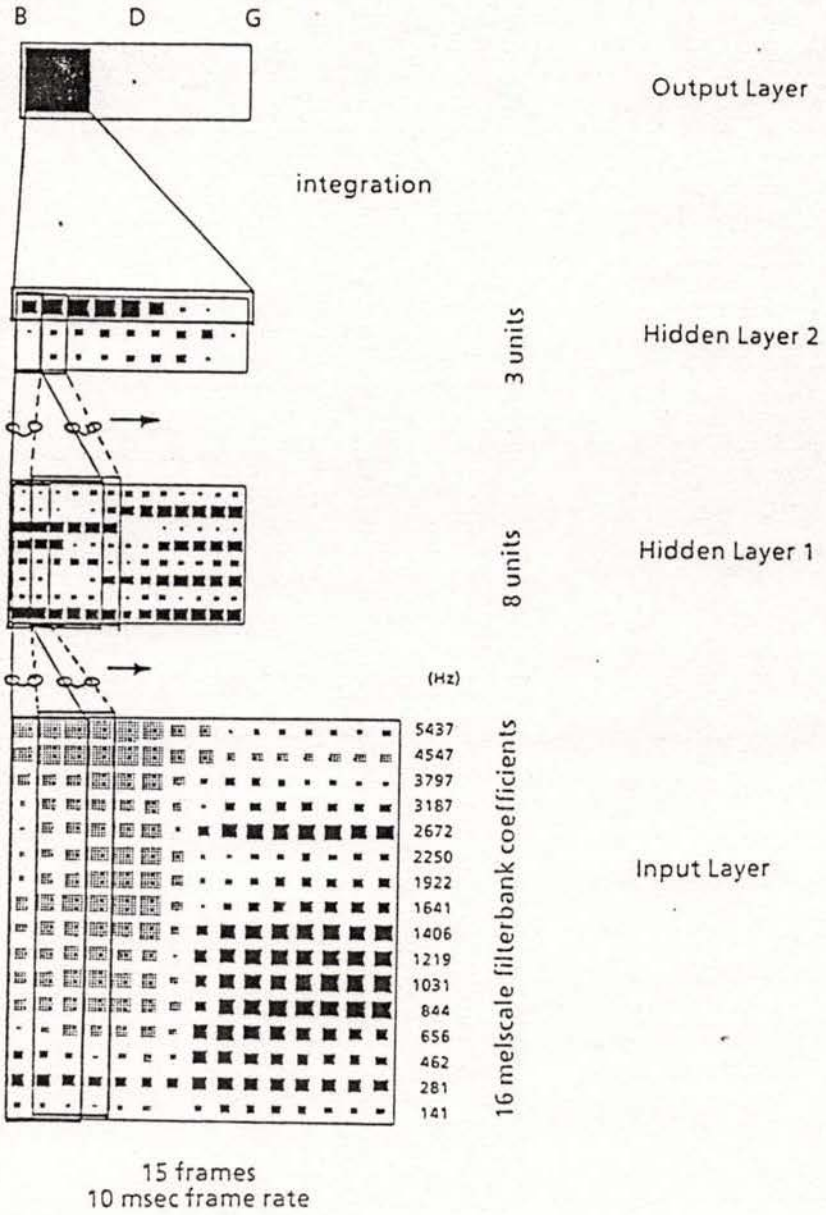


Figure 3 : T.D.N.N. with 3 classes.

4. Methodology

4.1. Introduction

This section introduces the methodology for our experiments. In most studies and comparisons of the speed of various connectionist algorithms [12, 13, 14 ,15, 16, 19] the criterion used is the speed of convergence of the algorithm on a set of training data. With problems like XOR or the encoder-decoder, this method is appropriate because we know the existence of an optimal solution and there is no open data. However in most real size problems, we do not know about the existence of this optimal solution. It is not guaranteed that we reach an error of zero. Moreover, the best solution for training data does not necessarily yield optimal results on *open* or *test* data. With small problems, it is also easy to obtain good statistics on the learning speed by performing a lot of learning runs (hereafter *trials*) with different initial conditions. This is much more difficult with larger problems where a trial may take up to several days. We first discuss these problems in a more detailed way with some examples, and then present our multi-staged benchmark.

4.2. When to stop learning

With training algorithms for pattern classification problems, it is always very difficult to know when to stop the training program, before it begins to overspecialize the system into recognizing the training samples, with a loss of performance for test data[17]. In phoneme recognition with Back-propagation, we have indeed found that zero error for a set of training samples is not desirable. Learning has to be stopped before this point is reached. A good way to determine completion time is to split the set of samples in two, and only use the first half to train the network, keeping the second one to track the generalization capacity.

We have also found as a rule of thumb that learning is often complete when the error rate on training data reaches a plateau. For instance, we see in table 2 that with the BDG task, learning should be stopped when the recognition rate on training data has been stable during 10 epochs.

4.3. Reporting learning time

Several ways of measuring learning time are possible:

- An epoch corresponds to the presentation of each of the patterns in the training set. As the size of our training set is fixed and as the algorithms we present do

Epochs	10	20	30	40	50
Training data	1.41	0.90	0.64	0.64	0.64
Test data	2.24	1.45	1.19	1.19	1.32
remarks			plateau	stop here	overshooting

Table 2 :BDG task with 780 training samples:
Error rates

not substantially modify the computing time per epoch, we have used epochs to rate performance.

- The number of iterations (pattern presentations) may give more accurate results, especially when the numbers of patterns are not the same for each epoch. This is for instance the case with our sample skipping procedure, and we will probably use this method for future experiments.

- The number of connections that have been passed through to learn the task is also very significant. It gives an idea of the required time when the the simulation program performance is known in terms of Millions of Connections Per Second. It allows comparisons with neural networks of different architectures.

- Floating point multiplications : generally a pass through a connection requires 3 floating point multiplications. This would be a good measure for comparisons of training time with other speech recognition methods such as Hidden Markov Models or Learning Vector Quantization.

- Computation time. Fundamental to satisfy our curiosity, but we do not favor it as a standard, as it is too hardware dependent.

4.4 . The Importance of Initial Conditions

Initial weights have a strong influence on final performance, both in terms of learning time and recognition rate. We have found that it is possible to improve learning speed by choosing weights which are adapted to the problem. For instance, in a TDNN, a physical unit is connected to another physical unit through several connections, each having a different delay. Even though their weights are different, learning is faster if these weights are initially set equal, as shown in table 3.

Epochs	10	20	30	40	50
Adapted weights	2.5 / 50	1.9 /70	1.7 /100	1.7 /100	1.6 /100
Random weights	2.8 /20	2.5 / 90	2.0 /100	1.9 /100	2.0 / 100

Table 3 : BDG task with 780 training samples .
Error rate / % of converging trials
(10 trials, Error averaged on converging trials)

However, the gain is very small, and we do not have any general theory, such as with Learning Vector Quantization [9, 10] which in large part owes its high learning speed to good initial conditions. We would like to adapt these ideas to multi-layered neural networks. Up to now, we have only been using a few practical results.

- Initial weights are random numbers with a gaussian distribution. The average is zero and the standard deviation is carefully chosen so that initial activations range over the steep part of the sigmoid function. A bad choice on this standard deviation may lead to configurations likely to never converge. A way to compute it may be found in [21].

- With a given learning task, it is possible to keep the same initial weights for most experiments. When initial weights give good performance for a certain learning method, performance is still good with slightly different learning methods, and we see in Table 4 that trials number 2 and 0 (among 10 trials starting from different initial weights) give very good performance for 3 different learning methods. Moreover, good initial weights with small training sets generally yield better than average performance with larger training sets.

Learning method	Standard	Sigmoid derivative + 0.01	overshooting control 10 instead of 1
Number of the 3 trials (ranging from 0 to 9) yielding the best recognition performance on test data.	2, 0, 3	2, 0, 8	0, 3, 2

Table 4 : BDG task with 780 training samples

4.5. A multi-staged benchmark.

To avoid tackling all these difficulties at the same time, our benchmark must include several stages.

The first stage deals with small problems, such as XOR and 838, or large ones with a very small number of training samples (always less than 100). Learning is completed when the error goes below some small fixed value. This is a kind of preselection for our different learning improvements. Methods leading to a catastrophic behavior are rejected.

The second stage deals with medium-sized problems, such as BDG and PTK. If learning takes one hour, it is possible to run ten trials with different initial conditions within one night. For each trial, we keep track over time of the recognition rates on training and test data. The problem is now: how to rate performance using all these results? Ideally, performance should be rated with two numbers: the best recognition rate and the number of epochs required to reach it. It is indeed very tempting to look over all the trials and take the trial, and the epoch for that trial, yielding the best recognition rate. Using this method, we can claim that our network is able, with some very favorable initial conditions, to yield a recognition rate of 99.5% for the BDG task rather than 98.6%. We reject this method as we want to achieve a good result reliably, within just one trial on average. Learning is only restarted when caught in a local minimum, which we have seldom observed with our algorithm.

We see in table 5 that for error rates on training data which are less than 2%, it is impossible to interpolate performance on test data from performance on training data, probably because our database is not large enough. However, with our speech recognition tasks, we have found that when the error rate on sample data is over 2%, performance on test data is significantly worse. For a given epoch, we call "converging trial" a trial which yields an error rate of less than 2% on sample data. We rate our recognition performance with two numbers: the percentage of converging trials and the error rate on test data averaged over the converging trials.

Trial	0	1	2	3	4	5	6	7	8	9
Training data	0.64	0.64	0.51	0.38	0.38	0.38	0.64	0.77	0.51	0.38
Test data	1.32	2.50	1.05	1.32	1.71	1.45	1.71	1.98	1.84	1.19

Table 5 :BDG task with 780 training samples:
Average Error rate after 50 epochs

4.6. Our simulation program.

During this research we have used several Back-Propagation simulation programs. Programmed in the C language on a UNIX system, they have been optimized to run in parallel on an Alliant supercomputer. Three versions are used here (we quote their respective speed in MCPS, Millions of Connections Per Second) .

- The first results reported by Waibel[2, 3] (learning the BDG task with a 98.6% recognition rate within 4 days) were obtained with an optimized program running on an Alliant with 4 processors.
- Most of the results we report here were obtained with the same program, but on an Alliant with 8 processors, yielding 0.8 MCPS.
- The most recent version runs at more than 2 MCPS on an Alliant with 8 processors.

5. Modeling the error surface

5.1. The sigmoid function

The presentation of a pattern modifies the weight connecting units i to j through $f(x_i)\ f'(x_j)\ \partial E/\partial y_j$. The Back-Propagation learning rate is then proportional to the values of the sigmoid function f and its derivative f'. But these functions flatten out at infinity, as seen in figures 4 and 5. There are several ways to make the functions non-zero at infinity.

- First, we can create a **symmetric sigmoid** whose value is never zero at infinity, by substracting 0.5 from the sigmoid. This generally gives a slightly better learning speed as shown in [15]. However, at the beginning of the learning phase, when the weights are small and the activations close to zero, learning may be very slow to initiate and break the symmetry .
- We may add a linear function to the sigmoid function: $f_1(x)=f(x)+1.x$ .The derivative becomes $f_1'(x)=f'(x)+1$. This amounts to adding a small positive constant 1 to the sigmoid derivative, which therefore cannot be zero (1 is generally between 0.01 and 0.1). Even when it improves performance, this method has to be used with the utmost care. We have found that with difficult tasks, the network may work with very large activations, in the domain of f far from 0. As a consequence, a small change in some weights may change the behaviour of the network considerably. The latter may lose most of its robustness and fault-tolerance.
- It is also possible to only **add a small constant** 1 to the sigmoid derivative without changing the sigmoid function. Therefore, during the backward phase, we multiply dE/dy by a factor which is no longer the real sigmoid derivative (this model is mathematically inconsistent but gives good results as shown in [12]) .

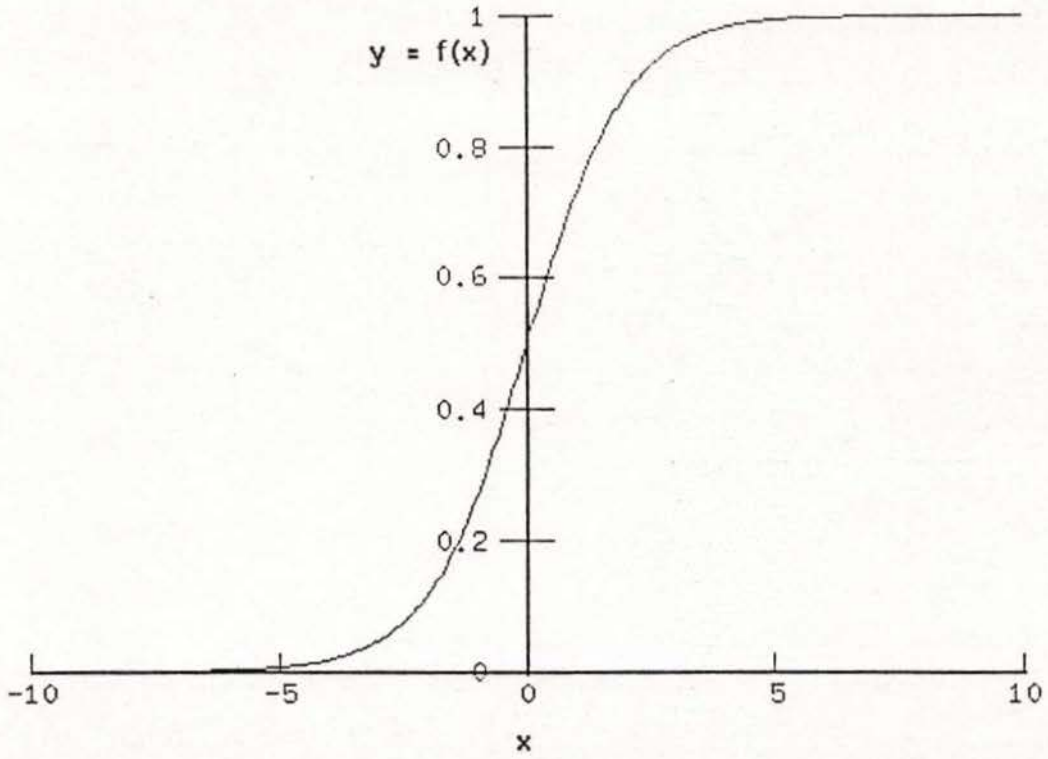


Figure 4: The sigmoid function: $f(x) = 1/(1 + e^{-x})$

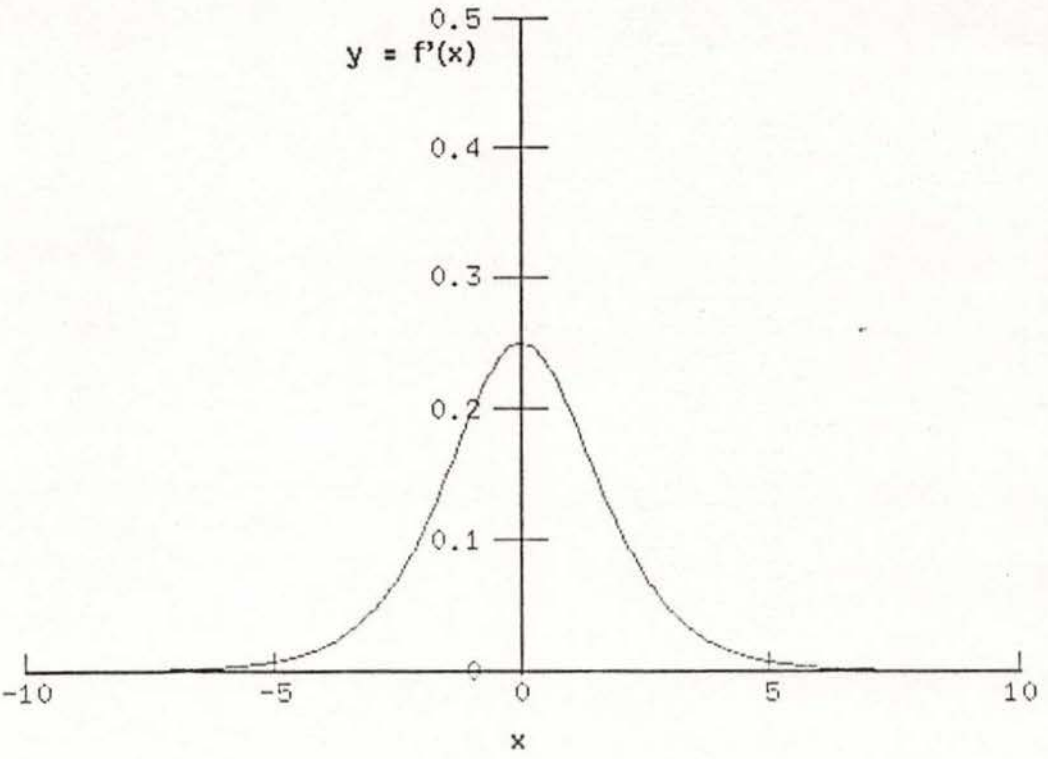


Figure 5: The sigmoid derivative. $f'(x) = f(x)\,(1 - f(x))$

These three methods, especially those dealing with a change in the sigmoid derivative, generally lead to an increase in speed, and guarantee convergence to a zero error global minimum. The results we found are consistent with [12]. However, this forced convergence is most often unwanted, for it leads to network configurations which have overlearnt the training set, yielding slightly worse generalization on test data. We see in table 6 that for the BDG task, the three models allow for fast convergence in all 10 trials, but the final average error rate on test data is higher than with the standard sigmoid.

Epochs	10	20	30	40	50
Standard Sigmoid	2.5 /50	1.9 /70	1.7 /100	1.7 /100	1.6 /100
Sigmoid 1	2.4 /50	2.1 /90	2.1 /90	2.1 /100	1.8 /100
Sigmoid 2 l=0.01	2.4 /90	1.9 /100	1.9 /100	1.9 /100	1.9 /100
Sigmoid 3 l=0.01	2.1 /100	2.0 /100	1.8 /100	2.0 /100	2.0 /100

Table 6 : BDG task with 780 training samples .
Error rate / % converging trials
(10 trials, Error averaged on converging trials)

5.2. A New Error.

Because of the multiplication by the sigmoid derivative, output units whose activations are close to zero or one have delta values which are close to zero, regardless of the size of the actual output error for these units. Then, in the case of a unit whose difference between the real and the desired output is close to one, the weight change on the input weights will be near zero. To cope with this problem, instead of the standard Mean Square Error (MSE) $E = - \Sigma_{samples} \Sigma_j (y_j - d_j)^2$, a **New Error measure** has been proposed by McClelland:

$$E = - \Sigma_{samples} \Sigma_j \ln(1 - (y_j - d_j)^2)$$

(y_j is the actual output and d_j is the desired output).

The derivative of this new error is :

$$\frac{\partial E}{\partial yj} = -\frac{1}{1-(d_j - y_j)} + \frac{1}{1+(d_j - y_j)}$$

After multiplication by the sigmoid derivative $y_j(1 - y_j)$, we find:

$$\frac{\partial E}{\partial xj} = \frac{(1-y_j)^2}{y_j - 2} \quad when \ d_j = 1$$

$$and \ \frac{\partial E}{\partial xj} = \frac{y_j^{\,2}}{y_j + 1} \quad when \ d_j = 0$$

We see that now **dE / dy_j is maximal when $y_j - d_j = 1$** . This new error ensures that the system will always learn and has roughly the same effect as adding a small constant to the sigmoid derivative, as we see in table 7:

Epochs	10	20	30	40	50
Standard Error	2.5 /50	1.9 /70	1.7 /100	1.7 /100	1.6 /100
New Error	2.4 /90	2.4 /100	2.1 /100	2.1 /100	2.1/100

Table 7 :BDG task with 780 training samples .
Error rate / % converging trials
(10 trials, Error averaged on converging trials)

We have found this new error to be particularly helpful with very large tasks with a lot of different classes, such as learning all the consonants, as described in section 8. Here, over 18 consecutive samples, the desired activation of any output unit is 1 one during 1 iteration and 0 during the 17 others. Outputting a zero activation for the 18 samples is an easy approximation for the network to learn. The sample whose output is 1 is not correctly learned, but no correction happens, as the sigmoid derivative is equal to zero. The new error prevents this problem and increases learning speed by several orders of magnitude. It is here very useful when the desired activation is 1.0.

To confirm this, we introduce a hybrid error, which corresponds to the new error when the desired output $d = 1$ and to the MSE error when $d = 0$. This hybrid error is then written :

$$\frac{\partial E}{\partial xj} = \frac{(1-y_j)^2}{y_j - 2} \quad when \ d_j = 1 \quad (same \ as \ new \ error)$$

$$and \quad \frac{\partial E}{\partial xj} = (1-y_j)y_j^2 when \ d_j=0 \quad (same as M.S.E)$$

Performance is the same as with the new error. As the BDG task does not have enough classes to show the efficiency of the new error, we have made the comparisons on a network which learns all the 23 phonemes (18 consonants + 5 vowels). Results are shown in table 8.

Method	Number of epochs to reach a 98% recognition rate on training data
M.S,E, Standard Sigmoid	Much larger than 200
M.S.E, Sigmoid + 0.01 x	120
New Error, Standard Sigmoid	20
Hybrid Error, Standard Sigmoid	20

Table 8 : Performance of the 23 Phonemes network with different Error functions.

5.3. The input activations

In staged learning, we take as input data for the second learning phase activations that were obtained after a first learning phase. With the standard sigmoid, these activationas range from 0 to 1. We have found experimentally that an important gain in speed was obtained when the **input** activations were ranging from -1 to +1 rather than from 0 to 1. This may be done by scaling the input data.

6. Learning strategy

We use the word strategy to describe the parts of our learning algorithm that deal with the way the training samples are presented to the network. The underlying ideas are very simple and primitive (nothing to do with real pedagogy), but they may be very efficient.

6.1 The Weight updating frequency

In this section, we discuss the problem of when to update weights. There seems to be two trends the Back-Propagation procedure: the Standard B.P., which updates the weights at each epoch and the stochastic B.P. , which updates the weights at each iteration. We have found that the optimal solution should be in-between the two procedures.

Splitting a large and often highly redundant training set into smaller subsets for the purpose of weight updating may be very advantageous. At the beginning of the learning phase, one subset is enough data for a network which is only acting as a rough classifier. As a consequence, updating weights over any subset may be as effective as updating weights over the whole training set, at the beginning of training. This remark leads to two different learning procedures. The first one is staged learning: the network first completely learns a single small subset of samples, which is progressively expanded to include the whole training set. The problem is that the network tends to become overspecialized for the first small subset. The second idea is to update weights often at the beginning and then progressively increase the weight updating period. This method has proved to be extremely efficient. If we keep the weight updating period well under the number of iterations for one epoch, we find two other advantages.

- At the end of the learning phase, fine-grained learning and a large training set are needed. However, the difference between two consecutive learning subsets may be considered as noise which prevents local minima.
- The other advantage of updating weights more often is more difficult to explain: it is not only faster, it also yields better generalization performance. A possible reason is that, as we add fewer sample errors for each update, learning steps are smaller and better controlled.

How do we change our learning procedure? We have an M samples training set and update weights after presentation of N patterns. In our BDG task, which uses three phoneme classes, we have found that using as the updating period any multiple of 3, ranging from 3 to 48, gives fair results. More generally, in problems

with C classes where C is much smaller than the number of input samples, N should be a multiple of C. Furthermore the momentum should become larger for smaller N's. Suppose α is the optimal momentum with an updating period of M samples, it would be logical to change it to $\alpha^{N/M}$ with an updating period of N samples. Practically, this relation is not well verified and we will propose a momentum scaling algorithm that will adjust the momentum to an optimal value.

When updating weights often, we have to take some basic precautions. The most important is to carefully mix the training samples. In our phoneme database, the consonants are not listed at random, but according to their following vowel. For instance, first come the BAs, then the BIs, BEs, BUs and BOs. To keep this order when updating weights very often may lead to oscillations which are periodic and very harmful to performance, as seen in table 8. We have also found that the weight updating period should not be changed too suddenly over time.

We propose now a procedure incrementing the weight updating period over the number of epochs. For the BDG task, it is:

- Training sample: **randomly mixed**, to avoid overspecialised training subsets.
- First learning epoch, weights are updated every 3 iterations.
- Each epoch, the size of the training subset is incremented by 3, until it reaches its maximum value of 48.

This is very easy to generalize to any kind of problem with a fixed number of classes. The initial value is generally the number of classes, and the increment should stay small. With the BDG task, as seen in table 8, the learning speed is multiplied by a factor of 5 to 10, the converging probability is increased and the recognition rate on test data is improved! The only problem is theoretical: our algorithm is now only an approximation of the gradient descent algorithm and some improving procedures derived from this algorithm are no longer possible.

Epochs	20	50	100	200	300
Epoch updating	0 /0	0 /0	2.0 /10	2.3 /100	2.0 /90
[P] Non mixed set	2.0 /10	2.1 /70	1.5 /70		
[P] Mixed set	2.5 /50	1.6 /100			

Table 8 : BDG task with 780 training samples .
Error rate / % converging trials
(10 trials, Error averaged on converging trials)

6.2. Skipping samples

When learning a large sample database, the learning program tends to spend most of its time trying to learn a small minority of unclassified samples, while most of the other samples already yield an output error close to zero. It is then a loss of time to perform both the forward and the backward pass on these learned samples. To prevent this, one has only to set a minimum error : if a sample output error is below this minimum, no backward pass is performed. This algorithm may be improved for samples whose error is much below this minimum: we consider that it will take a number of epochs proportional to the difference between the minimum error and the output error for this sample to be worth learning again.

With this method, we commonly skip 75 % of the training samples at the end of the learning phase, and save the same percentage of CPU time per Epoch. Let us take an example with a minimum error of 0.001. The old Error is incremented of 0.0002 at each epoch until it reaches back 0.001.

Epoch	increment Old Error	Forward pass	Compute Error	Backward pass
20	no	yes	0.0011	yes
21	no	yes	0.00046	no
22	0.00046	no	no	no
23	0.00066	no	no	no
24	0.00086	no	no	no
25	0.00106	yes	0.0014	yes

7. Controling learning

In this section, we propose algorithms for scaling the step size, the momentum and review Rumelhart's algorithm for weight decay.

7.1. The step size

Setting the step size is one of the most difficult problems with Back-Propagation. The literature [12, 13, 14, 16, 19] proposes many different algorithms to scale the step size and we have tried several of these. They seem however difficult to tune and, as they generally try to scale the step size to its maximum value, they lead to large jumps in the weight space which are difficult to control. Our work in this field is detailed in Appendix A, which discusses a dynamic step size algorithm we have widely used during our work. The problems of parameter tuning and the efficiency of our momentum scaling algorithm limit its utility.

We have kept one very simple and efficient feature of this algorithm: the overshooting control procedure. During learning, very brutal changes in learning strategy may appear spontaneously. As a consequence, the norm of the gradient vector grad (E) = ($\partial E/\partial w_1$, ...,$\partial E/\partial w_k$, ...) may be multiplied by a factor of ten within an updating epoch. We have therefore added a control that, at each updating iteration, limits the norm of the vector ε.grad(E) to a fixed value ω. An elegant way to perform this is to resize the step size according to:

$$\varepsilon' = \frac{\varepsilon}{1 + \frac{\varepsilon}{\omega} \sum_{i \in C} (\frac{\partial E}{\partial W_{ij}})^2}$$

$\omega = 1.0$ works with a large class of problems and prevents most oscillations.

7.2. The momentum

When the weights are updated at each **epoch**, it has been found that a momentum of 0.9 substantially increases learning speed compared to Back-Propagation with no momentum. Several explanations may explain this fact. In narrow steep regions of the weight space, the effect of the momentum is to focus the movement in a downhill direction by averaging out the components of the gradient that alternate in sign. Momentum also enables the network to jump over many local minima.

In our experiments we have indeed found that using this value is a good general method of increasing learning speed, but it is not optimal especially when we are updating the weights more often than once for each epoch.When the

momentum is too small (0.9), we have oscillations, which means that the average is made over too short a time.When it is too large, we have uncontrolled overshooting, as the network inertia is so large that it cannot change its direction in the weight space in time.

We propose now an algorithm to scale the momentum to an optimized value. First consider the ideal case of $\alpha = 1$: this means that there is no loss in momentum for the weight variation ΔW , which is then a perfect average over all the samples. This is interesting over one epoch, as it smoothes the differences between the samples. However, it is very dangerous over long periods of time, as the change computed long ago may not be desirable with a network which has changed a lot. To remedy this problem, the momentum should be reduced as the network changes. It has been found experimentally that one of the most characteristic symptoms of too large a momentum is the divergence of $\Sigma_i (W_{ij}(t))^2$ over time.

For a given unit j, the quantity we want to control is:

$$Q_j = \Sigma_i (W_{ij}(t))^2 - \Sigma_i (W_{ij}(t-1))^2 = 2 \Sigma_i W_{ij}(t-1) \Delta W_{ij}(t) + \Sigma_i (\Delta W_{ij}(t))^2$$

The weight variation is

$$\Delta W_{ij}(t) = \alpha_j \Delta W_{ij}(t-1) - \varepsilon_j \, dE/dW_{ij}$$

As our ε scaling algorithm limits the ε_j term, it is possi ble to limit the value of Q_j by scaling α_j with the relation:

$$\alpha_j = 1 / (1 + d \, |\Sigma_i W_{ij} \Delta W_{ij}|)$$

$d = 1.0$ gives good performance.

It has been found experimentally that a maximum value ranging from 0.99 to 0.999 is still needed to avoid overshooting. Setting a minimum value of 0.5 sometimes improves convergence. If we consider the weight vector as an object moving in high dimensional weight space, this scaling algorithm, stated in physical terms, makes it more difficult to move away the further the weights are from the origin.

Our experimental results in Table 9 are very encouraging. When weights are updated often, our momentum scaling algorithm makes a higher percentage of trials converge, as the network jumps over local minima. Moreover, generalization performance is better.

7.3.Weight Decay

With Neural Networks, it is very difficult to predict performance on test data from the results we get with sample data. In our experiments, we have only found one property that is correlated in some way with the generalization performance: the Root Mean Square average of the weights. For instance, when using a heightened sigmoid, the average weight size and the error rate on test data are both increasing functions of sigmoid heightening factor. The same pattern is

Epochs	10	20	30	40	50
$\alpha = 0.9$	3.3 /20	2.6 /50	2.7 /60	2.5 /60	2.4 /60
$\alpha = 0.95$	2.8 /10	1.8 /20	1.7 / 30	1.8 / 40	2.0 /50
α : dynamic	2.5 / 50	1.9 /70	1.7/100	1.7/100	1.6/100

Table 9 : BDG task with 780 training samples .

Error rate / % converging trials

(10 trials, Error averaged on converging trials)

(10 trials)

found when we increase the minimum momentum or step size. The idea to try to find a solution with minimal weights is as old as Neural Netwoks themselves. The simplest way to do it is to add a constant weight decay δ, so that each time the weights are updated, we have:

$$W_k = (1 - \delta) W_k + \Delta W_k$$

A good value for δ is very difficult to find: too small, it is unefficient and too large, it makes learning very slow.

Recently, Rumelhart [18] has proposed a new formalism that allows a better understanding of weight decay. The idea is that the simplest, most robust network which accounts for a data set will, on average, lead to the best generalization to the population from which the training set has been drawn. The method is to define a cost function which is minimized for this ideal network and derive a version of Back-Propagation which minimizes this cost function. But the big problem remains: how to define complexity? Rumelhart tries his method for two different complexity definitions: the number of units and the number of weights.

To minimize the number of weights, the proposed cost function for weight ij is written.

$$C_{weight} = \frac{w_{ij}^2}{1 + w_{ij}^2}$$

and the decay

$$\delta_{ij} = \frac{K.}{(1 + w_{ij}^2)^2} \qquad K \text{ is a constant.}$$

Using TDNN, we have found that this method constrains many weights to zero. However we have found a huge loss in learning speed while the increase in generalization capacity was hardly noticeable. With our large tasks, finding a good constant value of K is already a difficult problem, and we have not tried to make it vary with time.

To minimize the number of units, the proposed cost function for unit i is written

$$C_{unit} = \frac{\sum_j w_{ij}^2}{1 + \sum_j w_{ij}^2}$$

and the decay

$$\delta_i = \frac{K}{(1 + \sum_j w_{ij}^2)^2}$$

With TDNN, this algorithm achieves its goal quite readily and inhibits many units. The question is: is this really desirable? We have found no improvements in generalization capacity and we will see in the section on consonant recognition that the number of hidden units should not be minimized in TDNNs.

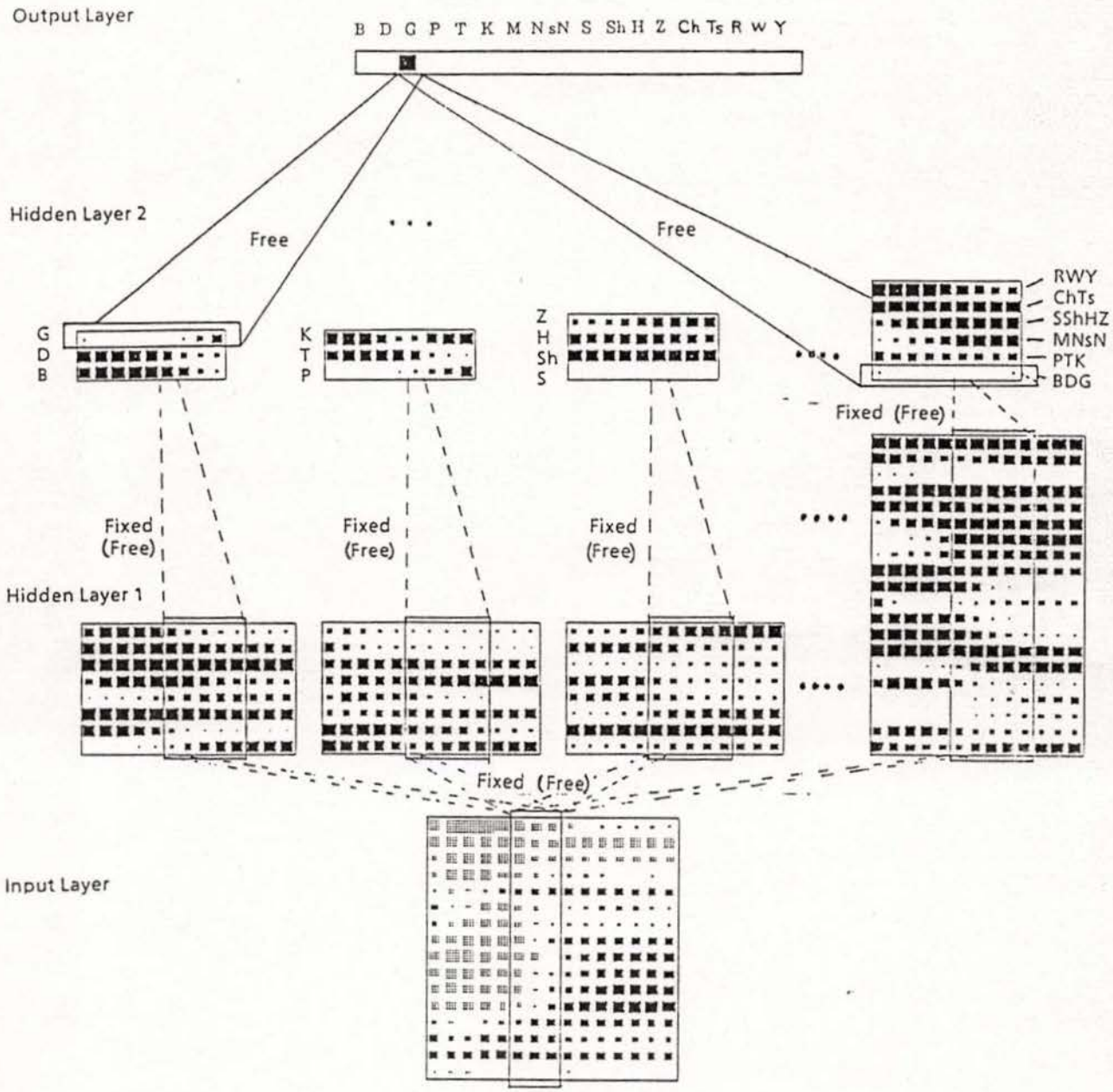


Figure 6 :Modular Construction of an All Consonant T.D.N.N.

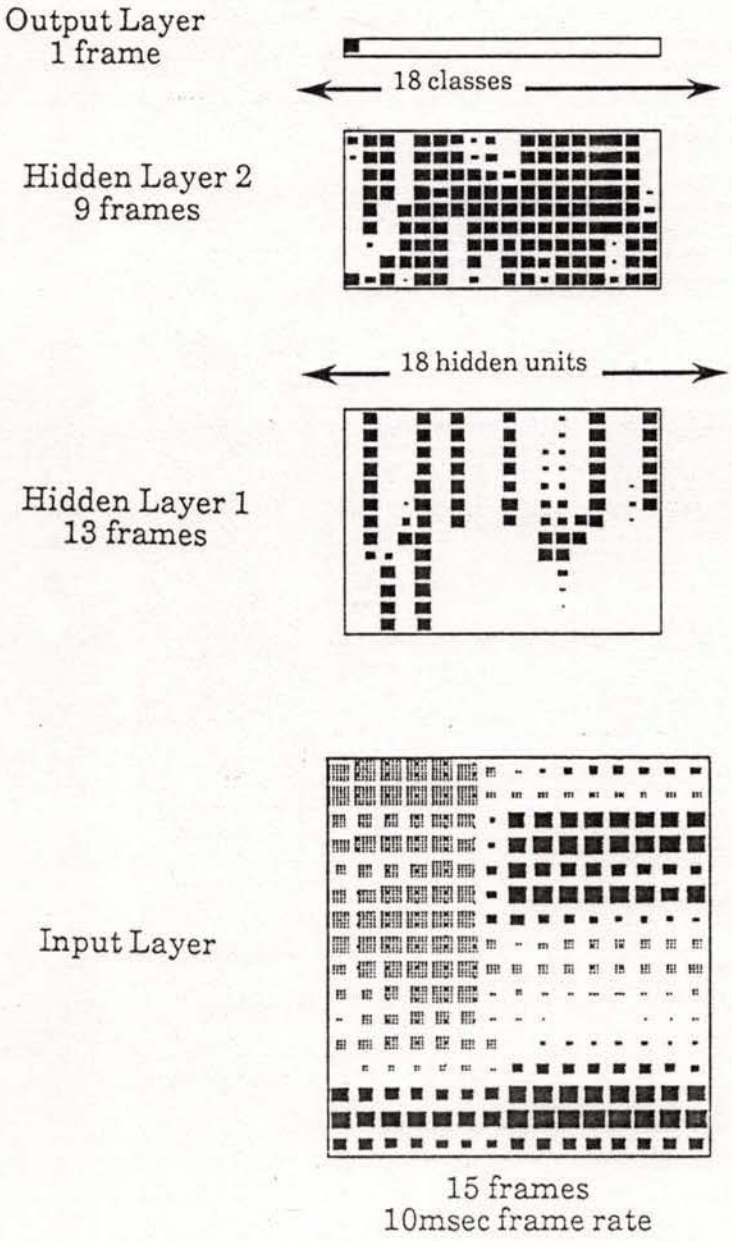


Figure 7a: Non-modular All Consonant T.D.N.N. with 18 physical hidden units.

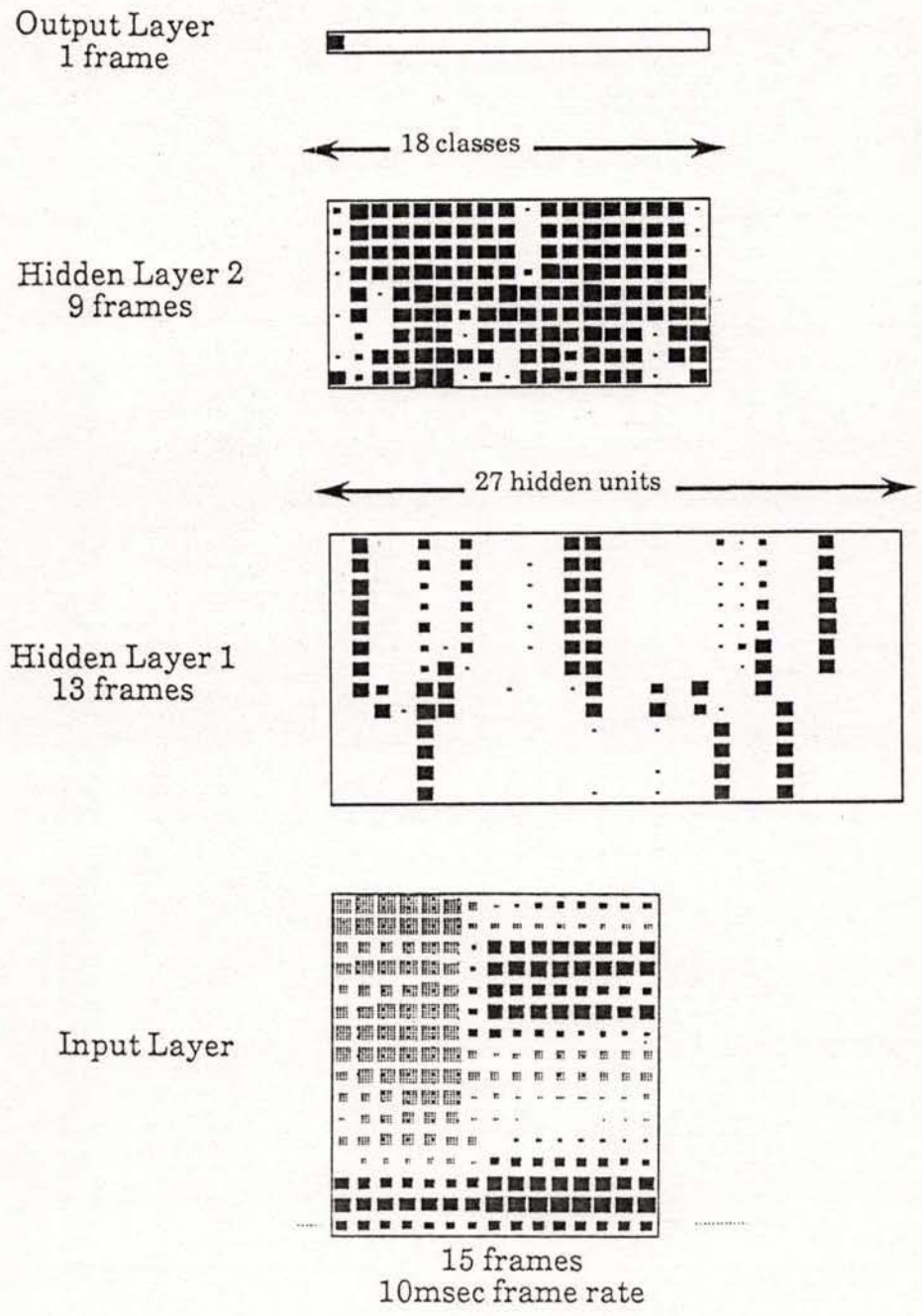


Figure 7b: Non-modular All Consonant T.D.N.N. with 27 physical hidden units.

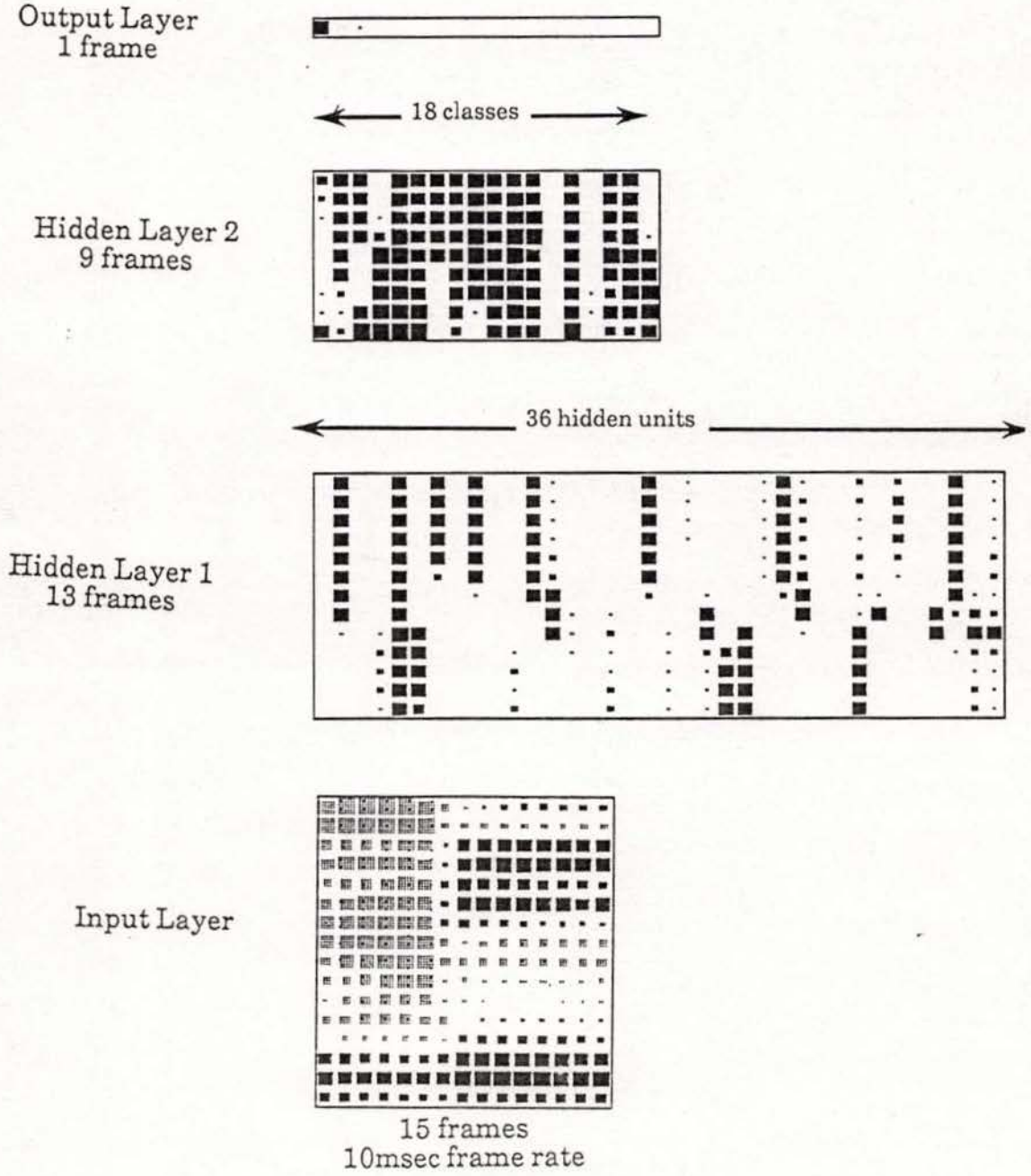


Figure 7c: Non-modular All Consonant T.D.N.N. with 36 physical hidden units.

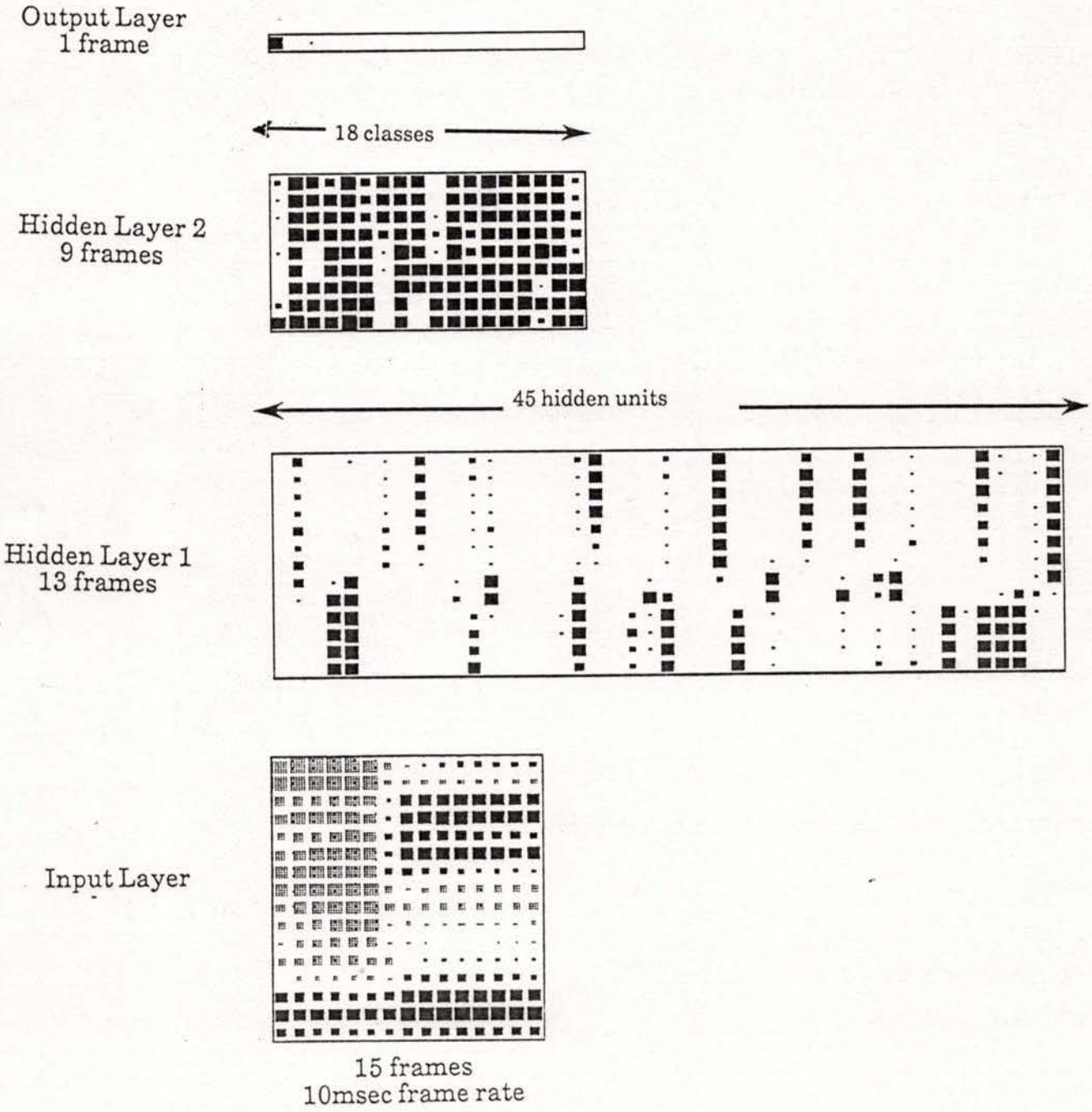


Figure 7d: Non-modular All Consonant T.D.N.N. with 45 physical hidden units.

8. Consonant recognition

This section is the last stage of our study. We show how our improved Back-Propagation algorithm enables to learn all Japanese consonants within 1 hour.

8.1 The Database.

	Original Training set	Large Training set	Small Training set	Original Testing set	Large Testing set	Small testing set
b	219	219	200	227	227	200
d	203	203	200	179	179	179
g	260	260	200	252	252	200
p	33	22	33	15	15	15
t	425	425	200	440	440	200
k	1154	500	200	1163	500	200
m	471	471	200	481	481	200
n	260	260	200	265	265	200
N	503	500	200	488	488	200
s	475	475	200	538	500	200
sh	310	310	200	316	316	200
h	214	214	200	207	207	200
z	116	116	116	115	115	115
ch	134	134	134	123	123	123
ts	212	212	200	177	177	177
r	753	500	200	722	500	200
w	72	72	72	78	78	78
y	159	159	159	174	174	174
total	5973	5063	3114	5960	5037	3061
Set Size	-	9000	3600	-	9000	3600

Table 10: Number of tokens per phoneme in our databases.

We use here the whole database available for one speaker, split equally into training and testing sets. Table 10 gives the number of tokens used for each phoneme. We see that the total number of different tokens for each set differs from the set size, as rare tokens such as /p/ are repeatedly presented.

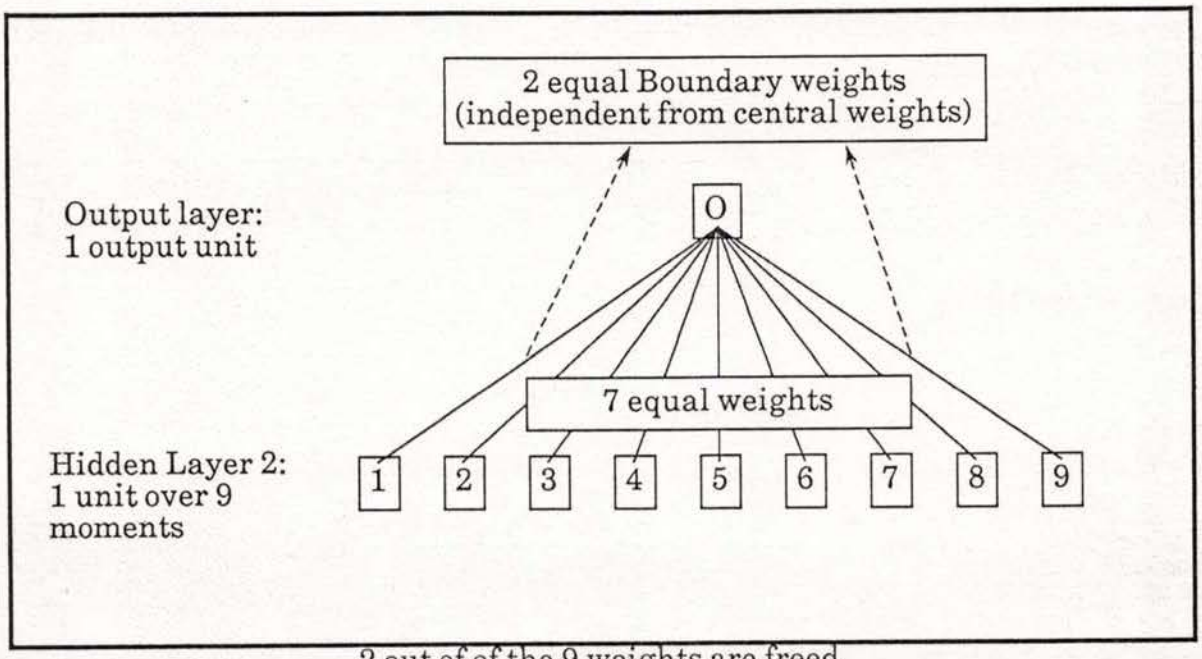
8.2. Our Learning Procedure

Among all the possible improvements presented in the previous sections, we have selected those giving the largest increase in performance, both in term of learning speed and generalization capacity.

- The standard sigmoid is used.
- The input activations are normalized to range from -1 to +1.
- As our tasks have a large number of classes, McClelland's New Error is used (section 5).
- The weights are updated according to the weight updating procedure presented in section 6. The initial updating period is 9, and is incremented by 3 at each epoch, until it reaches 72.
- Training samples are skipped when their output Error, which may range from 0 to 1 , is below 0.001. The maximum number of epochs during which they can be skipped is 5.
- The step size is limited by the overshooting control procedure (section 7). When it is not limited, the step size is 0.01 .
- The momentum is scaled by the procedure presented in section 7, with a minimum value of 0.5 and a maximum value of 0.99.

We have found that learning could be much faster when boundary effects in the summation of the last layer are taken into account. In TDNNs, the output is obtained by integrating the evidence of each unit in hidden layer 2 over time. This means that all the connections coming to an output unit must share the same weights. We have found that this configuration was not optimal, and that the boundary weights coming to the output units should be freed, as shown in the next figure.

With our TDNN networks, boundary weights tend to take a sign which is opposite to that of the central weights. We have not yet systematically explored this boundary effect in TDNNs (for instance in the other layers).



2 out of of the 9 weights are freed

8.3. Experiments on non-modular TDNNs.

Our first experiments with all-consonant recognition used an architecture derived from the 3-class architecture, but with 18 classes, as shown in Fig.7a,b,c,d. First we had to find the right number of units in the first hidden layer. The 3-class TDNN network has 8 physical units in this layer (as these are represented over 13 periods of time, there are a total of 13 * 8 = 104 TDNN units). This gives approximately 3 physical units per class. We have tried six different architectures, with different numbers of hidden units per class, as seen in table 11. (With all these architectures, the boundary weights of the last layer are free. When this is not the case, the network with 54 physical hidden units takes more than 6 hours to reach a 96% recognition rate on the training samples, which is more than ten times slower than with the network with free boundary weights).

Our recognition results after 20 and 30 iterations are shown in tables 12 and 13. We used the large training set for learning, and the small testing set for recognition. Further learning generally brings the recognition rate of the training data to 100%, with a slight loss in recognition performance on test data. These results bring very unexpected conclusions:

- While all the configurations yield a recognition rate of at least 99.5 % on training data, generalization capacity on test data seems to worsen while the number of hidden units decreases. This goes against the intuitive idea that the simplest network able to perform a given task should yield the best recognition

Number of Physical Hidden units per class	Number of physical Hidden units	Total number of connections
1	18	26388
1.5	27	39411
2	36	52434
2.5	45	65457
3	54	78480
4	72	104526

Table 11: Properties of the different networks

Number of Physical units in hidden layer 1	18	27	36	45	54	72
Elapsed CPU time (sec) with 6 CEs	1500	2000	2500	3000	3700	4800
Recognition(%) on Large training set	98.3	98.9	99.5	99.7	99.4	99.7
Recognition(%) on small test set	94.3	95.3	95.8	96.0	95.9	95.9

Table 12: Recognition rates after 20 epochs

rate. This result is consistent with what we found in Section 7 (for TDNNs, the decay on units proposed by Rumelhart reduces the number of active units without improving the generalization capacity). This may be explained by the fact that each weight contains much more information in the small network, and we see in figure 9 that the Root Mean Square over all the weights in the network is much larger for small networks. A consequence of larger weights may found when one looks at the activations of the Hidden units in Fig.7a,b: as their weighted sums are larger, units in small networks tend to have extremal activations of 0 or 1, as opposed to continuous values for larger networks (for instance the modular network in Fig.6). Fine tuning with these "boolean" activations is no longer possible.

- Moreover, it has been widely reported that it takes longer to learn when the number of hidden unit is smaller. This is not true for TDNN, and we see in

Number of Physical units in hidden layer 1	18	27	36	45	54	72
Elapsed CPU time (sec) with 6 CEs	2000	2500	3100	3600	4500	5600
Recognition(%) on large training set	98.7	99.5	99.7	99.8	99.7	99.9
Recognition(%) on small test set	94.4	94.9	96.3	95.4	96.2	95.6

Table 13: Recognition rates after 30 epochs

figure 8 that the evolution of the output error with the number of epochs is very similar for most of our architectures. As the CPU time per epoch is much shorter when there are less hidden units, learning is faster.

8.3 A T.D.N.N. modular design

In the previous architecture, we do not include any knowledge about phonetics. An adapted architecture would reduce the complexity of the high dimensional weight space and constrain learning for phonetically coherent solutions. As in [4], the following knowledge is included in the network architecture: the 18 consonants are sorted into 6 classes:

(/b/, /d/, /g/), (/p/, /t/, /k/), (/m/, /n/, /N/), (/s/, /sh/, /h/, /z/), (/ch/, /ts/) and (/r/, /w/, /y/)

Many experiments on modularity in Neural Networks are proposed in [4], to achieve consonant recognition with the large TDNNs. shown in fig.6 and whose characteristics are given in table 14. In [4], learning was performed in two main stages. During the first stage, each subnetwork was trained to recognize one consonant class, using all the training data available for this class. Then, the subnetworks were joined together and the weights were tuned for discrimination between the different consonant classes, using only the small training set described in table 10. The final recognition rate is 98.8% on training data and 95.9% on test data.

The same architecture has been tried without any staged learning, i.e.learning the training set from initial random weights is done in one single run. As shown in table 15, our algorithm only needs 1 hour to achieve a 96.7 % recognition rate

	Input	Layer 1	Layer 2	Output	Total
Units	241	884	216	18	1359
Physical units	16	68	24	18	126
Average fanin	0	49		19	
Connections	0	43316	12099	342	55757
Physical conns	0	3332	1350	48	4730

Table 14: Network numbers for the modular all-consonant network

on test data. Modular architecture seems to perform better than the non-modular architecture whose best performance is 96.3% (Table 13).

To check the influence of the training set size, the network has also been trained with the small set. Performance on test data is 95.3% as opposed to 96.7% on the large set. It is interesting to notice that the results obtained with staged learning (95.9%) lie between these two values, as both the large and the small training sets have been used: the former to train the subnetworks and the latter to tune the large all-consonant network.

Training set	Recogition rate on train samples	Recogition rate on test samples (small set)	CPU time
Small	99.4	95.3	1 hour
Large	99.2	96.7	1 hour

Table 15 : recogntion rates with modular Neural Networks.

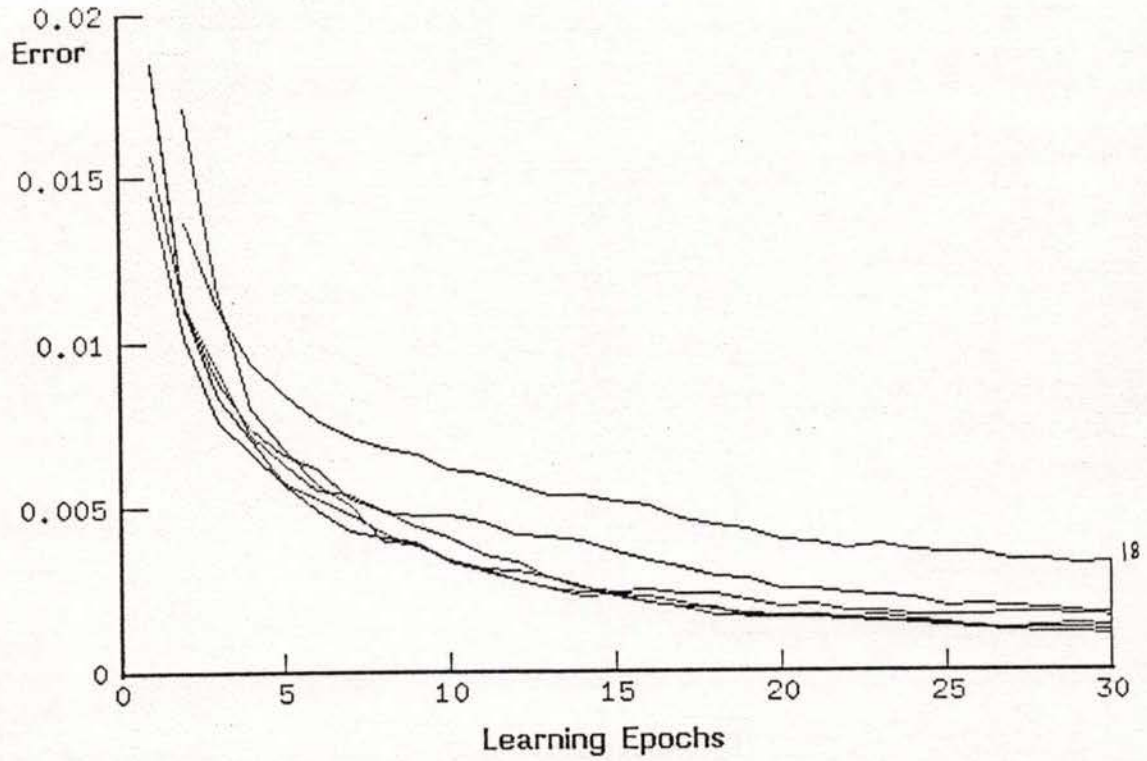


Figure 8 : Error Vs. Learning Epochs for All Consonants TDNNs with 18, 27, 36, 45, 54 and 72 hidden units.

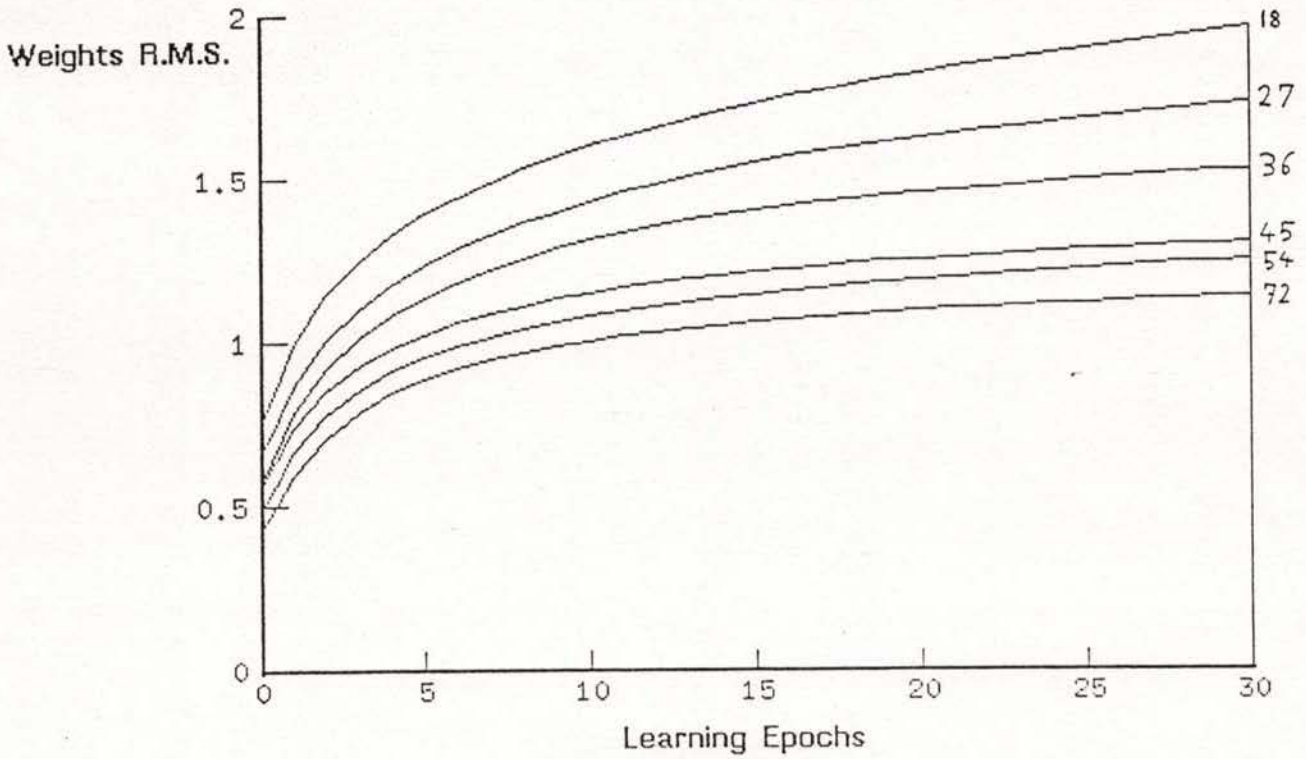


Figure 9 : Root Mean Square Weight Vs. Learning Epochs for All Consonants TDNNs with 18, 27, 36, 45, 54 and 72 hidden units.

9. CONCLUSION

9.1. Contributions

We have shown in this report that learning speed for the Back-Propagation procedure could be reduced to a large extent, thanks to improvements on the *modelling of the Error surface, learning strategy and control of the weight modifications*. Even though we have tried this methods mostly on TDNN, we have selected algorithm with sufficient robustness to work for a large variety of tasks.

This increased speed and the use of parallel computers have enabled us to obtain systematic results on large tasks and study problems which are not in the range of toy tasks, such as the classification performance on open data.

Modular design has been recommended for TDNNs to handle large speech tasks. The improvements proposed here allow an increase by a factor of 10 of the task size (number of training samples, connections and classes) of the basic modules proposed in [4].

9.2. The quest for generalization

We have found that an increase in learning speed very often comes with a loss in generalization capacity. With most of our algorithms, to tune parameters was to trade off speed for better generalization and we have found that generalization capacity becomes worse for large average weight values. A more precise definition of generalization is needed to control it.

With discrete tasks such as the 838 encoder-decoder, the problem is to train the network with only 7 of the 8 samples, and attain a configuration able to classify correctly the last one. This is achieved by using a minimal number(3) of hidden units, which guarantees an optimal internal representation. The number of hidden units may be automatically optimized through weight decay[18].

We have seen that this method does not seem to work well for classification of continuous patterns such as speech data in TDNNs. Bad generalization on open data is not necessarily due to a bad internal representation. Connection weights may still need some fine tuning, to optimize the boundaries between classes in the high dimensional input pattern space. From this viewpoint, the increase of generalization capacity from LVQ to LVQ2[9, 10], thanks to a fine tuning of the class boundaries, is very instructive. Similar methods would be useful for Back-Propagation.

These two remarks are consistent with the assumption that the connection weights should be small. With the former, many weights have to be equal to zero, and with the latter, weights have to be small to allow a fine tuning in the linear part (around 0) of the sigmoid function.

9.3. Future Prospects

In this work, some questions have remained partially unanswered. Among them how to choose the architecture and the initial conditions, when to stop learning and how to find an optimal step size for each unit. To solve these problems, we are thinking about two general categories of solutions.

The first one would insist on the theoretical aspect of learning in Neural Networks. Many mathematical models have already been proposed to set the parameters, but they are generally based on assumptions about linearity that are never verified and owe their success to their empirical performances. However, the fact that, in spite of their poor consistency, they sometimes dramatically improve performance, give us hopes that they are very rough approximations of models that are still to come.

The second and more pragmatic solution is to consider learning as a process too complex to formalize in great details. However, very general rules about the behavior of the network have been observed. The idea would be to integrate them in an Expert System. The latter would be able to restart the system if it finds something has gone wrong, to stop learning when some significative signs appear (for instance, the error reaches a plateau or overshoots), to give a complete diagnostic of what has happened during a learning run, or to modify the architecture of the network, even while learning.

APPENDIX A: Scaling the step size

Gradient descent methods theoretically requires an infinitely small step size, which is not realistic. Most methods which have been proposed to find an optimal step size generally deal with line search or Newton algorithms[13].

1) Line search.

Global line search requires a very time consuming heuristic, as several step sizes have to be tried for each iteration. However, this method may lead to subsantial gains in learning time by reducing the number of epochs[7].

2) Error zero-points Search

Many Error zero-points search algorithms have been proposed recently, yielding faster learning speed while reducing the number of parameters to tune, for one does not have to choose a step size or a momentum.

2.1) Global Error zero-points Search.

This finds the zero points of the Error considered as a linear function of the weights. Schmidhuber[19] found a substantial gain in speed for the NetTalk learning task. This method needs some alterations to work on TDNNs.

2.2) Zero point search local to units.

The problem becomes linear and a value for epsilon can be derived which minimizes a local energy. The computation is complicated and the result is far from the optimal step size we practically find, all the more soif we introduce a momentum. Moreover, using such a method, we have found that each unit tends to try to learn the solution at any price and forgets that the network is able to generate an alternative representation if the first one seems impossible to learn.

2.3) Zero point search local to connections.

The Quickprop procedure proposed by Fahlman[12] is local to each connection. With control parameters to avaoid divergence, it reportedly improves learning speed with the encoder problem by one order of magnitude, compared with standard Backprop. However this algorithm seems to require the computation of dEd/w over one epoch, as it uses in a critical way the difference between two consecutive values of dEd/w. This does not seem suited to our tasks where two consecutive dEd/w is represent very diffrent data and therefore be meaningless.

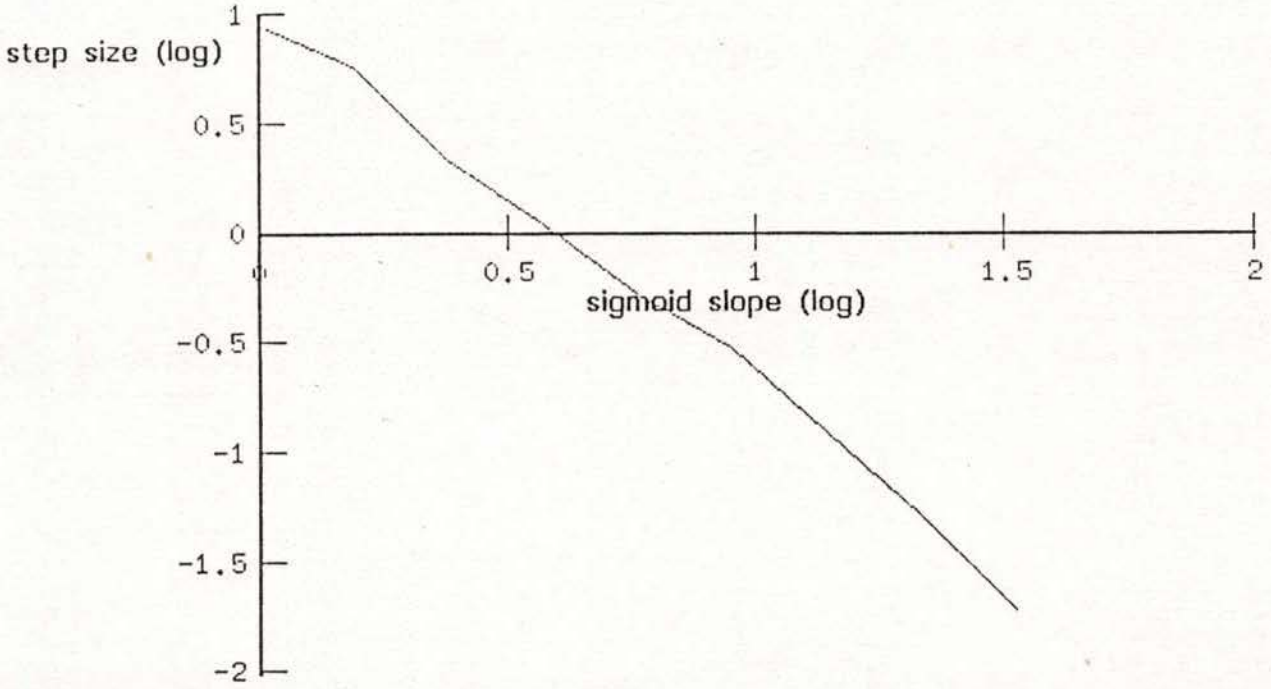


Figure 10 : Optimal Step Size Vs. Sigmoid Slope (XOR task)

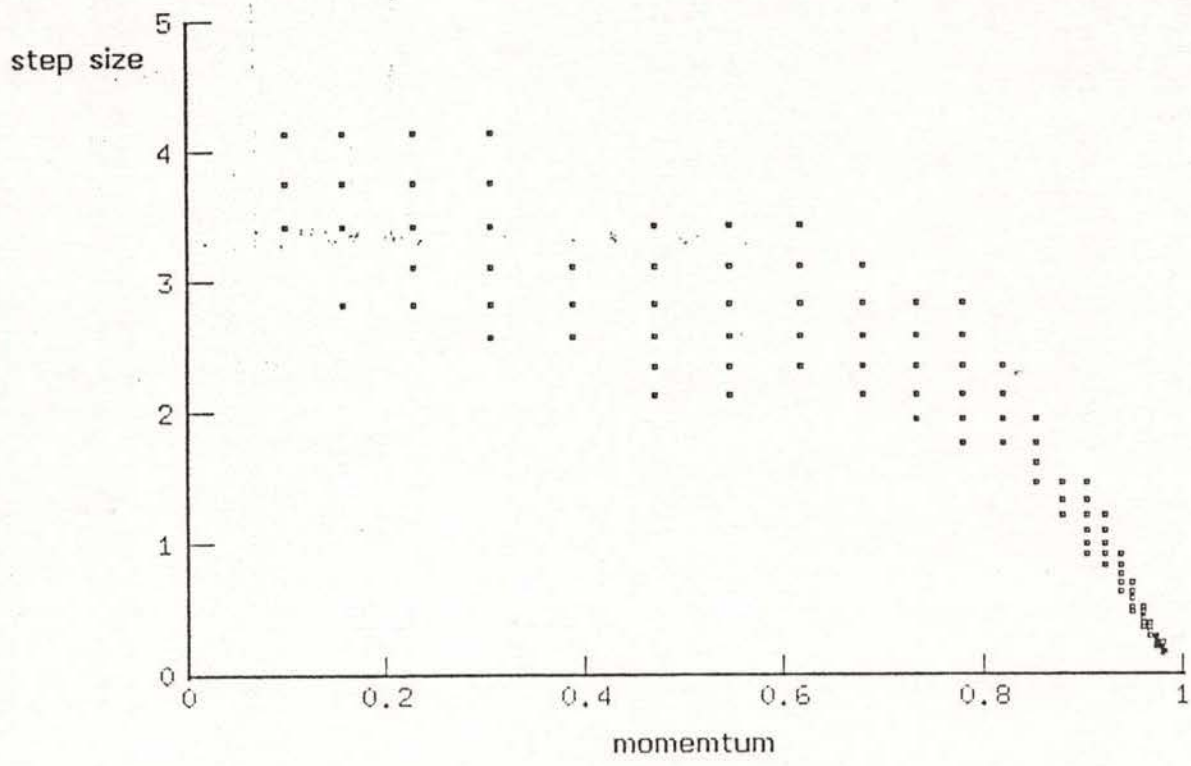


Figure 11 : Optimal Step Size Vs. Momentum (838 task)

3) Some observations

However, some observations may be very useful:

- If we call the momentum α, we should have $\varepsilon = cst\,(1 - \alpha)$. This relation only works well when $\alpha > 0.9$ (Fig 11). However, ε is always a decreasing function of α.

- For a given problem, if M is the number of samples, then $\varepsilon = cst/M$. This relation verifies very well on Fig 12. This is useful to find the step size for large training sets: we determine the optimal step size for a smaller set and then resize it to learn the larger set.

- ε depends on the sigmoid function according to the relation:

$$\varepsilon = \frac{K}{(f_{max})^2\ (f'_{max})^2}$$

We call the sigmoid slope s such that $f(x) = s/(1+e^{-x})$, we must then have:

$$\log(\varepsilon) = -4\log(s) + \text{constant}$$

This is well illustrated in Fig.10.

4) Dynamical Scaling of the step size

It has been shown that the optimal value of the step size may vary widely with time, which is self consistent with the large variations of slope and curvature on the energy surface. As a gauge of these variations, Franzini[16] has proposed the cosine of the angle between the error derivative at epoch t and that at epoch t-1.

$$\cos(\theta) = \frac{\sum_{i \in C} d0_i\, d1_i}{\sqrt{\left(\left(\sum_{i \in C} d0_i\right)\left(\sum_{i \in C} d0_i\right)\right)}}$$

$$where\ \ d0_i = \frac{\partial E}{\partial w_i}(t-1)\ \ and\ \ d1_i = \frac{\partial E}{\partial w_i}(t)$$

C is the set of connections of the network. The algorithm works in a very simple way, adapting the step size to the shape of the energy surface:

- Plateau : the learning trajectory is straight and $\cos(\theta) > 0$, we can accelerate the learning rate.

- Ravine : learning may give way to unwanted oscillations, in the worst case, we may miss the minimum we are looking for. Therefore, when $\cos(\theta) < 0$, we have to slow down the learning rate.

Following these ideas, we have tried several epsilon updating rules and kept to the simplest:

$$\varepsilon(t) = e^{u.\cos(\theta)}\ \varepsilon(t\text{-}1).$$

The choice of u is a trade-off between fast adaptation and few oscillations :

$u = 0.1$ seems optimal, as shown in Fig.13.

Moreover, it is useful to set extrema for ε. When learning with a momentum, we may set $d0_i = -\Delta w_i$ and $\theta = \text{Angle}(\Delta w(t\text{-}1), -\nabla E(t))$. Epsilon adaptation may be slower, but many oscillations are prevented.

We have seen before that the optimal step size generally depends on a given unit. If we take $C = C_u$ the set of input connections to unit u, the algorithm becomes local to this unit, updating the local step size ε_u. In our task where units may have very different roles, we have found a large increase in learning speed using this method. ε may vary by a factor of 100 from one unit to another, depending mostly on the layer. This is probably due to the fact that learning dynamics vary widely from one layer to another.

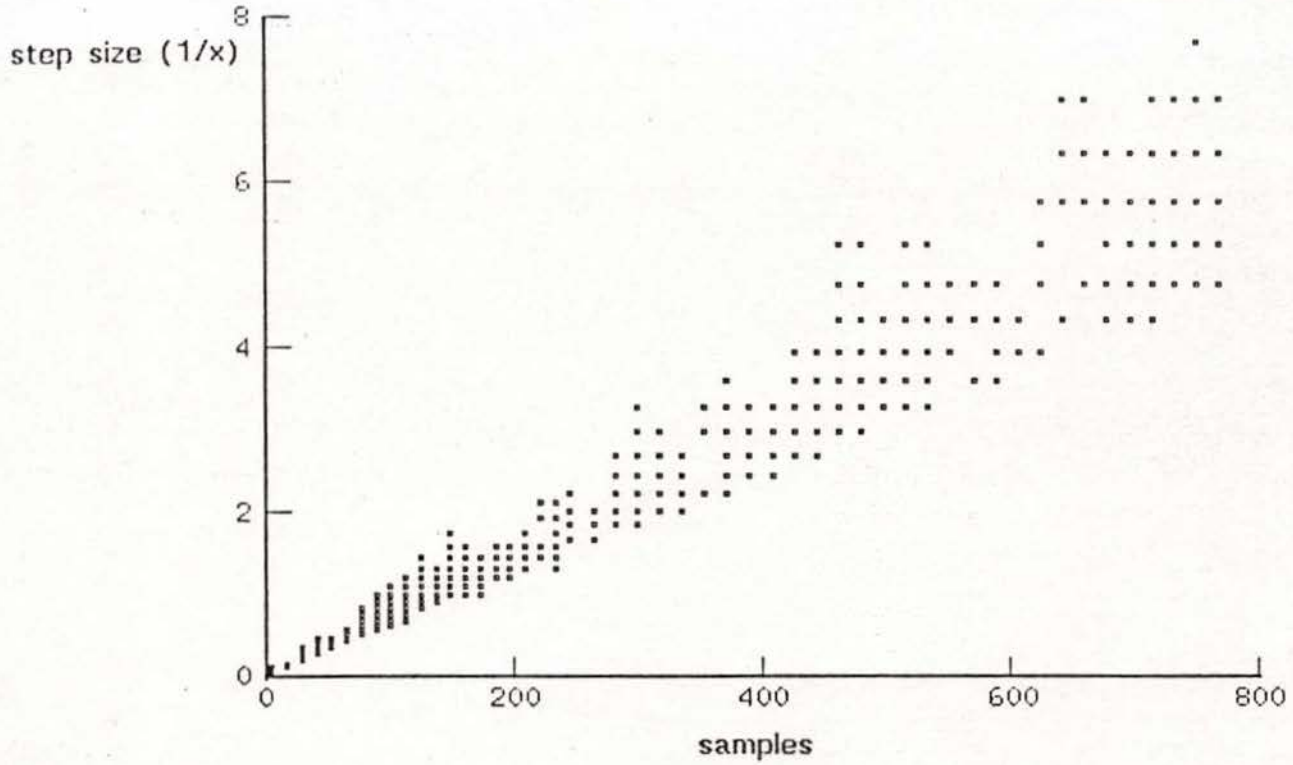


Figure 12: Optimal Step Size vs. Number of Training Samples (BDG task)

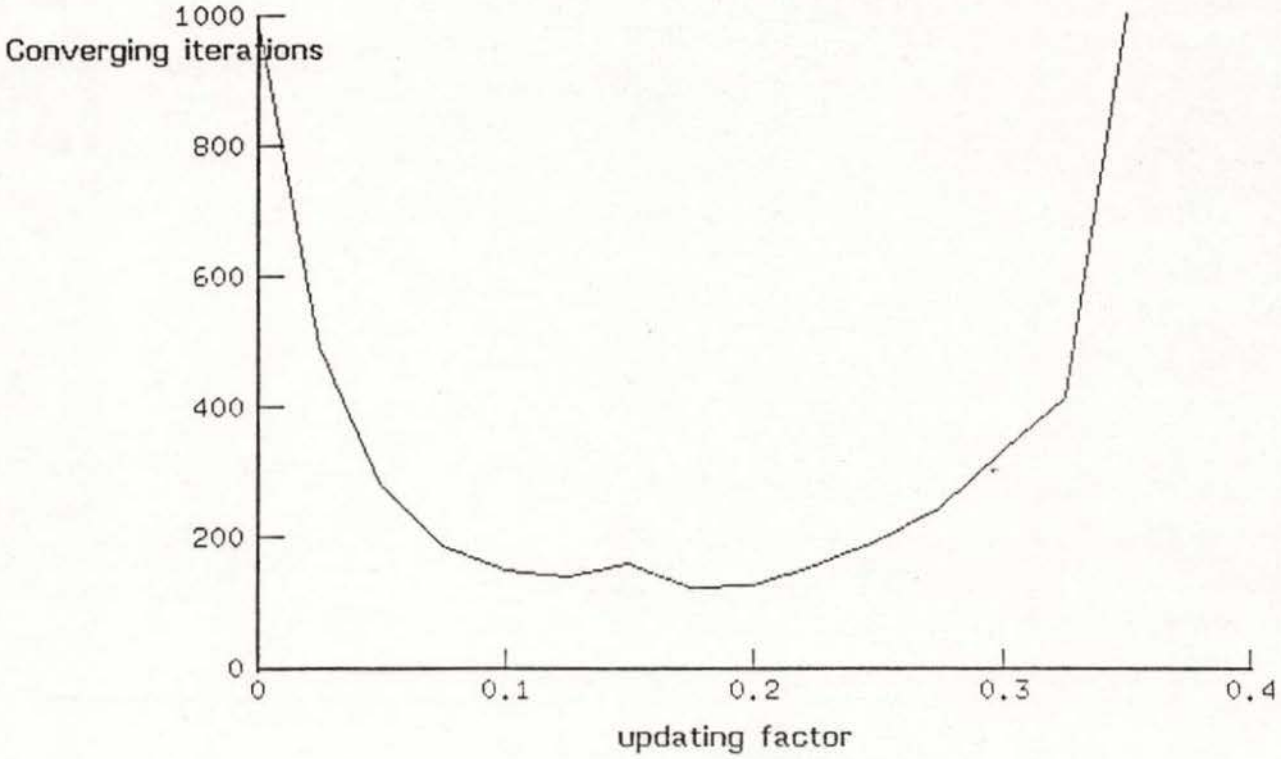


Figure 13: Number of converging Epochs vs. Step Size updating factor (838 task)

Acknowledgements

This work constitutes the internship report of one of the authors, Patrick Haffner, who would like to express his gratitude to all the people in the ATR Interpreting Telephony Research Laboratories who have helped him during this internship.

I would like to thank Dr. Akira Kurematsu, president of the ATR Interpreting Telephony Research Laboratories , for his enthusiastic support for research on Neural Networks.

I am also very grateful to Dr. Kiyohiro Shikano, Head of the Speech Processing Department, who gave me friendly scientific advice and opportunities to meet members of the Japanese scientific community. This internship would have been impossible without the dynamism of Dr. Hisao Kuwabara, Supervisor of the Speech Processing Department, who has done a lot for international scientific cooperation.

It has been for me a wonderful experience to work with Dr. Alex Waibel, who made scientific endeavor always fun, and with Dr. Hidefumi Sawai, who fed my TDNN simulation program with difficult but stimulating speech problems. The very dynamic "Neural Networt team" of ATR has many other members I wish to thank for their constant help and kindness.

I am also indebted to many in the Speech Processing Departement at ATR, for their help in the various stages of this research.

Special thanks to Erik McDermott, for checking this report. Moreover, this work owes a lot to a friendly scientific competition with him and to many fruitful discussions.

References

[1] D.E. Rumelhart and J.L. McClelland. *Parallel distributed Processing; Explorations in the Microstructures of Cognition.* Volume I and II, MIT press, Cambridge, MA, 1986.

[2] A. Waibel, T. Hanazawa, G. Hinton, K. Shikano and K. Lang. Phoneme Recognition Using Time-Delay Neural Networks. *Technical Report TR-1-0006, ATR Interpreting Telephony Research Laboratories, October 1987.*

[3] A. Waibel, T. Hanazawa, G. Hinton, K. Shikano and K. Lang. Phoneme Recognition Using Time-Delay Neural Networks. *IEEE transactions on Acoustics, Speech and signal processing, 1988.*

[4] A. Waibel, H. Sawai, K. Shikano. Modularity and Scaling in Large Phonemic Neural Networks.*Technical Report TR-1-0034, ATR Interpreting Telephony Research Laboratories, August 1988.*

[5] H.Sawai, A.Waibel, K.Shikano. Spotting Japanese CV-Syllables by Time-Delay Neural Networks. *Proc. from ICASSP 89, Glasgow, May 1989 (to be published)*

[6] P.Haffner, A.Waibel and K.Shikano. Fast Back-Propagation Methods for Neural Networks in Speech. In *Proc. from the Fall Meeting of the Acoustical Society of Japan, October 1988.*

[7] N.Nakamura, K.Shikano. A study of English Word Category Prediction Based on Neural Networks.*Technical Report TR-1-0052, ATR Interpreting Telephony Research Laboratories, November 1988.*Also in *Proceedings of the Second Joint Meeting of ASA and ASJ, Hawaii, November 1988.*

[8] Y. Sagisaka, K.Takeda, S.Katagiri, H.Kuwabara. Japanese Speech Database with Fine Acoustic-Phonetic Transcriptions. *Technical Report TR-1-0006, ATR Interpreting Telephony Research Laboratories, May 1987.*

[9] E. McDermott, S. Katagiri. Shift-Tolerant, Multi-phoneme Recognition Using Learning Vector Quantization. In *Technical Report SP-88-80 of the Institute of Electronics, Information and Communications Engineers, Tokyo, October 1988.* Also in *Proceedings of the Second Joint Meeting of ASA and ASJ, Hawaii, November 1988.*

[10] T.Kohonen, G.Barna, R.Chrisley. Statistical Pattern Recognition in Neural Networks: Benchmarking Studies. *IEEE, Proc of ICNN, Vol I, pp 61-68, July 1988.*

[11] W.H Huang, R.P.Lippmann. Neural Net and Traditional Classifiers. In *Proc of the Conference on Neural Information Processing Systems, Denver, Colorado.*

[12] S.E. Fahlman. An Empirical Study of Learning Speed in Back-Propagation Networks. *Technical report CMU-CS-88-162, Carnegie Mellon University,* June 1988.

[13] R.L. Watrous. Learning Algorithms for Connectionist Networks: Applied Gradient Methods for Non-Linear Optimization. In *Proceedings of the IEEE International Conference on Neural Networks,* pages 619-627. San Diego, CA, 1987.

[14] J.P. Cater. Successfully Using Peak Learning Rates of 10 (and greater) in Back-Propagation Networks with the Heuristic Learning Algorithm. In *Proceedings of the IEEE International Conference on Neural Networks,* pages 645-651. San Diego, CA, 1987.

[15] W.S. Stometta and B.A. Huberman. An Improved Three-Layer Back-Propagation Algorithm. In *Proceedings of the IEEE International Conference on Neural Networks,* pages 637-644. San Diego, CA, 1987.

[16] M.A. Franzini. Speech recognition with Back Propagation. In *Proceedings, Ninth Annual Conference of IEEE Engineering in Medecine and Biology Society. 1987.*

[17] C. Kamm, T. Landauer, S. Singhal. Training an adaptative Network to Spot Demisyllables in Continuous Speech. In *proceedings of the ATR Workshop on Neural Networks and PDP, July 1988, Osaka.*

[18] D.E.Rumelhart, Learning and Generalization: The Role of Minimal Networks. In *proceedings of the ATR Workshop on Neural Networks and PDP, July 1988, Osaka.*

[19] J. Schmidhuber: Accelerated Learning in Back-Propagation Nets. *Technical Note, Institut fur Informatik Technische Universitat Munchen, May 1988.*

[20] J. Pabon, D.Gossard: A Methodology to select appropriate Learning Rate Parameters in Feed-Forward Networks. In *Proc. of the 3rd Annual Aerospace Applications of Artificial Intelligence Conference, Dayton, Ohio, October 1987.*

[21] L.Y.Bottou. Reconnaissance de la Parole par Reseaux multi-couches. In *Neuro-Nimes, Nimes, France, November 1988.*