Signal Processing for DNA Sequencing

by

Petros T. Boufounos

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Electrical Engineering and Computer Science

and Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June, 2002

Author .....
Department of Electrical Engineering and Computer Science
April 4, 2002

Certified by .....
Alan V. Oppenheim, Ford Professor Of Engineering
Department of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses
Department of Electrical Engineering and Computer Science

Signal Processing for DNA Sequencing

by

Petros T. Boufounos

Submitted to the Department of
Electrical Engineering and Computer Science
on April 4, 2002
in partial fulfillment of the requirements for the degrees of
Bachelor of Science in Electrical Engineering and Computer Science and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

DNA sequencing is the process of determining the sequence of chemical bases in a particular DNA molecule—nature's blueprint of how life works. The advancement of biological science in has created a vast demand for sequencing methods, which needs to be addressed by automated equipment. This thesis tries to address one part of that process, known as *base calling*: it is the conversion of the electrical signal—the electropherogram—collected by the sequencing equipment to a sequence of letters drawn from {A,T,C,G} that corresponds to the sequence in the molecule sequenced.

This work formulates the problem as a pattern recognition problem, and observes its striking resemblance to the speech recognition problem. We, therefore, propose combining Hidden Markov Models and Artificial Neural Networks to solve it. In the formulation we derive an algorithm for training both models together. Furthermore, we devise a method to create very accurate training data, requiring minimal hand-labeling. We compare our method with the de facto standard, PHRED, and produce comparable results. Finally, we propose alternative HMM topologies that have the potential to significantly improve the performance of the method.

Thesis Supervisor: Alan V. Oppenheim
Title: Ford Professor Of Engineering

Acknowledgements

I could have never achieved something like that without my parents. They have always supported me and my decisions, both financially and morally. They have given me valuable advice, and treated me like a mature adult. Mom, dad, thank you for everything you have done for me.

I would also like to thank the greek community at MIT, and especially the group of κωλοβαρες. More specifically, Carl, Maria, Paris (the boss), Karrie, Elias, Andy (the big 'O'), Ilias, George (a.k.a. Lysi), and Nicolas for accepting me as a friend in my first days here, five and a half years ago, and for giving me helpful advice at my first steps in MIT. The next wave of students include George (Kotsalis), Alex, Stelios, Nikos, Marios, Peggy, Christina, and Theodore, all of whom I would also like to thank for their friendship and support at hard times.

Equally important was the international 'happybunch,' especially Arin, Joanna, and Kavita for reminding me every so often that Greeks are not the only nationality in this school. I will remember the endless conversations, and I am looking forward to their encyclopaedia analyzing anything you can imagine (and some things you can't...). Also, many thanks to Ozge, Danielle, Mike, Hugo, Emanuella, Ozlem, and Zeynep, all of whom are great friends to have.

Of course, my list of friends would not be complete if I did not include the bunch in Greece. Hip, George G., George F. P. (yes, George is a common Greek name!), Apostolos, Dafni, Anastasia, Yiannis, Elena L., Elena G., Christos, Despina, and my cousins (especially Costas, Vicky and Eleni) made sure I have no time to rest during my vacations.

I would also like to thank all the people in the Ehlrich lab, especially Sameh, Aram and Mark, for believing in me and making this research possible. Without them I wouldn't even have know about the existence of such a beautiful field at the intersection of electrical engineering, computer science, and biology. They have been a key factor in the way I think about research now.

I am also grateful to all the members of the DSP group at the RLE, especially to my advisor, Al Oppenheim. His teaching formally introduced me to the world of signal processing, giving a solid background to my intuition and to my experimental learning of signal processing. The DSP group is my new home for the Ph.D. thesis, and I am really looking forward to working with such a great bunch.

Of course, nothing would be possible if I did not have some great teachers in school and college, and I have something to remember from all of them. If I start enumerating, I am sure I will forget someone, that's why I will not do it. But they all put a lot of effort to make us, stubborn kids, learn something. I thank them for that.

Last but not least I would like to thank all the MIT faculty and administrative staff. They have a very hard job to do with all the students trying to outsmart them, and they are doing it very well.

Of course, I probably have forgotten a bunch of important people, and I would like to apologize for that. It is a daunting task to list everyone and I am only a human.

Table of Contents

CHAPTER 1

Introduction, Problem Statement, and Background

The introduction of methods for DNA sequencing has revolutionized the practice of biology, leading to projects such as the Fruitfly Genome project, the Mouse Genome project and—most significant of all—the Human Genome project. The ability to decode the genetic material is of prime importance to researchers trying among other to cure diseases, improve the resistance of crops to parasites, and explore the origin of species. Moreover, the explosion of fields such as computational biology and the demands of these fields for rapid and cheap sequencing has created a great need for automation. This need is addressed by the development of modern equipment with the ability to run unattended for a long time. These systems feed their output to computer systems for processing without human intervention.

In this thesis we will focus on one small part of the whole process: *basecalling*. It is the process of converting the signal generated from the equipment to a string of letters representing the sequence of bases that compose the processed DNA sample. The problem will be formulated as a pattern recognition problem. To solve it we will propose a method similar to the ones commonly used in speech recognition, combining *Hidden Markov Models* (HMMs) and *Artificial Neural Networks* (ANNs).

Since this thesis is addressed both to the biological and the signal processing community, the background development is quite extended. Sections that might seem basic to one community are completely unknown to the other. The main work for this thesis is presented starting at the end of chapter 2, at page 54. Anything before that has the form of a tutorial, with the intent of presenting some necessary background, establishing notation and building a broader perspective for some existing concepts. This perspective will be useful in the development of the key contributions of this work.

This chapter will present how the process of DNA sequencing works, and give a brief overview of Hidden Markov Models and of Artificial Neural Networks. In addition, the problem will be posed as a pattern recognition problem and show why HMMs are suited for solving it. Finally, we will explore existing work in both DNA sequencing and pattern recognition that is related to this project.

The DNA Sequencing Process

Before presenting the DNA sequencing process, it is important to understand the structure and the function of the molecule. DNA is a long double stranded molecule that encodes the structure of specific proteins in the cell, as well as information about how these proteins should be manufactured. This information is stored in the sequence of bases (*nucleotides*) that compose the molecule. These bases can be one of the four: adenine, thymine, cytosine, and guanine—which we can denote as A, T, C, and G respectively. They are located on a deoxyribose and phosphate based backbone, and are paired as shown in Figure 1. It is important to note that the only electrochemically stable pairings are A-T, and C-G, independent of the orientation. Therefore, the composition of one strand uniquely determines the composition of the other. The goal of the DNA sequencing process is to determine the sequence of bases in a given DNA molecule.

We can abstract this picture of DNA if we think of it as a long tape on which the genetic information is stored sequentially. For storing this information, nature does not use a binary system but a quarternary one, where each string is a sequence of letters drawn from the alphabet {A, T, C, G}. DNA sequenc-

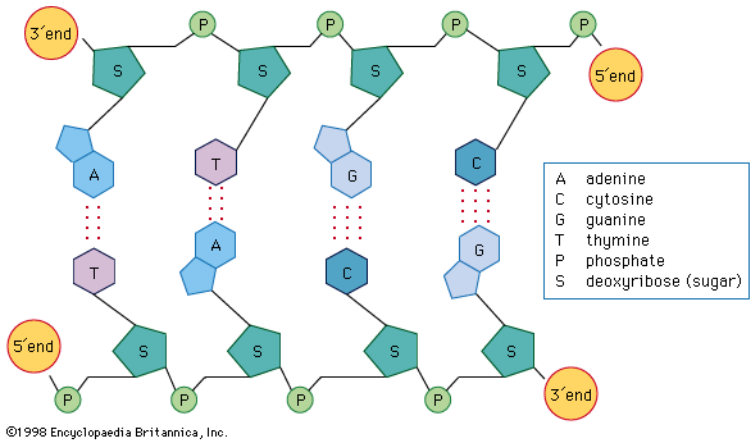

FIGURE 1. The 2-D structure of the DNA molecule. Note the pairings of the bases: A-T, and C-G (Image copyright: Encyclopædia Britannica, [6]).

ing is the process of reading the sequence of letters on a piece of DNA. We may ignore the presence of a second strand, since it is uniquely determined by the first one. This picture will be useful when we pose the problem as a pattern recognition one.

The sequencing process involves three steps: DNA sample preparation, electrophoresis, and processing of the electrophoresis output. We will briefly explore them, since they are important to our problem.

DNA Sample Preparation

The first step is quite straightforward but significant for the success of the process. For that step we will assume that we have a few copies of the same DNA molecule in a solution. We will not explore how to get these copies: it is beyond the scope of this work, in the realm of biological science. The goal of this process is to produce several copies of the DNA molecule, truncated at different lengths, and tagged with specific fluorescent tags.

In this step the original DNA molecules are first replicated several times using a biochemical process known as *polymerase chain reaction* (PCR). PCR involves cycling between three steps: *DNA denaturation*, *annealing*, and *DNA synthesis*, as shown in Figure 2. PCR is performed in the presence of *DNA*

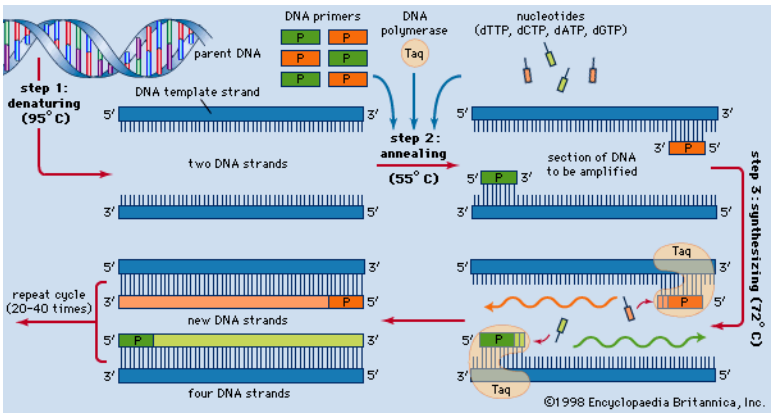

FIGURE 2. The three steps of the polymerase chain reaction (Image copyright Encyclopædia Britannica, [7]).

primers, *DNA polymerase*, and *nucleotides.* These are chemical species necessary to perform the reaction. Their function and usefulness will be explained below.

DNA denaturation: In this step the solution is heated to 95^{o}C. This causes the two strands of the molecule to separate to two complementary ones. These strands are the templates that will be used to create the two replicas of the molecule. As we mentioned before, because of the unique pairing of the bases, each strand uniquely defines the composition of its complementary one, and, therefore, one strand is sufficient to recreate the molecule.

Annealing: The purpose of this step is to initiate the replication of each strand. The solution cools down to 55^{o}C. At this temperature, the *primers* are able to attach to the 3' ends of the template strands. The primers are DNA fragments roughly 20 bases long, that are complementary to the ends of the

strands. This assumes that the sequence of these ends is known, in order to create the corresponding primers. Indeed, the method used to isolate the DNA from the cell provides for a way to know the sequences at the two ends of the molecule, or, at least, attach known sequences at both ends.

DNA synthesis: The final step occurs at 72^{o}C. At this temperature the enzyme responsible for DNA synthesis, *DNA polymerase*, attaches to the strands, at the end of the primers. The function of this enzyme is to extend the primers by attaching *nucleotides*, i.e. bases, according to the template strand. The nucleotides are attached one by one until the polymerase reaches the end of the strand and drops off the molecule. The DNA is now ready to undergo another cycle of replication.

Since from every DNA strand the reaction produces one molecule, and since each molecule has two strands, it is obvious that each cycle doubles the amount of DNA in solution. Therefore, assuming perfect yield, repeating the cycle N times yields 2^N times the original amount.

The replicas are subsequently treated with a similar process. The resulting solution contains pieces of the original strand, all with the same origin, but truncated at different lengths. These pieces are chemically labelled according to the final letter of the string with one of four fluorescent colors—say red, green, blue, and yellow. For example, if the sequence of the original strand was ATACG, the product of that process would be a chemical solution containing several replicas of five different strands: A, AT, ATA, ATAC, ATACG. The first, and the third strands would be labeled with the same fluorescent dye—red, for example—while the remaining three with the three remaining dyes—green, blue and yellow respectively, in our case.

To create this solution, one more cycle of the original PCR reaction is repeated, in a solution that contains not only nucleotides, but also “defective” nucleotides, labelled with the respective fluorescent tag. The role of the defective nucleotides is to fool the DNA polymerase to use them instead of the proper ones, effectively ending the synthesis step after the DNA strand has only been replicated partly. The position where the defective nucleotide will be inserted is random—the DNA polymerase will have to “select” it among regular and defective ones of the same type. Thus, the result will be different for each of the strands replicated, creating the solution described

above. Note, that the strands in the solution have different lengths, but all start from the same point. Also note that only the last nucleotide of each strand in the solution is a defective one. Therefore, each strand carries the fluorescent label of its last nucleotide[1].

Electrophoresis

The goal of the electrophoresis reaction is to separate, according to their size, the DNA fragments that were produced in the previous step. We can think of this reaction as a miniaturized and exaggerated version of Galileo's experiment from the leaning tower of Pizza. Galileo threw two balls of different sizes from the top of the tower. The smaller one, facing less air drag than the larger one, reached the bottom of the tower first. Effectively, the two balls were separated by size, precisely what our goal is for electrophoresis. Of course, we need to separate DNA in solution, and in equipment much smaller than the leaning tower of Pizza. Furthermore, we would like to control the force driving the molecules, therefore gravity is not an attractive option. Instead, we separate the molecules in a viscous gel using an electric field to produce the driving force.

Indeed, the reaction is based on the simple principle that under constant attractive force, larger molecules take longer time to traverse a viscous gel. Since DNA is a negatively charged molecule, in an electric field it will tend to move towards the positive electrode. By combining these two principles, we can create a simple electrophoresis apparatus: a simple container holding the gel, and a pair of electrodes generating the electric field, as shown in Figure 3. We place the solution of DNA fragments near the negative electrode and a detector near the positive end. When we turn on the electric field, DNA starts to move towards the positive electrode, and, hence, towards the detector. However, as shown in the figure, the smaller fragments can move easier and reach the detector faster. Given the right electric field strength and enough distance for DNA to travel, we can achieve complete separation between mol-

1. We should note here that there is an alternative process where the fluorescent tag is attached at the beginning of the strand. Although the details might be significant for a chemist or a biologist, they are not important to the development of the thesis, so they will not be discussed further.

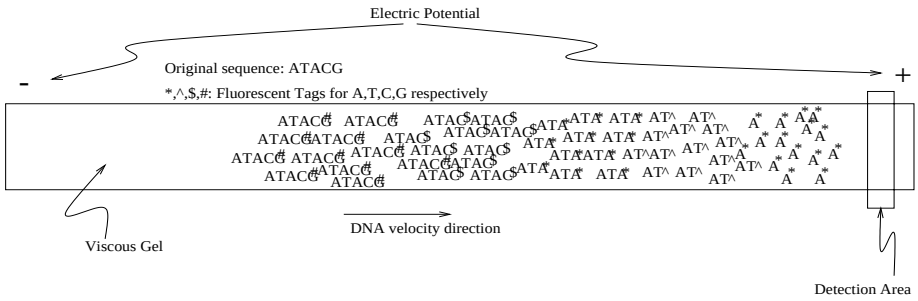

FIGURE 3. The electrophoresis process in a simple electrophoresis apparatus.

ecules of size differences as small as one nucleotide. Thus, we only need to read the fluorescent tags to determine the sequence.

This is the function of the detector. Usually the detector induces fluorescence using a laser to excite the fluorescent tags. It collects the data using a photodetector. There are several photodetector technologies, ranging from *photomultiplier tubes* (PMTs) to *charge-coupled devices* (CCDs). Each has its own trade-off in terms of simplicity to use, efficiency of fluorescence detection and spectral resolution. What is common, however, is the output of the detector: A four-channel signal—one channel per base—that represents the fluorescence of each of the four tags at each instant in time; it looks like Figure 4. Usually there is some linear cross-talk between the channels, due to imperfections in the optics of the equipment. Furthermore, there is some noise in the signal, both because of fluorescence from the gel and the container, and because of the noise in the electronics.

One small complication arises by the presence of the fluorescent tags on the molecules. The tags are quite big and different in size, depending on their fluorescent frequency. Therefore, the tags affect significantly the ability of the DNA molecule to find its way through the gel—a quantity known as mobility. Moreover, the effect will be different for molecules ending at different bases because they each carry a different tag. Hence, the fragments will not arrive in the correct sequence anymore. Fortunately, the effect is well understood, and can be undone using time warping on each of the four signals. This, how-

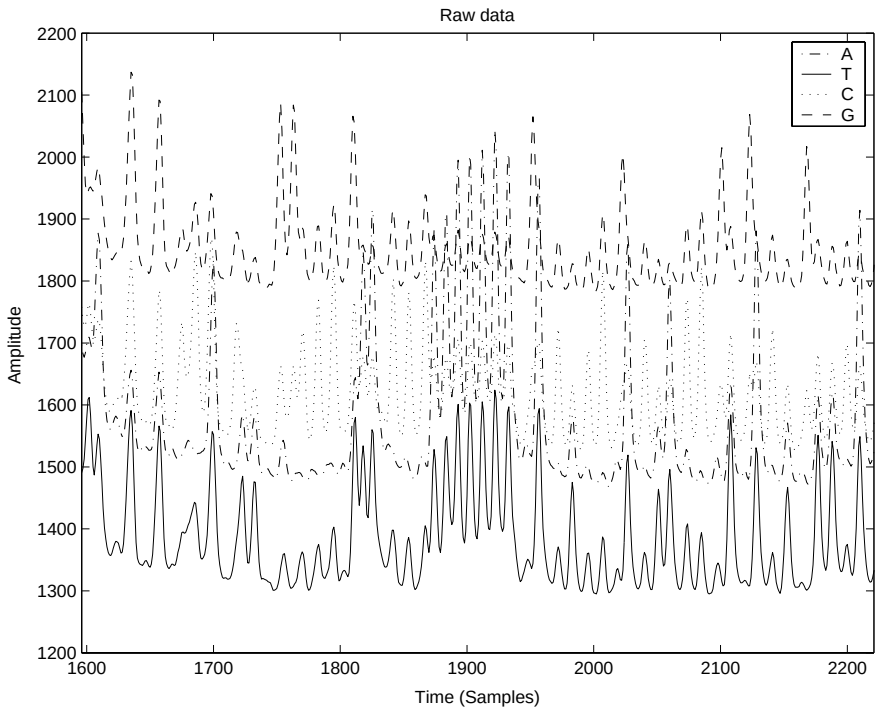

FIGURE 4. Sample raw electropherogram data

ever, is part of the signal processing, and we will explore it further in the next section.

Signal Processing

This is the last stage of the process and it involves processing the signal to produce the sequence of bases in the target molecule. This stage involves five processing functions: *denoising*, *color separation*, *baseline correction*, *mobility shift correction*, and *base calling*. The first four steps aim to condition the signal for the fifth one, without losing any useful information. They are necessary in order to undo all the imperfections in the chemistry, the optics, and the electronics of electrophoresis, as much as possible. The result of the whole process should be a sequence of letters and a confidence esti-

mate—usually in the form of probability of error or likelihood of the sequence.

Denoising: This process is aimed at removing any noise introduced in the signal. The sources of noise are many. For example, the gel often has fluorescent impurities. Furthermore, the electronics are imperfect and noisy. Even scattered light might make it back into the detector. Although the designers of the equipment try to reduce the noise presence, complete elimination is impossible. We usually model the noise as a white gaussian process. Since the actual DNA fluorescence is a very slowly varying signal—i.e. has significant power at low frequencies—low pass filtering is usually enough to filter the noise out.

Color Separation: This is a linear operation, which aims to eliminate the cross-talk between the four channels of the signal. This cross-talk is due to the imperfections of the optical filters that separate the fluorescence from each of the four tags. It is effectively a linear mixing of the four signals that can be undone trivially, assuming that the mixing coefficients are known. If we use the vector x to denote the desired signals, matrix M to denote the mixing matrix and the vector x_m to denote the mixed signals, then we can express x using $x_m=Mx$. Assuming that M is invertible—which is true since the mixing can be thought of as a rotation and a scaling, both invertible operations—we can calculate x using $x=M^{-1}x_m$. The matrix M, however, is not known, and should be determined. A number of techniques to determine the mixing matrix exist, mostly based on analysis of the second order statistics of the signal.

Baseline correction: This step aims to remove constant and very slowly varying offsets that occur to the signal due to a constant value of background fluoresecne. This fluorescence often depends on the experimental conditions, such as the temperature of the electrophoresis. Since these might not be constant during the run, there might be a small drift in the DC value of the signal. Furthermore, the background fluorescence is different for the four channels, so the DC value is different. The goal of this process is to remove a roughly constant waveform from the recorded signal; it is effectively a high-pass filter. The resulting signal should be zero—assuming that all the noise has been cleared in the first step—when no DNA is present in front of the detector.

Mobility Shift Correction: As we mentioned before, the presence of the fluorescent tags affects the mobility of the DNA molecules. This step aims to undo this effect by a time warping process. Ideally, the resulting signal should be equivalent to the signal that would have been obtained if all four of the tags had the same effect on the mobility of the DNA fragments.

Frequently, this step is combined with another time warping aimed at creating a signal with uniform peak spacing. Because of the nature of the experiment, the peak spacing near the end of the signal is much wider than the peak spacing at the beginning. In order to undo this effect, time can be warped so that it runs faster near the end of the run and slower at the beginning. This will result to a signal with uniform peak spacing. Uniform peak spacing might or might not be necessary, depending on the requirements of the next step.

Base Calling: This is the final step of the processing. The goal of this step is to translate the processed signal—which looks like Figure 5—into the sequence of letters that describe the DNA sequenced. This sequence should be accompanied by some form of confidence estimates. The confidence measures are usually in the form of likelihood measures or probability of error estimates. These measures usually reflect the quality and the resolution of the signal. Furthermore, they are very useful in further stages of processing of the output and in the quality control of the sequencing process. Base calling is the problem we try to address in this thesis.

A Brief Introduction to Pattern Recognition

As we will discuss later in this chapter, base calling is an example of a class of problems in the field of *pattern recognition*. Loosely speaking, this field—also known as *pattern classification*—examines ways to categorize data to several different classes. A typical problem, for example, is speech recognition, where speech utterances are translated—i.e. classified—to sequences of letters or words. Another example is face recognition: the face of a person must be recognized in an image, or an image must be categorized as ‘face’ or ‘not-face.’

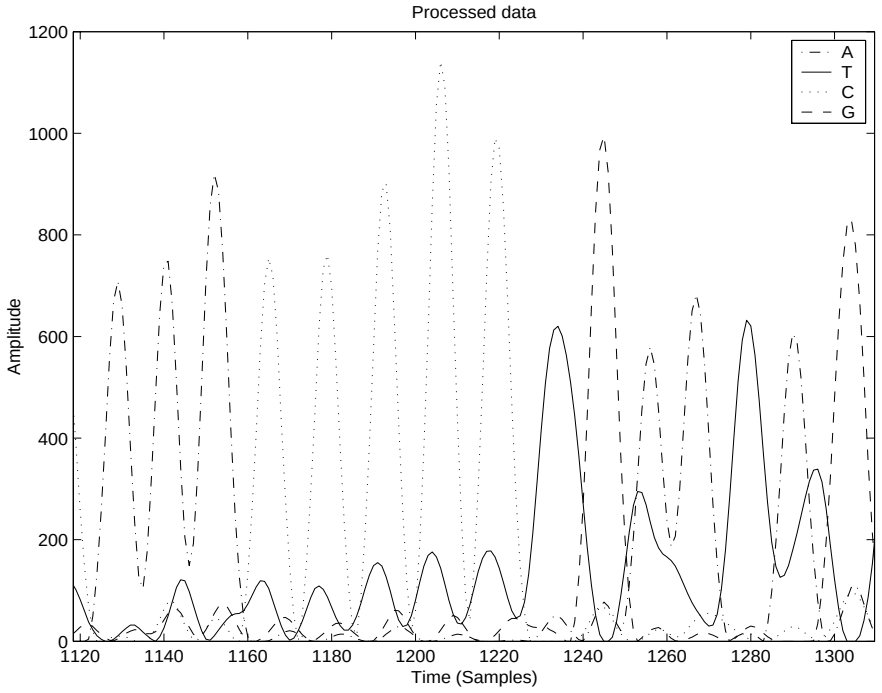

FIGURE 5. Processed electropherogram data, ready to be basecalled.

Modern techniques rely heavily on statistical models and probability theory to perform classification. Usually, a statistical model is assumed about the format and the distribution of certain features in the data. The parameters of the model are estimated using some example data, a process known as *training*. Once the parameters are estimated, the model can be used to classify new data. Sometimes the model is not trained with example data but it is formulated to find some 'natural' classification, by adjusting its parameters. The former way of training is often called *supervised learning*, while the latter is known as *unsupervised learning*.

In the rest of this section we will discuss some common considerations in the design of pattern recognition systems. We will proceed to examine two very effective techniques for parameter estimation, which we will use in the rest of

the thesis. Finally we will present two powerful tools used in pattern recognition: *Artificial Neural Networks* and *Hidden Markov Models*.

Considerations in the Design of Pattern Recognition Systems

When designing a pattern recognition system, there are several issues to consider. The success of the system depends on how well it can classify the data, and on how well it can generalize based on the training data. Therefore, the selection of the statistical model, and of the features of the data that the model will use are of prime importance. Furthermore, care should be taken in the training of the model, to ensure that it will perform well on actual data.

Model Selection: It might sound obvious that the statistical model selected should fit the actual statistics of the data. However, it is not trivial to formulate such a model, and ensure that its parameters can be estimated. Therefore, simplifications might often be needed. Furthermore, a model with a big parameter space might train very slowly, and might be prone to overfitting—which we will discuss later. On the other hand, a model with a very small parameter space might not be sufficient to describe the data; such a model would perform poorly. A general rule of thumb is to use a model as simple as needed to describe the data, but not simpler. This heuristic rule is also known as *Occam's razor* ([14] describes the rule in more detail).

Feature Selection: A separate, but very related, aspect of the design is the selection of the *features*—i.e. the functions of the raw data—that will be used for the classification. For example, the pitch of a voice signal might be the feature used to distinguish male from female voices in a classifier. Thus, a function of the raw waveform is used, and not the waveform itself.

While the unprocessed data can be used as features, this is often a bad choice for pattern recognition. For example, the data might have a very peculiar probability distribution, but a nonlinear function of the data might be normally distributed, making the models very easy to train. Also, the data might be multidimensional, while a function of the data might reduce the dimensionality, without throwing out useful information. Significant literature is devoted to feature selection, since it is a topic that often makes the difference between models that work and models that do not.

Overfitting: This issue often occurs when the model adjusts too much to the data given as examples, and is not able to generalize. Trying to achieve perfect classification rates on the training data is often a cause for overfitting. Complex models are prone to this problem, since they have a large parameter space, and therefore it is easier to find a parameter combination that classifies the training data very well. However, this is not the parameter combination that achieves good generalization.

There are a couple or ways to reduce overfitting. The simplest is to reduce the parameter space of the model, leading to models less prone to the problem. This is another instance of *Occum's razor.* An alternative way is to use a validation set of data, independent of the training set, on which the classification performance is tested. The performance of the model on the validation set should improve as training proceeds. Once the performance on the validation test starts deteriorating, this is an indication that the model is overfitting to the training data, and the training should be stopped.

Searching the Parameter Space

As we described above, in order to train the models we need to calibrate parameters based on the training data. Usually, parameters are selected so as to minimize some cost function, based on the model and the training set. Unfortunately, the parameter space of the functions is often very large, if not infinite, and an exhaustive search impossible. Several optimization techniques exist to perform such a search. I will present only two here: *gradient descent*, and *expectation-maximization (EM)*. These are the optimization methods most often used with the types of models I will consider later.

Gradient Descent: This is the simplest method to search the parameter space, but one of the most frequently used. The general idea is that the search starts at a random point in the multidimensional cost surface and perform a descent towards the steepest downhill path. We can picture a 1D version of this method by examining a gradient descent on the cost function depicted in Figure 6. Starting from a random point in the real axis, we get to the minimum of the cost function by following the derivative of the function.

It is obvious from the picture that this method does not guarantee settlement to the global minimum of the cost function, but only to a local one. This

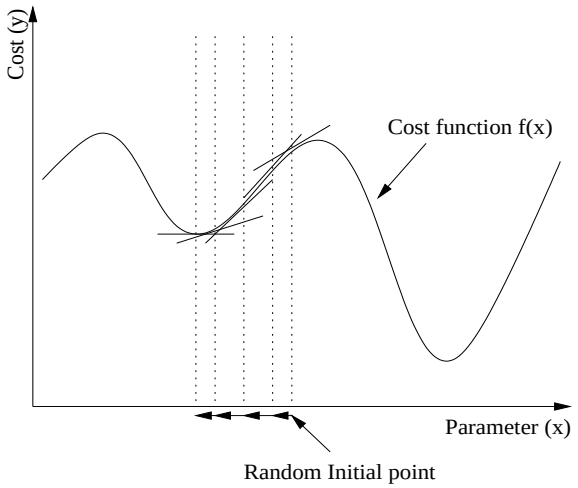

FIGURE 6. A simple one dimensional gradient descent example

might be an issue in certain cases. There exist ways to search around a larger local area, but not a global one. In this thesis, this method will be used to train Artificial Neural Networks.

An important parameter in the descent is the step size. Using a very small step size might result to a large number of iterations until the minimum is reached. Using a large one might result in oscillations around the minimum, without ever reaching it. Several tricks exist to overcome that difficulty. These include using a larger stepsize at the beginning and a smaller one as the training progresses. Also one could add momentum, which uses the previous step's direction and size to influence the current step's direction and size, effectively reducing the stepsize as the training approaches the minimum.

Expectation-Maximization: This is a two step iterative process especially designed for probability optimizations. The algorithm starts from certain parameter estimates, and calculates the expectations of the probability functions of the model based on these estimates (Expectation step). Then it estimates a new set of parameters that maximize the cost function given the

probabilities calculated before (Maximization step). The two steps are iterated until convergence is achieved.

An interesting view of the algorithm is that of a lower bound maximization. Indeed, as [17] shows, given a point in the parameter space, EM constructs a function of arbitrary form that is a lower bound to the cost function, such that they both have the same value at the specific point. The *M*-step chooses the step that maximizes over the lower bound function (the assumption is that the lower bound is easier to maximize), which guarantees improvement over the cost function. A version of the algorithm known as generalized EM only improves over the lower bound, which still improves over the cost function but less aggressively.

The algorithm exhibits the problem of locality that we observe in gradient descent. Although here convergence is guaranteed, we still have no guarantee that the global minimum will be achieved. However, it has the advantage of much faster convergence than gradient descent. Unfortunately, it is not applicable as widely as gradient descent.

Recognizing Static Patterns and Time Series

Since we are interested in processing DNA signals, we need to use a model that incorporates time. However, these models are easier to develop once we understand static models, and extend them to cover time series. Therefore, we will first examine *Gaussian mixture models* and *artificial neural networks* as probability estimators. Next we will combine these estimators and *markov chains* to create *Hidden Markov Models*. These are very powerful models, commonly used in speech recognition. In the remaining of this section I will present a brief overview. An expanded analysis will follow in the next chapter.

Gaussian Mixture Models: This is one of the simplest models for describing data. The assumption for such models is that the probability distribution of the features is a Gaussian mixture, with different parameters for each class. The distribution of a Gaussian mixture is the sum of scaled Gaussian density functions with different means and covariance matrices, such that the sum of the scale factors is equal to 1.

Training and classification is very easy with this type of model. To train this model, we only need to estimate the scale factor, the mean, and the covariance matrix for each of the Gaussians in the mixture. Training is usually performed using an EM strategy, developed further in the next chapter. Classification is done by selecting the most likely class given the data.

Artificial Neural Networks: These types of models were originally conceived as imitations of biological neural networks. However, the state of the art has evolved, and the field has been formalized and has progressed beyond the original simplicity. ANNs are usually comprised of layers of fundamental units, the *neurons*, that exhibit certain properties. It can be shown that under certain conditions, ANNs can be universal function estimators, a quality we will use for pattern recognition. Indeed, given that property, there are two ways to perform classification using an ANN: use them as functions that estimate the class directly, or use them as functions that estimate the likelihood of the class.

There are several advantages in using ANNs, and several drawbacks. The main advantage is that they can be trained to model any kind of data. However, because of this universality, the parameter space is huge. This often causes convergence issues: there is no guarantee that the gradient descent used to train a neural network will converge to a meaningful local minimum. For the same reason, ANNs are very prone to overtraining, an effect known in the literature as the *curse of dimensionality* (for more details see [14], p. 210-211).

Hidden Markov Models: These models are very good at describing continuous time processes. The assumption in these models is that the underlying process that produces the signal is a Markov chain. Markov chains are nondeterministic finite state machines that have state transitions governed by certain probability distribution functions. The Markov chain is ‘hidden’ in the following sense: the signal we observe is not deterministically related to the state of the Markov chain. Hence, we cannot have a certain mapping from the signal to the state transitions. Instead, each state emits the signal stochastically, with a certain probability model. The models used for these emissions are usually Gaussian mixture models or Neural Networks. The hidden markov chains are very effective in modeling time series, thus HMMs are often encountered in the speech recognition literature.

Training the model involves estimating the transition probabilities, and the emission probability density functions. This is usually performed by an instance of the EM algorithm known as the Baum-Welch algorithm. When Gaussian mixture models are used for the emission probabilities, the estimation can be incorporated into the Baum-Welch algorithm. However, when Neural Networks are used, they are usually trained separately from the remaining model, using manually labeled data. In this thesis we will develop a way to incorporate the ANN training into the whole training method, eliminating—at least, partly—the need for manual labeling.

Background

Before developing the proposed solution it is necessary to present the existing work in the field. This presentation will motivate the use of HMMs to solve the basecalling problem. Furthermore, it will provide us with some benchmark to compare our results. First we will examine existing work in DNA sequencing, and then we will give some necessary background on HMMs and attempts at combining them with ANNs.

DNA Sequencing

Existing work in the field is mostly concentrated in signal conditioning and basecalling. Giddings et al. [13] provide an overview of the signal processing steps described above, and propose a modular approach to building the basecalling system. Also, Giddings et al. [12] present a software system for data analysis in older slab gel electrophoresis machines. Berno [2] proposes a graph-theoretic approach to basecalling. Ewing et al. [9] describe Phred, the software mainly used by the Human Genome Project for analysis of the signals. Furthermore, Ewing and Green [10] describe how Phred assigns confidence estimates to the basecalled data. Lipshutz et al. [16] propose a method based on classification trees to perform the confidence estimation and correct uncalled parts of the signal. Finally, Lawrence et al. [15] suggest a linear discriminant analysis approach to assign position-specific confidence estimates on basecalled data. However, all approaches are empirical and depend significantly on heuristic rules.

More recently, Nelson [18] described some initial efforts to put statistical foundations on the problem, an approach that will isolate basecalling from the particular instrument used and provide confidence estimates derived straight from the methods used to basecall. This thesis intends to proceed in a similar direction.

Pattern Recognition

Pattern recognition is a very mature field, compared to DNA sequencing. Indeed, several good books, such as [5] and [14] exist to guide a beginner through the fundamentals of statistical pattern recognition, Gaussian mixture models, and artificial neural networks. Hidden Markov Models have been extensively studied in the field of speech recognition, and a very good review of the work can be found in Rabiner's tutorial [20]. Finally, some work on integrating HMMs with ANNs has appeared in [4] but the training methods used are not suitable for our case.

CHAPTER 2

Pattern Recognition for Static and Time-Varying Data

In this chapter we will develop further the pattern recognition techniques presented in the introduction. After developing the training methods for Gaussian Mixtures and Artificial Neural Networks, as usually developed in the literature, we will show how Markov Chains can be used to create a Hidden Markov Model framework for time varying signals. In this development we will follow closely but not exactly Rabiner's presentation in [20]. Finally, we will combine ANNs with the HMM framework and provide a method to train the system with sample data.

Gaussian Mixture Models

These models assume that the features in each class are distributed according to a mixture of *M* Gaussians. The advantage of such a model is that it can be trained using the EM algorithm, which usually implies rapid convergence. Still, unless the number of mixtures is large the model's descriptive power is limited. Furthermore, unless the amount of training data is large, convergence might become problematic. Nevertheless, these models are simple and powerful enough to be widely used. Therefore, we will examine them as the basic model for pattern recognition.

Model Parameters

As mentioned above, we will assume that there are *M* components in the mixture for each class. In other words, we will assume that if $\boldsymbol{x}$ is the feature vector we would like to classify, then the density of $\boldsymbol{x}$ originating from class *j* is:

$$p_j(\boldsymbol{x}) = \sum_{m=1}^{M} c_{jm} N(\boldsymbol{x}, \mu_{jm}, \boldsymbol{U}_{jm}) \tag{1}$$

Training this model involves estimating the parameters c_{jm}, μ_{jm}, and $\boldsymbol{U}_{jm}$ for each component *m* and class *j*. These parameters represent the mixture coefficients, the mixture means, and the mixture covariance matrices respectively. It can be shown that any distribution can be approximated arbitrarily well by a gaussian mixture model with a sufficiently large number of components. However, the number of components necessary is often extremely large for practical implementation.

Training the Model

The most common method to train a Gaussian Mixture model is the EM algorithm. The *Expectation* step of the two-step iterative algorithm involves estimating $\hat{p}_m(\boldsymbol{x}_i)$ for all the mixture components *m* and the data points $\boldsymbol{x}_i$ in class *j*. The estimate is calculated using the values of the parameters from the previous iteration.

The *Maximization* step estimates the parameters of the model using the following formulas:

$$\hat{\mu}_{jm} = \frac{\sum_{i=1}^{N} \hat{p}_m(\boldsymbol{x}_i)\boldsymbol{x}_i}{\sum_{i=1}^{N} \hat{p}_m(\boldsymbol{x}_i)} \tag{2}$$

$$\hat{\boldsymbol{U}}_{jm} = \frac{\sum_{i=1}^{N} \hat{p}_m(\boldsymbol{x}_i)(\boldsymbol{x}_i - \hat{\mu}_{jm})(\boldsymbol{x}_i - \hat{\mu}_{jm})^T}{\sum_{i=1}^{N} \hat{p}_m(\boldsymbol{x}_i)} \tag{3}$$

$$\hat{c}_{jm} = \frac{1}{N}\sum_{i=1}^{N} \hat{p}_m(\boldsymbol{x}_i)\,, \tag{4}$$

where N is the number of training points $\boldsymbol{x}_i$ that belong to the class j.

The two step algorithm should be repeated several times until the values of the estimated parameters converge. We should note that a subtle issue is the selection of the parameters before the first iteration: unfortunately, a random selection might not always be the best choice since it might make the algorithm to converge to a very inefficient local minimum of the cost function. Usually a more educated guess for the initialization is required, but this is an issue we will swipe under the rug.

Artificial Neural Networks

Artificial Neural Networks evolved from mathematical models of the neural networks in the human brain. Built from a fundamental block, the *neuron*, ANNs can be universal function estimators. In this section we will examine how these building blocks behave, how they are usually combined into networks, and how the networks are trained. Finally we will present a variation of the building block, that is very useful in networks designed to estimate probability mass functions. Since our approach to their development is not often encountered, we will give a rather extensive presentation compared to the Gaussian mixture models. However, we believe that this approach deserves the attention.

The Fundamental Building Block

Similarly to their biological counterparts, ANNs are built from units that combine several inputs into one output. These units look like the one in Figure 7.

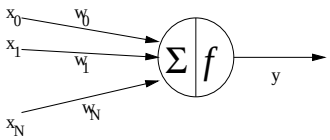

FIGURE 7. A *neuron*: the fundamental building block of a neural network

We will define the inputs as x_i for $i=0...N$, and the weights as w_i for the same i. Conventionally, we define $i_0=1$, and the corresponding weight w_i is called—for reasons that will be obvious later—the *bias*. Therefore, the unit we just defined has N independent inputs. Under these definitions, the output y of the neuron is defined as

$$y = f\left(\sum_{i=0}^{N} x_i w_i\right) = f\left(w_0 + \sum_{i=1}^{N} x_i w_i\right), \tag{5}$$

where the function *f(.)* is called the *activation function*. The second form of the equation justifies the term *bias* used for w_0: the term does not influence the data; it is just an additive constant that moves the sum in the input region of the activation function.

The notation above is useful to see the functionality of the neuron: it passes a weighted and biased sum of its inputs through the activation function. However, it will be convenient to use a compact vector notation once we start working with layers and networks of these units:

$$\boldsymbol{x} = \left[x_0 \ \ldots \ x_N\right]^T \tag{6}$$

$$\boldsymbol{w} = \left[w_0 \ \ldots \ w_N\right]^T \tag{7}$$

Then, the sum may be replaced by the dot product. Thus, the output *y* is equal to:

$$y = f(\boldsymbol{w}^T \boldsymbol{x}) \tag{8}$$

The activation function *f(.)* can be anything we desire, but certain functions are more popular than others. The original models used the unit step function:

$$u(x) = \begin{cases} 0, x < 0 \\ 1, x \geq 0 \end{cases} \tag{9}$$

This function matches the function of biological neurons. Artificial neurons using this activation function are often called *perceptrons*.

The big disadvantage of the unit step activation function is the discontinuity at 0, and the lack of a derivative. The discrete nature of this function does not allow for a gradient descent algorithm to operate on it, in order to perform training. This proved to be a significant drawback. Research in the area was set back for a long time, until the issue was resolved. In fact, no good solution has been found yet for this problem. Instead, a very useful alternative is used, the *sigmoid* function:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{10}$$

The graph of this function is shown in Figure 8. It is obvious that by scaling the input it can approximate the unit step function arbitrarily well. However, it has the added advantage that it is differentiable everywhere, it's derivative being:

$$f'(x) = \frac{e^{-x}}{(1 + e^{-x})^2} = f(x)(1 - f(x)) \tag{11}$$

This property is used extensively in training ANNs using a simple gradient descent algorithm.

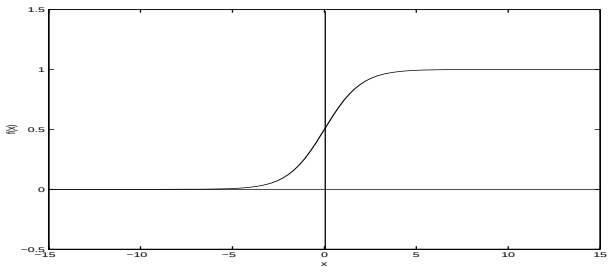

FIGURE 8. The sigmoid function

These two functions are not the only useful ones. Several others, such as the linear function $f(x)=x$, or the hyperbolic tangent function are often used, depending on the application. However, the sigmoid function is the most common. For the purposes of our development we will only use two functions: the sigmoid discussed above, and the *softmax* function. Because the latter exhibits some peculiarities, we will present it after we discuss how to organize neurons into layers and then into networks.

Layers and Networks

In order to handle the complexity of arbitrary networks of neurons, it is often desirable to limit their structure. We will assume that the networks are composed of layers of neurons. Each layer has multiple inputs and multiple outputs, as shown in Figure 9. All the inputs of the layer are observed by all its neurons, with different weight coefficients. In addition, all the neurons in the layer use the same activation function. The output of the layer is the vector of the outputs of each individual neuron of the layer. In fact we can extend the vector notation to matrix notation to describe the whole layer. If the layer has M neurons, then we can define the MxN weight matrix $\boldsymbol{W}$:

$$\boldsymbol{W} = \begin{bmatrix} - & \boldsymbol{w}_1^T & - \\ & \vdots & \\ - & \boldsymbol{w}_M^T & - \end{bmatrix}, \tag{12}$$

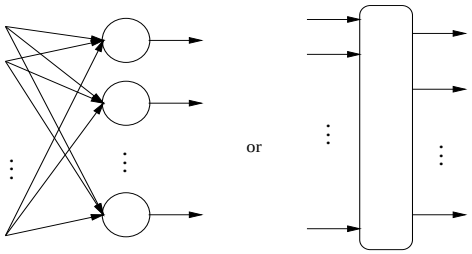

FIGURE 9. A multiple-input-multiple output *layer* of neurons. The right-hand part of the figure is just a schematic simplification of the left-hand part.

where $\boldsymbol{w}_i$ is the weight vector for the i^{th} neuron in the layer. If we also define $\boldsymbol{y}$, the output vector of the layer as

$$\boldsymbol{y} = \left[y_1 \ \ldots \ y_M \right]^T, \tag{13}$$

then the operation of the layer reduces to:

$$\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{W}\boldsymbol{x}) \tag{14}$$

where the vector function $\boldsymbol{f(x)}$ is just the activation function applied over all the elements of the vector, concatenated to the constant term that will apply the bias at the next layer:

$$\boldsymbol{f}(\boldsymbol{x}) = \left[1 \ f(x_1) \ \ldots \ f(x_M) \right]^T \tag{15}$$

Having defined a layer, we can now cascade multiple layers of different sizes to create a network, as shown in Figure 10. The connection between each neuron's output to another neuron's input is often called a *synapse*.

Often, we consider the inputs as one extra layer—the *input* layer. Also, the layer producing the outputs is called the *output* layer. Any layers between

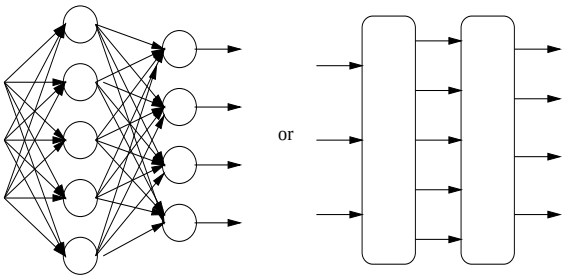

FIGURE 10. Two layers of neurons combined to give a neural network

these two are called *hidden* layers. These distinctions are important when deriving the formulas to train the network.

A multilayer network is a great function approximation tool. It can be shown that a single hidden layer is able to approximate any given function, if the layer has the right number of neurons (for details see [14], p. 208). This theorem provides a good justification for using ANNs to approximate functions. We should note however, that the size of the networks suggested by the theorem is large. In practice this makes networks prone to overfitting and less able to generalize—a great example of the curse of dimensionality: the network essentially 'memorizes' the training data. In applications we often encounter smaller networks. These are not good in memorizing the training data, therefore generalize better.

Error Backpropagation

Having discussed the network topology, we can use a very easy technique for training the network, caller *error backpropagation*. Although this method is just a gradient descent, we will see shortly how the network topology provides for a convenient way to find the gradient for each of the network weights w_i—the parameters of the model.

Before we discuss the method, it is important to establish some notation. We will denote each layer by a superscript to the appropriate parameter. The network will have L layers, each layer having a weight matrix $\mathbf{W}^l$, N^l inputs, and

M^l outputs. The input will be denoted by $\boldsymbol{x}^l$, and the output by $\boldsymbol{y}^l=\boldsymbol{x}^{l+1}$ for $l>1$. The first layer will be the input layer, and the last layer will be the output layer. The input-output relationship at each layer is given by

$$\boldsymbol{y}^l = \boldsymbol{f}^l(\boldsymbol{W}^l\boldsymbol{x}^l), \tag{16}$$

$$\text{where } \boldsymbol{f}^l(\boldsymbol{x}) = \begin{bmatrix} 1 & f^l(x_1) & \dots & f^l(x_M) \end{bmatrix}^T, \tag{17}$$

and $f^l(x)$ is the activation function of the l^{th} layer. It is easy to show that given $\boldsymbol{x}^l$ for any layer l, then the output of the network will be

$$\boldsymbol{y}^L = \boldsymbol{f}^L(\boldsymbol{W}^L\boldsymbol{f}^{L-1}(\dots(\boldsymbol{W}^{l+1}\boldsymbol{f}^l(\boldsymbol{W}^l\boldsymbol{x}^l)))). \tag{18}$$

The cost function we will optimize is the magnitude squared of the error. Specifically, we define the error vector $\boldsymbol{e}$ to be

$$\boldsymbol{e} = \boldsymbol{y}^L - \boldsymbol{y}_D, \tag{19}$$

where $\boldsymbol{y}_D$ is the desired output vector—taken from the training set. The magnitude squared of the error vector is given by:

$$c = \frac{1}{2}\|\boldsymbol{e}\|^2 = \frac{1}{2}\sum_{i=1}^{N^L} e_i^2. \tag{20}$$

The factor of 1/2 at the beginning of the expression does not affect any optimization; it is there in order to eliminate a factor of two that will appear when differentiating the cost function to perform the gradient descent. Indeed, to find the parameters w_{ij}^l—the weight coefficients for each synapse—we will need to find the derivative of c with respect to each parameter.

In addition, we will define the derivative—also known as the gradient—of the cost function with respect to any matrix as the matrix with elements:

$$\nabla_{\boldsymbol{M}} c = \left(\frac{\partial c}{\partial \boldsymbol{M}}\right)_{ij} = \frac{\partial c}{\partial (\boldsymbol{M})_{ij}}. \tag{21}$$

In most cases $\boldsymbol{M}$ will be one of the weight matrices $\boldsymbol{W}^l$. Still, sometimes, we might need to find the gradient with respect to a vector, in which case the above definition will still hold, the result being a vector instead. We will also need the gradient matrix of a vector function—we can think of it as the derivative of the function with respect to a vector—defined as the matrix with elements:

$$(\nabla \boldsymbol{f}(\boldsymbol{x}))_{ij} = \left(\frac{\partial}{\partial \boldsymbol{x}} \boldsymbol{f}(\boldsymbol{x})\right)_{ij} = \frac{\partial}{\partial x_i} f_j(\boldsymbol{x}). \tag{22}$$

The gradient matrix is the transpose of the Jacobian operator $\boldsymbol{Jf(x)}$, often used instead in the literature:

$$(\boldsymbol{Jf}(\boldsymbol{x}))_{ij} = ((\nabla \boldsymbol{f}(\boldsymbol{x}))^T)_{ij} = \frac{\partial}{\partial x_j} f_i(\boldsymbol{x}). \tag{23}$$

Finally, we will denote the conversion of a vector to a diagonal matrix using:

$$(\mathrm{diag}(\boldsymbol{x}))_{ij} = x_i \delta_{ij}, \tag{24}$$

where δ_{ij} is equal to 1 for $i=j$ and to 0 otherwise.

Given these definitions, it is easy to show that for any scalar function of a vector $g(\boldsymbol{x})$ the following relationships hold for any vector $\boldsymbol{y}$, any matrix $\boldsymbol{A}$, and any vector function $\boldsymbol{f(x)}$:

$$\frac{\partial}{\partial \boldsymbol{A}} g(\boldsymbol{Ax}) = \left(\frac{\partial}{\partial \boldsymbol{y}} g(\boldsymbol{y})\Big|_{\boldsymbol{y}=\boldsymbol{Ax}}\right)\boldsymbol{x}^T, \tag{25}$$

$$\frac{\partial}{\partial \boldsymbol{x}} g(\boldsymbol{f}(\boldsymbol{x})) = \frac{\partial}{\partial \boldsymbol{x}} \boldsymbol{f}(\boldsymbol{x}) \left(\frac{\partial}{\partial \boldsymbol{y}} g(\boldsymbol{y})\Big|_{\boldsymbol{y}=\boldsymbol{f}(\boldsymbol{x})}\right), \tag{26}$$

which we may combine to get:

$$\frac{\partial}{\partial \boldsymbol{A}} g(\boldsymbol{f}(\boldsymbol{A}\boldsymbol{x})) = \left(\frac{\partial}{\partial \boldsymbol{y}} \boldsymbol{f}(\boldsymbol{y})\Big|_{\boldsymbol{y}=\boldsymbol{A}\boldsymbol{x}}\right) \boldsymbol{x}^T \left(\frac{\partial}{\partial \boldsymbol{y}} g(\boldsymbol{y})\Big|_{\boldsymbol{y}=\boldsymbol{f}(\boldsymbol{A}\boldsymbol{x})}\right). \tag{27}$$

Next, we may define the cost function as a function of the input at layer l:

$$c^l(\boldsymbol{x}^l) = \frac{1}{2}\left\|\boldsymbol{y}^L - \boldsymbol{y}_D\right\|^2 = \frac{1}{2}\left\|\boldsymbol{f}^L(\boldsymbol{W}^L \boldsymbol{f}^{L-1}(\ldots(\boldsymbol{W}^{l+1}\boldsymbol{f}^l(\boldsymbol{W}^l\boldsymbol{x}^l)))) - \boldsymbol{y}_D\right\|^2, \tag{28}$$

and easily show that

$$c^l(\boldsymbol{x}^l) = c^{l+1}(\boldsymbol{f}^l(\boldsymbol{W}^l\boldsymbol{x}^l)). \tag{29}$$

Therefore the gradient with respect to the weight matrix is equal to:

$$\frac{\partial}{\partial \boldsymbol{W}^l} c^l(\boldsymbol{x}^l) = \left(\frac{\partial}{\partial \boldsymbol{x}} \boldsymbol{f}^l(\boldsymbol{x})\Big|_{\boldsymbol{x}=\boldsymbol{W}^l\boldsymbol{x}^l}\right) \boldsymbol{x}^l \left(\frac{\partial}{\partial \boldsymbol{x}^{l+1}} c^{l+1}(\boldsymbol{x}^{l+1})\Big|_{\boldsymbol{x}^{l+1}=\boldsymbol{f}^l(\boldsymbol{W}^l\boldsymbol{x}^l)}\right). \tag{30}$$

Furthermore, the gradient of the cost function with respect to the input is:

$$\frac{\partial}{\partial \boldsymbol{x}^l} c^l(\boldsymbol{x}^l) = \left(\frac{\partial}{\partial \boldsymbol{x}} \boldsymbol{f}^l(\boldsymbol{W}^l\boldsymbol{x}^l)\right)\left(\frac{\partial}{\partial \boldsymbol{x}^{l+1}} c^{l+1}(\boldsymbol{x}^{l+1})\Big|_{\boldsymbol{x}^{l+1}=\boldsymbol{f}^l(\boldsymbol{W}^l\boldsymbol{x}^l)}\right). \tag{31}$$

We can see that in these two equations there is a common recursive vector, the *local error signal*, denoted by $\boldsymbol{e}^l$:

$$\boldsymbol{e}^l = \frac{\partial}{\partial \boldsymbol{x}^l} c^l(\boldsymbol{x}^l)\Big|_{\boldsymbol{x}^l=\boldsymbol{f}^{l-1}(\boldsymbol{W}^{l-1}\boldsymbol{x}^{l-1})} \tag{32}$$

Substituting, the formulas for the gradients become:

$$\frac{\partial}{\partial \boldsymbol{W}^l} c^l(\boldsymbol{x}^l) = \left(\frac{\partial}{\partial \boldsymbol{x}} \boldsymbol{f}^l(\boldsymbol{x})\Big|_{\boldsymbol{x}=\boldsymbol{W}^l\boldsymbol{x}^l}\right) \boldsymbol{x}^l \boldsymbol{e}^{l+1}, \tag{33}$$

$$\text{and } \frac{\partial}{\partial \boldsymbol{x}^l} c^l(\boldsymbol{x}^l) = \left(\frac{\partial}{\partial \boldsymbol{x}} \boldsymbol{f}^l(\boldsymbol{W}^l \boldsymbol{x}^l)\right) \boldsymbol{e}^{l+1} = \boldsymbol{e}^l . \tag{34}$$

These formulas prompt to a nice recursion. Indeed, starting with the output layer we could calculate the gradient of the weight matrix and the local error signal. This signal is passed on to the previous layer, so that its gradient and local error signal is calculated, continuing the recursion until the input layer. The only issue to be resolved is the initiation of the recursion at the output layer. Specifically, we need to find formulas for the two gradients at $l=L$, as a function of the output data and the desired output data. However, it is easy to show that since

$$c = \frac{1}{2} \left\| \boldsymbol{y}^L - \boldsymbol{y}_D \right\|^2 , \tag{35}$$

$$\text{then } \frac{\partial c}{\partial \boldsymbol{y}^L} = \boldsymbol{y}^L - \boldsymbol{y}_D = \boldsymbol{e} , \tag{36}$$

where $\boldsymbol{e}$ is out error at the output, as defined originally. Thus, using the same formulas as before we can show that:

$$\frac{\partial}{\partial \boldsymbol{W}^L} c^L(\boldsymbol{x}^L) = \left(\frac{\partial}{\partial \boldsymbol{x}} \boldsymbol{f}^L(\boldsymbol{x}) \Big|_{\boldsymbol{x} = \boldsymbol{W}^L \boldsymbol{x}^L} \right) \boldsymbol{x}^L \boldsymbol{e} , \tag{37}$$

$$\text{and } \frac{\partial}{\partial \boldsymbol{x}^L} c^L(\boldsymbol{x}^L) = \left(\frac{\partial}{\partial \boldsymbol{x}} \boldsymbol{f}^L(\boldsymbol{W}^L \boldsymbol{x}^L)\right) \boldsymbol{e} = \boldsymbol{e}^L . \tag{38}$$

These equations also justify the term *local error signal* for $\boldsymbol{e}^l$. Each layer calculates the gradient of its weight matrix, and effectively propagates the error to the layer preceding it. Thus, this algorithm is called *error backpropagation*.

Having calculated each of the weight matrix gradients, we need to descend towards the bottom of the cost function. Therefore, we need to update all the matrices $\boldsymbol{W}^l$ using the following update step:

$$\Delta \boldsymbol{W}^l = \boldsymbol{W}^l_{new} - \boldsymbol{W}^l = -\eta \frac{\partial c}{\partial \boldsymbol{W}^l}. \tag{39}$$

The parameter η is the step size of the descent. As we mentioned in the introduction, picking the right value is an empirical issue, and might affect convergence.

To summarize, training reduces to the following two-pass algorithm:

1. Initialize the weight vectors to some values.
2. Select a sample point in the training data.
3. Pass that point through the network to calculate the output, storing all the intermediate—hidden—outputs. This step is also known as the *forward pass*.
4. Calculate the error at the output.
5. Use the *backpropagation algorithm* to calculate the gradients of the weight matrices. This is the *backward pass*.
6. Update the weight matrices using equation (39).
7. Select the next training sample point and repeat from 3, until the mean squared error is sufficiently low.

The first step in the process is quite significant. Current practice in the field dictates that initialization should be done with small, random zero-mean weights. Indeed, it is very difficult to find a good set of meaningful initial weights, therefore a random assignment is the best solution. Furthermore, the weights should have a small magnitude so that the derivative of the activation function is large initially. Had the weights been large in magnitude, then the algorithm would operate on areas of the sigmoid function where the derivative is small—sometimes zero, if we consider numerical quantization issues—so the gradient descent will not be able to move away from these points fast enough.

The *backpropagation algorithm* can be summarized as follows:

1. Set $l=L$ to select the output layer
2. Use equations (33) and (34) to calculate the gradient for the weight matrix and the local error signal for the current layer.

3. Set *l=l-1* and repeat 2 until the gradients for all the layers have been calculated.

There are variations of these two algorithms that are often used in practice. For example, to minimize the chances of the network falling into a small local minimum, some implementations randomly permute the order of the data that they present to the network. Other implementations use a set of data for cross-validation only, and stop the training early if the error in the cross-validation set starts to increase. This ensures that the network will not overtrain to the training data.

The are also variations aiming to protect the gradient descent from problems in the data. Indeed, certain implementations present a set of data to the network, calculate the gradients for each point in the set, and then use an average of the gradients to perform the update. This shields the training algorithm from aberrant or noisy data that might modify the cost landscape significantly, and force the gradient descent to take an undesired step. Another approach is to add *momentum* to the gradient, i.e. to update the matrices using a fraction of the $\Delta \boldsymbol{W}$ of the previous iteration together with the $\Delta \boldsymbol{W}$ calculated at the current iteration. Momentum, apart from the shielding it offers, often results in faster training.

Having established the training method, we only need to calculate the formula for the gradient of the specific functions we are going to use in the network. However, using equations (11) and (17), it is very easy to show that

$$\frac{\partial}{\partial \boldsymbol{x}} \boldsymbol{f}(\boldsymbol{x}) = \mathrm{diag}\Big(\big[0 \;\; f(x_1)(1-f(x_1)) \;\; \ldots \;\; f(x_N)(1-f(x_N)) \big] \Big), \tag{40}$$

$$\text{i.e. } \left(\frac{\partial}{\partial \boldsymbol{x}} \boldsymbol{f}(\boldsymbol{x}) \right)_{ij} = \begin{cases} 0, i = 1 \\ f(x_{i-1})(1-f(x_{i-1}))\delta_{ij}, i > 1 \end{cases} \tag{41}$$

where *f(x)* is the sigmoid function, and *N* is the length of the vector $\boldsymbol{x}$. Indeed, for all the hidden layers, this matrix is the correct derivative. However, in the output layer there is no constant output to provide the bias for the next layer since there exists no next layer. Therefore, the gradient at the output layer will be:

$$\frac{\partial}{\partial \boldsymbol{x}} \boldsymbol{f}^L(\boldsymbol{x}) = \text{diag}\Big(\big[f(x_1)(1-f(x_1)) \ \dots \ f(x_N)(1-f(x_N))\big]\Big), \tag{42}$$

$$\text{i.e. } \left(\frac{\partial}{\partial \boldsymbol{x}} \boldsymbol{f}(\boldsymbol{x})\right)_{ij} = f(x_i)(1-f(x_i))\delta_{ij} \tag{43}$$

Given these matrices it is trivial to implement a neural network to approximate any function with outputs constrained to be between 0 and 1. Still, this constraint will be sufficient to estimate probabilities of single events but not sufficient to estimate probabilities of mutually exclusive ones. For that we will introduce the *softmax* activation function for the output layer.

The *softmax* Activation Function

It is very often the case that a neural network will be used to estimate the probabilities of mutually exclusive events. An example scenario is a network that will recognize five kinds of classes. In that case, we might be interested in obtaining probability estimates about the data point being from each of these classes. The laws of probability would constrain such estimates to sum up to 1. However, a neural network with a sigmoid output layer would not guarantee such a constraint. For that we will introduce the *softmax* activation function.

The development in the previous sections showed that a layer is nothing more than a vector function. In the examples we examined, the output of a neuron only depended on the weighted sum of its input, resulting to an activation function with a diagonal gradient matrix, such as the one in equation (42). Having approached the network, however, from a vector function perspective, we may now define more complex activation functions at the same ease as the simple ones. Indeed, we will define the *softmax* activation function as follows:

$$\boldsymbol{f}(\boldsymbol{x}) = \left[\frac{e^{x_1}}{\sum_{i=1}^{N} e^{x_i}} \ \cdots \ \frac{e^{x_N}}{\sum_{i=1}^{N} e^{x_i}}\right]^T . \tag{44}$$

There are several nice properties associated with the softmax function. Its construction guarantees that the output vector will sum up to 1, as required for the probabilities of mutually exclusive events. Furthermore, it enhances the dominant input, driving the respective output close to 1, while driving the other outputs close to 0. Hence, it acts as a differentiable approximation of the *maximum* function, which assigns 1 to the output corresponding to the largest input and 0 to all other output. We should note here that the output of the previous layer should not be multiplied by a weighting matrix $\boldsymbol{W}$ but by a single constant α, which is the only parameter on which to perform the gradient descent. In other words, the output as a function of the output of the previous layer will be:

$$\boldsymbol{y}^l = \boldsymbol{f}(\alpha \boldsymbol{x}^{l-1}) = \left[\frac{e^{\alpha x_1^{l-1}}}{\sum_{i=1}^{N} e^{\alpha x_i^{l-1}}} \cdots \frac{e^{\alpha x_N^{l-1}}}{\sum_{i=1}^{N} e^{\alpha x_i^{l-1}}} \right]^T \tag{45}$$

The magnitude of α determines how closely the function approximates the *maximum* operation.

In order to implement a network with a softmax output, we need to calculate the gradient of the function with respect to its inputs. The difference in this case is that the gradient matrix will not be diagonal, since all the outputs depend on all the inputs. However, it is easy to show that:

$$\left(\frac{\partial}{\partial \boldsymbol{x}} \boldsymbol{f}(\boldsymbol{x}) \right)_{ij} = \frac{\partial}{\partial x_i} f_j(\boldsymbol{x}) = \begin{cases} f_j(\boldsymbol{x})(1 - f_j(\boldsymbol{x})), & i = j \\ -f_i(\boldsymbol{x}) f_j(\boldsymbol{x}), & i \neq j \end{cases}. \tag{46}$$

We can also express that in a form that might be more suitable for implementation be defining $\boldsymbol{1}$, the *Nx1* column vector that contains ones:

$$\mathbf{1} = \begin{bmatrix} 1 & \cdots & 1 \end{bmatrix} \tag{47}$$

and $\boldsymbol{I}$, the *NxN* identity matrix that contains ones in its diagonal, and zeros elsewhere:

$$\boldsymbol{I} = \text{diag}(\mathbf{1}) = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix}, \tag{48}$$

then we can write the gradient as

$$\frac{\partial}{\partial \boldsymbol{x}} \boldsymbol{f}(\boldsymbol{x}) = (\boldsymbol{I} - \boldsymbol{f}(\boldsymbol{x})(\mathbf{1}^T))\,\text{diag}(\boldsymbol{f}(\boldsymbol{x}))\,. \tag{49}$$

Indeed, this last form is very efficient to implement on matrix based systems.

Having formulated and analyzed the softmax activation function, we may form an ANN classifier using any network topology we desire in the hidden layers and appending a softmax layer at the end to ensure that the outputs obey the probability rules. With this formulation we will conclude the presentation of patter recognition techniques for static data and proceed to examine Hidden Markov Models and time varying data.

Hidden Markov Models

Hidden Markov Models are a very powerful class to describe time varying data. They have been successfully used in speech recognition engines. In fact, speech recognition research has driven most of their development. The models are extensions of a fundamental class of stochastic processes known as *Markov chain*s. Before we develop the models, therefore, we should become familiar with these processes. After we do so, we will present the extension to HMMs. Then we will present how training and classification is performed using these models. Finally, we will combine HMMs with ANNs to create a hybrid system, and show one approach to train it. For the development of all but the two final sections we will follow closely, but not exactly, Rabiner's ([20]) approach, and as similar as possible notation. For further information on HMMs, Rabiner's tutorial is an excellent start.

An Introduction to Markov Chains

One very powerful property of time series is the *Markov property.* A time series is *Markov* if knowledge of its value at some point in time incorporates all the knowledge about times before that point. In more rigorous terms, a process *q[n]* is Markov if:

$$p(q[n]|q[n-1], q[n-2], \ldots) = p(q[n]|q[n-1]). \quad (50)$$

We can also show that given the condition above, the following is also true for all values of $k>0$:

$$p(q[n]|q[n-k], q[n-k-1], \ldots) = p(q[n]|q[n-k]). \quad (51)$$

In other words, the value of the time series at any point incorporates all the history of the time series until that point. It is, therefore, said it incorporates the *state* of the system

Based on that premise, one can formulate a stochastic model with a finite number of states obeying the Markov property. A small such model can be seen in Figure 11. The model has N states, denoted by S_i, $1 \le i \le N$. The state of the model at time t is the time series $q[t]$. Time is discrete, starts at $t=1$ and extends until $t=T$. Furthermore, we will denote the probability of transition from one state to the other using $a_{ij}=P(q[t+1]=i|q[t]=j)$. Finally, we will use π_i to denote the probability that the model initially starts at state i. It is worth mentioning here that, as with the neural networks, matrix notation will prove very convenient. Therefore, we will use π to denote the initial probability column vector and the matrix $\boldsymbol{A}$, with elements $(\boldsymbol{A})_{ij}=a_{ij}$, to denote the transition probability matrix.

We can show that if $\boldsymbol{p}[t]$ is the vector such that $p_i[t]=P(q[t]=i)$, then:

$$\boldsymbol{p}[t] = \boldsymbol{A}^{t-1}\pi = \boldsymbol{A}^{t-k}\boldsymbol{p}[k] \quad (52)$$

Furthermore, because of the laws of probability, all these quantities should sum up to one:

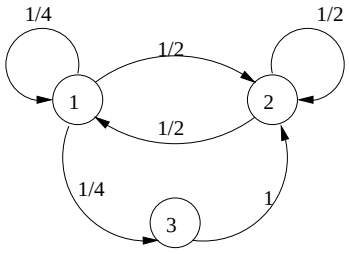

FIGURE 11. An example of a three-state markov chain. The numbers on the arrows of the graph reflect the transition probabilities a_{ij}.

$$\sum_{i=1}^{N} p_i[t] = \sum_{i=1}^{N} \pi_i = \sum_{j=1}^{N} a_{ij} = 1 \tag{53}$$

Another feature—or issue, depending on the application—of the model is an implicit state duration assumption. In particular, the distribution of the state duration d_i—that is, the number of times the state transitions back to itself—follows the geometric distribution:

$$P(d_i = d) = a_{ii}^{d-1}(1 - a_{ii}) \tag{54}$$

This distribution is a discrete approximation of the exponential distribution, the only memoryless continuous distribution. The mean of the geometric distribution is:

$$E(d_i) = \frac{1}{1 - a_{ii}}, \tag{55}$$

which implies that an estimate of the parameter a_{ii} may be calculated using an estimate $\tilde{d}_i$ of the duration:

$$a_{ii} \approx 1 - \frac{1}{\tilde{d}_i} \tag{56}$$

This will be used later to initialize the model parameters.

There are several other interesting properties of the markov chains—for example the matrix $\boldsymbol{A}$ has positive eigenvalues all less than or equal to *1*. However, these are not used in the development of HMMs, so we will not discuss them here. An extensive presentation may be found in Gallager [11], or in Papoulis [19]. For our purposes, we only need to "hide" the chain, to develop the HMMs we are after.

Hiding the Chain

When we try to model observed signals and processes, we need to extend Markov chains. Even if the process we observe can be closely approximated by a Markov chain, it is usually not possible to observe the current state of the process directly. Therefore, we have to make the assumption that the underlying model is hidden: we only observe what each state *emits*. We will assume that the emissions of each state follow a probabilistic model, and these are the features we observe. To train the model we will need to estimate both the parameters for the Markov chain and the parameters for the emission model.

To hide the chain, we will assume that there exists an observation vector $\boldsymbol{O}[t]$, which is emitted from the model at time *t*. Each state has emission density function $b_i(\boldsymbol{O})=p(\text{state } i \text{ emits vector } \boldsymbol{O})$, which is independent of time and independent of any previous state emission. In other word, when the model is in state *i*, the state emits a random vector $\boldsymbol{O}$, according to the density $b_i(\boldsymbol{O})$. We can also think about this model as follows: each state emits a series of i.i.d. random vectors $\boldsymbol{O}_i[t]$, and the vector $\boldsymbol{O}[t]$ gets the value of the vector $\boldsymbol{O}_j[t]$, where *j* is the state of the model at time *t*. We will assume that the emission density of each state is a mixture of *M* Gaussians, *M* being the same for all states. In other words:

$$b_j(\boldsymbol{O}) = \sum_{m=1}^{M} c_{jm} N(\boldsymbol{O}, \mu_{jm}, \boldsymbol{U}_{jm}), \tag{57}$$

where

$$\sum_{m=1}^{M} c_{jm} = 1, \tag{58}$$

so that the density has unit area.

It is obvious from the above discussion that the transition matrix $\boldsymbol{A}$, the initial probability vector π, and the set of state emission density functions $b_i(\boldsymbol{O})$, $1 \le i \le N$ completely define a hidden markov model. Henceforth, we will use λ to denote this set of parameters, i.e. to completely describe a model.

"The Three Basic Problems"

As Rabiner points out, there are three basic problems that are of interest when a process is modeled using HMMs. The three problems are presented in a different order ("Problem 3" is presented before "Problem 2") because this order emphasizes the conceptual links between the problems, which are crucial to understanding the intuition:

Estimating the probability of the observation sequence: Given an observation vector *O[t]*, and a model λ, how can we estimate efficiently the likelihood of the observation sequence?

Estimating the model parameters: Given an observation sequence produced by that model, how can we estimate the model parameters to maximize the likelihood of that observation sequence?

Estimating the state transitions: Given an observation sequence and a model, how can we determine a sequence of state transitions *q[t]* that corresponds to that observation sequence?

Solving the second problem is equivalent to training the model. In practice, this would be the first problem to tackle. However, the intuition we gain by solving the first problem will be very useful in estimating the model parameters. Therefore, we will examine how to classify using a trained model first—i.e. how to solve the first problem. The third problem can also be used for classification, depending on the application. In fact, we will use the third method for DNA basecalling, later in the thesis.

And Their Solutions

I will present a brief solution to the three problems here, meant as a reference. Rabiner presents a detailed treatment, a definite reading to obtain the full intuition.

Estimating the probability of the observation sequence: To do so, we will define the *forward* variable

$$\alpha_i[t] = p(\boldsymbol{O}[1], \boldsymbol{O}[2], \ldots, \boldsymbol{O}[t], q[t] = i|\lambda) \tag{59}$$

This variable can be computed inductively, in order *O(T)* using:

$$\alpha_j[t+1] = \left[\sum_{i=1}^{N} \alpha_i[t] a_{ij}\right] b_j(\boldsymbol{O}[t+1]), \tag{60}$$

where

$$\alpha_i[1] = \pi_i b_i(\boldsymbol{O}[1]), \tag{61}$$

and, at time *T*:

$$p(\boldsymbol{O}[1\ldots T]|\lambda) = \sum_{i=1}^{N} \alpha_i[T], \tag{62}$$

where we denote the whole observation vector with $\boldsymbol{O}[1...T]$, and the conditional density with $p(\boldsymbol{O}[1\ldots T]|\lambda)$.

Estimating the model parameters: Having defined and calculated the forward variable, we also define the *backward* variable, which we can calculate recursively:

$$\beta_i[t] = p(\boldsymbol{O}[t+1], \boldsymbol{O}[t+2], \ldots, \boldsymbol{O}[T]|q[T] = i, \lambda), \tag{63}$$

$$\text{where } \beta_i[T] = 1, \tag{64}$$

$$\text{and } \beta_i[t] = \sum_{j=1}^{N} a_{ij} b_j(\boldsymbol{O}[t+1]) \beta_j[t+1]. \tag{65}$$

This recursion is backwards, starting from $t=T$, down to 1. Given α, and β, we can now define

$$\gamma_i[t] = p(q[t] = i | \boldsymbol{O}[1 \ldots T], \lambda), \tag{66}$$

which can be easily calculated using:

$$\gamma_i[t] = \frac{\alpha_i[t]\beta_i[t]}{p(\boldsymbol{O}[1 \ldots T]|\lambda)} = \frac{\alpha_i[t]\beta_i[t]}{\sum_{j=1}^{N} \alpha_j[t]\beta_j[t]}. \tag{67}$$

Finally, we define

$$\xi_{ij}[t] = p(q[t] = i, q[t+1] = j | \boldsymbol{O}[1 \ldots T], \lambda), \tag{68}$$

and we use:

$$\xi_{ij}[t] = \frac{\alpha_i[t] a_{ij} b_j(\boldsymbol{O}[t+1]) \beta_j[t+1]}{P(\boldsymbol{O}[1 \ldots T]|\lambda)}. \tag{69}$$

Note that,

$$\gamma_i[t] = \sum_{j=1}^{N} \xi_{ij}[t] \tag{70}$$

which can be another way to compute γ, often more desirable for numerical stability reasons.

Using these variables, we perform an update step:

$$\bar{\pi} = \gamma_i[1] \tag{71}$$

$$\bar{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_{ij}[t]}{\sum_{t=1}^{T-1} \sum_{i=1} \gamma_i[t]} \tag{72}$$

$$\bar{c}_{jk} = \frac{\sum_{t=1}^{T} \gamma_{jk}[t]}{\sum_{t=1}^{T} \sum_{l=1}^{M} \gamma_{jl}[t]} = \frac{\sum_{t=1}^{T} \gamma_{jk}[t]}{\sum_{t=1}^{T} \gamma_j[t]} \tag{73}$$

$$\bar{\mu}_{jk} = \frac{\sum_{t=1}^{T} \gamma_{jk}[t] \cdot \boldsymbol{O}[t]}{\sum_{t=1}^{T} \gamma_{jk}[t]} \tag{74}$$

$$\bar{\boldsymbol{U}}_{jk} = \frac{\sum_{t=1}^{T} \gamma_{jk}[t] \cdot (\boldsymbol{O}[t] - \mu_{jk})(\boldsymbol{O}[t] - \mu_{jk})^T}{\sum_{t=1}^{T} \gamma_{jk}[t]}, \tag{75}$$

$$\text{where } \gamma_{jk}[t] = \gamma_j[t] \frac{c_{jk} N(\boldsymbol{O}[t], \mu_{jk}, \boldsymbol{U}_{jk})}{\sum_{m=1}^{M} c_{jm} N(\boldsymbol{O}[t], \mu_{jm}, \boldsymbol{U}_{jm})}. \tag{76}$$

This quantity can be thought of as the probability of being in state *j* at time *t*, with the k^{th} component of the mixture accounting for ***O[t]***. It is obvious that

$$\sum_{k=1}^{M} \gamma_{jk}[t] = \gamma_j[t]. \tag{77}$$

The two steps (calculating α, β, γ and ξ, and updating λ) are repeated a number of times, until the model error in the data converges in a meaningful sense. This algorithm is known as the *Baum-Welch algorithm*, and is essentially an implementation of the EM idea. The expectation step—i.e. the calculation of α, β, γ and ξ—is also known as the *forward-backward procedure*.

Before we continue we should point out that the update equations for the mixture model parameters (73-75) are remarkably similar to the update equations of the static mixture models (2-4), with $\hat{p}_m$ being substituted by γ_{jm}. We will exploit this to justify the expansion of Hidden Markov Models to use ANNs for density estimation.

Estimating the state transitions: To estimate the most likely sequence of state transitions for a given observation sequence we will use the Viterbi algorithm. Thus, we define:

$$\delta_i[t] = \max_{q[1], \ldots, q[t-1]} P(q[1] \ldots q[t-1], q[t] = i, \boldsymbol{O}[1], \ldots, \boldsymbol{O}[t] | \lambda), \quad (78)$$

which can be recursively computed using

$$\delta_j[t] = \left[\max_{1 \le i \le N} \delta_i[t-1] a_{ij} \right] b_j(\boldsymbol{O}[t]), \quad (79)$$

$$\delta_i[1] = \pi_i b_i(\boldsymbol{O}[1]) \quad (80)$$

We also need to keep track of the state sequence using:

$$\psi_j[t] = \operatorname*{argmax}_{1 \le i \le N} \left[\delta_i[t-1] a_{ij} \right] \quad (81)$$

$$\psi_j[1] = 0 \quad (82)$$

After computing $\delta_i[t]$ and $\psi_i[t]$ for all *i* and *t*, we can backtrack to find the estimated state sequence *q*[t]* using

$$q^*[T] = \operatorname*{argmax}_{1 \le i \le N} \left[\delta_i[T] \right] \quad (83)$$

$$\text{and } q^*[t] = \psi_{q^*[t+1]}[t+1]. \tag{84}$$

The estimated state sequence is the most likely sequence to produce the observation sequence, subject to the sequence obeying the transition constraints of the HMM. We could use γ from the forward-backward procedure to generate the sequence of the most likely state at each point in time, but this might result in state transitions that are not allowed.

Combining HMMs and ANNs

One of the first contributions of this thesis is the combination of HMMs with ANNs using a variant of the Baum-Welch algorithm that calls the ANN Backpropagation algorithm in the update step. ANNs are more powerful than Gaussian Mixtures models. Therefore it is desirable to combine ANNs with HMMs to model complicated signals such as electropherograms.

We will use a neural network that has a softmax output activation function. In other words, the output vector $\boldsymbol{y}$ of the network will sum to 1, so it can be considered a vector of probabilities of mutually exclusive events. Indeed, we will treat each component y_i as the probability that the feature vector $\boldsymbol{O}$—the input to the ANN—is emitted from state i. Obviously, the output of the ANN will have length N, the number of states in the HMM.

Given that network, we are able to use the forward-backward procedure described above, using $b_j(\boldsymbol{O}) = y_j$ instead of equation (57). In other words, we will use the ANN to model the emission probabilities of each state. This substitution makes equations (73-78) obsolete. Still, it does not affect the remaining equations which can be used to estimate $\gamma_j[t] = p(q[t] = j|\boldsymbol{O}[1\ldots T], \lambda)$.

Now we can perform the update step of the Baum-Welch variation, which is nothing more than treating $\gamma_i[t]$ as the target output points of our ANN for the respective input vectors $\boldsymbol{O}[t]$. These T sample points are fed to the error backpropagation algorithm to reestimate the ANN weight matrices. The two step backpropagation is executed for a couple of iterations, and then the new neu-

ral network is used to initiate a new iteration of the modified Baum-Welch algorithm. A small variation that we should probably add to avoid training bias is to present the training pairs $(\gamma_i[t], \boldsymbol{O}[t])$ in a different, random order to the neural network each time we perform the backpropagation.

The training process of the HMM could be summarized in Figure 12. The figure emphasizes the observations we made on the HMMs with state emissions modeled as mixtures of gaussians, and how these extended to train ANNs: the training procedure is essentially a two step process treating the static model as given in one step to determine the parameters of the markov chain, and treating the markov chain as given in the other step to train the static model. It is very similar to the EM algorithm, and we could in fact classify it as *generalized EM* algorithm. We should note however that we have not proved convergence and therefore cannot guarantee it for the general case.

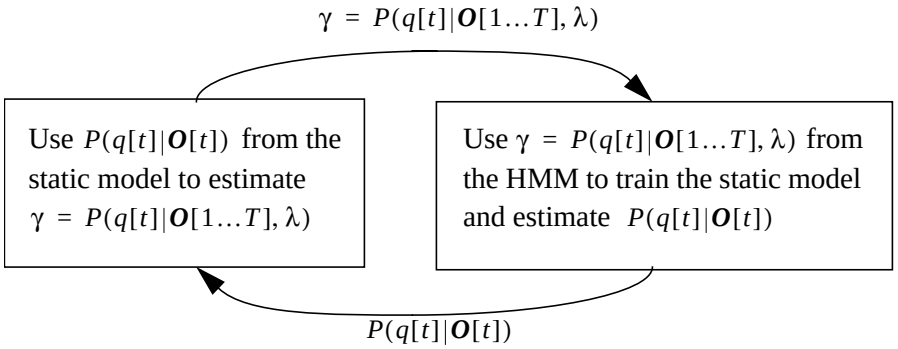

FIGURE 12. A schematic summary of the training process of the HMM. The training process applies to HMMs with state emissions estimated both using Gaussian mixtures and Artificial neural networks.

The new method creates a number of new design parameters in the HMM-ANN design space. Indeed, apart from the topology of the two networks, the training step and the number of backpropagation iterations for every Baum-Welch iteration also need to be specified. If we train the neural network heavily at each step, then the succeeding forward-backward step of the Baum-

Welch algorithm might not be able to move the system significantly on the parameter space, and the system will soon be stuck in a local minimum. If we decide not to iterate the backpropagation many times for each update step, then the algorithm will take a significant amount of Baum-Welch iterations to converge to a stable point. The right balance is usually achieved empirically by trial and error.

The scheme proposed above has some significant advantages over other similar HMM and ANN combinations such as [4] and [1]. Firstly, it trains the HMM and the ANN in a combined way: alternative proposals use ANNs to estimate state emission densities, but train them separately from the HMM, usually on manually labeled data. These methods increase the potential of human error or subjectivity and make the labeling process cumbersome and expensive. If, for example, a state i in a model transitions to itself or to one other terminal state $i+1$, it is not easy for a human to figure the exact transition point in the time series, leading to arbitrary state assignments and bias. On the other hand, a Baum-Welch reestimation approach will move the transition point until the statistics agree with the data.

Still, other methods combine HMM and ANN training and do not require separate training of the models and human labeling. Compared to these, the proposed method relies on integrating the most studied methods for training HMMs and ANNs—Baum-Welch and Error Backpropagation, respectively. This slows down the training process slightly, but allows us to exploit all the developments in these two fields to improve both the results and the efficiency of our implementation.

Summary

In this chapter we reviewed some basic probabilistic models, and examined how we can combine them to build stronger models. Specifically, we examined pattern recognition on static data, using Gaussian Mixture models and Artificial Neural Networks. We studied Markov chains and combined them with static models to create Hidden Markov Models for pattern recognition. To improve recognition ability, we designed a Hidden Markov Model using an Artificial Neural Network to estimate the state emission probabilities. Finally,

we proposed a training mechanism for the system that exploits extensively studied methods in both fields. This mechanism has several advantages—and some disadvantages—over similar proposals. We are now ready to explore the applications of these algorithms to DNA sequencing problems.

CHAPTER 3

DNA sequencing as a Pattern Recognition Problem

Armed with the tools we developed in the previous chapter, we are ready to tackle DNA basecalling. We will formulate the problem as a pattern recognition problem and acknowledge that HMMs are particularly suitable for the solution. We will develop a network topology that fits the problem, and train the model using sample data. For the training process we will need to develop a method to create a large sample of training data, and for that we will also create an algorithm to execute queries on databases using partly trained models. Finally, we will explore alternative topologies that can be used to approach the problem.

The Pattern Recognition Problem

As we mentioned in the introduction, DNA can be represented using a finite sequence s_i, $1 \le i \le B$ of letters drawn from the alphabet {A, T, C, G}, with an i.i.d. distribution. However, the DNA sequencing equipment returns a finite length discrete-time signal $e[t]$, $1 \le t \le T$ produced by the sequencing process, the *electropherogram*. To sequence this signal we need to assume that it is emitted from the DNA sequence s_i under a probabilistic model. Given that

model, the basecaller needs to estimate the most likely number $\hat{B}$, and sequence $\hat{s}_i$ of bases. In other words it needs to solve the following maximization problem:

$$(\hat{B}, \hat{s}_i, 1 \le i \le B) = \underset{B,\, s_i,\, 1 \le i \le B}{\operatorname{argmax}} P(B, s_i, 1 \le i \le B | e[t]) . \quad (85)$$

A first issue directly arising with this problem is the size of the discrete search space: A typical output of the equipment will correspond to about 700 bases, creating on the order of 4^{700} possible combinations of letters. Since the space is discrete, it is impossible to explore it with techniques such as gradient descent. Still, we can see that this probabilistic formulation of the problem hints to the use of pattern recognition techniques to model the data.

A deeper look at the problem will show that it is very similar to the speech recognition problem. Indeed, in speech recognition we are trying to form the most likely sequence of symbols from a set given the speech signal. Depending on the formulation, these symbols can be phonemes, syllables, or words. A simple example is the recognition of numbers spelled out by the user. the symbols are the set {‘zero’, ‘one’, ... ,‘nine’}, and we are trying to find a sequence—for example ‘one’, ‘eight’, ‘zero’, ‘zero’,...—which corresponds to what the user said.

This is a problem extensively studied by the speech recognition community, and we will try their approach in tackling the—almost identical—DNA sequencing problem. Note that we chose the simple number transcription problem, as opposed to the full speech recognition one, to avoid complications such as the vocabulary probabilities and the grammar models that are introduced in the complete speech recognition problems. These complications do not have a direct equivalent in DNA sequencing, since the fundamental assumption is that the sequence of symbols is an i.i.d. process with all the four letters being equiprobable.

One final point we need to stress is that maximizing the likelihood of the data is not necessarily the best thing to do. Ideally, we need to define a cost function of the predicted and the actual DNA sequence and try to minimize that function. For example that function could assign a cost of 1 to every prediction error, and a cost of zero to every correct call. The goal of the system,

then, should be to minimize the probability of base call error *which is not necessarily the same as maximizing the likelihood of equation (85).* However, this problem often becomes intractable for complex systems such as this one. If indeed the problem is tractable for a given cost function, then it should be easy to convert the algorithms to optimize the required metric. For more details, see [8].

HMM Topology

Having established the similarity of basecalling with speech recognition, we are ready to examine the particular structure of the HMMs we will use to solve the problem. Our "words" will be the four DNA bases, {A, T, C, G} and we can represent the i.i.d. sequence of bases as the markov model shown in Figure 13, where all transition probabilities are equal to 1/4. We use the

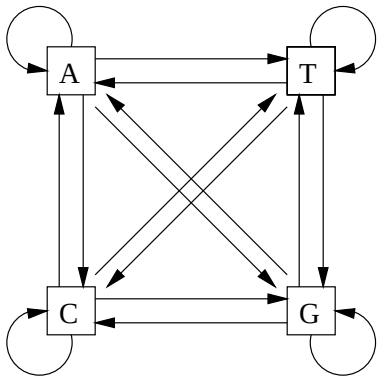

FIGURE 13. A model to produce an IID sequence of bases. All transition probabilities are equal to 1/4.

boxes instead of circles for representing the states in that model because the state transitions in that figure correspond to transitioning from one letter to another in the sequence we are trying to determine, and not time transitions in

the observed electropherogram. Referring to our speech recognition analogy, transitions in this model are equivalent to transitions from a word to the next.

The Bases Model

In order to implement a Hidden Markov Model we need to determine a model for each base that replaces the box of Figure 13. Such a model should have transitions that correspond to sampling time transitions and describe in more detail the structure of the observed signal for each base. To develop that model, we need to take a close look at the electropherogram and decide what a base looks like. Indeed, from Figure 14 we can determine that a "typical"

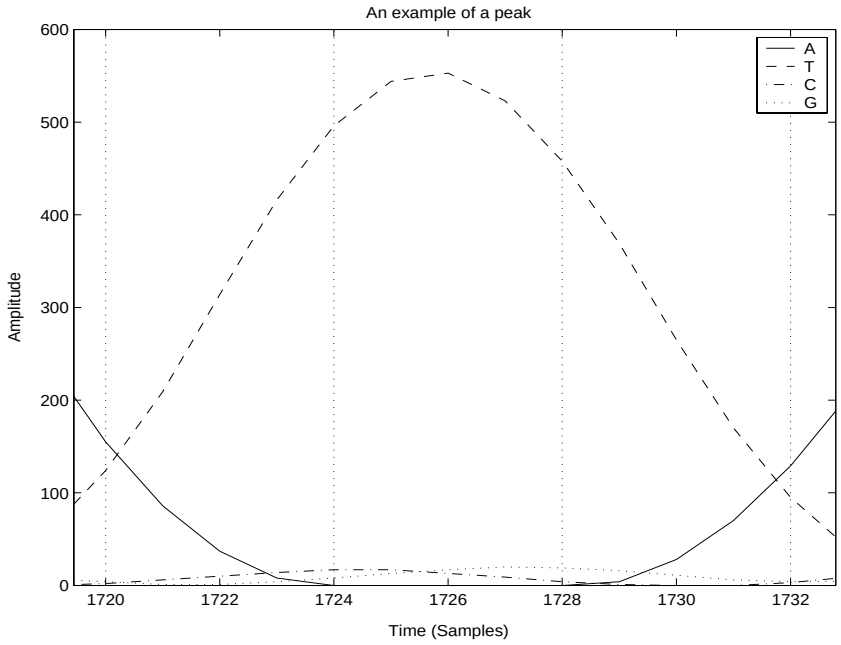

FIGURE 14. A typical base, as it shows up on the electropherogram. The figure is separated (somewhat arbitrarily here) into three segments: the rise, the inflection, and the fall.

base representation is just a peak. It is usually modeled as a gaussian peak convolved with a decaying exponential. A simple model for this peak could be a three state model, as shown in Figure 15: the base enters the detection

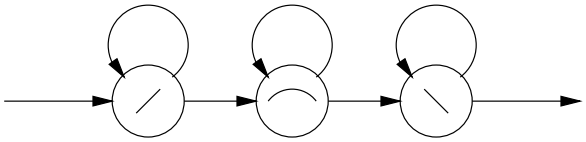

FIGURE 15. A simple Markov model describing the state transitions of a peak in time: rising, peaking, and falling.

system, the base is in the system, and the base exits the detection system, corresponding to the rise, the plateau, and the fall of the gaussian. We will see soon that it is a fairly good first order model.

The Basecalling Model

To determine the final HMM structure to be used for basecalling we can combine Figures 13 and 15 to get Figure 16. This creates a 12 state model, which we will use for the recognition once we determine the necessary parameters. Note that Figure 16 is visually cluttered so we will simplify it to Figure 17. Unfortunately, this model is useless for training since it is agnostic of the DNA sequence of letters that produced the training sequences. To remedy that, we will need to produce a model that incorporates this information.

The training model is best illustrated by an example. We will assume that the training sample we have at hand is the electropherogram of AATCA. This would produce a base sequence model as shown in the top part of Figure 18. As above, we can substitute the individual bases model in the boxes, and get the bottom part of the figure. We will feed that model to the training algorithm to represent the corresponding electropherogram. These types of models, with sequential states that only self loop or transition to the next state but never transition back to a previous state, are often called linear models.

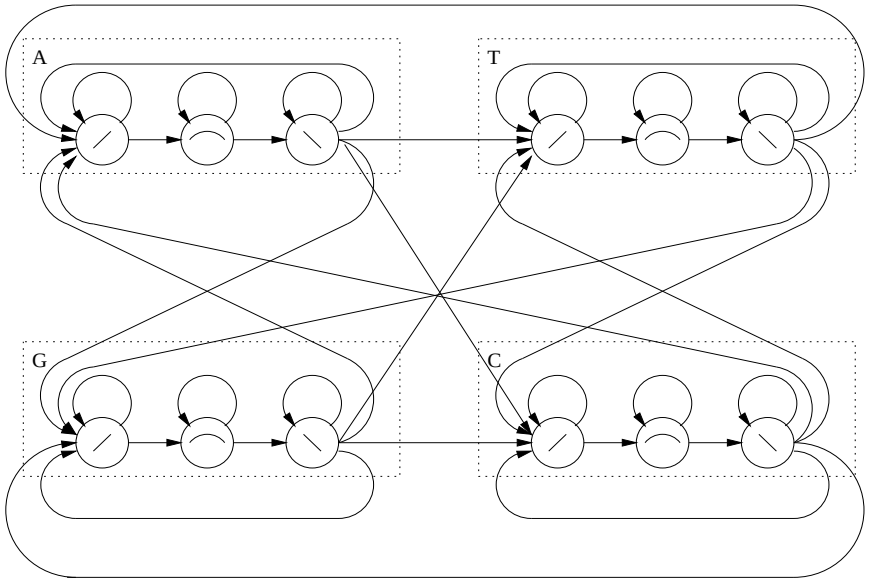

FIGURE 16. The final 12-state model to be used for basecalling

There is an important distinction between the linear model and the model of Figure 16. In the linear model each state is tied to the specific letter position it corresponds to. For example, consider the first and the fourth state of the linear model. Although they both have the same emission density functions, they represent two different system states: the first A entering the system, and the second A entering the system. The non-linear model above would cycle back to A right after exiting from the third state of it, if it was used to recognize that particular letter sequence. It is therefore obvious that the linear model is constrained to the particular base sequence that generated it, which makes it suitable for training.

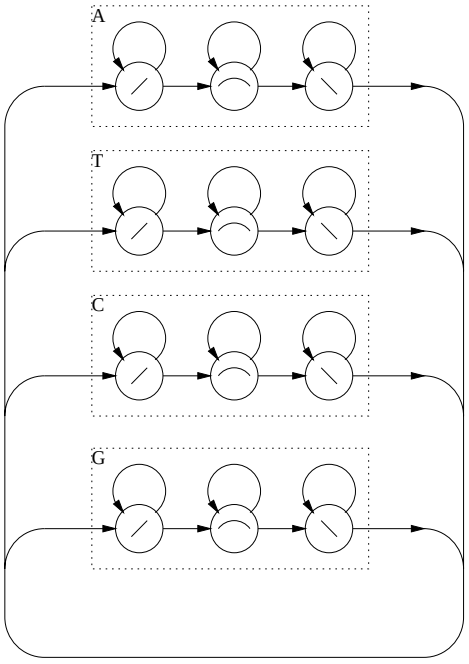

FIGURE 17. A schematic simplification of the model of Figure 16

System Training

In order to train the system we will need labelled electropherograms to create a training set. Unfortunately, labelled electropherogram data are next to impossible to find. Furthermore, hand labelling is a daunting and very expensive task which we would like to avoid as much as possible. Another option would be to use labelled data that have been generated by another base caller to train our system. This option has a number of problems. First, basecallers are not accurate enough for training purposes. Second, we train the system based on another system's decisions. This will result to a system emulating the other basecaller, including its defects. Finally, it creates a circular logic problem: if we need a basecaller to train another basecaller, then how do we

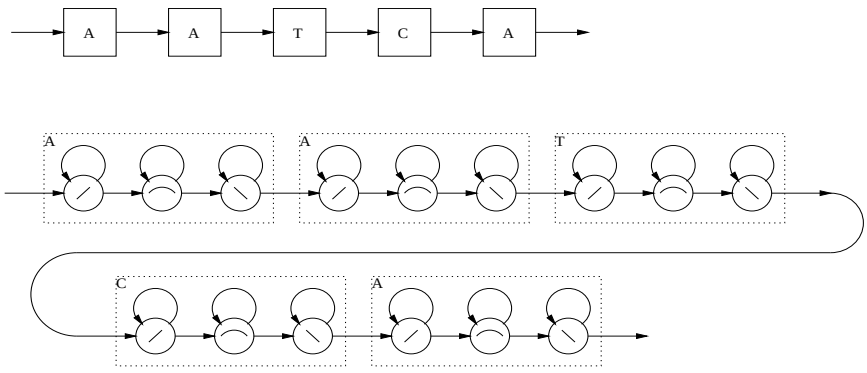

FIGURE 18. The linear model to be used for training purposes

train the first basecaller? Fortunately, we were able to devise a method to generate very good quality training sets with minor effort.

The consensus sequences.

To generate the data, we will exploit the availability of consensus sequences of the M13mp18 and the PBluescript genes. These genes—with a sequences of more than 7000 bases long—have been the subject of extensive study and are the DNA sequencing benchmarks. The published (in [21] and [22]) consensus sequences are the result of sequencing several times overlapping fragments of the genes, labelling them either by hand or by basecalling programs, and then combining the fragments to form consensus sequence that are correct with almost certainty for our purposes. Therefore, we can use electropherograms produced from sequencing these particular genes to train our system.

The solution is not perfect, yet. The problem is that electropherograms are usually on the order of 700 base pairs long but the exact starting point in the consensus sequence and their exact length is not known. The differences are due to variations to the conditions and the chemistry of the experiments, and are hard to predict. In order to generate a labelled training set from a set of electropherograms we will need first to identify which part of the sequence the electropherogram corresponds to.

This is an easier, but still laborious task. However, it could be automated if we had a trained Hidden Markov Model—which we do not. Still, we will show that instead we can use a poorly trained HMM to locate the electropherogram fragment in the consensus sequence. Thus we can bootstrap a poorly trained model to generate its own training data, by exploiting the side-information of the consensus sequence.

The training method.

In order to train our model, we will pick very few electropherograms and manually locate the corresponding fragments in the consensus sequence. We will use these electropherograms to train an HMM. This will produce a model that would make a poor basecaller. Still, this model is sufficient to run *queries*—which we will describe bellow—and match the unlabeled electropherograms to fragments in the consensus sequence. Thus, we can generate a significant number of electropherogram-sequence pairs to be used for training purposes. We will use the newly generated data to train a model that we can use afterwards for basecalling.

In order to train the basecalling model from labeled sample data, we will need to form one specific linear model for each training sequence. This model should encode the state sequence as defined by the base sequence of the training sample. A 100-bases long training sample will result to a 300-states long linear model. These models, together with the corresponding electropherograms will be passed trough the modified Baum-Welch reestimation procedure several times, until convergence of the parameters is achieved.

Executing Queries

Linear models are also the key to executing queries in the consensus sequence to find the location of an electropherogram. Indeed, we will use a variation of Viterbi algorithm on a linear model that corresponds to the consensus sequence. Although we will use this method for the specific purpose of creating training data, we believe that there might be a wide number of uses for it, especially for querying databases of discrete data using a continuous time signal.

Furthermore, the method can be extended in a trivial way to perform a query of a sequence into a signal, i.e. find a specific, small, state sequence—which will represent a base sequence—inside the electropherogram. This can have a number of applications, from SNP (single nucleotide polymorphism) detection to DNA figerprinting. Unfortunately, we do not have the time to explore these in this thesis.

The variation to the Viterbi algorithm

To perform the query, we will create a linear model of the corresponding consensus sequence. The model will have $N=3B$ states, where B is the number of bases of the sequence. The only variation we need to introduce is the initialization of the Viterbi algorithm to

$$\delta_i[1] = \frac{b_i(\boldsymbol{O}[1])}{N}, \tag{86}$$

instead of equation (80) of page 53.

The change reflects the fact that we do not really know where in the linear model our sequence starts. Therefore, we assign an equal probability to all the states of the model being the starting state, i.e. $\pi_i = \frac{1}{N}$. On that model we can run the Viterbi algorithm to find the optimal state sequence $q^*[t]$ for the given electropherogram. That state sequence can easily be converted to the base sequence corresponding to the electropherogram and create a labelled set to be used for training.

Alternative Topologies

The topologies we have proposed for basecalling are by no means the only possible topologies. In fact, our proposals could be modified—at the expense of complexity—to model the sequence better. We will mention some possible modifications here, but we will not explore them further. The goal of this work was to demonstrate that HMMs can be make good models for basecalling, not to find the best HMM topology.

Accommodating sequences of identical bases

Our models until now treat the transition from one base to another in the same way as the transition from one base to the same base. However, it is obvious from Figure 19 that the electropherogram of, for example, an AA sequence

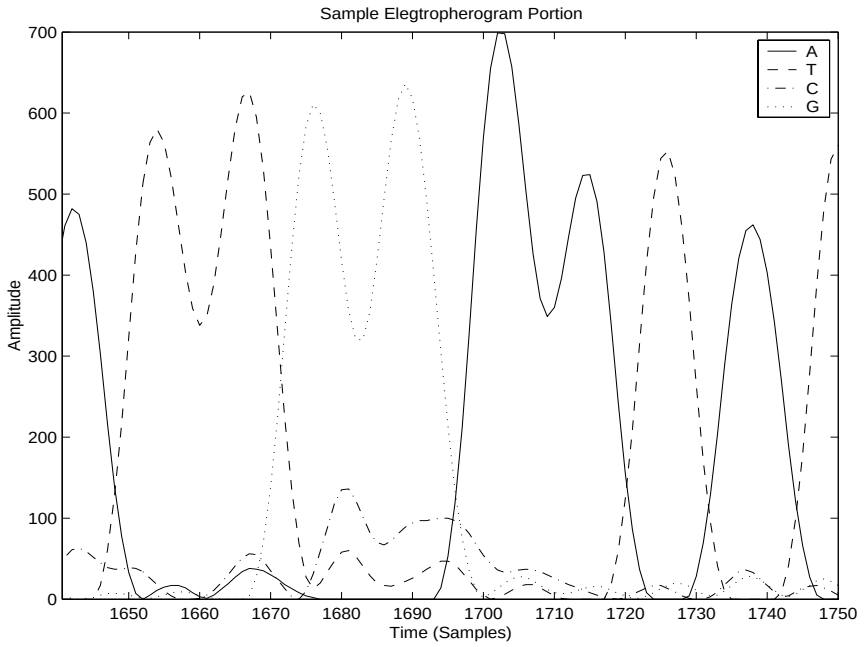

FIGURE 19. A sample electropherogram. It is obvious that the statistics of transition through consecutive peaks of the same base are much different that transitioning from a peak of one base to a peak of another.

has very different transition statistics than a TA sequence. That can be incorporated in the recognition model by modifying Figure 15 to look like one of the two models of Figure 20. Of course, the corresponding linear models used for training need to be modified accordingly. This model should improve the recognition performance in cases of repeated bases, at the expense of recognition—not of training—model complexity.

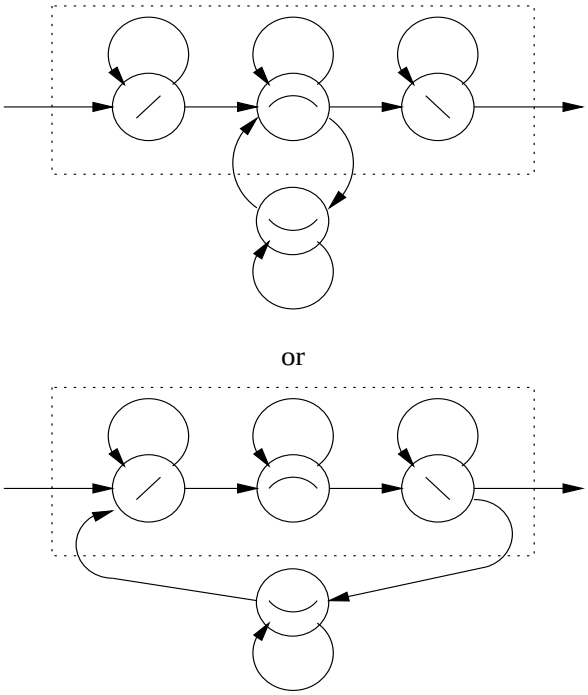

FIGURE 20. Two candidate base models to increase the recognition accuracy in sequences of identical bases.

Accommodating molecule compression effects

Our models assume that the transition statistics form one peak to the other in an electropherogram only depend on what the next peak is. That assumption partly implies that the width and the location of a peak only depends on the base creating that peak and not on what bases are around it. Although this is correct as a first order approximation, [3] shows that there is a correlation between interpeak spacing and the base before the one currently passing through the detector. They show that certain ending base pairs create compressions, making the fragments easier to pass through the gel. Thus the peaks arrive slightly earlier. For example, the peak corresponding to the final C in ATGATGC would be slightly advanced—but not out of order—com-

pared to a similar electropherogram for ATGATAC. They also show that there was no statistical evidence that earlier bases play a role, i.e. there is no reason to believe that as far as the peak for the final C is concerned, ATGATGC looks any different from ATGAAGC.

This destroys the Markov property of the model of Figure 13 since the statistics of any box in the model depend on where the transition happened from. However, it is easy to incorporate that in our model by augmenting the state space to include the previous base in the state. The resulting model is shown in Figure 21. The notation A|G means that the current peak corresponds to an A, given that it follows a G. Again, every box in the figure corresponds to a three state sequence, as in Figure 15.

The model is a very big improvement over our standard model, but at a cost. In fact, this model not only accommodates second order effects, but handles identical bases in a similar way as the model of Figure 20. The cost is that the state space of that model has increased fourfold. Still, this is not a significant issue: speech recognition models have significantly larger state spaces, of 42 phonemes and thousands of words, and the community has developed techniques to deal with such huge spaces.

Accommodating concurrent bases

In a different track than the previous models, one issue that often occurs in DNA sequencing problems is the concurrency of the peaks. Specifically, peaks near the end of the electropherogram might arrive with a very small time delay, not enough for the model to switch from one base to another. Figure 22 shows an instance of that problem. In this case our original model will just skip one of the two bases, making an error.

One possible modification to accommodate that issue is to run four different models on the same electropherogram, one for each base. The models should have inputs from all the signals of the electropherogram but only recognize the peak of their corresponding base. Furthermore, in order to be able to reconstruct the sequence from the individual outputs, it is important for the model to specify exactly where the peak location is. This can be accomplished by running four instances of the model in Figure 23, one for each base. We should notice that no self transitions are allowed in the middle state

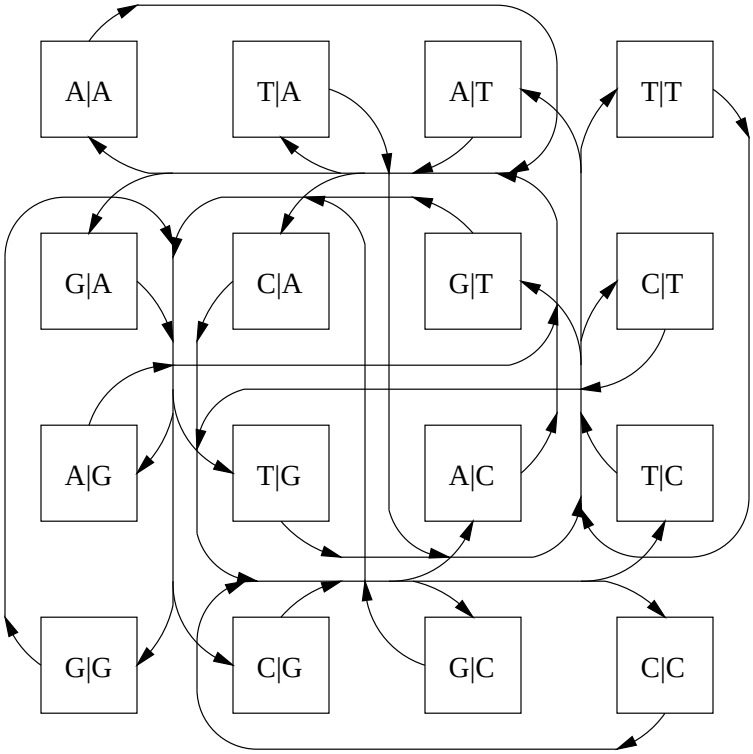

FIGURE 21. A more complicated model to accommodate second order effects in the electropherogram, such as GC compressions and sequences of identical bases.

of the base recognition sequence, so that the exact location of the peak is determined uniquely by the time instance that state is visited.

This model does not accommodate the issues that the first two models dealt with, only the base concurrency issue. However, one could extend this model, in a manner similar to the extensions above, to accommodate double peaks and second order effects.

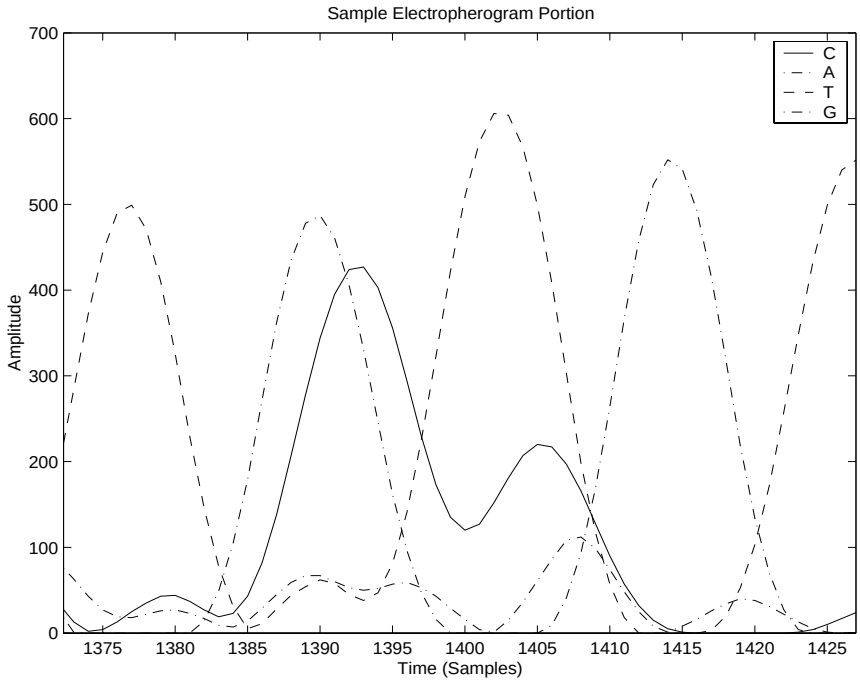

FIGURE 22. An instance of two peaks corresponding to different bases arriving very close in time. The model we initially proposed might not be able to transition in time, and ignore the existence of one of the peaks.

Summary

In this section we posed basecalling as a pattern recognition problem. We observed the striking similarities with the speech recognition problems and noticed that similar techniques can be used. Furthermore, we developed a very efficient method to execute queries in a long sequence to locate the position of an electropherogram, even using only partially trained models. This was a very good way to create labeled data, that are scarce and difficult to produce manually but are necessary for training the system. In addition, we

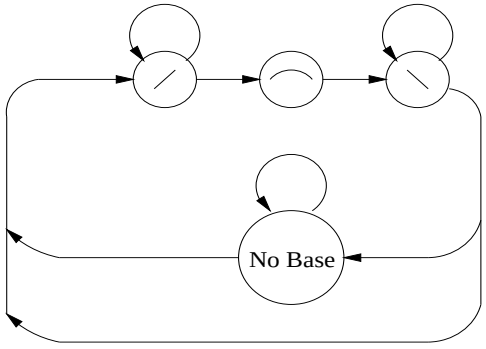

FIGURE 23. A suggestion to eliminate the concurrency problem. Four instances of the above model, one for each base, will be run concurrently on the electropherogram.

explored alternative topologies for our Hidden Markov Models, that can be used to accommodate effects that might cause error in the original model topology. We are now ready to run our models, see the results and draw the final conclusions for the project.

CHAPTER 4

Results, Conclusions, and Future Work

The main focus of this work is the method used. Still, it is important to provide some results to verify that the method works, and plant the seed for further research in the field. In this last chapter we will present the success rate we had on test samples using our models. From that we will conclude that the method is indeed promising, and we will give some directions for potential future research in the area.

Results

In order to evaluate the performance of our proposals we need to basecall electropherograms of known sequences. Again, we will resort to the M13mp18 and the PBluescript consensus sequences as benchmarks. It is important, however, not to use the same electropherograms as the ones used for training; the results in that case might be good just because the system overtrained and memorized them.

Having established the sequence to be used in the performance evaluations, we need to devise a strategy. To do so, we need to consider the types of errors that might occur in a basecalling system, and then figure out how to measure

them. For all our comparisons we will use PHRED as a reference, the standard program used by the genome center [9]. We should note that PHRED is heavily tuned and optimized, using a preprocessor tailored to the particular basecalling method. Therefore it has very good recognition rates. Comparisons of any method with PHRED should take that into account.

Error Evaluation

There are several types of error that may occur during basecalling. We will describe three of the here: *undercalls, overcalls,* and *miscalls*. All the other ones (such as inversions) can be described as combinations of these three. Here we will discuss briefly the three types of errors and then give a caution about how these errors should be reported.

Undercalls (Deletions): These occur when a basecaller does not detect a base that should be in the sequence. An example is the deletion of the second A in the sequence ATACCG when the basecaller translates the signal of that sequence to ATCCG instead.

Overcalls (Insertions): As the name suggests, these are the opposite of deletions. The basecaller inserts a base when it does not exist. For example, an insertion error would occur if the translation the electropherogram of the above sequence was ATACACG.

Miscalls (Substitutions): The problem here would be the substitution of one base for another. For example, if the electropherogram of ATACCG was translated to ATATCG, the basecaller would have made a substitution error.

In general these errors should be reported as a function of the read length, since the quality of the electropherogram degrades as the read length increases. For example, suppose a basecaller reads only the first 400 bases off a signal and has a 5% error rate and a second basecaller has the same error rate reading the first 600 bases of the same electropherogram. Since the latter part of the electropherogram is of much lower quality, one would expect the error rate to be significantly higher. Therefore, if only the first 400 bases of the second basecaller were evaluated, we would expect to have a significantly smaller error rate than 5%, a result much better than that of the first basecaller. Since some basecallers try to read electropherograms more aggressively (for

example, an HMM basecaller would be able to produce a reading for a whole electropherogram, even if the implementation decides not to present the last bases), when comparing error rates one should be careful to do so at the same read length.

Evaluation Results

Our initial evaluations showed that the basecaller based on Gaussian Mixture Models did not deliver worthy results, so we focused our attention on the basecaller based on ANNs. In order to evaluate its performance we called 10 different electropherograms of PBluescript, gathered from an ABI 3700 sequencing machine. To preprocess the data for our basecaller we used the preprocessor of the ABI software. We processed the same electropherograms using PHRED (which uses its own preprocessor).

The results were compared to the published PBluescript sequence using CROSS_MATCH [9], a program that implements the Smith-Waterman algorithm to compare text sequences. The implementation is specific to DNA sequences, an the program detects and reports the errors discussed above. The results were then tallied and averaged over the 10 sequences.

For features we used a 33-samples long window of the electropherogram (resulting to a feature vector of size 4x33=132), extending evenly around the current sample, i.e. 16 samples in each direction. The window was always normalized such that the maximum value is 1.

The neural network we used had three hidden sigmoid layers of size 120, 60, and 12 nodes from the input to the output respectively. The output layer was a softmax one. We chose three instead of two hidden layers that is usually the case because the output layer was a softmax function that only has one training parameter.

The results of our experiments are shown in Figure 24. Our basecaller had a significant number of deletion errors. However, PHRED made many more substitution and insertion errors. The total error rate is comparable, despite the fact that PHRED is heavily optimized, while our system is just a quick implementation with MatLAB scripts. We believe this figure is enough evidence for the validity of this proposal.

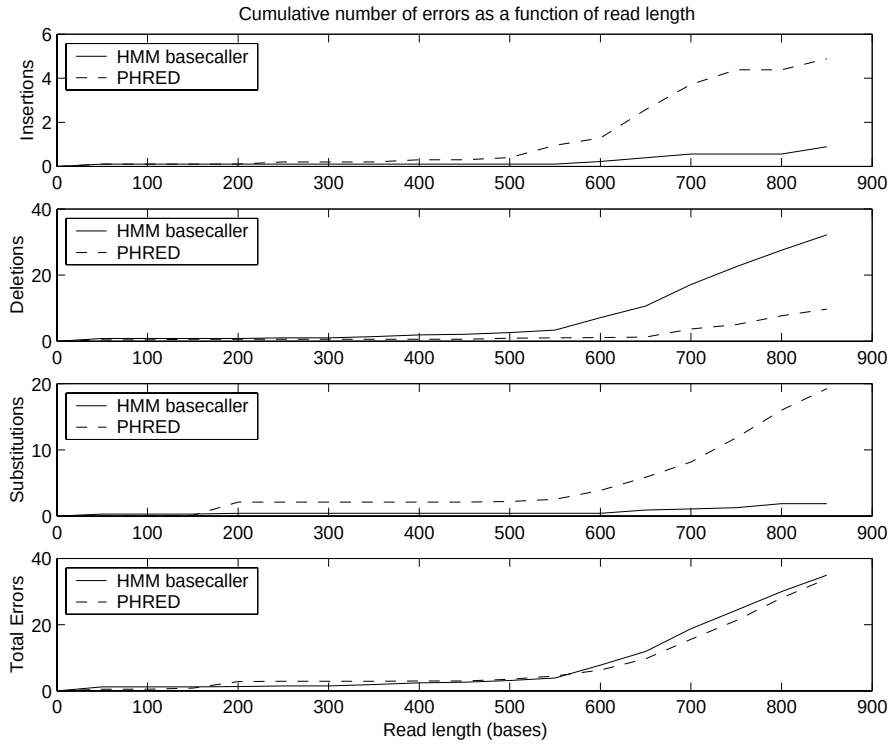

FIGURE 24. The results of comparing PHRED to a basecaller based on HMMs. We can see from the bottom figure that the total errors are comparable.

The number of deletion errors puzzled us, and we decided to look further. Our analysis showed that the largest number of deletion errors occurred in cases of repeated bases. For example, often in a sequence of four or five As, one or two As were dropped by our basecaller. The problem was traced to the difference in the transition statistics of AA compared to, say, AT. As discussed in the previous chapter, we believe that we can increase the accuracy of the basecaller just by implementing the model of Figure 20 on page 70.

What needs to be Done

Several issues have not been addressed in this thesis. Although we believe that our stated goal—to show that HMMs can be used successfully in basecalling—has been achieved, we have by no means exhausted the subject. There is research to be done in sever areas related to this topic. Furthermore, implementations need to address several details.

Preprocessing

One of the assumptions of this thesis was that the preprocessing of the electropherogram could not be affected. However, the preprocessing currently performed has the potential to eliminate useful information from the signal. We believe that most of the preprocessing can be removed, apart from the baseline correction and the mobility shift correction steps. The noise in the data is not significant to merit a low-pass filter that might destroy salient data features, especially when two bases of the same type arrive together and create two peaks that are very close to merging. Furthermore, the color separation step is unnecessary, since we do not base our predictions on the absolute height of each peak but on the statistics of the whole electropherogram. Although it is a linear transformation, and can always be inverted, it might pronounce some features that confuse the later steps of basecalling. In any case if these steps are necessary, the trained neural network should train to perform them internally, in a much better way than the rather heuristic approach currently used.

HMM Topologies

Another issue that deserves further research scrutiny is the HMM topology that delivers the best basecalling results. For example, apart from the simple three state per base model we have implemented, we have also proposed other models which we have not tested (see “Alternative Topologies” on page 68). These are motivated by the physical properties of the system, and have some potential to improve the accuracy. Furthermore, other topologies which we have not considered might provide better results.

Features and Emission models selection

The features we used for the Gaussian Mixture models were overly simplistic. On the other hand, the brute force approach we took with the neural network improved the results, at the expense of computation. By carefully selecting the feature set we could probably achieve a computational middle ground, with even better results. The computation complexity might be also be improved by using parametric models other than Gaussian Mixtures that are better at describing the data.

Extensions

One of the most intriguing aspect of this work, which we had no time to touch on, is the extension of these models to other biological applications. Specifically, *single nucleotide polymorphism* (SNP) detection involves analyzing similar electropherograms, looking for different things. Furthermore, DNA fingerprinting for forensic or other reasons can also exploit HMMs. Finally, recent developments in the field of *proteomics* require protein sequencing, a task performed essentially in the same way as DNA sequencing. This field too can exploit this work.

Conclusions

After formulating the DNA basecalling problem as a statistical pattern recognition one, we have noticed that it is strikingly similar to a subset of the speech recognition problems. Therefore, we exploited the extensive research in the field, specifically in the area of Hidden Markov Models. Our desire to use artificial neural networks for density estimation lead us to develop an embedded training method that treats the HMM and the ANN together, instead of training them separately and then combining them. This resulted to a system that exhibited a good performance compared with existing ones.

Furthermore, we developed a method to perform queries in a long sequence, even using only partially trained models. Even though we only used this method to generate training data, it can have a large variety of applications.

We believe that the field has by no means been exhausted. Alternative HMM topologies might have a significant impact in the quality of the results. Also, better feature selection might eliminate the need for ANNs as the emission model, which will significantly decrease training time and model complexity. Better signal preprocessing, tailored to an HMM basecaller, might improve the results further. Finally, there is a potential for these models to be applicable to other biological applications such as mutation detection, forensic analysis, and proteomic sequencing.

As the field of Biology uses more and more information processing as a tool, the importance of statistical models will increase rapidly. We believe that this work is a small step towards that direction.

References

[1] Bengio Y, De Mori R, Flammia G, Kompe R. *Global Optimization of a Neural Network-Hidden Markov Model Hybrid*. **IEEE Trans. on Neural Networks**. 1992 Mar; 3(2); 252-259.

[2] Berno AJ. *A graph theoretic approach to the analysis of DNA sequencing data*. **Genome Res**. 1996 Feb;6(2):80-91.

[3] Bowling JM, Bruner KL, Cmarik JL, Tibbetts C. *Neighboring nucleotide interactions during DNA sequencing gel electrophoresis*. **Nucleic Acids Res**. 1991 Jun 11;19(11):3089-97.

[4] Cohen M, Franco H, Morgan N, Rumelhart D, Abrash V, Konig Y. *Combining Neural Networks and Hidden Markov Models*. **Proceedings of the DARPA Speech and Natural Language Workshop** 1992, Harriman, NY.

[5] Duda RO, Hart PE. *Pattern Classification and Scene analysis.* 1973 John Willey& Sons, Inc. USA.

[6] Encyclopædia Britannica Online. *DNA*. <http://search.eb.com/bol/topic?eu=31232&sctn=1> [Accessed Feb 2002].

[7] Encyclopædia Britannica Online. *polymerase chain reaction*. <http://search.eb.com/bol/topic?eu=2536&sctn=1> [Accessed Feb 2002].

[8] Ephraim Y, Rabiner LR. *On the relations between modeling approaches for information sources*. **IEEE Transactions on Information Theory**, 36(2):372--380, March 1990.

[9] Ewing B, Hillier L, Wendl MC, Green P. *Base-calling of automated sequencer traces using phred. I. Accuracy assessment*. **Genome Res**. 1998 Mar;8(3):175-85.

[10] Ewing B, Green P. *Base-calling of automated sequencer traces using phred. II. Error probabilities*. **Genome Res**. 1998 Mar;8(3):186-94.

[11] Gallager R. *Discrete Stochastic Processes.* 1996 Kluwer Academic Publishers, Norwell, MA.

[12] Giddings MC, Brumley RL Jr, Haker M, Smith LM. *An adaptive, object oriented strategy for base calling in DNA sequence analysis*. **Nucleic Acids Res**. 1993 Sep 25;21(19):4530-40.

[13] Giddings MC, Severin J, Westphall M, Wu J, Smith LM. *A software system for data analysis in automated DNA sequencing*. **Genome Res.** 1998 Jun;8(6):644-65.

[14] Haykin S. *Neural Networks, A comprehensive Foundation, 2nd ed*. 1999 Prentice Hall, Upper Saddle River, NJ.

[15] Lawrence CB, Solovyev VV. *Assignment of position-specific error probability to primary DNA sequence data*. **Nucleic Acids Res**. 1994 Apr 11;22(7):1272-80.

[16] Lipshutz RJ, Taverner F, Hennessy K, Hartzell G, Davis R. *DNA sequence confidence estimation*. **Genomics**. 1994 Feb;19(3):417-24.

[17] Minka TP. *Expectation-Maximization as lower bound maximization*, Nov. 98; revised Nov 99; http://www-white.media.mit.edu/~tpminka/papers/em.html [Accessed Jan 2002].

[18] Nelson D, *Improving DNA Sequencing Accuracy And Throughput*. **Genetic mapping and DNA sequencing**. New York: Springer, c1996; 183-206

[19] Papoulis A. *Probability, Random Variables, and Stochastic Processes, 3rd ed.* 1991 WCB/McGraw-Hill, USA.

[20] Rabiner LR, *A tutorial on hidden Markov models and selected applications in speech recognition*. **Proceedings of the IEEE**, 1989 Feb.;77(2):257-286

[21] Short JM, Fernandez JM, Sorge JA, Huse WD. *Lambda ZAP: a bacteriophage lambda expression vector with in vivo excision properties*. **Nucleic Acids Res**. 1988 Aug 11;16(15):7583-600.

[22] Yanisch-Perron C, Vieira J, Messing J. *Improved M13 phage cloning vectors and host strains: nucleotide sequences of the M13mp18 and pUC19 vectors.* **Gene** 1985;33(1):103-19