LEARNING PREDICTION AND ABSTRACTION IN PARTIALLY OBSERVABLE MODELS

Marc Gendron-Bellemare

School of Computer Science
McGill University, Montréal

June 2007

A Thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfilment of the requirements for the degree of
Master of Science

Acknowledgments

I would like to thank Doina Precup, my supervisor, for coping with a countless number of tangents which I took over the course of my studies with her as my advisor. Especially important to me is the leeway that I was given in pursuing various ideas, which finally led this to thesis. I would also like to thank Dr. George Berkeley and Mr. David Hume for providing enlightening discussions at various points throughout my research. In no particular order, Cosmin Paduraru helped me with a proof of convergence; Philipp Keller provided many arguments against and for a variety of thesis-related topics; Robert Vincent was part the original discussion which resulted in this thesis.

I would also like to thank Marc Lanctot, who cleared the graduate studies path before me so that I could walk it safely. My thanks also go to all the graduate students of the RL Laboratory who graduated before me and prepared me for this endeavour.

Lastly, I thank Julia Grav for giving me moral support in bad times, encouragement in good times and an unrelenting belief in my work every step of the way.

Abstract

Markov models have been a keystone in Artificial Intelligence for many decades. However, they remain unsatisfactory when the environment modeled is partially observable. There are pathological examples where no history of fixed length is sufficient for accurate decision making. On the other hand, working with a hidden state (such as in POMDPs) has a high computational cost. In order to circumvent this problem, I suggest the use of a context-based model. My approach replaces strict transition probabilities by a linear approximation of probability distributions. The method proposed provides a trade-off between a fully and partially observable model. I also discuss improving the approximation by constructing history-based features. Simple examples are given in order to show that the linear approximation can behave like certain Markov models. Empirical results on feature construction are also given to illustrate the power of the approach.

Résumé

Depuis plusieurs décennies, les modèles de Markov forment une des bases de l'Intelligence Artificielle. Lorsque l'environnement modélisé n'est que partiellement observable, cependant, ceux-ci demeurent insatisfaisants. Dans certains cas, un historique de taille finie n'est pas suffisant pour une prise de décision correcte. D'un autre côté, faire appel au concept d'état caché (tel que pour les POMDPs) implique un coût computationnel plus élevé. Afin déviter ce problème, je propose l'utilisation d'un modèle se servant du contexte. Mon approche substitue une approximation linéaire des distributions de probabilités aux probabilités de transition strictes d'un modéle de Markov. La méthode proposée permet un compromis entre un modéle partiellement observable et un modéle complètement observable. Je traite aussi de la construction d'éléments liés à un historique afin d'améliorer l'approximation linéaire. Des exemples restreints sont présentés afin de monter qu'une approximation linéaire de certains modèles de Markov peut être atteinte. Des résultats empiriques au niveau de la construction d'éléments sont aussi présentés afin d'illustrer les bénéfices de mon approche.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

Introduction

Machine Learning is a branch of Artificial Intelligence which is concerned with algorithms that can discover patterns in data. In general, the goal is either to model future data (to predict it) or to improve the performance of a system from experience. In this thesis, I am particularly interested in studying how we can learn to predict sequences of symbols, also called time series. The simplest type of predictive models, *Markov Models*, work well for learning sequences from simple histories, but fail to transfer knowledge to new tasks efficiently. Part of the problem is that they are too strict in their definition of history, which is used to predict the future. It therefore seems crucial to look for a way of relaxing the way we look at the past when attempting to produce a prediction. My proposal is to use a more compact representation of the past than used by the above models in order to generalize better.

Most of the work on prediction either assumes a model of the abstract structure of the problem (such as HMMs [Rabiner, 1989]) or assumes that it will be built implicitly as part of the learning algorithm [Elman, 1990].

In this thesis I develop an algorithm which can achieve good prediction capacities. Although these associations do not help in the context of a single task, we can hope that they allow the system to carry abstraction information across tasks. The algorithm which I propose is relatively simple, making use of ideas from Sparse Distributed Memories [Kanerva, 1993] and neural networks. The first version of the algorithm uses a compact encoding of the history into a single, real-valued vector in order to predict sequences. This portion of my thesis has originally been described in [Bellemare and Precup, 2007]. The second version of the algorithm improves this work by adding the capacity to construct more complicated features which help prediction.

A review of existing ideas related to my proposal is given in Chapter 2, where I discuss the above algorithms and more recent work in sequence prediction. I investigate state-based representations, followed by suffix trees which attempt to depart from this notion of state. I also explain neural networks, related to my algorithm in their nature, and Sparse Distributed Memories, from which I borrow the notion of hard locations. In Chapter 3 I describe the basic version of the algorithm. In Chapter 4 I give experimental results concerning this basic algorithm that show that it can correctly model simple tasks. In Chapter 5 I detail a feature construction algorithm which gives additional representational power to the system, and give a proof of convergence for the feature construction process. Chapter 6 gives experimental results for this feature construction mechanism which shows that it outperforms the basic predictive model in many ways. Finally, 7 discusses the advantages and weaknesses of the algorithm and suggests a few lines of inquiries which may yield improved results.

1.1. Statement of Originality

Portions of this thesis have been previously published in the peer-reviewed conference proceedings of the *Twentieth International Joint Conference on Artificial Intelligence*.

CHAPTER 2

Background

Chapter Outline

In this chapter, I provide background information on temporal prediction and learning methods for such models.

2.1. Temporal Prediction

To predict is defined in the Merriam-Webster dictionary as "to declare or indicate in advance; *esp*: foretell on the basis of observation, experience or scientific reason"[1]. This is interesting in two ways: first, it suggests that an act of prediction should provide information about the future, without actually having observed it. Most importantly, the emphasis is put on the notion of experience and reason. Prediction does not need to be deterministic, but it certainly needs to provide reasonable estimates of future events. Furthermore, an agent which must take decisions in an environment must necessarily have a predictive model of the world incorporated in its system, implicitly or explicitly.

I am most interested in predicting sequences of observations, rather than one-shot events (for example, determining whether a patient has a given disease). The latter class of predictions has been extensively studied; belief networks [Neal, 1992] are geared towards this type of task. Temporal prediction can in fact be tackled using the similar technology of dynamic belief networks [Boyen and Koller, 1998]; however, the structure of the belief network often needs to be carefully crafted in advance. *Temporal prediction* as discusse in

[1]Merriam-Webster's Collegiate Dictionary, Tenth Edition. Merriam-Webster Inc., Springfield MA, 1999, p. 918.

this thesis can be viewed entirely as the following: prediction is the act of the determining as exactly as possible what event or events will come next given a certain *history* (a sequence of past events). This notion is crucial to intelligence as it allows an agent to plan the consequences of its actions over time. Another use of a good prediction model is to verify whether a given sequence behaves as expected, for example when monitoring factory processes over time. This field is referred to as anomaly detection. Before reviewing the various proposals for coping with temporal prediction, I will present a glossary of most variables used throughout this chapter. I have tried as much as possible to use the same variable for similar concepts, although for the sake of clarity have sometimes chosen to keep the notation from the original articles.

Name	Meaning
t	A time variable, used to represent discrete time steps
S, s, S_t	A state variable and an instance of a state; the state at time t
Z, z	An observation, in partially observable models
A, a	An action taken by an agent
C_i	A hard location or 'center'
D_i	An output vector corresponding to C_i
H_t	A history: the sequence of states $s_1, s_2, \ldots, s_t$ or observations
n	The number of possible states
m	The number of possible observations
k	The number of past states considered by the algorithm (e.g. k^{th} order Markov Model)
T	The length of an episode (sequence of states until termination)
L_C	The size of an input vector, or of an observation
L_D	The size of an output vector
$\vec{x}$	An input to the algorithm
β, $\vec{\beta}$	Some auxiliary output produced internally by the algorithm
y, $\vec{y}$	An output of the algorithm
o	A desired target value, used during training
$\vec{b}_t$	A belief vector at time t
W, W^l, $W_{i,j}$	A set or vector of weights, or a weight from i to j
κ	A normalizing constant

2.2. Markov Models

A standard framework for temporal prediction is that of Markov Models [Shannon, 1951]. In its simplest form, a Markov model contains a set of states in which the system can be. Given a discrete time scale, S_t is a random variable describing the state of the system at time t. Let $H_t = S_1, S_2, \ldots, S_t$ be the history at time t. For a simple Markov Chain, we have

$$P\{S_t|H_{t-1}\} = P\{S_t|S_1, S_2, \ldots, S_{t-1}\} = P\{S_t|S_{t-1}\} \tag{2.1}$$

This is a strong assumption, but which can be interpreted as follows: the state S_{t-1} contains all of the information about the system to be able to predict S_t. This is called the transition probability from state S_{t-1} to S_t. Classical physics, for example, falls under this model if it is assumed that the forces in presence are known - in fact, the system then becomes deterministic. In general, one can define a k^{th} order model as one where the distribution of S_t only depends on the last k events in the history.

The simplest way of learning in a Markov Model is simply to use a Maximum Likelihood approach. Given n possible states such that $S_t \in \{1, 2, \ldots, n\}$, and assuming a long sequence of states given as data, define $(N_i|s_1, s_2, \ldots, s_k)$ the number of times that state i is observed when the last k states were exactly $s_1, s_2, \ldots, s_k$. Then the maximum likelihood estimate for $P\{S_t = i|S_{t-1} = s_1, S_{t-2} = s_2, \ldots, S_{t-k} = s_k\}$ is

$$\hat{P}\{S_t = i|S_{t-1} = s_1, S_{t-2} = s_2, \ldots, S_{t-k} = s_k\} = \frac{(N_i|s_1, s_2, \ldots, s_k)}{\sum_{s'_1, s'_2, \ldots, s'_k} (N_i|s'_1, s'_2, \ldots, s'_k)} \tag{2.2}$$

This is a purely *tabular* method: the probability of each observation after each history is represented independently, as an entrry in a table and no dependence assumption between different histories is made. As such, this method requires a number of parameters exponential in the length of the history k, which makes it too cumbersome for most applications. Nevertheless, this framework is intuitively very powerful exactly because it allows complete independence between different histories.

2.3. Hidden Markov Models

A common relaxation of the problem, called Hidden Markov Models, is to assume that there is no way of directly knowing the state of the system. Rather, the system emits an *observation* Z_t at each time step (Fig. 2.1). In addition to S_t being determined solely by S_{t-1}, it is also assumed that $P\{Z_t|S_t, Z_{t-1}, S_{t-1}, \ldots, S_1\} = P\{Z_t|S_t\}$ - states emit observations independently of previous observations or states.

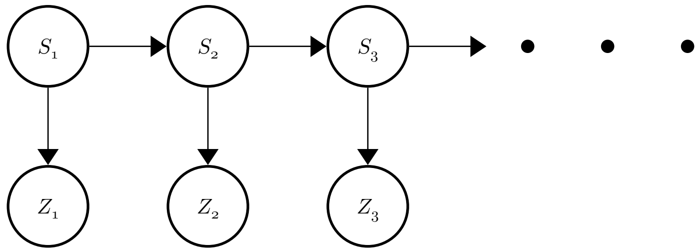

FIGURE 2.1. A Hidden Markov Model, where Y_t are observations and X_t, states, both occuring over time.

This framework is surveyed in [Rabiner, 1989]; it is often seen as a better model of real life systems, where there is only access to sensor data and never to the "true" state of the world. Unfortunately, it is well known that in general, no history of fixed length is sufficient to predict future events.

A Hidden Markov Model is described by the following parameters: $p_{i,j} = P\{S_t = j|S_{t-1} = i\}$, $o_i(k) = P\{Z_t = k|S_t = i\}$ and $\pi_i = P\{S_1 = i\}$, where S_1 is the initial state.

The main approach to using HMMs consists in working directly with the state by keeping track of a *belief vector* $\vec{b_t}$. This vector is defined as $\vec{b_t}(i) = P\{S_t = i|H_{t-1}\}$. Clearly, $\vec{b_1}(i) = \pi_i$. Then obtaining $\vec{b_t}$ simply involves applying the following update equation:

$$\vec{b_t}(i) = \frac{1}{\kappa} \sum_j p_{j,i} o_j(z_{t-1}) b_{t-1}(j) \tag{2.3}$$

κ is simply a normalizing constant such that $\vec{b_t}$ is a proper probability distribution. This equation can easily be obtained by using the Law of Total Probability. Given this, it becomes possible to predict future observations by simply considering the fact that

$$P\{Z_t|H_{t-1}\} = \sum_i P\{Z_t|S_t = i\} P\{S_t = i|H_{t-1}\} \tag{2.4}$$

2.3.1. Learning From Data

As described in Section 2.2, one task of interest is to be able to learn the parameters of the model from the data. In this case, data is usually one or many sequence(s) of observations, with either $p_{i,j}$, $o_i(k)$ or π_i - or all three - being unknown.

The main issue in this case is that, since S_t is never observed, the learning algorithm must “fantasize” it. The Baum-Welch algorithm [Rabiner, 1989] does exactly this. This algorithm is a type of Expectation-Maximization algorithm: it makes an initial guess for the parameters to be estimated, then generates fictitious state data based on the observation sequences, and re-estimates the parameters from these.

Formally, for a known observation sequence $z_1, \ldots, z_T$ of length T, define $\alpha_t(i) = P\{Z_1 = z_1, Z_2 = z_2, \ldots, Z_t = z_t, S_t = i\}$. This value can be computed inductively as follows:

$$\begin{aligned} \alpha_1(i) &= \pi_i o_i(z_1) \\ \alpha_{t+1}(j) &= \left(\sum_{i=1}^{n} \alpha_t(i) p_{i,j} o_j(z_{t+1}) \right) \quad \text{for } 1 \le t \le T-1 \end{aligned}$$

Of interest is the fact that $P\{Z_1, \ldots, Z_T\} = \sum_{i=1}^{n} \alpha_T(i)$: the probability of the observation sequence can easily (in time $O(nT)$) be computed when the model is known. Anomaly detection would require a good estimate of these probabilities. The computation of $\alpha_t(i)$ is known as the *forward pass*.

Similarly, the value $\beta_t(i) = P\{Z_{t+1} = z_{t+1}, Z_{t+2} = z_{t+2}, \ldots, Z_T = z_T | S_t = i\}$ can be computed inductively. This is known as the *backward pass*, and is given by the following equations:

$$\begin{aligned}\beta_T(i) &= 1 \\ \beta_t(i) &= \sum_{j=1}^{m} p_{i,j} o_j(z_{t+1}) \beta_{t+1}(j)\end{aligned}$$

These two sets of values can then be used to compute $\xi_t(i,j) = P\{S_t = i, S_{t+1} = j | Z_1, \ldots, Z_T\}$, through the following formula (which is similar to the belief update equation):

$$\xi_t(i,j) = \frac{1}{\kappa} \alpha_t(i) p_{i,j} o_j(z_{t+1}) \beta_{t+1}(j) \tag{2.5}$$

κ ensures that $\sum_{i,j} \xi_t(i,j) = 1$. The important thing to note is that $\xi_t(i,j)$, along with an additional variable $\gamma_t(i) = \sum_{j=1}^{n} \xi_t(i,j)$, gives us a probability of a transition occuring from state i to j or, respectively, of being in state i. As such, for a given trajectory $\sum_{t=1}^{T} \xi_t(i,j)$ represents a fictitious "count" of how many times a transition from i to j occcured. Applying the same reasoning to $\gamma_t(i)$ allows the estimation of the transition probabilities as

$$\hat{p}_{i,j} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)} \tag{2.6}$$

Equations for estimates of $o_j(k)$ and π_i can also be derived in this fashion. It is clear from this description that the learning process here is one of Expectation-Maximization. The main issue that arises is that the number of states must be specified. This is a major constraint as the system might not have sufficient representational power to properly predict observations if, say, too few states are used. On the other end of the spectrum, adding too many states would result in rote learning. For example, one solution is to assume that no state is ever visited twice, and to create sequences of states which explicitly mimic the observation sequences.

My proposed algorithm, which I will describe in Chapter 3, does not require any assumption on the underlying state. Rather, the assumption is that there are sufficiently many observations to be able to approximately predict the behavior of the system solely through the observation history. This is clearly not something that holds in general; but high sensory information tasks such as sound and vision often have more observations than states, in which case it might be useful to not have to guess the exact number of states.

2.3.2. Partially Observable Markov Decision Processes

Although so far I have discussed prediction over a sequence of observations, a large part of the research in this area is devoted to *action-conditional* predictions. Although this thesis does not attempt to work on this issue, it is important to mention the distinction between the two. In a Partially Observable Markov Decision Process (POMDP) there is an agent which must decide on an *action* A_t to take at every time step. It is in other points similar to a HMM, except that at every step, the agent receives a real-valued reward. The agent's goal is to select the action which maximizes the sum of discounted future rewards. The transition probabilities become $P\{S_{t+1} = i | S_t = j, A_t = a\}$. Observation probabilities also depend on the action taken.

This framework is related to this thesis because it is used for modelling planning tasks [Kaelbling *et al.*, 1998]. As such, it is important to obtain a good predictive model: since the true state is never observed, the system must learn to predict future rewards and observations from its history. The main difference with learning to predict in this framework, as opposed to HMMs, is that the learner picks actions, therefore influencing the sequence of observations. Nevertheless, if one assumes that actions are picked with a certainly probability which only depends on the state (and are not determined by the agent), then the framework becomes equivalent to an HMM.

2.4. Probabilistic Suffix Trees and Related Models

Another approach to learning to predict sequences of observations is that of using a *suffix tree*. It was shown in [Ron *et al.*, 1996] that Probabilistic Suffix Trees (PSTs) could be used to model Probabilistic Suffix Automata (PSAs). PSAs are of interest because they are another way of representing states and observations. Formally, a PSA is a special case of a Probabilistic Finite Automata: at every time step, the system transits from one state to

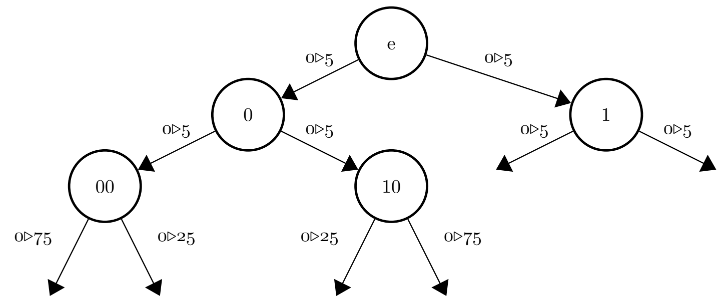

FIGURE 2.2. An example Probabilistic Suffix Tree. Left branches indicate the addition of a ‘0’ prefix, whereas right ones indicate the addition of a ‘1’ prefix.

another probabilistically. Upon entering a state, a symbol (an observation) is deterministically emitted. In the case of the PSA, more than one symbol can be emitted at every step; observations are strings of symbols of length at least 1. Furthermore, the set of possible observations has the property that it is *suffix free*: no observation string is a suffix of another. The PSA is a very concise way of viewing a sequence of observations, solely requiring that any state be distinguishable from another through some variable length string. [Ron *et al.*, 1996] also makes the assumption that the length of an observation string be bounded.

2.4.1. Probabilistic Suffix Trees

A Probabilistic Suffix Tree is a tree whose nodes are labeled with observation strings. Furthermore, it has the property that a node’s parent’s label is its label’s suffix. In order words, going down one level in the tree adds one symbol at the beginning of the nodes’ label. Finally, each branch in the tree is labeled with the probability of that the corresponding symbol was observed. Figure 2.2 shows an example of a PST.

If one were to build a PST from a PSA, then the leaves would represent states in the automaton: in this case, the distribution over the next symbol is well defined. Since a PSA’s states are all labeled by a unique suffix, there is clearly a way of modeling it using a PST. In [Ron *et al.*, 1996], the authors describe an algorithm for constructing a PST from data (assumed to be generated by a PSA). They also give PAC bounds on the required number of samples for their algorithm to converge, and show its performance on recovering

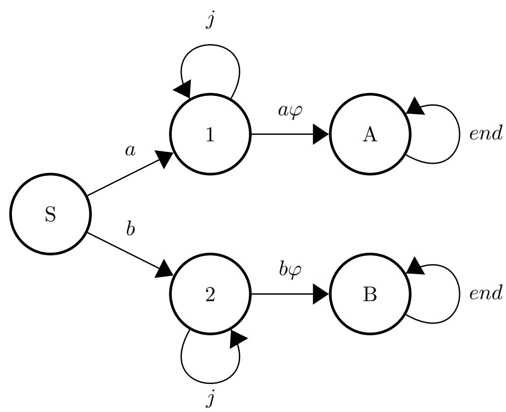

FIGURE 2.3. Example Markov Chain where no history of a fixed length is sufficient to predict the outcome with complete accuracy.

corrupted text and modeling DNA sequences. Although intuitive, this framework suffers from the drawback that a suffix tree is a memory intensive data structure, compared to more compact representations such as HMMs. It can also require many data points, since each leaf is seen as an additional state. The authors also acknowledge the fact that the PST algorithm cannot perfectly model general Probabilistic Finite Automata: typical tasks on which HMMs concisely represent the observation sequences (such as ones generated by the automata in Figure 2.3) may not be well represented by PSTs.

2.4.2. Looping Suffix Trees

In [Holmes and Isbell, 2006] the authors propose an improvement over the PST model which adds representational power to it. In short, their contribution to PSTs is to allow nodes which explicitly 'loop' back to a previous suffix. In terms of observation sequences, this is described as follows: suppose that there are two sequences, h and q, such that $h = eq$, i.e. q is a suffix of h. Furthermore, suppose that there is no predictive difference between h and q: concatenating an additional sequence p to h or q (to yield ph and pq, respectively) results in the same predicted symbol. The authors call e an *excisable* string, which adds no predictive information. They show in [Holmes and Isbell, 2006] that predictions of the form $e*q$ (the infinite set $\{q, eq, eeq, eeeq, \dots\}$) must be indistinguishable. From this, they improve Probabilistic Suffix Trees such that if the suffix eq is reached in the tree, then the algorithm 'loops back' to the node representing q. The paper focuses mainly on theoretical results concerning the representational power of the approach; little empirical evaluation has been done with it. Also, the addition of loops does not solve the problems inherent with using a suffix tree for representing history.

2.4.3. Utile Suffix Memory

Related work which also uses a suffix tree for prediction is that of [McCallum, 1995]. Utile Suffix Memory, one of the proposed algorithms, incorporate the PST in a Reinforcement Learning (RL) framework. RL is concerned with finding sequences of actions which lead to optimal reward. Reward is usually a numerical value given at each time step; a RL agent's goal is to maximize the (discounted) sum of such rewards. In [McCallum, 1995], PSTs are used to represent the history as a variable length Markov Model, but focusing on rewards. As such, each node also stores the value of a given suffix. This information - rather than predictive accuracy - is used to determine whether an additional symbol should be added at the beginning of a suffix. In order to do so, the author uses a statistical test to determine whether the distribution of rewards differs significantly between potential child nodes of a leaf. This approach is interesting because it provides a clearer goal for the predictive algorithm. It also behaves in an on-line fashion, as opposed to the algorithms proposed in [Ron *et al.*, 1996; Holmes and Isbell, 2006] which require a large amount of training sequences to determine whether to add nodes. The main issue here is that using reward for determining whether to split is not always good. For example, it might be the case that the rewards being given change over time. Although a predictive model that only uses observations will still be valid, one that is reward-driven might have to re-learn its structure. This approach also suffers from the same problem as PST, namely that suffixes of infinite length are not correctly handled. Again, Figure 2.3 is a case where this algorithm would not perform well. The reason is that both Utile Suffix Memory and PSTs cannot account for tasks in which the history *prefix* determines the distribution over future observations. Other examples can be crafted in which suffix-based models fail.

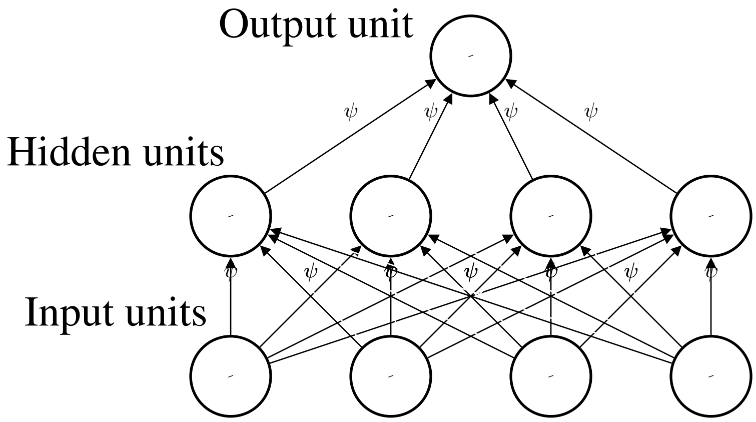

FIGURE 2.4. Example feed-forward neural network with one hidden layer.

2.5. Neural Networks for Temporal Prediction

2.5.1. Feedforward Neural Networks

The term “neural networks” represents a large family of algorithms and architectures sharing common characteristics. In general, a neural network can be decomposed into *units*, which perform some functional task based on their inputs. Inputs are either external to the system (e.g. image data) or come from other units, and are often modified by *weights*. The most frequently used type of neural network is a feedforward neural network, depicted in Figure 2.4.

Formally, a feedforward multilayer neural network has L layers, where the outputs of layer $l-1$ become the inputs of layer l. A more generalized version allows the outputs of layer $l-1$ to serve as inputs to layers $l, \ldots, L$. The first layer contains the input vector (the *input units*); the last layer, usually a single unit, produces the output. The other units are called *hidden units*. There is one matrix of weights W^l per layer, where W_i^l is the vector of weights for unit i in layer l. Let β_i^l be the output of unit i in layer l, and $\vec{\beta}^l$ the corresponding vector of such outputs. Also, let x be the input to the network. Then in the feedforward network described above, the following update equations are used to compute the output y:

$$\begin{aligned}\vec{\beta}^0 &\leftarrow x \\ \beta_i^l &\leftarrow \sigma(\vec{w}_i^l \cdot \vec{\beta}^{l-1}) \quad \text{for } l = 1 \ldots L \\ y &\leftarrow \beta^L\end{aligned}$$

σ is the standard sigmoid function, which is used to model a soft threshold function ($\sigma(-\infty) = 0$ and $\sigma(+\infty) = 1$). For a given real-valued input x, it is defined as

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.7}$$

As the above update equations suggest, computing the output of the network is simply a question of propagating the input signal up to the last layer, hence the qualificative "feedforward".

Learning in neural networks can be accomplished through the Backpropagation algorithm, originally proposed in [Rumelhart *et al.*, 1986]. The algorithm proceeds by propagating the error between the output y and a target value o backwards, from layer L to layer 1. It does so using gradient descent. Many improvements over the basic Backpropagation scheme have been suggested, such as adding a stochastic term which helps prevent local minima.

The vast literature on neural networks prohibits a complete survey of the field. This architecture and its variants have been used for all types of tasks, ranging from Reinforcement Learning in game-playing [Tesauro, 1995] to supervised clustering of data [Hadsell *et al.*, 2006]. In this case, however, the chief interest is towards temporal prediction. This task differs from other machine learning problems because the data is not independent and identically distributed (iid). This is a reformulation of the fact that the last observation is not sufficient to determine the future. For feedforward neural networks, this is an issue because there is no obvious way of defining how past inputs affect the current output. The simplest way is to expand the network's input layer to containing input vectors from times

$t-1, t-2, \ldots$ as well as the current one (at time t). By the same rationale that k^{th} order Markov Models fail on certain tasks, this must also fail.

2.5.2. Recurrent Neural Networks

Recurrent neural networks attempt to solve exactly the temporal prediction problem. Given a sequence of inputs, the algorithm should keep *contextual information* in a closed form to predict future observations accurately (the state in HMMs can be seen as a closed form context). In order to do so, such networks allow for weights over time. Formally, the inputs to unit i on layer l are a vector $< \vec{\beta_t^{l-1}}, \vec{beta_{t-1}^0}, \vec{\beta_{t-1}^1}, \ldots, \vec{\beta_{t-1}^L} >$. Adding $\vec{\beta_{t-1}^0}$ to the inputs provides a second order model; adding inputs from the other units, however, yields more representational power. The set of inputs from the previous time step is called the *context vector*. Correspondingly, recurrent neural networks can have a weight between every unit: $W_{ij}^{l_1 l_2}$ represents the weight from unit i on layer l_1 to unit j on layer l_2 (for simplicity, the previous time step can be viewed as an additional set of layers). [Elman, 1990] shows, through a series of experimental results, that such an augmented network can learn temporal structure and predict observations based on the past. He also points out to emerging clustering in how the predictions are ordered: in a simple natural language task, words which can be used in similar contexts result in similar activations by the network. This suggests that Elman's machine not only provides temporal prediction, but also a coarse form of abstraction based on these predictions. One way to interepret this is to say that some hidden units in the network learn to summarize contextual information necessary to predict the future.

Learning weights in a recurrent network, however, is more complex because the gradient of the error (as in standard backpropagation) propagates through time as well as through layers. In effect,the network should be unfolded $t-1$ times to train the weights at time t. Disregarding this leads to biased learning. Furthermore, recurrent neural networks can in general be very unstable. This is becacuse having a loop adds the possibly of resonance. If, for example, unit i on layer l has a link to itself such that $\sigma(W_{ii}^{ll}\beta_i^l) > \beta_i^l$ for some β_i^l, then over time unit i could converge to a high output value. If this happens to the whole network, then nothing useful can be recovered from its outputs. One proposed way of handling this is to limit the norm of the weights [Hochreiter and Schmidhuber, 1997], but this requires a more careful learning rule than simple backpropagation through time.

2.5.3. Dynamic Neural Networks

Work in the field of neural networks has also been geared towards constructing architectures from data. Indeed, a major flaw of neural networks is that their use often requires careful design, to avoid local minima and to optimize the way units learn. One approach which is of interest is cascade-correlation neural networks [Fahlman and Lebiere, 1990; Baluja and Fahlman, 1994]. This is only one in many, however; for example, meiosis networks [Hanson, 1990] attempt to do something very similar. There are also non-neural networks approaches to the problem, such as the one proposed in [Utgoff and Precup, 1998]. Cascade-correlation networks, however, are of interest because they relate to the approach I take to add features, which will be described in Chapter 5.

Cascade-correlation neural networks begin with a network with no hidden units, connecting the output unit(s) directly to the input layer. The system is then trained on the data in this fashion for a certain period of time, which is called the output phase. After this time, the algorithm determines whether the data is sufficiently well modeled using an error criterion. If not, it trains a number of *candidate* hidden units to maximize the correlation between their output and the error on the data. These candidate hidden units are placed right before the output layer, and are connected to all units before them (initially, only the input layer). The training of the candidate units is called the input phase. After this phase, the system picks the candidate which has the highest correlation and adds it the system as a hidden unit. Once this is done, however, its input weights are frozen, so that it stops learning. In effect, the algorithm incrementally adds features that respond to input patterns which are incorrectly modeled.

Results in [Fahlman and Lebiere, 1990] show that having such an architecture is beneficial in two ways: the network topology does not have to be specified beforehand, and learning is actually simpler since there are fewer parameters to optimize at any given time.

Unfortunately, learning in this architecture can also be harder, especially if the data is noisy. This is because the system has the double task of learning to output the correct value and to build a structure which helps this learning. It still remains a very appealing framework, at the very least because in theory its representational capacities are not bounded by too few hidden units, nor its learning by too many.

2.6. Graphical Models

2.6.1. Dynamic Bayesian Networks

Dynamic Bayesian Networks can be viewed as a generalization of Markov Models. They are based on simple Bayesian networks, which are a graphical way of representing conditional independencies between random variables. In a Bayesian network, there is a set of random variables $\{Z_1, Z_2, \dots, Z_m\}$, each represented by a node in a graph. A directed edge from Z_i to Z_j indicates that the probability distribution over the values of Z_j depends on the value of Z_i. For a dynamic Bayesian network, the random variables are considered at different time steps, where $Z_i(t)$ is the random variable Z_i at time t. In a DBN, there can be edges from nodes at time steps $1 \dots t-1$ to a node at time t. Random variables can be modeled as a conditional probability table, in which the probability of $Z_i = o_i$ is explicitly specified for each assignment of values of its parent nodes. They can also be modeled as other probability distributions depending solely on fixed parameters and their parent nodes.

Bayesian networks and DBNs are of interest because they represent an explicit factorization of the random variables, which permits simpler inference. A DBN can be made to generalize the concept of Hidden Markov Model through *hidden variables*: nodes whose value is never observed. In this case, there are well known inference algorithms for computing a belief over the possible assignments of values to the hidden variables. These variables can represent the state of the system, whereas the observed variables represent observations, as in HMMs. Unfortunately, as discussed in [Boyen and Koller, 1998], all hidden variables generally become correlated after a few time steps. Keeping track of the belief state then involves storing a vector which is exponentially large in the number of random variables.

[Boyen and Koller, 1998] give results regarding the efficiency of approximations of a belief state, showing that it decreases exponentially with the number of parents of each variable.

Good results with dynamic Bayesian networks have been obtained in a variety of tasks. Of interest is the representation of POMDPs and HMMs as dynamic Bayesian networks [Theocharous *et al.*, 2004]. In this paper, the authors present the advantage of the approach. They focus on a Hallway task [McCallum, 1995], where the agent observes whether there is a wall facing it and where the state is composed of a location and a orientation. The task is hierarchical, in that two levels of state are defined: the physical location of the robot and whether it is at a junction in the maze. Using a DBN has the advantage that it can encode succinctly the state variables, yielding a much smaller representation. The authors show that their method has lower inference complexity, due to the structure, than simply using a hierarchical POMDP. Space complexity, however, is greater than in other methods. Although the concept of using structure to perform learning and inference more efficiently is appealing, it has the severe drawback of requiring more knowledge of the environment (in order to shape the DBN used). In [Theocharous *et al.*, 2004], for example, they explicitly encode which state transitions are possible. Nevertheless, their empirical results involving a simulated hallway task and real robot experiments show improved performance from the use of DBNs.

2.6.2. Boltzmann Machines

Boltzmann Machines [Fahlman *et al.*, 1983] combine the features of neural networks with that of Bayesian Networks. They are composed of L layers of units, as in a feedforward network. However, these units are binary random variables, rather than real-valued. In general, a unit in layer l is connected to all units in layers $l-1$ and $l+1$, with the exception of units in layers 0 and L. The first layer contains observed variables, such as an input vector; the value of each variable is fixed. The other variables, however, are hidden. As in feedforward neural networks, there is a weight $W_{ij}^{l_1 l_2}$ which regulates how the value of one variable influences another's. Formally, let β_i^l be the (binary) value of unit i in layer l. The probability that unit i on layer l takes on the value 1 is given by

$$P\{\beta_i^l = 1 | \vec{\beta}^{l-1}, \vec{\beta}^{l+1}\} = \sigma(\sum_j \beta_j^{l-1} W_{ji}^{l-1,l} + \sum_j \beta_j^{l+1} W_{ji}^{l+1,l}) \tag{2.8}$$

Here, σ represents the sigmoid function (Equation 2.7). That is to say, the probability of $\beta_i^l = 1$ depends on the value of unit i's neighbors. If a weight is positive and the corresponding unit's value is 1, then it increases the likelihood that unit i also take value 1. The converse is true when weights are negative.

Boltzmann Machines are a powerful tool because they explicitly define a full probability distribution over the hidden units, while preserving the simplicity of a neural network. The weights can be trained to reflect a batch of input data, in such a way that they form a generative model over the data [Hinton and Sejnowski, 1986]. There are two ways in which this can be used: first, new inputs can be generated by making the input variables unknown, and randomly generating values for all hidden variables until they reach a stable state. This procedure is known as *Gibbs sampling*, and was discussed in [Neal, 1992] for generating samples in a sigmoid belief network. This generative model also provides a way for detecting anomalous patterns, by looking at how unlikely a given input pattern given the weights.

There have been many applications of this framework. In [Taylor *et al.*, 2007], the authors discuss a generative model of human motion. In this particular case, the Boltzmann Machines act over time, rather than for a fixed set of variables. This is done in a way parallel to the way time can be incorporated in Bayesian networks. Because Boltzmann Machines provide a way of creating new observations through the Gibbs sampling procedure, complete sequences of simple human motion can be generated. Especially interesting is the fact that, if the data is divided into clusters of sequences that resemble eachother (in [Taylor *et al.*, 2007], walking and running), then the system will generate either of these sequences with high probability, but sometimes switching to the other.

The main drawback of the approach, however, is its complexity. It is well known that Gibbs sampling, for one, can take a long time to converge [Neal, 1992]. Learning is also time-consuming, as it involves finding weights that best model the whole distribution.

There have been a few attempts at reducing this complexity, such as in [Hinton *et al.*, 2006] and through Restricted Boltzmann Machines, which have only one layer of hidden units.

2.7. Sparse Distributed Memories

2.7.1. Description

Sparse Distributed Memories (SDMs), as the name implies, were developed by Kanerva [Kanerva, 1988; 1993] in order to provide a data storage system closer to human memory. Drawing on various physiological considerations, these memories trade storage accuracy for generalization. An example given in [Kanerva, 1993] is that of images: it is very unlikely that the same image will be observed twice, especially given a certain amount of noise in an agent's sensors. However, it is not a strong claim to assume that images which are related will have similar features. A model that can recover stored data based on input similarity is then more likely to be useful to an intelligent system than one that requires strict equality between inputs. Kanerva's proposal is to replace the standard addressing system found in computer architectures by *hard locations*. Many locations are triggered by a given input (observation), and the output is formed based on data at each location.

Formally, a Sparse Distributed Memory is composed of a set of hard locations $\{C_1, C_2, \ldots, C_n\}$. Each hard location C_i is a binary vector of length L_C, and is associated with a data vector D_i of length L_D composed of integers. Given an input vector $\vec{x}$ of length L_C, the SDM generates the output as follows: each location C_i is matched against $\vec{x}$, and locations 'close enough' (e.g. within a certain Hamming distance of $\vec{x}$) are activated. This activation is recorded through a vector $\vec{\alpha}$, such that $\alpha_i = 1$ if $\vec{x}$ is near C_i, and 0 otherwise. The output $\vec{y}$ is produced by adding up the data vectors corresponding to activated locations and thresholding the result. Formally, let β_i be the components of an intermediate vector β; $\vec{y}$ is given by

$$\begin{aligned}\vec{\beta} &= \sum_{i:\alpha_i=1} D_i \\ y_i &= sign(\beta_i)\end{aligned}$$

It is important to note that some elements of D_i may be negative. In effect, this scheme distributes the storage of assocations $\vec{x} \rightarrow \vec{y}$ over many locations. The result is that SDMs can compensate for noisy inputs and as well as recover data for previously unseen observations. Learning in SDMs is fairly straightforward: the algorithm modifies the data vectors corresponding to the training input to more closely reflect the training output.

SDMs have been shown in many applications to perform well when the input vector is of high dimension [Kostiadis and Hu, 2001; Rao and Fuentes, 1996]. One of the drawbacks of the algorithm is that an overly simple distance function is often not enough. In the case of images, for example, it is well know that slight changes in pose [2] can cause large shifts in the input space. Although there is no clear solution to this problem, simple objects can end up requiring many hard locations simply because of the activation scheme. More importantly, the lack of a probabilistic interpretation of the world in SDMs prevents its use in general temporal prediction. In effect, the system averages out noise in order to produce 'ideal' versions of outputs. This means that, if based on the history H_t, the system should predict symbol 'A', 'B' and 'C' with some probability mass, SDMs will learn to output the weighted average of those three symbols.

2.7.2. Using Sparse Distributed Memories for Temporal Prediction

There have been many attempts at using SDMs for temporal prediction. I will describe one briefly here; however, it is important to keep in mind that such applications all make the strong assumption that given sufficient history, the next symbol can be deterministically predicted. This is an assumption that was also made in [Holmes and Isbell, 2006] (see Section 2.4.2).

In [Bose *et al.*, 2005a], the authors describe an associative memory which can learn to predict sequences. This paper is of interest to this thesis because it combines SDMs with other concepts that are more geared towards temporal prediction, such as the use of a *context layer* (see Section 2.5.2). The main contribution of the paper is to show that one can combine fixed history models (such as the simple Markov Models described in Section

[2]For simple objects, pose can be thought of as the slant and tilt. For a human face, for example, we obtain different poses by rotating the object along the y-axis.

2.1) with such a context layer. The latter is trained to modify itself based on its value at the previous time step and the input: in effect, it reflects the state of the system. The authors then construct an input vector based on both history and context which is fed to a modified SDM, resulting in an output which is its prediction.

Of interest is the fact that such a scheme allows sequences to be learned in a single pass. The authors' empirical results show that combining both context and history results in better performance in that regard. The main problem with the proposed algorithm is that, as stated above, it is not designed to predict distributions over future symbols. Also, the design of the algorithm is rather *ad hoc* and no theoretical guarantees are given. The authors have implemented their system as a special type of neural network in [Bose *et al.*, 2005b]. This paper gives empirical results on a simple sequence memorization task, but the authors focus on the one pass learning aspect rather than good learning over a large set of examples.

2.8. Predictive State Representations

Predictive State Representations were proposed recently [Littman *et al.*, 2002] as an alternative to representing state information. They are grounded around the notion of a *test*, which is a sequence of actions and observations. In a PSR, the algorithm keeps tracks of the probability of different tests succeeding. Knowing the probability of success of such tests yields state information. For example, if an agent faces a wall but knows that it would observe a coffee machine by turning left once, then its 'position' relative to the coffee machine is encoded in the test's probability of success. It was shown in [Littman *et al.*, 2002] that a special type of PSR, the linear PSR, can represent a POMDP with at most as many tests as there are states in the POMDP. This is related to [Ron *et al.*, 1996] and [McCallum, 1995] in their conception of the state as a history 'suffix'; PSRs, however, take a drastically different approach by considering future events rather than past events.

The probability of success of the tests chosen to represent the state can be updated over time through an equation similar to the belief update in HMMs (see Section 2.3). Learning, unfortunately, is more complicated. For linear PSRs, for example, it is known that in order to achieve a minimal number of tests, one must generate the set of linearly independent tests. Doing so is nontrivial if the probabilities of success are estimated. For that reason, experimental results also focus on small tasks, although recent work [Tanner *et al.*, 2007] has shown that PSRs can be used to construct abstractions over the state space.

CHAPTER 3

The Algorithm

Chapter Outline

In this chapter, we detail the workings of our algorithm.

3.1. Motivation

Some of the algorithms which were detailed in the previous chapter are interesting because they do not explicitly define what a state of the system must be. This is in line with my proposal of avoiding the notion of state altogether and instead attempting to work solely at the level of observations. Recent successes, however, often come from algorithms which produce output vectors, rather than 'symbols'. Neural networks fall in the first category; Boltzmann Machines, for example, have shown great representational power [Taylor *et al.*, 2007]. Symbolic approaches, on the other hand, are often easier to develop because they usually deal with a finite set of observations. A good compromise between the two is found in Sparse Distributed Memories, which produces data vectors which vary at the level of individual elements, while preserving the notion of finiteness associated with symbolic representations. Indeed, the focus of Kanerva's work has been to constrtuct hard locations that can capture sensor data without having to explicitly reduce it. This is a scheme which can be more porrtable to different tasks. In effect, the hard locations constitute features - vector elements - but these need not be in any way predetermined. These features are special in that they actually represent 'prototype' percepts. It therefore seems reasonable to replace the feature-driven approach of neural networks by a more symbolic percept-driven approach.

The second improvement that I propose is closely tied to the concept of state. In recurrent neural networks, for example, the state - that which allows us to predict from history - is represented by the values of the hidden units. In Probability Suffix Trees, especially in Utile Suffix Memory, leaf nodes represent states which hold meaningful predictive information. In all cases, there is a very definite and compact notion of state - but it is constructed by the algorithms, rather than provided by the designer. Storing the whole history more explicitly, however, can be interesting when the correct prediction varies significantly for closely related histories. Furthermore, it should give more control over the history than through assuming that the learning procedure can construct a correct state representation. To exemplify this, one only needs to consider the fact that hidden units in neural networks have a tendency to spread a given mode across many hidden units [Elman, 1990; Neal, 1992]. This can be detrimental to both performance and learning time, and also makes it harder to interpret the resulting structure. Hence, an algorithm which condenses the history information into a *context vector* without relying on the internal architecture of the system might perform better.

An additional argument in favor of using hard locations rather than an implicit state assumption comes from the forgetting problem, often found in architectures with hidden units. Forgetting can occur in neural networks when the distribution of training examples changes through the training. In this case, the weights are modified towards the new distribution. Because the learning process distributes states over many hidden units, this can lead to the modification of other partts of the distribution. Increased performance over the new training set is then observed, to the detriment of the original one. Unfortunately, there are many examples of such forgetting, such as when attempting to model a value function in Reinforcement Learning [Rivest and Precup, 2003]. A more localized approach such as one which uses hard locations often performs better in this situation by only modifying the parts of the distribution which have changed.

3.2. Proposed Algorithm: Context-Driven Prediction

Before discussing the proposed framework, a few assumptions must be made regarding the environment which I am interested in modeling, and a few terms must be defined. First, we define a *percept* as a real-valued vector representing an observation. The term of percept here relates to a real-valued vector, as opposed to an observation (which is often viewed as a symbolic element). Secondly, no one-to-one prediction assumption is made, as is the case in SDMs and Looping Suffix Trees: a given history might result in any probability distribution over its predictions. Formally, an input percept $\vec{x}$ maps to a *distribution* over future percepts.

Define the *memory* as a set of cells, C. Each of these cells acts as a hard location, and therefore has a single percept associated with it, which is denoted C_i. Each cell also has a *saliency* value s_i associated with it and a vector of weights, W_i. W_i represents directed influences between cells, and so W_i has the same size as C. In general, the weight matrix is denoted by W and the saliency vector corresponding to C by $\vec{s}$.

When the system is presented with an input percept $\vec{x}$, an activation vector α is computed, similar to the activation in SDMs. Here, however, α is a real-valued vector with elements taking values between 0 and 1, where $\alpha_i = 1$ indicates a perfect match between $\vec{x}$ and C_i and $\alpha_i = 0$ indicates no match. Usually, $\sum_i \alpha_i = 1$. In this work, I use the simplest activation function, namely $\alpha_i = 1$ if $C_i = \vec{x}$, and $\alpha_i = 0$ otherwise. In effect, this assumes that percepts are actually symbolic and not subject to noise; but the proposed algorithm is designed for a more general type of percept.

By themselves, SDMs do not allow any association based on past observations. To circumvent this, the saliency value of cells is used as an activation trace. More formally, at every time step the saliency vector is computed as

$$\vec{s} \leftarrow \gamma\vec{s} + \vec{\alpha}$$

This is similar to the cumulative eligibility trace in Reinforcement Learning [Sutton, 1988], where γ is a decay factor. A restricted form of this type of encoding has also been used for prediction in [Bose *et al.*, 2005a; Furber *et al.*, 2004], where the k most important symbols are stored in the same fashion as is done here. In their case, however, this is used spatially rather than temporally, to order the relevance of the different symbols. If $\alpha_i = 1$ for exactly one percept and 0 everywhere else, $\vec{s}$ represents the past sequence of observations. With infinite machine precision, the history can also be recovered from $\vec{s}$. Getting an exact prediction based on any history, however, requires that the algorithm be able to detect minute variations in $\vec{s}$. Finally, note that for the purposes of the system, $\vec{s}$ does not need to be an exponentially decaying trace of observations: its goal is to serve as *context* to obtain better predictions. Neural networks can be viewed as using a context which is learned through error propagation; it would certainly be possible to generate $\vec{s}^t$ from $\vec{s}^{t-1}$ and $\vec{\alpha}^t$, the previous time step's context and the current input. Using a decaying trace has the advantage of being straightforward to implement. It is also easier to produce theoretical results than if $\vec{s}$ was generated based on a learned procedure.

This algorithm diverges from SDMs as it attempts to predict percepts that match its hard locations, rather than separating them from the output words. Secondary activation, denoted by $\vec{\beta}$, is defined as a vector representing a prediction weight for each cell. Of interest is the prediction of the percepts (represented by cells) which may be observed. It is assumed here that hard locations represent actual percepts, as opposed to averages, as used in traditional SDMs. Computing $\vec{\beta}$ is simply done as:

$$\vec{\beta} = W\vec{s}$$

From this equation it should be noticed that the weight matrix acts as a set of influences: experiencing a percept shifts the distribution over predictions towards related percepts. Since $\vec{\beta}$ represents the prediction of related, or associated, percepts, its values should be in the correct prediction order with higher values for percepts that are more likely. Any function of $\vec{\beta}$ which preserves this ordering and results in a valid probability distribution can be used to predict the next time step. I chose to use a simple Boltzmann distribution based on $\vec{\beta}$. Formally, the probability distribution is given by

$$P\{C_i|\vec{s}\} = \frac{e^{\tau\beta_i}}{\sigma}$$

The distribution's entropy is controlled by τ, a standard temperature parameter between 0 and ∞. Here, $\sigma = \sum_i e^{\tau\beta_i}$ so that $\sum_i^m P\{C_i|\vec{s}\} = 1$. For simplicity, $\tau = 1$ will be assumed throughout the rest of this thesis. To make things more easily comparable with Boltzmann Machines, we also add a bias term to β, simply by defining a weight $W_{0,i}$ for each cell, and adding it to β_i. This simply represents a prior probability on hard location i[1].

3.3. Learning

After having discussed how the algorithm predicts events, I now describe how learning can be accomplished. For now, assume that there already is a known set of hard locations. Let $\vec{x}^t$ be the percept observed at time t, and similarly C^t, the set of cells, W^t the weight matrix and $\vec{s}^t$ the saliency vector. Denote the activation due to $\vec{x}^t$ by $\alpha(\vec{x}^t)$.

Assuming that the goal is to predict $\vec{x}^t$ whenever $\vec{s}$ is present, W^t should be modified to produce a probability distribution similar to $\alpha(\vec{x}_t)$. Formally, let π be the probability distribution on C. Then let $\sigma = \sum_i e^{\beta_i}$, and recall that $\beta_k = \sum_i W_{i,k}s_i$. The Kronecker delta function is $\delta_{ij} = 1$ if $i = j$, 0 otherwise. First note that:

$$\begin{aligned}
\frac{\partial}{\partial W_{j,i}}\pi_k &= \frac{\partial}{\partial W_{j,i}}\frac{e^{\beta_k}}{\sigma} = \frac{e^{\beta_k}}{\sigma^2}\left(\sigma\frac{\partial}{\partial W_{i,j}}\beta_k - e^{\beta_i}s_j\right) \\
&= s_j\pi_k(\delta_{ik} - \pi_i)
\end{aligned}$$

Suppose a percept $\vec{x}_t$ is observed. Then the standard definition of the likelihood of $\vec{x}_t$, given the current parameter set θ, is simply $P\{\vec{x}_t|\theta\}$. In general, however, $\vec{x}_t$ does not correspond to a single cell C_i. This means that α will have more than one non-zero value.

[1]This is not strictly necessary because in my experiments I use a start symbol. In studying the prediction outcomes of the system, I ignore the prediction on this symbol. As such, for any prediction that the algorithm would like to make, there is a non-zero saliency vector to support it.

To countervene this problem [2], assume that α_i represents the non-normalized probability that $\vec{x}_t$ truly is an instance of the prototype percept C_i - a reasonable assumption if the SDM represents a mixture of Gaussians through its locations. Then if L denotes the likelihood in question,

$$L = P\{\vec{x}_t|\theta\} = \sum_i P\{C_i|\theta\}P\{\vec{x}_t = C_i\} = \frac{1}{\kappa}\sum_i \alpha_i\pi_i = \frac{\vec{\alpha}\cdot\pi}{\kappa}$$

Here, $\kappa = \sum_i \alpha_i$ so that $\sum_i \frac{\alpha_i}{\kappa} = 1$. Then, taking the logarithm of L and differentiating with respect to $W_{j,i}$, this yields

$$\begin{aligned} \mathscr{L} = \log L = \log(\vec{\alpha}\cdot\pi) - \log\kappa &= \log(\vec{\alpha}e^{\beta}) - \log(\sigma) - \log\kappa \\ \frac{\partial}{\partial W_{j,i}}\mathscr{L} = \frac{\alpha_i e^{\beta_i}s_j}{\vec{\alpha}\cdot e^{\beta}} - \frac{e^{\beta_i}s_j}{\sigma} &= s_j\left(\frac{\alpha_i e^{\beta_i}}{\vec{\alpha}\cdot e^{\beta}} - \pi_i\right) = s_j(\hat{\pi}_i - \pi_i) \end{aligned}$$

where $\hat{\pi}_i$ represents a weighted probability distribution on the C_i's. Of interest is the fact that the gradient does not depend on κ: hence there is no need to worry about $\sum_i \alpha_i$. For the special case when $\alpha_i = 1$ for exactly one C_i, then $\hat{\pi}_i = 1$ if $p_t = C_i$, and 0 otherwise. Let ϵ be the vector of errors, with $\epsilon_i = \pi_i - \alpha_i$. Clearly $\hat{\pi}_i - \pi_i = \epsilon_i$, so that

$$\frac{\partial}{\partial W_{j,i}}\mathscr{L} = s_j\epsilon_i \tag{3.1}$$

This is a very compact formula. Recall that the log-likelihood of a batch of data is the sum of the log-likelihood of the individual patterns. Let s_j^t be the saliency of cell j at time t, and similarly ϵ_i^t the error on the prediction of i at time t. Then, for T training samples,

$$\frac{\partial}{\partial W_{i,j}}\mathscr{L} = \sum_t s_j^t\epsilon_i^t \tag{3.2}$$

[2]The problem lies in what α represents. In Sparse Distributed Memories, more than one hard location is activated because the input resembles many ideal percepts. In learning such a case, a correct predictive model should attribute a probability of 1.0 to the event of observing α, a mixture of ideal percepts. Rather, here, the algorithm can only give a distribution over ideal percepts.

This can be written in matrix form as $\Delta = SE^T$, where Δ is the matrix of gradients, S the matrix of saliencies over time and E the matrix of errors over time. This is the batch version of the algorithm. Whether using a single sample or a batch of data, the weights can be modified through a standard update rule with learning rate $c \in (0, 1)$:

$$W_{i,j} \leftarrow W_{i,j} - c\frac{\partial}{\partial W_{i,j}}\mathscr{L} \tag{3.3}$$

$W_{0,i}$ can be updated in a similar fashion, if one considers that there is a bias saliency element which always has value 1. As such, for the symbolic case at least, the resulting maximum likelihood gradient derivation is quite elegant.

There is a second learning problem, which I have ignored here, but is of interest. The hard locations do not have to be pre-defined, or fixed. In a way, learning to *recognize* a percept can be just as hard as prediction. This issue will be discussed further in Section 7. The whole algorithm is detailed in the Algorithm 1 table.

Algorithm 1 The Context-Driven Prediction Algorithm

```
Initialize: W ← 0
loop
    Begin a new episode
    s⃗ ← 0
    repeat
        Compute π from s⃗ (prediction)
        Observe the next percept x⃗
        W_{i,j} ← W_{i,j} − c ∂/∂W_{i,j} 𝓛
        s⃗ ← γs⃗ + α(x⃗)
    until end of episode
end loop
```

3.4. Discussion

Solely considering finite state automata with probabilistic observations, it can already be seen that the algorithm cannot fully model all such automata - for example the one in Figure 3.1 - at least with the basic algorithm detailed in this chapter.

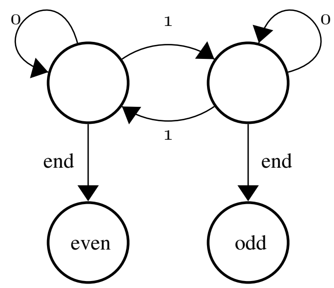

FIGURE 3.1. Example of a Finite State Automata which cannot be represented by our model. This system simply counts the number of 1's that have been outputted - the parity of the sequence - and outputs even or odd at the end of the sequence (or probabilistically at any point in time).

One major underlying assumption is that percepts represent state information. Observing the same percept many times, therefore, does not explicitly induce a different distribution on future events - as is the case in a simple Markov Model - but rather increases the likelihood of being in a state which produces the said observation. This is certainly a flaw of the model, but can also be seen as restricting our hypothesis class in order to obtain better learning. Also, there might be cases where the correct predictions can still be made.

CHAPTER 4

Experimental Results

Chapter Outline

This chapter discusses various experiments that were performed with the algorithm in order to verify its capacities.

4.1. Goals and Methodology

In order to verify the validity of the proposed algorithm, it is at least necessary to ensure that it can handle the same basic cases as k^{th} order Markov Models can represent. Of utmost importance, clearly, is the need for the algorithm to learn unbiased probability distributions. Moreover, Markov Models are known to be powerful because inverting the order in which two symbols appear can yield a completely different distribution. Intuitively, one would expect the algorithm to perform very badly on such a task, since it collapses the history into a single real-valued vector. As will be seen below, however, the proposed framework does preserve order-dependence, at least to some extent.

For all the experiments below, unless stated otherwise, the following parameters were used: $\gamma = 0.5$, $c = 0.05$, the agents learned in an on-line fashion and the log-likelihood of the observations was used to derive a gradient. Also, I fixed the number of possible percepts to 10, so that across the tests (which take at most 10 observations) the system begins with the same uniform distribution. In tasks where less than 10 symbols are used, the expected behavior of the algorithm is to reduce their probability of occurence to 0 over time.

As opposed to control and reinforcement learning tasks, where there is a clear goal, the algorithm is simply attempting to model the system. Unfortunately, comparing probability distributions is not always an easy task, especially when performance throughout an episode is of interest. When only interested in the prediction of a single, end symbol I will show the resulting probabilities. In most cases, however, I will rely on the Kullback-Leibler (KL) divergence (D_{KL}) for comparing two probability distributions. Formally, given a "ground truth" discrete probability distribution P and an estimate of this distribution, Q, the KL divergence measure is given by

$$D_{KL} = \sum_i P_i \log(\frac{P_i}{Q_i}) \tag{4.1}$$

This is a concise summary of how good the estimate is with respect to the true distribution. This divergence measure takes values between 0 and ∞, and like entropy, depends on the number of possible percepts. Since I am using a constant number of symbols throughout the following experiments, the latter problem is not of concern. However, this measure only gives the divergence for a given time step within an episode. To palliate this problem, I report the mean divergence over one or many episodes. In the experiments described in this chapter, I always use 500 test episodes (without learning) to estimate the average KL divergence at a given point in time.

4.2. Markov Properties

4.2.1. Frequency Estimates

The first and simplest experiment tests whether the algorithm can correctly estimate probabilities. To do this, I set up the following trivial task: the agent first receives a start symbol and then receives either a 1 or a 2, such that $P\{1\} = p$ and $P\{2\} = 1-p$. This ends the episode, so that the only thing which the agent has to do is to predict the probability of each symbol occuring. For all of the data presented below, I ran tests on three different probabilities, namely $p \in \{0.5, 0.75, 0.9\}$.

		Mean	Std. Deviation
$p = 0.5$	$P\{1\}$	0.477	0.068
	$P\{2\}$	0.478	0.069
$p = 0.75$	$P\{1\}$	0.722	0.063
	$P\{2\}$	0.235	0.060
$p = 0.9$	$P\{1\}$	0.874	0.027
	$P\{2\}$	0.079	0.023

TABLE 4.1. Average predicted observation frequency based on the actual frequency.

Episodes	10^3	10^4	10^5
	Mean (x$10^{-2} \pm 10^{-2}$)	Mean (x$10^{-2} \pm 10^{-2}$)	Mean (x$10^{-2} \pm 10^{-2}$)
$p = 0.5$	4.67 $\pm$ 1.56	1.24 $\pm$ 1.21	1.34 $\pm$ 1.61
$p = 0.75$	4.43 $\pm$ 1.23	1.06 $\pm$ 0.67	0.94 $\pm$ 1.23
$p = 0.9$	4.21 $\pm$ 0.50	0.68 $\pm$ 0.47	0.31 $\pm$ 0.34

TABLE 4.2. Average Kullback-Leibler divergence $\pm$ one standard deviation for the frequency task, in function of p and the number of training episodes.

Table 4.1 show the results of this experiment in terms of the end estimate for the probability distribution over 1 and 2. As discussed before, this algorithm induces global distribution over percepts, which necessarily leads to all percepts (including those that never appear in the experiment) being given non-zero probability weight. Nevertheless, the results are encouraging, with fairly low standard derivation. The data comes from 30 independent trials, each of 1000 episodes (here, 1000 samples).

With additional samples and a lower learning rate, these estimates can be shown to converge to the true probabilities, which should not come as a surprise. To verify this, Table 4.2 reports the KL divergence for three different numbers of episodes, namely 1000, 10000 and 100000.

The mean value of D_{KL} decreases, as the number of samples is increased, regardless of p. There is clearly a precision limit, due to the learning rate c; this is reflected by the fact that the average divergence does not change between 10000 and 100000 episodes when $p = 0.5$, and from the fact that the standard deviation of the results does not seem to be influenced by additional samples. It should also be noted that the actual weights that the system learns stabilize, rather than diverging.

One important issue in machine learning is whether updates to the model should occur in an *on-line* or *batch* fashion. In the former method, the system (here, weights) is updated at every step. In the latter, the system waits until it has accumulated a sufficient amount of experience before computing the gradient and updating the weights. It therefore seems reasonable to explore this issue here, at least briefly, to determine whether the algorithm is better suited to be used on-line or in batch, and also to obtain some insight as to what might affect its performance. Figure 4.1 shows the KL divergence resulting from using different batch sizes, over 100000 episodes - sufficiently long to reach convergence. Since with an increased batch size also comes increased variance in the gradient magnitude, it is customary to reduce the learning rate appropriately. Here, I chose to have a learning rate of $0.05/S$, where S is the batch size. A baseline D_{KL} is also given for the on-line case. Note that there is a (subtle) difference between a batch of one episode (the leftmost value on Fig. 4.1) and the on-line method: the former incorporates the gradient for predicting the start symbol. For this experiment, only $p = 0.5$ and $p = 0.9$ are reported, as other values yielded similar results.

Clearly, the on-line learning method performs no worse than the simple batch algorithm. Despite the reduced learning rate, as the batch size is increased there is a constant degradation in the end performance of the algorithm. This can be explained in this case by the following fact. Suppose that the weights have been learned to certain values, but that there is a difference between the predictions of the two symbols. These differences might have arised from the initial sampling distribution being slightly different from the actual distribution. To obtain a correct predictive model, one weight must be increased and the other, decreased. However, when using a batch method, the system risks accumulating enough gradients such that when the weights are updated, the larger one becomes much smaller. In such a way, an irremediable oscillation occurs between the two weights. This is fatal in this model, as I am using an exponential function over $\vec{\beta}$: the oscillations cause large changes in the probability distribution. This phenomenon has been observed in the

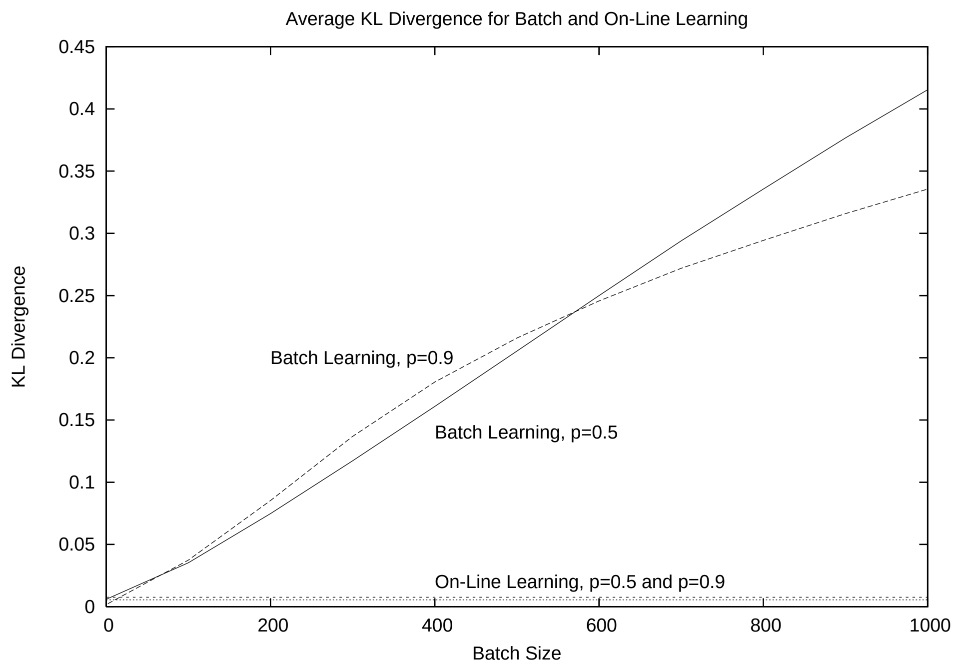

FIGURE 4.1. Average KL Divergence in function of the batch size in the frequency task; comparison between batch and on-line learning.

other tasks presented in this thesis; as such, the on-line version of the algorithm was retained as being more stable. It should be noted, however, that in more complex situations it might be desirable to use a batch in order to smooth out the gradient.

Note that this effect occurs just the same when the probability of observing symbol 1 is much closer to 1.0, namely $p = 0.9$. Although there is certainly an improvement for large batch sizes, the on-line method still outperforms its batch counterpart. There is also a difference in the KL divergence between the two reported values of the on-line method, but these should not be considered significantly different.

4.2.2. Order Dependence

The second most important property of Markov Models, which makes them so attractive, is their ability to distinguish between the order of symbols over time. This is something which is commonly observed in language, where the ordering of the words in a sentence can completely change its meaning. One only has to consider the inversion of the verb and the subject in English and French in an interrogative sentence to have an example of this:

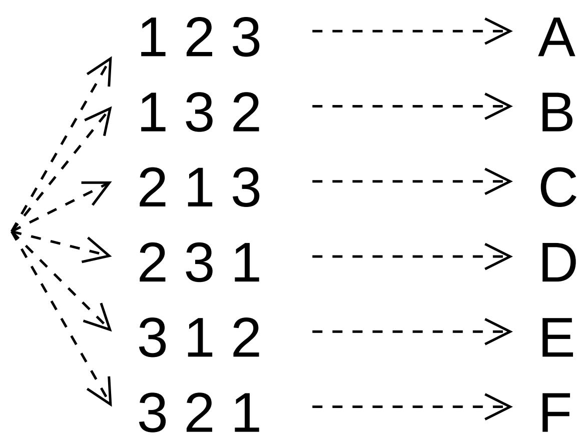

FIGURE 4.2. Example of order dependence task, where the same three symbols are given, following by a fourth which depends on their order.

"Did you..." clearly corresponds to a different sense than "You did...". It is therefore critical that this ordering should be preserved in any predictive model, at least as much as possible; failing this, there would be a much more limited set of environments that can be modeled.

To verify that the algorithm can handle this order dependence, I chose to use the following simple task. The agent is presented with n symbols; the order in which these symbols occur contains all the information necessary to predict with full accuracy the last symbol. Figure 4.2 gives an example of this task.

The experimental task here consists of randomly selecting one of the $n!$ possible permutations, outputting it, and then producing a last symbol deterministically. From the point of view of the agent, the $n-1$ first symbols cannot be fully predicted. The n^{th} symbol, although determined by the order of the preceding ones, adds complexity to the task which the algorithm must solve.

Of interest then is the average probability of the end symbol, when $n \in \{2, 3, 4\}$. For simplicity, let the first n observations denote the common symbols, and the following $n!$ ones denote the end symbol. For $n = 2$, this gives the following possible sequences: $1-2-3$ and $2-1-4$. Note that for $n = 4$, this means that there are 28 $(4 + 4!)$ symbols in total. Since the Boltzmann distribution, as mentioned above, is of a global nature, this implies that the probability of observing a symbol, prior to any learning, would be lower

Episodes	100	1000	5000	20000
	Mean $\pm$ S.D.	Mean $\pm$ S.D.	Mean $\pm$ S.D.	Mean $\pm$ S.D.
$n = 2$	0.722 $\pm$ 0.033	0.960 $\pm$ 0.002	0.990 $\pm \sim 0$	0.990 $\pm \sim 0$
$n = 3$	0.247 $\pm$ 0.011	0.625 $\pm$ 0.014	0.889 $\pm$ 0.005	0.967 $\pm$ 0.001
$n = 4$	0.307 $\pm$ 0.207	0.400 $\pm$ 0.268	0.47 $\pm$ 0.295	0.54 $\pm$ 0.319

TABLE 4.3. Average estimated probability of observing the correct symbol given a specific context sequence for the order dependence task, and corresponding standard deviation. Values for three context lengths are reported.

when n is increased, if exactly the necessary number of observations was used. To avoid biasing the results based on this, I chose to have all experiments be performed with 28 symbols. Although this slows down the learning for $n = 2$ and $n = 3$ without improving anything, it makes the results comparable across context lengths.

Table 4.3 gives the resulting estimated probabilities of observing the correct end symbol, averaged over the $n!$ possibilities. These values are obtained based on 30 independent runs. The reported standard deviations are over all symbols and trials. I also give the estimates after three different number of episodes, in order to compare how quickly the algorithm can converge. In this experiment, I used a learning rate of 0.5 to speed up the learning process; this did not seem to significantly degrade the quality of the solutions. [1]

Clearly, regardless of n, the algorithm learns to disambiguate between different contexts. It should be noted that having an estimated probability of 60% for a given symbol, when $n = 4$, implies that the remaining possible symbols - there are 23 of them - share a 40%. Therefore the disambiguation remains good, despite lower values. Although the rate at which learning occurs is significantly reduced when n is increased, the reasons for this can be easily given. First of all, from the factorial nature of the experiment, there are three times as many contexts for $n = 3$ than for $n = 2$, and four times more when comparing $n = 4$ and $n = 3$. As such, the expected number of training episodes featuring a given context is considerably diminished; one should expect to require three (or twelve) times more episodes to obtain similar results. There also is the trivial fact that adding a symbol increases the complexity of the task. Therefore, the apparently much weaker results

[1]Alternatively, I could have also computed the variance within a given trial across the different contexts. However, there was little difference in those predictions for $n = 2$ and $n = 3$, such that it was deemed unnecessary to include these results here.

at $n = 3$ and especially $n = 4$ should not be blamed on the algorithm. In a more natural framework, it might be the case that there is a constant number - say two or three - of different orders which can occur. In such a case, one can hope that the algorithm would have similar learning capacities even with longer context sequences. Due to the apparent weak performance of the algorithm with $n = 4$, I ran additional experiments with lower learning rates to determine whether this might have been in cause. However, this only resulted in lower variance, without significantly modifying the average estimated probability.

Another reason for the slower learning when n is increased occurs in other tasks, and so should be pointed out on its own. I have used the term *disambiguation* above without truly defining it; but clearly the proper meaning should be "to be able to predict the correct symbol with probability significantly higher than the other symbols". Learning to disambiguate symbols, then, implies that the system should obtain a set of weights that allows it to do this - this is obviously only possible when there is a unique percept which follows a given context sequence. In the order dependence task, this learning causes the weights to be tuned to respond to the decay in saliency of a percept as it becomes less recent. That is, each end symbol i tunes its weight vector W_i to pick up a certain saliency trace, such that it has the highest β only when this particular trace occurs.

This suggests that the relationship between the order dependence and the discount factor γ, whose value I have until now dismissed, should be further explored. To do this, I let γ vary between 0.1 and 0.9. Clearly, values of 0 and 1 are uninteresting to study, as the former implies having no history at all, whereas the second ignores the temporal ordering of the symbols. As it has been seen above, different lengths of context require different amounts of episodes to converge; in the following experiment, I tried correcting this by training the system for 100, 600 or 2400 episodes per trial for $n = 2, 3$ or 4, respectively. One rationale for this is that there are respectively 2, 6 and 24 different possible context sequences for the lengths studied. Furthermore, the learning with $n = 2$ was sufficiently fast that taking only 100 episodes - rather than 200 - yielded results that are more easily compared in terms of γ.

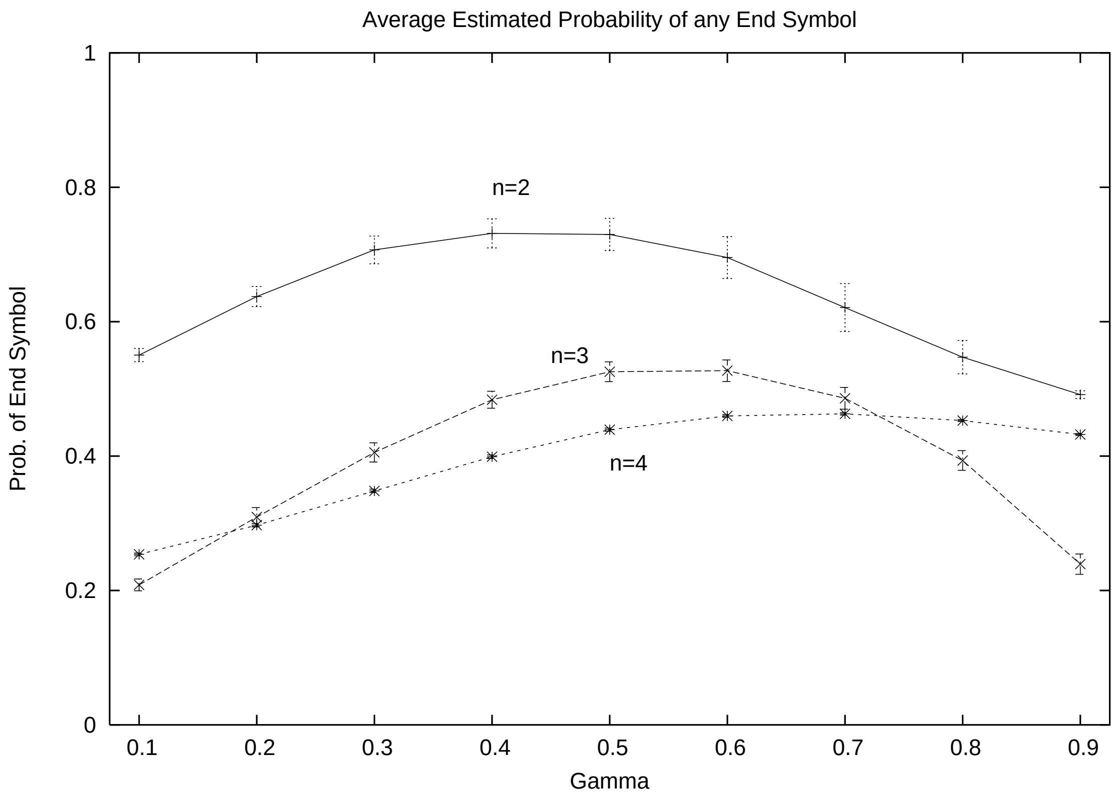

FIGURE 4.3. Average estimated probability of observing the correct end symbol, given a context sequence, in function of γ. Here, the error bars are one standard deviation away from the mean, but the sample variance is only over the different trials (and not over the different end symbols).

Figure 4.3 plots the performance of the algorithms with respect to γ, depending on n. One should remember that there is no reason for which the algorithm should not converge to the correct probabilities, regardless of n, provided that the learning rate is small enough and that sufficiently many training episodes are used. The transient behavior of the algorithm is of course of greater interest, since in a real task there will be a restricted number of samples.

To begin the analysis of Figure 4.3, first note that in general, the estimated probability as a function of γ has a concave shape, with a maximum at some intermediate value. This intuitively makes sense, since values of γ near 0 and 1 have a behavior close to these (despite having radically different limiting properties). Therefore, there should be - and indeed, this is what can be observed from Fig. 4.3 - a value of γ for which learning is optimal. It can also be seen from the three curves that this optimal value depends on n. As

n increases, the optimal value tends to 1. There are other possible explanations for this; it is possible that if the learning rates and training episodes were carefully adjusted, the same optimum would be observed. Finally, notice the similarity of these curves to experiments on *eligibility traces* [Singh and Sutton, 1996], where an intermediate value of λ was found to be optimal, both for replacing and accumulating traces. Higher variance with smaller context lengths may simply be due to the amount of training episodes used.

This task is reminescent of what was done in [Elman, 1990]. In fact, the learning procedure exhibits much of the same structure, for example the fact that the last symbols of the context sequence can be predicted more easily than the beginning ones. Such a task is interesting as it suggests that the algorithm can cope both with unpredictable events (such as the first symbol) and completely determined objects, with degrees in-between. This is in line with my earlier experiment on frequency estimation, but takes it slightly further.

4.3. Influence of the Context Length

In the two sections above, I verified that the algorithm can correctly estimate probabilities based on frequency of observation; I have also shown that, at least for short context sequences, the system can disambiguate solely based on the *order* in which the contextual symbols appear. The next logical step is to give evidence towards my earlier claim that the algorithm does not suffer from window-size problems, as opposed to simple Markov Models. The easiest way to do this is to have a task where the elements of context which help us predict the future symbols occur at varying lengths in the past. In this section, I will denote the length of the intervening context 'junk' (symbols which do not help differenciate between the possible ends) by L.

4.3.1. Fixed-Length Context

The training and testing sequences used for this section are of the following form: first, the system receives a *start symbol*, which uniquely determines the last symbol of the sequence, named - simply enough - the *end symbol*. Between those two percepts, there is a

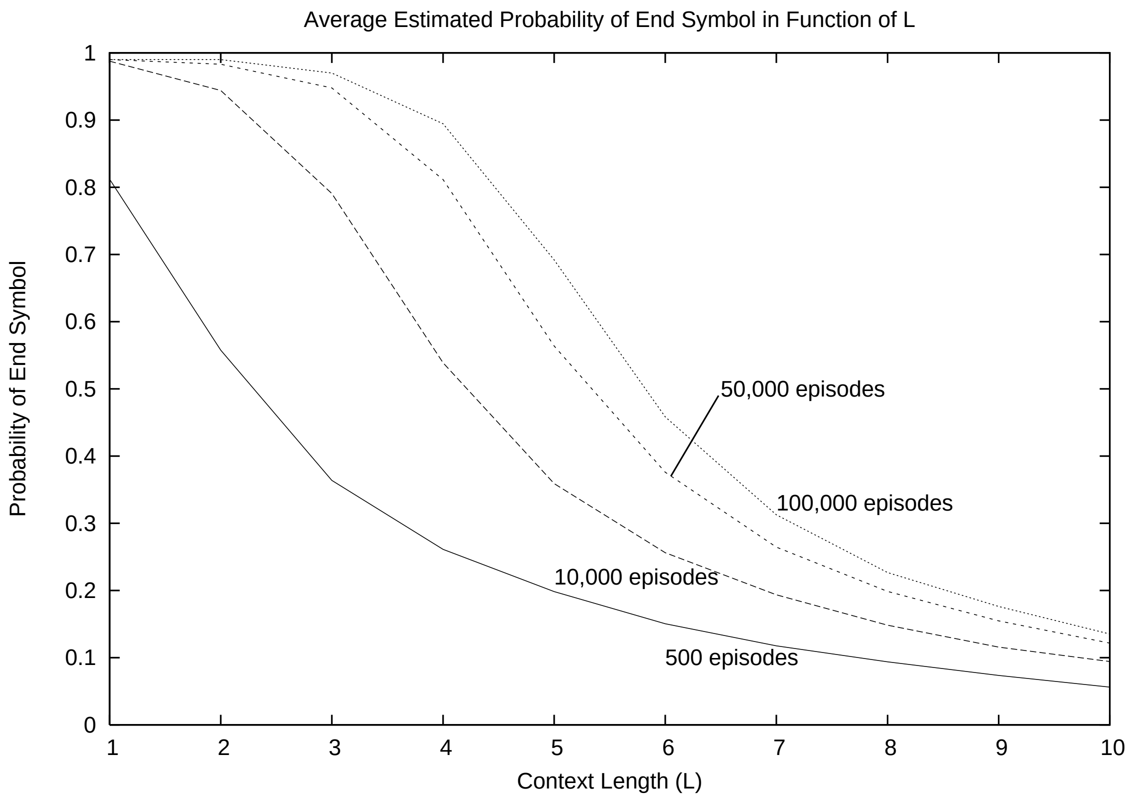

FIGURE 4.4. Average estimated probability of observing the correct end symbol in function of the length of the intervening context 'junk'.

string of symbols of fixed length. The symbols, however, do not occur in any meaningful order and so do not inform the system in any way as to what the end symbol will be.

The first experiment on this task was to determine the performance of the algorithm after a certain number of episodes, based on L, ranging from 1 to 10. I also used $\gamma = 0.5$ and a learning rate of 0.1. The number of training episodes used to evaluate performance were 500, 10000, 50000 and 100000. The lowest value was chosen based on the results of Section 4.2.2, to show the speed of convergence when L is small. The largest value was chosen to show whether the algorithm can converge, even for large values of L.

Figure 4.4 shows the results of this experiment. As expected, the end probability monotonically decreases when L increases, regardless of the number of episodes. There is also a maximum prediction which is reached for the lower values of L, influenced by oscillation issues related to learning rates. As a whole, however, the algorithm performs very well, clearly being able to distinguish between two end symbols based on something that is far in the past. If the learning rate issues are ignored, there is in fact no reason why the algorithm should not converge, given sufficient training episodes. This is because there is a positive weight gradient between the start and end symbols, and because it is easy to hand-craft a solution to this task.

The actual learned structure, however, slightly differs from this. In line with neural network learning, I observed that the prediction comes to rely on the junk data as well as the actual context information in order to predict the end symbol. This does not necessarily have a negative impact on learning; as a corollary to the weights being trained this way, the algorithm is able - to some extent - to predict the non-occurence of the end symbol. In other words, the probability of observing the last symbol rises sharply at the end of the sequence (Figure 4.5). This is quite interesting, as it suggests that the system, despite not having being designed to account for time steps, can effectively perform crude counting. In fact, closer examination shows that the system correctly estimates the uniform distribution on junk symbols and the start symbol.

Unfortunately, even with many episodes (100000), the algorithm still fails to produce good results when $L \geq 8$. One obvious reason is the fact that that I used $\gamma = 0.5$. As such, at the time of predicting the end symbol, the saliency of the relevant context element is 2^{-L}. Of interest, then, is the study of the behavior of the algorithm when we modify γ. Taking $L = 8, 9, 10$, I consider the results of training the system for 10000 episodes when γ ranges from 0.1 to 0.9, as in Section 4.2.2. It is clear that an increased γ must yield better result for this task; but since it has been shown in the previous section that there is an optimal γ at which the order of symbols is best disambiguated, it appears useful to know whether a good γ for large context lengths falls close to this optimal value. Figure 4.6 gives the results of this test.

The graph confirms the supposition that a higher γ always yield better results. However, there is a clearly higher increase when going from $\gamma = 0.7$ to 0.8. This is present in all three curves; remember, now, that the optimal value of γ in the task where order was important (Section 4.2.2) was found to be lower than 0.9, possibly in the range of 0.6 to

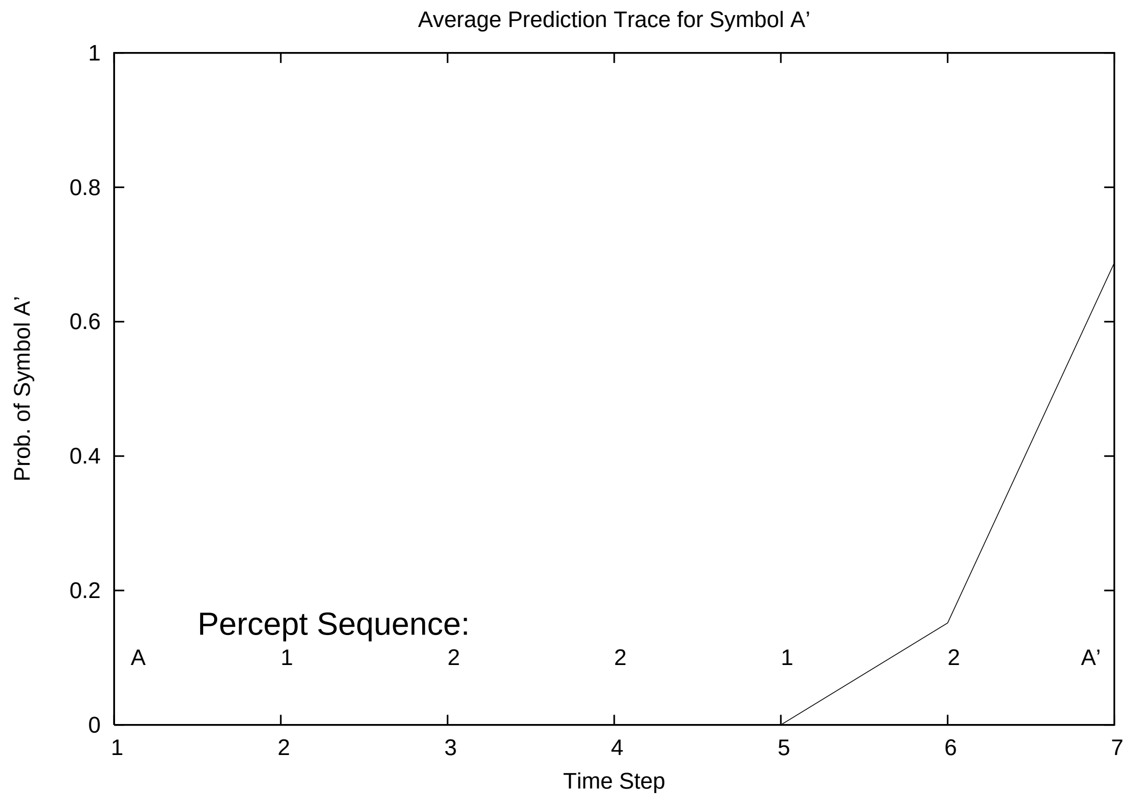

FIGURE 4.5. Average prediction trace for the end symbol A'. The x-axis represents timesteps through the episode. Values are given for a test episode ran after 100,000 training episodes, with $L = 5$, and averaged over 30 trials.

0.8, depending on the length of the context sequence to be disambiguated. It would then be reasonable, if one wished to combine both properties together and optimize them, to take $\gamma = 0.8$. Although further, more extensive tests clearly need to be undertaken before such a conclusion can be verified, it is interesting to again point out the relationship between γ, the saliency decay factor, and the λ value which controls the rate at which eligibility traces decay. In [Singh and Sutton, 1996], the optimal value for λ was also experimentally found to be 0.8.

Regardless of the optimal value, $\gamma > 0.6$ is certainly needed, since even for $\gamma = 0.6$ the algorithm has a low learning rate for the large n studied above. To be consistent with the experiment on the symbol ordering, then, only values of γ between 0.5 and 0.8 should be used.

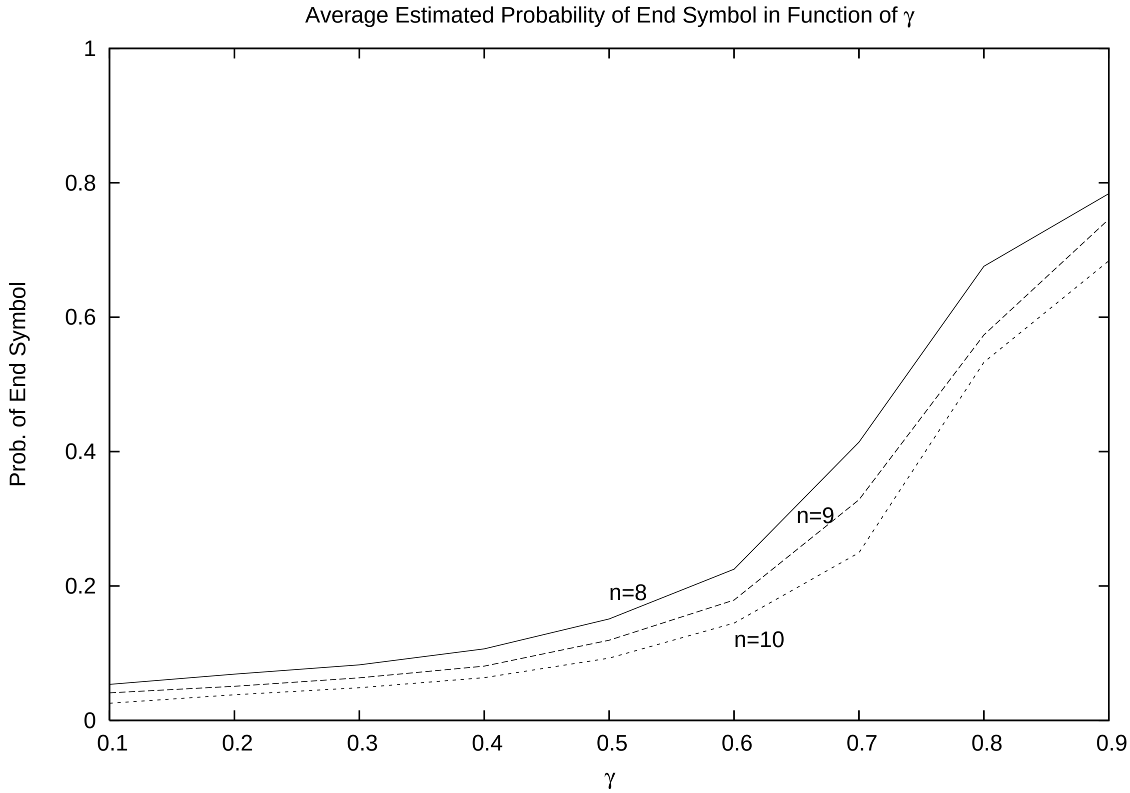

FIGURE 4.6. Average estimated probability of observing the correct end symbol in function of the saliency decay factor, γ. Three values of n were used, namely $n = 8$, $n = 9$ and $n = 10$. Training was done in all cases with 10000 episodes.

To conclude this section, my proposed algorithm has the advantage over Markov Models that it can learn to predict based on any history length, without resorting to a hidden state. The skeptic reader, however, should wonder whether such a property holds even when the 'junk' takes a less ordered aspect. Indeed, in the above task, the number of intervening symbols between the important context and the target symbol is known; furthermore, the type of junk that is observed is pre-determined, such that the algorithm can rely on it to *count* how long it should wait before emitting the correct symbol. As such, it would be of interest to look at how the algorithm reacts to context sequences of *variable* length.

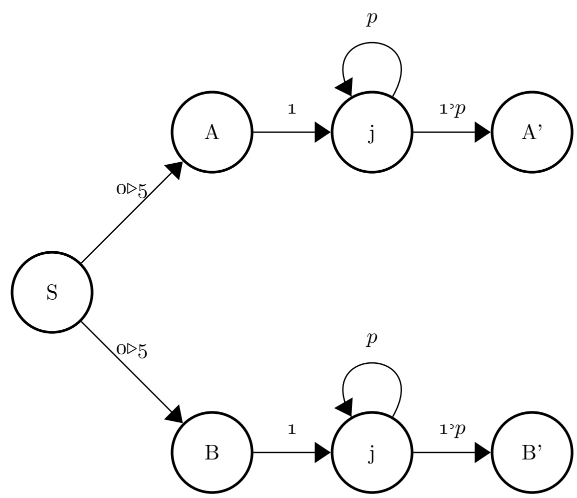

FIGURE 4.7. Finite State Automaton describing the Variable Length Context task used in this section.

4.3.2. Variable Length Context

The logical extension of the task used above is to simply make the length of the ‘junk’ sequence non-deterministic. More formally, consider a Finite State Automaton with probabilistic transitions as given in Fig. 4.7.

This task is necessarily harder than the previous one, since the algorithm cannot rely on counting. In fact, the correct probability distribution that it should predict stays the same throughout the sequence - any amount of junk should be ignored. It is easy to see that the algorithm cannot properly handle this situation, since the saliency of the first symbol must decrease with each time step. Rather than show the probability of the end symbol, I will therefore give results in term of the KL divergence, discussed in Section 4.1. Here, D_{KL} is given over the whole episode, averaged over all timesteps (so that results can be compared across sequences of different lengths). At the very least, one would expect the algorithm to learn to reduce D_{KL} to a manageable value.

Let p be the probability of transiting to one of the end states. Note that the expected number of junk symbols that should be observed before the end symbol is $\frac{1}{p}$. For experimental purposes, I will consider values of $p \in \{0.1, 0.25, 0.333, 0.5\}$, yielding expected

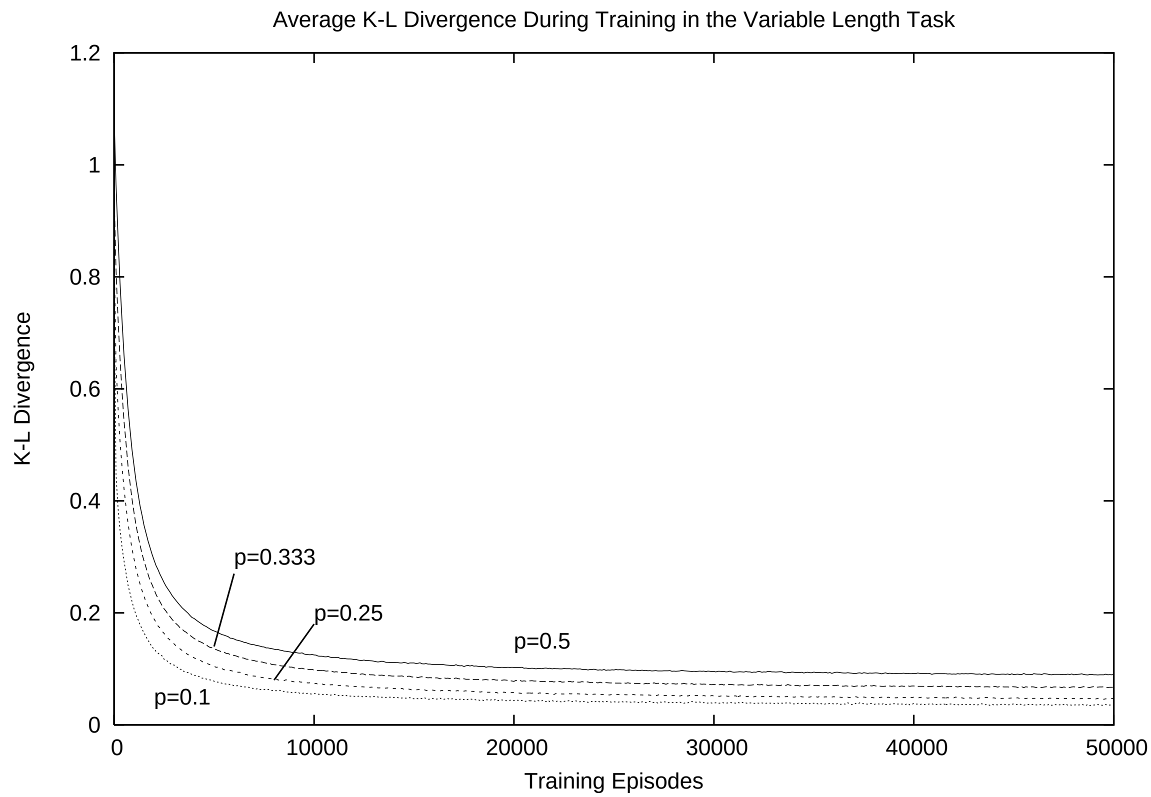

FIGURE 4.8. Average KL divergence in function of training time and p for the Variable Length task.

junk lengths of 10, 4, 3 and 2, respectively. I ran the algorithm on these four tasks for 50000 episodes, reporting the KL divergence every 100 steps. Due to the difficulty of this problem, I chose a smaller learning rate than before, namely 0.01. Also, based on the conclusions drawn from the previous experiment, I picked $\gamma = 0.8$.

Nothing is truly surprising in the results depicted in Figure 4.8. The algorithm converges after sufficiently many episodes, although it cannot bring D_{KL} down to 0. The reason for this is clear from the explanations above: there is no deterministic way for the algorithm to decide when the symbol will appear, such that it must have some error. More interestingly, however, is the fact that as p is decreased towards 0 the divergence decreases faster, and seems to settle down at a lower value. One would expect the higher p values to produce better results, since it is easier for the algorithm to predict when the end symbol will occur. However, having a small p means that it is in fact a better guess to predict that the end symbol will *not* occur with high probability. That is to say, if two outcomes are considered (junk or end symbol) then the entropy of the system is at its highest when $p = 0.5$. In general, however, it can safely be claimed that the algorithm can learn - at least partially - the structure of the problem.

One question that remains is whether the weights of the model stabilize, or slowly grow. The latter situation, even if it produced a correct distribution, would not be desirable. First of all, higher weights means that smaller changes in their magnitude will cause a greater effect on the estimated probabilities, and therefore higher variance in the results. Also, large weights make it harder for the algorithm to learn from new data, by reducing the probability of unobserved outcomes. Finally, it would be interesting to see the weights stabilize simply in order to say that the algorithm truly converges. In order to answer whether this is the case, I show in Figure 4.9 the progression of various weights over the training time, for a given trial.

Conclusions drawn from this plot should not necessarily be taken too seriously, as it is always hard to evaluate an algorithm's performance based on the magnitude of its weights. Nevertheless, the three types of weights that we show in Fig. 4.9 reflect the three expected behaviors: the greatest influence on the end symbol is found in the start symbol - which defines the end symbol -, while a small influence is given to the junk symbols, as discussed previously. Finally, the start symbols (A and B) have a negative influence on the other (respectively B', A') end symbol, which makes sense. The weights $W_{0,A'}$ and $W_{0,B'}$ seem to be stabilizing over time, which is good. However, the other pairs of weights continue to increase (decrease) over time, although the rate at which they do so decreases.

Through this experiment, I have hopefully shown that the algorithm is sufficiently resilient to probabilistic tasks and can accomodate for them. Clearly, there are better models than this one for such a task: but my claim is that this necessarily requires the notion of a hidden state, which I seek to avoid.

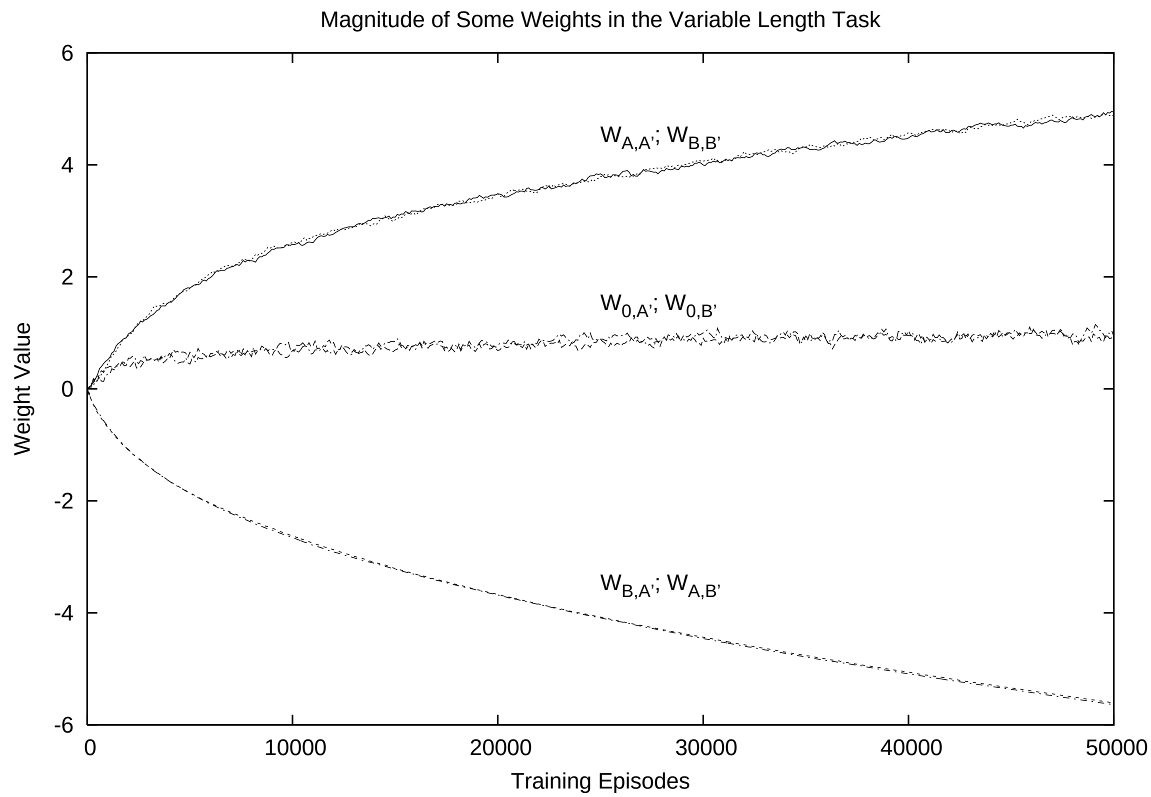

FIGURE 4.9. Weights evolving over time for the Variable Length task. Six weights are represented here, namely three per possible sequence (A, junk, A' or B, junk, B'). The top weights show the influence of the beginning symbol on the end symbol; the middle ones show the influence of junk symbol 1 on the end symbol; and the bottom ones show the (negative) influence of the opposite start symbol on a given end symbol. Weights here were plotted for a single run.

4.4. Discussion

The goal of the above experiments was to demonstrate that my proposed algorithm can learn to predict simple sequences which can be learned by k^{th} order Markov Models. The main loss is that the exactness of a Markov Model is lost in modeling the history more succintly. On the other hand, this means that this history does not need to be represented as a cumbersome table. I showed that frequencies can be learned just as well as in a Markov Model. Symbol ordering is weakly encoded, which gives good results in some cases. However, there are clearly cases where the algorithm must fail, such as the one which was presented in Section 3.4. Finally, the size of the sequence which the algorithm must learn to 'ignore' (the value of k) is also weakly handled in the model, by the virtue of the exponentially decaying saliency value.

Prediction Suffix Trees and their ilk are arguably better at producing variable length Markov Models, but all the algorithms presented in this thesis require a large amount of history stored. The proposed algorithm, on the other hand, proceeds in an on-line fashion, which is often desirable of learning algorithms. Its slow learning compared to other frameworks can be partially attributed to this; neural networks often learn at a similar rate, because they also rely on gradient descent and on-line learning. Yet, this is not a neural network, as the only abstraction occurs at the level of the hard locations.

In general, only models which keep a strong notion of state can learn a task like the one presented at the beginning of the chapter. PSTs, for example, will require the construction of ever longer tree branches to explain the behavior of the system. In this respect, my algorithm might perform better on this task because it cannot add new branches *ad infinitum*.

Another interesting aspect is the simplicity of the algorithm. Partially because of this, it exhibits robust behavior across a range of parameters, which is often not the case for gradient-descent approaches such as neural networks. Its oversimplified hypothesis space, however, might be the cause for poor performance in larger tasks. For example, the context vector cannot fully account for repeated symbols in a sequence. One way to circumvent this would be to add features which are only active if a symbol is repeated twice. The simplest way of achieving this is by letting such a feature's inputs be the activation of another hard location. It could then be activated only if the other location's activation was γ, indicating a symbol which occured one time step ago, combined with re-setting the activation to 1 rather than $1 + \gamma$, as is done in replacing eligibility traces [Singh and Sutton, 1996]. The next chapter will focus on the addition of such units in an attempt at generalizing the algorithm to handle more complex problems.

CHAPTER 5

Feature Construction

Chapter Outline

In this chapter, I propose to enhance the prediction algorithm by automatically constructing additional features.

5.1. The Need For Feature Construction

The experiments in the previous chapter showed that my proposed algorithm can indeed model various probability distributions, in a way that resembles a k^{th} order Markov Model, without requiring the user to explicitly define k. However, this method does not work perfectly in all instances. One such problematic case is when symbols are repeated; I brought the issue forward previously, without attempting to resolve it.

To exemplify the problem, suppose that the agent is faced with the task of learning how to correctly predict symbols based on two equally probable strings of observations, namely 'AAB' and 'B'. As before, we can suppose that we have three different symbols, A, B and 0, where 0 is a special symbol which is never observed but represents the beginning of the sequence (and therefore our two sequences are '0AAB' and '0B'). Let $w_{i,j}$ denote the weight from i to j, where i and $j \in \{0, A, B\}$. Also, let s_i be the saliency of a given observation; for conciseness, I will denote the whole vector by $\vec{s} =< s_0, s_A, s_B >$. Recall that π_i is the probability of a given observation and β_i is its activation, such that

$$\pi_i = \frac{e^{\tau\beta_i}}{\sum_j e^{\tau\beta_j}} \tag{5.1}$$

(i) The two sequences are equiprobable; so in order to correctly predict the initial symbol (A or B), the agent must assign a weight $w_{0,A} = w_{0,B}$ such that $\pi_A = \pi_B = \frac{1}{2}$ after observing only 0.

(ii) Note that in the first sequence, '0AAB', the following saliency vectors occur: $< 1, 0, 0 >$, $< \gamma, 1, 0 >$ and $< \gamma^2, 1 + \gamma, 0 >$. Provided that the first symbol is correctly predicted, then a perfect prediction would be as follows:
If $\vec{s} =< \gamma, 1, 0 >$ then $\pi_A = 1$; if $\vec{s} =< \gamma^2, 1 + \gamma, 0 >$ then $\pi_B = 1$.

(iii) It is impossible to achieve $\pi_A = 1$ with my algorithm, since clearly $\pi_i > 0 \; \forall i$. A relaxation of this is to require that $\pi_A >> \pi_B$ in the first case and $\pi_A << \pi_B$ in the second case. This leads to $\beta_A >> \beta_B$ and $\beta_B >> \beta_A$, respectively. Expanding these variables gives:

$$
\begin{aligned}
w_{0,A}\gamma + w_{A,A} &>> w_{0,B}\gamma + w_{A,B} \\
w_{0,A}\gamma^2 + w_{A,A}(1+\gamma) &<< w_{0,B}\gamma^2 + w_{A,B}(1+\gamma)
\end{aligned}
\tag{5.2}
$$

(iv) The second sequence, '0B', enforces that $w_{0,A} = w_{0,B}$. This leads to the obvious impossibility here that $w_{A,A} >> w_{A,B}$ and $w_{A,B} >> w_{A,A}$. Therefore, no assignment of the weights $w_{i,j}$ can produce a correct predictive model of this task. Note that the second sequence is necessary in this counter-example, as the system of inequalities would otherwise have a solution.

The question of interest is then whether this is caused by the repetition of symbols, or by something more fundamentally flawed in the proposed algorithm. One hypothetical reason for this problem is that the data is not linearly separable. Linear separability refers to the notion of separating labeled points on a plane with a single line. The points are labeled either positive or negative, and are to be separated according to this criterion. Much work has been done on linear separability, especially in the field of neural networks [Rujan and Marchand, 1989]. Since in this task $s_B = 0$ at all time steps where prediction is needed,

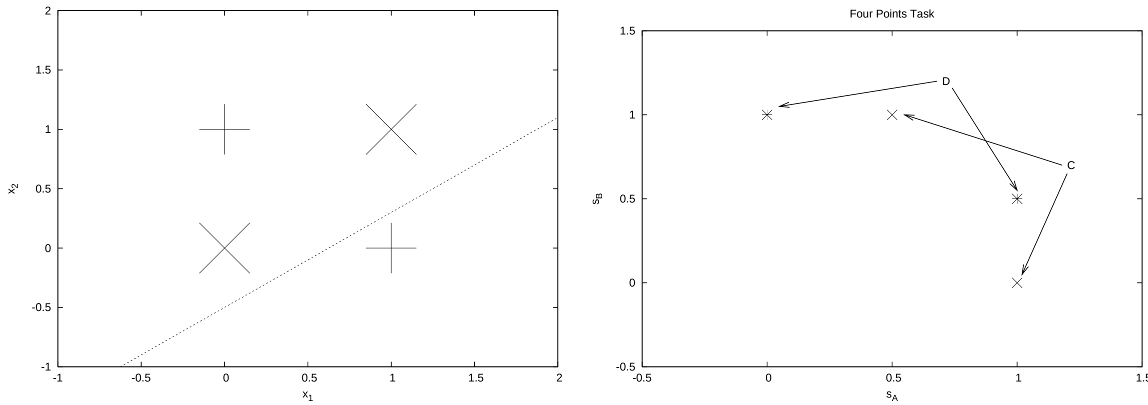

FIGURE 5.1. Left: A two-dimensional version of the parity task. Right: The four points task in saliency space.

we can visualize the task as three points to be linearly separated on the real line. The first of these points is at $s_A = 0$, the second as $s_A = 1$ and the third, as $s_A = 1 + \gamma$. Clearly, this is not something which can be achieved.

If linear inseparability is the cause of the algorithm's failure, then an additional hypothesis can be formulated: there should be a set of training sequences which does not contain repeated symbols, but which cannot be correctly modeled by the algorithm. The simplest task one can study is the *parity problem*, considered to be the paradigm of linearly inseparable binary functions. Formally, the parity problem has n binary inputs, denoted $\{x_1, \dots x_n\} = \vec{x}$. The parity function $p(\vec{x})$ is 1 if there is an odd number of 1's in the input vector, and 0 otherwise. A two-dimensional version is shown in Figure 5.1. It is easy to see that regardless of how the separating line is placed, it cannot neatly divide the space into a region containing only positive points and one containing only negative points.

In order to respect the nature of the saliency vector - it represents history - the parity task must be modified. This is because it is impossible for the system to observe the saliency vector $< 1, 1 >$, since it would imply that two percepts had been triggered simultaneously. Rather, I propose the following task: there are four symbols, A, B, C and D. Each episode consists in the agent receiving one of the following four equally likely sequences: $\{ABC, AC, BAD, BD\}$. These four sequences are represented in terms of the saliency of A and B in Figure 5.1. I call this task the 'Four Points Task', simply because it consists in four points in space which cannot be linearly separated. Again, it is easy to see that no single straight line will separate the data into a region containing the points labeled 'D', and one containing the points labeled 'C'. This task is interesting because no percept is repeated; if the model must fail, it is solely due to linear inseparability.

Empirically, I verified that the algorithm, as it is, indeed cannot give a correct distribution over the four percepts. In all runs, the last symbol of one of the two sequences of length 3 (or of both) was incorrectly predicted to occur with 50% or less (when in fact we can exactly predict it). The two smaller sequences are more accurately predicted, although the modeled probabilities still fall short of the real values. Interestingly enough, removing *any* of the four points causes the algorithm to learn the correct model. This is a strong indication that linear separability is indeed the root of the problem here. More formal empirical results will be given later.

5.2. Constructing Features Based on Error

The issues put into light by the Four Points Task suggest that linear separability should be handled explicitly in the proposed predictive model. In classical supervised learning algorithms which rely on a threshold function, separability is handled by adding additional variables into the equation. In neural networks, the hidden layer(s) play this role: they divide the input space into partitions which can then be combined in order to obtain a good (but no longer linear) separation of the state space. Cascade-correlation neural networks, discussed in 2.5.3, offer a way of doing this automatically, in effect adding hidden units when the data is not linearly separable.

In the same way, I propose that the predictive algorithm can make use of extra features which depend on the current saliency vector and can be used to improve prediction accuracy. For example, suppose that in the 'AAB' / 'B' task, there is a feature - a variable - ϕ which is not directly observed but such that $\phi = 1$ if $s_A > 1$. Then suppose the weight matrix W is extended such that for each original percept i, there is now a weight $w_{\phi,i}$. The β values are then computed as $\beta_i = (\sum_j w_{j,i} s_j) + w_{\phi,i}\phi$. Equation 5.2 then becomes (after simplification of the $w_{0,A}$ and $w_{0,B}$ terms):

$$\begin{aligned} w_{A,A} &>> w_{A,B} \\ w_{A,A}(1+\gamma) &<< w_{A,B}(1+\gamma)+w_{\phi,B} \end{aligned}$$

There is a simple assignment of weights which satisfies both inequalities. This suggests that adding one or more such features can help in modelling more complex tasks.

One way to visualize this is to consider the addition of an extra dimension to the saliency space, and the value of points along this dimension is ϕ. From a linear separability point of view, this allows for the separation of points that were previously unseparable. For the Four Points Task, the simplest way to resolve the problem is to add a feature that takes value 1 when one of the four points of interest is present.

My proposed refinement to the algorithm is therefore the following:

(i) Learn a model using the basic algorithm
(ii) Look for which points cannot be linearly separated
(iii) Add features that allow their separation
(iv) Repeat until all predictions are correct

There are, however, a few problems with this proposal. First and foremost, it is not clear how to determine whether a set of points is linearly separable or not. More subtle is the fact that the algorithm's goal is not to classify points as positive or negative examples, as in classical supervised learning, but rather to give a probability distribution over future observations. Therefore, this is a regression problem, which is usually accepted as being harder than classification. There are also other issues that I will discuss below.

5.2.1. Finding Linearly Inseparable Points

In order to come to a working algorithm for the prediction problem, it is useful to view it as a supervised learning task. In supervised learning, the algorithm is given a set of input vectors, X, to which corresponds a set of output vectors Y. For simplification, assume that

Y is univariate. The task of the system is to learn a map from inputs to output, such that when presented with a new instance, $\hat{x}$, the system can produce the appropriate $\hat{y}$.

It is clear that from a supervised learning point of view, removing data can only make the training easier (although the results on independent data might suffer). More specifically, if a set of points X is linearly separable (such that the output y_i can be predicted from x_i simply by determining on which side of a hyperplane x_i lies) then removing points from X preserves its linear separability.

Using this fact, I suggest the use of a partial training set in order to determine whether the model is incorrectly predicting some observations. More formally, let P be the set of all possible saliency vectors which we can observe. If there are a finite number of percepts and the episodes are of finite length, then $|P|$ is also finite. Let M be the predictive model - which might not be completely correct, due to linear inseparability; call this the 'full model'. Finally, let $\hat{M}$ be a second predictive model with weights initialized to zero; call it the 'partial model'. For a given saliency vector $\vec{s}$, denote by $\pi(\vec{s})$ the probability distribution on future percepts induced by $\vec{s}$ in the model. Then

(i) Construct $\hat{P}$ by randomly sampling without replacement from P, such that $|\hat{P}| < |P|$
(ii) Train M on P and $\hat{M}$ on $\hat{P}$ for a certain number of episodes
(iii) Let $P_\phi = \{\vec{s} : \vec{s} \in \hat{P},\ \pi(\vec{s}) \sim \hat{\pi}(\vec{s})\}$
(iv) Construct a feature ϕ such that $\phi = 1$ when $\vec{s} \in P_\phi$, 0 otherwise

What does it mean to train $\hat{M}$ on $\hat{P}$? In the full model case, the system receives one percept per time step (x_t), which it uses to update the distribution that should result from the context vector ($\vec{s}_{t-1}$). A proper distribution $\pi(\vec{s})$ is obtained $\forall \vec{s} \in P$ because the sampling of x_t is done according to some underlying, environment-driven distribution. The partial model, which should be trained only on patterns $\vec{s} \in \hat{P}$, still requires data from the environment, rather than taking inputs from $\hat{P}$ and outputs from a corresponding set, as in supervised learning. The way I resolve this is by computing the saliency vector at every step for both models, but only updating the weights in $\hat{M}$ when this vector is present in $\hat{P}$.

Step 3 needs a similarity measure between two probability distributions. The KL divergence has been discussed previously; there are other measures which are seemingly better suited here. These will be described in the experimental section. The last step requires that one weight for each percept i, $w_{\phi,i}$, be added every time we add a new feature. Since we do not need to predict the added feature, only m variables are created, where m is the total number of percepts.

5.2.2. The Algorithm

My implementation of the algorithm determines whether a particular ϕ_i should be 'on' (i.e., take a value of 1) by directly storing a set of triggering vectors F_i for each feature. If $\vec{s} \in F_i$, then ϕ_i is set to 1. Another question that arises is how to determine when to test for the construction of a new feature. I chose to do this after a fixed number of elapsed episodes, l. Finally, there are many possible ways - none definitive - of determining whether two probabilitty distributions are similar. In my work, I chose to focus on two simple methods which both give a numerical value representing the difference between two distributions. The first one is the KL divergence, which has been discussed previously. The other is the total variation distance, defined as

$$D_{TV}(\pi, \hat{\pi}) = \frac{1}{2} \sum_i |\pi(i) - \hat{\pi}(i)| \tag{5.3}$$

Although at first glance simplistic, the total variation distance is of interest because it takes values strictly between 0 and 1. The KL divergence, on the other hand, takes values between 0 and ∞, making it harder to compare two results. In the probabilistic literature, the total variation distance is also a standard measure.

To conclude, the modified algorithm is presented below. For easier understanding, here is a glossary of the variables used in the algorithm:

Name	Usage	Name	Usage
W	Weights (Full model)	$\vec{s}$	A saliency vector
$\hat{W}$	Weights (Partial model)	$\vec{\nu}$	An extended saliency vector
P	Set of possible saliency vectors	$\vec{\beta}$	Un-normalized prediction vector
$\hat{P}$	Random subset of P used to train $\hat{W}$	$\vec{\pi}$	Prediction vector
N_f	Number of constructed features	$\vec{x}$	A percept
F_i	Set of inputs for which feature i is 'on'	$\alpha(\vec{x})$	Activation of cells by $\vec{x}$
γ	Saliency decay rate	τ	The Boltzmann 'temperature' parameter
c	Model learning rate		
δ	Divergence threshold	l	Number of training episodes before adding features (phase length)

5.3. Theoretical Results

In this section, I show that the modified algorithm, under certain conditions, must necessarily converge. Let $\Omega \in P$ denote the set of saliency vectors for which the model produces the correct distribution, and let Ω^C be the set of saliency vectors for which the model does not produce the correct distribution. In general, the Boltzmann distribution cannot reduce the total variation error to 0, as it always assigns a non-zero probability to every element. Instead, I consider a point $\vec{s}$ to be a member of Ω if $dist(\pi(\vec{s}), p(\vec{s})) \leq \delta$, where p is the true distribution of percepts that follows the history $\vec{s}$. In the treatment below, k is the fixed size of $\hat{P}$, a random subset of P.

LEMMA 1. *Given W and provided that $k \leq |P| - 1$, for any $\vec{q}, \vec{s} \in \Omega^C$ there exists $P_\theta, P_{\theta'} \in P$ such that $P_\theta - P_{\theta'} = \vec{q}$ and $P_{\theta'} - P_\theta = \vec{s}$. Furthermore, if P_θ and $P_{\theta'}$ are taken as the sets of inputs for which features θ and θ' take the value of 1, then there exists a new set of weights $\hat{W}$ such that $\Omega \cup \{\vec{q}\} \subseteq \hat{\Omega}$, where $\hat{\Omega}$ is the set of histories correctly predicted by $\hat{W}$.*

Algorithm 2 The Modified Context-Driven Prediction Algorithm

```
Initialize W ← 0; P is assumed given
N_f ← 0
loop
  Ŵ ← 0
  P̂ ← random subset of P
  repeat
    s⃗ ← 0
    repeat
      ν⃗ ←EXTEND(s)
      Observe next percept x⃗
      UPDATE-MODEL(ν⃗, α(x⃗), W)
      if s⃗ ∈ P̂ then
        UPDATE-MODEL(ν⃗, α(x⃗), Ŵ)
        s⃗ ← γs⃗ + α(x⃗)
      end if
    until end of episode
  until l episodes have been run
  F ← ∅
  for all s⃗ ∈ P̂ do
    if (dist(π(s⃗), π̂(s⃗) > δ) then
      F ← F ∪ s⃗
    end if
  end for
  if F ≠ ∅ then
    F_{N_f} ← F
    N_f ← N_f + 1
  end if
end loop
```

Algorithm 3 EXPAND($\vec{s}$)

```
for i = 1 . . . N_F do
  if s⃗ ∈ F_i then
    φ ← 1
  else
    φ ← 0
  end if
end for
ν⃗ ← s⃗· < φ_1, φ_2, . . . , φ_{N_f} >
```

Algorithm 4 UPDATE-MODEL$(\vec{x}, \alpha, W)$

$\beta \leftarrow W\vec{\nu}$
$\sigma \leftarrow \sum_i e^{\tau\beta_i}$
for $i = 1 \ldots m$ **do**
  $\pi_i \leftarrow \frac{e^{\tau\beta_i}}{\sigma}$
end for
Compute $\nabla W = \frac{\partial}{\partial W_{i,j}}\mathscr{L}$
$W \leftarrow W + c\nabla W$

PROOF. Construct P_θ and $P_{\theta'}$ as follows: $P_\theta = \{\vec{q}\} \cup \{\vec{s}_1, \ldots, \vec{s}_{k-1}\}$ where $\vec{s}_1, \ldots, \vec{s}_{k-1}$ are $k-1$ different points in P other than $\vec{q}$ and $\vec{s}$. Then $P_{\theta'} = \{\vec{s}_1, \ldots, \vec{s}_{k-1}\} \cup \{\vec{s}\}$, where $\vec{s} \notin P_\theta$. Then set $\vec{W}_\theta$, the weight vector corresponding to θ, proportional to $p(\vec{q})$, the true probability distribution over future percepts when $\vec{q}$ is the history. Finally, set $\vec{W}_{\theta'}$ to $-\vec{W}_{\theta'}$. Clearly, $\vec{s}_1, \ldots, \vec{s}_{k-1}$ remain unchanged by this operation: for any $\vec{s}_i$, $i \in 1 \ldots k-1$, both θ and θ' are activated when the context vector is $\vec{s}_i$. If ϕ_i denotes the vector of activations $W\vec{s}_i$ before adding the features θ and θ', and $\hat{\phi}_i$ the same vector after the features are added, then $\hat{\phi}_i = W\vec{s}_i + \vec{W}_\theta + \vec{W}_{\theta'} = W\vec{s}_i = \phi_i$. With this construction, $\pi(\vec{q})$ can be made arbitrarily close to $p(\vec{q})$ without removing points from Ω. Therefore, $\Omega \cup \vec{q} \subseteq \hat{\Omega}$, as required. □

Note that $\pi(\vec{s})$ is most likely made incorrect by the above construction.

LEMMA 2. *Given a set of histories P and provided $k \leq |P| - 1$, there exists a set of features $F = \{P_{\theta_1}, P_{\theta'_1}, \ldots, P_{\theta_{N-1}}, P_{\theta'_{N-1}}\}$ to which correspond a weight matrix W such that $\Omega = P$ and $N = |P|$.*

PROOF. Denote the set of points in Ω^C by $\{\vec{q}_1, \vec{q}_2, \ldots, \vec{q}_N\}$. Denote $\vec{q}_N$ as $\vec{b}$ and order Ω^C such that $\vec{b}$ has exactly one non-zero element. We can iteratively apply Lemma 1 with $\vec{q} = \vec{q}_i$ and $\vec{s} = \vec{q}_{i+1}$, from $i = 1$ to $N-1$ to yield a set of features $F = \{P_{\theta_1}, P_{\theta'_1}, \ldots, P_{\theta_{N-1}}, P_{\theta_{N-1}}\}$. Lemma 1 tells us that there is a set of weights $\tilde{W}$ such that $P - \{\vec{b}\} \subseteq \tilde{\Omega}$, where $\tilde{\Omega}$ is the set of histories that $\tilde{W}$ correctly models.

In the worst case, $\vec{b} \notin \tilde{\Omega}$. Let j be the index such that $b_j \neq 0$. Let $W_{\theta_1}, \ldots, W_{\theta_{N-1}}$ and $W_{\theta'_1}, \ldots, W_{\theta'_{N-1}}$ be the weight vectors corresponding to the constructed features $\theta_1, \ldots, \theta_{N-1}$ and $\theta'_1, \ldots, \theta'_{N-1}$. Note that $W_{\theta'_i} = -W_{\theta_i}\ \forall i = 1 \ldots N-1$. Denote the weight vector corresponding to the j^{th} element of the context vectors by W_j. Let $q_{i,j}$ be the j^{th} element of the vector q_i, and denote the vector of activations Wq_i by ϕ_i. We have that $\Omega = P$ if $\phi_i = a_i \, \forall i = 1 \ldots N$, where a_i is proportional to $p(\vec{q_i})$ and large. With the above construction, this yields the following:

$$\begin{aligned}
\phi_1 &= W_{\theta_1} + W_j q_{1,j} = a_1 \\
\phi_i &= W_{\theta_i} - W_{\theta_{i-1}} + W_j q_{i,j} = a_i \quad \forall i = 2 \ldots N-1 \\
\phi_N &= -W_{\theta_{N-1}} + W_j q_{i,j}
\end{aligned}$$

This system of equations can be put in matrix form, with $W_{\theta_1}, \ldots, W_{\theta_{N-1}}, W_j$ as variables, and shown to have a solution, provided $\sum_{i=1}^{N-1} q_{i,j} + 1 \neq 0$, which is clearly true in this case since $q_{i,j} \geq 0 \, \forall i$. The solution gives a set of weights W such that $\phi_i = a_i \, \forall i = 1 \ldots N$, and therefore such that $P = \Omega$. To conclude, there is a set of features, as constructed above, for which we can find a weight matrix W that causes all history vectors to be correctly modeled. □

THEOREM 1. *Given a set of histories P, a sufficiently small learning rate and provided that $k \leq |P| - 1$, an algorithm which adds features based on random subsets of P must converge to $\Omega = P$.*

Note that the above theorem assumes that features are triggered by any saliency vector in their subset. This provides a second possible implementation of the algorithm, albeit one which would be expected to perform worse, especially for high k.

PROOF. Lemma 2 tells us that there exists a set of features F to which corresponds a weight matrix W such that $P = \Omega$. Since features are added uniformly randomly, it is easy to see that all features in F must eventually be added. Furthermore, it is well known that gradient descent on a linear approximation converges to the global minimum. Therefore, we are assured that

(i) There is a time t_1 at which all $f \in F$ will have been added.

(ii) There is a time t_2 after which W_{t_2} is sufficiently close to W, or is a better solution.

By Lemma 2, it follows that after t_2 $\Omega = P$. $\square$

THEOREM 2 (Main Theorem). *Given a set of histories P, a sufficiently small learning rate and provided that $k \leq |P| - 1$, the modified algorithm presented in this chapter must converge to $\Omega = P$.*

Note that Lemma 2 cannot directly be used, as it requires that $|P_i| = k \; \forall i$.

PROOF. For a history $\vec{q}$ to be added to a feature set P_i, it must be that $\vec{q} \in \Omega^C$, by the algorithm's definition. Consider a random subset P_{θ_i} from Lemma 2. In this case, only a subset $Q_{\theta_i} \subseteq P_{\theta_i}$, such that $Q \subset \Omega^C$, actually corresponds to the feature. However, for any $\vec{q} \in Q_{\theta_i}$, then either $\vec{q} \in Q_{\theta'_i}$ or $\vec{q} \notin \left(P_{\theta_i} \cup P_{\theta'_i}\right) - \left(P_{\theta_i} \cap P_{\theta'_i}\right)$; the converse is also true. In effect, for two subsets P_{θ_i} and $P_{\theta'_i}$, their intersections must contain the same points in Ω^C. From this, it can be seen that Lemma 2 can still be applied. The rest of the proof follows Theorem 1. $\square$

CHAPTER 6

Experimental Results In Feature Construction

Chapter Outline

This chapter discusses various experiments that were performed with the modified algorithm in order to verify its capacities.

6.1. Goals and Methodology

This section takes a similar direction as the previous one which described experimental results concerning the simple algorithm. My chief goal here is to study how feature construction helps the prediction of sequences, and whether this can be done with greater efficiency than if a simple Markovian Model was used. Recall that a full k^{th} order model with m different possible symbols is usually stored as a probability table, requiring $O((m-1)m^k)$ parameters. It is possible to reduce this by storing experience in a tree and only considering sequences that in effect occur, as was done in [Ron *et al.*, 1996; McCallum, 1995]. At the very least, the system should not add *more* features than the number of sequences which are observed; in the best case, it should do so only for the restricted number of sequences which are incorrectly predicted by the basic model. The number of parameters which my algorithm requires is $O(m^2 + mN_f)$.

Throughout this chapter I will refer to the algorithm which adds no features as the 'basic' algorithm. Unless otherwise mentioned, the parameters used were $\gamma = 0.8$ and $c = 0.1$. The random subsets $\hat{P}$ are made by sampling without replacement half of the points in the full set of possible histories, P. Results are given as an average of 30 independent trials.

Most of the performance reported in this chapter takes two forms: the KL divergence (see Section 4.1), which compares two probability distributions, and the number of features added, which is a measure of the complexity of the resulting system. The KL divergence reported during training is an average over 200 test episodes. I used the total variation divergence criterion in the experiments, as it was found to be more robust.

6.2. The Four Points Task

6.2.1. Description

A good starting point to determine the validity of the feature construction part of the algorithm is to use the Four Points task (see Section 5.1), which I showed cannot be learned using the basic algorithm. To summarize the task, there are four possible observations: A, B, C and D. Each episode consists in one of the following four sequences: ABC, BAD, AC or BD. These occur with equal probability, such that the agent should initially predict A or B w.p. 0.5, then the following symbol w.p. 0.5. If the second symbol is a B or an A, then the agent should deterministically predict the last symbol. This part is the one which should demonstrate that the feature construction algorithm performs better than the basic one. For this task, each agent was trained for 10,000 episodes. The number of training episodes after which the system determines whether to add a new feature, l, was set to 1000.

6.2.2. Comparison With The Basic Algorithm

Figure 6.1 plots the performance of the basic algorithm against that of the improved algorithm. I chose to display three values of δ, the divergence threshold, to show that the superiority of this approach is not due to a tweaking of parameters.

It is clear that beyond a certain number of training episodes, the basic algorithm stops learning. Tests done with additional training episodes did not improve that performance.

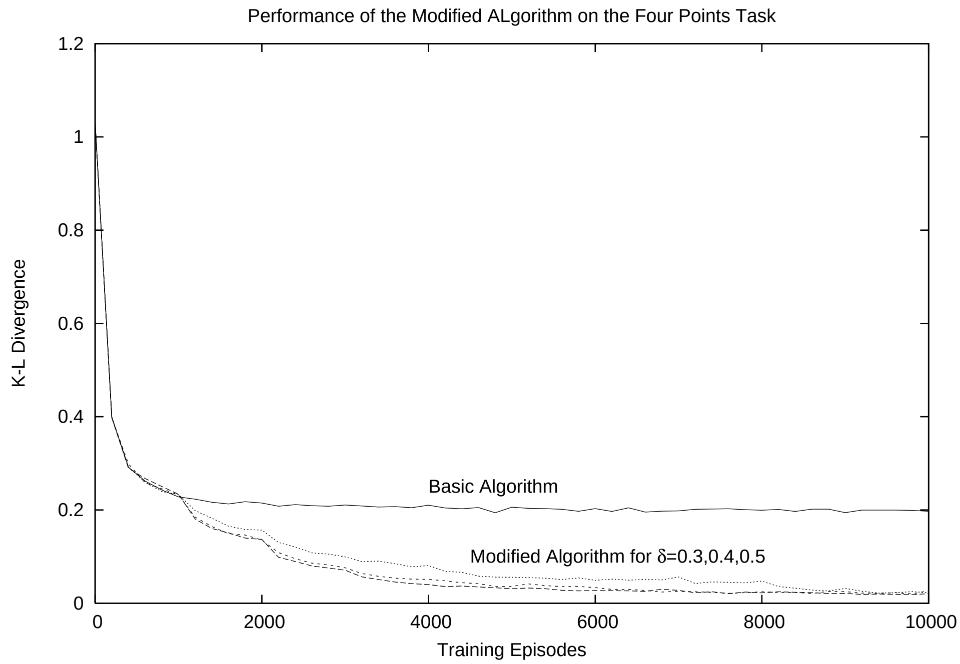

FIGURE 6.1. Performance of the basic algorithm, compared with that of the feature construction algorithm, in the Four Points task.

The feature construction algorithm, on the other hand, learns as quickly as the basic algorithm for the first 1000 episodes, then has the chance to add a feature, which results in better performance afterwards. On average, with $\delta = 0.3$ the agent added 1.35 features (between 1 and 2); this value was 1.11 for $\delta = 0.4$. For $\delta = 0.5$, the agent added only one feature, which is all that is required to learn the task. In other cases, the second feature was added before learning with the first one had been completed, yielding redundancy.

The next logical step is to study the effect of the divergence threshold on the task; especially, whether the algorithm is sensitive to the choice of threshold. Figure 6.2 shows how the performance of the algorithm (in terms of the average KL divergence over 200 test episodes) varies with the choice of δ. Simultaneously, it also shows the number of features added in fuction of δ. Unsurprisingly, performance increases as δ is reduced, whereas the number of features increases. One thing to keep in mind is that with higher values of δ, units were added at a later point during the training, which resulted in a higher D_{KL} simply because the algorithm could not progress beyond a certain point without an additional feature. In all cases, the number of features added is relatively small, and the

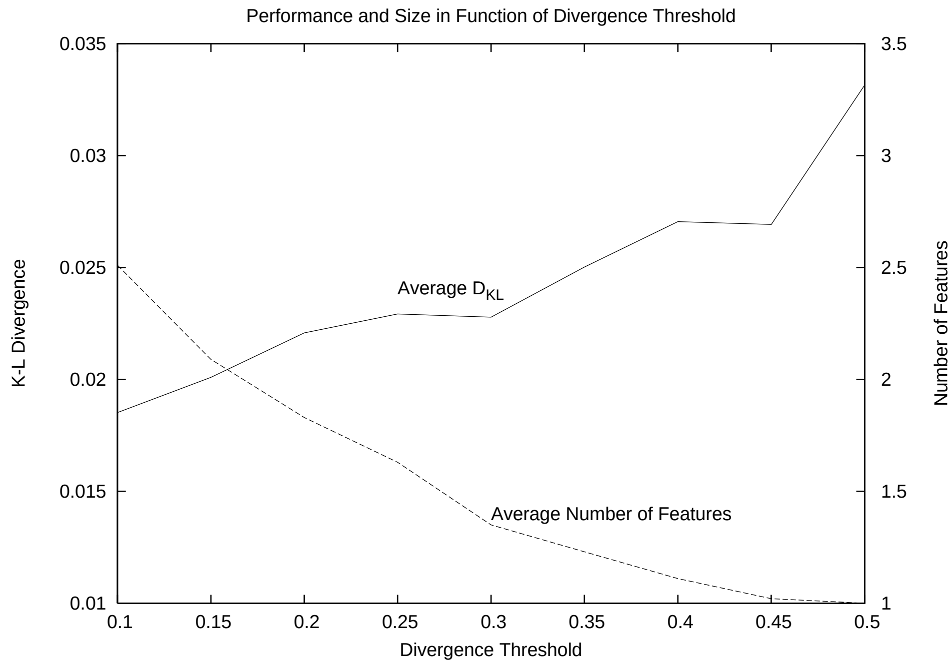

FIGURE 6.2. Effect of the divergence threshold δ on the feature construction algorithm, in terms of the number of features added to solve the task and the end KL divergence.

performance is much better than the basic algorithm's. Note that the results shown in Figure 6.2 were averaged over 100 trials rather than 30, to obtain a smoother graph. The average KL divergence for the basic algorithm is not reported on this graph as it was much higher, namely 0.197.

The feature construction therefore seems robust enough to large changes in δ, and can clearly help predicting symbols. Good values of δ are found to be between 0.3 and 0.4. Since it seems reasonable to desire that the system add features not only efficiently but also early in the training phase, a value of $\delta = 0.3$ seems the best choice, at least for this task.

6.3. The Hallway Task

The Four Points task has little merit beyond showing, empirically, that my proposed feature construction algorithm helps the system predict future percepts. A more interesting task is one that has appeared previously in the literature. For this purpose, I chose a task found in [McCallum, 1995], called the Hallway task, which has since been re-implemented as a regular POMDP.

6.3.1. Description

The Hallway task consists in a simple two-dimensional maze where the agent can move in any of the four cardinal directions. Its sensors restrict it to detect walls immediately surrounding it. The result is that many states are perceptually aliased. In effect, there are 16 (2^4) possible observations, corresponding to encoding the presence of each wall as a bit. As was discussed in Section 2.3.2, actions are not part of my proposed framework. One way to go around this problem is to view actions as observations. For example, the system could be set up to receive two percepts at every time step, one of type 'action' and the other of type 'observation'. There are many issues involved with this, however, the main one being that when there are very few actions, their weights become trained to produce an average distribution over the possible observations. Rather than doing this, I chose to create a new percept, which consists of an action-observation pair. Formally, given an observation y_t and the action following it, a_t, the percept p_t that the system receives is $< y_t, a_t >$. Everything that has been discussed so far can be done in this way. It will be important to keep this in mind in the following sections, as this means that the agent predicts the following action-observation pair. To keep results intelligible, I have chosen to report observation probabilities rather than action-observation probabilities, when necessary. KL divergence values, however, were computed using the latter. The maze I chose to use for training is depicted in Figure 6.3. The experimental setup in this task is a modified version of the previous one, since convergence times were much slower. The learning rate c was set to 0.1, and the number of training episodes to 500,000. To allow features to be added with more precision, I chose $l = 2000$.

The original task is a POMDP, which means that the goal of the agent is to discover the policy - a mapping from state, or belief state, to actions - which yields the highest sum of rewards. For my proposed algorithm, reward is not directly handled. As such, I chose to ignore this completely, and focus solely on predicting future observations. This also means

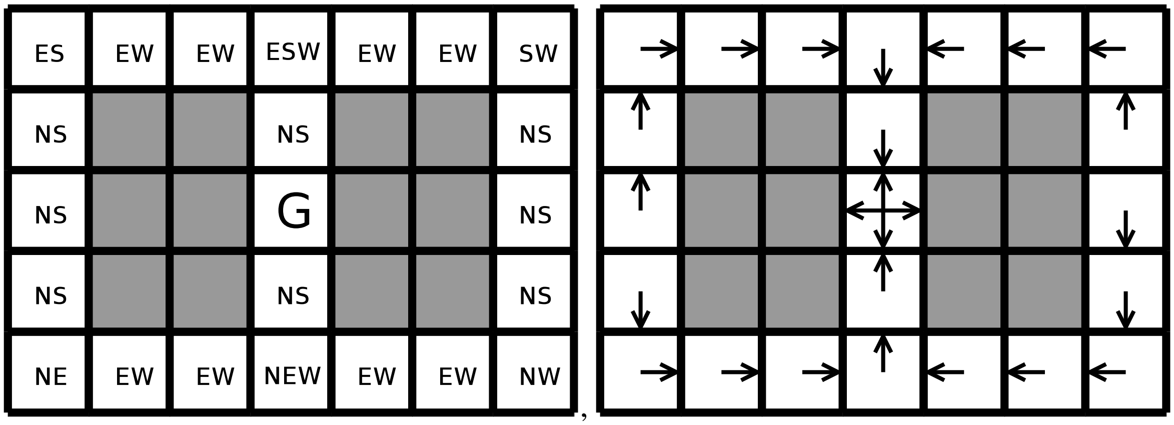

FIGURE 6.3. **Left:** A map of the hallway used for experiments. Each state is labeled with the corresponding observation: the directions that are not blocked by walls. The center state, labeled 'G', is the end state. **Right:** A depiction of the policy followed by the agent.

that there is no possibility for the agent to *improve* its policy, as there is no well-defined notion of improvement (unless one considers reducing the model's error). The goal state then becomes simply an end state - one in which the episode terminates. In this task, an episode starts in any of the non-goal states; the agent then follows a deterministic policy (this requirement will be lifted in Section 6.3.5). Figure 6.3 depicts this policy.

6.3.2. Comparison With The Basic Algorithm

The starting point of my empirical study of the algorithm's behavior when used in the Hallway task is a comparison with the algorithm which does not add features. Figure 6.4 plots the performance (in terms of the KL divergence) for this algorithm and for three values of δ, namely 0.3, 0.4 and 0.5. It also compares the growth of the feature vector over time for these three values of δ.

Clearly, for all values of δ the modified algorithm perform better than its basic counterpart. As expected, as δ is decreased the algorithm performs better and learns faster, but at the cost of many more features. In fact, for $\delta = 0.3$ the algorithm sometimes constantly adds one feature at the end of every phase of partial model learning. Many of these features are redundant as the algorithm could model the task without them if it were given more training time. As such, it seems that a value of $\delta = 0.4$ is a reasonable compromise

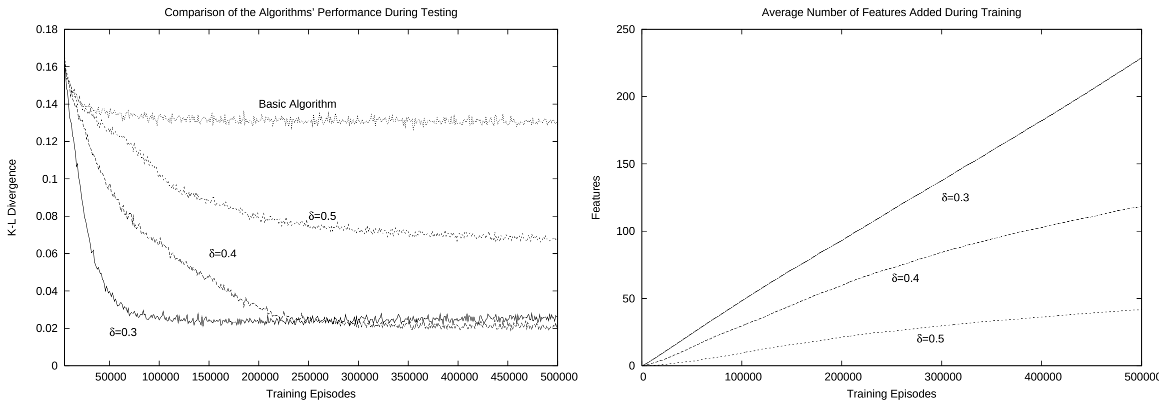

FIGURE 6.4. **Left:** Performance of the basic algorithm and of the feature construction algorithm for three values of δ (0.3, 0.4 and 0.5). **Right:** Growth of the feature vector for the same three values of δ.

between performance (it attains the same value as $\delta = 0.3$ in about as many episodes) and size. For $\delta = 0.3$ and 0.4, the number of features added seems to decrease over time, which is a good sign. It should be noted that the total number of histories that the system can encounter in this task is 317. Adding this many features means having one dimension per possible history, which would be overfitting. Here, on the other hand, the algorithm is adding fewer units, at least for $\delta \geq 0.4$. This suggests that it is achieving some level of compression. An average KL divergence, however, does not indicate where the algorithm is making mistakes and how they are making them. The next section will address this need by looking at a restricted set of testing examples.

6.3.3. Looking at an Ambiguous Sequence

One way to study more closely the performance of prediction algorithms is to focus on one single sequence from start to goal, where it is known that features are needed. The sequence which I chose for these simulations is shown in Figure 6.5.

Looking at a single sequence biases the empirical results, as it might be that the sequence I chose is one on which feature construction does particularly well. However, the feature construction algorithm acts at the level of individual, mispredicted saliency vectors.

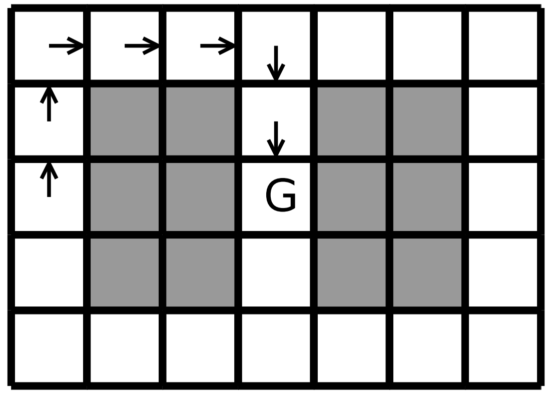

FIGURE 6.5. The testing sequence used to measure the performance of the feature construction algorithm in the Hallway task.

This approach makes sense if one considers reward, since it is often the case that very unlikely states contain high rewards (e.g. tasks that require a very precise policy, such as the Bicycle Driving task of [Randlov and Alstrom, 1998]).

The KL divergence at each step in the test sequence is reported in Figure 6.6, for the same parameter settings as before. The results are averaged over 30 independent trials. This particular sequence was chosen because it exhibits the same behavior as the 'AAB' / 'B' example given in Section 5.1 in the repeated 'NS', 'NS' observations. In fact, I trained some agents on a simpler version of the hallway task, where the system either starts in the 'ES' corner, or the start state used in this testing sequence. The results were exactly as suggested by the derivation in Section 5.1, namely that the basic algorithm cannot solve this task. As an additional verification, I trained the basic algorithm for 1 million episodes (10 times as long). No significant improvement could be seen from the results reported below.

First and foremost, it is clear that the basic algorithm completely fails at predicting the 'ES' observation, which is expected. The 'ESW' observation also seems to be significantly badly predicted, as well as the 'NS' observations and the second 'EW'. Clearly it is learning, since for some observations (including the goal) it has reduced the KL divergence to a small value. It should be pointed out that the first 'EW' percept is easily predictable because it depends on the 'ES' observation, which occurs only once and so gives the state of the system.

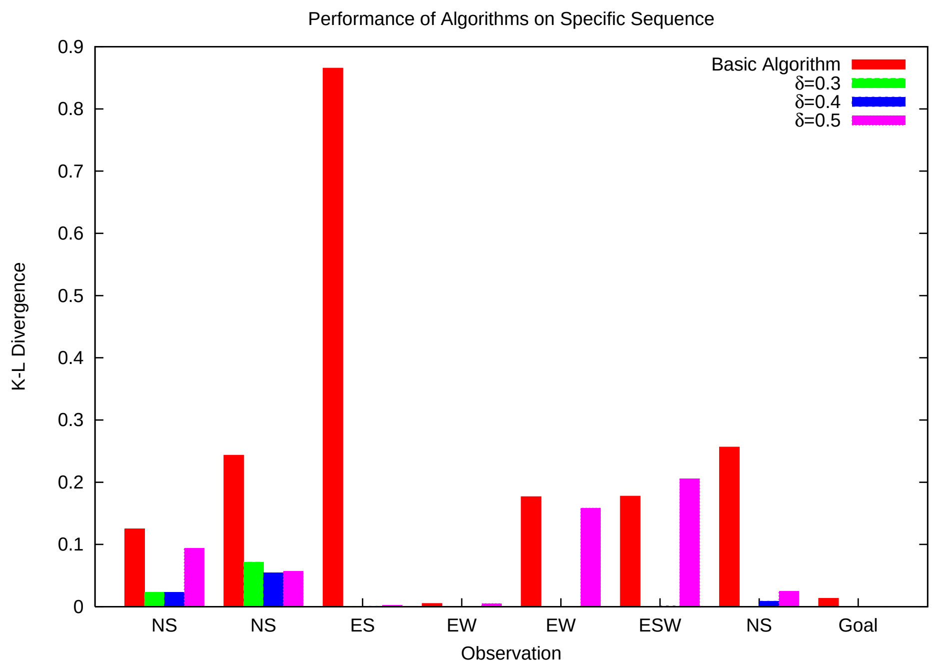

FIGURE 6.6. Performance of the basic and feature construction algorithms on a specific sequence. The observation at each time step is given under the data.

For the 'ES' and the 'NS' observations, the modified algorithm is systematically able to add features to correctly predict this. Its poor performance on the first two 'NS' can be explained by the fact that the algorithm must predict a distribution over future percepts ('NS' is one of them, along with the corner observations). The learning rate of 0.1 might restrict the algorithm from learning it completely. Nevertheless, features seem to have improved the performance at this level compared to the basic algorithm.

Finally, the case when $\delta = 0.5$ clearly shows that a higher threshold prevents from adding units which significantly help predict. This is exemplified in the second 'EW' and the 'ESW' observations, which the $\delta = 0.5$ algorithms cannot learn. On the other hand, there is no clear difference between a choice of $\delta = 0.3$ and $\delta = 0.4$, aside from many more features. The results strongly suggest that feature construction can significantly reduce the prediction error for individual patterns in the Hallway task, without requiring many additional features.

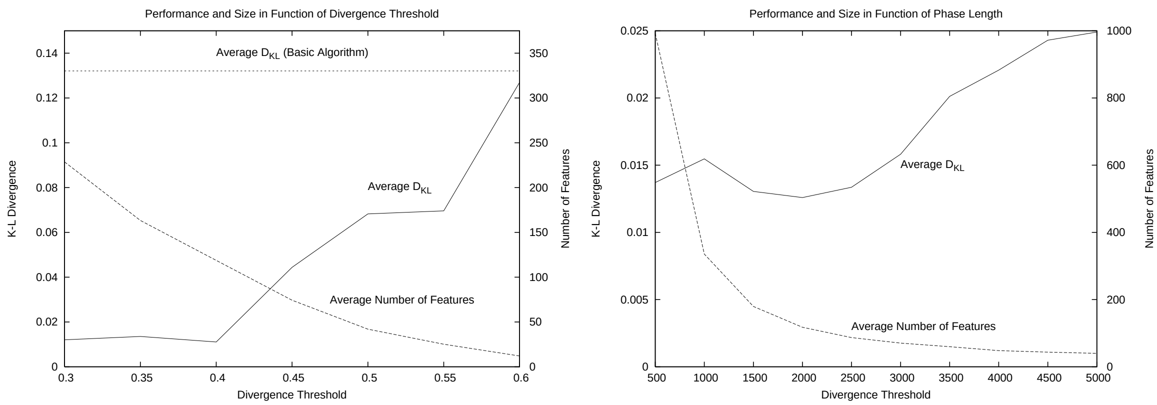

FIGURE 6.7. **Left:** Performance of the feature construction algorithm in function of δ. This performance is measured in terms of the testing sequence. **Right:** The same performance, when l is varied. δ here is 0.4.

6.3.4. Effect of Parameters on Performance

Studying the effect of the δ parameter is crucial as it controls the memory complexity of the algorithm. Clearly, if δ is too low then the system will perpetually add units; this is something which occurs in other on-line feature construction algorithms [Bellemare, 2006]. For δ too high, on the other hand, the algorithm might fail to detect patterns of interest. Furthermore, the number of episodes l between rounds of feature construction is tied to δ in the following way: in order to properly detect discrepancies between the partial and full models, the algorithm must have sufficient time to correctly re-learn the partial model. As such, I will study in this section the effects of these two parameters on the performance of the algorithm.

Figure 6.7 depicts the result of varying those parameters. Unsurprisingly, the number of features added seems to grow as δ is reduced. In all cases the algorithm does not reach the limit of 250 units it can add over time. The performance of the algorithm, however, also decreases monotonically for $\delta \geq 0.4$; there are two reasons for this. First, runs with higher values of δ do not add critical features required to minimize the error. Secondly, they may add such features at a later point in training time, due to the imperfection of the partial model. What is more interesting, however, is the fact that for $\delta \leq 0.4$, there seems to be no significant change in performance, although more features are added for lower values of δ.

This is potentially due to the fact that at this point, the algorithm has learned all that there is to be learned about the task (for a given learning rate). Regardless of δ, the algorithm performs better than the basic algorithm. This is good because in a task where there might be a large amount of noise or where there is a danger of overfitting, a high value of δ may be selected while preserving better performance. As a side note, the global KL divergence behaves in a similar fashion as the performance over the test sequence.

The results of varying the number of episodes between feature construction rounds also give a very interesting picture. Incidentally, my choice of $l = 2000$, which was only motivated by the *ad hoc* observation that $l = 1000$ yielded a high amount of features, turns out to be the best parameter choice. Apart from presenting the lowest D_{KL}, it also has a manageable number of features. In fact, for $l \leq 1000$, there are more features than histories, due to the fact that the partial model cannot be trained suitably before a feature is added. Although the results for $l \leq 2500$ are not necessarily significantly different, beyond that point the performance of the algorithm suffers. This can be explained by the fact that the system adds units at a slower rate, causing higher end D_{KL}. Of course, this measure would reach the same level if more training episodes were used, but this is something to be avoided when possible. It is also interesting to note that the number of features added seems to be exponential as l is decreased. For $l = 5000$, this is 40 features, which is quite reasonable. However, there is no telling that the algorithm would not add more features, were it given more time. Finally, one last issue to consider is that l might need to be much greater for stochastic tasks such as the one presented in the following section. Note that the average KL divergence for the basic algorithm on this task, as reported before, is 0.132, much higher than the divergence for any choice of l when $\delta = 0.4$.

6.3.5. Using a Probabilistic Policy

So far, the simulation results which I have shown all depend on the agent taking a deterministic policy. This is the kind of task on which Suffix Trees [Ron *et al.*, 1996; McCallum, 1995; Holmes and Isbell, 2006] often perform well, because the next observation is unambiguously determined given a sufficient history. In this section, I consider

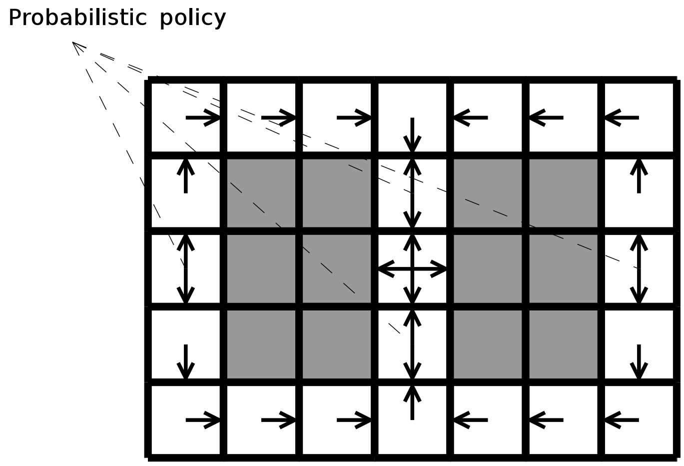

FIGURE 6.8. A depiction of the stochastic policy followed by the agent.

modifying the policy so that in some states, the agent uniformly randomly pick its next action (see Figure 6.8). The choice of these four locations is not arbitrary: they all produce the 'NS' observation, and they exhibit pairwise symmetry. For example, if the agent started only the center left and right states, then it should predict observing 'NS', 'NS' and then one of the two corner observations, based on the actions taken. Initially, I attempted to make the policy be stochastic in all states, but by the virtue of stochastic processes the expected episode length turned out to be a fairly large number.

The main challenge posed by this change in the Hallway task is that even if the starting state is known, the episode length is variable. This property of some Markov Chains was studied in Section 4.3.2 and the algorithm was found to be able to cope moderately with it. The feature construction algorithm can palliate to the problem of variable length by adding features. However, it cannot cleanly deal with such a system in general, since this requires explicitly considering loops, as is done in [Holmes and Isbell, 2006], or using a belief vector, as is done in HMMs and POMDPs. For the first 1000 episodes, empirical tests yielded between 1200 and 1400 possible sequences, much more than the previous 317.

Figure 6.9 compares the performance of the basic and modified algorithms when trained on the stochastic policy with their performance when trained with the deterministic policy. Results on the stochastic policy for $\delta = 0.2$ are also shown. For $\delta = 0.4$, the algorithm

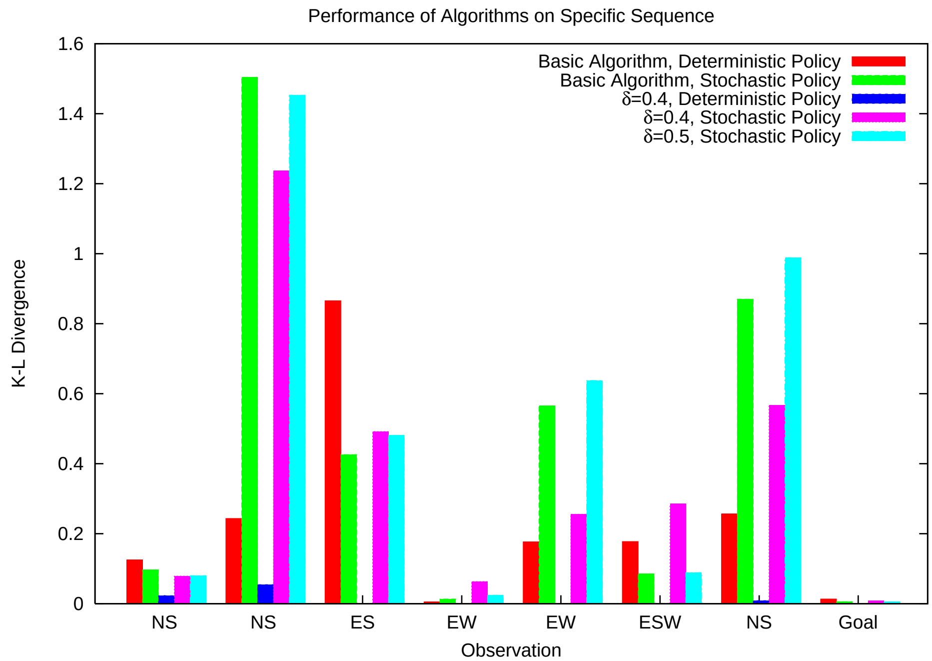

FIGURE 6.9. Performance of the algorithm on the stochastic policy Hallway task.

trained on the deterministic policy had an average of 118.8 features added (std. deviation of 6.9); when trained on the stochastic policy, the average was 140.07 (std. deviation of 18.8). For $\delta = 0.5$, these numbers were 42.0 (4.6) and 25.3 (9.0), respectively. One thing to note is that the basic algorithm does not always perform better for a given observation when the policy is deterministic: this can be explained due to the change in the probability of observing a symbol (such as the 'ES' symbol, which now occurs less often). When $\delta = 0.4$, feature construction always helps on this sequence, except for the 'ES' and 'ESW' symbols which are not predicted any better. For $\delta = 0.5$, the results are more troubling: the feature construction algorithm never performs better than the basic algorithm! One thing to keep in mind here is that the size of the subset of points on which the partial model is trained is roughly 650. Features may hinder the algorithm more than help by adding additional dimensions which do not provide sufficient information to separate the points correctly. This is an issue that should not arise if the partial model was perfectly trained. As discussed in Section 6.3.4, this is not sometimes which is achievable under the current scheme.

6.4. Discussion

Feature construction in this context is clearly beneficial to the performance of the algorithm. Both the Four Points task and the Hallway task yield much lower KL divergence when features are added. This comes with added complexity, which is reflected in the variability in the results when the δ and l parameters are varied. This is a known issue in feature construction algorithms [Adams and Waugh, 1995], which must cope with both learning a task and its structure simultaneously. The basic algorithm, akin to feedforward neural networks, is forced into a given structure, and therefore is less sensitive to changes in parameters. One way to handle this is to idevise stronger divergence measures, but this often yields a more complex algorithm. This topic will be discussed in greater detail in Chapter 7.

In order of importance, δ affects the performance more than l, but changing l gives drastically different number of features added. Of interest is the fact that in both cases, the number of features added smoothly decreases as the parameter values increase. Unfortunately, the choice of parameters yielding the smallest D_{KL} depends on the task.

Another issue which should be studied is that in all cases presented above, the modified algorithm yielded much better performance than the basic algorithm. In the Four Points task, this is because the said task was crafted to demonstrate the principle. In the Hallway task, however, it might be that the algorithm is simply overfitting the data presented to it. It is not clear whether it is desirable to do so. On one hand, learning to handle especially complicated sequences is good, because it ensures that the algorithm can cope with any task. On the other hand, if such sequences occur very rarely - as is the case here -, then adding features for them might cause overfitting. Worse, rarely occuring sequences which are badly predicted are unlikely to be accurately learned by the partial model in a reasonable amount of time, and so features representing them may be repeatedly added. This can be the case for a stochastic task such as the one which was presented above, where performance greatly suffers.

CHAPTER 7

Conclusions and Future Work

7.1. Contributions

In this thesis I have explored using a context vector in order to summarize history. I have discussed how to use this vector to predict future observations through a simple linear scheme combined with a Boltzmann distribution. I have presented the algorithm in such a way that it can be used on continuous state spaces, although my work has been restricted to symbolic (discrete) state spaces. Experimental results with tasks involving simple probabilities show that such a framework can indeed learn, in an on-line fashion, a wide range of probability distributions. The second part of my thesis builds on the first to develop a feature construction algorithm which can help the prediction process by discovering hard-to-learn points. I have briefly shown that some points cannot in fact be learned at all using the basic model. I have given empirical results showing that the additional features created by the algorithm allow it to predict nearly perfectly future observations.

7.2. Discussion

Many issues, however, remain unsolved. The drawback of the algorithm is the learning speed. It requires a large amount of samples, even for very simple tasks, where a simple Markov Model could learn with only a few episodes. There are two clearly identifiable causes of this. First, the Boltzmann distribution is by its very nature a global one (and in fact a Bayesian approach to statistics). The initial distribution is uniform, which is desirable; however, the probability of a certain observation is never zero, regardless of the weight matrix. This leads to slower learning, because the algorithm must go against a prior distribution to learn the model. Markov Models and Probabilistic Suffix Trees do not suffer from this because they take a frequentist's view of the problem. The problem becomes dramatic for my algorithm when the number of symbols is increased, because each individual symbol must overcome the 'mass' of more symbols to be predicted. The second issue that causes slower learning is the fact that the algorithm proceeds in an on-line fashion. PSTs [Ron *et al.*, 1996; Holmes and Isbell, 2006] and Utile Suffix Memory [McCallum, 1995] assume a batch of data, which they use to construct new features and learn about the world. My proposed framework, on the other hand, uses each sample only once, never looking at it in conjunction with others. This makes my approach very similar to on-line training of neural networks, which can suffer from slower learning due to the two issues outlined above.

Although the basic algorithm is clearly outperformed by the feature construction approach, the latter introduces more parameters and instability in the learning process. This is because determining how to add features, especially when done without a batch of data to compute statistics, is not a noise-free procedure. Especially, as can be seen in the Hallway task of Section 6.3, the partial model can take a very long time to converge. In fact, I have empirically studied the KL divergence of the partial model (which is reset every so often) and that of the full model. Although not directly comparable, there is a point during the learning process where the partial model cannot reach better performance, even though it is trained on a smaller set of examples. This is in part due to the tuning of the weights which happens at a later stage during training. This suggests that the partial model might not be able to accurately discern which points to keep as part of a feature's set.

In Chapter 2, I discussed many other approaches that can learn to predict sequences. Rather than carrying out an empirical comparison of these various algorithms with respect to mine, I chose to focus on studying the behavior of the algorithm on different tasks. Because this is a relatively new algorithm, I was interested in determining its flaws and weaknesses. Some of the tasks were crafted explicitly for this purposes.

Furthermore, the proposed algorithm has the merit of being fairly straightforward to implement. Batch methods such as PSTs and Utile Suffix Memory require statistical tests and special data structures. It is also not too sensitive to the choice of parameters, as opposed to recurrent neural networks which require cross-validation and fine tuning. Finally, it requires fewer samples than what is usually reported as necessary for Predictive State Representations.

7.3. Future Work

My proposed algorithm can be expanded in many ways, as most parts do not depend on the behavior of other sections. The clearest way to improve it would be to obtain a better method for detecting whether a given set of histories yields incorrect prediction. One way of doing this would be to break down the partial model into many smaller partial models. For example, rather than training one set of weights $\hat{W}$ on a subset $\hat{P}$, one could train N sets $\hat{W}_1, \ldots, \hat{W}_N$ on a corresponding number of different (and smaller) subsets of data. The learning rate could then potentially be raised, leading to faster convergence of each restricted model. Reducing the size of the subset on which the partial model is trained must necessarily allow for the discovery of more incorrect predictions.

Another way to improve the algorithm computationally would be to summarize the histories which trigger a feature, rather than keep them in a set. In effect, the hidden units of a neural network do exactly this: they trigger when a pattern from a given set occurs in their inputs. However, since the sigmoid unit which is used is a threshold function which cannot act on a local portion of the state space, a sigmoid unit could not be used to generate the algorithm's features. On the other hand, a mixture of a few Gaussians may be sufficient.

Although a good value for the divergence threshold δ was found in the empirical results of Chapter 6, there is an inherent issue that arises with using this to determine whether to add a feature. Some histories may be linked with a high divergence, because they are especially hard to predict; one may want to have a single feature for each such history. On the other hand, histories that have relatively low error may be bunched up together. It would then be interesting to let the algorithm add features based on the sum of the errors, rather than individual errors, and only add features while the sum is below a certain value.

There is another aspect of the algorithm which I have not discussed so far, but which was originally the focus of my work. The name 'context vector' comes from the fact that the vector is meant to represent a filtered history. As such, it would be of interest to make it vary over time in a better way rather than simply keeping track of the history. In effect, this is what is done in [Elman, 1990] and other recurrent neural network approaches. Modifying the context vector through a set of weights, however, gives rise to the problem of *backpropagating through time* [Hochreiter and Schmidhuber, 1997]. In order to properly modify the weights of a recurrent sigmoid unit, one must unfold the neural network to the beginning of the sequence. This is because errors propagate through time as well as through 'space'. This is clearly a very computationally expensive approach, since for long sequences the error has to be propagated backwards many times.

REFERENCES

[Adams and Waugh, 1995] A. Adams and Sam Waugh. Function evaluation and the cascade-correlation architecture. In *Proceedings of the IEEE International Conference on Neural Networks*, pages 942–946, 1995.

[Baluja and Fahlman, 1994] Shumeet Baluja and Scott Fahlman. Reducing network depth in the cascade-correlation learning architecture. Technical report, Technical Report CMU-CS-94-209, Carnegie Mellon University, 1994.

[Bellemare and Precup, 2007] Marc G. Bellemare and Doina Precup. Context-driven predictions. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*, 2007.

[Bellemare, 2006] Marc G. Bellemare. Cascade-correlation algorithms for on-line reinforcement learning. In *Proceedings of the North East Student Colloquium on Artificial Intelligence*, 2006.

[Bose *et al.*, 2005a] Joy Bose, Steve B. Furber, and Jonathan L. Shapiro. An associative memory for the on-line recognition and prediction of temporal sequences. In *Proceedings of the International Joint Conference on Neural Networks*, 2005.

[Bose *et al.*, 2005b] Joy Bose, Steve B. Furber, and Jonathan L. Shapiro. A spiking neural sparse distributed memory implementation for learning and predicting temporal sequences. In *Proceedings of the International Conference on Artificial Neural Networks*, 2005.

[Boyen and Koller, 1998] Xavier Boyen and Daphne Koller. Tractable inference for complex stochastic processes. In *Proceedings of the Fourteenth Annual Conference on Uncertainty in Artificial Intelligence*, pages 33–42, 1998.

[Elman, 1990] Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990.

[Fahlman and Lebiere, 1990] Scott E. Fahlman and C. Lebiere. The cascade-correlation learning architecture. In *Advances in Neural Information Processing Systems 2*, volume 2, 1990.

[Fahlman *et al.*, 1983] Scott E. Fahlman, Geoffrey E. Hinton, and T.J. Sejnowski. Massively parallel architectures for ai: Netl, thistle, and boltzmann machines. In *Proceedings of the National Conference on Artificial Intelligence*, 1983.

[Furber *et al.*, 2004] S. B. Furber, W.J. Bainbridge, J.M. Cumpstey, and S. Temple. Sparse distributed memory using n-of-m codes. *Neural Networks*, 10, 2004.

[Hadsell *et al.*, 2006] R. Hadsell, S. Chopra, and Y. LeCun. Dimensionality reduction by learning an invariant mapping. In *Proceedings of the Computer Vision and Pattern Recognition Conference 2006*. IEEE Press, 2006.

[Hanson, 1990] Stephen J. Hanson. Meiosis networks. In *Advances in Neural Information Processing Systems 2*, volume 2, 1990.

[Hinton and Sejnowski, 1986] Geoffrey E. Hinton and T. J. Sejnowski. Learning and relearning in boltzmann machines. In *Parallel distributed processing: explorations in the microstructure of cognition*, pages 282–317. MIT Press, 1986.

[Hinton *et al.*, 2006] Geoffrey E. Hinton, S. Osindero, and Y. Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18:1527–1554, 2006.

[Hochreiter and Schmidhuber, 1997] Sepp Hochreiter and Jurgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[Holmes and Isbell, 2006] M. Holmes and C. Isbell. Looping suffix tree-based inference of partially observable hidden state. In *Proceedings of the Twenty-Third International Conference on Machine Learning*, 2006.

[Kaelbling *et al.*, 1998] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101:99–134, 1998.

[Kanerva, 1988] Pentti Kanerva. *Sparse Distributed Memory*. The MIT Press, 1988.

[Kanerva, 1993] Pentti Kanerva. Sparse distributed memory and related models. In M. Hassoum, editor, *Associative Neural Memories*, chapter 3. Oxford University Press, 1993.

[Kostiadis and Hu, 2001] Kostas Kostiadis and Huosheng Hu. Kabage-rl: Kanerva-based generalization and reinforcement learning for possession football. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2001.

[Littman *et al.*, 2002] Michael L. Littman, Richard S. Sutton, and Satinder Singh. Predictive representations of state. In *Advances in Neural Information Processing Systems 14*, pages 1555–1561, 2002.

[McCallum, 1995] Andrew K. McCallum. *Reinforcement Learning with Selective Perception and Hidden State*. PhD thesis, University of Rochester, 1995.

[Neal, 1992] R.M. Neal. Connectionist learning of belief networks. *Artificial Intelligence*, 56(1):71–113, 1992.

[Rabiner, 1989] Lawrence R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–287, 1989.

[Randlov and Alstrom, 1998] J. Randlov and P Alstrom. Learning to drive a bicycle using reinforcement learning and shaping. In *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 463–471, 1998.

[Rao and Fuentes, 1996] Rajesh P.N. Rao and Olac Fuentes. Learning navigational behaviors using a predictive sparse distributed memory. *IEEE Transitions on Neural Networks*, 7(4):926–941, 1996.

[Rivest and Precup, 2003] F. Rivest and D. Precup. Combining td-learning with cascade-correlation networks. In *Proceedings of the Twentieth International Conference on Machine Learning*, 2003.

[Ron *et al.*, 1996] D. Ron, Y. Singer, and N. Tishby. The power of amnesia: Learning probabilistic automata with variable memory length. *Machine Learning*, 25:117–149, 1996.

[Rujan and Marchand, 1989] Pal Rujan and Mario Marchand. A geometric approach to learning in neural networks. In *Proceedings of the International Joint Conference on Neural Networks*, 1989.

[Rumelhart *et al.*, 1986] D.E. Rumelhart, Geoffrey E. Hinton, and R.J. Williams. Learning internal representations by error propagation. In *Parallel distributed processing: explorations in the microstructure of cognition*, pages 282–317. MIT Press, 1986.

[Shannon, 1951] Claude E. Shannon. Prediction and entropy of printed english. *Bell System Technical Journal*, 30:50–64, 1951.

[Singh and Sutton, 1996] Satinder P. Singh and Richard S. Sutton. Reinforcement learning with eligibility traces. *Machine Learning*, 22:123–158, 1996.

[Sutton, 1988] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44, 1988.

[Tanner *et al.*, 2007] B. Tanner, V. Bulitko, A. Koop, and C. Paduraru. Grounding abstraction in predictive state representations. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*, 2007.

[Taylor *et al.*, 2007] G. Taylor, G. Hinton, and S. Roweis. Modelling human motion using binary latent variables. In *Advances in Neural Information Processing Systems 19*, 2007.

[Tesauro, 1995] Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3), 1995.

[Theocharous *et al.*, 2004] Georgios Theocharous, Kevin Murphy, and Leslie Kaelbling. Representing hierarchical pomdps as dbns for multi-scale robot localization. In *Proceedings of the International Conference on Robotics and Automation*, 2004.

[Utgoff and Precup, 1998] Paul E. Utgoff and Doina Precup. Constructive fuunction approximation. In *Feature extraction, construction, and selection: A data-mining perspective*, pages 219–235. Kluwer, 1998.

Document Log:

Manuscript Version 0

Typeset by $\mathcal{A}\mathcal{M}\mathcal{S}$-LaTeX — 13 June 2007

MARC GENDRON-BELLEMARE

MCGILL UNIVERSITY, 3480 UNIVERSITY ST., MONTRÉAL (QUÉBEC) H3A 2A7, CANADA, *Tel.* : (514) 398-7071

E-mail address: `marcgb@cs.mcgill.ca`

Typeset by $\mathcal{A}\mathcal{M}\mathcal{S}$-LaTeX