13 MACHINE LEARNING: PROBABILISTIC

Probability theory is nothing but common sense reduced to calculation....

—LAPLACE

The purpose of computing is insight, not numbers...

—RICHARD HAMMING

13.0 Stochastic and Dynamic Models of Learning

As noted in earlier chapters, there are two primary reasons for using probabilistic tools to understand and predict phenomena in the world: first, events may be genuinely probabilistically related to each other, and second, deterministic causal relationships between situations in the changing world may be so complex that their interactions are best captured by stochastic models. The AI community has adopted and deployed probabilistic models for both of these reasons, and these stochastic technologies have had a very important influence on the design, power, and flexibility of machine learning algorithms.

Bayes' rule, first presented in Section 5.3, is the basis for probabilistic models of machine learning. Bayesian approaches support interpretation of new experiences based on prior learned relationships: understanding present events is a function of learned expectations from prior events. Similarly, Markov models and their variants offer a basis for stochastic approaches to learning. In a Markov chain the probability of an event at any point in time is a function of the probability with which events occurred at previous time periods. In a *first-order Markov chain*, introduced in Section 9.3, the probability of an event at time `t` is a function only of the probability of its predecessor at time `t - 1`.

Chapter 13 has three sections, first Section 13.1 continues the presentation of Markov models introducing *Hidden Markov Models* (or HMMs) and several important variants and extensions. Section 13.2 extends the presentation of Bayesian networks began in Chapter 9.3, focusing especially on *dynamic Bayesian networks* (or DBNs) and several

extensions. Section 13.3 continues the presentation of reinforcement learning started in Section 10.7, with the addition of probabilistic measures for reinforcement.

13.1 Hidden Markov Models (HMMs)

In the early twentieth century the Russian mathematician Andrey Markov proposed a mathematics for modeling probabilistically related discrete stepped events. As compared with a deterministic sequence of events, as one might expect when a computer executes a fixed set of instructions, Markov's formalisms supported the idea that each event can be probabilistically related to the events that preceded it, as in patterns of phonemes or words in a spoken sentence. In our attempts to program computers to learn phenomena in a changing world, Markov's insights have proved exceedingly important.

13.1.1 Introduction to and Definition of Hidden Markov Models

A *hidden Markov model* (or *HMM*) is a generalization of the traditional *Markov chain* (or *process*) introduced in Section 9.3. In the Markov chains seen to this point, each state corresponded to a discrete physical - and observable - event, such as the weather at a certain time of day. This class of models is fairly limited and we now generalize it to a wider class of problems. In this section we extend the Markov model to situations where observations are themselves probabilistic functions of current hidden states. Thus, the resulting model, called a hidden Markov model, is a doubly embedded stochastic process.

An HMM may be described as an observable stochastic process supported by a further non-observable, or hidden, stochastic process. An example use for an HMM would be to support computer based word recognition through the interpretation of noisy acoustic signals. The phone patterns themselves, that is, the phonemes that the speaker intends to use as a part of producing particular words of a language, make up the hidden level of the Markov model. The observations, that is, the noisy acoustic signals that are heard, are a stochastic function of these intended phonemes. The phonemes that the speaker intended can be seen only probabilistically through the top level (observable) stream of the noisy acoustic signals. We define an HMM:

DEFINITION

HIDDEN MARKOV MODEL

A graphical model is called a *hidden Markov model (HMM)* if it is a Markov model whose states are not directly observable but are hidden by a further stochastic system interpreting their output. More formally, given a set of states $S = s_1, s_2, ..., s_n$, and given a set of state transition probabilities $A = a_{11}, a_{12}, ..., a_{1n}, a_{21}, a_{22}, ..., ..., a_{nn,}$ there is a set of observation likelihoods, $O = p_i(o_t)$, each expressing the probability of an observation o_t (at time t) being generated by a state s_t.

We now give two examples of HMMs, adapted from Rabiner (1989). First, consider the situation of coin flipping. Suppose there is a person in a room flipping coins one at a time. We have no idea what is going on in the room, there may in fact be multiple coins, randomly selected for flipping, and each of the coins may have its own biased outcomes, i.e., they may not be fair coins. All we observe outside the room is a series of outcomes from flipping, e.g., observable O = H, H, T, H, T, H, ... Our task is to design a model of coins flipped inside the room that produces the resulting observable string.

Given this situation, we now attempt to model the resulting set of observations emanating from the room. We begin with a simple model; suppose there is but one coin being flipped. In this case we only have to determine the bias of the coin as it is flipped over time producing the set of head/tail observations. This approach results in a directly observable (zero-order) Markov model, which is just a set of Bernoulli trials. This model may, in fact, be too simple to capture reliably what is going on in the coin flipping problem.

We next consider a two coin model, with two hidden states s_1 and s_2, as seen in Figure 13.1. A probabilistic transition matrix, A, controls which state the system is in, i.e., which coin is flipped, at any time. Each coin has a different H/T bias, b_1 and b_2. Suppose we observed the string of coin flips: O = H, T, T, H, T, H, H, H, T, T, H. This set of observations could be accounted for by the following path through the state transition diagram of Figure 13.1: s_2, s_1, s_1, s_2, s_2, s_2, s_1, s_2, s_2, s_1, s_2.

A third model is presented in Figure 13.2, where three states (three coins) are used to capture the coin flip output. Again, each coin/state, s_i, has its own bias, b_i, and the state of the system (which coin is flipped) is a function of some probabilistic event such as the roll of a die and the transition matrix, A. The three state machine could account for the H/T output with the state sequence: s_3, s_1, s_2, s_3, s_3, s_1, s_1, s_2, s_3, s_1, s_3.

The difficult issue for the coin flipping problem is the choice of an optimal model. The simplest model has one parameter, the bias of a single coin. The two state model has four parameters, the state transitions and the biases of each coin. The three state model of Figure 13.2 has nine parameters. With more degrees of freedom, the larger model is more powerful for capturing a (possibly) more complex situation. However, the complexity issues of determining the parameters for the larger model can doom the exercise. For example, the actual observables might be produced by a simpler situation, in which case the larger model would be incorrect, under specified, and would require much more data

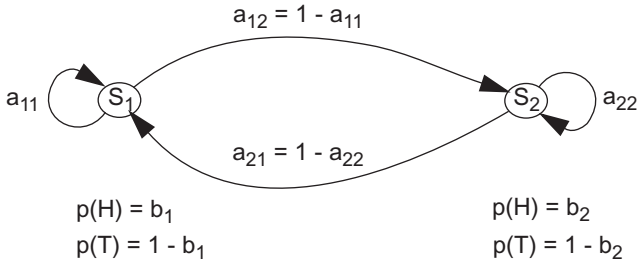

Figure 13.1 A state transition diagram of a hidden Markov model of two states designed for the coin flipping problem. The a_{ij} are determined by the elements of the 2 x 2 transition matrix, A.

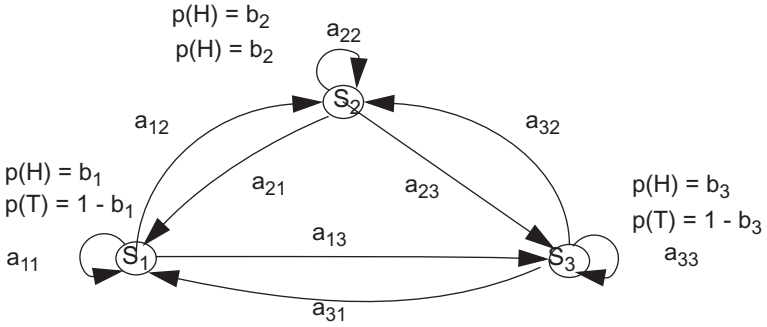

Figure 13.2 The state transition diagram for a three state hidden Markov model of coin flipping. Each coin/state, S_i, has its bias, b_i.

to establish any degree of confidence. Similar problems, such as *overfitting*, also haunt smaller models. Finally, in each model there is an implicit assumption of *stationarity*, that is, that the transition and bias probabilities of the system do not change across time.

For a second example, consider the problem of N urns, each urn containing a collection of M differently colored balls. The physical process of obtaining observations is, according to some random process, to pick one of the N urns. Once an urn is selected a ball is removed and its color is recorded in the output stream. The ball is then replaced and the random process associated with the current urn selects the next (which might be the same) urn to continue the process. This process generates an observation sequence consisting of colors of the balls. It is obvious that the simplest HMM corresponding to this process is the model in which each state corresponds to a specific urn, the values of the transition matrix for that state produce the next state choice, and finally, a model in which the ball color probability is defined by the number and color of balls for each state (urn). However, unless the modeler has some prior information, it would be a very difficult task indeed to determine all the parameters required to select an optimal HMM.

In the next section we consider several variant forms of HMMs.

13.1.2 Important Variants of Hidden Markov Models

In the previous section, we observed that HMMs are extremely powerful yet simple models that can represent uncertain domains. The computational simplicity of the HMMs seen so far follows from the Markovian principle of a first-order system: that the present state depends only on the previous state. In this section, we explore how we can modify the basic principles of an HMM to achieve even richer representational capabilities. The regular HMMs of Section 13.1.1 have limitations in specific domains and we next demonstrate HMM variants that can address many of these limitations.

A widely used extension to HMMs is called the *auto-regressive HMM* where a previous observation can also influence the value of the current observation. This extension is used to capture more information from the past and factor it into the interpretation of the present state. In order to model complex processes made up of combinations of sets of simpler subprocesses *factorial HMMs* are often used. One extension of HMMs, *hierarchi-*

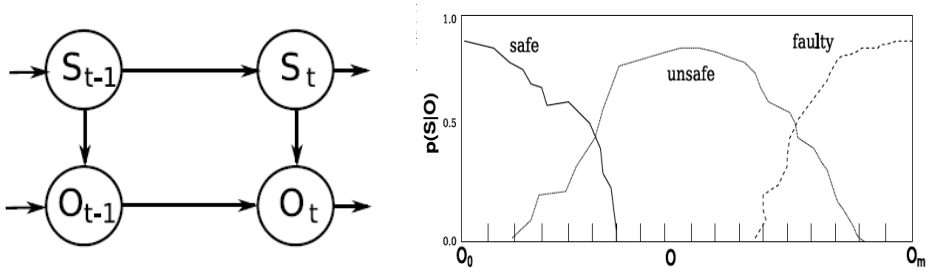

Figure 13.3. a. The hidden, S, and observable, O, states of the AR-HMM where $p(O_t \mid S_t , O_{t-1})$.
b. The values of the hidden S_t of the example AR-HMM: safe, unsafe, and faulty.

cal HMMs or *(HHMS)*, assumes that each state of a system is itself an HMM. We next consider these and other extensions of HMMs in more detail.

Auto-regressive HMMs (AR-HMMs)

In hidden Markov models the first-order property hypothesizes that the present state is only dependent on the immediate previous state and that the observed variables are not corelated. This can be a limitation, in which the present state does not adequately capture the information available in previous states. In diagnostic reasoning, for example, this can lead to inferior generalizations, especially when modeling time-series data. The fact is that in complex environments the observable values at one time period often do correlate with subsequent observations. This must be factored into the structure of the HMM.

We overcome this aspect of limited generalization by allowing the previous observation to influence the interpretation of the current observation along with the usual HMM influences of the previous state. This HMM model is called the *auto-regressive hidden Markov model* whose emission model is defined by the non-zero probability model:

$$p(O_t \mid S_t , O_{t-1}).$$

where O is the emit or observation state and S represents the hidden layer, as seen in Figure 13.3a. The difference between the traditional HMM and the AR-HMM is the existence of a link (influence) between O_{t-1} and O_t. The motivation for this influence is straightforward: the observation state at one time period will be an indicator of what the next observation will likely be. In other words, the observed output values of a time-series analysis will change according to some underlying distribution.

The AR-HMM often leads to models with a higher likelihood of convergence than regular HMMs, especially while modeling extremely complex and noisy time-series data. We demonstrate the AR-HMM with an example adapted from the work of Chakrabarti et al. (2007). The task is to monitor the running health of a helicopter rotor system. Figure 13.4 represents a sample of the data that is observed from the running system. The data is

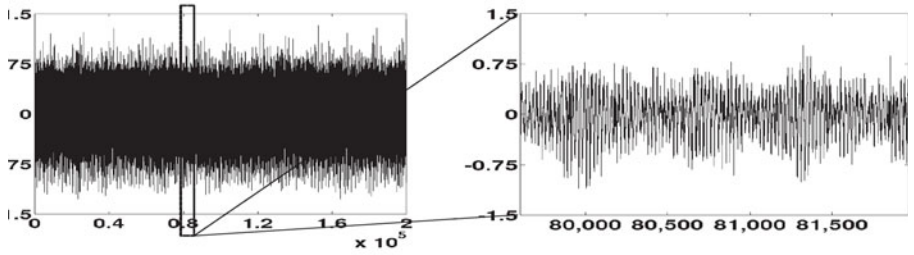

Figure 13.4 A selection of the real-time data across multiple time periods, with one time slice expanded, for the AR-HMM.

the output of a number of sensors, including information from heat and vibration measures on multiple components. Figure 13.5 shows the results of processing the data of Figure 13.4 using a fast Fourier transform (FFT), to produce equivalent data on the frequency domain. These data points are then used to train an AR-HMM. The three states of the hidden node, S_t, in Figure 13.3, are safe, unsafe, and faulty.

The goal of this project was to investigate techniques for fault detection and diagnosis in mechanical systems. Using a set of time series data from sensors monitoring mechanical processes, the task was to build a quantitative model of the entire process and to train it on data sets (seeded faults such as a cracked shaft) for later use in the real-time diagnosis and prediction of future faults. After the AR-HMM was trained it was released to work in real-time. Although the reported accuracy (Chakrabarti et al. 2006) was just over 75%, it is important to see how such prognostic tools can be designed.

Factorial HMMs

In our previous discussions on HMMs and their variants, we described systems whose state space was partially hidden. We could only get clues to the state space through a series of observations. We tried to infer the sequence of underlying hidden states using information from the observed sequence of data emitted by the system.

But what happens if the underlying process is very complex and cannot be efficiently

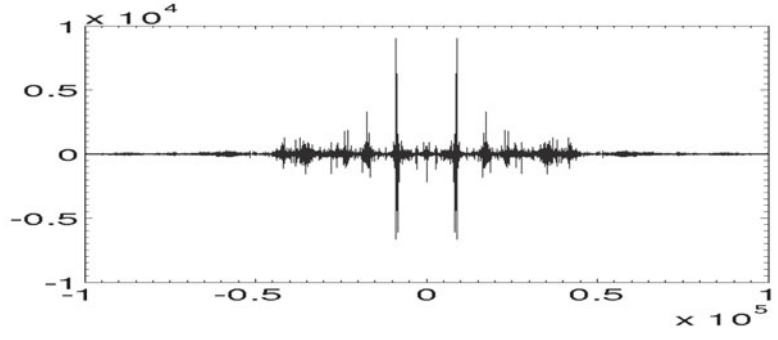

Figure 13.5 The time-series data of Figure 13.4 processed by a fast Fourier transform on the frequency domain. This was the data submitted to the AR-HMM for each time period.

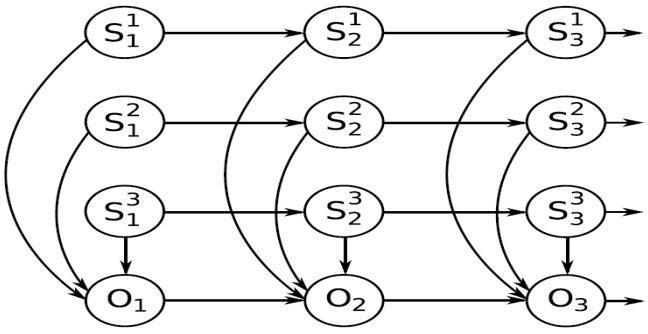

Figure 13.6 An auto regressive factorial HMM, where the observable state O_t, at time t is dependent on multiple (S_t) subprocess, S^i_t, and O_{t-1}.

represented as a simple set of states? What happens if the underlying process is actually a combination of several sub-processes? Our regular HMM cannot sufficiently capture the information about processes in these situations. We need a richer model to represent more complex situations.

Factorial HMMs are a possible solution. Figure 13.6 shows an auto-regressive factorial HMM. We see that the underlying process can be split into n sub-processes, where each of these n processes influences the current observation, O_t. Also, each sub-process follows the first-order Markov property, that is, its current state depends only on its previous state(s). This relation to the observable random variable can be defined by the CPD:

$$p(O_t \mid O_{t-1}, S^1_t, S^2_t, \ldots, S^i_t)$$

Hierarchical HMMs (HHMMs)

Hierarchical HMMs are used to model extremely complex systems. In the *hierarchical hidden Markov model* (or *HHMM*) each individual state is considered a self contained probabilistic model. More precisely, each state of the HHMM can itself be an HHMM. This implies that states of the HHMM emit sequences of observations rather than just single observations as is the case for the standard HMMs and the variants seen to this point.

When a state in an HHMM is reached, it will activate its own probabilistic model, i.e., it will activate one of the states of the underlying HHMM, which in turn may activate its underlying HHMM, and so on, the base case being the traditional HMM. The process is repeated until a state, called a *production state*, is activated. Only production states emit observation symbols in the sense of the usual hidden Markov model. When the production state has emitted a symbol, control returns to the state that activated the production state. The states that do not directly emit observation symbols are called *internal states*. The activation of a state in an HHMM under an internal state is called a *vertical* transition. After a vertical transition is completed a *horizontal* transition occurs to a state at the same level. When a horizontal transition leads to a terminating state, control is returned to the

state in the HHMM higher up in the hierarchy, that produced the last vertical transition.

It should be noted that a vertical transition can result in more vertical transitions before reaching a sequence of production states and finally returning to the top level. Thus the production states visited give rise to a sequence of observation symbols that are produced by the state at the top level.

N-Gram HMMs

As noted earlier, in traditional first-order Markov models the current state, given the immediately previous state, is independent of all earlier states. Although this rule reduces the computational complexity for using Markov processes, it may be the case that the previous state alone cannot completely capture important information available in the problem domain. We explicitly address this situation with the *n-gram hidden Markov model* (or *N-gram HMM*). These models, especially in their *bi-gram* and *tri-gram* forms, are especially important in stochastic approaches to natural language understanding, presented in more detail in Section 15.4.

With bi-gram Markov models the interpretation of the observable current state depends only on interpretation of the previous state, as $p(O_t \mid O_{t-1})$; this is equivalent to the traditional first-order Markov chain. Even though the first-order Markov assumption is obvious, it is interesting to think about what bi-gram processing means, for example, in natural language understanding. It hypothesizes that, for word or phoneme recognition, given a sequence of previous words, the probability of a particular word occurring is best estimated when we use only the interpretation of the immediately previous word. The bi-gram is thus hypothesized to give better results than knowledge of no previous word information or knowledge of a larger sequence of previous words. For example, given a collection of language data, p(lamb | little) is a better predictor of the current observed word being lamb, than either p(lamb | a little), p(lamb | had a little), or p(lamb | Mary had a little).

Similarly, tri-gram models are Markov models in which the interpretation of the observable current state depends on the previous two states. The tri-gram model is represented by $P(O_t \mid O_{t-1}, O_{t-2})$. Continuing the natural language example, the use of tri-gram models imply that p(lamb | a little) is a better predictor of the current word being lamb than p(lamb | little), or of more previous words, including p(lamb | Mary had a little).

We can extend the n-gram Markov model to any length n that we feel captures the invariances present in the data we are trying to model. An n-gram Markov model is a Markov chain in which the probability of the current state depends on exactly the n-1 previous states.

As we see in more detail in Chapter 15, n-gram models are especially useful for capturing sequences of phones, syllables, or words in speech understanding or other language recognition tasks. We can set the value of n depending on the desired complexity of the domain we want to represent. If n is too small, we may have poorer representational power and miss the ability to capture the predictive power of longer strings. On the other hand, if the value of n is too large, then the computational cost of validating the model can be very high or even impossible. The main reason for this is that the prior expectations for phone or word combinations are taken from large data bases (often combined data bases)

of collected language phenomena called a *corpus*. We simply may not have enough data to validate an n-gram with a very large n. For example, we may have no information at all on p(lamb | Mary had a little) while we have lots of information of p(lamb | little), p(lamb | a little), or p(lamb | the). Typically, in speech or text recognition tasks, we use n-grams with the value of n no higher than 3.

There are a number of other variants of HMMs that we have not described here, including *mixed-memory HMMs* that use several low order Markov models to overcome the complexity issues. A further extension of auto-regressive HMMs, called *buried Markov models*, allows for nonlinear dependencies between the observable nodes. *Input-output HMMs* are used for mapping sequences of inputs to a sequence of outputs. In situations where interactions between subprocesses of a complex process are more complicated than just a composition, a *coupled HMM* is often used. A *hidden semi-Markov model* generalizes an HMM to avoid the geometric decay effect of HMMs by making the probability of transitions between hidden states depend on the amount of time that has passed since the last state transition. We refer to Section 13.4 for references.

In the next section, we apply the bi-gram HMM technology to the problem of interpreting English phoneme combinations, and use dynamic programming with the Viterbi algorithm to implement an HMM search.

13.1.3 Using HMMs and Viterbi to Decode Strings of Phonemes

Our final section on hidden Markov models demonstrates an important use of the HMM technology: identifying patterns in spoken natural language. We will assume the presence of a large language corpus that supports prior expectations on phone combinations. Finally, we use dynamic programming algorithms, introduced in Section 4.1, for implementing HMM inference. When dynamic programming is used to find the maximum probability of an a posteriori string, it is often called the *Viterbi algorithm*.

The following example of computational analysis of human speech patterns and the use of the Viterbi algorithm to interpret strings of phonemes is adapted from Jurafsky and Martin (2008). The input into this probabilistic machine is a string of phones, or basic speech sounds. These are derived from the decomposition of the acoustic signal produced through spoken language use. It is unusual in automated speech understanding that acoustic signals would be unambiguously captured as a string of phonemes. Rather speech signals are interpreted as specific phones probabilistically. We assume unambiguous interpretation of signals to simplify our presentation of the Viterbi algorithm processing of the HMM.

Figure 13.7 presents a segment of a database of words, related by the closeness of the sets of phonemes that make up their acoustic components. Although this set of words, neat, new, need, and knee, is but a small piece of the full English vocabulary, it can be imagined that a very large number of these related clusters could support a speech understanding system. Figure 13.7 is an example of a probabilistic finite state machine, as first introduced in Section 5.3. We assume our language corpus to be a collection of similar probabilistic finite state machines that can give prior probability measures of different

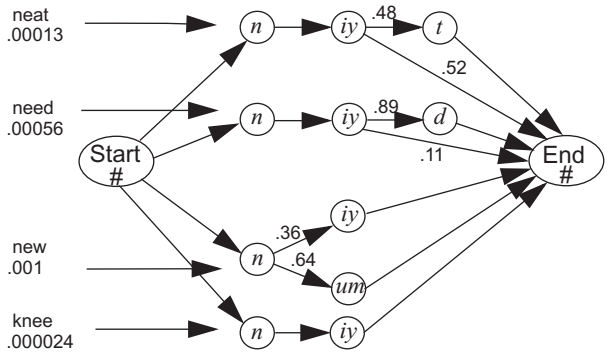

Figure 13.7 A PFSM representing a set of phonetically related English words. The probability of each word occurring is below that word. Adapted from Jurasky and Martin (2008).

possible phone combinations, and how finally, these phones can be seen as parts of words.

The goal of the analysis is to determine which English word from our database of words best represents the input of an acoustic signal. This requires use of the HMM technology, because the representation of possible words is itself stochastic. In the non-deterministic finite state machine of Figure 13.7, the string of phones gives us the set of observations to interpret. Suppose the string of observations is made up of the phones #, n, iy, #; where # indicates a pause of break between sounds. We use the Viterbi algorithm to see which path through the probabilistic finite state machine best captures these observations. In the forward mode Viterbi iteratively finds the next best state and its value, and then sets a pointer to it. In the backward mode we retrace these pointers to obtain the best path. Thus, the output of Viterbi is one of the optimal paths of states through the graph associated with the probability of that path.

Using Viterbi, each state of the search is associated with a value. The value for being in state s_i at time t is viterbi[s_i, t]. The value associated with the next state s_j in the state machine at time t + 1 is viterbi[s_j, t + 1]. The value for the next state is computed as the product of the score of the present state, viterbi[s_i, t], times the transition probability of going from the present state to the next state, path[s_i, s_j], times the probability of observation s_j given s_i, p(s_j | s_i). The transition probability, path[s_i, s_j], is taken from the non-deterministic finite state machine and p(s_j | s_i) is taken from known observation likelihoods of pairs of phones occurring in the English language.

We next present pseudocode for the Viterbi algorithm. Note its similarity with dynamic programming, Section 4.1. The array for storing and coordinating values must support iteration over R rows - equal to the number of phones in the probabilistic finite state machine (PFSM) plus two, to handle each state plus the start and end states. It must also iterate over C columns - equal to the number of observations plus two, to handle use of the empty phone #. The columns also indicate the time sequence for observations and each state must be linked with the appropriate observed phone, as seen in Figure 13.8.

Start = 1.0	#	*n*	*iy*	#	end
neat .00013 2 paths	1.0	1.0 x .00013 = .00013	.00013 x 1.0 = .00013	.00013 x .52 = .000067 (2 paths)	
need .00056 2 paths	1.0	1.0 x .00056 = .00056	.00056 x 1.0 = .00056	.00056 x .11 = .000062 (2 paths)	
new .001 2 paths	1.0	1.0 x .001 = .001	.001 x .36 = .00036 (2 paths)	.00036 x 1.0 = .00036	
knee .000024 1 path	1.0	1.0 x .000024 = .000024	.000024 x 1.0 = .000024	.000024 x 1.0 = .000024	
Total best					.00036

Figure 13.8 A trace of the Viterbi algorithm on several of the paths of Figure 13.7. Rows report the maximum value for Viterbi on each word for each input value (top row). Adapted from Jurafsky and Martin (2008).

```
function Viterbi(Observations of length T, Probabilistic FSM)
begin
    number := number of states in FSM
    create probability matrix viterbi[R = N + 2, C = T + 2];
    viterbi[0, 0] := 1.0;
    for each time step (observation) t from 0 to T do
        for each state s_i from i = 0 to number do
            for each transition from s_i to s^j in the Probabilistic FSM do
                begin
                    new-count := viterbi[s_i, t] x path[s_i, s^j] x p(s^j | s_i);
                    if ((viterbi[s^j, t + 1] = 0) or (new-count > viterbi[s^j, t + 1]))
                        then
                            begin
                                viterbi[s^i, t + 1] := new-count
                                append back-pointer [s^j , t + 1] to back-pointer list
                            end;
                end;
    return viterbi[R, C];
    return back-pointer list
end.
```

Figure 13.8, adapted from Jurafsky and Martin (2008) presents a trace of the Viterbi algorithm processing a component of the probabilistic finite state machine of Figure 13.7

and the phone sequence #, n, iy, # (the observations of length T = 4). The back trace links indicate the optimal path through the probabilistic finite state machine. This path indicates that the most likely interpretation of the string of observed phones is the word new.

In the next section we present another graphical model, a generalization of Markov models, called the dynamic Bayesian network (or DBN).

13.2 Dynamic Bayesian Networks and Learning

We often encounter problems in modeling where the underlying process to be characterized is best described as a sequence or progression of states. Data collected from sequential processes are typically indexed by a parameter such as the time stamp of data acquisition or a position in a string of text. Such sequential dependencies can be both deterministic and stochastic. Dynamic processes that are inherently stochastic demand models that are capable of modeling their non-determinism.

Although most deterministic sequences can be characterized well by finite state machines (FSMs, Section 3.1), some important sequences can be arbitrarily long. Therefore, modeling each instance of these sequences as sets of individual states becomes problematic as the numbers of state transitions grow exponentially with respect to the number of states at any given instant. Probabilistic modeling of such sequences, therefore, enables us to keep models tractable, while still capturing sequence information relevant to the problem at hand.

In Section 13.2 we describe two different approaches to learning, structure learning and parameter learning, in the context of graphical models. We briefly describe techniques for structure learning including the use of *Markov chain Monte Carlo* (or *MCMC*) sampling. We also detail a popular meta-algorithm called *Expectation Maximization* (or *EM*) for parameter learning. We show how a recursive dynamic programming algorithm called the *Baum-Welch algorithm* adapts the EM meta-algorithm for use with both DBMs and HMMs.

In Section 13.3 we explain how Markov models can be used for decision support by introducing the *Markov decision process* (or *MDP*) and the *partially observed MDP* (or *POMDP*). We then take reinforcement learning (as originally introduced in Section 10.7) and supplement it with probabilistic state and feedback models. Finally, we end Chapter 13 using reinforcement learning using an MDP to design a robot navigation system.

13.2.1 Dynamic Bayesian Networks

In Section 9.3 we introduced the Bayesian belief network, a very popular and successfully applied formalism for probabilistic modeling. However, for modeling dynamic or sequential processes BBNs are quite inflexible. They do not have the provision to model time series data including the causal processes or sequential data correlations that are often encountered in complex tasks, including natural language understanding and real-time diagnostics. The problem is that the conditional dependencies of a BBN are all assumed to be active at the same instant.

In Section 9.3 we introduced the *dynamic Bayesian network* (or *DBN*), although we deferred detailed discussion to this chapter. Briefly, a dynamic, sometimes called *temporal*, Bayesian network is a sequence of identical Bayesian networks whose nodes are linked in the (directed) dimension of time. dynamic Bayesian networks are a sequence of directed graphical models of stochastic processes. They are generalizations of popular stochastic tools like hidden Markov models (HMMs) and linear dynamical systems (LDSs) that represent hidden (and observed) states as state variables, which can have complex interdependencies across periods of time. The graphical structure provides a way to specify conditional dependencies, and hence offers a compact parameterization for the model.

The individual BNs that make up the DBNs are not dynamic in the sense that they themselves do not change with time, the *stationarity* assumption, but they still support the modeling of dynamic processes. Thus DBNs work under the assumption that their defining parameters are time-invariant. However, as we will see, hidden nodes can be added to represent current operating conditions, thereby creating mixtures of models to capture periodic non-invariances in the time dimension.

At each time period (slice), a DBN is akin to a static BBN that has a set Q_t, of N_h random variables representing the hidden states and a set O_t of N_o random variables representing the states that are observable. Each random variable can either be discrete or continuous. We have to define a model that represents the conditional probability distributions between states in a single time slice and a transition model that represents the relationship between the BBNs for consecutive time slices. Therefore, if $Z_t = \{Q_t \cup O_t\}$ is the union of both sets of variables, the transition and observation models can be defined as the product of the conditional probability distributions in a two time period BBN with no cycles, call it B_t. The notation $(1{:}N)$ means for all variables 1 through N:

$$p(Z_t\,(1{:}N) \mid Z_{t-1}(1{:}N)) = \prod_{i=1}^{N} p(Z_t(i) \mid Pa(Z_t(i))),$$

where $Z_t(i)$ is the ith node in time slice t, $Pa(Z_t(i))$ are the parents of $Z_t(i)$ and $N = N_h + N_o$. Note that here, for simplicity, we have restricted the definition to first-order Markov processes where variables at time t are influenced only by their predecessor at time $t - 1$. We can represent the unconditional initial state distribution $p(Z_1(1{:}N))$, as an unconditional Bayes net B_1. B_1 and B_t, for all t, define the DBN. We show two time slices of a simple DBN in Figure 13.9.

As should be obvious, the dynamic Bayesian network is a generalization of the hidden Markov models presented in Section 13.1. In Figure 13.9 the Bayesian networks are linked between times t and $t+1$. At least one node of the two networks must be linked across time.

13.2.2 Learning Bayesian Networks

Learning is central to intelligent action. Intelligent agents often record events observed in the past in the expectation that this information might serve some utility in the future. Over its lifetime an intelligent agent encounters many events, perhaps many more than it

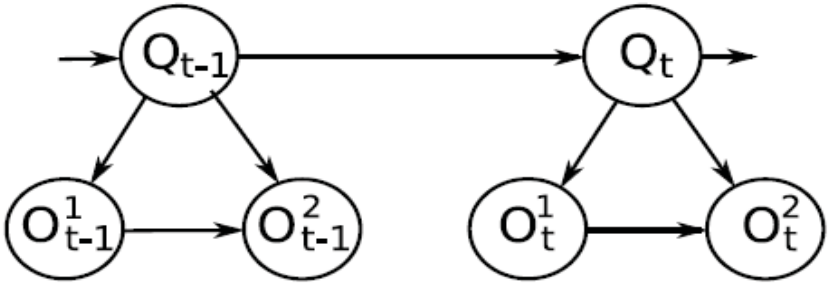

Figure 13.9 A DBN example of two time slices. The set Q of random variables are hidden, the set O observed, t indicates time.

has resources for storing. Therefore, it becomes necessary to encode event information with a concise abstraction rather than with a full documentation of every observation. Moreover, compaction of information assists quicker retrieval. Learning, therefore, can be viewed as a combination of data compression, indexing, and storage for efficient recovery.

Any new observation by an agent may be slightly different from previous observations, but may still belong to a general class of prior events. In that case it is desirable to learn this event as an instance of the prior class, ignoring the less important details of the specific event. A new event might also be very different from all prior events, and then the agent will need to learn it as a new class or as a specific classless anomaly. An agent needs to create a sufficient number of classes that fully characterize the parameter space, as well as be able to successfully handle new anomalous events. Thus, good learning is a balance of both generality and specificity.

In the context of probabilistic graphical models, we encounter two general learning problems that fit our previous discussion, *structure learning* where entire new relationships (graphical models) are learned and *parameter learning* where components of an existing model are recalibrated.

Structure Learning and Search

Structure learning addresses the general problem of determining the existence of statistical dependencies among variables. For example, the answers to the following questions would help determine the structure of a probabilistic graphical model of the weather. Are dark clouds related to the chances of rain? Is there a correlation between the position of Saturn and the number of hurricanes on any particular day? Is global warming related to the presence of typhoons? Is weather in the Arctic related to changing boundaries of the Sahara desert? As is evident, in a probabilistic model, a positive answer to such questions would warrant a conditional dependency between the variables in question and a negative answer would permit us to treat these variables as independent.

In designing a probabilistic graphical model, every variable could be dependent on every other variable, while preserving the constraint that the resulting network is a directed acyclic graph. In the worst case then, we end up with a fully connected DAG as

the model. However, this is very uncommon in most practical applications of any size, because the creation of the conditional probability tables becomes intractable. An important insight supporting the design of graphical models is that we can very often determine that several variables exhibit independencies with respect to each other in the context of the problem being modeled. Since the complexity of inference in Bayesian belief networks depends on the number of dependencies between variables in the model, we wish to have as few dependencies as possible while still preserving the validity of the model. In other words, we want to prune away as many (unnecessary) dependencies as possible from the fully connected graph.

Structure learning can thus be thought of as a problem of the search for the optimal structure in the space of all possible structures for a given set of variables representing some application domain. As noted by Chickering et al. (1994), an optimal solution for this search across existing data sets is NP-hard. The optimal structure describes the data well while remaining as simple as possible.

However, this learned structure must describe both previously observed data and also fit new, possibly relevant, information. Thus, when we create (learn) a complex structure that characterizes the accumulated data exactly, we may end up with a structure that is too specific and as a result fails to generalize well on new data. In the optimization literature, this situation is sometimes called *over-fitting*.

Although searching the exponential space of possible directed acyclic graph structures by brute force methods is rather unrealistic for practical applications with many variables, there are some principles that do help tackle this problem. The first rule of thumb is Occam's razor, or *the principle of parsimony.* Among two competing structures that perform comparably, the simpler structure should be preferred over the more complex. Occam's razor would dictate, that in some cases, it might be a good idea to search for possible structures going from the simple to the complex, terminating search when we find a structure that is simple enough yet succeeds in being a suitable model for the task at hand. Using this simplicity heuristic imposes a Bayesian prior on the structure search.

It is often difficult, however, to define a priori a measure of complexity for a structure. The choice of a reasonable measure of complexity that imposes a total order on possible structures is not obvious. There might be classes of structures that are equally complex, in which case we might be able to impose a partial order on the structure space. Even among structures of the same class, however, an exhaustive search through all the possibilities is often impractical.

As with many difficult problems in AI, structure induction in the general case is simply not possible for reasonably complex situations. As a result, we apply heuristics in an attempt to make structure search tractable. Given a working, although unsatisfactory, model of a situation, one heuristic method is to perform a greedy local search close to the failed components of the model. Given that the goal is to find only a local optimization within the graphical structure, this might offer a good enough solution to the modeling problem. Another heuristic would be to rely on the human expert in the problem domain for advice to help rethink the structure of the model. Again the human expert might focus on the breakdown points of the current model. A more general approach to the problem of model induction is through sampling the space of possible models. We present next a popular technique for this type sampling, Markov chain Monte Carlo.

Markov Chain Monte Carlo (MCMC)

Markov chain Monte Carlo (or *MCMC*) is a class of algorithms for sampling from probability distributions. MCMC is based on constructing a Markov chain that has the desired distribution as its equilibrium distribution. The state of the chain after a large number of steps, sometimes called *burn in time*, is then used as a sample for the desired distributions of a possible DAG structure for the application domain. Of course, the quality of the Markov chain sample improves as a function of the number of samples taken.

In the context of structure search we assume that the possible structures for a given set of variables form the state space of a Markov chain. The transition matrix for these states is populated by some weight based technique. Weights may start as uniform, where a Monte Carlo random walk among these structures would weigh all structural modifications equally likely.

Over time, of course, higher weights are used for transitions between closely related structures. If structure A is different from structure B in only one link, then a Monte Carlo walk of this space is a search in the heuristically organized neighborhoods of these structures. Each structure, when sampled, is associated with a score, typically a *maximum a posteriori* (or *MAP*) based score, the p(structure | data). The Monte Carlo sampling technique traverses across the structure space, choosing with a higher probability the transitions that improve the likelihood of the data, rather than transitions that make the score worse. After multiple iterations, the MCMC approach converges on better structures, improving the posterior distribution of p(structure | data) when compared to its starting point. Again, there is the problem of discovering local maxima in the sampled space; this problem can be handled by techniques such as *random restart* or *simulated annealing*.

Parameter Learning and Expectation Maximization (EM)

Parametric learning may be contrasted with structure learning across the space of graphical models. Given an existing model of a problem domain, parameter learning attempts to infer the optimal joint distribution for a set of variables given a set of observations. Parameter learning is often used as a subroutine within a general structure search. When a complete data set exists, parameter learning for a distribution reduces to counting.

In many interesting applications of graphical models, however, there can be a problem of having an incomplete set of data for a variable. A further - and related - problem is the existence of possible hidden or latent variables within the model. In cases like these the *expectation maximization* (or *EM*) algorithm serves as a powerful tool to infer the likely estimates for joint distributions. The EM algorithm was first explained, and given its name, in a classic paper by A. Dempster, N. Laird, and D. Rubin (1977). This paper generalized the methodology and developed the theory behind it.

EM has proven to be one of the most popular methods of learning in statistical and probabilistic modeling. The EM algorithm is used in statistics for finding maximum likelihood estimates of parameters in probabilistic models, where the model depends on possible unobserved latent variables. EM is an iterative algorithm which alternates between performing an expectation (E) step, that computes an expectation of the likelihood of a variable by including latent variables as if they were observed, and the maximization (M)

step, which computes the maximum likelihood estimates of the parameters from the expected likelihood found in the E step. The parameters found on the M step are then used to begin another E step, and the process is repeated until convergence, that is, until there remains very little oscillation between the values at the E and M steps. We present an example of the expectation maximization algorithm in Section 13.2.3.

Expectation Maximization for HMMs (Baum-Welch)

An important tool for parameter learning is Baum-Welch, a variant of the EM algorithm. It computes maximum likelihood estimates and posterior mode estimates for parameters (transition and emission probabilities) of an HMM or other temporal graphical model, when given only the sets of emission (observable) training data. The Baum-Welch algorithm has two steps:

1. Calculate the forward and the backward probabilities for each temporal state;
2. On the basis of 1, determine the frequency of the transition-emission pair values and divide this by the probability of the entire sequence.

Step 1 of Baum-Welch uses the *forward-backward algorithm*. This is the algorithm supporting dynamic programming, as we first saw in Section 4.1 and have seen several times since, including in Section 13.1.3, when it was used to interpret strings of phones.

Step 2 of the Baum-Welch algorithm is used to compute the probability of a particular output sequence, given the parameters of the model, in the context of hidden Markov or other temporal models. Step 2 amounts to calculating the expected count of the particular transition-emission pair. Each time a transition is found, the value of the ratio of the transition over the probability of the entire sequence increases and this value can then be made the new value of the transition.

A brute force procedure for solving the problem of HMM parameter learning is the generation of all possible sequences of observed events and hidden states with their probabilities using two transition matrices. The joint probability of two sequences, given the model, is calculated by multiplying the corresponding probabilities in the matrices. This procedure has a time complexity of $O(2TN^T)$, where T is the length of the sequences of observed events and N the total number of symbols in the state alphabet. This time complexity cost is intractable for realistic problems, thus the method of choice for implementing Baum-Welch utilizes dynamic programming.

We next demonstrate a simple example of parameter learning using loopy belief propagation (Pearl 1988) with the expectation maximization algorithm (Dempster et al. 1977).

13.2.3 Expectation Maximization: An Example

We illustrate expectation maximization (EM) with an example and use Pearl's (1988) loopy belief propagation algorithm to show the inference steps. This example has been implemented in, and the Markov random field representation is taken from loopy logic

(Pless et al. 2006), a first-order stochastic modeling language discussed briefly in Section 9.3. Although simpler methods are available for solving the constraints of the burglar alarm example we present, our goal is to demonstrate an important inference scheme for BBNs, loopy belief propagation, and the EM algorithm, in an understandable setting.

Consider the Bayesian belief network of Figure 13.10, a BBN with 3 nodes, alarm A, burglary B, and earthquake Q. This model describes a burglar alarm system A_t, active at an observation time t, causally dependent on there being a possible burglary B_t, or an earthquake Q_t. This model, with some assumed probability measures, can be described in loopy logic by the program:

$$A_t \mid B_t , Q_t = [[[.999,.001],[.7,.3]],[[.8,.2],[.1,.9]]]$$

We have simplified the actual loopy logic syntax to make the presentation clearer.

In this program we assume that all of the variables are boolean. We also assume that the BBN from Figure 13.10 contains the conditional probability distribution (CPD) specified above. For example, the table specifies that the probability for the alarm not to go off, given that there is a burglary and an earthquake, is .001; the probability for the alarm to go off, given neither a burglary nor that an earthquake occurred, is .1.

Note also that this program specifies a general class of BBNs, or rather a BBN for every time t. This is somewhat similar to the representational approach taken by Kersting and DeRaedt (2000). However, loopy logic (Pless et al. 2006) extends the approach taken by Kersting and DeRaedt in several ways. For instance, loopy logic converts probabilistic logic sentences, like the one above, into a Markov random field to apply the inference algorithm, loopy belief propagation (Pearl 1988).

A *Markov random field*, or *Markov network*, is a probabilistic graphical model (see Section 9.3) whose structure is an undirected graph (as compared to the directed graphical models of Bayesian belief networks). The nodes of the structure graph of the Markov random field correspond to random variables and the links between the nodes correspond to the dependencies between the connected random variables. In loopy logic the Markov random field is used as an intermediate and equivalent representation for the original graphical model (just as the clique tree was used in Section 9.3), The Markov random field supports probabilistic inference, in our case loopy belief propagation (Pearl 1988).

The quantitative components of the Markov random field are specified by a set of potential functions, whose domain is a clique of the graph structure of the Markov random

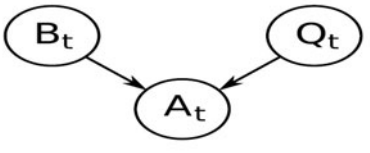

Figure 13.10 A Bayesian belief net for the burglar alarm, earthquake, burglary example.

field, and that define a correspondence between the set of all possible joint assignments to the variables from the clique and the set of nonnegative real numbers.

In the loopy logic system, a special version of a Markov random field is used. A potential function is assigned to each dependency between random variables. Consequently, the Markov random field can be represented by a bipartite graph with two types of nodes, *variable nodes* and *cluster nodes*. Variable nodes correspond to random variables, while cluster nodes correspond to potential functions on random variables specified by the neighbors of the cluster node. Figure 13.11 illustrates the bipartite Markov random field that captures the potential functions of the BBN of Figure 13.10.

In loopy logic we can specify facts about the model and ask queries; for instance, we can state:

B_1 = t.
Q_1 = f.
A_1 = ?

An interpretation of these three statements is that we know that at time 1 there was a burglary, that there was no earthquake, and that we would like to know the probability of the alarm going off. Using these three statements along with $A_t \mid B_t$, Q_t = [[[.999,.001], [.7,.3]], [[.8,.2], [.1,.9]]], the loopy logic system constructs the Markov random field with the facts and potential functions incorporated as in Figure 13.11.

Note that in Figure 13.11 the facts from the program are incorporated into the Markov random field as leaf cluster nodes and the conditional probability distribution (CPD) table is encoded in the cluster node connecting the three variable nodes. In order to infer the answer for this query, the system applies the loopy belief propagation algorithm to the Markov random field. Since we specified simple deterministic facts, and the original graphical model has no loops, the inference algorithm is exact (Pearl 1988) and converges in one iteration returning the distribution for alarm, [0.8,0.2].

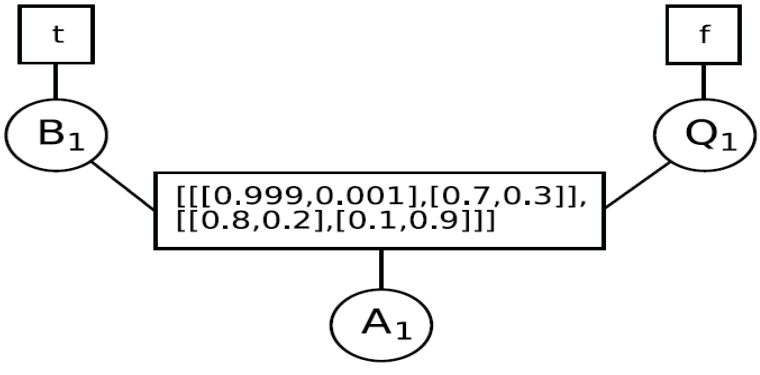

Figure 13.11 A Markov random field reflecting the potential functions of the random variables in the BBN of Figure 13.10, together with the two observations about the system.

In addition, loopy logic supports learning by allowing its users to assign learnable distributions to selected rules of a loopy logic program. These parameters can be estimated using a message passing algorithm derived from expectation maximization (Dempster et al. 1977). To demonstrate this, we next assume that we do not have knowledge of the conditional probability distribution of the BBN of Figure 13.10. In loopy logic, to obtain values for this distribution, the user sets the CPD to be a learnable distribution at an arbitrary time t. To denote this we use the variable L in the program:

$A_t \mid B_t, Q_t = L$

We can now present a collected set of observations to the loopy logic interpreter over a set of time periods using the fact representation, below. For simplicity, we collect only three data points at each of three time intervals:

$Q_1 = f$
$B_1 = t$
$A_1 = t$
$Q_2 = t$
$B_2 = f$
$A_2 = t$
$Q_3 = f$
$B_3 = f$
$A_3 = f$

Loopy logic uses a combination of the loopy belief propagation inference scheme coupled with expectation maximization to estimate the learnable parameters of a model. To accomplish this, the interpreter constructs the Markov random field for the model, as before. For the alarm BBN shown in Figure 13.10, its corresponding Markov random field, with the learnable parameter L linked to each cluster node whose rule unifies with the learnable rule presented earlier ($A_t \mid B_t, Q_t = L$), is presented in Figure 13.12.

We next populate the known distributions of the BBN, and make a random initialization to the cluster nodes R1, R2, and R3 as well as to the learnable node L. Finally, this system is presented with the data representing the three time periods, where known facts are incorporated into the leaf cluster nodes as before. Figure 13.12 shows the result.

The expectation maximization algorithm is an iterative process implemented through message passing. After each iteration the current approximation of the conditional probability table is the learnable node's message to each of its linked cluster nodes. When the learnable node is updated, each neighboring cluster node sends a message to L (dash-dot arrows labeled M in Figure 13.12). This message is the product of all messages coming into that cluster node from the neighboring variable nodes. These (unnormalized) tables are an estimate of the joint probability at each cluster node. This is a distribution over all states of the conditioned and conditioning variables. The learnable node takes the sum of all these cluster messages and converts them to a normalized conditional probability table.

By doing inference (loopy belief propagation) on the cluster and variable nodes, we compute the message for the learnable nodes. Reflecting the bipartite structure of the underlying Markov random field, loopy belief propagation is done with two stages of message passing: sending messages from all cluster nodes to their neighboring variable

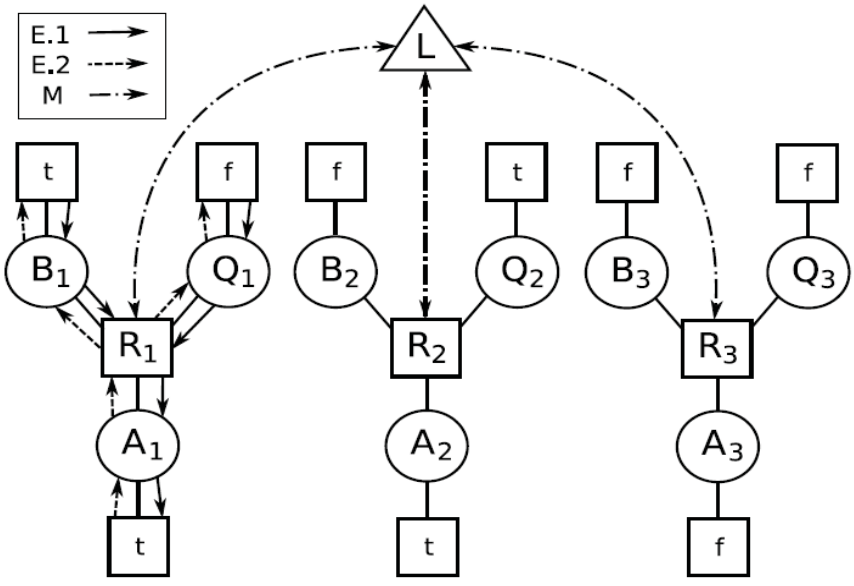

Figure 13.12. A learnable node L is added to the Markov random field of Figure 13.11. The Markov random field iterates across three time periods. For simplicity, the EM iteration is only indicated at time 1.

nodes, the solid arrows labeled E.1 in Figure 13.12, and then back again, the dashed arrows labeled E.2 in Figure 13.12.

Applying this belief propagation algorithm until convergence yields an approximation of the expected values. This is equivalent to the expectation step in the EM algorithm. The averaging that takes place over all learnable cluster nodes corresponds to the maximization step returning a maximum likelihood estimate of the parameters in a learnable node. Iteration that supports convergence in the variable and cluster nodes followed by updating the learnable nodes and then continuing to iterate is equivalent to the full EM algorithm.

To summarize the algorithm, and as displayed in Figure 13.12:

In the expectation or E step:

1. All cluster nodes send their messages to the variable nodes.
2. All variable nodes send their messages back to the cluster nodes updating the information of the cluster nodes.

In the maximization or M step:

1. The cluster nodes send messages, updated in the E step. to learnable nodes where they are averaged.

In the algorithm just described, the nodes can be changed in any order, and updates of cluster and variable nodes may be overlapped with the updates of learning nodes. This

iterative update process represents a family of EM style algorithms, some of which may be more efficient than standard EM (Pless et al. 2006, Dempster et al., 1977) for certain domains. An algorithmic extension that this framework also supports is the *generalized belief propagation* algorithm proposed by Yedidia et al.(2000).

13.3 Stochastic Extensions to Reinforcement Learning

In Section 10.7 we first introduced reinforcement learning, where with reward reinforcement, an agent learns to take different sets of actions in an environment. The goal of the agent is to maximize its long-term reward. Generally speaking, the agent wants to learn a policy, or a mapping between reward states of the world and its own actions in the world.

To illustrate the concepts of reinforcement learning, we consider the example of a recycling robot (adapted from Sutton and Barto 1998). Consider a mobile robot whose job is to collect empty soda cans from business offices. The robot has a rechargeable battery. It is equipped with sensors to find cans as well as with an arm to collect them. We assume that the robot has a control system for interpreting sensory information, for navigating, and for moving its arm to collect the cans. The robot uses reinforcement learning, based on the current charge level of its battery, to search for cans. The agent must make one of three decisions:

1. The robot can actively search for a soda can during a certain time interval;
2. The robot can pause and wait for someone to bring it a can; or
3. The robot can return to the home base to recharge its batteries.

The robot makes decisions either in some fixed time period, or when an empty can is found, or when some other event occurs. Consequently, the robot has three actions while its state is being determined by the charge level of the battery. The reward is set to zero most of the time, it becomes positive once a can is collected, and the reward is negative if the the battery runs out of energy. Note that in this example the reinforcement-based learning agent is a part of the robot that monitors both the physical state of the robot as well as the situation (state) of its external environment: the robot-agent monitors both the state of the robot as well as its external environment.

There is a major limitation of the traditional reinforcement learning framework that we have just described: it is deterministic in nature. In order to overcome this limitation and to be able to use reinforcement learning in a wider range of complex tasks, we extend the deterministic framework with a stochastic component. In the following sections we consider two examples of stochastic reinforcement learning: the *Markov decision process* and the *partially observable Markov decision process*.

13.3.1 The Markov Decision Process (or MDP)

The examples of reinforcement learning presented thus far, both in Section 10.7 and in the introduction of Section 13.3, have been deterministic in nature. In effect, when the agent

executes an action from a specific state at time t, it always results in a deterministically specified new state at time t +1: the system is completely deterministic.

However, in many actual reinforcement learning applications, this might not be the situation: the agent does not have complete knowledge or control over the next state of the world. Specifically, taking an action from state s_t might lead to more than one possible resulting state at time t+1. For example, in the game of chess, when we make a certain move, we often do not know what move our opponent will take. We do not have complete knowledge of what state the world will be in as a result of taking our move. In fact, this uncertainty is true for many multi-person games. A second example is playing the lottery or other games of chance. We select a move uncertain of the likely reward. A third example is in single agent decision making (not a contest against one or more other agents) when the world is so complex that a deterministic response to the choice of a state is not computable. Again, in this situation a response based on probabilities might be the best we can justify. In all these situations, we need a more powerful framework for setting up the reinforcement learning problem. One such framework is the *Markov decision process* (or *MDP*).

MDPs are based on the first-order Markov property, which states that a transition to a next state is represented by a probability distribution that depends only on the current state and possible actions. It is independent of all states prior to the current state. MDPs are discrete-time stochastic processes consisting of a set of states, a set of actions that can be executed from each state, and a reward function associated with each state. As a consequence, MDPs provide a very powerful framework that can be used to model a wide class of reinforcement learning situations. We next define the MDP:

D E F I N I T I O N

A MARKOV DECISION PROCESS, or MDP

A Markov Decision Process is a tuple <S, A, P, R> where:

S is a set of states, and

A is a set of actions.

$p_a(s_t , s_{t+1}) = p(s_{t+1} \mid s_t , a_t = a)$ is the probability that if the agent executes action $a \in A$ from state s_t at time t, it results in state s_{t+1} at time t+1. Since the probability, $p_a \in P$ is defined over the entire state-space of actions, it is often represented with a transition matrix.

R(s) is the reward received by the agent when in state s.

We should observe that the only difference between the MDP framework for reinforcement learning and the reinforcement learning presentation of Section 10.7 is that the transition function is now replaced by a probability distribution function. This is an important modification in our theory enabling us to capture uncertainty in the world, when the world is either non-computably complex or is in fact stochastic. In such cases the reward response to an agent's action is best represented as a probability distribution.

13.3.2 The Partially Observable Markov Decision Process (POMDP)

We observed that MDPs can be used to model a game like chess, in which the next state of the world can be outside an agent's deterministic control. When we make a move in chess the resulting state of the game depends on our opponent's move. Thus, we are aware of the current state we are in, we are aware of which action (move) we are taking, but the resultant state is not immediately accessible to us. We only have a probability distribution over the set of possible states that we are likely to be in, depending on our opponent's move.

Let's now consider the game of poker. Here, we are not even certain of our current state! We know the cards we hold, but we do not have perfect knowledge of our opponent's cards. We can only guess, perhaps with knowledge from bidding, which cards our opponent has. The world is even more uncertain than the world of MDPs. Not only do we not know which state we are in, but we must also execute an action based on our guess of the actual state of the world. As a result, our guess of the action-induced new state of the world will be even more uncertain. Hence, we need a richer framework than the MDP to model this doubly uncertain situation. One framework for accomplishing this task is the *partially observable Markov decision process* (or *POMDP*). It is called a POMDP because we can only partially observe the state we are in as well as what our opponent's response will be. Instead of perfect knowledge of our current state, we only have a probability distribution over the possible set of states we could possibly be in. We define the POMDP:

DEFINITION

A PARTIALLY OBSERVABLE MARKOV DECISION PROCESS, or POMDP

A Partially Observable Markov Decision Process is a tuple $<S, A, O, P, R>$ where:

- S is a set of states, and
- A is a set of actions.
- O is the set of observations denoting what the agent can see about its world. Since the agent cannot directly observe its current state, the observations are probabilistically related to the underlying actual state of the world.
- $p_a(s_t, o, s_{t+1}) = p(s_{t+1}, o_t = o \mid s_t, a_t = a)$ is the probability that when the agent executes action a from state s_t at time t, it results in an observation o that leads to an underlying state s_{t+1} at time $t+1$.
- $R(s_t, a, s_{t+1})$ is the reward received by the agent when it executes action a in state s_t and transitions to state s_{t+1}.

To summarize, the MDP can be considered to be a special case of the POMDP, when the observation o gives us an accurate indication of the current state s. An analogy can be helpful in understanding this distinction: the MDP is related to the Markov chain as the POMDP is related to the hidden Markov model.

There are several algorithms for building MDPs and POMDPs. The complexity of these algorithms is obviously greater than that encountered in the purely deterministic

cases presented earlier. In fact the algorithms for solving POMDPs are computationally intractable. That is, it can be impossible to find an optimal solution for a problem using a POMDP in polynomial time, and a solution may indeed be uncomputable. As a result, we address these problems with algorithms that approximate an optimal solution. References for solution algorithms for MDPs and POMDPs include Sutton and Barto (1998) and Puterman (1998). We next present an example implementation of an MDP.

13.3.3 An Example Implementation of the Markov Decision Process

To illustrate a simple MDP, we take up again the example of the recycling robot (adapted from Sutton and Barto 1998) introduced at the beginning of Section 13.3. Recall that each time this reinforcement learning agent encounters an external event it makes a decision to either actively search for a soda can for recycling, we call this search, wait for someone to bring it a can, we call this wait, or return to the home base to recharge its battery, recharge. These decisions are made by the robot based only on the state of the energy level of the battery. For simplicity, two battery levels are distinguished, low and high: the state space of S = {low, high}. If the current state of the battery is high, the battery is charged, otherwise it's state is low. Therefore the action sets of the agent are A(low) = {search, wait, recharge} and A(high) = {search, wait}. If A(high) included recharge, we would expect the agent to learn that a policy using that action would be suboptimal!

The state of the battery is determined independent of actions taken. If it is high, then we expect that the robot will be able to complete a period of active search without the risk of running down the battery. Thus, if the state is high, then the agent stays in that state with probability a and changes its state to low with probability 1- a. If the agent decides to perform an active search while being in state low, the energy level of the battery remains the same with probability b and the battery runs out of energy with probability 1- b.

We next create a reward system: once the battery is run down, S = low, the robot is rescued, its battery is recharged to high, and it gets a reward of -3. Each time the robot finds a soda can, it receives a reward of 1. We denote the expected amount of cans the robot will collect (the expected reward the agent will receive) while actively searching and while it is waiting by R_search and R_wait, and assume that R_search > R_wait. We also assume that the robot cannot collect any soda cans while returning for recharging and that it cannot collect any cans during a step in which the battery is low. The system just described may be represented by a finite MDP whose transition probabilities ($p_a(s_t, s_{t+1})$) and expected rewards are given in Table 13.1.

To represent the dynamics of this finite MDP we can use a graph, such as Figure 13.13, the transition graph of the MDP for the recycling robot. A transition graph has two types of nodes: state nodes and action nodes. Each state of the agent is represented by a state node depicted as a large circle with the state's name, s, inside. Each state-action pair is represented by an action node connected to the state node denoted as a small black circle. If the agent starts in state s and takes action a, it moves along the line from state node s to action node (s_t , a). Consequently, the environment responds with a transition to the next state's node via one of the arrows leaving action node (s_t, a), where each arrow corresponds to a triple (s_t, a, s_{t+1}), where s_{t+1} is the next state. The arrow is labeled with the

s_t	s_{t+1}	a_t	$p_a(s_t,s_{t+1})$	$R_a(s_t,s_{t+1})$
high	high	search	a	R_search
high	low	search	1 - a	R_search
low	high	search	1 - b	-3
low	low	search	b	R_search
high	high	wait	1	R_wait
high	low	wait	0	R_wait
low	high	wait	0	R_wait
low	low	wait	1	R_wait
low	high	recharge	1	0
low	low	recharge	0	0

Table 13.1. Transition probabilities and expected rewards for the finite MDP of the recycling robot example. The table contains all possible combinations of the current state, s_t, next state, s_{t+1}, the actions and rewards possible from the current state a_t.

transition probability, $p_a(s_t , s_{t+1})$, and the expected reward for the transition, $R_a(s_t , s_{t+1})$, is represented by the arrow. Note that the transition probabilities associated with the arrows leaving an action node always sum to 1.

We see several of the stochastic learning techniques described in this chapter again in Chapter 15, Understanding Natural Language.

13.4 Epilogue and References

This chapter presented machine learning algorithms from the dynamic and stochastic perspective. The heart of probabilistic machine learning is Bayes' and Markov's algorithms. Both these tools have been simplified (the Bayesian belief net and first-order Markov chains) to make stochastic approaches to machine learning both powerful and tractable.

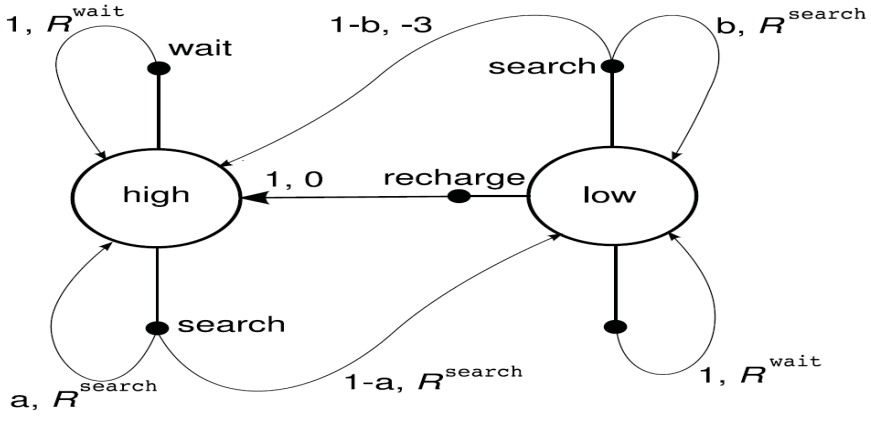

Figure 13.13. The transition graph for the recycling robot. The state nodes are the large circles and the action nodes are the small black states.

Bayes' theorem was first presented in Section 5.3 and the Bayesian belief network in Section 9.3. Markov chains as well as the first-order Markov assumption was presented in Section 9.3. Their uses and extensions make up the material of Chapter 13.

There are a number of introductory texts and tutorials in stochastic approaches to modeling and learning. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference* by J. Pearl (1988), is an early seminal presentation that clearly focused the field of graphical models. We also recommend *Bayesian Networks and Decision Graphs* by F.V. Jensen (1996), *Probabilistic Networks and Expert Systems* by R.G. Cowell et al. (1999), *Graphical Models* by S. Lauritzen (1996), *An Introduction to Bayesian Networks* by F. Jensen (2001), and *Learning in Graphical Models* by M.I. Jordan (1998).

Several further authors present probabilistic learning from a more statistical viewpoint, including J. Whittaker (1990), *Graphical Models in Applied Multivariate Statistics* and D. Edwards (2000), *Introduction to Graphical Modeling.* An engineering/communication perspective is taken by B. Frey (1998), *Graphical Models for Machine Learning and Digital Communication*.

J. Pearl's (2000) *Causality* is an important contribution to the philosophy supporting graphical models. Along with attempting to clarify the notion of causality itself, he delineates its importance and utility within the design of probabilistic models.

Tutorial introductions to graphical models include E. Charniak (1991) *Bayesian Networks without Tears*, and P. Smyth (1997) *Belief Networks, Hidden Markov Models, and Markov Random Fields: A Unifying View.* Excellent on-line tutorial introductions to the field include K. Murphy's *A Brief Introduction to Graphical Models and Bayesian Networks* [www.cs.ubc.ca/~murphyk/Bayes/bnintro.html#appl] as well as Murphy's *A Brief Introduction to Bayes' Rule* at [www.cs.ubc.ca/~murphyk/Bayes/bayesrule.html]. A more complete listing of tutorials can be found at [www.cs.engr.uky.edu/~dekhtyar/dblab/resources.html#bnets].

The EM algorithm was explained and given its name in a classic 1977 paper by A. Dempster, N. Laird, and D. Rubin. They pointed out that method had been "proposed many times in special circumstances" by other authors, but their 1977 paper generalized the method and developed the theory behind it.

Inference in Markov and dynamic Bayesian networks comes in two forms, exact, where results can be computationally expensive but are demonstrably correct, and approximate, where their cost is less but the results are approximate, and can be local maxima. For exact reasoning see S.M Aji and R. McEliece (2000) *The Generalized Distributive Law* and F. Kschischang et al. (2000) *Factor Graphs and the Sum Product Algorithm*. For approximate algorithms see M.I. Jordan et al. (1999) *An Introduction to Variational Methods for Graphical Models*, T. Jaakkola and M.I. Jordan (1998) *Variational Probabilistic Inference and the QMR-DT Database*, and J. Yedidia et al. (2000) *Generalized Belief Propagatio*n. Loopy belief propagation, an approximation algorithm for graphical models is described in J. Pearl (1988) *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*.

In recent years the creation of various forms of first-order stochastic reasoning languages has offered an important research area. We suggest D. Koller and A. Pfeffer (1998) *Probabilistic Frame-Based Systems*, N. Friedman et al. (1999) *Learning Probabilistic Relational Models*, K. Kersting and L. DeRaedt (2000) *Bayesian Logic Programs*, R. Ng

and V. Subrahmanian (1992) Probabilistic Logic Programming, and Pless et al. (2006) *The Design and Testing of a First-Order Stochastic Modeling Language*.

Important contributors to reinforcement learning include R.S. Sutton (1988) *Learning to Predict by the method of Temporal Differences*, R.S. Sutton and A.G. Barto (1998) *Reinforcement Learning: An Introduction,* and M. L. Puterman (1994) *Markov Decision Processes*.

The author wishes to thank several of his present and recent graduate students, especially Dan Pless, the creator of the first-order stochastic inference language loopy logic (Pless et al. 2006), Chayan Chakrabarti (Chakrabarti et al. 2006), Roshan Rammohan, and Nikita Sakhanenko (Sakhanenko et al. 2007) for many of the ideas, figures, examples, and exercises of Chapter 13.

13.5 Exercises

1. Create a hidden Markov model to represent the scoring sequence of an American football game where touchdowns are 6 points which can be followed by either a 0, 1, or 2 extra point attempt. Of course, there are two teams and either can score. If a team has the ball and does not score they emit a 0. So suppose the emit stream of scores is 6,1,6,0,3,0,6,1,0,3,0; what would your HMM look like?
   a. Discuss this problem and what the best HMM might look like.
   b. Test your HMM on two more made up streams of scores.
   c. Track an actual game and see how well your system predicts the scoring stream.
2. Create a hidden Markov model to predict the half inning score sequence of an American baseball game. Suppose the half inning sequence is 0, 0, 0, 1, 0, 1, 1, 2, 2, 0, 0, 0, 0, 2, 0, 0, 0, 0. For simplicity, constrain the scoring for each team to 0, 1, or 2 runs during their half inning at bat.
   a. How would you change the model to allow any number of runs per half inning?
   b. How might you train your system to obtain more realistic score values?
3. Create a figure to represent the hierarchical hidden Markov model of Section 13.1.2. What type problem situation might your HHMM be appropriate for? Discuss the issue of fitting the structures of HMMs to application domains.
4. Given the example of the Viterbi algorithm processing the probabilistic finite state machine of Figure 13.7 and 13.8:
   a. Why is new seen as a better interpretation than knee for the observed phones?
   b. How are alternative states in the probabilistic finite state machine handled by the Viterbi algorithm, for example, the choice of the phones uw and iy in the word new?
5. Given the hidden Markov model and Viterbi algorithm of Section 9.3.6, perform a full trace, including setting up the appropriate back pointers, that shows how the observation #, n, iy, t, # would be processed.

6. Hand run the robot described in the Markov decision process example of Section 13.3.3. Use the same reward mechanism and select probabilistic values for a and b for the decision processing.

    a. Run the robot again with different values for a and b. What policies give the robot the best chances for reward?

7. Program the MDP supported robot of Section 13.3.3 in the language of your choice. Experiment with different values of a and b that can optimize the reward. There are several interesting possible policies: If recharge is a policy of A(high), would your robot learn that this policy is suboptimal? Under what circumstances would the robot always search for empty cans, i.e., the policy for A(low) = recharge is suboptimal?

8. Jack has a car dealership and is looking for a way to maximize his profits. Every week, Jack orders a stock of cars, at the cost of d dollars per car. These cars get delivered instantly. The new cars get added to his inventory. Then during the week, he sells some random number of cars, k, at a price of c each. Jack also incurs a cost u for every unsold car that he has to keep in inventory. Formulate this problem as a Markov decision process. What are the states and actions? What are the rewards? What are the transition probabilities? Describe the long-term return.

9. Consider the general domain of grid-world navigation tasks, where there is a goal state, obstacles, and a discount factor $\gamma < 1$. The actions are stochastic, so the agent may slip into a different cell when trying to move. There are five possible actions: go north, south, east, west, or stay in the same location. Consider the situation in which negative costs are incurred when bumping into walls. Can you draw a 3x3 example environment in which the best action in at least one state is to stay? If so, specify the actions, rewards and transition probabilities. If not, explain why.

10. When we go out to dinner, we always like to park as close as possible to the restaurant. Assume the restaurant is situated on a very long street running east to west, which allows parking on one side only. The street is divided into one-car-length sections. We approach the restaurant from the east, starting D units away. The probability that a parking spot at distance s from the restaurant is unoccupied is ps, independent of all other spots. Formulate this problem as a Markov decision process. Be sure to specify all the elements of your MDP! (Adapted from Puterman, 1994).

11. How would you change the MDP representation of Section 13.3 to a POMDP? Take the simple robot problem and its Markov transition matrix created in Section 13.3.3 and change it into a POMDP. Hint: think of using a probability matrix for the partially observable states.

12. We discussed the card game Poker briefly in Section 13.3. Given that the present (probabilistic) state of a player is either good bet, questionable bet, or bad bet, work out a POMDP to represent this situation. You might discuss this using the probabilities that certain possible poker hands can be dealt.

13. Work out the complexity cost for finding an optimal policy for the POMDP problem exhaustively.