EXTENDENDO MODELOS MARKOVIANOS

USANDO DESCIDA DE GRADIENTE

ANDRE LLOYD DWIGHT PERLEE HARDER

EXTENDENDO MODELOS MARKOVIANOS USANDO DESCIDA DE GRADIENTE

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Adriano Alonso Veloso

Belo Horizonte

Março de 2017

ANDRE LLOYD DWIGHT PERLEE HARDER

EXTENDING MARKOV MODELS THROUGH GRADIENT DESCENT OPTIMIZATION

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Adriano Alonso Veloso

Belo Horizonte

March 2017

Harder, Andre Lloyd Dwight Perlee

D1234p Extending Markov Models through Gradient Descent Optimization / Andre Lloyd Dwight Perlee Harder. — Belo Horizonte, 2017
xxiv, 108 f. : il. ; 29cm

Dissertação (mestrado) — Federal University of Minas Gerais
Orientador: Adriano Alonso Veloso

1. Machine Learning. 2. Markov Models. 3. Neural Networks. I. Título.

CDU 519.6*82.10

[Folha de Aprovação]

Quando a secretaria do Curso fornecer esta folha,
ela deve ser digitalizada e armazenada no disco em formato gráfico.

Se você estiver usando o `pdflatex`,
armazene o arquivo preferencialmente em formato PNG
(o formato JPEG é pior neste caso).

Se você estiver usando o `latex` (não o `pdflatex`),
terá que converter o arquivo gráfico para o formato EPS.

Em seguida, acrescente a opção `approval={`*nome do arquivo*`}`
ao comando `\ppgccufmg`.

Se a imagem da folha de aprovação precisar ser ajustada, use:
`approval=[`*ajuste*`][`*escala*`]{`*nome do arquivo*`}`
onde *ajuste* é uma distância para deslocar a imagem para baixo
e *escala* é um fator de escala para a imagem. Por exemplo:
`approval=[-2cm][0.9]{`*nome do arquivo*`}`
desloca a imagem 2cm para cima e a escala em 90%.

Dedicuum cest laborae a quelquis personatum que ajudorat a facirelo.

Acknowledgments

Agradeço aos prótons por serem tão positivos, aos nêutrons pela sua neutralidade e aos elétrons pela sua carga.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur? Quis autem vel eum iure reprehenderit qui in ea voluptate velit esse quam nihil molestiae consequatur, vel illum qui dolorem eum fugiat quo voluptas nulla pariatur?

At vero eos et accusamus et iusto odio dignissimos ducimus qui blanditiis praesentium voluptatum deleniti atque corrupti quos dolores et quas molestias excepturi sint occaecati cupiditate non provident, similique sunt in culpa qui officia deserunt mollitia animi, id est laborum et dolorum fuga. Et harum quidem rerum facilis est et expedita distinctio. Nam libero tempore, cum soluta nobis est eligendi optio cumque nihil impedit quo minus id quod maxime placeat facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet ut et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum hic tenetur a sapiente delectus, ut aut reiciendis voluptatibus maiores alias consequatur aut perferendis doloribus asperiores repellat.

"Truth and lie are opposite things."

(Unknown)

Resumo

O estudo de sequências de eventos e séries temporais se encontra no cerne do pensamento científico, tal que a meta principal de uma grande parcela de nossas empreitadas intelectuais se sumariza em prever, explicar, ou modelar esse tipo de fenômeno, que se instancia no mundo por via de series de, por exemplo, eventos climáticos, linguagem, observações empíricas, e ações motoras. Um grupo de métodos que tem sido exitoso em modelar séries e sequências são os Modelos Markovianos de tempo e estado discreto. Estes modelos oferecem um ferramental variado, podendo ser empregados em tarefas como predição, explanação e simulação, às custas de uma modelagem mais simplificada do mundo. O uso destes modelos, no entanto, vem acompanhado de dificuldades em adequar-los a contextos específicos: A criação de estados semanticamente úteis, e novas estratégias para expressar a função de transição tradicionalmente requerem alterações complexas sobre otimizadores como o algoritmo de Baum Welch.

Neste trabalho, hipotetizamos que a otimização por via de Descida de Gradiente é um substituto eficaz para formas mais tradicionais de se otimizar Cadeias de Markov discretas. Ademais, acreditamos que o uso de tais técnicas permitirá uma maior flexibilidade ao usuário na definição da semântica dos estados e do modelo de transição. Para esse fim, desenvolvemos uma estratégia para obter otimizadores de descida de gradiente para essa classe de modelos, que validamos por via de comparações objetivas e subjetivas com a saída obtida pelo algoritmo Baum-Welch. Em seguida, propomos dois modelos práticos que objetivam demonstrar a flexibilidade do novo otimizador. O primeiro utiliza um modelo hibrido de Rede Convolucional/Modelo Markoviano na classificação de dígitos do MNIST, enquanto o segundo ilustra um modelo de transição mais complexo, hierárquico, que trata dados multi-dimensionais e multi-escala, o qual validamos sobre histogramas de temperatura e da operação de um servidor. Nossos resultados mostram que descida de gradiente é um método viável para otimizar Modelos Markovianos Discretos, e que o modelo resultante é altamente flexível.

Palavras-chave: Modelos Markovianos, Descida de Gradiente, Aprendizado de Máquina.

Abstract

The study of sequences of events and time series has been at the heart of scientific reasoning since its conception, as ultimately the goal of many of our intellectual endeavors is to predict, explain or model these phenomenon, which can be as diverse as a series of weather patterns, language, empirical observations or motor actions. One group of methods that have been successfully employed in modeling series of events are discrete-state and time Markov Models. These models offer a wide variety of tools for tasks such as prediction, explanation and simulation, at the cost of a more simplified model of the world. A significant downside of using this class of models, however, can be the effort required to adapt them to a desired context: The development of useful state semantics and novel strategies for expressing the transition model traditionally requires unreasonably complex adaptations to optimizers such as the Baum Welch algorithm.

In our work, we posit that Gradient Descent optimization can be used as an effective substitute for more traditional forms of optimizing discrete-time and state Markov Models. Furthermore, we believe that doing so affords users a greater flexibility in defining state semantics and transition models. To this end, we devise a strategy for mechanically obtaining a Gradient Descent Optimizer for discrete-time and state Markov Models, which we then scrutinize by comparing it objectively and subjectively to models obtained through the Baum-Welch algorithm. Next, we propose two practical models which aim to demonstrate the flexibility attained through the use of a Gradient Descent Optimizer. The first uses a hybrid Convolutional Neural Network/Markov Model classifier to achieve digit classification on MNIST, while the second illustrates a more complex, hierarchical transition model aimed at multidimensional and multi-timescale data, which we validate on a temperature histogram and on values from the operation of a server. Our results show that Gradient Descent is indeed a viable method for optimizing discrete-state and time Markov Models, and that the resulting models are highly flexible.

Palavras-chave: Markov Models, Gradient Descent, Machine Learning.

List of Figures

2.1 Example 3-state Markov Model 10
2.2 Example Hidden Markov Model 11
2.3 Bigram output example 13
2.4 Information flow graph for the Perceptron 26
2.5 Information flow graph for the Multilayer Perceptron 29

3.1 Weekday traffic example 45
3.2 Simple weekday traffic Markov Chain 45
3.3 Simple weekday traffic Markov Chain simulation outputs 45
3.4 Ring Markov Chain for traffic modeling 46
3.5 Weekend traffic example 47

4.1 *Simple Markov Chain* example 59
4.2 *Ambiguous Markov Chain* example 59
4.3 *Simple HMM* example 61
4.4 Example Ring Markov Model 61
4.5 *Simple Markov Chain* GD/BW likelihood results 63
4.6 *Ambiguous Markov Chain* GD/BW likelihood results 63
4.7 *Simple HMM* GD/BW likelihood results 63
4.8 *Large Random HMM* GD/BW likelihood results 64
4.9 *Chain HMM* GD/BW likelihood results 64
4.10 Local minima example 65
4.11 Examples from MNIST dataset 67
4.12 Visual Markov Model likelihood $\times$ iteration 69
4.13 Visual Markov Model likelihood comparison 69
4.14 Temperature histogram dataset example 72
4.15 CPU and RAM histogram dataset example 73
4.16 Multilayer Perceptron implemented in our Domain Specific Language . . . 76

4.17 Hidden Markov Model implemented in our Domain Specific Language . . . 77
4.18 Log likelihood results on the temperature data 80
4.19 Log likelihood results on the CPU data 80
4.20 Log likelihood results on the RAM data 81
4.21 Temperature distributions predicted by the SMM 82
4.22 Temperature distributions predicted by the HMM 84
4.23 Multi-series likelihood comparison 85

List of Tables

4.1 *Large Random HMM* example 60
4.2 Optimizer parameters used in training our Gradient Descent/Baum Welch comparison tests 62
4.3 SMM experimental parameters 74

Contents

Acknowledgments **xi**

Resumo **xv**

Abstract **xvii**

List of Figures **xix**

List of Tables **xxi**

1 Introduction **1**

1.1 Statement 3

1.2 Objective 4

1.3 Specific objectives 4

1.4 Challenges and contributions 5

1.5 Structure 6

2 Background **7**

2.1 Markov Models 7

2.1.1 Mathematical Formulation of the Markov Chain 8

2.1.2 Mathematical Formulation of the Hidden Markov Model 10

2.1.3 Predicting the future 12

2.1.4 Simulating new data 13

2.1.5 External data 13

2.1.6 The Baum Welch algorithm 18

2.2 Gradient Descent Optimization 20

2.2.1 Artificial Neural Networks 24

2.2.2 Automatic differentiation 30

3 Extending Markov Models Through Gradient Descent **33**

3.1 Gradient Descent Formulation 34
3.2 Proposed Models 41
3.2.1 Co-Optimized Visual Markov Model 41
3.2.2 Stacked Markov Model 44

4 Experiments **57**
4.1 Effectiveness of Gradient Descent Optimization on Markov Models . . . 57
4.1.1 Setup 58
4.1.2 Results 62
4.2 Digit Classification 66
4.2.1 Setup 67
4.2.2 Results 68
4.3 Stacked Markov Model 70
4.3.1 Setup 71
4.3.2 Results 79

5 Conclusions and Future Work **87**
5.1 Future Work 88

Bibliography **91**

Appendix A SMM Formulation **95**

Chapter 1

Introduction

The study of causality and, more generally, sequences of events, has arguably been at the heart of scientific reasoning and understanding since their very beginnings. Physics, for example, aims to model the natural world through the lens of mathematics in such a way that a repeatable sequence of actions will yield a consistent set of consequences; On the other side of the spectrum, Psychology and Sociology attempt to do the same in the fields of human life and society, by formulating models that, given some sequence of events, result in a particular set of observations. The crucial role which causal sequences have played in our understanding of the world is far from arbitrary: From a utilitarian perspective, understanding the past and present only serve a purpose in so much as doing so enables us to shape our actions in the future, and thus, to the extent that knowledge of the future is achievable and useful, the only way to go about attaining it is by constructing these sequences of events, and coupling known past occurrences to possible future outcomes.

It is impossible, however, to reason about sequences without also addressing the elements which comprise them. Newton's laws of motion, for instance, while effective in establishing a sequence of cause and effect with an infinitely large range of applications, rely on less sequence-oriented concepts such as mass, velocity, acceleration and energy; Similarly, the kind of behavioral relationships modeled in Pavlovian conditioning also requires base-concept definitions, albeit of a more complex nature, such as what constitutes a conditioning stimuli, and what qualifies as conditioned behavior. While in many cases these elements can be defined in a mostly objective way – such as the concept of mass – it is often the case that a more subjective and intuitive approach must be taken – such as the concept of "conditioned behavior" in psychology, or "severe structural damage" in civil engineering.

Historically, in order to conceptualize these more subjective elements, we have relied on our innate processing abilities and intuition, however, this paradigm has been shifting over the past few decades. Factors such as the exponential growth of computing power, the availability of massive amounts of free data, and a renaissance in the understanding of algorithms have lead to both a demand and solutions for machine-based pattern recognition applications to problems which are subjective in nature. Instances of these kinds of systems have become ubiquitous. For example, it was estimated that, in the third quarter of 2014, spam accounted for an average of 68% of all email messages (a total of 56 billion messages per day)[Kohavi, 2014], and, on a daily basis, over 3.5 billion Google searches continue to be made, with most users receiving acceptable results, selected from a database of over 30 trillion pages[gst, 2016]. In both of these cases, even though the task is largely a subjective one – who, after all, can say unambiguously what is and isn't spam, or what is a good result for a search query – users see satisfactory results operating on scales 'which would be impossible to achieve manually through human labor.

In these instances and others, the identification of base elements is sufficient for completing the task, but this is not always the case. In particular, contexts where the base elements may be ambiguously defined, deriving part of their meaning from their relationship with the sequence, pose a significant challenge. Consider, for instance, the task of visually tracking a person through video: At some angles, the person may be clearly identifiable (perhaps their face is showing), at others, it may be hard to distinguish between them and someone else (the person might be turned around, for example). In cases such as this, knowledge may be spread out over time, requiring some form of data-integration in order to establish the desired outcome[Saxena et al., 2008] (in the given example, this might be the challenge of establishing reasonable patterns of motion for a person – hardly a simple task in its own right – and then joining the resulting probabilities with those extracted from each position of each frame in order to come up with a reasonable guess as to where the person is in at each moment in time).

The approaches to solving the problem of modeling sequences of non-trivial observations, such as the above, can be generally grouped into two camps, which we will refer to as two-stage and single-stage models. In two-stage models, one attempts to model the identification of the basic elements through some kind of supervised classifier, and then separately devise a sequence model – often probabilistic in nature. There are many advantages to this kind of approach, for instance, it allows the task to be broken down into sub-tasks (namely the derivation of the observation and sequence models), which can then also make use of expert knowledge when being developed (for example, an expert in facial recognition might apply his skills as needed in doing the facial recognition portion of a person-tracking system, whereas a computer graphics professional might be able to use his skills to develop a model for movement).

These same advantages can however become limitations, as expert knowledge may be imprecise, and a lack of cross-discipline understanding and integration may lead to severe performance penalties at the point where the models are joined. These difficulties lead to the second set of approaches, single-stage models, which fully reject the separation of these two components. In these models, such as Recurrent Neural Networks (RNNs)[Pearlmutter, 1995], both the local and temporal aspects of the data are treated homogeneously, which, while effective in eliminating performance losses across domains, often comes at the cost of interpretability and allowing the inclusion of insight from experts, for example. In spite of this, methods in this group have gained much ground in the last decade, but they still are far from fully replacing the more traditional two-stage systems in use to this day.

In light of these two fundamentally different approaches to sequence modeling, each with different strengths and weaknesses, the question which arises – and which motivates this thesis – is whether there is a way to bridge the gap between two- and single-stage models. Objectively, what this thesis attempts to do is to demonstrate that one can design two-stage sequence models, but then use optimization techniques from single-stage models to eliminate the performance losses from having independent modules, in particular, this is demonstrated for Markov Models, one of the more popular sequence modeling techniques used in the temporal portion of two-stage models, and Gradient Descent-optimized models, such as Neural Networks, which have been shown to be effective in a wide range of classificatory applications.

1.1 Statement

Many sequence-based machine-learning models are defined as two-part systems, wherein first sequence elements are identified, and then, separately, the sequential nature of the data is addressed. We posit that the segregation of models into these two parts can lead to performance losses, but that these losses can be minimized by joining them in a single optimization process. In this dissertation, we derive this joint-optimization process for the broad class of Markov-Model-based sequence optimizers, and explore a few of its ramifications for this class of models.

1.2 Objective

The goal of this work is to demonstrate that the utility and effectiveness of Markov Model-based sequence analysis methods can be improved through the use of Gradient Descent optimization. To this end, we derive and implement the Gradient Descent optimizer for Markov Models, which in turn allows for much richer model designs, and for its use and conjoint optimization with other Gradient Descent-optimizable methods, such as Neural Networks and linear models.

With this tool in hand, we aim to validate the hypothesis that Markov Models can retain many of their desirable properties, such as interpretability, use as a predictive method, simulation and data denoising, while simultaneously making use of/being used in Gradient Descent-based methods, such as Neural Networks and linear models. We posit that the classificatory power of these latter methods can be conjoint with the usefulness of the former in a more effective and broader way by conjointly optimizing these components.

1.3 Specific objectives

Less generally, the specific objectives we pursued in this work are as follows:

- To study and understand concepts related to parameter optimization, specifically, using the methods of Gradient Descent optimization and the Baum Welch algorithm, as well as come to an understanding of other relevant forms of optimization, and what their strengths and weaknesses are with respect to our area of interest;

- To validate the effectiveness of Gradient Descent Optimization as a competitive method for optimizing discrete Markov Models, exploring what advantages and/or disadvantages this technique has over more traditional Markov Model optimizers;

- To study and improve upon traditional techniques of automatic differentiation, with the goal of facilitating model design and prototyping;

- To illustrate the flexibility Gradient Descent Optimization affords to the development of Markov Models by designing a set of novel Markov Models which perform interesting tasks, and then quantify and compare their performance with those of more traditional methods.

1.4 Challenges and contributions

The challenges and contributions of this work are as follows:

- **A Gradient Descent model for Markov Models:** One of the principal challenges of this project was in developing a way of expressing a Markov Model's parameters in such a way that Gradient Descent Optimization would be effective on it. Many parameter formulations, objective functions and gradient manipulations were attempted before coming up with with the relatively simple model presented here. These difficulties were initially compounded by the having to manually recalculate and then re-implement all the gradients, though this barrier was eventually resolved through the use of automatic differentiation.

- **A flexible, yet efficient method for automatic differentiation:** We aimed in this thesis to make the design process for Markov Models flexible. This in turn required a method for easily expressing novel designs that also eliminated the significant effort of constructing its optimizer. While the available automatic differentiation libraries are exceedingly efficient, they require users to reformulate their thought process to work with tensors, making the already difficult process of model conception even harder. With this in mind, we invested a significant portion of our efforts into designing an imperative programming language which would be more natural for its users, while still producing a reasonably efficient optimizer. The resulting language supports full optimization for complex features such as recursion, for loops, if conditionals, methods and classes, multiple assignments to the same variable, C++ inlining, as well as other features. We also implemented Hessian-Free optimization in the compiler, which, as far as we know, is the first time this has been implemented in a fully automatic system. Sadly, the convexity constraints of this form of second order optimization made it impractical for general use.

- **Adding contextual knowledge to Markov Models:** One of the main drawbacks of using Markov Models is their lack of memory (as all context information must be inferred from the model being in some specific state). We devised a way of dealing with this without fundamentally changing how these models work through the use of multiple Markov Chains operating at different time scales. Developing this model was difficult, however, as obtaining an actionable likelihood expression which expressed our desired structure was not trivial.

- **Allowing Markov Models to utilize visual data directly:** One of the contributions of our work is in the development of the Visual Markov Model, which uses a Convolutional Neural Network[LeCun et al., 1998a] to do the bulk of the image analysis, before moving onto the sequence operations inherent to Markov Models. Dealing with visual data has been a challenge for Markov Models, as manually devising visual feature detectors is difficult and time consuming. Here, however, we were able to construct the visual features automatically, by conjointly optimizing the Markov Model and the Convolutional Network at its base.

1.5 Structure

The remainder of this dissertation is structured as follows: Chapter 2 details the concepts and related works needed to understand and contextualize our thesis, namely those of Markov Models and Gradient Descent; Chapter 3 details our method for optimizing Markov Models, and proposes two novel forms of Markov Models whose aim is to illustrate the flexibility of the proposed mode of optimization; Chapter 4 details the experiments we conducted, as well as their results; And finally, Chapter 5 concludes this dissertation, reviewing what was done as well as detailing several avenues for future research.

Chapter 2

Background

2.1 Markov Models

Several common methods for modeling sequences and time series fall under the category of *Markov Models*Murphy [2012]. These methods are often employed due to their ease of understanding, interpretability, parametrization from data, and versatility. While an exceedingly large diversity exists within these kinds of models, several key principles are generally shared between them, which is what allows us to generalize our results over the larger set of models used in the literature. Broadly speaking, this class of models can be categorized as discrete-state and -time probabilistic time-series models which assume the validity of the Markov property on the system which they model, a definition which we will now expand on:

In referring to probabilistic discrete-state models, we refer to methods which, at any given time, represent the state of the environment being modeled as some distribution of probabilities over a countable set of possible contexts, or states. While many real-world environments have an uncountably large set of possible states (caused, for instance, by the presence of real-valued variables), in practice, slicing the space into discrete configurations can often be sufficiently precise, especially when accompanied by the use of probability distributions, which, by spreading out probabilities over many similar states, can at times counter the approximation errors incurred by discretizing the environment.

The manner in which the probabilistic distribution over these states evolves, however, has yet to be addressed, and it is at this point that both discrete-time and the Markov property from our definition come into play. The Markov property[Murphy, 2012], named after the eponymous mathematician Andrey Markov, who studied the field in the mid-20th century, simply states that the probability distribution over future states can be fully determined using only knowledge of the present state distribution, not requiring any knowledge of the distribution at any point before it, or even knowing when precisely the present is. If we also accept as a premise that time can be reduced to a countable sequence of moments, then it follows that if we can establish rules for transitioning between the distribution at time t and the distribution at time $t+1$, these rules will be applicable at any time $t \in T$, and can thus be used to predict the distribution arbitrarily far into the future, as well as having other useful properties, examples of which are addressed in Sections 2.1.3, 2.1.4 and 2.1.5.

Finally, putting everything together, that is, joining the assumptions of discrete-state and -time modelability with the Markov property, we come to a more comprehensive classification of the Markov Models which are dealt with in this work: These methods first assume that their environments can be discretized, both in time and semantically into states, without significant degradation of performance. Then, these methods overlay a probability distribution over these states, and attempt to model the way that these state probability distributions evolve from one moment to the next through some set of rules. Examples of models which typically fall under this category are Markov Chains (MC), Hidden Markov Models (HMM)[Murphy, 2012] and Hierarchical Markov Models[Fine et al., 1998]. Other examples for which the results of this thesis remain accurate, but which take only a subset of these properties are Latent Dirichlet Allocation[Blei et al., 2003] and n-gram models[Brown et al., 1992]. Many other examples exist, and a few will be discussed in more detail later in this thesis.

In this section we will begin by deriving the mathematical formulation behind two of the more popular forms of Markov Models, the Markov Chain and the Hidden Markov Model (Sections 2.1.1 and 2.1.2). Following this, we will demonstrate three of the many useful operations which can be done in this kind of model: Prediction (Section 2.1.3), Simulation (Section 2.1.4), and External data analysis (Section 2.1.5). Finally, we conclude with an overview of the Baum Welch algorithm for obtaining HMM transition matrices from observed data (Section 2.1.6).

2.1.1 Mathematical Formulation of the Markov Chain

In its most basic form, a Markov Chain can be defined by a finite set of states $s_i \in S$, and a set of initial- and transition-probabilities over these states, which describe how the probabilities of these states start, and then change from one moment to the next. Mathematically, we introduce the following notation:

$$\begin{aligned} s_i &: \text{The state } i \text{ of the model world} \\ s_i \in S &: \text{All the states in the model world} \\ s_i^{(t)} &: \text{The world at time } t \text{ is in state } i \text{ (an event)} \end{aligned} \tag{2.1}$$

By construction, we define that two of these events cannot be true simultaneously, as the world can only be in one state at a time. This is to say that:

$$\begin{aligned} P[s_i^{(t)}] &: \text{The probability of being in state } i \text{ at time } t \\ P[S^{(t)}] &= \begin{cases} P[s_i^{(t)}] & \text{if } s_i^{(t)} \wedge \neg s_j^{(t)} \forall j \neq i \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{2.2}$$

We can then define transition probabilities $P[s_j^{(t+1)}|s_i^{(t)}]$ on top of these events, representing the probability of the world evolving from state i to j in one time step. This allows us to define the probability of being in a particular state in the future given only the present state distribution.

$$\begin{aligned} P[s_i^{(t+1)}|s_j^{(t)}] &: \text{Transition probability (defined by the model)} \\ P[s_i^{(t+1)}] &= \sum_{\forall S^{(t)}} P[s_i^{(t+1)}|S^{(t)}]P[S^{(t)}] \\ &= \sum_{j \in |S|} P[s_i^{(t+1)}|s_j^{(t)}]P[s_j^{(t)}] \\ P[s_i^{(0)}] &: \text{Initial state-probability distribution (defined by the model)} \end{aligned} \tag{2.3}$$

Note that, in the above, we use the definition from Equation 2.2 to eliminate all $S^{(t)}$ where more than one $s_i^{(t)}$ is active, as these have probability 0. Note also that, if the number of states in the Markov Model is finite, we can use Equation 2.3 to iteratively compute $S^{(t+1)}$ in its entirety, which can in turn then be used to calculate $S^{(t+2)}$, and so on (more details on this in Section 2.1.3). Graphically, this model is often represented in a manner such as seen and explained in Figure 2.1.

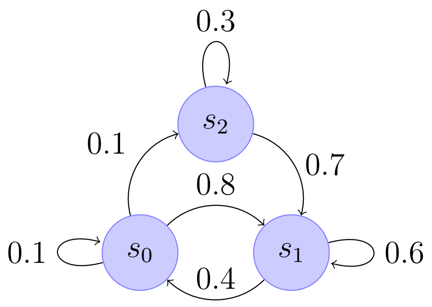


Figure 2.1. A Markov Chain with 3 states, represented as nodes, and transition probabilities $P[s_i^{(t+1)}|s_j^{(t)}]$, represented as weighted directed edges from i to j. At each moment, a single state is active; Which state will be active in the next time step depends entirely on which state is currently active, and is uniformly sampled from its transition probabilities.

2.1.2 Mathematical Formulation of the Hidden Markov Model

The use of purely Markov Chain approaches, such as delimited in Section 2.1.1, is associated with difficulties such as how to divide the world into a finite set of states, and then how to determine which state the world is in at a given moment. This sort of problem is particularly common when treating real-world data, be that a sequence of words taken from a sentence, a series of stock values (or any other kind of data where the true state of the world might be obscured at times by interfering variables), or where there may be no clear designation on how the world should be organized into states.

The solution proposed by Hidden Markov Models (HMM) is to assume that, while behind the scenes the world still operates on a Markov Chain whose states are "hidden", or *latent*, we can observe evidence of the hidden state of the world through imprecise observations. In essence, what the HMM proposes is that what we can observe are only the results of the world being in these particular states, which not only may be variable so as to account for noise, but may also be real-valued and ambiguous, allowing for us to organize the internal states of the model more freely.

Specifically, the Hidden Markov Model (HMM) attempts to model these latent states by separating the observed data from the model's states, associating each new input $x^{(t)} \in X$ with each state s_i through a conditional probability. Philosophically, what the Hidden Markov Model assumes is that, when the world is in state s_i, some event occurs which will stochastically sample some new observation $x^{(t)}$ from the set of all possible observations U with some distribution assigned to s_i. Thus, the observations seen in the world depend solely on the hidden state of the world at that present moment – another assumption of the Markov property – a fact which can also allow us to infer which hidden state the model is in based on real world data (see Section 2.1.5).

Mathematically, the HMM expands on the Markov Chain equations (Equation 2.4) with the addition of the observation conditional probability:

$$\begin{aligned}
x \in U &: \text{One possible observation within the set of all possible observations} \\
x^{(t)} \in X &: \text{The observation at time } t \text{ within the set of all observed values} \\
P[x|s_i] &: \text{Probability of observing } x \text{ in state } s_i \\
P[x^{(t)}|s_i^{(t)}] &= P[x^{(t)}|s_i] \ \forall\ t \\
P[x^{(t)}] &= \sum_{s_i \in S} P[x^{(t)}|s_i^{(t)}] P[s_i^{(t)}]
\end{aligned} \tag{2.4}$$

Graphically, Figure 2.2 exemplifies a customary representation of a Hidden Markov Model.

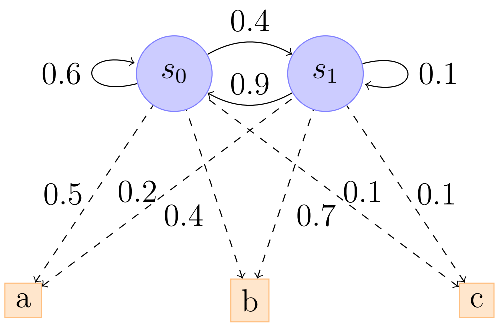


Figure 2.2. A Hidden Markov Model with two hidden states s_0 and s_1 representing the world, and a, b and c, the three possible observations of this hidden world. As is the case with the Markov Chain, the current state of the model evolves by picking one of the states at each time step, based on the previous state and its transition probabilities. Different from a Markov Chain, however, the state of the model is not assumed to be observable. Instead, we observe a, b or c, and assert that the probability of seeing them when in each state is given by the dotted edges (e.g.: the probability of observing b when in state s_0 is 0.4). From this we can try and infer the true state of the model (for example, observing b is a good indication that the model might be in state s_1.

2.1.3 Predicting the future

Motivated by our general desire for anticipating the future, many applications for sequence models involve prediction. This is a difficult task in nearly any real-world scenario, however, as there are often many unexpected variables which, when put together, can change the observable world in significant and unpredictable ways.

One class of methods which are particularly prone to this sort of unpredictability are next-element classification algorithms – In these cases, a model is built which, given some portion of a series, predicts what the next element of that series will be. While effective in predicting a single time step in the future, reusing this predicted element as an input for predicting the next element ignores scenarios where more than one outcome is possible at a given moment, as well as allowing for inaccuracies in the model to become compounded at every step, with no indication that this is occurring.

Markov Models, on account of their probabilistic representation of the world, offer a uniquely effective way of solving this problem. Put simply, Markov Models avoid the issues associated with making a single prediction by maintaining an entire distribution over the state-space, thus taking into account variability in possible outcomes, and minimizing the problems associated with compounding predictive mistakes by allowing inaccuracies to be expressed as an increased vagueness in the predicted state probability distribution.

In order to compute a Markov Model's prediction, all that is needed is to apply Equation 2.3 iteratively. For instance, assuming that we have a distribution $\{P[s_i^{(t)}]\ \forall s_i \in S\}$, we can use this equation to compute $\{P[s_i^{(t+1)}]\ \forall s_i \in S\}$; then, to obtain $\{P[s_i^{(t+2)}]\ \forall s_i \in S\}$, we can reapply the same equation, only this time taking the previous distribution, $\{P[s_i^{(t+1)}] \forall s_i \in S]\}$, as input (recalling that we only need the previous distribution since we assume the Markov property).

Using this method, one can predict arbitrarily far into the future. Over time, however, the uncertainties associated with ambiguous state transitions – that is, when a single state can transition to two or more other states with non-zero probability – build up, rendering less punctual distributions. This is to say that very certain predictions (such as assigning a high probability for one particular state at a given moment) will often become diluted after a larger number of iterations, reflecting the overall uncertainty inherent in compounding the inaccuracies of multiple predictions[1].

Other algorithms exist that can predict the future state-probability distributions in Markov Chains more efficiently, as well as others which can calculate the distribution

[1]The opposite effect can more rarely be observed if the system is convergent in nature. In cases such as these, the further in the future one goes, the more likely it is for the system to be in a particular state.

```
And Metheg-Ammah took and travel unto her beautiful land.
By of one curse in marvelous riches is lost to riches.
What shall findeth given mine blade?
```

Figure 2.3. A few sentences generated using a bigram model, a particular sub-case of Markov Models, can be seen above. While somewhat sensible, particularly when observing words pair by pair, the finer points of syntax are lost on account of textual data not strictly following the Markov property, which, when compounded over larger sentences, leads to confusing, at times even humorous constructions.

after an arbitrarily long period of time (referred to in the literature as the stationary, or ergodic distribution)[Murphy, 2012]. While useful in many contexts and certainly applicable to the results of this thesis, this is not the focus of our work, and thus will not be discussed further.

2.1.4 Simulating new data

Generative sequence models have their uses in many contexts, ranging from chatbots[Hutchens and Alder, 1998][Graves, 2013] to music generation[Paiement et al., 2005]. In cases where the data being modeled strictly obeys the Markov property, Markov Models can generate good quality results. When this is not the case, however, the generated sequences can seem nonsensical, or even be gibberish. Take, for instance, the bigram model from natural language processing; The bigram model can be seen as a trivial Markov Chain where each word is a state. While useful in applications such as spam detection, it is difficult for bigram models to adhere to the finer points of syntax. For example, [Macedo, 2016] generated text samples using a bigram model derived from the King James Bible, producing results such as seen in Figure 2.3.

Considerations about the adequacy of the Markov Property in one scenario or another, simulation can be achieved in Markov Chains by, at each iteration, sampling a new value for $s_i^{(t+1)}$ from the $\{P[s_j^{(t+1)}|s_i^{(t)}]\ \forall s_j \in S\}$ distribution. In a Hidden Markov Model, in addition to using this process for simulating the hidden Markov Chain, an observation must also be sampled from its respective distribution. This process can be repeated indefinitely in order to obtain arbitrarily long sequences. Algorithms 1 and 2 detail this process in pseudocode for Markov Chains and Hidden Markov Models.

2.1.5 External data

The methods presented thus far assume that a great deal of information about the state of the model is already known, namely the probability distribution over the state-space.

```
function sampleMarkovChain($startingState$, $count$, $TransitionProb[][]$)
    $currentState \leftarrow startingState$
    $sequence \leftarrow []$
    while $count > 0$ do
        $sequence \leftarrow sequence + [currentState]$
        $currentState \leftarrow$ sample($TransitionProb[currentState]$)
        $count \leftarrow count - 1$
    end while
    return $sequence$
end function
```

Algorithm 1: Simulates a Markov Chain for a specified number of iterations. The *sample* function receives an array of transition weights, and randomly picks one of the indices with probability proportional to its value in the array.

```
function sampleHMM($startingState$, $count$, $TransitionProb[][]$, $obsModel[]$)
    $states \leftarrow$ sampleMarkovChain($startingState$, $count$, $TransitionProb$)
    $observations \leftarrow []$
    for all $s \in states$ do
        $observations \leftarrow observations +$ [ sample($obsModel[s]$)]
    end for
    return $observations$
end function
```

Algorithm 2: Simulates a Hidden Markov Model by first simulating its Markov Chain, and then generating the associated observations.

In applications where only the external data is given, however, these values must be deduced. Another interesting issue which is associated with this is that of model adequacy: Just how well does the particular Markov Model actually represent some particular set of data? An effective answer to this question will not only aid us in choosing between different possible models, but it also happens to create a framework for optimization (which we discuss in Section 3.1).

First off, let us elaborate on how external data can influence the state-probability vector. In the case of a pure Markov Chain, this probability vector is trivially deducible from the input, which appears as a sequence of states and can be converted to a sequence of distributions by taking a hot-one encoding of the state-sequence. In the case of the HMM, however, determining $P[s_i^{(t)}|X]$ is less trivial.

In the general case, $P[s_i^{(t)}|X]$ can be obtained in an HMM using the forward-backward algorithm (a more in depth explanation of this algorithm can be seen in [Devijver, 1985]), however, for the sake of prediction and model adequacy, it turns out that it is only necessary to compute $P[s_i^{(t)}|x^{(0)} \dots x^{(t)}]$ – the state distribution immediately after all the observations have ended – which we now do:

First, from our assumption of the Markov property, we can state that any two observations are independent, so long as the occurrence of one of the intermediate states is defined:

$$\begin{aligned} P[x^{(t_0)}, x^{(t_1)}|s_i^{(t)}] &= P[x^{(t_0)}|s_i^{(t)}]P[x^{(t_1)}|s_i^{(t)}] \quad \forall\, t_0 \leq t \leq t_1 \wedge t_0 \neq t_1 \\ P[x^{(0)} \dots x^{(t)}|s_i^{(t)}] &= P[x^{(0)} \dots x^{(t-1)}|s_i^{(t)}]P[x^{(t)}|s_i^{(t)}] \end{aligned} \tag{2.5}$$

Using this, we isolate $x^{(t)}$ from $x^{(0)} \dots x^{(t-1)}$ in $P[s_i^{(t)}|x^{(0)} \dots x^{(t)}]$ using two applications of the Bayes theorem:

$$\begin{aligned} P[s_i^{(t)}|x^{(0)} \dots x^{(t)}] &= \frac{P[x^{(0)} \dots x^{(t)}|s_i^{(t)}]P[s_i^{(t)}]}{P[x^{(0)} \dots x^{(t)}]} \\ &= \frac{P[x^{(0)} \dots x^{(t-1)}|s_i^{(t)}]P[x^{(t)}|s_i^{(t)}]P[s_i^{(t)}]}{P[x^{(0)} \dots x^{(t)}]} \\ &= \frac{\left(\frac{P[s_i^{(t)}|x^{(0)}\dots x^{(t-1)}]P[x^{(0)}\dots x^{(t-1)}]}{P[s_i^{(t)}]}\right) P[x^{(t)}|s_i^{(t)}]P[s_i^{(t)}]}{P[x^{(0)} \dots x^{(t)}]} \\ &= \frac{P[s_i^{(t)}|x^{(0)} \dots x^{(t-1)}]P[x^{(0)} \dots x^{(t-1)}]P[x^{(t)}|s_i^{(t)}]}{P[x^{(0)} \dots x^{(t)}]} \end{aligned} \tag{2.6}$$

Now, using the fact that the probability distribution over all states must add up to 1, we can remove the pure observation probabilities (e.g.: $P[x^{(0)}...x^{(t)}]$) in order to obtain a more computable value:

$$\begin{aligned} 1 &= \sum_{s_i^{(t)} \in S} P[s_i^{(t)}|x^{(0)} \dots x^{(t)}] \\ &= \sum_{s_i^{(t)} \in S} \frac{P[s_i^{(t)}|x^{(0)} \dots x^{(t-1)}]P[x^{(0)} \dots x^{(t-1)}]P[x^{(t)}|s_i^{(t)}]}{P[x^{(0)} \dots x^{(t)}]} \\ &= \left(\frac{P[x^{(0)} \dots x^{(t-1)}]}{P[x^{(0)} \dots x^{(t)}]}\right) \sum_{s_i^{(t)} \in S} P[s_i^{(t)}|x^{(0)} \dots x^{(t-1)}]P[x^{(t)}|s_i^{(t)}] \end{aligned} \tag{2.7}$$

$$\begin{aligned} P[s_i^{(t)}|x^{(0)}\dots x^{(t)}] &= \frac{P[s_i^{(t)}|x^{(0)}\dots x^{(t)}]}{\sum_{s_j^{(t)}\in S} P[s_j^{(t)}|x^{(0)}\dots x^{(t)}]} \\ &= \frac{P[s_i^{(t)}|x^{(0)}\dots x^{(t-1)}]P[x^{(t)}|s_i^{(t)}]}{\sum_{s_j^{(t)}\in S} P[s_j^{(t)}|x^{(0)}\dots x^{(t-1)}]P[x^{(t)}|s_j^{(t)}]} \end{aligned} \tag{2.8}$$

Finally, we show that $P[s_i^{(t)}|x^{(0)}\dots x^{(t-1)}]$ can be defined recursively in function of $P[s_j^{(t-1)}|x^{(0)}\dots x^{(t-1)}]$:

$$\begin{aligned} P[s_i^{(t)}|x^{(0)}\dots x^{(t-1)}] &= \sum_{s_j^{(t-1)}\in S} P[s_i^{(t)}|s_j^{(t-1)}, x^{(0)}\dots x^{(t-1)}]P[s_j^{(t-1)}|x^{(0)}\dots x^{(t-1)}] \\ &= \sum_{s_j^{(t-1)}\in S} P[s_i^{(t)}|s_j^{(t-1)}]P[s_j^{(t-1)}|x^{(0)}\dots x^{(t-1)}] \end{aligned} \tag{2.9}$$

Putting Equations 2.8 and 2.9 together, we get:

$$P[s_i^{(t)}|x^{(0)}\dots x^{(t)}] = \frac{P[x^{(t)}|s_i^{(t)}]\sum_{s_j^{(t-1)}\in S} P[s_i^{(t)}|s_j^{(t-1)}]P[s_j^{(t-1)}|x^{(0)}\dots x^{(t-1)}]}{\sum_{s_j^{(t)}\in S} P[x^{(t)}|s_j^{(t)}]\sum_{s_k^{(t-1)}\in S} P[s_j^{(t)}|s_k^{(t-1)}]P[s_k^{(t-1)}|x^{(0)}\dots x^{(t-1)}]} \tag{2.10}$$

Since $P[s_i^{(t)}|s_j^{(t-1)}]$ is the state transition probability, given by the model, and $P[x^{(t)}|s_i^{(t)}]$ is the observation model, also given, all terms in Equations 2.10 are defined either by the model or by recursion, and thus, using this, it follows that if we can compute the probability distribution at t, we can also compute it at $t+1$. Since the probability distribution at time $t = 0$ is given by the model, this in turn allows for us to compute values of this type for an arbitrarily long sequence.

The above result, referred to in the literature as the *forward algorithm*, is important for several reasons: First, it allows us to compute the state-probability distribution at the end of the observed data, which is the basis for prediction (as seen in Section 2.1.3), but also, and perhaps more importantly, it is one of the components needed to quantify how well the model represents the provided data (which we will use in Section 3.1 to derive our optimization routine).

While there are many ways we could go about measuring how well a model can represent data, one method which stands out is the notion of *likelihood.* In probability, the term *likelihood* is employed with respect to some specific set of data and a model, and it represents the probability assigned by the model for that particular data occurring in its world. *Likelihood* can be a good way of measuring and comparing the adequacy of models since, if one model assigns a much greater probability to some collection of samples than another model, then that is an indication that the former better represents the example data.

For example, consider two different models for coin tosses, one which assumes the coin will land on each side roughly evenly, and another which gives a 3/4 chance for landing on heads, and a 1/4 chance for landing on tails. Now suppose that, in flipping a particular coin 100 times we got a specific sequence of heads and tails, and ended up with 81 heads and 19 tails. This scenario is possible in both models, however the first one will give a probability of $0.5^{100} \approx 7.9 * 10^{-31}$ for that particular sequence, whereas the second model will give a probability of $0.75^{81}0.5^{19} \approx 1.4*10^{-16}$ for the same sequence. The second model is a million billion times more likely to have generated that particular series of tosses, and indeed is a better fit for the data (since having such an exceedingly large disparity between heads and tails wouldn't be expected from an unbiased coin).

In the case of the Markov Chain, observations appear as a sequence of states. The likelihood function can be computed on top of these values by comparing the expected distribution achieved by applying the model on the previous given state, and comparing it with the new one (that is, by computing how likely it would have been for a sampling process of the Markov Chain to have produced that particular sequence):

$$\begin{aligned}
&o^{(t)} \in O : \text{Observed state indices sequence}\\
&[o^{(t)} = i] \leftrightarrow s_i^{(t)}\\
&\qquad L = P[O|Model] = P[o^{(0)} \dots o^{(T)}]\\
&\qquad\quad = P[o^{(0)}] \times P[o^{(1)}|o^{(0)}] \times P[o^{(2)}|o^{(0)}, o^{(1)}] \times \dots \times P[o^{(t)}|o^{(0)} \dots o^{(t-1)}]\\
&\qquad\quad = P[o^{(0)}] \prod_{t=1}^{T} P[o^{(t)}|o^{(0)} \dots o^{(t-1)}] = P[s_{o^{(0)}}^{(0)}] \prod_{t=1}^{T} P[s_{o^{(t)}}^{(t)}|s_{o^{(0)}}^{(0)} \dots s_{o^{(t-1)}}^{(t)}]\\
&\qquad\quad = P[s_{o^{(0)}}^{(0)}] \prod_{t=1}^{T} P[s_{o^{(t)}}^{(t)}|s_{o^{(t-1)}}^{(t)}]
\end{aligned} \tag{2.11}$$

All terms in Equation 2.11 are either given by the model, or received as inputs. The likelihood derivation for the HMM is similar:

$$\begin{aligned}
L &= P[X|Model] \\
&= P[x^{(0)} \dots x^{(T)}] \\
&= P[x^{(0)}] \times P[x^{(1)}|x^{(0)}] \times P[x^{(2)}|x^{(0)}, x^{(1)}] \times \dots \times P[x^{(t)}|x^{(0)} \dots x^{(t-1)}] \\
&= P[x^{(0)}] \prod_{t=1}^{T} P[x^{(t)}|x^{(0)} \dots x^{(t-1)}] \\
&= \left(\sum_{s_i^{(0)} \in S} P[x^{(0)}|s_i^{(0)}] P[s_i^{(0)}] \right) \prod_{t=1}^{T} \sum_{s_i^{(t)} \in S} P[x^{(t)}|s_i^{(t)}, x^{(0)} \dots x^{(t-1)}] P[s_i^{(t)}|x^{(0)} \dots x^{(t-1)}] \\
&= \left(\sum_{s_i^{(0)} \in S} P[x^{(0)}|s_i^{(0)}] P[s_i^{(0)}] \right) \prod_{t=1}^{T} \sum_{s_i^{(t)} \in S} P[x^{(t)}|s_i^{(t)}] P[s_i^{(t)}|x^{(0)} \dots x^{(t-1)}]
\end{aligned} \tag{2.12}$$

Where the $P[s_i^{(t)}|x^{(0)} \dots x^{(t-1)}]$ term in Equation 2.12 can be computed recursively by Equation 2.10. The remainder of the terms in the equation are given by the model.

Note that *log-likelihood* (denoted by $l = \log L$), is generally preferred over likelihood as it improves readability (i.e., comparing -3 and -4 is easier to compare visually than 0.001 and 0.0001) and generally has better numerical stability, on account of the limited precision of floating point numbers (an IEEE754 32 bit float[iee, 1985] will underflow for likelihoods less than $\approx 1.18 \times 10^{-38}$, which occur frequently). For operations such as optimization, transforming into the log-domain is harmless, as the log function in the domain of L ($L \in [0, 1] \rightarrow \log L \in [-\infty, 0]$) is bijective and monotonically increasing, meaning one can always retrieve L from l, and that an optimizer which maximizes l also maximizes L.

2.1.6 The Baum Welch algorithm

Up until now, the discussion has revolved around the background and utilization of pre-defined Markov Models for assorted tasks. But what of cases where only the data is known, from which a model must be derived? Cases such as this are perhaps the most common use-case for this sort of method, as it is rarely the case that a problem presents itself as a fully-defined Markov Model. Expanding on the discussion from Section 2.1.5, in which we illustrated that it is not sufficient that a model can explain some set of data, but rather that it must also exhibit a good likelihood estimation of it, we discuss in this section the means by which the parameters of Markov Chains and Hidden Markov Models can be obtained through the Baum Welch algorithm.

Discovered in the late 1960s by Leonard E. Baum and Lloyd R. Welch, the Baum Welch algorithm[Baum et al., 1970] is an extension of the Expectation-Maximization algorithm[Moon, 1996] which iteratively alters the parameters of an HMM by jumping between local maximas of a proxy to log-likelihood. While it doesn't guarantee convergence to the optimal solution, it can very quickly and very effectively find good solutions, and, as such, is widely used as the de-facto means of building Hidden Markov Models (although also being extensible to Markov Chains through the use of a restricted observation model).

A more detailed look into how the algorithm is derived can be seen in [Tu, 2015], however, the outline of the steps involved in coming up with it are as follows:

1. Establish the proxy function which, when optimized will also optimize the log likelihood, but that also facilitates derivation in the following steps
2. Come up with an analytic expression of the proxy function's value over the observation and state model
3. Insert Lagrange multipliers where applicable (in observations and state probabilities) to ensure that the gradient of the proxy function is 0 only when the restraints that probability distributions must sum to one are true
4. Calculate the gradient of the altered proxy function w.r.t each parameter
5. Equal the gradients to zero – in order to find the maxima – and then solve the resulting system nonlinear equations in order to find the equation for local maxima.

The end result of the above process will be a formula which depends on the current configuration of the model (initial probabilities, transition weights and observation distribution function), and which, when computed, will return a point which has a superior likelihood to the present position, and is a local maxima with respect to the current parameters.

While this algorithm is highly useful in the context of HMMs, it is ill suited for the purposes of our thesis. For instance, even within Hidden Markov Models, the complexity involved in extending the Baum Welch algorithm to other observation models is non-trivial, and can even be impossible (if, in doing so, one bumps into an system of equations which can't be solved analytically, such as when using Neural Networks as inputs); Furthermore, without significant changes, the Baum Welch algorithm is restricted to optimizing likelihood as its objective function, restricting the use of HMMs to their current context; Finally, it is the goal of this thesis to not only deal with HMMs, but also the much broader class of Markov Models, for which constructing a derivation of the expectation-maximization algorithm may not yet exist, and would be unreasonable to expect the average user of these methods to attempt to create one every time he or she attempted to mix Markov Models with other methods in a novel way. With this in mind, we move onto Section 2.2, where we propose Gradient Descent Optimization as a solution to these problems.

2.2 Gradient Descent Optimization

In the context of machine learning, function optimization plays a key role in allowing the data scientist to focus on the underlying architecture and structure of the processing involved, rather than the specific parameters associated with his or her particular task. Take, for instance, the process of trying to create a computational model which can identify if a given picture contains a chair or not. In this task, a data scientist might think of a wooden chair, and conceive that it might contain a large set of specific angles, representing the angles between its legs, the back support and the seat. Then, he or she might design a feature detector to look for these angles within the image at particular locations. While this strategy may be ultimately effective, the true difficulty for these kinds of tasks arises in the details, such as, in this example, what precise locations should be examined for the given features, what kinds of angles are to be expected, and what to do with backgrounds which might contain additional, yet irrelevant, angles.

In order to deal with these details, any specific numeric parameters must first be established. Rather than giving this tedious, often humanly impossible task to the data scientist, a function optimizer is usually employed which, when given some goal/objective/error function, will then find a good fit for the parameters of the model which maximizes it (in the case of the chair problem, this could be to maximize the number of correctly classified example images).

A first approach to this problem is the Hill Climbing algorithmRussell and Norvig [1995], which is described in Algorithm 3. While useful in many applications, one of the drawbacks of this approach is the computational complexity involved in having to repeat this process for every single parameter in the model numerous times, as

```
function hillClimb($X$, $iters$, $params$, $stepSizes$, $stepDecay$, $objFun$)
    $objVal \leftarrow$ objFun($params$, $X$)
    while $iters > 0$ do
        for all $i \in |params|$ do
            $candA \leftarrow \{params[0..i-1], params[i] + stepSizes[i], params[i+1...]\}$
            $objValA \leftarrow$ objFun($candA$, $X$)
            $candB \leftarrow \{params[0..i-1], params[i] - stepSizes[i], params[i+1...]\}$
            $objValB \leftarrow$ objFun($candB$, $X$)
            if $objValA > objVal$ then
                $params \leftarrow candA$
                $objVal \leftarrow objValA$
            end if
            if $objValB > objVal$ then
                $params \leftarrow candB$
                $objVal \leftarrow objValB$
            end if
        end for
        $stepSizes \leftarrow stepDecay \times stepSizes$
        $iters \leftarrow iters - 1$
    end while
    return $params$
end function
```

Algorithm 3: Example of the Hill Climbing algorithm, an algorithm whose goal is to find the set of parameters which maximizes the objective function. It does this by sequentially varying each parameter, and checking if, after the change, the final solution improved. If it did, it then takes that to be the new value of the parameter, otherwise, nothing changes. In this case, the algorithm stops when a fixed number of iterations has elapsed. Other variants of the algorithm change the way candidate parametrizations are constructed, what the stopping conditions are, or introduce methods for handling multiple instances of input, but the general principles remain the same. Note that the algorithm can trivially altered so it minimizes the objective function by flipping the sign on *objFun*.

both the total parameter and iteration counts may number in the tens of thousands, therefore requiring an exceedingly large number of iterations before completion. A faster approach would be to find a way to simultaneously measure how changes in each parameter would affect the objective function, and this is precisely where the Gradient Descent algorithm comes in.

In its essence, the gradient is a way of measuring how changes in one value affect another, dependent amount. Consequently, Gradient Descent requires an analytical formula for calculating the gradient of the objective function with respect to each parameter, which it then uses to construct an algorithm which iteratively increments

```
function GRADIENTDESCENT($X$, $iters$, $params$, $\alpha$, $\alpha Decay$, $\nabla objFun$)
    while $iters > 0$ do
        $grad \leftarrow \nabla$OBJFUN($X$, $params$)
        $params \leftarrow params - \alpha \times grad$
        $\alpha \leftarrow \alpha \times \alpha Decay$
    end while
    return $params$
end function
```

Algorithm 4: Example of the Gradient Descent algorithm. Similar to the Hill Climbing algorithm (Algorithm 3), the Gradient Descent algorithm aims to find the set of parameters which optimizes the objective function. In contrast with the Hill Climbing algorithm, however, the algorithm computes the direction of improvement for all parameters simultaneously, and then updates all of them accordingly. The example illustrates minimization of the objective function, however maximization can also be achieved by flipping the sign on the objective function or on the *params* update step. Many variants of the Gradient Descent algorithm exist, which mainly change how the weight update step occurs, or what to do if there are multiple instances of input. Note that, similar to the Hill Climbing algorithm, the Gradient Descent algorithm can be turned into the Gradient Ascent algorithm, which maximizes its objective function, merely by flipping the sign on $\nabla objFun$.

the parameters of a model, in proportion to the magnitude of the current gradient, with the goal of minimizing (or maximizing) the objective function. The method and reasoning behind how this algorithm works is detailed in Algorithm 4, however, a useful analogy is that of a marble on a hilly surface: If the marble's position on the surface represents the current set of parameters being optimized, and the height of this surface at any point is the objective function, the algorithm loosely describes the rolling movement the marble will take going from a random starting position towards lower and lower points, traveling faster in directions which are steeper, and eventually settling down once it's fallen into some hole or divot.

The Gradient Descent algorithm isn't perfect, however. In addition to being limited to derivable functions, it has many flaws, for example, it is not guaranteed to find the optimal solution, since, like the Hill Climbing algorithm, it can get stuck in local optima, where any changes within the local neighborhood in the parameter space would lead to a worsening of the objective function, even if beyond this neighborhood there might be a more optimal configuration (in the analogy of the marble, the marble has settled in some pit on the surface from where it can't get out, but which also isn't the absolute lowest point). Unlike in the Hill Climbing algorithm, Gradient Descent is not guaranteed to always improve its solution from iteration to iteration, since the gradient is only a measure of how the function changes locally, but its movement is done in steps. Finally, convergence can also be slow using this method, if, for example, small changes in a parameter only affects the objective function a tiny amount, but very large changes could lead to significant improvements, in which case the value will slowly crawl to where it should be (again, in the marble analogy, it may be rolling down a very slight hill, and, even if the hill is very deep, it may still take a long time to get there on account of it not being very steep).

In spite of these difficulties, the Gradient Descent algorithm has been extensively used in many different methods, perhaps most famously within the context of Neural Networks (see Section 2.2.1). There are many reasons for this, such as the fact that it works on many more kinds of problems than other forms of optimization (for example, linear and Hessian optimization, which both require convexity constraints[Martens and Sutskever, 2012]), however, we argue that the main reason for this is the chain rule, which allows one to calculate the gradient of an appropriate objective function with relative ease, and introduces inherent modularity to the solutions obtained through these means.

$$\begin{aligned} f(x) &= g(h(x)) \\ \frac{\delta f}{\delta x} &= \frac{\delta}{\delta x} g(h(x)) \\ &= \frac{\delta g}{\delta h}\frac{\delta h}{\delta x} \\ f'(x) &= g'(h(x))h'(x) \end{aligned} \tag{2.13}$$

$$\begin{aligned} f(x,y) &= g(h(x,y)) \\ \frac{\delta f}{\delta x} &= \frac{\delta g}{\delta h}\frac{\delta h}{\delta x} \\ \frac{\delta f}{\delta y} &= \frac{\delta g}{\delta h}\frac{\delta h}{\delta y} \end{aligned} \tag{2.14}$$

The chain rule, shown in Equation 2.13, allows for us to split the gradient computation into independent components. For example, consider some objective function $f(g(h(p)))$ which depends on the enclosed parameter p. In order to calculate the gradient $\frac{\delta f}{\delta p}$ we would only need to calculate $\frac{\delta f}{\delta g}$, $\frac{\delta g}{\delta h}$, and $\frac{\delta h}{\delta p}$. While intuitively this may seem even more complex, after all, we are calculating three derivatives instead of one, the three functions might individually be relatively simple to derive, whereas the expression as a whole might be less more complicated. Furthermore, this modularization can significantly reduce computational effort, such as in Equation 2.14, where the term $\frac{\delta g}{\delta h}$ appears in both $\frac{\delta f}{\delta x}$ and $\frac{\delta f}{\delta y}$, needing, therefore, to only be calculated once.

The modular property of Gradient Descent is arguably the foundation which has allowed for and explains the popularity of large Neural Network libraries. Unlike in more traditional machine learning methods, the aforementioned properties permit that, if the data scientist chooses to create a novel model which is more suited for his or her context, he can do so by mashing together computational units (representing more basic arithmetic operations), with little regard to how an optimizer his design will be made (as the library knows how to construct one from its basic operations). This in turn frees the data scientist to focus on finding an appropriate model for his or her context, and allows the library designers to come up with more efficient optimizers, a wider range of arithmetic operations, and better computational resource management algorithms.

Considering the prominent position Neural Networks exhibit within the class of Gradient Descent-optimized models, we will examine, in the remainder of this section, precisely how these networks are constructed (Section 2.2.1). Then, in conclusion, we have Section 2.2.2, where we explore how modern Neural Network libraries are able to leverage Gradient Descent in order to autonomously construct optimizers for new models – an exceedingly relevant component of this project, as they offer a means to facilitate the interaction of Markov Models and other Gradient Descent methods, such as Neural Networks, without requiring specialist knowledge from the user.

2.2.1 Artificial Neural Networks

The notion of an artificial neural network was first proposed by McCulloch and Pitts in 1943, and was designed as a form of bio-inspired computation, inspired by early insights into the human visual processing system. Since then, however, the biological nature of this technique gradually waned in importance, as researchers realized the versatility inherent in this kind of model. Early on, one of the main motivators for the use of artificial Neural Networks was their ability to deal with novel sources of data with fewer requirements for complex feature engineering. Furthermore, artificial Neural Networks were one of the first models that could deal with nonlinear classification tasks.

In their essence, artificial neural networks are generally built out of two main components, named neurons and weights, which serve as analogies for neurons and synapses in the brain. In these kinds of models, neurons achieve activation through some means, which in turn affects the ability of other neurons, to which they are linked via weights, to also achieve activation. Information is input into the network by inducing activity in one set of neurons, and is extracted by reading out the activation pattern of some set of neurons after some number of iterations. The network's output is then evaluated in some, the results of which often being used to alter the parameters of the network so it will perform better in the future.

While these general principles tend to be the same across the many different kinds of Neural Networks, the specific means by which a neuron can affect another one's ability to activate/fire varies. A full review of the different kinds of neural networks goes beyond the scope of this thesis – and indeed, many forms of neural networks don't even use Gradient Descent for their optimization routines – thus, we will focus on the Perceptron family of neural networks, as they are the most relevant to our thesis and to the contemporary machine learning community in general.

In the Perceptron family of neural networks[Bishop, 2007], neurons accumulate a real valued charge which is used in conjunction with the synaptic weights, the main parameters of the model, in order to alter the charge of other neighbors to which the present neuron is connected. The simplest member of this family is the Perceptron itself, which is comprised of a single neuron, and one weight for each of its real-valued inputs. The mathematics for the Perceptron are shown in Equation 2.15.

$$\begin{aligned}
x_i \in X &: \text{Collection of inputs} \\
w_i \in W &: \text{Collection of real-valued weights, one for each input} \\
b &: \text{Real valued bias parameter} \\
s(X) &= b + \sum_i w_i x_i \\
out &= a(s(X))
\end{aligned} \tag{2.15}$$

As seen in Equation 2.15, the sum of the neuron's weighted inputs $s(X)$ is used in $a(s)$, referred to in the literature as its activation function, to compute the output of the model. $a(s)$ is defined to be some non-linearity (such as a sigmoid, arc-tangent, or relu function), and, except for the bias parameter b, is the only difference between this model and linear regression. Before we can construct an optimizer for this model, however, one must first establish an objective function. For example, if the Perceptron is being used in regression, one possible objective function could be to minimize the square error (seen in Equation 2.16).

$$\begin{aligned} y &: \text{expected output for some input X} \\ err &= (y - out)^2 \end{aligned} \tag{2.16}$$

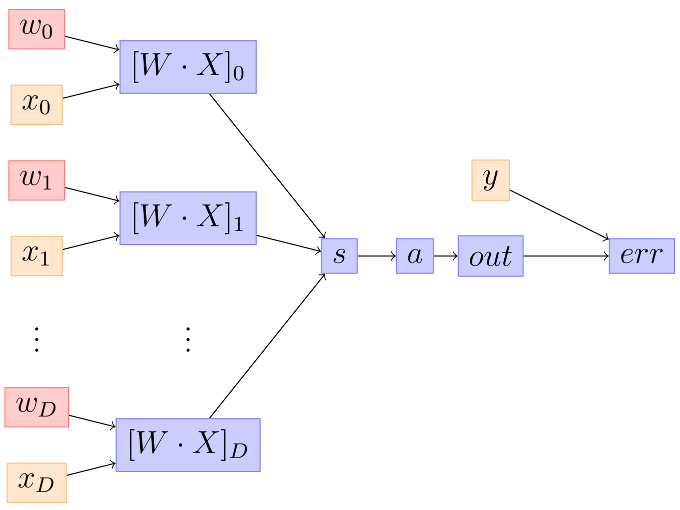


Figure 2.4. Information flow between inputs (orange), parameters (red), and calculated outputs (blue) in a Perceptron. The specific operations at each node are given by Equation 2.15. Different objective functions may yield to different graph configurations to the right of the output.

Using the chain rule (Equation 2.13), we can derive the gradient for the Perceptron function w.r.t. its parameters. While in this case the derivative is relatively simple, it will be useful in the future, when models become more complex, to build a graph, such as seen in Figure 2.4, in order to understand the relations between each term being calculated. In this graph, an edge is created whenever a symbol is used in the definition of another, going from used to user[2]. We can use this graph to determine what partial derivatives are needed, and in what order they need to be computed:

The gradient of the objective/error function with respect to a node n_i, $\frac{\delta err}{\delta n_i}$ is the sum of the gradients of the objective function with respect to each of the nodes n_j which use n_i as a symbol, multiplied by their partial derivative $\frac{\delta n_j}{\delta n_i}$. That is:

[2]In the case of recursive definitions where values are updated on further iterations (such as RNNs) a new instance of the node must be created each time its value is changed.

$$\frac{\delta err}{\delta n_i} = \sum_j \frac{\delta err}{\delta n_j} \frac{\delta n_j}{\delta n_i} \tag{2.17}$$

Where j is the set of all nodes pointed to by n_i in the information flow graph. The order by which partial derivatives must be calculated is also given by the graph, and can be taken to be the reverse order in which the terms were originally calculated, or, more generally, any order whereby the gradient with respect to the nodes pointed to by the current node have been calculated before its own gradient calculation takes place. Using this, we calculate the gradients for a Perceptron with a squared error minimization objective function and a sigmoid activation function (that is, $a(s) = \frac{1}{1+e^{-s}}$):

$$\begin{aligned}
\frac{\delta err}{\delta out} &= -2(y - out)\\
\frac{\delta out}{\delta a} &= 1\\
\frac{\delta a}{\delta s} &= \frac{1}{1-e^{-s}}\left(1 - \frac{1}{1-e^{-s}}\right)\\
\frac{\delta s}{\delta b} &= 1\\
\frac{\delta s}{\delta w_i} &= x_i
\end{aligned} \tag{2.18}$$

$$\begin{aligned}
\frac{\delta err}{\delta out} &= -2(y - out)\\
\frac{\delta err}{\delta a} &= \frac{\delta err}{\delta out}\\
\frac{\delta err}{\delta s} &= \left(\frac{\delta err}{\delta a}\right)\left(\frac{1}{1-e^{-s}}\left(1 - \frac{1}{1-e^{-s}}\right)\right)\\
\frac{\delta err}{\delta b} &= \frac{\delta err}{\delta s}\\
\frac{\delta err}{\delta w_i} &= \left(\frac{\delta err}{\delta s}\right)x_i
\end{aligned} \tag{2.19}$$

$$\begin{aligned}\frac{\delta err}{\delta w_i} &= \frac{\delta err}{\delta out} \times \frac{\delta out}{\delta a} \times \frac{\delta a}{\delta s} \times \frac{\delta s}{\delta w_i} \\ &= -2(y - out) \times 1 \times \frac{1}{1-e^{-s}}(1 - \frac{1}{1-e^{-s}}) \times 1 \\ &= -2(y - out)\left(\frac{1}{1-e^{-s}}\right)\left(1 - \frac{1}{1-e^{-s}}\right)\end{aligned} \tag{2.20}$$

Equation 2.18 shows the partial derivatives which need to be computed in order to compute Equation 2.19, which gives the gradient of the objective function with respect to each parameter. Equation 2.20 summarizes these results in a single symbolic formula, however, in practice, a complete symbolic equation such as seen there is rarely derived, as it can result in redundant calculations and is generally unnecessary.

A more complex member of the Perceptron family is the Multilayer Perceptron (MLP)[Bishop, 2007]. The main feature it introduces is the number of computational units used: While in the Perceptron a single neuron is utilized, the Multilayer Perceptron uses multiple neurons, organized into layers, where the outputs of one layer serve as inputs to the next layer. The equations for it are similar to those for the Perceptron, only now they are vectorized by the layer index and the index of the neuron within the layer:

$$\begin{aligned} x_i \in X &: \text{Collection of inputs} \\ w^l_{ij} \in W &: \text{Real-valued weight from neuron } i \text{ of layer } l \text{ to neuron } j \text{ of layer } l+1 \\ b^l_i &: \text{Real valued bias of neuron } i \text{ of layer } l \\ a^0_i &= x_i \\ s^l_i &= b^l_i + \sum_j w^{l-1}_{ji} a^{l-1}_j \\ a^l_i &= a(s^l_j) \\ out &= [a^L_i \ \forall i] \end{aligned} \tag{2.21}$$

Note that now *out* can be a vector instead of a scalar. We follow a similar procedure as before, and derive the information flow graph in Figure 2.5, followed by the partial and full gradient computations in Equations 2.22 and 2.23.

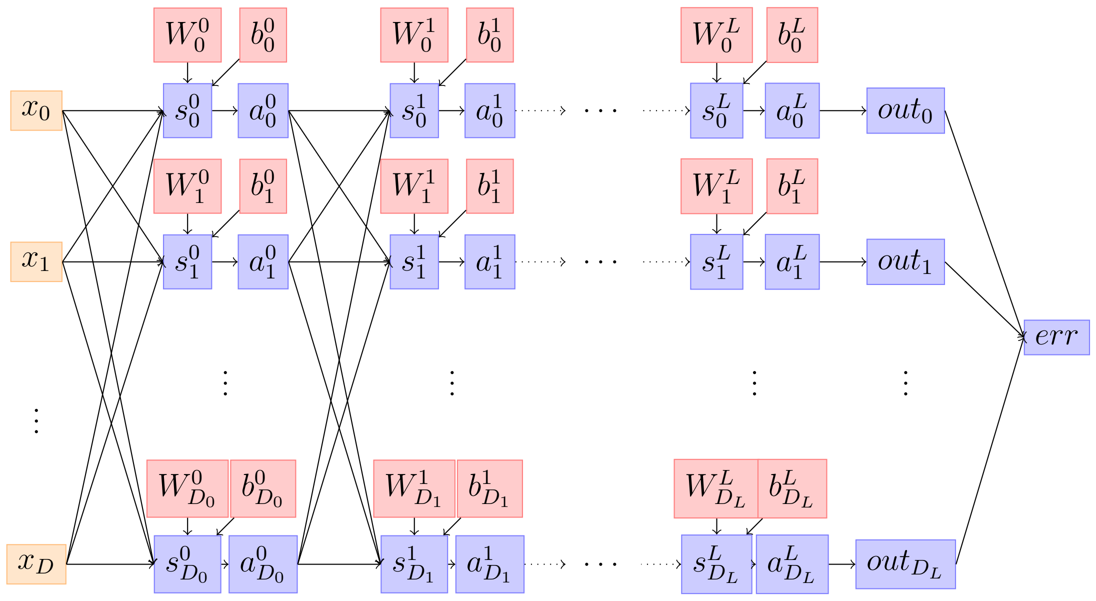


Figure 2.5. Information flow in a Multilayer Perceptron. The diagram's style is the same as in Figure 2.4, however, in the interest of legibility, the individual weight scalars $w_{i0}^l, w_{i1}^l, \ldots, w_{iD_l}^l$ were grouped into W_i^l.

$$\begin{aligned}
\frac{\delta err}{\delta a_i^L} &= \frac{\delta err}{\delta out_i} \\
\frac{\delta a_i^l}{\delta s_i^l} &= a'(s_j^l) \\
\frac{\delta s_i^l}{\delta b_i^l} &= 1 \\
\frac{\delta s_i^l}{\delta a_j^{l-1}} &= w_{ji}^{l-1} \\
\frac{\delta s_i^l}{\delta w_{ji}^{l-1}} &= a_j^{l-1}
\end{aligned} \tag{2.22}$$

$$\begin{aligned}
\frac{\delta err}{\delta a_i^L} &= \frac{\delta err}{\delta out_i} \\
\frac{\delta err}{\delta s_i^l} &= \left(\frac{\delta err}{\delta a_i^l}\right) a'(s_j^l) \\
\frac{\delta err}{\delta b_i^l} &= \frac{\delta err}{\delta s_i^l} \\
\frac{\delta err}{\delta a_j^{l-1}} &= \sum_i \left(\frac{\delta err}{\delta s_i^l}\right) w_{ji}^{l-1} \\
\frac{\delta err}{\delta w_{ji}^{l-1}} &= \left(\frac{\delta err}{\delta s_i^l}\right) a_j^{l-1}
\end{aligned} \tag{2.23}$$

Equations 2.22 and 2.23 compute the gradient for the MLP in much the same manner as was done for the Perceptron. Of import are the differences pertaining to $\frac{\delta err}{\delta a_j^{l-1}}$, where a summation must be introduced since we now have that a single component, a_j^{l-1}, is used as a symbol in multiple other functions (as seen in the information flow graph, Figure 2.5), as well as the omission of $\frac{\delta err}{\delta out_i}$ and $a'(s_j^l)$, as these were respectively exemplified by the squared-error and sigmoid functions in the Perceptron example, whereas here they have been left as placeholders.

Other members of the Perceptron family include Recurrent Neural Networks[Pearlmutter, 1995], Convolutional Neural Networks[LeCun et al., 1998a] and Autoencoders[Bengio et al., 2009]. While each of these models has its own uses and peculiarities, and variations on how the gradient is used do exist – be that implementation, scaling, initialization or stopping procedures – the process described here for obtaining a functional optimizer for an arbitrary model utilizing Gradient Descent stands, giving credence to the interoperability of these kinds of models, as well as their simplicity of use from the perspective of the data-scientist, so long as he can relegate the gradient computation to a library.

2.2.2 Automatic differentiation

As outlined in the beginning of this section (and further exemplified in Section 2.2.1), the process for deriving a Gradient Descent optimizer from a model specification is relatively simple given a comfortable understanding of the calculus and methods involved. While "simple", requiring this particular skill-set from the average data scientist, or even enthusiast who just wants do design a machine learning algorithm to model their problem, however, is hardly reasonable. To this end, the notion of automatic differentiation[Baydin et al., 2015] was proposed, which aims to automate the already mechanical process of computing the numerical derivative of a function.

The general principles behind automatic differentiation are those already seen in this section: One uses the chain rule (Equation 2.13) to calculate the partial derivatives of the model (such as was done for the Perceptron and MLP in Equations 2.18 and 2.22), and then combines these as needed in order to produce the desired gradient. The specific way in which this is done varies, however, with the two main methods being referred to as forward and reverse mode automatic differentiation.

Reverse mode automatic differentiation follows the progression utilized in Section 2.2.1: In order to calculate $\frac{\delta f}{\delta x}$ in $f(g(h(x)))$, for example, one first calculates $\frac{\delta f}{\delta g}$, which is then multiplied by $\frac{\delta g}{\delta h}$, which is then finally multiplied by $\frac{\delta h}{\delta x}$. The "reverse" term is added to the name of this algorithm in reference to the order in which computations are done, following the reverse order in which calculations were performed in the first place. The main drawback of this method is the need to recall the partial derivatives in reverse order, which often requires recomputation and/or the storage of intermediate results (in the example, one would need to store or recompute $h'(x)$, $g'(h(x))$ and $f'(g(h(x)))$).

One way to avoid the problem of recomputing partial derivatives is to utilize forward mode automatic differentiation. In this method, instead of propagating the gradient backwards from the error source, it is instead initiated when the parameters are defined, and then updated as the calculation evolves. For example, once again using $\frac{\delta f}{\delta x}$ in $f(g(h(x)))$ as our target, forward mode automatic differentiation would start with $\frac{\delta x}{\delta x} = 1$, and then multiply that value in sequence by $\frac{\delta h}{\delta x}$, $\frac{\delta g}{\delta h}$, and $\frac{\delta f}{\delta g}$. The advantage of not needing to recall the partial derivatives occurs since one can calculate the partial derivative with respect to the variable (x in this case) at the same time each computation of the model is being done, at which point all the information needed is already available.

Forward mode automatic differentiation is not without its drawbacks, however, as the computational effort required to compute the gradient scales linearly with the number of parameters. This occurs since every node must keep track of its gradient with respect to every parameter, unlike in reverse mode where the gradients are only split off on a parameter basis when that leaf is reached while traversing the computation graph. A similar penalty is incurred with reverse mode automatic differentiation in the case of multiple objective functions, though such is not the case in most machine learning algorithms, as even when there are multiple objective functions one desires to combine them somehow, not derive different models to maximize each of them separately.

There are several approaches for implementing automatic differentiation. On one side, Neural Network libraries such as *TensorFlow*[Abadi et al., 2016], *Theano*[Bergstra et al., 2011] and *Torch*[Collobert et al., 2002] rely on the pre-coding of hundreds of complex tensor operations (such as matrix and dot products, mass applications of specific functions, and even convolution), with pre-coded gradient calculations. The advantages here are two-fold: first, the use of block operations allows for function-level parallelism to be tailored to each specific operation, facilitating, for instance, the use of GPUs, and second, the increased complexity of each operation reduces the total number of operations required, granting library designers' handmade optimizations of each operation a greater overall significance.

On the other side, automatic differentiation can be implemented on a more granular level, as was implemented for the purpose of this thesis. In this paradigm, only a handful of operations and their gradients need to be encoded, and the user has more liberty to design their functions, unimpeded by the lack of some specific complex operation which the library designers deemed unnecessary. The drawback, of course, is efficiency, as the entire burden of optimization falls on to the automatic differentiation implementation, which has yet to surpass the human level-efficacy of handmade tensor operations. Effective parallelism becomes a significant issue here, as automatic parallelization is itself an active area of research in general [Banerjee et al., 1993].

Considering the positive and negative aspects of both of these implementation paradigms, we made use of both techniques in this thesis. We found tensor-based approaches useful in dealing with simpler concepts, as they provided more speed, and then, in order to prototype more complex models where a tensor-based approach would be cumbersome, we made use of our own Domain Specific Language.

Chapter 3

Extending Markov Models Through Gradient Descent

As outlined in Chapter 1, we posit in this thesis that the use of Gradient Descent in the context of Markov Models can be a means for extending the usability of this kind of method. In particular, we claim that, by using Gradient Descent optimization, one can co-optimize the observation function along side the transition model, leading to a more diverse set of observation functions, and, additionally, better sequence-oriented architectures for this category of methods. In this chapter, we explore the means by which we intend to demonstrate these statements, and, in so doing, show that Gradient Descent, as applied to Markov Models, is both an effective optimizer and yields a versatile set of tools.

In order to accomplish this, we first derive and examine the Gradient Descent formulation for a Hidden Markov Model (Section 3.1). While ultimately we make use automatic differentiation to construct our optimizers, manually constructing an optimizer is an important exercise not only from a chronological perspective – as building the optimizer manually was the first step taken in the direction of this thesis – but is also necessary for understanding how to define these models in such a way that automatic differentiation can effectively operate on this class of models.

Next we define two novel uses for Markov Models, whose aim is to showcase the versatility of models constructed out of the union of Markov Models and Gradient Descent. The first example, Section 3.2.1, is a visual data classifier which uses convolutional layers as its basis, but relies on a Markov Model's likelihood estimation to come up with the classification. The second example, Section 3.2.2, stacks multiple layers of Markov Models, employing the sequence-likelihood given by one model as an observation for another. This solves some of some of the limitations of standard

HMMs, and allows us to model more complex behavior in time series using this kind of method.

3.1 Gradient Descent Formulation

Markov Models, as discussed in Section 2.1, make use of a set of parameters in order to establish the probabilities of a transition occurring between one state and another at one moment in time ($P[s_i^{(t+1)}|s_j^{(t)}]$), as well as a set of initial probabilities ($\{P[s_i^{(0}] \; \forall i\}$). Furthermore, as seen in Section 2.1.5, we can use the statistical quantity *likelihood* as a metric for how well a model represents some set of real data. Combining the concepts of likelihood, the existence of real-valued model parameters, and the Gradient Descent optimization techniques discussed in Section 2.2, we can formulate optimization strategies for Markov Models.

Generally speaking, in order to optimize the performance of a Markov Model on some set of set of real data using Gradient Descent, we start with a random set of parameters, which we then iteratively improve upon using the Gradient Descent algorithm, Algorithm 4. Specifically, we utilize Gradient Ascent, as we wish to maximize the likelihood of the model over the data. Since a likelihood function is generally easy to compute in Markov Models, this strategy can generally be applied to all members of the family, however, for the sake of brevity, we will only manually derive the Gradient Descent optimizer for the HMM, leaving derivations of further models to automatic differentiation (which functions as detailed in Section 2.2.2).

To begin, our first step in establishing a Gradient Descent optimizer for the Hidden Markov Model is to formulate some set of equations which, when computed in sequence, will give us the total likelihood of the model for the given data. For this we can rely on Equation 2.12, from which we construct Algorithm 5, which computes the log-likelihood of a HMM with respect to some data.

Next, in Equation 3.1, we compute the derivatives of each term from Algorithm 5, which we then use to construct Algorithm 6. Algorithm 6 is constructed by reversing the control flow of Algorithm 5, and then replacing each term with its gradient expressions (as computed in Equation 3.1). Note that exactly reversing the control flow is sufficient for ensuring that each node's children's gradients are always computed before their own, and, when coupled with making all gradient expressions incremental (e.g.: $\frac{\delta err}{\delta f} \leftarrow \frac{\delta err}{\delta f} + \frac{\delta err}{\delta g}\frac{\delta g}{\delta f}$), allows for a completely mechanistic (albeit at times inefficient) method for computing the gradients.

```
function logLikelihoodHMM($X$, $P^{(0)}$, $P_{ij}$, $observationProb[]$)
    $q^{(0..0)} \leftarrow P^{(0)}$
    for all $t$ from 0 to $|X| - 1$ do
        $o^{(t)} \leftarrow \{\ observationProb[i](X^{(t)})\ \forall i\}$
        $p^{(0..t)} \leftarrow q^{(0..t)} \circ o^{(t)}$
        $\alpha^{(t)} \leftarrow \sum_i p_i^{(0..t)}$
        $q^{(0..t+1)} \leftarrow (\alpha^{(t)})^{-1} p^{(0..t)} \cdot P_{ij}$
    end for
    return $l \leftarrow \sum_t \log \alpha^{(t)}$
end function
```

Algorithm 5: Computes the log-likelihood of a given sequence for a Hidden Markov Model. For clarification, $observationProb[i](X^{(t)})$ is a function, associated with state i which gives $P[X^{(t)}|s_i]$, $a \circ b$ is the pairwise product between two vectors, and α is a normalization factor used for numerical stability (otherwise, $p^{(0..t)}$ risks underflowing).

$$
\begin{aligned}
\frac{\delta err}{\delta P_i^{(0)}} &= \frac{\delta err}{\delta q_i^{(0..0)}} \\
\frac{\delta err}{\delta observationProb[i]} &= \sum_t \frac{\delta err}{\delta o_i^{(t)}} \\
\frac{\delta err}{\delta o_i^{(t)}} &= \frac{\delta err}{\delta p^{(0..t)}} q_i^{(0..t)} \\
\frac{\delta err}{\delta q_i^{(t)}} &= \frac{\delta err}{\delta p^{(0..t)}} o_i^{(0..t)} \\
\frac{\delta err}{\delta p_i^{(0..t)}} &= \frac{\delta err}{\delta \alpha^{(t)}} + (\alpha^{(t)})^{-1} \left(\sum_j P_{ij} \frac{\delta err}{\delta q_j^{(0..t+1)}} \right) \\
\frac{\delta err}{\delta \alpha^{(t)}} &= (\alpha^{(t)})^{-1} \frac{\delta err}{\delta l} - (\alpha^{(t)})^{-2} \left(\frac{\delta err}{\delta q^{(0..t+1)}} \cdot \left(p^{(0..t)} \cdot P_{ij}\right) \right) \\
\frac{\delta err}{\delta P_{ij}} &= \sum_t (\alpha^{(t)})^{-1} p_i^{(0..t)} \frac{\delta err}{\delta q_j^{(0..t+1)}}
\end{aligned}
\tag{3.1}
$$

Algorithms 5 and 6 can be used together so as to compute the likelihood and gradient of a set of parameters over some data. This can be done by running the likelihood computation (Algorithm 5), and then running the gradient computation (Algorithm 6) taking $\frac{\delta err}{\delta l}$ to be -1 (any constant will do here, with the magnitude scaling all $\frac{\delta err}{\delta *}$ terms, and the sign determining whether we will maximize or minimize likelihood). Finally, with gradients in hand, the next step would be to run the parameter update step (as described in the Gradient Descent Algorithm, Algorithm 4).

A challenge which appears at this point are the co-dependencies between model parameters, namely that they all must represent valid probability distributions. These restrictions are summarized in Equation 3.2 (we assume for now that *observationProb* is constructed so as to be a valid probability distribution. Later in this section we will discuss what that entails)

$$\begin{aligned}
P_i^{(0)} &\in [0,1] \quad \forall i \\
P_{ij} &\in [0,1] \quad \forall i,j \\
\sum_i P_i^{(0)} &= 1 \\
\sum_j P_{ij} &= 1 \quad \forall i
\end{aligned} \tag{3.2}$$

There are several methods which can be employed in order to ensure the restrictions posed in Equation 3.2 hold. The Baum Welch algorithm (Section 2.1.6), for example, uses Lagrange multipliers in order to ensure that local minima exist only if the restrictions are met, and then jumps between these minima by setting parameters accordingly. This, however, cannot be applied here due to Gradient Descent, which allows for parameters to not always be at a local minima (meaning that, when not in these states, we potentially have invalid probability distributions); Another approach is to fix parameters after the application of the gradient through some form of normalization (e.g.: by adding a constant factor to all values so they all exceed 0 and then scaling them so they sum to 1). Care must be taken when devising these kinds of methods, as manipulating the gradient or values directly often has unintended consequences.

A more organic solution, which was adopted in our formulation, is shown in Equations 3.3 and 3.4. In it, instead of manually fixing the gradient or values, we redefine the model's parameters in function of a set of restrictionless real-valued parameters, which by construction retain the desired restrictions. In summary, our method entails using real valued parameters ($w \in [-\infty, \infty]$), which are squared ($w^2 \in [0, \infty]$), and then normalized so their sum is 1, thus representing a valid distribution.

$$\begin{aligned}
&W_i^{(0)} \in \mathbb{R} \quad \forall i \\
&P_i^{(0)} = \frac{\left(W_i^{(0)}\right)^2 + \epsilon}{\sum_j \left(\left(W_j^{(0)}\right)^2 + \epsilon\right)}
\end{aligned} \tag{3.3}$$

```
function LogLikelihoodHMMGradient($\frac{\delta err}{\delta l}$, {PARTIALS})
    for all $t$ from $|X| - 1$ to $0$ do
        $\frac{\delta err}{\delta \alpha^{(t)}} \leftarrow (\alpha^{(t)})^{-1} \frac{\delta err}{\delta l}$
    end for
    for all $t$ from $|X| - 1$ to $0$ do
        $\frac{\delta err}{\delta \alpha^{(t)}} \leftarrow \frac{\delta err}{\delta \alpha^{(t)}} - (\alpha^{(t)})^{-2} \left( \frac{\delta err}{\delta q^{(0..t+1)}} \cdot \left( p^{(0..t)} \cdot P_{ij} \right) \right)$
        for all $i$ do
            $\frac{\delta err}{\delta p_i^{(0..t)}} \leftarrow (\alpha^{(t)})^{-1} \left( \sum_j P_{ij} \frac{\delta err}{\delta q_j^{(0..t+1)}} \right)$
            for all $j$ do
                $\frac{\delta err}{\delta P_{ij}} \leftarrow \frac{\delta err}{\delta P_{ij}} + (\alpha^{(t)})^{-1} p_i^{(0..t)} \frac{\delta err}{\delta q_j^{(0..t+1)}}$
            end for
        end for
        for all $i$ do
            $\frac{\delta err}{\delta p_i^{(0..t)}} \leftarrow \frac{\delta err}{\delta p_i^{(0..t)}} + \frac{\delta err}{\delta \alpha^{(t)}}$
        end for
        for all $i$ do
            $\frac{\delta err}{\delta o_i^{(t)}} \leftarrow \frac{\delta err}{\delta p^{(0..t)}} q_i^{(0..t)}$
            $\frac{\delta err}{\delta q_i^{(t)}} \leftarrow \frac{\delta err}{\delta p^{(0..t)}} o_i^{(0..t)}$
        end for
        for all $i$ do
            $\frac{\delta err}{\delta observationProb[i]} \leftarrow \frac{\delta err}{\delta observationProb[i]} + \frac{\delta err}{\delta o_i^{(t)}}$
        end for
    end for
    for all $i$ do
        $\frac{\delta err}{\delta P_i^{(0)}} \leftarrow \frac{\delta err}{\delta q_i^{(0..0)}}$
    end for
    return $\{ \frac{\delta err}{\delta P_i^{(0)}}, \frac{\delta err}{\delta P_{ij}}, \frac{\delta err}{\delta observationProb[i]} \}$
end function
```

Algorithm 6: Computes the gradient of the likelihood with respect to each parameter for a Hidden Markov Model. It receives as input the gradient with respect to its output, $\frac{\delta err}{\delta l}$, and a set of partial derivatives {PARTIALS} (the terms which are multiplied by $\frac{\delta err}{\delta *}$ terms). The partial derivatives (calculated in Equation 3.1) all depend on terms computed in the *logLikelihoodHMM* method (Algorithm 5). Note that here we assume that any symbol that has not yet been declared has the value 0. The above portrayal may not represent the most efficient design, as improvements, such as grouping the bodies of similar loops, may be more compact, however it does illustrate that the gradient can be obtained from a rather straight-forward transformation.

$$\begin{aligned} &W_{ij} \in \mathbb{R} \quad \forall i, j \\ &P_{ij} = \frac{W_{ij}^2 + \epsilon}{\sum_k \left(W_{ik}^2 + \epsilon\right)} \end{aligned} \tag{3.4}$$

While the squaring and normalizing alone is sufficient to create a valid probability distribution, the addition of a small constant ϵ (e.g.: $\epsilon = 10^{-5}$) is needed to ensure that probabilities are never exactly equal to zero. As seen in the gradient computation, Equation 3.5, $\frac{\delta err}{\delta w}$ is scaled by w, meaning that, in the event that $w = 0$, the optimizer will be in a rut and will never alter its value. Furthermore, a probability mistakenly taken to be zero leads to a log-likelihood of $-\infty$, which in turn propagates to the gradients which also become ∞, $-\infty$ or `nan`. Zero-probabilities should be avoided in this kind of optimization, and indeed, requiring non-zero probabilities is generally not a problem as a true zero-probability can be inferred in an optimized model by finding w values which have been minimized to 0 (or very close to it).

$$\begin{aligned} w_i &\in \mathbb{R} \quad \forall i \\ p_i &= \frac{w_i^2 + \epsilon}{\sum_j \left(w_j^2 + \epsilon\right)} \\ \frac{\delta err}{\delta w_i} &= \sum_k \frac{\delta err}{\delta p_k} \frac{\delta p_k}{\delta w_i} \\ &= \frac{2w_i}{\sum_j \left(w_j^2 + \epsilon\right)} \left(\frac{\delta err}{\delta p_i}\right) + \sum_k \left(\frac{\delta err}{\delta p_k}\right) \left(-\frac{2w_i \left(w_k^2 + \epsilon\right)}{\left(\sum_j \left(w_j^2 + \epsilon\right)\right)^2}\right) \\ &= \left(\frac{2w_i}{\sum_j \left(w_j^2 + \epsilon\right)}\right) \left(\left(\frac{\delta err}{\delta p_i}\right) - \sum_k p_k \left(\frac{\delta err}{\delta p_k}\right)\right) \end{aligned} \tag{3.5}$$

Finally, on the matter of the restrictions on the observation probabilities, there are a couple points to be detailed. Firstly, it is important to note that the techniques used for ensuring that parameters form valid probability distributions can also be applied to observation functions which define a distribution over a finite number of values. While useful in many applications (and indeed the main use of Hidden Markov Models), the restrictions imposed by requiring that *observationProb* always returns a strict probability distribution over the set of states are limiting, and can often have unintended consequences.

Take, for instance, a Hidden Markov Model which is attempting to model $\mathbb{R}^2$ data using a set of Bivariate Normal distributions. In this hypothetical setup, each state has one Bivariate Normal distribution associated with it, however, since density functions don't return valid probability distributions, we attempt to make it valid by dividing the density function of each state by the total sum of all density functions (that is, $P[X^{(t)}|s_i] = \frac{pdf_i(X^{(t)})}{\sum_j pdf_j(X^{(t)})}$, where pdf_i is the distribution's density function). Intuitively, one might expect that, upon optimizing this model on some data, the distributions will move to cluster around the points, mimicking their pattern in 2D space (as one might see in a Gaussian mixture model), however, this is not the case. What indeed happens is that, due to normalization, there is no incentive for the mean value of the distribution to approximate the centroids of the clusters, as points get assigned higher probabilities with each state based solely on their *relative distance* from each of the means[1]. Consequently, the mean's of the distributions will often fly off into infinity, in opposite directions, which, while still ever-improving state-probability distributions, leads to nonsensical, and often ineffective models.

An informative solution to this problem is to simply drop the normalization term. Upon doing so, the Bivariate Normal Distributions can no longer move away from the centroids of the data without a significant penalty to the objective function. The only real consequence of this change is that the objective function of our Markov Model now no longer represents a true likelihood value, defined as the probability of the observed sequence having originated from the model. In this case, the change in semantics is analogous to how the probability density at some point differs semantically from the more conventional notion of event probability.

More generally, we can allow for the observation probability function to be rather flexible, so long as one is willing to discard the interpretability of the objective function being a true likelihood value. Aside from permitting the performance comparison between distinct probabilistic models, maintaining proper likelihood semantics is unnecessary for most applications of Markov Models such as prediction, error correction, simulation and classification. We summarize the minimal requirements for the array returned by the observation function *observationProb* needed to ensure successful optimization of a Markov Model using our optimization method:

- All items in the array must be non-negative: Since the array is being used in place of the state observation probability distribution ($P[X^{(t)}|s_i]$), its values multiply with those of the current state probabilities in order to update them (since, as

[1]In particular, points closer with respect to their *mahalanobis* distance d to the means of each state are assigned exponentially higher probabilities, as the Bivariate Normal distribution's density function is proportional to e^{-d^2}

seen in Equation 2.12, $P[s_i^{(t)}|X^{(0)}\dots X^{(t)}] \propto P[s_i^{(t)}|X^{(0)}\dots X^{(t-1)}]P[X^{(t)}|s_i])$. A negative value here would yield a nonsensical "negative probability"

- At least one value in the array must be non-zero: A fully zero array must be avoided, as it stops the model by permanently making all future state probability distributions zero (this is undefined behavior, as it creates a paradox between the assumption that the world is always in one of the model's states, and the observation that it isn't in any of them)

- There is no means for the optimizer to perversely increase the values of the array in order to infinitely increase the objective function: If there is no trade-off in the way parameters are used to compose the values of the array, then the optimizer may attempt to simply maximize these values, as they scale linearly with the objective function. This problem mainly expresses itself through eventual numerical stability problems, as values run towards infinity, but its aim to capitalize on the effects of the runaway phenomenon can also come at the cost of a coherent transition model which actually represents the data (that is, by ignoring the particularities of the data and simply aiming to maximize the array's values).

While the above restrictions do preclude the use of some techniques (such as non-normalized linear regression and relu activation functions, both of which can learn to become independently infinite), methods such as probability density and distribution functions, and soft-max and sigmoid activation functions are all feasible solutions (given reasonable assumptions about their implementation). Note that in the case of probability densities, so long as the data being analyzed indeed follows a continuous distribution (where the probability of any two points being in the exact same place is zero), there will always be a downside to the model assigning excessive probabilities to one region, as doing so implies in reducing it in another. In many cases, softmax and softmin can be used as a means to ensure parameters don't become infinitely large or small (i.e.: setting a minimum standard deviation in a Gaussian to deal with occasional point-distributions), though the model cannot rely exclusively on these limitations (as a poorly designed model will simply learn to saturate parameters at their limit values).

In conclusion, we have illustrated our mechanistic method for obtaining a Gradient Descent optimizer for Markov Models through the example of the Hidden Markov Model. We have detailed a few particularities pertaining to how to describe the model parameters in a way that ensures that the probability distributions being modeled are valid, and presented an extension to how the observation function can be designed that allows for it to be more flexibly used, at the cost of the interpretability of the objective function. The techniques detailed here can be carried out on other models through automatic differentiation, as indeed is the case for the models in Section 3.2, so long as an algorithm for computing the likelihood function can be devised (such as Algorithm 5 was for a HMM).

3.2 Proposed Models

3.2.1 Co-Optimized Visual Markov Model

In Section 3.1 we discussed how Gradient Descent optimization could be utilized to co-optimize more complex observation functions. Here, we propose a model which makes use of this property by extending the visual computation abilities of Convolutional Neural Networks to Hidden Markov Models.

Traditionally, computer graphics have not been a common area of application for Hidden Markov Models. The reason behind this lies in part due to the high dimensionality and structure of this kind of data, which make it difficult to establish a useful observation function. In particular, image data often has complicated properties such as invariance towards translation and rotation (e.g.: An orange in an image is still an orange if we move it about and rotate it), and a vast number of real valued data points (representing multi-channel colors over a wide grid of pixels). Defining a method which makes use of these properties while simultaneously being able to be co-optimized by traditional optimization techniques for Markov Models is nearly impossible (to the knowledge of this author, no such cases are known to exist), which consequently has lead to the adoption of two-stage solutions, where a model better equipped to deal with visual data is trained to extract visual features, which are then fed into a Markov Model. The development of the model to extract these visual features, however, is often complex and laborious, often negating any advantages one might have had in using a Markov Model in the first place.

Convolutional Neural Networks (CNNs), on the other hand, are especially designed for this kind of information, being able to take advantage of the structures inherent in images by defining its parameters in small, 2D windows, which are then reused over the entirety of the image. The use of shared weights has the effect of increasing the generalization ability of the model, as a feature developed for one part of the image get applied to its entirety. Furthermore, parameter reuse can reduce the total number of parameters, which in turn reduces the need for training data, and can help mitigate overfitting and excessive use of computational resources. CNNs, aren't, however, sequence models, and as such can not perform sequence-based tasks.

While video-data analysis would be obvious choice to demonstrate the effectiveness of combining these two classes of models, we will instead opt for static image analysis on account of its manageability and less stringent computational requirements. Nonetheless, the model we propose here can be extended to this form of data without too much difficulty.

To begin, a first step in defining any Markov Model is defining what the semantics of the states will be. In the case of image data, one possible approach – which aligns itself well with more traditional uses of Markov Models and visual data – is to define states as evidence of some event having occurred in the image as a whole (e.g.: a state which means that "the image contains an apple"). While this is a perfectly valid approach in many scenarios, we are more interested in how Markov Models can actually aid in image analysis. A more interesting model starts to emerge if, in addition to representing some feature, each state of our Markov Model also represents a position in the image (that is, each state represents an ($x \times y \times$ feature) 3-tuple). In this conception, a path through a set of states represents a collection of image features which were recognized in specific positions of the image.

We can improve this model by borrowing the concept of weight sharing from CNNs, and transpose the same transition models between states of equal features and relative positions (i.e.: the probability of transitioning from $s_{0,2,f_0} = (0 \times 3 \times f_0)$ to $s_{2,3,f_1}$ is equal to that from $s_{4,5,f_0}$ to $s_{6,6,f_1}$). This has the effect of granting position invariance to our model, as a path which represents some concept in one part of the image will also represent the same concept in another part of it, should the path be transposed.

Using shared weights helps to reduce the total number of parameters in the model, but we are still left with N^4F^2 transitions (where the image is assumed to be $N \times N$, with a total of F features, making a total of N^2F states, all of which connect to each other). To address this issue we simplify our transition model and define that instead of having one parameter per transition, transitions are defined jointly, with each feature being associated with a Bivariate Normal projected towards some relative position in the image. Now, for example, after recognizing some feature at $s_{0,2,f_0}$, we would define our new distribution over S as being a normalized Gaussian centered around $x = 0 + \mu_{x,f0}$.

While there are many uses for such a model, we employ it here on the task of classification. By constructing multiple transition models, each of which has presumably learned to recognize different relevant features within the image (e.g.: one model has transition models adapted to finding and following the curvature of a banana, whereas another checks for the shape of an apple), we can compare the likelihoods of the different models on an unlabeled image, and come up with a measure of how well the image represents the concept codified by each transition model, relative to the others (picking, for example, the highest likelihood as our guess).

We now formalize this model in a step by step way, defining the associated equations as we go, so as to remove any ambiguities in our description:

1. The convolutional and pooling layers of the CNN are run, resulting in a 3D matrix of values, one for each feature map and position;
2. The values for the feature maps are normalized, forming a state probability distribution (Equation 3.6);
3. The initial state probabilities and transition models are computed from our Gaussian parameters (Equation 3.7);
4. For some number of iterations, we first apply the observation probabilities to the state probabilities (which begins as the initial state probability distribution), and then propagate them using the transition model we defined (Equation 3.8)
5. Having iterated a few times, the total likelihood of the model is computed by summing up the probabilities left in all states (Equation 3.9);

$$\begin{aligned} c_{i,j,f} &: \text{The CNN's output for feature } f \text{ at position } (i,j) \\ o_{i,j,f} &= \frac{c_{x,y,f}}{\sum_{\forall x',y',f'} c_{x',y',f'}} \end{aligned} \tag{3.6}$$

$$\begin{aligned} g_{[\mu_x,\mu_y,\sigma_x,\sigma_y,\rho,N]}(i,j) &= e^{-(1-\rho^2)^{-1}\left(\frac{(x-\mu_x)^2}{\sigma_x^2}-\frac{2\rho(x-\mu_x)(y-\mu_y)}{\sigma_x^2}+\frac{(y-\mu_y)^2}{\sigma_y^2}\right)} \mid [x,y] = \frac{2[i,j]-N}{N} \\ G_{[\mu_x,\mu_y,\sigma_x,\sigma_y,\rho,N]}(i,j) &= \frac{g_{[\mu_x,\mu_y,\sigma_x,\sigma_y,\rho,N]}(i,j)}{\sum_{\forall i,j} g_{[\mu_x,\mu_y,\sigma_x,\sigma_y,\rho,N]}(i,j)} \\ P[s^{(0)}_{i,j,f}] &= G_{[\mu_{f0,x},\mu_{f0,y},\sigma_{f0,x},\sigma_{f0,y},\rho_{f0},N]}(i,j) \\ P[s^{(t+1)}_{k,l,g}|s^{(t)}_{i,j,f}] &= \left(\frac{1}{N}\right) G_{[\mu_{f,x},\mu_{f,y},\sigma_{f,x},\sigma_{f,y},\rho_f,N]}(k-i,l-j) \end{aligned} \tag{3.7}$$

$$\begin{aligned} P[s^{(0)}_{i,j,f}|\text{obs}] &= P[s^{(0)}_{i,j,f}] \\ P[s^{(t+1)}_{i,j,f}|\text{obs}] &= \sum_{k,l,g} P[s^{(t+1)}_{i,j,f}|s^{(t)}_{k,l,g}]P[s^{(t)}_{k,l,g}|\text{obs}]o_{k,l,g} \end{aligned} \tag{3.8}$$

$$L^{(t)} = \sum_{\forall k,l,g} P[s_{i,j,f}^{(t)}|\text{obs}] \tag{3.9}$$

Note that the model can be extended to deal with video data by simply changing $o_{i,j,f}$ in Equation 3.6 to be frame specific (which in turn requires running the convolutional layers on each frame of video), and then using the appropriate $o_{i,j,f}^{(t)}$ in the conditional probability expression (Equation 3.8). Doing so, however, requires extensive computational resources.

3.2.2 Stacked Markov Model

In contrast to Section 3.2.1, where we propose a model that illustrates the versatility of a Markov Model's observation functions when coupled with Gradient Descent, here we aim to illustrate that this kind of methodology also allows us to be more creative with the types of Markov Models we can design, which in turn allows us to address some of the classical hurdles faced by more traditional models such as HMMs. We will start by illustrating a scenario, and will then propose a novel Markov Model which addresses some of the problems that a more traditional approach encounters.

Consider the task of developing a Markov Chain for the half-hourly traffic of a single street. Suppose that we are interested in knowing only if there is either light traffic, or heavy traffic, and that a typical day looks something like Figure 3.1. For the sake of our analysis, we decide to model this scenario with a Markov Chain, which isn't unreasonable, as traffic generally follows the Markov Principle (in that future traffic is usually conditionally independent of previous traffic, given the traffic in the present), has patterns (such as times of day which usually have more traffic), but is ultimately stochastic (as there can be variations in daily pattern, such as accidents or a road blockage).

A first approach to modeling this problem would be to assign each scenario with a single state in the model, with the 4 transition probabilities representing the street remaining in light or heavy traffic, or switching between these two states. We illustrate this model in Figure 3.2. While this approach works, and can be used for many desirable things (such as predicting what the traffic will be in an hour), it doesn't particularly grasp the nuances of traffic flow, and as such has difficulties predicting too far into the future. Furthermore, a simulation of this Markov Chain yields outputs such as Figure 3.3, which aren't particularly realistic, completely disregarding facts such as the traffic spikes at 8 AM and 6 PM.

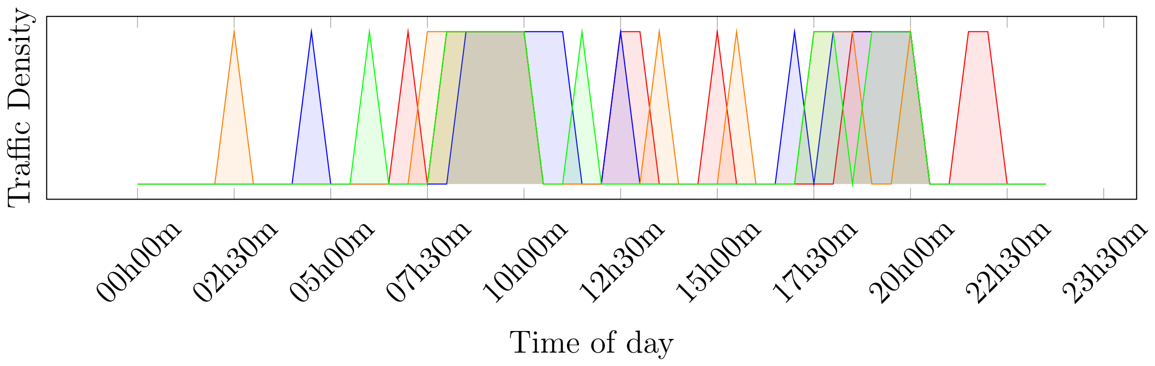


Figure 3.1. Examples of typical traffic for a weekday on our hypothetical street. Traffic is densest at 8AM and 6PM and surrounding times, with occasional spikes outside this window.

$$\begin{tikzcd} L \arrow[loop above, "0.86"] \arrow[r, bend left, "0.14"] & H \arrow[loop above, "0.67"] \arrow[l, bend left, "0.33"] \end{tikzcd}$$

Figure 3.2. Markov Chain which aims to model the weekday traffic of Figure 3.1.

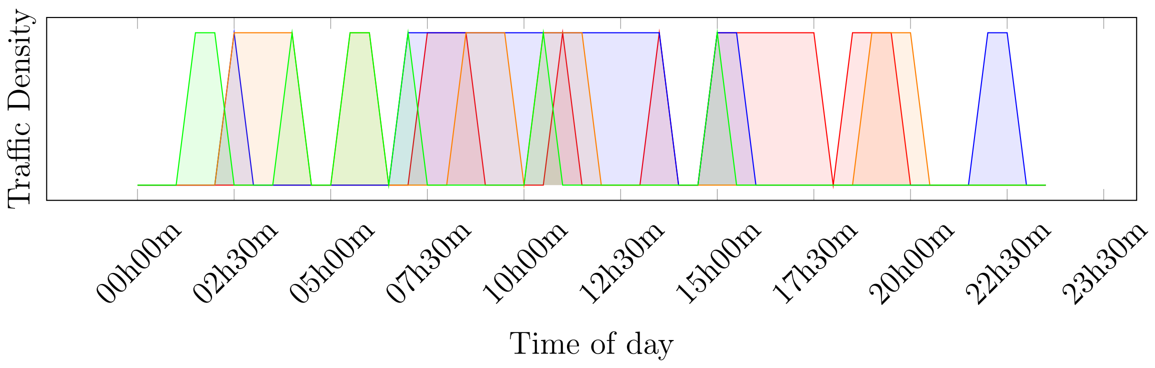


Figure 3.3. Traffic pattern sampled from the simple traffic model shown in Figure 3.2

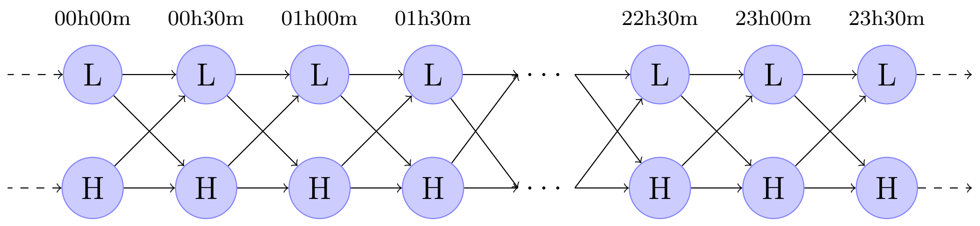


Figure 3.4. Ring Markov Chain for traffic modeling; We represent only two observations, *low-* and *heavy-traffic*, but define one state for each half-hour mark in the day in a ring (not shown: the states at 23h30 connect to those at 00h00m).

The reason our simple Markov Chain failed to capture the details of traffic on the street is the fact that Markov Models have no memory, or, more precisely, they store no data beyond what can be inferred from the state they're currently in. A two-state system such as we proposed stores only one bit of data, and that's not enough to remember the time of day. More states, however, allow for more data, which leads us to our second approach: Instead of only designating low and high traffic, we add states representing low and high traffic at specific times of day, which we serially link up in a ring, such as shown in Figure 3.4.

The ring model captures the daily traffic patterns reasonably well, and, as needed, allows for variations due to anomalies. This would be perfect if it were not for Saturday and Sunday. Suppose that weekend traffic on this street resembles that of Figure 3.5. On these days our ring model for weekdays fails miserably. A naive solution would be to simply expand our ring with a state for every half hour in the week – instead of every half hour of the day as was previously the case – but this quickly starts becoming nonviable, as the number of states explodes, going from 96 to 672. Even more distressing would be to try and model variations which occur on the scale of months, such as children from a hypothetical school on this street being on summer vacation (and thus reducing traffic spikes). Maintaining a large number of states is not only computationally expensive, but it can also be expensive in terms of data, as an optimizer attempting to obtain the transition probabilities of a given state would be forced to rely solely on the data from its specific time of day and moment within the week, being unable to use examples from other days to aid its understanding.

An insight which can help solve this problem is the idea of abstraction. If we construct two higher order events, one for *weekday-traffic* and another for *weekend-traffic*, each an instance of the aforementioned ring model, then capturing the variations of daily traffic patterns throughout the week would be as picking the correct ring model between the *weekend-* and *weekday-traffic* models. We can ensure we always make the correct choice by creating yet another ring model, only this time with 14 states (one for

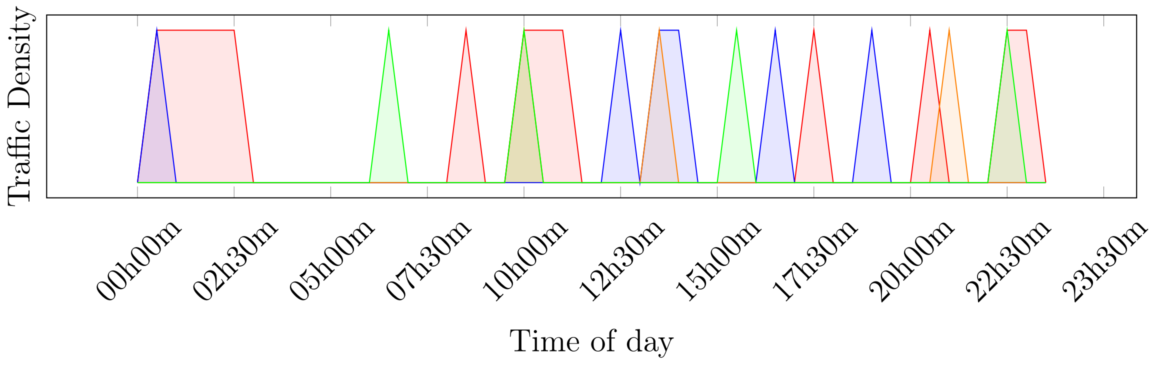


Figure 3.5. Examples of typical traffic for a weekend on our hypothetical street. Traffic is rarely dense, but when it is, it is more likely to be so in the late morning and early afternoon.

each of *weekend-* and *weekday-traffic* models we wish to chose from, on each day of the week). In fact, what if instead of a single model for our days of the week we also created models for other kinds of weeks, for instance, *vacation-week* and *long-weekend-week*, and then chose between them using an even higher order markov chain; This leads us to a recursive definition, from which we can begin formalizing our model.

We refer to this collection of layered Markov Chains and transition models as a Stacked Markov Model (SMM). In the SMM, each state is associated with a unique transition model for the Markov Chain in the layer below it (just as how the *weekday-traffic* state defines a set of transitions between a series of *low-* and *heavy-traffic* states). The active state in layer $l + 1$ defines the transition model for layer l, however it does not uniquely define the transition models for layers $l - 1$ and lower, as these depend only on the state in layer l (which is shared by all sates in layer $l + 1$; In our example, this is analogous to how *vacation-week* alters the transition between *weekday-* and *weekend-traffic*, but doesn't redefine the transitions these states themselves impose on *low-* and *heavy-traffic*). As a consequence of our structure, each layer l represents a wider timescale than the layer beneath it, with a single iteration of layer l requiring some number of iterations of layer $l - 1$ (which we will refer to as Δ_l). We define more formal notation for this model in Equation 3.10.

$$
\begin{aligned}
\text{layer } l &: \text{Set of states } S_l \text{ used in all Markov Chains of layer } l \\
s_{l.i}^{(t_l)} &: \text{State } i \text{ of layer } l \text{ is active at iteration } t_l \text{ of layer } l \\
M_{l.i} &: \text{Set of transition models used for layer } l-1 \text{ when state } s_{l.i} \text{ is active} \\
\Delta_l &: \text{Number of iterations layer } l \text{ runs for an iteration of layer } l+1 \\
t_l = t_{l+1}\Delta_l &: \text{Relationships between iterations of adjacent layers}
\end{aligned}
\tag{3.10}
$$

At the base of the SMM the observation model semantics remain the same as for a Markov Chain or a Hidden Markov Model (following the guidelines put forth in Section 3.1). For a higher layer l, we recursively define that the probability of observing some state $s_{l.i}^{(t_l)}$ is the likelihood of the its transition model $M_{l.i}$ over the observations of layer $l-1$. We express this mathematically in Equation 3.11.

$$
\begin{aligned}
obs(s_{0.i}^{(t_0)}) &= P[s_{0.i}^{(t_0)}|X^{(t_0)}] \\
obs(s_{l.i}^{(t_l)}) &= P[obs(S_{l-1}^{(t_l\Delta_l)})\dots obs(S_{l-1}^{((t_l+1)\Delta_l-1)})|M_{l.i}]
\end{aligned}
\tag{3.11}
$$

Note that the set of observation functions $obs(S_l)$ may not represent a valid probability distribution in layers 1 and above, as they don't add up to 1. This isn't as much of a problem as it might seem, however, as the distributions obey the guidelines set forth in Section 3.1, and normalization can be used here without side effects.

The final component needed to define the model is how the initial probabilities are defined (in our example, this would appear, for example, as a choice as to how we would define traffic at the start of a *weeday-traffic* day, or what kind of day we expect to start a *vacation-week*). The simplest approach here would be to specify some initial probability vector $P[S_l^{(0)}]$ for each transition model $M_{l.i}$, and then reinitialize the state-probability vector every time each Markov Chain initiates the first of its Δ_l iterations (for example, defining that *weekday-traffic* weeks start out at midnight with a probability of 0.05 of *heavy-traffic*, and 0.95 for *low-traffic*). This model has some computational advantages when computing the model's state probabilities given data, such as the fact that, in this approach, the data X can be split into windows of size Δ_0, whose likelihoods are independently computed for each transition model given by layer 1, yielding $obs(S_1^{(t)})\ \forall t$, which in turn could be windowed into blocks of size Δ_1, and so on – the independence in this case allowing for parallelism.

While this is a solution that works, we can do better: Suppose, for example, that while inferring the state distributions over some data, some state has a near 1 probability at the end of a set of Δ_l iterations ($P[s_{l.i}^{(t_l)}|X] \approx 1$), and that all transition models $M_{l+1.j}\ \forall j$ dictate that the transition to some state $s_{l.k}$ occurs with near certainty ($P[s_{l.k}^{(t+1)}|s_{l.i}^{(t)}] \approx 1$). In this case, it makes no sense to completely reset the probability distribution $P[S_l^{(t_l+1)}]$, which would completely ignores the fact that $s_{l.k}$ is the most likely next event. Instead, we can define a single initial probability distribution for each layer, and then attempt to create an inference rule which defines the new initial probability distribution at the start of each iteration. Equations 3.12 and 3.13 illustrate the aforementioned "simplest approach" and one of the inference rules we utilized.

$$P_{l+1.i}^{(0)} : \text{Distribution for layer } l \text{ whenever } t_l \bmod \Delta_l = 0.$$

$$P[s_{l.i}^{(t_l+1)}|S_l^{(t_l)}, s_{l+1.m}^{(t_l+1)}] = \begin{cases} P_{l+1.i}^{(0)}[i], & \text{if } t_l + 1 \mod \Delta_l = 0 \\ \sum_j P[s_{l.j}^{(t_l)}|s_{l+1.m}^{(t_l+1)}]M_{l+1.m}^{(t_l)}[j,i] & \text{otherwise} \end{cases} \tag{3.12}$$

$$P[s_{l.i}^{(t_l+1)}|S_l^{(t_l)}, s_{l+1.m}^{(t_l+1)}] = \begin{cases} \sum_n P[s_{l+1.n}^{(t_l+1)}] \sum_j P[s_{l.j}^{(t_l)}|s_{l+1.m}^{(t_l+1)}]M_{l+1.n}^{(t_l)}[j,i], & \text{if } t_l + 1 \mod \Delta_l = 0 \\ \sum_j P[s_{l.j}^{(t_l)}]M_{l+1.m}^{(t_l)}[j,i] & \text{otherwise} \end{cases} \tag{3.13}$$

At this point we have finished defining the intuition and formalism behind the SMM. However, before moving on to its uses and parametrization, a qualitative example of the model's architecture in relation to an example may be useful to for the reader to solidify some of the finer details of the model. Let us, therefore, recall our traffic modeling problem from before. In this scenario, the SMM we construct might hypothetically have 3 layers:

- layer 0 has two states, representing $S_0 = \{\text{heavy-traffic}, \text{light-traffic}\}$. Being the base layer, it defines no transition models, and every iteration of the Markov Chain at this layer is associated with a single observation of traffic;
- layer 1 is composed of $S_1 = \{\text{work-day}, \text{non-work-day}\}$, each of which defines a unique set of $\Delta_0 = 48$ transition models between the two states of layer 0 (*heavy-* and *light-traffic*, forming a total of 96 transition models). We choose $\Delta_0 = 48$ on account of this being the number of half-hours in a day, which allows us to split our traffic observations into windows of 1 day, and use a unique transition model for each state of layer 1 and each half hour of the day.

- layer 2 is composed of $S_2 = \{\text{work-week, vacation-week, early-weekend, extended-weekend}\}$, and, since each state represents a week, and each iteration of layer 1 spans a day, a week is then given by $\Delta_1 = 7$ transition models, instantiated by each of its 4 states.

Using this example we can also illustrate the reasoning behind the observation function of each state being the likelihood of its transition model over its inferior layer (Equation 3.11). Suppose we send to our algorithm the traffic pattern of a normal week, starting on a Monday. For the first 5 days, $obs(S_1)$ will be much greater for the *work-day* state than for the *non-work-day* state (as it should be), a pattern which is reversed on the last 2 days. This happens because $obs(S_1)$ is computed using a likelihood calculation, which, as discussed in Section 2.1.5, is exponentially greater when a model adequately represents its inputs. Now, when computing the observation function for layer 2, we see that states such as *vacation-week* and *early-weekend* all propose transition models with low likelihood, given that their transition models expect the observations of layer 1 to be in some order with more than two *non-work-day* observations. Again, the correct state, *work-week*, has the highest likelihood, and is the clear winner given the observations, thus demonstrating that our observation function is able to capture our intended abstractions.

In the remainder of this section, we will exemplify how the SMM retains some of the desirable properties we associate with Markov Models, such as simulation (Section 3.2.2.1) and prediction (Section 3.2.2.2), before we move on to its likelihood computation and parametric optimization (Section 3.2.2.3), which allows us to use the model in practice. Finally, we end our discussion on the SMM with a slight extension, which permits it to deal with multi-dimensional data (Section 3.2.2.4).

3.2.2.1 Simulation

In Section 2.1.4 we discussed the simulation of Markov Chains and Hidden Markov Models. In this section, we discuss the relatively straightforward extension of these techniques to the SMM.

As outlined in the introduction to this section, the SMM operates as a set of stacked Markov Chains, where the states of layer $l+1$ define a transition model for the states at layer l. We can look at this model from a generative perspective, where whenever a state $s_{l+1.i}$ is selected at some layer $l+1$, the Markov Chain of layer l is recursively initialized and iterated using the associated transition model $M_{l+1.i}$ for Δ_l iterations. We detail this method in Algorithm 7.

```
function simulateSMM(layer, $S_{\text{parent}}$)
    $S_{\text{curr}} \leftarrow S_{\text{PARENT}}$.sample0()
    for all $i$ from 0 to $\Delta_{\text{layer}}$ do
        if layer $> 0$ then
            simulateSMM(layer$-1$, $S_{\text{curr}}$)
        else
            emitObservation($S_{\text{curr}}$)
        end if
        $S_{\text{curr}} \leftarrow S_{\text{PARENT}}$.TransitionModel.sample($s_{\text{curr}}$)
    end for
end function
```

Algorithm 7: Recursively simulates an SMM. To start the process, a state must be picked from the top layer. Arbitrarily large sequences can be achieved by if we consider $\Delta_L = \infty$ (where L is the number of layers in the model)

The simulation algorithm (Algorithm 7) expresses the initial probability distribution through the `sample0` function, whose semantics depend on the specific inference rules used. The probabilistic inference rules (such as Equation 3.12 and Equation 3.13) can be made into a sampling function by simply applying these rules at the point described in the algorithm, and then sampling uniformly from the resulting $P[S_l^{(t)}]$ distribution, using the obtained value as the new starting state.

3.2.2.2 Prediction

Similar to Markov Chains, a SMM can also be used to make probabilistic predictions arbitrarily far into the future. The mechanism whereby this is achieved is similar to that of a HMM, with the main complicating factor being the fact that now we are dealing with multiple layers of Markov Chains, each of which has its own probability distribution.

Before delving into the mathematics for prediction within this model, it is insightful to recall the relationship between simulation and prediction. The simulation of an SMM, as discussed in Section 3.2.2.1, is a stochastic, generative process which generates samples according to the expected outcome of a starting distribution. This process could in theory be used to make increasingly accurate predictions by making numerous simulations, and then compiling statistics on the resulting data. Naturally, this method is inefficient, but it does set the scene for what we might expect to see from the mathematics: First, the hierarchical nature of the generative process is entirely top-down, which is to say that the choice in states at upper levels is not influenced by the choice in states at lower levels (rather the reverse is true); Second, that the sampling process for a layer $l-1$ does not depend on the state that is active in layer $l+1$, so long as some state in layer l is chosen.

With these considerations in mind, we can confidently deduce the equations for prediction using our model, as is done in Equation 3.14, where we assume that the probability distributions are known at each layer at their respective iterations t_l, for which we compute the distribution of the states in the model for the next time step.

$$\begin{aligned}
P[s_{l.i}^{(t_l+1)}|s_{l+1.m}^{(t_{l+1})}] &= \begin{cases} \text{inference rule,} & \text{if } t_l+1 \mod \Delta_l = 0 \\ \sum_j P[s_{l.i}^{(t_l+1)}|s_{l.j}^{(t_l)}, s_{l+1.m}^{(t_{l+1})}]P[s_{l.j}^{(t_l)}|s_{l+1.m}^{(t_{l+1})}] & \text{otherwise} \end{cases} \\
P[s_{L.i}^{(t_L+1)}] &= \sum_j P[s_{L.j}^{(t_L)}]P[s_{L.i}|s_{L.j}] \quad \text{(Markov Chain prediction for top layer } L) \\
P[s_{l.i}^{(t_l+1)}] &= \sum_m P[s_{l.i}^{(t_l+1)}|s_{l+1.m}^{(t_{l+1})}]P[s_{l+1.m}^{(t_{l+1})}]
\end{aligned} \tag{3.14}$$

Note that with Equation 3.14 we can compute predictions $P[s_{l.i}^{(t_l+1)}]$ for any layer (including the observable layer), recursively in function of the predictions of higher layers and the previous conditional state probability distribution (with exception of the top layer, which has no higher layer, and is defined in the same way as a Markov Chain). The conditional state probability $P[s_{l.i}^{(t_l+1)}|s_{l+1.m}^{(t_{l+1})}]$ must initially be given as inputs, however future values can be computed recursively through the given expression. Should the observation function of the base layer be indirect (such as in a HMM), then in order to predict actual observations an additional step would be needed, identical to what is already done in a HMM.

3.2.2.3 Likelihood and Optimization

As discussed in Sections 2.1.5 and 3.1, likelihood encapsulates the stochastic nature of Markov Models, and as such is both appropriate and sufficient for optimizing this class of models through Gradient Descent. We aim here, therefore, to express the way in which likelihood is calculated for the SMM.

By construction, the SMM was designed as a set of layered Markov Chains, each acting like a HMM which receives the likelihood of a window of observations of the lower adjacent layer as its observation function. This design makes the likelihood calculation relatively straightforward, since we already know how to calculate the likelihood for a Hidden Markov Model (see Section 2.1.5). We begin in Equation 3.15 by breaking down the likelihood function into a set of conditional probabilities:

$$\begin{aligned}
L &= P[x^{(0)}, x^{(1)}, \dots, x^{(T)}|Model] \\
&= P[x^{(0)}]P[x^{(1)}|x^{(0)}]P[x^{(2)}|x^{(0)}, x^{(1)}] \dots P[x^{(Y)}|x^{(0)}, \dots, x^{(T-1)}] \\
&= P[x^{(0)}] \prod_{t=1}^{T} P[x^{(i)}|x^{(0)} \dots x^{(t-1)}] \\
&= \left(\sum_i P[x^{(0)}|s_{0.i}^{(0)}]P[s_{0.i}^{(0)}] \right) \prod_{t=1}^{T} \left(\sum_i P[x^{(i)}|s_{0.i}^{(t)}, x^{(0)} \dots x^{(t-1)}]P[s_{0.i}^{(t)}|x^{(0)} \dots x^{(t-1)}] \right) \\
&= \left(\sum_i P[x^{(0)}|s_{0.i}^{(0)}]P[s_{0.i}^{(0)}] \right) \prod_{t=1}^{T} \left(\sum_i P[x^{(i)}|s_{0.i}^{(t)}]P[s_{0.i}^{(t)}|x^{(0)} \dots x^{(t-1)}] \right)
\end{aligned} \tag{3.15}$$

Next, in Equation 3.16, we recursively define the conditional probabilities in a manner similar to what was done for Hidden Markov Models (Equation 2.12). This equation is also very similar to the SMM's prediction equation (Equation 3.14), differing only in that the state probability distribution is weighted by the observation function at each iteration (again, this being by design).

$$\begin{aligned}
o_{l.i}^{(t)} &= \begin{cases} P[x^{(t)}|s_{0.i}^{(t)}], & \text{if } l = 0 \\ P[o_{l-1}^{\Delta_{l-1}t_l} \dots o_{l-1}^{((\Delta_{l-1}+1)t_l-1)}|s_{l.i}^{(t)}] & \text{otherwise} \end{cases} \\
P[s_{l.i}^{(t_l+1)}|s_{l+1.m}^{(t_{l+1})}, \{x^{(0 \dots t_l)}\}] &= \begin{cases} \text{inference rule}, & \text{if } t_l + 1 \mod \Delta_l = 0 \\ \sum_j P[s_{l.i}^{(t_l+1)}|s_{l.j}^{(t_l)}, s_{l+1.m}^{(t_{l+1})}]P[s_{l.j}^{(t_l)}|s_{l+1.m}^{(t_{l+1})}, \{x^{(0 \dots t_l)}\}] & \text{otherwise} \end{cases} \\
P[s_{l.j}^{(t_l)}|s_{l+1.m}^{(t_{l+1})}, \{x^{(0 \dots t_l)}\}] &= \frac{o_{l.j}^{(t_l)} P[s_{l.j}^{(t_l)}|s_{l+1.m}^{(t_{l+1})}, \{x^{(0 \dots t_l-1)}\}]}{\sum_k o_{l.k}^{(t_l)} P[s_{l.j}^{(t_l)}|s_{l+1.m}^{(t_{l+1})}, \{x^{(0 \dots t_l-1)}\}]} \\
P[s_{l.i}^{(t_l+1)}|\{x^{(0 \dots t_l-1)}\}] &= \sum_m P[s_{l.i}^{(t_l+1)}|s_{l+1.m}^{(t_{l+1})}, \{x^{(0 \dots t_l-1)}\}]P[s_{l+1.m}^{(t_{l+1})}, \{x^{(0 \dots t_l-1)}\}]
\end{aligned} \tag{3.16}$$

Using Equations 3.15 and 3.16, we can construct an algorithm that computes the likelihood of a given set of inputs with respect to the model, for which we can consequently use automatic Gradient Descent to come up with an optimizer (this being a clear case where automatic Gradient Descent is needed, as computing the gradients by hand can be a monumental task for a model such as this). We show the complete code for the SMM in Appendix A, as written in our Domain Specific Language.

3.2.2.4 Multidimensional Data

So far, the justification for using a SMM has been on the basis that it could drastically reduce the number of states required to represent context information. A small extension can be made to the model, however, that would also allow it to have a similar effect on multidimensional data.

Consider if, in our original street traffic example, we now have a set of light/heavy traffic histograms for a number of streets, and we now wish to create a conjoint *weekday-traffic* model for all of them. One approach would be to separate each histogram into its own model, and then process them independently. In contexts where the traffic on these roads are completely unrelated this would indeed be the correct solution, however, it is likely that traffic on one street will affect another one (e.g.: build-up on a main avenue might cause traffic to also build up on the in-flowing roads).

A naive approach to solving this would be to simply join all the possible states of each street into a single set of unified states, each representing some configuration of light and heavy traffic for each street in the set. The trouble here is that with n streets we end up with 2^n states, which not only becomes computationally intractable for Markov Chain computations, but also requires impossibly large amounts of data in order to infer the transitions within the model.

While having this large number of states may be needed in extreme cases, this is rare in practice. Consider, for example, the task of modeling a neighborhood comprised of 2 main avenues and some 15 streets. The naive approach would have us create over 131 thousand states, each with a full set of probabilities to each other state. This would allow for each traffic pattern to have radically different effects on traffic everywhere else, however, in practice, phenomenon such as this are more local. Our hypothetical neighborhood might only have 4 main behaviors, representing each of the two avenues being either full or free (when each of the avenues is full, for example, traffic on some subset of the streets associated with the full avenue might also get backed up).

We can use the SMM to solve this problem by abstracting the notion that a certain set of streets all tend to have heavy traffic together in a higher level state, in effect creating a mechanism for storing and sharing context information between each street's models. In our example scenario, this could be accomplished by creating a model such as the following:

- layer 0 is a collection of sets of states, where each set is comprised of $S_0 = \{\text{heavy-traffic}, \text{light-traffic}\}$, much like before, and there is one set of states for each street we are modeling

- layer 1 is a set of states, representing *all-ok*, *ave-1-blocked*, *ave-2-blocked*, and *both-blocked*, spanning $\Delta_0 = 4$ steps of layer 0 (2 hours in real time). Each state of layer 1 defines a unique transition model for each of the sets of states and times of layer 0, forming a total of 4(layer 1 states) $\times$ $4(\Delta_0)$ transition models.
- layer 2 is equivalent to layer 1 in the previous example, changing $\Delta_1 = 12$.
- layer 3 is identical to layer 2 in the previous example.

Mathematically, the main change this extension causes is in defining the observation function for the layers which have more than one child (whose states, consequently, now each define a different transition model for each child layer). We define the new observation function in Equation 3.17.

$$o_{l.i}^{(t)} = \begin{cases} P[x^{(t)}|s_{0.i}^{(t)}], & \text{if } l = 0 \\ \prod_{c\in\text{Children}} P[o_c^{\Delta_c t_l} \dots o_c^{((\Delta_c+1)t_l-1)}|s_{l.i}^{(t)}] & \text{otherwise} \end{cases} \tag{3.17}$$

The choice in defining the observation function in the manner described in Equation 3.17 as the product of the likelihoods on all the child layer's likelihoods is on account of the independence of the children's series, which is especially evident when looking at the model from a generative perspective. In this perspective, we define that higher level states define the order in which patterns of states at lower layers will occur, which in turn means that if multiple series are to be generated, they are done so independently of one another, disregarding what was done in any of the other series other than their direct parents. This has the consequence of making the probability of their joint observation also independent, a fact which we exemplify for a layer with two children in Equation 3.18.

$$\begin{aligned} o_a^{(t_l)} &: \text{Observations of layer } l\text{'s child layer } a \text{ from } \Delta_a t_l \text{ for } \Delta_a \text{ iterations} \\ o_b^{(t_l)} &: \text{Observations of layer } l\text{'s child layer } b \text{ from } \Delta_b t_l \text{ for } \Delta_b \text{ iterations} \\ o_a^{(t_l)} \perp\!\!\!\perp o_b^{(t_l)} | s_{l.i}^{(t_l)} &: \text{(by the independence of the generation method of each series)} \\ o_{l.i}^{(t_l)} &= P[o_a^{(t_l)}, o_b^{(t_l)} | s_{l.i}^{(t_l)}] \\ &= P[o_a^{(t_l)} | s_{l.i}^{(t_l)}] P[o_b^{(t_l)} | s_{l.i}^{(t_l)}] \end{aligned} \tag{3.18}$$

Finally, it is interesting to note that, while in our traffic example we defined a set of series for each street with equal Δ values and heights, completely different data sources with different scales can, in practice, be merged in this manner, with the model forming a tree of layers – a rather flexible design.

Chapter 4

Experiments

In this chapter we describe the experiments and results which we use to defend the hypotheses of our thesis. These are, namely, that Gradient Descent optimization is an effective optimizer for Markov Models, and that its use allows for a greater flexibility and ease in how a user can define and use discrete Markov Models. Consequently, our experiment design aims to define objective and subjective metrics for effectiveness and flexibility, and then measure these values in relevant scenarios.

The remainder of this chapter is structured as follows: In Section 4.1 we explore the use of Gradient Descent optimization on Markov Models, specifically, Hidden Markov Models. Our goal here is to demonstrate that, while somewhat more computationally expensive, Gradient Descent optimization can be an effective tool for optimizing this class of models when compared to more traditional methods; In Section 4.2 we propose a digit classification task using the model we introduced in Section 3.2.1. The aim of this section is to demonstrate the flexibility of the observation function, a Convolutional Neural Network in this case; Finally, Section 4.3 explores the Stacked Markov Model, proposed in Section 3.2.2 in the context of Temperature, CPU and Memory histogram modeling, aiming to illustrate flexibility in defining transition models and state semantics using our methodology.

4.1 Effectiveness of Gradient Descent Optimization on Markov Models

Central to our thesis is the notion that Discrete Markov Models can be effectively optimized using Gradient Descent Optimization. This claim is not trivial, since Gradient Descent optimization has several downsides, such as slow convergence rates, elevated computational complexity, and convergence to local minima (all previously discussed in Section 2.2). Since we cannot obtain a comparative measure of performance in a vacuum, we present a set of experiments which aim to demonstrate the effectiveness of the Gradient Descent optimizer on Hidden Markov Models in particular, for which the Baum Welch algorithm has been the well established state of the art optimizer for decades.

In order to guide our analysis, we pose the following research questions:

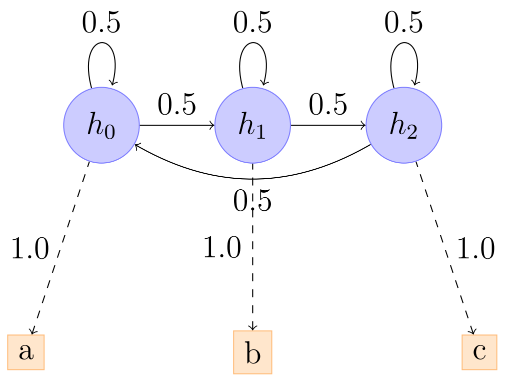
- **Can Gradient Descent optimization obtain an adequate Markov Model from data?**: A basic requirement for the validity of our hypotheses is that the models generated adequately represent the data.

- **How does Gradient Descent optimization compare to more conventional methods?**: While it is not our goal to exceed the performance of optimizers specialized on HMMs, it is useful to understand how it compares to more traditional approaches in order to justify its use.

We support our claims in these matters by comparing the likelihoods obtained by our model with those obtained by the Baum Welch optimizer, the optimal solution, and randomized Hidden Markov Models.

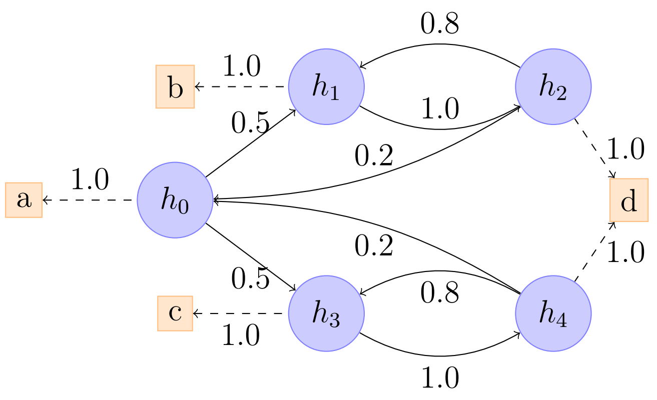
4.1.1 Setup

The task set forth in these experiments is to construct the best possible HMM given a sequence of inputs. In order to eliminate questions related to the adequacy of these sequences for this class of models (such as whether the data indeed follows the Markov Property), we synthesize our own data through the simulation of HMMs of our choosing. We chose 5 unique models, each of which represents a different set of difficulties, which we illustrate and detail in Figures 4.1 - 4.4 and Table 4.1.



Example: `abbccccabcabbcccaabccaabcaaaaaabccaaaabbbbbbcccaa`

Figure 4.1. Simple Markov Chain: Here, we have a Hidden Markov Model with a very simple transition and emission model. It is a rather simple task for an optimizer, since there are no ambiguities on the outputs (each state has a single and unique emission, `a`, `b` or `c`), and all transitions are equally likely (meaning that even with a relatively small amount of data there will likely be sufficient examples of each transition being taken).



Example: `abdabdabdcddacdabdcddcddcddcddcddcddcddcddcdacddc`

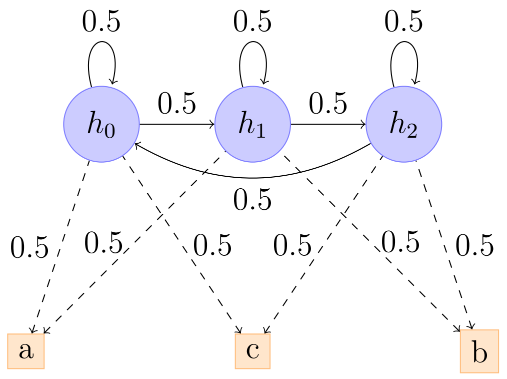
Figure 4.2. Ambiguous Markov Chain: In this model we introduce a small amount of ambiguity in the observation of `d`, which is shared by two states. A local minima in this scenario is to merge h_2 and h_4, but this comes with the penalty of poorly estimating the occurrence of `b`'s and `c`'s. Local minimas occur in scenarios such as this on account of the random initialization of parameters almost always creating some state which has a higher bias towards `d` than the others. If this bias is large enough, the model will deduce that it can best increase its likelihood over the input by using this state whenever `d` appears. As it gets used more, its transitions become even more optimized, increasing its attractiveness towards alternatives, at which point it becomes very unlikely that the model will allocate some other state that splits the merged value, as it hasn't undergone the same level of improvement, and is thus worse in every way.

	Dst									
Src	h_0	h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8	h_9
h_0	0.000	0.103	0.035	0.148	0.107	0.088	0.064	0.164	0.151	0.140
h_1	0.115	0.046	0.052	0.313	0.141	0.037	0.001	0.002	0.119	0.174
h_2	0.008	0.092	0.119	0.122	0.079	0.046	0.133	0.110	0.145	0.146
h_3	0.149	0.176	0.177	0.108	0.069	0.047	0.052	0.148	0.004	0.070
h_4	0.095	0.112	0.136	0.118	0.078	0.033	0.120	0.076	0.074	0.158
h_5	0.076	0.031	0.047	0.089	0.169	0.109	0.208	0.158	0.072	0.041
h_6	0.027	0.173	0.132	0.058	0.081	0.013	0.185	0.130	0.029	0.172
h_7	0.096	0.139	0.077	0.053	0.108	0.129	0.144	0.137	0.090	0.027
h_8	0.172	0.167	0.099	0.062	0.085	0.068	0.153	0.057	0.082	0.055
h_9	0.079	0.100	0.177	0.005	0.198	0.114	0.010	0.106	0.038	0.173

	Emission						
Src	`a`	`b`	`c`	`d`	`e`	`f`	`g`
h_0	0.065	0.322	0.266	0.192	0.113	0.005	0.037
h_1	0.167	0.176	0.177	0.048	0.194	0.132	0.106
h_2	0.166	0.044	0.143	0.072	0.266	0.064	0.245
h_3	0.040	0.294	0.024	0.003	0.399	0.119	0.121
h_4	0.198	0.028	0.159	0.102	0.195	0.162	0.156
h_5	0.187	0.139	0.018	0.199	0.143	0.041	0.273
h_6	0.255	0.180	0.059	0.055	0.253	0.147	0.051
h_7	0.124	0.283	0.011	0.175	0.224	0.144	0.039
h_8	0.224	0.067	0.168	0.129	0.044	0.173	0.195
h_9	0.189	0.199	0.059	0.255	0.037	0.033	0.228

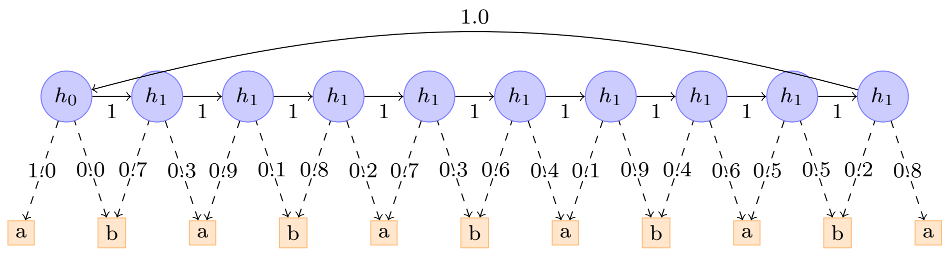
Example: `beadecffgfeceaabdaabcdcegcgggfaaeaebbbacafffffacg`

Table 4.1. Large Random HMM: Here we define a uniformly random model with 10 hidden states, and 7 emission values (observable states). In random models such as this, synthesized sequences will often lack sufficient examples of specific transitions (since there are many states, some less common than others, with some transitions being less common than others as well), thus the training data is likely to not match perfectly the real data, increasing the chances for overfitting. On the other hand, identifying the exact model in a random scenario such as this may not be particularly necessary for a reasonably approximation of the data, as the generated sequences are already nearly random.



Example: `abbacaaaabaaabccbaaabccaabcaaabcabccbbbcaabbbcaac`

Figure 4.3. Simple HMM: In this model we have a Hidden Markov Model similar to Figure 4.1, but now with fully ambiguous emissions. While there is more ambiguity here than in the model for Figure 4.2, the consequences of a mistake are less dire.



Example: `ababaababaaaababbbaaababbabaaaababbbbabaabaaaabab`

Figure 4.4. Chain HMM: Similar to our original *work-day*/*non-work-day* ring model we proposed in Section 3.2.2, the model here runs in a loop, emitting `a`'s and `b`'s stochastically as it goes. Local minima are a serious problem here, as identifying such a long chain without constraining the transitions so they follow a ring shape is difficult.

From each of these models we derive two sequences of length 2000, representing a training-series, from which the optimizers must derive their parameters, and a test-series, on which we can evaluate the quality of the abstraction the optimizers built. We chose this number of samples experimentally as it was a reasonably small number of samples where training and test sets no longer had such distinct features in most models (that is, where overfitting was minimized).

The training-series are then each used to generate a set of 20 models for each optimizer. Parametrically, the number of iterations and Gradient Descent α values were chosen so as to ensure convergence of the optimizers. We summarize these parameters in Table 4.2. Note that we chose the number of hidden and observable states to match

Model	Hidden	Observable	α	iters GD	iters BW
Small MC	3	3	0.01	1000	200
Ambiguous MC	5	4	0.01	1000	200
Small HMM	3	3	0.01	1000	200
Large Random	10	7	0.10	3000	400
Chain	10	2	0.01	3000	400

Table 4.2. Optimizer parameters used on the training sequences. Additionally, 300 and 50k iterations were used in Baum Welch (BW)/Gradient Descent (GD) respectively. The α parameter refers to the update scaling (detailed in the Gradient Descent algorithm, Algorithm 4)

those of the ground truth (giving them more than that would be fine, and often results in better likelihood metrics, but was unnecessary in these experiments).

To evaluate our results, we use likelihood as our measure, which we compute on the train- and test-sequences. We then compare these results with the likelihoods obtained using the correct model (ground truth) and a set of 20 random models of equal design. An analysis of the mean and standard deviation of these likelihoods in each set was done, aiming to better highlight the relationships between the upper and lower bounds of the performance of the optimizers.

Code for this experiment (including the optimizers and likelihood computation) was coded by the author in *C++*. Intermediate calculations in the likelihood and Baum Welch algorithm were, however, verified against the *HiddenMarkov R* package[Harte, 2016], and the Hidden Markov Model optimizer was constructed through automatic differentiation using the compiler we wrote. Experiments were run on a desktop computer with an Intel *i7-4770 CPU @ 3.40 GHz* processor and 28.0 GB of DDR3 RAM @ 800Mhz.

4.1.2 Results

Using the setup described in Section 4.1, we obtained models and likelihoods for each of the problems from Figures 4.1 - 4.4 and Table 4.1. The likelihoods of the models generated by each approach (Random model (RND), Ground Truth (OPT), and optimized via Gradient Descent (GD) and Baum Welch (BW)) for the training- and test-sequences are shown and analyzed in Figures 4.5 through 4.9.

Results are generally favorable, showing that our Gradient Descent optimizer performs virtually identically to the Baum Welch algorithm in most scenarios, though with a slightly higher variability.

In the first case, *Simple MC* (Figure 4.1), one of the downsides of using Gradient

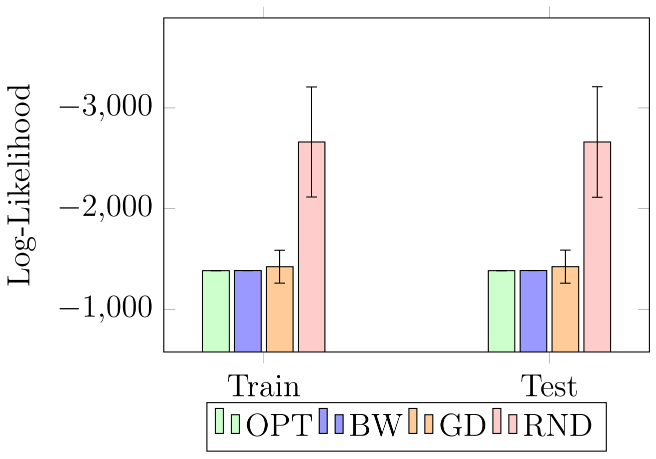


Figure 4.5. Simple Markov Chain (Problem from Figure 4.1) train- and test-series likelihoods.

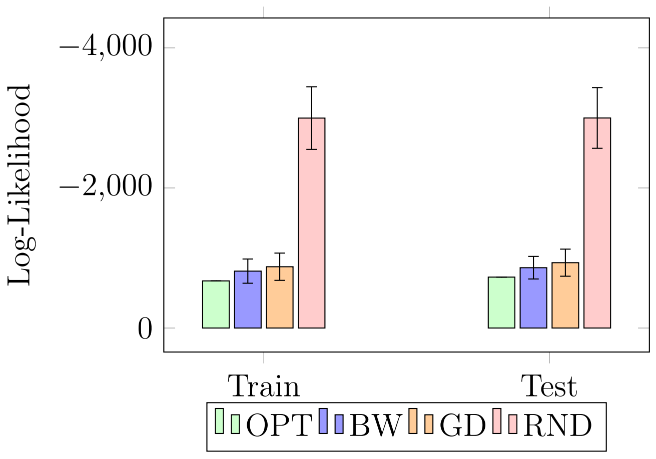


Figure 4.6. Ambiguous Markov Chain (Problem from Figure 4.2) train- and test-series likelihoods.

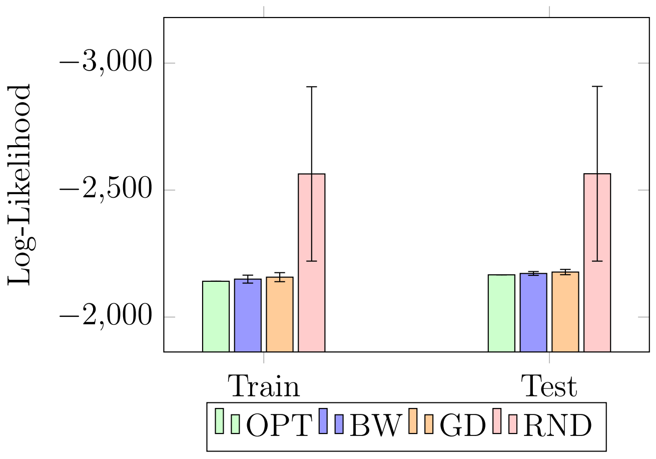


Figure 4.7. Simple HMM (Problem from Figure 4.3) train- and test-series likelihoods.

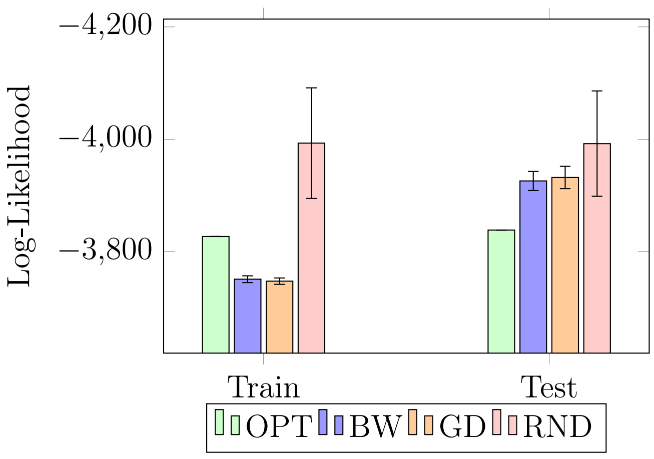


Figure 4.8. Large Random HMM (Problem from Table 4.1) train- and test-series likelihoods. As expected, the large amount of states led to a disconnect between the training- and testing-sequences' distributions, causing overfitting. Even so, results are relatively accurate, thanks to the mostly random nature of the sequence. Larger sample sizes could improve the performance of both optimizers.

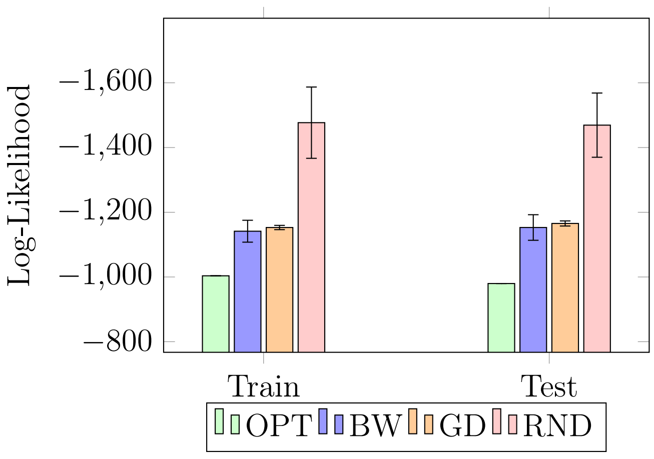


Figure 4.9. Chain HMM (Problem from Figure 4.4) train- and test-series likelihoods.

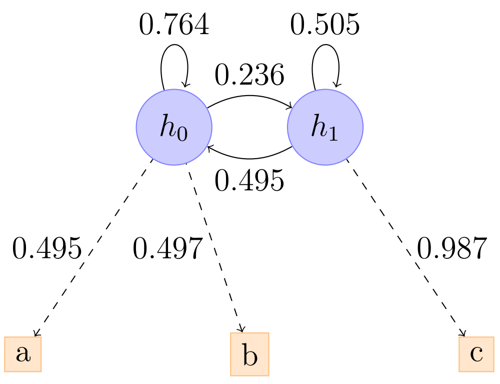


Figure 4.10. Local minima the Gradient Descent optimizer got stuck in when using the training sample from the Simple Markov Chain (Problem from Figure 4.1). Weights were truncated and a few of the near-0 values were omitted.

Descent becomes evident by inspecting the error bars, which appear as large as they do because of a single failure in the Gradient Descent optimization process. The culprit, shown in 4.10, illustrates the two-state local minima the optimizer got stuck in, where somewhere along the way the third state became unreachable. An analysis of the unreachable third state reveals it specialized in emitting `a`'s and `c`'s, but with a lower probability than h_0 has for `a` and a worse probability than h_1 has for `c`. We can speculate from this that it was likely worse at modeling the data than h_0 and h_1 were, and thus, when coupled with a poor random choice in the initial set of transitions, this would have prompted the other two states to prefer each other, gradually excluding it from the model.

Ambiguity in the transition and emission model were handled well by both the optimizers, however, as expected, extreme examples such as the *Large Random HMM* and *Chain HMM* challenges (from Figure 4.4 and Table 4.1, results in Figures 4.8 and 4.9) provide for too many local minima, which were too much for either model to handle. Being that as it may, the results on both models are still rather accurate (taking note of the scale on the axis on the *Large Random HMM* results), and neither optimizer has a clear advantage.

The computational resources required by the optimizers was relatively negligible considering modern hardware. On the most expensive example, the *Large Random HMM*, the Baum Welch algorithm took an average of 0.62 seconds to complete, and used approximately 700KB of RAM; The Gradient Descent optimizer used significantly more resources, but was still at a meager 3.05 seconds and 4MB of ram.

We now revisit our research questions, and argue them individually:

- **Can Gradient Descent optimization obtain an adequate Markov Model**

from data?: In the first three examples we proposed the Gradient Descent optimizer was able to retrieve precisely the original model using only the data, getting snagged on local minima only very occasionally. In the last two, more complex examples, the performance of the model obtained through GD is reasonably close to that of the optimum, but has more difficulties on account of the size of the problem.

- **How does Gradient Descent optimization compare to more conventional methods?**: We compared the performance of our optimizer with that of the Baum Welch algorithm, and found them to be roughly equivalent in most cases. This might be because the Baum Welch algorithm is also a method which aims to find a point where the gradient is zero, and is merely faster at traveling between local minima than Gradient Descent. Resource-wise the Baum-Welch algorithm is nearly an order of magnitude more efficient, but both approaches make fairly reasonable uses of computational resources.

4.2 Digit Classification

In contrast to Section 4.1, where we explicitly compared the effectiveness of Gradient Descent optimization with more traditional optimizers, here we focus on the flexibility afforded to the models by this approach. To this end, we explore our visual Markov Model from Section 3.2.1 in the context of digit classification on the MNIST dataset. Our goal with these experiments is to gain insights into the following research questions:

- **Is the process of extending the observation function expedient?**: We posited that using Gradient Descent made designing novel observation functions more practical for final users. We answer this analyzing if the experience designing this model reflects this added benefit.

- **Are Markov Models effective when using more complex observation functions?**: While in theory we can append nearly arbitrary observation functions to a Markov Model, does its structure lend itself well to creating a usable gradient on the observation layer? Some models, such as deep Multilayer Perceptrons and Recurrent Neural Networks will unintentionally obscure the gradient in a phenomenon known as the vanishing gradient problem[Hochreiter, 1998], thus, it is a legitimate issue which must be addressed.

- **Are shared and conditionally defined transition probabilities effectively optimized by Gradient Descent?**: Shared transition probabilities, whereby

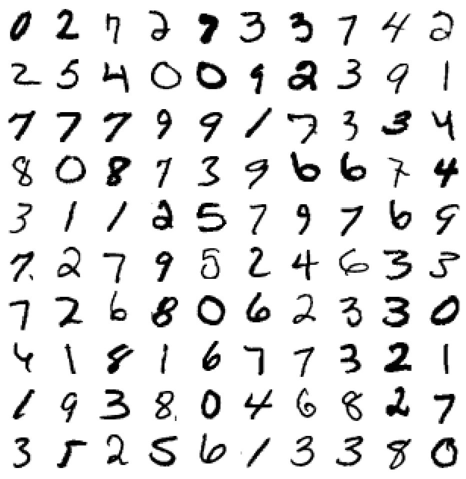


Figure 4.11. Examples of the hand-written digits from the MNIST dataset. In the data, each digit is a separate 28x28 pixel grayscale matrix.

multiple state transitions share a single parameter and/or are defined in function of the same set of parameters have the potential to allow Markov Models to be composed of a larger number of states without some of the usual side effects such as overfitting and slow convergence rates. Similar to our question about observation function optimization, it is necessary to verify if the gradient is viable for more complex forms of modeling these values.

4.2.1 Setup

We train our model on the MNIST dataset[LeCun et al., 1998b], one of the most well studied datasets in the area of image analysis. The data is comprised of 65k hand-written digits, split into training- and test-sets at a ratio of 55/10. Each image in the dataset is 28x28 pixels, and is a grayscale value between 0 and 255 (normalized to be between 0 and 1 in our case). A few instances of the dataset are illustrated in Figure 4.11.

The model itself was implemented in python within *TensorFlow*, a tensor-based automatic differentiation and general machine learning framework owned by Google[Abadi et al., 2016]. While less flexible than other forms of automatic differentiation, the framework also offers GPU support, which speeds up the training process immensely.

We evaluate the performance of our model using test-set accuracy, and compare it with some of the results reported by [LeCun et al., 1998a], the authors of the dataset, as well as a CNN which uses the same convolutional layer architecture as our Visual Markov Model.

Parametrically, we used 2 convolutional relu layers, with 32 and 64 feature maps, both with 5×5 kernels and 2×2 max pooling, and ran our transition model for 3 iterations in order to calculate the likelihoods (additional iterations were tested, but found to not significantly affect model performance). In the MLP portion of our CNN baseline we utilized 1024 neurons. While a few different values were tested, the main goal of this experiment was to explore the viability of the method, rather than achieve optimal performance.

We ran our experiments on a desktop computer with an Intel *i7-4770 CPU @ 3.40 GHz* processor and 28.0 GB of DDR3 RAM @ 800Mhz. Also relevant is the GPU, the NVIDIA GTX 970 (4GB video RAM), as it is used by TensorFlow in the optimization process.

4.2.2 Results

Figure 4.12 illustrates the performance of the Visual Markov Model on MNIST's test-set as optimization progresses. From it we can conclude that our concerns about the viability of the gradient are unfounded, as the model starts at random and, by the end, reaches 96.02% accuracy.

Figure 4.13 compares the performance of our model with other methods. In it, we see that we are able to exceed the performance of methods which are not specialized in visual data analysis (linear regression, K-nearest-neighbors, and a two layer Multilayer Perceptron), but fail to surpass more specialized algorithms (including the CNN which uses the same convolutional architecture). While somewhat discouraging, these results are expected as we are competing with the state of the art in static image processing using a model designed for more generalized image sequence analysis tasks.

The main drawback of the model we presented is its excessive use of computational resources and time: Where on one hand training the CNN we used as a baseline took under two minutes and used only a few hundred MB of RAM, on the other, the Visual Markov Model required nearly 3 hours and over 10GB of memory. While these requirements aren't out of the ordinary for many machine learning algorithms, they scale with N^4 (where N is the size of the grid created by the convolutional layers), making the model, as it stands, unviable for larger images.

We now revisit our research questions:

- **Is the process of extending the observation function expedient?**: Considering the complexity of the model we proposed – a level which would be untenable using more traditional optimization techniques – the model implementation was

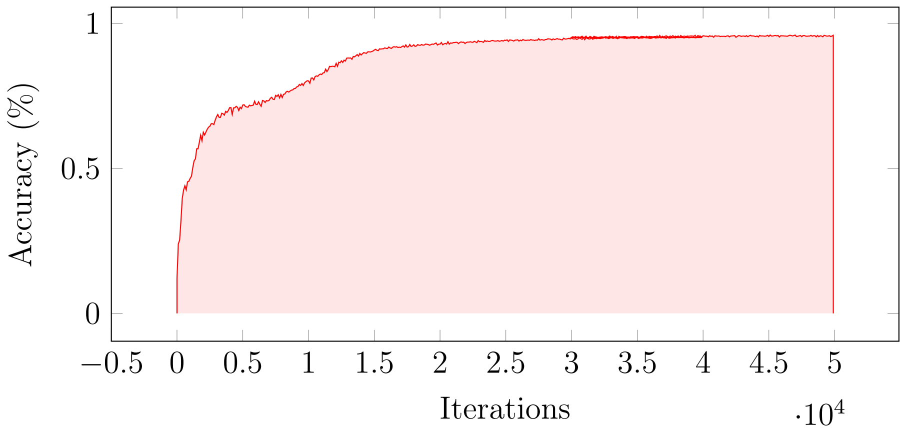


Figure 4.12. Test-set error of the visual Markov Model on the MNIST dataset. By 50k iterations the model has essentially converged, with a best test-set score of 96.02%

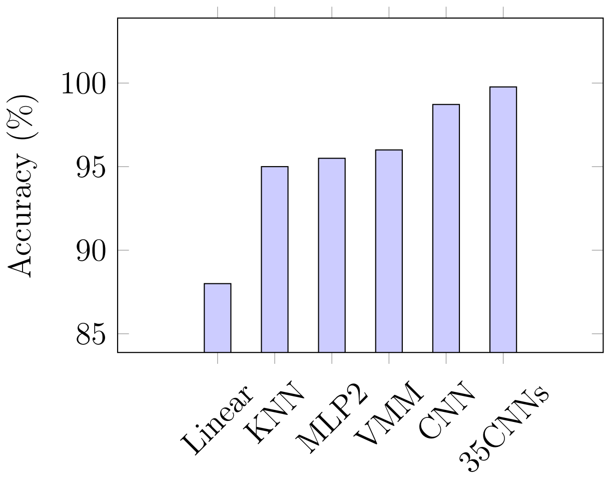


Figure 4.13. Performance comparison between the Visual Markov Model (*VMM*) and Linear regression (Linear), K-Nearest Neighbors with euclidean distance (*KNN*), a 2 layer MLP with 300 hidden units (*MLP2*) [LeCun et al., 1998a], our CNN, trained with the same architecture as the base of the VMM and 1000 hidden units in the MLP layer (*CNN*), and finally, a CNN committee[Ciregan et al., 2012].

relatively simple, certainly not requiring expert knowledge about Markov Model optimization.

- **Are Markov Models effective when using more complex observation functions?**: Our model achieved a commendable 96.02% accuracy on our training data, which could only be achieved by successfully co-optimizing the convolutional layers. From this we can conclude that, indeed, Markov Models can effectively optimize complex observation functions.

- **Are shared and conditionally defined transition probabilities effectively optimized by Gradient Descent?**: The same evidence used to justify the effectiveness of our model in optimizing the observation function also applies here, as failure to optimize the complex transition model we proposed would inhibit the level of accuracy achieved. Furthermore, an inspection of the means and standard deviations of the model's parameters reveals that it indeed optimized them as intended, learning to construct paths between relevant features, and forming projections outside of the image grid for features which were found to not be useful.

4.3 Stacked Markov Model

In this set of experiments we aim to explore the effectiveness of the Stacked Markov Model, a model which we introduced in Section 3.2.2, whose purpose in the scope of this thesis is to illustrate the benefits of having a more flexible transition model, which comes naturally from the use of Gradient Descent optimization.

To this end, we pose the following research questions to guide our analysis:

- **Is the process of defining novel forms of Markov Models expedient using Gradient Descent?**: While expediency is somewhat subjective, we take it here to relate to the effort and knowledge required by the user to develop a model and optimizer which expresses their desired architecture. While some effort and knowledge will always be required, we analyze this research question considering the relative difficulty of achieving the same without the tools we have proposed. Our discussion on this issue is mainly centered in our implementation section, Section 4.3.1.3.

- **Is there justification for adopting more complex transition models?**: Having a more flexible framework for defining customized transition models is only useful insofar as there are benefits to be gained from novel architectures. While this may be a sensible assumption, it is nevertheless something which must be analyzed. We explore this issue differently in each of our experimental sections (Sections 4.3.2.1, 4.3.2.2 and 4.3.2.3).

More specifically, and more related to the second question, we aim to validate the properties of the SMM, which, in the case that they are shown to exist, may justify its use outside of this thesis. These secondary questions are as follows:

- **Is the SMM effective at modeling time series with long term contextual issues?**: The main reasoning behind the construction of the SMM was that series often don't strictly follow the Markov Property, and that memory in Markov Models is an expensive resource. The SMM aims to solve these issues by storing context information in higher level Markov Chains, which vary at a slower rate, and influence lower levels by defining their transition models. While this is theoretically the case, it remains to be seen if the model can indeed learn to represent contextual information.

- **Can the SMM model multiple time series simultaneously, sharing contextual information as needed?**: We extended the SMM in Section 3.2.2.4 so as to include multidimensional data, positing that the model could learn to efficiently share a parent layer should there be sufficient correlation between their data. We explore this very scenario in the experiment described in Section 4.3.1.6.

4.3.1 Setup

4.3.1.1 Datasets

In contrast to the previous experimental sections, here we will focus more on more realistic real-world sequence data (as opposed to the synthetic examples we used to validate our optimizer in Section 4.1, and the somewhat artificial sequence-interpretation of digit classification seen in Section 4.2). While it is the case that any sequence can be modeled by the Markov Models we've investigated, we will focus on data which is informative of the properties we wish to explore, namely the intersection between stochasticity and structure (which is to say that we don't desire data that is far too random for any context information to be relevant, but also not overly structured, so as to discard the need for the stochastic properties of Markov Models.

To this end, we focus our experiments on 3 histograms: The first, a plot of the outside temperature near the airport in Belo Horizonte (Brazil), presents itself as a

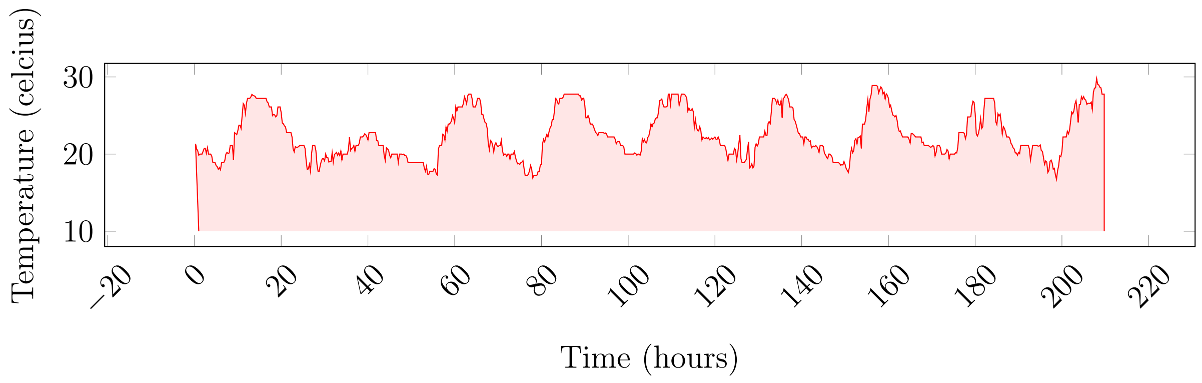


Figure 4.14. Sample from the outdoors temperature data from the Confins International Airport in Belo Horizonte, Brazil. While the behavior changes day by day depending on a multitude of factors (e.g.: cloud coverage, wind speed, rain, etc), patterns and regularities emerge thanks to the day/night cycle.

good candidate on account of it having structure in the form of periodicity (such as the rise and fall of temperature during the day), but also stochasticity (numerous events can affect the measured temperature on a given day, ranging from cloud coverage to humidity to wind speed). We illustrate a portion of this series in Figure 4.14.

The other two histograms are of the CPU and Memory usage of a machine within a network which, in addition to displaying the usual patterns of activity typical of an idle computer, also periodically runs a series of computationally intense operations. We show examples of these series in Figure 4.15.

In all three cases the data comes in as a sequence of floating point numbers, with the temperature being sampled at a frequency of 15 minutes, totaling 32152 values (or 334 days) and a total of 20546 CPU and Memory samples were used, sampled at a rate of one a minute (24 days).

For the purposes of testing, we split each series by a 70/30 ratio into a training- and a test-sequence. The importance of context to our experiments means that it is best that the training and test data be contiguous in time, thus the test-sequence was cut from the end of each series. A concern here was that having such a large contiguous batch of samples might affect the homogeneity of the training and test sequences. To ensure this was not the case, the training- and test-sequences were visually inspected to ensure that they exhibited similar patterns.

In one case the floating point values were used directly (with the data normalized beforehand so it had a 0-mean and a standard deviation of 1). However in the other two experiments we discretize the values into buckets. A copy of the samples were

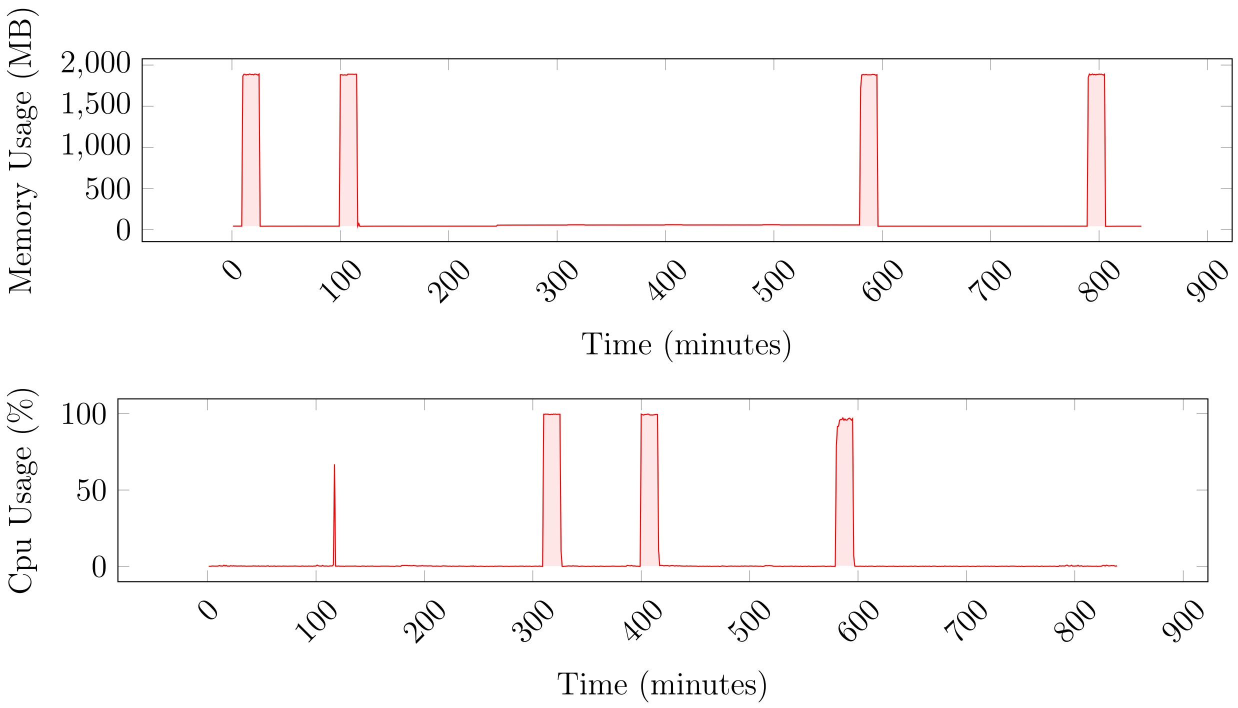


Figure 4.15. Sample from the machine's CPU and memory usage. Activity generally remains nominal, with small variations in activity, except for periodic peeks caused by a timed process. The peeks, caused by synthetic stress added to the system are often correlated between series, but, as can be seen above, this is not always the case.

sorted, and then analyzed so that the ranges for each bucket could be determined in such a way so as to ensure all buckets had an equal amount of samples – a technique designed to minimize the effects outliers have on uniformly sized buckets. A total of 18 buckets were used, a somewhat arbitrary amount that provided good granularity on the data while not making it overly sparse.

4.3.1.2 Baseline and Evaluation Measures

As stated at the beginning of this section, one of our research questions is whether or not there is any utility in allowing a user to define their own architectures, such as we have done with the SMM. It stands to reason that if conventional techniques were to have similar performances to any novel models a user could come up with, then this supposed benefit would not be real (as there would be no reason to utilize anything but the well established models). Conversely, if we can design a Markov Model that performs better than more traditional approaches in some set of contexts, as we have aimed to do with the SMM, then the use of alternate optimizers is justified,

Layer	States	Δ_l	Time scale
0	18	5	0h15m
1	20	3	1h15m
2	1	∞	3h45m

Table 4.3. Parameters used to define the SMMs we trained on Temperature, CPU and Memory data. Note that the bottom layer has 18 states, one for each bucket in the histogram's discretization, whereas the top layer has only a single state, as is always the case, which allows it to merely define a transition model and Δ value for the layer below it). Δ_2 is infinite here so as to prevent our recursive definition of the model from trying to run non-existent layers above layer 2.

as there would be evidence to sugest that there are contexts where a user might design a transition model architecture which performs better than conventional techniques. It is with this in mind that we chose the Hidden Markov Model as our baseline in these experiments.

While simplistic, the HMM is a good baseline for the SMM on account of the following: (1) Both are Markov Models, which allows us to support our hypothesis that Gradient Descent can improve the flexibility of Markov Model design, and that having this flexibility is useful within this class of models, (2) both models can be compared using their likelihoods over a sequence, a well established measure that captures both specialization and generalization of the models over their data, and (3), while more complex Markov Models exist, the HMM remains the most widely used model of the category, and as such, demonstrating that its performance can be surpassed without the need for specialist knowledge would best argue for its use by the machine learning community.

Parameters for the HMM and its optimizer were obtained by iteratively testing different state counts and optimization iterations, with near optimal parameters being chosen. For the HMM, we utilized 30 hidden states and 300 iterations of the Baum Welch algorithm. These values were chosen to be as large as possible before overfitting started having a negative impact on test-sequence performance. In the case of the SMM, the large amount of resources required by the model reduced the total number of configurations that could be tested, thus parameters were determined manually by varying individual values until satisfactory result were obtained. We utilized a 3 layer model, whose architecture is summarized in Table 4.3.

4.3.1.3 Implementation

We implemented our baseline, the Baum Welch algorithm, in *C++*, checking its intermediate and final outputs against the *HiddenMarkov R* package, so as to minimize the chance that an implementation error would affect our results. The SMM and its optimizer, on the other hand, underwent a much less straight forward development process.

Initial development went into trying to devise an extension to the Baum Welch algorithm which would optimize the SMM. This proved to be impossible, however, as relaxing the requirements for the observation function to be a probability distribution (allowing it to be a PDF, for example), inhibited important steps in the deduction of the algorithm (namely setting up the Lagrange multipliers, whose parameters serve as a left hand side when trying to analytically solve for setting the gradient to zero). The first versions of the SMM's optimizer using Gradient Descent were done manually, requiring close to 4 pages of relatively simple, albeit tiresome differential calculus. Mistakes can be made at any step in this process, with a mistake in the design process cascading to the derivation phase, and then in turn to the software implementation.

The laborious nature of this process led to the design and implementation of our own Domain Specific Language (DSL), accompanied by a compiler that supported automatic differentiation. We mention this whole process in light of our hypothesis that automatic differentiation can make Markov Model design accessible to the average user, which is exceedingly clear when considering the long stream of difficulties associated with trying to develop our model in a conventional way which made it virtually impossible to bring it to fruition, in contrast to what became possible after automating the gradient implementation. As a testament to the effectiveness of this procedure, the same prototype which before took pages of calculus and hours of work now was summarized to 50 or so lines of C-like syntax, and worked flawlessly.

As per the DSL, a full accounting of its syntax and inner working goes beyond the scope of the text for this thesis, nevertheless, a few examples are useful in arguing its accessibility for the final user. Figure 4.16 exemplifies the implementation of a simple Multilayer Perceptron in the language, whereas Figure 4.17 illustrates a simple Markov Model (the implementation of the SMM can be found in Appendix A). Different from the more common tensor approach to automatic differentiation, the commands in this language have a much higher granularity, allowing for a C-like syntax, as well as features such as for-loops, conditional expressions, auxiliary methods and classes, value reassignment, C++ inlining and recursion.

The compiler implements backward-mode automatic differentiation, and, due to recursion, uses a call-stack-like system to keep track of the information needed to precisely reverse the control flow in the backward pass, as well as store the Jacobians needed in computation (refer to Section 2.2 for the general model for automatic differentiation). Needed values are stored sequentially so as to maximize spacial and

```
hyperplaneRegressor{
    init(int _dims){
        int dims = _dims;
        float w[dims] = random();
    }
    f(float * x -> float y){
        y = 0;
        for (i until dims) y += x[i] * w[i];
    }
}
mlpLayer{
    init(int inDims, int outDims){
        int units = outDims;
        hyperplaneRegressor HR[units](inDims);
    }
    f(float * x -> float * y){
        for (u until units) y[u] = sigm(HR[u].f(x));
    }
    fLSE(float * x, float * y -> float LSE){
        float yEst[units] = sigm(HR[_0].f(x));
        LSE = 0;
        for (u until units) LSE += (y[u] - yEst[u])^2;
    }
}
mlp{
    init(int * _dims, int _layerCnt){
        int layerCnt = _layerCnt;
        int dims[layerCnt] = _dims[_0];
        mlpLayer layer[layerCnt - 1](dims[_0], dims[_0 + 1]);
    }
    f(float * x -> float * y){
        float aux[layerCnt][dims[_0]];
        for (i until dims[0]) aux[0][i] = x[i];
        for (l until layerCnt - 1)
            layer[l].f(aux[l] -> aux[l + 1]);
        for (i until dims[layerCnt - 1])
            y[i] = aux[layerCnt - 1][i];
    }
}
```

Figure 4.16. Multilayer Perceptron implemented in our Domain Specific Language.

```
HMM{
    init(int _sCnt, int _oCnt){
        int sCnt = _sCnt, oCnt = _oCnt;
        float pD[sCnt][sCnt] = random();
        float oD[sCnt][oCnt] = random();
    }
    fn(int * obs -> float logL){

        // Denormalize weights
        float pNF[sCnt], oNF[sCnt];
        for (i until sCnt){
            for (j until sCnt) pNF[i] += pD[i][j]^2 + 0.0001;
            for (j until oCnt) oNF[i] += oD[i][j]^2 + 0.0001;
        }
        float p[sCnt][sCnt] = (pD[_0][_1]^2 + 0.0001)/pNF[_0];
        float o[sCnt][oCnt] = (oD[_0][_1]^2 + 0.0001)/oNF[_0];

        // Compute logL
        logL = 0;
        float distSrc[sCnt], distDst[sCnt]; distSrc[0] = 1.0;
        for (t until obs.size){
            for (s until sCnt) distSrc[s] *= o[s][obs[t]];
            for (src until sCnt)
                for (dst until sCnt)
                    distDst[dst] += distSrc[src]*p[src][dst];
            float pSum = 0;
            for (s until sCnt) pSum += distDst[s];
            for (s until sCnt) {
                distSrc[s] = distDst[s] / pSum;
                distDst[s] = 0;
            }
            logL += log(pSum);
        }
    }
}
```

Figure 4.17. Hidden Markov Model implemented in our Domain Specific Language.

temporal locality in both passes, and Jacobian calculations are designed to require the smallest amount of re-computation.

As a consequence of these factors the resulting optimizer is rather efficient in time – faster than many tested hand-written optimizers – but exceedingly inefficient in memory use. Each expression requires one Jacobian for every variable on its right hand side, meaning that runtime memory use scales with the number of computations, and the ability to reassign values to variables (e.g.: an increment) means that the results from the forward pass aren't always available to the backward pass for use in its partial derivatives.

Difficulties aside, once constructed, the compiler allowed for the development of the SMM with virtually no need for special considerations on how it was going to be optimized, illustrating the accessibility of these kinds of techniques to the average user.

4.3.1.4 Experiment Design: Series Likelihood Maximization

In this experiment we aim to quantify the relative performance of our model, the SMM, with respect to our baseline, the HMM, on the three discretized datasets discussed in Section 4.3.1.1. We train 20 instances of each model on the training sets, and plot the average and standard deviation of the log-likelihoods.

Specifically, we aim to answer the question of whether the SMM is competitive compared to other more traditional Markov Models. More generally, we argue that superior performance in these contexts demonstrates the need for novel forms of Markov Model optimization.

4.3.1.5 Experiment Design: Temperature Prediction

One of the main motivations for utilizing Markov Models is the set of tools which are associated with them. While these tools are generally applicable to any discrete Markov Model, it is useful to demonstrate that they function well in practice. Here, we make use of the predictive ability of this class of models, by computing the probability distribution of temperatures some time in the future.

Methodologically, we utilize the normalized real values of the temperature series directly, with a Gaussian pdf observation function. We impose a minimum values for $\sigma = 0.05$, using a soft-max function so as to prevent states from developing point-distributions. While not a valid observation function, the resulting model and pseudo-likelihood are still useful for our purposes (as discussed in 3.1), as the density functions of the states can be added together, weighted by the probability of their states, in order to form a mixture model, which in turn gives us a distribution of probable temperatures.

Beyond exemplifying the use of prediction and of non-probabilistic observation functions, the experiment allow us to highlight the ability of the SMM to hold onto contextual information, as the accuracy of the predicted distribution depends on the model properly learning the periodicity of the temperature cycle. Since predictive accuracy alone is difficult to judge in a vacuum, we contrast the SMM's predictions with those from a simpler HMM, similarly trained on continuous data.

We us the same set of parameters for the HMM and SMM as described in the discrete cases in Section 4.3.1.2, only now we replace layer 0's observation function with the aforementioned Gaussians.

4.3.1.6 Experiment Design: Multi-series Analysis

In our final experiment, we explore the multi-series capabilities of the SMM, as laid out in Section 3.2.2.4. Our main goal is to test our prediction that the SMM can learn a joint transition model at higher level Markov Chains which translates well to multiple time series, so long as they are related. Success in this matter would signify a better approach towards multidimensional data than is commonly available in Markov Models, as the conventional approaches either assume conditional independence of the different dimensions of the data, or attempts to model both values simultaneously in each of the states, leading to an explosion in the number of model parameters.

We make use of the CPU and Memory series in this example, as phenomenon which affect one series often will also affect the other one. To gauge our success, we compare the sum of the likelihoods of the models trained independently with that of a joint model, which maintains the same set of higher level Markov Chains, differing only in the addition of an extra, equal base layer (that is, in the parameters from Table 4.3, an exact copy of layer 0 is added to handle the second series).

4.3.2 Results

4.3.2.1 Experiment: Series Likelihood Maximization

Figures 4.18 through 4.20 compare the performance of our model to our baseline HMM. As hoped, the addition of higher level Markov Chains and sequentially structured sequences lead to significant improvements an all tested examples.

Furthermore, an inspection of the generated models reveals that, as expected, the transitions in the Markov Chains mimic common shapes in the data, for instance, in the model trained on the temperature series, stability, rising temperatures and lowering temperatures are 3 common patterns which are learnt by the model.

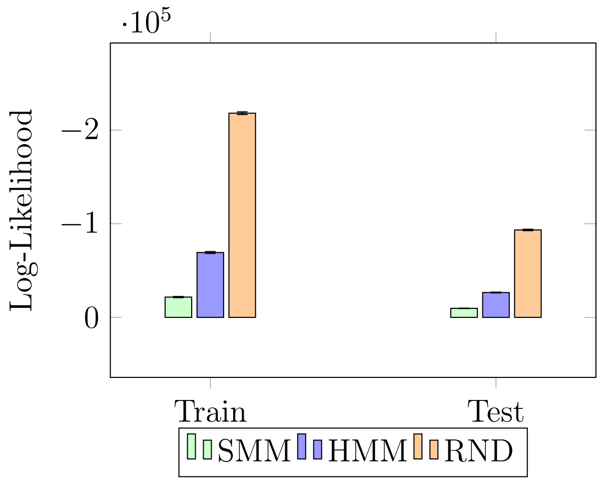


Figure 4.18. Log likelihood obtained on the temperature data histogram (Figure 4.14) by our model (the SMM), our baseline (the HMM), and, for reference, a random Markov Model (RND).

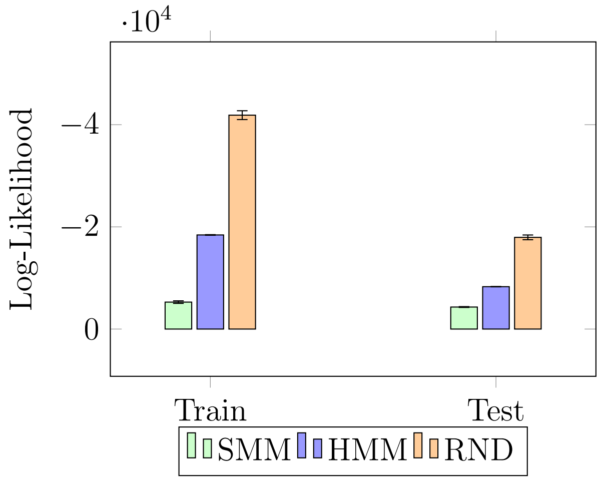


Figure 4.19. Log likelihood obtained on the CPU data histogram (Figure 4.15) by our model (the SMM), our baseline (the HMM), and, for reference, a random Markov Model (RND).

These results support our claims that the SMM can learn to store contextual information in higher-level states, and, more generally, that flexibility in defining a Markov Model's transitions can be a highly useful trait.

4.3.2.2 Experiment: Temperature Prediction

In this experiment we employed the predictive tools of Markov Models on our SMM, which was trained directly on the real valued inputs from the temperature histogram using a Gaussian observation function. Our results, shown in Figure 4.21 illustrates

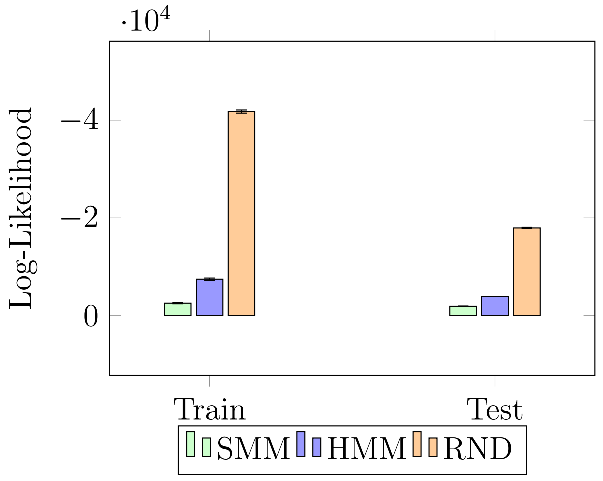


Figure 4.20. Log likelihood obtained on the memory data histogram (Figure 4.15) by our model (the SMM), our baseline (the HMM), and, for reference, a random Markov Model (RND).

the sliding distribution our model predicts for 15 minutes, 6 hours, and one day into the future.

From the results we can see that the model has acquired a firm grasp on the periodicity of temperature data, with there generally being a good match between the most probable estimate (the lightest color in each column) and the actual temperature value. The model does however display some difficulties inferring temperatures outside of the norm at points which are too far in the future. For example, the 15 minute prediction is rather accurate, while the 6 hour prediction tends to predict the general expected temperature at that hour, with only a slight relationship to its current observations. The 24 hour estimate shows no signs of having used any non-periodic information (e.g.: the model doesn't relate it being hot now with it being hotter than usual in 24 hours time). This kind of behavior is reasonable considering the data, since much of what can affect temperature measurements at one time (such as cloud cover or wind) can change within a few hours.

For comparison, we show in Figure 4.22 the same set of distributions, only now using a Hidden Markov Model on the data – a model which has no way of efficiently remembering context information. Not only are its predictions far less certain, but they are actually static: A local minima for temperature estimation is to always guess that the temperature in 15 minutes will be the same as it is now – a reasonably good guess in the short term, but hardly useful in practice. Furthermore, the lack of an understanding of periodicity means that the model's probabilities tend to approach the ergodic distribution as we increase k, as can be seen by the horizontal lines which

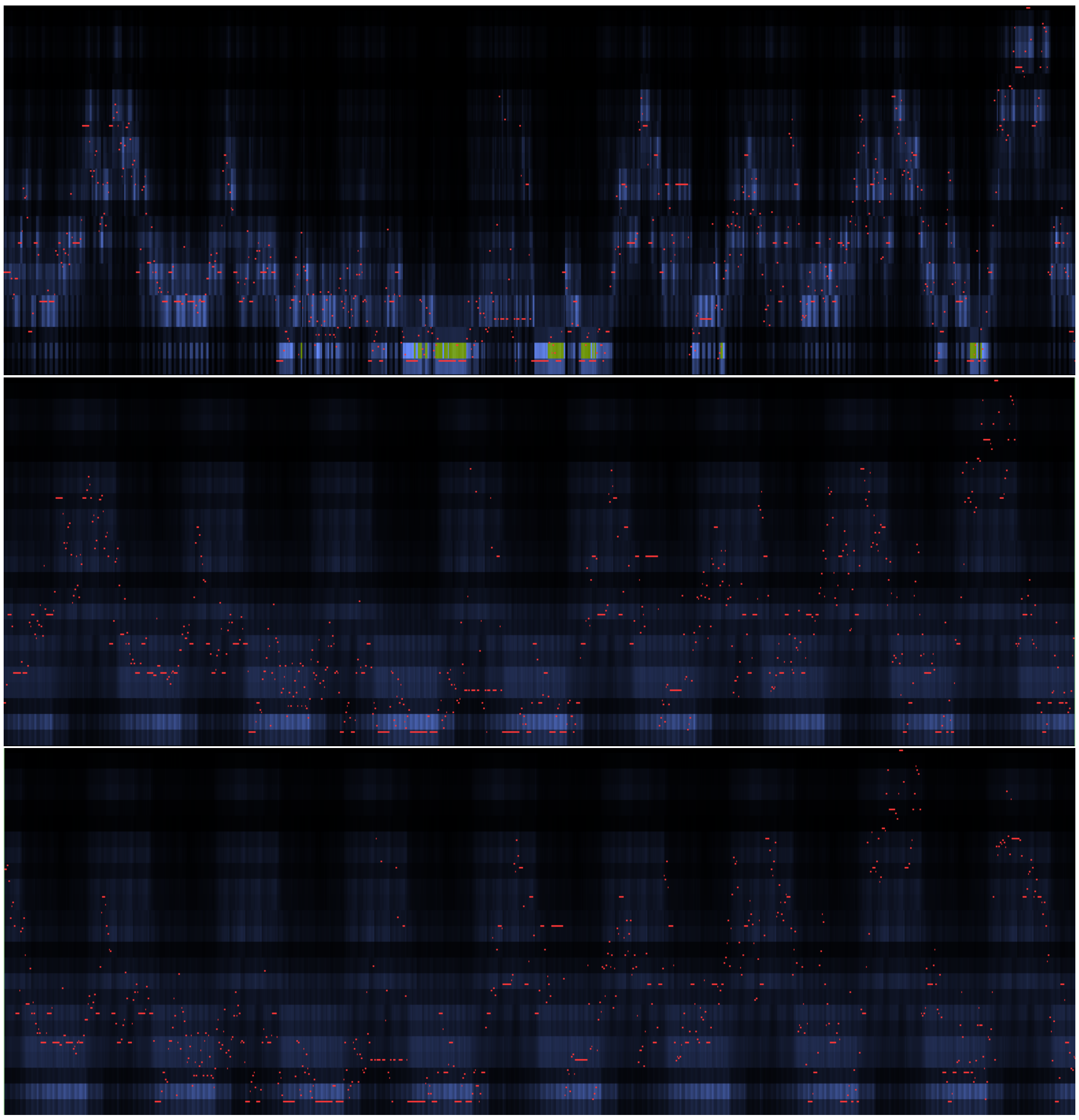


Figure 4.21. The SMMs prediction of the temperature in 15 minutes (top), 6 hours (middle), and 1 day (bottom). Each column represents one time index, and each row a temperature, with lighter colors representing a higher probability density than darker colors. To draw some column c, we first compute the probability distribution of the SMM given all data preceding c (that is, we compute $P[S_l^{(c)}|X^{(0..c)}]\ \forall l$). Then, we propagate these probabilities for some number k of steps ($k = 1$ for a 15 minute estimate, $k = 24$ for 6 hours, etc; This is equivalent to computing $P[S_l^{(c+k)}|X^{(0..c)}]\ \forall l$). We then plot the sum of the observation functions, weighted by each state ($\sum_i P[y|s_{0.i}^{(c+k)}]P[s_{0.i}^{(c+k)}|X^{(0..c)}]$). Succinctly, what this gives us the probabilistic estimate in $k + c$ steps, given the data until c. Finally, we also add red dots where the actual temperature was recorded at each time $c + k$, for comparison.

develop over the distribution, which represent some state in the model acquiring a higher probability and then never loosing it.

4.3.2.3 Experiment: Multi-series Analysis

In our last experiment we focus on the multi-series analysis properties of the SMM. We wanted to know if, when given two correlated series (CPU and RAM usage) if the model could jointly represent them through shared parent layers. Figure 4.23 illustrates these results.

In these results we can see that, even though the independently trained models together had vastly more parameters than the conjoint model, their performance was only marginally better. From this we can infer that, while some aspects which were abstracted from each series may have been lost when creating the conjoint model, they still had sufficient in common so as to enable their co-optimization. Thus, the results support our claim that the model is capable of abstracting common elements from multiple time series in order to jointly represent them.

4.3.2.4 Discussion

We now revisit our research questions:

- **Is the process of defining novel forms of Markov Models expedient using Gradient Descent?**: As discussed at length in Section 4.3.1.3, implementing the SMM using conventional techniques is quite literally impossible, on account of the violation of the key assumptions about probability distributions. The use of automatic differentiation, on the other hand, made for a rather straightforward design process, were time could be invested in the model, instead of the means by which it would be optimized.

- **Is there justification for adopting more complex transition models?**: Our ability to significantly outperform our baseline, the HMM, illustrates that there are scenarios where flexibility in designing more complex transition models can be highly beneficial.

- **Is the SMM effective at modeling time series with long term contextual issues?**: Our results from our temperature prediction experiment, as well as the inspection of the SMM's models, reveal that the model does indeed learn the patterns of inferior layers, thus codifying contextual knowledge of the layer below.

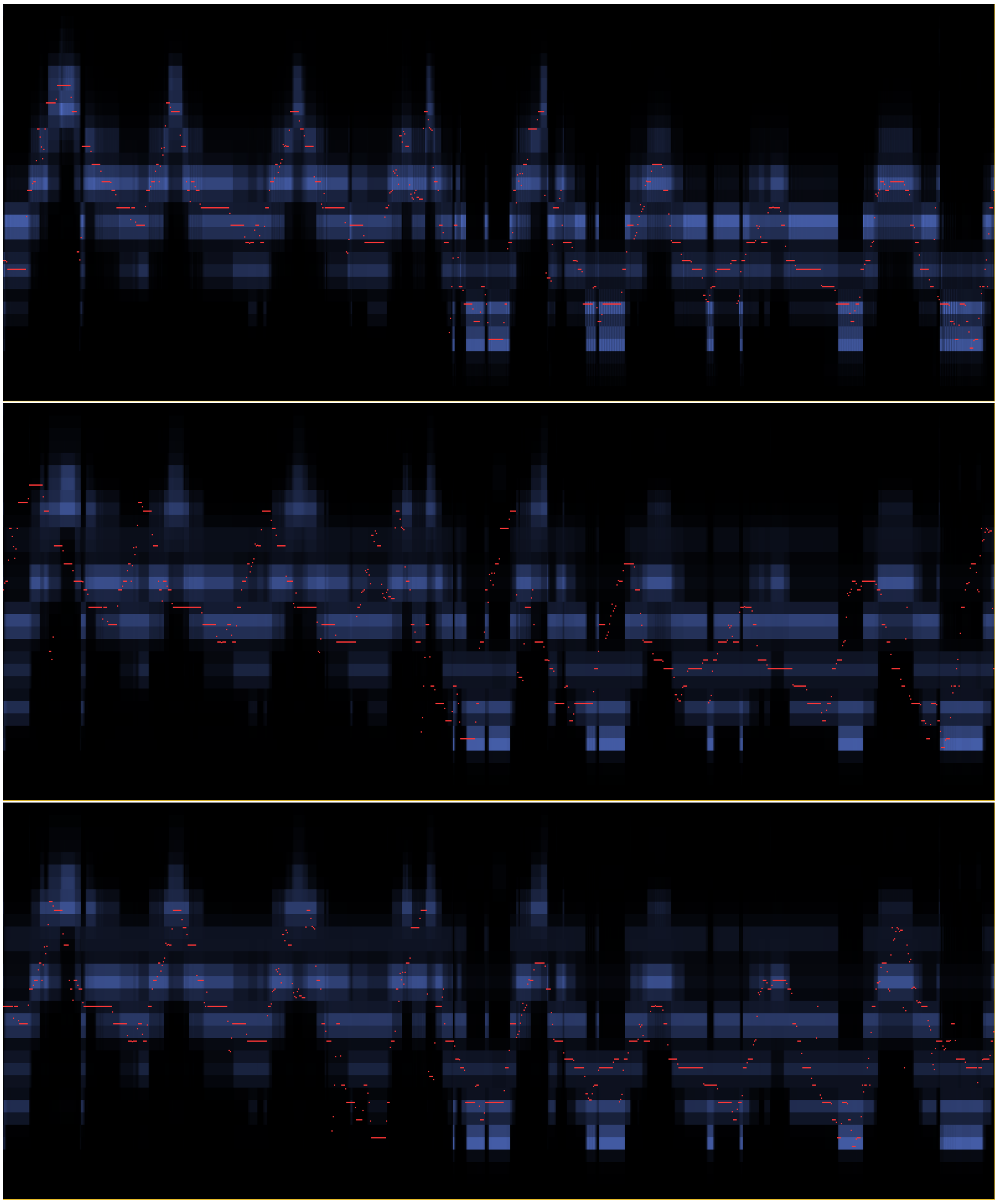


Figure 4.22. The HMMs running prediction of the temperature in 15 minutes (top), 6 hours (middle), and 1 day (bottom). The plot shown here is generated in the same way as that of Figure 4.21

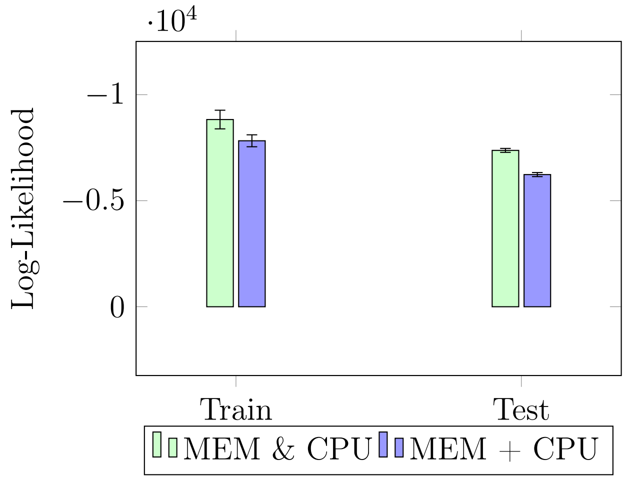


Figure 4.23. Log likelihood of the models trained together (*SMM & CPU*) vs the sum of their likelihoods when trained independently (*SMM + CPU*).

- **Can the SMM model multiple time series simultaneously, sharing contextual information as needed?**: Our final experiment demonstrated that, as designed, the SMM was able to deal with multiple correlated time series with only a negligible penalty to its performance.

Chapter 5

Conclusions and Future Work

In this dissertation we posited that Gradient Descent Optimization could be used to improve the flexibility and usefulness of Discrete Markov Models. We made this claim on the basis that the development of novel Markov Model designs through conventional means requires expert knowledge not common to most data scientists, whereas gradient-optimized models are much more accessible to users, in large part due to automatic differentiation. We argued that optimizing Markov Models in this way could open up new, exciting possibilities, as Markov Models still offer an unparalleled tool-set, such as prediction, simulation, and regression, whereas gradient-optimized methods such as Neural Networks offer state-of-the-art data analysis.

In order to argue these points, we began by tackling the issue of developing a Gradient Descent optimizer for the Hidden Markov Model. While our general goal was to go beyond simple models such as this one, we reasoned that, should Gradient Descent be shown as an effective alternative to the state of the art techniques for optimizing this class of models, then there would be reason to believe that it could be used effectively on other instances of Markov Models, for which there are no traditional approaches for optimization. Our results indicate that Gradient Descent is a highly competitive alternative to state of the art techniques, with its only downside being a reasonable increase in resource usage.

Having illustrated that Gradient Descent could be an effective optimizer for Markov Models, we next argued that the flexibility it afforded was indeed useful. We did this by constructing two novel forms of Markov Models. The first model we proposed makes use of a Convolutional Neural Network as its input, which is then co-optimized along side the state transition model, with the aim of building a visual classifier. The model highlights the advantages of our approach, as the convolutional layers develop the feature detectors needed by the transition model, without the need for the time consuming manual process that would normally be required. Furthermore, the model explores concepts such as conditionally defined and share transition probabilities, which can further aid in Markov Model design. We tested the model on the MNIST dataset, a digit classification task, and achieved promising results, allowing us to assert that complex observation functions could be used and conjointly optimized with Markov Models.

Finally, we further exemplified the utility of our approach through our second model, the Stacked Markov Model. In it, we devise a solution to the memory problem in Markov Models which involves storing contextual information within secondary Markov Models, who transition at progressively wider timescales. We test out this model on 3 real-world histograms, and show that we are able to significantly outperform our baseline, the Hidden Markov Model, on the same task, thus justifying the need for novel Markov Model architectures.

From these three tasks we believe that we have proved our hypothesis beyond a reasonable doubt, and hope that this can serve to motivate users of Neural Networks to think about ways to include Markov Models in their approaches, as well as conversely, to motivate Markov Model users to think outside the box, and make use of the analytic abilities of techniques such as Neural Networks. Less abstractly, our dissertation contributes with two novel techniques for data analysis, one of which we have already shown to significantly outperform traditional approaches.

5.1 Future Work

Most importantly, the techniques presented here open up a vast expanse of possible marriages between Markov Models and gradient-optimized approaches such as Neural Networks. Foreseeable useful applications revolve around the analysis of real-world sequence data, as real-world data processing often benefits from techniques such as Neural Networks, whereas the tool-set inherent to Markov Models makes them ideal for understanding the sequential aspects of the data. Common examples of these sorts of problems are audio and video analysis, but more obscure applications exist, including topics such as the analysis of series from Smart Cities or information flow on the internet. Exploring these possibilities is liable to be a very fruitful direction for future research.

Our optimization technique offers directions of research related to improving the effectiveness of Gradient Descent on Markov Models, as it is generally the case that every new class of models can lead to some adaptation of the Gradient Descent algorithm which improves optimization speed and performance. In particular, we briefly investigated the use of second order optimization techniques, such as Hessian Free Optimization, which also allow for automatic differentiation, as a means to significantly improve optimization performance. These attempts were not realized, however, due to hurdles associated with convexity constraints on the objective function. Nevertheless, future research into this area may well reveal a better way of optimizing models such as ours.

Less abstractly, the models we devised here can also be improved upon, and applied in different scenarios. The Stacked Markov Model has obvious applications in text analysis, as many techniques in natural language processing already are forms of Markov Models (such as n-gram models and Latent Dirichlet Allocation), and stand to benefit from contextual information. The high dimensionality of text data is, however, a challenge for this kind of model, namely because if we allocate one state per word, then the resulting transition model ends up having millions of parameters, becoming computationally intractable. Further work also needs to be done in the SMM in defining architectures for multiple time series, since, as it stands, there aren't good indicators for determining which series could efficiently share a common parent layer, other than testing out the resulting model, a relatively expensive process if trying to find the optimal tree model with more than a few dozen series.

Finally, with respect to the Visual Markov Model, further analysis on other data is required in order to truly evaluate its potential. Before this can be done, however, its excessive use of computational resources must be addressed, as the size of the model it generates quickly becomes intractable for larger images. Nevertheless, adaptations to this model could be made which would allow it to look for patterns in video data, as well as hand-written document optical character recognition, problems for which there still aren't particularly great solutions.

Bibliography

(1985). *IEEE standard for binary floating-point arithmetic*. Institute of Electrical and Electronics Engineers, New York. Note: Standard 754–1985.

(2016). Google search statistics.

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., et al. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*.

Banerjee, U., Eigenmann, R., Nicolau, A., and Padua, D. A. (1993). Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211--243.

Baum, L. E., Petrie, T., Soules, G., and Weiss, N. (1970). A maximization technique occurring in the statistical analysis of probabilistic functions of markov chains. *The annals of mathematical statistics*, 41(1):164--171.

Baydin, A. G., Pearlmutter, B. A., Radul, A. A., and Siskind, J. M. (2015). Automatic differentiation in machine learning: a survey. *arXiv preprint arXiv:1502.05767*.

Bengio, Y. et al. (2009). Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1--127.

Bergstra, J., Bastien, F., Breuleux, O., Lamblin, P., Pascanu, R., Delalleau, O., Desjardins, G., Warde-Farley, D., Goodfellow, I., Bergeron, A., et al. (2011). Theano: Deep learning on gpus with python. In *NIPS 2011, BigLearning Workshop, Granada, Spain*, volume 3. Citeseer.

Bishop, C. (2007). Pattern recognition and machine learning (information science and statistics), 1st edn. 2006. corr. 2nd printing edn. *Springer, New York*.

Blei, D. M., Ng, A. Y., and Jordan, M. I. (2003). Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993--1022.

Brown, P. F., Desouza, P. V., Mercer, R. L., Pietra, V. J. D., and Lai, J. C. (1992). Class-based n-gram models of natural language. *Computational linguistics*, 18(4):467--479.

Ciregan, D., Meier, U., and Schmidhuber, J. (2012). Multi-column deep neural networks for image classification. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 3642--3649. IEEE.

Collobert, R., Bengio, S., and Mariéthoz, J. (2002). Torch: a modular machine learning software library. Technical report, Idiap.

Devijver, P. A. (1985). Baum's forward-backward algorithm revisited. *Pattern Recognition Letters*, 3(6):369--373.

Fine, S., Singer, Y., and Tishby, N. (1998). The hierarchical hidden markov model: Analysis and applications. *Machine learning*, 32(1):41--62.

Graves, A. (2013). Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*.

Harte, D. (2016). *HiddenMarkov: Hidden Markov Models.* Statistics Research Associates, Wellington. R package version 1.8-7.

Hochreiter, S. (1998). The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107--116.

Hutchens, J. L. and Alder, M. D. (1998). Introducing megahal. In *Proceedings of the Joint Conferences on New Methods in Language Processing and Computational Natural Language Learning*, pages 271--274. Association for Computational Linguistics.

Kohavi, L. (2014). Internet threats trend report. *CYREN, Inc., October (http://www.cyren.com/tl_files/downloads/CYREN_ Q3_ 2014_ Trend_ Report.pdf)*.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998a). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278--2324.

LeCun, Y., Cortes, C., and Burges, C. J. (1998b). The mnist database of handwritten digits.

Macedo, Y. (2016). Directed dialogue generation using n-gram models. *Unpublished.*

Martens, J. and Sutskever, I. (2012). Training deep and recurrent networks with hessian-free optimization. In *Neural networks: Tricks of the trade*, pages 479--535. Springer.

Moon, T. K. (1996). The expectation-maximization algorithm. *IEEE Signal processing magazine*, 13(6):47--60.

Murphy, K. P. (2012). *Machine learning: a probabilistic perspective*. MIT press.

Paiement, J.-F., Eck, D., and Bengio, S. (2005). A probabilistic model for chord progressions. In *Proceedings of the Sixth International Conference on Music Information Retrieval (ISMIR)*, number EPFL-CONF-83178.

Pearlmutter, B. A. (1995). Gradient calculations for dynamic recurrent neural networks: A survey. *IEEE Transactions on Neural networks*, 6(5):1212--1228.

Russell, S. and Norvig, P. (1995). Artificial intelligence, a modern approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, 25:27.

Saxena, S., Brémond, F., Thonnat, M., and Ma, R. (2008). Crowd behavior recognition for video surveillance. In *International Conference on Advanced Concepts for Intelligent Vision Systems*, pages 970--981. Springer.

Tu, S. (2015). Derivation of baum-welch algorithm for hidden markov models.

Appendix A

SMM Formulation

The following is a complete listing of the code which defines the SMM. It is written in our Domain Specific Language, and from it our compiler creates a working Gradient Descent Optimizer.

```


// Manages probabilities stored as real values thar are squared and sum-
    normalized
norm :: {
    vector (float * in -> float * out){
       for (i until in.size) out[i] = in[i]^0.5;
    }
    matrix (float ** in -> float ** out) {
       for(i until in.size) norm::vector(in[i] -> out[i]);
    }
    matrix3D (float *** in -> float *** out) {
       for(i until in.size) norm::matrix(in[i] -> out[i]);
    }
}

denorm :: {
    vector (float * in -> float * out) : extern ;
    matrix (float ** in -> float ** out) : extern ;
    matrix3D (float *** in -> float *** out) : extern ;
}
/&

namespace denormAux {

    inline float epsilon(){ return 0.0001f; };
```

```

template<bool w_buffered, bool p_buffered>
inline void denorm_forward_aux(
    util::array::array_t<float, 1, w_buffered> w,
    util::array::array_t<float, 1, p_buffered> p
){
    float sum = 0;
    for (size_t i : p.range()) sum += p[i] = w[i]*w[i];
    float norm = (1 - w.size() * epsilon())/sum;
    for (size_t i : p.range()) p[i] = norm * p[i] + epsilon();
}
template<bool w_buffered, bool p_buffered, size_t dims>
inline void denorm_forward_aux(
    util::array::array_t<float, dims, w_buffered> w,
    util::array::array_t<float, dims, p_buffered> p
){
    for (size_t i : w.range()) denorm_forward_aux(w[i], p[i]);
}

template<bool clearDerivative = true, bool w_buffered, bool
    d_dw_buffered, bool d_dp_buffered>
inline void denorm_backward_aux(
    util::array::array_t<float, 1, w_buffered> w,
    util::array::array_t<float, 1, d_dw_buffered> d_dw,
    util::array::array_t<float, 1, d_dp_buffered> d_dp
){
    if (clearDerivative) d_dw.clear();

    float sumSq = 0, wSumSq = 0;
    for (size_t i : w.range()) {
        float sq = w[i] * w[i];
        sumSq += sq;
        wSumSq += d_dp[i] * sq;
    }
    wSumSq /= sumSq;

    float norm = (2.0f / sumSq) * (1 - w.size() * epsilon());
    for (size_t i : d_dw.range())
        d_dw[i] += norm * w[i] * (d_dp[i] - wSumSq);
}
template<bool clearDerivative = true, bool w_buffered, bool
    d_dw_buffered, bool d_dp_buffered, size_t dims>
inline void denorm_backward_aux(
    util::array::array_t<float, dims, w_buffered> w,
    util::array::array_t<float, dims, d_dw_buffered> d_dw,
    util::array::array_t<float, dims, d_dp_buffered> d_dp
){
    for (size_t i : w.range()) denorm_backward_aux<clearDerivative>(w[i], d_dw[i], d_dp[i]);
}

template<size_t dims>
struct genericDenormMem{
    typedef typename util::array::array_t<float, dims, false> noBuff_t;
    typedef typename util::array::array_t<float, dims,  true> buffed_t;

    noBuff_t inNoBuff;
    buffed_t inBuffed;

    template<bool buffered, typename = typename std::enable_if<buffered == false>::type>
    noBuff_t getVal(){ return inNoBuff; }
    template<bool buffered, typename = typename std::enable_if<buffered == true>::type>
    buffed_t getVal(){ return inBuffed; }

    template<bool buffered, typename = typename std::enable_if<buffered == false>::type>
    void setVal(noBuff_t src){ inNoBuff = src; }
    template<bool buffered, typename = typename std::enable_if<buffered == true>::type>
    void setVal(buffed_t src){ inBuffed = src; }
};

// Denormalizer memorizes the source array by reference. It is assumed to be unchanging!
template<bool in_buffered, bool out_buffered, size_t dims>
inline void denorm_forward(
    genericDenormMem<dims> * mem,
    util::array::array_t<float_t, dims, in_buffered> in,
    util::array::array_t<float_t, dims, out_buffered> out
) {
    mem->template setVal<in_buffered>(in);
    denormAux::denorm_forward_aux(in, out);
}
template<bool d_din_buffered, bool d_dout_buffered, size_t dims>
inline void denorm_backward(
    genericDenormMem<dims> * mem,
        util::array::array_t<float_t, dims, d_din_buffered> d_din,
        util::array::array_t<float_t, dims, d_dout_buffered> d_dout
    ) {
        denormAux::denorm_backward_aux(mem->template getVal<d_din_buffered
            >(), d_din, d_dout);
    }

}

struct denorm_vector_memt   : public denormAux::genericDenormMem<1> {};
struct denorm_matrix_memt   : public denormAux::genericDenormMem<2> {};
struct denorm_matrix3D_memt : public denormAux::genericDenormMem<3> {};

template<typename in_t, typename out_t>
inline void denorm_vector_forward(denorm_vector_memt * mem, in_t &in,
    out_t &out) { denormAux::denorm_forward(mem, in, out); }
template<typename in_t, typename out_t>
inline void denorm_vector_backward(denorm_vector_memt * mem, in_t &d_din,
     out_t &d_dout) { denormAux::denorm_backward(mem, d_din, d_dout); }
template<typename in_t, typename out_t>
inline void denorm_matrix_forward(denorm_matrix_memt * mem, in_t &in,
    out_t &out) { denormAux::denorm_forward(mem, in, out); }
template<typename in_t, typename out_t>
inline void denorm_matrix_backward(denorm_matrix_memt * mem, in_t &d_din,
     out_t &d_dout) { denormAux::denorm_backward(mem, d_din, d_dout); }
template<typename in_t, typename out_t>
inline void denorm_matrix3D_forward(denorm_matrix3D_memt * mem, in_t &in,
     out_t &out) { denormAux::denorm_forward(mem, in, out); }
template<typename in_t, typename out_t>
inline void denorm_matrix3D_backward(denorm_matrix3D_memt * mem, in_t &
    d_din, out_t &d_dout) { denormAux::denorm_backward(mem, d_din, d_dout)
    ; }

&/

MM{

    init(int states){
        float q0[states] = random()^2;
        float w[states][states] = random()^2;
    }

    getP(-> float ** p){ denorm::matrix(w -> p); }
    getP0(-> float * p0){ denorm::vector(q0 -> p0); }
```

```
}


StandardMMSequence{

   init(int states, int length){
      float q0[states] = random();
      float w[length][states][states] = random();
   }

   getP(-> float *** p){ denorm::matrix3D(w -> p); }
   getP0(-> float * p0){ denorm::vector(q0 -> p0); }
}
SplitMMSequence{

   init(int states, int length){
      float q0[states] = random();
      float wNovelty = random();
      float wPrior[states] = random();
      float wPrev[states][states] = random();
      float wSeq[length][states][states] = random();
   }

   // Sets / Retrieves the initial and iteration probability matrix
   getP(-> float *** p){
      float aux[wSeq.size][wSeq[_0].size][wSeq[_0][_1].size] = wNovelty +
           wPrior[_2] + wPrev[_1][_2] + wSeq[_0][_1][_2];
      denorm::matrix3D(aux -> p);
   }
   getP0(-> float * p0){ denorm::vector(q0 -> p0); }

}
TiedMMSequence{

   init(int states, int length){
      float q0[states] = random();
      float w[states][states] = random();
   }

   // Retrieves the initial and iteration probability matrix
   getP(-> float *** p){
      denorm::matrix(w -> p[0]);
      for (i from 1 until p.size)
         for(j until p[0].size)
```

```
            for(k until p[0][i].size)
               p[i][j][k] = p[0][j][k];
    }
    getP0(-> float * p0){ denorm::vector(q0 -> p0); }
}

// Updates a state distribution
MMForward :: {
    dist (float * dist, float * obs, float ** p, float * distAux -> float
        sumP){
        sumP = 0;
        distAux.clear;
        for (i until dist.size) {
            float normSrc = dist[i] * obs[i];
            for (j until distAux.size) distAux[j] += normSrc * p[i][j];
            sumP += normSrc;
        }
        float norm = 1 / sumP;
        for (i until dist.size) dist[i] = distAux[i] * norm;
    }
    distSeq (float * dist, float ** obs, float *** p, float * distAux, int
         t0, int T -> float logP){
        logP = 0;
        for (t from t0 until T) logP += log(MMForward::dist(dist, obs[t], p
            [t % p.size], distAux));
    }
    // Propagate the source distribution if the observation probability is
         discrete
    discrete (float * dist, int obsIdx, float ** p -> float sumP){
        for (i until dist.size) dist[i] = p[obsIdx][i];
        sumP = dist[obsIdx];
    }
    discreteSeq (float * dist, int * obsIdx, float *** p, int t0, int T ->
         float logP){
        if (T == t0) logP = 0;
        else {
            float epsilon = 0.00001;
            logP = log(dist[obsIdx[t0]] + epsilon);
            for (t from t0 until T - 1) logP += log(p[t % p.size][obsIdx[t
                ]][obsIdx[t+ 1]] + epsilon);
            int lastBlock = (T - 1) % p.size;
            for (i until dist.size) dist[i] = p[lastBlock][obsIdx[T-1]][i];
        }
    }
```

```
}

// Stores the transition weights used in a SMM
SMMTransitions{

    init(int * stateCnt, int * dims, int * length, int * type){

        int uniqueTypes = 3;
        int nodes = stateCnt.size;
        int typeCounts[uniqueTypes];
        for (i until nodes) typeCounts[type[i]]++;

        int typeIndexToNodeMap[uniqueTypes][typeCounts[_0]];
        int nodeToTypeIndexMap[nodes][2];
        typeCounts.clear;
        for (i until nodes) {
            int currType = type[i];
            int currOffset = typeCounts[currType];
            typeIndexToNodeMap[currType][currOffset] = i;
            nodeToTypeIndexMap[i][0] = currType;
            nodeToTypeIndexMap[i][1] = currOffset;
            typeCounts[currType]++;
        }

        StandardMMSequence standard[typeCounts[0]][stateCnt[
            typeIndexToNodeMap[0][_0]]](dims[typeIndexToNodeMap[0][_0]],
            length[typeIndexToNodeMap[0][_0]]);
        SplitMMSequence split[typeCounts[1]][stateCnt[typeIndexToNodeMap
            [1][_0]]](dims[typeIndexToNodeMap[1][_0]], length[
            typeIndexToNodeMap[1][_0]]);
        TiedMMSequence tied[typeCounts[2]][stateCnt[typeIndexToNodeMap[2][
            _0]]](dims[typeIndexToNodeMap[2][_0]], length[typeIndexToNodeMap
            [2][_0]]);
    }

    getP(-> float ***** p){

        //float p[nodeCnt][stateCnt[_0]][DT[_0]][dims[_0]][dims[_0]];
        for(i until standard.size)
            for(j until standard[i].size) standard[i][j].getP(-> p[
                typeIndexToNodeMap[0][i]][j]);
        for(i until split.size)
            for(j until split[i].size) split[i][j].getP(-> p[
                typeIndexToNodeMap[1][i]][j]);
```

```
      for(i until tied.size)
         for(j until tied[i].size) tied[i][j].getP(-> p[
            typeIndexToNodeMap[2][i]][j]);
   }
   getP0(-> float *** p0){

      //float p0[nodeCnt][stateCnt[_0 + 1]][stateCnt[_0 + 1]];
      for(i until standard.size)
         for(j until standard[i].size) standard[i][j].getP0(-> p0[
            typeIndexToNodeMap[0][i]][j]);
      for(i until split.size)
         for(j until split[i].size) split[i][j].getP0(-> p0[
            typeIndexToNodeMap[1][i]][j]);
      for(i until tied.size)
         for(j until tied[i].size) tied[i][j].getP0(-> p0[
            typeIndexToNodeMap[2][i]][j]);
   }
}

// Creates the computational graph required for SMM computation
SMMTree{

   init(int * _dims, int * _stateCnt, int * _DT, int * _CDT, int **
       _inputMap, int ** _children){
      int nodeCnt = _children.size;
      int dims[nodeCnt] = _dims[_0];
      int stateCnt[nodeCnt] = _stateCnt[_0];
      int DT[nodeCnt] = _DT[_0];
      int CDT[nodeCnt] = _CDT[_0];
      int inputMap[nodeCnt][2] = _inputMap[_0][_1];
      int children[nodeCnt][_children[_0].size] = _children[_0][_1];
   }


   fn_init(int nID, int t0, int T -> int * tBegin, int * tEnd) {
      tBegin[nID] = t0 / CDT[nID]; t0 %= CDT[nID];
      tEnd[nID] = T / CDT[nID]; T %= CDT[nID];
      for (c until children[nID].size)
         fn_init(children[nID][c], t0, T -> tBegin, tEnd);
   }
   fn_rec(
      int nID, int t0, int T, int isBegin, int isEnd, int * tBegin, int *
          tEnd,
      float ** u, float *** up, float *** s, float *** sAux,
    float *** obsDist, int ** obsSeq, float ** inMetaObsProb, float **
        outMetaObsProb,
    float ***** p
->
    float logL
){

    // Clear the above layer's meta-obsProb
    if (!isBegin) outMetaObsProb[nID].clear;

    // If this is the bottom layer do standard markov computation
    if(children[nID].size == 0){

        // Propagate the leaf node using either sequence or distribution
             data
        int inputType = inputMap[nID][0];
        int inputSrc = inputMap[nID][1];
        //      Distribution based observations
        if (inputType == 0){
            for (t from t0 until T){
                float sumP = 0;
                float sumMeta = 0;
                for (i until stateCnt[nID]){
                    float auxf = MMForward::dist(s[nID][i], obsDist[
                        inputSrc][t], p[nID][i][t % p[nID][i].size], sAux[
                        nID][i]);
                    outMetaObsProb[nID][i] += log(auxf);
                    float metai = exp(outMetaObsProb[nID][i]) * u[nID][i];
                    sumP += auxf * metai;
                    sumMeta += metai;
                }
                logL += log(sumP/sumMeta);
            }
        //      Sequence based observations
        } else if (inputType == 1){
            for (i until stateCnt[nID])
                outMetaObsProb[nID][i] += MMForward::discreteSeq(s[nID][i
                    ], obsSeq[inputSrc], p[nID][i], t0, T);
        }

    // Otherwise, do SMM recursion
    } else for( t from t0 until T + isEnd ) {
        int tt = t % DT[nID];
        int firstChildID = children[nID][0];

        // Recompute the layer's children's universal probabilities
        u[firstChildID].clear;
        for (i until stateCnt[nID])
            for (j until dims[nID])
                u[firstChildID][j] += u[nID][i] * s[nID][i][j];
        for (c from 1 until children[nID].size){
            int childID = children[nID][c];
            for (j until dims[nID]) u[childID][j] = u[firstChildID][j];
        }

        // Recurse on lower layers
        int new_isBegin; if (isBegin && t == t0) new_isBegin = 1; else
            new_isBegin = 0;
        int new_isEnd; if (isEnd && t == T) new_isEnd = 1; else
            new_isEnd = 0;
        for (childIdx until children[nID].size) {
            int childID = children[nID][childIdx];
            int new_baseT = t * DT[childID];
            int new_t0 = new_baseT + new_isBegin * tBegin[childID];
            int new_T = new_baseT; if (new_isEnd) new_T += tEnd[childID];
                else new_T += DT[childID];
            fn_rec(childID, new_t0, new_T, new_isBegin, new_isEnd, tBegin
                , tEnd, u, up, s, sAux, obsDist, obsSeq, inMetaObsProb,
                outMetaObsProb, p -> logL);
        }

        // Iterate the models at the present layer once
        if(t != T){

            // Compute the input meta-obsProb
            for (i until dims[nID])
                inMetaObsProb[nID][i] = outMetaObsProb[firstChildID][i];
            if (children[nID].size > 1){
                for (i from 1 until children[nID].size){
                    int childID = children[nID][i];
                    for (j until dims[nID])
                        inMetaObsProb[nID][j] *= outMetaObsProb[childID][j];
                }
                float sum = 0;
                for (i until dims[nID]) sum += inMetaObsProb[nID][i];
                float norm = 1/sum;
                for (i until dims[nID]) inMetaObsProb[nID][i] *= norm;
            }

        // Compute the universal transition probability
        up[nID].clear;
        for (i until stateCnt[nID]){
            float ui = u[nID][i];
            for (j until dims[nID])
                for (k until dims[nID])
                    up[nID][j][k] += p[nID][i][tt][j][k] * ui;
        }

        // Merge children
        for (c until children[nID].size){
            int childID = children[nID][c];
            for(i until stateCnt[childID]){
                float norm = u[childID][i] * inMetaObsProb[nID][i];
                for(j until dims[childID])
                    sAux[childID][i][j] = s[childID][i][j] * norm;
            }
            s[childID].clear;
            for (iSrc until stateCnt[childID]) {
                for (iDst until stateCnt[childID]) {
                    float p_srcDst = up[nID][iSrc][iDst];
                    for (j until dims[childID])
                        s[childID][iDst][j] += sAux[childID][iSrc][j] *
                            p_srcDst;
                }
            }
            for (i until stateCnt[childID]) {
                float sum;
                for (j until dims[childID]) sum += s[childID][i][j];
                float norm = 1 / sum;
                for (j until dims[childID]) s[childID][i][j] *= norm;
            }
        }

        // Iterate
        float sumU = 0;
        for ( i until stateCnt[nID]) {
            float sumP;
            MMForward::dist(s[nID][i], inMetaObsProb[nID], p[nID][i][
                tt], sAux[nID][i] -> sumP);
            outMetaObsProb[nID][i] += log(sumP);
        }
        float normU = 1/sumU;
```

```
                for ( i until stateCnt[nID]) u[nID][i] *= normU;
            }
        }

        if (!isEnd /*&& nID > 0*/){

            // Normalize the metaObsProb out of log-space
            float maxObsProb = outMetaObsProb[nID][0];
            for (i from 1 until stateCnt[nID])
                if (outMetaObsProb[nID][i] > maxObsProb) maxObsProb =
                    outMetaObsProb[nID][i];
            float sum;
            for (i until stateCnt[nID]){
                float expdVal = exp(outMetaObsProb[nID][i] - maxObsProb);
                outMetaObsProb[nID][i] = expdVal;
                sum += expdVal;
            }
            float norm = 1 / sum;
            for (i until stateCnt[nID]) outMetaObsProb[nID][i] *= norm;
        }
    }
}

SMM{

    init(int * _dims, int * _stateCnt, int * _DT, int * _CDT, int *
        _transitionType, int ** _inputMap, int ** _children){
        int nodeCnt = _stateCnt.size;
        int stateCnt[_stateCnt.size] = _stateCnt[_0];
        int dims[_dims.size] = _dims[_0];
        int DT[_DT.size] = _DT[_0];
        SMMTransitions transitions(_stateCnt, _dims, _DT, _transitionType);
        SMMTree computationModel(_dims, _stateCnt, _DT, _CDT, _inputMap,
            _children);
    }
    fn(float *** obsDist, int ** obsSeq, int t0, int T -> float logL){
        float p[nodeCnt][stateCnt[_0]][DT[_0]][dims[_0]][dims[_0]];
        float s[nodeCnt][stateCnt[_0]][dims[_0]];
        transitions.getP(->p);
        transitions.getP0(->s);

        int tBegin[nodeCnt], tEnd[nodeCnt];
        computationModel.fn_init(0, t0, T -> tBegin, tEnd);

        float u[nodeCnt][stateCnt[_0]], up[nodeCnt][dims[_0]][dims[_0]];
        float inMetaObsProb[nodeCnt][dims[_0]], outMetaObsProb[nodeCnt][
            stateCnt[_0]];
        float sAux[nodeCnt][stateCnt[_0]][dims[_0]];

        u[0][0] = 1;
        logL = 0;
        computationModel.fn_rec(
            0, tBegin[0], tEnd[0], 1, 1, tBegin, tEnd,
            u, up, s, sAux, obsDist, obsSeq, inMetaObsProb, outMetaObsProb,
                p -> logL
        );
    }
}


// Collection of normal distributions
normalMixture{
    init(int _dims, float muMin, float muMax, float sigmaMin, float
        sigmaMax){
        int dims = _dims;
        float mu[dims] = random()*(muMax - muMin) + muMin;
        float tau[dims] = 1/(random()*(sigmaMax - sigmaMin) + sigmaMin)^2;
    }

    seqToDiscrete(float * sequenceIn -> float ** stateProbsOut){
        float auxA[dims] = (tau[_0]/6.283)^0.5;
        float auxB[dims] = -tau[_0]/2;
        for (t until sequenceIn.size){
            for (d until dims){
                float val = auxA[d] * exp(auxB[d] * (sequenceIn[t] - mu[d])
                    ^2) + 0.0001;
                stateProbsOut[t][d] = val;
            }
        }
    }
    discreteToDist(float ** stateProbsIn, int T, float distMin, float
        distMax, int distDims -> float ** distOut) {
        float auxA[dims] = (tau[_0]/6.283)^0.5;
        float auxB[dims] = -tau[_0]/2;
        for (t until T){
            for (d until distDims){
                float x = distMin + d * (distMax - distMin) / distDims;
                distOut[t][d] = 0;
            for (s until dims)
                distOut[t][d] += stateProbsIn[t][s] * auxA[s] * exp(auxB[s
                    ] * (x - mu[s])^2) + 0.0001;
        }
    }
  }
}

normalMixtureSMM{

    init(
        int * dims, int * stateCnt, int * DT, int * CDT, int *
            transitionType, int ** inputMap, int ** children,
        int * _inputDims, float * muMin, float * muMax, float * sigmaMin,
            float * sigmaMax
    ){
        int inputDims[_inputDims.size] = _inputDims[_0];
        normalMixture NM[_inputDims.size](_inputDims[_0], muMin[_0], muMax[
            _0], sigmaMin[_0], sigmaMax[_0]);
        SMM M(dims, stateCnt, DT, CDT, transitionType, inputMap, children);
    }

    fn(float ** x, int t0, int T -> float logL){
        float obsDist[NM.size][x[_0].size][inputDims[_0]];
        int obsSeq[0][0];
        for (i until NM.size)
            NM[i].seqToDiscrete(x[i] -> obsDist[i]);
        M.fn(obsDist, obsSeq, t0, T -> logL);
    }
}
```