Università degli Studi di Roma "Tor Vergata"
Dottorato di Ricerca in Sistemi e Tecnologie per lo Spazio
Ciclo XIX

Design of Hardware Architectures for HMM–Based Signal Processing systems

With Applications to Advanced Human-Machine Interfaces

Alessandro Malatesta

February 2007

Supervisor:
Prof. Gian Carlo Cardarilli

Co–Supervisor
Ing. Luigi Arnone

Contents

Preface **vii**

1 An overview of human-machine interfaces **1**

1.1 Introduction to human-machine interaction 1

1.2 Human-machine interfaces 2

1.2.1 Different kinds of user interfaces for HCI 3

1.3 New generation interfaces 4

1.3.1 Hands-free interfaces 4

1.3.2 Evolution of standard interfaces 5

1.4 Human-machine interfaces in space systems 6

1.4.1 Multimodal interfaces 6

1.5 Dealing with time series: the HMM 7

1.5.1 Pattern recognition systems 8

Statistical classification 9

1.5.2 Time series in pattern recognition 10

2 Use of HMMs in pattern recognition **13**

2.1 Introduction to Hidden Markov Models 13

2.1.1 Discrete Markov processes 13

2.1.2 From Markov chains to Hidden Markov Models 16

2.1.3 Characterization of HMMs 18

2.2 Use of HMMs for pattern recognition 20

2.2.1 Use of HMMs for classification 22

2.2.2 Exploring the hidden layer: optimal state sequences 25

2.2.3 HMM Training 27

2.2.4 HMM topologies 30

2.2.5 HMM with a continuous observation space 31

2.2.6 Some more variations: null transitions, tied states, state durations 32

Null transitions 33

Tied states and shared distributions 34

Explicit state duration 35

2.2.7 Multiple observation sequences . . . 36
2.3 Limitations and common problems . . . 37
2.3.1 Choice of initial parameters for training . . . 37
2.3.2 Effects on under-training . . . 37
2.3.3 The choice of the model . . . 39

3 Development of a dedicated hardware architecture 41
3.1 Towards hardware . . . 41
3.1.1 Choice of an algorithmic framework . . . 42
3.1.2 Splitting it down into pieces: Token Passing . . . 44
3.1.3 Considerations on hardware implementation . . . 47
3.2 System level overview . . . 48
3.2.1 From the conceptual model to the architecture . . . 49
3.2.2 Defining the hardware platform . . . 49
3.2.3 Refining the system architecture . . . 51
3.2.4 Keeping track of tokens . . . 53
3.3 Going deeper: the HMM processor . . . 54
3.3.1 Overview on the HMM processor architecture . . . 56
3.4 Details of the architecture's components . . . 58
3.4.1 The Token Passer . . . 58
3.4.2 The Emission Probability Computer . . . 59
3.4.3 The Token Life Monitor . . . 64

4 Implementation and testing of the architecture 67
4.1 A design flow for the architecture . . . 67
4.1.1 Choice of the design tools and strategies . . . 69
4.2 Implementation details . . . 70
4.2.1 The Top Level: HMM Processor . . . 70
4.2.2 The HMM processor control . . . 72
4.2.3 The processing blocks . . . 76
Memories . . . 76
Computation of emission probability . . . 79
Token Passing . . . 83
Notification of tokens' lifespan . . . 85
Finalization and results' output . . . 86
4.3 Test of the architecture . . . 87
4.3.1 System Implementation . . . 87
4.3.2 Functional Verification . . . 92

5 Selected applications for human-machine interfaces **99**
5.1 Applications to speech recognition 100
5.1.1 Introduction to ASR 100
5.1.2 Implementation of a phonetic recognizer 104
5.2 Applications to brain-computer interfaces 112
5.2.1 Introduction to Brain–Computer Interfaces 112
Principles of Electroencephalography 115
5.2.2 Implementation of a Binary EEG Classifier 117
Data Recording 118
Signal Processing 122
EEG Classification 127
Hardware Tests 136

6 Conclusions **139**

Acknowledgments **141**

List of Figures **147**

List of Tables **149**

Bibliography **159**

Preface

This thesis presents a research work that passes through different stages of evolution. The first ideas were developed during my Master Thesis through a collaboration with STMicroelectronics (Advanced System Technologies, Agrate, Italy).

The first efforts were towards the realization of a natural language speech recognition system. The subject was appealing because of the huge amount of possible applications of that technology in the development of advanced human–machine interfaces (space systems, support for disabled people and consumer applications, among them). The Hidden Markov Model (HMM) was chosen as the main algorithmic tool to be used to build the system. The early analysis on the related algorithmic framework brought to the identification of the major performance bottlenecks in that class of systems and to the development of some hardware solutions aimed to achieve better performances, and open new possibilities for the development of systems more advanced than the state of the art allowed.

After that work, my collaboration with STM continued through the Ph.D., initially on the same subject. After the early work I realized how the usefulness of the HMM framework, and the wide range of possible applications, would have justified the effort put in the development of an hardware architecture that provided a reconfigurable tool to be used in any pattern recognition task, and not only in speech recognition. So the development of that architecture begun and was carried out through different incremental improvements, and using different methodologies and tools.

As a parallel line of research, the algorithms implemented in the hardware architecture were tested on different applications, with a particular focus on advanced human–machine interfaces. Anyway most of the work have been concentrated on two of these applications.

The first one was natural language speech recognition. This application was chosen mainly because it is an application still far from being fully functional and useable, but also because it is among the ones in which the HMMs are most successful. Moreover a good part of the problems related to this applications are due to the tradeoff between speed and accuracy: the introduction of a speed improvement through hardware seemed thus a promising solution. Finally, treating this application allowed also to maintain a continuity with the early work.

The second application was introduced in the project thanks to a fruitful collaboration with the Laboratory of Non–Linear Systems at the EPFL (École Polytechnique Fédérale de Lausanne). The problem of Brain Computer Interfaces was faced and some really interesting results on the classification of the electroencephalographic signal were obtained. Even in this case the

HMMs showed to be a really valuable and effective tool.

The topics in this thesis are organized as following:

Overview on Human–Machine Interfaces: The topic of human–machine interface design is introduced and the main issues are stressed. The field of interest is identified as the development of advanced interfaces. Particular attention is paid to pattern recognition systems for time series, and the choice of the HMM algorithmic framework is motivated.

Hidden Markov Models for pattern recognition: The theory of HMMs is introduced, and its applications to the problem of pattern recognition are discussed.

Development of a hardware architecture: The chosen algorithms are analyzed from the point of view of the implementation as a digital hardware system. Then the development of a dedicated hardware architecture for HMM processing is described in detail, from system level operation down to the processing sub–systems and basic components.

Implementation and Testing: The core subsystem of the hardware architecture is implemented at register transfer level and described using an hardware description language. Some FPGA–based fast–prototyping hardware platforms are selected for testing. The mapping of the design on the hardware platforms is described, along with several mapping configuration variations. Finally the hardware tests made for functional verification of the system are discussed.

Application: The application of the developed system to two real–world pattern recognition systems is described. The first application is a phonetic recognizer (a sub–system of a speech recognizer); the second application is a binary classifier for electroencephalographic signal (brain–computer interface). The description of the BCI application also brings some new research results on the classification of the EEG signal.

Chapter 1

An overview of human-machine interfaces

1.1 Introduction to human-machine interaction

Since the appearance of machines, developed in order to aid humans in all kind of different tasks, one of the main issues has always been to provide a proper way of control on such machines, namely a *user interface*. Of course the concept of machine can be easily extended to any kind of device that can be controlled by a user (nowadays the most common example is a computer).

The main goal of a user interface should be to provide a suitable control on the system to which it is applied, while being comfortable and natural for the user (user friendliness). Nonetheless a good user interface should provide also enough *feedback* to grant proper operation; in other words the quantity of informations given on the controlled system's status, should be enough to allow the user to take decisions on which kind of control to apply to the system in every moment during the interaction.

So, we could say that a good user interface should have a good balance between *usability* for the person that interacts with it, and *effectiveness* of control on the system. The quality of the *feedback* plays its role in both factors, since it increases the usability and allows for better control.

As one can imagine, as systems get more and more complex, the user interface could become really difficult to manage. Such increase in complexity brought to the development of different kind of techniques that, properly mixed together, can allow for interaction with the most advanced devices and machines.

As a wider topic, *Human Machine Interaction (HMI)* is concerned with

- methodologies and processes for designing interfaces (i.e., given a task and a class of users, design the best possible interface within given constraints, optimizing for a desired property such as 'learnability' or efficiency of use)

- methods for implementing interfaces (e.g. software toolkits and libraries; efficient algorithms)
- techniques for evaluating and comparing interfaces
- developing new interfaces and interaction techniques
- developing descriptive and predictive models and theories of interaction

A long term goal of HMI is to design systems that minimize the barrier between the human's cognitive model of what user wants to accomplish and the computer's understanding of the user's task. This topic is addressed by the theory of *Computer Supported Cooperative Networks (CSCW)*. CSCW focuses on the study of tools and techniques of groupware *as well as* their psychological, social and organizational effects [1].

People with hands on HMI are usually designers concerned with the practical application of design methodologies to real-world problems. In the field of *Human Computer Interaction (HCI)* their work often revolves around designing graphical user interfaces and web interfaces. On the other hand researchers in HMI are interested in developing new design methodologies, experimenting with new hardware devices, prototyping new software systems, exploring new paradigms for interaction, and developing models and theories of interaction.

A number of diverse methodologies for HMI design have emerged since the rise of the field in the 1980s. Most design methodologies start from a model of how users, designers, and technical systems interact. Early methodologies, for example, treated users' cognitive processes as predictable and quantifiable and encouraged designers to look to cognitive science results in areas such as memory and attention when designing user interfaces. Modern models tend to focus on a constant feedback and communication between users, designers, and engineers, and push for technical systems to be adapted to the types of experiences users want to have, rather than forcing the user to adapt his experience to a completed system [2].

Among the modern approaches, *user-centered design (UCD)* is widely practiced design philosophy rooted in the idea that users must take center-stage in the design of any HMI system. Users and designers work together to articulate the wants, needs, and limitations of the user and create a system that addresses these elements. Often, user-centered design projects are informed by ethnographic studies of the environments in which users will be interacting with the system.

1.2 Human-machine interfaces

As already noted, the concept of HMI can be applied to a wide variety of systems, ranging from industrial machinery, to vehicles, electronic devices and so on. Anyway, since *a computer* (but we could say 'a machine' in the widest sense) *shall not waste your time or require you to do more work than is strictly necessary*[1], low complexity systems should not need highly complex

[1]Jef Raskin, Expert in human-computer interaction who began the Macintosh project for Apple Computer, in *The Humane Interface: New Directions for Designing Interactive Systems*.

interfaces. Furthermore is well known that really complex systems, even if of mechanical type, often need a computer to handle the interface and control tasks, in order to allow the user to concentrate on the overall system operation instead of technical details.

Given these premises, I'm going to focus the attention on *human-computer interaction* (HCI from now on) [3] without a great loss of generality, understood that the computer itself can be in turn used to provide commands to almost any kind of complex machine or device.

1.2.1 Different kinds of user interfaces for HCI

Nowadays the most commonly used user interfaces for HCI can be classified as follows:

Graphical User Interfaces (GUI): Provide a complex graphical feedback on a display, usually with buttons and other similar means [4]. Input is provided by devices such as keyboards, pointing devices (like mice) and touch devices (like touch screens). This class is used almost in any kind of application.

Command line interfaces: The user provides commands through a keyboard in the form of character strings. The systems provides feedback by printing text on a display. This kind of interface is one of the earliest, and it is still used in UNIX terminals, usually for system administration tasks.

Tactile interfaces: Make use of *haptic* input and feedback. This means that input is provided through devices that are sensible to touch and tactile manipulation [5, 6, 7]. Feedback also can be provided through vibrating devices that can be handheld (really popular is the *Force Feedback* technology used in gaming) or applied on the skin in different parts of the body [8]. This interfaces are used for complex manipulation tasks (like 3D modeling and interaction with virtual environments) and for aid to visually impaired users.

Gesture interfaces: The user provides the commands through gestures. Gestures can be traced with a stylus on a touch-sensible device, or just in the air and recognized through a video camera. Often it is possible to find this class of interfaces in handheld devices, but in general the field of application is wide.

It is clear that the shared feature of the interfaces described, is that they are all hand controlled (in opposition to the interfaces that I will describe in Section 1.3 and that are going to be called, in this work, *advanced (or new generation) user interfaces*).

Most of the real world user interfaces are often a mix of the classes described here, according to the specific needs. This kind of interfaces are often referred to as *multimodal interfaces* [9]. The main point in developing multimodal interfaces is on integrating humans, computational devices, and environments in a seamless manner, leveraging the unique capabilities of each to satisfy system-level requirements. Since multimodal interfaces usually make use, along with the classes of interfaces already introduced, of more advanced techniques, more details will be given in Section 1.3.

1.3 New generation interfaces

As already noted, usually when we talk about human computer interface, we think about the kind of interfaces we are used to in our everyday life: a mouse, a keyboard, a screen. And even if we leave apart our computer experience, we realize that the most common tasks, like getting money or buying tickets from an automatic machine, involves a really similar way of interaction, based on screen prompts and button or touch-screen input. All of this interaction is based on using our hands to provide the input, and of our eyes to get the feedback (audio feedback is a common exception, but it is rarely crucial in the task).

This common idea of HMI is due to the fact that the greatest part of the interfaces we deal with in our everyday life is of that kind. Nevertheless technology is rapidly evolving towards totally different paradigms, and some hands-free interfaces are already available along with some interesting evolutions of the popular *touch and sight* kind of interaction.

1.3.1 Hands-free interfaces

There are a lot of cases in which having the ability to control a system without the use of our hands can turn extremely useful, both because the user needs his hands to perform a different task in parallel or because the user cannot use his hands owing to some kind of disability (e.g. neurological pathologies, accidents).

Some of the most promising technologies that can be used in hands-free interfaces are the following:

Vocal interfaces: The main technologies involved in a fully vocal interface are *speech recognition* for the input part [10] and *speech synthesis* for the feedback part [11]. Speech recognizers analyze audio waveform and provide the corresponding text. The extracted text is then processed in order to extract its meaning and translate it in an actual command for the systems [12]. Typical speech recognizers make use of techniques like *Hidden Markov Models* [13], *Neural Networks* [14] and other kinds of statistical classifiers. Typical drawbacks of this technology is low accuracy under certain conditions (namely very large vocabulary, speaker independentness, continuous speaking, highly noisy environments) especially when these conditions are present altogether [15]. On the other side successful applications have been deployed with medium sized vocabularies and specific training for speaker adaptation [16, 17]. More details on speak recognition can be found in Section 5.1.

Brain-computer interfaces: In brain-computer interfaces (BCI) the input is provided by biological signals coming from the user's brain [18]. Common techniques for signal acquisition are *Electroencephalogram* (EEG) [19] and *Electrocorticogram* (ECoG) [20] but also more advanced like *Functional Magnetic Resonance Imagery* (fMRI) [21]. The signal is classified (which technique is most suitable is still matter of discussion), and used to provide commands to a system (e.g. a wheelchair [22]). The research on BCI is still ongoing, but some results have already been obtained with invasive techniques (like ECoG [23]). Non invasive techniques (EEG) are still affected by the problem of really low SNR, but the effort on this part of the problem is consistent and begins to provide significant results [24]. More details on BCI systems can be found in Section 5.2.

Image-based interfaces: This class of interfaces relies on capturing the user with a video camera, and interpreting the footage in some way in order to extract commands or some other informations useful for the interface's operation. Most of the techniques used in this kind of interfaces make use of some form of object/feature tracking in the captured video [25]. *Gesture recognition* [26] interprets some movements of the user and translates them in commands; gestures can be produced with hands, arms, head, face expressions and, in principle, with almost any part of the body but, since we are talking about hands-free control, the most interesting features are clearly the one extracted from the image of the user's head (head movements and facial features). *Expression recognition* [27] can provide useful informations on the context in which the interface is used (e.g. the mood of the user). An even more advanced technique is *eye tracking* [28], in which eye movements can be used to replace a pointing device, and eye blinking as a selection command. Finally, most of the informations that can be extracted from a footage of the user (position, posture, movements, and so on) can be successfully used as a complement to other kind of interfaces when developing the so called *multimodal interfaces*. More detail on multimodal interfaces can be found in Section 1.4.1. An example of image-based interface can be found in the experiment setup described in Section 5.2.

1.3.2 Evolution of standard interfaces

When talking about new generation or advanced interfaces, we surely do not talk only of hands-free interfaces. Even the most widely used kinds of interfaces are quickly evolving. In this section I'll give a brief overview of the directions that this evolution has taken, showing some examples.

Web-based interfaces: This is the most straightforward and common evolution of graphical user interfaces. The principle is the same: the user controls a system through a graphical interface shown on a display, using some kind of pointing device. The fundamental difference is that the system is remotely controlled: the interface is on a web server that accepts inputs and provides outputs by generating web pages which are sent through the Internet and viewed by the user using a web browsing application. Newer implementations utilize Java, AJAX, or similar technologies to provide realtime control [29, 30].

Attentive user interfaces: They track the user attention deciding when to interrupt the user, the kind of warnings, and the level of detail of the messages presented through the feedback channel of the interface. This technology is basically a feature that can be applied to any kind of interface to raise the level of adaptability to the user's behavior [31]. The next step in this kind of technology are the *Non-command user interfaces* [32], which observe the user to infer his needs and intentions, without requiring that he formulates explicit commands, and the *Zero-Input interfaces*, that grab inputs from a set of sensors instead of querying the user with input dialogs [33].

Zooming user interfaces: These are graphical user interfaces in which information objects are represented at different levels of scale and detail, and where the user can change the scale of the viewed area in order to show more detail [34]. In some applications can be used also along with tracking technologies in order to provide hands free operation [35].

1.4 Human-machine interfaces in space systems

All the interfaces described so far find their application in a wide variety of systems, with no particular restrictions. Anyway, a really interesting field of application is HCI/HMI in space environment. This is mainly due to the fact that, if some tasks that are commonly performed on earth with the aid of machines can be also performed without them (clearly at the expense of some more effort and time), the same cannot be said when we are talking about space. In other words, since man cannot go in space without the aid of machines, the human-machine interaction becomes an even more critical issue, and *the two [. . .] should be wed in such a way that the respective strengths of each complement the weaknesses of the other*[2].

As everybody knows, the application of any commonly used technology in space becomes sensibly more difficult, with respect to the terrestrial use, since the behavior of both the users and the devices changes radically. As regards the systems themselves, the designer should deal with several problems. In mechanical systems, for example, the reduced gravity (or the total absence of it) makes the designer's task quite tougher [36]. The same is true for electronic systems, since the the currents injected by the particles that compose the radiation present in space give rise to the need of different design strategies, both from the point of view of the production process [37, 38] and of the hardware architectures [39]. Even the users show a definitely different behavior due to the different gravity (different use of muscles), the different living spaces, the variations in day-night cycles, irradiation (light flashes and other similar effects) [40, 41].

Given all these premises, it is clear that interfaces for space systems are more likely to make use of advanced techniques than interfaces for terrestrial systems. In particular all the major space groups, like *NASA* [42], are focusing on *multimodal interfaces* in order to exploit the most useful characteristics of every technique.

1.4.1 Multimodal interfaces

A multimodal interface can include speech and natural language interfaces, multimedia systems, adaptive and intelligent interfaces and displays, ubiquitous computing, and mobile and

[2]Dr. Eugene Konecci, NASA Director of Biotechnology and Human Research (speech, 1964)

wearable computing [9]. The aim is to find ways to couple humans and computers into cooperative, efficient systems. For example, team interfaces that reflect mixed-initiative models of content, activity, and dialogue, will require advances in spoken and visual languages, flexible movement among levels of abstraction, and consistent coordination of multiple representations.

Multimodal interfaces go far beyond simple displays and voice-response systems. They help decision makers accomplish high-bandwidth, heterogeneous, real-time data integration and interpretation. In space systems applications include remote science, model-based predictive control, onboard system management and procedure execution, information management for ground operations.

A representative example might be an intelligent maneuvering system for the International Space Station [43]. The ISS will offer serious challenges to humans performing intricate docking and assembly maneuvers. An intelligent maneuvering system could enhance the performance of astronauts by integrating tele-robotic systems with software for monitoring, scheduling, and decision support. Such an intelligent system would reduce the probability of mishaps, while freeing operators from routine monitoring and manipulation tasks.

Another broad area of application is enhancement of air traffic safety and capacity. A new generation of performance aids is needed both on the ground and in the air. In the cockpit, a new generation of displays and interfaces could mitigate information overload and improve situation awareness. Some groups are also developing systems that can assist pilots in flying severely damaged aircraft by reconfiguring the flight-control computer system.

This kind of interfaces must be useful even under difficult conditions, and their safety and reliability must be established in advance through rigorous design and testing. However, engineering design of such flexible, robust performance-enhancement systems is still beyond the state of the art. Research is focused on developing better models of attention, memory, conceptual structure, decision-making, learning, and higher-level perception needed for principled design and testing of novel human-computer systems. These will permit early, reliable design choices (minimizing life cycle cost) and reduce system risk.

1.5 Dealing with time series: the HMM

From the considerations made in Section 1.4 it seemed reasonable to focus this thesis on what I defined *advanced interfaces* (see Section 1.3) since these are in some way the building blocks of multimodal interfaces (see Section 1.4.1), and multimodal interfaces are the ones that promise to provide the user with the most comfortable and efficient interaction experience.

When designing user interfaces that are supposed to deal with a lot of informations to be passed from the user to the system and back, often some kind of *machine learning* [44] is involved, since a suitable interpretation of user input, based also on prior knowledge of the task and of its context, allows to infer from the inputs more information than is actually contained in it.

Is well known that machine learning is concerned with the development of algorithms and

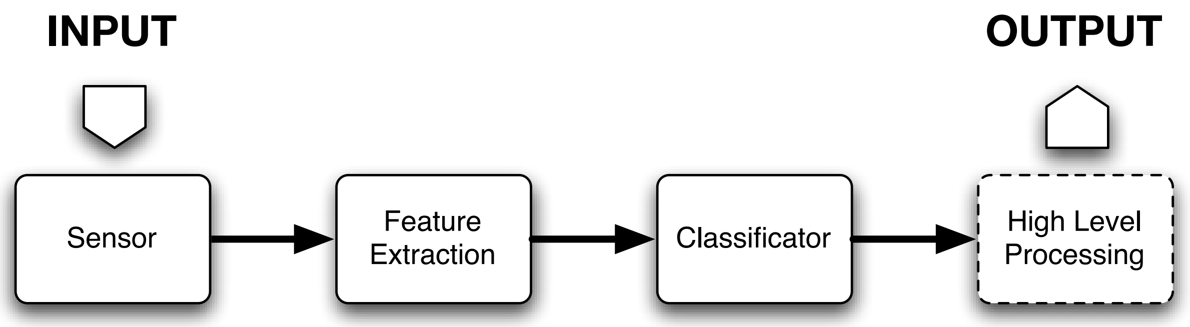


Figure 1.1: A block diagram that describes a generic pattern recognition system.

techniques that allow computers to 'learn'. Machine learning has a wide spectrum of applications including natural language processing, data retrieval, medical diagnosis, bioinformatics and cheminformatics, security, stock market analysis, classification of DNA sequences, speech and handwriting recognition, object recognition in computer vision, robotics [45]. Some machine learning systems attempt to eliminate the need for human intuition in the analysis of the data, while others adopt a collaborative approach between human and machine. Which of the two approaches to choose is tightly related to the goals and specifications of the selected application.

Anyway in all the mentioned applications, the goal is to extract meaning from some kind of input (that can be provided in quite a lot of different ways). That can be stated also as identifying a *pattern* in the input and assign it to a known *class*. The techniques that allow to do this belong to a branch of machine learning known as *pattern recognition* [46].

1.5.1 Pattern recognition systems

Pattern recognition aims to classify data (patterns) based on either a priori knowledge or on statistical information extracted from the patterns. A complete pattern recognition system, typically consists of:

- a *sensor* that gathers the *observations* to be classified or described
- a *feature extraction* mechanism that computes numeric or symbolic information from the observations
- a *classification* or description scheme that does the actual job of classifying or describing observations, relying on the extracted features

A fourth part could be added, and it is the one that translates the result of the classification into informations directly usable by the system through high level processing. A general representation of this kind of system is depicted in Figure 1.1.

The classification or description scheme is usually based on the availability of a set of patterns that have already been classified or described. This set of patterns is called the *training set* and the resulting learning strategy is characterized as *supervised learning*. Learning can also be *unsupervised*, in the sense that the system is not given an a priori labeling of patterns, instead it establishes the classes itself based on the statistical regularities of the patterns [47].

The classification or description scheme usually uses either the *statistical* (*decision theoretic*) approach or the *syntactic* (*structural*) approach. Statistical pattern recognition is based on statistical characterizations of patterns, assuming that the patterns are generated by a probabilistic system. Structural pattern recognition is based on the structural interrelationships of features.

When the classification task is highly complex and involves a high number of classification targets and high variability in the inputs (due usually to differences between the sources that produce the input and to external noise scenarios), usually statistical classification performs better since a well chosen training data-set can take into account variations that are difficult to identify with a structural approach. Most often some techniques taken from structural pattern recognition are used in statistical frameworks to put constraints and improve generalization and *outlier* inputs.

Statistical classification

In statistical classification individual items are subdivided into groups on the base of quantitative information related to one or more characteristics inherent the items. The reference quantities used in the classification process are taken from a training set made of previously labeled items of the same type.

Formally the problem could be stated as follows: given a training data-set

$$\{(\boldsymbol{x_1}, y_1), \ldots, (\boldsymbol{x_n}, y_n)\}$$

the aim is to build a classifier $h : \boldsymbol{X} \rightarrow Y$ that can map an object $\boldsymbol{x} \in \boldsymbol{X}$ to its correct classification label $y \in Y$.

If, for example, the problem at hand is filtering spam, then $\boldsymbol{x_i}$ could be some representation of an e-mail, and the classification labels would be $Y = \{Spam, Non - Spam\}$.

Even if statistical classification techniques are many, each of them always solve one of three precise mathematical problems.

The first one is to find a map from a feature space (which is typically a multi-dimensional space) to a set of labels. This is equivalent to partitioning the feature space in different regions and assigning a label to each of them. Algorithms that solve this problem (like the nearest-neighbour algorithm) typically do not provide a confidence level or class probabilities, unless some kind of post-processing is applied. This problem can be solved also by first applying unsupervised clustering to the feature space and then trying to apply labels to the resulting clusters.

The second problem is to consider the classification task as an estimation problem where the goal is to estimate a goal function of the form

$$P(class|\vec{x}) = f\left(\vec{x};\vec{\theta}\right) \tag{1.1}$$

where the input feature vector is $\vec{x}$ and the function f is usually parametrized through some parameter vector $\vec{\theta}$. In the bayesian approach to this problem, instead of choosing a single parameter vector $\vec{\theta}$, the result is integrated over all the possible parameter vectors, weighting each of them according to their likelihood given the training dataset D. This means that the goal function in (1.1) becomes

$$P(class|\vec{x}) = \int f\left(\vec{x};\vec{\theta}\right) P\left(\vec{\theta}|D\right) d\vec{\theta} \tag{1.2}$$

The third and last problem is closely related to the second one, and it consists of estimating the probabilities of the features conditioned to the classes, namely

$$P(\vec{x}|class) \tag{1.3}$$

After estimating this function, one can easily find the class probability of equation (1.1) by means of the Bayes' rule

$$P(class|\vec{x}) = \frac{P(\vec{x}|class)P(class)}{P(\vec{x})} \tag{1.4}$$

1.5.2 Time series in pattern recognition

A wide range of algorithms can be applied for pattern recognition, from very simple Bayesian classifiers to much more powerful neural networks. Anyway, the patterns to be classified are usually groups of measurements or observations, defining points in an appropriate multidimensional space. In particular when dealing with user interfaces, the temporal dimension is always present, since it is natural for the user's interaction with the systems to evolve in time. Given this premise, it is clear that the input measurements will be some kind of *time series*.

In statistics and signal processing, a time series is a sequence of data points measured at successive times with a defined *sampling rate* (usually uniform). A common notation for representing time series is

$$X = \{X_1, X_2, \ldots, X_N\} \tag{1.5}$$

where X_i are the samples corresponding to the time interval i and N is the length of the sequence.

Time series analysis comprises methods that attempt to understand certain characteristics of the data, like how it was generated (i.e. by which kind of source) or to make forecasts on the evolution of the time series itself.

Models for time series data can have many forms. Three broad classes of practical importance are the autoregressive (AR) models, the integrated (I) models, and the moving average (MA) models. Combining these basic techniques, one cane obtain more complex models, like

ARMA models. Further details on time series analysis can be found in [48]. These three classes depend linearly on previous data points. Non-linear dependence on previous data points is of interest because of the possibility of producing a chaotic time series [49], and is thus useful when dealing with chaotic systems.

Of course there are a lot of tools that can be applied for time series analysis. Even if not with the intent to be exhaustive, some of them are here listed:

Autocorrelation methods, making use of the autocorrelation function and the spectral density function;

Frequency analysis, that usually involver the application of the Fourier transform to investigate the series in the frequency domain;

Filtering, linear or nonlinear, often used to remove unwanted noise;

PCA, principal components analysis (or also empirical orthogonal function analysis), used to reduce the number of dimensions of the problem;

Time-frequency analysis techniques, like wavelet transforms, short-time Fourier transform, chirplet transform and fractional Fourier transform;

Chaotic analysis techniques, like correlation dimension, recurrence plots, recurrence quantification analysis and Lyapunov exponents;

Again, for further details refer to [48, 49].

From a pattern recognition system-level point of view, the methods we mentioned for time series analysis are the ones to be used in the front-end for *feature extraction* (cf. Figure 1.1). In pattern recognition, features are the individual measurable heuristic properties of the phenomena being observed. Choosing discriminating and independent features is crucial to any pattern recognition algorithm for being successful in classification. Features are usually numeric, but structural features such as strings and graphs are used in *syntactic pattern recognition* [50].

While different areas of pattern recognition obviously have different features, once the features are decided, they are classified by a much smaller set of algorithms. These include nearest neighbor classification in multiple dimensions, neural networks or statistical techniques such as Bayesian approaches.

Just to propose some examples, in character recognition, features may include horizontal and vertical profiles, number of internal holes, stroke detection. In speech recognition, features for recognizing phonemes can include noise ratios, length of sounds, relative power, filter matches. In spam detection algorithms, features may include whether certain email headers are present or absent, whether they are well formed, what language the email appears to be, the grammatical correctness of the text, markovian frequency analysis.

In all these cases, and many others, extracting features that are measurable by a computer is absolutely not trivial, and with the exception some neural networking and genetic techniques that automatically infer features, hand selection of good features forms the basis of almost all classification algorithms.

When it comes to the classification task itself, the choice of the algorithms to apply depends obviously on the kind of input the system has to decode, and on the kind of output to extract. This directly reflects on the kind of features coming out from the front-end (since, as already noted, they extract from the data the informations that are most important for the extraction of the desired meaning).

While there is a wide choice of algorithms suitable for classification (see Page 11), not all of them are the best choice if the system has to deal with time series (i.e. when system's input evolves in time).

A typical example is the speech recognition task, especially the continuous speech case. The most natural representation for speech is indeed a time series (a sequence of samples or other kind of features in time). Some techniques, like neural networks, are most suitable for the classification of static patterns (pictures for example) or already segmented inputs (like single spoken words), but have problems when the input is a continuous stream of data. In the case of neural networks some efforts to solve this problem have been made with *time-delay neural networks* [51], but some other problems arise because the increase in complexity in the training procedures is significant.

Some other techniques, instead, seem more suitable for time series classification, since they can deal with continuously changing inputs in the most natural way. The Hidden Markov Models (HMM) in particular, have some characteristics that make them one of the best choices for time series classification. Among the most interesting, there is the ability to automatically segment a continuous stream of data, taking into account each segment's context in time.

For this and for other reasons that will be better described in Chapter 2, this thesis focuses on the use of HMMs for pattern recognition on time series, along with the development of methods and tools to improve systems based on this paradigm.

Of course good results have been obtained also with other approaches, and many examples can be found in literature (see for example [52, 53]) but, as I'm going to show in Chapters 3 and 4, algorithms that can be applied to solve the problem through HMMs seem to be particularly suitable when developing real-time system with a mixed hardware-software architecture. In the perspective of building advanced user interfaces for complex systems, this feature make the use of HMMs one of the most interesting choices.

Chapter 2

Use of HMMs in pattern recognition

For reasons explained in 1.5.2, this thesis is going to focus on the use of HMMs for time series processing. In this chapter I am going to introduce the basic theory of Hidden Markov Models. Then I will show how this theory is applied to pattern recognition problems. Finally I will discuss the common problems and limitations with respect to other methods along with the advantages in using such technique.

2.1 Introduction to Hidden Markov Models

The theory of HMMs is definitely not new, and it has been applied to many different different problems since its first publications by Baum and his collaborators [54] between the end of the sixties and the beginning of the seventies. To be precise it begun to be applied some years later, along with the publication of the first tutorial documents that provided enough informations and details to allow for practical use.

2.1.1 Discrete Markov processes

The HMM is actually an extension of the concept of Markov chain. Markov chains are used to model systems that can be described as a set of N distinct states $S_1, S_2, \ldots, S_N$. In discrete Markov processes the time is sampled at regular steps $t = 1, 2, \ldots, N$. At each time step the system can change its state (or stay in the same state) according to a set of transition probabilities associated to the states. In a complete statistical description of the system, the probability of being in a certain state S_j at a certain time t would depend on the system's state at every time step before t. Formally this means that we can express the probability of being in state S_j at time t as

$$P[q_t = S_j | q_{t-1} = S_i, q_{t-2} = S_k, \cdots] \tag{2.1}$$

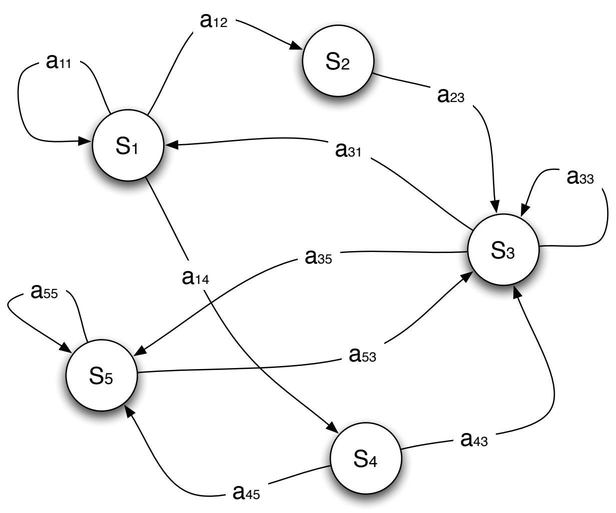


Figure 2.1: A graphical representation of a generic, not fully connected, 5-states markov chain.

A widely used approximation is to consider only first order processes, that is state probability in Equation (2.1) is assumed to depend only on previous system's state:

$$P[q_t = S_j | q_{t-1} = S_i, q_{t-2} = S_k, \cdots] \simeq P[q_t = S_j | q_{t-1} = S_i] \tag{2.2}$$

A further and really common approximation is to consider that state transition probabilities do not change in time. This means that it is possible to describe the system through a set of static state transition probabilities of the following form:

$$a_{ij} = P[q_t = S_j | q_{t-1} = S_i] \qquad \begin{array}{ll} \forall & 1 \leq i, j \leq N \\ \forall & t \end{array} \tag{2.3}$$

and, since transition probabilities should satisfy the basic stochastic constraints, it follows that

$$\begin{array}{ll} \sum_{j=1}^{N} a_{ij} = 1 & \forall \quad i \\ & \\ a_{ij} \geq 0 & \forall \quad i, j \end{array} \tag{2.4}$$

A typical representation of a Markov process, given the approximations described by Equation (2.3), is shown in Figure 2.1.

The kind of process just described is said to be an *observable process*. That is true because the model's output is the temporal sequence of states itself, and each state corresponds to an observable event. The model can be completely described by means of an $N \times N$ *transition matrix* $\boldsymbol{A}$, containing the transition probabilities a_{ij}, and an $N \times 1$ vector of initial state probabilities $\boldsymbol{\Pi}$, containing the probability π_i of each state to be the first of the sequence produced by the model.

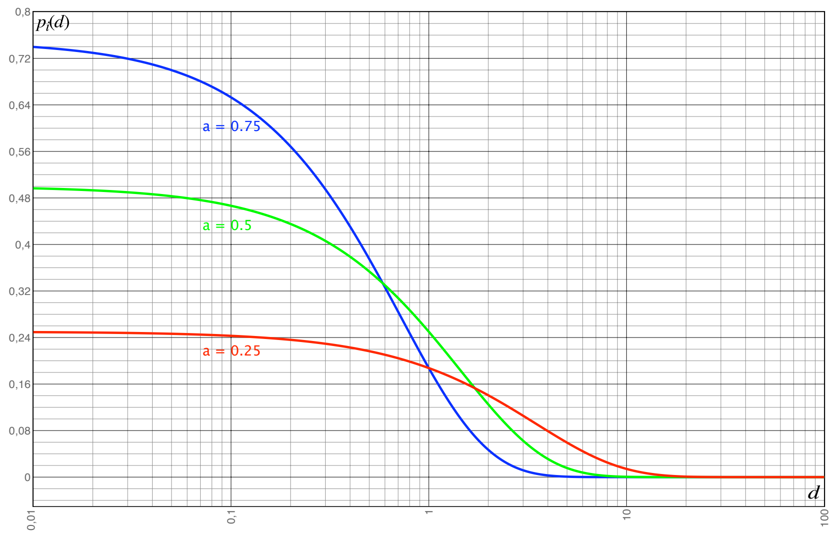


Figure 2.2: Probability density function for duration d in the same state for a first order, time invariant Markov process. Interpolated curves are shown for different values of the state's self-transition probability.

To clarify the operation of a Markov chain, let's assume we want to calculate the probability of a specified state sequence, given the model $\lambda = (\mathbf{\Pi}, \boldsymbol{A})$ with three states $\boldsymbol{S} = \{S_a, S_b, S_c\}$. If the chosen sequence is, for example, $\boldsymbol{O} = \{S_c, S_a, S_b, S_a\}$, its probability can be simply calculated using the expression

$$\begin{aligned} P(\boldsymbol{O}|\lambda) &= P(S_c, S_a, S_b, S_a|\lambda) \\ &= P(S_c) \cdot P(S_a|S_c) \cdot P(S_b|S_a) \cdot P(S_a|S_b) \\ &= \pi_c \cdot a_{ca} \cdot a_{ab} \cdot a_{ba} \end{aligned} \tag{2.5}$$

Another interesting problem is, given the model's current state is S_i, to compute the probability to stay in the same state for a time d. Formally the problem reduces to the calculation of the probability for the sequence

$$\boldsymbol{O} = \{S_i^{(1)}, S_i^{(2)}, \cdots, S_i^{(d)}, S_j^{(d+1)} \neq S_i\} \tag{2.6}$$

This probability can be calculated with the same method used in Equation (2.5), that is

$$P(\boldsymbol{O}|\lambda, q_1 = S_i) = (a_{ii})^{d-1}(1 - a_{ii}) = p_i(d) \tag{2.7}$$

where q_t denotes the sytem's state at time t. The quantity $p_i(d)$ is the discrete probability density function of duration d in state i. The exponential trend of this quantity, shown also in Figure 2.2, is a typical feature of Markov chains. Finally, given the duration *pdf*, it is possible to compute the expected value for the permanence in a given state i (measured in time steps) using the formula

$$\overline{d_i} = \sum_{d=1}^{\infty} d \cdot p_i(d) = \sum_{d=1}^{\infty} d \cdot (a_{ii})^{d-1} \cdot (1 - a_{ii}) = \frac{1}{1 - a_{ii}} \tag{2.8}$$

Though Markov processes can be useful when applied to simple problems, they have not enough modeling power and flexibility to handle complex behaviors. The Hidden Markov Models provide an extension to this theory that allows to build models for really complex phenomena.

2.1.2 From Markov chains to Hidden Markov Models

The fundamental difference between Markov chains and HMMs is that in the latter there are two processes going on concurrently: an *observable* process and a *hidden* process (from which the name *Hidden* Markov Model).

The concept is that if we look at a complex system, we can see the outputs of that system, while there are some other underlying processes that remain *hidden* to the observer. The overall behavior of HMMs is similar to that of Markov chains but, assuming to be in state S_i at time t, the observation is generated by a probability density function tied to S_i, instead of being the state itself (see Figure 2.3). So the hidden process is the one that controls the transitions between states, while the observable one rules the emission of symbols (or observations) according to the *present* state's PDF.

In order to better emphasize the differences between Markov chains and HMMs, let's consider a simple coin toss experiment. As one can expect the outcome of any coin toss experiment will be of the kind $\boldsymbol{O} = \{O_1, O_2, O_3, \cdots, O_T\} = \{Head, Cross, Cross, \cdots, Head\}$. The observer anyway does not know the number of coins used to perform the experiment, neither any other details on the process. At this point the question is: how to build a Markov model capable of reproducing accurately the observed phenomenon?

A possible choice could be to assume that a single coin is used for the experiment, and use a two–state Markov chain in which the states represent the *Head* and *Cross* events (see Figure 2.4.a). Such kind of model is observable and requires only one parameter to be specified (e.g. the probability of observing a *Head* event).

Using the HMMs, on the other hand, allow for more complex scenarios to be specified. One can assume, for example, that the coin is randomly chosen (in a set of two or three coins) before the toss. In the HMM each state would represent one of the coins, and each coin would have its own PDF for the emission of the Head/Cross events. Furthermore, the choice of the coin would be ruled by the transition probabilities between the states.

A two–state HMM (like the one shown in Figure 2.4.b) would require four parameters to be specified, whilst a three–state HMM (like the one in Figure 2.4.c) would require nine.

From a pure theoretical point of view, more complex HMMs are capable of modeling with higher detail the proposed experiment. Nonetheless the complexity that an HMM can reach in real applications is strongly limited by practical issues like, for example, the need for suitable

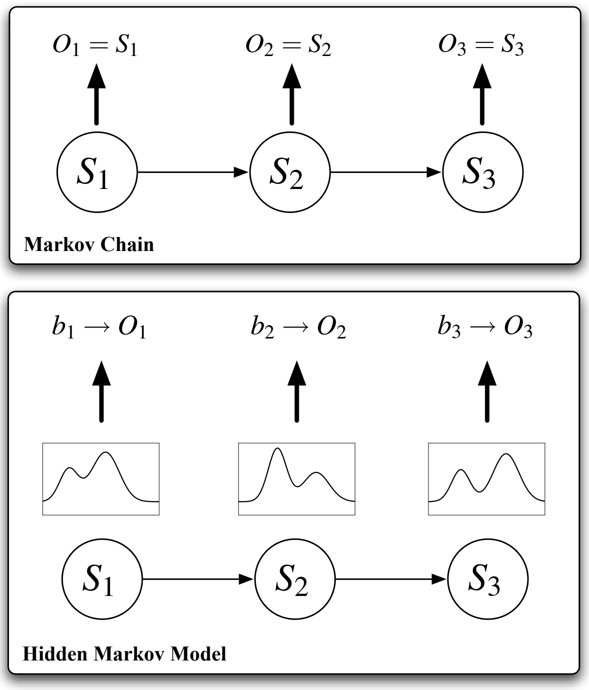


Figure 2.3: A comparison of observation output between Markov Chains and HMMs.

training data-sets. Furthermore it could also be that the experiment was actually done with a single coin. This eventuality would allow the Markov chain to outperform the HMMs since, in this particular case, the HMM approach, even if more powerful, is clearly less suitable for the problem.

Indeed the real usefulness of HMMs comes out when dealing with more challenging problems. For this purpose we consider another experiment (see Figure 2.5). We assume to have N different jars, each one full of colored balls. Let's say that the possible colors are M. The observation are produced in the following way: first a jar is randomly chosen, then a ball is extracted from the chosen jar and its color recorded as an observation, and finally the extracted ball is put back in the jar. The process is repeated as many times as needed to obtain the sequence of desired length.

It is quite clear to see that the most straightforward way to model this experiment is by using a HMM in which each state represents one of the jars. In that way the state emission PDFs will model the distribution of different colors inside each jar, while the transition probabilities will rule the choice of the jar from which to extract the color at each time step. It is also easy to guess that modeling such a problems using simple Markov chains would be extremely difficult.

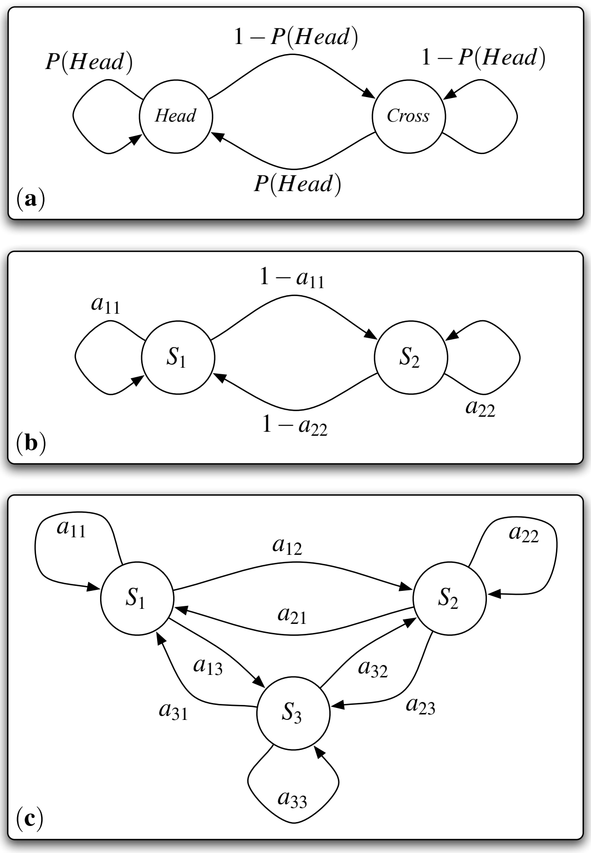


Figure 2.4: Models for a coin toss experiment: (a) first order, time invariant Markov chain; (b) two–state HMM; (c) three–state HMM.

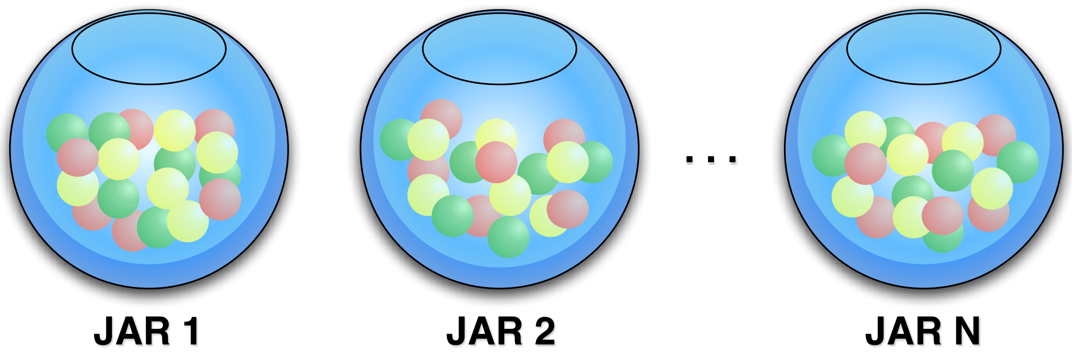
2.1.3 Characterization of HMMs

Now that is better understood how HMMs work and on which kind of problems is worth to apply them, it is necessary to define the theory in a more precise way.

An HMM can be completely described through the following set of parameters:

Number of states, N: For most of the applications the states have a precise correspondence with some of the elements that make up the application itself. In the particular class of *ergodic* models, the states are connected in such a way that it is possible to reach any state starting from any other, in a finite number of steps. The set of states is described using the notation $\boldsymbol{S} = \{S_1, S_2, \ldots, S_N\}$, while q_t is used to specify the state at time t.

Number of observable symbols, M: This is the dimension of the discrete alphabet used. The sequences output by the system that the model describes can be composed only by symbols from this alphabet. The set of symbols is expressed by the notation $\boldsymbol{V} = \{v_1, \ldots, v_M\}$.



$$
\begin{array}{ccc}
P(Color1) = b_1(1) & P(Color1) = b_2(1) & P(Color1) = b_N(1) \\
P(Color2) = b_1(2) & P(Color2) = b_2(2) & P(Color2) = b_N(2) \\
\vdots & \vdots \quad \cdots & \vdots \\
P(ColorM) = b_1(M) & P(ColorM) = b_2(M) & P(ColorM) = b_N(M)
\end{array}
$$

Figure 2.5: Representation of the *N–Jars* experiment for the production of color sequences, and its relations with HMM emission probabilities.

Transition probability matrix, A: The values in the transition probability matrix are the transition probabilities associated with the arcs that connect states (see for example Figure 2.4.c at page 18 for a reference HMM diagram). In particular

$$a_{ij} = P[q_{t+1} = S_j | q_t = S_i] \qquad 1 \leq i, j \leq N \tag{2.9}$$

In the particular case in which it is possible to reach any state of the model from any other in a single step, it is $a_{ij} > 0$ for every couple (i, j). Otherwise $a_{ij} = 0$ for some (i, j). This means that there is no direct connection between some couples of states.

Emission probability distributions, B: These are the discrete probability distributions associated with each of the states in the model. Each PDF defines the emission of symbols (or observations) from the state to which it is associated. For the set of PDFs the notation $\boldsymbol{B} = \{\boldsymbol{b}_1(k), \boldsymbol{b}_2(k), \ldots, \boldsymbol{b}_N(k)\}$ is used, where

$$\boldsymbol{b}_j(k) = P[v_k \text{ at time step } t | q_t = S_j] \qquad 1 \leq j \leq N, \quad 1 \leq k \leq M \tag{2.10}$$

Initial state probability distribution, Π: This vector contains the probability for each state to be the first of a sequence. This set of probability is addressed with $\boldsymbol{\Pi} = \{\pi_1, \pi_2, \ldots, \pi_N\}$ where

$$\pi_i = P[q_1 = S_i] \qquad 1 \leq i \leq N \tag{2.11}$$

After defining a model $\lambda(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{\Pi})$ (the quantities N,M and $\boldsymbol{V}$ will follow from these ones), it is possible to use the HMM as a sequence generator. The output observations would be of the type $\boldsymbol{O} = \{O_1, O_2, \ldots, O_T\}$ where each symbol O_t would be taken from the alphabet $\boldsymbol{V}$, and T is the length of the sequence.

A sequence is generated using the following algorithm:

1. time is initialized at $t = 1$;
2. an initial state $q_1 = S_i$ is chosen according to the initial state probability distribution $\boldsymbol{\Pi}$;
3. an observation $O_t = v_k$ is chosen according to the emission probability distribution $b_i(k)$ tied to the state $q_t = S_i$;
4. a transition towards a new state $q_{t+1} = S_j$ is performed according to the PDF of transition from state S_i (the PDF is actually the i^{th} row of the transition matrix $\boldsymbol{A}$);
5. time is incremented by one step ($t = t + 1$) and, if $t < T$, the process is repeated from step 3 (otherwise it ends).

Of course an HMM can be used not only to generate sequences, but also to model a sequence generator and check if an existing sequence has been generated from a similar system. This last use is most common in pattern recognition and allows to use the HMM as a classifier. In next section I'll show in detail how to use a HMM in this way.

2.2 Use of HMMs for pattern recognition

Assumed that we choose to use HMMs in our pattern recognition system, it is first of all necessary to solve some practical problems that arise whenever one tries to applied the theory described in the previous sections.

In general, all the needs of an HMM user can be fulfilled by solving three major problems:

Problem 1: Given an observation sequence $\boldsymbol{O} = \{O_1, O_2, \ldots, O_T\}$ and a model $\lambda(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{\Pi})$, how is it possible to compute efficiently the probability $P(\boldsymbol{O}|\lambda)$ (likelihood of the sequence given the model)? This is referred to as the *scoring problem*.

Problem 2: Given an observation sequence $\boldsymbol{O} = \{O_1, O_2, \ldots, O_T\}$ and a model $\lambda(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{\Pi})$, how is it possible to identify the sequence of model states $\boldsymbol{Q} = \{q_1, q_2, \ldots, q_K\}$ (with $K \geq T$) that best explains the production of the sequence? This is referred to as the *best state sequence problem*.

Problem 3: How is it possible to adjust the model's parameters $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{\Pi})$ in such a way that the probability $P(\boldsymbol{O}|\lambda)$ is maximized? This is referred to as the *training problem*.

The first problem is about how to compute the probability that an observation sequence has been generated by a given model. From another point of view it is about the evaluation of how much a certain model is suitable for modeling a given sequence (model scoring). The latter interpretation is the most interesting since, assuming to have several models available for a certain observation sequence, scoring them allows for the choice of the best one to be used for spotting sequences of the same kind.

The second problem is about the hidden part of the HMM: its solution provides informations on that part. It is anyway important to stress that in most cases there is no *correct* state sequence for a given observation sequence. In practice optimization methods are used to provide the best possible (but still sub-optimal) solution. In 2.2.2 will be shown how the choice of the optimization criteria strongly depends on the kind of use intended for the resulting state sequence (like making statistics, or trying to understand the internal behavior of the model, or validating a certain chosen topology).

The third and last problem is about how to optimize the model's parameters in such a way that it best represents the desired observation sequence. The sequences used for the optimization process are called *training sequences* and they make up the *training data set*. The process itself is referred to as *model training*.

To clarify the role of each of the three problems just described, I will show the example of a simple classifier for isolated spoken words. In this kind of systems the most straightforward solution is to use an *N-State* HMM to model each of the W words to be recognized. The input speech could be coded by a sequence of vectors containing spectral features, quantized so that they can be represented using a code-book containing a finite number M of symbols. This means that the observation space would be clustered into M regions. Thus a typical training sequence for the HMM that models the word w_i would be a series of code-book vectors obtained from the coding of several repetitions of that word.

The fist step to develop such a system is to build a model for each word. This implies the solution of the training problem, since the model parameters must be optimized to fit the training data set.

Then, if we want to refine the models, it is necessary to understand the meaning of each state of the HMM and, eventually, correct the model topology and the other parameters with the aim to obtain a model that better reproduces the real generator of the observations. Here the solution to the best state sequence problems comes in handy.

Finally, when the model for each of the W words is set up, it is possible to evaluate the system's performance on input sequences that are not part of the training data set (test data set). The classification process implies the evaluation of the likelihood of each model given the test input sequence, and the selection of the best scoring model. This process clearly relies on the solution to the scoring problem.

In the following sections the most popular solutions to each of the three problems will be shown.

2.2.1 Use of HMMs for classification

Since this thesis is focused on pattern recognition, the solution to the scoring problem is obviously one of the most interesting from that point of view. The aim, as already said, is to compute the probability of observing the sequence $\boldsymbol{O} = \{O_1, O_2, \ldots, O_T\}$ given the model $\lambda(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{\Pi})$, thus $P(\boldsymbol{O}|\lambda)$. The simplest method could seem to enumerate all the possible state sequences of length T of the kind $\boldsymbol{Q} = \{q_1, q_2, \ldots, q_T\}$. The probability of the observation sequence $\boldsymbol{O}$ given a state sequence $\boldsymbol{Q}$ and the model λ is given by

$$P(\boldsymbol{O}|\boldsymbol{Q}, \lambda) = \prod_{t=1}^{T} P(O_t|q_t, \lambda) = b_{q_1}(O_1) \cdot b_{q_2}(O_2) \cdots b_{q_T}(O_T) \tag{2.12}$$

where is assumed the statistical independence of observations. The probability of the state sequence $\boldsymbol{Q}$ is, on the other hand

$$P(\boldsymbol{Q}|\lambda) = \pi_{q_1} \cdot a_{q_1 q_2} \cdot a_{q_2 q_3} \cdots a_{q_{T-1} q_T} \tag{2.13}$$

The joint probability of $\boldsymbol{O}$ and $\boldsymbol{Q}$ is simply the product of (2.12) and (2.13), thus

$$P(\boldsymbol{O}, \boldsymbol{Q}|\lambda) = P(\boldsymbol{O}|\boldsymbol{Q}, \lambda) \cdot P(\boldsymbol{Q}|\lambda) \tag{2.14}$$

Finally the probability of the observation sequence given the model can be obtained by summing the joint probability in (2.14) over all the possible observation sequences of length T in the following way:

$$\begin{aligned} P(\boldsymbol{O}|\lambda) &= \sum_{\forall \boldsymbol{Q}(T)} \Big[P(\boldsymbol{O}|\boldsymbol{Q}, \lambda) \cdot P(\boldsymbol{Q}|\lambda) \Big] = \\ &= \sum_{\forall \boldsymbol{Q}(T)} \Big[\pi_{q_1} \cdot b_{q_1}(O_1) \cdot a_{q_1 q_2} \cdot b_{q_2}(O_2) \cdot a_{q_2 q_3} \cdots a_{q_{T-1} q_T} \cdot b_{q_T}(O_T) \Big] \end{aligned} \tag{2.15}$$

The main drawback with this method is the computational weight. The number of operations needed is proportional to $2TN^T$ since there are N^T possible sequences of length T, and the computation of the joint probability for each sequence requires approximately $2T$ operations (mostly multiplications). Taking into account a typical situation in which the HMM has $N = 5$ states and the sequence contains $T = 100$ symbols, the number of operations would be close to 10^{72}. This is clearly unmanageable.

A definitely more efficient method is the *forward–backward algorithm*. This method requires firstly to define the *forward variable* as

$$\alpha_t(i) = P(O_1, O_2, \ldots, O_t, q_t = S_i|\lambda) \tag{2.16}$$

The forward variable represents the joint probability of observing the partial sequence $\{O_1, O_2, \ldots, O_t\}$ and of the state S_i at time t given the model λ. The forward variables, for every couple (i, t), can be calculated by induction using the following algorithm:

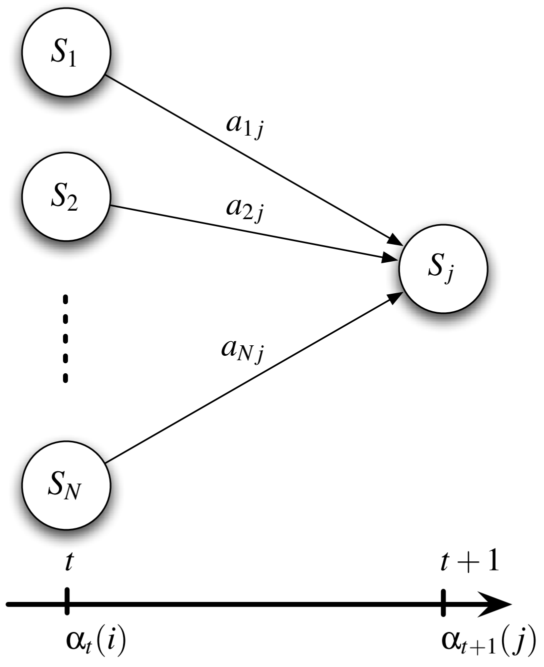


Figure 2.6: Graphical illustration of the recursion step in the calculus of the forward variables.

1. Initialization:
$$\alpha_1(j) = \pi_j \cdot b_j(O_1) \qquad \forall \quad 1 \leq j \leq N \tag{2.17}$$

2. Induction:
$$\alpha_{t+1}(j) = b_j(O_{t+1}) \cdot \sum_{i=1}^{N} \left[\alpha_t(i) \cdot a_{ij}\right] \qquad \forall \quad 1 \leq j \leq N, \quad 1 \leq t \leq (T-1) \tag{2.18}$$

3. Termination:
$$P(\boldsymbol{O}|\lambda) = \sum_{j=1}^{N} \alpha_T(j) \tag{2.19}$$

The first step initializes each of the N forward variables with the joint probability of state S_j to be the initial one and of the first symbol O_1 of the sequence to be emitted from that state.

The second step, the induction, is illustrated graphically in Figure 2.6: the main point is that each state S_j can be reached at time $t+1$ with a transition from one of the N possible states at time t. The quantity $\alpha_t(i) \cdot a_{ij}$ in (2.18) is the joint probability of observing the sequence $\{O_1, \ldots, O_t\}$, and reaching state S_j at time $t+1$ passing through state S_i at time t. Summing this quantity over all the possible states at time t yields the probability of being in state S_j at time t, taking into account all the possible partial sequences at time t. It is clear now that, multiplying that quantity by the probability $b_j(O_{t+1})$ of emitting the symbol O_{t+1} from state S_j at time $t+1$, the forward variable $\alpha_{t+1}(j)$ is found. The induction step must be calculated for each of the N states at each time step from 1 to $T-1$.

The third and last step provides the needed probability $P(\boldsymbol{O}|\lambda)$ as the sum of the N forward variables at time T. That is because each forward variable $\alpha_T(j)$ represents the probability of observing the whole sequence $\boldsymbol{O}$ and ending in state S_j. Summing over all the forward variables at time T simply takes into account each of the states as a possible final state.

Calculating $P(\boldsymbol{O}|\lambda)$ using the forward variables requires approximately N^2T operations (again, mainly multiplications) instead of the $2TN^T$ of the direct method. In the same practical case shown at Page 22 for the direct method (for $N = 5$ and $T = 100$), the number of operations needed by the forward variable method is 3000 against the 10^{72} of the direct method. The difference spans 69 orders of magnitude and makes the computation fairly manageable.

In a way really similar to the one just seen, it is possible to define a *backward variable* $\beta_t(j)$ in the following way:

$$\beta_t(j) = P(O_{t+1}, O_{t+2}, \ldots, O_T | q_t = S_j, \lambda) \tag{2.20}$$

The backward variable is complementary to the forward one. It represents the joint probability of observing the partial sequence $\{O_{t+1}, \ldots, O_T\}$ and being in state S_i at time t, given the model λ. Also the steps needed to compute it are analogous to the ones in (2.17–2.19):

1. Initialization:

$$\beta_T(j) = 1 \qquad\qquad \forall \quad 1 \leq j \leq N \tag{2.21}$$

2. Induction:

$$\beta_t(j) = \sum_{i=1}^{N} \left[a_{ij} \cdot b_j(O_{t+1}) \cdot \beta_{t+1}(j) \right] \qquad\qquad \forall \quad 1 \leq j \leq N, \quad (T-1) \geq t \geq 1 \tag{2.22}$$

As one can see, the initialization step defines arbitrarily the value of the backward variable at time T to be 1 for every state. The induction step, on the other hand, takes into account all the possible states S_j at time $t+1$ and the transitions a_{ij} that connects them to state S_i, and computes the probabilities for the partial sequences in a *backward* fashion. Again the emission of the symbol O_{t+1} is taken into account by the contribution of the emission probabilities $b_j(O_{t+1})$.

The graphical representation of this method is almost the same of the one shown in Figure 2.6 of Page 23, but mirrored horizontally. The number of operations needed for the computation of the N backward variables is again, like the case of the forward variables, proportional to N^2T.

Even if the forward variable has no role in the solution of the classification problem, it will have a fundamental role in the solution of the other two problems, as the reader will see in the following sections.

2.2.2 Exploring the hidden layer: optimal state sequences

As already noted, getting informations on the inner behavior of an HMM is extremely useful in order to optimize the model itself, and make it fit in the best way to the entity that has to be modeled. The inner behavior I am talking about is basically the sequence of states that actually produce the observed symbols.

The problem of reconstructing such a sequence has more than one solution, mainly because there is no correct solution in an absolute sense. The main issue is defining what is the *optimal state sequence*. There are many optimum criteria that can be applied.

An example is the choice of the states that have *individually* the highest probability, in such a way that the number of *single* correct states is maximized. To solve the problem in this way, it is useful to define the following variable:

$$\gamma_t(i) = P(q_t = S_i|\boldsymbol{O}, \lambda) \tag{2.23}$$

This variable clearly represents the probability of being in state S_i at time t given an observation sequence and a model. It can be easily expressed, using the forward and backward variables, in the following way:

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{P(\boldsymbol{O}|\lambda)} = \frac{\alpha_t(i)\beta_t(i)}{\sum_{j=1}^{N} \alpha_t(j)\beta_t(j)} \tag{2.24}$$

Here the forward variable takes into account the part of the observation sequence that goes from the beginning to time step t, and the backward variable for the remaining part of the sequence. The denominator is a normalization factor that actually makes $\gamma_t(i)$ a probability density function, since it yields:

$$\sum_{i=1}^{N} \gamma_t(i) = 1 \tag{2.25}$$

The computation of the γ variable enables to find, for every t, the state with highest probability, through the solution of

$$\underset{1\leq i\leq N}{argmax}[\gamma_t(i)] \tag{2.26}$$

Anyway, even if this solution maximizes the number of individually most likely states, there could be some problems with the resulting sequence. In fact the transition matrix is not considered at all in the optimization process and, if the model has some transitions with zero probability ($a_{ij} = 0$ for some (i, j)), the method could produce a sequence not consistent with the model. Actually this problem could be overcome by optimizing groups of two or three states, or take at each time step the best state toward which there is a non–zero transition.

As a matter of fact it is possible to compute directly the whole sequence using a more complex, yet more efficient technique, based on dynamic programming methods: the *Viterbi algorithm*. The Viterbi algorithm [55] allows to obtain a *unique* most likely sequence of states $\boldsymbol{Q} = \{q_1, \ldots, q_T\}$ given an observation sequence $\boldsymbol{O} = \{O_1, \ldots, O_T\}$, since it considers the sequence as a whole, and not as a collection of single entities.

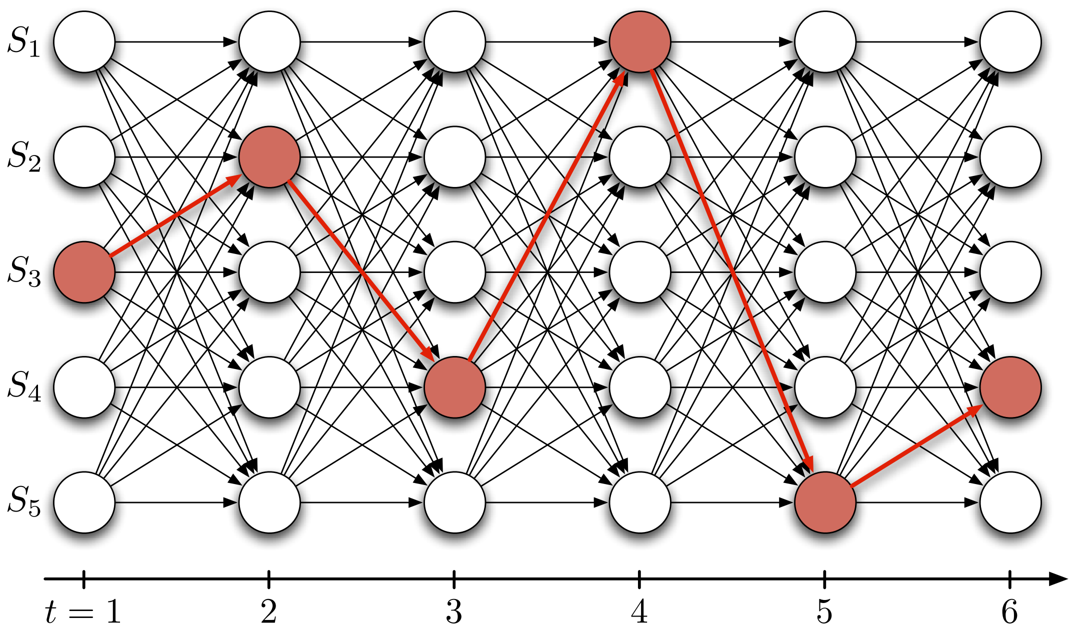


Figure 2.7: Trellis representation for a fully connected 5–state HMM over a time span of six steps. A generic path through the trellis is marked, corresponding to the state sequence $\boldsymbol{Q} = \{q_1, \ldots, q_6\} = \{S_3, S_2, S_4, S_1, S_5, S_4\}$.

In order to treat the Viterbi algorithm, it is useful to interpret the time evolution of an HMM by displaying it like a *trellis* structure. A trellis representation for a 5–state HMM over a small window of time is shown in Figure 2.7. The horizontal axis represents the time and, for each time step, all the HMM's states are shown along the vertical axis. In this kind of diagram a sequence of states can be drawn as a path through the trellis. As we will see in the following, treating state sequences as paths makes the analysis of the subject more understandable.

Given these premises, to introduce the Viterbi algorithm the quantity $\delta_t(i)$ must be defined as

$$\delta_t(j) = \max_{q_1, q_2, \ldots, q_{t-1}} \Big[P(q_1, q_2, \ldots, q_t = j; O_1, O_2, \ldots, O_t | \lambda) \Big] \tag{2.27}$$

This is the state sequence (*path* with maximum probability, among all the possible paths that end in state S_j at time t and take into account the partial observation sequence $\{O_1, \ldots, O_t\}$. It is easy to see that, for induction, the δ variable at next time step can be calculated as

$$\delta_{t+1}(j) = \max_{1 \leq i \leq N} \Big[\delta_t(i) \cdot a_{ij} \Big] \cdot b_j(O_{t+1}) \tag{2.28}$$

To obtain the state sequence itself, it is necessary to keep track of the argument i that maximizes Equation (2.28). Usually this is done by means of a set of variables labeled $\psi_t(j)$. The complete procedure used to reconstruct the whole best path through the trellis, for a given observation sequence, is the following:

1. Initialization:

$$\begin{aligned} \delta_1(j) &= \pi_j \cdot b_j(O_1) \\ \psi_1(j) &= 0 \end{aligned} \qquad \forall \quad 1 \leq j \leq N \tag{2.29}$$

2. Induction:

$$\begin{aligned} \delta_t(j) &= \max_{1 \leq j \leq N} \Big[\delta_{t-1}(i) \cdot a_{ij} \Big] \cdot b_j(O_t) \\ \psi_t(j) &= \underset{1 \leq j \leq N}{argmax} \Big[\delta_{t-1}(i) \cdot a_{ij} \Big] \end{aligned} \qquad \begin{aligned} &\forall \quad 1 \leq j \leq N \\ &\forall \quad 2 \leq t \leq T \end{aligned} \tag{2.30}$$

3. Termination:

$$\begin{aligned} p^* &= \max_{1 \leq j \leq N} \Big[\delta_T(j) \Big] \\ q_T^* &= \underset{1 \leq j \leq N}{argmax} \Big[\delta_T(j) \Big] \end{aligned} \tag{2.31}$$

4. Backtracking:

$$q_t^* = \psi_{t+1}(q_{t+1}^*) \qquad \forall \quad (T-1) \geq t \geq 1 \tag{2.32}$$

2.2.3 HMM Training

The last and most complex problem to solve is how to train an HMM model. In other words, how to tune the parameters in such a way that the model becomes suitable for producing observation sequences of the same kind of the ones in a chosen training data set. More precisely the goal is to optimize the parameters $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{\Pi})$ so that the probability of the training observation sequence is maximized.

Since an analytical solution to this problem is not yet available, the most common approach consists in finding a parameter set that locally maximizes the probability $P(\boldsymbol{O}|\lambda)$. One of the most common methods used to achieve this result is the *Baum–Welch* algorithm [54], that performs an iterative estimation of the parameters. An equivalent method, that will not be treated here, is the *EM* procedure (*Expectation–Maximization*, [56]) that makes use of gradient descent techniques.

To describe the Baum–Welch algorithm, the variable $\xi_t(i,j)$ is defined as

$$\xi_t(i,j) = P(q_t = S_i, q_{t+1} = S_j | \boldsymbol{O}, \lambda) \tag{2.33}$$

that expresses the probability of transition between the states S_i and S_j across the time interval $(t, t+1)$, given and observation sequence and a model. The ξ variable can be directly expressed in terms of the forward and backward variables:

$$\xi_t(i,j) = \frac{\alpha_t(i) a_{ij} b_j(O_{t+1}) \beta_{t+1}(j)}{P(\mathbf{O}|\lambda)} = \frac{\alpha_t(i) a_{ij} b_j(O_{t+1}) \beta_{t+1}(j)}{\sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_t(i) a_{ij} b_j(O_{t+1}) \beta_{t+1}(j)} \tag{2.34}$$

Here the numerator provides $P(q_t = S_i, q_{t+1} = S_j, \mathbf{O}|\lambda)$, that is the joint probability of the two states at the required time points and of the observation sequence. The denominator transforms the value in a probability measure through normalization (like in 2.25 at Page 25).

Furthermore, by considering all the possible states at time $t+1$ as a destination for the transition, it is possible to obtain a relation with the γ variable:

$$\gamma_t(i) = P(q_t = S_i|\mathbf{O}, \lambda) = \sum_{j=1}^{N} \xi_t(i,j) \tag{2.35}$$

One can observe that by summing the variable $\gamma_t(i)$ over all the possible values of t for the given observation, the result is a quantity that can be interpreted as the probability of passing through the state S_i or, in other words, the probability of transition from the state S_i during the time interval spanned by the observation.

In a similar way, by summing the variable $\xi_t(i,j)$ over all the possible time values given the observation (up to $T-1$), it yields the probability of transition between the states S_i and S_j during the time interval spanned by the observation.

Thus

$$\begin{aligned} \textstyle\sum_{t=1}^{T-1} \gamma_t(i) &= P(\text{transition from } S_i) \\ \textstyle\sum_{t=1}^{T-1} \xi_t(i,j) &= P(\text{transition from } S_i \text{ to } S_j) \end{aligned} \tag{2.36}$$

The relations (2.36) can be used to obtain a method for the recursive estimation of the parameters of a HMM, that is based on the concept of *event frequency*. From this point of view the initial state probabilities π_i are estimated as the probability of each state S_i at time $t = 1$. The transition probabilities A_{ij} are the ratio of the probability of transition between the states S_i and S_j and that of the transitions from the state S_i. Finally the emission probabilities $b_j(k)$ are obtained from the ratio between the probability of passing through state S_j while emitting the symbol v_k, and the simple probability of passing through state S_j.

Formally the formulas obtained are

$$\overline{\pi}_i = \gamma_1(i) \tag{2.37}$$

$$\overline{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i,j)} \tag{2.38}$$

$$\overline{b}_j(k) = \frac{\sum^{T}{}_{\substack{t=1 \\ O_t = v_k^T}} \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)} \tag{2.39}$$

The application of equations (2.37–2.38–2.39) to an HMM produces a new model with modified parameters:

$$\lambda(\boldsymbol{A},\boldsymbol{B},\boldsymbol{\Pi}) \Longrightarrow \overline{\lambda}(\overline{\boldsymbol{A}},\overline{\boldsymbol{B}},\overline{\boldsymbol{\Pi}}) \tag{2.40}$$

Baum [54] showed that by repeating the estimation process, starting from the parameters produced by the previous iteration, two outcomes are possible:

1. The new parameters correspond to an improved model, that is most likely to produce the training sequence. Thus
$$P(\boldsymbol{O}|\overline{\lambda}) > P(\boldsymbol{O}|\lambda) \tag{2.41}$$

2. The starting parameters of λ represent a critical point of the likelihood function. This means that it is not possible to further improve the model. Thus
$$P(\boldsymbol{O}|\overline{\lambda}) = P(\boldsymbol{O}|\lambda) \tag{2.42}$$

In the second case one says that the model is an estimate of *maximum likelihood* for the HMM given the training sequence. It is important to stress that the forward–backward algorithm allows to find only local maxima for the likelihood function, and that the function itself is, in real world applications, extremely complex, and contains a high number of relative maxima.

The formulae shown for the iterative estimation of HMM parameters can be found also through the maximization of the *Baum's auxiliary function*:

$$Q(\lambda,\overline{\lambda}) = \sum_{Q} P(Q|\boldsymbol{O},\lambda)\log[P(\boldsymbol{O},Q|\overline{\lambda})] \tag{2.43}$$

This function can be maximized using the known techniques of constrained optimization. Baum showed that

$$\max_{\overline{\lambda}}[Q(\lambda,\overline{\lambda})] \Longrightarrow P(\boldsymbol{O}|\overline{\lambda}) \geq P(\boldsymbol{O}|\lambda) \tag{2.44}$$

The one of the most interesting features of the Baum–Welch procedure is that the stochastic constraints on the model parameters

$$\begin{array}{lcll} \sum_{i=1}^{N}\overline{\pi}_i & = & 1 & \\ \sum_{j=1}^{N}\overline{a}_{ij} & = & 1 & 1\leq i\leq N \\ \sum_{k=1}^{M}\overline{b}_j(k) & = & 1 & 1\leq j\leq N \end{array} \tag{2.45}$$

are automatically satisfied.

A last method, not exposed here, to obtain the equations (2.37–2.38–2.39), is to maximize the quantity $P(\boldsymbol{O}|\lambda)$ using the Lagrange multipliers and the constraints in (2.45).

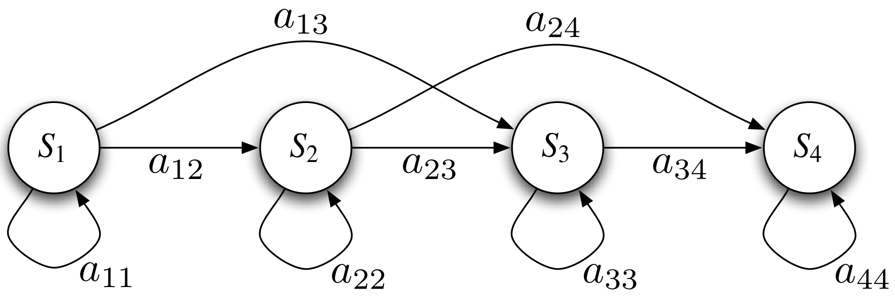


Figure 2.8: A 4–state Bakis (or left–right) HMM.

2.2.4 HMM topologies

Up to now, even if implicitly, only ergodic (fully connected) HMMs have been taken into account, without focusing on the effect of zero–probability state transitions.

In many applications, anyway, an ergodic model is not the optimal way to represent the observation's properties. A widely used non–ergodic model, used mostly when the observation sequence is derived from a time series, is the *left–right* or *Bakis* HMM (shown in Figure 2.8). In these models the state sequences have a common characteristic: while the time index increases, the state index always increases (or remains the same). From a graphical point of view, taking Figure 2.8 as a reference, the active state "moves" from left to right as time increases. It is fairly clear that such kind of models is particularly suitable to reproduce signals that vary with time (like speech, for example).

Formally a Bakis HMM can be described with the following relations:

$$\begin{aligned} a_{ij} &= 0 \qquad \forall\, j \le i \\ \pi_i &= \begin{cases} 0 & i \neq 1 \\ 1 & i = 1 \end{cases} \end{aligned} \tag{2.46}$$

This means that there are no *backwards* transitions, and any state sequence starts in state S_1 and ends in state S_N. Usually another constraint is added to left–right models, in order to avoid "long jumps":

$$a_{ij} = 0 \qquad \forall\, j > i + \Delta \tag{2.47}$$

In the model shown in Figure 2.8, for example, the value of the parameter Δ is 2, in fact there is no transition between states S_1 and S_4.

Though most of the applications use the topologies described, the possible variations are many. To add flexibility to a Bakis model, for example, one can use a parallel version, like the one shown in Figure 2.9.

It is important to stress that changes in the topology of the model, do not interfere in any way with the training procedures described in 2.2.3. This happens because the shown training algorithms have no effect on parameters that have zero value at the beginning of the computations.

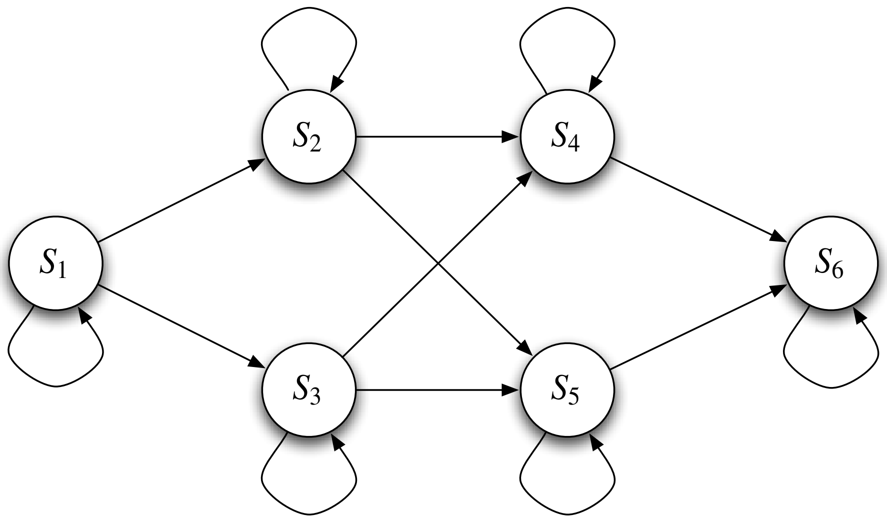


Figure 2.9: A HMM with a parallel Bakis topology.

2.2.5 HMM with a continuous observation space

The HMMs taken into consideration in the previous sections were capable of producing observation sequences made up only of symbols taken from a finite alphabet. This feature translated in the use of discrete PDFs for the emission probabilities.

The main issue with discrete models is that, obviously, real world signals are continuous: even if it is possible, and widely used, to quantize continuous feature spaces into finite codebooks, the unavoidable degradation introduced by this process could be unacceptable for certain applications. This problem can be avoided only with the use of continuous PDFs.

Of course this introduces some complications in handling the models. In particular, to ensure the iterative estimation of the model parameters to be consistent, it is necessary to put some restriction on the kind of continuous distributions that can be used.

The most general form of PDF for which there are means for estimating model parameters with iterative procedures, is the following:

$$b_j(\boldsymbol{O}) = \sum_{m=1}^{M} c_{jm} \cdot \mathcal{N}\left[\boldsymbol{O}, \boldsymbol{\mu}_{jm}, \boldsymbol{U}_{jm}\right] \qquad 1 \leq j \leq N \tag{2.48}$$

where $\boldsymbol{O}$ is the observation sequence to be modeled, c_{jm} a mixture coefficient for the m^{th} component of state j, and $\mathcal{N}$ a concave or elliptically symmetric PDF (like a gaussian, for example), defined by its mean vector $\boldsymbol{\mu}_{jm}$ and covariance matrix $\boldsymbol{U}_{jm}$ (for the m^{th} component of state j).

The mixture coefficients must satisfy the statistical constraints

$$\begin{aligned} \textstyle\sum_{m=1}^{M} c_{jm} &= 1 \qquad 1 \le j \le N \\ c_{jm} &\ge 0 \qquad \begin{array}{c} 1 \le j \le N \\ 1 \le m \le M \end{array} \end{aligned} \tag{2.49}$$

so that the function has the characteristics of a PDF, that is

$$\int_{-\infty}^{\infty} b_j(x)dx = 1 \qquad 1 \le j \le N \tag{2.50}$$

A mixture like the one described by equation (2.48) can be applied in a wide variety of problems, since it is capable of approximating with any precision almost every continuous PDF.

The formulae for the iterative estimation of the parameters are found, in the case of continuous PDFs, in a way similar to the one shown in Section 2.2.3 for the discrete case (see [57, 58]). The parameters are updated at each iteration according to the following:

$$\bar{c}_{jk} = \frac{\sum_{t=1}^{T} \gamma_t(j,k)}{\sum_{t=1}^{T} \sum_{k=1}^{M} \gamma_t(j,k)} \tag{2.51}$$

$$\bar{\boldsymbol{\mu}}_{jk} = \frac{\sum_{t=1}^{T} \gamma_t(j,k) \cdot \boldsymbol{O}_t}{\sum_{t=1}^{T} \gamma_t(j,k)} \tag{2.52}$$

$$\overline{\boldsymbol{U}}_{jk} = \frac{\sum_{t=1}^{T} \gamma_t(j,k) \cdot (\boldsymbol{O}_t - \boldsymbol{\mu}_{jk}) (\boldsymbol{O}_t - \boldsymbol{\mu}_{jk})'}{\sum_{t=1}^{T} \gamma_t(j,k)} \tag{2.53}$$

The quantity $\gamma_t(j,k)$ is here the probability of state S_j at time t when the emission of the observation vector $\boldsymbol{O}_t$ is modeled only by means of the k^{th} component of the mixture. It can be computed using the following formula:

$$\gamma_t(j,k) = \left[\frac{\alpha_t(j)\beta_t(j)}{\sum_{j=1}^{N} \alpha_t(j)\beta_t(j)} \right] \cdot \left[\frac{c_{jk}\,\mathcal{N}(\boldsymbol{O}_t, \boldsymbol{\mu}_{jk}, \boldsymbol{U}_{jk})}{\sum_{m=1}^{M} c_{jk}\,\mathcal{N}(\boldsymbol{O}_t, \boldsymbol{\mu}_{jk}, \boldsymbol{U}_{jk})} \right] \tag{2.54}$$

It is interesting to note that if the mixture degenerates to a single gaussian component ($M = 1$), then the relation (2.54) reduces to the value $\gamma_t(j)$ defined in 2.24 (Page 25) for the discrete case.

2.2.6 Some more variations: null transitions, tied states, state durations

The HMMs can be adapted to many different situations by introducing some useful variations on the theory exposed in the previous sections.

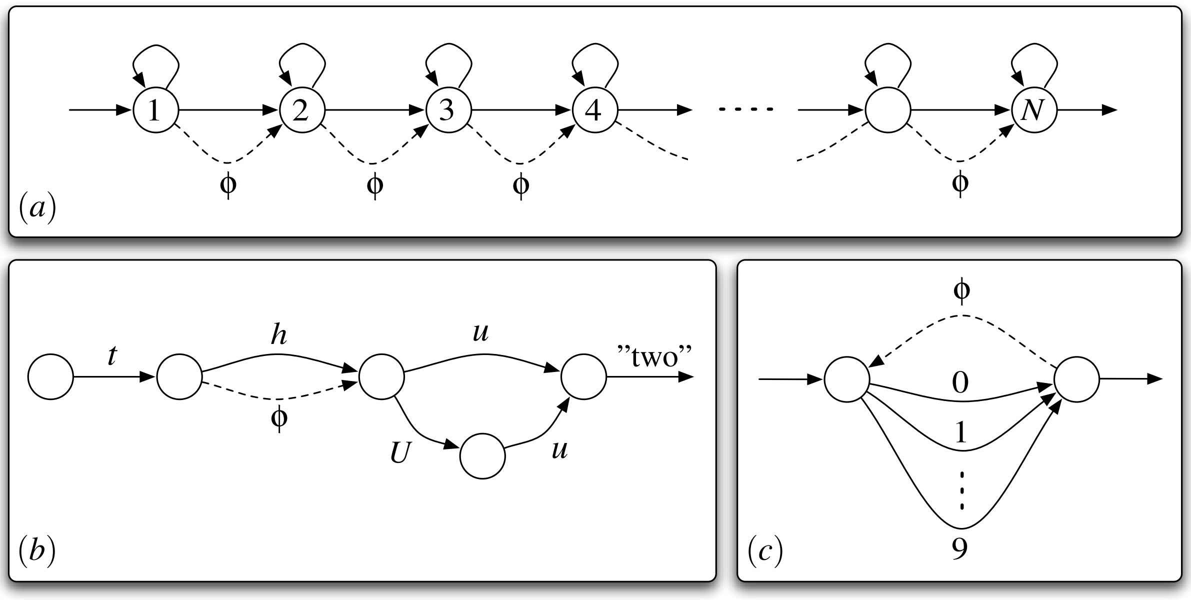


Figure 2.10: Examples of application for null transitions: (a) a HMM with a high number of states able to produce also short sequences through the use of null transitions; (b) finite state machine that models the production of the word "two" at a phonetic level; (c) finite state machine that models a grammar for the production of digit sequences.

Null transitions

Null transitions are usually labeled in the graphical representation of HMMs with the symbol ϕ, and the corresponding connection will be drawn as a dotted line. A null transition is always performed instantly, so that any number of null transitions can be done without advancing in time. Often, when using null transitions, the HMM framework is changed so that the emission of symbols is tied to the transitions themselves instead of the states. Some examples of HMMs containing null transitions are shown in Figure 2.10.

Figure 2.10.a shows a left–right HMM with a large number of states and null transitions between every couple of adjacent states, in which it is so possible to jump forward over any number of states instantly. This topology allows to generate both short and really long sequences with a single model.

Figure 2.10.b shows a *finite state network* (FSN) that is capable of modeling sequences with "optional" segments. In particular it refers to an acoustic model for multiple pronunciations of the word "two", to be used in a speech recognition system. Here every connection corresponds to a phonetic HMM, modeling a single sound. It can be seen that the model allows for up to four pronunciations of the word "two" to be recognized: *t-h-u*, *t-h-U-u*, *t-u*, and *t-U-u* (these last two passing through the null transition). Here the use of the null transition allow the sound *h* to be an option in the pronunciation of the word.

The last example, in Figure 2.10.c, shows a *grammar* model for the production of digit sequences, implemented again by means of a FSN. The null transition in this case allows for the generation of sequences of any length.

Tied states and shared distributions

Another interesting variation on the HMM theory is the introduction of *tied states*. The idea is to cluster the state parameters of a group of models into a set of classes whose number is considerably lower than the total number of states. After the clustering, all the states belonging to a certain cluster will share a common set of parameters. This solution can reduce greatly the number of independent parameters thus simplifying model training.

Furthermore, the application of clustering can sometimes flatten some slight differences between different states that actually model similar (if not identical) events occurring in the sequences to be modeled, in such a way that robustness is improved along with classification accuracy. Finally having a smaller number of parameters to estimate, enables to use more complex models that, in this way, can be trained with a training data set that, without shared parameters, would produce under-training problems (see Section 2.3.2).

Starting from a general set of models that do not share any PDF, a procedure should be applied in order to merge the most similar sets of PDFs into a new one that can suitably replace the the whole merged set. Firstly, to perform the clustering procedure, a measure of the distortion introduced by the merging process is needed. The increase of entropy is one of the commonly used measures for this purpose.

In the example of discrete PDFs (the extension to the continuous case can be easily made by replacing sums with integrals), taking two distributions α and β, containing respectively the elements a_i and b_i, the entropy of the distribution obtained by merging the two can be expressed as

$$H_{\alpha+\beta} = -\sum_i \frac{a_i + b_i}{\sum_i a_i + \sum_i b_i} \log\left(\frac{a_i + b_i}{\sum_i a_i + \sum_i b_i}\right) \tag{2.55}$$

Calculating the increase in entropy produced by the merging yields

$$\Omega(\alpha, \beta) = H_{\alpha+\beta}\left(\sum_i a_i + \sum_i b_i\right) - H_\alpha \sum_i a_i - H_\beta \sum_i b_i \geq 0 \tag{2.56}$$

It can be shown that $\Omega(\alpha, \beta)$ reaches its minimum (equal to zero) only when $\alpha = \beta$, that is when we are merging two identical distributions. Thus the clustering can be made by merging pairs of PDFs in such a way that the relation (2.56) is minimized.

A typical clustering algorithm that implements a clustering procedure for distribution sharing is the following:

1. Estimate the parameters for all the models.

2. Create a cluster for each distribution present in the model set (one for each state of each HMM).

3. *Merging:* find all the similar pairs of PDFs using the variation of entropy as a measure, and merge them into a new set of clusters.

4. *Shifting:* for each PDF contained in each cluster, move it in another cluster if the movement (shift) causes a reduction in the overall entropy. Repeat the shifting procedure until no more improvement is possible.

5. Repeat the procedure from step 3 until a predefined convergence criterion is met (a chosen number of clusters, for example).

Step 4 is the most complex and implements a heuristic optimization algorithm. The mentioned overall entropy is a weighted sum of the entropy for each of the clusters (*grand weighted entropy*). In order to reduce the computational weight of this step, a termination rule is often used. It is furthermore important to note that moving a distribution from one cluster to another is computationally expensive since it involves a new evaluation of the entropy for the two clusters.

To address this issue the matrix of entropy increment $\mathbf{\Omega}$ can be used to spot the shifts that produce minor improvements. The matrix $\mathbf{\Omega}$ contains all the values $\Omega(\alpha, \beta)$ for all the possible couples (α, β) and is calculated the first time that step 3 (see Page 35) is executed. Thus before moving a certain distribution α from cluster $\mathcal{P}$ to cluster $\mathcal{Q}$, the mean entropy increment before and after the shift is evaluated for that distribution:

$$\begin{aligned} \Omega(\alpha) &= \frac{\sum_{\substack{\beta \neq \alpha \\ \beta \in \mathcal{P}}} \Omega(\alpha,\beta)}{size(\mathcal{P})-1} \\ \Omega'(\alpha) &= \frac{\sum_{\beta \in \mathcal{Q}} \Omega(\alpha,\beta)}{size(\mathcal{Q})} \end{aligned} \tag{2.57}$$

The total entropy will be calculated only if the ratio $\Omega'(\alpha)/\Omega(\alpha)$ is less than a certain threshold.

Explicit state duration

One of the problems that could arise when modeling with HMMs is the lack of control on the time of permanence in each state (or *state duration*). As already seen in Section 2.1.1 the state duration PDF $p_i(d)$ (see (2.8), Page 16) has an exponential form (see Figure 2.2, Page 15). For many real–world signals, this kind of duration PDF is not suitable and would be more useful to model it with an arbitrary function. This can be obtained, for any state S_i, setting to zero the self transition probability a_{ii} and introducing an explicit duration PDF $p_i(d)$ in such a way that the number d of observations produced by that state will depend directly on the PDF itself. For a model of this kind the emission algorithm would be the following:

1. A starting state $q_1 = S_i$ is chosen according to the initial state PDF $\mathbf{\Pi}$.

2. A duration d_1 for state q_1 is chosen according to the duration PDF $p_{q_1}(d_1)$ (to avoid too much computation the PDF is usually truncated to a maximum duration D).

3. A sequence $\{O_1, O_2, \dots, O_{d_1}\}$ of length d_1 is generated according to the *joint* emission probability $b_{q_1}(O_1, O_2, \dots, O_{d_1})$. In practice the observations are usually considered independent in such a way that the joint PDF can be easily calculated as the product of the factors $b_{q_1}(O_t)$ for all $t \in (1, \dots, d_1)$.

4. The following state $q_2 = S_j$ is chosen according to the transition probability distribution $a_{q_1 q_2}$ under the constraint $a_{q_1 q_1} = 0$.

5. The steps are repeated for the following states.

The use of HMMs with explicit state duration requires some variations (not treated in this work) in the formulas for the computation of $P(\boldsymbol{O}|\lambda)$ and for the iterative estimation of parameters.

Even if in some applications the use of explicit duration clearly improves the quality of the models, the increase in computational cost and memory use is significant. In fact, if the duration PDF are all truncated to the same maximum duration D, then the computational cost will be $D^2/2$ times the one needed for a standard HMM, while the memory occupation will be increased approximately by D. This means that for $D = 25$, a reasonable value for many applications, the computational cost will be 300 times the original one. Of course also the number of parameters to estimate will increase, with the already mentioned consequences on training data availability. To partially deal with the increase in number of parameters to estimate, parametric functions, like gaussians, can be used as duration PDF.

A final and important consideration is that when using HMM topologies in which the number of states is proportional to the duration of the event to be modeled, like the Bakis one (see 2.2.4), the use of explicit duration produces no quality improvements in the model, and the only effect is to make the computation longer.

2.2.7 Multiple observation sequences

In many of the possible topologies for an HMM, among which the Bakis one, a single observation sequence is not enough to estimate all the parameters of the model in a reliable way. This happens since, in these kind of topologies, the states are transient for definition, and each of them can produce only a limited number of observations, usually not enough for a good training. The model's parameters must be thus estimated using multiple observation sequences.

The changes to apply on the iterative re–estimation procedure are here explained. First of all let us assume to have a set $\boldsymbol{O}$ containing K independent observation sequences:

$$\begin{aligned} \boldsymbol{O} &= \left[\boldsymbol{O}^{(1)}, \boldsymbol{O}^{(2)}, \dots, \boldsymbol{O}^{(K)}\right] \\ \textit{where} \quad \boldsymbol{O}^{(k)} &= \left[O_1^{(k)}, O_2^{(k)}, \dots, O_{T_k}^{(k)}\right] \end{aligned} \tag{2.58}$$

The goal of the training procedure is to maximize the probability of all the set of observations:

$$P(\boldsymbol{O}|\lambda) = \prod_{k=1}^{K} P\left(\boldsymbol{O}^k|\lambda\right) = \prod_{k=1}^{K} P_k \tag{2.59}$$

Since the formulae for the iterative re–estimation of parameters are based on event frequency (see (2.37,2.38,2.39) at Page 28), it is sufficient to sum the occurrence frequencies for each of the sequences in order to obtain the needed formulae:

$$\bar{a}_{ij} = \frac{\sum_{k=1}^{K} \frac{1}{P_k} \sum_{t=1}^{T_k-1} \alpha_t^{(k)}(i) a_{ij} b_j\left(O_{t+1}^{(k)}\right) \beta_{t+1}^{(k)}(j)}{\sum_{k=1}^{K} \frac{1}{P_k} \sum_{t=1}^{T_k-1} \alpha_t^{(k)}(i) \beta_t^{(k)}(i)} \tag{2.60}$$

$$\bar{b}_i(\ell) = \frac{\sum_{k=1}^{K} \frac{1}{P_k} \sum_{\substack{t=1 \\ O_t=v_\ell}}^{T_k-1} \alpha_t^{(k)}(i) \beta_t^{(k)}(i)}{\sum_{k=1}^{K} \frac{1}{P_k} \sum_{t=1}^{T_k-1} \alpha_t^{(k)}(i) \beta_t^{(k)}(i)} \tag{2.61}$$

2.3 Limitations and common problems

2.3.1 Choice of initial parameters for training

As already said in Section 2.2.3, the likelihood function in the training process contains many relative maxima. This is one of the problems that makes HMM training definitely non trivial. There is no way to know if the solution found is the optimal one. Anyway, some methods have been developed in order to reduce as much as possible the probability to fall into highly sub–optimal solutions.

In practice it is really important to choose a suitable starting point for the training, in order to avoid falling into local maxima. The starting point is obviously defined by the initial parameter set. Unfortunately there is no direct method to choose these parameters. The most common methods are fundamentally empiric, and experience has shown that, for $\boldsymbol{\Pi}$ and $\boldsymbol{A}$, random parameters with uniform distribution, provide good training results in most of the cases. Issues arise when choosing the initial parameters for the emission distributions $\boldsymbol{B}$: starting from a good estimate is fundamental when dealing with continuous mixtures made of gaussian components.

The starting estimate for the $\boldsymbol{B}$ parameters can be obtained in several ways, and most of them make use of a manual (or semi–automatic) segmentation of the observation according to the HMM states. Anyway the details of this process are not in the goal of this work, and can be found in [59].

2.3.2 Effects on under-training

It is unavoidable that the training is always going to be made using training sequences of finite length. This constrain implies that some of the events represented by each model will occur in

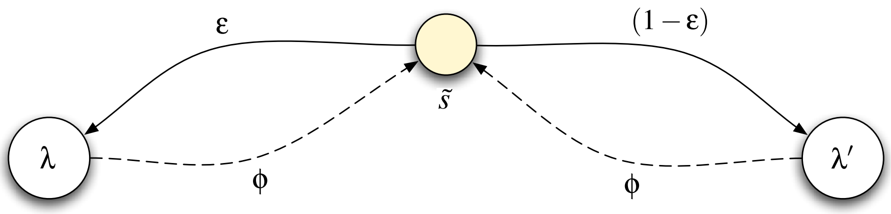


Figure 2.11: Representation of the *deleted interpolation* process as an extended HMM.

the training sequences with different frequencies. In other words the content of the training set is never perfectly balanced among all the parameters to train. In particular for some events, their occurrence in the training data set could be too low to grant a satisfying estimation of the corresponding parameters.

The most straightforward solutions are the increase in dimension of the training set, or the reduction of the number of parameters in the model. Anyway none of them is practical, since finding a greater amount of *good* training data is usually really difficult and time consuming,. On the other hand, changing the model itself can affect its correct behavior because the topology, for example, tend to have some logical connection with the event to be modeled, and the states often represent some specific sub–events that cannot be carelessly removed.

A more practical, and thus widely used, solution is to interpolate the under–trained parameters with the parameters of a reduced version of the model that can be correctly trained with the available training data. The reduced model is generated by tying some parameters to be the same, in a way similar to the one seen when talking about clustering for distribution sharing (see Section 2.2.6). By naming $\lambda = (\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{\Pi})$ the main model and $\lambda' = (\boldsymbol{A}', \boldsymbol{B}', \boldsymbol{\Pi}')$ the reduced model, the interpolation is done using the formula:

$$\tilde{\lambda} = \varepsilon\lambda + (1 - \varepsilon)\lambda' \tag{2.62}$$

The variable $\varepsilon \in [0, 1]$ is the weight of the main model's parameters, while $(1 - \varepsilon)$ is the weight of the reduced model's parameters. A good interpolated model is obtained through the optimization of the ε quantity, and the optimal value depends on the amount of training data available. The extreme case of under-training corresponds to $\varepsilon = 0$, while $\varepsilon = 1$ indicates that there is no need for parameter interpolation.

The problem of the optimization of ε has been solved by Jelinek and Mercer [60]. The solution is obtained by interpreting (2.62) as an extended HMM like the one shown in Figure 2.11, and by applying the forward–backward procedure to it. In the HMM representation the weights represent the transition coefficients from a neutral state $\tilde{s}$ towards the two states representing the original and reduced model. Two null transitions allow for returning into the neutral state. In that way the value ε can be estimated through standard HMM methods. Usually the training data set T is split in two subsets T_1 and T_2 such that $T = T_1 \cup T_2$. The set T_1 is used for training both λ and λ', while the set T_2 is used for the estimation of ε, assuming that λ and λ' are fixed. An evolution of this procedure, known as *deleted interpolation*, repeats the described training procedure with multiple partitions of the training data set.

A further, really common method used to avoid the introduction of errors due to undertraining, is to put constraints on the parameters to be estimated, in order to be sure that certain (upper or lower) thresholds are not exceeded. On a discrete model, for example, it would be possible to apply a constraint of the following form:

$$b_j(k) \geq \delta \tag{2.63}$$

while on a continuous model the same constraint would have the form:

$$\boldsymbol{U}_{jk}(r,r) \geq \delta \tag{2.64}$$

The constraints can be applied by means of a postprocessor that corrects the values exceeding the thresholds after each training iteration, and renormalizes all the other parameters in such a way that no statistical constraint is violated.

2.3.3 The choice of the model

A fundamental step in the implementation of an HMM based system, is the choice of the model topology, of the number of states, the kind of symbols to be emitted (discrete or continuous PDF, single component or mixture, and so on).

Again there is no exact method to make these choices. The most important factor to take into account is, anyway, the nature of the signal to be modeled.

Talking, for example, about speech recognition, the most common models are left–right. That is because the speech signal is represented as a time series and it seems to be logical to associate each HMM state to a certain acoustic event. Furthermore, acoustic events have a natural time evolution that closely matches the behavior of a Bakis model, without transitions towards previous states.

The choice of the kind of PDF instead, is often related to the balance between processing speed and accuracy. Continuous models are definitely more accurate than discrete ones, but they are computationally much heavier than their discrete counterparts. When the system to be implemented is highly complex, sometimes there is no choice but to use discrete models in order to grant a real–time operation.

Chapter 3

Development of a dedicated hardware architecture

3.1 Towards hardware

In the previous chapters, the HMM has been introduced, and it has been shown how this modeling paradigm can be useful to develop pattern recognition systems. It is also well known that most of the systems that rely on pattern recognition are likely to benefit from real–time operation. Some examples are speech recognition, video tracking, identification (through voice, video, and other methods), brain–computer interfaces. Other applications, like DNA decoding, in theory do not need real–time operation, but they usually deal with such huge amounts of data that an increase in computation speed would be really useful in order to shorten processing times.

Both real–time operation and increase in processing speed can be achieved through different techniques. A first approach is to use more powerful devices to perform the computation needed (an example can be found in [61]). This approach is of course the simplest, since it does not need any effort to be spent on the system itself, but is nonetheless the most expensive, both economically and from the point of view of power consumption (really important if the system has to be used in portable devices). A second approach is to optimize the algorithms in such a way that the functionality is maintained but the computational weight and the memory requirements are reduced as much as possible (see [62]). On the other hand these changes on the algorithms have always some effect on the accuracy of the system and, in some cases, such kind of degradation cannot be tolerated.

The method I am going to propose in this work, and that was first introduced in [15], is somewhat a different interpretation of the first of the two approaches just described. The idea is to develop some dedicated hardware that takes care of the most time consuming operations. In that way that computation is speeded up in a more efficient way with respect to what is possible through the use of general purpose computers because the hardware is designed expressly for the specific task, and also because some features are available, parallel processing above all, that cannot be widely and efficiently applied in software. Furthermore the increase in cost can be strongly limited thanks to the availability of programmable devices and soft–processors.

A first step towards the development of an hardware architecture is the analysis of typical problems that are solved with the use of HMMs, in order to identify the parts of the system that are the most computationally costly and, among them, the ones for which a mapping to hardware is possible. This work has already be done for speech-recognition systems in [15, 63, 64, 65]. In this thesis I am going to extend it to a general pattern recognition system and, in practice, to any system that needs to process time series using HMMs.

3.1.1 Choice of an algorithmic framework

Doing pattern recognition on a time series, as we could see in the previous chapters, is basically a matter of correctly splitting the sequence (*segmentation*) and associate a meaning to each of the segments (*classification*). When using HMMs this can be accomplished by finding some kind of building blocks (or *atoms*) in the sequence, and by modeling each of these atoms with a single HMM. In almost any application it is possible to find a finite set of atoms that, through concatenation, can produce any possible input sequence.

A fairly understandable example is that of speech modeling: any language make use of a certain set of sounds to build all the words in its dictionary, and this set usually contains a really limited number of elements (around fifty for most languages, see [66]). By building an HMM for each sound, one can recognize (or produce) any word by just knowing the correct sequence of sounds.

Given a set of models that describe all the atoms needed to build the sequences of interest, a *search space* must be built to allow for the application of the Viterbi algorithm (see page 27 in Section 2.2.2). A search space is actually a trellis like the one shown in Figure 2.7 at Page 26. It is built by connecting together different instances of the atom–models according to some rules that will insure the production of the expected sequences or sub–sequences.

Again this concept can be easily explained using the example of speech modeling. Here HMM chains are built by connecting the sound–models according to a set of rules that specify how to compose, through sequences of atoms, each of the words to be modeled (these rules are usually referred to as *pronunciation* or *spelling dictionary*). The result of this first step is a set of HMM chains, one for each word, each made up of several atom–HMMs. One could obtain a search space directly from this set of multi–atom HMMs by allowing transitions from each of the chain's exit states to all other enter states. An alternative is to create transitions between word–models according to a set of transition probabilities that are consistent with the grammar rules of the language to be modeled.

The process of search space building is illustrated in Figure 3.1. It is clear how the number of states in the trellis can explode due to the fact that the number of multi–atom chains can be very high (in speech, with $N = 5$ states in HMMs and $W = 10000$ words in dictionary, the trellis would contain 50000 states).

The high–level architecture of a generic pattern recognition system based on the standard

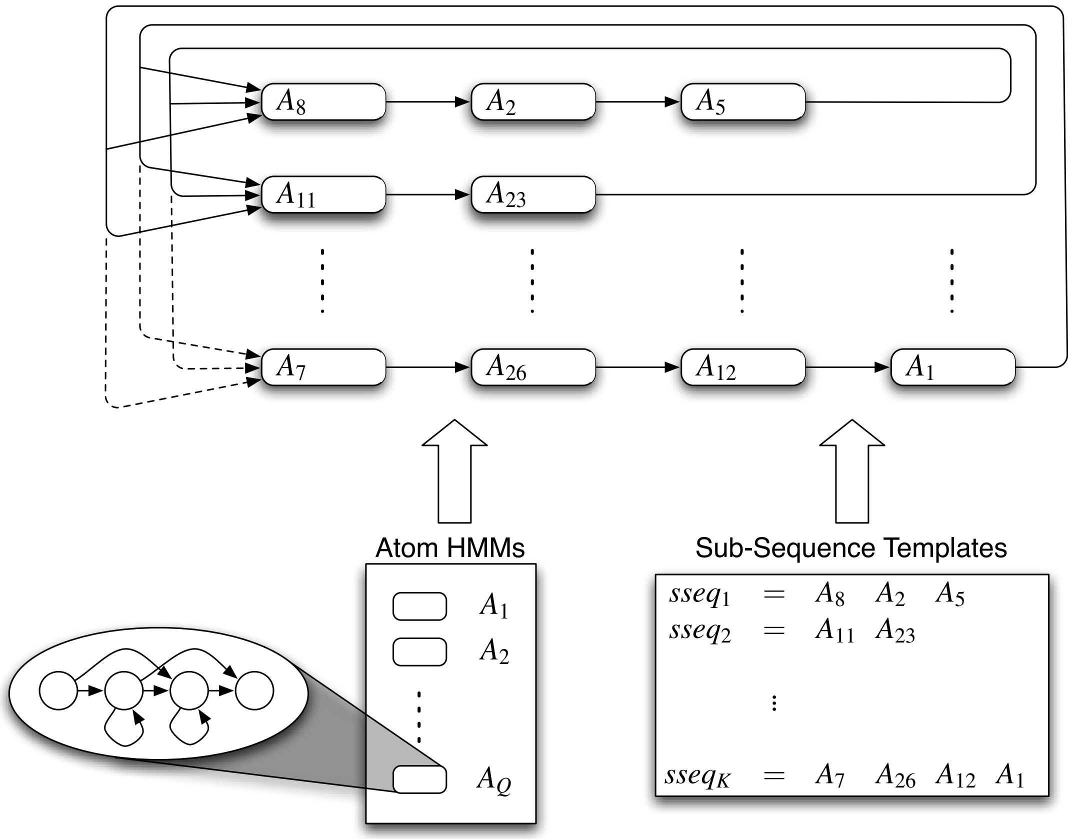


Figure 3.1: Search space building process. A high level representation of the trellis is shown, along with the model–set (*atom–HMMs*) and the HMM chain building rules (*sub–sequence templates*).

processing method just described is shown in Figure 3.2. A signal processing front–end takes care of converting the input signal into a stream of observations suitable for the specific task. The kind of signal processing applied is strongly application–specific. As a general rule, the task of the front–end is to reduce the components of the signal that do not carry information useful for the task (like noise) while enhancing the useful ones. The influence of the application on the design choices to be made for the front–end becomes clear when comparing classification and identification tasks. In classification tasks (like image classification and speech recognition) the aim is to ignore the differences between different realizations of the same event. This means that the front–end should regard as noise any part of the signal that carries informations on the source that produced the event, while enhancing all the informations related to the event itself. In identification tasks (like person identification through voice or image) the situation is actually the exact opposite: since the events are only a carrier for informations related to the source, the front–end will discard the actual content of the signal and enhance the source–specific features. The core of the system is the Viterbi decoder itself. It receives the observations and feeds

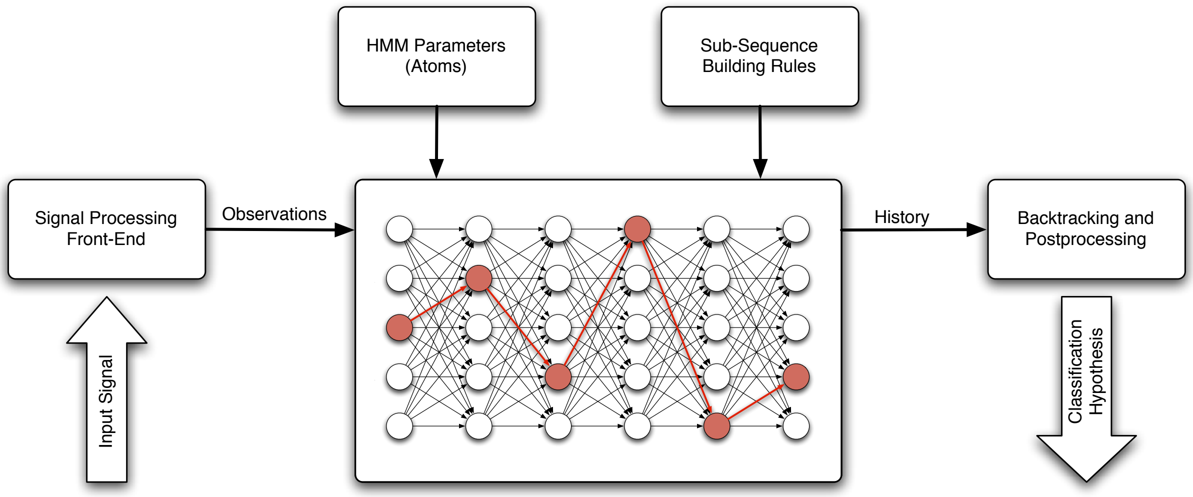

HMM Parameters (Atoms)
Sub-Sequence Building Rules
Signal Processing Front-End
Observations
History
Backtracking and Postprocessing
Input Signal
Classification Hypothesis

Figure 3.2: A standard HMM–based pattern recognition system.

them into the search space, producing a stream of "history" data that contains the scores for all the states at each time step. The history data is then handled by a post–processing unit that does backtracking and eventually applies post processing algorithms in order to produce the classification hypothesis (in the form of a single class, or a sequence of classes, according to the content of the observation sequence).

From the point of view of hardware implementation, a search space built in such a way would need lots of memory and would not easily allow for parallel processing (one should keep in mind that, in principle, each HMM state can be seen as a small processing unit). That is because the application of the standard Viterbi algorithm require all the states in the trellis to be evaluated for each incoming observation, and there is no direct way to split the Viterbi decoder into smaller subsystems. Furthermore all the processing history (i.e. the state probabilities) must be stored to allow for the final backtracking step and by consequence, since the number of states is really high, the memory requirements would grow even more.

These problems can actually be solved by replacing the standard formulation of the Viterbi algorithm with an alternative one that allows for a more functional architecture: the *Token Passing* algorithm.

3.1.2 Splitting it down into pieces: Token Passing

The *Token Passing* is a conceptual model for the processing of connected HMMs, firstly introduced by Young et Al. in [67] for speech recognition systems.

In the Token Passing paradigm the Viterbi algorithm is implemented by reorganizing the decoder into different distinct conceptual levels. Each HMM state S_i is supposed to be able to hold, at each time step t, one or more packets of data called *tokens*. Each token contains an alignment probability $\psi_i(t)$ between the partial observation sequence $\{O_1, \ldots, O_t\}$ and a partial state sequence $\{q_1, \dots, q_t = S_i\}$ that ends in state S_i at time t.

A token can be *passed* from a state S_i to another state S_j if there is a non–zero probability transition between the two states, thus if $a_{ij} \neq 0$. In the *passing* process the token's alignment probability $\psi_i(t)$ is updated according to the value of the transition probability a_{ij}.

Given these basic rules, the formulation of the Viterbi algorithm can be converted into the application of the following steps to each state in each HMM for every incoming observation $\boldsymbol{O}_t$:

1. For all the states S_i from which there is a transition towards the current state s_j, collect the token there contained updating it with the transition probability a_{ij}.

2. Sort the list of collected tokens according to the alignment score, and keep only the N best ones (where the value for N is am application dependent parameter).

3. Update the alignment probability of the retained tokens with the emission probability of the current state $b_j(\boldsymbol{O}_t)$.

Following this interpretation of the decoding process allows to treat the temporary data in the network of HMM states as a flow of tokens through the system, and the system itself as a set of processing elements that perform different tasks. As a matter of fact the system itself is conceptually split into different layers (see Figure 3.3). The lowest layer is the *atom–level* one: it manages the flow (and score update) of tokens between the states belonging to the same atom–HMM chain, according to the input observation sequences and the model parameters. In other words this layer associates segments of the input sequences to the corresponding atomic units that are supposed to make up the signal to be modeled. The second layer, on the other side, controls the flow of tokens between atom-HMMs while the third will control the flow of tokens between groups of logically connected HMMs, and so on. Therefore the upper layers are usually the ones concerned with the application of higher level rules. The number of layers used can actually be arbitrarily high, depending on the complexity of the signal to be modeled.

In speech modeling, for example, the lower layer could model the base speech sounds, the second layer the word chains (made up by sequences of base sounds), and the third layer sequences of words, hence sentences. These three levels actually represent acoustic, lexical and grammatical rules in the production of speech.

In this framework the architecture of the HMM–based pattern recognition system shown in Figure 3.2 at Page 44, can be changed as shown in Figure 3.4. The feature extraction front–end, given the same application, is the same used in the standard implementation, while the search space is now broken into a low level HMM processing block that elaborates incoming data by treating each atom-HMM independently and concurrently according to the Token Passing algorithm. This segmentation of the search space is the main feature of the Token Passing approach that will enable us to introduce parallelism in the hardware architecture. On the other hand, the cross–model transitions are handled by a specific module that controls the flow of the tokens exiting the atom–HMMs back into the models themselves. In particular, tokens exiting the last state of each atom–HMM are passed to the block called *Token Flow Controller* that, by

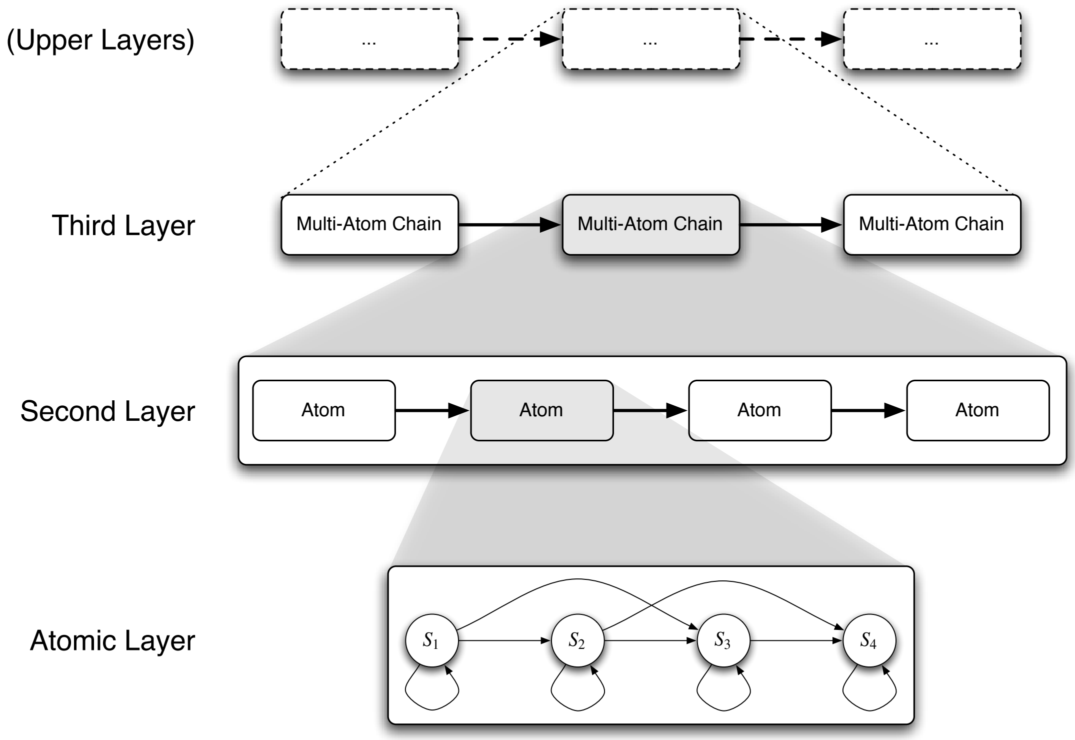


Figure 3.3: Interpretation of the HMM decoding as a layered process through the use of the Token Passing conceptual model.

applying application specific constraints on the possible atom sequences, selects which of the incoming tokens have to be passed back to which atom–HMM, and applies the needed corrections to the tokens' alignment probabilities. Furthermore the Token Flow Controller serves also the task of generating and storing the history as a *lattice* of recognized atom sequences along with the relative start and end time points. From this lattice, via backtracking and/or other post–processing techniques and thanks to higher level rules, the final classification hypothesis is produced.

The most interesting characteristic of this approach is to split the recognition task into several distinct sub–tasks:

The atomic level task: token flow management between the atom-HMMs' internal states.

The *atom grouping* task: token flow management between atomic models belonging to the same logical aggregate, among the ones specified by the relative rule set.

The higher level tasks: token flow management between atom aggregates, according to high level rules.

This subdivision clearly makes the system more versatile and greatly simplifies data handling.

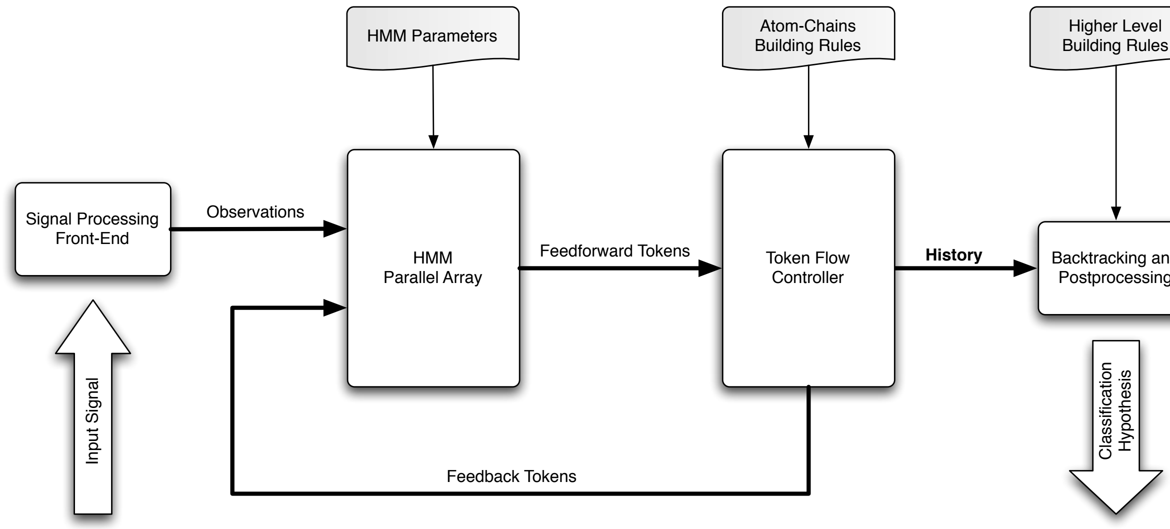

HMM Parameters
Atom-Chains Building Rules
Higher Level Building Rules
Signal Processing Front-End
Observations
HMM Parallel Array
Feedforward Tokens
Token Flow Controller
History
Backtracking and Postprocessing
Input Signal
Classification Hypothesis
Feedback Tokens

Figure 3.4: An HMM–based pattern recognition system that implements the Token Passing conceptual model.

3.1.3 Considerations on hardware implementation

In order to develop an hardware architecture that implements an HMM–based pattern recognition system (or part of it), it is worth analyzing in greater detail the algorithms presented in the previous sections, mainly to quantify the advantages brought by the choice of the Token Passing paradigm instead of the standard approach.

In the search space used in the standard Viterbi algorithm, the system needs to access memories that have to hold the atom–HMMs' parameters, the atom sequences building rules, and the higher level rules. The alignment probability of each HMM state in the search space should be held in another memory that contains also the informations on the topology of the whole network of states. This last structure can turn out to be a significant problem since its dimensions grow exponentially as the number of atom sequence templates to take into account grow. Just to have an idea, using a 3–states HMM with 20000 atom sequences (in speech recognition, for example, it is really easy because atom sequences usually correspond to words), the search space would easily contain 300000 states. Moreover, as already stressed, during the decoding process the temporal evolution oh the search space must be recorded to allow for backtracking and post–processing, so a third memory is needed to store such data in the form of a lattice containing atom sub–sequences.

Such a pattern recognition system implemented in hardware would be characterized by a strictly sequential behavior: the use of a search space built as a single, huge network does not allow for clustering the states into logically connected blocks that could be processed in parallel. This means that it is not possible to exploit the greatest advantage given by hardware implementation over software, that is parallelism. In addition, the memory is completely centralized, and all the accesses are made from the search block. This implies the use of big

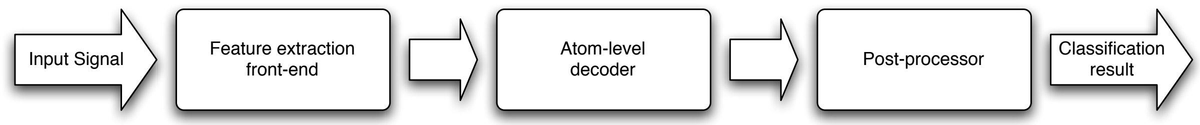


Figure 3.5: The main logical blocks that make up a pattern recognition system's architecture.

memories with consequent great latency in data retrieval. On the other hand for HMM parameters it is possible to reduce the quantity of data to be stored through the use of shared mixtures (see Section 2.2.6, Page 34). Unfortunately the use of shared mixtures does not completely solve the problem of data access latency, since the same issue comes up with the other memories (that hold states' probabilities and topology informations), and also introduces a loss of accuracy in the classifier.

When using instead the Token Passing paradigm, the situation is significantly different. Here the search space is segmented into clusters, each cluster corresponding to a single atom–HMM. From the point of view of hardware implementation, this feature enables us to parallelize the operations to be performed on each of the clusters, thus considerably speeding up the evaluation process. The state clustering also allows to distribute the memory containing the model parameters amongst the separate processing units, thus requiring smaller memories and reducing data access latency. Data access latency can be reduced also for the access to the states' alignment probabilities, essentially because these probabilities are now handled as a data flow (a flow of tokens in our case) and not stored in a central memory.

In the next sections I am going to show in more detail a possible system architecture, its main components and how to implement them.

3.2 System level overview

As it is possible to notice in Figure 3.4, there are several building blocks that need to be implemented in order to obtain a fully functional pattern recognition system. The whole architecture can actually be logically split in three main parts, like shown in Figure 3.5: the front–end, that translates the input signal into features suitable for he decoder, the atom–level decoder, that segments and classifies the input sequences of observations into sequences of atom identifiers, and the post–processor, that applies higher level rules and, using the classification data from the decoder, reconstructs the original content of the input sequence in a comprehensible form.

In this work I am not going to focus on the implementation of the feature extraction front–end, firstly because its content strongly depends on the application, secondly because it performs a set of quite common signal processing tasks that have already been treated in literature (see for example [68] and [69]) and that can also be performed by a suitable DSP. So in the following I will treat the front–end as a black–box that provides the observations stream as required.

3.2.1 From the conceptual model to the architecture

Since the aim of this work is to exploit all the possibilities offered by the hardware implementation, dealing with the possibility to parallelize operations is one of the main points in the definition of an efficient architecture.

As said in Section 3.1, the use of the Token Passing conceptual model allows to deal with each atom–HMM as an independent machine, so that several of these machine can process data in parallel. So implementing a parallel array of HMM processing devices seems the most natural choice. According to the considerations made in Section 3.1.3 about the advantages of having locally distributed memory resources, each processor will have his own memories to store model parameters and temporary data (that is one or more tokens per HMM state).

The HMM processors array will then be controlled by a device that controls the flow of tokens through the system. The token flow control module will be in charge of starting and stopping the activity of the HMM processors, controlling when the front–end provides the observations, collecting tokens output by the HMM array, and processing them in order to choose which ones should be fed back into which of the array elements. The token flow control device also have access to a memory containing the atom chain building rules, and uses them to record the occurrence of any allowed sequence (on the basis of the tokens collected from the HMM array) to allow for higher level post–processing. This kind of records are usually stored in the form of a lattice, each element containing a sub–sequence identifier, start and end times for the event, and eventually pointers to the most likely previous and next events.

Finally the post processing part receives the lattice information and applies the high level rules to extract the most probable sequence. It is important to notice that high level rule sets often take huge amounts of space to be stored and that the post-processing step usually make use of algorithms, like dynamic programming ones (see [70]), that are particularly suitable to be implemented as software, also because the data structures they work on (the lattice records) are described with a high level of abstraction. These last considerations will show up to be really important when choosing the devices that will have to perform each of the tasks described. A representation of the architecture described so far is shown in Figure 3.6.

3.2.2 Defining the hardware platform

The next step is to choose the features of the hardware platform on which the architecture is to be implemented. Taking into account all the considerations made so far, a possible choice could be the following:

- A device capable of sampling the input signal. This depends on the application. An ADC would be needed if the original signal is an analog one. This part will not be treated in this work since can be implemented through standard methods and also because it is highly application specific.
- A device that implements the signal processing front–end. The choice depends on the kind of processing to be applied. The device could be a general purpose or specific pur-

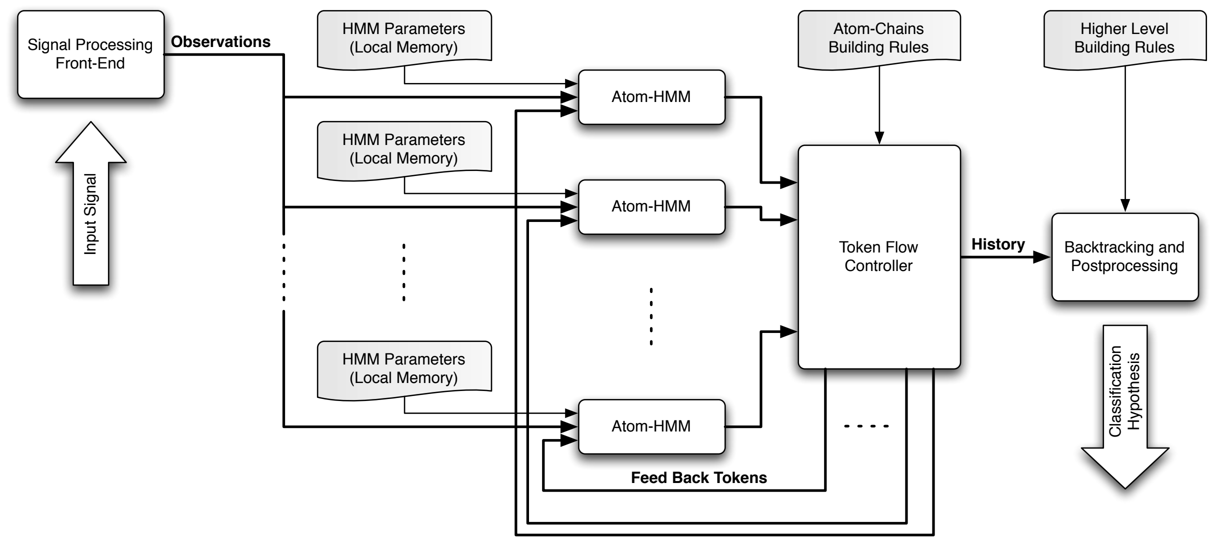

Signal Processing Front-End
Observations
Input Signal
HMM Parameters (Local Memory)
HMM Parameters (Local Memory)
HMM Parameters (Local Memory)
Atom-HMM
Atom-HMM
Atom-HMM
Atom-Chains Building Rules
Token Flow Controller
Feed Back Tokens
History
Higher Level Building Rules
Backtracking and Postprocessing
Classification Hypothesis

Figure 3.6: Representation of a parallel architecture implementing the Token Passing conceptual model.

pose DSP or, in case there is no ready made device able to perform the needed operations, a custom hardware implemented on ASIC or on a PLD (like an FPGA). Also this digital signal processing part is widely present in literature and will not be treated in this thesis.

- A device that implements the atomic–layer of the decoder, that is basically the parallel HMM processor array and the token flow control module (see Figure 3.6). This is the main focus of this thesis. Since it is based on the novel hardware architecture here described, it is going to be implemented as custom hardware, as described in 3.3 (Page 54). In particular the device will be an FPGA because it allows for several versions to be tested and the workflow is fast, flexible. Furthermore many high performance fast–prototyping boards are included in inexpensive development kits available on the market (some examples can be found in [71] and [72]).

- A device that implements the post–processor. Here the kind of processing to be performed is more suitable to be handled through software than through hardware. For that reason the best choice seem to be a microprocessor or a microcontroller. The integration with the custom hardware on the FPGA can be also made easier since several vendors offer, as an IP, some highly configurable soft–processors (see [73] and [74]) that can be mapped directly onto another (or eventually the same) FPGA.

- An interface to retrieve the classification results. This is another standard task. The destination of the recognition data could be a specific device or even a general purpose computer. In the latter case there are many possible alternatives among the standard interfaces, mainly driven by the needed bandwidth: RS232, USB, Ethernet, Wi-Fi, Bluetooth, PCI and so on. No more details are going to be given here on this part.

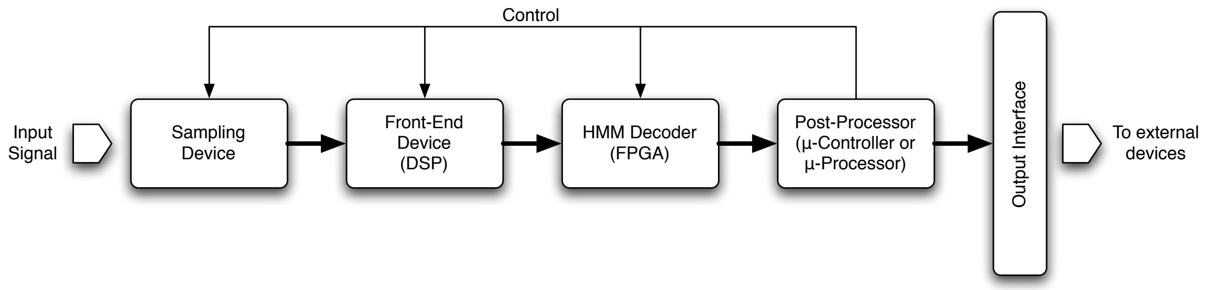


Figure 3.7: A possible hardware platform on which a Token Passing based pattern recognition system can be implemented.

Figure 3.7 illustrates, in a simplified form, a platform like the one just described. Here the microcontroller/microprocessor is used also to provide overall control over the whole system (thus, we can call it a *System Controller*).

3.2.3 Refining the system architecture

The simplest realization of this kind of architecture is shown in Figure 3.8. Here the number of HMM processors is selected on the basis of the number of atom-HMM used in the application. This means that during initialization the System Controller takes care of loading the model parameters from a global memory to the local memories. The main drawback is that, after fixing the number of HMM processors, the architecture would be either oversized, if the application requires few processors, or undersized, if the application requires more processors than available. In the former case there is a waste of processing resources, while in the latter it is not possible at all to run the application. An alternative would be to tailor the architecture to each application by instantiating in each case the required number of HMM processors. This last approach is viable if the system runs on a programmable device, but prevents the implementation on ASIC. Finally, let us consider an application that requires a really high number of HMM processors and a platform with enough resources to map all the processors on hardware: in that case it can easily happen that the input data rate is so slow that the HMM processors remain idle for most of the time between one observation and the following. This, again, wastes processing resources (clock cycles, in this case) that could be used for other purposes.

In practice there is no real solution to the waste of resources that occurs when the number of HMM processors is less than the ones really used by the application. On the other hand some improvement can be applied to the management of the available resources, so that other problems are solved; particularly when there are not enough HMM processors to evaluate concurrently all the atom–HMMs needed by the application, but the input rate is slow enough (with respect to the system's speed) to allow for reusing each HMM processor several times for each incoming observation.

Let us consider a system in which N_H HMM processors are available and an application that makes use of $N_A > N_H$ atom–HMMs. In general it would not be possible to run this

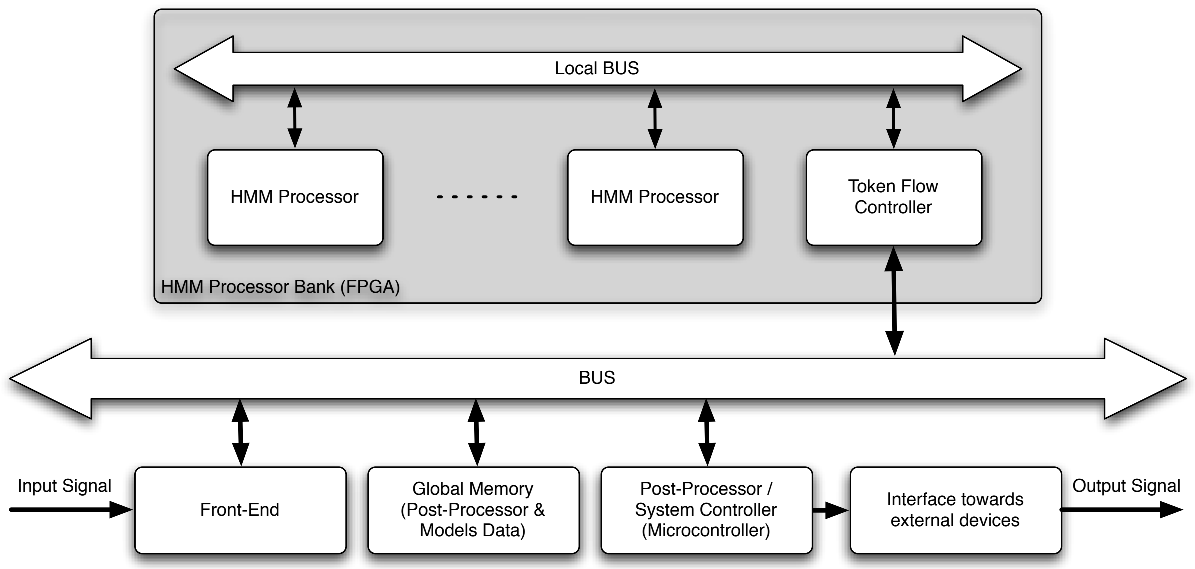

Local BUS
HMM Processor
HMM Processor
Token Flow Controller
HMM Processor Bank (FPGA)
BUS
Input Signal
Front-End
Global Memory (Post-Processor & Models Data)
Post-Processor / System Controller (Microcontroller)
Interface towards external devices
Output Signal

Figure 3.8: System level architecture that use static HMM processors.

application if each HMM processor works with a fixes parameter set loaded in the system initialization phase. Anyway the problem can be solved by using a more complex control over the system, and under certain constraints. At each time step t the system operation would be the following:

1. Store the observation $\boldsymbol{O}_t$ in a buffer.
2. Load the first N_H model parameters from a global memory into the HMM processors' local memories.
3. Run the HMM processors on the currently stored observation $\boldsymbol{O}_t$ (feeding and recollecting the tokens appropriately).
4. Load the next N_H (or $N < N_H$) remaining models and continue from step 3 until all the N_A atom–HMMs are evaluated. Then continue to step 5.
5. Increment time to $t + 1$ and, if there are more observations to process, continue from step 1.

The constraints that have to be met with this type of control are mainly temporal. Without reusing HMM processors, the time needed to process an observation would be T_H, that is the time needed by an HMM processor to evaluate an observation. In this case the only constraint is $T_H \leq T_O$, where T_O is the time between observations. To evaluate the time needed with resource reutilization, instead, let us assume that $N_A = QN_H + R$ with $R < N_H$. The time needed to process a single observation would be the time needed to load all the models into the local HMM processor memories plus the actual processing time, that is clearly $(Q+1)T_H$. With these assumptions, the time constrain for a system enabled to process at each frame more atom–HMMs than HMM processors is:

$$(Q+1)T_H + N_A T_L < T_O \tag{3.1}$$

Even if it is not straightforward to verify if this constraint is met, it is clear that in some common HMM–based application the solution would be viable. In classification of biological and speech signals, for example, the observation frequency seldom reaches more than a few hundred Hertz. Nonetheless, most FPGA–based fast prototyping boards can easily work with clock frequencies of 50 MHz (modern FPGAs themselves can reach 400 MHz and more). This means having a minimum *cycle–budget* of 10^4 to 10^5 clock cycles, that should be enough to meet the constrain in (3.1) at least for $Q = 1$.

From the architectural point of view some changes should be introduced in order to allow for HMM–processor reutilization (called *dynamic HMM-processor allocation* from now on). First of all, observations should be stored for all the duration of the multi–pass process. This is usually not a problem since it can be easily implemented as an output buffer in the front–end or an input buffer in the devices receiving the data. Since the model parameters are now loaded into the HMM processors' local memories several times for each time step (instead of only one during initialization), a local memory is added into the HMM decoder to speed up the process. This memory is loaded with *all* the models during initialization and during run time, the HMM processors are loaded from that memory, reducing off–chip data transfers. Finally the data is now transferred by means of two buses: one used for the observations and (only during system start–up) for loading the decoder's local memory with the model parameters, and the other for the token traffic. This last change is not strictly necessary for the proper working of the dynamic HMM processor allocation, but increases the throughput between the decoder and the rest of the system, thus speeding up the computation and allowing for more processing passes to be done for each observation. The new system architecture is shown in Figure 3.9.

3.2.4 Keeping track of tokens

Another issue to deal with when trying to implement an HMM–based pattern recognition system onto a hardware platform, is how to keep track of tokens.

At system level, the event of an HMM emitting a token can be interpreted as the recognition of the class modeled by that HMM, with a likelihood given by the token's alignment probability. The first time this happens, the system level controller creates and stores a record containing an ID for the HMM and the time step in which the event occurred. The emitted token will be updated with a back–pointer to the record. The next time the same token will be emitted from an HMM, another record will be created and back–linked to the previously created one. The token, in turn, will be updated with the back–pointer to the last record created. This means that each token is actually the tail of a back–linked chain of class records. This concept is illustrated in Figure 3.10. The main problem to face when using this approach

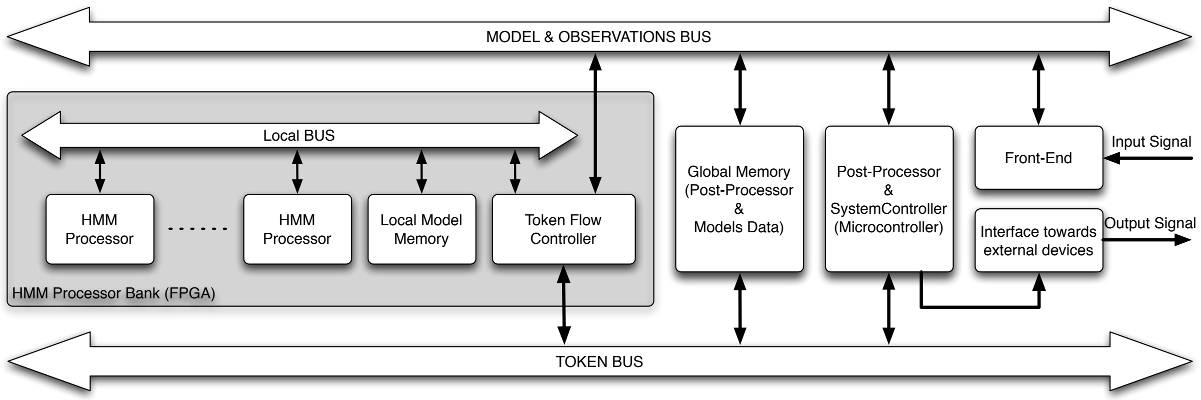

MODEL & OBSERVATIONS BUS
Local BUS
HMM Processor
HMM Processor
Local Model Memory
Token Flow Controller
HMM Processor Bank (FPGA)
Global Memory (Post-Processor & Models Data)
Post-Processor & SystemController (Microcontroller)
Front-End
Input Signal
Interface towards external devices
Output Signal
TOKEN BUS

Figure 3.9: System level architecture that use dynamic allocation of HMM processors (control signals are omitted).

is that, with a high number of HMMs, the storage space needed by all the records produced is likely to explode in few time steps. The common solution to this is to limit the number of active tokens by keeping only the K ones with highest alignment probability (*K–best*). On the other hand, since each state can hold, in real world applications, a limited number of tokens, some pruning mechanism must be implemented in order to keep only the most likely in each state. The main consequence of pruning is that some tokens entering an HMM may never come out of it because they are discarded. In these cases the corresponding record chains become useless and should be removed to free up precious space. All these considerations imply the necessity of implementing inside each HMM processor a mechanism of notification for pruned tokens. Details on this will be given in Section 3.3 along with a more comprehensive description of the HMM processor and the architectural solutions found to implement it.

3.3 Going deeper: the HMM processor

The HMM processor is actually the core element of the dedicated architecture presented in this work. As already said, it is in charge of performing the computations and control tasks needed to apply the token passing algorithm on each atom–HMM's internal state. The operations to be implemented are basically the computation of emission probabilities, the token passing between states (including the update of scores with transition probabilities), the overall control, and the communication with the other devices in the architecture (see Figure 3.9, Page 54).

To define the HMM processor architecture, some assumptions have been made on the way the HMMs are treated. First of all, to any N–state HMM, two more non–emitting states (here called *dummy states*) are added: an entry state and an exit state. These two states work like input and output buffers. Tokens entering the HMM are stored in the entry state and tokens being emitted from the HMM are stored in the exit state. Passing through one of these *dummy* states, do not make the observation time advance. This means that passing a token from the

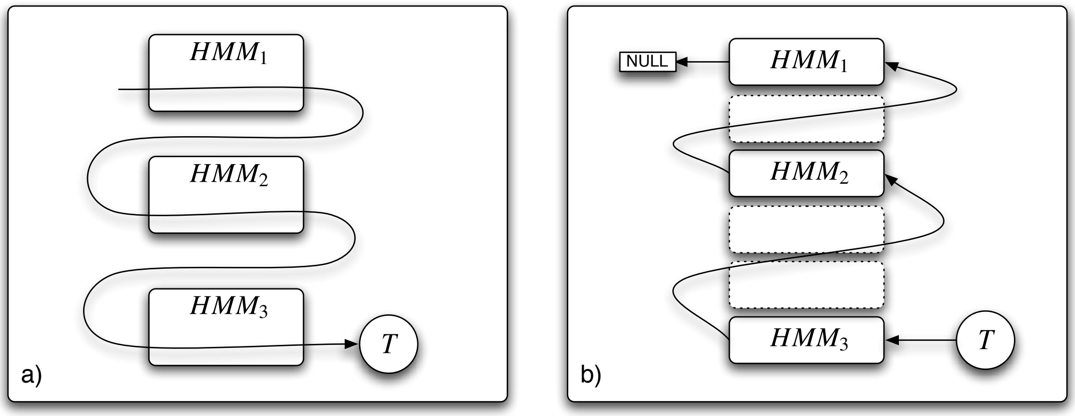


Figure 3.10: Representation of a typical chain of records created by a token being passed through the state space. *a)* shows the possible trajectory of a token in the state space; *b)* shows the corresponding chain of records.

exit state of an atom–HMM H_1 to the entry state of another atom–HMM H_2, produces the exact behavior obtained by directly connecting with a transition the last *emitting* state of H_1 to the first *emitting* state of H_2. This concept is better explained in Figure 3.11. It is going to be clearer in 3.4.1 (Page 58), how the use of dummy states really simplifies the token passing process and the management of cross–model transitions.

Furthermore, I am going to show how parallel processing can be exploited also within the single HMM processor. This can be done by mirroring the memory in which HMM–internal tokens are stored, so that the first half is used for reading data, and the second for storing results. In that way, both token passing and emission probability computations can be done concurrently for each emitting state in the HMM.

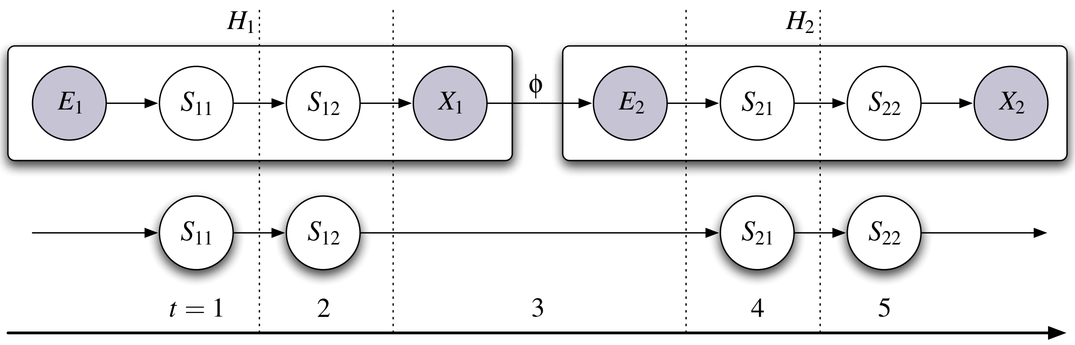


Figure 3.11: Illustration of the behavior of dummy states in HMMs. The state chain in the first row is equivalent to the one in the second row. Observation time is shown in the lower part.

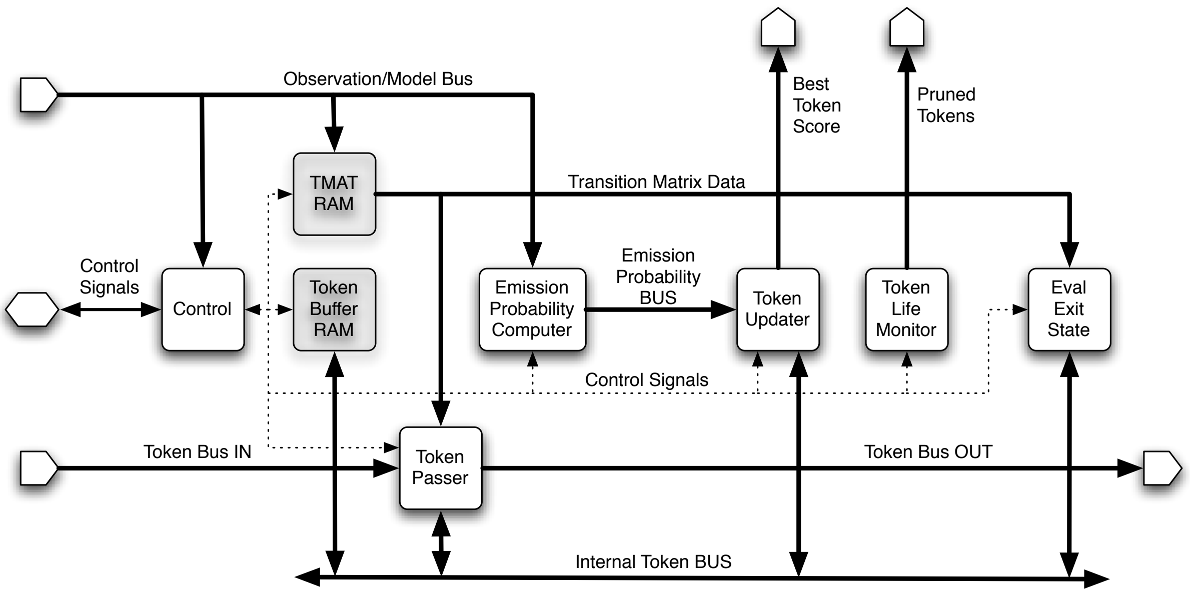


Figure 3.12: Architecture of the HMM processor.

3.3.1 Overview on the HMM processor architecture

An overall view of the architecture developed for the HMM processor is shown in Figure 3.12. Of course it is a simplified representation, and there is no much detail on how data and control signals are distributed among the various components of the subsystem (for details see Section 3.4), but it shows all the building blocks and their relationship.

All the architecture is managed by a dedicated control unit that provides also an interface for external control. The FSM implemented by the HMM processor control unit is shown in Figure 3.13.

After the reset signal, the HMM processor goes to *Wait for Model* state. In this state the system is not able to process anything because there are no model parameters available in the internal memory.

In order simplify the system level control, when instantiating an array of HMM processors, an unique identification number (*HMMID*) is assigned to each instance. This allows to easily address a specific device. As an example, when the proper ID number is received, the HMM control changes its state to *Load Model*, and the model parameters are loaded locally from the model bus (that will be used during processing also to receive the observations). Model parameters are split and stored in two different locations inside the HMM processor's architecture: the transition matrix is held in a memory accessible by the devices that perform token passing tasks (*TMAT RAM*), while the gaussian mixtures are held in a memory internal to the device that computes the emission probability (*Emission Probability Computer*).

After model parameters are correctly loaded, the system switches to *Wait Command* state,

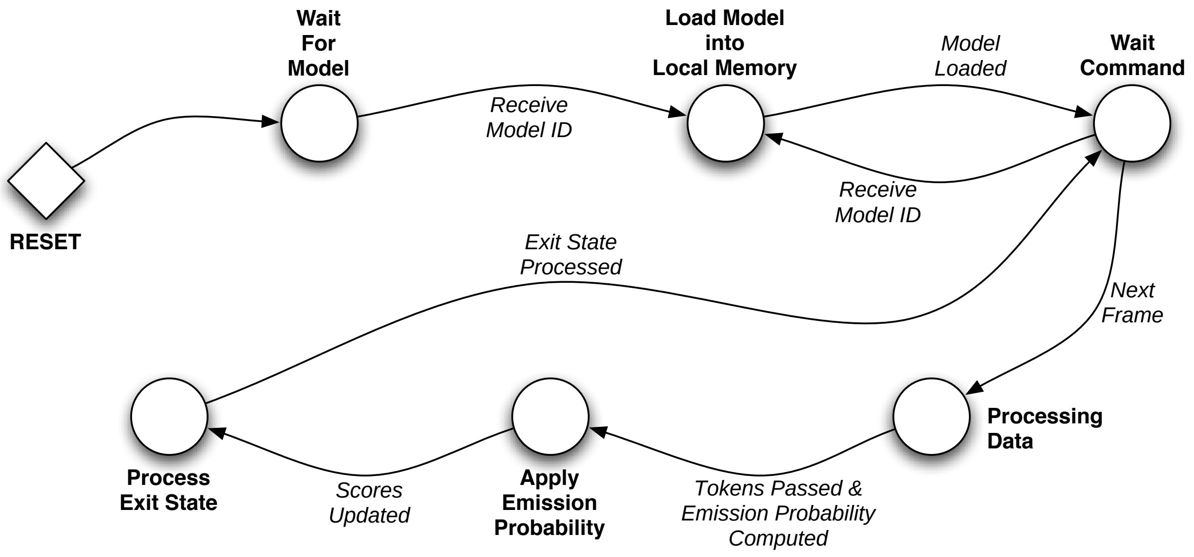


Figure 3.13: Finite State Machine that implements the overall control for the HMM processor architecture.

and is actually ready for processing. In *Wait Command* state, two options are available: beginning processing and loading a new model. The latter option allows to load the HMM processor with a new set of parameters during run time, thus allowing for reutilization of a single processing device for multiple logical atom–HMMs.

The actual HMM processing is split in three distinct stages: main processing stage, update of the token scores with emission probabilities, and selection of emitted tokens.

The first state of the process is called *Processing Data*. In this state two devices in the architecture are activated in parallel: the *Token Passer* and the *Emission Probability Computer*. The Token Passer collects tokens sent to the HMM from the external token flow control unit, and applies the token passing algorithm for each emitting state. Tokens are collected and stored using a specific memory (the *Token Buffer RAM*) containing two twin banks, each capable of storing one token per HMM state (including the dummy states). [1] The choice of the twin banks for the tokens' storage is due to the fact that one bank is needed to read tokens (from the algorithmic point of view it contains the tokens computed in the previous time step) and the other bank to store results during the processing stages. At the end of each time step the second bank is dumped into the first one [2] so that the process can be repeated in the following time step.

[1]A possible extension to the computation model used in this work is to allow for holding multiple tokens in each state. This choice can produce an increase in classification accuracy but requires a more complex control, more hardware processing resources and much more bandwidth for the data flowing through the system. Since the aim here is to introduce a basic architecture, variations like this one are not going to be treated in this work.

[2]Again, a possible improvement would be to use a *ping–pong* strategy with the two memory banks, so to avoid dumping the second bank into the first one at each time step, and thus speeding up the process. This solution can be applied without much effort, but I am not using it in this work for the sake of clarity. It will surely be implemented in future architecture tuning stages.

Concurrently with the Token Passer, the Emission Probability Computer is activated. This last module loads from outside, through the observation bus, the features for the time step to be processed, and computes the emission probabilities for each emitting state.

When both the Token Passer and the Emission Probability Computer are done, the system goes in *Apply Emission Probability* state. In this state, the *Token Updater* module gets the computed emission probabilities from the Emission Probability Computer, and updates the tokens stored in the second bank of the token buffer memory. The Token Updater module also keeps track of the best token's score in the HMM and make it available on an output port. Best tokens are used at system level control to tune the thresholds for pruning.

The last processing stage, after the scores are updated, is the *Process Exit State*, during which the *Eval Exit State* module is activated. This module collects tokens from all the connected emitting states, updates their scores with the appropriate transition probabilities, and selects the one with highest probability as the one to be emitted from the HMM at the current time step.

When also this step is completed, the system returns in the *Wait Command* state, ready for the next time step.

Along with the modules that perform the actual data processing, also the *Token Life Monitor* module is active. It is in charge of providing to the system level controller informations on the pruned tokens, so that the records in the upper processing layers can be correctly updated. The Token Life Monitor is basically a counter array that keeps track concurrently of new tokens fed into the HMM, token replication due to multi–destination passing, and token pruning. Details on its internal architecture can be found in 3.4.3 (Page 64).

3.4 Details of the architecture's components

Some of the architecture components described in the previous section have undoubtedly a complex behavior. For this reason in this section a deeper analysis of these components is made, with the aim to clarify the actual implementation that will be presented in Chapter 4.

3.4.1 The Token Passer

The structure of the *Token Passer* module is shown in Figure 3.14.a, while the FSM that controls its behavior is shown in Figure 3.14.b.

The *Get Incoming Tokens* state works also as a wait state when the module is idle. When an activation signal is received from the external HMM processor controller, the *Token Collector* sub–module takes care of reading the tokens fed into the HMM's entry state from *outside* the HMM processor, and write them in the Token Buffer memory.

Then the module goes into the *Pass Tokens and Computer Score* state. In this state the *Score Computer* sub–module applies token passing on the HMM's internal states: tokens from time step $t-1$ (stored in the first of the twin banks of the token buffer memory) are passed according to the transition probabilities, and stored in the second bank of the token buffer and their score

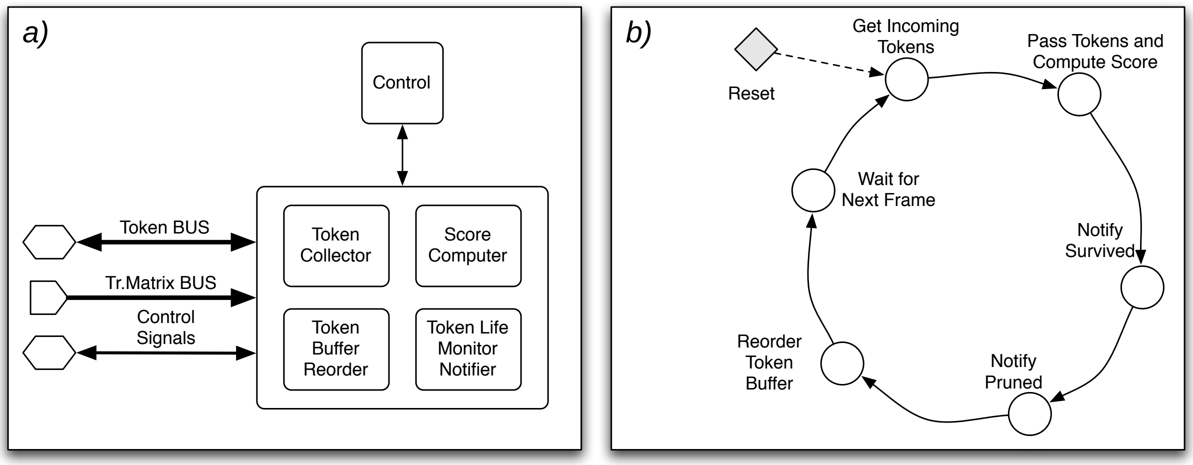


Figure 3.14: Architecture of the *Token Passer* module: *a)* shows the building blocks; *b)* shows the FSM that controls the processing.

updated with the corresponding transition probability. For an N–states HMM, the token buffer is capable of holding a list of at most N tokens per state (corresponding to the limit case of a state that has transition coming from all the others HMM states). After the collection and update, the token with the highest alignment probability (also referred to as *best token*) is selected. This will be the only token to survive until next time step. [3]

When the token passing is done, the state changes to *Notify Survived* and then to *Notify Pruned*. In these states the *Token Life Monitor Notifier* sub–module is activated, and signals are sent to the *Token Life Monitor* module (see Page 64) to notify which new tokens have been kept in the HMM, which existing tokens have survived and, eventually, multiplied (through passing to multiple states), and which tokens have been pruned.

After notification tasks have been performed, the *Token Buffer Reorder* module writes the selected best token for each state to reserved locations into the *Token Buffer* memory. After that, the state changes to *Wait for Next Frame* until the system passes to next time step, and the cycle begins again from state *Get Incoming Tokens*.

3.4.2 The Emission Probability Computer

The *Emission Probabilty Computer* is likely to be the most articulated module in the HMM processor architecture. Its architecture is shown in Figure 3.15. It can be noticed that, compared to the other modules, it is the only one that takes full advantage of parallel processing, since the emission probabilities for all the emitting states in the HMM are computed concurrently by an array of independent devices.

[3]This because I am using a one–token–per–state approach. In multiple–tokens–per–state approaches, a predefined number M of tokens per state is held, and the token list must be sorted according to likelihood, so that in the next time steps it will be easier to keep only the highest probability tokens.

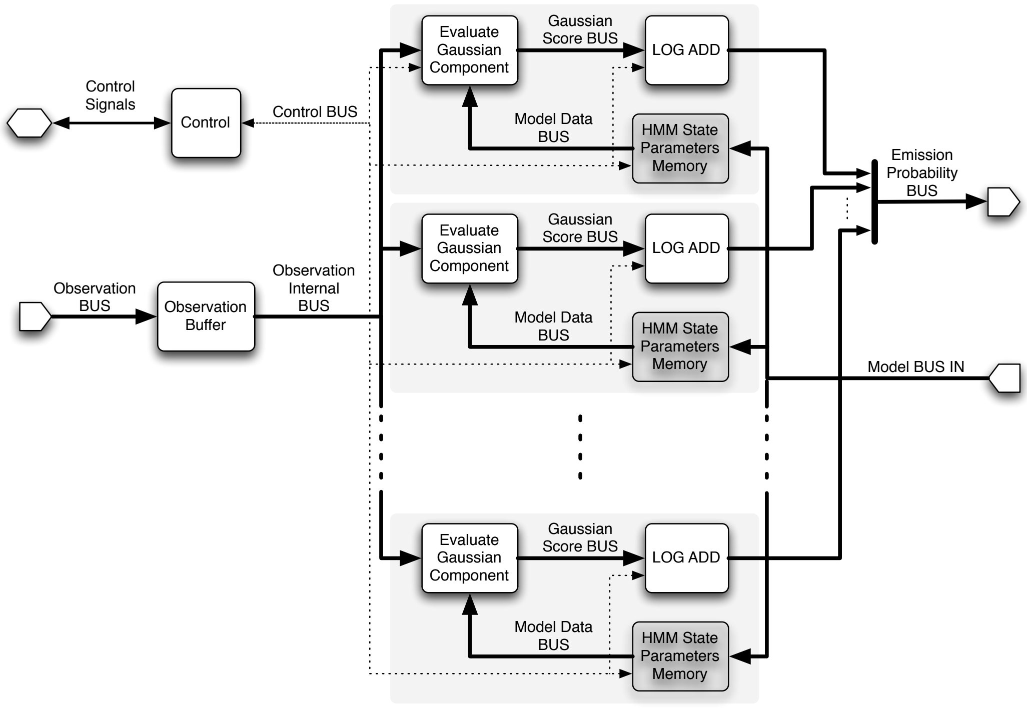


Figure 3.15: Architecture of the *Emission Probability Computer* module.

Since the processing of each state's probability is not necessarily synchronous, the observation for the current time step is loaded, from the external observation bus directly connected to the front–end (see Figure 3.9), in a local buffer.

In the architecture there is a number of emission probability computing blocks equal (at least) to the number of emitting states in the HMM. Each processing block loads, into its local memory, the needed model parameters for the corresponding state from the model bus during the HMM processor's model loading phase, or during runtime model re–loading (see Section 3.3.1, Page 56).

In order to understand how a single emission probability processing block works, it is necessary to give some more detail on how scores are treated in the proposed architecture. So, let us consider the operations performed on the alignment scores during a token passing session. I showed in Section 2.2.2 how most of the processing is due to the evaluation of the recursion step (equation 2.30, Page 27) in the Viterbi algorithm. Thus, on the score of each token, passing through a state S_j, the following operations are applied for every t in the token's lifetime:

$$\delta_t(j) = \max_{1\leq j\leq N}\left[\delta_{t-1}(i)\cdot a_{ij}\right]\cdot b_j(\boldsymbol{O}_t) \tag{3.2}$$

In can be easily seen that the computation of this value requires $N+1$ multiplication, plus the operations needed to compute the emission probability $b_j(\boldsymbol{O}_t)$. As already showed in 2.2.5, the computation of the emission probability is done according to (2.48, Page 31), thus:

$$\begin{aligned} b_j(\boldsymbol{O}_t) &= \sum_{m=1}^{M} c_{jm} \cdot \mathcal{N}\left[\boldsymbol{O}_t, \boldsymbol{\mu}_{jm}, \boldsymbol{U}_{jm}\right] \\ &= \sum_{m=1}^{M} c_{jm} \cdot \left[(2\pi)^{-N/2} |\boldsymbol{U}_{jm}|^{-1/2}\right] \cdot e^{-\frac{1}{2}(\boldsymbol{O}_t - \boldsymbol{\mu}_{jm})\boldsymbol{U}_{jm}^{-1}(\boldsymbol{O}_t - \boldsymbol{\mu}_{jm})'} \end{aligned} \tag{3.3}$$

Assuming that the factors $(2\pi)^{-N/2}|\boldsymbol{U}_{jm}|^{-1/2}$ can be pre–computed and stored for a given set of model parameters, that the covariance matrix is directly stored in its inverse form, and that the gaussians components are L–dimensional, the computation of an emission probability requires $2L - 1$ sums and $2L^2$ multiplications to obtain the exponent, plus two more multiplications and the evaluation of the exponential in order to obtain the term of the sum. In synthesis the number of operations needed are $M(2L-1) + M - 1 \propto 2LM$ for the sums and to $2M(L^2+1) \propto 2ML^2$ for the multiplications, plus the evaluation of M exponentials.

The number of operations needed is definitely high since it is quite common, in particular when using also the derivatives of the feature coefficients, to work with high values of L (for speech recognition it is common to have $L = 36$ and $M = 4$, so that the number of multiplications gets easily over 10^4). Furthermore, one should remember that the computations are made on *probabilities* that, by definition, are always less than unity. The relation (3.2) implies a large number of multiplications between values of that kind and, clearly, the product of a large number of factors less than one leads to problems of underflow when using fixed–point values and a finite number of bits for the representation, as is our case.

Fortunately all these issues have a solution. First of all it is possible to simplify the computations themselves by assuming that the covariance matrices $\boldsymbol{U}_{jm}$ used to describe the gaussian components are all *diagonal*, thus implying the independentness of the observation coefficients belonging to the same vector $\boldsymbol{O}_t$. This assumption does not introduce sensible changes in the system's accuracy because the independence is a true condition in most of the cases[4], but nonetheless it greatly reduces the number of multiplications needed. The introduction of diagonal covariance matrices changes the (3.3) in the following way:

$$\begin{aligned} b_j(\boldsymbol{O_t}) &= \sum_{m=1}^{M} \left[c_{jm} \cdot \mathcal{N}\left(\boldsymbol{O}, \boldsymbol{\mu}_{jm}, \boldsymbol{U}_{jm}\right)\right] \\ &= \sum_{m=1}^{M} \left[c_{jm} \cdot \left((2\pi)^{-N/2} \prod_{i=1}^{L} \sigma_{jmi}^{-1}\right) \cdot \exp\left(-\frac{1}{2}\sum_{i=1}^{L}(O_{ti} - \mu_{jmi})^2 \sigma_{jmi}^{-2}\right)\right] \end{aligned} \tag{3.4}$$

Here the subscript i is used to indicate the single elements of vectors, like O_i. In the case of the diagonal covariance matrix, we know that the elements on the diagonal U_{ii} represent the variance of the multidimensional gaussian along the dimension i, so they are indicated with σ^2.

[4] In fact an accurately designed front–end provides a set of parameters as much independent as possible (maximizing the amount of informations carried by each parameter), mainly because reducing the number of parameters needed to model the signal also reduces the complexity of the classifier's models.

It is straightforward to verify that to evaluate the relation (3.4) are necessary $M(L+1)-1 \propto ML$ sums, $2M(L+1) \propto 2ML$ multiplications and the evaluation of M exponentials, with a neat advantage with respect to the evaluation of (3.3).

Now it is still necessary to solve the problem of underflow. This issue can be dealt with by just changing the representation of numbers from linear to logarithmic. By using this technique the dynamic range of values is highly compressed, so that the risk of underflow becomes minimal. Furthermore all the products are transformed in sums, thus dramatically reducing the resources needed to implement the operations in hardware. In logarithmic scale the relation (3.4) becomes:

$$\begin{aligned} \ln[b_j(\boldsymbol{O}_t)] &= \ln\left\{\sum_{m=1}^{M}\left[c_{jm}\cdot\left((2\pi)^{-N/2}\prod_{i=1}^{L}\sigma_{jmi}^{-1}\right)\cdot\exp\left(-\frac{1}{2}\sum_{i=1}^{L}(O_{ti}-\mu_{jmi})^2\sigma_{jmi}^{-2}\right)\right]\right\} \\ &= \underset{1\le m\le M}{\text{logadd}}\left[\ln(c_{jm})-\frac{N}{2}\ln(2\pi)-\sum_{i=1}^{L}ln(\sigma_{jmi})-\frac{1}{2}\sum_{i=1}^{L}(O_{ti}-\mu_{jmi})^2\sigma_{jmi}^{-2}\right] \qquad (3.5)\end{aligned}$$

In the last equation the notation *logadd* is used to indicate the operation that in literature [75, 76] is usually called *soft–combining* or *soft–add*, and that can be defined for a generic set of K variables x_k through the following relation:

$$\ln\left(\sum_{k=1}^{K}x_k\right) = \underset{1\le k\le K}{\text{logadd}}\left[\ln(x_k)\right] \qquad (3.6)$$

The main advantage in making use of the *logadd* operation, is that it can be easily estimated thanks to an approximation known as *Jacobian logarithm*, widely used in coding and decoding algorithms for telecommunications. With this approximation, the *logadd* operation can be computed in the following way:

$$\begin{aligned} \text{logadd}\left[\ln(x),\ln(y)\right] &\cong \max\left[\ln(x),\ln(y)\right]+\ln\left[1+e^{-|\ln(x)-\ln(y)|}\right] \\ &\hat{=} \max\left[\ln(x),\ln(y)\right]+f_{MAP}\left(|\ln(x)-\ln(y)|\right) \qquad (3.7)\end{aligned}$$

How to deal with the *logadd* operation depends on the target application: in some cases it is enough to just ignore the f_{MAP} term and replace the operation with a simple maximum selection; in other cases, when the number of bits used for the representation is not too high, the values of the f_{MAP} can be stored in a look–up table; finally, if none of the previous solutions is viable, the computation of the f_{MAP} can be done through the use of well known techniques for the evaluation of the exponential and of the logarithm (like the CORDIC algorithm [77, 78]).

When using the logarithmic scale, some of the model parameters (namely c_{jm} and a_{ij}) are stored directly in logarithmic scale. Also the constants are pre–calculated when possible: in this case, for the equation (3.7), a constant G_{jm} is assumed to be stored along with the model parameters, and it is defined as:

$$G_{jm} = -(N/2)\ln(2\pi)-\sum_{i=1}^{L}\ln(\sigma_{jmi}) \qquad (3.8)$$

Method	**Sums**	**Multiplications**	**Other**
Original	$2ML$	$2ML^2$	M exponentials
DCM	ML	$2ML$	M exponentials
DCM & LS	ML	$2ML$	$(M-1)$ logadds

Table 3.1: Computation of emission probabilities: comparison of the number of operations needed in function of the method used (DCM stands for *diagonal covariance matrices* and LS for *logarithmic scale*).

From now on, when talking about emission probability calculation, the variables $b_j(\boldsymbol{O}_t, c_{jm}$ and a_{ij} are assumed to be in logarithmic scale while the others in linear scale[5]. Moreover I assume to store the inverse variances ($ivar_{jmi} = \sigma_{jmi}^{-2}$) in order to replace divisions with multiplications. The relation (3.5) will be thus written in the following way:

$$b_j(\boldsymbol{O}_t) = \underset{1 \le m \le M}{\text{logadd}} \left[G_{jm} + c_{jm} - \frac{1}{2} \sum_{i=1}^{L} (O_{ti} - \mu_{jmi})^2 ivar_{jmi} \right] \tag{3.9}$$

It is straightforward to see that this implementation needs $M(L+2) \propto ML$ sums/subtractions, $2LM$ multiplications and $M-1$ *logadd* operations. Also the $N+1$ multiplications needed to evaluate the alignment probability in (3.2), become $N+1$ sums with the introduction of the logarithmic scale.

In Table 3.1 is shown a comparison of the number of operations needed by the three evaluation methods for $b_j(\boldsymbol{O}_t)$ (original, with diagonal covariance matrices, with diagonal covariance matrices and logarithmic scale). It can be seen that a great reduction in the number of multiplications is obtained by introducing the diagonal covariance matrices. On the other hand the method that uses logarithmic requires more operations (the *logadd* operation requires the computation of an exponential *and* a logarithm, even if in some circumstances it can be avoided). Nonetheless, using logarithmic scale cannot be avoided because of the already discussed numerical representation problems.

So, referring again to Figure 3.15 (Page 60), each of the processing blocks contains a couple of specific sub–modules: the first one (*Evaluate Gaussian Component*) computes in turn each of the M arguments of the *logadd* function as they are written in (3.9), and sends them to the second block (*LOG ADD*) that applies the *logadd* function (or whichever function has been chosen to replace it). The two blocks work in pipeline: when the first result is passed to the *LOG ADD* block, it is simply stored as the last result. For all the following values, the operation is applied taking as arguments the incoming value and the stored one. The result is then stored replacing the old one, and the cycle repeats until all the M terms have been sent. At this point the value for $b_j(\boldsymbol{O}_t)$ is ready and kept until requested from outside the Emission Probability Computer.

[5] Note that there will be no change of notation. So, for example, the value $\ln[b_j(\boldsymbol{O}_t)]$ will be written just as $b_j(\boldsymbol{O}_t)$ and so on for all the variables in logarithmic scale. This is done to avoid unpleasantly cluttered formulas.

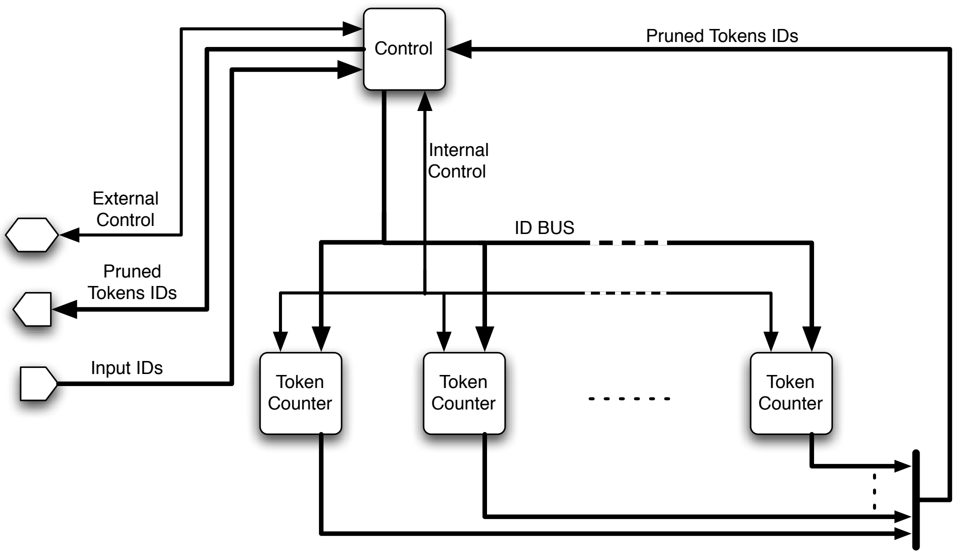


Figure 3.16: Architecture of the *Token Life Monitor* module.

3.4.3 The Token Life Monitor

The *Token Life Monitor* is an essential component for system–level control. Its purpose is to notify to the system controller if some tokens that are supposed to be inside the HMM have been pruned or not. In other word this module allows the system controller to know if it has to wait for certain tokens to be emitted from the HMM, or if it can discard all the tracking information related to them, and free up some precious storage resources. The architecture of this module is shown in Figure 3.16.

As a matter of fact the Token Life Monitor is just a bank of counters with an application specific controller. During the processing stages the *Token Passer* module sends through the *Input IDs* bus, some numbers that identify univocally a certain token traveling through the system. Along with the IDs, some control signal specify if the token has just been fed into the HMM, if it was pruned or if it was duplicated. When a new token is fed into the HMM, a new counter is activated for it, and that counter keep track of the number of instances of that token inside the HMM until there are no more inside the HMM, because of pruning or emission. At that point the counter is free and can be used for other new tokens. When a counter is freed, the corresponding token ID is notified as *pruned*. Of course this happens also when a token is emitted from the HMM and there are no more instances left: in that case is up to the system controller to realize that the notification of pruning has happened concurrently with the token emission.

There are some factors that require particular attention when designing this module. First of all the number of counters should be enough to track all the different tokens that can be present in the HMM at the same time. When holding a single token in each state this number is just the number of HMM states, but when keeping a list of tokens in each state the problem must be handled carefully, in order to avoid "losing" incoming tokens with high probability (because there are no free counters to track them). Also the number of bits in each counter must be tuned according to the expected lifetime of a token inside the HMM. Another important issue is how to obtain unambiguous IDs for *every* token traveling in the system. Also this can be done through the estimation of the tokens' lifetime: by using a progressive counter that flips over a period longer than that lifetime, there is never the risk of having two tokens with the same ID.

Now that the whole architecture have been described thoroughly, the actual implementation of the system, along with the methodologies used, can be described. Chapter 4 gives all the details on that topic.

Chapter 4

Implementation and testing of the architecture

In this chapter I will show the approach used and the results obtained for the actual implementation in hardware of the architecture described in Chapter 3. First of all, the overall design flow will be described along with the tools and techniques used. Then the implementation details will be discussed and, finally, the results of the tests made on the implemented design will be shown.

4.1 A design flow for the architecture

In Section 3.2 I explained how the proposed architecture can be mapped on a platform containing one or more FPGAs for the custom hardware, a front–end for the feature extraction, and a device (most likely a microcontroller or a soft–processor) capable of performing the tasks of a system level controller plus the high level pattern recognition operations (see Figure 3.7, Page 51).

Of course it could be possible to design the whole hardware platform but, at this early stage of development, using a fast prototyping platform provides the same level of effectiveness, the possibility to experiment several solutions, and a highly reduced cost. In Section 3.2.2 I already mentioned that on the market there are many high performance development kits that can be suitable for the goals of this work.

A typical design flow for these kind of fast–prototyping platforms is shown in Figure 4.1. First of all, the custom hardware present in the system is described at RTL level using a chosen *hardware description language* (depending on the one supported by the design tools used). Then it is simulated in order to check if the behavior is the one expected. To simulate the described subsystem (referred to, in this context, as *Unit Under Test*, or *UUT*) a testbench is needed to provide the proper stimuli. The testbench can be described at a behavioral or also RTL level (sometimes, for testing purposes in the later stages, it can be useful to have a sythesizable testbench). Depending on the RTL simulation results, some modifications could be needed on

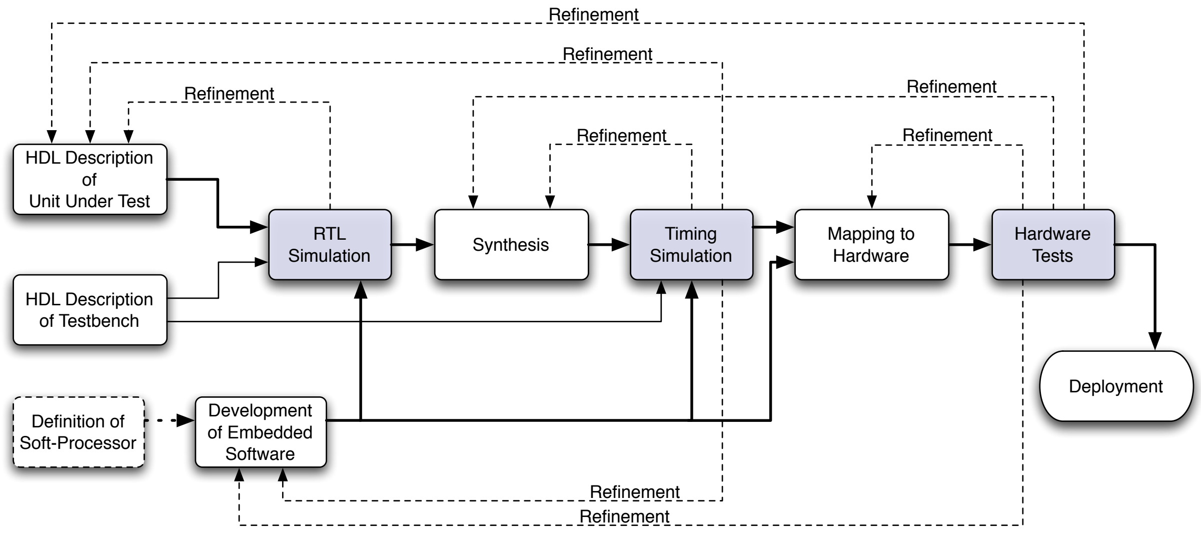


Figure 4.1: Typical design flow adopted for fast–prototyping hardware platforms. The step of defining a soft–processor is applied only when the microprocessor / microcontroller is not implemented using a ready–made device.

the description. Otherwise it is possible to pass to the timing simulation stage, using the code produced by the synthesizer. At this stage all the necessary time constraints should be specified in order to obtain more significative simulation results. The timing simulation allows to know if the UUT can be run at the selected speed on the target device, or if some architectural changes are needed to match the specifications. Even from this point it could be necessary to step back and make some changes in the HDL description or in the synthesizer's parameters. the last main step is the mapping onto a programmable device. Here the constraints to be met concern the resources available on the target device. An eventual overmapping can only be corrected by downscaling the UUT or by changing the target device. Less severe cases of overmapping can be dealt with by changing some implementation strategies in the synthesis stage. The very final step is the test of the system running on the hardware platform. At this stage, the needed stimuli can be provided by an external pattern generator or by a testbench mapped directly into the device (the choice depends on the application).

Some changes on the design flow must be taken into account when the architecture contains also a software part, as in our case. If the device running the software is a ready–made device (like a DSP or microcontroller), some testbench that reproduces its behavior should be developed. In some cases the manufacturer provides models that enable the designer to simulate the device in an RTL simulator, along with the rest of the architecture. On the other hand, if one chooses to use a soft–processor, the device must be configured in a way suitable for the application (*Definition of Soft–Processor* in Figure 4.1). In this latter case most of the integrated development environments enable the designer to simulate the soft–processor (and the software running on it) along with any peripheral and custom hardware present in the architecture, greatly simplifying the design and test process. A step that is independent from whether one uses a soft–processor or a ready–made device, is the development and testing of the embedded software. The strategies that can be used are many. For example it is possible to develop and test the hardware and the software part separately (using a specific testbench for each part), and then put them together for final testing. Another option is to develop and test thoroughly one of the two parts and then use it as the testbench for the development of the other one. The choice is usually made according to the features of the specific architecture to be developed.

4.1.1 Choice of the design tools and strategies

As just said, the architecture proposed in Chapter 3 requires the development of both a software and a hardware part. The implementation of the whole architecture is undoubtedly a complex task and goes actually beyond the goals of this work. For that reason in this chapter, and in the ones following, I will describe only the implementation of the custom hardware for the HMM processors, also because it is the most innovative and critical part of the architecture. Even if the implementation of the rest of the system is left to future work, particular attention will be payed on the effects of any design choice made for the part described onto the whole architecture. Furthermore implementation guidelines will be provided for the part that will be implemented in a later stage.

The architecture of a single HMM processor have been described at RTL level using the VHDL language [79]. A testbench has also been described in VHDL, reproducing the expected behavior of the rest of the system (i.e. the system controller, the front–end and the external memories).

The design has then been simulated using both the internal simulator of the software used for design entry[1] and with *Modelsim* [81]. Since HMM model parameters were needed in order to test the proper operation of the system, they were generated using *HTK*, an open–source HMM modeling toolkit [82]. From HTK, also the HMM simulator was used in order to verify the correctness of the results produced by the system (for more details on testing see Section 4.3).

After checking the correct behavior of the design, a platform has been chosen for synthesis and mapping. As already said only the custom hardware is going to be implemented for this thesis. This means that at this stage any generic development board would be suitable. For this thesis *Xilinx* [83] devices are used, so *Xilinx ISE* integrated development environment (*IDE*) [84] has been used. This IDE provides tools for synthesis and mapping on Xilinx FPGAs and also an *Embedded Development Kit* (*EDK*) that allows to map custom soft–processors (Xilinx's IP is called *MicroBlaze*) onto the FPGA itself. Since Xilinx produces a wide variety of FPGAs, it is possible to first synthesize the design and then choose a specific device according to the amount of needed hardware resources.

The synthesis process produces also an HDL code with timing informations. This was used

[1] In the present work *Aldec Active HDL* was used [80].

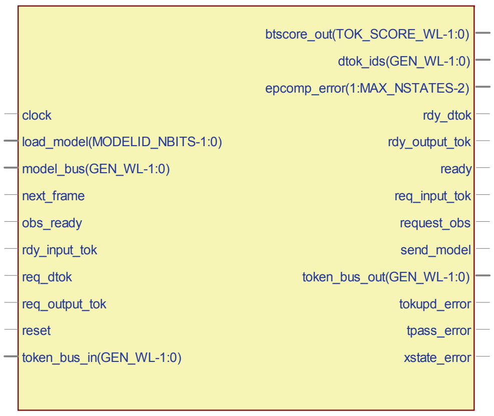


Figure 4.2: The HMM processor entity as generated by the design entry software. Input ports are on the left and output ports on the right.

to perform a simulation taking into account also signals' propagation times. This insures that the design is going to behave in the same way as the simulated model also when mapped onto the selected device.

The final step of the process was to map the design onto a development board and check that the behavior was the one expected.

More details on the topics explained here will be given in the next sections.

4.2 Implementation details

In this section I will explain in detail how each component of the architecture shown in Chapter 3 was described in VHDL, along with its features and its interface. I will use a top–down approach starting from the HMM processor itself.

It is important to stress that an effort has been made to make the VHDL code completely device independent. This allows to easily port the design on other platforms with a minimum effort. Furthermore the code has been conceived to be as much parametric as possible, making possible to scale it in many ways to fit the needs of the application in which the hardware is going to be used. For this latter purpose a package has been written, containing all the parameters that characterize the system, like number of HMM states, word length to be used for probability computations, gaussian mixtures' size parameters and so on.

4.2.1 The Top Level: HMM Processor

The HMM processor was described as a self consistent entity named `hmmp`. Its interface is shown in Figure 4.2. As any other entity in the design it has a `clock` and a `reset` port. Then

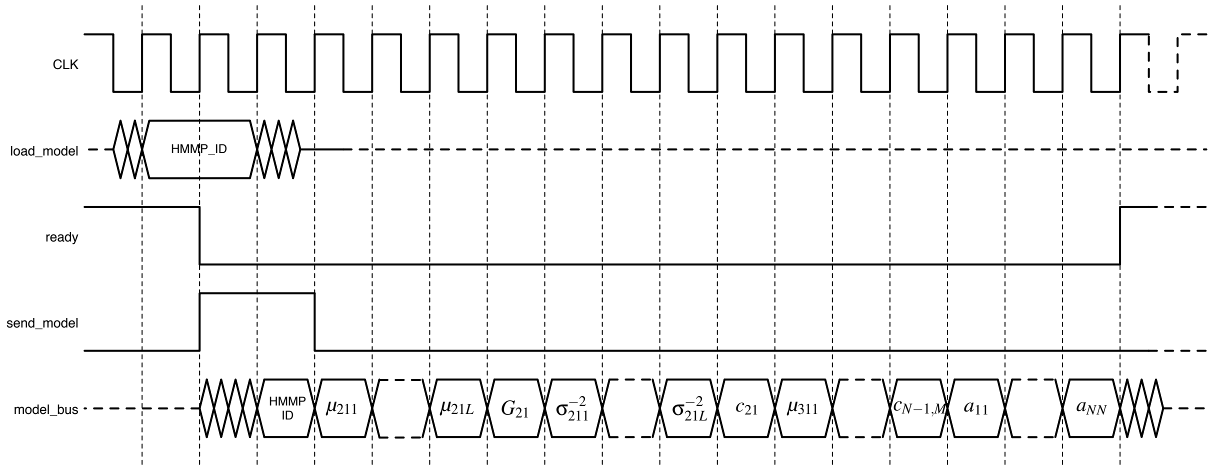


Figure 4.3: Expected timing diagram for model parameters loading into the internal HMM processor's memory.

there is a set of *error* signals that notify if there was some problem in processing data (the meaning of each error signal will be explained later, in the section dealing the corresponding entity). The bus `btscore_out` provides the best token at the end of each time step. This is used from the system level controller for pruning purposes. The `ready` signal notifies that the HMM processor is ready for processing. This basically means that a set of model parameters has been properly loaded and that the entity is idle. The `load_model` port is directly connected to the external device that is in charge of loading HMM model parameters into the HMM processors. When instantiated, the HMM processor is assigned an unique identification number through the generic `HMMP_ID`. If this number is received on the `load_model` port and the HMM processor is idle, the process of loading a new model is initiated by asserting the `send_model` signal. Then the model is received through the port `model_bus`. The first value received must be the model ID itself to notify that the rest of the data is incoming. The model data is sent in the following way: for each of the emitting states[2] S_j, the M gaussian components of the mixture are sent sequentially, following the order $\boldsymbol{\mu}_{jm}$, G_{jm}, $\boldsymbol{ivar}_{jm}$ and c_{jm} (refer to Equation (3.9), Page 63). Finally, after all the mixtures are sent, the matrix of transition coefficients is loaded. During the loading process the `ready` signal goes low and it is asserted again when the process is complete. The expected waveforms during the loading process are shown in Figure 4.3. The number of clock cycles needed for model loading is proportional to the number of parameters in the model set thus:

$$N_{cycles} \propto 2M(N-2)(L+1) + N^2 \tag{4.1}$$

The `next_frame` input port is used to notify to the HMM processor that the system is passing to the next time step. The handshaking signals `request_obs` and `obs_ready` are used to

[2] In this implementation, the emitting states for an N–state HMM are indicated with the indexes from 2 to $N-1$ since the indexes 1 and N are reserved for the entry and exit state respectively.

request a new observation from the front–end (or, to be more precise, from the buffer in which it is stored). Observation data is sent through the `model_bus` since model and observation data–flows never overlap in time (see also Section 3.3.1, Page 56). The flow of tokens is split over two buses with their own handshaking signals: the `token_bus_in` (for receiving tokens), with the control signals `req_input_tok` and `rdy_input_tok`, and the `token_bus_out` (for outputting tokens), with the control signals `req_output_tok` and `rdy_output_tok`. Finally some ports are dedicated to the notification of token pruning to the system level controller: the output bus `dtok_ids` is used to send the identification numbers of tokens that have been pruned during the last time step, while the two signals `req_dtok` and `rdy_dtok` are used for handshaking.

When models are loaded (and the `ready` signal is asserted) the assertion of the signal `next_frame` actually starts the HMM processor[3]. First of all the `ready` signal is set to logical zero, and two parallel processes are activated. The first process gets the observations for the current time step from `model_bus`, using the signals `request_obs` and `obs_ready` for handshaking. The observations are stored in a local buffer and then the emission probabilities are computed for all the emitting states. The second process begins by getting the incoming tokens from `token_bus_in`, using the signals `req_input_tok` and `rdy_input_tok` for handshaking. Then tokens are passed between internal HMM states, and the alignment probabilities are updated accordingly. After both processes are over the tokens' scores are updated with the computed transition probabilities. Finally the token passing is applied towards the HMM's exit state, and the token to be output is selected. At this point the `ready` signal is asserted again and, *before going to next frame*, the system controller must collect the emitted token from `token_bus_out`, using the signals `req_output_tok` and `rdy_output_tok` for handshaking. Moreover, the ID numbers of the tokens pruned during the last time step must be collected through the `dtok_out` bus, using the signals `req_dtok` and `rdy_dtok` for handshaking. The IDs will be sent sequentially while the signal `rdy_dtok` stays high after the request. The whole cycle restarts when the signal `next_frame` is asserted again. If any of the error signal is asserted during processing, then the results generated must not be considered reliable.

The handshaking procedure used for communicating with external devices is really simple: basically the device requesting data asserts the request signal and waits for the ready signal. When the ready signal is asserted it means that the first requested value is on the bus. The ready signal stays high as long as there is valid data on the bus. This protocol is illustrated in Figure 4.4

4.2.2 The HMM processor control

The detailed version of the architecture in Figure 3.12 (Page 56) is shown in Figure 4.5. All

[3] In this implementation is mandatory to reset to logical zero the `next_frame` signal *before* the end of the time step, in order to avoid that the HMM processor restarts processing with unpredictable results on the overall system behavior.

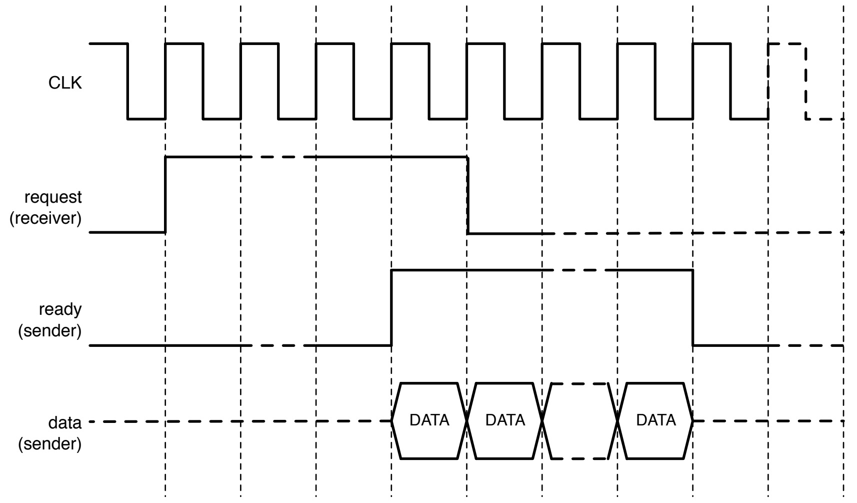


Figure 4.4: Expected signaling for a generic data transfer through a bus with two control signals.

the building blocks are implemented by a VHDL entity. The entity `hmmp_control` (shown in Figure 4.6) communicates with the external control device and provides to the entities that make up the architecture the control signals needed to insure that the data processing is performed correctly. The function of the signals `ready` and `next_frame` have already been described, and also the one of `load_model` and `send_model`. The signals `modmem_wen` and `modmem_waddr` are used to control writes on the local model memory during the model loading stage. Then four couples of signals are used to activate all the other entities in the architecture and to monitor their state:

- `compute_emprob` and `emprob_computed` for the *Emission Probability Computer* (entity `emprob_computer`)
- `pass_tokens` and `tokens_passed` for the *Token Passer* (entity `tokenpasser`)
- `update_tokens` and `tokens_updated` for the *Token Updater* (entity `token_updater`)
- `process_xit_state` and `xit_state_done` for the *Evaluate Exit State* module (entity `eval_xit_state`)

The `hmmp_control` entity also stores all the informations on the model during the loading stage. These informations are put on a set of output ports so that they can be read at any moment from the architecture components that need it for processing. These information are namely the length L of each observation vector (port `swidth`), the number M of gaussian components in each mixture (port `ngaussians`) and the number N of states in the HMM (port `nstates`). Furthermore, also the static model ID is available. That should be connected to an output port of the HMM processor to allow the system controller to distinguish between

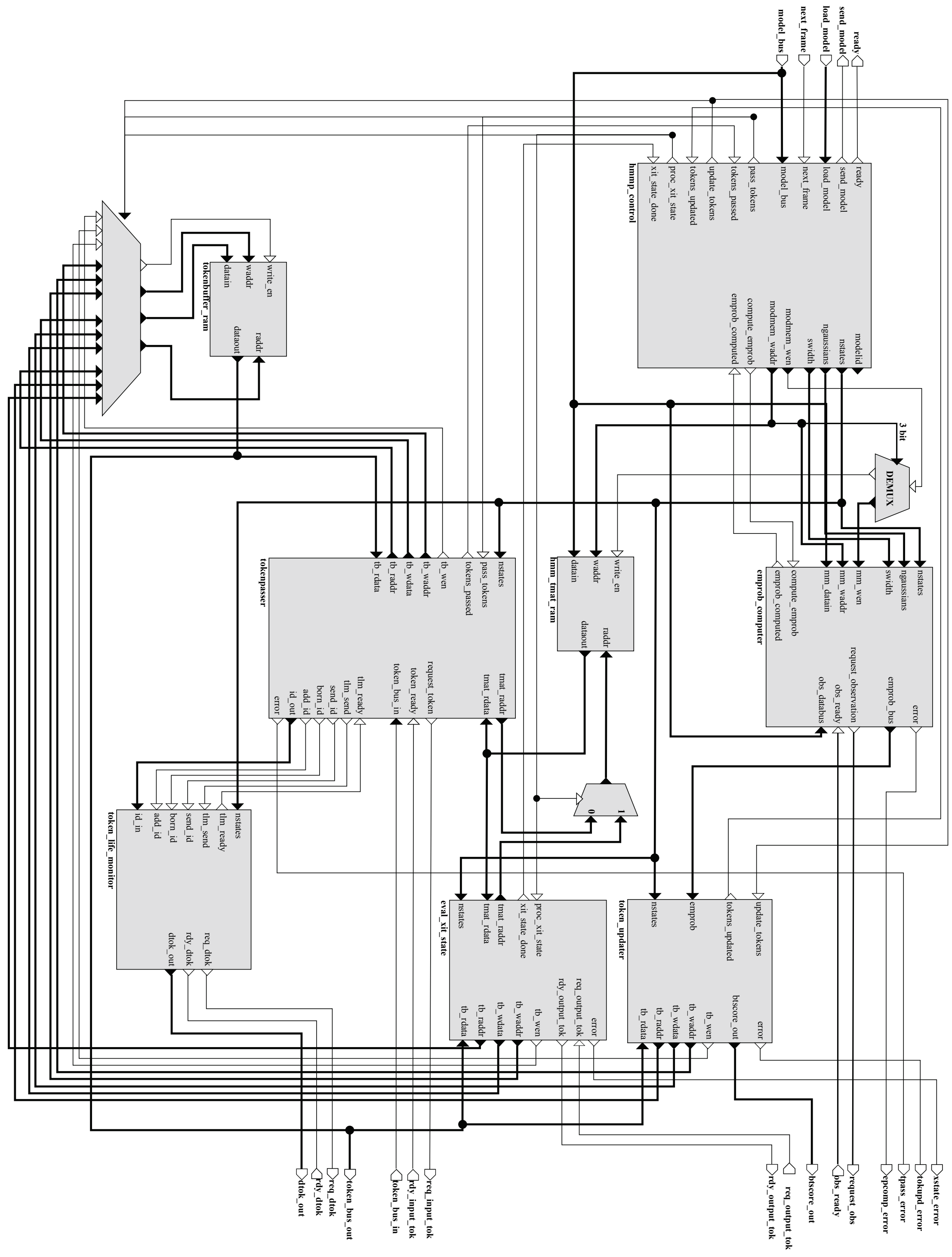


Figure 4.5: Detailed architecture of the HMM processor, as described in VHDL for implementation.

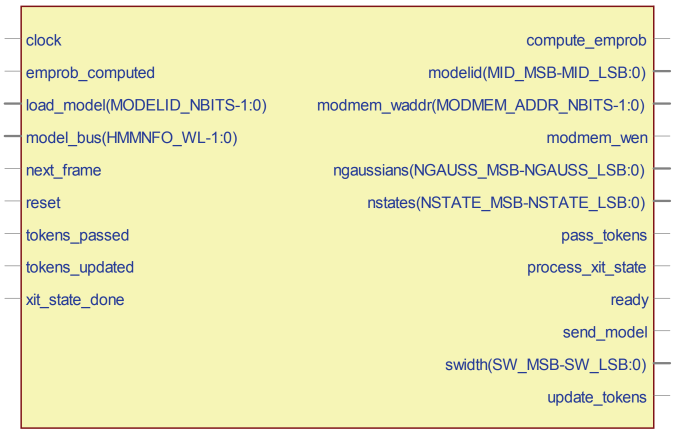


Figure 4.6: The `hmmp_control` entity as generated by the design entry software. Input ports are on the left and output ports on the right.

the physical processors. In this implementation it is left open since the tests presented here are made with a single HMM.

The representation of the state machine implemented in the `hmmp_control` (actually, the detailed version of the one in Figure 3.13, Page 57) is shown in Figure 4.7. The states' names are shown in boldface, along with the signals asserted when the state is active. In particular, for any `state = S`, is assumed a statement of the kind:

```
signal <= '1' when state = S else '0';
```

On the transition arcs are indicated the events that produce the change of state. The behavior of this FSM is substantially identical to the one described in 3.3.1 (Page 56). Furthermore, since in the `loading_model` state the local controller handles the whole parameter loading operation, another state machine is activated to perform the task. Its diagram is shown in Figure 4.8. This second FSM directly controls the handshaking with the external device that provides the model parameters. The loading process is divided into several stages. In the first stage the model is requested and the system waits until the data is available on the `model_bus`. Then the addresses are generated to load the gaussian mixtures into the local memory inside the `emprob_computer` entity. In this stage also the values for the number of states, mixtures and length of observation vector are collected and written on the ports `nstates`, `ngaussians` and `swidth` respectively. When all the gaussians are loaded, the transition coefficients are sent and the proper addresses are generated in order to write them into the local transition matrix memory (entity `hmm_tmat_ram`). Finally, when all the model has been stored, the `model_loaded` signal is asserted and the control goes back to the main FSM.

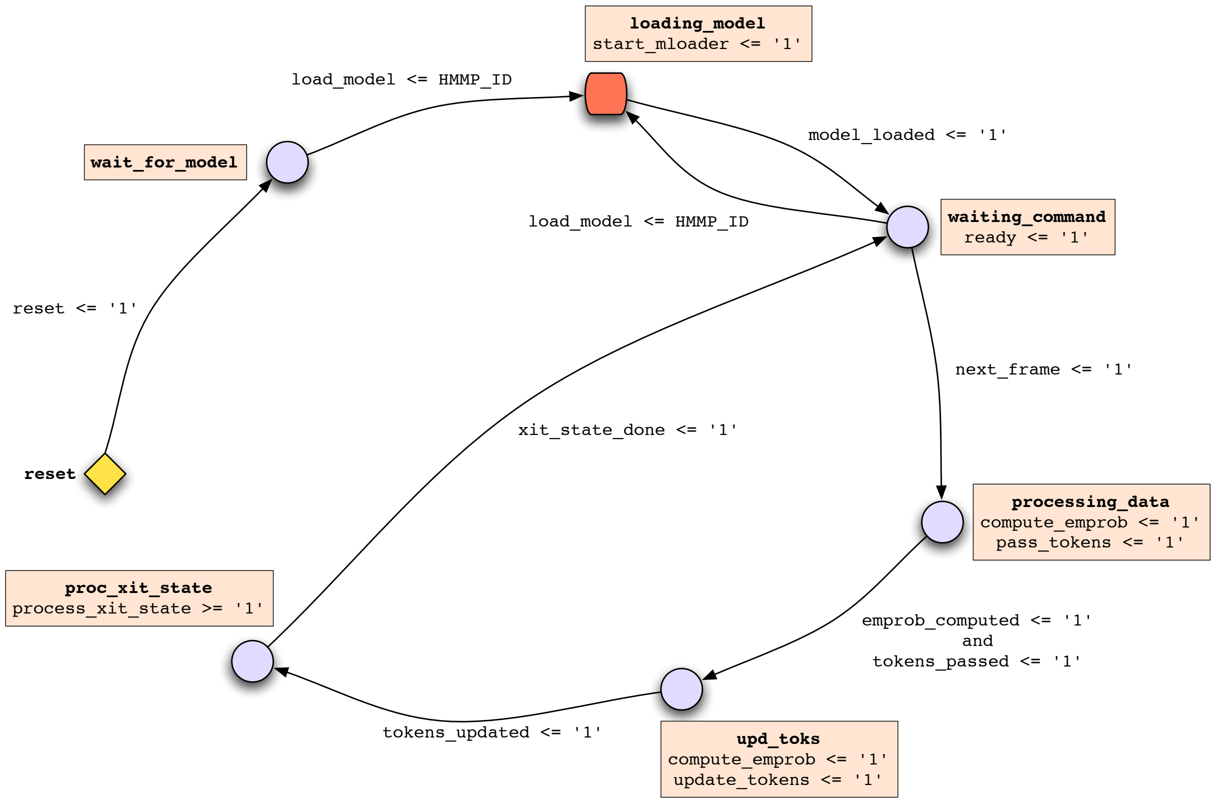


Figure 4.7: Detailed view of the state machine that controls the HMM processor.

4.2.3 The processing blocks

In this section I will refer to Figure 4.5 and give details on the implementation of the entities that make up the HMM processor's architecture. First of all I will describe the internal memories and how data is stored inside them, so that the following introduction of the processing blocks themselves will be clearer. The processing blocks will be introduced according to the order of activation given by the state machine in Figure 4.7.

Memories

The main memories used inside the HMM processor at the top level are two: one for the tokens (entity `tokenbuffer_ram`), one for the transition matrix (entity `hmm_tmat_ram`).

The `token_buffer` memory must be a dual port RAM since read and write accesses are made simultaneously during processing. Furthermore, when using dynamic allocation of HMM processors, an external memory is used to store the token buffers of all the non allocated HMMs (the HMMs that are active but not currently mapped onto any HMM processor). Thus while loading a new model the current token buffer is saved to the external memory while the one belonging to the new model is loaded from the same memory. A single token is stored as

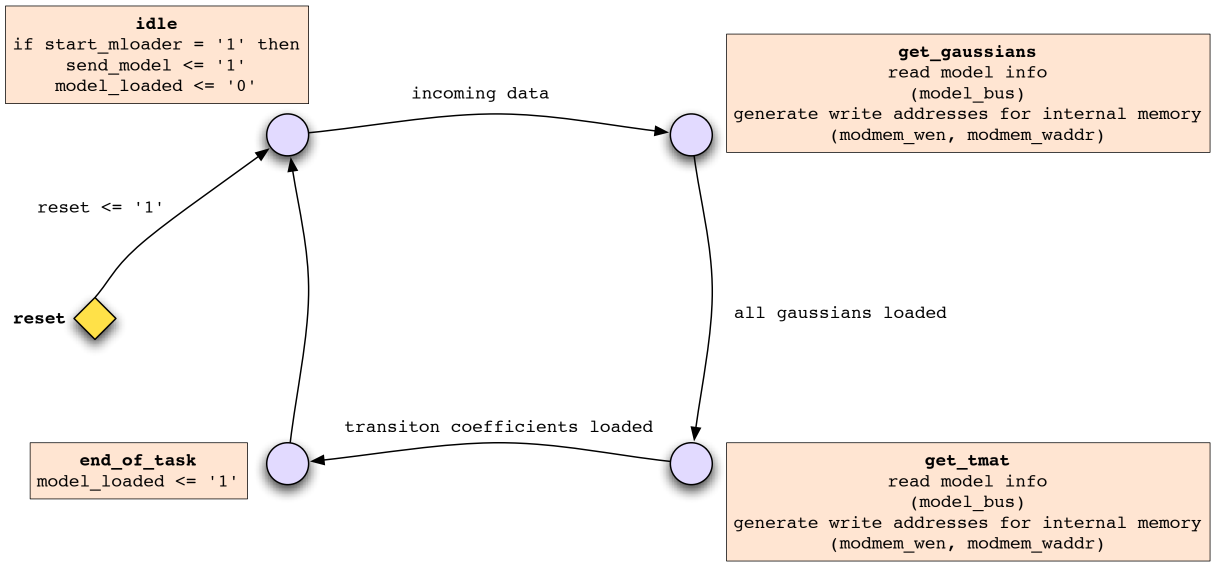

idle
if start_mloader = '1' then
send_model <= '1'
model_loaded <= '0'
incoming data
get_gaussians
read model info
(model_bus)
generate write addresses for internal memory
(modmem_wen, modmem_waddr)
reset <= '1'
reset
all gaussians loaded
transiton coefficients loaded
end_of_task
model_loaded <= '1'
get_tmat
read model info
(model_bus)
generate write addresses for internal memory
(modmem_wen, modmem_waddr)

Figure 4.8: Detailed view of the state machine that controls the loading of model parameters.

two memory *words*: the first one containing the HMM identification number of the last model from which the token was emitted and the alignment score; the second one contains a back-pointer to the last record of the chain stored in the lattice (see Section 3.2.4, Page 53 for further details). Since the number NB_{HMMID} of bits needed to represent the HMM ID depends on the number N_{HMMS} of HMM used by the application (to be precise $NB_{HMMID} = \lceil \log_2(N_{HMMS}) \rceil$), the word length WL must be chosen is such a way that there are enough bits NB_{SCORE} to represent the alignment scores (that because $NB_{SCORE} = WL - NB_{HMMID}$). The structure of a single token is shown in Figure 4.9(c). A single token buffer must be able to contain, for each of the N states, N or more tokens[4] and the whole buffer must be mirrored (see Section 3.3.1, Page 56). So the `token_buffer` RAM, in our implementation where N tokens are kept for each state, will contain a number of words given by $NW_{TB} = 4N^2$. The addresses are composed by concatenating a bit to identify one of the two banks, followed by the state number, the token number (in the state's list) and a bit to specify one of the two fields. The number of bits needed for addressing will be $NB_{TBADDR} = 2 + 2\lceil \log_2(N) \rceil$. The structure of the token buffer is shown in Figure 4.9(a).

The `hmm_tmat_ram` memory does not need to be a dual port RAM since write and read access never overlap in time (the former occurs during model loading, the latter during processing). This memory must contain one word for each transition coefficient, that is N^2 words. The items are stored in such a way that the transition coefficient a_{ij} will be addressed by simply concatenating the representations in $\lceil \log_2(N) \rceil$ bits of the numbers i and j. So the number of

[4] N is the minimum to be able to collect the token coming from all the other states (in the hypothesis of a fully connected HMM). More than N tokens are needed only when collecting multiple tokens from every incoming connection in order to produce multiple hypotheses.

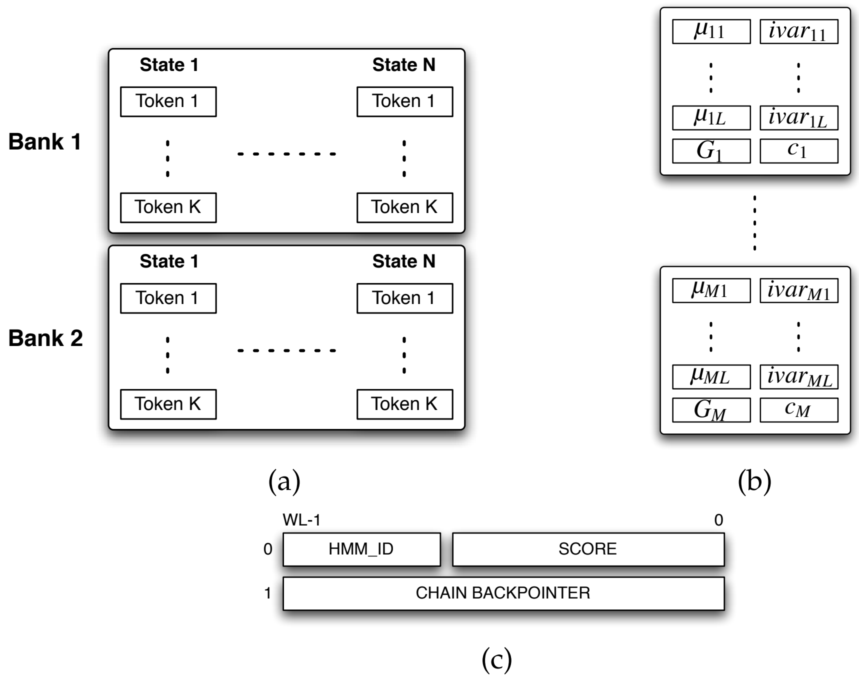


Figure 4.9: (a) Structure of the token buffer RAM. (b) Structure of a single HMM state RAM. (c) Structure of a single token as stored in memory.

bits needed for addressing is always $2\lceil\log_2(N)\rceil$.

All the other memories in the HMM processor are inside the `emprob_computer` entity (see ahead in this section for details). One is used to locally store the observations (entity `observation_ram`), and the others are used to keep gaussian mixtures. Since the entity processes each HMM state in parallel, there is a memory (entity `hmm_state_ram`) that holds a single gaussian mixture for each of the $N-2$ emitting states in the HMM.

The `observation_ram` is a single port RAM capable of holding a complete observation vector, thus L parameters. The addressing is sequential.

The `hmm_state_ram` is also a single port RAM. It must contain a whole gaussian mixture, so N means, N inverse variances, a mixture weight and a G constant. In total it is $NW_{SMEM} = 2M(L+1)$. The address of mean and inverse variance elements is obtained by concatenating the mixture number (from 0 to $M-1$, represented with $\lceil\log_2(M)\rceil$ bits), a bit that will be `'0'` for means and `'1'` for inverse variances, and the parameter number (from 0 to $L-1$, represented with $\lceil\log_2(L+1)\rceil$ bits). The G constant and the mixture weight are addressed as the $(L+1)^{th}$ element (index L) of the mean and inverse variance vectors, respectively. The structure of the state RAM is shown in Figure 4.9(b).

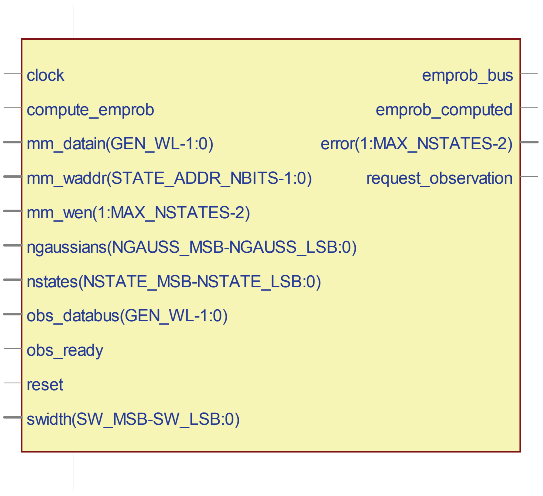


Figure 4.10: The entity `emprob_computer` as generated by the design entry tool.

Computation of emission probability

The interface of the `emprob_computer` entity is shown in Figure 4.10. Control on the entity is provided through the signal `compute_emprob` (enable) and feedback on the entity's status through the signal `emprob_computed`. The model parameters are loaded on the internal memories through the `mm_datain` port, using the signals `mm_wen` and `mm_waddr` to control the internal memories (write enable and write address, respectively). Through the input ports `nstates`, `ngaussians` and `swidth` the HMM controller provides size informations on the model, that are needed for the computations. During processing the observation data is got directly from outside the HMM processor into the local buffer using the `obs_databus` port and the `request_observation` and `obs_ready` signals for handshaking. Computation results are output through the `emprob_bus`. Eventual errors are notified by asserting the `error` signal. The `emprob_computer` entity's internal architecture is shown in detail in Figure 4.11. The entity is made up by a control unit `ec_cpu_ctrl`, a buffer `observation_ram` for the current observation vector, and a processing block for each of the $N-2$ emitting states in the HMM[5] (in Figure 4.11 is shown a 5–state HMM). Each processing block contains a RAM `hmm_state_ram` to store the gaussian mixture relative to the corresponding emitting state, a block `eval_gaussian` able to compute the score for a single gaussian component, and a `log_add` module to compute the soft–add operation among the M values produced by `eval_gaussian`.

The control unit implements the state machine shown in Figure 4.12. The entity becomes active only when the signal `compute_emprob` is asserted. The process starts in the state `idle` and, if an observation vector has not yet been loaded into the local buffer, it switches

[5] In practice the architecture is built to be able to handle a certain maximum number N_{MAX} of HMMs. So the processing blocks in `emprob_computer` will be $N_{MAX}-2$, and some of them will remain idle when the actual number of states in the loaded model is less than the maximum allowed.

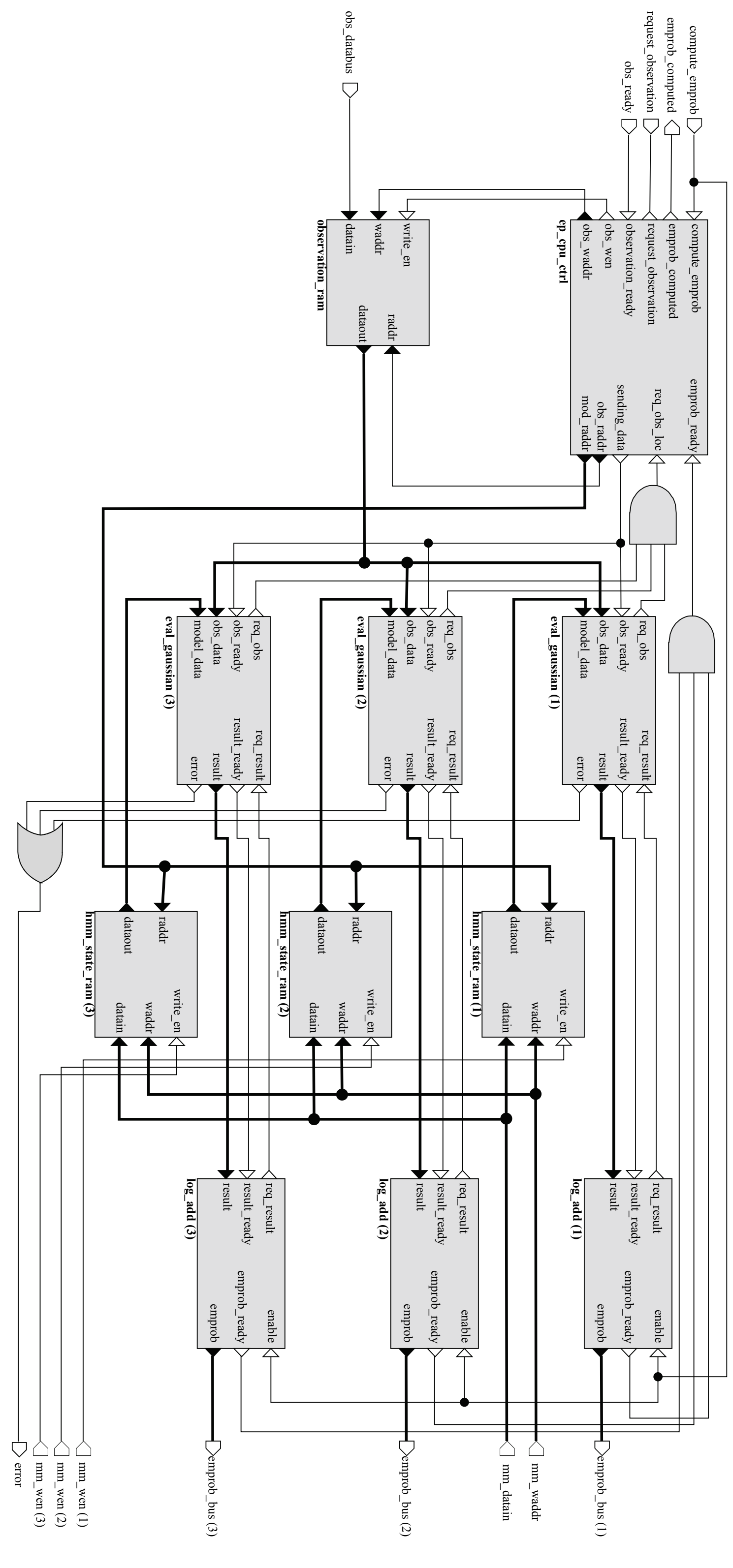


Figure 4.11: Details of the `emprob_computer` entity's internal architecture. This diagram shows the case of a 5–state HMM.

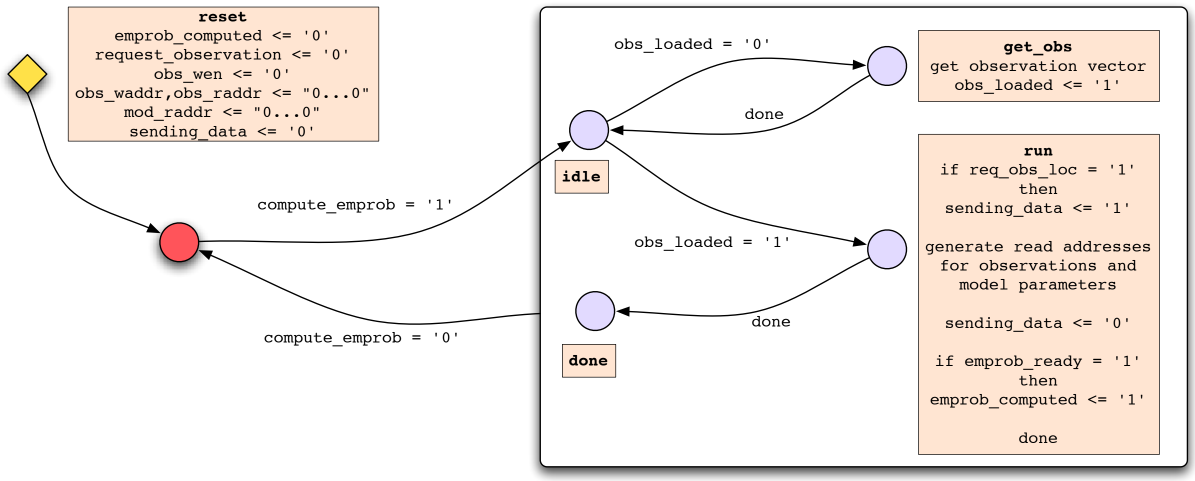


Figure 4.12: The FSM implemented by the Emission Probability Computer's control unit `ep_cpu_ctrl`.

to `get_obs` state. In this state the observation vector is loaded and then the FSM goes back to `idle` state. When there is an observation vector in the local buffer, from `idle` state the FSM switches to the `run` state. In this state the controller first waits for the `req_obs_loc` to be asserted (i.e. all the `eval_gaussian` blocks are ready to process). Then the addresses are generated in order to send the observation values and the model parameters, in the correct order, to the `eval_gaussian` entities. When the first data are ready on the observation and model buses, the `sending_data` signal is asserted to notify the `eval_gaussian` blocks to begin processing. Finally when the computation is over, the `emprob_ready` signal is asserted. This means that it can be notified to the external entities, through the `emprob_computed` signal, that emission probabilities can be read and used. The process ends in the `done` state and stays like that until the signal `compute_emprob` goes low and high again for the next frame.

All the `eval_gaussian` blocks, when instantiated, have assigned an unique identification number from 0 to $N_{MAX} - 2$. If the number of model states, provided through the port `nstates`, is less than the ID, the block disables automatically. Otherwise it gets ready for processing. The processing itself begins on assertion of the signal `obs_ready`. This means that the first value of the current observation vector is ready on the bus. For each of the M gaussians, the value

$$P_{jm} = G_{jm} + c_{jm} - \frac{1}{2}\sum_{i=1}^{L}(O_{ti} - \mu_{jmi})^2 ivar_{jmi} \tag{4.2}$$

is computed. The value is put on the `result` bus and the signal `result_ready` is asserted as soon as the `req_result` signal goes high (so that the value can be safely read). After that the process repeats with the next gaussian component. Actually the parameters used in the computation of (4.2) are floating point numbers (in this work they were generated using the HMM toolkit *HTK* [82]). Thus some normalization was performed to allow a correct representation and avoid precision issues with the operations to be performed. After considering the typical ranges of the parameters, and assuming that the calculations are done with a WL bits precision, I obtained the following formula for the computation performed by the hardware:

$$2^{(WL-\Delta_1)}P_{jm} = -2^{-1}\left\{G'_{jm} + \sum_{i=1}^{L}\left[\left(2^{-(WL-\Delta_1)}(O'_{ti}-\mu'_{jmi})^2\right)\cdot 2^{-(WL-\Delta_2)}ivar'_{jmi}\right]\right\} \tag{4.3}$$

Here scaled variables are used. The values are stored directly in the scaled form and are defined as follows:

$$G'_{jm} = 2^{(WL-\Delta_1)}(G_{jm}+c_{jm}) \tag{4.4}$$

$$O'_{ti} = 2^{(WL-\Delta_1)}O_{ti} \tag{4.5}$$

$$\mu'_{jmi} = 2^{(WL-\Delta_1)}\mu_{jmi} \tag{4.6}$$

$$ivar'_{jmi} = 2^{(WL-\Delta_2)}ivar_{jmi} \tag{4.7}$$

The values for Δ_1 and Δ_2 are chosen on the basis of the specific model parameters being used, and of the available word length WL. I will show in Section 4.3 that in the actual tests, good results have been obtained with $\Delta_1 \in [8,12]$ and $\Delta_2 - \Delta_1 = 3$.

In the arithmetic architecture (see Figure 4.13) the most significant bit of each variable involved in the computation is doubled in order to allow for overflow and underflow checks throughout the process. Double precision is used temporarily to store multiplication results. The shift values Δ_1 and Δ_2 can be set through the package parameters $LSO = WL - \Delta_1$ and $LSV = WL - \Delta_2$, in order to match the normalization used for any specific parameter set.

The last element in the computing block of the `emprob_computer` entity is the `log_add`. This block has the same ID–based activation mechanism used also for the `eval_gaussian` entity (port `nstates` is used to get the number of states in the model, and port `id` to get the entity's identification number). When active and enabled through the `enable` signal, the module gets the M gaussian score from `eval_gaussian` using the signals `req_result` and `result_ready` for handshaking. Data is received through the `result` port. Whenever a new value is received, the request signal `req_result` is put low, and a side process is activated that actually accumulates the results using the soft–add operation. In this implementation this module is simplified and just keeps the maximum among the values received. As can be seen in 4.3, the results are good even with this approximation. Anyway the extension is quite straightforward because every FPGA manufacturer provides an IP that implements the Cordic algorithm [77], with which the f_{MAP} function (see (3.7), Page 62) can be easily evaluated. In these cases the constraints are given by the tradeoff between precision and used hardware resources. When the side process has finished to compute a single soft–add operation, the main process gets a new gaussian score and the cycle repeats until all the M values have been received (the number of gaussians to read is obtained through the `ngaussians` port). When the

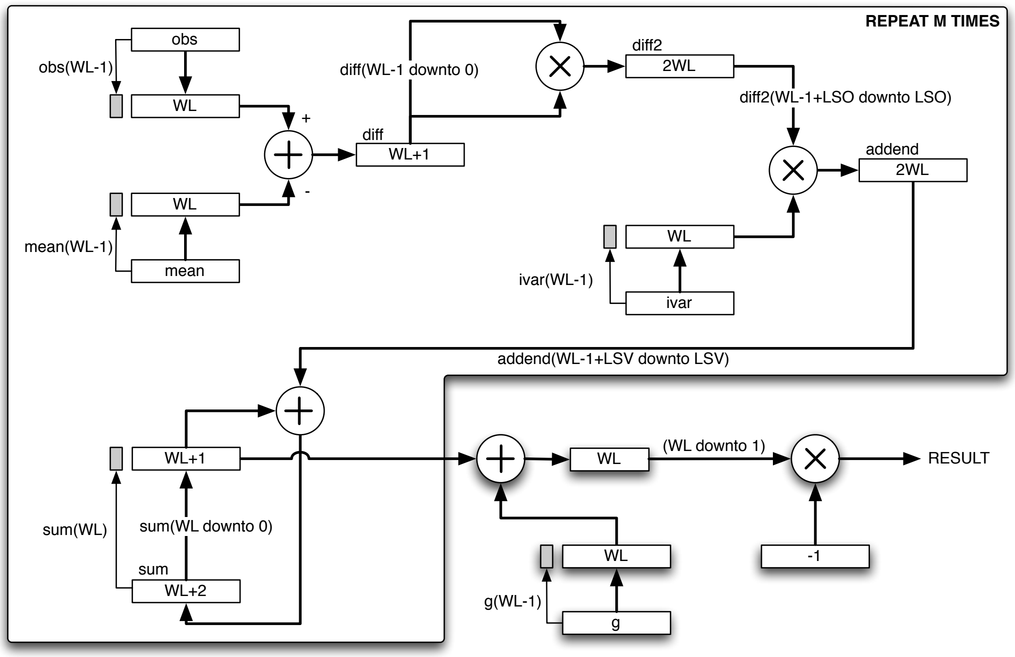


Figure 4.13: Evaluation of a single gaussian component from a mixture. Boxes represent registers (word length is indicated inside). The hardware for error detection is not shown.

result is ready it is scaled to the word length NB_{SCORE} used for token scores, and registered on the output port `emprob`. Contextually the signal `emprob_ready` is asserted, so that external units know when to safely read the value. The entity `log_add` restart processing only when `enable` goes down and then up again.

Token Passing

The token passing between internal states of the HMM is handled by the entity `tokenpasser`. The technique used is to sequentially process each emitting state by collecting all the incoming tokens, applying the transition probabilities, and selecting the token with the highest alignment probability in each state. The sequential behavior has been chosen because there is no big advantage in processing the states in parallel since all of them would read simultaneously from the same memory (the first bank of the `tokenbuffer_ram`). The entity's interface is shown in Figure 4.14. The signals `pass_tokens` and `tokens_passed` are used for enabling the process and notifying its end. The process is controlled by the state machine shown in Figure 4.15 that is substantially identical to the one shown in Figure 3.14(b) (Page 59). Upon enable, the entity starts in the state `get_incoming`. In this state the tokens fed from outside are collected through the port `token_bus_in` using the signals `request_token` and `token_ready` for handshaking. Incoming tokens are saved into the entry state location of the token buffer's first bank (writes to the token buffer are managed through the ports `tb_wen`, `tb_waddr` and

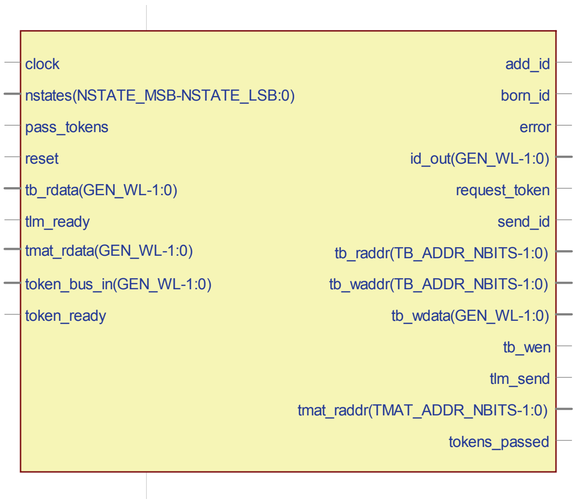


Figure 4.14: The `tokenpasser` entity interface as generated by the design entry software. Input ports on the left, output ports on the right.

`tb_wdata` for write enable, addresses and data respectively). When collection has been performed, the process begins the actual token passing by switching to the `passing` state. At this point each of the emitting states is processed sequentially. For each state, tokens are collected from *all* the N states in the HMM[6], and the corresponding transition probability is applied to their alignment score. Reads from the token buffer are handled using the ports `tb_raddr` and `tb_rdata` (for addressing and data respectively). All the collected data is read from token buffer's first bank: at any time step t the first bank of the token buffer will always contain the results from time step $t-1$. The transition probabilities are read from the `tmat_ram` through `tmat_rdata`, using `tmat_raddr` for addressing. For each state track is kept of the token that has highest score after the application of the transition probability. Also, any best token coming from the entry state, is marked as *newborn*. These informations are used by the *Token Life Monitor* to keep track of the birth and death of tokens in the HMM. Results of the token passing computations are stored in the second bank of the token buffer. After the token passing, the state switches to `notify_survive`. Here IDs of the survived tokens are read from the token buffer and sent to the *Token Life Monitor*. The `tokenpasser` waits for the `tlm_ready` signal from the `token_life_monitor` entity, then asserts `tlm_send` to notify that data is being sent. A set of flag–signals provides the needed informations on the token whose ID is being sent through the `id_out` bus. The `send_id` signal just tells the `token_life_monitor` that the ID on `id_out` must be recorded. The signal `born_id` notifies that the ID belongs to a newly born token, so that a new counter in the Token Life Monitor must be initialized for that ID. Finally the `add_id` signal indicates if the token counter must be incremented (signal

[6] This is done to simplify the control: states from which there is no transition towards the current state have a zero transition probability that, when applied to any token's alignment score, will cause the immediate pruning of the token itself.

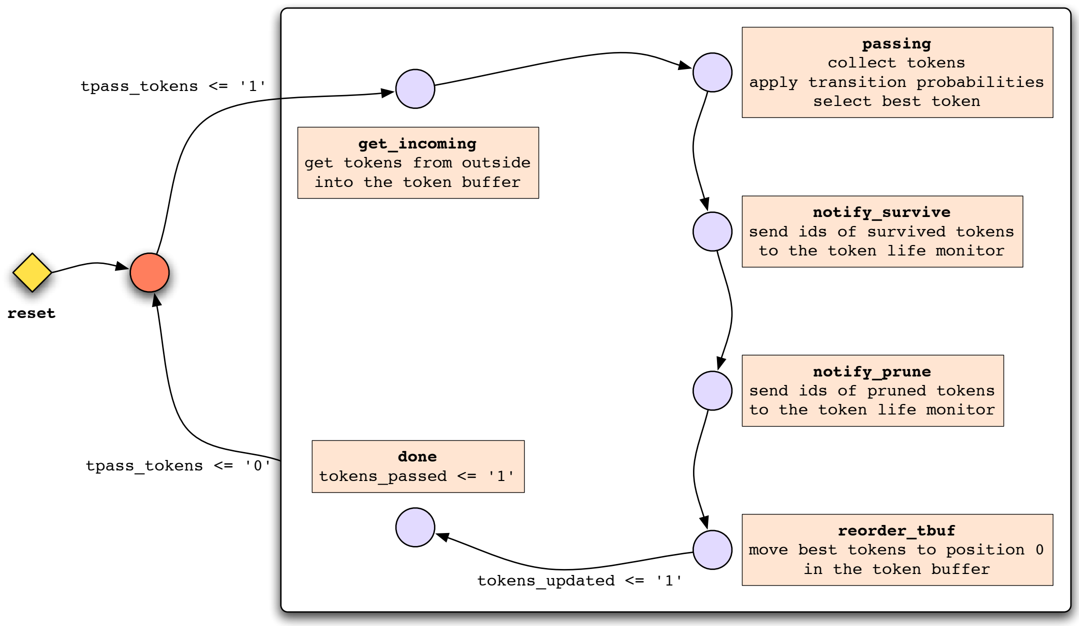


Figure 4.15: The FSM that controls the `tokenpasser` entity.

high) or decremented (signal low). Thus in `notify_survive` state `add_id` will be always high. The tasks performed in the next state, `notify_prune`, are almost the same as the ones performed in `notify_survive`. In particular both the signal `born_id` and `add_id` will be always low (because no new counters are initialized during pruning, and counters are always decremented). The last stage of processing corresponds to the `reorder_tbuf` state. Here the best token of each state is moved, in the token buffer's second bank, in order to be first of the state's token list[7]. Finally, when the `done` state is reached, the signal `tokens_passed` is asserted to notify that the results of the token passing can be read from the second bank of the token buffer.

Notification of tokens' lifespan

The `token_life_monitor` entity's task is strictly related to the token passing process since token pruning mechanism depends only on the token passing stage. Its interface is shown in Figure 4.16. After reset the process asserts the `tlm_ready` signal and waits for the assertion of `tlm_send` to activate. When active it manages an array of counters according to the IDs incoming from the `id_in` bus and the flag signals `born_id` and `add_id`, as already described. Whenever a token's counter reaches zero, the token's id is stored in a list of pruned token's IDs, and the counter is freed in order to be used for the next newborn token. When the `tlm_send`

[7] In a multiple–tokens–per–state implementation, the `reorder_tbuf` state would be in charge of sorting the token lists by score, with the best tokens on top.

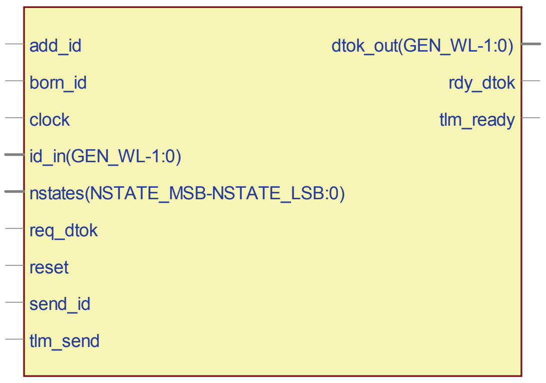


Figure 4.16: The `token_life_monitor` entity interface as generated by the design entry software. Input ports on the left, output ports on the right.

signal goes low (meaning that the `tokenpasser` entity has sent all the token informations for the current time step), the process goes in a state in which it waits for an outside device to request the list of pruned token's IDs. The request is made through the `req_dtok` signal. Upon request the signal `rdy_dtok` is asserted and the IDs are sent sequentially through the `dtok_out` bus. After the last value is sent, the `rdy_signal` is put low, the signal `tlm_ready` is asserted again, and the entity goes idle again, waiting for the next `tlm_send` assertion.

Finalization and results' output

The last tasks to accomplish in order to complete a time step processing, is the application of the emission probabilities and the selection of the token to output.

The `token_updater`, whose interface is shown in Figure 4.17(a), is in charge of performing the first task. Upon enable through the `update_tokens` signal, the entity collects the best token for each state from token buffer's second bank (through the `tb_raddr` and `tb_rdata` signals), collects the corresponding emission probability from the `emprob` bus, updates the token's score, and writes the result in the *first bank* of the token buffer. Furthermore, the best token among all the states is selected and its score is registered on the `btscore_out` port. At the end of the process the signal `tokens_updated` is asserted.

The selection of the token to be output is instead performed by the `eval_xit_state` entity (its interface is shown in Figure 4.17(b)). The entity is enabled through the `proc_xit_state` signal. Upon enable the process gets the tokens from token buffer's first bank (through the ports `tb_raddr` and `tb_rdata`), and updates the scores with the transition probabilities got through `tmat_raddr` and `tmat_rdata`. The highest scoring token is selected and written as the first in the exit state list into the first bank of the token buffer. Finally the process waits for the assertion of `req_output_tok`. When the data is requested, the read address for the output

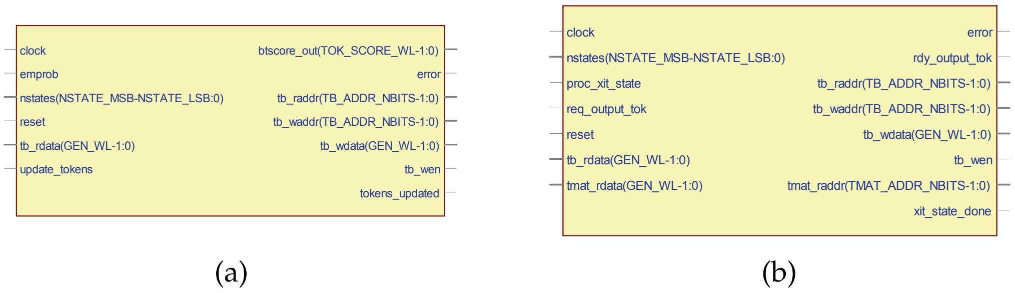


Figure 4.17: Entity interfaces of (a) `token_updater` and (b) `eval_xit_state`, as generated by the design entry software. Input ports on the left, output ports on the right.

token is generated[8] and the `rdy_output_token` signal is asserted. Upon process completion the signal `xit_state_done` is asserted until `eval_xit_state` goes low and up again.

4.3 Test of the architecture

The testing process passes through the actual implementation into hardware of the architecture (synthesis and mapping of the VHDL code) to end up with tests aimed to verify that the intended functionality has been preserved. In this section this process will be described along with the considerations that affected the various design choices.

4.3.1 System Implementation

Referring to the system description provided in Section 4.2 (Page 70), the core component of the architecture can be easily identified as the *HMM Processor* (entity `hmmp`) or, to be more specific, in a parallel array of these processors. In principle one or more of these processors can be mapped onto a single device. The choice of the number of processors in the parallel array basically depend on the resources provided by the programmable device, and on the number of bits used for the fixed point representation of the quantities to be processed (is obvious that this last characteristic directly reflects on the amount of resources needed to map a single processor).

So the first thing to do is to choose a target device. A single HMM Processor has been synthesized, with three different word lengths for fixed point representation (32, 24 and 16 bits), in order to check for resource occupation. For all these word lengths, some preliminary tests have been made in order to choose the optimal values for the parameters Δ_1 and Δ_2 (refer to 4.2.3, Page 79). The final values chosen were $\Delta_1 = 10$ and $\Delta_2 = 7$. On the basis of the results two FPGAs have been selected for mapping the design. The first one was a Xilinx Virtex-II 3000

[8] It should be noted that the output bus of the token buffer has also a direct connection to one of the HMM processor's ports (refer to the architecture diagram in Figure 4.11, Page 80).

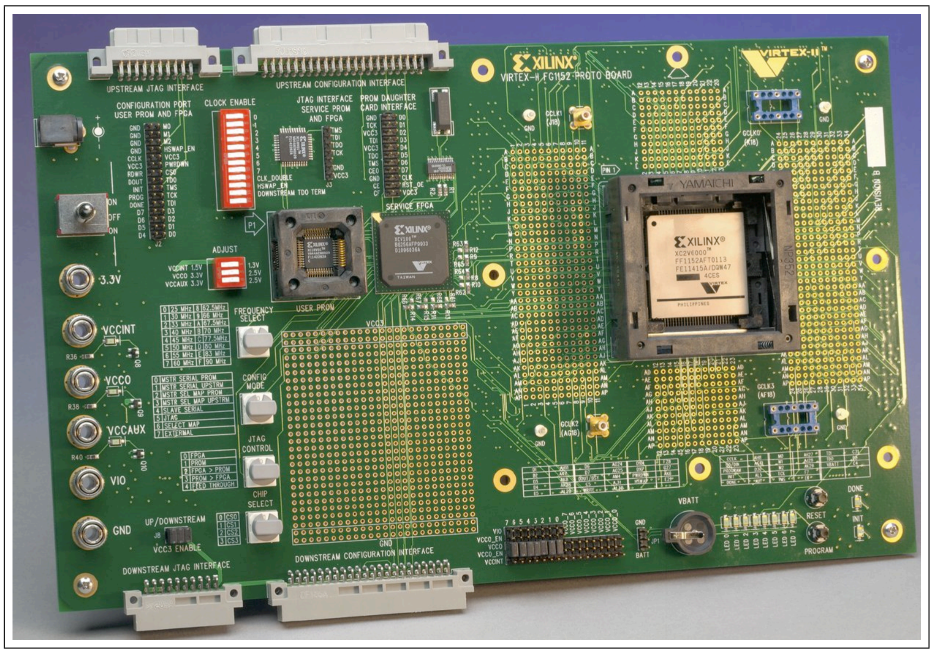


Figure 4.18: Xilinx Virtex FF1152 Proto Board used for the implementation tests on the Virtex II 3000 device.

	XC2V3000	XC4VSX55
Logic Cells	32256	55296
Block RAM (Kb)	1728	5760
18x18bit Multipliers	96	512
Digital Clock Managers	12	8
User I/O	720	640

Table 4.1: Details of the Xilinx FPGAs chosen as target devices for the architecture.

(XC2V3000). This is a device with three million equivalent gates, for which a prototyping board was available to make the tests (Xilinx HW-AFX-FF1152-200 Proto Board, shown in Figure 4.18). The second device was a Xilinx Virtex-4 VSX55 (XC4VSX55). This second device was not available on a board but has been taken as a reference because it provides the most suitable set of resources for the architecture to be mapped. The details for both of the devices are shown in Table 4.1.

The implementation strategy adopted was the following: firstly a single HMM Processor was mapped onto each of the devices in three different versions: 32-bit, 24-bit and 16-bit word length. Then resource occupation was checked for each case, and finally the highest possible number of HMM Processors (for each device and each work length) was estimated and then mapped onto the device.

Before showing the implementation results, it is important to define the target performance that the mapped device should achieve in order to allow for correct operation in the context described so far. The most straightforward measure for a digital system's performance is of course the working clock frequency. Given the main application and architectural parameters, it is possible to define a minimum clock frequency that the system should reach in order to be able to provide real–time operation. This minimum clock frequency can be easily found by computing the number of clock cycles (or *cycle budget*) needed, in a worst–case scenario, to perform all the operations that must be completed in a second. In a worst–case hypothesis we should assume $N_H < N_A$ (refer to Equation (3.1), Page 53), meaning that it will be necessary to use each HMM Processor several times for each incoming observation. So let us call CB_H the cycle budget needed to load the model parameters into the HMM Processor (CB_L) plus the cycle budget needed to fully process an observation (CB_P). In a multi–pass processing scheme (as seen in Section 3.2.3, Page 51), the number of *passes* is defined as $N_{passes} = \lceil N_A/N_H \rceil$ so that, for each incoming observation, the cycle budget will be $CB_H \cdot N_{passes}$. At this point, the minimum clock frequency is easily found by taking into account the observation frequency f_{obs} in the following way:

$$f_{clock} \geq f_{min} = CB_H \cdot N_{passes} \cdot f_{obs} = (CB_L + CB_P) \left\lceil \frac{N_A}{N_H} \right\rceil \cdot \frac{1}{T_{obs}} \tag{4.8}$$

While the values N_H, N_A and T_{obs} are found easily, the values CB_L and CB_P must be evaluated through a detailed analysis of the architecture or, more reasonably, through simulation. A waveform output from a simulation session is shown in Figure 4.19. In this simulation a single–gaussian model ($M = 1$, $N = 5$) trained with HTK is loaded and processed using as input a stream of observation vectors of length $L = 39$ each. In the figure both the loading time (5460 *ns*) and the processing time (5060 *ns*) for the HMM Processor have been marked. Since the simulation clock period is 20 *ns*, it is found that $CB_L = 273$ and $CB_P = 253$. Using these parameters it is possible to plot the minimum clock frequency in function of the observation period. This plot is shown in Figure 4.20 for $N_A = 60$, $N_H \in [1, 2, 4]$ and $T_{obs} \in [1ms, 10ms]$. It can be seen that, with single gaussian models (plot on the left), even in the worst–case situation of a single HMM processor and an observation period of 1 *ms*, the minimum clock frequency is less than 35 *MHz* (easily reachable with modern devices and accurate architecture definition). The situation changes when the models become more complex. The plot on the right in Figure 4.20 shows the common case of a model that uses a mixture of 4 gaussian components to define the emission probability PDF. Here a single HMM processor could not be enough since the cycle budget for a single observation is almost 4 times the one shown for a single gaussian model. On the other hand it is possible to reach manageable frequencies by using more HMM processors in the array[9] (in fact it can be seen that with two processors the minimum frequency for 1 *ms* observation period drops suddenly from more than 120 *MHz* down to 60 *MHz*).

[9] In some situations where the computational load is really high, a possible approach is to predict which HMMs will be active during the next frame and process only the ones that are supposed to be *active*. This allows to process at each time step only a subset of the whole atomic–HMM list. This technique is widely used in software for speech recognition.

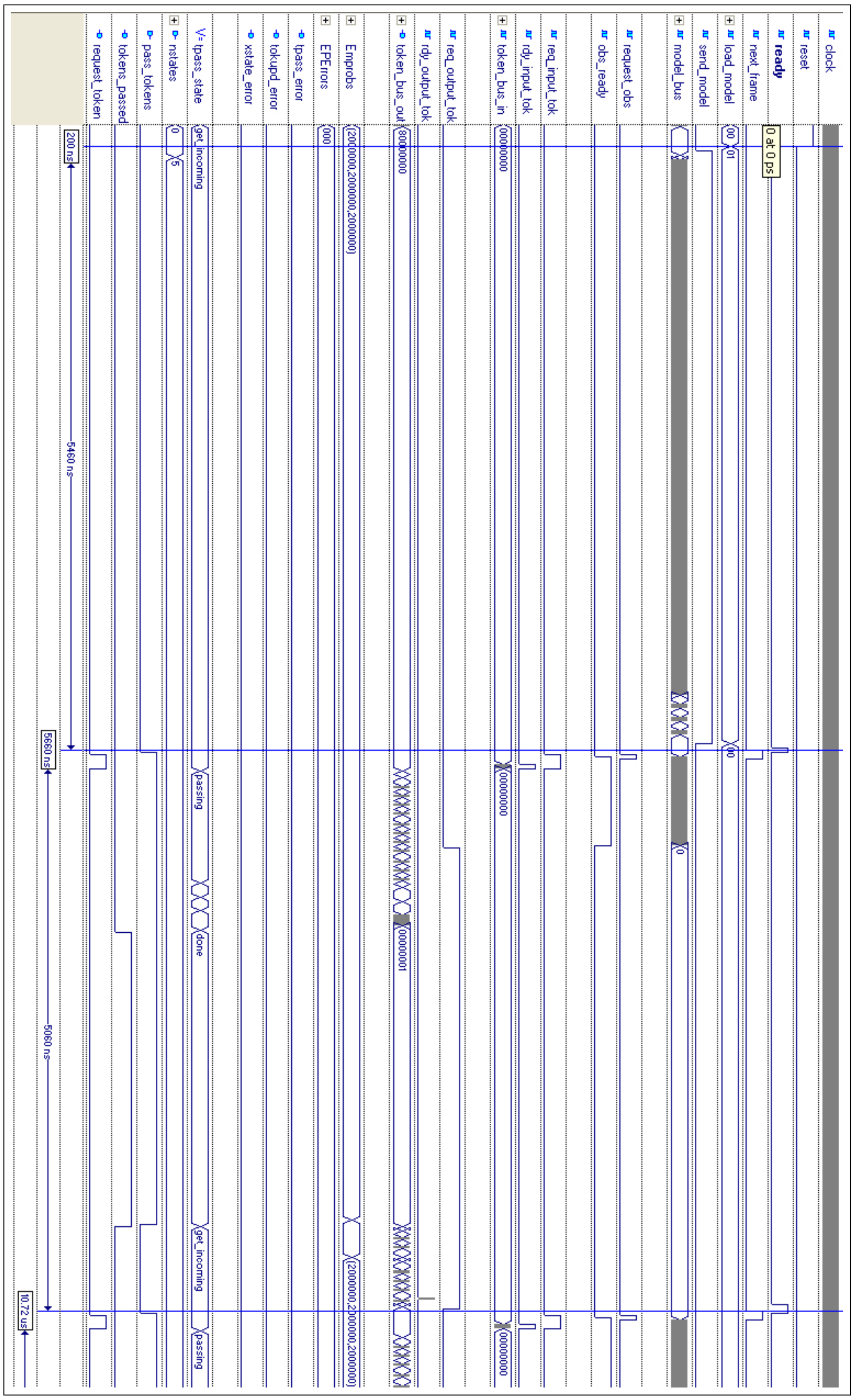


Figure 4.19: Waveforms obtained from the simulation of the HMM Processor's VHDL RTL description. The simulation clock runs at 50 *MHz*. The model loaded is a $N = 5$ HMM with gaussians of dimension $L = 36$. Loading and processing times are marked: respectively 5460 *ns* and 5060 *ns*.

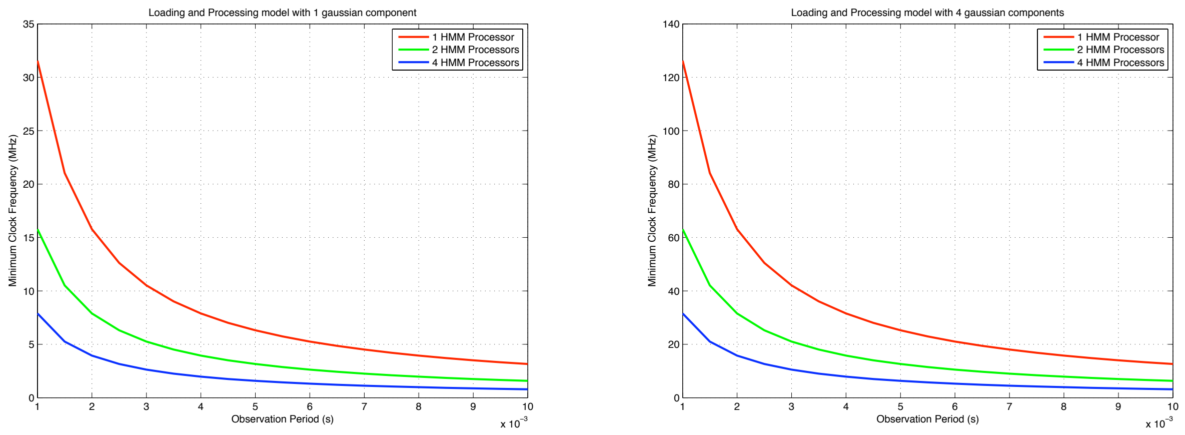


Figure 4.20: Minimum clock frequency in function of the observation period. On the left: loading and processing a HMM with single gaussian PDF for emission probabilities. On the right: loading and processing a HMM with 4–gaussian mixture PDF for emission probabilities.

Now that it is clear what should be the system's working frequency given the architectural specification and the application's details, some considerations can be made on the actual implementation results.

The implementation results for the Virtex II device are shown in Table 4.2. In the 32 *bits* and 24 *bits* case no more than 2 HMM Processors can be mapped on the device. This is actually due to the limited number of embedded multipliers (in fact in both cases only half of the FPGA slices are used): one should remember that the available multipliers are 18x18 bits (two bits are reserved for parity checks), so more than one multiplier is used for a single multiplication whenever the factors are wider than 16 *bits*. Anyway the working frequency allow for observation periods of less than 1 *ms* (note that in the table is shown only the minimum observation period for the single gaussian case. For the generic M–gaussian model that value should be multiplied by M). The 16 *bits* case is different because fewer multipliers are used thanks to the shorter word length. In this case is possible to fit up to six HMM processor into the device, and the limiting factor is actually the available resources for logic. Here the working frequency allows for observation periods far smaller than 1 *ms*. In order to give an idea of the level of resource occupation inside the device, Figure 4.21 shows some snapshots of the various configurations mapped. Taking into account all the implementation results for the Virtex 2 device, it seems that the application that would provide a better use of the available hardware resources would be one that can take advantage of the 16 *bits* precision array of 6 HMM Processors.

The implementation results for the Virtex 4 device are shown in Table 4.3. With this device the performance achievable is clearly higher with respect to the Virtex 2 device. The maximum number of HMM processors on a single device allow for a high level of parallelism in the system. Also the working frequencies are much higher. Both these features are reflected in the possibility to process observations at a much higher rate (over 30 *KHz*). Here, since the

Word Length	N_{HMMP}	f_{MAX} (clock)	T_{OBS} (min)	Occupied Slices	BRAMs Used	Embedded 18x18b Multipliers Used
32 *bits*	1	61 *MHz*	0.6 *ms*	27 %	8 %	41 %
	2	57 *MHz*	0.3 *ms*	55 %	16 %	83 %
24 *bits*	1	67 *MHz*	0.5 *ms*	22 %	8 %	41 %
	2	65 *MHz*	0.3 *ms*	45 %	16 %	83 %
16 *bits*	1	79 *MHz*	0.5 *ms*	16 %	8 %	10 %
	6	76 *MHz*	78 μs	97 %	50 %	62 %

Table 4.2: Mapping on Virtex II device (XC2V3000). The left part of the table shows the system's structural and performance–related parameters versus the word length. The right part shows the resource occupation versus the word length. The T_{obs} values are computed assuming a model set with $L = 39$ and $M = 1$.

Word Length	N_{HMMP}	f_{MAX} (clock)	T_{OBS} (min)	Occupied Slices	RAMB Used	DSP48 Slices (18x18b Multipliers)
32 *bits*	1	77 *MHz*	0.5 *ms*	14 %	2 %	7 %
	7	74 *MHz*	66 μs	99 %	17 %	54 %
24 *bits*	1	75 *MHz*	0.5 *ms*	11 %	2 %	7 %
	9	70 *MHz*	53 μs	99 %	22 %	70 %
16 *bits*	1	108 *MHz*	0.3 *ms*	16 %	8 %	10 %
	11	108 *MHz*	30 μs	99 %	27 %	21 %

Table 4.3: Mapping on Virtex 4 device (XC4VSX55). Again, the T_{obs} values refer to a model set with $L = 39$ and $M = 1$.

number of embedded multipliers is really high (refer to Table 4.1, Page 88) the constraint for the maximum number of HMM processors in a single device is the quantity of logic slices. Figure 4.22 shows some snapshots of the various configurations mapped. The configuration that makes better use of the available hardware resources is the array of 9 HMM processors with a precision of 24 *bits*, so the ideal application to be run onto this device is one that needs a high number of parallel processors with a medium precision of the numeric representation.

All of the implementation solutions shown in this section have been tested in order to insure the intended functionality and behavior. The next section will provide the details about how this functional verification has been done.

4.3.2 Functional Verification

The testing of the implementation described in Section 4.2 was mainly aimed to verify the functionality of the HMM decoding process. To do this, it is necessary to have a reference system capable of performing the same kind of processing, in order to compare the results. The chosen system was the open source toolkit HTK [82], that contains, among its many tools, a

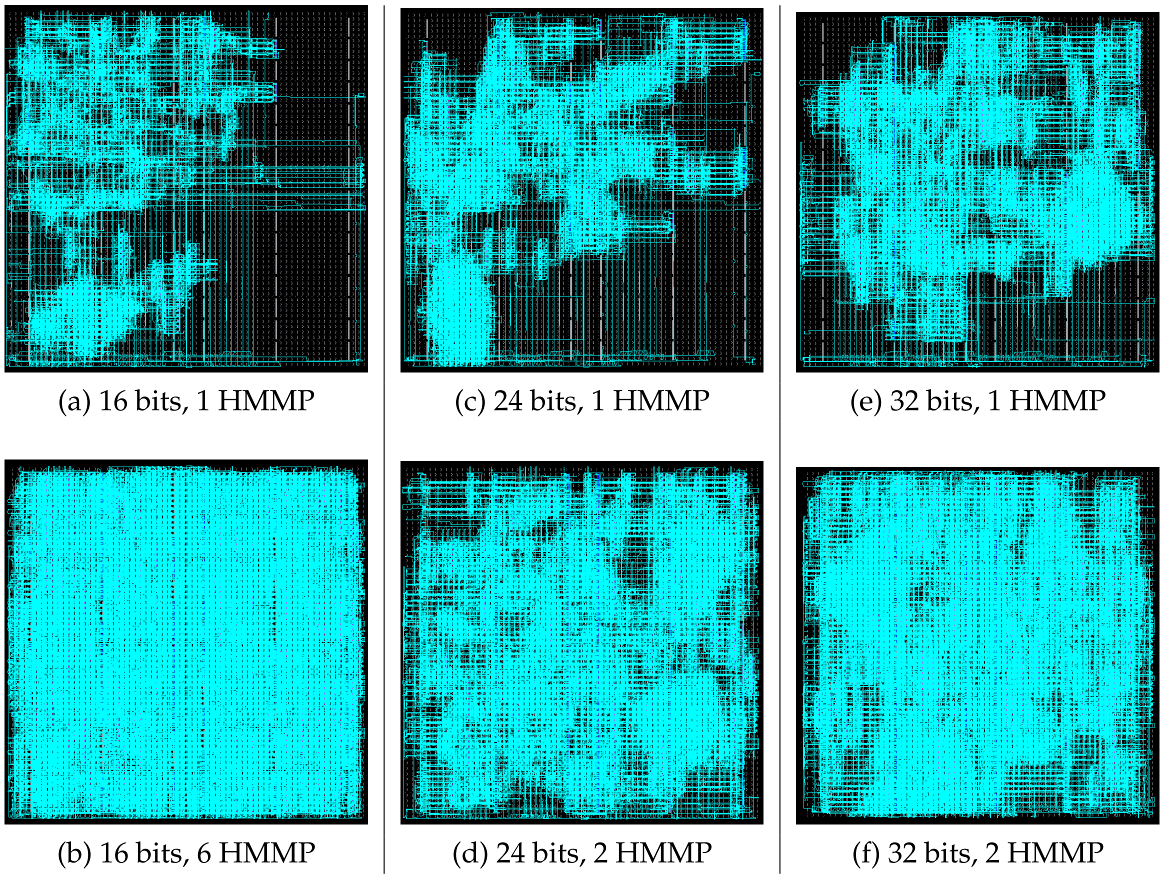
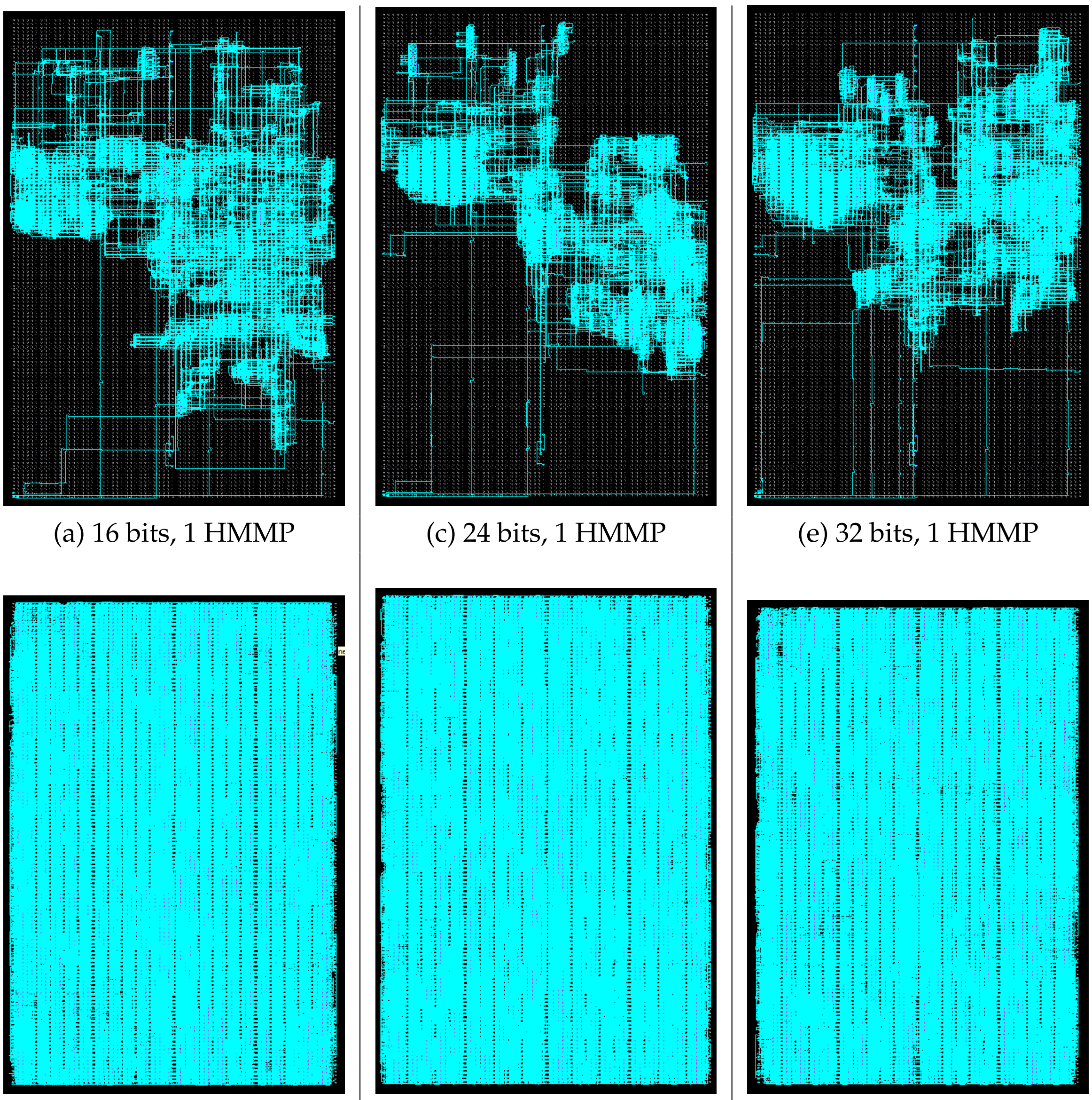

(a) 16 bits, 1 HMMP
(c) 24 bits, 1 HMMP
(e) 32 bits, 1 HMMP
(b) 16 bits, 6 HMMP
(d) 24 bits, 2 HMMP
(f) 32 bits, 2 HMMP

Figure 4.21: Different configurations of the HMM Processor array placed and routed onto a Virtex II 3000.

fully configurable HMM decoder. The HTK is structured as a set of command–line tools that allow to perform all the operations related to the design and analysis of a system based on HMMs (See Figure 4.3.2). Many of the tools are specific to the speech recognition problem, nonetheless the toolkit can be used to model any kind of HMM–based system.

For this work, the tools used are the ones for data preparation (tools `HCopy`, `HList` and `HLStats`), the ones for model training (tools `HCompV` and `HERest`), the Viterbi decoder (tool `HVite`) and the one to output results (tool `HResults`). The overall processing flow for HTK is shown in Figure 4.3.2. Detailed informations on the HTK tools are not going to be provided here. For further details refer to the *HTKBook* [59].

Several model-sets have been trained using HTK in order to test the architecture. The source signals were from the two applications described later in more detail in Chapter 5: an EEG signal classification task and a speech recognition task. The two tasks represent two problems that must be handled in a totally different way from the point of view of hardware resources utilization.


(a) 16 bits, 1 HMMP (c) 24 bits, 1 HMMP (e) 32 bits, 1 HMMP

(b) 16 bits, 11 HMMP (d) 24 bits, 9 HMMP (f) 32 bits, 7 HMMP

Figure 4.22: Different configurations of the HMM Processor array placed and routed onto a Virtex 4 SVX55.

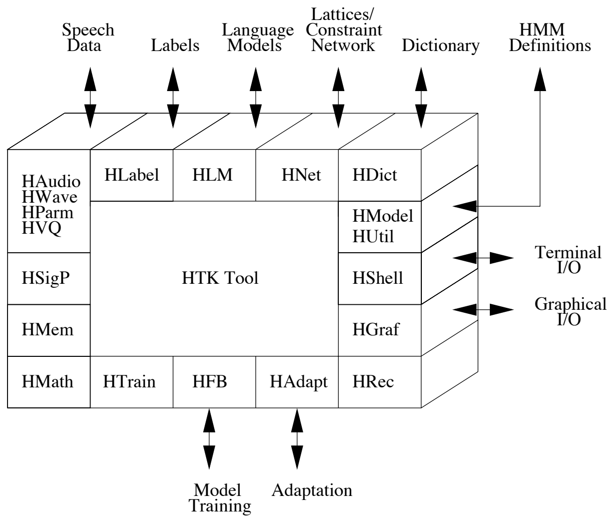


Figure 4.23: Software architecture of the HTK Toolkit.

The first tests were made on automatic speech recognition(ASR). The speech recognition task taken into account was actually the implementation of a phonetic recognizer. The goal is to correctly split a continuous speech sequence and to label each phonetic sound. The model-set contains a number N_A of HMMs around 50 that can be processed in several passes per time step, using always the same observation vector, and switching models into the same HMM processors. Since the computations must be as precise as possible, the highest possible number of bits should be used for data representation, thus requiring more FPGA resources and allowing for less processors on a single device. This translates into being forced to use a multi–pass strategy. The model-set was trained on a randomly chosen half of a speech database. The other half was used for testing. The recognition of the test set has been done with both `HVite` and a HMM processor device. The configuration of the hardware device was done according to application–specific needs. In particular, in this application, the observation rate is such that $T_{obs} = 10\ ms$. Also, the models make use of mixtures with $M = 4$ gaussian components for emission probability PDFs. Referring to the plot on the right in Figure 4.20 (Page 91) it is clear that any configuration capable of running at a minimum clock frequency of 20 MHz would be suitable for the application. Since, as it can be seen in Table 4.2 and 4.3 (Pages 92), all the configurations can reach that working frequency, all of them have been tested on the application. The Virtex II configurations were tested on the *Xilinx HW-AFX-FF1152-200 Proto Board* (whose architectural diagram is shown in Figure 4.25 for reference) while the Virtex 4 configurations were tested through simulation of a *post–place and route* (black–box) simulation model automatically generated with the IDE software *Xilinx ISE*. Given the word length, all the configurations produced the same behavior (with longer idle time for the faster ones). As expected, a slight decrease in accuracy was observed when decreasing the representation precision. The detailed results can be found in Section 5.1.2.

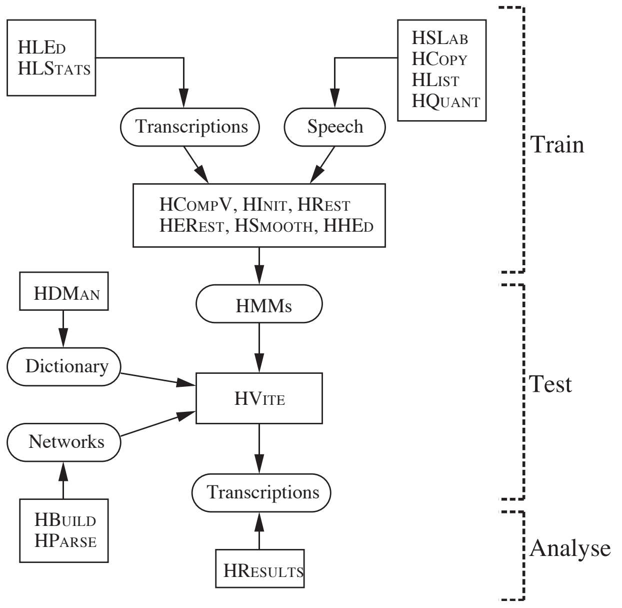


Figure 4.24: HTK processing stages, along with the tools involved in each stage.

The Brain–Computer Interface task is a binary classifier intended to discriminate between a command and a no–command pattern in a multi–channel electroencephalographic recording. The task relies on information taken from a high number of channels ($N_A \cong 60$ in the tests shown here). So a couple of HMMs was trained for each of the channels (one for each of the two classes to be spotted), providing a model–set of about 120 HMMs. It is clear that such a number of HMMs will never fit in an single hardware parallel array (and as I will show in a while, there is no need to do that) so also in this case a multi–pass technique must be applied. Since the sampling frequency of electroencephalogram (EEG) is of few hundreds Hertz (far less than the typical 16 *KHz* of recorded speech signals), the observation period will be larger than in the speech recognition task, in order to collect in a single observation vector enough samples (thus enough information). In the tests made in this work, the EEG was sampled at 256 *Hz* and a front–end was used with an observation period $T_{obs} = 10\ ms$. Even with such a high number of HMMs to be processed in each frame, the observation rate is low enough that any of the shown hardware configurations could be able to do the processing with no effort. On the other hand, EEG preprocessing could be really computationally intensive because advanced techniques, like independent component analysis for example, are often used. Moreover, one of the most interesting applications of BCI are interface systems for impaired people: in this context, reducing power consumption for portability is fundamental, and can be achieved also by scaling down the systems' clock frequency. Given these considerations, all of the configurations showed in the previous section could be interesting for this purpose: the single–processor ones because they are small enough to be used in really compact systems, and the parallel ones because they can accomplish the task required working at really low frequencies (below

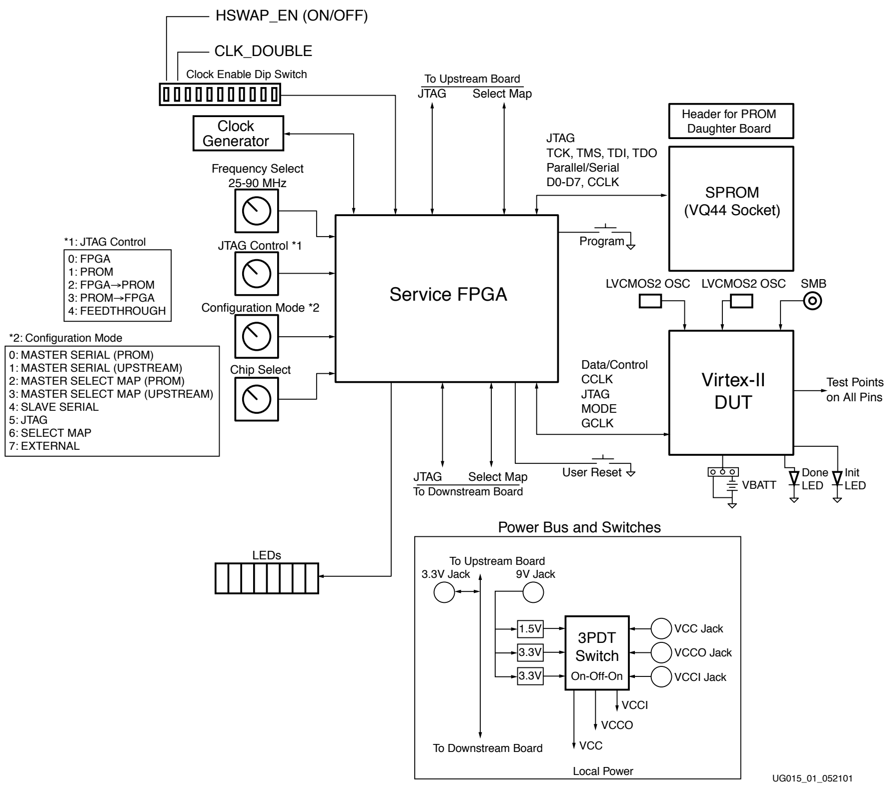


Figure 4.25: Diagram of the Xilinx Virtex II Proto Board used for the implementation tests (HW-AFX-FF1152-200).

5 *MHz* with a parallel array of 11 HMM processors) with lower power consumption. As for the speech recognition application, the Virtex II configurations were tested on the Xilinx Virtex II Proto Board, and the Virtex 4 configurations were tested through simulation of a *post–place and route* simulation model. Also for the BCI task, all the configurations were found functionally equivalent with a slight decrease in accuracy towards lower word lengths. Detailed results are presented in Chapter 5.2.2.

A last thing worth saying is that in both of the applications taken into account for the functional verification the HMM Processors array is only a part of the system. Both the application need also a signal processing front–end (from signal to feature, like the ones shown in Figure 5.2, Page 105, and Figure 5.18, Page 128) and post–processing back–end (from lattice to result, not treated in this work). In speech recognition the front–end is quite standard signal processing while for BCI that part can reach high levels of complexity (this needing an important computational effort). On the other hand the back–end is for both the applications a complex, time consuming and computationally expensive task because it should translate raw data from the classifier into information with a high level of abstraction (for example a sentence in ASR and a sequence of commands in BCI). From these considerations it is clear how the HMM Processor array cannot use all the observation period T_{obs} to process an observation, but should be faster in order to leave a certain time margin to allow for all the other operations to be performed. The optimal processing time for the HMM Processor array depends on the application and on the specific characteristics of the front–end and back–end implementations. In particular in a simple processing framework it would be

$$T_{obs} \geq T_{fe} + T_{hmm} \tag{4.9}$$

where T_{fe} is the processing time of the front–end and T_{hmm} the processing time of the HMM Processor array. The back–end processing time T_{be} has not been taken into account since in most cases it processes several items output from the front–end/classifier chain in order to provide an output item itself. The time taken by the back–end to provide the output is usually tuned on the application level of intended user interactivity, but will not be treated here since it is outside the aims of this work. As an alternative the three parts of the system can be seen as cascaded blocks, thus allowing for a pipeline. In that case the constraint is

$$T_{max} = max(T_{fe}, T_{hmm}) \leq T_{obs} \tag{4.10}$$

meaning again that the HMM Processors array's processing time must be just below the observation period.

The next chapter will show how all the considerations made here have been applied to real systems.

Chapter 5

Selected applications for human-machine interfaces

As already explained in the previous chapters, the HMMs are suitable for many pattern recognition tasks that can be useful in the context of a multimodal human–machine interface. Of course a multimodal interface requires an extremely complex high level control in order to collect the data from all the various input channels and merge them into actual control commands, but to grant the necessary machine–user interactivity, also all the interface elements should perform their task quickly and efficiently to avoid slowdowns that can affect all the system. It is not realistic to think that many highly complex recognition tasks *and* the overall interface control can be run on a single computer, even if really powerful. The introduction of an hardware HMM engine like the one described in the previous chapters could provide the means to build the most complex and computationally demanding elements of the interface in an inexpensive way.

To show that the described architecture can be actually applied to problems of interest, two application that are likely to be part of a multimodal interface have been chosen.

The first application is automatic speech recognition (ASR). Since speech is among the most used forms of interaction between humans, it is obvious that it would be one of the most promising among the advanced techniques to be used in a multimodal interface. Research on this application is still ongoing, and good results have been obtained. The main problem is that the complexity of the system grows rapidly when trying to provide an interface that allows the user to speak in a natural way. Techniques are known to deal with the problem, but real time processing is far from being achieved with software solutions, because of extremely high computational needs. All the details on the approach used to deal with this application are provided in Section 5.1

The second application is brain–computer interface (BCI). Is known that biological signals are a huge source of informations on the state and attitude of the user. The interesting thing is that all these informations are provided unconsciously. Moreover it has been shown that biological signals can be also modulated in a conscious way. The advantage in using this kind of input is that, with proper training of the user, it would require a minimum level of attention, and allow for other tasks to be performed by the user himself. Research on BCI is still at the beginning, but encouraging results have already been obtained with Electroencephalogram (EEG). The main problems are given by high levels of noise in the signal, mixing of the signal sources and limited knowledge on the mechanisms that produce the signal itself. The use of biological signals in interfaces is thus still a challenge. Details on the work I did on BCI during my research, can be found in 5.2.

5.1 Applications to speech recognition

5.1.1 Introduction to ASR

Automatic speech recognition is formally the conversion of speech, sampled as audio waveforms, into the corresponding text. The algorithmic framework used in speech recognition is not new anymore [10]. A great effort has been made in the past years in order to develop the necessary technologies. Nonetheless all the ASR systems available nowadays provide only basic functions, a limited vocabulary and, usually, the need for some adaptation or training time for each new user. In fact ASR is actually used only as a side option in commercial devices and services, because its capabilities and its reliability still do not allow for it to be the main mean of communication between the user and the system.

There are many classes of speech recognition, and whether a speech recognizer belongs to a class or another depends on a set of fundamental parameters that are used to provide the most relevant features.

Speaking mode. There are two main modes of speech production when talking about ASR systems: *continuous speech* and *isolated word*. In continuous speech recognition the user can speak fluently, as in normal speech production, while in the other case he is forced to put pauses between words in order to make the recognizer react as expected. Of course continuous speech provides the most natural way of communication, at the expense of higher computational needs.

Speaking style. According to the application the user will be requested to speech in a certain way. Application for text dictation in word processors, for example, can expect that the speaker pronounces sentences with a regular structure and words that are suitable for a written text. The opposite situation is represented by a control interface that allows the user to provide commands using natural speaking. In that case the sentences would be fragmented, not always grammatically correct, and there could be words taken from jargon and sounds produced in moment of hesitation. All these factor must be taken into account and introduce a high level of complexity in the process of sentence building and meaning extraction.

User adaptation. In most of the cases a speech recognizer needs some specific training for each new user, and must know which user is interacting with the system in each moment (speaker dependentness). This means that before first use, the speaker must read some predefined text so that the system spots the main characteristics of his voice. To make the training procedure faster, sometimes the recognizer just *adapts* some predefined models to the user's voice. This needs less training text to be spoken, and less computations. In opposition to the *speaker dependent* systems there are the *speaker independent* systems. These systems are able to recognize any user without any specific training. It is clear that this last kind of system can find many application in interrogation of publicly accessible databases, for example. However the training of speaker independent systems is more difficult and needs huge amounts of accurately selected training data. Furthermore the models are always less robust than speaker dependent ones, so that also the effects of noise are stronger.

Vocabulary size. The size of the vocabulary is strictly tied to the recognizer's target application. Small dictionaries can be used for command and control applications, while large ones are needed for dictation or smooth interaction. Anyway large vocabularies make the system's computational demand explode and require substantial storage capacity, thus they can be used only in systems that run on powerful platforms.

Grammar. The presence or absence of a set of grammar rules usually depends on the size of the dictionary or some task specific needs. Grammar rules put constraints on the possible sequences of words thus reducing the overall complexity in the production of the output recognition hypothesis. A grammar could be of several types, but the most common in complex tasks are statistical grammars and sets of rules based on the language to be modeled.

Task perplexity. The perplexity actually measures the hardness of the recognition task. Let us assume that a recognizer has spotted a certain word inside a sentence. The perplexity is defined as the mean number of words that, given a vocabulary and a grammar, can be considered as possible successors of the current one. It is important to note that the perplexity is a useful measure only in presence of a grammar. When there is no grammar, the possible successors of any word are simply *all* the words in the dictionary.

Noise level. The level of noise in the input signal directly affects the recognition performance. The signal processing front–end is in charge of dealing with the noise, so the expected noise level is an important factor to take into account when designing the whole system. In some cases environment models are used to identify the scenario and better isolate the signal from the environmental background.

Sampling quality. The channel through which the speech signal can reach the recognizers are many. Different approaches should be adopted whether the input is taken from the phone line, from a high performance microphone or any other mean. Since the aim is to make speech recognizers usable in any environment and condition, it is common to assume that commercial level microphones are used. This clearly reflects in the need of a further design effort in order to increase system's robustness.

Parameter	Possible Parameter Range
Speech Mode	Isolated Words $\rightarrow$ Continuous Speech
Speaking Style	Dictation $\rightarrow$ Spontaneous Speech
User Adaptation	Speaker–Dependent $\rightarrow$ Speaker–Independent
Vocabulary Size	Small ($W < 100$ words) $\rightarrow$ Large ($W > 10000$ words)
Grammar	None $\rightarrow$ Rule Set $\rightarrow$ Statistical
Perplexity	Low ($PP < 10$) $\rightarrow$ High ($PP > 100$)
SNR	High ($> 30dB$) $\rightarrow$ Low ($< 10dB$)
Sampling Quality	Telephone Band $\rightarrow$ High Fidelity

Table 5.1: Parameters that characterize a generic speech recognition system. Typical range is shown (from low complexity to high complexity).

A summary of the speech recognition characterizing parameters is shown in Table 5.1 along with some typical values (ordered from smallest to highest design effort needed to implement the feature).

The diagram of a generic automatic speech recognition system is shown in Figure 5.1. The front–end is in charge of extracting from the speech signal the feature that are most likely to carry informations on the meaning, discarding as much as possible all the informations related to noisy factors like background sounds, mood and state of the speaker, intonation[1], dialectal inflexions.

The feature vectors produced by the front–end are fed into the low level classifier. Since the speech signal is highly variable and the mechanisms of language production really complex,

[1] Some advanced techniques try to extract in a parallel feature channel, also the informations given by the intonation (referred to as *prosody*), in order to use them when reconstructing the sentence (for example, knowing if a sentence is interrogative or not gives a cue on the kind of syntactical constructions that are more likely to have been used).

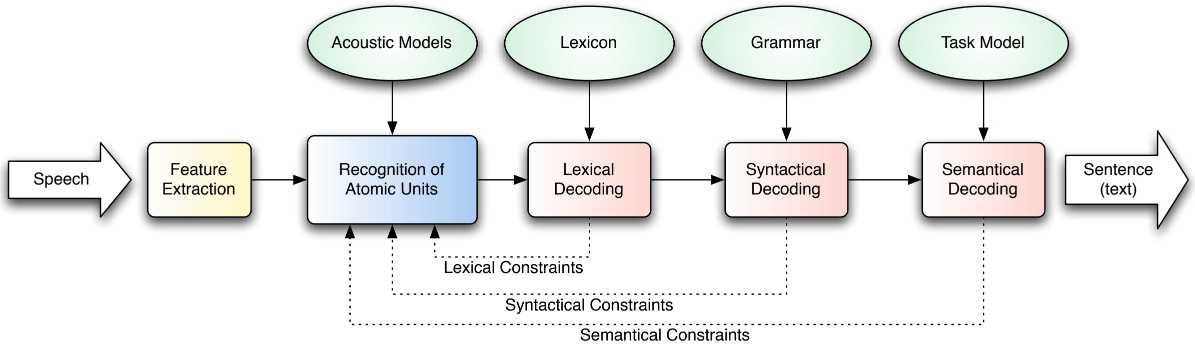


Figure 5.1: Representation of the main components of a generic ASR system.

the classifier is always a statistical one. Statistical classifiers can be trained on real data and take automatically into account complex phenomena that are too difficult to model explicitly. The low level classifier uses a set of acoustic models to spot the atomic units (chosen as the base building blocks of speech) into the input stream. The choice of an atomic unit for speech is another crucial factor and depends strongly on the target application. In simple systems with small vocabularies, it is possible to choose the *word* as the base unit. This implies the training of a model for each of the words in the dictionary. Word based systems are very simple because they totally remove the lexical processing stage. On the other hand it becomes impossible to use this approach as soon as the vocabulary grows too much, either because too much training data would be necessary, and because too many models should be evaluated concurrently. The solution when dealing with large vocabularies is to use *phonemes* as atomic units. Phonemes are recognized also by the human brain as the building blocks of speech [85] and phonetic alphabets have been identified for all the major spoken languages [66]. It has also been shown that all the sounds in almost every language can be reproduced through a set of 50 or 60 phonemes. This allows to build an acoustic model set containing a reasonable amount of models, recognize the phoneme sequences, and leave the task of building words out of phoneme sequences to a dedicated part of the system. Since they are the most powerful and flexible, from now on I will refer only to phoneme–based ASR systems. The output of the phonetic recognizer is one (or more, if keeping track of multiple hypotheses) phonetic sequences.

After the phonetic recognition, the next task is to aggregate phoneme sequences in order to build words. This task is called *lexical decoding*, and makes use of a lexical model known also as pronunciation dictionary. The pronunciation dictionary provides the phonetic sequence needed to build every word present in the dictionary. Given this information it is possible to check whether a certain phonetic sequence corresponds to an existing word or, if not (maybe because of some errors in the phonetic recognition), which is the word that most resembles the phonetic sequence. So the lexical decoding can have the capability of recovering some of the eventual error introduced by the phonetic recognizer. The output of the lexical decoder should be one or more sequences of words.

In some application, like command and control interfaces, word recognition is enough for the needs of the interface. In all the other cases a further step must be performed, that is the grouping of sequences of words into separate sentences. This is known as *syntactical decoding*. The syntactical decoder makes use of a grammar to look in a stream of words for sequences that are compatible with the specified grammatical rules. The output of the syntactical decoder is a set of grammatically correct sentences.

The final step to be performed in order to make the output of the ASR system usable in a real interface, is the *semantic decoding*. The semantic decoding is a highly complex task because it associates words or sentences (according to the kind of speech recognizer) to an actual command or action to be performed. This association is made thanks to a set of rules that specifies the actual task to be performed.

In most of the practical situation is preferable to apply the lexical, grammatical and (less frequently) syntactical constraints, directly during the phoneme recognition process. These constraints reduce the perplexity of the task and allow for a dramatic reduction of the error, mainly because it becomes less probable to produce phonetic sequences that do not match the ones allowed by the lexical and grammatical rules. Usually, when applying constraints in this way, keeping track of multiple recognition hypotheses can greatly improve the recognition accuracy, at the expense of course of heavier computational needs.

Automatic speech recognition has been chosen to be studied in this work mainly because most of the ASR systems already developed are based on HMM classifiers [86, 87, 88]. This allows us to test the hardware architecture described in Chapter 3 directly on an algorithmic framework that has already been accurately verified. Furthermore this thesis focuses on the implementation of advanced speech recognizers, that is the ones that are still beyond the state of the art. In particular from now on, whenever talking about ASR, I will refer to real–time continuous–speech speaker–independent recognition with large vocabulary. Systems that belong to this class are also called *natural language* speech recognizers, because the user can speak to the interface in the same way he speaks to any other person.

Up to now, all the known and working implementations of ASR systems are, basically, software applications. It has been shown [15] that to obtain real–time continuous-speech speaker–independent systems through software implementations on commercial platforms, it is necessary to reduce dictionary size or simplify the task to avoid unacceptable degradation of the accuracy. The only way to obtain really accurate software implementations of natural language speech recognizers, is to run the system in batch mode because of the really long processing time. So, with the software approach, the choice seem to be between real natural language ASR and real time operation.

The alternative proposed here is to make use of dedicated hardware in order to speed up the accurate software batch systems. Since the processing core of the class of ASR systems taken into consideration is the HMM classifier, the hardware architecture described in the previous chapters seems to be a suitable solution. The introduction of hardware acceleration in ASR systems would therefore allow to develop highly accurate real time natural language speech recognizers, but also ASR systems with a good level of complexity (even if not natural language) that can be run on low processing power platform like portable devices.

5.1.2 Implementation of a phonetic recognizer

The aim of this section is to show how the hardware architecture described in the previous chapters can be applied to a real implementation of a speech recognizer. Of course the implementation of a whole speech recognizer is an extremely complex task and it is beyond the goals of this thesis.

A choice has been made in order to perform a meaningful test without the need to build a really advanced system from scratch. The chosen ASR related task was the implementation of a *phonetic recognizer*. Referring to Figure 5.1 (Page 102) the part of the system I decided to implement is the one identified as "recognition of atomic units". As already noted, a phonetic recognizer can be used as a stand–alone device, and its output phoneme sequences can

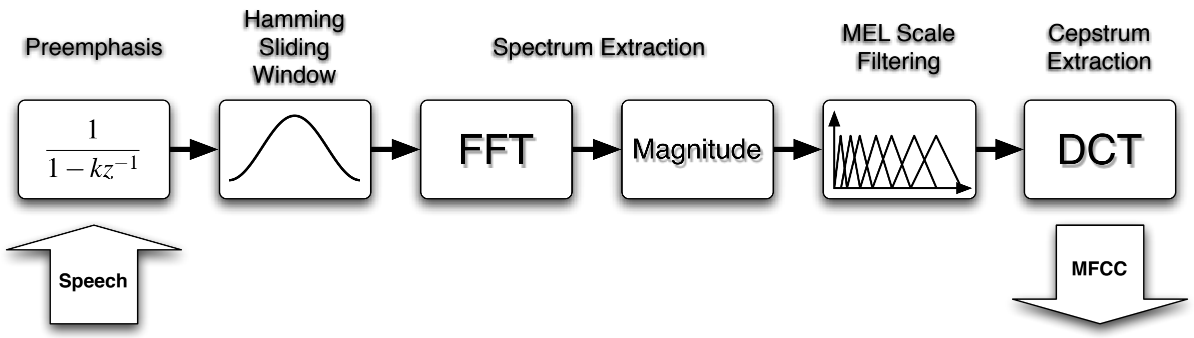


Figure 5.2: Block diagram of a MFCC front–end for speech recognition.

be processed at a later stage with higher level rules, or it can be interconnected with the other cascaded blocks in order to receive further constraints to be applied during the phonetic recognition task itself.

The HTK's Viterbi decoder, `HVite`, was configured as a phonetic recognizer in order to provide a suitable reference system. This was done by using a dictionary containing one entry for each phoneme in the model set and a grammar that allows for any word sequence to be produced. In that way the output of `HVite` will be an unconstrained sequence of words that exactly matches the phoneme sequence produced by the low level decoder.

A model set was trained using a database of 200 isolated italian words spoken by 13 different speakers (7 males and 6 females), so 2600 isolated words in total. Each word was recorded as a single channel PCM waveform with a sampling frequency of 16 *KHz*. The *Sampa* computer readable phonetic alphabet for italian [66] was used as the phonetic set (see Table 5.2 for details). This phonetic alphabet contains 43 sounds for consonants (29 obstruents and 14 sonorants) and 7 vocalic sounds for a total of 50 phonemes, each of them modeled by a HMMs.

The front–end signal processing was done offline and the feature vectors stored in order to speed up the tests. The HTK tool `HCopy` was used as the offline signal processing front–end. The tool was set up to extract *Mel Frequency Cepstral Coefficients* (MFCC) from the input speech. This kind of signal processing is quite common when treating speech for ASR systems and it was shown to be one of the most effective [89, 90]. A diagram of the MFCC front–end is shown in Figure 5.2. First of all a pre-emphasis filter is applied to the speech signal. This is done by applying the first order difference equation

$$s'_n = s_n - ks_{n-1} \tag{5.1}$$

where s_n is the generic n^{th} audio sample, and k is the pre-emphasis coefficient that, in the tests shown here, has been chosen to be $k = 0.97$ and, in general, is always in the range $0 \leq k \leq 1$.

The pre-emphasized signal is then split into overlapped windows and each window boundary is smoothed through sample–by–sample multiplication with an Hamming function defined

Symbol	Example	
	Word	Transcription
Consonants		
Plosives		
p	**pane**	**p**ane
b	**banco**	**b**anko
t	**tana**	**t**ana
d	**danno**	**d**anno
k	**cane**	**k**ane
g	**gamba**	**g**amba
pp	**coppa**	kO**pp**a
bb	**gobba**	gO**bb**a
tt	**zitto**	tsi**tt**o
dd	**cadde**	ka**dd**e
kk	**nocca**	nO**kk**a
gg	**fugga**	fu**gg**a
Affricates		
ts	**zitto**	**ts**itto
dz	**zona**	**dz**Ona
tS	**cena**	**tS**ena
dZ	**gita**	**dZ**ita
tts	**bozza**	bO**tts**a
ddz	**mezzo**	mE**ddz**o
ttS	**braccio**	bra**ttS**o
ddZ	**oggi**	O**ddZ**i
Fricatives		
f	**fame**	**f**ame
v	**vano**	**v**ano
s	**sano**	**s**ano
z	**sbaglio**	**z**baLLo
S	**scendo**	**S**endo
ff	**beffa**	bE**ff**a
Fricatives (Cont.)		
vv	**bevvi**	be**vv**i
ss	**cassa**	ca**ss**a
SS	**ascia**	a**SS**a
Nasals		
m	**molla**	**m**Olla
n	**nocca**	**n**Okka
J	**gnocco**	**J**Okko
mm	**grammo**	gra**mm**o
nn	**panna**	pa**nn**a
JJ	**bagno**	ba**JJ**o
Liquid		
r	**rete**	**r**ete
l	**lama**	**l**ama
L	**gli**	**L**i
rr	**ferro**	fE**rr**o
ll	**colla**	kO**ll**a
LL	**foglia**	fO**LL**a
Semivowels		
j	**ieri**	**j**Eri
w	**uomo**	**w**Omo
Vowels		
i	**mite**	m**i**te
e	**rete**	ret**e**
E	**meta**	m**E**ta
a	**rata**	rat**a**
O	**moto**	m**O**to
o	**dove**	d**o**ve
u	**muto**	m**u**to

Table 5.2: The *Sampa* phonetic alphabet for the italian language. Each phoneme symbol is shown along with an example word and its transcription. Phonemes are grouped into phonetic classes.

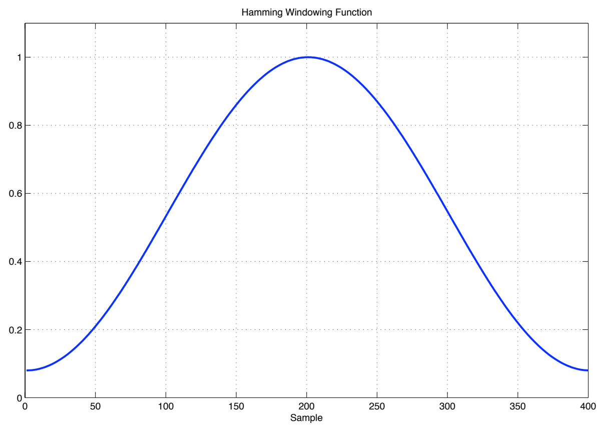


Figure 5.3: Hamming windowing function used in the front–end.

as

$$H_n = 0.54 - 0.46 \cos \left[2\pi \frac{n-1}{N-1} \right] \tag{5.2}$$

where N is the arbitrary number of samples in the window. The hamming function is shown in Figure 5.3. In the tests made, the sliding window was 25 ms wide with an overlap of 10 ms (400 samples per window with a 160 sample overlap at 16 KHz). So the output of the sliding window stage was a stream of vectors containing 400 samples each, with a rate of one vector every 10 ms.

After windowing, the magnitude of the Fourier transform is extracted. This is done through FFT (512 points in the tests) and amplitude computation. The output of the magnitude extraction stage was a vector of 256 magnitude values for each input window.

The next processing stage is the application of a bank of N_M triangular filters evenly distributed along the MEL frequency scale. This frequency scale is used because it reproduces the frequency resolution of the human ear, so it is likely to be effective in speech recognition. The MEL scale is defined as

$$MEL(f) = 2595 \log_{10} \left(1 + \frac{f}{700} \right) \tag{5.3}$$

and an example MEL filter bank is shown in Figure 5.4. The filter bank is applied to the FFT magnitude vectors by multiplying each filter coefficient with the corresponding magnitude value, and summing together the results for each of the filters. The results is a vector containing one value m_j for each of the triangular filters in the array. In the tests $N_M = 26$ triangular filters were used, distributed from zero frequency to the Nyquist frequency. Thus the output of the filter bank was a vector of 26 elements for each input FFT magnitude vector.

The final step in order to obtain the actual cepstral coefficients is to apply the cosine transform to the filter bank's output. This is done by computing a Discrete Cosine Transform, defined as

$$c_i = \sqrt{\frac{2}{N_M}} \sum_{j=1}^{N_M} m_j \cos\left[\frac{\pi i}{N_M}(j-0.5)\right] \tag{5.4}$$

where $0 \le i \le N_C$ and N_C is an arbitrarily chosen number of cepstral coefficients. The DCT allows to remove the high intercorrelation between the MEL coefficients, thus reducing the number of needed parameters. Moreover the coefficient c_0 is particularly useful because it is proportional the the signal's energy, so it can be used as a measure for it. In the tests a number of cepstral coefficient $N_C = 12$ was used, plus the coefficient c_0 as an energy measure.

Finally, to better catch the dynamics of the feature coefficients, first and second order differences were computed, adding two more sets of 13 coefficients to each feature vector. The difference with calculated as follows:

$$d_t = \frac{\sum_{\theta=1}^{\Theta} \theta(c_{t+\theta} - c_{t-\theta})}{2\sum_{\theta=1}^{\Theta} \theta^2} \tag{5.5}$$

where Θ is half the width of the regression window, that was set to 2 for both first and second order differences. The first and last vectors of each stream on which differences were calculated, were suitably replicated in order to allow for correct computation. The final feature vectors in the tests contained 39 coefficients each: this is the final stream width L used throughout the final system.

The feature vectors have been calculated for all the database and stored to avoid re-computation for each test. Then the database was randomly split in two parts containing respectively 10% and 90% of the data, taking care to keep in each subset an even distribution of phoneme occurrences and a male–female speaker balance the most close to 50%. The bigger part was used for training the HMMs while the smaller part was used for testing purposes. This partition was done 10 times, the tests repeated for each partition and the results obtained by averaging the outcome of each test. In this way the results are supposed to statistically more reliable.

So 10 model-sets were trained using the corresponding database partitions. The training was made by first initializing each model's parameters with average values computed from

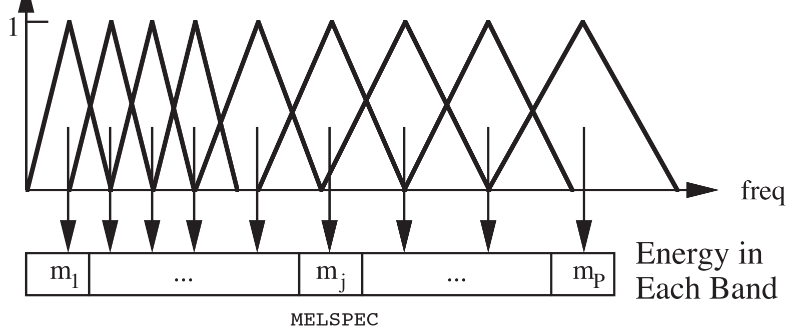


Figure 5.4: A bank of triangular filters evenly distributed along the MEL frequency scale.

the training data-set. This procedure is known as *flat–start* ad was done using the HTK tool `HCompV`. Then the model was re-estimated with the Baum–Welch iterative procedure using the HTK tool `HERest`. The number of iterations was chosen in order to obtain the best recognition performance on the test sets. The training procedure was done as follows:

1. Apply flat–start procedure and a first pass of the Baum–Welch algorithm providing as initial parameters the ones obtained from the flat–start.
2. Run the recognizer with the model obtained from a single iteration.
3. Apply another iteration of the Baum–Welch algorithm providing as initial parameters the ones from the model obtained with the last iteration.
4. Run the recognizer with the model obtained from the last iteration.
5. If the accuracy of the current model is better than the one from previous step, continue from Step 3, otherwise discard the current model, keep the one from the previous step and continue from Step 6.
6. If more components in each gaussian mixture are needed, split each gaussian component into two almost overlapping components using the HTK tool `HHEd` and continue from Step 7. Otherwise the training procedure is over.[2]
7. Apply an iteration of the Baum–Welch algorithm providing as initial parameters the ones obtained from the component splitting process and continue from Step 2.

One HMM for each of the phonemes in the Sampa phonetic alphabet shown in Figure 5.2 (Page 106) was trained, plus one HMM to model the silence (or background noise) at the beginning and at the end of each word. The correct association between training data segments and the corresponding HMM chains was made by providing a phone level transcription for every word in the training data sets. The emission PDFs were modeled as mixtures of 4 gaussian components. The HMM topology used for all the phonemes is shown in Figure 5.5 (a left–right 5–state HMM). The topology for the silence model was almost the same, with the exception of a transition from S_2 to S_4 and a transition from S_4 to S_2, introduced to allow for longer silences to be modeled.

For each of the 10 partitions of the speech database, the performance was evaluated both with `HVite` and with all the hardware configurations described in Section 4.3 (Page 87). In the hardware tests, the models' parameters and the test observation stream was preloaded onto an on–board memory. Since the phonetic recognizer requires a much simpler control with respect to a full speech recognizer, the system controller was written directly in VHDL and mapped

[2] The splitting process requires the specification of a target number of components in the mixtures. The `HHEd` tool picks the component with the highest weight and splits it by duplicating the component, dividing the weight of each by 2, and perturbing the means by 0.2σ in opposite directions. The process is repeated until the required number of mixtures is reached. It is recommendable to increase the number of component in a progressive way, always interleaving the splitting operation by a suitable number of training iterations.

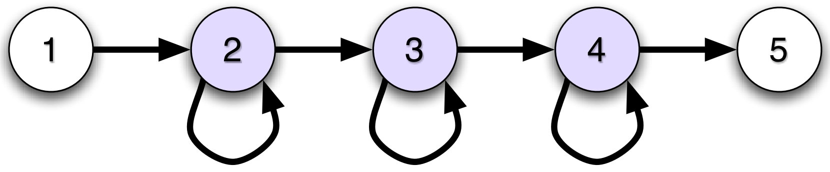


Figure 5.5: The topology of the HMM used to model each phoneme in the phonetic recognizer. The emitting states are drawn in a darker tint.

together with the rest of the design (instead of being implemented on a microcontroller, as said in Section 3.2, Page 48), in order to control model loading, to feed the observations from the memory into the system, and to provide the necessary system level control signals. The tokens output from the HMM processors in the mapped array were routed to an output bus and fed into a logic analyzer to view and store the results. It should be remember that since in the used configuration the HMM classifier is totally unconstrained, the system's output at each time step is just the phoneme corresponding to the HMM that outputs the token with the highest probability (some simple post–processing can be applied to avoid phonemes with durations too short to be realistic).

The comparison of the results obtained from `HVite` and the hardware implementation are shown in Figure 5.6. The plot reports two performance measures, averaged over the 10 trials made with the different partitions of the speech database. The first is the percentage of correctly recognized phonemes (Spotted Phoneme Percentage, SPP). This measure is computed in the following way:

$$SPP = 100\left(\frac{NP_C}{NP}\right) \tag{5.6}$$

where NP_C is the number of correctly recognized phonemes and NP is the total number of phonemes that make up the input sequence. The second measure is the *Phone Recognition Rate* (PRR). This measure takes into account also the number of errors present in the recognized sequence. It is defined as

$$PRR = 100\left(\frac{NP_C - N_E}{NP}\right) = 100\left[\frac{NP_C - (NP_I + NP_D + NP_S)}{NP}\right] \tag{5.7}$$

where the total number of errors N_E is given by the sum of *insertion* errors NP_I, *deletion* errors NP_D and *substitution* errors NP_S. It is clear that the PRR will be always less or equal to the SPP.

The comparison has been made between the results from HTK's `HVite` and the hardware architecture implemented with three different word lengths (for score computations): 32, 24 and 16 bits. As already checked in 4.3.2 (Page 92) all the configurations of the hardware architecture had exactly the same functional behavior for a given word length (i.e. they provided the same results for the same input), so the results referred to a certain word length are applicable to any HMM processor configuration using the same number of bits for number representation. Moreover, when interpreting the results, it is important to remember that the `HVite` software

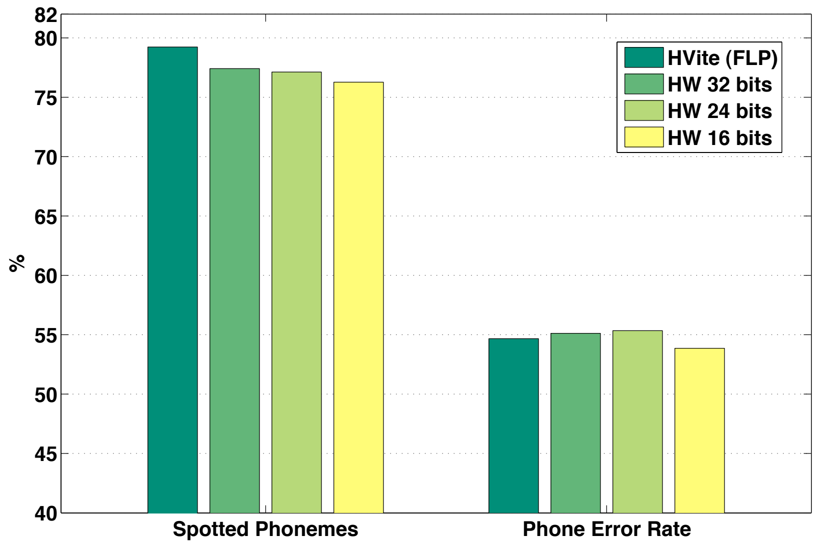


Figure 5.6: Comparison of the recognition performance (average over 10 trials): HTK's `HVite` (software, floating point arithmetic) vs. HMM processor array (hardware, fixed point). Both systems were configured as phonetic recognizers. Percentage of spotted phonemes and Phone Recognition Rate are shown. For the hardware implementation, results obtained with different word lengths are reported.

implementation makes use of a 32 bits floating point arithmetic, while all the hardware configuration use fixed point arithmetic.

From the plot in Figure 5.6 it can be seen that the percentage of recognized phonemes from the hardware system is really close to the one given by the software decoder, and the mismatch seems to be due to the quantization introduced by the fixed point representation. As expected the accuracy decreases as the precision is reduced but, even with 16 bits of precision the accuracy is still close to the one provided by *HVite*. On the other hand the results on the phone recognition rate are quite surprising. The PRR of the 32 bits and 24 bits versions of the hardware recognizer is higher than the one provided by `HVite`. This means that even if the number of correctly spotted phonemes is higher for the software recognizer, the hardware version introduces a lower number of errors, thus providing a better overall result. Actually in many of the tests made during the development of this work, it was a common result that the introduction of a small amount of quantization in statistical classifiers improved robustness to noise and could lead to better performance. Clearly, as can be seen from the PRR of the 16 bits version of the hardware recognizer, too much quantization produces again a degradation in performance.

As a final consideration, it can be noticed that even the HTK's recognizer's performance is really low. This is obviously due to the fact that `HVite` is designed to give the best performance through the use of a dictionary and a grammar, and by keeping track of multiple recognition hypotheses during the decoding process. Some tests have been made by providing all of these elements (thus making `HVite` work as a full speech recognizer), and the result obtained was a *Word Error Rate*[3] of 99.8%. The little amount of errors is probably due to a slight model under-training, but with a bigger speech database the HTK recognizer has been proven to easily reach an accuracy of 100%. The purpose of this test was to show that the level of accuracy reached by the hardware architecture (really close to the one given by `HVite` in phonetic recognition mode) is likely to be suitable to obtain a really high accuracy whenever the support of lexical and syntactical constraints is added. However, as already said, the implementation of a whole ASR systems goes out of the subject treated in this thesis and will be left to future developments.

5.2 Applications to brain-computer interfaces

5.2.1 Introduction to Brain–Computer Interfaces

Research on *Brain–Computer Interfaces* (BCIs) has been going on for more than 30 years, but from the mid-1990s there has been a dramatic increase in working experimental systems. Years of animal experimentation have produced early working implants in humans designed to restore damaged hearing, sight and movement. The common thread throughout the research is the remarkable cortical plasticity of the brain, which often adapts to interaction with external devices. With recent advances in technology and knowledge, leading edge research could now conceivably attempt to produce BCIs that augment human functions rather than simply restoring them. However referring to BCI still does not identify a precise and well defined subject. Since the related techniques are still developing, different frameworks and approaches are continuously proposed as the research on the field goes on. However the common factor is always the same: taking signals from the brain of human or animal subjects, and process the recordings in order to transform them in informations that can be used to drive a system's interface. Also the opposite is a matter of research, that is exploiting the knowledge on the brain in order to provide signals that can be understood by it, thus creating a working communication channel from a machine to the nervous system of a living organism. Since this last topic is more related to life functions support than to human–machine interfaces, from now on I will refer only to the communication from the brain to the machine.

The main factors that characterize the different approaches to BCI are the subjects and the way the neural signal is acquired. Two are the main ways to record a signal from the brain: through invasive measures and through non–invasive measures. Usually the invasive measures are done only on non–human subjects because they imply some risk for the health of the subject. Anyway there are some situations in which invasive recordings are made also on human subjects.

[3] The word error rate (WER) is defined as the ratio between the number of correctly recognized words to the total number of words fed into the recognizer.

The most commons invasive technique are the *electrocorticogram* (ECoG) and the implant of electrodes.

ECoG: The electrocorticography is done by placing arrays of electrodes directly on the cerebral cortex, and allows to record really clear signals directly from single neurons or small groups of them. The biggest drawback to ECoG is the requirement of surgery in order to place the electrodes under the skull, directly onto the brain's surface, with all the risk factors that this procedure implies. In fact ECoG recordings on human subject are done almost only when brain surgery must be performed to cure some kind of pathologies (ECoG monitoring is usually done on epileptic patients). Furthermore the duration of ECoG recording is limited by the reaction of the living organisms to the presence of the electrodes (rejection or scar–tissue build–up).

Implanted Electrodes: The implant of electrodes is done to monitor specific deep regions of the brain, with a resolution that can reach the single neuron. The invasiveness of this approach is even higher than in the ECoG recordings because of the presence of wires in the brain along with the electrode itself (with the same problems of biological tolerance), but the signal has an extremely high SNR and spatial resolution.

On the opposite side there are the non–invasive techniques. The most widely used for BCI are the *Magnetoencephalography* (MEG), the *Functional Magnetic Resonance Imaging* (fMRI) and the *Electroencephalography* (EEG).

EEG: Electroencephalography is the neurophysiologic measurement of the electrical activity of the brain by recording from electrodes placed on the scalp. The resulting traces are known as an electroencephalogram and represent an electrical signal (post-synaptic potentials) from a large number of neurons. The actual measurements are voltage differences between different parts of the brain. The EEG is capable of detecting changes in electrical activity in the brain with a sub–millisecond resolution. It is one of the few techniques available that has such high temporal resolution, together with MEG, but it is the only non–invasive technique that allow to directly measure brain activity. On the other hand it has poor spatial resolution, provides a highly noisy signal, and it is really difficult to recover the original sources from the measured signals.

MEG: The MEG is an imaging technique used to measure the magnetic fields produced by electrical activity in the brain via extremely sensitive devices such as SQUIDs[4]. MEG signals derive from the net effect of ionic currents flowing in the dendrites of neurons during synaptic transmission. In accordance with Maxwell's equations, any electrical current will produce an orthogonally oriented magnetic field. It is this field which is measured with MEG. In order to generate a signal that is detectable, approximately 50,000 active neurons are needed. Since current dipoles must have similar orientations to generate magnetic fields that reinforce each other, it is often the layer of pyramidal cells in

[4] SQUID — Superconducting Quantum Interference Devices, are used to measure extremely small magnetic fields. They are one of the most sensitive magnetometers known.

the cortex, which are generally perpendicular to its surface, that give rise to measurable magnetic fields. Furthermore, it is often bundles of these neurons located in the sulci of the cortex with orientations parallel to the surface of the head that project measurable portions of their magnetic fields outside of the head. With current MEG techniques only cortical activity can be detected, while monitoring of deep activity is still not achievable. Also, the action potentials do not usually produce an observable field.

fMRI: Functional magnetic resonance imaging is the use of magnetic resonance imagery (MRI) to measure the hemodynamic response (changes in blood flow and blood oxygenation) related to neural activity in the brain or spinal cord of humans or other animals. It is actually one of the most recently developed forms of neuroimaging. When nerve cells are active they consume oxygen. The local response to this oxygen utilization is an increase in blood flow to regions of increased neural activity, occurring after a delay of approximately 1 to 5 seconds. Hemoglobin, that carries oxygen in blood, is diamagnetic when oxygenated but paramagnetic when deoxygenated. The magnetic resonance (MR) signal of blood is therefore slightly different depending on the level of oxygenation. These differential signals can be detected, using an appropriate MR pulse sequence, as a so called *blood–oxygen–level dependent* (BOLD) contrast. The precise relationship between neural signals and BOLD is under active research but, in general, informations on neural activity can be obtained with moderately good spatial and temporal resolution: images are usually taken every 1 to 4 seconds, and the voxels in the resulting image typically represent cubes of tissue about 2 to 4 millimeters in humans. Recent technical advancements have advanced spatial resolution to the millimeter scale.

When thinking about a user interface that makes use of biological signals, one should consider the principles already exposed in Chapter 1 when choosing how to acquire the input signal. From the point of view of user–friendliness the use of invasive methods should be clearly excluded, unless some cases in which it is unavoidable and in which a tailored solution should be developed. So the choice should be made between the various non–invasive techniques. On the other hand the adoption of MEG and fMRI raises the problems costs and portability because the equipment is extremely expensive and unwieldy. Furthermore the time resolution would not be high enough to provide a suitable interactivity for the most common task. In contrast EEG is quite inexpensive, the recording equipment can be really small and the temporal resolution can be high enough for the reaction times expected by a typical user. The factors that could raise problems with the use of EEG are the lower quality of the recorded signals and the need for a usually high number of electrodes that must be correctly placed. Anyway these are problems that can be dealt with, respectively, through signal processing techniques and the use of small sets of electrodes. For the reason just exposed, in the application presented here the EEG was chosen as the system's input channel.

Principles of Electroencephalography

The first recording of the electric field of the human brain was made by the German psychiatrist Hans Berger in 1924 [91]. He gave this recording the name electroencephalogram (EEG). The recorder signals are made up mainly of *spontaneous activity*. The amplitude of the EEG is about 100 μV when measured on the scalp, and about 1 mV when measured on the surface of the brain. The bandwidth of this signal was at the beginning thought to be ranging roughly from 1 Hz to about 50 Hz, but new results show that some components can reach much higher frequencies [92, 93, 94]. As the phrase *spontaneous activity* implies, this activity goes on continuously in the living individual. *Evoked potentials* are other components of the EEG that arise in response to a stimulus (which may be electric, auditory, visual, and so on). Such signals are usually below the noise level and thus not readily distinguished, and one must use various techniques like trains of stimuli and signal averaging to improve the SNR.

The number of nerve cells in the brain has been estimated to be on the order of 10^{11}. Cortical neurons are strongly interconnected. In the cortex the surface of a single neuron may be covered with 1000 to 100000 synapses [95]. Many are the models developed to describe an excitable neuron [96], but the main common characteristics are that the resting voltage is around $-70\ mV$, and the peak of the action potential is positive. The amplitude of the nerve impulse is about 100 mV and lasts about 1 ms. The bioelectric impressed current density $\boldsymbol{J}_i$ associated with neuronal activation produces an electric field, which can be measured on the surface of the head or directly on the brain tissue. For the EEG the basis for the impressed current density $\boldsymbol{J}_i$ appears to arise from the action of a chemical transmitter on post-synaptic cortical neurons. The action causes localized depolarization (an excitatory potential, EPSP) or hyperpolarization (an inhibitory potential, IPSP). The result in either case is a nonzero primary source. For distant field points the effect is the same of a net dipole for each active cell. Since neural tissue is generally composed of a very large number of small, densely packed cells, the whole brain appears as a continuous volume source distribution. Although in principle the EEG can be analytically evaluated, the complexity of the brain structure and its electrophysiological behavior actually preclude any computation.

The spontaneous EEG is usually recorded using the internationally standardized $10-20$ system. In this system 21 electrodes are located on the surface of the scalp, as shown in Figure 5.7. The positions are determined as follows: the reference points are *nasion*, which is the delve at the top of the nose, and *inion*, which is the bony lump at the base of the skull on the mid–line at the back of the head. From these points, the skull perimeters are measured in the transverse and median planes. Electrode locations are determined by dividing these perimeters into 10% and 20% intervals as shown in Figure 5.7 [97]. In addition to the 21 electrodes of the international $10-20$ system, intermediate 10% electrode positions are also used. The locations and nomenclature of these electrodes have been standardized by the American Electroencephalographic Society [98] (refer to Figure 5.8). In this setup, four electrodes have different names compared to the $10-20$ system ($T7$, $T8$, $P7$, and $P8$) and are drawn black with white text in the Figure 5.8. Besides the international 10-20 system, many other electrode systems exist for recording electric potentials on the scalp. Bipolar or unipolar electrodes can be used in the

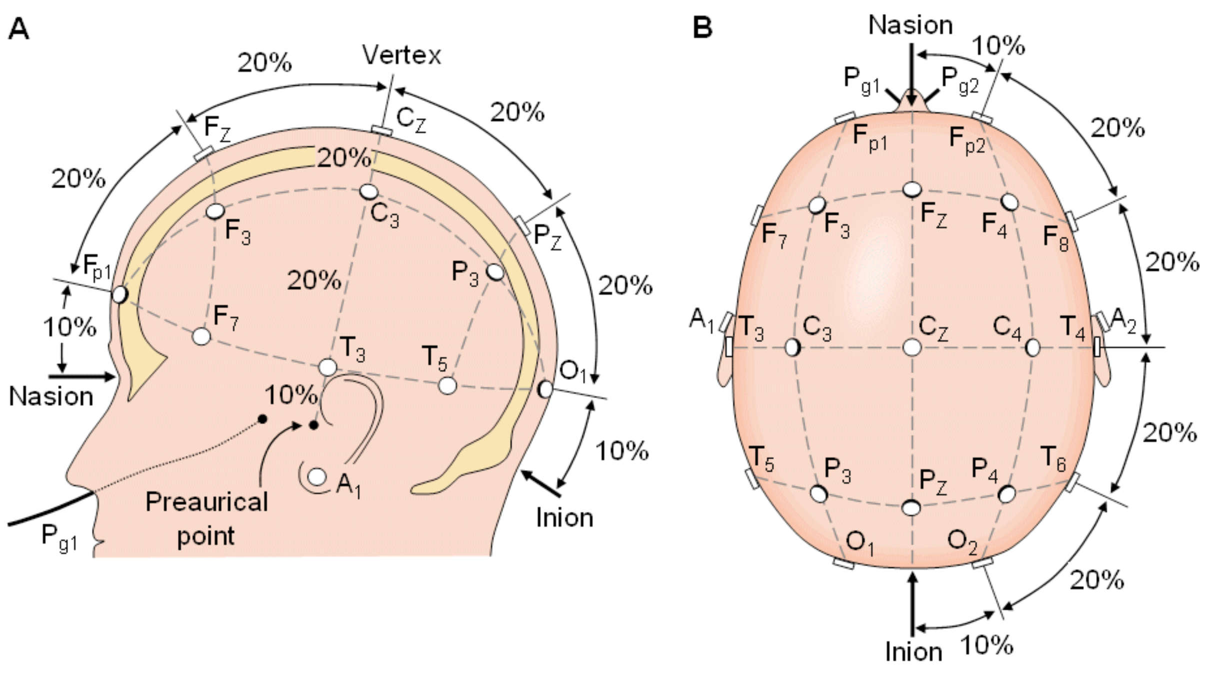


Figure 5.7: The international 10 − 20 system for EEG electrode placement, seen from (A) left and (B) above the head. Labels: A = Ear lobe, C = central, Pg = nasopharyngeal, P = parietal, F = frontal, Fp = frontal polar, O = occipital.

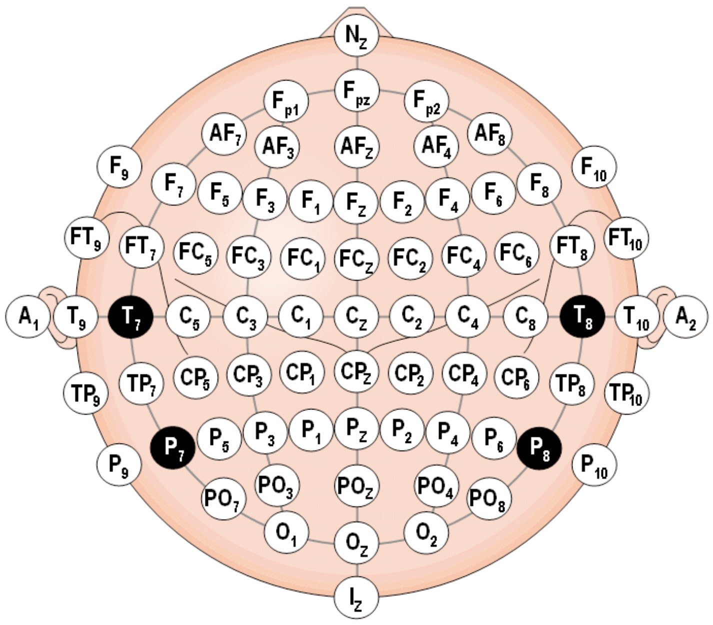


Figure 5.8: Location and nomenclature of the intermediate 10% electrodes, as standardized by the American Electroencephalographic Society.

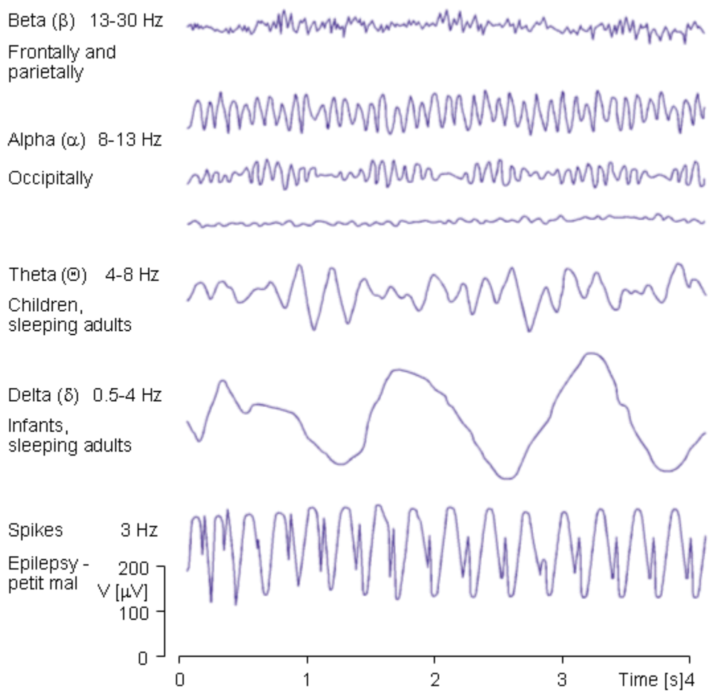


Figure 5.9: Examples of EEG waveforms in the various frequency bands.

EEG measurement. In the first method the potential difference between a pair of electrodes is measured. In the latter method the potential of each electrode is compared either to a reference electrode or to the average of all electrodes. The most recent guidelines for EEG-recording are published in [99].

The traditional use of the EEG in a clinic environment, the habit of visual inspection and the limited performance of early recording systems, brought to the classification of the signal into differently looking *waves* named *alpha*, *beta*, *delta*, and *theta* waves, as well as spikes associated with epilepsy. An example of each waveform is given in Figure 5.9. The alpha waves have a frequency $f_\alpha \in [8, 13]$ Hz and can be measured from the occipital region in an awake person when the eyes are closed. The frequency range of the beta waves is $f_\beta \in [13, 30]$ Hz and this kind of waveforms is detectable over the parietal and frontal lobes. The delta waves have a frequency range $f_\delta \in [0.5, 4]$ Hz and are detectable in infants and sleeping adults. The theta waves have a frequency range $f_\theta \in [4, 8]$ Hz and, as the delta waves, are recorded from children and sleeping adults.

5.2.2 Implementation of a Binary EEG Classifier

Many are the ways the EEG signal can be processed in order to extract informations from it, and many are the ways to use this kind of informations in a practical way. Since research is still ongoing on the subject and, unlike the ASR application, there are no publicly available systems to take as a reference in order to reproduce their behavior, part of the work done for this thesis was to identify a possible EEG–based BCI application and check for the applicability of HMM techniques on that application.

The choice was to develop an application that could be a starting point for an EEG–based user interface. Thinking about the most common user interfaces currently available, since the feedback is always visual the base operations that must be performed in order to interact with the interfaces themselves are *pointing* some target and *selecting* it. The movement of a cursor or pointer through EEG is a widely studied problem (of which some examples can be found in [100, 101]), whereas the selection task is almost not treated at all. Managing to identify a selection task can be interpreted as being able to identify the *will* to perform an action through the EEG. This *will* is the event I aimed to identify through analysis of the EEG signal. So the pattern recognition task will consist of discriminating between two classes: *select* and *don't select*.

Data Recording

First of all, it is necessary to obtain some EEG recordings that emphasize (or at least make identifiable in time) the parts of the signal that are likely to contain the events that will allow for classification. A dedicated recording setup was developed for the purpose. It consisted of a simulated point-and-click BCI task reproducing the selection of a button appearing on a display. Since the task to be performed through the EEG was only the selection, the movement of the cursor was controlled through head movement using a commercial web–cam and an free–ware software (*VisualMouse*, [102]) capable of converting head movements into the movements of a mouse pointer. The software used to provide the visual stimuli was written in Python programming language [103] using an open–source library designed on purpose to produce stimuli for vision research experiments (*Vision Egg* [104]).

The experiment was performed as follows:

- The subject, with the electrodes already in place for EEG recording, sat in front of a flat–panel 19″, 60 *Hz* display where the stimuli are going to be displayed. In order to reduce any disturbance, the display was placed in such a way that nothing but the screen is on the visual field of the subject. Also auditory stimuli were excluded. A web–cam is pointed on the subject's face its output is used by the *VisualMouse* software to drive the movements of the on–screen pointer (see Figure 5.10).

- The subject was given some time to learn how to control the mouse pointer with head movements. Then he was instructed on how to react to the visual stimuli that were going to be presented. After that the experiment begun.

- The visual stimuli were organized in *trials*. Each trial was performed in the following way (refer to Figure 5.11):

    1. Blank screen for 1 second (here *blank* means that the screen is filled with the background color, that is black in this experiment).

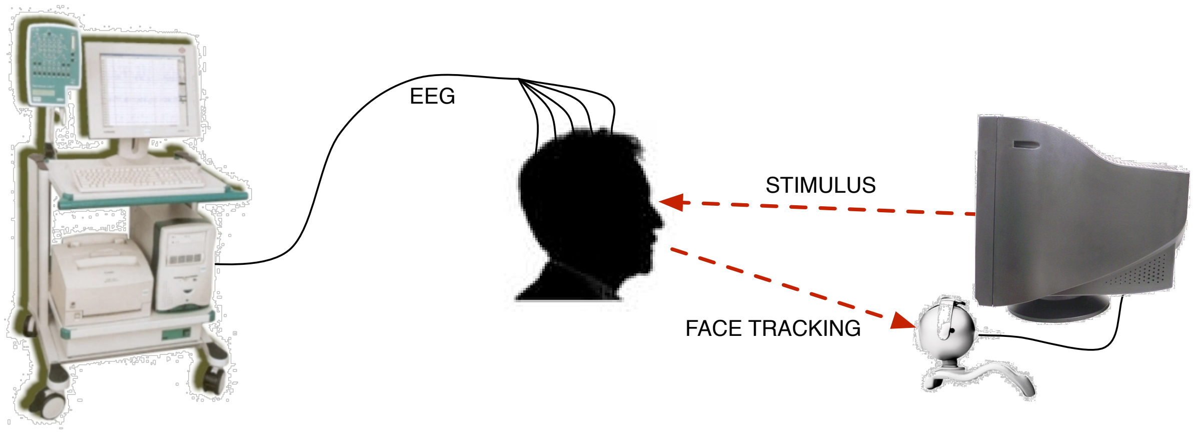


Figure 5.10: Representation of the experimental setup used for the EEG selection recordings.

2. Flash of a sync target in the upper–left corner of the screen with a 100 *ms* duration. This flash was captured through a photodiode placed on the upper–left corner of the display, directly from the EEG recording equipment, and the spikes saved in a separate channel, in order to be used later for alignment of the EEG waveforms.
3. After 500 *ms* after the sync flash, appearance of a colored visual cue (green or red) in the center of the screen with duration 2 seconds. The subject was instructed to remember the color of the cue and to perform a selection, in a later stage, only if the presented cue was green.
4. Blank screen for 2 seconds. This time is intended to let any reaction evoked by the colored cue to settle in the EEG traces, so that it will not interfere with the part of the signal to be classified (the one corresponding to the actual selection).
5. A white target square appears in a peripheral part of the display, and an arrow pointer appears in the center. At this point the subject is requested to move the arrow on the target square using head movements and, when the pointer reaches the target, to keep the pointer on the target and, if the cue was green, try to select the button *mentally*. If the cue was red the subject must just keep the pointer on the target. In any case the target disappears after 3 seconds of permanence of the pointer on it. It is extremely important to stress that no guidelines were given to the subjects on how to actually perform the selection. The aim was to check if there is some way of performing an EEG selection that is spontaneous and does not require any specific training. Of course this approach will produce a smaller amount of meaningful trials just because in some cases the subject's selection waveforms would be identical to the non–selection ones.
6. The disappearance of the target is followed by 1 second of blank screen.

- The trials were grouped in *trial sets*. Each trial sets consisted of 8 trials with each target

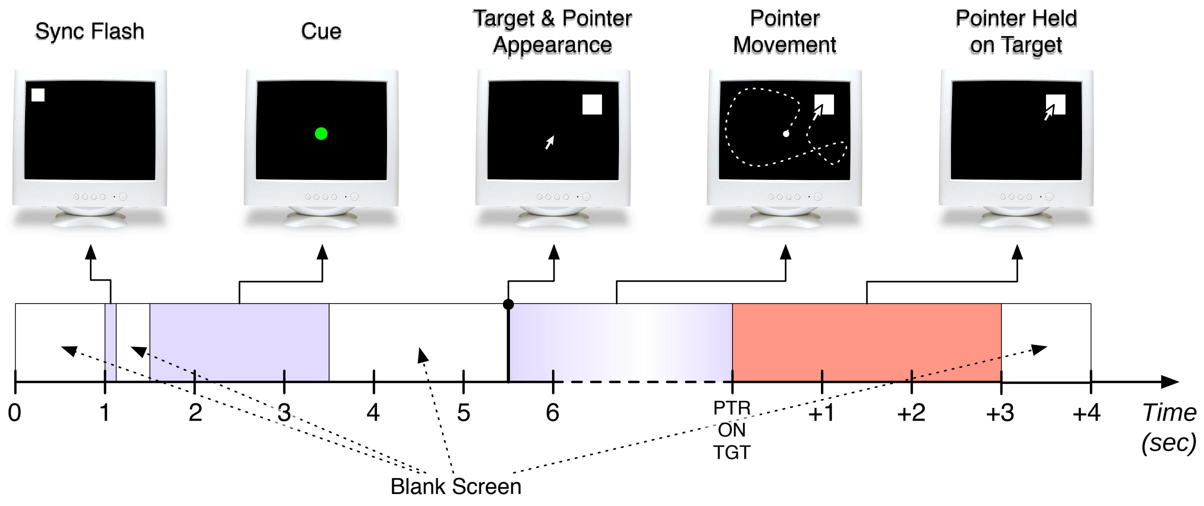


Figure 5.11: Timeline for a single trial in the EEG selection experiment.

presented in a different position. The position of the 8 targets in a trial set was determined by even distribution around a circle centered in the display (see Figure 5.12). The appearance was pseudo–random: in the same trial set the target appears only once in each of the 8 positions. Also the cues were presented in a pseudo–random order, with an even distribution of the colors (i.e. 50% red, 50% green). This kind of setup insures that, for any number of trial sets, there will be half selection trials and half non–selection trials. Furthermore the pseudo random order of presentation of stimuli avoids the elicitation of EEG potentials due to stimulus expectation [105], elicitation that would be present if the pattern of target presentation was known. Several trial sets were performed for each subject.

During the experiments the Python script that provides the stimuli also saves informations on the ongoing experiment into two files: the first file contains the mouse coordinates for all the duration of the experiment, sampled with the refresh frequency of the screen (60 *Hz* in this experiment). A sample of coordinates recorded during a recording session is shown in Figure 5.13(a). The second file contains a sequence of events related to what happens during the recording. Each event is saved in the form

```
timecode eventname
```

A sample of an event file produced during one of the recording sessions is shown in Figure 5.13(b).

The event names are representative of the various phases of the trial:

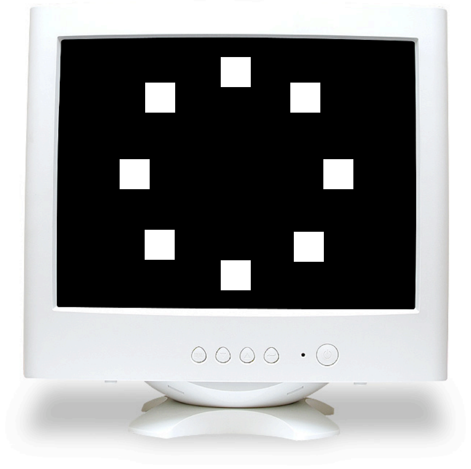


Figure 5.12: Possible locations for the appearance of targets in a 8 target trial set.

`sync-on` / **`sync-off`**	Appearance or disappearance of the sync square on the display.
`go-on` / **`go-off`**	Appearance or disappearance of a selection cue (a green spot in the used set–up).
`nogo-on` / **`nogo-off`**	Appearance or disappearance of a non–selection cue (a red spot in the used set–up).
`target-on` / **`target-off`**	Appearance or disappearance of a target square (in whichever position).
`ptr-on-target`	The pointer is moved from a blank part of the screen onto a target square.
`ptr-off-target`	The pointer is moved off from a target square to a blank part of the screen.

A typical event sequence is like the one made up by the first 11 events in Figure 5.13(b): the sync square flashes on and off, a cue appears and disappears, a target is presented, the pointer goes on the target (and possibly off the target and again on it, for any number of times), and finally (after 3 seconds of permanence in the `ptr-on-tgt` state) the target disappears. As it can be seen later in this section, the part of the signal in which the selection–related features are expected to be found is the last one in each trial, in particular the time time span corresponding to the 3 seconds of pointer permanence onto the target.

The experiment was performed on 4 male subjects, and 10 trial sets were recorded for each subject (in total of 320 trials, half of them corresponding to an attempted selection from the subject).

The recording equipment used had 64 channel plus one electrode for on–scalp reference.

```
37.5  7.3  (320, 251)
37.6  7.4  (313, 268)
37.7  7.5  (310, 281)
37.8  7.6  (288, 307)
37.9  7.7  (265, 316)
38.0  7.8  (259, 318)
38.1  7.9  (233, 321)
38.2  8.0  (235, 285)
38.3  8.1  (273, 230)
38.4  8.2  (317, 206)
38.5  8.3  (359, 197)
38.6  8.4  (426, 197)
38.7  8.5  (434, 200)
38.8  8.6  (463, 222)
38.9  8.7  (467, 225)
39.0  8.8  (473, 227)
39.1  8.9  (476, 227)
39.2  9.0  (479, 231)
```

(a)

```
59.6  sync-on
59.8  sync-off
60.2  go-on
62.2  go-off
64.2  target-on
66.4  ptr-on-tgt
66.8  ptr-off-tgt
67.5  ptr-on-tgt
68.5  ptr-off-tgt
68.9  ptr-on-tgt
73.2  target-off
74.2  sync-on
74.4  sync-off
74.8  nogo-on
76.9  nogo-off
78.8  target-on
81.1  ptr-on-tgt
85.5  target-off
```

(b)

Figure 5.13: Samples of the reports produced during a recording session: (a) mouse coordinates, with format `total_time trial_time (xcoord,ycoord)`; (b) events, with format `total_time event_name`.

The sampling rate was of 256 *Hz* with 16 bits per sample. The electrode placement was made according to the $10 - 10$ international system, taking only a subset of the specified locations (refer to Figure 5.8, Page 116).

A sample of the recorded signal is shown in Figure 5.14. After a first inspection it could be seen that the recordings were really noisy with a strong 50 *Hz* component, probably coming in from the on–scalp ground contact. Furthermore, as expectable in an EEG recording, the presence of muscular artifacts was significant.

Signal Processing

Signal processing on the recordings was performed using Mathworks' Matlab [106] and the open–source toolbox *EEGLab* [107], developed by the *Swartz Center of Computational Neuroscience* at the University of California, San Diego. The recordings were imported in Matlab along with the event markers saved by the stimulus script during the experiment. The event markers were then aligned to the signal thanks to the channel containing the spikes corresponding to the sync square flashes captured with the photodiode.

The recordings were processed by first removing the DC component via high–pass filter-

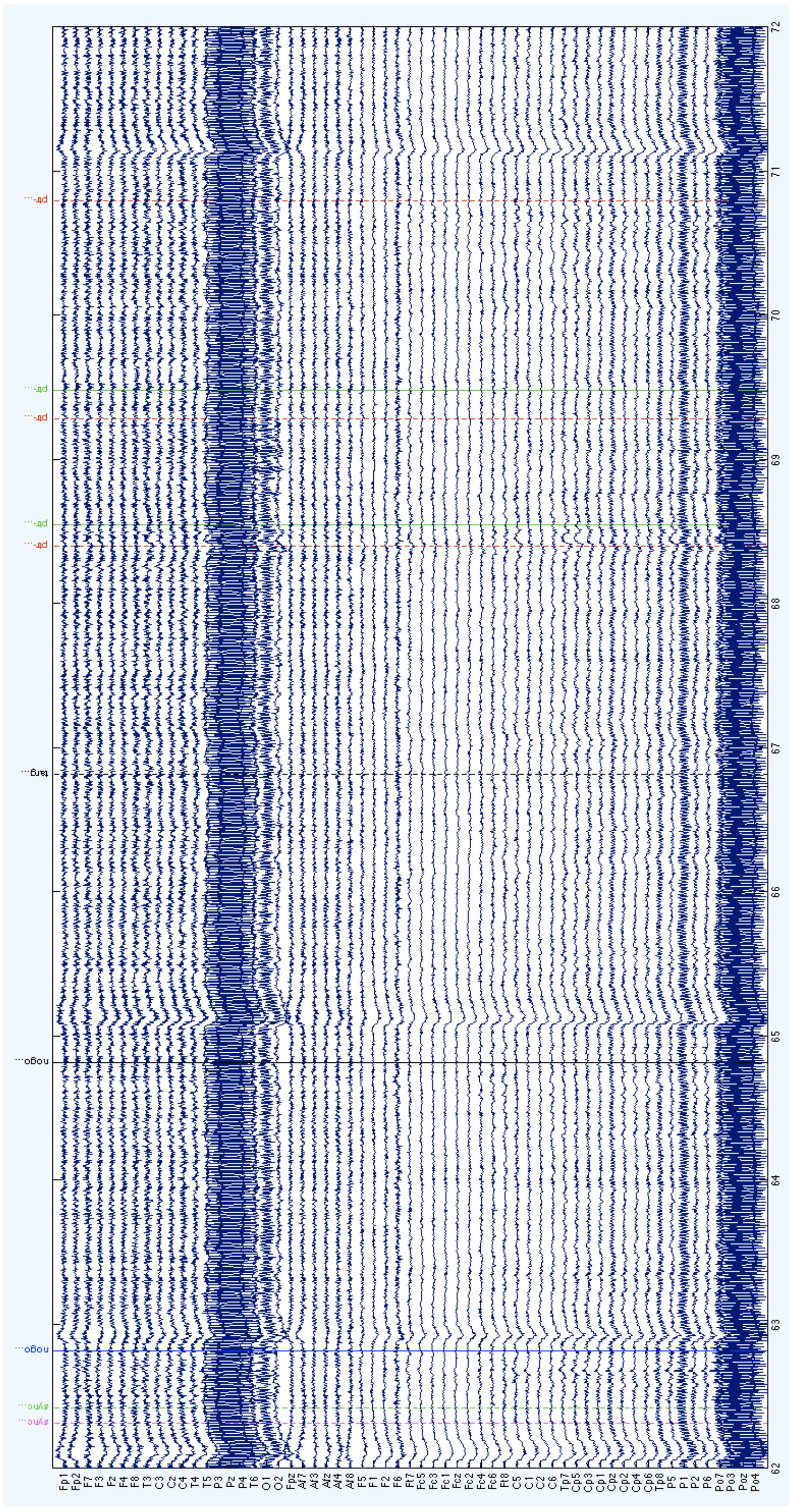


Figure 5.14: A sample of the EEG signal recorded during the experiments. Also the events are marked.

ing. Then *Independent Component Analysis* (ICA) was performed, and finally the signal was segmented in epochs containing the three second of signal before the disappearance of the target in each trial (i.e. from the last `ptr_on_tgt` event to the `tgt_off` in each trial).

The ICA method allows to solve the so called *cocktail party problem*, that is recovering some source signals from a set of observed signals that are a linear mixture of the original ones. Formally, given a set of N source signals $\{s_1(t) \dots s_N(t)\}$, it is assumed that each of the observed signals $\{x_1(t) \dots x_N(t)\}$ can be expressed as

$$x_i(t) = \sum_{j=1}^{N} a_{ij} s_j(t) \qquad \forall\ 1 \le i \le N \tag{5.8}$$

where the signals are treated like random variables, and the sum made sample by sample. The matrix notation is the most commonly used when treating the problem, thus

$$\boldsymbol{x} = \boldsymbol{A}\boldsymbol{s} \tag{5.9}$$

where both the mixing matrix $\boldsymbol{A}$ and the components $\boldsymbol{s}$ are assumed to be unknown. By assuming that the source components are *statistically independent* (for definitions on random variables refer to [108]) and with a *non–gaussian probability distribution*, through the ICA method is possible to find an unmixing matrix $\boldsymbol{W}$ such that

$$\boldsymbol{s} = \boldsymbol{W}\boldsymbol{x} \tag{5.10}$$

For a detailed overview of ICA theory and techniques refer to [109].

The reason for using ICA is that the signal recorded from each electrode using the EEG can be seen as a mixture of signals produced by an extremely high number of sources inside the brain. Clearly the number of electrodes used does not allow to recover all the single sources, but only clusters of synchronized neighboring neurons (that are actually the main sources of the signals recorded through electroencephalography). Moreover the fact that short distance neural connections are always stronger than long distance ones, makes the signals produced by the neuronal clusters likely to be independent one from another. Anyway the effectiveness of the use of ICA on EEG sources separation has been proved more than once (see, for example [110]. Furthermore, the ability of ICA of separating independent components has been shown to be highly effective also for artifact removal [111, 112].

The application of the ICA on the recorded signal provided interesting results. The number of components obtained was 59, because of the removal of some unusable channels. In particular the component accounting for most of the recorded signal power could be easily associated with artifactual sources. Figure 5.15, for example, refers to the component with the highest power among the ones identified. The upper–left corner of the figure shows a top view of the projection of the averaged signal on the scalp along with the electrodes' positions (for reference). The upper–right corner shows, for each of 40 *selection* trials from the same subject, the time evolution of the signal's amplitude (indicated by the color map). The bottom trace is the average over all the trials (also called *event-reated potential*, ERP). Here the time span is referred

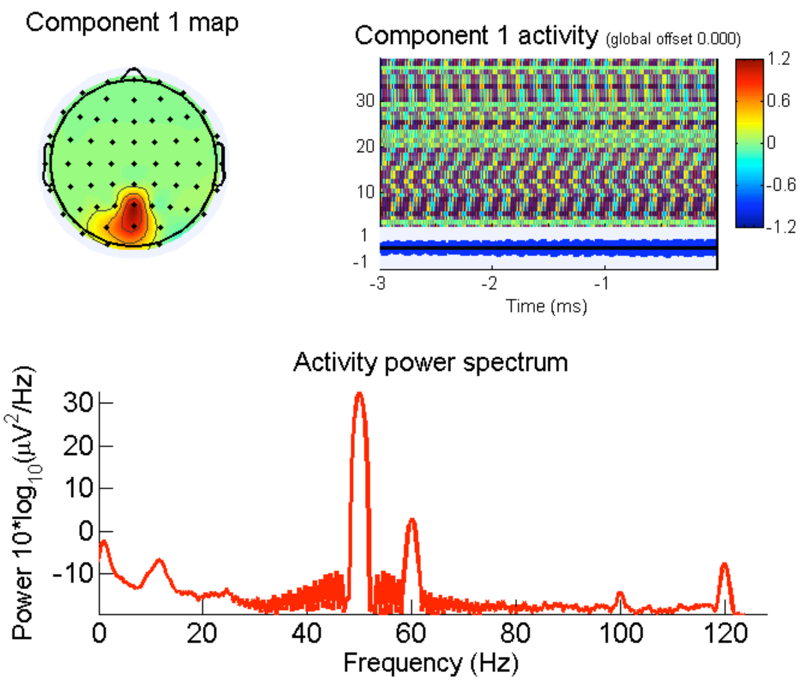


Figure 5.15: Independent component 1 — Plots relative to 40 selection epochs taken from the same subject. *Top-left:* Projection of the component over the scalp. *Top-right:* Map of the signal amplitude for each trial and ERP. *Bottom:* ERP Spectrum.

to what I will call from now on, an *epoch*, that is the three seconds of pointer's permanence on the target (all the following considerations on experiment trials will be referred to the corresponding epoch). Lastly the bottom of the figure shows the full spectrum of the component's signal averaged over the 40 epochs. Several considerations can be made on the contents of Figure 5.15: firstly the projection of the activity is localized exactly around the ground electrode; secondly the ERP has a random time evolution, phenomenon that can be seen also in the non consistent outcomes of the single trials; finally the signal's spectrum has a strong peak at 50 *Hz* and a secondary peak at 60 *Hz*. Noticeable second harmonics of these frequencies (100 *Hz* and 120 *Hz*) are also present. From these observations it is clear how the ICA processing has isolated as the first independent component two main sources of noise coming from some kind of ground loop: a 50 *Hz* source (with its second harmonic) corresponding to the power grid's AC frequency, and a 60 *Hz* source (also with its second harmonic) corresponding to the display's refresh frequency. The second independent component has almost the same characteristics of the first one, with the exception of a lower power.

Figure 5.16 shows another interesting artifactual component extracted through the ICA (the third one in order of decreasing power). Again, the epochs are taken from a single subject's recordings, and correspond to the trials in which a selection attempt was requested. In this case the projection of the component's activity is concentrated in the frontal region of the scalp.

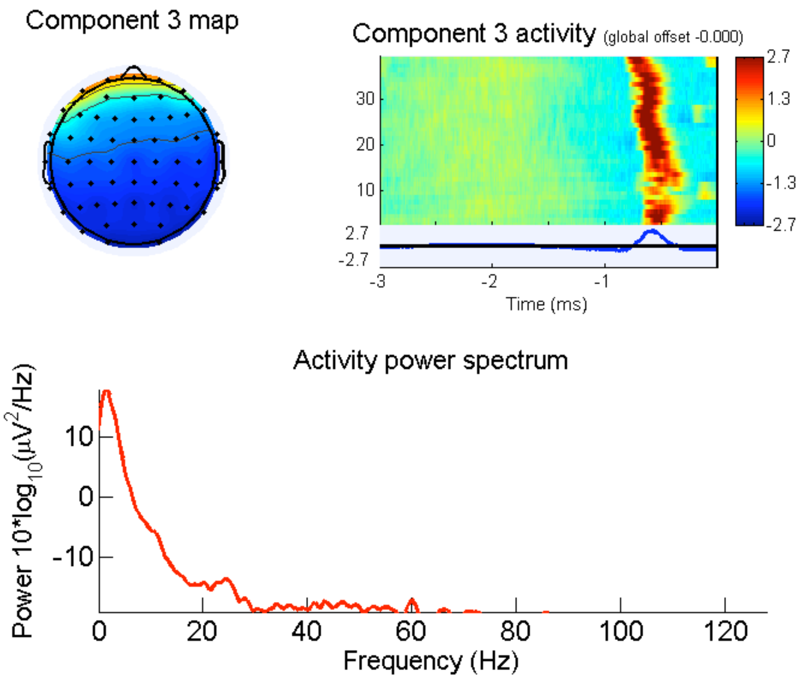


Figure 5.16: Independent component 3 — Plots relative to 40 selection epochs taken from the same subject. *Top-left:* Projection of the component over the scalp. *Top-right:* Map of the signal amplitude for each trial and ERP. *Bottom:* ERP Spectrum.

The ERP shows a clear peak towards the end of the epoch. The inter–epoch consistency is high, confirming the peak of activity towards the epoch's end. Finally, in the signal's spectrum, a marked peak is present at really low frequencies. A straightforward interpretation these features can be found by observing the subjects during the selection trials: since the subjects had no known means of making a mental selection, most of them tried to concentrate on the target, and this action was always accompanied by contraction of the forehead muscles. The electrical activity that produce these contractions is easily spotted by the EEG electrodes because of its high intensity with respect to brain signals. The ICA processing manages to isolate it in a really clear way, and the results are like the ones presented in Figure 5.16.

Of course, not every independent component obtained through ICA processing correspond to some artifactual source. An example of a typical brain source component is shown in Figure 5.17. Here the scalp projection shows a left parietal–occipital activity. The ERP has a clearly recognizable positivity towards the end of the epoch, and the single trial are consistent only towards the end of the epoch, as if the subject's control on his brain activity needs some time to be exploited, and the effects become stable with a certain delay after the beginning of the selection process. Also the signal's spectrum is typical of brain sources, with high values at lower frequencies and a constant decrease towards higher ones.

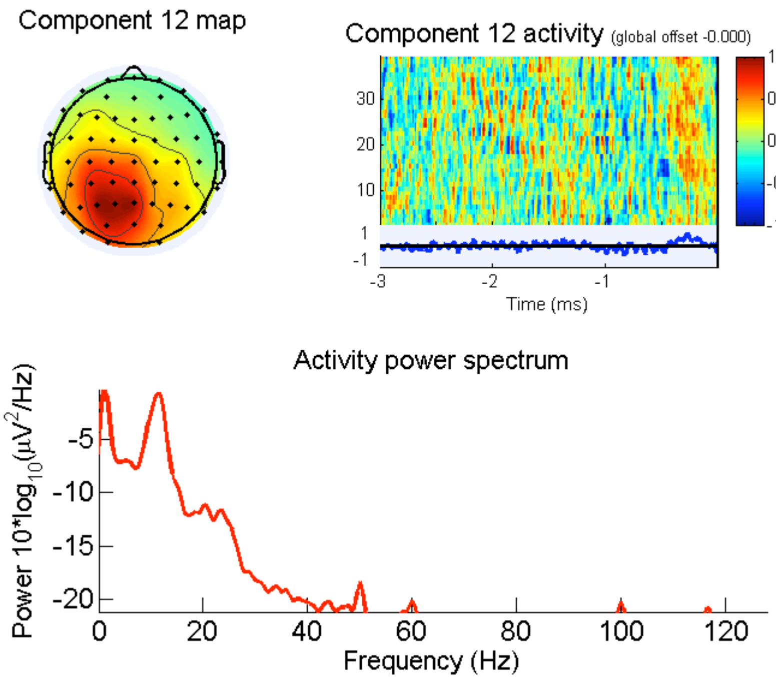


Figure 5.17: Independent component 12 — Plots relative to 40 selection epochs taken from the same subject. *Top-left:* Projection of the component over the scalp. *Top-right:* Map of the signal amplitude for each trial and ERP. *Bottom:* ERP Spectrum.

EEG Classification

The classification tests for this experiment have been attempted on the 59 ICA components extracted from the EEG recordings (as already said, some channels had to be removed because of technical problems during the recording process). The main problem is that the temporal position inside the single epochs of the (eventually present) discriminating events, is not known neither easily identifiable. Here the automatic segmentation capabilities of the HMM classifiers seem to be a really promising way of solving the problem. That is why the HMM has been selected for the first classification tests.

The overall structure of the pattern recognition system set up for this task is similar to the one shown in Figure 1.1 (Page 8): the independent components are fed in the system as 59 independent streams, processed by a feature extraction front–end, and then classified.

The front–end was an *Linear Prediction Coding Cepstrum* extractor (LPC–Cepstrum). It was implemented, for each of the input channels, by a sliding window with Hamming profile, a *Linear Prediction Filter*, and a DCT (see Figure 5.18). The windowing operation was already described in Section 5.1.2 (Page 104) for the ASR front–end. The window used in the front–end was 20 *ms* wide with 10 *ms* overlap. The *linear prediction analysis* is implemented through an

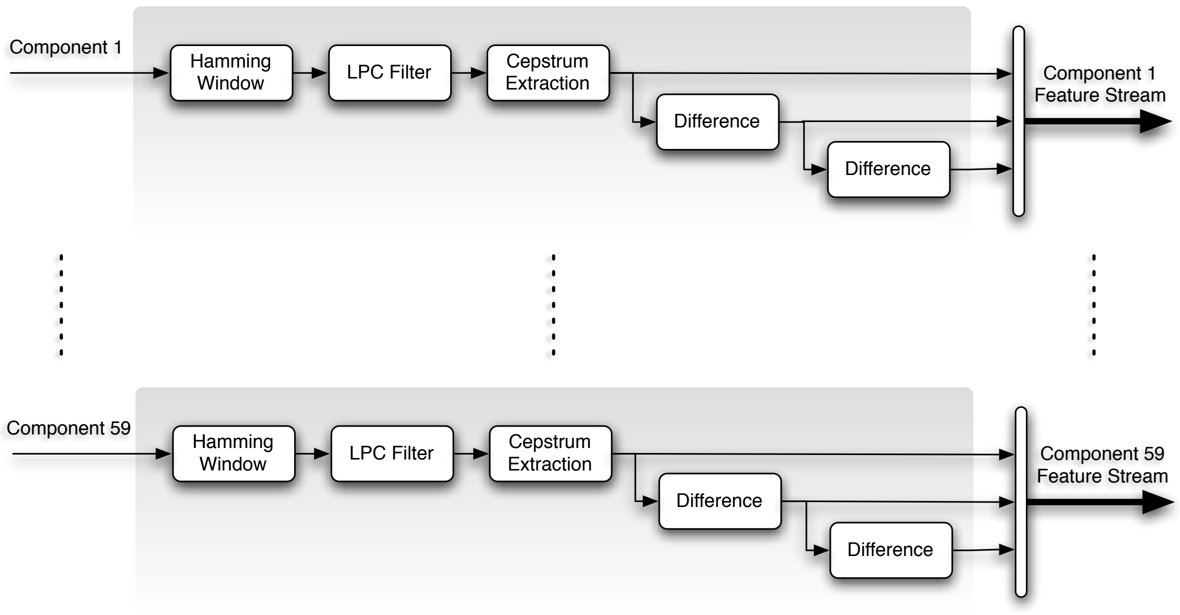


Figure 5.18: Feature extraction front–end used in the EEG pattern recognition system.

all–pole filter with a transfer function of the form

$$H(z) = \left(\sum_{i=0}^{p} a_i z^{-i} \right)^{-1} \tag{5.11}$$

where p is the number of filter poles and $a_0 = 0$. The filter coefficients a_i are chosen in such a way to minimize the filter's prediction error, in the mean square sense, summed all over the analysis window. This minimization was done using the *autocorrelation method*: given the window samples $\{s_1 \dots s_N\}$, the first $p+1$ terms of the autocorrelation sequence are calculated using

$$r_i = \sum_{j=1}^{N-i} s_j s_{j+i} \qquad 0 \le i \le p \tag{5.12}$$

The filter coefficients are then computed recursively using a set of auxiliary coefficients k_i and by initializing the prediction error E to r_0. For any couple $k_j^{(i-1)}, a_j^{(i-1)}$ related to a filter of order $i-1$, the filter of order i can be calculated in three steps:

1. Computation of a new set of reflection coefficients:

$$k_j^{(i)} = k_j^{(i-1)} \qquad \forall\ 1 \le j \le (i-1) \tag{5.13}$$

$$k_i^{(i)} = \frac{r_i + \sum_{j=1}^{i-1} a_j^{(i-1)} r_{i-j}}{E^{(i-1)}} \tag{5.14}$$

2. Update of the prediction energy:

$$E^{(i)} = \left[1 - \left(k_i^{(i)}\right)^2\right] E^{(i-1)} \tag{5.15}$$

3. Computation of the new filter coefficients:

$$a_j^{(i)} = a_j^{(i-1)} - k_i^{(i)} a_{i-j}^{(i-1)} \qquad \forall\ 1 \le j \le (i-1) \tag{5.16}$$

$$a_i^{(i)} = -k_i^{(i)} \tag{5.17}$$

This process is repeated recursively until the required filter order p is reached. Finally, in order to obtain the LPC–cepstrum, it would be necessary to first apply a Fourier transform to the LPC filtered data in order to extract the spectrum amplitude, and then apply the DCT. Anyway in the case of LPC analysis, a simpler recursive formula exists that can be applied directly on the filter coefficients to obtain the cepstral coefficients:

$$c_n = -a_n - n^{-1} \sum_{i=1}^{n-1} (n-i) a_i c_{n-i} \tag{5.18}$$

In the EEG front–end an LPC filter of order 14 was used and 12 cepstral coefficients were extracted from each window of samples. To the vector of cepstral coefficients an energy coefficient was appended, computed as

$$E = \log\left(\sum_{n=1}^{N_s} s_n^2\right) \tag{5.19}$$

where N_s is the number of samples in the window. As in the ASR application, the features along with their first and second order differences have been extracted using HTK's `HCopy` tool, and stored to avoid re-computation during the tests. The final feature vectors had a frequency of 100 *Hz* (10 *ms* period) and contained 39 parameters each.

For the implementation of the classifier, two HMMs were trained for each channel: one trained on the features relative to the selection events and the other trained on the features relative to the non–selection events. The topology used for all the HMMs is shown in Figure 5.19. The HMM has a 5–state left–right topology with emission probabilities PDFs modeled using 4–component gaussian mixtures. The three emitting states are logically associated with the expected structure of the input signal: the central state is supposed to model the actual discriminating event, while the other two state shave been introduced in order to take into consideration the leading and trailing background signal (where background refers to any part of the signal not related to the event to be classified). The transitions are conceived in such a way that the emission sequence can be of the form

$$< sequence >::= [\mathbf{O}_{LBG}, \{\mathbf{O}_{LBG}\}], \mathbf{O}_E, \{\mathbf{O}_E\}, [\mathbf{O}_{TBG}, \{\mathbf{O}_{TBG}\}];$$

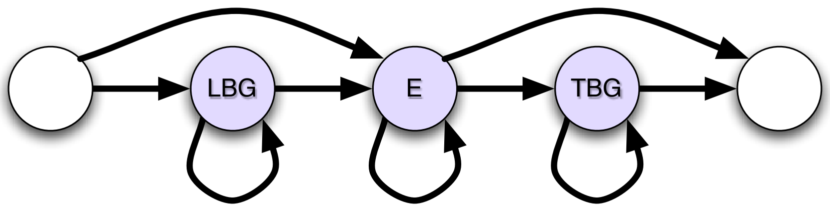


Figure 5.19: The topology of the HMMs used in the EEG classifier. The emitting states are drawn in a darker tint and marked as following: *LBG* to model the leading background, *TBG* to model the trailing background, *E* to model the discriminating event.

where $\boldsymbol{O}_{LBG}$ is an emission from the leading background state, $\boldsymbol{O}_E$ an emission from the event state, $\boldsymbol{O}_{TBG}$ an emission from the trailing background state, and the extended Bakus–Naur form (EBNF) has been used to describe the repetition rules (square braces for optional items an curly braces for optional repetitions).

The training was performed using the HTK tools (`HCompV`, `HERest` and `HHEd`) and the procedure was the same used for the ASR classifier (see Page 109). The whole available data set contained 40 selection epochs and 40 non–selection epochs for each of the 4 recorded subjects: 160 selection epochs plus 160 non–selection epochs in total. Since the data set was small and problems of under–training could arise, 20 different partitions were made out of the available data: each partition consisted of a training data set with 95% of the epochs (304 epochs), and a test data set with the remaining 5% (16 epochs). Both the training and test data sets were made in such a way that each one was made up of 50% selection epochs and 50% non–selection epochs. For all of the 59 channels in each data set partition, a couple of HMMs was trained separately. Thus the output of the training sessions was a collection of 20 models sets, each of them consisting of 118 HMMs (two for each of the 59 input channels).

The test sessions were performed by using HTK's `HVite` to recognize all of the 20 test data sets using the corresponding model set. Actually for each model–set/training–set pair, 59 separate recognition sessions had to be performed, one for each input channel, thus providing 59 separate accuracy measures. As a matter of fact *each model set behave like a set of* 59 *binary classifiers running in parallel*.

The accuracy obtained for each channel in the various sessions was then averaged in order to make the measure statistically more significant. The results were analyzed and it could be noted that while some channels did not allow at all for discrimination of the selection and non–selection epochs (providing a *chance level* accuracy around 50%), other channels performed clearly above 50% accuracy thus confirming the initial hypothesis on the possibility to correctly classify the intent of selection in the EEG signal. A first interesting result is shown in Figure 5.20. It shows the accuracy percentage of two artifactual components (already described in this section, Figures 5.15 and 5.16), in function of the training iteration index. The chance level (50% accuracy) is marked for reference. As it could have been expected, the artifactual component related to the forehead muscular contraction has the highest score among all the components (on the right in the figure). That because the muscular contractions are likely to be present only

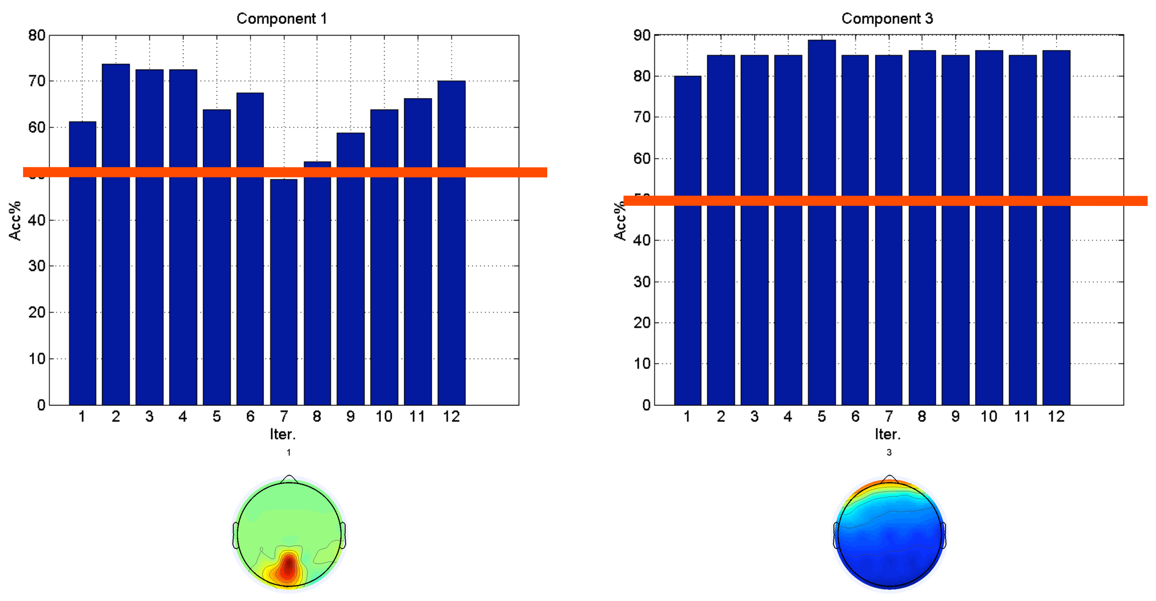


Figure 5.20: Results of EEG Classification through HMM. Accuracy percentage vs. training iteration index. The 50% chance level is marked for reference. *Left:* Component 1 (containing 50 *Hz* and 60 *Hz* noise). *Right:* Component 3 (forehead muscular contractions).

when a selection is required to the subject. When the selection os not required the subject tends to be more relaxed and the contraction do not appear (or have a really low intensity). A more unusual result has been obtained with the artifactual component containing the ground loop noise (Figure 5.20 on the left). Even here the classification accuracy is quite high (save for a drop halfway during the training process). Since the 50 *Hz* and 60 *Hz* noise could not have been produced by a brain source (even because the intensity is too high), it is probable that part of the brain or muscular signal has been collected by the ICA together with the noise (in fact this can be also noticed from the signal's spectrum at lower frequencies, see Figure 5.15, Page 125). Since the noise does not change between the selection and non–selection epochs, the HMM training procedure automatically discarded it while emphasizing the other residual components of the signal.

Indeed these two results on artifactual components, even if interesting in some sense (a classifier could be developed, for example, using only the muscular activity as input), are not useful when it comes to the aims of this tests, that is developing a classifier for brain activity. On the other hand significative classification results have been obtained with component that, for their signal characteristics, clearly represent some kind of brain activity. The results for the best scoring brain sources are shown in Figure 5.21. Both of the components scored around 70% accuracy and seem to be complementary, reflecting a symmetrical parietal–occipital activity in the brain (from the scalp projection it can be noticed that the activities have opposite sign, but this is simply due to the ICA technique that can estimate components save for the sign). These

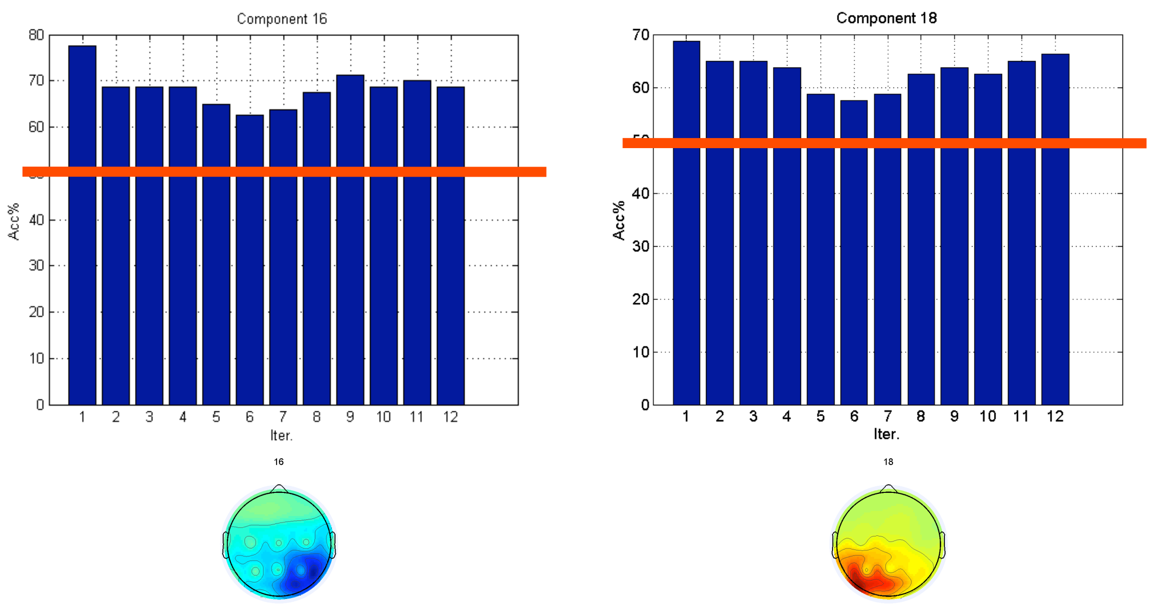


Figure 5.21: EEG Classification through HMM. Accuracy vs. training iteration. Chance level marked for reference. *Left:* Component 16 (right parietal–occipital brain source). *Right:* Component 18 (left parietal–occipital brain source).

components are actually related to brain sources (neuronal clusters) that show to have a causal relationship with the selection attempts made by the subjects during the experiment.

Other highly scoring components are the ones shown in Figure 5.22. The classification of these components can be accurate up to 70% – 75% but their nature is not clearly identifiable as in the examples shown in Figure 5.21. The component shown on the left of the figure shows an occipital activity while the one shown of the right shows a right–occipital activity. This kind of peripheral localization is sometimes characteristic of muscular artifacts. Further analysis would be needed for a correct identification of these sources, but this is out of the aims of the current work and will not be treated here.

Finally, other components provide lower but still significant classification accuracies. These components are shown in Figure 5.23. All of them seem to be generated by brain sources, because of their localization, of their spectrum and of the low power levels of the signal. The component shown in the upper left of the figure shows a frontal activity and levels of classification accuracy around 60% – 65%. The one shown in the upper right is localized in the left–central zone of the scalp, with the same levels of accuracy (60% – 65%). Finally, the other two components refer to right central–temporal activities, with accuracies around 60%.

The results of the classification tests just described confirmed the hypothesis made at the beginning of this section about the possibility to use the EEG signal to accomplish selection tasks. It is worth to stress that no tuning at all have been performed on the system and much higher

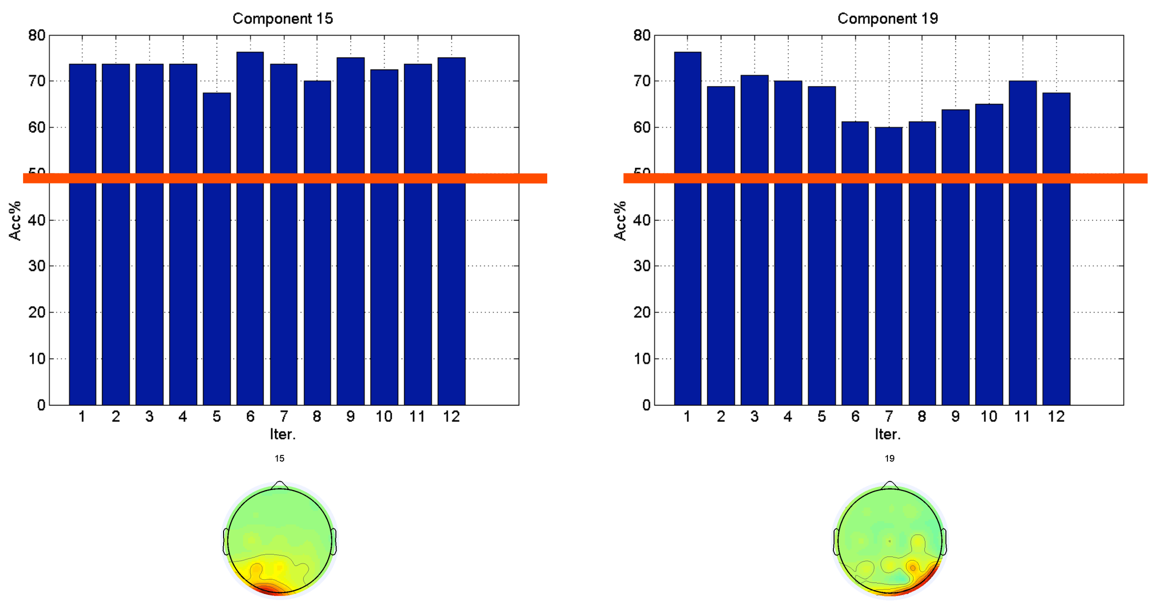


Figure 5.22: EEG Classification through HMM. Accuracy vs. training iteration. Chance level marked for reference. *Left:* Component 15 (not clearly identified occipital source, brain or muscular). *Right:* Component 19 (not clearly identified left occipital source, brain or muscular).

accuracies are likely to be obtained. The points to develop are: a more accurate definition of the front–end (experimenting also other methods for feature extraction), some refinements on the HMM classifier (on the topology and emission probability PDFs for example), and the acquisition of a bigger EEG database in order to better train the models. Furthermore, even if here the case of a subject–independent classifier was studied, with the aim of a multi–user class of interfaces, also the possibility of tailoring the classifier on a single subject could be explored. Clearly, in that case, the reachable accuracy is going to be even higher than the one expectable in a subject–independent system.

Given these preliminary results, the natural evolution of the system as a whole would be to identify the independent components that allow for better classification (we saw that their number is going to be of the order of 10 or less), and extract the classification results by mixing the outputs of the single classifiers. The number of needed electrodes for the measurements can be reduced by selecting only the ones that most contribute to the used components, thus making the sampling equipment more compact, portable and wearable.

As a final step, in order to double–check the results obtained. The same classification tests have been repeated with another classifier. The chosen classifier was the *Support Vector Machine* (SVM). The basic SVM is a linear classifier: given a set of multi–dimensional data points of the form

$$\{(\boldsymbol{x}_1, c_1), \ldots, (\boldsymbol{x}_n, c_n)\} \tag{5.20}$$

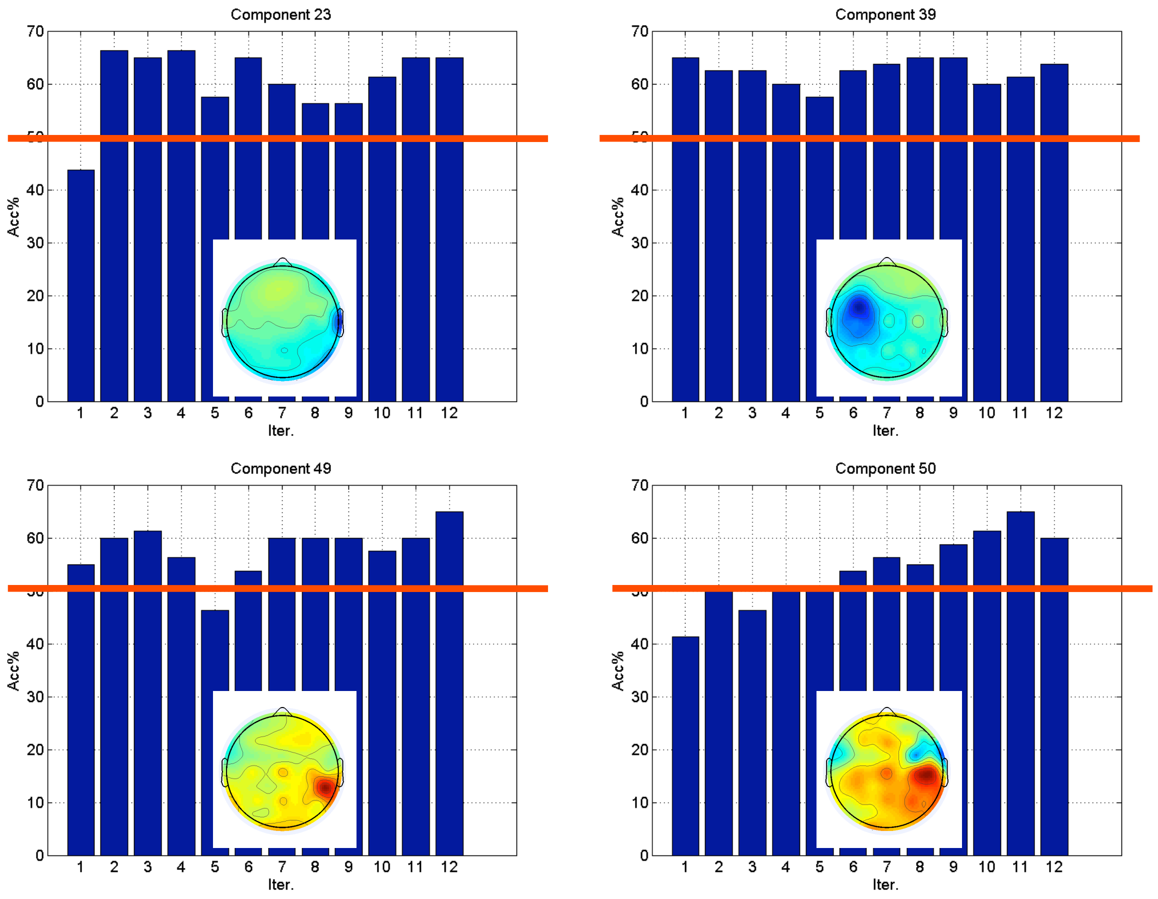


Figure 5.23: EEG Classification through HMM. Accuracy vs. training iteration. Chance level marked for reference. *Upper Left:* Component 23 (frontal brain source). *Upper Right:* Component 39 (left central brain source). *Lower Left:* Component 49 (right temporal unidentified source, brain or muscular). *Lower Right:* Component 50 (right central–temporal brain source).

where x_i is a generic vector and c_i an identifier that specifies to which of two classes the corresponding vector belongs, a SVM trained on such a data set is able to discriminate the point belonging to the two classes if they are separable by means of a hyper-plane

$$w \cdot x - b = 0 \tag{5.21}$$

where w is a vector pointing perpendicular to the mentioned hyper-plane. In order to achieve the *maximum margin* possible, one should find the two hyper-planes that are parallel to the one described by (5.21) and the most close to the points belonging to the two classes:

$$\begin{aligned} w \cdot x - b &= 1 \\ w \cdot x - b &= -1 \end{aligned} \tag{5.22}$$

The *support vectors* are the one containing the elements of the data set lying on these two planes. The main limitation of the SVM is that it can classify only linearly separable classes. Anyway this limitation can be easily overcome through the use of the *kernel trick*, a technique that allows to convert a linear classifier into a non–linear one by using a non–linear function to map the original observations into a higher-dimensional space: linear classification in the new space turns out to be equivalent to non–linear classification in the original space. More details on SVMs and their application to pattern recognition can be found in [113].

The the same front–end used with the HMM classifier was used with the SVMs (refer to Figure 5.18, Page 128). Also the data–set partitions for training and testing were the same (20 partition with 90% of the epochs used for training and 10% for testing, even distribution of selection and non–selection epochs). In every session one SVM was trained for each of the 59 channels. All the tests were repeated with four different kinds of SVM: a linear one and three non–linear ones using the kernel trick with different kernel functions (quadratic, polynomial and gaussian radial basis function). The training and test of the SVM have been performed using Matlab and the functions `svmtrain()` and `svmclassify()` from the Bioinformatics Toolbox.

The non–linear classified performed poorly, without going beyond the chance level of classification accuracy. On the other hand, as can be seen from the plot in Figure 5.24, the linear classifier provided good results. The figure shows, for each of the 59 channels, the accuracy averaged over the results from the 20 data–set partitions. It is easily seen that part of the channels score over 60%. In order to make a comparison, the 11 channels that provided the highest accuracy in the tests with the HMM classifiers, have been marked in red. It can be seen that most of the channels that provided good results with the HMMs, provide good accuracy also when classified with a linear SVM. This arises the hypothesis (still to be verified) that these components carry some kind of information that is directly related to the brain dynamics activated by the selection process. On the other hand, some channels that had high accuracy in the HMM tests, scored poorly with the SVMs, and vice versa. This last phenomenon could be due to the specific ability of each of the two classifiers of identifying a certain kind of content. This can be favorably exploited in the implementation of a system, by using both HMM and SVM classifiers, and feeding in each of them the channels that provide higher accuracy with

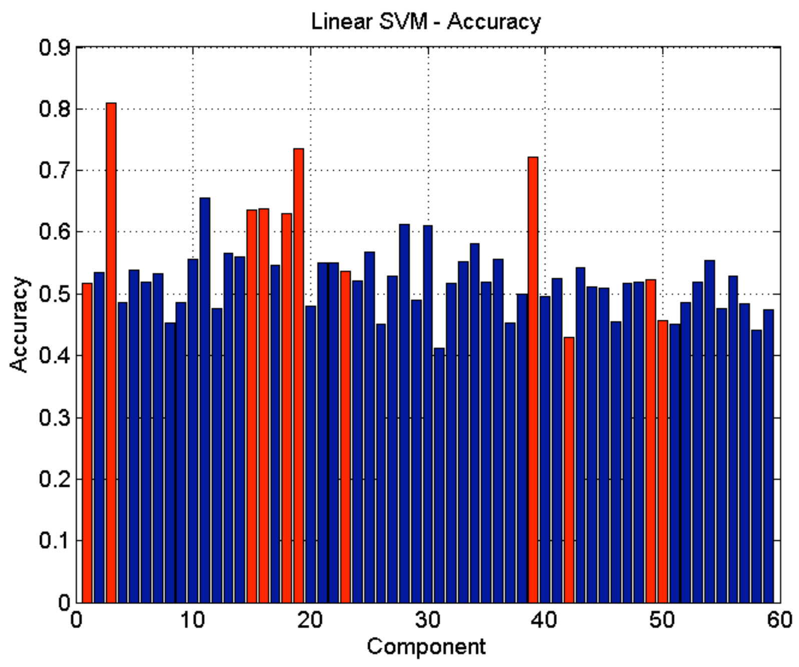


Figure 5.24: EEG Classification through linear SVM. Averaged accuracy for all the component classifiers. The channels that provided highest accuracy with the HMM classifiers are marked in red.

that particular classifier. However this last test confirmed the hypothesis, made firstly on the base of the results from the HMM tests, that the kind of EEG classification here proposed is feasible and could lay the basis for a new class of BCI based on spontaneous brain commands.

Hardware Tests

The HMM based EEG classifier described in the previous pages was implemented as a parallel array of HMM processors for comparison with the software implementation. The system required 108 HMMs and, according to the considerations already made in Section 4.3.2 (Page 92) the minimum operating frequency should be evaluated in order to decide which of the implementation solution proposed in 4.3.1 (Page 87) is suitable for the application. Figure 5.25 shows a plot of the minimum clock frequency required with $N_A = 108$ atomic HMMs and an observation period $T_{obs} = 10\ ms$ (according to Equation 4.8, Page 89). The plot shows that all the described implementations are capable of running the system because, even in the worst case situation of a single HMM processor mapped on the device, the minimum required clock frequency is less that 25 MHz, whereas the lowest working frequency among the shown implementations is 57 MHz (refer to Table 4.2, Page 92 and Table 4.3, Page 92). On the other hand it has to be noted that the parallel array of HMMs is effective in computation only when the same observation vector is provided to all the HMMs. The EEG classifier application to be

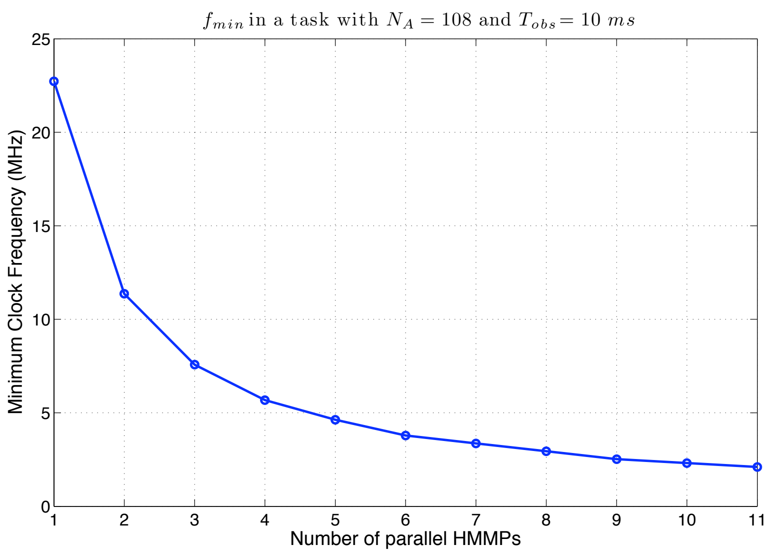


Figure 5.25: Minimum clock frequency in function of the number of parallel HMM processors for an implementation of a 59-channel EEG classifier. Fixed parameters: $N_A = 108$, $T_{obs} = 10\ ms$.

implemented here has to handle 59 different observation streams, and each observation vector is always processed by no more than 2 HMM. For this reason in this case it has no sense to use more than 2 HMM processors in parallel[5]. For this reason the tests have been performed only on the configurations with one and two HMM processors on a single device. Also, a slight variation on the system–level control of the architecture had to be made in order to deal with the multiple input streams: let us indicate with $(H_c^{\oplus}, H_c^{\ominus})$ a couple made up by the selection and non–selection HMM for channel c, and $\boldsymbol{O}_t^{(c)}$ the observation at time t for channel c. For each time step t the model parameters for the couple $(H_1^{\oplus}, H_1^{\ominus})$ is loaded and the observation $\boldsymbol{O}_t^{(1)}$ is processed; then the procedure is repeated for $(H_2^{\oplus}, H_2^{\ominus})$ to process $\boldsymbol{O}_t^{(2)}$, and so on, up to $(H_{N_C}^{\oplus}, H_{N_C}^{\ominus})$ to process $\boldsymbol{O}_t^{(N_C)}$, where N_C indicates the number of input channels (in this case $N_C = 59$). Only when all the channels have been processed for time step t, the time is incremented to $t+1$ and the cycle begins again reloading the models $(H_1^{\oplus}, H_1^{\ominus})$ for the first channel, in order to process $\boldsymbol{O}_{t+1}^{(1)}$. Apart from these changes in data feeding, the system's behavior is the same as in the ASR application.

The comparison has been made between the results from HTK's `HVite` and the hardware architecture implemented with three different word lengths (for score computations): 32, 24 and 16 bits. As already checked in 4.3.2 (Page 92) all the configurations of the hardware archi-

[5] The only way to process more than 2 HMMs in parallel in this application is to provide multiple observation buses to the architecture and route each of them to a couple of HMM processors. This has not been done in the present work, even because the number of buses is so high that any advantage would be lost because of the huge increment in needed hardware resources.

tecture had exactly the same functional behavior for a given word length (i.e. they provided the same results for the same input), so the results referred to a certain word length are applicable to any HMM processor configuration using the same number of bits for number representation.

From the the comparison of the results obtained with the software and hardware implementations it could be seen that the accuracy is really similar between the two. The mismatch introduced by the quantization in the hardware is, for this application, almost unnoticeable even with 16 bits of precision. The decrease in accuracy with respect to the `HVite`'s floating point software implementation is in fact of about 2% for the 16 bits implementation, and less than 1% for the 24 and 32 bits implementations.

It has therefore been shown that the architecture based on the developed HMM processors can be effectively applied also to BCI related classification tasks.

Chapter 6

Conclusions

In this thesis a new approach has been described for the development of human–computer interfaces. In particular the case of pattern recognition systems based on Hidden Markov Models have been taken into account.

The work has been focused on the development of dedicated hardware architectures, but also some new results have been obtained on the classification of electroencephalographic signals through the use of HMMs.

The original contributions of this thesis can be summarized as follows:

The Hardware Architecture: A system–level architecture has been developed to be used in HMM based pattern recognition systems. The architecture has been conceived in order to be able to work as a stand–alone system: its output could be fed directly in a system's command interpreter.

The HMM Processor: A VHDL description has been made of a flexible and completely reconfigurable hardware HMM processor and the design was successfully simulated. A parallel array of these processors is actually the core processing block of the developed architecture.

Hardware Mapping: Two suitable FPGA based, fast prototyping platforms have been identified to be the targets for the implementation tests. Different configurations of parallel HMM processor arrays have been set up and mapped on the target FPGAs. Some solutions have been selected to be the best in terms of balance between performance and resources utilization.

Functional Verification: A software HMM based pattern recognition system has been chosen to be the reference system for the functionality of the implemented subsystems. A set of tests have been developed with the aim to test the correct functionality of the hardware. The implemented system was compared to the reference system on the basis of the tests' results, and it was found that the behavior was the one expected and the required functionality was correctly achieved.

Application tests: The implementation of the parallel HMM array was tested through its application to two real–world applications: a speech recognition task and a brain–computer interface task. In both cases the architecture showed to be functionally suitable and powerful enough to handle the task without problems. The application of the hardware processing to speech recognition opens new perspectives in the design of this kind of systems because of the dramatic increment in performance. The application to brain–computer interface is really interesting because of a new approach in the classification of EEG that shows how could be possible a future development of interfaces based on the classification of spontaneous thought.

The possible evolution directions of the work started with this thesis are many. Effort could be spent of the implementation of the developed architecture as a stand–alone reconfigurable system suitable for any kind of HMM–based pattern recognition task. The potential performance of such a system could open the way to extremely complex real–time pattern recognition systems, and thus to the realization of truly multimodal interfaces, with a variety of applications, from space to aid systems for the impaired.

Acknowledgments

Finally the last page! Many are the people I would like to thank for the support I received during this long research period.

First of all I would like to thank all the people in the group of Digital Electronic Systems at the university of Rome, "Tor Vergata". In particular thanks to Prof. Gian Carlo Cardarilli, for the chance I had to work on this project an for the support. Thanks also to all the people that have been working besides me in the labs during these three year: Andrea, Domenico, Gianluca and all the others. They have been invaluable either during the (usually) fruitful technical and scientific discussions and during the few spare time: their mood and company made everything much easier.

Thanks also to the people from STM's Advanced System Technologies Labs, in Agrate, with whom I shared the work on speech recognition during the first part of my research period. Very special thanks to Eng. Luigi Arnone, that helped me mostly in dealing with the things I found completely incomprehensible (and that seemed to be his favorites).

My appreciation goes also to all the people from the LANOS group at the EPFL: I spent five unforgettable months in Lausanne also thanks to their availability and competence. Thanks mostly to Prof. Martin Hasler because he managed to dedicate some of his precious time to me even if he had really a few: I really felt at home, and working in LANOS was a pleasure.

Talking about my permanence in Switzerland, I must thank also all the friends I met inside and outside of the Valentin: probably my work was so productive because I had always a really good time when I was NOT working!

Thanks to Maria, that had lots of patience with me. She have been my main cure for the stress and the losses of motivation.

And finally, as usually, thanks to my family, that support me in everything I do... well, I should say "in every good thing I do", and I think this is the case.

Thanks also to everyone I forgot to thank in this lines: hope you will understand.

Thank you.

Ringraziamenti

Ovviamente non ho resistito alla tentazione di spendere due parole anche nella mia lingua madre.

Bene, finalmente l'ultima pagina! Penso sia giusto dedicarla alle persone che mi sono state in qualche modo d'aiuto durante questo lungo periodo di ricerca.

Innanzitutto vorrei ringraziare il gruppo di Elettronica Digitale dell'Università degli Studi di Roma, "Tor Vergata". Grazie in particolare al Prof. Gian Carlo Cardarilli, per avermi offerto la possibilità di lavorare a questo progetto e per il supporto. Grazie anche a tutte le persone a cui ho lavorato accanto durante i miei tre anni di permanenza nel dipartimento: Andrea, Domenico, Gianluca e tutti gli altri che sono "apparsi" per periodi più o meno lunghi. Tutti loro stati indispensabili al mio lavoro, sia per i produttivi scambi di idee che per la loro compagnia e il loro buon umore (salvo rari casi!).

Grazie anche alle persone di STM — Advanced System Technologies (Agrate) con cui ho avuto il piacere di collaborare, in particolare durante il mio lavoro sul riconoscimento vocale. Un ringraziamento particolare all'Ing. Luigi Arnone che mi è stato particolarmente d'aiuto per tutti i problemi che mi apparivano incomprensibili, e con cui lui, puntualmente, sembra avere una dimestichezza particolare!

Grazie a tutti i membri del LANOS all'EPFL: ho trascorso cinque mesi indimenticabili a Losanna anche grazie a loro, alla loro disponibilità e alla loro competenza. Un grazie particolare va al Prof. Martin Hasler, che è riuscito a dedicarmi parte del su prezioso tempo: posso dire di essermi sentito veramente a casa, e lavorare al LANOS è stato un vero piacere.

Restando sempre sul tema della mia permanenza in Svizzera, voglio ringraziare anche tutti gli amici conosciuti sia fuori che dentro il Valentin: probabilmente il mio lavoro è stato cosí produttivo grazie al tempo libero che trascorrevo con loro!

Grazie a Maria, che ha avuto davvero molta pazienza con me. È stata davvero il mio rimedio principale contro lo stress e i crolli di motivazione.

Un'ultima parola voglio spenderla per ringraziare la mia famiglia che, come sempre, mi supporta in tutto ciò che faccio (o meglio: in tutto ciò che faccio di buono!).

Grazie anche a tutti quelli che ho dimenticato in queste righe: spero che lo stress valga come scusa valida.

Grazie!

List of Figures

1.1 A block diagram that describes a generic pattern recognition system. 8

2.1 A graphical representation of a generic 5-states markov chain 14
2.2 Duration probability density function for a first order, time invariant Markov process 15
2.3 A comparison of observation output between Markov Chains and HMMs. 17
2.4 Markov models for a coin toss experiment 18
2.5 Representation of the *N–Jars* experiment for the production of color sequences, and its relations with HMM emission probabilities. 19
2.6 Graphical illustration of the recursion step in the calculus of the forward variables. 23
2.7 Trellis representation for a fully connected 5–state HMM over time 26
2.8 A 4–state Bakis (or left–right) HMM. 30
2.9 A HMM with a parallel Bakis topology. 31
2.10 Examples of application for null transitions 33
2.11 Representation of the *deleted interpolation* process as an extended HMM. 38

3.1 Representation of a search space building process 43
3.2 A standard HMM–based pattern recognition system. 44
3.3 Interpretation of the HMM decoding as a layered process through the use of the Token Passing conceptual model. 46
3.4 An HMM–based pattern recognition system that implements the Token Passing conceptual model. 47
3.5 The main logical blocks that make up a pattern recognition system's architecture. 48
3.6 Representation of a parallel architecture implementing the Token Passing conceptual model. 50
3.7 A possible hardware platform on which a Token Passing based pattern recognition system can be implemented. 51
3.8 System level architecture that use static HMM processors. 52
3.9 System level architecture that use dynamic allocation of HMM processors 54
3.10 Representation of a typical chain of records created by a token being passed through the state space 55

3.11 Illustration of the behavior of dummy states in HMMs 55
3.12 Architecture of the HMM processor. 56
3.13 Finite State Machine that implements the overall control for the HMM processor architecture. 57
3.14 Architecture of the *Token Passer* module 59
3.15 Architecture of the Emission Probability Computer module 60
3.16 Architecture of the *Token Life Monitor* module. 64

4.1 Typical design flow adopted for fast–prototyping hardware platforms 68
4.2 The HMM processor entity 70
4.3 Expected timing diagram for model parameters loading into the internal HMM processor's memory. 71
4.4 Expected signaling for a generic data transfer through a bus with two control signals. 73
4.5 Detailed architecture of the HMM processor, as described in VHDL for implementation. 74
4.6 The `hmmp_control` entity 75
4.7 Detailed view of the state machine that controls the HMM processor. 76
4.8 Detailed view of the state machine that controls the loading of model parameters. 77
4.9 Storage structures implemented in the architecture 78
4.10 The entity `emprob_computer` 79
4.11 Details of the `emprob_computer` entity's internal architecture 80
4.12 The FSM implemented by the Emission Probability Computer's control unit `ep_cpu_ctrl`. 81
4.13 Hardware evaluation of a single gaussian component from a mixture 83
4.14 The `tokenpasser` entity 84
4.15 The FSM that controls the `tokenpasser` entity. 85
4.16 The `token_life_monitor` entity 86
4.17 Entities for the `token_updater` and `eval_xit_state` modules 87
4.18 Xilinx Virtex FF1152 Proto Board used for the implementation tests on the Virtex II 3000 device 88
4.19 Waveforms obtained from the simulation of the HMM Processor's VHDL RTL description 90
4.20 Architecture's minimum clock frequency in function of the observation period . 91
4.21 Different configurations of the HMM Processor array placed and routed onto a Virtex II 3000. 93
4.22 Different configurations of the HMM Processor array placed and routed onto a Virtex 4 SVX55. 94
4.23 Software architecture of the HTK Toolkit. 95
4.24 HTK processing stages, along with the tools involved in each stage. 96

4.25 Diagram of the Xilinx Virtex II Proto Board used for the implementation tests (HW-AFX-FF1152-200). 97

5.1 Representation of the main components of a generic ASR system. 102
5.2 Block diagram of a MFCC front–end for speech recognition. 105
5.3 Hamming windowing function used in the front–end. 107
5.4 A bank of triangular filters evenly distributed along the MEL frequency scale. . . 108
5.5 The topology of the HMM used to model each phoneme in the phonetic recognizer 110
5.6 Comparison of hardware and software speech recognition performance 111
5.7 The international $10 - 20$ system for EEG electrode placement 116
5.8 Location and nomenclature of the intermediate 10% EEG electrodes 116
5.9 Examples of EEG waveforms in the various frequency bands. 117
5.10 Representation of the experimental setup used for the EEG selection recordings. 119
5.11 Timeline for a single trial in the EEG selection experiment. 120
5.12 Possible locations for the appearance of targets in a 8 target trial set. 121
5.13 Samples of the reports produced during an EEG recording session 122
5.14 A sample of the EEG signal recorded during the experiments. Also the events are marked. 123
5.15 Plots for independent component 1 extracted from EEG 125
5.16 Plots for independent component 3 extracted from EEG 126
5.17 Plots for independent component 12 extracted from EEG 127
5.18 Feature extraction front–end used in the EEG pattern recognition system. 128
5.19 The topology of the HMMs used in the EEG classifier 130
5.20 Results of EEG Classification through HMM - Independent components 1 and 3 . 131
5.21 EEG Classification through HMM - Independent components 16 and 18 132
5.22 EEG Classification through HMM - Independent components 15 and 19 133
5.23 Results of EEG Classification through HMM - Independent components 23, 39, 49 and 50 134
5.24 Results of EEG Classification through SVM 136
5.25 Minimum clock frequency in function of the number of parallel HMM processors in a 59-channel EEG classifier 137

List of Tables

3.1 Computation of emission probabilities: comparison of the number of operations needed in function of the method used 63

4.1 Details of the Xilinx FPGAs chosen as target devices for the architecture. 88
4.2 Details of mapping on Virtex II device (XC2V3000) 92
4.3 Details of mapping on Virtex 4 device (XC4VSX55) 92

5.1 Parameters that characterize a generic speech recognition system 102
5.2 The *Sampa* phonetic alphabet for the italian language 106

Bibliography

[1] L. Bannon and K. Schmidt. *CSCW — four characters in search of a context. Studies in computer supported cooperative work — theory, practice and design.* J. M. Bowers and S. Benford. Amsterdam, North Holland., 1991. 1.1

[2] Stuart K. Card, Thomas P. Moran, and Allen Newell. *The Psychology of Human–Computer Interaction*. Erlbaum, Hillsdale, 2 edition, 1983. 1.1

[3] Ronald M. Baecker, Jonathan Grudin, William A. S. Buxton, and Saul Greenberg. *Readings in human–computer interaction. Toward the Year 2000*. Morgan Kaufmann, San Francisco, 2 edition, 1995. 1.2

[4] T. Mandel. *The elements of user interface design*. Wiley New York, 2 edition, 1997. 1.2.1

[5] W. Schiff. *Tactual Perception: A Sourcebook*. Cambridge University Press, 1982. 1.2.1

[6] M. Kurze. Rendering drawings for interactive haptic perception. *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 423–430, 1997. 1.2.1

[7] WK Edwards, ED Mynatt, and K. Stockton. *Providing access to graphical user interfaces—not graphical screens*. ACM Press New York, NY, USA, 1994. 1.2.1

[8] SD Laycock and AM Day. Recent Developments and Applications of Haptic Devices. *Computer Graphics Forum*, 22(2):117–132, 2003. 1.2.1

[9] S. Oviatt. Multimodal interfaces. *The Human-Computer Interaction Handbook: Fundamentals, Evolving Technologies and Emerging Applications*, pages 286–304, 2003. 1.2.1, 1.4.1

[10] L. Rabiner and B.H. Juang. *Fundamentals of speech recognition*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1993. 1.3.1, 5.1.1

[11] W.B. Kleijn, K.K. Paliwal, et al. *Speech coding and synthesis*. Elsevier, 1995. 1.3.1

[12] A. Collins, J.S. Brown, and K.M. Larkin. *Inference in Text Understanding*. University of Illinois at Urbana-Champaign, 1977. 1.3.1

[13] L. R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989. 1.3.1

[14] R.P. Lippmann. Review of neural networks for speech recognition. *Neural Computation*, 1(1):1–38, 1990. 1.3.1

[15] Alessandro Malatesta. *Studio e definizione di architetture harware per riconoscitori vocali speaker–independent per parlato continuo e vocabolario esteso*. Master's thesis, Università degli studi di Roma, "Tor Vergata", July 2003. 1.3.1, 3.1, 5.1.1

[16] L. Comerford, D. Frank, P. Gopalakrishnan, R. Gopinath, and J. Sedivy. The IBM Personal Speech Assistant. *Acoustics, Speech, and Signal Processing, 2001. Proceedings.(ICASSP'01). 2001 IEEE International Conference on*, 1, 2001. 1.3.1

[17] A. Averbuch, L. Bahl, R. Bakis, P. Brown, G. Daggett, S. Das, K. Davies, S. De Gennaro, P. de Souza, E. Epstein, et al. Experiments with the Tangora 20,000 word speech recognizer. *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP'87.*, 12, 1987. 1.3.1

[18] J.R. Wolpaw, N. Birbaumer, W.J. Heetderks, D.J. McFarland, P.H. Peckham, G. Schalk, E. Donchin, L.A. Quatrano, C.J. Robinson, and T.M. Vaughan. Brain–Computer Interface Technology: A Review of the First International Meeting. *IEEE TRANSACTIONS ON REHABILITATION ENGINEERING*, 8(2), 2000. 1.3.1

[19] E.E.D.T. Niedermeyer, FH Lopes Da Silva, and FH Lopes da Silva. *Electroencephalography: Basic Principles, Clinical Applications, and Related Fields*. Lippincott Williams & Wilkins, 2005. 1.3.1

[20] H. Jasper. Electrocorticography. *Epilepsy and the functional anatomy of the human brain. Boston: Little, Brown*, pages 692–738, 1954. 1.3.1

[21] EA DeYoe, P. Bandettini, J. Neitz, D. Miller, and P. Winans. Functional magnetic resonance imaging (FMRI) of the human brain. *J Neurosci Methods*, 54(2):171–87, 1994. 1.3.1

[22] K. Tanaka, K. Matsunaga, and HO Wang. Electroencephalogram-Based Control of an Electric Wheelchair. *Robotics, IEEE Transactions on [see also Robotics and Automation, IEEE Transactions on]*, 21(4):762–766, 2005. 1.3.1

[23] E.C. Leuthardt, K.J. Miller, G. Schalk, R.P.N. Rao, and J.G. Ojemann. Electrocorticography-Based Brain Computer Interface—The Seattle Experience. *IEEE TRANSACTIONS ON NEURAL SYSTEMS AND REHABILITATION ENGINEERING*, 14(2):195, 2006. 1.3.1

[24] B. Blankertz, K.R. Muller, G. Curio, TM Vaughan, G. Schalk, JR Wolpaw, A. Schlogl, C. Neuper, G. Pfurtscheller, T. Hinterberger, et al. The BCI competition 2003: progress and perspectives in detection and discrimination of EEG single trials. *Biomedical Engineering, IEEE Transactions on*, 51(6):1044–1051, 2004. 1.3.1

[25] T. Zinßer, C. Graßl, and H. Niemann. Efficient Feature Tracking for Long Video Sequences. *Pattern Recognition, 26th DAGM Symposium*, 3175:326–333, 2004. 1.3.1

[26] Y. Wu and T.S. Huang. Vision-Based Gesture Recognition: A Review. *Urbana*, 51:61801, 1999. 1.3.1

[27] I. Cohen, N. Sebe, A. Garg, L.S. Chen, and T.S. Huang. Facial expression recognition from video sequences: temporal and static modeling. *Computer Vision and Image Understanding*, 91(1-2):160–187, 2003. 1.3.1

[28] A. Duchowski et al. *Eye Tracking Methodology: theory and practice*. Springer, 2003. 1.3.1

[29] J.J. Garrett. Ajax: A New Approach to Web Applications. *Archived at http://www. adaptivepath. com/publications/essays/archives/000385. php*, 2005. 1.3.2

[30] D.A. Chappell and T. Jewell. *Java Web services*. O'Reilly Sebastopol, CA, 2002. 1.3.2

[31] R. Vertegaal. Attentive user interfaces: Introduction. *Communications of the ACM*, 46(3):30–33, 2003. 1.3.2

[32] J. Nielsen. Noncommand user interfaces. *Communications of the ACM*, 36(4):83–99, 1993. 1.3.2

[33] T. Sharon, H. Lieberman, and T. Selker. A zero-input interface for leveraging group experience in web browsing. *Proceedings of the 8th international conference on Intelligent user interfaces*, pages 290–292, 2003. 1.3.2

[34] B.B. Bederson and J.D. Hollan. Pad++: a zooming graphical interface for exploring alternate interface physics. *Proceedings of the 7th annual ACM symposium on User interface software and technology*, pages 17–26, 1994. 1.3.2

[35] M. Pomplun, N. Ivanovic, E.M. Reingold, and J. Shen. Empirical Evaluation of a Novel Gaze-Controlled Zooming Interface. *Usability Evaluation and Design: Cognitive Engineering, Intelligent Agents and Virtual Reality. Proceedings of the 9th International Conference on Human-Computer Interaction*, 2001. 1.3.2

[36] VS Avduevskii, VI Polezhaev, and VV Sazonov. Mechanics of Zero Gravity and Gravitationally Sensitive Systems (First Issue). *Cosmic Research*, 39(2):105–105, 2001. 1.4

[37] H. Spieler. Introduction to radiation-resistant semiconductor devices and circuits. *AIP Conference Proceedings*, 390(1):23–49, 2006. 1.4

[38] P. Alfke and R. Padovani. Radiation tolerance of high-density FPGAs. *ELECTRON ENG LONDON*, 72(880):2, 2000. 1.4

[39] P.K. Lala. *Self-checking and fault-tolerant digital design*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2000. 1.4

[40] R.J. White and M. Averner. Humans in space. *Nature*, 409:1115–1118, 2001. 1.4

[41] CM Lathers, JB Charles, and MW Bungo. Humans in space: NASA's contributions to medical technology and health care. *The Journal of Clinical Pharmacology*, 30(3):223–225, 1990. 1.4

[42] Nasa — national aeronautics and space administration, website. `http://www.nasa.gov`. 1.4

[43] J.D. Schierman, D.G. Ward, J.R. Hull, and N. Gandhi. Intelligent Guidance and Trajectory Command Systems for Autonomous Space Vehicles. *AIAA 1 st Intelligent Systems Technical Conference*, pages 1–24, 2004. 1.4.1

[44] J. G. Carbonell, R. S. Michalski, and T. M. Mitchell. An overview of machine learning. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, pages 3–23. Springer, Berlin, Heidelberg, 1984. 1.5

[45] T.M. Mitchell. *Machine Learning*. McGraw-Hill Higher Education, 1997. 1.5

[46] K. Fukunaga. *Introduction to Statistical Pattern Recognition*. Academic Pr, 1990. 1.5

[47] VN Vapnik. An overview of statistical learning theory. *Neural Networks, IEEE Transactions on*, 10(5):988–999, 1999. 1.5.1

[48] G.E.P. Box and G. Jenkins. *Time Series Analysis, Forecasting and Control*. Holden-Day, Incorporated, 1990. 1.5.2

[49] H. Kantz and T. Schreiber. *Nonlinear Time Series Analysis*. Cambridge University Press, 2004. 1.5.2

[50] H. Bunke. Structural and syntactic pattern recognition. *Handbook of pattern recognition & computer vision*, pages 163–209, 1993. 1.5.2

[51] Georg Dorffner. Neural Networks for Time Series Processing. *Neural Network World*, 6(4):447–468, 1996. 1.5.2

[52] H. Byun and S.W. Lee. Applications of Support Vector Machines for Pattern Recognition: A Survey. *SVM*, pages 213–236, 2002. 1.5.2

[53] A. Baraldi and P. Blonda. A survey of fuzzy clustering algorithms for pattern recognition. II. *Systems, Man and Cybernetics, Part B, IEEE Transactions on*, 29(6):786–801, 1999. 1.5.2

[54] L.E. Baum, T. Petrie, G. Soules, and N. Weiss. A Maximization Technique Occurring in the Statistical Analysis of Probabilistic Functions of Markov Chains. *The Annals of Mathematical Statistics*, 41(1):164–171, 1970. 2.1, 2.2.3, 2.2.3

[55] A. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *Information Theory, IEEE Transactions on*, 13(2):260–269, 1967. 2.2.2

[56] TK Moon. The expectation-maximization algorithm. *Signal Processing Magazine, IEEE*, 13(6):47–60, 1996. 2.2.3

[57] L. Liporace. Maximum likelihood estimation for multivariate observations of Markov sources. *Information Theory, IEEE Transactions on*, 28(5):729–734, 1982. 2.2.5

[58] B.H. Juang, S. Levinson, and M. Sondhi. Maximum likelihood estimation for multivariate mixture observations of markov chains (Corresp.). *Information Theory, IEEE Transactions on*, 32(2):307–309, 1986. 2.2.5

[59] S. Young, G. Evermann, M. Gales, T. Hain, D. Kershaw, X. Liu, G. Moore, J. Odell, D. Ollason, D. Povey, V. Valtchev, and P. Woodland. *The HTKbook (for HTK Version 3.4)*. Cambridge University Engineering Department, December 2006. 2.3.1, 4.3.2

[60] F. Jelinek and R.L. Mercer. Interpolated estimation of Markov source parameters from sparse data. *Pattern Recognition in Practice*, 23:381, 1980. 2.3.2

[61] R. Dujari. *Parallel Viterbi search algorithm for speech recognition*. Master's thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1992. 3.1

[62] M.K. Ravishankar. *Efficient Algorithms for Speech Recognition*. PhD thesis, Carnegie Mellon University, 2005. 3.1

[63] GC Cardarilli, A. Malatesta, M. Re, L. Arnone, and S. Bocchio. Hardware oriented architectures for continuous–speech speaker–independent ASR systems. *Signal Processing and Information Technology, 2004. Proceedings of the Fourth IEEE International Symposium on*, pages 346–352, 2004. 3.1

[64] A. Malatesta, GC Cardarilli, M. Re, L. Arnone, and S. Bocchio. Development and validation of hardware architectures for real–time high–performance speech recognition systems. *Research in Microelectronics and Electronics, 2005 PhD*, 2, 2005. 3.1

[65] D. Bianchi, GC Cardarilli, A. Del Re, A. Malatesta, and M. Re. FPGA implementation of a general purpose HMM processor based on Token passing algorithm. *Circuit Theory and Design, 2005. Proceedings of the 2005 European Conference on*, 1, 2001. 3.1

[66] Sampa: Computer Readable Phonetic Alphabet.
`http://www.phon.ucl.ac.uk/home/sampa/index.html`. 3.1.1, 5.1.1, 5.1.2

[67] S.J. Young, NH Russell, and JHS Thornton. Token Passing: A Simple Conceptual Model for Connected Speech Recognition Systems. Technical report, University of Cambridge, Dept. of Engineering, July 1989. 3.1.2

[68] B. Delaney, N. Jayant, M. Hans, T. Simunic, and A. Acquaviva. A low–power, fixed–point, front–end feature extraction for adistributed speech recognition system. *Acoustics,*

Speech, and Signal Processing, 2002. Proceedings.(ICASSP'02). IEEE International Conference on, 1, 2002. 3.2

[69] N. Kumar, W. Himmelbauer, G. Cauwenberghs, and AG Andreou. An analog VLSI chip with asynchronous interface for auditory feature extraction. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on [see also Circuits and Systems II: Express Briefs, IEEE Transactions on]*, 45(5):600–606, 1998. 3.2

[70] T.H. Cormen, R.L. Rivest, C.E. Leiserson, and C. Stein. *Introduction to algorithms*. MIT Press, 2001. 3.2.1

[71] Altera, Development Kits.
`http://www.altera.com/products/devkits/kit-index.html`. 3.2.2

[72] Xilinx, Development Boards.
`http://www.xilinx.com/products/devboards/index.htm`. 3.2.2

[73] Altera, Nios II Soft Processor.
`http://www.altera.com/products/ip/processors/nios2/ni2-index.html`. 3.2.2

[74] Xilinx, MicroBlaze Soft Processor.
`http://www.xilinx.com/xlnx/xebiz/designResources/ip_product_details.jsp?key=micro_blaze`. 3.2.2

[75] J.F. Cheng and T. Ottosson. Linearly approximated log-MAP algorithms for turbo decoding. *Vehicular Technology Conference Proceedings, 2000. VTC 2000-Spring Tokyo. 2000 IEEE 51st*, 3, 2000. 3.4.2

[76] P. Robertson, E. Villebrun, and P. Hoeher. A comparison of optimal and sub-optimal MAP decoding algorithmsoperating in the log domain. *Communications, 1995. ICC 95 Seattle, Gateway to Globalization, 1995 IEEE International Conference on*, 2, 1995. 3.4.2

[77] R. Andraka. A survey of CORDIC algorithms for FPGA based computers. *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pages 191–200, 1998. 3.4.2, 4.2.3

[78] X. Hu, R.G. Harber, and S.C. Bass. Expanding the range of convergence of the CORDIC algorithm. *IEEE Transactions on Computers*, 40(1):13–21, 1991. 3.4.2

[79] VASG: VHDL Analysis and Standardization Group. `http://www.eda.org/vhdl-200x`. 4.1.1

[80] Aldec active hdl, design entry software. `http://www.aldec.com/products/active-hdl`. 1

[81] Modelsim hdl simulator. `http://www.model.com`. 4.1.1

[82] HTK, The Hidden Markov Model Toolkit. `http://htk.eng.cam.ac.uk`. 4.1.1, 4.2.3, 4.3.2

[83] Xilins programmable logic devices. `http://www.xilinx.com`. 4.1.1

[84] Xilinx ise foundation, integrated development environment. `http://www.xilinx.com/ise/logic_design_prod/foundation.htm`. 4.1.1

[85] R. Naeaetaenen, A. Lehtokoski, M. Lennes, M. Cheour, M. Huotilainen, A. Iivonen, M. Vainio, P. Alku, R.J. Ilmoniemi, A. Luuk, et al. Language-specific phoneme representations revealed by electric and magnetic brain responses. *Nature*, 385(6615):432–434, 1997. 5.1.1

[86] PC Woodland, JJ Odell, V. Valtchev, and SJ Young. Large vocabulary continuous speech recognition using HTK. *Acoustics, Speech, and Signal Processing, 1994. ICASSP-94., 1994 IEEE International Conference on*, 2, 1994. 5.1.1

[87] W. Walker, P. Lamere, P. Kwok, B. Raj, R. Singh, E. Gouvea, P. Wolf, and J. Woelfel. Sphinx-4: A flexible open source framework for speech recognition. *Sun Microsystems Laboratories, Tech. Rep. TR-2004-139, November*, 2004. 5.1.1

[88] R. Roth, L. Gillick, J. Orloff, F. Scattone, G. Gao, S. Wegmann, and J. Baker. Dragon Systems' 1994 Large Vocabulary Continuous Speech Recognizer. *Proc. ARPA Spoken Language Systems Technology Workshop, Austin*, 1995. 5.1.1

[89] R. Vergin, D. O'Shaughnessy, and A. Farhat. Generalized Mel Frequency Cepstral Coefficients for Large-Vocabulary Speaker-Independent Continuous-Speech Recognition. *IEEE TRANSACTIONS ON SPEECH AND AUDIO PROCESSING*, 7(5):525, 1999. 5.1.2

[90] S. Imai. Cepstral analysis synthesis on the mel frequency scale. *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP'83.*, 8, 1983. 5.1.2

[91] H. Berger. Uber das Elektrenkephalogramm des Menschen. *Arch Psychiatr Nervenkr*, 87:527–570, 1929. 5.2.1

[92] HC Sing, MA Kautz, DR Thorne, SW Hall, DP Redmond, DE Johnson, K. Warren, J. Bailey, and MB Russo. High-frequency EEG as measure of cognitive function capacity: a preliminary report. *Aviat Space Environ Med*, 76(7 Suppl):C114–35, 2005. 5.2.1

[93] HC Sing, MA Kautz, DR Thorne, SW Hall, DP Redmond, and MB Russo. High frequency EEG as a potential indicator of alertness, attention and cognition. *Aviation Space and Environmental Medicine*, 77(3):83, 2006. 5.2.1

[94] K. Tanji, K. Suzuki, A. Delorme, H. Shamoto, and N. Nakasato. High-Frequency γ-Band Activity in the Basal Temporal Cortex during Picture-Naming and Lexical-Decision Tasks. *Journal of Neuroscience*, 25(13):3287–3293, 2005. 5.2.1

[95] P. Nunez. *Electric fields of the brain*. New York: Oxford University Press, 1981. 5.2.1

[96] W. Gerstner and W.M. Kistler. *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press, 2002. 5.2.1

[97] H.H. Jasper. The ten-twenty electrode system of the International Federation. *Electroencephalography and Clinical Neurophysiology*, 10(1):371–375, 1958. 5.2.1

[98] F. Sharbrough, GE Chatrian, and RP Lesser. American Electroencephalographic Society Guidelines for Standard Electrode Position Nomenclature. *J. Clin. Neurophysiol*, 8:200–202, 1991. 5.2.1

[99] R.L. Gilmore. *American Electroencephalographic Society Guidelines in Electroencephalography, Evoked Potentials, and Polysomnography*. Raven Press, 1994. 5.2.1

[100] G.E. Fabiani, D.J. McFarland, J.R. Wolpaw, and G. Pfurtscheller. Conversion of EEG Activity Into Cursor Movement by a Brain–Computer Interface (BCI). *IEEE TRANSACTIONS ON NEURAL SYSTEMS AND REHABILITATION ENGINEERING*, 12(3):331, 2004. 5.2.2

[101] J.R. Wolpaw and D.J. McFarland. Control of a two-dimensional movement signal by a noninvasive brain-computer interface in humans. *Proceedings of the National Academy of Sciences*, 101(51):17849–17854, 2004. 5.2.2

[102] Visual mouse, face tracking freeware software. `http://www.mousevision.com`. 5.2.2

[103] The python programming language. `http://www.python.org/`. 5.2.2

[104] Vision egg, python library for vision research experiments. `http://www.visionegg.org/`. 5.2.2

[105] S.C. de Oliveira, A. Thiele, and K.P. Hoffmann. Synchronization of Neuronal Activity during Stimulus Expectation in a Direction Discrimination Task. *Journal of Neuroscience*, 17(23):9248–9260, 1997. 5.2.2

[106] The matworks — matlab®. `http://www.mathworks.com/products/matlab`. 5.2.2

[107] Eeglab — matlab toolbox for electrophysiological data processing. `http://www.sccn.ucsd.edu/eeglab`. 5.2.2

[108] A. Papoulis. *Probability, random variables and stochastic processes*. McGraw-Hill, 3rd edition, 1991. 5.2.2

[109] A. Hyvarinen and E. Oja. Independent component analysis: Algorithms and applications. *Neural Networks*, 13(4):411–430, 2000. 5.2.2

[110] L. Zhukov, D. Weinstein, and C. Johnson. Independent component analysis for EEG source localization. *Engineering in Medicine and Biology Magazine, IEEE*, 19(3):87–96, 2000. 5.2.2

[111] R. Vigário, J. Särelä, V. Jousmäki, M. Hämäläinen, and E. Oja. Independent Component Approach to the Analysis of EEG and MEG Recordings. *IEEE TRANSACTIONS ON BIOMEDICAL ENGINEERING*, 47(5):589, 2000. 5.2.2

[112] C.A. Joyce, I.F. Gorodnitsky, and M. Kutas. Automatic removal of eye movement and blink artifacts from EEG data using blind component separation. *Psychophysiology*, 41(2):313–325, 2004. 5.2.2

[113] C.J.C. Burges. A Tutorial on Support Vector Machines for Pattern Recognition. *Data Mining and Knowledge Discovery*, 2(2):121–167, 1998. 5.2.2

Index

algorithm
  choice, 42
algorithmic framework, 42
applications
  ASR, 100
  BCI, 112
  to human–machine interface, 99
architecture, 41
  component description, 58
  derivation, 49
  design flow, 67
  emission probability computer, 59
  HMM processor, 56
  implementation, 67
  refinements, 51
  testing, 67, 87
  token life monitor, 64
  token passer, 58
ASR
  applications, 100
  implementation
    phonetic recognizer, 104
  introduction, 100

BCI
  applications, 112
  introduction, 112
brain–computer interfaces, *see* BCI

choice of the model, 39
classification
  EEG, 127
  statistical, 9
  use of HMM, 22
classifier
  EEG
    implementation, 117
components
  of the architecture, 58
computation
  of emission probability, 79
conceptual model
  towards architecture, 49
continuous observation space
  HMM, 31
control
  HMM processor, 72

dedicated hardware, 41
design flow, 67
design tools, 69
discrete Markov processes, 13

EEG
  classification, 127
  classifier
    implementation, 117
  hardware tests, 136
  principles, 115
  recording, 118
  signal processing, 122
electroencephalogram, *see* EEG
electroencephalography, *see* EEG
emission probability
  computation, 79
emission probability computer, 59

finalization, 86
functional verification, 92

hands–free interfaces, 4
hardware, 41
  platform, 49
  test
    EEG, 136
hardware implementation, 47
HCI, 3
hidden layer
  of HMM, 25
hidden Markov models, *see* HMM
HMI, 1, 2
  applications, 99
HMM, 7
  characterization, 18
  choice of initial parameters, 37
  choice of the model, 39
  continuous, 31
  for classification, 22
  from Markov chains to, 16
  hidden layer, 25
  in pattern recognition, 13, 20
  introduction, 13
  limitations, 37
  multiple observation sequences, 36
  null transitions, 33
  optimal state sequence, 25
  processor, 54
    architecture, 56
    control, 72
    top–level, 70
  state duration, 32, 35
  tied states, 32, 34
  token passing, 44
  topologies, 30
  training, 27
    initial parameters, 37
  undertraining, 37
human–computer interfaces, 3
human–machine interaction, *see* HMI
human–machine interfaces, *see* HMI

implementation
  architecture, 67
  details, 70
  EEG classifier, 117
  phonetic recognizer, 104
  strategies, 69
  system, 87
  token passing, 83
initial parameters, 37
interfaces
  evolution of, 5
  hands–free, 4
  human–computer, 3
  human–machine, 2
  in space systems, space systems
    interfaces for, 6
  multimodal, 6
  new generation, 4
  user, 3

lifespan
  of token
    notification, 85

Markov
  discrete process, 13
Markov chains, 13
  extension to HMM, 16
memory, 76
multimodal interfaces, 6
multiple observation sequences, 36

notification
  of token lifespan, 85
null transitions, 33

optimal state sequence
  in HMM, 25
output of result, 86

pattern recognition
  HMM, 13, 20
  of time series, 10
pattern recognition systems, 8

phonetic recognizer
  implementation, 104
platform definition, 49
processing blocks, 76
processor
  HMM, 54
    architecture, 56
    control, 72
    top–level, 70

recording
  EEG, 118
refinement of the architecture, 51
result output, 86

shared distributions, *see* tied states
shared mixtures, *see* tied states
signal processing
  EEG, 122
speech recognition, *see* ASR
state duration, 32, 35
statistical classification, 9
system
  implementation, 87
system overview, 48

testing
  architecture, 67, 87
tests
  hardware
    EEG, 136
tied states, 32, 34
time–series, 7
  in pattern recognition, 10
TLM, *see* token life monitor
token
  lifespan of
    notification, 85
token life monitor, 64
token passer, 58
token passing, 44
  implementation, 83
tokens
  tracking, 53
tools
  design, 69
top–level, 70
topology
  HMM, 30
tracking
  of tokens, 53
training
  HMM, 27
  initial parameters, 37

undertraining, 37
user interfaces, 3

verification
  functional, 92