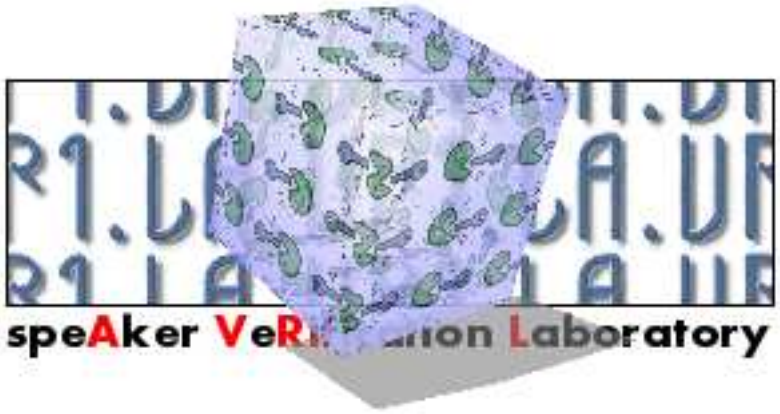
A.VRI.L

speAker VeRIfication Laboratory



An MLP based approach

Amos Brocco
e-mail: amos.brocco@unifr.ch

30th July 2003

Bachelor Project 2002-2003

Department of Informatics, University of Fribourg, Switzerland

Supervised by Prof. Dr. Dijana Petrovska

Abstract

This paper investigates an approach in speaker verification using a multi-layer perceptron's neural network trained with Mel Frequency Cepstral Coefficients (MFCC) parametrization from two classes: one of client speaker voice and the other from of other speakers . The role of the trained MLP is then to discriminate between these two classes by returning the percentage error of a given input versus each class. Many experiments were performed in order to find out the best neural network configuration.

Contents

1 Introduction: What is A.VRI.L ?

A.VRI.L is a project aimed to create a speaker verification laboratory based on a Multi Layer Perceptron's (MLP) neural network. Traditionally this task is performed using Hidden Markov Models, (HMM) or with the HMM-MLP combination. Speaker verification will then be divised in three parts: first a model of the speaker (MFCC) is created using the HTK [1] tool. The MLP network is trained with this file and a combination of several "non-client" speaker models (also called "world"). One of the difficulties of this approach is the modest amount of client data available, compared to world data, which can be bypassed by supplying several times the same client model. Once the MLP is trained for a given speaker, its internal node configuration is saved to a file, which can be later used to perform speaker verification. It's important to clear distinguish speaker verification from speaker identification: this project is focused on the first task, that is, distinguish between a client speaker and impostor not to resolve speaker identity. The development of this type of speaker verification techniques could follow two ways; the first way, on which this project focuses, analyzes a global modeling and training approach: coding of an MLP (using the Torch 3 [2] library) and training using global MFCC. A global MFCC is intended as a global modelization of speaker voice. The other way would have provided a more complicated approach using multiple MLP for each phonetic class: voice modeling would have to be classified accordingly to phonetic classes and then, data for each classes would have been used to train an MLP. Speaker verification would have then been the result of multiple verifications conducted on different phonetic classes followed by a "score" recombination. As said before, this project only focuses on the first way, this means that only the global parametrization and discrimination approach has been subject of study and has been developed; a segmentational approach, starting from what has been done should not be very hard, considering that the most critical part (the development of the neural network software) is working and performing very good. [1]

2 Speaker verification technique

Human voice is determined by a number of factors: in order to perform a speaker verification we first need to a way to parametrize the voice (i.e. extract characteristic voice informations): there are many different ways to do that, the one that will be used in A.VRI.L is based on the so called Mel Frequency Cepstral Coefficients, which are the standard representation used by the HTK toolkit. Speaker modeling was not directly performed for this project, because for the development and research phase a pre-built set of MFCC files have been chosen (this choice was imposed in order to be able to compare results obtained with other methods). Once we have access to the speaker model, we can train a custom built MLP to distinguish (discriminate) two types or classes of speakers: the good one (the client or "true" speaker) and the bad ones ("world"). As training is finished, the MLP is ready for speaker verification. Note: as this is a research project it is not yet clear if it will give good or bad perfomance (compared to existing methonds) or if it will be reliable for general use. In the next section an almost detailed description of the MLP is given.

[1] Note: in order to better understand things presented in this text, the reader is invited to browse the source code (as this code is distributed under the GNU/GPL license) found on cdrom.

3 Project roadmap

This bachelor project started in the month of November 2002; since I was completely stranger regarding speech and voice technology and neural networks, it wasn't until end of december 2002 that I started developing something.

Here's a brief roadmap of this "nine months long" bachelor project:

- Start of November 2002: First meeting with Dr. Dijana Petrovska and discussion about the subject.
- November and December: Documentation phase about speaker verification and neural networks.
- January 2003: Start of development of the neural network.
- February: First almost working neural network, but some problems with MFCC input.
- March: Neural network working and correcly loads MFCC files.
- April - May: Speaker verification tests with bad results. Start of the improvement subproject.
- June: Male-Female discrimination tests with poor performance.
- End of June, July: MLP-MFCC problem found and solved, resulting in good Male-Female discrimination results as well as pretty good results with speaker verification

After the development and testing phase it was time to collect results and write this report; there is still much space for improvement so results presented later on in this document are not the very best.

4 The software

As it has been said before, the main part of this project is the development of the neural network. In this section a description of the neural network is provided; this part should help the reader understanding how the main software part of A.VRI.L works, but could also serve as an help for developers who would like to "hack" on the code or just understand basically how to deal with the Torch library. As in this section only small pieces of code are listed, which are written in C++, I suggest browsing the complete code for better understanding, as well as reading some Torch documentation, found on their website[2, 5]; the latter section describes the additional software developed with Python to perform automated testing sessions and to manage input data.

4.1 The neural network

A.VRI.L uses a multilayer perceptrons network that is trained to distinguish between the "true" speaker and the impostor speakers (also named "world"). In this section we will give a detailed analyse of the MLP, which is based on the Torch C/C++ library (selected pieces of code are listed). Neural network programming was the most problematic part of the programming phase: as I'm going to better explain later in this text, much of the problem encountered during the practical part (i.e. testing) of this project, were due to a bad implementation (due to a lack of documentation in the Torch 3 Library package) of a specific part of the MLP.

4.1.1 Main concepts of Torch

In order to better understand what follows, I'll present here the four important concepts of the Torch library (freely taken from the Torch Tutorial[5]):

> "There are only four important concepts in Torch. Each of them are implemented in a generic class. And almost all classes of Torch are subclasses of one of them. Here they are:
>
> **DataSet**: this class handles the data. Sublcasses could be for static or dynamic data, for data that can fit in memory or on disk, etc...
>
> **Machine**: a black-box that, given an (optional) input and some (optional) parameters, returns an output. It could be for instance a neural network, or a mixture of gaussians.
>
> **Trainer**: this class is able to train and test a given machine over a given dataset.
>
> **Measurer**: when given to a trainer, it prints in different files the measures of interest. It could be for example the classification error, or the mean-squared error.
>
> The *general idea* of Torch is very simple: first, the **DataSet** produces one "training example"...
>
> This training example is given to a machine which computes an output...
>
> ...and with that a trainer tries to tune the machine.
>
> As your surely begin to understand, for a given machine, you need a special trainer.
> Usually when you create a new class of machine learning machine, you have to write the corresponding trainer. Examples:

- there are many "gradient machines" (**GradientMachine**) (including a multilayer perceptron) which can be trained using a "gradient machine trainer" (**StochasticTrainer**)
- (...)"

4.1.2 The core of A.VRI.L MLP

The MLP is composed of four layers: the input's linear layer, the hidden sigmoidal layer, the intermediate output linear layer and the log-softmax output layer. This configuration was suggested by literature, to be used for basic classification/discrimination; additional research about the configuration of the neural network falls outside the project scope, so no changes were made to this configuration. Furtherwise, adding more than one hidden layers would not improve performances of the MLP, because, as Kolmogorov theorem says, an MLP with one hidden layer of sufficient size can approximate any continous function to any desired accuracy.

A.VRI.L make use of a supervised training approach, i.e. using a training by example method, additionally it uses a backpropagation learning model (outputs are back-propagated during training).

A "mathematical oriented" (but simple) explaination of what the MLP is supposed to do is the following:

> *"Given a group of examples (**speaker models**) represented by vectors (**of a MFCC parametrization**), each described by a sequence of values, whose class is known (**either "client speaker" or "world"**), the*

> *goal of the neural network is to find a link between a vector and the corresponding class (**during training**), then estimating the probability that a given example (**given by vectors**) is part of a class (**testing phase**).*"

Let's have an "in-depth look" at "how" the multilayer perceptron has been coded using the Torch 3 Library (the code presented here came from the latest version [4], which is currently used). It's important to mention that, as Torch is a C++ library, all parts of the neural network are represented by objects[2]; for example the expression

```
ConnectedMachine avril_mlp;
```

creates an instance of a **ConnectedMachine** class named **avril_mlp**; the class ConnectedMachine is the core of the multilayer perceptron: all other parts (i.e. the layers) are built on top of it. Layers are created almost in the same way:

```
Linear *c1 = new(allocator)
             Linear(int(data_htk->inputs->frame_size), hidden);
Sigmoid *c2 = new(allocator) Sigmoid(hidden);
Linear *c3 = new(allocator) Linear(hidden, NOUTPUTS);
LogSoftMax *c4 = new(allocator) LogSoftMax(NOUTPUTS);
```

Layer objects require additional parameters as number of inputs and outputs (as shown in the above code). The number of input nodes corresponds to the size of the input vector, also called frame, which, in our case, is 26. By default the number of outputs is 2, as it's the number of classes required for our speaker verification purposes (either the "client speaker" class or the "world" class). For this project a number of 13 hidden units has been chosen for the MLP (suggested by literature), but this number has to be changed depending on data size (a large number of hidden units will provide better verification quality but also result in poor performances and requires much training data). Next it's time to connect every object to each other, by adding layers to the ConnectedMachine object:

```
avril_mlp.addFCL(c1);
avril_mlp.addFCL(c2);
avril_mlp.addFCL(c3);
avril_mlp.addFCL(c4);
```

Finally we need to create intra-layer connections (each layer has a number of connections from and to adiacent layers):

```
avril_mlp.build();
```

As we've seen, building the core section of the MLP is pretty simple; next task is to code the data I/O part, i.e. functions that load training and testing data to "feed" the neural network core.

4.1.3 Dealing with data

As it has been said, input data has to be in HTK format: the HTK toolkit can produce a variety of files starting from sound data (in this case the voice), but the one that are used in this project are those containing Mel Cepstral Coefficients, formely named MFCC (you'll find a detailed description of this data format and

[2] To better understand the code I suggest reading the Torch Tutorial (available at [1]) or browsing the class hierachy (althrough not well documented) found on Torch's website.

parameters used for conversion in section 4.1). Hopefully the Torch 3 library can deal directly with this file format[3], so no further data conversion is necessary. In this section I'll give an overview of the basic method to gain access to MFCC vector sequences, because, despite the fact that the MFCC format is well supported by Torch, loading data is not a trivial thing.

The neural network has to load many files during a training session, either files of the client speaker or files from the world class; inside the Torch library the basic object used to manage data is the **Sequence**: loading data consist in creating sequences of vectors for inputs and outputs. First, for each file, MFCC vectors (containing 26 values) are extracted and inserted into an input sequence:

```
data_htk = new(allocator) HTKDataSet(input_files[k],
               true, -1 , NULL, NULL, false);
data_htk->setExample(0);
```

Next, for each input vector, an output one of size two is associated to; the output vector is used as identifier for the appartenance class: 0 1 represents class 1 (for the client speaker), 1 0 represents class 2 (for the world). In the testing phase this identifier is set to 0 0, but in fact it is ignored by the neural network. This output sequence, of the same size (i.e. number of vectors) as the input one, is created on the fly:

```
data_CLASS = new(allocator) Sequence(number_of_vectors,2);
for(int v = 0; v < number_of_vectors; v++) {
    data_CLASS->frames[v][theclass] = 1;
    data_CLASS->frames[v][1-theclass] = 0; }
```

(Note: the allocator object shown above is used by Torch to manage global memory allocation)

Finally, the input and output sequences of each files are merged together to form the two main sequences: one containing the input vectors, the other containing the output identifiers. This part has shown to be the most critical one regarding the memory space needed: as it should be easy to understand, each vector contains 26 real values, each requiring 32 bits; as there is a vector for every 10 ms of voice data, the total memory required could raise up to several hundred of Megabytes.[4]

```
for(int u = 0; u < n_train; u++) {
    data_input[(...)] = new(allocator)
                        Sequence(1, data_htk->n_inputs);
    data_input[(...)]->copyFrom(data_htk->inputs->frames[u]);
    data_target[(...)] = new(allocator) Sequence(1, 2);
    data_target[(...)]->copyFrom(data_CLASS->frames[u]); }
```

As you can see by browsing the complete source code[4], there is additional code to manage balancing of vectors (that means, loading the same number of vectors for each class) and other small features, not presented in the above code.[5] Next task in data loading is to merge the complete sequence of inputs and outputs to create a big main input DataSet; as written before, the only way a Machine can load data is through a DataSet object.

[3] In fact the Torch 3 Library was choosen because of this ability

[4] Refer to section 4.1 for details

[5] A complete code review would have fall outside the scope of this text. Nevertheless the complete source code is well commented and the reader is encouraged to take a look to it.

```
data_mem = new(allocator) MemoryDataSet();
data_mem->setInputs(data_input,sequence_size);
data_mem->setTargets(data_target,sequence_size);
```

As soon as the input DataSet is created a preprocessing of the data is performed: this is needed because the "scale" of vectors have not to be a discriminating factor during training nor during testing. Data is processed by normalizing all the sequences (of frames, vectors) contained in the input data in a way that the variance of the ensemble of all frames is 1 for all frame columns, and the mean is 0 (by dividing by the standard deviation). It's important to note that the normalization values applied during the training phase are saved to the created MLP model and used for preprocessing of the testing data.

Additional preprocessing is needed to set the number of classes associated to the input data as well as the data format, but I will not get into details as these are to be considered as specific implementation tricks proper to the Torch library (the use has been suggested by reading the Torch tutorial, but no investigation about their meaning has been made).

It may seems that data management with the Torch library is a little bit tricky: this observation is partially true, but after considering the powerful capabilities of this library and the extreme complexity of correct memory management in C++, this is a price that is worth to pay.

4.1.4 Measurers, criterions and the trainer

In order to get an output from the neural network, measurers are needed: measurers are used to compute the classification error of the given inputs (vectors) compared to the current targets. Output is saved to a file and is used to make decisions about a data test. Output values are close tied to the layer configuration: in fact as the output layer is a LogSoftMax, what we get is a logarithm of the error.

```
ClassMeasurer *class_meas = new(allocator)
ClassMeasurer(avril_mlp.outputs, data, class_format,
                    cmd.getXFile("the_class_err"));
measurers.addNode(class_meas);
```

Finally we need a trainer for the MLP: training is done using a stochastic gradient trainer. This trainer takes the gradient machine (as our MLP) and trains it with a gradient descent algorithm (i.e. data and node output will be propagated through the neural network); as it has been said before, back-propagation is also used. In order to work, A.VRI.L trainer has to know how much it has to train the MLP: it's important to set stop conditions for training, and that is done by the criterion. A.VRI.L make use of a negative log-likelihood criterion, provided by the Torch library: during training the trainer will try to maximize the NLL. Additionally it's possible to set up others training stop conditions, such as end accuracy (the difference between the previous error and the current error), the learning rate, the learning rate decay or the maximum number of iteration.

```
criterion = new(allocator) ClassNLLCriterion(class_format);
StochasticGradient trainer(&avril_mlp, criterion);
trainer.train(htk_data, &measurers);
```

Another possibility is to train using cross validation, to avoid "over-training" of the MLP. This is done by selecting a subset of available speaker models (approximatively the 20%) which class appartenance is known, to validate learning results. Data selection for cross-validation is done in the Python script in a random way.

Once the training has been performed, we can provide test examples to the MLP (which training state has been stored to a file) to perform speaker verification, which are loaded the same way as training data.

4.1.5 How stuff works

During training, the neural network loads the provided input MFCC files (containing parametrization of a client speaker voice and of various speakers voices forming the "world") and extracts all the vectors, associating them with the corresponding class/target (class 1 for vectors which belong to the client, else class 2). Data is then preprocessed and normalized; next the trainer loads this data into the MLP core and adjust the node parameters accordingly to input and targets (*figure 1*).

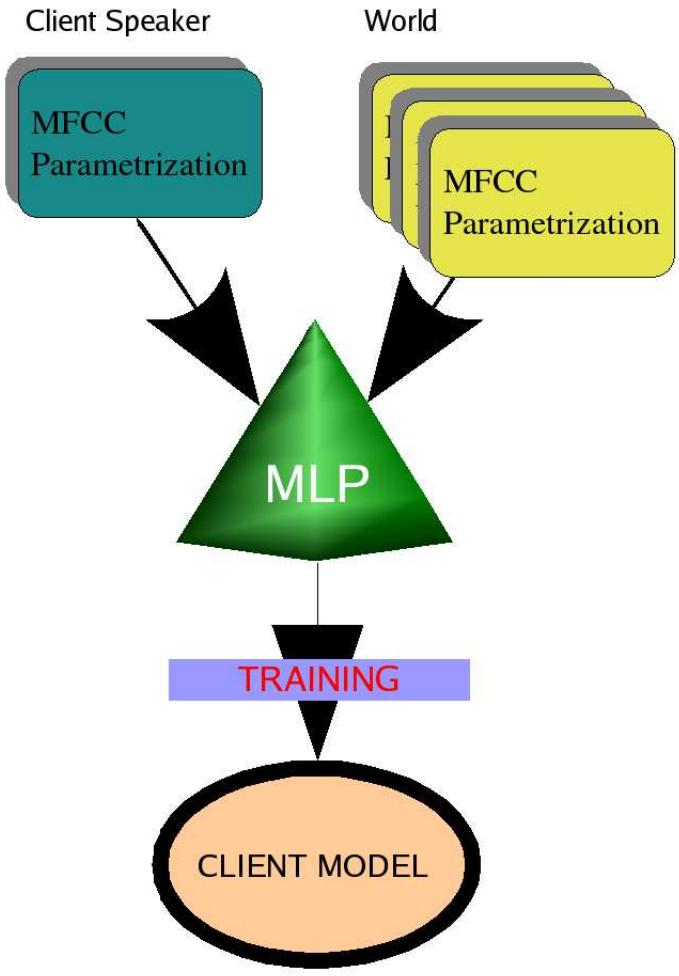


Figure 1: The training phase

As there is a big difference between available data for the client speaker and the world, it was choosen to use the client data repeated in a way such that the number of vectors for each class is the same. After training (which is stopped depending

on criterion's settings), an MLP model, containing internal nodes' configuration, is saved: this model would be used to verify the identity of the client it represents.

When testing data, a previously saved MLP configuration is loaded (which corresponds to a speaker we want to verify); MFCC vectors (from the parametrization of the unknown speaker) are "passed" to the MLP core, and the output (generally a logarithmic percentage error versus each class) is saved to a file, providing a way of judgement for the speaker verification task (*figure 2*)[6].

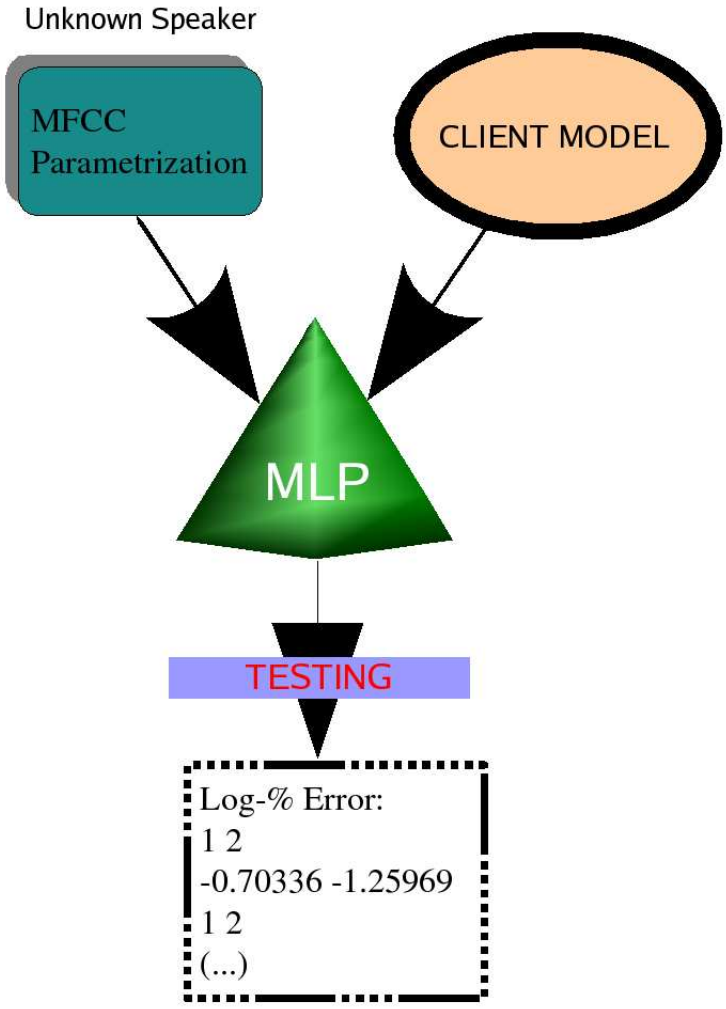


Figure 2: The testing phase

4.2 Additional tools

In this project, the neural network has been conceived as a stand-alone piece of software, but for speaker verification research purposes, additional tools are needed to perform training and testing phases: multiple Python scripts have been developed in order to take care of the training, testing and result collection phases.

[6] Additional detail can be found in Appendix C.

These scripts manage files lists as well as output results and statistics, and are also responsible for the cross validation part: they select a given percentage of the input data to use for cross validation.

Without them it would have been very difficult to perform several thousand of testing sessions and get reliable results. I will not get into details of these scripts here; if you are interested in performing tests please refer to Appendix C.

5 Research environment

In this section a detail on other important parts (mainly the input data) of this project is given.

5.1 MFCC in depth

Speech can be modeled as a convolution between a glottal excitation source, noted as *g[n]*, and the vocal tract impulse response, noted as *v[n]* (*figure* 3); it is believed that the vocal tract characteristics is important for the speaker recognition task, so we would like to separate out this filter response.

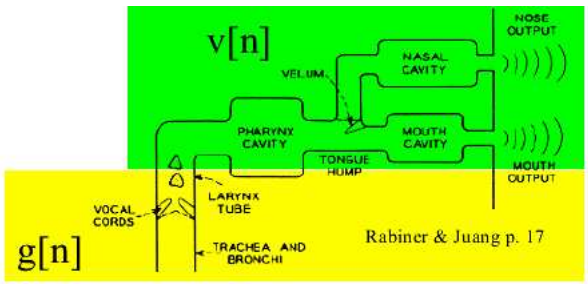


Figure 3: Speech production

A homomorphic transform converts the convolution of glottal excitation and vocal tract impulse response to a sum of them, that is:

$$y[n] = g[n] * v[n] \rightarrow \hat{y}[n] = \hat{g}[n] + \hat{v}[n]$$

This trasform can done using the logarithm function, in fact:

$$Y(\omega) = G(\omega)V(\omega)$$
$$\downarrow$$
$$\textit{Homomorphic Transform}$$
$$\downarrow$$
$$\log Y(\omega) = \log G(\omega) + \log V(\omega)$$

A special type of trasform is the cepstrum transform, which can be performed by computing the power of the spectrum, then taking the logarithm (special attention must be paid to avoid a "log 0" situation) and finally using an inverse discrete Fourier transform to move back to time domain; the Mel cepstrum has been motivated by human perception of sound: it uses a filterbank to separate the spectrum into channels: lower frequency channels are linearly spaced, higher frequency channels

are logarithmically spaced. Mel Frequency Cepstral Coefficients, better known as *MFCC*, are the workhorse features used most often in speech recognition: the basic frequency resolution is based on the Mel approximation to the bandwidths of the ear's tuned resonators, wich is fixed-bandwidth at low frequencies and constant-Q (bandwidth proportional to center frequency) at ahigh frequencies. These Mel-spectra are then cepstrally transformed (the discrete cosine transform of the log magnitude spectrum) and typicaly truncated to give a compact (8-16 element, 16 in our case) largely decorrelated feature vector capturing the broad freatures of the original spectrum. It's important to note that this type of trasformation is no longer homomorphic, but approximatively so.

5.2 Speech database

The speech database used for this project came from NIST (the american National Institute for Standards and Technology); audio samples have been recorded from telephone talks (in english language) and their length varied from about 30 seconds for the ones used for test to 2 minutes for the ones used during training. Infortunately some samples were not very clear and results in tests may have been influenced by this factor.

For the world class is was choosen to use only 30 seconds data during training (due to memory limitations), so that is possible to load much different people' short parametrizations instead of having few long ones.

For the speaker verification task a set of 73 female and 73 male client speakers has been choosen for training; for each speaker a voice sample of 2 minutes was used. From this set a client at time was choosen and trained against data from the world set. This latter set of audio samples contained voice from 49 females and 39 males, each of them providing 30 seconds of voice data.

For the male-female discrimination subproject, presented later on in this report, the same database was used, but instead of training a client versus the world, many female models (representing each 2 minutes of voice) were trained against many male models (again representing each 2 minutes of voice), and during test data from the above testing set, of about 30 seconds, was used.

Audio was parametrized using the HTK toolkit to create MFCC models; parameters used were[7]:

- Window size of 10 ms (this means that every vector represents 10 ms)
- Target rate of 80000 * 8ms frames
- 3 streams, with preemphasis coefficient of 0.95
- 24 channels and 22 filters
- 12 cepstral coefficients
- No energy normalisation

This kind of parameters were used in order to compare results of this speaker verification method with existing ones. It is possible that choosing other values could have improved the discrimination performance, but no research in this direction has been done.

[7] A detailed list of parameters used for MFCC creation can be found in Appendix B

5.3 Hardware and software platform

This project was developed on various x86 workstations running Linux (RedHat 7.3). For the test phase a dual Intel Xeon (2.4 Ghz) with 2GB of RAM was used. In order to be able to load as much voice data as needed during training, a large amount of memory is needed.

The neural network has been successfully compiled with the GNU C++ Compiler (*gcc*) either version 2.96 or 3.2.2; for Python, version 2.2 was used. Applications and tools developed and used for this project do not make use or represents any non OpenSource software.

5.4 About testing sessions

Each configuration change (either in MLP parameters or number of training files,...) was followed by an intensive testing session: each time a training and testing of all available data was performed, in order to get the clearest results as possible.

For the speaker verification task, 23408 tests for each configuration were conducted[8]: 1751 had to be verified and the remaining 21657 had to be rejected.

For the Male-Female discrimination task, 1131 voice parametrizations have been tested.

6 First results and improvement strategy

The following table shows the best results obtained for the speaker verification task, conducted only on female voice:

Correct Verified	Wrong Rejected	Wrong Verified	Correct Rejected
7.20 %	92.80 %	6.06 %	93.94 %
3.77 %	93.23 %	2.66 %	97.34 %

As we see in the above table, results for first test were not very incouraging. A mere 7.20 % in correct speaker verification (considering that there is also a 6.06 % of wrong verification) lead to no doubt: there was a problem, either in the neural network or in the parametrization of files (which could not have been well suited for neural network based discrimination). A big problem was that still many bugs existed in the multilayer perceptron code, mostly because I ignored the way Torch loads data for training (now I know that this was the source of problems), but at that time, I ignored the possible cause of this poor performace. To try to increase the correct discrimination percentage, a subproject focused on male-female discrimination was started.

6.1 Neural network optimization

In order to get better results from the neural network training, I've started working on a small sub-project that was supposed to be easier to manage and whose results, with actual methods, are good. The goal of this new challenge was to discriminate between male and female voices using the same MFCC parametrization as for speaker verification; as this task is supposed to be easier to perform, I could have expected to find out the cause of bad performance encountered with the speaker verification task.

During these tests, various parameters have been tested, such as changing the learning rate decay, the weight decay,... as well as the number of hidden units and

[8]One test means that one parametrization (MFCC file) of 30 seconds of voice data was "submitted" to the neural network and results were read.

training files. As for the original project, the training and testing phase is driven by a Python script that also computes simple stats on results.

Unfortunately, after having tried almost 500 different configurations, I could not get male-female discriminations better than 50-55 %, which is really poor considering that actual methods give a percentage above 90 %.

At this point it was clear that there was a bug in the neural network engine: I discovered that the way the multilayer perceptron loaded data was not correct, leading to a model trained only on the last parametrization entered; I quickly fixed this and tried another time for male-female discrimination: it was successful, and results were so good I could not believe it; the only major drawback is that the neural network has now to load the complete training data into memory.

6.2 Male-Female discrimination results

I this section results from the Male-Female discrimination task before and after the MFCC-loading bug are presented.

Before MFCC-load fix:

Sex	Correct Identified
Females	51.77 %
Males	55.36 %

After MFCC-load fix:

Sex	Correct Identified
Females	97.00 %
Males	93.25 %

As we see, results with the correct neural network are clearly better. Parameters used for test varies and only better results are show, but after fixing the problem I had no time to perform accurate test with other configurations, mainly because I would like to try again with speaker verification.

7 Final results

Results obtained with the corrected neural network in the speaker verification task were, as for male-female discrimination, much better than before. Unfortunately, as already said, it was not possible to perform many test, so results presented here have not to be considered as the best, althrough they are named as "final results" for this project. This time test were conducted both for male and female voices:

Sex	Correct Verified	Wrong Rejected	Wrong Verified	Correct Rejected
Females	52.31 %	47.69 %	3.93 %	96.07 %
Males	54.35 %	45.65 %	2.96 %	97.04 %

As we see in the above table, results improved by almost 50 % and are now lined up with other speaker verification methods performance. What follows (figure 4) is the DET (Detection Error Trade-Off) curve calculated on these results.

[9] Training with 77 * 30 seconds data (limited by available memory); parameters for the MLP were 100 iterations, 0.01 learning rate, 1e-5 end accuracy, 0 learning rate decay, 0 weight decay, 13 hidden units

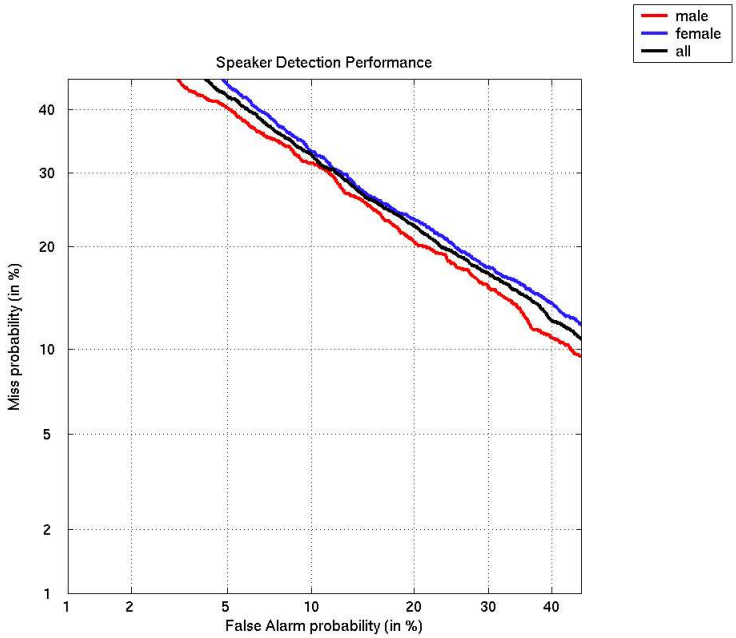


figure 4: DET curve for speaker verification

8 Conclusions

As we have seen, the speaker verification task can be successfully performed using an MLP-only technique; a longer testing session, now that big problems have been solved, could give yet better results. We have found that the main problem is the way of presenting data to the MLP. A better understanding of neural networks and the Torch Library could have been an advantage but the available time was also an issue.

As said at the beginning of this text, a further way in this speaker verification technique would focus on a segmental approach, where result of multiple verification conducted on different phonetic classes are then combinated for a judgement. Finding out classes that provide better characterization of speaker's voice would surely improve speaker verification (but this method also requires much more data).

9 Acknowledgements

Special thanks to Asmaa, Andrea, Johnny and Philippe, for their precious help and support,

also thanks to Dr. Dijana Petrovska and Prof. Dr. R.Ingold.

A Appendix : Installing Torch 3

To install the Torch 3 C++ library package, first download the sourcecode from the official website[2]; next, decompress the package in a directory at your choice.

Then copy the makefile corresponding to your operating system from the './config' directory to the root Torch source code directory (which in facts is the './config' parent directory).

Next edit the previously copied file and change the following line from:

- ***PACKAGES =***

to

- ***PACKAGES = convolutions datasets distributions gradients matrix nonparametrics speech***

You may have to change additional parameters as needed by your system configuration. Save the file and then issue a *'make depend'* followed by a *'make all'* commands from the sourcecode's root directory. That's all: the Torch compiled library is found in the *'./lib'* subdirectory. If you want to compile your own programs, the simplest way (which, of course, I'm currently using), is to copy your source file to an examples directory (for example *'./examples/discriminatives'*) and then launch a *'make* YOUR_PROGRAM_SOURCE_FILENAME_WITHOUT_EXTENSION*'*: for example, if your program is called *'foo.cc*', launch a *'make foo'* to compile it; the compiled file is usually found in a subdirectory called *'Linux_OPT_FLOAT*'. For further information about Torch as well as a programming reference can be found on the official website[2]; I suggest reading the Torch tutorial[5] as starting point for working with this library.

B Appendix : Hands on HTK

B.1 Installing HTK

In this section I will give a brief guide to install the HTK toolkit on a Linux x86 workstation.[10]

First you need to download the HTK source code (current version is 3.2) from the official website[1] (you need to complete a registration form before you can download it); next decompress the downloaded package in a directory of your choice and from a terminal (being in the HTK toolkit path) issue the one following commands lists to set global variables, depending on which type of shell you are using[11]:

If you're using a c-shell-like shell (for example csh, tcsh,...) enter the following command list (press 'Return' after each line):

- ***setenv HTKCF '-ansi -02 -DOSS_AUDIO'***
- ***setenv HTKLF '-L/usr/X11/lib'*** *(maybe you need to change this path)*
- ***setenv HTKCC 'gcc'*** *(if you are using another compiler change this)*
- ***setenv Arch LINUX***
- ***setenv Objcopy "echo"***
- ***setenv PRILF '-x'***
- ***setenv CPU linux***
- ***setenv SHRLF '-shared'***
- ***setenv LIBEXT 'so'***

[10] This is not intended to be a complete HTK reference: for further information about the toolkit please refer to documentation available on the official HTK website, which address is found in the 'References' part of this text.

[11] To determine which type of shell you are using issue the following command: *'echo $SHELL'*

If you are using a ash-shell-like shell (ash, bash, sh,...) enter the following:

- ***export HTKCF='-ansi -02 -DOSS_AUDIO'***
- ***export HTKLF='-L/usr/X11/lib'*** *(maybe you need to change this path)*
- ***export HTKCC='gcc'*** *(if you are using another compiler change this)*
- ***export Arch=LINUX***
- ***export Objcopy="echo"***
- ***export PRILF='-x'***
- ***export CPU=linux***
- ***export SHRLF='-shared'***
- ***export LIBEXT='so'***

Next move to ***'.../htk/HTKLib'*** and execute a ***'make all'***. When finished, create a new directory where the HTK binary files will be copied, for example ***'.../htk/Hbin/bin.linux'*** (the destination folder has to be named ***bin.XXX***, where XXX is the system architecture you are running on), and export this location as a global variable with the following commands:

- ***setenv HBIN '.../htk/Hbin'*** *(for a c-shell-like shell)*
- ***export HBIN='.../htk/Hbin'*** *(for an ash-shell-like shell)*

Note that you may need to change the given path.

Next, make all the files in ***'.../htk/HTKTools'*** by issuing a ***'make all'*** command: executable files are now created in the specified directory (***'.../htk/Hbin/bin.linux'***). You can now copy these files to you default bin directory, such as ***'/usr/bin'*** or add the ***'.../htk/Hbin/bin.linux'*** folder to your default path[12].

B.2 HCopy MFCC models

This section contains the HCopy' settings used to create MFCC parametrization of voice data. A general way to produce MFCC files starting from Wave files (.WAV) with this command, is to use the following syntax:

- **HCopy -T 1 -C <parametersfile> <input.wav> <output.mfcc>**

The parameters' file is a text file containing a list of parameters to be used for the conversion (as, for example, the ones you found later on in this section); you have to provide the filename of the .WAV file to parametrize and a filename for the MFCC output file. Please refer to HTK Book (freely available at HTK website [1]) for further information on using the HCopy tool.

```
SOURCEKIND = WAVEFORM
# Parameter kind of source
SOURCEFORMAT = NOHEAD
# File format of source
SOURCERATE = 1250
# 8 Khz rate in 100ns unit
```

[12] To do this please refer to your favorite Linux/Unix manual.

```
TARGETKIND = MFCC_E_D
# Parameter kind of target
TARGETRATE = 80000.0
# 8ms frame rate
SAVECOMPRESSED = F
SAVEWITHCRC = F
WINDOWSIZE = 160000.0
# 16ms Window
USEHAMMING = T
NSTREAMS = 3
PREEMCOEF = 0.95
# 1-az^-1
NUMCHANS = 24
# Number of filters
CEPLIFTER = 22
NUMCEPS = 12
# Number of cep. coefficients
ENORMALIZE = F
# Energy normalisation
FORCEOUT = T
```

C Appendix : A.VRI.L software guide

In this section a brief description of A.VRI.L software (neural network and additional scripts) is provided, in order to make it possible to "replicate" tests or to perform new ones.

C.1 A.VRI.L MLP

Note: A.VRI.L's neural network has many limitations over a general-purpose MLP, mainly due to the extreme customization needed for this research project:

- *it has been developed to perform discrimination of only two classes of data*
- *can only deal with MFCC input files*
- *its functionalities have been developed to suit the speaker verification task needs (and therefore only the needed ones have been implemented)*

So don't expect it will be useful for other purposes without huge modifications[13]*; additionally use it at your own risk.*

By the way, let's have a look at A.VRI.L MLP and how to use it...

C.1.1 Compiling the source

To compile the A.VRI.L neural network source code, simply copy it into the ***examples/discriminatives*** directory of your Torch installation (if you have not yet installed Torch please refer to Appendix A). Next call ***make avrilXXXX*** (depending on the version of the source code, typically ***avril94d1***), and you'll find the executable in the subdirectory created in the current folder (whose name depends on the operating system you are using).

[13] *In fact I recommend you to write your MLP from scratch if you intend to use it for other purposes.*

C.1.2 Command line arguments

By calling the A.VRI.L neural network' executable without any parameter, a list of valid arguments is shown. For each parameter the valid type is specified (for example <INT>); additionally a default value have been set (for example [100]).

```
# # USAGE: ./AVRIL94D1 [OPTIONS] <INPUTS> <TARGETS>
```

Training mode arguments and options:

```
# VALID ARGUMENTS:
<INPUTS> -> THE TRAIN INPUTS LIST OF MFCC FILES OR FILENAME IN -
SINGLE MODE (<STRING>)
```

- <INPUTS> is the name of the file containing the list of input MFCC files for training (Note: this file has to end with an empty line)

```
<TARGETS> -> THE TRAIN TARGETS LIST FILE OR CLASS TAG IN -
SINGLE MODE (<STRING>)
```

- <TARGETS> is the name of the file containing a list of output tags to identify the class of the corresponding MFCC file given in the input list (Note: this file has to end with an empty line)

```
TRAINING OPTIONS:
-ITER <INT> -> MAX ITERATION [100]
```

- Number of iterations for each vector during training, default is 100

```
-LR <REAL> -> LEARNING RATE [0.01]
```

- Learning rate, default is 0.01

```
-E <REAL> -> FINAL ACCURACY [1E-05]
```

- End accuracy, default is 1e-05

```
-LRD <REAL> -> LEARNING RATE DECAY [0]
```

- Learning rate decay, default is 0

```
-KFOLD <INT> -> NUMBER OF FOLDS, IF YOU WANT TO DO CROSS-VALIDATION [-1]
```

- Number of folds for cross-validation, default is -1 (none)

```
-WD <REAL> -> WEIGHT DECAY [0]
```

- Weight decay, default is 0

```
-OM -> USE OUTPUTMEASURER
```

- Output measurer toggle; if set, the output error will be written to file in the specified (using the -dir option) directory. The file format will be (for example):
  1 2
  -0.0345 -1.5390
  Which means that the log of the percentage error for class 1 (client) is -0.0345, and for class 2 (world) si -1.5390. When making a judgement remember that these are log values!

```
-NLLM -> USE CLASSNLL MEASURER
```

- Another type of measurer; please refer to the Torch[2] documentation for more detail.

  -TM -> USE TEMPORAL MEAN AFTER LOGSOFTMAX

- Temporal mean switch; adds a temporal mean layer as output

  -RT -> RETRAINING MODE

- Sets retraining mode: do not overwrite an existing MLP model.

  -BALANCE -> EQUAL VECTORS COUNT FOR EACH CLASS

- Loads a balanced number of vectors for each class; it would be the number of vectors of the class with the lowest vector count.

  -SINGLE -> SIMPLE INPUT FILE AND CLASS

- Set this if a sigle input file and class is given instead of a list in the above arguments.

  -TAGZERO <STRING> -
  > CLASS ZERO TAG (ANYTHING DIFFERENT WILL BE CONSIDERED AS CLASS ONE) [0]

- Set the identifier tag (as a string) for the first class (client) to be used as target. Note that anything different will be considered as a second class identifier (the tag is case sensitive). Default value is 0.

  OTHER OPTIONS:
  -SEED <INT> -> THE RANDOM SEED [-1]

- Set a random seed at your choice, used to initialize neural network node's weight.

  -VILIST <STRING> -> THE VALIDATION INPUT LIST FILE OR FILENAME IN -
  SINGLE MODE []

- The filename of the list with MFCC files used for validation (if the -SINGLE switch is set this will be the name of a single MFCC file)

  -VTLIST <STRING> -> THE VALIDATION TARGET LIST FILE OR TAG IN -
  SINGLE MODE []

- The list of target identifiers of validation files (if using a single file you can enter the identifier directly)

  -DIR <STRING> -> DIRECTORY TO SAVE MEASURES [.]

- Directory/path were measurers save output to. Default path is the current one.

  -SAVE <STRING> -> THE MODEL FILE [AVRILMODEL]

- The model filename, used to save MLP parameters and neural network weights. In order to perform speaker verification tests you need to save training sessions to a model with this option. Default filename si **'AVRILMODEL'.**

  -HIDDEN <INT> -> HIDDEN UNITS [13]

- Number of hidden units. Default is 13.

  -LENDIAN -> LITTLE ENDIAN MODE

- Set little-endianess for files, default is big-endian.

  # # OR: ./AVRIL94D1 –TEST [OPTIONS] <MODEL> <INPUTS FILE>

Testing mode arguments and options:

  # VALID ARGUMENTS:
  <MODEL> -> THE MODEL FILE (<STRING>)

- The model file to load.

  <INPUTS FILE> -> THE TEST LIST FILE (<STRING>)

- The list containing MFCC files to test on the given model.

  OTHER OPTIONS:
  -LOAD <INT> -> MAX NUMBER OF EXAMPLES TO LOAD FOR TEST [-1]

- Number of vectors to test. Default is -1, which means that all vectors are loaded.

  -SINGLE -> SIMPLE INPUT FILE AND CLASS

- Single file mode: indicates that the filename provided as input file is a single MFCC file instead of a list.

  -DIR <STRING> -> DIRECTORY TO SAVE MEASURES [.]

- Directory/path were measurers save output to. Default path is the current one.

  -BIN -> BINARY MODE FOR FILES

- (Same as for training)

  -OM -> USE OUTPUTMEASURER

- (Same as for training)

  -NLLM -> USE CLASSNLL MEASURER

- (Same as for training)

C.1.3 General usage guideline

Generally, to train the neural network on a list of MFCC files, create two text files (remember to leave an empty line at the end of each one):

- One file containing the list of the MFCC files (call it, for example, ***'list.in'***) to train
- The other containing targets, one per line, associated with the input files (either '0' or '1'; call it, for example ***'list.out'***)

Next, to train for a model called ***'EXMODEL'*** launch A.VRI.L MLP with the following command:

- ***./avril94d1 -save EXMODEL list.in list.out***

To test a file on the model previously saved issue the following command:

- ***./avril94d1 –test -single -om EXMODEL FILETOTEST.mfcc***

As the '-OM' option was used, we get an output of OutputMeasurer for each vector which is saved to a file called 'the_class_err'[14]; the format of this output is the following:

> ***1 2***
> ***<log % error for first class> <log % error for second class>***
> *(repeated n-times, one for each vector)*

Remember that the percentage error is calculated as a logarithm, so, output values are negative. An example output could be:

> 1 2
> -0.3567 -1.4667
> 1 2
> -0.4711 -1.2677

Which means that for the first vector the log of the percentage error for the first class (the client speaker, in the speaker verification task) is -0.3567, while the value regarding the second class (the 'world') is -1.4667. In this case, considering also the output for the second vector, the lowest percentage error is found for the second class, so the file belongs to this one (and is therefully rejected, in the speaker verification task).

C.2 Additional scripts

To perform tests, many Python scripts were used; I will not describe them, as they are very specific to the test environment I used and subject to change often. If you want to use these tools I encourage you to read the script source and try to understand them as much as possible. If there is something that is not clear feel free to contact me.

References

[1] HTK, version 3.2, is developed by the Speech Vision and Robotics group, Cambridge University Engineering Department; http://svr-www.eng.cam.ac.uk/

[2] R. Collobert, S. Bengio, and J. Mariéthoz. Torch: a modular machine learning software library. Technical Report IDIAP-RR 02-46, IDIAP, 2002. WebSite: www.torch.ch

[3] Petrovska-Delacrétaz, Dijana, Cernocky, Jan, Hennebert, Jean and Chollet, Gérard, Segmental Approaches for Automatic Speaker Verification, Digital Signal Processing 10 (2000), 198-212

[4] A.VRI.L MLP 0.94d sourcecode, found on cdrom (under 'MLP Sources')

[5] R. Collobert, Torch Tutorial, found on www.torch.ch

[14] As general rule, always set the '-OM' option in order to get an output from the neural network. If no measurer is set (either with the '-OM' or '-NLL' switch) no output/result from the MLP is given.