

15

PROBABILISTIC REASONING OVER TIME

In which we try to interpret the present, understand the past, and perhaps predict the future, even when very little is crystal clear

Agents in uncertain environments must be able to keep track of the current state of the environment, just like the logical agents in Part III. The task is made more difficult by partial and noisy percepts and uncertainty about how the environment changes over time. At best, the agent will be able to obtain only a probabilistic assessment of the current situation. This chapter describes the representations and inference algorithms that make this possible, building on the ideas introduced in Chapter 14.

The basic approach is described in Section 15.1: a changing world is modelled using a random variable for each aspect of the world state *at each point in time*. The relations among these variables describe how the state evolves. Section 15.2 defines the basic inference tasks and describes the general structure of inference algorithms for temporal models. Then we describe three specific kinds of models: **hidden Markov models**, **Kalman filters**, and **dynamic Bayesian networks** (which include hidden Markov models and Kalman filters as special cases). Finally, Section 15.6 explains how temporal probability models form the core of modern speech recognition systems. Learning plays a central role in the construction of all these models, but detailed investigation of learning algorithms is left until Part VI.

15.1 TIME AND UNCERTAINTY

We have developed our techniques for probabilistic reasoning in the context of **static** worlds, in which each random variable has a single fixed value. For example, when repairing a car, we assume that whatever is broken remains broken during the process of diagnosis; our job is to infer the state of the car from observed evidence, which also remains fixed.

Now consider a slightly different problem—treating a diabetic patient. As in the case of car repair, we have evidence such as recent insulin doses, food intake, blood sugar measurements, and other physical signs. The task is to assess the current state of the patient, including actual blood sugar level and insulin level. Given this information, the doctor (or patient) makes a decision about food intake and insulin dose. Unlike the case of car repair,

here the *dynamic* aspects of the problem are essential. Blood sugar levels, and measurements thereof, can change rapidly over time, depending on recent food intake and insulin doses, metabolic activity, time of day, and so on. To assess the current state from the history of evidence and to predict the outcomes of treatment actions, we must model these changes.

The same considerations arise in many other contexts, ranging from tracking the economic activity of a nation, given approximate and partial statistics, to understanding a sequence of spoken words, given noisy and ambiguous acoustic measurements. How can dynamic situations like these be modelled?

States and observations

TIME SLICE

The basic approach we will adopt is very similar to the idea underlying situation calculus, as described in Chapter 10: the process of change can be viewed as a series of snapshots, each of which describes the state of the world at a particular time. Each snapshot or **time slice** contains a set of random variables, some of which are observable and some of which are not. For simplicity, we will assume that the same subset of variables is observable in each time slice (although this is not strictly necessary in anything that follows). We will use \mathbf{X}_t to denote the set of unobservable state variables at time t and \mathbf{E}_t to denote the set of observable evidence variables. The observation at time t is $\mathbf{E}_t = \mathbf{e}_t$ for some set of values \mathbf{e}_t .

Consider the following oversimplified example. Suppose you are the security guard at some secret underground installation. You want to know if it's raining today, but your only access to the outside world occurs each morning when you see the director coming in with, or without, an umbrella. For each day t , the set \mathbf{E}_t thus contains a single evidence variable U_t (whether the umbrella appears), and the set \mathbf{X}_t contains a single state variable R_t (whether it is raining). Other problems may involve larger sets of variables. In the diabetes example, we might have evidence variables such as $MeasuredBloodSugar_t$, $PulseRate_t$, etc., with state variables such as $BloodSugar_t$, $StomachContents_t$, and so on.¹

The interval between time slices also depends on the problem. For diabetes monitoring, a suitable interval might be an hour rather than a day. In this chapter, we will generally assume a fixed, finite interval; this means that times can be labelled by integers. We will assume that the state sequence starts at $t = 0$; for various uninteresting reasons, we will assume that evidence starts arriving at $t = 1$ rather than $t = 0$. Hence our umbrella world is represented by state variables R_0, R_1, R_2, \dots and evidence variables U_1, U_2, \dots . We will use the notation $a : b$ to denote the sequence of integers from a to b , and the notation $\mathbf{X}_{a:b}$ to denote the corresponding set of variables from \mathbf{X}_a to \mathbf{X}_b . For example, $U_{1:3}$ corresponds to the variables U_1, U_2, U_3 .

Stationary processes and the Markov assumption

Having decided on the set of state and evidence variables for a given problem, the next step is to specify the dependencies among the variables. We could follow the procedure laid down in Chapter 14, placing the variables in some order and asking questions about conditional

¹ Notice that $BloodSugar_t$ and $MeasuredBloodSugar_t$ are not the same variable; this is how we deal with noisy measurements of actual quantities.

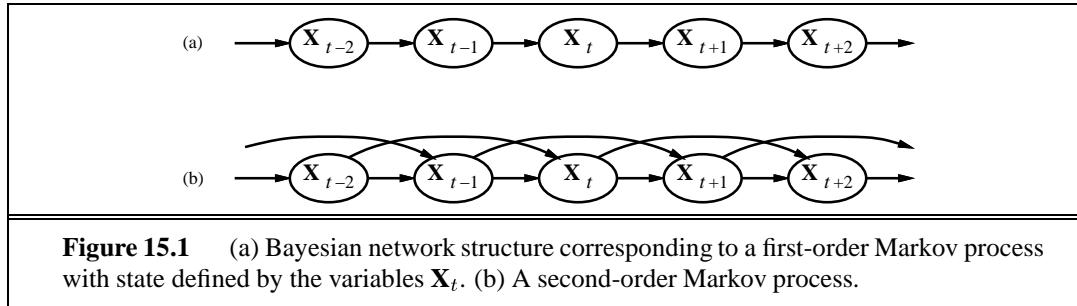


Figure 15.1 (a) Bayesian network structure corresponding to a first-order Markov process with state defined by the variables \mathbf{X}_t . (b) A second-order Markov process.

independence of predecessors given some set of parents. One obvious choice is to order the variables in their natural temporal order, since cause usually precedes effect and we prefer to add the variables in causal order.

We would quickly run into an obstacle, however: the set of variables is unbounded, since it includes the state and evidence variables for every time slice. This actually creates two problems: first, we might have to specify an unbounded number of conditional probability tables—one for each variable in each slice; and second, each one might involve an unbounded number of parents.

The first problem is solved by assuming that changes in the world state are caused by a **stationary process**—that is, a process of change that is governed by laws that do not themselves change over time. (Don’t confuse *stationary* with *static*: in a *static* process, the state itself does not change.) In the umbrella world, then, the conditional probability that the umbrella appears, $\mathbf{P}(U_t | \text{Parents}(U_t))$, is the same for all t . Given the assumption of stationarity, therefore, we need specify conditional distributions only for the variables within a “representative” time slice.

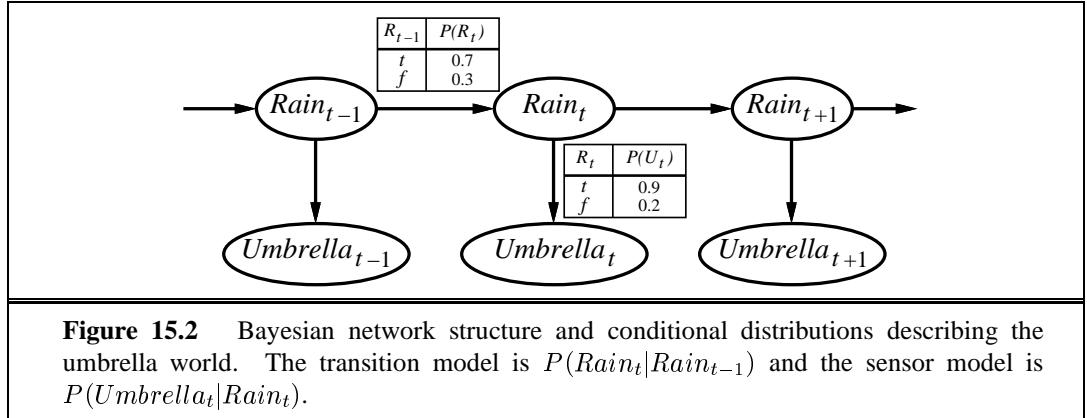
The second problem, that of handling the potentially infinite number of parents, is solved by making what is called a **Markov assumption**, that is, that the current state depends on only a *finite* history of previous states. Processes satisfying this assumption were first studied in depth by the Russian statistician A. A. Markov and are called **Markov processes** or **Markov chains**. They come in various flavors; the simplest is the **first-order Markov process**, in which the current state depends only on the previous state and not on any earlier states. Using our notation, the corresponding conditional independence assertion states that, for all t ,

$$\mathbf{P}(\mathbf{X}_t | \mathbf{X}_{0:t-1}) = \mathbf{P}(\mathbf{X}_t | \mathbf{X}_{t-1}) \quad (15.1)$$

Hence, in a first-order Markov process, the laws describing how the state evolves over time are contained entirely within the conditional distribution $\mathbf{P}(\mathbf{X}_t | \mathbf{X}_{t-1})$, which we call the **transition model**.² The transition model for a second-order Markov process is the conditional distribution $\mathbf{P}(\mathbf{X}_t | \mathbf{X}_{t-2}, \mathbf{X}_{t-1})$. Figure 15.1 shows the Bayesian network structures corresponding to first-order and second-order Markov processes.

In addition to restricting the parents of the state variables \mathbf{X}_t , we must also restrict the

² The transition model is the probabilistic analogue of the Boolean update circuits in Chapter 7 and the successor-state axioms in Chapter 10.



parents of the evidence variables \mathbf{E}_t . Typically, we will assume that the evidence variables at time t depend only on the current state:

$$\mathbf{P}(\mathbf{E}_t | \mathbf{X}_{0:t}, \mathbf{E}_{0:t-1}) = \mathbf{P}(\mathbf{E}_t | \mathbf{X}_t) \quad (15.2)$$

SENSOR MODEL

The conditional distribution $\mathbf{P}(\mathbf{E}_t | \mathbf{X}_t)$ is called the **sensor model** (or sometimes the **observation model**), because it describes how the “sensors”—that is, the evidence variables, are affected by the actual state of the world. Notice the direction of the dependence: the “arrow” goes from state to sensor values because the state of the world *causes* the sensors to take on particular values. In the umbrella world, for example, the rain *causes* the umbrella to appear. (The inference process, of course, goes in the other direction; the distinction between the direction of modelled dependencies and the direction of inference is one of the principal advantages of Bayesian networks.)

In addition to the transition model and sensor model, we also need to specify a prior probability $\mathbf{P}(\mathbf{X}_0)$ over the states at time 0. These three distributions, combined with the the conditional independence assertions in Equations (15.1) and (15.2), give us a specification of the complete joint distribution over all the variables. For any finite t , we have

$$\mathbf{P}(\mathbf{X}_0, \mathbf{X}_1, \dots, \mathbf{X}_t, \mathbf{E}_1, \dots, \mathbf{E}_t) = \mathbf{P}(\mathbf{X}_0) \prod_{i=1}^t \mathbf{P}(X_i | \mathbf{X}_{i-1}) \mathbf{P}(\mathbf{E}_i | \mathbf{X}_i) \quad (15.3)$$

The independence assumptions correspond to a very simple structure for the Bayesian network describing the whole system. Figure 15.2 shows the network structure for the umbrella example, including the conditional distributions for the transition and sensor models.

RANDOM WALK

The structure in the figure assumes a first-order Markov process, because the probability of rain is assumed to depend only on whether it rained the previous day. Whether such an assumption is reasonable depends on the domain itself. The first-order Markov assumption says that the state variables contain *all* the information needed to characterize the probability distribution for the next time slice. Sometimes the assumption is exactly true—for example, if a particle is executing a **random walk** along the x -axis, changing its position by ± 1 at each time step, then using the x -coordinate as the state gives a first-order Markov process. Sometimes the assumption is only approximate, as in the case of predicting rain just based on

whether it rained the previous day. There are two possible fixes if the approximation proves too inaccurate:

1. Increasing the order of the Markov process model. For example, we could make a second-order model by adding $Rain_{t-2}$ as a parent of $Rain_t$, which might give slightly more accurate predictions.
2. Increasing the set of state variables. For example, we could add $Temperature_t$ and $Pressure_t$ to help in predicting the weather.

Exercise 15.1 asks you to show that the first solution—increasing the order—can always be reformulated as an increase in the set of state variables, keeping the order fixed. Notice that adding state variables may improve predictive power but also increases the prediction *requirements*, since we also have to predict the new variables. Thus, we are looking for a “self-sufficient” set of variables, which really means that we have to understand the “physics” of the process being modelled. The requirement for accurate modelling of the process is obviously lessened if we can add new sensors (e.g., measurements of temperature and pressure) that provide information directly about the new state variables.

Consider, for example, the problem of tracking a robot wandering randomly on the X–Y plane. One might propose that the position and velocity are a sufficient set of state variables: one can simply use Newton’s laws to calculate the new position, and the velocity may change unpredictably. If the robot is battery-powered, however, then battery exhaustion would tend to have a systematic effect on the change in velocity. Because this in turn depends on how much power was used by all previous maneuvers, the Markov property is violated. We can restore the Markov property by including the charge level $Battery_t$ as one of the state variables that comprise \mathbf{X}_t . This helps in predicting the motion of the robot, but in turn requires a model for predicting $Battery_t$ given $Battery_{t-1}$ and the velocity. In some cases this can be done reliably; accuracy would be improved by *adding a new sensor* that measures the battery level.

15.2 INFERENCE IN TEMPORAL MODELS

Having set up the structure of a generic temporal model, we can formulate the basic inference tasks that must be solved. They are as follows:

FILTERING
MONITORING
BELIEF STATE

PREDICTION

- ◊ **Filtering or monitoring:** this is the task of computing the **belief state**—the posterior distribution over the current state, given all evidence to date. That is, we wish to compute $\mathbf{P}(\mathbf{X}_t | \mathbf{e}_{1:t})$, assuming that evidence arrives in a continuous stream beginning at $t = 1$. In the umbrella example, this would mean computing the probability of rain today, given all the observations of the umbrella-carrier made so far. Filtering is what a rational agent needs to do in order to keep track of the current state so that rational decisions can be made (see Chapter 17). It turns out that an almost identical calculation provides the **likelihood** of the evidence sequence, i.e., $P(\mathbf{e}_{1:t})$.
- ◊ **Prediction:** This is the task of computing the posterior distribution over the *future* state, given all evidence to date. That is, we wish to compute $\mathbf{P}(\mathbf{X}_{t+k} | \mathbf{e}_{1:t})$ for some $k > 0$.

In the umbrella example, this might mean computing the probability of rain three days from now, given all the observations of the umbrella-carrier made so far. Prediction is useful for evaluating possible courses of action.

SMOOTHING

HINDSIGHT

- ◊ **Smoothing or hindsight:** This is the task of computing the posterior distribution over a *past* state, given all evidence up to the present. That is, we wish to compute $\mathbf{P}(\mathbf{X}_k | \mathbf{e}_{1:t})$ for some k such that $0 \leq k < t$. In the umbrella example, this might mean computing the probability that it rained last Wednesday, given all the observations of the umbrella-carrier made up to today. Hindsight provides a better estimate of the state than was available at the time, because it incorporates more evidence.
- ◊ **Most likely explanation:** Given a sequence of observations, we may wish to find the most likely sequence of states that generated those observations. That is, we wish to compute $\arg \max_{\mathbf{x}_{1:t}} P(\mathbf{x}_{1:t} | \mathbf{e}_{1:t})$. For example, if the umbrella appears on each of the first three days and is absent on the fourth, then the most likely explanation is that it rained on the first three days and did not rain on the fourth. Algorithms for this task are useful in many applications, including speech recognition—where the aim is to find the most likely sentence, given a series of sounds—and reconstruction of bit strings transmitted over a noisy channel.

In addition to these tasks, methods are also needed for *learning* the transition and sensor models from observations. Just as with static Bayesian networks, DBN learning can be done as a by-product of inference. Inference provides an estimate of what transitions actually occurred and of what states generated the sensor readings, and these estimates can be used to update the models. The updated models provide new estimates, and the process iterates to convergence. The overall process is an instance of the **EM algorithm** (see Section 19.3). One point to note is that learning requires the full smoothing inference, rather than filtering, because it provides better estimates of the states of the process. Learning with filtering may fail to converge correctly; consider, for example, the problem of learning to solve murders—hindsight is *always* required to infer what happened at the murder scene.

Algorithms for the four inference tasks listed in the preceding paragraph can be described first at a generic level, independent of the particular kind of model employed. Further improvements specific to each family of models will be described in the corresponding sections.

Filtering and prediction

Let us begin with filtering. We will show that this can be done in a simple online fashion: given the result of filtering up to time t , one can easily compute the result for $t + 1$ given the new evidence \mathbf{e}_{t+1} . That is,

$$\mathbf{P}(\mathbf{X}_{t+1} | \mathbf{e}_{1:t+1}) = f(\mathbf{e}_{t+1}, \mathbf{P}(\mathbf{X}_t | \mathbf{e}_{1:t}))$$

RECURSIVE ESTIMATION

for some function f . This process is often called **recursive estimation**. We can view the calculation as actually being composed of two parts: first, the current state distribution is projected forward from t to $t + 1$, then it is updated using the new evidence \mathbf{e}_{t+1} . This

two-part process emerges quite simply:

$$\begin{aligned}\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t+1}) &= \mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t}, \mathbf{e}_{t+1}) \quad \text{dividing up the evidence} \\ &= \alpha \mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1}, \mathbf{e}_{1:t}) \mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t}) \quad \text{using Bayes' rule} \\ &= \alpha \mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1}) \mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t}) \quad \text{by the Markov property of evidence}\end{aligned}$$

The second term, $\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t})$ represents a one-step prediction of the next state, and the first term updates this with the new evidence; notice that $\mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1})$ is obtainable directly from the sensor model. Now we obtain the one-step prediction for the next state by conditioning on the current state \mathbf{X}_t :

$$\begin{aligned}\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t+1}) &= \alpha \mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1}) \sum_{\mathbf{x}_t} \mathbf{P}(\mathbf{X}_{t+1}|\mathbf{x}_t, \mathbf{e}_{1:t}) P(\mathbf{x}_t|\mathbf{e}_{1:t}) \\ &= \alpha \mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1}) \sum_{\mathbf{x}_t} \mathbf{P}(\mathbf{X}_{t+1}|\mathbf{x}_t) P(\mathbf{x}_t|\mathbf{e}_{1:t}) \quad \text{using the Markov property} \quad (15.4)\end{aligned}$$

Within the summation, the first factor is simply the transition model, and the second is the current state distribution. Hence, we have the desired recursive formulation. We can think of the filtered estimate $\mathbf{P}(\mathbf{X}_t|\mathbf{e}_{1:t})$ as a “message” $\mathbf{f}_{1:t}$ that is propagated forward along the sequence, modified by each transition and updated by each new observation. The process is

$$\mathbf{f}_{1:t+1} = \alpha \text{ FORWARD}(\mathbf{f}_{1:t}, \mathbf{e}_{t+1})$$

where FORWARD implements the update described in Equation (15.4).

 When all the state variables are discrete, the time for each update is constant (independent of t), and the space required is also constant. (The constants depend, of course, on the size of the state space and the specific type of the temporal model in question.) *The time and space requirements for updating must be constant if an agent with limited memory is to keep track of the current state distribution over an unbounded sequence of observations.*

Let us illustrate the filtering process for two steps in the basic umbrella example (see Figure 15.2). We assume that our security guard has some prior belief as to whether it rained on day 0, just before the observation sequence begins. Let’s suppose this is $\mathbf{P}(R_0) = \langle 0.5, 0.5 \rangle$. Now we process the two observations as follows:

- On day 1, the umbrella appears, so $U_1 = \text{true}$. The prediction from $t = 0$ to $t = 1$ is

$$\begin{aligned}\mathbf{P}(R_1) &= \sum_{r_0} \mathbf{P}(R_1|r_0) P(r_0) \\ &= \langle 0.7, 0.3 \rangle \times 0.5 + \langle 0.3, 0.7 \rangle \times 0.5 = \langle 0.5, 0.5 \rangle\end{aligned}$$

and updating with the evidence for $t = 1$ gives

$$\begin{aligned}\mathbf{P}(R_1|u_1) &= \alpha \mathbf{P}(u_1|R_1) \mathbf{P}(R_1) = \alpha \langle 0.9, 0.2 \rangle \langle 0.5, 0.5 \rangle \\ &= \alpha \langle 0.45, 0.1 \rangle \approx \langle 0.818, 0.182 \rangle\end{aligned}$$

- On day 2, the umbrella appears, so $U_2 = \text{true}$. The prediction from $t = 1$ to $t = 2$ is

$$\begin{aligned}\mathbf{P}(R_2|u_1) &= \sum_{r_1} \mathbf{P}(R_2|r_1) P(r_1|u_1) \\ &= \langle 0.7, 0.3 \rangle \times 0.818 + \langle 0.3, 0.7 \rangle \times 0.182 \approx \langle 0.627, 0.373 \rangle\end{aligned}$$

and updating with the evidence for $t = 2$ gives

$$\begin{aligned}\mathbf{P}(R_2|u_1, u_2) &= \alpha \mathbf{P}(u_2|R_2) \mathbf{P}(R_2|u_1) = \alpha \langle 0.9, 0.2 \rangle \langle 0.627, 0.373 \rangle \\ &= \alpha \langle 0.565, 0.075 \rangle \approx \langle 0.883, 0.117 \rangle\end{aligned}$$

Intuitively, the probability of rain increases from day 1 to day 2 because rain persists. Exercise 15.2(a) asks you to investigate this tendency further.

The task of **prediction** can simply be seen as filtering without the addition of new evidence. In fact, the filtering process already incorporates a one-step prediction, and it is easy to derive the following recursive computation for predicting the state at $t + k + 1$ from a prediction for $t + k$:

$$\mathbf{P}(\mathbf{X}_{t+k+1}|\mathbf{e}_{1:t}) = \sum_{\mathbf{x}_{t+k}} \mathbf{P}(\mathbf{X}_{t+k+1}|\mathbf{x}_{t+k}) P(\mathbf{x}_{t+k}|\mathbf{e}_{1:t}) \quad (15.5)$$

Naturally, this computation involves only the transition model and not the sensor model.

MIXING TIME

It is interesting to consider what happens as we try to predict further and further into the future. As Exercise 15.2(b) shows, the predicted distribution for rain converges to a fixed point $\langle 0.5, 0.5 \rangle$, after which it remains constant for all time. This is the **stationary distribution** of the Markov process defined by the transition model (see also page 522). A great deal is known about the properties of such distributions and about the **mixing time**—roughly, the time taken to reach the fixed point. In practical terms, this dooms to failure any attempt to predict the *actual* state for a number of steps that is more than a small fraction of the mixing time. The more uncertainty there is in the transition model, the shorter will be the mixing time and the more the future is obscured.

In addition to filtering and prediction, we can also use a forward recursion to compute the the **likelihood** of the evidence sequence, i.e., $P(\mathbf{e}_{1:t})$. This is a useful quantity if we want to compare different possible temporal models that might have produced the same evidence sequence; for example, in Section 15.6, we compare different words that might have produced the same sound sequence. For this recursion, we use a likelihood message $\ell_{1:t} = \mathbf{P}(\mathbf{X}_t, \mathbf{e}_{1:t})$. It is a simple exercise to show that

$$\ell_{1:t+1} = \text{FORWARD}(\ell_{1:t}, \mathbf{e}_{t+1}).$$

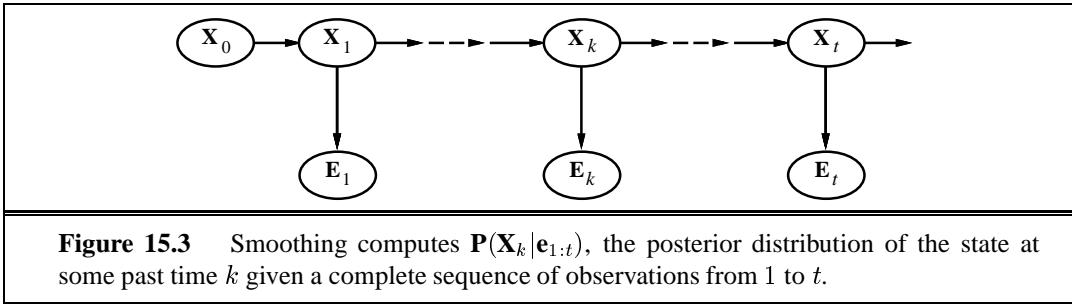
Having computed $\ell_{1:t}$, we obtain the actual likelihood by summing out \mathbf{X}_t :

$$L_{1:t} = P(\mathbf{e}_{1:t}) = \sum_{\mathbf{x}_t} \ell_{1:t}(\mathbf{x}_t). \quad (15.6)$$

Smoothing

As we said earlier, **smoothing** is the process of computing the distribution over past states given evidence up to the present, that is, $\mathbf{P}(\mathbf{X}_k|\mathbf{e}_{1:t})$ for $1 \leq k < t$ (see Figure 15.3). This is done most conveniently in two parts—the evidence up to k and the evidence from $k + 1$ to t :

$$\begin{aligned}\mathbf{P}(\mathbf{X}_k|\mathbf{e}_{1:t}) &= \mathbf{P}(\mathbf{X}_k|\mathbf{e}_{1:k}, \mathbf{e}_{k+1:t}) \\ &= \alpha \mathbf{P}(\mathbf{X}_k|\mathbf{e}_{1:k}) \mathbf{P}(\mathbf{e}_{k+1:t}|\mathbf{X}_k, \mathbf{e}_{1:k}) \quad \text{using Bayes' rule} \\ &= \alpha \mathbf{P}(\mathbf{X}_k|\mathbf{e}_{1:k}) \mathbf{P}(\mathbf{e}_{k+1:t}|\mathbf{X}_k) \quad \text{using conditional independence} \\ &= \alpha \mathbf{f}_{1:k} \mathbf{b}_{k+1:t}\end{aligned} \quad (15.7)$$



where we have defined a “backward” message $\mathbf{b}_{k+1:t} = \mathbf{P}(\mathbf{e}_{k+1:t} | \mathbf{X}_k)$, analogous to the forward message $\mathbf{f}_{1:k}$. The forward message $\mathbf{f}_{1:k}$ can be computed by filtering forward from 1 to k , as given by Equation (15.4). It turns out that the backward message $\mathbf{b}_{k+1:t}$ can be computed by a recursive process that runs *backwards* from t :

$$\begin{aligned}
 \mathbf{P}(\mathbf{e}_{k+1:t} | \mathbf{X}_k) &= \sum_{\mathbf{x}_{k+1}} \mathbf{P}(\mathbf{e}_{k+1:t} | \mathbf{X}_k, \mathbf{x}_{k+1}) \mathbf{P}(\mathbf{x}_{k+1} | \mathbf{X}_k) \quad \text{conditioning on } \mathbf{X}_{k+1} \\
 &= \sum_{\mathbf{x}_{k+1}} P(\mathbf{e}_{k+1:t} | \mathbf{x}_{k+1}) \mathbf{P}(\mathbf{x}_{k+1} | \mathbf{X}_k) \quad \text{by conditional independence} \\
 &= \sum_{\mathbf{x}_{k+1}} P(\mathbf{e}_{k+1}, \mathbf{e}_{k+2:t} | \mathbf{x}_{k+1}) \mathbf{P}(\mathbf{x}_{k+1} | \mathbf{X}_k) \\
 &= \sum_{\mathbf{x}_{k+1}} P(\mathbf{e}_{k+1} | \mathbf{x}_{k+1}) P(\mathbf{e}_{k+2:t} | \mathbf{x}_{k+1}) \mathbf{P}(\mathbf{x}_{k+1} | \mathbf{X}_k)
 \end{aligned} \tag{15.8}$$

where the last step follows by conditional independence of \mathbf{e}_{k+1} and $\mathbf{e}_{k+2:t}$ given \mathbf{X}_{k+1} . Of the three factors in this summation, the first and third are obtained directly from the model, and the second is the “recursive call.” Using the message notation, we have

$$\mathbf{b}_{k+1:t} = \text{BACKWARD}(\mathbf{b}_{k+2:t}, \mathbf{e}_{k+2:t})$$

where BACKWARD implements the update described in Equation (15.8). As with the forward recursion, the time and space required for each update are constant, independent of t .

Given this derivation, we can now see that the two terms in Equation (15.7) can both be computed by recursions through time, one running forward from 1 to k using the filtering equation (15.4) and the other running backward from t to $k + 1$ using Equation (15.8). Note that the backward phase is initialized with $\mathbf{b}_{t+1:t} = \mathbf{P}(\mathbf{e}_{t+1:t} | \mathbf{X}_t) = \mathbf{1}$, where $\mathbf{1}$ is a vector of 1s. (Why?)

Let us now illustrate this algorithm for the umbrella example by computing the smoothed estimate for the probability of rain at $t = 1$, given umbrella observations on day 1 and day 2. From Equation (15.7), this is given by

$$\mathbf{P}(R_1 | u_1, u_2) = \alpha \mathbf{P}(u_2 | R_1) \mathbf{P}(R_1 | u_1) \tag{15.9}$$

The second term we already know to be $\langle .818, .182 \rangle$, from the forward filtering process described earlier. The first term can be computed by applying the backward recursion in Equation (15.8):

$$\mathbf{P}(u_2 | R_1) = \sum_{r_2} P(u_2 | r_2) P(r_2 | R_1)$$

$$= (0.9 \times 1 \times \langle 0.7, 0.3 \rangle) + (0.2 \times 1 \times \langle 0.3, 0.7 \rangle) = \langle 0.69, 0.41 \rangle$$

Plugging this into Equation (15.9), we find that the smoothed estimate for rain on day 1 is

$$\mathbf{P}(R_1|u_1, u_2) = \alpha \langle 0.69, 0.41 \rangle \times \langle 0.818, 0.182 \rangle \approx \langle 0.883, 0.117 \rangle$$

Thus, the smoothed estimate is *higher* than the filtered estimate (0.818) in this case. This is because the umbrella on day 2 makes it more likely to have rained on day 2; in turn, because rain tends to persist, this makes it more likely to have rained on day 1.

Both the forward and backward recursions take a constant amount of time per step, hence the time complexity of smoothing with respect to evidence $\mathbf{e}_{1:t}$ is $O(t)$. This is the complexity for smoothing at a particular time step k . If we want to smooth the whole sequence to get the correct posterior estimate of what actually happened, one obvious method is simply to run the whole smoothing process once for each time step to be smoothed. This results in a time complexity of $O(t^2)$. A better approach uses a very simple application of dynamic programming to reduce this to $O(t)$. A clue appears in the preceding analysis of the umbrella example, where we were able to reuse the results of the forward filtering phase. The key to the linear-time algorithm is to *record the results* of forward filtering over the whole sequence. Then we run the backward recursion from t down to 1, computing the smoothed estimate at each step k from the computed backward message $\mathbf{b}_{k+1:t}$ and the stored forward message $\mathbf{f}_{1:k}$. The algorithm, aptly called the **forward–backward algorithm**, is shown in Figure 15.4.

FORWARD–BACKWARD ALGORITHM

The alert reader will have spotted that the Bayesian network structure shown in Figure 15.3 is a **polytree** in the terminology of Chapter 14. This means that a straightforward application of the clustering algorithm also yields a linear-time algorithm that computes smoothed estimates for the entire sequence. One can show that the forward–backward algorithm is in fact a special case of the polytree propagation algorithm used with clustering methods.

The forward–backward algorithm forms the backbone of the computational methods used in many applications that deal with sequences of noisy observations, ranging from speech recognition to radar tracking of aircraft. As described, it has two practical drawbacks. The first is that its space complexity can be too high for applications where the state space is large and the sequences are long. It uses $O(|\mathbf{f}|t)$ space where $|\mathbf{f}|$ is the size of the representation of the forward message. The space requirement can be reduced to $O(|\mathbf{f}|\log t)$ with a concomitant increase in the time complexity by a factor of $\log t$, as shown in Exercise 15.3. In some cases (see Section 15.3), a constant-space algorithm can be used with no time penalty.

FIXED-LAG SMOOTHING

The second drawback of the basic algorithm is that it needs modification to work in an *online* setting where smoothed estimates must be computed for earlier time slices as new observations are continuously added to the end of the sequence. The most common requirement is for **fixed-lag smoothing**, which requires computing the smoothed estimate $\mathbf{P}(\mathbf{X}_{t-d}|\mathbf{e}_{1:t})$ for fixed d . That is, smoothing is done for the time slice d steps behind the current time t ; as t increases, the smoothing has to keep up. Obviously, we can run the forward–backward algorithm over the d -step “window” as each new observation is added, but this seems inefficient. In Section 15.3, we will see that fixed-lag smoothing can, in some cases, be done in constant time per update, independent of the lag d .

```

function FORWARD-BACKWARD(ev, prior) returns a vector of probability distributions
  inputs: ev, a vector of evidence values for steps  $1, \dots, t$ 
           prior, the prior distribution on the initial state,  $\mathbf{P}(\mathbf{X}_0)$ 
  local variables: fv, a vector of forward messages for steps  $0, \dots, t$ 
                     b, a representation of the backward message, initially all 1s
                     sv, a vector of smoothed estimates for steps  $1, \dots, t$ 

  fv[0]  $\leftarrow$  prior
  for  $i = 1$  to  $t$  do
    fv[ $i$ ]  $\leftarrow$  FORWARD(fv[ $i - 1$ ], ev[ $i$ ])
  for  $i = t$  downto 1 do
    sv[ $i$ ]  $\leftarrow$  NORMALIZE(fv[ $i$ ]  $\times$  b)
    b  $\leftarrow$  BACKWARD(b, ev[ $i$ ])
  return sv

```

Figure 15.4 The forward–backward algorithm for computing posterior probabilities of a sequence of states given a sequence of observations. The FORWARD and BACKWARD operators are defined by Equations (15.4) and (15.8) respectively.

Finding the most likely sequence

Suppose that $[true, true, false, true, true]$ is the umbrella sequence for the security guard’s first five days on the job. What is the most likely weather sequence that explains this? Does the absence of the umbrella on day 3 mean that it wasn’t raining, or did the director forget to bring it? If it didn’t rain on day 3, perhaps (because weather tends to persist) it didn’t rain on day 4 either, but the director brought the umbrella just in case. In all, there are 2^5 possible weather sequences we could pick. Is there a way to find the most likely one, short of enumerating all of them?

One approach we could try is the following linear-time procedure: use the smoothing algorithm to find the posterior distribution for the weather at each time step, then construct the sequence using the most likely weather at each step according to the posterior. Such an approach should set off alarm bells in the reader’s head, because the posteriors computed by smoothing are distributions over *single* time steps, whereas to find the most likely *sequence* we must consider *joint* probabilities over all the time steps. The results may in fact be quite different (see Exercise 15.4).

There *is* a linear-time algorithm for finding the most likely sequence, but it requires a little more thought. It relies on the same Markov property that yielded efficient algorithms for filtering and smoothing. The easiest way to think about the problem is to view each sequence as a *path* through a graph whose nodes are the possible *states* at each time step. Such a graph is shown for the umbrella world in Figure 15.5(a). Now consider the task of finding the most likely path through this graph, where the likelihood of any path is the product of the transition probabilities along the path and the probabilities of the given observations at each state. Let’s focus in particular on paths that reach the state $Rain_5 = true$. Because of the Markov property, we can make the following simple observation: the most likely path



to the state $Rain_5 = true$ consists of the most likely path to *some* state at time 4 followed by a transition to $Rain_5 = true$; and the state at time 4 that will become part of the path to $Rain_5 = true$ is whichever maximizes the likelihood of that path. In other words, *there is a recursive relationship between most likely paths to each state \mathbf{x}_{t+1} and most likely paths to each state \mathbf{x}_t* . We can write this relationship as an equation connecting the probabilities of the paths:

$$\begin{aligned} & \max_{\mathbf{x}_1 \dots \mathbf{x}_t} \mathbf{P}(\mathbf{x}_1, \dots, \mathbf{x}_t, \mathbf{X}_{t+1} | \mathbf{e}_{1:t+1}) \\ &= \mathbf{P}(\mathbf{e}_{t+1} | \mathbf{X}_{t+1}) \max_{\mathbf{x}_t} \left(\mathbf{P}(\mathbf{X}_{t+1} | \mathbf{x}_t) \max_{\mathbf{x}_1 \dots \mathbf{x}_{t-1}} P(\mathbf{x}_1, \dots, \mathbf{x}_{t-1}, \mathbf{x}_t | \mathbf{e}_{1:t}) \right) \end{aligned} \quad (15.10)$$

Equation (15.10) is *identical* to the filtering equation (15.4) except that

1. the forward message $\mathbf{f}_{1:t} = \mathbf{P}(\mathbf{X}_t | \mathbf{e}_{1:t})$ is replaced by the message

$$\mathbf{m}_{1:t} = \max_{\mathbf{x}_1 \dots \mathbf{x}_{t-1}} \mathbf{P}(\mathbf{x}_1, \dots, \mathbf{x}_{t-1}, \mathbf{X}_t | \mathbf{e}_{1:t}),$$

that is, the probabilities of the most likely path to each state \mathbf{x}_t ; and

2. the summation over \mathbf{x}_t in Equation (15.4) is replaced by the maximization over \mathbf{x}_t in Equation (15.10).

Thus, the algorithm for computing the most likely sequence is very similar to filtering: it runs forward along the sequence, computing the \mathbf{m} message at each time step using Equation (15.10). The progress of this computation is shown in Figure 15.5(b). At the end, it will have the probability for the most likely sequence reaching *each* of the final states. One can thus easily select the most likely sequence overall (the state outlined in bold). In order to identify the actual sequence, as opposed to just computing its probability, the algorithm will also need to keep pointers from each state back to the best state that leads to it (shown in bold); the sequence is identified by following the pointers back from the best final state.

VITERBI ALGORITHM

The algorithm we have just described is called the **Viterbi algorithm**, after its inventor. Like the filtering algorithm, its complexity is linear in t , the length of the sequence. Unlike filtering, however, its space requirement is also linear in t . This is because the Viterbi algorithm needs to keep the pointers that identify the best sequence leading to each state.

15.3 HIDDEN MARKOV MODELS

HIDDEN MARKOV MODEL

The preceding section developed algorithms for temporal probabilistic reasoning using a very general framework, independent of the specific form of the transition and sensor models. In this and the following two sections, we discuss more concrete models and applications that illustrate the power of the basic algorithms and in some cases allow further improvements.

We begin with the **hidden Markov model** or **HMM**. An HMM is a temporal probabilistic model in which the state of the process is described by a *single, discrete* random variable. The possible values of the variable are the possible states of the world. The umbrella example described in the preceding section is therefore an HMM, since it has just one state variable, $Rain_t$. Additional state variables can be added to a temporal model while staying within the

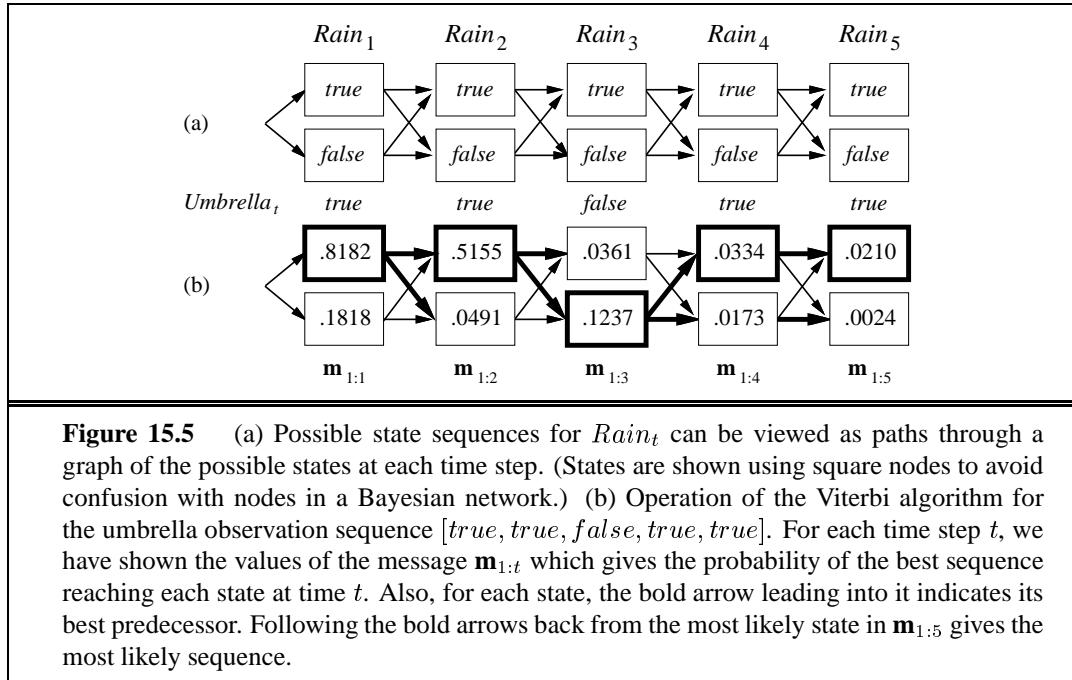


Figure 15.5 (a) Possible state sequences for $Rain_t$ can be viewed as paths through a graph of the possible states at each time step. (States are shown using square nodes to avoid confusion with nodes in a Bayesian network.) (b) Operation of the Viterbi algorithm for the umbrella observation sequence [true, true, false, true, true]. For each time step t , we have shown the values of the message $\mathbf{m}_{1:t}$ which gives the probability of the best sequence reaching each state at time t . Also, for each state, the bold arrow leading into it indicates its best predecessor. Following the bold arrows back from the most likely state in $\mathbf{m}_{1:5}$ gives the most likely sequence.

HMM framework, but only by combining all the state variables into a single “megavariable” whose values are all possible tuples of values of the individual state variables. HMMs usually have a single, discrete evidence variable as well, but this restriction is less important. We will see that the restricted structure of HMMs allows for a very simple and elegant matrix implementation of all the basic algorithms.³ Section 15.6 shows how HMMs are used for speech recognition.

Simplified matrix algorithms

With a single, discrete state variable X_t , we can give concrete form to the representations of the transition model, the sensor model, and the forward and backward messages. Let the state variable X_t have values denoted by integers $1, \dots, S$, where S is the number of possible states. The transition model $\mathbf{P}(X_t|X_{t-1})$ becomes an $S \times S$ matrix \mathbf{T} , where

$$\mathbf{T}_{ij} = P(X_t = j | X_{t-1} = i)$$

That is, \mathbf{T}_{ij} is the probability of a transition from state i to state j . For example, the transition matrix for the umbrella world is

$$\mathbf{T} = \mathbf{P}(X_t|X_{t-1}) = \begin{pmatrix} 0.7 & 0.3 \\ 0.3 & 0.7 \end{pmatrix}$$

We also put the sensor model in matrix form. In this case, because the value of the evidence variable E_t is known to be e_t (say), we need only use that part of the model specifying the

³ For this reason, the reader unfamiliar with basic operations on vectors and matrices might wish to consult Appendix A before continuing.

probability that e_t appears. For each time step t , we construct a diagonal matrix \mathbf{O}_t whose diagonal entries are given by the values $P(e_t|X_t = i)$ and whose other entries are 0. For example, on day 1 in the umbrella world, $U_1 = \text{true}$, so from the table in Figure 15.2 we have

$$\mathbf{O}_1 = \begin{pmatrix} 0.9 & 0 \\ 0 & 0.2 \end{pmatrix}$$

Now, if we use column vectors to represent the forward and backward messages, the computations become simple matrix–vector operations. The forward equation (15.4) becomes

$$\mathbf{f}_{1:t+1} = \alpha \mathbf{O}_{t+1} \mathbf{T}^\top \mathbf{f}_{1:t} \quad (15.11)$$

and the backward equation (15.8) becomes

$$\mathbf{b}_{k+1:t} = \mathbf{T} \mathbf{O}_{k+1} \mathbf{b}_{k+2:t} \quad (15.12)$$

From these equations, we can see that the time complexity of the forward–backward algorithm (Figure 15.4) applied to a sequence of length t is $O(S^2 t)$, as each step requires multiplying an S -element vector by an $S \times S$ matrix. The space requirement is $O(St)$, because the forward pass stores t vectors of size S .

Besides providing an elegant description and implementation of the filtering and smoothing algorithms for HMMs, the matrix formulation also reveals opportunities for improved algorithms. The first is a simple variation on the forward–backward algorithm that allows smoothing to be carried out using *constant* space, independent of the length of the sequence. The idea is that smoothing for any particular time slice k requires the simultaneous presence of both the forward and backward messages, $\mathbf{f}_{1:k}$ and $\mathbf{b}_{k+1:t}$, according to Equation (15.7). The forward–backward algorithm achieves this by storing the \mathbf{f} s computed on the forward pass so that they are available during the backward pass. Another way to achieve this is with a single pass that propagates both \mathbf{f} and \mathbf{b} in the same direction. For example, the “forward” message \mathbf{f} can be propagated backwards if we manipulate Equation (15.11) to work in the other direction:

$$\mathbf{f}_{1:t} = \alpha' (\mathbf{T}^\top)^{-1} \mathbf{O}_{t+1}^{-1} \mathbf{f}_{1:t+1}$$

The modified smoothing algorithm works by first running the standard forward pass to compute $\mathbf{f}_{t:t}$ (forgetting all the intermediate results), then running the backward pass for both \mathbf{b} and \mathbf{f} together, using them to compute the smoothed estimate at each step. Since only one copy of each message is needed, the storage requirements are constant (independent of t , the length of the sequence). There is, of course, one significant restriction on this algorithm: it requires that the transition matrix be invertible and that the sensor model have no zeroes—that is, every observation is possible in every state.

A second area where the matrix formulation reveals an improvement is in *online* smoothing with a fixed lag. The fact that smoothing can be done with constant space suggests that there should exist an efficient recursive algorithm for online smoothing—that is, one whose time complexity is independent of the length of the lag. Let us suppose that the lag is d —that is, we are smoothing at time slice $t - d$ where the current time is t . By Equation (15.7), we need to compute

$$\alpha \mathbf{f}_{1:t-d} \mathbf{b}_{t-d+1:t}$$

for slice $t - d$. Then, when a new observation arrives, we need to compute

$$\alpha \mathbf{f}_{1:t-d+1} \mathbf{b}_{t-d+2:t+1}$$

for slice $t - d + 1$. How can this be done incrementally? First, we can compute $\mathbf{f}_{1:t-d+1}$ from $\mathbf{f}_{1:t-d}$ using the standard filtering process, Equation (15.4).

Computing the backward message incrementally is more tricky, because there is no simple relationship between the old backward message $\mathbf{b}_{t-d+1:t}$ and the new backward message $\mathbf{b}_{t-d+2:t+1}$. Instead, we will examine the relationship between the old backward message $\mathbf{b}_{t-d+1:t}$ and the backward message at the front of the sequence, $\mathbf{b}_{t+1:t}$. To do this, we apply Equation (15.12) d times:

$$\mathbf{b}_{t-d+1:t} = \left(\prod_{i=t-d+1}^t \mathbf{T} \mathbf{O}_i \right) \mathbf{b}_{t+1:t} = \mathbf{B}_{t-d+1:t} \mathbf{1} \quad (15.13)$$

where the matrix $\mathbf{B}_{t-d+1:t}$ is the product of the sequence of \mathbf{T} and \mathbf{O} matrices. \mathbf{B} can be thought of as a “transformation operator” that transforms a later backward message into an earlier one. A similar equation holds for the new backward messages *after* the next observation arrives:

$$\mathbf{b}_{t-d+2:t+1} = \left(\prod_{i=t-d+2}^{t+1} \mathbf{T} \mathbf{O}_i \right) \mathbf{b}_{t+2:t+1} = \mathbf{B}_{t-d+2:t+1} \mathbf{1} \quad (15.14)$$

Examining the product expressions in Equations (15.13) and (15.14), we see that they have a simple relationship: to get the second product, “divide” the first product by the first element \mathbf{TO}_{t-d+1} and multiply by the new last element \mathbf{TO}_{t+1} . In matrix language, then, there is a simple relationship between the old and new \mathbf{B} matrices:

$$\mathbf{B}_{t-d+2:t+1} = \mathbf{O}_{t-d+1}^{-1} \mathbf{T}^{-1} \mathbf{B}_{t-d+1:t} \mathbf{TO}_{t+1} \quad (15.15)$$

This equation provides an incremental update for the \mathbf{B} matrix, which in turn (through Equation (15.14)) allows us to compute the new backward message $\mathbf{b}_{t-d+2:t+1}$. The complete algorithm, which requires storing and updating \mathbf{f} and \mathbf{B} , is shown in Figure 15.6.

15.4 KALMAN FILTERS

Imagine watching a small bird flying through dense jungle foliage at dusk: you glimpse brief, intermittent flashes of motion; you try hard to guess where the bird is and where it will appear next so that you don’t lose it. Or imagine you are a WWII radar operator peering at a faint, wandering blip that appears once every ten seconds on the screen. Or, going back further still, imagine you are Kepler trying to reconstruct the motions of the planets from a collection of highly inaccurate angular observations taken at irregular and imprecisely measured time intervals. In all these cases, you are trying to estimate the state (position and velocity, for example) of a physical system from noisy observations over time. The problem can be formulated as inference in a temporal probability model, where the transition model describes the physics of motion and the sensor model describes the measurement process.

```

function FIXED-LAG-SMOOTHING( $e_t, hmm, d$ ) returns a probability distribution over  $\mathbf{X}_{t-d}$ 
  inputs:  $e_t$ , the current evidence for time step  $t$ 
     $hmm$ , a hidden Markov model with  $S \times S$  transition matrix  $\mathbf{T}$ 
     $d$ , the length of the lag for smoothing
  static:  $t$ , the current time, initially 1
     $\mathbf{f}$ , a probability distribution, the forward message  $\mathbf{P}(X_t | e_{1:t})$ , initially PRIOR[ $hmm$ ]
     $\mathbf{B}$ , the  $d$ -step backward transformation matrix, initially the identity matrix
     $e_{t-d:t}$ , double-ended list of evidence from  $t - d$  to  $t$ , initially empty
  local variables:  $\mathbf{O}_{t-d}, \mathbf{O}_t$ , diagonal matrices containing the sensor model information

  add  $e_t$  to the end of  $e_{t-d:t}$ 
   $\mathbf{O}_t \leftarrow$  diagonal matrix containing  $\mathbf{P}(e_t | X_t)$ 
  if  $t > d$  then
     $\mathbf{f} \leftarrow$  FORWARD( $\mathbf{f}, e_t$ )
    remove  $e_{t-d-1}$  from the beginning of  $e_{t-d:t}$ 
     $\mathbf{O}_{t-d} \leftarrow$  diagonal matrix containing  $\mathbf{P}(e_{t-d} | X_{t-d})$ 
     $\mathbf{B} \leftarrow \mathbf{O}_{t-d}^{-1} \mathbf{T}^{-1} \mathbf{B} \mathbf{O}_t$ 
  else  $\mathbf{B} \leftarrow \mathbf{B} \mathbf{O}_t$ 
   $t \leftarrow t + 1$ 
  if  $t > d$  then return NORMALIZE( $\mathbf{f} \times \mathbf{B}$ ) else return null

```

Figure 15.6 An algorithm for smoothing with a fixed time lag of d steps, implemented as an online algorithm that outputs the new smoothed estimate given the observation for a new time step.

KALMAN FILTERING

This section describes the special representations and inference algorithms that have been developed to solve these sorts of problems; the method we will describe is called **Kalman filtering** after its inventor.

Clearly, we will need several *continuous* variables to specify the state of the system. For example, the bird’s flight might be specified by position (X, Y, Z) and velocity $(\dot{X}, \dot{Y}, \dot{Z})$ at each point in time. We will also need suitable conditional densities to represent the transition and sensor models; as in Chapter 14, we will use **linear Gaussian** distributions. This means that the next state \mathbf{X}_{t+1} must be a linear function of the current state \mathbf{X}_t , plus some Gaussian noise. This turns out to be quite reasonable in practice. Consider, for example, the X -coordinate of the bird, ignoring the other coordinates for now. Let the interval between observations be Δ , and let us assume constant velocity; then the position update is given by

$$X_{t+\Delta} = X_t + \Delta \dot{X}$$

If we add Gaussian noise to account for variation in velocity and so on, then we have a linear Gaussian transition model:

$$P(X_{t+\Delta} = x_{t+\Delta} | X_t = x_t, \dot{X}_t = \dot{x}_t) = N(x_t + \Delta \dot{x}_t, \sigma)(x_{t+\Delta})$$

The Bayesian network structure for a system with position \mathbf{X}_t and velocity $\dot{\mathbf{X}}_t$ is shown in Figure 15.7. Note that this is a very specific form of linear Gaussian model; the general form will be described later in this section, and covers a vast array of applications beyond the sim-

ple motion examples of the first paragraph. The reader may wish to consult Appendix A for some of the mathematical properties of Gaussian distributions; for our immediate purposes, the most important is that a **multivariate Gaussian** distribution for d variables is specified by a d -element mean μ and a $d \times d$ covariance matrix Σ .

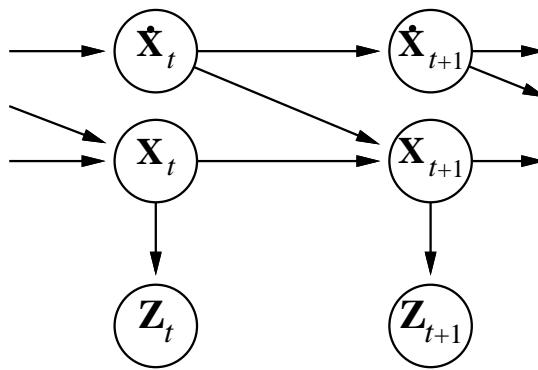


Figure 15.7 Bayesian network structure for a linear dynamical system with position \mathbf{X}_t , velocity $\dot{\mathbf{X}}_t$, and position measurement \mathbf{Z}_t .

Updating Gaussian distributions

We alluded in Chapter 14 to a key property of the linear Gaussian family of distributions: it remains closed under the standard Bayesian network operations. Here, we make this claim precise in the context of filtering in a temporal probability model. The required properties correspond to the two-step filtering calculation in Equation (15.4):

1. If the current distribution $\mathbf{P}(\mathbf{X}_t | \mathbf{e}_{1:t})$ is Gaussian and the transition model $\mathbf{P}(\mathbf{X}_{t+1} | \mathbf{x}_t)$ is linear Gaussian, then the one-step predicted distribution given by

$$\mathbf{P}(\mathbf{X}_{t+1} | \mathbf{e}_{1:t}) = \int_{\mathbf{x}_t} \mathbf{P}(\mathbf{X}_{t+1} | \mathbf{x}_t) P(\mathbf{x}_t | \mathbf{e}_{1:t}) d\mathbf{x}_t \quad (15.16)$$

is also a Gaussian distribution.

2. If the predicted distribution $\mathbf{P}(\mathbf{X}_{t+1} | \mathbf{e}_{1:t})$ is Gaussian and the sensor model $\mathbf{P}(\mathbf{e}_{t+1} | \mathbf{X}_{t+1})$ is linear Gaussian, then, after conditioning on the new evidence, the updated distribution

$$\mathbf{P}(\mathbf{X}_{t+1} | \mathbf{e}_{1:t+1}) = \alpha \mathbf{P}(\mathbf{e}_{t+1} | \mathbf{X}_{t+1}) \mathbf{P}(\mathbf{X}_{t+1} | \mathbf{e}_{1:t}) \quad (15.17)$$

is also a Gaussian distribution.

Thus, the FORWARD operator for Kalman filtering takes a Gaussian forward message $\mathbf{f}_{1:t}$, specified by a mean μ_t and covariance matrix Σ_t , and produces a new multivariate Gaussian forward message $\mathbf{f}_{1:t+1}$, specified by a mean μ_{t+1} and covariance matrix Σ_{t+1} . So, if we start with a Gaussian prior $\mathbf{f}_{1:0} = \mathbf{P}(\mathbf{X}_0) = N(\mu_0, \Sigma_0)$, filtering with a linear Gaussian model produces a Gaussian state distribution for all time.

This seems to be a nice, elegant result, but why is it so important? The reason is that, except for a few special cases such as this, *filtering with continuous or hybrid (discrete and*



(continuous) networks generates state distributions whose representation grows without bound over time. This is not easy to prove in general, but Exercise 15.5 shows what happens for a simple example.

A simple, one-dimensional example

We have said that the FORWARD operator for the Kalman filter maps a Gaussian into a new Gaussian. This translates into computing a new mean and covariance matrix from the previous mean and covariance matrix. Deriving the update rule in the general (multivariate) case requires rather a lot of linear algebra, so will stick to a very simple, univariate case for now; later we will give the results for the general case. Even for the univariate case, the calculations are somewhat tedious, but we feel they are worth seeing because the usefulness of the Kalman filter is tied so intimately to the mathematical properties of Gaussian distributions.

RANDOM WALK

The temporal model we will consider describes a **random walk** of a single continuous state variable X_t with a noisy observation Z_t . An example might be the “consumer confidence” index, which can be modelled as undergoing a random, Gaussian-distributed change each month and is measured by a random consumer survey that also introduces Gaussian sampling noise. The prior distribution is assumed to be Gaussian with variance σ_0^2 :

$$P(x_0) = \alpha e^{-\frac{1}{2} \left(\frac{(x_0 - \mu_0)^2}{\sigma_0^2} \right)}$$

(For simplicity, we will use the same symbol α for all normalizing constants in this section.) The transition model simply adds a Gaussian perturbation of constant variance σ_x^2 to the current state:

$$P(x_{t+1}|x_t) = \alpha e^{-\frac{1}{2} \left(\frac{(x_{t+1} - x_t)^2}{\sigma_x^2} \right)}$$

and the sensor model assumes Gaussian noise with variance σ_z^2 :

$$P(z_t|x_t) = \alpha e^{-\frac{1}{2} \left(\frac{(z_t - x_t)^2}{\sigma_z^2} \right)}$$

Now, given the prior $P(X_0)$, we can compute the one-step predicted distribution using Equation (15.16):

$$\begin{aligned} P(x_1) &= \int_{-\infty}^{\infty} P(x_1|x_0)P(x_0) dx_0 = \alpha \int_{-\infty}^{\infty} e^{-\frac{1}{2} \left(\frac{(x_1 - x_0)^2}{\sigma_x^2} \right)} e^{-\frac{1}{2} \left(\frac{(x_0 - \mu_0)^2}{\sigma_0^2} \right)} dx_0 \\ &= \alpha \int_{-\infty}^{\infty} e^{-\frac{1}{2} \left(\frac{\sigma_0^2(x_1 - x_0)^2 + \sigma_x^2(x_0 - \mu_0)^2}{\sigma_0^2 \sigma_x^2} \right)} dx_0 \end{aligned}$$

COMPLETING THE SQUARE

This integral looks rather hairy. The key to progress is to notice that the exponent is the sum of two expressions that are *quadratic* in x_0 , and hence is itself a quadratic in x_0 . A simple trick known as **completing the square** allows the rewriting of any quadratic $ax_0^2 + bx_0 + c$ as the sum of a squared term $a(x_0 - \frac{-b}{2a})^2$ and a residual term $c - \frac{b^2}{4a}$ that is independent of x_0 . The residual term can be taken outside the integral, giving us

$$P(x_1) = \alpha e^{-\frac{1}{2} \left(c - \frac{b^2}{4a} \right)} \int_{-\infty}^{\infty} e^{-\frac{1}{2} \left(a(x_0 - \frac{-b}{2a})^2 \right)} dx_0$$

Now the integral is just the integral of a Gaussian over its full range, which is simply 1. Thus, we are left with just the residual term from the quadratic.

The second key step is to notice that the residual term has to be a quadratic in x_1 ; in fact, after simplification, we obtain

$$P(x_1) = \alpha e^{-\frac{1}{2} \left(\frac{(x_1 - \mu_0)^2}{\sigma_0^2 + \sigma_x^2} \right)}$$

That is, the one-step predicted distribution is a Gaussian with the same mean μ_0 and a variance equal to the sum of the original variance σ_0^2 and the transition variance σ_x^2 . A momentary exercise of intuition reveals that this is intuitively reasonable.

To complete the update step, we need to condition on the observation at the first time step, namely z_1 . From Equation (15.17), this is given by

$$\begin{aligned} P(x_1|z_1) &= \alpha P(z_1|x_1)P(x_1) \\ &= \alpha e^{-\frac{1}{2} \left(\frac{(z_1 - x_1)^2}{\sigma_z^2} \right)} e^{-\frac{1}{2} \left(\frac{(x_1 - \mu_0)^2}{\sigma_0^2 + \sigma_x^2} \right)} \end{aligned}$$

Once again, we combine the exponents and complete the square (Exercise 15.6), obtaining

$$P(x_1|z_1) = \alpha e^{-\frac{1}{2} \left(\frac{(x_1 - \frac{(\sigma_0^2 + \sigma_x^2)z_1 + \sigma_z^2\mu_0}{\sigma_0^2 + \sigma_x^2 + \sigma_z^2})^2}{\frac{(\sigma_0^2 + \sigma_x^2)\sigma_z^2}{(\sigma_0^2 + \sigma_x^2 + \sigma_z^2)}} \right)} \quad (15.18)$$

Thus, after one update cycle, we have a new Gaussian distribution for the state variable.

From the Gaussian formula in Equation (15.18), we can see that the new mean and standard deviation can be calculated from the old mean and standard deviation as follows:

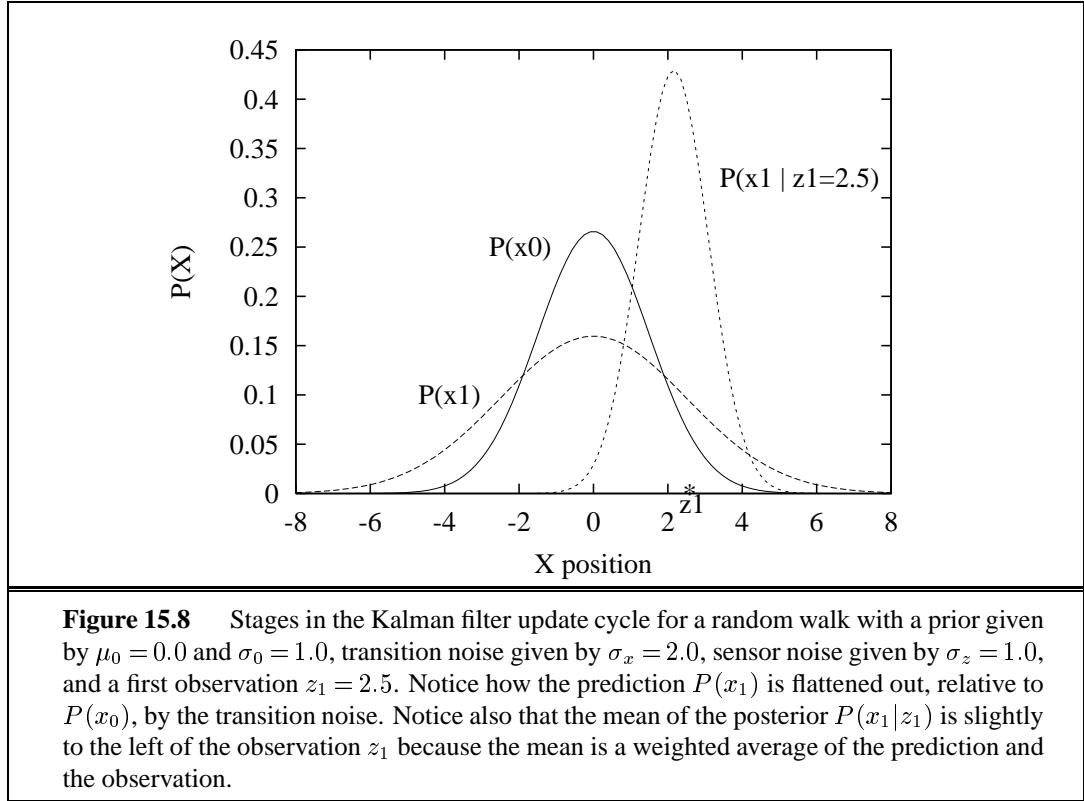
$$\begin{aligned} \mu_{t+1} &= \frac{(\sigma_t^2 + \sigma_x^2)z_{t+1} + \sigma_z^2\mu_t}{\sigma_t^2 + \sigma_x^2 + \sigma_z^2} \\ \sigma_{t+1}^2 &= \frac{(\sigma_t^2 + \sigma_x^2)\sigma_z^2}{\sigma_t^2 + \sigma_x^2 + \sigma_z^2} \end{aligned} \quad (15.19)$$

Figure 15.8 shows one update cycle for particular values of the transition and sensor models.

The preceding pair of equations plays exactly the same role as the general filtering equation (15.4) or the HMM filtering equation (15.11). Because of the special nature of Gaussian distributions, however, the equations have some interesting additional properties. First, we can interpret the calculation for the new mean μ_{t+1} as simply a *weighted mean* of the new observation z_{t+1} and the old mean μ_t . If the observation is unreliable, then σ_z^2 is large and we pay more attention to the old mean; if the old mean is unreliable (σ_t^2 is large) or the process is highly unpredictable (σ_x^2 is large), then we pay more attention to the observation. Second, notice that the update for the variance σ_{t+1}^2 is *independent of the observation*. We can therefore compute in advance what the sequence of variance values will be. Third, the sequence of variance values quickly converges to a fixed value that depends only on σ_x^2 and σ_z^2 , thereby substantially simplifying the subsequent calculations (see Exercise 15.7).

The general case

The preceding derivation, painful as it was, illustrates the key property of Gaussian distributions that allows Kalman filtering to work: the fact that the exponent is a quadratic form.



This is true not just for the univariate case. The full multivariate Gaussian distribution has the form

$$N(\boldsymbol{\mu}, \boldsymbol{\Sigma})(\mathbf{x}) = \alpha e^{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})}$$

Multiplying out the terms in the exponent, it is clear that the exponent is also a quadratic function of the random variables x_i in \mathbf{x} . As in the univariate case, the filtering update preserves the Gaussian nature of the state distribution.

Let us first define the general temporal model used with Kalman filtering. Both the transition model and the sensor model allow for a *linear* transformation with additive Gaussian noise. Thus, we have

$$\begin{aligned} P(\mathbf{x}_{t+1}|\mathbf{x}_t) &= N(\mathbf{F}\mathbf{x}_t, \boldsymbol{\Sigma}_x)(\mathbf{x}_{t+1}) \\ P(\mathbf{z}_t|\mathbf{x}_t) &= N(\mathbf{H}\mathbf{x}_t, \boldsymbol{\Sigma}_z)(\mathbf{z}_t) \end{aligned} \tag{15.20}$$

where \mathbf{F} and $\boldsymbol{\Sigma}_x$ are matrices describing the linear transition model and transition noise covariance, and \mathbf{H} and $\boldsymbol{\Sigma}_z$ are the corresponding matrices for the sensor model. Now the update equations for the mean and covariance, in their full, hairy horribleness, are as follows:

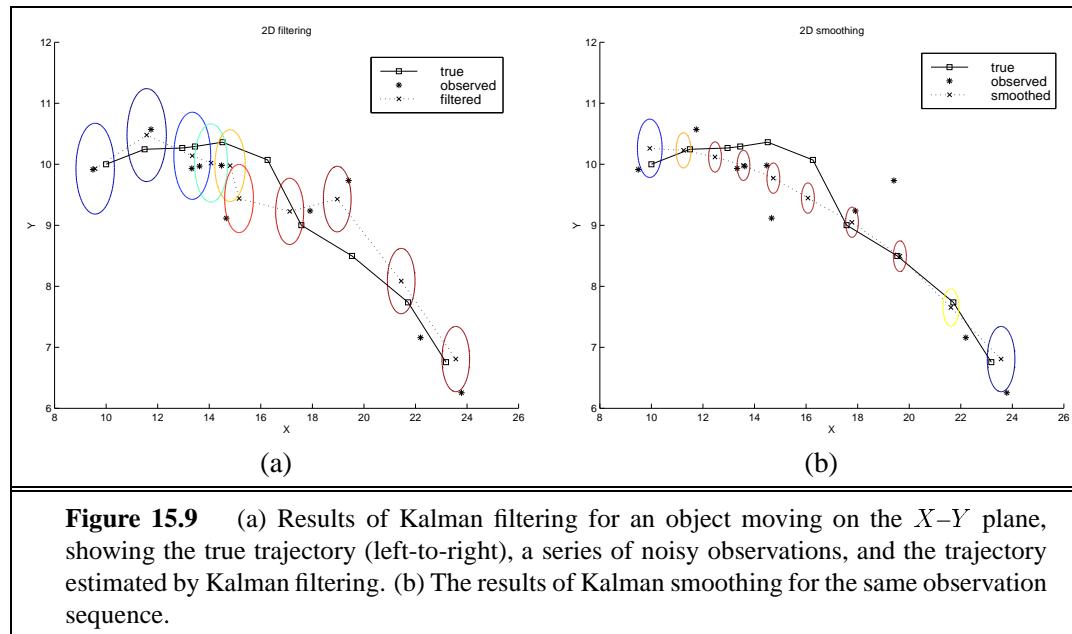
$$\begin{aligned} \boldsymbol{\mu}_{t+1} &= \mathbf{F}\boldsymbol{\mu}_t + \mathbf{K}_{t+1}(\mathbf{z}_{t+1} - \mathbf{H}\mathbf{F}\boldsymbol{\mu}_t) \\ \boldsymbol{\Sigma}_{t+1} &= (\mathbf{I} - \mathbf{K}_{t+1})(\mathbf{F}\boldsymbol{\Sigma}_t\mathbf{F}^\top + \boldsymbol{\Sigma}_x) \end{aligned} \tag{15.21}$$

where $\mathbf{K}_{t+1} = (\mathbf{F}\boldsymbol{\Sigma}_t\mathbf{F}^\top + \boldsymbol{\Sigma}_x)\mathbf{H}^\top(\mathbf{H}(\mathbf{F}\boldsymbol{\Sigma}_t\mathbf{F}^\top + \boldsymbol{\Sigma}_x) + \boldsymbol{\Sigma}_z)^{-1}$ is called the **Kalman gain matrix**. Believe it or not, these equations make some intuitive sense. For example, consider

the update for the mean state estimate μ . The term $\mathbf{F}\mu_t$ is the *predicted* state at $t + 1$, so $\mathbf{H}\mathbf{F}\mu_t$ is the *predicted* observation. Therefore the term $\mathbf{z}_{t+1} - \mathbf{H}\mathbf{F}\mu_t$ represents the error in the predicted observation. This is multiplied by \mathbf{K}_{t+1} to correct the predicted state; therefore \mathbf{K}_{t+1} is a measure of *how seriously to take the new observation relative to the prediction*. As in Equation (15.19), we also have the property that the variance update is independent of the observations. The sequence of values for Σ_t and \mathbf{K}_t can therefore be computed offline, and the actual calculations required during online tracking are quite modest.

To illustrate these equations at work, we have applied them to the problem of tracking an object moving on the X - Y plane. The state variables are $\mathbf{X} = (X, Y, \dot{X}, \dot{Y})^\top$ so \mathbf{F} , Σ_x , \mathbf{H} , and Σ_z are 4×4 matrices. Figure 15.9(a) shows the true trajectory, a series of noisy observations, and the trajectory estimated by Kalman filtering, along with the covariances indicated by the one-standard-deviation contours. The filtering process does a reasonably good job of tracking the actual motion, and, as expected, the variance quickly reaches a fixed point.

As one might expect, one can also derive equations for *smoothing* as well as filtering with linear Gaussian models. The smoothing results are shown in Figure 15.9(b). Notice how the variance in the position estimate is sharply reduced, except at the ends of the trajectory (why?); and that the estimated trajectory is much smoother.



Applicability of Kalman filtering

The Kalman filter and its elaborations are used in a vast array of applications. The “classical” application is in radar tracking of aircraft and missiles. Related applications include acoustic tracking of submarines and ground vehicles and visual tracking of vehicles and people. In a

EXTENDED KALMAN FILTER

slightly more esoteric vein, Kalman filters are used to reconstruct particle trajectories from bubble chamber photographs and ocean currents from satellite surface measurements. The range of application is much larger than just the tracking of motion: any system characterized by continuous state variables and noisy measurements will do. Such systems include pulp mills, chemical plants, nuclear reactors, plant ecosystems, and national economies.

The fact that Kalman filtering can be applied to a system does not mean that the results will be valid or useful. The assumptions made—linear Gaussian transition and sensor models—are very strong. The **extended Kalman filter** or EKF attempts to overcome nonlinearities in the system being modelled. A system is nonlinear if the transition model cannot be described as a matrix multiplication of the state vector, as in Equation (15.20). The EKF works by modelling the system as *locally* linear in \mathbf{x}_t in the region of $\mathbf{x}_t = \mu_t$, the mean of the current state distribution. This works well for smooth, well-behaved systems, and allows the tracker to maintain and update a Gaussian state distribution that is a reasonable approximation to the true posterior.

What does it mean for a system to be “unsmooth” or “poorly behaved”? Technically, this means that there is significant nonlinearity in system response within the region that is “close” (according to the covariance Σ_t) to the current mean μ_t . To understand this in nontechnical terms, consider the example of trying to track a bird as it flies through the jungle. The bird appears to be heading at high speed straight for a tree-trunk. The Kalman filter, whether regular or extended, can only make a Gaussian prediction of the location of the bird, and the mean of this Gaussian will be centered on the trunk, as shown in Figure 15.10(a). A reasonable model of the bird, on the other hand, would predict evasive action to one side or the other, resulting in the prediction shown in Figure 15.10(b). Such a model is highly nonlinear because the bird’s decision varies sharply depending on its precise location relative to the trunk.

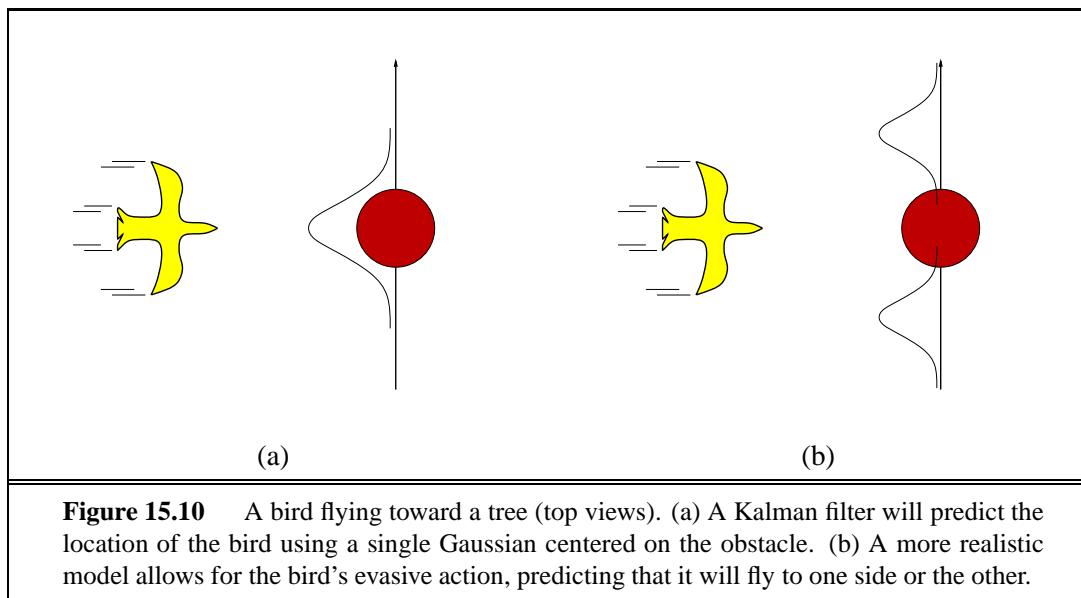


Figure 15.10 A bird flying toward a tree (top views). (a) A Kalman filter will predict the location of the bird using a single Gaussian centered on the obstacle. (b) A more realistic model allows for the bird’s evasive action, predicting that it will fly to one side or the other.

SWITCHING KALMAN FILTER

In order to handle such examples, we clearly need a more expressive language for representing the behavior of the system being modelled. Within the control theory community, where problems such as evasive maneuvering by aircraft raise the same kinds of difficulties, the standard solution is the **switching Kalman filter**. In this approach, multiple Kalman filters run in parallel, each using a different model of the system—for example, one for straight flight, one for sharp left turns, one for sharp right turns. A weighted sum of predictions is used, where the weight depends on how well each filter fits the current data. We will see in the next section that this is simply a special case of the general dynamic Bayesian network model, obtained in this case by adding a discrete “maneuver” state variable to the network shown in Figure 15.7. Switching Kalman filters are discussed further in Exercise 15.5.

15.5 DYNAMIC BAYESIAN NETWORKS

DYNAMIC BAYESIAN NETWORK

A **dynamic Bayesian network** or **DBN** is a Bayesian network that represents a temporal probability model of the kind described in Section 15.1. We have already seen examples of DBNs: the umbrella network in Figure 15.2 and the Kalman filter network in Figure 15.7. In general, each slice of a DBN can have any number of state variables \mathbf{X}_t and evidence variables \mathbf{E}_t . For simplicity, we will assume that the variables and their links are exactly replicated from slice to slice, and that the DBN represents a first-order Markov process, so that each variable can have parents only in its own slice or the immediately preceding slice.



It should be clear that every hidden Markov model can be represented as a DBN with a single state variable and a single evidence variable. It is also the case that every discrete-variable DBN can be represented as an HMM: as explained in Section 15.3, we can combine all the state variables in the DBN into a single state variable whose values are all possible tuples of values of the individual state variables. Now if every HMM is a DBN and every DBN can be translated into an HMM, what’s the difference? The difference is that, *by decomposing the state of a complex system into its constituent variables, the DBN is able to take advantage of sparseness in the temporal probability model*. Suppose, for example, that a DBN has 20 Boolean state variables, each of which has three parents in the preceding slice. Then the DBN transition model has $20 \times 2^3 = 160$ probabilities, whereas the corresponding HMM has 2^{20} states and therefore 2^{40} , or roughly a trillion, probabilities in the transition matrix. This is bad for at least two reasons: first, the HMM itself requires much more space; second, the huge transition matrix makes HMM inference much more expensive; and third, the problem of learning such a huge number of parameters makes the pure HMM model unsuitable for large problems. The relationship between DBNs and HMMs is roughly analogous to the relationship between ordinary Bayesian networks and full tabulated joint distributions.

We have already explained that every Kalman filter model can be represented in a DBN with continuous variables and linear Gaussian conditional distributions (Figure 15.7). It should be clear from the discussion at the end of the preceding section that *not* every DBN can be represented by a Kalman filter model. In a Kalman filter, the current state distribution is always a single multivariate Gaussian distribution—that is, a single “bump” in a particular

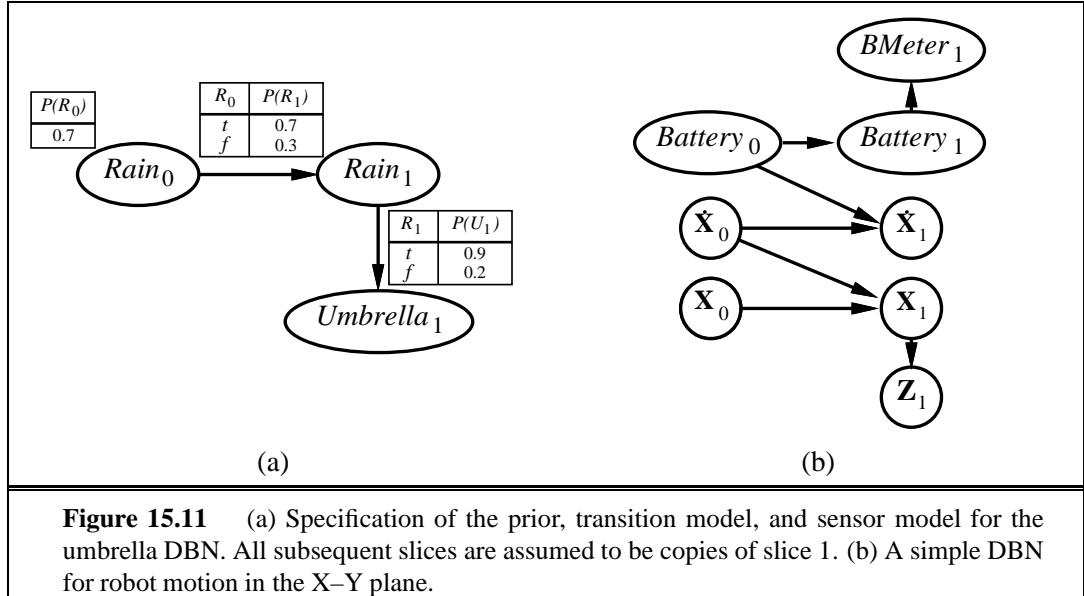


Figure 15.11 (a) Specification of the prior, transition model, and sensor model for the umbrella DBN. All subsequent slices are assumed to be copies of slice 1. (b) A simple DBN for robot motion in the X–Y plane.

location. DBNs, on the other hand, can handle arbitrary distributions. For many real-world applications, this flexibility is essential. Consider, for example, the current location of my keys. They might be in my pocket, on the bedside table, on the kitchen counter, or dangling from the front door. A single Gaussian bump that included all these places would have to allocate significant probability to the keys being in mid-air in the front hall. Aspects of the real world such as purposive agents, obstacles, and pockets introduce “nonlinearities” and “discontinuities” that necessitate complex combinations of discrete and continuous variables in order to get reasonable models.

Constructing DBNs

To construct a DBN, one must specify three kinds of information: the prior distribution over the state variables, $\mathbf{P}(\mathbf{X}_0)$; the transition model $\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{X}_t)$; and the sensor model $\mathbf{P}(\mathbf{E}_t|\mathbf{X}_t)$. To specify the transition and sensor models, one must also specify the topology of the connections between successive slices and between the state and evidence variables. Because the transition and sensor models are assumed to be stationary—i.e., the same for all t —it is most convenient simply to specify them for the first slice. For example, the complete DBN specification for the umbrella world is given by the three-node network shown in Figure 15.11(a). From this specification, the complete (semi-infinite) DBN can be constructed as needed by copying the first slice.

Let us now consider a more interesting example: monitoring a battery-powered robot moving in the X–Y plane, as introduced in Section 15.1. First, we need state variables, which will include both $\mathbf{X}_t = (X_t, Y_t)$ for position and $\dot{\mathbf{X}}_t = (\dot{X}_t, \dot{Y}_t)$ for velocity. We will assume some method of measuring position—perhaps a fixed camera or onboard GPS (Global Positioning System)—yielding measurements \mathbf{Z}_t . The position at the next time step depends

on the current position and velocity, as in the standard Kalman filter model. The velocity at the next step depends on the current velocity and the state of the battery. We add $Battery_t$ to represent the actual battery charge level, which has as parents the previous battery level and the velocity, and we add $BMeter_t$, which measures the battery charge level. This gives us the basic model shown in Figure 15.11(b).

It is worth looking in more depth at the nature of the sensor model for $BMeter_t$. Let us suppose, for simplicity, that both $Battery_t$ and $BMeter_t$ can take on discrete values 0 through 5—rather like the battery meter on a typical laptop computer. If the meter is always accurate, then the CPT $\mathbf{P}(BMeter_t|Battery_t)$ should have probabilities of 1.0 “along the diagonal” and probabilities of 0.0 elsewhere. In reality, noise always creeps into measurements. For continuous measurements, a Gaussian distribution with a small variance might be used instead.⁴ For our discrete variables, we can approximate a Gaussian using a distribution in which the probability of error drops off in the appropriate way, so that the probability of a large error is very small. We will use the term **Gaussian error model** to cover both the continuous and discrete versions.

GAUSSIAN ERROR MODEL

TRANSIENT FAILURE

Anyone with hands-on experience of robotics, computerized process control, or other forms of automatic sensing will readily testify to the fact that small amounts of measurement noise are often the least of one’s problems. Real sensors *fail*. When a sensor fails, it does not necessarily send a signal saying, “Oh, by the way, the data I’m about to send you is a load of nonsense.” Instead, it simply sends the nonsense. The simplest kind of failure is called a **transient failure**, where the sensor occasionally decides to send some nonsense. For example, the battery level sensor might have a habit of sending a zero when someone bumps the robot, even if the battery is fully charged.

Let’s see what happens when a transient failure occurs with a Gaussian error model that doesn’t accommodate such failures. Suppose, for example, that the robot is sitting quietly and observes twenty consecutive battery readings of 5. Then the battery meter has a temporary seizure and the next reading is $BMeter_{21} = 0$. What will the simple Gaussian error model lead us to believe about $Battery_{21}$? According to Bayes’ rule, the answer depends on both the sensor model $\mathbf{P}(BMeter_{21} = 0|Battery_{21})$ and the prediction $\mathbf{P}(Battery_{21}|BMeter_{1:20})$. If the probability of a large sensor error is significantly less likely than the probability of a transition to $Battery_{21} = 0$, even if the latter is very unlikely, then the posterior distribution will assign high probability to the battery being empty. A second reading of zero at $t = 22$ will make this conclusion almost certain. If the transient failure then disappears and the reading returns to 5 from $t = 23$ onwards, the estimate for the battery level will quickly return to 5, as if by magic. This course of events is illustrated in the upper curve of Figure 15.12(a), which shows the expected value of $Battery_t$ over time using a discrete Gaussian error model.

Despite the recovery, there is a time ($t = 22$) when the robot is convinced its battery is empty; presumably, then, it should send out a mayday signal and shut down. Alas, its oversimplified sensor model has led it astray. How can this be fixed? Consider a familiar

⁴ Strictly speaking, a Gaussian distribution is problematic because it assigns nonzero probability to large negative charge levels. The **beta distribution** is sometimes a better choice for a variable whose range is restricted.



example from everyday human driving: on sharp curves or steep hills, one's "fuel tank empty" warning light sometimes turns on. Rather than looking for the emergency phone, one simply recalls that the fuel gauge sometimes gives a very large error when the fuel is sloshing around in the tank. The moral of the story is the following: *in order for the system to handle sensor failure properly, the sensor model must include the possibility of failure.*

The simplest kind of failure model for a sensor allows a certain probability that the sensor will return some completely incorrect value, regardless of the true state of the world. For example, if the battery meter fails by returning 0, we might say that

$$P(BMeter_t = 0 | Battery_t = 5) = 0.03$$

TRANSIENT FAILURE MODEL

which is presumably much larger than the probability assigned by the simple Gaussian error model. Let's call this the **transient failure model**. How does it help when we are faced with a reading of 0? Provided that the *predicted* probability of an empty battery, according to the readings so far, is much less than 0.03, then the best explanation of the observation $BMeter_{21} = 0$ is that the sensor has temporarily failed. Intuitively, we can think of the belief about the battery level as having a certain amount of "inertia" that helps to overcome temporary blips in the meter reading. The upper curve in Figure 15.12(b) shows that the transient failure model can handle transient failures without a catastrophic change in beliefs.

So much for temporary blips. What about a persistent sensor failure? Sadly, failures of this kind are all too common. If the sensor returns 20 readings of 5 followed by 20 readings of 0, then the transient sensor failure model described in the preceding paragraph will result in the robot gradually coming to believe that its battery is empty, when in fact it may be that the meter has failed. The lower curve in Figure 15.12(b) shows the belief "trajectory" for this case. By $t = 25$ —five readings of 0—the robot is convinced that its battery is empty. Obviously, we would prefer the robot to believe that its battery meter is broken—if indeed this is the more likely event.

PERSISTENT FAILURE MODEL

PERSISTENCE ARC

Unsurprisingly, to handle persistent failure we will need a **persistent failure model** that describes how the sensor behaves under normal conditions and after failure. To do this, we need to augment the hidden state of the system with an additional variable, say $BM Broken$, that describes the status of the battery meter. The persistence of failure must be modelled by an arc linking $BM Broken_0$ to $BM Broken_1$. This **persistence arc** has a CPT that gives a small probability of failure in any given time step, say 0.001, but specifies that the sensor stays broken once it breaks. When the sensor is OK, the sensor model for $BMeter$ is identical to the transient failure model; when the sensor is broken, it says $BMeter$ is always 0, regardless of the actual battery charge.

The persistent failure model for the battery sensor is shown in Figure 15.13(a). Its performance on the two data sequences (temporary blip and persistent failure) is shown in Figure 15.13(b). There are several things to notice about these curves. First, in the case of the temporary blip, the probability that the sensor is broken rises significantly after the second 0 reading, but immediately drops back to zero once a 5 is observed. Second, in the case of persistent failure, the probability that the sensor is broken rises quickly to almost 1 and stays there. Finally, once the sensor is known to be broken, the robot can only assume that its battery discharges at the "normal" rate, as shown by the gradually descending level of

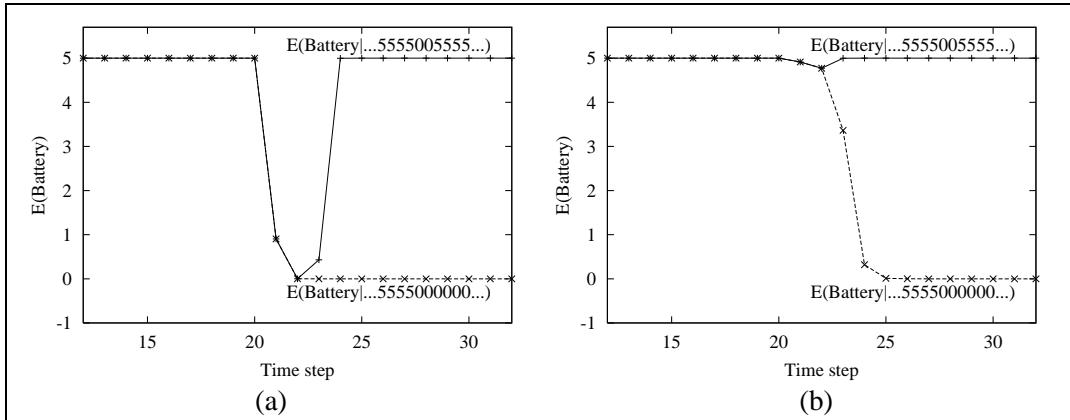
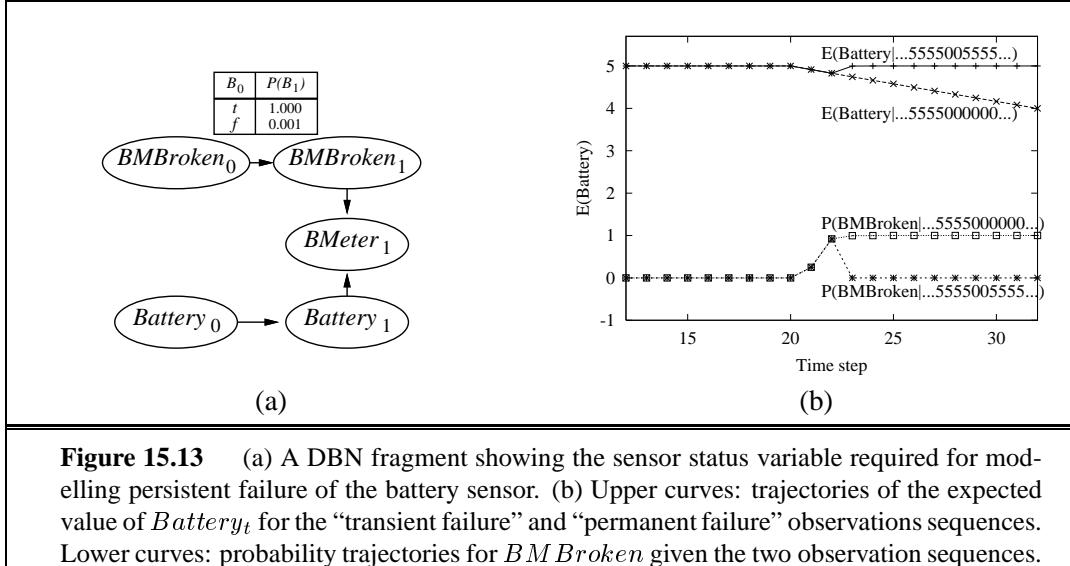


Figure 15.12 (a) Upper curve: trajectory of the expected value of $Battery_t$ for an observation sequence consisting of all 5s except for 0s at $t = 21$ and $t = 22$, using a simple Gaussian error model. Lower curve: trajectory when the observation remains at 0 from $t = 21$ onwards. (b) The same experiment run using the transient failure model. Notice that the transient failure is handled well but the persistent failure results in excessive pessimism.



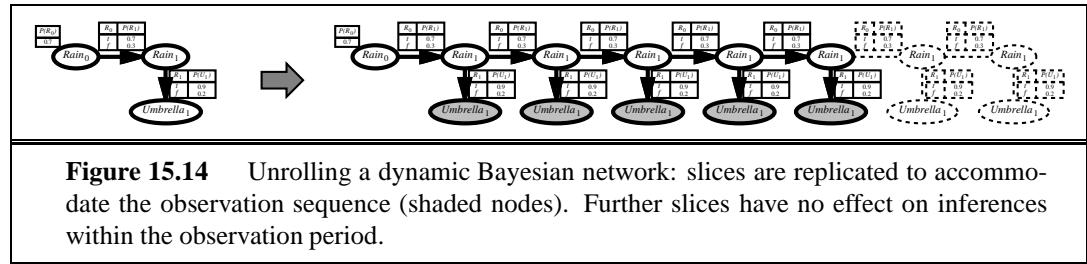
$E(Battery_t | \dots)$. A more refined model would include the influence of the robot's activities on the battery level, which we have so far ignored.

So far, we have only scratched the surface of the problem of representing complex processes. The variety of transition models is huge, encompassing topics as disparate as modelling of the human endocrine system and modelling multiple vehicles driving on a freeway. Sensor modelling is also a vast subfield in itself, but even subtle phenomena, such as sensor drift, sudden decalibration, and the effects of exogenous conditions (such as weather) on sensor readings, can be handled by explicit representation within dynamic Bayesian networks.

UNROLLING

Exact inference in DBNs

Having sketched some ideas for representing complex processes as DBNs, we now turn to the question of inference. In a sense, this question has already been answered: dynamic Bayesian networks *are* Bayesian networks, and we already have algorithms for inference in Bayesian networks. Given a sequence of observations, one can construct the full Bayesian network representation of a DBN by replicating slices until the network is large enough to accommodate the observations, as in Figure 15.14. This is called **unrolling**. (Technically, the DBN is equivalent to the semi-infinite network obtained by unrolling for ever. Slices added beyond the last observation have no effect on inferences within the observation period and can be omitted.) Once the DBN is unrolled, one can use any of the inference algorithms—variable elimination, join-tree methods, and so on—described in Chapter 14.



Unfortunately, a naive application of unrolling would not be particularly efficient. If we want to perform filtering or smoothing with a long sequence of observations $e_{1:t}$, the unrolled network would require $O(t)$ space and thus grows without bound as more observations are added. Moreover, if we simply run the inference algorithm anew each time an observation is added, the inference time per update will also increase as $O(t)$.

Looking back to Section 15.2, we see that constant time and space per filtering update can be achieved if the computation can be done in a recursive fashion. Essentially, the filtering update in Equation (15.4) works by *summing out* the state variables of the previous time step to get the distribution for the new time step. Summing out variables is exactly what the **variable elimination** (Figure 14.10) algorithm does, and it turns out that running variable elimination with the variables in temporal order exactly mimics the operation of the recursive filtering update in Equation (15.4). The modified algorithm keeps at most two slices in memory at any one time: starting with slice 0, we add slice 1, then sum out slice 0, then add slice 2, then sum out slice 1, and so on. In this way, we can achieve constant space and time per filtering update. (The same performance can be achieved by making suitable modifications to the join tree algorithm.) Exercise 15.10 asks you to verify this fact for the umbrella network.

So much for the good news; now for the bad news. It turns out that the “constant” for the per-update time and space complexity is, in almost all cases, exponential in the number of state variables. What happens is that as the variable elimination proceeds, the factors grow to include all the state variables (or, more precisely, all those state variables that have parents in the previous time slice). The maximum factor size is $O(d^{n+1})$ and the update cost is $O(d^{n+2})$.



This is much less than the cost of HMM updating, which is $O(d^{2n})$, but it is still infeasible for large numbers of variables. This grim fact is somewhat hard to accept. What it means is that *even though we can use DBNs to represent very complex temporal processes with many sparsely connected variables, we cannot reason efficiently and exactly about those processes.* The DBN model itself, which represents the prior joint distribution over all the variables, is factorable into its constituent CPTs, but the posterior joint distribution conditioned on an observation sequence—that is, the forward message—is generally *not* factorable. So far, no-one has found a way around this problem, despite the fact that many important areas of science and engineering would benefit enormously from its solution. Thus, we must fall back on approximate methods.

Approximate inference in DBNs

Chapter 14 described two approximation algorithms: likelihood weighting (Figure 14.14) and Markov chain Monte Carlo (MCMC, Figure 14.15). Of the two, the former is most easily adapted to the DBN context. We will see, however, that several improvements are required over the standard likelihood weighting algorithm before a practical method emerges.



Recall that likelihood weighting works by sampling the nonevidence nodes of the network in topological order, weighting each sample by the likelihood it accords to the observed evidence variables. As with the exact algorithms, we could apply likelihood weighting directly to an unrolled DBN, but this would suffer from the same problems in terms of increasing time and space requirements per update as the observation sequence grows. The problem is that the standard algorithm runs each sample in turn all the way through the network. Instead, we can simply run all N samples together through the DBN one slice at a time. The modified algorithm fits the general pattern of filtering algorithms, with the set of N samples as the forward message. The first key innovation, then, is to *use the samples themselves as an approximate representation of the current state distribution.* This meets the requirement of a “constant” time per update, although the constant depends on the number of samples required to maintain a reasonable approximation to the true posterior distribution. There is also no need to unroll the DBN, because we need only the current slice and the next slice in memory.

In our discussion of likelihood weighting in Chapter 14, we pointed out that the algorithm’s accuracy suffers if the evidence variables are “downstream” of the variables being sampled, because in that case the samples are generated without any influence from the evidence. Looking at the typical structure of a DBN—say, the umbrella DBN in Figure 15.14—we see that indeed the early state variables will be sampled without the benefit of the later evidence. In fact, looking more carefully, we see that *none* of the state variables has *any* evidence variables among its ancestors! Hence, although the weight of each sample will depend on the evidence, the actual set of samples generated will be *completely independent* of the evidence. For example, even if the boss brings in the umbrella every day, the sampling process may still hallucinate endless days of sunshine. What this means in practice is that the fraction of samples that remain reasonably close to the actual series of events drops exponentially with t , the length of the observation sequence; in other words, to maintain a given level

of accuracy, we need to increase the number of samples exponentially with t . Figure 15.15(a) shows some experimental results for likelihood weighting applied to the umbrella network. Clearly we need a better solution.

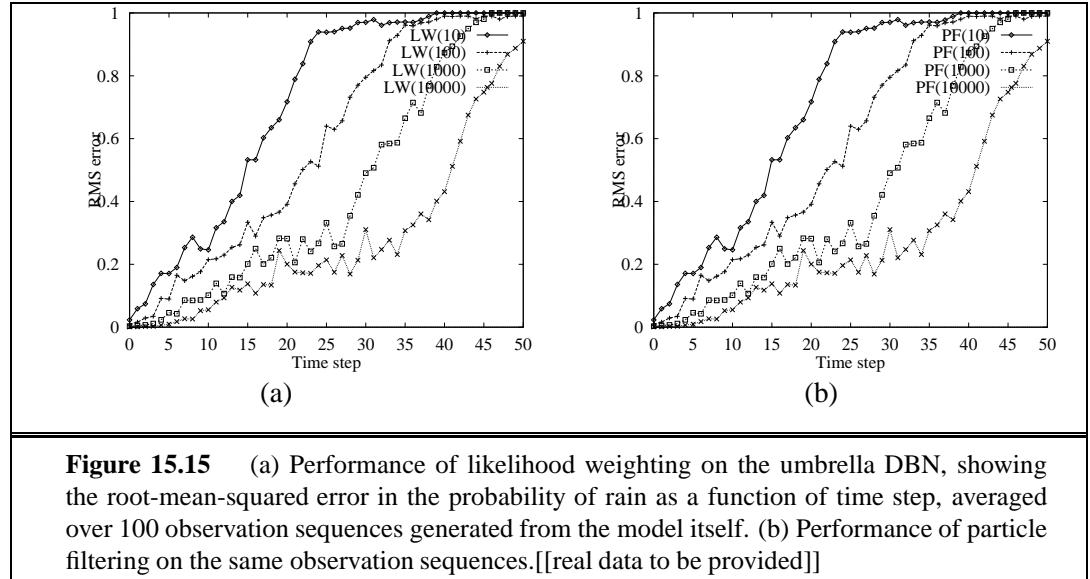


Figure 15.15 (a) Performance of likelihood weighting on the umbrella DBN, showing the root-mean-squared error in the probability of rain as a function of time step, averaged over 100 observation sequences generated from the model itself. (b) Performance of particle filtering on the same observation sequences. [[real data to be provided]]



PARTICLE FILTERING

The second key innovation is to *focus the set of samples on the high-probability regions of the state space*. This can be done by throwing away samples that have very low weight, according to the observations, while multiplying those that have high weight. In this way, the population of samples will stay reasonably close to reality. If we think of samples as a resource for modelling the posterior distribution, then it makes sense to use more samples in regions of the state space where the posterior is higher.

A family of algorithms called **particle filtering** is designed to do just this. Particle filtering works as follows. First, a population of N samples is created by sampling from the prior distribution at time 0, $\mathbf{P}(\mathbf{X}_0)$. Then the update cycle is repeated for each time step:

- Each sample is propagated forward by sampling the next state value \mathbf{x}_{t+1} given the current value \mathbf{x}_t for the sample, using the transition model $\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{x}_t)$.
- Each sample is weighted by the likelihood it assigns to the new evidence, $P(\mathbf{e}_{t+1}|\mathbf{x}_{t+1})$.
- The population is *resampled* to generate a new population of N samples. Each new sample is selected from the current population; the probability that a particular sample is selected is proportional to its weight. The new samples are unweighted.

The algorithm is shown in detail in Figure 15.16, and its operation for the umbrella DBN is illustrated in Figure 15.17.

We can show that this algorithm is consistent—gives the correct probabilities as N tends to infinity—by considering what happens during one update cycle. We will assume the sample population starts with a correct representation of the forward message $\mathbf{f}_{1:t}$ at time t :

```

function PARTICLEFILTERING(e,N,dbn) returns a set of samples for the next time step
  inputs: e, the new incoming evidence
    N, the number of samples to be maintained
    dbn, a DBN with slice 0 variables X0 and slice 1 variables X1 and E1
  static: S, a vector of samples of size N
  local variables: W, a vector of weights of size N

  if e is empty then /* initialization phase */
    for i = 1 to N do
      S[i]  $\leftarrow$  sample from P(X0)
  else do /* update cycle */
    for i = 1 to N do
      S[i]  $\leftarrow$  sample from P(X1|X0 = S[i])
      W[i]  $\leftarrow$  P(e|X1 = S[i])
  S  $\leftarrow$  WEIGHTEDSAMPLEWITHREPLACEMENT(N,S,W)
  return S

```

Figure 15.16 The particle filtering algorithm implemented as a recursive update operation with state (the set of samples). Each of the sampling steps involves sampling the relevant slice variables in topological order, much as in PRIOR-SAMPLE. The WEIGHTED-SAMPLE-WITH-REPLACEMENT operation can be implemented to run in $O(N)$ expected time.

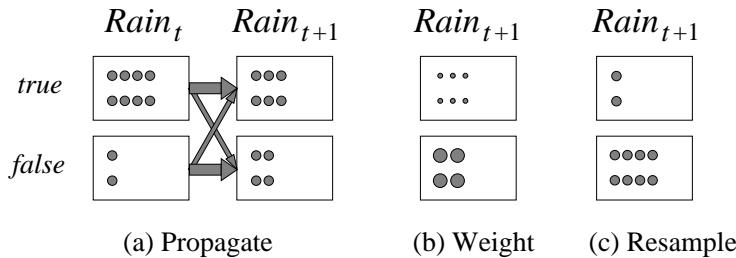


Figure 15.17 The particle filtering update cycle for the umbrella DBN with $N = 10$, showing the sample populations of each state. (a) At time t , 8 samples indicate *Rain* and 2 indicate \neg *Rain*. Each is propagated forward by sampling the next state using the transition model. At time $t + 1$, 7 samples indicate *Rain* and 3 indicate \neg *Rain*. (b) \neg *Umbrella* is observed at $t + 1$. Each sample is weighted by its likelihood for the observation, as indicated by the size of the circles. (c) A new set of 10 samples is generated by weighted random selection from the current set, resulting in 4 samples that indicate *Rain* and 6 that indicate \neg *Rain*.

writing $N(\mathbf{x}_t | \mathbf{e}_{1:t})$ for the number of samples occupying state \mathbf{x}_t after observations $\mathbf{e}_{1:t}$ have been processed, we therefore have

$$N(\mathbf{x}_t | \mathbf{e}_{1:t})/N = P(\mathbf{x}_t | \mathbf{e}_{1:t}) \quad (15.22)$$

for large N . Now we propagate each sample forward by sampling the state variables at $t + 1$ given the values for the sample at t . The number of samples reaching state \mathbf{x}_{t+1} from each

\mathbf{x}_t is the transition probability times the population of \mathbf{x}_t ; hence the total number of samples reaching \mathbf{x}_{t+1} is

$$N(\mathbf{x}_{t+1}|\mathbf{e}_{1:t}) = \sum_{\mathbf{x}_t} P(\mathbf{x}_{t+1}|\mathbf{x}_t) N(\mathbf{x}_t|\mathbf{e}_{1:t})$$

Now we weight each sample by its likelihood for the evidence at $t + 1$. A sample in state \mathbf{x}_{t+1} receives weight $P(\mathbf{e}_{t+1}|\mathbf{x}_{t+1})$. The total weight of the samples in \mathbf{x}_{t+1} after seeing \mathbf{e}_{t+1} is therefore

$$W(\mathbf{x}_{t+1}|\mathbf{e}_{1:t+1}) = P(\mathbf{e}_{t+1}|\mathbf{x}_{t+1})N(\mathbf{x}_{t+1}|\mathbf{e}_{1:t})$$

Now for the resampling step. Since each sample is replicated with probability proportional to its weight, the number of samples in state \mathbf{x}_{t+1} after resampling is proportional to the total weight in \mathbf{x}_{t+1} before resampling:

$$\begin{aligned} N(\mathbf{x}_{t+1}|\mathbf{e}_{1:t+1})/N &= \alpha W(\mathbf{x}_{t+1}|\mathbf{e}_{1:t+1}) \\ &= \alpha P(\mathbf{e}_{t+1}|\mathbf{x}_{t+1})N(\mathbf{x}_{t+1}|\mathbf{e}_{1:t}) \\ &= \alpha P(\mathbf{e}_{t+1}|\mathbf{x}_{t+1}) \sum_{\mathbf{x}_t} P(\mathbf{x}_{t+1}|\mathbf{x}_t) N(\mathbf{x}_t|\mathbf{e}_{1:t}) \\ &= \alpha NP(\mathbf{e}_{t+1}|\mathbf{x}_{t+1}) \sum_{\mathbf{x}_t} P(\mathbf{x}_{t+1}|\mathbf{x}_t)P(\mathbf{x}_t|\mathbf{e}_{1:t}) \quad \text{by Equation (15.22)} \\ &= \alpha' P(\mathbf{e}_{t+1}|\mathbf{x}_{t+1}) \sum_{\mathbf{x}_t} P(\mathbf{x}_{t+1}|\mathbf{x}_t)P(\mathbf{x}_t|\mathbf{e}_{1:t}) \\ &= P(\mathbf{x}_{t+1}|\mathbf{e}_{1:t+1}) \quad \text{by Equation (15.4)} \end{aligned}$$

Therefore the sample population after one update cycle correctly represents the forward message at time $t + 1$.

Particle filtering is *consistent*, therefore, but is it *efficient*? In practice, it seems the answer is yes—particle filtering seems to maintain a good approximation to the true posterior using a constant number of samples. There are, as yet, no theoretical guarantees; particle filtering is currently an area of intensive study. Many variants and improvements have been proposed and the set of applications is growing rapidly. Because it is a sampling algorithm, particle filtering can be used easily with hybrid and continuous DBNs, allowing it to be applied to areas such as tracking complex motion patterns in video (Isard and Blake, 1996) and predicting the stock market (de Freitas *et al.*, 1999).

15.6 SPEECH RECOGNITION

SPEECH
RECOGNITION

In this section, we look at one of the most important applications of temporal probability models—**speech recognition**. The task is to identify the sequence of words uttered by a speaker, given the acoustic signal. Speech is the dominant modality for communication between humans, and reliable speech recognition by machines would be immensely useful. Still more useful would be **speech understanding**—the identification of the *meaning* of the utterance. For this, we must wait until Chapter 22.

Speech provides our first contact with the raw, unwashed world of real sensor data. These data are *noisy*, quite literally; there can be background noise as well as artifacts introduced by the digitization process; there is variation in the way that words are pronounced, even by the same speaker; different words can sound the same; and so on. For these reasons, speech recognition has come to be viewed as a problem of probabilistic inference.

At the most general level, we can define the probabilistic inference problem as follows. Let $Words$ be a random variable ranging over all possible sequences of words that might be uttered, and let $signal$ be the observed acoustic signal sequence. Then the most likely interpretation of the utterance is the value of $Words$ that maximizes $P(words|signal)$. As is often the case, applying Bayes' rule is helpful:

$$P(words|signal) = \alpha P(signal|words)P(words)$$

ACOUSTIC MODEL $P(signal|words)$ is the **acoustic model**. It describes the sounds of words—for example, that “ceiling” begins with a soft “c” and sounds very similar to “sealing”. (Words that sound the same are called **homophones**.) $P(words)$ is known as the **language model**. It specifies the prior probability of each utterance—for example, that “high ceiling” is a much more likely word sequence than “high sealing.”

BIGRAM MODEL The language models used in speech recognition systems are usually very simple. The **bigram model** that we describe later in this section gives the probability of each word following each other word. The acoustic model is much more complex. At its heart is an important discovery made in the field of **phonology** (the study of how language sounds), namely, that all human languages use a limited repertoire of about 40 or 50 sounds, called **phones**. Roughly speaking, a phone is the sound that corresponds to a single vowel or consonant, but there are some complications: combinations of letters such as “th” and “ng” produce single phones, and some letters produce different phones in different contexts (for example, the “a” in *rat* and *rate*). Figure 15.18 lists all the phones in English with an example of each.

PHONOLOGY **PHONES** The existence of phones makes it possible to divide the acoustic model into two parts. The first part deals with **pronunciation** and specifies, for each word, a probability distribution over possible phone sequences. For example, “ceiling” is pronounced [s iy l ih ng], or sometimes [s iy l ix ng], or sometimes even [s iy l en]. The phones are not directly observable, so, roughly speaking, speech is represented as a hidden Markov model whose state variable X_t specifies which phone is being uttered at time t .

PRONUNCIATION **SIGNAL PROCESSING** The second part of the acoustic model deals with the way that phones are realized as acoustic signals—that is, the evidence variable E_t for the hidden Markov model gives the observed features of the acoustic signal at time t , and the acoustic model specifies $P(E_t|X_t)$, where X_t is the current phone. This model must allow for variations in pitch, speed, and volume, and relies on techniques from **signal processing** to provide signal descriptions that are reasonably robust against these kinds of variations.

The remainder of the section describes the models and algorithms from the bottom up, beginning with acoustic signals and phones, then individual words, and finally entire sentences. We conclude with a description of how all these models are trained and how well the resulting systems work.

Vowels		Consonants B-N		Consonants P-Z	
Phone	Example	Phone	Example	Phone	Example
[iy]	<u>beat</u>	[b]	<u>bet</u>	[p]	<u>pet</u>
[ih]	<u>bit</u>	[ch]	<u>Chet</u>	[r]	<u>rat</u>
[eh]	<u>bet</u>	[d]	<u>debt</u>	[s]	<u>set</u>
[æ]	<u>bat</u>	[f]	<u>fat</u>	[sh]	<u>shoe</u>
[ah]	<u>but</u>	[g]	<u>get</u>	[t]	<u>ten</u>
[ao]	<u>bought</u>	[hh]	<u>hat</u>	[th]	<u>thick</u>
[ow]	<u>boat</u>	[hv]	<u>high</u>	[dh]	<u>that</u>
[uh]	<u>book</u>	[jh]	<u>jet</u>	[dx]	<u>butter</u>
[ey]	<u>bait</u>	[k]	<u>kick</u>	[v]	<u>vet</u>
[er]	<u>Bert</u>	[l]	<u>let</u>	[w]	<u>wet</u>
[ay]	<u>buy</u>	[el]	<u>bottle</u>	[wh]	<u>which</u>
[oy]	<u>boy</u>	[m]	<u>met</u>	[y]	<u>yet</u>
[axr]	<u>diner</u>	[em]	<u>bottom</u>	[z]	<u>zoo</u>
[aw]	<u>down</u>	[n]	<u>net</u>	[zh]	<u>measure</u>
[ax]	<u>about</u>	[en]	<u>button</u>		
[ix]	<u>roses</u>	[ng]	<u>sing</u>		
[aa]	<u>cot</u>	[eng]	<u>Washington</u>	[-]	(silence)

Figure 15.18 The DARPA phonetic alphabet, or ARPAbet, listing all the phones used in American English. There are several alternative notations, including an International Phonetic Alphabet (IPA), which contains the phones in all known languages.

Speech sounds

SAMPLING RATE

QUANTIZATION FACTOR

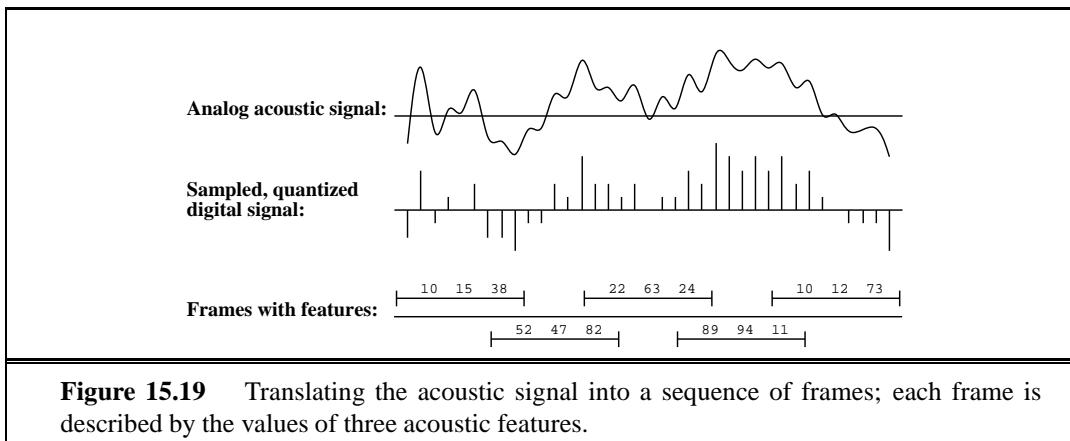
FRAMES

FEATURES

Sound waves are periodic changes in pressure that propagate through the air. Sound can be measured by a microphone whose diaphragm is displaced by the pressure changes and generates a continuously varying current. An analog-to-digital converter measures the size of the current—which corresponds to the amplitude of the sound wave—at discrete intervals determined by the **sampling rate**. For speech, a sampling rate between 8 and 16 kHz (i.e., 8 to 16,000 times per second) is typical. The precision of each measurement is determined by the **quantization factor**; speech recognizers typically keep 8 to 12 bits. That means that a low-end system, sampling at 8 kHz with 8-bit quantization, would require nearly half a megabyte per minute of speech. It would be impractical to construct and manipulate $P(\text{signal}|\text{phone})$ distributions with so much signal information; therefore, we need to develop more concise descriptions of the signal.

First, we observe that although the sound frequencies in speech may be several kHz, the *changes* in the content of the signal occur much less often, perhaps at no more than 100 Hz. Therefore, speech systems summarize the properties of the signal over extended intervals called **frames**. A frame length of about 10 msec (i.e., 80 samples at 8 kHz) is short enough to ensure that few short-duration phenomena will be smudged out by the summarization process. Within each frame, we represent what is happening with a vector of **features**. For

example, we might want to characterize the amount of energy at each of several frequency ranges. Other important features include overall energy in a frame, and the difference from the previous frame. Picking out features from a speech signal is like listening to an orchestra and saying “here the French horns are playing loudly and the violins are playing softly.” Figure 15.19 shows the sequence of transformations from the raw sound to a sequence of frames. Note that the frames overlap; this prevents us from losing information if an important acoustic event just happens to fall on a frame boundary.



In our example, we have shown frames with just three features. Real systems may have tens or even hundreds of features. If there are n features and each has, say, 256 possible values, then a frame is described by a point in n -dimensional space and there are 256^n possible frames. For $n > 2$ it would be impractical to represent the distribution $P(\text{features}|\text{phone})$ as an explicit table, so we need further compression. There are two possible approaches:

VECTOR QUANTIZATION

- The method of **vector quantization** or VQ divides the n -dimensional space into, say, 256 regions labelled C1 through C256. Each frame can then be represented with a single label rather than a vector of n numbers. Thus, the tabulated distribution $P(VQ \text{ label}|\text{phone})$ has 256 probabilities specified for each phone. Vector quantization is no longer popular in large-scale systems.
- Instead of discretizing the feature space, we can use a parameterized continuous distribution to describe $P(\text{features}|\text{phone})$. For example, we could use a Gaussian distribution with a different mean and covariance matrix for each phone. This works well if the acoustic realizations of each phone are clustered in a single region of feature space. In practice, the sounds can be spread over several regions, and a **mixture of Gaussians** must be used. A mixture is a weighted sum of k individual distributions, so $P(\text{features}|\text{phone})$ has k weights, k mean vectors of size n , and k covariance matrices of size n^2 —that is, $O(kn^2)$ parameters for each phone.

MIXTURE OF GAUSSIANS

Of course, some information is lost in going from the full speech signal to a VQ label or a set of mixture parameters. The art of signal processing lies in choosing features and regions (or Gaussians) so that the loss of *useful* information is minimized. A given speech sound

can be pronounced so many ways: loud or soft, fast or slow, high-pitched or low, against a background of silence or noise, and by any of millions of different speakers each with different accents and vocal tracts. Signal processing hopes to eliminate the variations while keeping the commonalities that define the sound.

STOP CONSONANTS

THREE-STATE PHONE

TRIPHONE

There are two more refinements we need to make to the simple model we have described so far. The first deals with the temporal structure of phones. In normal speech, most phones have a duration of 50-100 milliseconds, or 5-10 frames. The probability model $P(features|phone)$ is the same for all these frames, whereas most phones have a good deal of internal structure. For example, [t] is one of several **stop consonants** in which the flow of air is cut off for a short period before a sharp release. Examining the acoustic signal, we find that [t] has a silent beginning, a small explosion in the middle, and (usually) a hissing at the end. This internal structure of phones can be captured by the **three-state phone** model; each phone has Onset, Mid, and End states, and each state has its own distribution over features.

The second refinement deals with the context in which the phone is uttered. The sound of a given phone can change depending on the surrounding phones.⁵ For example, the [t] in “tar” has a short hiss at the end, prior to the voiced [aa r], whereas the [t] in “star” does not. Both of these [t] sounds are produced by closing the tongue against the roof of the mouth just behind the teeth, whereas the [t] in “eighth” is often produced with the tongue pressed against the front teeth because it is followed immediately by a [th] sound. These contextual effects are partially captured by the **triphone** model, in which the acoustic model for each phone is allowed to depend on the preceding and succeeding phones. Thus, the [t] in “star” is written [t(s,aa)], i.e., [t] with left-context [s] and right-context [aa].

The combined effect of the three-state and triphone models is to increase the number of possible states of the temporal process from n phones in the original phone alphabet ($n \approx 50$ for the ARPAbet) to $3n^3$. Experience shows that the improved accuracy more than offsets the extra expense in terms of inference and learning.

Words

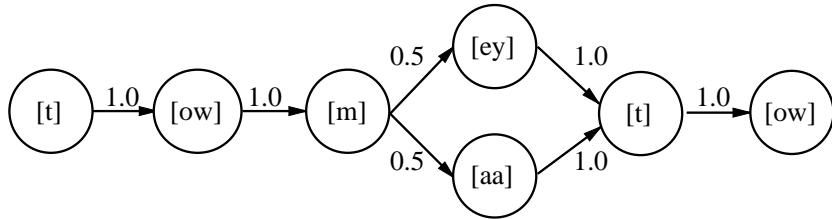
We can think of each word as specifying a distinct probability distribution $\mathbf{P}(X_{1:t}|word)$, where X_i specifies the phone state in the i th frame. Typically, we separate this distribution into two parts. The **pronunciation model** gives a distribution over phone sequences (ignoring metric time and frames), while the **phone model** describes how a phone maps into a sequence of frames.

Consider the word “tomato.” It is well-known that you say [t ow m ey t ow] and I say [t ow m aa t ow], so the pronunciation model has to account for dialects. The top of Figure 15.20 shows a transition model that provides for this variation. There are only two possible paths through the model, one corresponding to the phone sequence [t ow m ey t ow] and the other to [t ow m aa t ow]. The probability of a path is the product of the probabilities on the arcs that make up the path:

$$P([towmeytow]|“tomato”) = P([towmaatow]|“tomato”) = 0.5$$

⁵ In this sense, the “phone model” of speech should be thought of as a useful approximation rather than an immutable law.

(a) Word model with dialect variation:



(b) Word model with coarticulation and dialect variations:

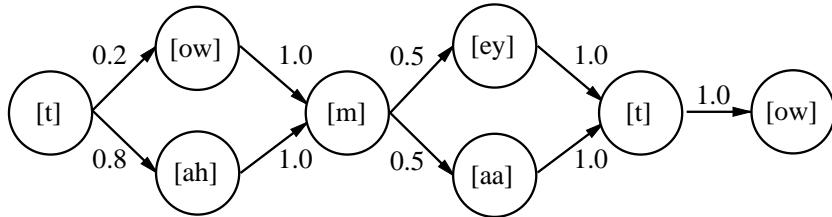


Figure 15.20 Two pronunciation models of the word “tomato.” Each model is shown as a transition diagram with states as circles and arrows showing allowed transitions with their associated probabilities. (a) A model allowing for dialect differences. The 0.5 numbers are estimates based on the two authors’ preferred pronunciations. (b) A model with a coarticulation effect on the first vowel, allowing either the [ow] or the [ah] phone.

COARTICULATION

The second source of phonetic variation is **coarticulation**. Remember that speech sounds are produced by moving the tongue and jaw and forcing air through the vocal tract. When the speaker is talking slowly and deliberately, there is time to place the tongue in just the right spot before producing a phone. But when the speaker is talking quickly (or sometimes even at a normal pace), the movements slur together. For example, the [t] phone is produced with the tongue at the top of the mouth, whereas the [ow] has the tongue near the bottom. When spoken quickly, the tongue often goes to an intermediate position, and we get [t ah] rather than [t ow]. The bottom half of Figure 15.20 gives a more complicated pronunciation model for “tomato” that takes this coarticulation effect into account. In this model there are four distinct paths and we have

$$\begin{aligned} P([\text{towmeytow}] | \text{“tomato”}) &= P([\text{towmaatow}] | \text{“tomato”}) = 0.1 \\ P([\text{tahmeytow}] | \text{“tomato”}) &= P([\text{tahmaatow}] | \text{“tomato”}) = 0.4 \end{aligned}$$

Similar models can be constructed for every word we want to be able to recognize.

The model for a three-state phone is shown as a state transition diagram in Figure 15.21. The model is for a particular phone, [m], but all phones will have models with similar topology. For each phone state, we show the associated acoustic model assuming that the signal is represented by a VQ label. For example, the model asserts that $P(E_t = C_1 | X_t = [\text{m}]_{\text{Onset}}) = 0.5$. Notice the self-loops in the figure; for example, the $[\text{m}]_{\text{Mid}}$ state persists with probability 0.9. This means that the $[\text{m}]_{\text{Mid}}$ state has an expected duration of 10 frames. In this way,

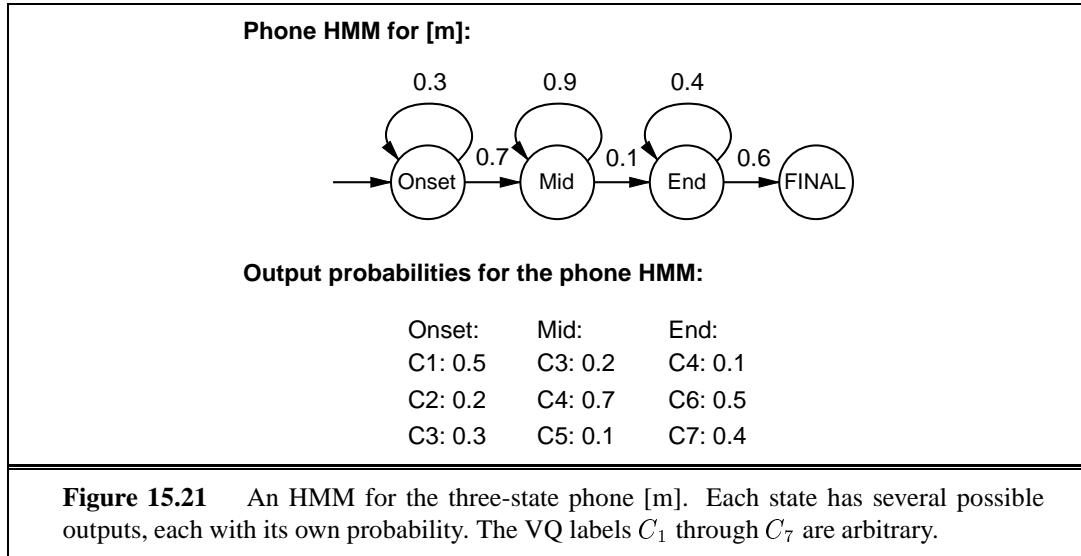


Figure 15.21 An HMM for the three-state phone [m]. Each state has several possible outputs, each with its own probability. The VQ labels C_1 through C_7 are arbitrary.

we can specify the relative durations of phones; of course, the probabilistic model allows for variations, such as arise with fast and slow speech.

We can construct similar models for each phone, possibly depending on the triphone context. Each word model, when combined with the phone models, gives a complete specification of an HMM. The model specifies the transition probabilities between phone states from frame to frame, as well as the acoustic feature probabilities for each phone state.

If we want to recognize **isolated words**—that is, words spoken without any surrounding context and with clear boundaries—then we need to find the word that maximizes

$$P(\text{word}|e_{1:t}) = \alpha P(e_{1:t}|\text{word})P(\text{word})$$

The prior probability $P(\text{word})$ can be obtained from actual text data, as described later. The quantity $P(e_{1:t}|\text{word})$ is the likelihood of the sequence of acoustic features according to the word model. Section 15.2 covered the computation of such likelihoods; in particular, Equation (15.6) gives a simple recursive computation whose cost is linear in t and in the number of states of the Markov chain. To find the most likely word, we can perform this calculation for each possible word model, multiply by the prior, and select the best word accordingly.

Sentences

ISOLATED WORDS
CONTINUOUS SPEECH

To have a conversation with a human, a machine needs to be able to recognize **continuous speech** rather than just isolated words. One might think that continuous speech is nothing more than a sequence of words, to each of which we can apply the algorithm from the previous section. This approach fails for two reasons. First, we have already seen (page 552) that the sequence of most likely words is not the most likely sequence of words. For example, in the movie *Take the Money and Run*, a bank teller interprets Woody Allen's sloppily written hold-up note as saying “I have a gub.” A good language model would suggest “I have a gun” as a much more likely sequence, even though the last word looks more like “gub” than “gun”. The

Word	Unigram count	Previous words							
		of	in	is	on	to	from	model	agent
the	33508	3833	2479	832	944	1365	597	28	24
on	2573	1	0	33	2	1	0	0	6
of	15474	0	0	29	1	0	0	88	7
to	11527	0	4	450	21	4	16	9	82
is	10566	3	6	1	4	2	1	47	127
model	752	8	1	0	1	14	0	6	4
agent	2100	10	3	3	2	3	0	0	36
idea	241	0	0	0	0	0	0	0	0

Figure 15.22 A partial table of unigram and bigram counts for the words in this book. There are 513,893 total words; “the” is the most common at 33,508. The bigram “of the” is the most common at 15,474. That is, one out of every 15 words is “the” and one out of every 33 word pairs is “of the.” Some counts are higher than expected (e.g. 4 for “on is”) because the bigram counts ignore punctuation—one sentence might end with “on” and the next begin with “is.”

SEGMENTATION

second issue we must face with continuous speech is **segmentation**, the problem of deciding where one word ends and the next begins. Anyone who has tried to learn a foreign language will appreciate this problem: at first all the words seem to run together. Gradually, one learns to pick out words from the jumble of sounds. In this case, first impressions are correct; a spectrographic analysis shows that in fluent speech, the words really *do* run together with no silence between them. We learn to identify word boundaries despite the lack of silence.

Let us begin with the language model, whose job in speech recognition is to specify the probability of each possible sequence of words. Using the notation $w_1 \dots w_n$ to denote a string of n words and w_i to denote the i th word of the string, we can write an expression for the probability of a string using the chain rule as follows:⁶

$$\begin{aligned} P(w_1 \dots w_n) &= P(w_1) P(w_2|w_1) P(w_3|w_1 w_2) \dots P(w_n|w_1 \dots w_{n-1}) \\ &= \prod_{i=1}^n P(w_i|w_1 \dots w_{i-1}) \end{aligned}$$

BIGRAM

Most of these terms are quite complex and difficult to estimate or compute. Fortunately, we can approximate this formula with something simpler and still capture a large part of the language model. One simple, popular, and effective approach is the **bigram** model. This model approximates $P(w_i|w_1 \dots w_{i-1})$ with $P(w_i|w_{i-1})$. In other words, it makes a first-order Markov assumption for word sequences.

A big advantage of the bigram model is that it is easy to train the model by counting the number of times each word pair occurs in a representative corpus of strings and using the counts to estimate the probabilities. For example, if “a” appears 10,000 times in the training

⁶ Strictly speaking, the probability of a word sequence depends strongly on the *context* of the utterance; for example, “I have a gun” is much more common on notes passed to a bank teller than it is in, say, the Wall Street Journal. Few speech recognizers handle context, other than by training a special-purpose language model for a particular task.

corpus and it is followed by “gun” 37 times, then $\hat{P}(gun_i|a_{i-1}) = 37/10,000$, where by \hat{P} we mean the estimated probability. After such training one would expect “I have” and “a gun” to have relatively high estimated probabilities, while “I has” and “an gun” would have low probabilities. Figure 15.22 shows some bigram counts derived from the words in this chapter.

TRIGRAM

It is possible to go to a **trigram** model that provides values for $P(w_i|w_{i-1}w_{i-2})$. This is a more powerful language model, capable of determining that “ate a banana” is more likely than “ate a bandana.” For trigram models, and to a lesser extent for bigram and unigram models, there is a problem with counts of zero. We wouldn’t want to say that a combination of words that didn’t happen to appear in the training corpus is improbable. The process of **smoothing** gives a small non-zero probability to such combinations. It is discussed on page 817.

Bigram or trigram models are not as sophisticated as some of the grammar models we will see in Chapters 22 and 23, but they account for local context-sensitive effects better, and manage to capture some local syntax. For example, the fact that the word pairs “I has” and “man have” get low scores is reflective of subject-verb agreement. The problem is that these relationships can only be detected locally: “the man have” gets a low score, but “the man over there have” is not penalized.

Now we consider how to combine the language model with the word models, so that we can handle word sequences properly. We’ll assume a bigram language model for simplicity. With such a model, we can combine all the word models (which are comprised in turn of pronunciation models and phone models) into one large HMM model. A state in a single-word HMM is a frame labelled by the current phone and phone state (for example, $[m]_{\text{Onset}}$); a state in a continuous-speech HMM is also labelled with a word, as in $[m]_{\text{Onset}}^{\text{tomato}}$. If each word has an average of p three-state phones in its pronunciation model, and there are W words, then the continuous-speech HMM has $3Wp$ states. Transitions can occur between phone states within a given phone; between phones in a given word, and between the final state of one word and the initial state of another. The transitions between words occur with probabilities specified by the bigram model.

Once we have constructed the combined HMM, we can use it to analyze the continuous speech signal. In particular, the Viterbi algorithm embodied in Equation (15.10) can be used to find the most likely state sequence. From this state sequence, we can extract a word sequence simply by reading the word labels from the states. Thus, the Viterbi algorithm solves the word segmentation problem by using dynamic programming to consider (in effect) all possible word sequences and word boundaries simultaneously.

Notice that we didn’t say “we can extract *the most likely* word sequence.” The most likely word sequence is not necessarily the one that contains the most likely state sequence. This is because the probability of a word sequence is the sum of probabilities over all possible state sequences consistent with that word sequence. Comparing two word sequences, say “a back” and “aback,” it might be that case that there are ten alternative state sequences for “a back,” each with probability 0.03, but just one state sequence for “aback,” with probability 0.20. Viterbi chooses “aback,” but “a back” is actually more likely.

In practice, this difficulty is not life-threatening, but it is serious enough that other

A* DECODER

approaches have been tried. The most common is the **A* decoder**, which makes ingenious use of A* search (see Chapter 4) to find the most likely word sequence. The idea is to view each word sequence as a path through a graph whose nodes are labelled with words. The successors of a node are all the words that can come next; thus, the graph for all sentences of length n or less has n layers, each of width at most W , where W is the number of possible words. With a bigram model, the cost $g(w_1, w_2)$ of an arc between nodes label w_1 to w_2 is given by $-\log P(w_2|w_1)$; in this way, the total path cost of a sequence $w_1 \dots w_n$ is

$$\sum_{i=1}^n -\log P(w_i|w_{i-1}) = -\log \prod_{i=1}^n P(w_i|w_{i-1}).$$

With this definition of path cost, finding the shortest path is exactly equivalent to finding the most likely word sequence. For the process to be efficient, we also need a good heuristic $h(w_i)$ to estimate the cost of completing the word sequence. Obviously, this has something to do with how much of the speech signal is not yet covered by the words on the current path. As yet, no especially interesting heuristics have been devised for this problem.

Building a speech recognizer

The quality of a speech recognition system depends on the quality of all its components—the language model, the word pronunciation models, the phone models, and the signal processing algorithms used to extract spectral features from the acoustic signal. We have discussed how the language model may be constructed, and we leave the details of signal processing to other textbooks. That leaves the pronunciation and phone models. The *structure* of the pronunciation models—such as the tomato models in Figure 15.20—is usually developed by hand. Large pronunciation dictionaries are now available for English and other languages, although their accuracy varies greatly. The structure of the three-state phone models is the same for all phones, as shown in Figure 15.21. That leaves the probabilities themselves. How are these to be obtained, given that the models may require hundreds of thousands or millions of parameters?

The only plausible method is to learn the models from actual speech data, of which there is certainly no shortage. The next question is how to do the learning. We give the answer in full in Chapter 19, but we can give the main ideas here. Consider the bigram language model; we explained how to learn it by looking at frequencies of word pairs in actual text. Can we do the same for, say, phone transition probabilities in the pronunciation model? The answer is yes, but only if someone goes to the trouble of annotating every occurrence of each word with the right phone sequence. This is a difficult and error-prone task, but has been carried out for some standard data sets containing several hours of speech. If we know the phone sequences, we can estimate transition probabilities for the pronunciation models from frequencies of phone pairs. Similarly, if we are given the phone state for each frame—an even more excruciating manual labelling task—then we can estimate transition probabilities for the phone models. Given the phone state and the acoustic features in each frame, we can also estimate the acoustic model, either directly from frequencies (for VQ models) or using statistical fitting methods (for mixture-of-Gaussian models; see Chapter 19).

The cost and rarity of hand-labelled data, and the fact that the available hand-labelled



data sets may not represent the kinds of speakers and acoustic conditions found in a new recognition context, could doom this approach to failure. *Fortunately, the expectation-maximization or EM algorithm learns HMM transition and sensor models without the need for labelled data.* Estimates derived from hand-labelled data can be used to initialize the models; after that, EM takes over and trains the models for the task at hand. The idea is simple: given an HMM and an observation sequence, we can use the smoothing algorithms from Sections 15.2 and 15.3 to compute the probability of each state at each time step, and, by a simple extension, the probability of each state-state pair at consecutive time steps. These probabilities can be viewed as *uncertain labels* in place of the definite labels provided by hand. From the uncertain labels, we can estimate new transition and sensor probabilities, and the EM procedure repeats. The method is guaranteed to increase the fit between model and data on each iteration, and generally converges to a much better set of parameter values than those provided by the initial, hand-labelled estimates.

State-of-the-art speech systems use enormous data sets and massive computational resources to train their models. For isolated word recognition under good acoustic conditions (no background noise or reverberation) with a vocabulary of a few thousand words and a single speaker, accuracy can be over 99%. For unrestricted continuous speech with a variety of speakers, 60–80% accuracy is common, even with good acoustic conditions. With background noise and telephone transmission, accuracy degrades further. Although fielded systems have improved continuously for decades, there is still room for many new ideas.

15.7 SUMMARY

This chapter has addressed the general problem of representing and reasoning about probabilistic temporal processes. The main points are as follows:

- The changing state of the world is handled using a set of random variables to represent the state at each point in time.
- Representations can be designed to satisfy the **Markov property**, so that the future is independent of the past given the present. Combined with the assumption that the process is **stationary**—i.e., the dynamics do not change over time—this greatly simplifies the representation.
- A temporal probability model can be thought of as containing a **transition model** describing the evolution and a **sensor model** describing the observation process.
- The principal inference tasks in temporal models are **filtering**, **prediction**, **smoothing**, and computing the **most likely explanation**. Each of these can be achieved using simple, recursive algorithms whose runtime is linear in the length of the sequence.
- Three families of temporal models were studied in more depth: **hidden Markov models**, **Kalman filters**, and **dynamic Bayesian networks** (which include the other two as special cases).
- **Speech recognition** and **tracking** are two important applications for temporal probability models.

- Unless special assumptions are made, as in Kalman filters, exact inference with many state variables appears to be intractable. In practice, the **particle filtering** algorithm seems to be an effective approximation algorithm.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Many of the basic ideas for estimating the state of dynamical systems came from the mathematician C. F. Gauss (1809). Gauss developed a deterministic least-squares algorithm for the problem of estimating orbits from astronomical observations. The Russian mathematician A. A. Markov (1913) developed what was later called the **Markov assumption** in his analysis of stochastic processes; he estimated a first-order Markov chain on letters from the text of *Eugene Onegin*. Significant classified work on filtering was done during World War II by Wiener (1942) for continuous-time processes and by Kolmogorov (1941) for discrete-time processes. Although this work led to important technological developments over the next twenty years, its use of a frequency-domain representation made many calculations quite cumbersome. Direct state-space modelling of the stochastic process turned out to be simpler, as shown by Swerling (1959) and Kalman (Kalman, 1960). The latter paper introduced what is now known as the Kalman filter for forward inference in linear systems with Gaussian noise. Important results on smoothing were derived by Rauch *et al.* (1965), and the impressively named **Rauch-Tung-Striebel smoother** is still a standard technique today. Many early results are gathered in Gelb (1974). Bar-Shalom and Fortmann (1988) give a more modern treatment with a Bayesian flavor, as well as many references to the vast literature on the subject.

In many applications of Kalman filtering, one must deal not only with uncertain sensing and dynamics but also with uncertain *identity*—that is, if there are multiple objects being monitored, the system must determine which observations were generated by which objects before it can update each of the state estimates. This is the problem of **data association** (Bar-Shalom and Fortmann, 1988; Bar-Shalom, 1992). With n observations and n tracks (a fairly benign case), there are $n!$ possible assignments of observations to tracks; a proper probabilistic treatment must take all of them into account, and this can be shown to be NP-hard (Cox, 1993; Cox and Hingorani, 1994). Polynomial-time approximation methods based on MCMC appear to work well in practice (Pasula *et al.*, 1999). It is interesting to note that the data association problem is an instance of probabilistic inference in a *first-order* language—unlike most probabilistic inference problems, which are purely propositional, data association involves *objects* as well as the *identity relation*. It is therefore intimately connected to the first-order probabilistic languages that were mentioned in Chapter 14. Recent work has shown that reasoning about identity in general, and data association in particular, can be carried out within the first-order probabilistic framework (Pasula and Russell, 2001).

The hidden Markov model and associated algorithms for inference and learning, including the forward–backward algorithm, were developed by Baum and Petrie (1966). Similar ideas also appeared independently in the Kalman filtering community (Rauch *et al.*, 1965). The forward–backward algorithm was one of the main precursors of the general formulation

of the EM algorithm (; see also Chapter 19 Dempster *et al.*, 1977). Constant-space smoothing appears in Binder *et al.* (1997), as does the divide-and-conquer algorithm developed in Exercise 15.3.

Dynamic belief networks (DBNs) can be viewed as a sparse encoding of a Markov process, and were first used in AI by Dean and Kanazawa (1989b), Nicholson (1992), and Kjaerulff (1992). The last work includes a generic extension to the HUGIN belief net system to provide the necessary facilities for dynamic belief network generation and compilation. Dynamic Bayesian networks have become popular for modelling a variety of complex motion processes in computer vision (Huang *et al.*, 1994; Intille and Bobick, 1999). The link between HMMs and DBNs, and between the forward–backward algorithm and Bayesian network propagation, was made explicitly by Smyth *et al.* (1997). A further unification with Kalman filters (as well as several other statistical models) appears in Roweis and Ghahramani (1999).

The particle filtering algorithm described in Section 15.5 has a particularly interesting history. The first sampling algorithms for filtering were developed in the control theory community by Handschin and Mayne (1969), and the resampling idea that is the core of particle filtering appeared in a Russian control journal (Zaritskii *et al.*, 1975). It was later reinvented in statistics as **sequential importance-sampling resampling** or **SIR** (Rubin, 1988; Liu and Chen, 1998), in control theory as particle filtering (Gordon *et al.*, 1993; Gordon, 1994), in AI as **survival of the fittest** (Kanazawa *et al.*, 1995), and in computer vision as **condensation** (Isard and Blake, 1996). The paper by Kanazawa *et al.* (1995) includes an improvement called **evidence reversal** whereby the state at time $t + 1$ is sampled conditional on both the state at time t and the evidence at time $t + 1$. This allows the evidence to influence sample generation directly, and was proved (independently) by Doucet (1997) to reduce the approximation error.

Alternative methods for approximate filtering include the **decayed MCMC** algorithm (Marthi *et al.*, 2002) and the factored approximation method of Boyen *et al.* (1999). Both of these methods have the important property that the approximation error does not diverge over time. Variational techniques (see Chapter 14) have also been developed for temporal models. Ghahramani and Jordan (1997) discuss an approximation algorithm for the **factorial HMM**, a DBN in which two or more independently evolving Markov chains are linked by a shared observation stream. Jordan *et al.* (1998) cover a number of other applications.

The prehistory of speech recognition began in the 1920s with Radio Rex, a voice-activated toy dog. Rex jumped in response to sound frequencies near 500 Hz, which corresponds to the [eh] vowel in “Rex!” Somewhat more serious work began after World War II. At AT&T Bell Labs, a system was built for recognizing isolated digits (Davis *et al.*, 1952) using simple pattern matching of acoustic features. Phone transition probabilities were first used in a system built at University College, London by Fry (1959) and Denes (1959). Starting in 1971, the Defense Advanced Research Projects Agency (DARPA) of the United States Department of Defense funded four competing five-year projects to develop high-performance speech recognition systems. The winner, and the only system to meet the goal of 90% accuracy with a 1000-word vocabulary, was the HARPY system at CMU (Lowerre, 1976; Lowerre

and Reddy, 1980).⁷ The final version of HARPY was derived from a system called DRAGON built by CMU graduate student James Baker (1975), which was the first to use HMMs for speech. Almost simultaneously, Jelinek (1976) at IBM had developed another HMM-based system. From that point onwards, probabilistic methods in general, and HMMs in particular, came to dominate speech recognition research and development. Recent years have been characterized by incremental progress, larger data sets and models, and more rigorous competitions on more realistic speech tasks. Some researchers have explored the possibility of using DBNs instead of HMMs for speech, with the aim of using the greater expressive power of DBNs to capture more of the complex hidden state of the speech apparatus (Zweig and Russell, 1998; Richardson *et al.*, 2000).

Several good textbooks on speech recognition are available (Rabiner and Juang, 1993; Jelinek, 1997; Gold and Morgan, 2000; Huang *et al.*, 2001). Waibel and Lee (1990) collect important papers in the area, including some tutorial ones. The presentation in this chapter drew on the survey by Kay, Kawron, and Norvig (1994), and on the textbook by Jurafsky and Martin (2000). Speech recognition research is published in *Computer Speech and Language*, *Speech Communications*, and the IEEE *Transactions on Acoustics, Speech, and Signal Processing*, and at the DARPA Workshops on Speech and Natural Language Processing and the Eurospeech, ICSLP, and ASRU conferences.

EXERCISES

15.1 Show that any second-order Markov process can be rewritten as a first-order Markov process with an augmented set of state variables. Can this always be done *parsimoniously*—that is, without increasing the number of parameters needed to specify the transition model?

15.2 In this exercise we examine what happens to the probabilities in the umbrella world in the limit of long time sequences.

- a. Suppose we observe an unending sequence of days on which the umbrella appears. Show that, as the days go by, the probability of rain on the current day increases monotonically towards a fixed point. Calculate this fixed point.
- b. Now consider *forecasting* further and further into the future, given just the first two umbrella observations. First, compute the probability $P(R_{2+k}|U_1, U_2)$ for $k = 1 \dots 20$ and plot the results. You should see that the probability converges towards a fixed point. Calculate the exact value of this fixed point.

15.3 This exercise develops a space-efficient variant of the forward–backward algorithm

⁷ The second-ranked system in the competition, HEARSAY-II (Erman *et al.*, 1980), had a great deal of influence on other branches of AI research because of its use of the **blackboard architecture**. It was a rule-based expert system with a number of more or less independent, modular **knowledge sources** which communicated via a common **blackboard** from which they could write and read. Blackboard systems are the foundation of modern user interface architectures.

described in Figure 15.4. We wish to compute $\mathbf{P}(\mathbf{X}_k | \mathbf{e}_{1:t})$ for $k = 1, \dots, t$. This will be done with a divide-and-conquer approach.

- Suppose, for simplicity, that t is odd, and let the halfway point be $h = (t + 1)/2$. Show that $\mathbf{P}(\mathbf{X}_k | \mathbf{e}_{1:t})$ can be computed for $k = 1, \dots, h$ given just the initial forward message $\mathbf{f}_{1:0}$, the backward message $\mathbf{b}_{h+1:t}$, and the evidence $\mathbf{e}_{1:h}$.
- Show a similar result for the second half of the sequence.
- Given the results of (a) and (b), a recursive, divide-and-conquer algorithm can be constructed by first running forward along the sequence and then backwards from the end, storing just the required messages at the middle and the ends. Then the algorithm is called on each half. Write out the algorithm in detail.
- Compute the time and space complexity of the algorithm as a function of t , the length of the sequence. How does this change if we divide into more than two pieces?

15.4 On page 552, we outlined a flawed procedure for finding the most likely state sequence, given an observation sequence. The procedure involves finding the most likely state at each time step, using smoothing, and returning the sequence composed of these states. Show that, for some temporal probability models and observation sequences, this procedure returns an impossible state sequence (i.e., the posterior probability of the sequence is zero).

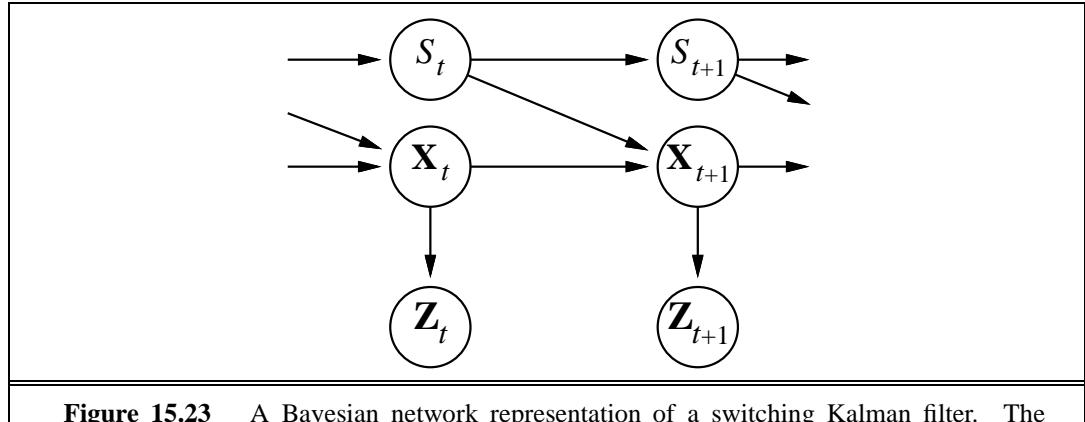


Figure 15.23 A Bayesian network representation of a switching Kalman filter. The switching variable S_t is a discrete state variable whose value determines the transition model for the continuous state variables \mathbf{X}_t . For any discrete state i , the transition model $\mathbf{P}(\mathbf{X}_{t+1} | \mathbf{X}_t, S_t = i)$ is a linear Gaussian model, just as in a regular Kalman filter. The transition model for the discrete state, $\mathbf{P}(S_{t+1} | S_t)$, can be thought of as a matrix just as in a hidden Markov model.

15.5 Often we wish to monitor a continuous-state system whose behavior switches unpredictably among a set of k distinct “modes.” For example, an aircraft trying to evade a missile may execute a series of distinct maneuvers that the missile may attempt to track. A Bayesian network representation of such a **switching Kalman filter** model is shown in Figure 15.23.

- Suppose that the discrete state S_t has k possible values and that the prior continuous state estimate $\mathbf{P}(\mathbf{X}_0)$ is a multivariate Gaussian distribution. Show that the prediction

$\mathbf{P}(\mathbf{X}_1)$ is a **mixture of Gaussians**—that is, a weighted sum of Gaussians such that the weights sum to 1.

- b. Show that if the current continuous state estimate $\mathbf{P}(\mathbf{X}_t|\mathbf{e}_{1:t})$ is a mixture of m Gaussians, then the updated state estimate $\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t+1})$ will be a mixture of km Gaussians in the general case.
- c. What aspect of the temporal process do the weights in the Gaussian mixture represent?

Together, the results in (a) and (b) show that the representation of the posterior grows without limit even for switching Kalman filters, which are the simplest hybrid dynamic models.

15.6 Complete the missing step in the derivation of Equation (15.18), the first update step for the one-dimensional Kalman filter.

15.7 Let us examine the behavior of the variance update in Equation (15.19).

- a. Plot the value of σ_t^2 as a function of t , given various values for σ_x^2 and σ_z^2 .
- b. Show that the update has a fixed point σ^2 such that $\sigma_t^2 \rightarrow \sigma^2$ as $t \rightarrow \infty$, and calculate it.
- c. Give a qualitative explanation for what happens as $\sigma_x^2 \rightarrow 0$ and as $\sigma_z^2 \rightarrow 0$.

15.8 Show how to represent an HMM as a recursive relational probabilistic model, as suggested in Section 14.6.

15.9 In this exercise, we analyze in more detail the persistent failure model for the battery sensor in Figure 15.13(a).

- a. Figure 15.13(b) stops at $t = 32$. Describe qualitatively what should happen as $t \rightarrow \infty$ if the sensor continues to read 0.
- b. Suppose that the external temperature affects the battery sensor, in such a way that transient failures become more likely as temperature increases. Show how to augment the DBN structure in Figure 15.13(a) and explain any required changes to the CPTs.
- c. Given the new network structure, can battery readings be used by the robot to infer the current temperature?

15.10 Consider applying the variable elimination algorithm to the umbrella DBN unrolled for three slices, where the query is $\mathbf{P}(R_3|U_1, U_2, U_3)$. Show that the complexity of the algorithm—the size of the largest factor—is the same whether the rain variables are eliminated in forward or backward order.

15.11 The model of “tomato” in Figure 15.20 allows for a coarticulation on the first vowel by giving two possible phones. An alternative approach is to use a triphone model in which the [ow(t,m)] phone automatically includes the change in vowel sound. Draw a complete triphone model for “tomato,” including the dialect variation.

plans generated by value iteration.) For problems in which the discount factor γ is not too close to 1, a shallow search is often good enough to give near-optimal decisions. It is also possible to approximate the averaging step at the chance nodes, by sampling from the set of possible percepts instead of summing over all possible percepts. There are various other ways of finding good approximate solutions quickly, but we defer them to Chapter 21.

Decision-theoretic agents based on dynamic decision networks have a number of advantages compared with other, simpler agent designs presented in earlier chapters. In particular, they handle partially observable, uncertain environments and can easily revise their “plans” to handle unexpected evidence. With appropriate sensor models, they can handle sensor failure and can plan to gather information. They exhibit “graceful degradation” under time pressure and in complex environments, using various approximation techniques. So what is missing? One defect of our DDN-based algorithm is its reliance on forward search through state space, rather than using the hierarchical and other advanced planning techniques described in Chapter 11. There have been attempts to extend these techniques into the probabilistic domain, but so far they have proved to be inefficient. A second, related problem is the basically propositional nature of the DDN language. We would like to be able to extend some of the ideas for first-order probabilistic languages to the problem of decision making. Current research has shown that this extension is possible and has significant benefits, as discussed in the notes at the end of the chapter.

17.5 DECISIONS WITH MULTIPLE AGENTS: GAME THEORY

GAME THEORY

This chapter has concentrated on making decisions in uncertain environments. But what if the uncertainty is due to other agents and the decisions they make? And what if the decisions of those agents are in turn influenced by our decisions? We addressed this question once before, when we studied games in Chapter 5. There, however, we were primarily concerned with turn-taking games in fully observable environments, for which minimax search can be used to find optimal moves. In this section we study the aspects of **game theory** that analyze games with simultaneous moves and other sources of partial observability. (Game theorists use the terms **perfect information** and **imperfect information** rather than fully and partially observable.) Game theory can be used in at least two ways:

1. **Agent design:** Game theory can analyze the agent’s decisions and compute the expected utility for each decision (under the assumption that other agents are acting optimally according to game theory). For example, in the game **two-finger Morra**, two players, O and E , simultaneously display one or two fingers. Let the total number of fingers be f . If f is odd, O collects f dollars from E ; and if f is even, E collects f dollars from O . Game theory can determine the best strategy against a rational player and the expected return for each player.⁴

⁴ Morra is a recreational version of an **inspection game**. In such games, an inspector chooses a day to inspect a facility (such as a restaurant or a biological weapons plant), and the facility operator chooses a day to hide all the nasty stuff. The inspector wins if the days are different, and the facility operator wins if they are the same.

2. **Mechanism design:** When an environment is inhabited by many agents, it might be possible to define the rules of the environment (i.e., the game that the agents must play) so that the collective good of all agents is maximized when each agent adopts the game-theoretic solution that maximizes its own utility. For example, game theory can help design the protocols for a collection of Internet traffic routers so that each router has an incentive to act in such a way that global throughput is maximized. Mechanism design can also be used to construct intelligent **multiagent systems** that solve complex problems in a distributed fashion.

17.5.1 Single-move games

We start by considering a restricted set of games: ones where all players take action simultaneously and the result of the game is based on this single set of actions. (Actually, it is not crucial that the actions take place at exactly the same time; what matters is that no player has knowledge of the other players' choices.) The restriction to a single move (and the very use of the word "game") might make this seem trivial, but in fact, game theory is serious business. It is used in decision-making situations including the auctioning of oil drilling rights and wireless frequency spectrum rights, bankruptcy proceedings, product development and pricing decisions, and national defense—situations involving billions of dollars and hundreds of thousands of lives. A single-move game is defined by three components:

- | PLAYER | <ul style="list-style-type: none"> • Players or agents who will be making decisions. Two-player games have received the most attention, although n-player games for $n > 2$ are also common. We give players capitalized names, like <i>Alice</i> and <i>Bob</i> or O and E. | | | | | | | | | |
|-----------------|---|------------------|-----------------|-----------------|-----------------|------------------|------------------|-----------------|------------------|------------------|
| ACTION | <ul style="list-style-type: none"> • Actions that the players can choose. We will give actions lowercase names, like <i>one</i> or <i>testify</i>. The players may or may not have the same set of actions available. | | | | | | | | | |
| PAYOFF FUNCTION | <ul style="list-style-type: none"> • A payoff function that gives the utility to each player for each combination of actions by all the players. For single-move games the payoff function can be represented by a matrix, a representation known as the strategic form (also called normal form). The payoff matrix for two-finger Morra is as follows: | | | | | | | | | |
| STRATEGIC FORM | <table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <th></th> <th>$O: \text{one}$</th> <th>$O: \text{two}$</th> </tr> <tr> <th>$E: \text{one}$</th> <td>$E = +2, O = -2$</td> <td>$E = -3, O = +3$</td> </tr> <tr> <th>$E: \text{two}$</th> <td>$E = -3, O = +3$</td> <td>$E = +4, O = -4$</td> </tr> </table> | | $O: \text{one}$ | $O: \text{two}$ | $E: \text{one}$ | $E = +2, O = -2$ | $E = -3, O = +3$ | $E: \text{two}$ | $E = -3, O = +3$ | $E = +4, O = -4$ |
| | $O: \text{one}$ | $O: \text{two}$ | | | | | | | | |
| $E: \text{one}$ | $E = +2, O = -2$ | $E = -3, O = +3$ | | | | | | | | |
| $E: \text{two}$ | $E = -3, O = +3$ | $E = +4, O = -4$ | | | | | | | | |

For example, the lower-right corner shows that when player O chooses action *two* and E also chooses *two*, the payoff is $+4$ for E and -4 for O .

- | | |
|------------------|---|
| STRATEGY | Each player in a game must adopt and then execute a strategy (which is the name used in game theory for a <i>policy</i>). A pure strategy is a deterministic policy; for a single-move game, a pure strategy is just a single action. For many games an agent can do better with a mixed strategy , which is a randomized policy that selects actions according to a probability distribution. The mixed strategy that chooses action a with probability p and action b otherwise is written $[p: a; (1 - p): b]$. For example, a mixed strategy for two-finger Morra might be $[0.5: \text{one}; 0.5: \text{two}]$. A strategy profile is an assignment of a strategy to each player; given the strategy profile, the game's outcome is a numeric value for each player. |
| PURE STRATEGY | |
| MIXED STRATEGY | |
| STRATEGY PROFILE | |
| OUTCOME | |

SOLUTION

A **solution** to a game is a strategy profile in which each player adopts a rational strategy. We will see that the most important issue in game theory is to define what “rational” means when each agent chooses only part of the strategy profile that determines the outcome. It is important to realize that outcomes are actual results of playing a game, while solutions are theoretical constructs used to analyze a game. We will see that some games have a solution only in mixed strategies. But that does not mean that a player must literally be adopting a mixed strategy to be rational.

PRISONER'S DILEMMA

Consider the following story: Two alleged burglars, Alice and Bob, are caught red-handed near the scene of a burglary and are interrogated separately. A prosecutor offers each a deal: if you testify against your partner as the leader of a burglary ring, you’ll go free for being the cooperative one, while your partner will serve 10 years in prison. However, if you both testify against each other, you’ll both get 5 years. Alice and Bob also know that if both refuse to testify they will serve only 1 year each for the lesser charge of possessing stolen property. Now Alice and Bob face the so-called **prisoner’s dilemma**: should they testify or refuse? Being rational agents, Alice and Bob each want to maximize their own expected utility. Let’s assume that Alice is callously unconcerned about her partner’s fate, so her utility decreases in proportion to the number of years she will spend in prison, regardless of what happens to Bob. Bob feels exactly the same way. To help reach a rational decision, they both construct the following payoff matrix:

	<i>Alice:testify</i>	<i>Alice:refuse</i>
<i>Bob:testify</i>	$A = -5, B = -5$	$A = -10, B = 0$
<i>Bob:refuse</i>	$A = 0, B = -10$	$A = -1, B = -1$

DOMINANT STRATEGY STRONG DOMINATION

WEAK DOMINATION

PARETO OPTIMAL

PARETO DOMINATED

DOMINANT STRATEGY EQUILIBRIUM EQUILIBRIUM

Alice analyzes the payoff matrix as follows: “Suppose Bob testifies. Then I get 5 years if I testify and 10 years if I don’t, so in that case testifying is better. On the other hand, if Bob refuses, then I get 0 years if I testify and 1 year if I refuse, so in that case as well testifying is better. So in either case, it’s better for me to testify, so that’s what I must do.”

Alice has discovered that *testify* is a **dominant strategy** for the game. We say that a strategy s for player p **strongly dominates** strategy s' if the outcome for s is better for p than the outcome for s' , for every choice of strategies by the other player(s). Strategy s **weakly dominates** s' if s is better than s' on at least one strategy profile and no worse on any other. A dominant strategy is a strategy that dominates all others. It is irrational to play a dominated strategy, and irrational not to play a dominant strategy if one exists. Being rational, Alice chooses the dominant strategy. We need just a bit more terminology: we say that an outcome is **Pareto optimal**⁵ if there is no other outcome that all players would prefer. An outcome is **Pareto dominated** by another outcome if all players would prefer the other outcome.

If Alice is clever as well as rational, she will continue to reason as follows: Bob’s dominant strategy is also to testify. Therefore, he will testify and we will both get five years. When each player has a dominant strategy, the combination of those strategies is called a **dominant strategy equilibrium**. In general, a strategy profile forms an **equilibrium** if no player can benefit by switching strategies, given that every other player sticks with the same

⁵ Pareto optimality is named after the economist Vilfredo Pareto (1848–1923).

strategy. An equilibrium is essentially a **local optimum** in the space of policies; it is the top of a peak that slopes downward along every dimension, where a dimension corresponds to a player's strategy choices.



NASH EQUILIBRIUM

The mathematician John Nash (1928–) proved that *every game has at least one equilibrium*. The general concept of equilibrium is now called **Nash equilibrium** in his honor. Clearly, a dominant strategy equilibrium is a Nash equilibrium (Exercise 17.16), but some games have Nash equilibria but no dominant strategies.

The *dilemma* in the prisoner's dilemma is that the equilibrium outcome is worse for both players than the outcome they would get if they both refused to testify. In other words, *(testify, testify)* is Pareto dominated by the $(-1, -1)$ outcome of *(refuse, refuse)*. Is there any way for Alice and Bob to arrive at the $(-1, -1)$ outcome? It is certainly an *allowable* option for both of them to refuse to testify, but it is hard to see how rational agents can get there, given the definition of the game. Either player contemplating playing *refuse* will realize that he or she would do better by playing *testify*. That is the attractive power of an equilibrium point. Game theorists agree that being a Nash equilibrium is a necessary condition for being a solution—although they disagree whether it is a sufficient condition.

It is easy enough to get to the *(refuse, refuse)* solution if we modify the game. For example, we could change to a **repeated game** in which the players know that they will meet again. Or the agents might have moral beliefs that encourage cooperation and fairness. That means they have a different utility function, necessitating a different payoff matrix, making it a different game. We will see later that agents with limited computational powers, rather than the ability to reason absolutely rationally, can reach non-equilibrium outcomes, as can an agent that knows that the other agent has limited rationality. In each case, we are considering a different game than the one described by the payoff matrix above.

Now let's look at a game that has no dominant strategy. Acme, a video game console manufacturer, has to decide whether its next game machine will use Blu-ray discs or DVDs. Meanwhile, the video game software producer Best needs to decide whether to produce its next game on Blu-ray or DVD. The profits for both will be positive if they agree and negative if they disagree, as shown in the following payoff matrix:

	<i>Acme:bluray</i>	<i>Acme:dvd</i>
<i>Best:bluray</i>	$A = +9, B = +9$	$A = -4, B = -1$
<i>Best:dvd</i>	$A = -3, B = -1$	$A = +5, B = +5$



There is no dominant strategy equilibrium for this game, but there are *two* Nash equilibria: *(bluray, bluray)* and *(dvd, dvd)*. We know these are Nash equilibria because if either player unilaterally moves to a different strategy, that player will be worse off. Now the agents have a problem: *there are multiple acceptable solutions, but if each agent aims for a different solution, then both agents will suffer*. How can they agree on a solution? One answer is that both should choose the Pareto-optimal solution *(bluray, bluray)*; that is, we can restrict the definition of “solution” to the unique Pareto-optimal Nash equilibrium *provided that one exists*. Every game has at least one Pareto-optimal solution, but a game might have several, or they might not be equilibrium points. For example, if *(bluray, bluray)* had payoff $(5, 5)$, then there would be two equal Pareto-optimal equilibrium points. To choose between

COORDINATION GAME

ZERO-SUM GAME

MAXIMIN

them the agents can either guess or *communicate*, which can be done either by establishing a convention that orders the solutions before the game begins or by negotiating to reach a mutually beneficial solution during the game (which would mean including communicative actions as part of a sequential game). Communication thus arises in game theory for exactly the same reasons that it arose in multiagent planning in Section 11.4. Games in which players need to communicate like this are called **coordination games**.

A game can have more than one Nash equilibrium; how do we know that every game must have at least one? Some games have no *pure-strategy* Nash equilibria. Consider, for example, any pure-strategy profile for two-finger Morra (page 666). If the total number of fingers is even, then O will want to switch; on the other hand (so to speak), if the total is odd, then E will want to switch. Therefore, no pure strategy profile can be an equilibrium and we must look to mixed strategies instead.

But *which* mixed strategy? In 1928, von Neumann developed a method for finding the *optimal* mixed strategy for two-player, **zero-sum games**—games in which the sum of the payoffs is always zero.⁶ Clearly, Morra is such a game. For two-player, zero-sum games, we know that the payoffs are equal and opposite, so we need consider the payoffs of only one player, who will be the maximizer (just as in Chapter 5). For Morra, we pick the even player E to be the maximizer, so we can define the payoff matrix by the values $U_E(e, o)$ —the payoff to E if E does e and O does o . (For convenience we call player E “her” and O “him.”) Von Neumann’s method is called the the **maximin** technique, and it works as follows:

- Suppose we change the rules as follows: first E picks her strategy and reveals it to O . Then O picks his strategy, with knowledge of E ’s strategy. Finally, we evaluate the expected payoff of the game based on the chosen strategies. This gives us a turn-taking game to which we can apply the standard **minimax** algorithm from Chapter 5. Let’s suppose this gives an outcome $U_{E,O}$. Clearly, this game favors O , so the true utility U of the original game (from E ’s point of view) is *at least* $U_{E,O}$. For example, if we just look at pure strategies, the minimax game tree has a root value of -3 (see Figure 17.12(a)), so we know that $U \geq -3$.
- Now suppose we change the rules to force O to reveal his strategy first, followed by E . Then the minimax value of this game is $U_{O,E}$, and because this game favors E we know that U is *at most* $U_{O,E}$. With pure strategies, the value is $+2$ (see Figure 17.12(b)), so we know $U \leq +2$.

Combining these two arguments, we see that the true utility U of the solution to the original game must satisfy

$$U_{E,O} \leq U \leq U_{O,E} \quad \text{or in this case,} \quad -3 \leq U \leq 2 .$$



To pinpoint the value of U , we need to turn our analysis to mixed strategies. First, observe the following: *once the first player has revealed his or her strategy, the second player might as well choose a pure strategy*. The reason is simple: if the second player plays a mixed strategy, $[p: \text{one}; (1-p): \text{two}]$, its expected utility is a linear combination $(p \cdot u_{\text{one}} + (1-p) \cdot u_{\text{two}})$ of

⁶ or a constant—see page 162.

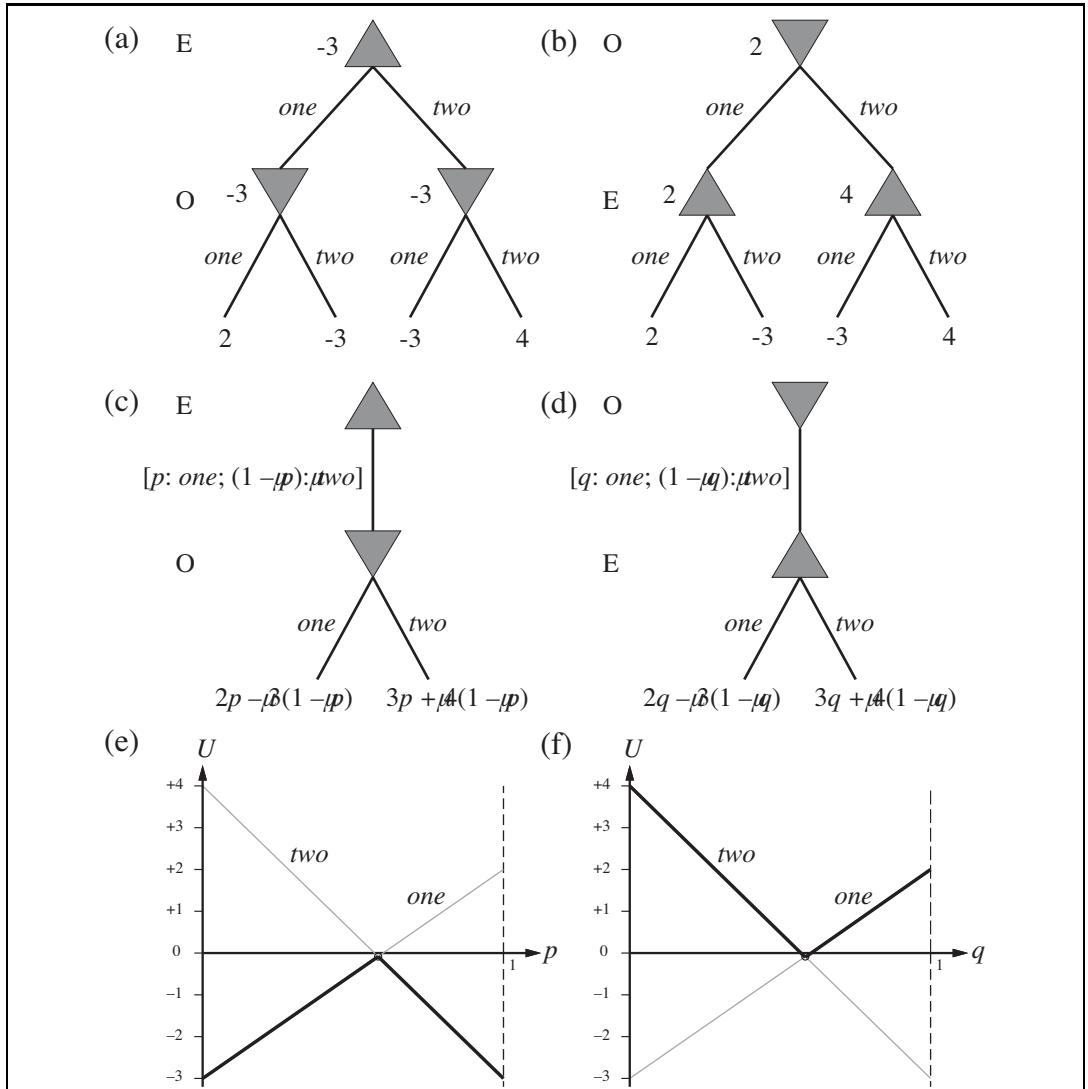


Figure 17.12 (a) and (b): Minimax game trees for two-finger Morra if the players take turns playing pure strategies. (c) and (d): Parameterized game trees where the first player plays a mixed strategy. The payoffs depend on the probability parameter (p or q) in the mixed strategy. (e) and (f): For any particular value of the probability parameter, the second player will choose the “better” of the two actions, so the value of the first player’s mixed strategy is given by the heavy lines. The first player will choose the probability parameter for the mixed strategy at the intersection point.

the utilities of the pure strategies, u_{one} and u_{two} . This linear combination can never be better than the better of u_{one} and u_{two} , so the second player can just choose the better one.

With this observation in mind, the minimax trees can be thought of as having infinitely many branches at the root, corresponding to the infinitely many mixed strategies the first

player can choose. Each of these leads to a node with two branches corresponding to the pure strategies for the second player. We can depict these infinite trees finitely by having one “parameterized” choice at the root:

- If E chooses first, the situation is as shown in Figure 17.12(c). E chooses the strategy $[p: \text{one}; (1-p): \text{two}]$ at the root, and then O chooses a pure strategy (and hence a move) given the value of p . If O chooses *one*, the expected payoff (to E) is $2p - 3(1-p) = 5p - 3$; if O chooses *two*, the expected payoff is $-3p + 4(1-p) = 4 - 7p$. We can draw these two payoffs as straight lines on a graph, where p ranges from 0 to 1 on the x -axis, as shown in Figure 17.12(e). O , the minimizer, will always choose the lower of the two lines, as shown by the heavy lines in the figure. Therefore, the best that E can do at the root is to choose p to be at the intersection point, which is where

$$5p - 3 = 4 - 7p \quad \Rightarrow \quad p = 7/12.$$

The utility for E at this point is $U_{E,O} = -1/12$.

- If O moves first, the situation is as shown in Figure 17.12(d). O chooses the strategy $[q: \text{one}; (1-q): \text{two}]$ at the root, and then E chooses a move given the value of q . The payoffs are $2q - 3(1-q) = 5q - 3$ and $-3q + 4(1-q) = 4 - 7q$.⁷ Again, Figure 17.12(f) shows that the best O can do at the root is to choose the intersection point:

$$5q - 3 = 4 - 7q \quad \Rightarrow \quad q = 7/12.$$

The utility for E at this point is $U_{O,E} = -1/12$.

Now we know that the true utility of the original game lies between $-1/12$ and $-1/12$, that is, it is exactly $-1/12$! (The moral is that it is better to be O than E if you are playing this game.) Furthermore, the true utility is attained by the mixed strategy $[7/12: \text{one}; 5/12: \text{two}]$, which should be played by both players. This strategy is called the **maximin equilibrium** of the game, and is a Nash equilibrium. Note that each component strategy in an equilibrium mixed strategy has the same expected utility. In this case, both *one* and *two* have the same expected utility, $-1/12$, as the mixed strategy itself.

MAXIMIN EQUILIBRIUM



Our result for two-finger Morra is an example of the general result by von Neumann: *every two-player zero-sum game has a maximin equilibrium when you allow mixed strategies*. Furthermore, every Nash equilibrium in a zero-sum game is a maximin for both players. A player who adopts the maximin strategy has two guarantees: First, no other strategy can do better against an opponent who plays well (although some other strategies might be better at exploiting an opponent who makes irrational mistakes). Second, the player continues to do just as well even if the strategy is revealed to the opponent.

The general algorithm for finding maximin equilibria in zero-sum games is somewhat more involved than Figures 17.12(e) and (f) might suggest. When there are n possible actions, a mixed strategy is a point in n -dimensional space and the lines become hyperplanes. It's also possible for some pure strategies for the second player to be dominated by others, so that they are not optimal against *any* strategy for the first player. After removing all such strategies (which might have to be done repeatedly), the optimal choice at the root is the

⁷ It is a coincidence that these equations are the same as those for p ; the coincidence arises because $U_E(\text{one}, \text{two}) = U_E(\text{two}, \text{one}) = -3$. This also explains why the optimal strategy is the same for both players.

highest (or lowest) intersection point of the remaining hyperplanes. Finding this choice is an example of a **linear programming** problem: maximizing an objective function subject to linear constraints. Such problems can be solved by standard techniques in time polynomial in the number of actions (and in the number of bits used to specify the reward function, if you want to get technical).

The question remains, what should a rational agent actually *do* in playing a single game of Morra? The rational agent will have derived the fact that $[7/12: \text{one}; 5/12: \text{two}]$ is the maximin equilibrium strategy, and will assume that this is mutual knowledge with a rational opponent. The agent could use a 12-sided die or a random number generator to pick randomly according to this mixed strategy, in which case the expected payoff would be $-1/12$ for E . Or the agent could just decide to play *one*, or *two*. In either case, the expected payoff remains $-1/12$ for E . Curiously, unilaterally choosing a particular action does not harm one's expected payoff, but allowing the other agent to know that one has made such a unilateral decision *does* affect the expected payoff, because then the opponent can adjust his strategy accordingly.

Finding equilibria in non-zero-sum games is somewhat more complicated. The general approach has two steps: (1) Enumerate all possible subsets of actions that might form mixed strategies. For example, first try all strategy profiles where each player uses a single action, then those where each player uses either one or two actions, and so on. This is exponential in the number of actions, and so only applies to relatively small games. (2) For each strategy profile enumerated in (1), check to see if it is an equilibrium. This is done by solving a set of equations and inequalities that are similar to the ones used in the zero-sum case. For two players these equations are linear and can be solved with basic linear programming techniques, but for three or more players they are nonlinear and may be very difficult to solve.

17.5.2 Repeated games

REPEATED GAME

So far we have looked only at games that last a single move. The simplest kind of multiple-move game is the **repeated game**, in which players face the same choice repeatedly, but each time with knowledge of the history of all players' previous choices. A strategy profile for a repeated game specifies an action choice for each player at each time step for every possible history of previous choices. As with MDPs, payoffs are additive over time.

Let's consider the repeated version of the prisoner's dilemma. Will Alice and Bob work together and refuse to testify, knowing they will meet again? The answer depends on the details of the engagement. For example, suppose Alice and Bob know that they must play exactly 100 rounds of prisoner's dilemma. Then they both know that the 100th round will not be a repeated game—that is, its outcome can have no effect on future rounds—and therefore they will both choose the dominant strategy, *testify*, in that round. But once the 100th round is determined, the 99th round can have no effect on subsequent rounds, so it too will have a dominant strategy equilibrium at $(\text{testify}, \text{testify})$. By induction, both players will choose *testify* on every round, earning a total jail sentence of 500 years each.

We can get different solutions by changing the rules of the interaction. For example, suppose that after each round there is a 99% chance that the players will meet again. Then the expected number of rounds is still 100, but neither player knows for sure which round

PERPETUAL PUNISHMENT

will be the last. Under these conditions, more cooperative behavior is possible. For example, one equilibrium strategy is for each player to *refuse* unless the other player has ever played *testify*. This strategy could be called **perpetual punishment**. Suppose both players have adopted this strategy, and this is mutual knowledge. Then as long as neither player has played *testify*, then at any point in time the expected future total payoff for each player is

$$\sum_{t=0}^{\infty} 0.99^t \cdot (-1) = -100 .$$

A player who deviates from the strategy and chooses *testify* will gain a score of 0 rather than -1 on the very next move, but from then on both players will play *testify* and the player's total expected future payoff becomes

$$0 + \sum_{t=1}^{\infty} 0.99^t \cdot (-5) = -495 .$$

Therefore, at every step, there is no incentive to deviate from $(\text{refuse}, \text{refuse})$. Perpetual punishment is the “mutually assured destruction” strategy of the prisoner’s dilemma: once either player decides to *testify*, it ensures that both players suffer a great deal. But it works as a deterrent only if the other player believes you have adopted this strategy—or at least that you might have adopted it.

TIT-FOR-TAT

Other strategies are more forgiving. The most famous, called **tit-for-tat**, calls for starting with *refuse* and then echoing the other player’s previous move on all subsequent moves. So Alice would refuse as long as Bob refuses and would testify the move after Bob testified, but would go back to refusing if Bob did. Although very simple, this strategy has proven to be highly robust and effective against a wide variety of strategies.

We can also get different solutions by changing the agents, rather than changing the rules of engagement. Suppose the agents are finite-state machines with n states and they are playing a game with $m > n$ total steps. The agents are thus incapable of representing the number of remaining steps, and must treat it as an unknown. Therefore, they cannot do the induction, and are free to arrive at the more favorable $(\text{refuse}, \text{refuse})$ equilibrium. In this case, ignorance is bliss—or rather, having your opponent believe that you are ignorant is bliss. Your success in these repeated games depends on the other player’s *perception* of you as a bully or a simpleton, and not on your actual characteristics.

17.5.3 Sequential games

EXTENSIVE FORM

In the general case, a game consists of a sequence of turns that need not be all the same. Such games are best represented by a game tree, which game theorists call the **extensive form**. The tree includes all the same information we saw in Section 5.1: an initial state S_0 , a function $\text{PLAYER}(s)$ that tells which player has the move, a function $\text{ACTIONS}(s)$ enumerating the possible actions, a function $\text{RESULT}(s, a)$ that defines the transition to a new state, and a partial function $\text{UTILITY}(s, p)$, which is defined only on terminal states, to give the payoff for each player.

To represent stochastic games, such as backgammon, we add a distinguished player, *chance*, that can take random actions. *Chance*’s “strategy” is part of the definition of the

20 STATISTICAL LEARNING METHODS

In which we view learning as a form of uncertain reasoning from observations.

Part V pointed out the prevalence of uncertainty in real environments. Agents can handle uncertainty by using the methods of probability and decision theory, but first they must learn their probabilistic theories of the world from experience. This chapter explains how they can do that. We will see how to formulate the learning task itself as a process of probabilistic inference (Section 20.1). We will see that a Bayesian view of learning is extremely powerful, providing general solutions to the problems of noise, overfitting, and optimal prediction. It also takes into account the fact that a less-than-omniscient agent can never be certain about which theory of the world is correct, yet must still make decisions by using some theory of the world.

We describe methods for learning probability models—primarily Bayesian networks—in Sections 20.2 and 20.3. Section 20.4 looks at learning methods that store and recall specific instances. Section 20.5 covers **neural network** learning and Section 20.6 introduces **kernel machines**. Some of the material in this chapter is fairly mathematical (requiring a basic understanding of multivariate calculus), although the general lessons can be understood without plunging into the details. It may benefit the reader at this point to review the material in Chapters 13 and 14 and to peek at the mathematical background in Appendix A.

20.1 STATISTICAL LEARNING

The key concepts in this chapter, just as in Chapter 18, are **data** and **hypotheses**. Here, the data are **evidence**—that is, instantiations of some or all of the random variables describing the domain. The hypotheses are probabilistic theories of how the domain works, including logical theories as a special case.

Let us consider a *very* simple example. Our favorite Surprise candy comes in two flavors: cherry (yum) and lime (ugh). The candy manufacturer has a peculiar sense of humor and wraps each piece of candy in the same opaque wrapper, regardless of flavor. The candy is sold in very large bags, of which there are known to be five kinds—again, indistinguishable from the outside:

- h_1 : 100% cherry
- h_2 : 75% cherry + 25% lime
- h_3 : 50% cherry + 50% lime
- h_4 : 25% cherry + 75% lime
- h_5 : 100% lime

Given a new bag of candy, the random variable H (for *hypothesis*) denotes the type of the bag, with possible values h_1 through h_5 . H is not directly observable, of course. As the pieces of candy are opened and inspected, data are revealed— D_1, D_2, \dots, D_N , where each D_i is a random variable with possible values *cherry* and *lime*. The basic task faced by the agent is to predict the flavor of the next piece of candy.¹ Despite its apparent triviality, this scenario serves to introduce many of the major issues. The agent really does need to infer a theory of its world, albeit a very simple one.

BAYESIAN LEARNING

Bayesian learning simply calculates the probability of each hypothesis, given the data, and makes predictions on that basis. That is, the predictions are made by using *all* the hypotheses, weighted by their probabilities, rather than by using just a single “best” hypothesis. In this way, learning is reduced to probabilistic inference. Let \mathbf{D} represent all the data, with observed value \mathbf{d} ; then the probability of each hypothesis is obtained by Bayes’ rule:

$$P(h_i|\mathbf{d}) = \alpha P(\mathbf{d}|h_i)P(h_i). \quad (20.1)$$

Now, suppose we want to make a prediction about an unknown quantity X . Then we have

$$\mathbf{P}(X|\mathbf{d}) = \sum_i \mathbf{P}(X|\mathbf{d}, h_i)\mathbf{P}(h_i|\mathbf{d}) = \sum_i \mathbf{P}(X|h_i)P(h_i|\mathbf{d}), \quad (20.2)$$

where we have assumed that each hypothesis determines a probability distribution over X . This equation shows that predictions are weighted averages over the predictions of the individual hypotheses. The hypotheses themselves are essentially “intermediaries” between the raw data and the predictions. The key quantities in the Bayesian approach are the **hypothesis prior**, $P(h_i)$, and the **likelihood** of the data under each hypothesis, $P(\mathbf{d}|h_i)$.

HYPOTHESIS PRIOR

LIKELIHOOD

I.I.D.

For our candy example, we will assume for the time being that the prior distribution over h_1, \dots, h_5 is given by $\langle 0.1, 0.2, 0.4, 0.2, 0.1 \rangle$, as advertised by the manufacturer. The likelihood of the data is calculated under the assumption that the observations are **i.i.d.**—that is, independently and identically distributed—so that

$$P(\mathbf{d}|h_i) = \prod_j P(d_j|h_i). \quad (20.3)$$

For example, suppose the bag is really an all-lime bag (h_5) and the first 10 candies are all lime; then $P(\mathbf{d}|h_5)$ is 0.5^{10} , because half the candies in an h_3 bag are lime.² Figure 20.1(a) shows how the posterior probabilities of the five hypotheses change as the sequence of 10 lime candies is observed. Notice that the probabilities start out at their prior values, so h_3 is initially the most likely choice and remains so after 1 lime candy is unwrapped. After 2

¹ Statistically sophisticated readers will recognize this scenario as a variant of the **urn-and-ball** setup. We find urns and balls less compelling than candy; furthermore, candy lends itself to other tasks, such as deciding whether to trade the bag with a friend—see Exercise 20.3.

² We stated earlier that the bags of candy are very large; otherwise, the i.i.d. assumption fails to hold. Technically, it is more correct (but less hygienic) to rewrap each candy after inspection and return it to the bag.

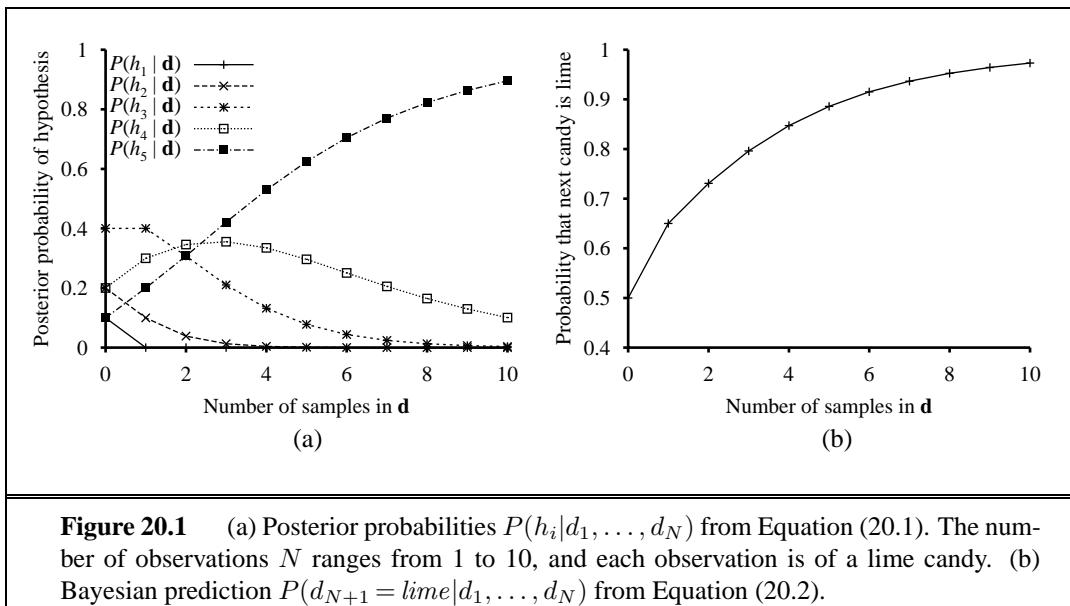


Figure 20.1 (a) Posterior probabilities $P(h_i|d_1, \dots, d_N)$ from Equation (20.1). The number of observations N ranges from 1 to 10, and each observation is of a lime candy. (b) Bayesian prediction $P(d_{N+1} = \text{lime}|d_1, \dots, d_N)$ from Equation (20.2).

lime candies are unwrapped, h_4 is most likely; after 3 or more, h_5 (the dreaded all-lime bag) is the most likely. After 10 in a row, we are fairly certain of our fate. Figure 20.1(b) shows the predicted probability that the next candy is lime, based on Equation (20.2). As we would expect, it increases monotonically toward 1.



The example shows that *the true hypothesis eventually dominates the Bayesian prediction*. This is characteristic of Bayesian learning. For any fixed prior that does not rule out the true hypothesis, the posterior probability of any false hypothesis will eventually vanish, simply because the probability of generating “uncharacteristic” data indefinitely is vanishingly small. (This point is analogous to one made in the discussion of PAC learning in Chapter 18.) More importantly, the Bayesian prediction is *optimal*, whether the data set be small or large. Given the hypothesis prior, any other prediction will be correct less often.

The optimality of Bayesian learning comes at a price, of course. For real learning problems, the hypothesis space is usually very large or infinite, as we saw in Chapter 18. In some cases, the summation in Equation (20.2) (or integration, in the continuous case) can be carried out tractably, but in most cases we must resort to approximate or simplified methods.

A very common approximation—one that is usually adopted in science—is to make predictions based on a single *most probable* hypothesis—that is, an h_i that maximizes $P(h_i|\mathbf{d})$. This is often called a **maximum a posteriori** or MAP (pronounced “em-ay-pee”) hypothesis. Predictions made according to an MAP hypothesis h_{MAP} are approximately Bayesian to the extent that $\mathbf{P}(X|\mathbf{d}) \approx \mathbf{P}(X|h_{\text{MAP}})$. In our candy example, $h_{\text{MAP}} = h_5$ after three lime candies in a row, so the MAP learner then predicts that the fourth candy is lime with probability 1.0—a much more dangerous prediction than the Bayesian prediction of 0.8 shown in Figure 20.1. As more data arrive, the MAP and Bayesian predictions become closer, because the competitors to the MAP hypothesis become less and less probable. Although our example doesn’t show it, finding MAP hypotheses is often much easier than Bayesian learn-

ing, because it requires solving an optimization problem instead of a large summation (or integration) problem. We will see examples of this later in the chapter.

In both Bayesian learning and MAP learning, the hypothesis prior $P(h_i)$ plays an important role. We saw in Chapter 18 that **overfitting** can occur when the hypothesis space is too expressive, so that it contains many hypotheses that fit the data set well. Rather than placing an arbitrary limit on the hypotheses to be considered, Bayesian and MAP learning methods use the prior to *penalize complexity*. Typically, more complex hypotheses have a lower prior probability—in part because there are usually many more complex hypotheses than simple hypotheses. On the other hand, more complex hypotheses have a greater capacity to fit the data. (In the extreme case, a lookup table can reproduce the data exactly with probability 1.) Hence, the hypothesis prior embodies a trade-off between the complexity of a hypothesis and its degree of fit to the data.



We can see the effect of this trade-off most clearly in the logical case, where H contains only *deterministic* hypotheses. In that case, $P(\mathbf{d}|h_i)$ is 1 if h_i is consistent and 0 otherwise. Looking at Equation (20.1), we see that h_{MAP} will then be the *simplest logical theory that is consistent with the data*. Therefore, maximum *a posteriori* learning provides a natural embodiment of Ockham’s razor.

Another insight into the trade-off between complexity and degree of fit is obtained by taking the logarithm of Equation (20.1). Choosing h_{MAP} to maximize $P(\mathbf{d}|h_i)P(h_i)$ is equivalent to minimizing

$$-\log_2 P(\mathbf{d}|h_i) - \log_2 P(h_i).$$

MINIMUM DESCRIPTION LENGTH

Using the connection between information encoding and probability that we introduced in Chapter 18, we see that the $-\log_2 P(h_i)$ term equals the number of bits required to specify the hypothesis h_i . Furthermore, $-\log_2 P(\mathbf{d}|h_i)$ is the additional number of bits required to specify the data, given the hypothesis. (To see this, consider that no bits are required if the hypothesis predicts the data exactly—as with h_5 and the string of lime candies—and $\log_2 1 = 0$.) Hence, MAP learning is choosing the hypothesis that provides maximum *compression* of the data. The same task is addressed more directly by the **minimum description length**, or MDL, learning method, which attempts to minimize the size of hypothesis and data encodings rather than work with probabilities.

MAXIMUM-LIKELIHOOD

A final simplification is provided by assuming a **uniform** prior over the space of hypotheses. In that case, MAP learning reduces to choosing an h_i that maximizes $P(\mathbf{d}|H_i)$. This is called a **maximum-likelihood** (ML) hypothesis, h_{ML} . Maximum-likelihood learning is very common in statistics, a discipline in which many researchers distrust the subjective nature of hypothesis priors. It is a reasonable approach when there is no reason to prefer one hypothesis over another *a priori*—for example, when all hypotheses are equally complex. It provides a good approximation to Bayesian and MAP learning when the data set is large, because the data swamps the prior distribution over hypotheses, but it has problems (as we shall see) with small data sets.

20.2 LEARNING WITH COMPLETE DATA

PARAMETER
LEARNING
COMPLETE DATA

Our development of statistical learning methods begins with the simplest task: **parameter learning** with **complete data**. A parameter learning task involves finding the numerical parameters for a probability model whose structure is fixed. For example, we might be interested in learning the conditional probabilities in a Bayesian network with a given structure. Data are complete when each data point contains values for every variable in the probability model being learned. Complete data greatly simplify the problem of learning the parameters of a complex model. We will also look briefly at the problem of learning structure.

Maximum-likelihood parameter learning: Discrete models

Suppose we buy a bag of lime and cherry candy from a new manufacturer whose lime–cherry proportions are completely unknown—that is, the fraction could be anywhere between 0 and 1. In that case, we have a continuum of hypotheses. The **parameter** in this case, which we call θ , is the proportion of cherry candies, and the hypothesis is h_θ . (The proportion of limes is just $1 - \theta$.) If we assume that all proportions are equally likely *a priori*, then a maximum-likelihood approach is reasonable. If we model the situation with a Bayesian network, we need just one random variable, *Flavor* (the flavor of a randomly chosen candy from the bag). It has values *cherry* and *lime*, where the probability of *cherry* is θ (see Figure 20.2(a)). Now suppose we unwrap N candies, of which c are cherries and $\ell = N - c$ are limes. According to Equation (20.3), the likelihood of this particular data set is

$$P(\mathbf{d}|h_\theta) = \prod_{j=1}^N P(d_j|h_\theta) = \theta^c \cdot (1 - \theta)^\ell.$$

LOG LIKELIHOOD

The maximum-likelihood hypothesis is given by the value of θ that maximizes this expression. The same value is obtained by maximizing the **log likelihood**,

$$L(\mathbf{d}|h_\theta) = \log P(\mathbf{d}|h_\theta) = \sum_{j=1}^N \log P(d_j|h_\theta) = c \log \theta + \ell \log(1 - \theta).$$

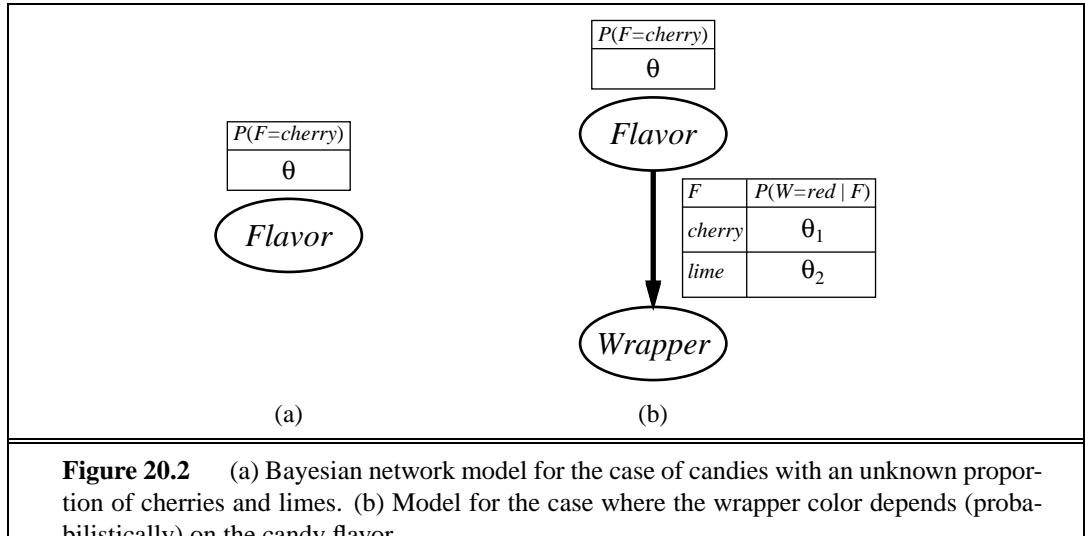
(By taking logarithms, we reduce the product to a sum over the data, which is usually easier to maximize.) To find the maximum-likelihood value of θ , we differentiate L with respect to θ and set the resulting expression to zero:

$$\frac{dL(\mathbf{d}|h_\theta)}{d\theta} = \frac{c}{\theta} - \frac{\ell}{1 - \theta} = 0 \quad \Rightarrow \quad \theta = \frac{c}{c + \ell} = \frac{c}{N}.$$

In English, then, the maximum-likelihood hypothesis h_{ML} asserts that the actual proportion of cherries in the bag is equal to the observed proportion in the candies unwrapped so far!

It appears that we have done a lot of work to discover the obvious. In fact, though, we have laid out one standard method for maximum-likelihood parameter learning:

1. Write down an expression for the likelihood of the data as a function of the parameter(s).
2. Write down the derivative of the log likelihood with respect to each parameter.
3. Find the parameter values such that the derivatives are zero.



The trickiest step is usually the last. In our example, it was trivial, but we will see that in many cases we need to resort to iterative solution algorithms or other numerical optimization techniques, as described in Chapter 4. The example also illustrates a significant problem with maximum-likelihood learning in general: *when the data set is small enough that some events have not yet been observed—for instance, no cherry candies—the maximum likelihood hypothesis assigns zero probability to those events.* Various tricks are used to avoid this problem, such as initializing the counts for each event to 1 instead of zero.



Let us look at another example. Suppose this new candy manufacturer wants to give a little hint to the consumer and uses candy wrappers colored red and green. The *Wrapper* for each candy is selected *probabilistically*, according to some unknown conditional distribution, depending on the flavor. The corresponding probability model is shown in Figure 20.2(b). Notice that it has three parameters: θ , θ_1 , and θ_2 . With these parameters, the likelihood of seeing, say, a cherry candy in a green wrapper can be obtained from the standard semantics for Bayesian networks (page 495):

$$\begin{aligned} P(Flavor = \text{cherry}, \text{Wrapper} = \text{green} | h_{\theta, \theta_1, \theta_2}) \\ = P(Flavor = \text{cherry} | h_{\theta, \theta_1, \theta_2}) P(\text{Wrapper} = \text{green} | Flavor = \text{cherry}, h_{\theta, \theta_1, \theta_2}) \\ = \theta \cdot (1 - \theta_1) . \end{aligned}$$

Now, we unwrap N candies, of which c are cherries and ℓ are limes. The wrapper counts are as follows: r_c of the cherries have red wrappers and g_c have green, while r_ℓ of the limes have red and g_ℓ have green. The likelihood of the data is given by

$$P(\mathbf{d}|h_{\theta,\theta_1,\theta_2}) = \theta^c(1-\theta)^\ell \cdot \theta_1^{r_c}(1-\theta_1)^{g_c} \cdot \theta_2^{r_\ell}(1-\theta_2)^{g_\ell}.$$

This looks pretty horrible, but taking logarithms helps:

$$L = [c \log \theta + \ell \log(1 - \theta)] + [r_c \log \theta_1 + g_c \log(1 - \theta_1)] + [r_\ell \log \theta_2 + g_\ell \log(1 - \theta_2)] .$$

The benefit of taking logs is clear: the log likelihood is the sum of three terms, each of which contains a single parameter. When we take derivatives with respect to each parameter and set

them to zero, we get three independent equations, each containing just one parameter:

$$\begin{aligned}\frac{\partial L}{\partial \theta} &= \frac{c}{\theta} - \frac{\ell}{1-\theta} = 0 & \Rightarrow \theta &= \frac{c}{c+\ell} \\ \frac{\partial L}{\partial \theta_1} &= \frac{r_c}{\theta_1} - \frac{g_c}{1-\theta_1} = 0 & \Rightarrow \theta_1 &= \frac{r_c}{r_c+g_c} \\ \frac{\partial L}{\partial \theta_2} &= \frac{r_\ell}{\theta_2} - \frac{g_\ell}{1-\theta_2} = 0 & \Rightarrow \theta_2 &= \frac{r_\ell}{r_\ell+g_\ell}.\end{aligned}$$

The solution for θ is the same as before. The solution for θ_1 , the probability that a cherry candy has a red wrapper, is the observed fraction of cherry candies with red wrappers, and similarly for θ_2 .

These results are very comforting, and it is easy to see that they can be extended to any Bayesian network whose conditional probabilities are represented as tables. The most important point is that, *with complete data, the maximum-likelihood parameter learning problem for a Bayesian network decomposes into separate learning problems, one for each parameter.*³ The second point is that the parameter values for a variable, given its parents, are just the observed frequencies of the variable values for each setting of the parent values. As before, we must be careful to avoid zeroes when the data set is small.



Naive Bayes models

Probably the most common Bayesian network model used in machine learning is the **naive Bayes** model. In this model, the “class” variable C (which is to be predicted) is the root and the “attribute” variables X_i are the leaves. The model is “naive” because it assumes that the attributes are conditionally independent of each other, given the class. (The model in Figure 20.2(b) is a naive Bayes model with just one attribute.) Assuming Boolean variables, the parameters are

$$\theta = P(C = \text{true}), \theta_{i1} = P(X_i = \text{true}|C = \text{true}), \theta_{i2} = P(X_i = \text{true}|C = \text{false}).$$

The maximum-likelihood parameter values are found in exactly the same way as for Figure 20.2(b). Once the model has been trained in this way, it can be used to classify new examples for which the class variable C is unobserved. With observed attribute values x_1, \dots, x_n , the probability of each class is given by

$$\mathbf{P}(C|x_1, \dots, x_n) = \alpha \mathbf{P}(C) \prod_i \mathbf{P}(x_i|C).$$

A deterministic prediction can be obtained by choosing the most likely class. Figure 20.3 shows the learning curve for this method when it is applied to the restaurant problem from Chapter 18. The method learns fairly well but not as well as decision-tree learning; this is presumably because the true hypothesis—which is a decision tree—is not representable exactly using a naive Bayes model. Naive Bayes learning turns out to do surprisingly well in a wide range of applications; the boosted version (Exercise 20.5) is one of the most effective general-purpose learning algorithms. Naive Bayes learning scales well to very large problems: with n Boolean attributes, there are just $2n + 1$ parameters, and *no search is required to find h_{ML} , the maximum-likelihood naive Bayes hypothesis.* Finally, naive Bayes learning has no difficulty with noisy data and can give probabilistic predictions when appropriate.



³ See Exercise 20.7 for the nontabulated case, where each parameter affects several conditional probabilities.

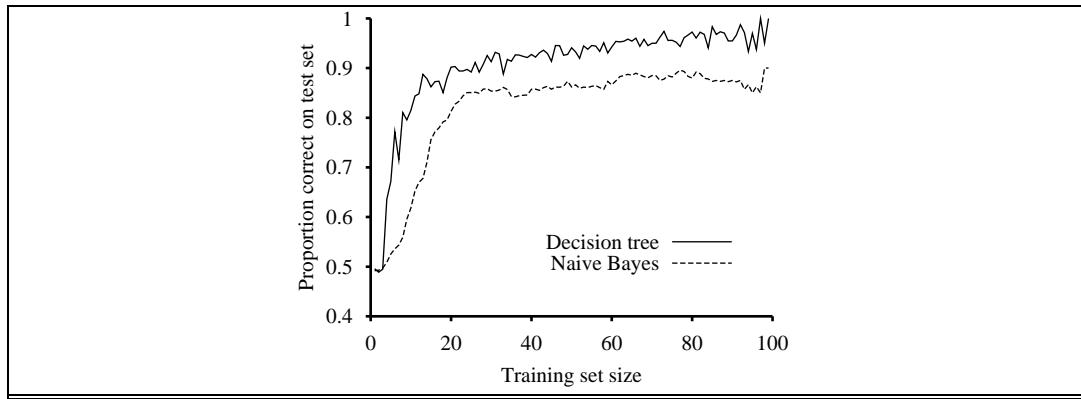


Figure 20.3 The learning curve for naive Bayes learning applied to the restaurant problem from Chapter 18; the learning curve for decision-tree learning is shown for comparison.

Maximum-likelihood parameter learning: Continuous models

Continuous probability models such as the **linear-Gaussian** model were introduced in Section 14.3. Because continuous variables are ubiquitous in real-world applications, it is important to know how to learn continuous models from data. The principles for maximum-likelihood learning are identical to those of the discrete case.

Let us begin with a very simple case: learning the parameters of a Gaussian density function on a single variable. That is, the data are generated as follows:

$$P(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

The parameters of this model are the mean μ and the standard deviation σ . (Notice that the normalizing “constant” depends on σ , so we cannot ignore it.) Let the observed values be x_1, \dots, x_N . Then the log likelihood is

$$L = \sum_{j=1}^N \log \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x_j-\mu)^2}{2\sigma^2}} = N(-\log \sqrt{2\pi} - \log \sigma) - \sum_{j=1}^N \frac{(x_j - \mu)^2}{2\sigma^2}.$$

Setting the derivatives to zero as usual, we obtain

$$\begin{aligned} \frac{\partial L}{\partial \mu} &= -\frac{1}{\sigma^2} \sum_{j=1}^N (x_j - \mu) = 0 & \Rightarrow \quad \mu &= \frac{\sum_j x_j}{N} \\ \frac{\partial L}{\partial \sigma} &= -\frac{N}{\sigma} + \frac{1}{\sigma^3} \sum_{j=1}^N (x_j - \mu)^2 = 0 & \Rightarrow \quad \sigma &= \sqrt{\frac{\sum_j (x_j - \mu)^2}{N}}. \end{aligned} \tag{20.4}$$

That is, the maximum-likelihood value of the mean is the sample average and the maximum-likelihood value of the standard deviation is the square root of the sample variance. Again, these are comforting results that confirm “commonsense” practice.

Now consider a linear Gaussian model with one continuous parent X and a continuous child Y . As explained on page 502, Y has a Gaussian distribution whose mean depends linearly on the value of X and whose standard deviation is fixed. To learn the conditional

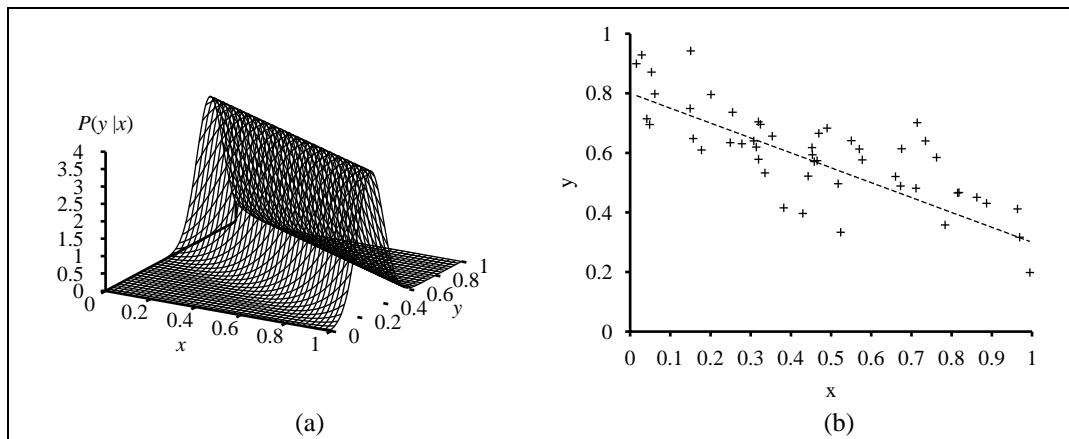


Figure 20.4 (a) A linear Gaussian model described as $y = \theta_1 x + \theta_2$ plus Gaussian noise with fixed variance. (b) A set of 50 data points generated from this model.

distribution $P(Y|X)$, we can maximize the conditional likelihood

$$P(y|x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(y-(\theta_1 x + \theta_2))^2}{2\sigma^2}}. \quad (20.5)$$

Here, the parameters are θ_1 , θ_2 , and σ . The data are a collection of (x_j, y_j) pairs, as illustrated in Figure 20.4. Using the usual methods (Exercise 20.6), we can find the maximum-likelihood values of the parameters. Here, we want to make a different point. If we consider just the parameters θ_1 and θ_2 that define the linear relationship between x and y , it becomes clear that maximizing the log likelihood with respect to these parameters is the same as *minimizing* the numerator in the exponent of Equation (20.5):

$$E = \sum_{j=1}^N (y_j - (\theta_1 x_j + \theta_2))^2.$$

The quantity $(y_j - (\theta_1 x_j + \theta_2))$ is the **error** for (x_j, y_j) —that is, the difference between the actual value y_j and the predicted value $(\theta_1 x_j + \theta_2)$ —so E is the well-known **sum of squared errors**. This is the quantity that is minimized by the standard **linear regression** procedure. Now we can understand why: minimizing the sum of squared errors gives the maximum-likelihood straight-line model, *provided that the data are generated with Gaussian noise of fixed variance.*

Bayesian parameter learning

Maximum-likelihood learning gives rise to some very simple procedures, but it has some serious deficiencies with small data sets. For example, after seeing one cherry candy, the maximum-likelihood hypothesis is that the bag is 100% cherry (i.e., $\theta = 1.0$). Unless one's hypothesis prior is that bags must be either all cherry or all lime, this is not a reasonable conclusion. The Bayesian approach to parameter learning places a hypothesis prior over the possible values of the parameters and updates this distribution as data arrive.

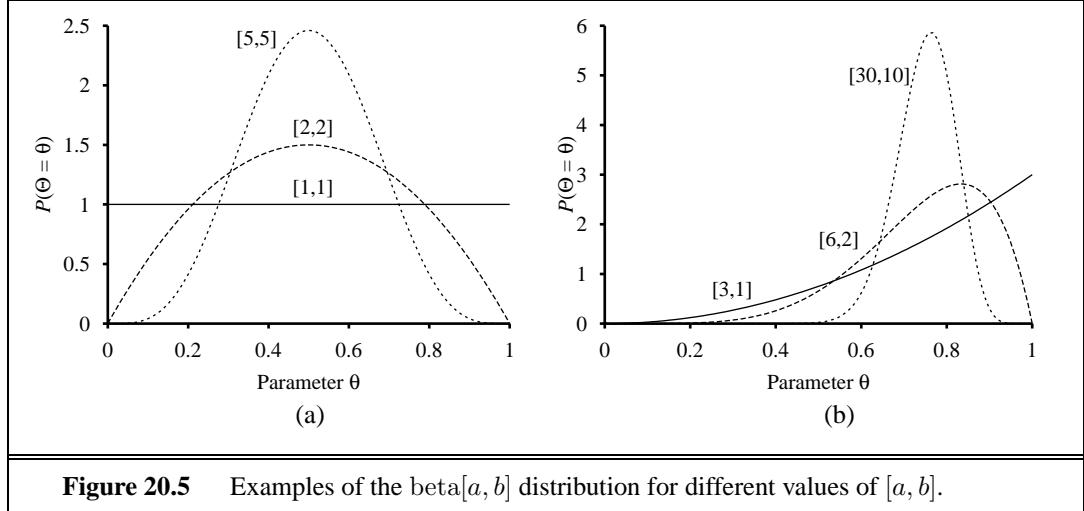


Figure 20.5 Examples of the beta $[a, b]$ distribution for different values of $[a, b]$.

The candy example in Figure 20.2(a) has one parameter, θ : the probability that a randomly selected piece of candy is cherry flavored. In the Bayesian view, θ is the (unknown) value of a random variable Θ ; the hypothesis prior is just the prior distribution $\mathbf{P}(\Theta)$. Thus, $P(\Theta = \theta)$ is the prior probability that the bag has a fraction θ of cherry candies.

If the parameter θ can be any value between 0 and 1, then $\mathbf{P}(\Theta)$ must be a continuous distribution that is nonzero only between 0 and 1 and that integrates to 1. The uniform density $P(\theta) = U[0, 1](\theta)$ is one candidate. (See Chapter 13.) It turns out that the uniform density is a member of the family of **beta distributions**. Each beta distribution is defined by two **hyperparameters**⁴ a and b such that

$$\text{beta}[a, b](\theta) = \alpha \theta^{a-1} (1 - \theta)^{b-1}, \quad (20.6)$$

for θ in the range $[0, 1]$. The normalization constant α depends on a and b . (See Exercise 20.8.) Figure 20.5 shows what the distribution looks like for various values of a and b . The mean value of the distribution is $a/(a + b)$, so larger values of a suggest a belief that Θ is closer to 1 than to 0. Larger values of $a + b$ make the distribution more peaked, suggesting greater certainty about the value of Θ . Thus, the beta family provides a useful range of possibilities for the hypothesis prior.

Besides its flexibility, the beta family has another wonderful property: if Θ has a prior beta $[a, b]$, then, after a data point is observed, the posterior distribution for Θ is also a beta distribution. The beta family is called the **conjugate prior** for the family of distributions for a Boolean variable.⁵ Let's see how this works. Suppose we observe a cherry candy; then

$$\begin{aligned} P(\theta | D_1 = \text{cherry}) &= \alpha P(D_1 = \text{cherry} | \theta) P(\theta) \\ &= \alpha' \theta \cdot \text{beta}[a, b](\theta) = \alpha' \theta \cdot \theta^{a-1} (1 - \theta)^{b-1} \\ &= \alpha' \theta^a (1 - \theta)^{b-1} = \text{beta}[a+1, b](\theta). \end{aligned}$$

⁴ They are called hyperparameters because they parameterize a distribution over θ , which is itself a parameter.

⁵ Other conjugate priors include the **Dirichlet** family for the parameters of a discrete multivalued distribution and the **Normal–Wishart** family for the parameters of a Gaussian distribution. See Bernardo and Smith (1994).

VIRTUAL COUNTS

Thus, after seeing a cherry candy, we simply increment the a parameter to get the posterior; similarly, after seeing a lime candy, we increment the b parameter. Thus, we can view the a and b hyperparameters as **virtual counts**, in the sense that a prior $\text{beta}[a, b]$ behaves exactly as if we had started out with a uniform prior $\text{beta}[1, 1]$ and seen $a - 1$ actual cherry candies and $b - 1$ actual lime candies.

PARAMETER INDEPENDENCE

By examining a sequence of beta distributions for increasing values of a and b , keeping the proportions fixed, we can see vividly how the posterior distribution over the parameter Θ changes as data arrive. For example, suppose the actual bag of candy is 75% cherry. Figure 20.5(b) shows the sequence $\text{beta}[3, 1]$, $\text{beta}[6, 2]$, $\text{beta}[30, 10]$. Clearly, the distribution is converging to a narrow peak around the true value of Θ . For large data sets, then, Bayesian learning (at least in this case) converges to give the same results as maximum-likelihood learning.

The network in Figure 20.2(b) has three parameters, θ , θ_1 , and θ_2 , where θ_1 is the probability of a red wrapper on a cherry candy and θ_2 is the probability of a red wrapper on a lime candy. The Bayesian hypothesis prior must cover all three parameters—that is, we need to specify $\mathbf{P}(\Theta, \Theta_1, \Theta_2)$. Usually, we assume **parameter independence**:

$$\mathbf{P}(\Theta, \Theta_1, \Theta_2) = \mathbf{P}(\Theta)\mathbf{P}(\Theta_1)\mathbf{P}(\Theta_2).$$

With this assumption, each parameter can have its own beta distribution that is updated separately as data arrive.

Once we have the idea that unknown parameters can be represented by random variables such as Θ , it is natural to incorporate them into the Bayesian network itself. To do this, we also need to make copies of the variables describing each instance. For example, if we have observed three candies then we need $Flavor_1$, $Flavor_2$, $Flavor_3$ and $Wrapper_1$, $Wrapper_2$, $Wrapper_3$. The parameter variable Θ determines the probability of each $Flavor_i$ variable:

$$P(Flavor_i = \text{cherry} | \Theta = \theta) = \theta.$$

Similarly, the wrapper probabilities depend on Θ_1 and Θ_2 . For example,

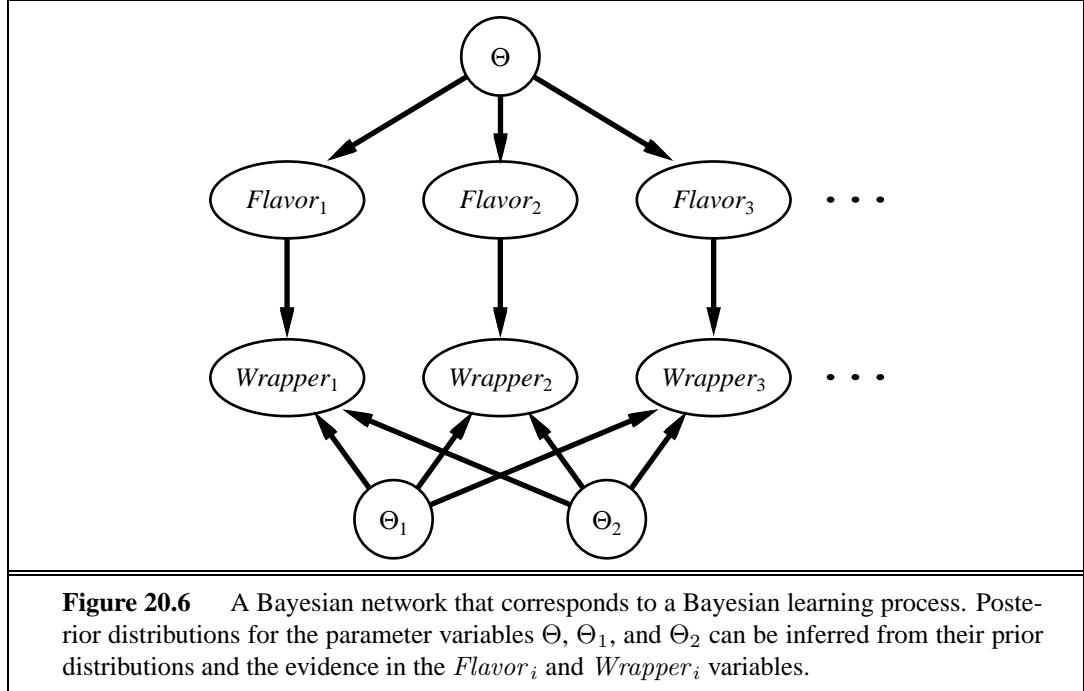
$$P(Wrapper_i = \text{red} | Flavor_i = \text{cherry}, \Theta_1 = \theta_1) = \theta_1.$$



Now, the entire Bayesian learning process can be formulated as an *inference* problem in a suitably constructed Bayes net, as shown in Figure 20.6. Prediction for a new instance is done simply by adding new instance variables to the network, some of which are queried. This formulation of learning and prediction makes it clear that Bayesian learning requires no extra “principles of learning.” Furthermore, *there is, in essence, just one learning algorithm*, i.e., the inference algorithm for Bayesian networks.

Learning Bayes net structures

So far, we have assumed that the structure of the Bayes net is given and we are just trying to learn the parameters. The structure of the network represents basic causal knowledge about the domain that is often easy for an expert, or even a naive user, to supply. In some cases, however, the causal model may be unavailable or subject to dispute—for example, certain corporations have long claimed that smoking does not cause cancer—so it is important to



understand how the structure of a Bayes net can be learned from data. At present, structural learning algorithms are in their infancy, so we will give only a brief sketch of the main ideas.

The most obvious approach is to *search* for a good model. We can start with a model containing no links and begin adding parents for each node, fitting the parameters with the methods we have just covered and measuring the accuracy of the resulting model. Alternatively, we can start with an initial guess at the structure and use hill-climbing or simulated annealing search to make modifications, retuning the parameters after each change in the structure. Modifications can include reversing, adding, or deleting arcs. We must not introduce cycles in the process, so many algorithms assume that an ordering is given for the variables, and that a node can have parents only among those nodes that come earlier in the ordering (just as in the construction process Chapter 14). For full generality, we also need to search over possible orderings.

There are two alternative methods for deciding when a good structure has been found. The first is to test whether the conditional independence assertions implicit in the structure are actually satisfied in the data. For example, the use of a naive Bayes model for the restaurant problem assumes that

$$\mathbf{P}(Fri/Sat, Bar | WillWait) = \mathbf{P}(Fri/Sat | WillWait)\mathbf{P}(Bar | WillWait)$$

and we can check in the data that the same equation holds between the corresponding conditional frequencies. Now, even if the structure describes the true causal nature of the domain, statistical fluctuations in the data set mean that the equation will never be satisfied *exactly*, so we need to perform a suitable statistical test to see if there is sufficient evidence that the independence hypothesis is violated. The complexity of the resulting network will depend

on the threshold used for this test—the stricter the independence test, the more links will be added and the greater the danger of overfitting.

An approach more consistent with the ideas in this chapter is to the degree to which the proposed model explains the data (in a probabilistic sense). We must be careful how we measure this, however. If we just try to find the maximum-likelihood hypothesis, we will end up with a fully connected network, because adding more parents to a node cannot decrease the likelihood (Exercise 20.9). We are forced to penalize model complexity in some way. The MAP (or MDL) approach simply subtracts a penalty from the likelihood of each structure (after parameter tuning) before comparing different structures. The Bayesian approach places a joint prior over structures and parameters. There are usually far too many structures to sum over (superexponential in the number of variables), so most practitioners use MCMC to sample over structures.

Penalizing complexity (whether by MAP or Bayesian methods) introduces an important connection between the optimal structure and the nature of the representation for the conditional distributions in the network. With tabular distributions, the complexity penalty for a node’s distribution grows exponentially with the number of parents, but with, say, noisy-OR distributions, it grows only linearly. This means that learning with noisy-OR (or other compactly parameterized) models tends to produce learned structures with more parents than does learning with tabular distributions.

20.3 LEARNING WITH HIDDEN VARIABLES: THE EM ALGORITHM

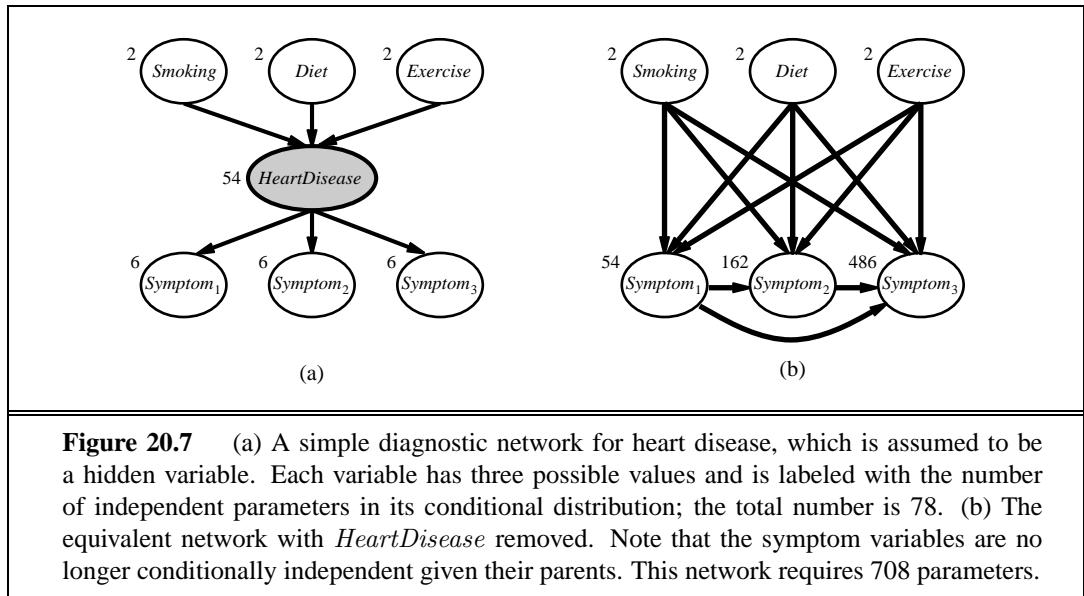
LATENT VARIABLES

The preceding section dealt with the fully observable case. Many real-world problems have **hidden variables** (sometimes called **latent variables**) which are not observable in the data that are available for learning. For example, medical records often include the observed symptoms, the treatment applied, and perhaps the outcome of the treatment, but they seldom contain a direct observation of the disease itself!⁶ One might ask, “If the disease is not observed, why not construct a model without it?” The answer appears in Figure 20.7, which shows a small, fictitious diagnostic model for heart disease. There are three observable predisposing factors and three observable symptoms (which are too depressing to name). Assume that each variable has three possible values (e.g., *none*, *moderate*, and *severe*). Removing the hidden variable from the network in (a) yields the network in (b); the total number of parameters increases from 78 to 708. Thus, *latent variables can dramatically reduce the number of parameters required to specify a Bayesian network*. This, in turn, can dramatically reduce the amount of data needed to learn the parameters.



Hidden variables are important, but they do complicate the learning problem. In Figure 20.7(a), for example, it is not obvious how to learn the conditional distribution for *HeartDisease*, given its parents, because we do not know the value of *HeartDisease* in each case; the same problem arises in learning the distributions for the symptoms. This section

⁶ Some records contain the diagnosis suggested by the physician, but this is a causal consequence of the symptoms, which are in turn caused by the disease.



EXPECTATION-MAXIMIZATION

describes an algorithm called **expectation–maximization**, or EM, that solves this problem in a very general way. We will show three examples and then provide a general description. The algorithm seems like magic at first, but once the intuition has been developed, one can find applications for EM in a huge range of learning problems.

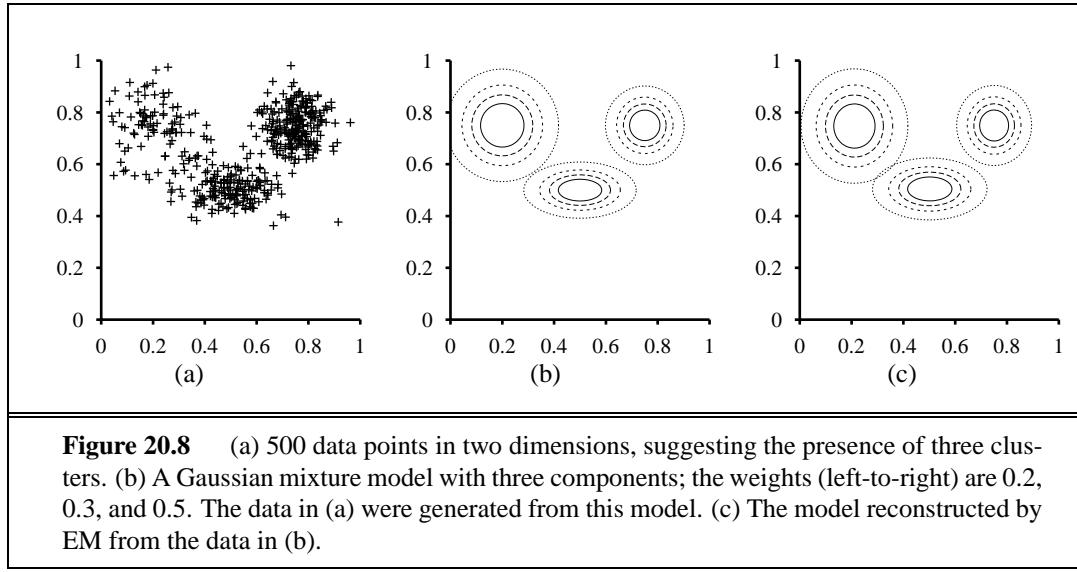
UNSUPERVISED CLUSTERING

Unsupervised clustering: Learning mixtures of Gaussians

Unsupervised clustering is the problem of discerning multiple categories in a collection of objects. The problem is unsupervised because the category labels are not given. For example, suppose we record the spectra of a hundred thousand stars; are there different *types* of stars revealed by the spectra, and, if so, how many and what are their characteristics? We are all familiar with terms such as “red giant” and “white dwarf,” but the stars do not carry these labels on their hats—astronomers had to perform unsupervised clustering to identify these categories. Other examples include the identification of species, genera, orders, and so on in the Linnæan taxonomy of organisms and the creation of natural kinds to categorize ordinary objects (see Chapter 10).

MIXTURE DISTRIBUTION COMPONENT

Unsupervised clustering begins with data. Figure 20.8(a) shows 500 data points, each of which specifies the values of two continuous attributes. The data points might correspond to stars, and the attributes might correspond to spectral intensities at two particular frequencies. Next, we need to understand what kind of probability distribution might have generated the data. Clustering presumes that the data are generated from a **mixture distribution**, P . Such a distribution has k **components**, each of which is a distribution in its own right. A data point is generated by first choosing a component and then generating a sample from that component. Let the random variable C denote the component, with values $1, \dots, k$; then the mixture



distribution is given by

$$P(\mathbf{x}) = \sum_{i=1}^k P(C=i) P(\mathbf{x}|C=i),$$

where \mathbf{x} refers to the values of the attributes for a data point. For continuous data, a natural choice for the component distributions is the multivariate Gaussian, which gives the so-called **mixture of Gaussians** family of distributions. The parameters of a mixture of Gaussians are $w_i = P(C=i)$ (the weight of each component), μ_i (the mean of each component), and Σ_i (the covariance of each component). Figure 20.8(b) shows a mixture of three Gaussians; this mixture is in fact the source of the data in (a).

The unsupervised clustering problem, then, is to recover a mixture model like the one in Figure 20.8(b) from raw data like that in Figure 20.8(a). Clearly, if we *knew* which component generated each data point, then it would be easy to recover the component Gaussians: we could just select all the data points from a given component and then apply (a multivariate version of) Equation (20.4) for fitting the parameters of a Gaussian to a set of data. On the other hand, if we *knew* the parameters of each component, then we could, at least in a probabilistic sense, assign each data point to a component. The problem is that we know neither the assignments nor the parameters.

The basic idea of EM in this context is to *pretend* that we know the parameters of the model and then to infer the probability that each data point belongs to each component. After that, we refit the components to the data, where each component is fitted to the entire data set with each point weighted by the probability that it belongs to that component. The process iterates until convergence. Essentially, we are “completing” the data by inferring probability distributions over the hidden variables—which component each data point belongs to—based on the current model. For the mixture of Gaussians, we initialize the mixture model parameters arbitrarily and then iterate the following two steps:

1. E-step: Compute the probabilities $p_{ij} = P(C = i|\mathbf{x}_j)$, the probability that datum \mathbf{x}_j was generated by component i . By Bayes' rule, we have $p_{ij} = \alpha P(\mathbf{x}_j|C = i)P(C = i)$. The term $P(\mathbf{x}_j|C = i)$ is just the probability at \mathbf{x}_j of the i th Gaussian, and the term $P(C = i)$ is just the weight parameter for the i th Gaussian. Define $p_i = \sum_j p_{ij}$.
2. M-step: Compute the new mean, covariance, and component weights as follows:

$$\begin{aligned}\boldsymbol{\mu}_i &\leftarrow \sum_j p_{ij} \mathbf{x}_j / p_i \\ \boldsymbol{\Sigma}_i &\leftarrow \sum_j p_{ij} \mathbf{x}_j \mathbf{x}_j^\top / p_i \\ w_i &\leftarrow p_i.\end{aligned}$$

The E-step, or *expectation step*, can be viewed as computing the expected values p_{ij} of the hidden **indicator variables** Z_{ij} , where Z_{ij} is 1 if datum \mathbf{x}_j was generated by the i th component and 0 otherwise. The M-step, or *maximization step*, finds the new values of the parameters that maximize the log likelihood of the data, given the expected values of the hidden indicator variables.

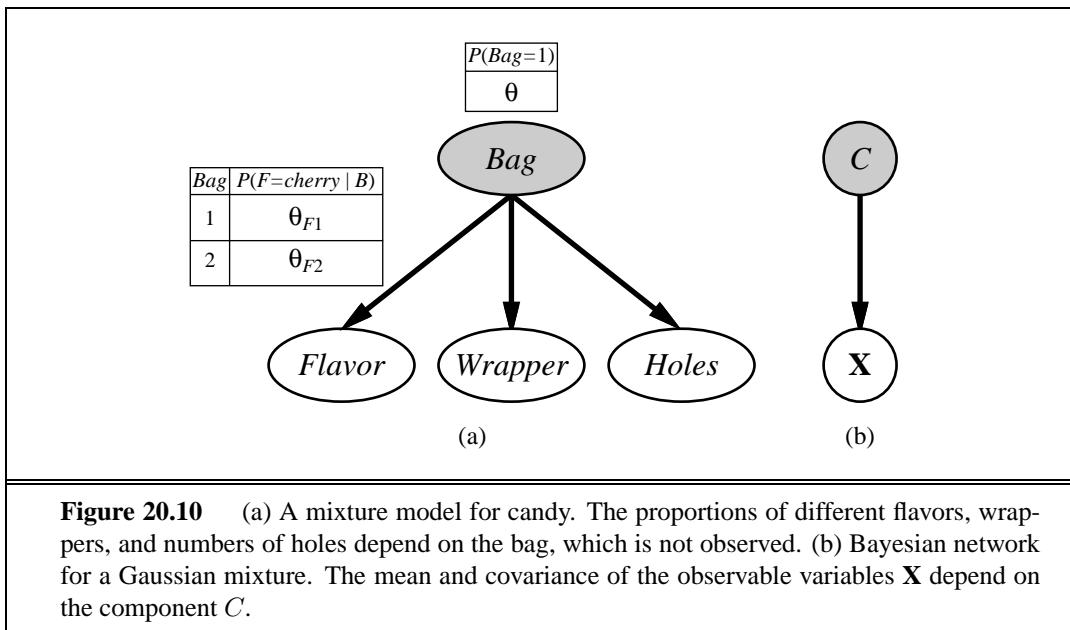
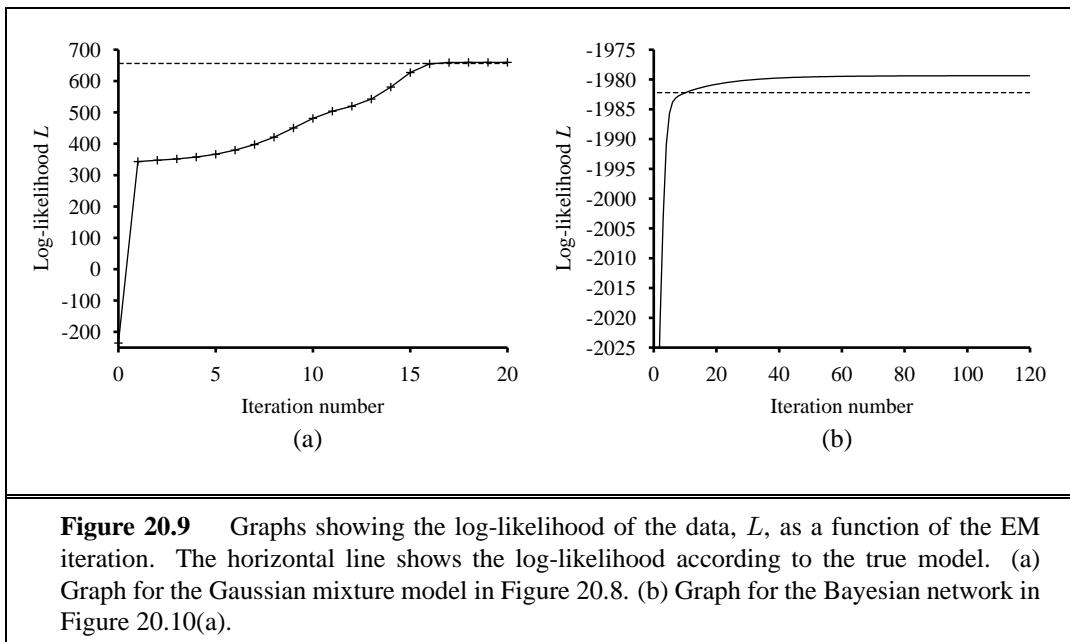


The final model that EM learns when it is applied to the data in Figure 20.8(a) is shown in Figure 20.8(c); it is virtually indistinguishable from the original model from which the data were generated. Figure 20.9(a) plots the log likelihood of the data according to the current model as EM progresses. There are two points to notice. First, the log likelihood for the final learned model slightly *exceeds* that of the original model, from which the data were generated. This might seem surprising, but it simply reflects the fact that the data were generated randomly and might not provide an exact reflection of the underlying model. The second point is that *EM increases the log likelihood of the data at every iteration*. This fact can be proved in general. Furthermore, under certain conditions, EM can be proven to reach a local maximum in likelihood. (In rare cases, it could reach a saddle point or even a local minimum.) In this sense, EM resembles a gradient-based hill-climbing algorithm, but notice that it has no “step size” parameter!

Things do not always go as well as Figure 20.9(a) might suggest. It can happen, for example, that one Gaussian component shrinks so that it covers just a single data point. Then its variance will go to zero and its likelihood will go to infinity! Another problem is that two components can “merge,” acquiring identical means and variances and sharing their data points. These kinds of degenerate local maxima are serious problems, especially in high dimensions. One solution is to place priors on the model parameters and to apply the MAP version of EM. Another is to restart a component with new random parameters if it gets too small or too close to another component. It also helps to initialize the parameters with reasonable values.

Learning Bayesian networks with hidden variables

To learn a Bayesian network with hidden variables, we apply the same insights that worked for mixtures of Gaussians. Figure 20.10 represents a situation in which there are two bags of candies that have been mixed together. Candies are described by three features: in addition to the *Flavor* and the *Wrapper*, some candies have a *Hole* in the middle and some do not.



The distribution of candies in each bag is described by a **naive Bayes** model: the features are independent, given the bag, but the conditional probability distribution for each feature depends on the bag. The parameters are as follows: θ is the prior probability that a candy comes from Bag 1; θ_{F1} and θ_{F2} are the probabilities that the flavor is cherry, given that the candy comes from Bag 1 and Bag 2 respectively; θ_{W1} and θ_{W2} give the probabilities that the wrapper is red; and θ_{H1} and θ_{H2} give the probabilities that the candy has a hole. Notice that

the overall model is a mixture model. (In fact, we can also model the mixture of Gaussians as a Bayesian network, as shown in Figure 20.10(b).) In the figure, the bag is a hidden variable because, once the candies have been mixed together, we no longer know which bag each candy came from. In such a case, can we recover the descriptions of the two bags by observing candies from the mixture?

Let us work through an iteration of EM for this problem. First, let's look at the data. We generated 1000 samples from a model whose true parameters are

$$\theta = 0.5, \theta_{F1} = \theta_{W1} = \theta_{H1} = 0.8, \theta_{F2} = \theta_{W2} = \theta_{H2} = 0.3 . \quad (20.7)$$

That is, the candies are equally likely to come from either bag; the first is mostly cherries with red wrappers and holes; the second is mostly limes with green wrappers and no holes. The counts for the eight possible kinds of candy are as follows:

	$W = \text{red}$		$W = \text{green}$	
	$H = 1$	$H = 0$	$H = 1$	$H = 0$
$F = \text{cherry}$	273	93	104	90
$F = \text{lime}$	79	100	94	167

We start by initializing the parameters. For numerical simplicity, we will choose⁷

$$\theta^{(0)} = 0.6, \theta_{F1}^{(0)} = \theta_{W1}^{(0)} = \theta_{H1}^{(0)} = 0.6, \theta_{F2}^{(0)} = \theta_{W2}^{(0)} = \theta_{H2}^{(0)} = 0.4 . \quad (20.8)$$

First, let us work on the θ parameter. In the fully observable case, we would estimate this directly from the *observed* counts of candies from bags 1 and 2. Because the bag is a hidden variable, we calculate the *expected* counts instead. The expected count $\hat{N}(Bag = 1)$ is the sum, over all candies, of the probability that the candy came from bag 1:

$$\theta^{(1)} = \hat{N}(Bag = 1)/N = \sum_{j=1}^N P(Bag = 1 | flavor_j, wrapper_j, holes_j)/N .$$

These probabilities can be computed by any inference algorithm for Bayesian networks. For a naive Bayes model such as the one in our example, we can do the inference “by hand,” using Bayes’ rule and applying conditional independence:

$$\theta^{(1)} = \frac{1}{N} \sum_{j=1}^N \frac{P(flavor_j | Bag = 1)P(wrapper_j | Bag = 1)P(holes_j | Bag = 1)P(Bag = 1)}{\sum_i P(flavor_j | Bag = i)P(wrapper_j | Bag = i)P(holes_j | Bag = i)P(Bag = i)} .$$

(Notice that the normalizing constant also depends on the parameters.) Applying this formula to, say, the 273 red-wrapped cherry candies with holes, we get a contribution of

$$\frac{273}{1000} \cdot \frac{\theta_{F1}^{(0)} \theta_{W1}^{(0)} \theta_{H1}^{(0)} \theta^{(0)}}{\theta_{F1}^{(0)} \theta_{W1}^{(0)} \theta_{H1}^{(0)} \theta^{(0)} + \theta_{F2}^{(0)} \theta_{W2}^{(0)} \theta_{H2}^{(0)} (1 - \theta^{(0)})} \approx 0.22797 .$$

Continuing with the other seven kinds of candy in the table of counts, we obtain $\theta^{(1)} = 0.6124$.

⁷ It is better in practice to choose them randomly, to avoid local maxima due to symmetry.

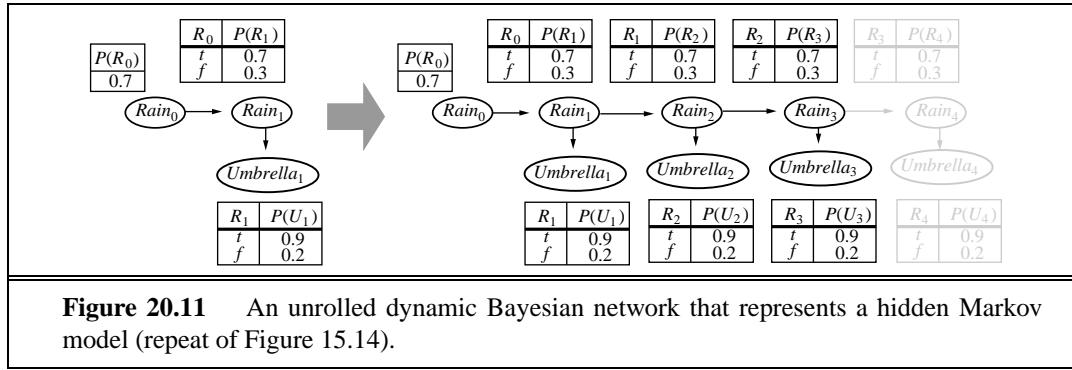


Figure 20.11 An unrolled dynamic Bayesian network that represents a hidden Markov model (repeat of Figure 15.14).

Now let us consider the other parameters, such as θ_{F1} . In the fully observable case, we would estimate this directly from the *observed* counts of cherry and lime candies from bag 1. The *expected* count of cherry candies from bag 1 is given by

$$\sum_{j: Flavor_j = \text{cherry}} P(\text{Bag} = 1 | Flavor_j = \text{cherry}, wrapper_j, holes_j) .$$

Again, these probabilities can be calculated by any Bayes net algorithm. Completing this process, we obtain the new values of all the parameters:

$$\begin{aligned} \theta^{(1)} &= 0.6124, \quad \theta_{F1}^{(1)} = 0.6684, \quad \theta_{W1}^{(1)} = 0.6483, \quad \theta_{H1}^{(1)} = 0.6558, \\ \theta_{F2}^{(1)} &= 0.3887, \quad \theta_{W2}^{(1)} = 0.3817, \quad \theta_{H2}^{(1)} = 0.3827 . \end{aligned} \quad (20.9)$$

The log likelihood of the data increases from about -2044 initially to about -2021 after the first iteration, as shown in Figure 20.9(b). That is, the update improves the likelihood itself by a factor of about $e^{23} \approx 10^{10}$. By the tenth iteration, the learned model is a better fit than the original model ($L = -1982.214$). Thereafter, progress becomes very slow. This is not uncommon with EM, and many practical systems combine EM with a gradient-based algorithm such as Newton–Raphson (see Chapter 4) for the last phase of learning.



The general lesson from this example is that *the parameter updates for Bayesian network learning with hidden variables are directly available from the results of inference on each example. Moreover, only local posterior probabilities are needed for each parameter.* For the general case in which we are learning the conditional probability parameters for each variable X_i , given its parents—that is, $\theta_{ijk} = P(X_i = x_{ij} | Pa_i = pa_{ik})$ —the update is given by the normalized expected counts as follows:

$$\theta_{ijk} \leftarrow \hat{N}(X_i = x_{ij}, Pa_i = pa_{ik}) / \hat{N}(Pa_i = pa_{ik}) .$$

The expected counts are obtained by summing over the examples, computing the probabilities $P(X_i = x_{ij}, Pa_i = pa_{ik})$ for each by using any Bayes net inference algorithm. For the exact algorithms—including variable elimination—all these probabilities are obtainable directly as a by-product of standard inference, with no need for extra computations specific to learning. Moreover, the information needed for learning is available *locally* for each parameter.

Learning hidden Markov models

Our final application of EM involves learning the transition probabilities in hidden Markov models (HMMs). Recall from Chapter 15 that a hidden Markov model can be represented by a dynamic Bayes net with a single discrete state variable, as illustrated in Figure 20.11. Each data point consists of an observation *sequence* of finite length, so the problem is to learn the transition probabilities from a set of observation sequences (or possibly from just one long sequence).

We have already worked out how to learn Bayes nets, but there is one complication: in Bayes nets, each parameter is distinct; in a hidden Markov model, on the other hand, the individual transition probabilities from state i to state j at time t , $\theta_{ijt} = P(X_{t+1} = j | X_t = i)$, are *repeated* across time—that is, $\theta_{ijt} = \theta_{ij}$ for all t . To estimate the transition probability from state i to state j , we simply calculate the expected proportion of times that the system undergoes a transition to state j when in state i :

$$\hat{\theta}_{ij} \leftarrow \sum_t \hat{N}(X_{t+1} = j, X_t = i) / \sum_t \hat{N}(X_t = i).$$

Again, the expected counts are computed by any HMM inference algorithm. The **forward-backward** algorithm shown in Figure 15.4 can be modified very easily to compute the necessary probabilities. One important point is that the probabilities required are those obtained by **smoothing** rather than **filtering**; that is, we need to pay attention to subsequent evidence in estimating the probability that a particular transition occurred. As we said in Chapter 15, the evidence in a murder case is usually obtained *after* the crime (i.e., the transition from state i to state j) occurs.

The general form of the EM algorithm

We have seen several instances of the EM algorithm. Each involves computing expected values of hidden variables for each example and then recomputing the parameters, using the expected values as if they were observed values. Let \mathbf{x} be all the observed values in all the examples, let \mathbf{Z} denote all the hidden variables for all the examples, and let $\boldsymbol{\theta}$ be all the parameters for the probability model. Then the EM algorithm is

$$\boldsymbol{\theta}^{(i+1)} = \operatorname{argmax}_{\boldsymbol{\theta}} \sum_{\mathbf{z}} P(\mathbf{Z} = \mathbf{z} | \mathbf{x}, \boldsymbol{\theta}^{(i)}) L(\mathbf{x}, \mathbf{Z} = \mathbf{z} | \boldsymbol{\theta}).$$

This equation is the EM algorithm in a nutshell. The E-step is the computation of the summation, which is the expectation of the log likelihood of the “completed” data with respect to the distribution $P(\mathbf{Z} = \mathbf{z} | \mathbf{x}, \boldsymbol{\theta}^{(i)})$, which is the posterior over the hidden variables, given the data. The M-step is the maximization of this expected log likelihood with respect to the parameters. For mixtures of Gaussians, the hidden variables are the Z_{ij} s, where Z_{ij} is 1 if example j was generated by component i . For Bayes nets, the hidden variables are the values of the unobserved variables for each example. For HMMs, the hidden variables are the $i \rightarrow j$ transitions. Starting from the general form, it is possible to derive an EM algorithm for a specific application once the appropriate hidden variables have been identified.

As soon as we understand the general idea of EM, it becomes easy to derive all sorts of variants and improvements. For example, in many cases the E-step—the computation of

postiors over the hidden variables—is intractable, as in large Bayes nets. It turns out that one can use an *approximate* E-step and still obtain an effective learning algorithm. With a sampling algorithm such as MCMC (see Section 14.5), the learning process is very intuitive: each state (configuration of hidden and observed variables) visited by MCMC is treated exactly as if it were a complete observation. Thus, the parameters can be updated directly after each MCMC transition. Other forms of approximate inference, such as variational and loopy methods, have also proven effective for learning very large networks.

Learning Bayes net structures with hidden variables

In Section 20.2, we discussed the problem of learning Bayes net structures with complete data. When hidden variables are taken into consideration, things get more difficult. In the simplest case, the hidden variables are listed along with the observed variables; although their values are not observed, the learning algorithm is told that they exist and must find a place for them in the network structure. For example, an algorithm might try to learn the structure shown in Figure 20.7(a), given the information that *HeartDisease* (a three-valued variable) should be included in the model. If the learning algorithm is not told this information, then there are two choices: either pretend that the data is really complete—which forces the algorithm to learn the parameter-intensive model in Figure 20.7(b)—or *invent* new hidden variables in order to simplify the model. The latter approach can be implemented by including new modification choices in the structure search: in addition to modifying links, the algorithm can add or delete a hidden variable or change its arity. Of course, the algorithm will not know that the new variable it has invented is called *HeartDisease*; nor will it have meaningful names for the values. Fortunately, newly invented hidden variables will usually be connected to pre-existing variables, so a human expert can often inspect the local conditional distributions involving the new variable and ascertain its meaning.

As in the complete-data case, pure maximum-likelihood structure learning will result in a completely connected network (moreover, one with no hidden variables), so some form of complexity penalty is required. We can also apply MCMC to approximate Bayesian learning. For example, we can learn mixtures of Gaussians with an unknown number of components by sampling over the number; the approximate posterior distribution for the number of Gaussians is given by the sampling frequencies of the MCMC process.

So far, the process we have discussed has an outer loop that is a structural search process and an inner loop that is a parametric optimization process. For the complete-data case, the inner loop is very fast—just a matter of extracting conditional frequencies from the data set. When there are hidden variables, the inner loop may involve many iterations of EM or a gradient-based algorithm, and each iteration involves the calculation of posteriors in a Bayes net, which is itself an NP-hard problem. To date, this approach has proved impractical for learning complex models. One possible improvement is the so-called **structural EM** algorithm, which operates in much the same way as ordinary (parametric) EM except that the algorithm can update the structure as well as the parameters. Just as ordinary EM uses the current parameters to compute the expected counts in the E-step and then applies those counts in the M-step to choose new parameters, structural EM uses the current structure to compute

expected counts and then applies those counts in the M-step to evaluate the likelihood for potential new structures. (This contrasts with the outer-loop/inner-loop method, which computes new expected counts for each potential structure.) In this way, structural EM may make several structural alterations to the network without once recomputing the expected counts, and is capable of learning nontrivial Bayes net structures. Nonetheless, much work remains to be done before we can say that the structure learning problem is solved.

20.4 INSTANCE-BASED LEARNING

PARAMETRIC
LEARNING

NONPARAMETRIC
LEARNING

INSTANCE-BASED
LEARNING

NEAREST-NEIGHBOR

So far, our discussion of statistical learning has focused primarily on fitting the parameters of a *restricted* family of probability models to an *unrestricted* data set. For example, unsupervised clustering using mixtures of Gaussians assumes that the data are explained by the *sum* of a *fixed* number of *Gaussian* distributions. We call such methods **parametric learning**. Parametric learning methods are often simple and effective, but assuming a particular restricted family of models often oversimplifies what's happening in the real world, from where the data come. Now, it is true when we have very little data, we cannot hope to learn a complex and detailed model, but it seems silly to keep the hypothesis complexity fixed even when the data set grows very large!

In contrast to parametric learning, **nonparametric learning** methods allow the hypothesis complexity to grow with the data. The more data we have, the wigglier the hypothesis can be. We will look at two very simple families of nonparametric **instance-based learning** (or **memory-based learning**) methods, so called because they construct hypotheses directly from the training instances themselves.

Nearest-neighbor models

The key idea of **nearest-neighbor** models is that the properties of any particular input point \mathbf{x} are likely to be similar to those of points in the neighborhood of \mathbf{x} . For example, if we want to do **density estimation**—that is, estimate the value of an unknown probability density at \mathbf{x} —then we can simply measure the density with which points are scattered in the neighborhood of \mathbf{x} . This sounds very simple, until we realize that we need to specify exactly what we mean by “neighborhood.” If the neighborhood is too small, it won’t contain any data points; too large, and it may include *all* the data points, resulting in a density estimate that is the same everywhere. One solution is to define the neighborhood to be just big enough to include k points, where k is large enough to ensure a meaningful estimate. For fixed k , the size of the neighborhood varies—where data are sparse, the neighborhood is large, but where data are dense, the neighborhood is small. Figure 20.12(a) shows an example for data scattered in two dimensions. Figure 20.13 shows the results of k -nearest-neighbor density estimation from these data with $k = 3, 10$, and 40 respectively. For $k = 3$, the density estimate at any point is based on only 3 neighboring points and is highly variable. For $k = 40$, the estimate provides a good reconstruction of the true density shown in Figure 20.12(b). For $k = 40$, the neighborhood becomes too large and structure of the data is altogether lost. In practice, using

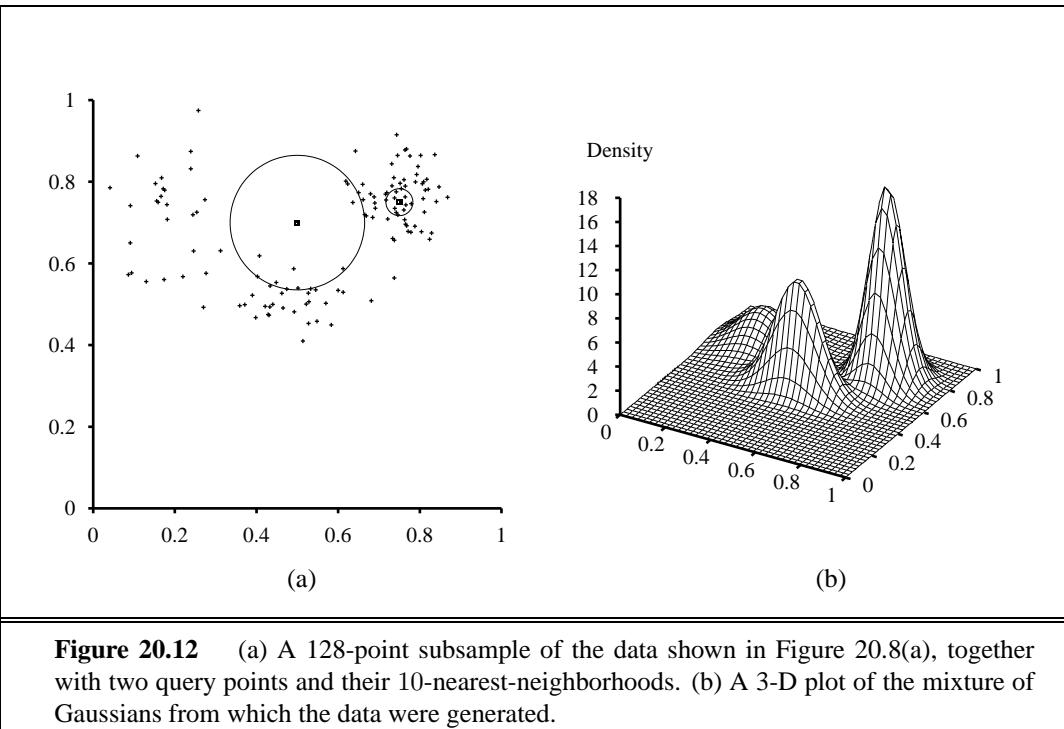


Figure 20.12 (a) A 128-point subsample of the data shown in Figure 20.8(a), together with two query points and their 10-nearest-neighborhoods. (b) A 3-D plot of the mixture of Gaussians from which the data were generated.

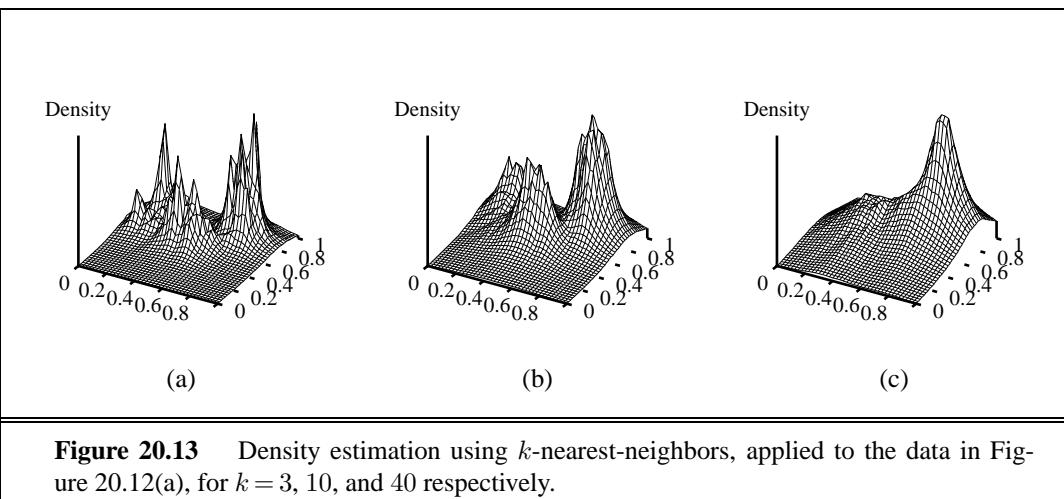


Figure 20.13 Density estimation using k -nearest-neighbors, applied to the data in Figure 20.12(a), for $k = 3, 10$, and 40 respectively.

a value of k somewhere between 5 and 10 gives good results for most low-dimensional data sets. A good value of k can also be chosen by using cross-validation.

To identify the nearest neighbors of a query point, we need a distance metric, $D(\mathbf{x}_1, \mathbf{x}_2)$. The two-dimensional example in Figure 20.12 uses Euclidean distance. This is inappropriate when each dimension of the space is measuring something different—for example, height and weight—because changing the scale of one dimension would change the set of nearest neighbors. One solution is to standardize the scale for each dimension. To do this, we measure

MAHALANOBIS
DISTANCE
HAMMING DISTANCE

the standard deviation of each feature over the whole data set and express feature values as multiples of the standard deviation for that feature. (This is a special case of the **Mahalanobis distance**, which takes into account the covariance of the features as well.) Finally, for discrete features we can use the **Hamming distance**, which defines $D(\mathbf{x}_1, \mathbf{x}_2)$ to be the number of features on which \mathbf{x}_1 and \mathbf{x}_2 differ.

Density estimates like those shown in Figure 20.13 define joint distributions over the input space. Unlike a Bayesian network, however, an instance-based representation cannot contain hidden variables, which means that we cannot perform unsupervised clustering as we did with the mixture-of-Gaussians model. We can still use the density estimate to predict a target value y given input feature values \mathbf{x} by calculating $P(y|\mathbf{x}) = P(y, \mathbf{x})/P(\mathbf{x})$, provided that the training data include values for the target feature.

It is also possible to use the nearest-neighbor idea for direct supervised learning. Given a test example with input \mathbf{x} , the output $y = h(\mathbf{x})$ is obtained from the y -values of the k nearest neighbors of \mathbf{x} . In the discrete case, we can obtain a single prediction by majority vote. In the continuous case, we can average the k values or do local linear regression, fitting a hyperplane to the k points and predicting the value at \mathbf{x} according to the hyperplane.

The k -nearest-neighbor learning algorithm is very simple to implement, requires little in the way of tuning, and often performs quite well. It is a good thing to try first on a new learning problem. For large data sets, however, we require an efficient mechanism for finding the nearest neighbors of a query point \mathbf{x} —simply calculating the distance to every point would take far too long. A variety of ingenious methods have been proposed to make this step efficient by preprocessing the training data. Unfortunately, most of these methods do not scale well with the dimension of the space (i.e., the number of features).

High-dimensional spaces pose an additional problem, namely that nearest neighbors in such spaces are usually a long way away! Consider a data set of size N in the d -dimensional unit hypercube, and assume hypercubic neighborhoods of side b and volume b^d . (The same argument works with hyperspheres, but the formula for the volume of a hypersphere is more complicated.) To contain k points, the average neighborhood must occupy a fraction k/N of the entire volume, which is 1. Hence, $b^d = k/N$, or $b = (k/N)^{1/d}$. So far, so good. Now let the number of features d be 100 and let k be 10 and N be 1,000,000. Then we have $b \approx 0.89$ —that is, the neighborhood has to span almost the entire input space! This suggests that nearest-neighbor methods cannot be trusted for high-dimensional data. In low dimensions there is no problem; with $d = 2$ we have $b = 0.003$.

Kernel models

KERNEL MODEL
KERNEL FUNCTION

In a **kernel model**, we view each training instance as generating a little density function—a **kernel function**—of its own. The density estimate as a whole is just the normalized sum of all the little kernel functions. A training instance at \mathbf{x}_i will generate a kernel function $K(\mathbf{x}, \mathbf{x}_i)$ that assigns a probability to each point \mathbf{x} in the space. Thus, the density estimate is

$$P(\mathbf{x}) = \frac{1}{N} \sum_{i=1}^N K(\mathbf{x}, \mathbf{x}_i) .$$

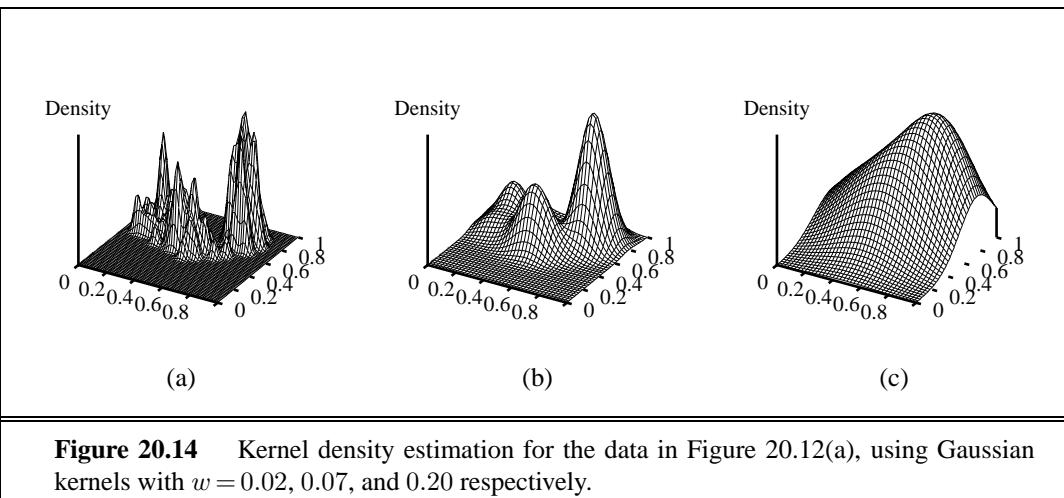


Figure 20.14 Kernel density estimation for the data in Figure 20.12(a), using Gaussian kernels with $w = 0.02, 0.07$, and 0.20 respectively.

The kernel function normally depends only on the *distance* $D(\mathbf{x}, \mathbf{x}_i)$ from \mathbf{x} to the instance \mathbf{x}_i . The most popular kernel function is (of course) the Gaussian. For simplicity, we will assume spherical Gaussians with standard deviation w along each axis, i.e.,

$$K(\mathbf{x}, \mathbf{x}_i) = \frac{1}{(w^2\sqrt{2\pi})^d} e^{-\frac{D(\mathbf{x}, \mathbf{x}_i)^2}{2w^2}},$$

where d is the number of dimensions in \mathbf{x} . We still have the problem of choosing a suitable value for w ; as before, making the neighborhood too small gives a very spiky estimate—see Figure 20.14(a). In (b), a medium value of w gives a very good reconstruction. In (c), too large a neighborhood results in losing the structure altogether. A good value of w can be chosen by using cross-validation.

Supervised learning with kernels is done by taking a *weighted* combination of *all* the predictions from the training instances. (Compare this with k -nearest-neighbor prediction, which takes an unweighted combination of the nearest k instances.) The weight of the i th instance for a query point \mathbf{x} is given by the value of the kernel $K(\mathbf{x}, \mathbf{x}_i)$. For a discrete prediction, we can take a weighted vote; for a continuous prediction, we can take weighted average or a weighted linear regression. Notice that making predictions with kernels requires looking at *every* training instance. It is possible to combine kernels with nearest-neighbor indexing schemes to make weighted predictions from just the nearby instances.

20.5 NEURAL NETWORKS

A **neuron** is a cell in the brain whose principal function is the collection, processing, and dissemination of electrical signals. Figure 1.2 on page 11 showed a schematic diagram of a typical neuron. The brain's information-processing capacity is thought to emerge primarily from *networks* of such neurons. For this reason, some of the earliest AI work aimed to create artificial **neural networks**. (Other names for the field include **connectionism**, **parallel distributed processing**, and **cybernetics**.)

tributed processing, and neural computation.) Figure 20.15 shows a simple mathematical model of the neuron devised by McCulloch and Pitts (1943). Roughly speaking, it “fires” when a linear combination of its inputs exceeds some threshold. Since 1943, much more detailed and realistic models have been developed, both for neurons and for larger systems in the brain, leading to the modern field of **computational neuroscience**. On the other hand, researchers in AI and statistics became interested in the more abstract properties of neural networks, such as their ability to perform distributed computation, to tolerate noisy inputs, and to learn. Although we understand now that other kinds of systems—including Bayesian networks—have these properties, neural networks remain one of the most popular and effective forms of learning system and are worthy of study in their own right.

Units in neural networks

UNITS
LINKS
ACTIVATION
WEIGHT

ACTIVATION
FUNCTION

BIAS WEIGHT

Neural networks are composed of nodes or **units** (see Figure 20.15) connected by directed **links**. A link from unit j to unit i serves to propagate the **activation** a_j from j to i . Each link also has a numeric **weight** $W_{j,i}$ associated with it, which determines the strength and sign of the connection. Each unit i first computes a weighted sum of its inputs:

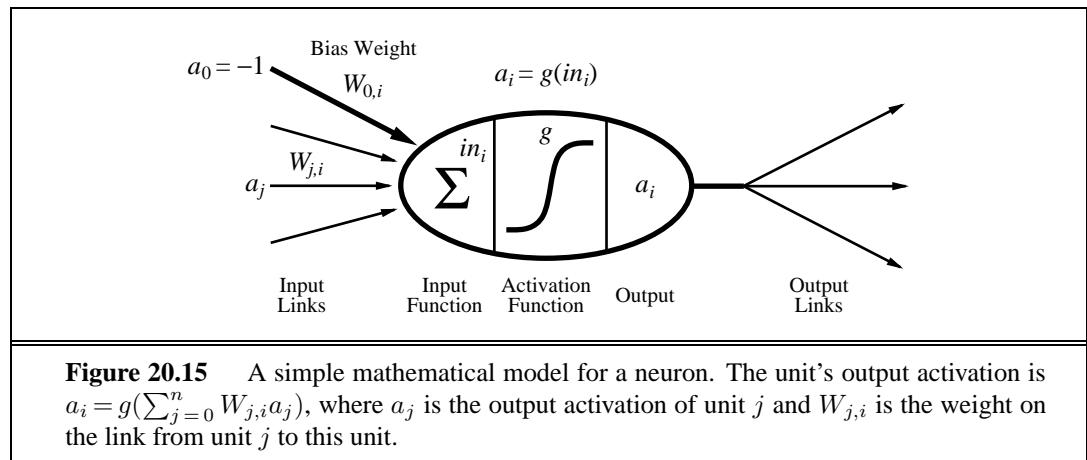
$$in_i = \sum_{j=0}^n W_{j,i} a_j .$$

Then it applies an **activation function** g to this sum to derive the output:

$$a_i = g(in_i) = g\left(\sum_{j=0}^n W_{j,i} a_j\right) . \quad (20.10)$$

Notice that we have included a **bias weight** $W_{0,i}$ connected to a fixed input $a_0 = -1$. We will explain its role in a moment.

The activation function g is designed to meet two desiderata. First, we want the unit to be “active” (near +1) when the “right” inputs are given, and “inactive” (near 0) when the “wrong” inputs are given. Second, the activation needs to be *nonlinear*, otherwise the entire neural network collapses into a simple linear function (see Exercise 20.17). Two choices for g



THRESHOLD
SIGMOID FUNCTION
LOGISTIC FUNCTION

are shown in Figure 20.16: the **threshold** function and the **sigmoid function** (also known as the **logistic function**). The sigmoid function has the advantage of being differentiable, which we will see later is important for the weight-learning algorithm. Notice that both functions have a threshold (either hard or soft) at zero; the bias weight $W_{0,i}$ sets the *actual* threshold for the unit, in the sense that the unit is activated when the weighted sum of “real” inputs $\sum_{j=1}^n W_{j,i} a_j$ (i.e., excluding the bias input) exceeds $W_{0,i}$.

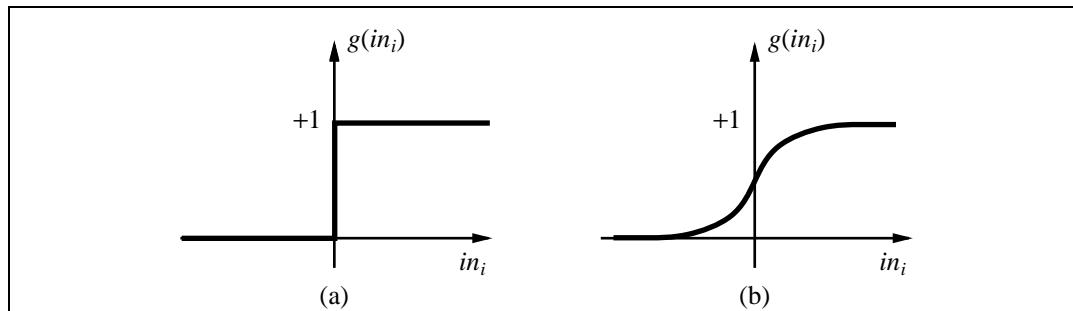


Figure 20.16 (a) The **threshold** activation function, which outputs 1 when the input is positive and 0 otherwise. (Sometimes the **sign** function is used instead, which outputs ± 1 depending on the sign of the input.) (b) The **sigmoid** function $1/(1 + e^{-x})$.

We can get a feel for the operation of individual units by comparing them with logic gates. One of the original motivations for the design of individual units (McCulloch and Pitts, 1943) was their ability to represent basic Boolean functions. Figure 20.17 shows how the Boolean functions AND, OR, and NOT can be represented by threshold units with suitable weights. This is important because it means we can use these units to build a network to compute any Boolean function of the inputs.

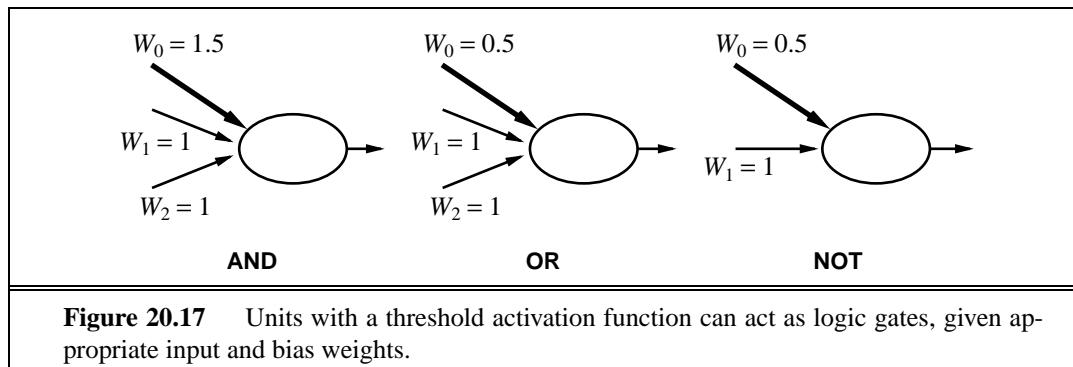


Figure 20.17 Units with a threshold activation function can act as logic gates, given appropriate input and bias weights.

Network structures

FEED-FORWARD NETWORKS
RECURRENT NETWORKS

There are two main categories of neural network structures: acyclic or **feed-forward networks** and cyclic or **recurrent networks**. A feed-forward network represents a function of its current input; thus, it has no internal state other than the weights themselves. A recurrent

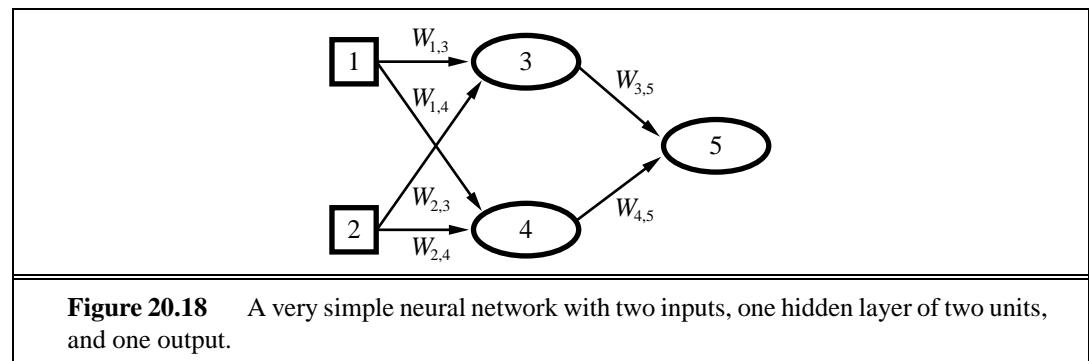
network, on the other hand, feeds its outputs back into its own inputs. This means that the activation levels of the network form a dynamical system that may reach a stable state or exhibit oscillations or even chaotic behavior. Moreover, the response of the network to a given input depends on its initial state, which may depend on previous inputs. Hence, recurrent networks (unlike feed-forward networks) can support short-term memory. This makes them more interesting as models of the brain, but also more difficult to understand. This section will concentrate on feed-forward networks; some pointers for further reading on recurrent networks are given at the end of the chapter.

HIDDEN UNITS

Let us look more closely into the assertion that a feed-forward network represents a function of its inputs. Consider the simple network shown in Figure 20.18, which has two input units, two **hidden units**, and an output unit. (To keep things simple, we have omitted the bias units in this example.) Given an input vector $\mathbf{x} = (x_1, x_2)$, the activations of the input units are set to $(a_1, a_2) = (x_1, x_2)$ and the network computes

$$\begin{aligned} a_5 &= g(W_{3,5}a_3 + W_{4,5}a_4) \\ &= g(W_{3,5}g(W_{1,3}a_1 + W_{2,3}a_2) + W_{4,5}g(W_{1,4}a_1 + W_{2,4}a_2)). \end{aligned} \quad (20.11)$$

That is, by expressing the output of each hidden unit as a function of *its* inputs, we have shown that output of the network as a whole, a_5 , is a function of the network's inputs. Furthermore, we see that the weights in the network act as *parameters* of this function; writing \mathbf{W} for the parameters, the network computes a function $h_{\mathbf{W}}(\mathbf{x})$. By adjusting the weights, we change the function that the network represents. This is how learning occurs in neural networks.



LAYERS

A neural network can be used for classification or regression. For Boolean classification with continuous outputs (e.g., with sigmoid units), it is traditional to have a single output unit, with a value over 0.5 interpreted as one class and a value below 0.5 as the other. For k -way classification, one could divide the single output unit's range into k portions, but it is more common to have k separate output units, with the value of each one representing the relative likelihood of that class given the current input.

Feed-forward networks are usually arranged in **layers**, such that each unit receives input only from units in the immediately preceding layer. In the next two subsections, we will look at single layer networks, which have no hidden units, and multilayer networks, which have one or more layers of hidden units.

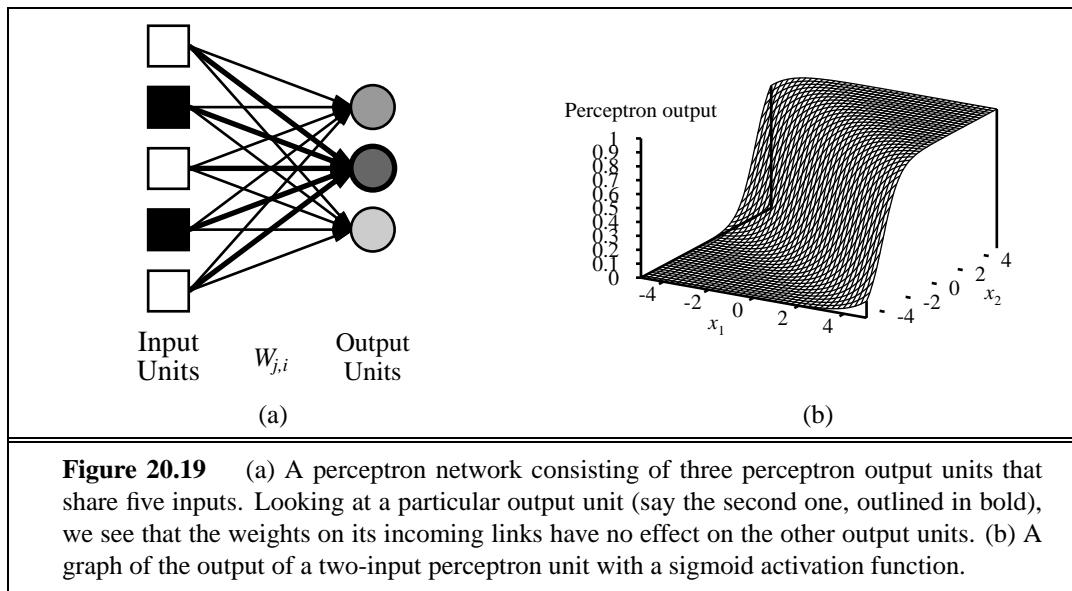


Figure 20.19 (a) A perceptron network consisting of three perceptron output units that share five inputs. Looking at a particular output unit (say the second one, outlined in bold), we see that the weights on its incoming links have no effect on the other output units. (b) A graph of the output of a two-input perceptron unit with a sigmoid activation function.

Single layer feed-forward neural networks (perceptrons)

SINGLE-LAYER
NEURAL NETWORK
PERCEPTRON

A network with all the inputs connected directly to the outputs is called a **single-layer neural network**, or a **perceptron** network. Since each output unit is independent of the others—each weight affects only one of the outputs—we can limit our study to perceptrons with a single output unit, as explained in Figure 20.19(a).

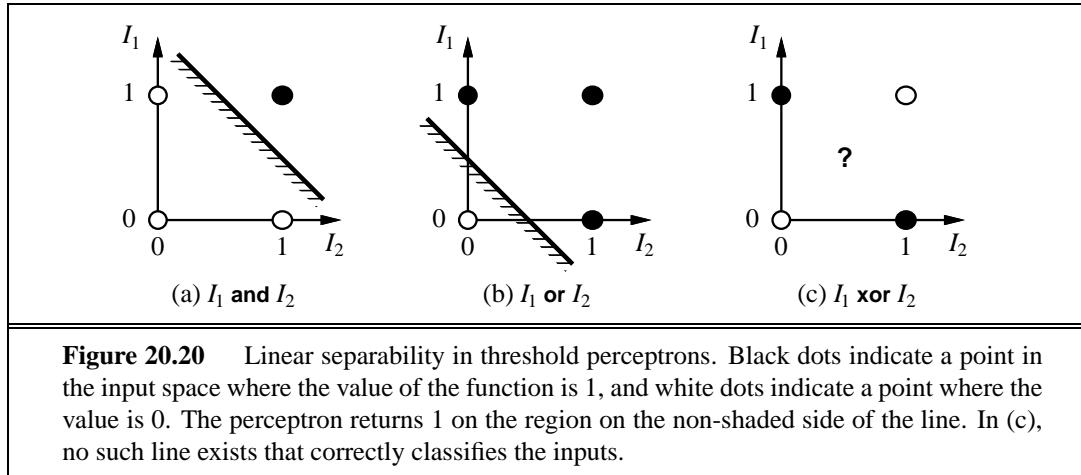
Let us begin by examining the hypothesis space that a perceptron can represent. With a threshold activation function, we can view the perceptron as representing a Boolean function. In addition to the elementary Boolean functions AND, OR, and NOT (Figure 20.17), a perceptron can represent some quite “complex” Boolean functions very compactly. For example, the **majority function**, which outputs a 1 only if more than half of its n inputs are 1, can be represented by a perceptron with each $W_j = 1$ and threshold $W_0 = n/2$. A decision tree would need $O(2^n)$ nodes to represent this function.

Unfortunately, there are many Boolean functions that the threshold perceptron cannot represent. Looking at Equation (20.10), we see that the threshold perceptron returns 1 if and only if the weighted sum of its inputs (including the bias) is positive:

$$\sum_{j=0}^n W_j x_j > 0 \quad \text{or} \quad \mathbf{W} \cdot \mathbf{x} > 0 .$$

LINEAR SEPARATOR

Now, the equation $\mathbf{W} \cdot \mathbf{x} = 0$ defines a *hyperplane* in the input space, so the perceptron returns 1 if and only if the input is on one side of that hyperplane. For this reason, the threshold perceptron is called a **linear separator**. Figure 20.20(a) and (b) show this hyperplane (a line, in two dimensions) for the perceptron representations of the AND and OR functions of two inputs. Black dots indicate a point in the input space where the value of the function is 1, and white dots indicate a point where the value is 0. The perceptron can represent these functions because there is some line that separates all the white dots from all the black

LINEARLY
SEPARABLE

WEIGHT SPACE

dots. Such functions are called **linearly separable**. Figure 20.20(c) shows an example of a function that is *not* linearly separable—the XOR function. Clearly, there is no way for a threshold perceptron to learn this function. In general, *threshold perceptrons can represent only linearly separable functions*. These constitute just a small fraction of all functions; Exercise 20.14 asks you to quantify this fraction. Sigmoid perceptrons are similarly limited, in the sense that they represent only “soft” linear separators. (See Figure 20.19(b).)

Despite their limited expressive power, threshold perceptrons have some advantages. In particular, *there is a simple learning algorithm that will fit a threshold perceptron to any linearly separable training set*. Rather than present this algorithm, we will *derive* a closely related algorithm for learning in sigmoid perceptrons.

The idea behind this algorithm, and indeed behind most algorithms for neural network learning, is to adjust the weights of the network to minimize some measure of the error on the training set. Thus, learning is formulated as an optimization search in **weight space**.⁸ The “classical” measure of error is the **sum of squared errors**, which we used for linear regression on page 720. The squared error for a single training example with input \mathbf{x} and true output y is written as

$$E = \frac{1}{2} Err^2 \equiv \frac{1}{2}(y - h_{\mathbf{w}}(\mathbf{x}))^2,$$

where $h_{\mathbf{w}}(\mathbf{x})$ is the output of the perceptron on the example and T is the true output value.

We can use gradient descent to reduce the squared error by calculating the partial derivative of E with respect to each weight. We have

$$\begin{aligned} \frac{\partial E}{\partial W_j} &= Err \times \frac{\partial Err}{\partial W_j} \\ &= Err \times \frac{\partial}{\partial W_j} g\left(y - \sum_{j=0}^n W_j x_j\right) \\ &= -Err \times g'(in) \times x_j, \end{aligned}$$

⁸ See Section 4.4 for general optimization techniques applicable to continuous spaces.

where g' is the derivative of the activation function.⁹ In the gradient descent algorithm, where we want to *reduce* E , we update the weight as follows:

$$W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j , \quad (20.12)$$

where α is the **learning rate**. Intuitively, this makes a lot of sense. If the error $Err = y - h_w(\mathbf{x})$ is positive, then the network output is too small and so the weights are *increased* for the positive inputs and *decreased* for the negative inputs. The opposite happens when the error is negative.¹⁰

The complete algorithm is shown in Figure 20.21. It runs the training examples through the net one at a time, adjusting the weights slightly after each example to reduce the error. Each cycle through the examples is called an **epoch**. Epochs are repeated until some stopping criterion is reached—typically, that the weight changes have become very small. Other methods calculate the gradient for the whole training set by adding up all the gradient contributions in Equation (20.12) before updating the weights. The **stochastic gradient** method selects examples randomly from the training set rather than cycling through them.

EPOCH

STOCHASTIC GRADIENT

```

function PERCEPTRON-LEARNING(examples, network) returns a perceptron hypothesis
  inputs: examples, a set of examples, each with input  $\mathbf{x} = x_1, \dots, x_n$  and output  $y$ 
           network, a perceptron with weights  $W_j, j = 0 \dots n$ , and activation function  $g$ 
  repeat
    for each e in examples do
       $in \leftarrow \sum_{j=0}^n W_j x_j[e]$ 
       $Err \leftarrow y[e] - g(in)$ 
       $W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j[e]$ 
    until some stopping criterion is satisfied
  return NEURAL-NET-HYPOTHESIS(network)

```

Figure 20.21 The gradient descent learning algorithm for perceptrons, assuming a differentiable activation function g . For threshold perceptrons, the factor $g'(in)$ is omitted from the weight update. NEURAL-NET-HYPOTHESIS returns a hypothesis that computes the network output for any given example.

Figure 20.22 shows the learning curve for a perceptron on two different problems. On the left, we show the curve for learning the majority function with 11 Boolean inputs (i.e., the function outputs a 1 if 6 or more inputs are 1). As we would expect, the perceptron learns the function quite quickly, because the majority function is linearly separable. On the other hand, the decision-tree learner makes no progress, because the majority function is very hard (although not impossible) to represent as a decision tree. On the right, we have the restaurant

⁹ For the sigmoid, this derivative is given by $g' = g(1 - g)$.

¹⁰ For threshold perceptrons, where $g'(in)$ is undefined, the original **perceptron learning rule** developed by Rosenblatt (1957) is identical to Equation (20.12) except that $g'(in)$ is omitted. Since $g'(in)$ is the same for all weights, its omission changes only the magnitude and not the direction of the overall weight update for each example.

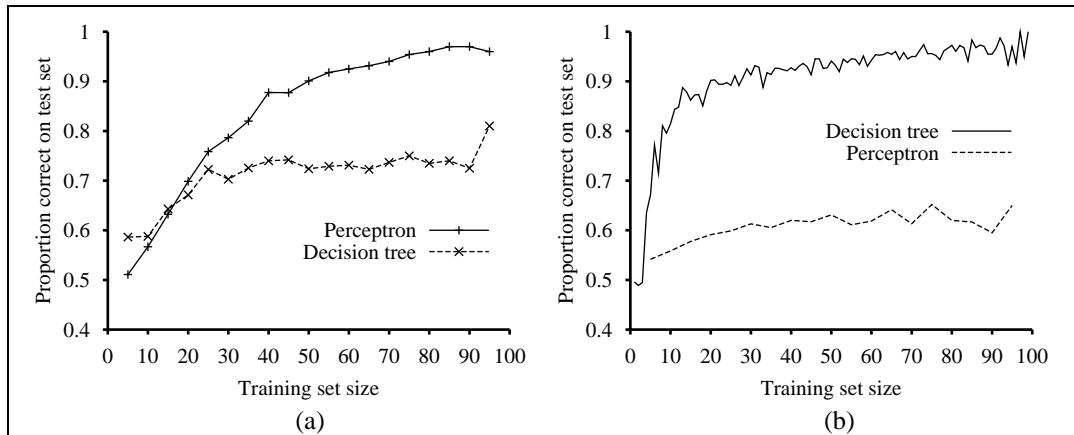


Figure 20.22 Comparing the performance of perceptrons and decision trees. (a) Perceptrons are better at learning the majority function of 11 inputs. (b) Decision trees are better at learning the *WillWait* predicate in the restaurant example.

example. The solution problem is easily represented as a decision tree, but is not linearly separable. The best plane through the data correctly classifies only 65%.

So far, we have treated perceptrons as deterministic functions with possibly erroneous outputs. It is also possible to interpret the output of a sigmoid perceptron as a *probability*—specifically, the probability that the true output is 1 given the inputs. With this interpretation, one can use the sigmoid as a canonical representation for conditional distributions in Bayesian networks (see Section 14.3). One can also derive a learning rule using the standard method of maximizing the (conditional) log likelihood of the data, as described earlier in this chapter. Let's see how this works.

Consider a single training example with true output value T , and let p be the probability returned by the perceptron for this example. If $T = 1$, the conditional probability of the datum is p , and if $T = 0$, the conditional probability of the datum is $(1 - p)$. Now we can use a simple trick to write the log likelihood in a form that is differentiable. The trick is that a 0/1 variable in the *exponent* of an expression acts as an **indicator variable**: p^T is p if $T = 1$ and 1 otherwise; similarly $(1 - p)^{(1-T)}$ is $(1 - p)$ if $T = 0$ and 1 otherwise. Hence, we can write the log likelihood of the datum as

$$L = \log p^T (1 - p)^{(1-T)} = T \log p + (1 - T) \log(1 - p). \quad (20.13)$$

Thanks to the properties of the sigmoid function, the gradient reduces to a very simple formula (Exercise 20.16):

$$\frac{\partial L}{\partial W_j} = Err \times a_j.$$

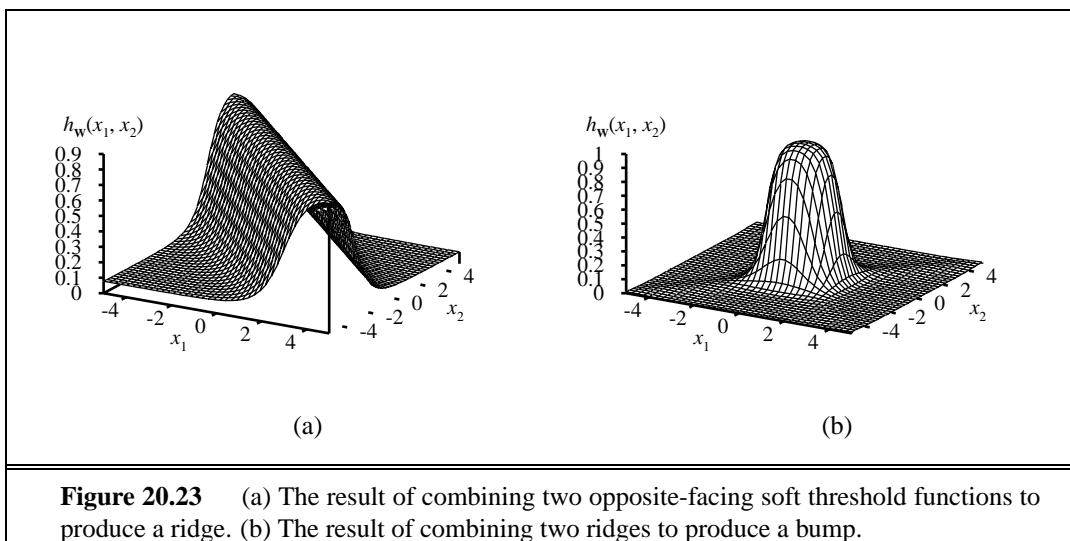
INDICATOR VARIABLE

Notice that the *weight-update vector for maximum likelihood learning in sigmoid perceptrons is essentially identical to the update vector for squared error minimization*. Thus, we could say that perceptrons have a probabilistic interpretation even when the learning rule is derived from a deterministic viewpoint.



Multilayer feed-forward neural networks

Now we will consider networks with hidden units. The most common case involves a single hidden layer,¹¹ as in Figure 20.24. The advantage of adding hidden layers is that it enlarges the space of hypotheses that the network can represent. Think of each hidden unit as a perceptron that represents a soft threshold function in the input space, as shown in Figure 20.19(b). Then, think of an output unit as a soft-thresholded linear combination of several such functions. For example, by adding two opposite-facing soft threshold functions and thresholding the result, we can obtain a “ridge” function as shown in Figure 20.23(a). Combining two such ridges at right angles to each other (i.e., combining the outputs from four hidden units), we obtain a “bump” as shown in Figure 20.23(b).



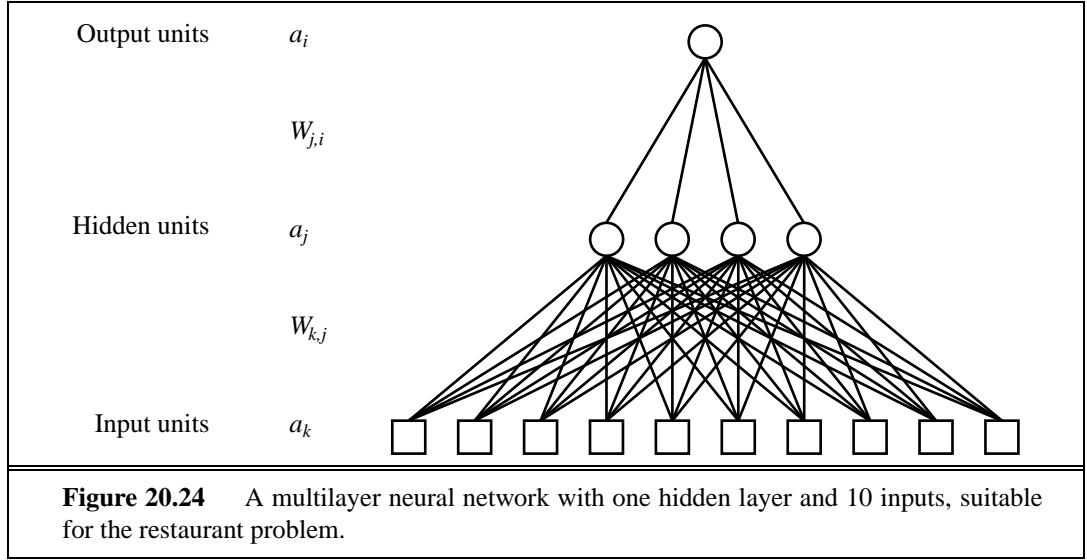
With more hidden units, we can produce more bumps of different sizes in more places. In fact, with a single, sufficiently large hidden layer, it is possible to represent any continuous function of the inputs with arbitrary accuracy; with two layers, even discontinuous functions can be represented.¹² Unfortunately, for any *particular* network structure, it is harder to characterize exactly which functions can be represented and which ones cannot.

Suppose we want to construct a hidden layer network for the restaurant problem. We have 10 attributes describing each example, so we will need 10 input units. How many hidden units are needed? In Figure 20.24, we show a network with four hidden units. This turns out to be about right for this problem. The problem of choosing the right number of hidden units in advance is still not well understood. (See page 748.)

Learning algorithms for multilayer networks are similar to the perceptron learning algorithm shown in Figure 20.21. One minor difference is that we may have several outputs, so

¹¹ Some people call this a three-layer network, and some call it a two-layer network (because the inputs aren't "real" units). We will avoid confusion and call it a "single-hidden-layer network."

¹² The proof is complex, but the main point is that the required number of hidden units grows exponentially with the number of inputs. For example, $2^n/n$ hidden units are needed to encode all Boolean functions of n inputs.



BACK-PROPAGATION

we have an output vector $\mathbf{h}_W(\mathbf{x})$ rather than a single value, and each example has an output vector \mathbf{y} . The major difference is that, whereas the error $\mathbf{y} - \mathbf{h}_W$ at the output layer is clear, the error at the hidden layers seems mysterious because the training data does not say what value the hidden nodes should have. It turns out that we can **back-propagate** the error from the output layer to the hidden layers. The back-propagation process emerges directly from a derivation of the overall error gradient. First, we will describe the process with an intuitive justification; then, we will show the derivation.

At the output layer, the weight-update rule is identical to Equation (20.12). We have multiple output units, so let Err_i be i th component of the error vector $\mathbf{y} - \mathbf{h}_W$. We will also find it convenient to define a modified error $\Delta_i = Err_i \times g'(in_i)$, so that the weight-update rule becomes

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i . \quad (20.14)$$

To update the connections between the input units and the hidden units, we need to define a quantity analogous to the error term for output nodes. Here is where we do the error back-propagation. The idea is that hidden node j is “responsible” for some fraction of the error Δ_i in each of the output nodes to which it connects. Thus, the Δ_i values are divided according to the strength of the connection between the hidden node and the output node and are propagated back to provide the Δ_j values for the hidden layer. The propagation rule for the Δ values is the following:

$$\Delta_j = g'(in_j) \sum_i W_{j,i} \Delta_i . \quad (20.15)$$

Now the weight-update rule for the weights between the inputs and the hidden layer is almost identical to the update rule for the output layer:

$$W_{k,j} \leftarrow W_{k,j} + \alpha \times a_k \times \Delta_j .$$

The back-propagation process can be summarized as follows:

```

function BACK-PROP-LEARNING(examples, network) returns a neural network
  inputs: examples, a set of examples, each with input vector  $\mathbf{x}$  and output vector  $\mathbf{y}$ 
  network, a multilayer network with  $L$  layers, weights  $W_{j,i}$ , activation function  $g$ 

  repeat
    for each e in examples do
      for each node  $j$  in the input layer do  $a_j \leftarrow x_j[e]$ 
      for  $\ell = 2$  to  $M$  do
         $in_i \leftarrow \sum_j W_{j,i} a_j$ 
         $a_i \leftarrow g(in_i)$ 
      for each node  $i$  in the output layer do
         $\Delta_i \leftarrow g'(in_i) \times (y_i[e] - a_i)$ 
      for  $\ell = M - 1$  to 1 do
        for each node  $j$  in layer  $\ell$  do
           $\Delta_j \leftarrow g'(in_j) \sum_i W_{j,i} \Delta_i$ 
        for each node  $i$  in layer  $\ell + 1$  do
           $W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$ 
    until some stopping criterion is satisfied
  return NEURAL-NET-HYPOTHESIS(network)

```

Figure 20.25 The back-propagation algorithm for learning in multilayer networks.

- Compute the Δ values for the output units, using the observed error.
- Starting with output layer, repeat the following for each layer in the network, until the earliest hidden layer is reached:
 - Propagate the Δ values back to the previous layer.
 - Update the weights between the two layers.

The detailed algorithm is shown in Figure 20.25.

For the mathematically inclined, we will now derive the back-propagation equations from first principles. The squared error on a single example is defined as

$$E = \frac{1}{2} \sum_i (y_i - a_i)^2 ,$$

where the sum is over the nodes in the output layer. To obtain the gradient with respect to a specific weight $W_{j,i}$ in the output layer, we need only expand out the activation a_i as all other terms in the summation are unaffected by $W_{j,i}$:

$$\begin{aligned} \frac{\partial E}{\partial W_{j,i}} &= -(y_i - a_i) \frac{\partial a_i}{\partial W_{j,i}} = -(y_i - a_i) \frac{\partial g(in_i)}{\partial W_{j,i}} \\ &= -(y_i - a_i) g'(in_i) \frac{\partial in_i}{\partial W_{j,i}} = -(y_i - a_i) g'(in_i) \frac{\partial}{\partial W_{j,i}} \left(\sum_j W_{j,i} a_j \right) \\ &= -(y_i - a_i) g'(in_i) a_j = -a_j \Delta_i , \end{aligned}$$

with Δ_i defined as before. To obtain the gradient with respect to the $W_{k,j}$ weights connecting the input layer to the hidden layer, we have to keep the entire summation over i because each output value a_i may be affected by changes in $W_{k,j}$. We also have to expand out the activations a_j . We will show the derivation in gory detail because it is interesting to see how the derivative operator propagates back through the network:

$$\begin{aligned}
 \frac{\partial E}{\partial W_{k,j}} &= - \sum_i (y_i - a_i) \frac{\partial a_i}{\partial W_{k,j}} = - \sum_i (y_i - a_i) \frac{\partial g(in_i)}{\partial W_{k,j}} \\
 &= - \sum_i (y_i - a_i) g'(in_i) \frac{\partial in_i}{\partial W_{k,j}} = - \sum_i \Delta_i \frac{\partial}{\partial W_{k,j}} \left(\sum_j W_{j,i} a_j \right) \\
 &= - \sum_i \Delta_i W_{j,i} \frac{\partial a_j}{\partial W_{k,j}} = - \sum_i \Delta_i W_{j,i} \frac{\partial g(in_j)}{\partial W_{k,j}} \\
 &= - \sum_i \Delta_i W_{j,i} g'(in_j) \frac{\partial in_j}{\partial W_{k,j}} \\
 &= - \sum_i \Delta_i W_{j,i} g'(in_j) \frac{\partial}{\partial W_{k,j}} \left(\sum_k W_{k,j} a_k \right) \\
 &= - \sum_i \Delta_i W_{j,i} g'(in_j) a_k = -a_k \Delta_j ,
 \end{aligned}$$

where Δ_j is defined as before. Thus, we obtain the update rules obtained earlier from intuitive considerations. It is also clear that the process can be continued for networks with more than one hidden layer, which justifies the general algorithm given in Figure 20.25.

Having made it through (or skipped over) all the mathematics, let's see how a single-hidden-layer network performs on the restaurant problem. In Figure 20.26, we show two

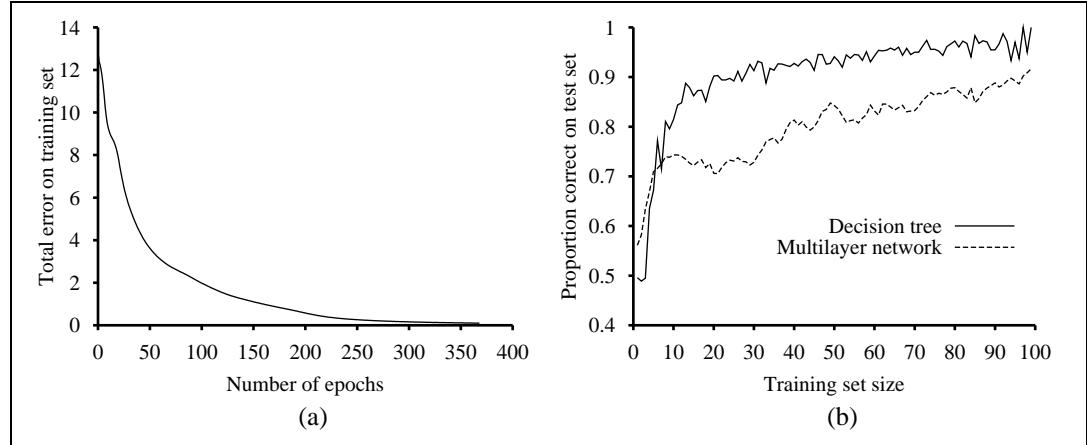


Figure 20.26 (a) Training curve showing the gradual reduction in error as weights are modified over several epochs, for a given set of examples in the restaurant domain. (b) Comparative learning curves showing that decision-tree learning does slightly better than back-propagation in a multilayer network.

TRAINING CURVE

curves. The first is a **training curve**, which shows the mean squared error on a given training set of 100 restaurant examples during the weight-updating process. This demonstrates that the network does indeed converge to a perfect fit to the training data. The second curve is the standard learning curve for the restaurant data. The neural network does learn well, although not quite as fast as decision-tree learning; this is perhaps not surprising, because the data were generated from a simple decision tree in the first place.

Neural networks are capable of far more complex learning tasks of course, although it must be said that a certain amount of twiddling is needed to get the network structure right and to achieve convergence to something close to the global optimum in weight space. There are literally tens of thousands of published applications of neural networks. Section 20.7 looks at one such application in more depth.

Learning neural network structures

So far, we have considered the problem of learning weights, given a fixed network structure; just as with Bayesian networks, we also need to understand how to find the best network structure. If we choose a network that is too big, it will be able to memorize all the examples by forming a large lookup table, but will not necessarily generalize well to inputs that have not been seen before.¹³ In other words, like all statistical models, neural networks are subject to **overfitting** when there are too many parameters in the model. We saw this in Figure 18.1 (page 652), where the high-parameter models in (b) and (c) fit all the data, but might not generalize as well as the low-parameter models in (a) and (d).

If we stick to fully connected networks, the only choices to be made concern the number of hidden layers and their sizes. The usual approach is to try several and keep the best. The **cross-validation** techniques of Chapter 18 are needed if we are to avoid **peeking** at the test set. That is, we choose the network architecture that gives the highest prediction accuracy on the validation sets.

If we want to consider networks that are not fully connected, then we need to find some effective search method through the very large space of possible connection topologies. The **optimal brain damage** algorithm begins with a fully connected network and removes connections from it. After the network is trained for the first time, an information-theoretic approach identifies an optimal selection of connections that can be dropped. The network is then retrained, and if its performance has not decreased then the process is repeated. In addition to removing connections, it is also possible to remove units that are not contributing much to the result.

Several algorithms have been proposed for growing a larger network from a smaller one. One, the **tiling** algorithm, resembles decision-list learning. The idea is to start with a single unit that does its best to produce the correct output on as many of the training examples as possible. Subsequent units are added to take care of the examples that the first unit got wrong. The algorithm adds only as many units as are needed to cover all the examples.

OPTIMAL BRAIN DAMAGE

TILING

¹³ It has been observed that very large networks *do* generalize well *as long as the weights are kept small*. This restriction keeps the activation values in the *linear* region of the sigmoid function $g(x)$ where x is close to zero. This, in turn, means that the network behaves like a linear function (Exercise 20.17) with far fewer parameters.

20.6 KERNEL MACHINES

SUPPORT VECTOR
MACHINE
KERNEL MACHINE

MARGIN

QUADRATIC
PROGRAMMING

Our discussion of neural networks left us with a dilemma. Single-layer networks have a simple and efficient learning algorithm, but have very limited expressive power—they can learn only linear decision boundaries in the input space. Multilayer networks, on the other hand, are much more expressive—they can represent general nonlinear functions—but are very hard to train because of the abundance of local minima and the high dimensionality of the weight space. In this section, we will explore a relatively new family of learning methods called **support vector machines** (SVMs) or, more generally, **kernel machines**. To some extent, kernel machines give us the best of both worlds. That is, these methods use an efficient training algorithm *and* can represent complex, nonlinear functions.

The full treatment of kernel machines is beyond the scope of the book, but we can illustrate the main idea through an example. Figure 20.27(a) shows a two-dimensional input space defined by attributes $\mathbf{x} = (x_1, x_2)$, with positive examples ($y = +1$) inside a circular region and negative examples ($y = -1$) outside. Clearly, there is no linear separator for this problem. Now, suppose we re-express the input data using some computed features—i.e., we map each input vector \mathbf{x} to a new vector of feature values, $F(\mathbf{x})$. In particular, let us use the three features

$$f_1 = x_1^2, \quad f_2 = x_2^2, \quad f_3 = \sqrt{2}x_1x_2. \quad (20.16)$$

We will see shortly where these came from, but, for now, just look at what happens. Figure 20.27(b) shows the data in the new, three-dimensional space defined by the three features; the data are *linearly separable* in this space! This phenomenon is actually fairly general: if data are mapped into a space of sufficiently high dimension, then they will always be linearly separable. Here, we used only three dimensions,¹⁴ but if we have N data points then, except in special cases, they will always be separable in a space of $N - 1$ dimensions or more (Exercise 20.21).

So, is that it? Do we just produce loads of computed features and then find a linear separator in the corresponding high-dimensional space? Unfortunately, it's not that easy. Remember that a linear separator in a space of d dimensions is defined by an equation with d parameters, so we are in serious danger of overfitting the data if $d \approx N$, the number of data points. (This is like overfitting data with a high-degree polynomial, which we discussed in Chapter 18.) For this reason, kernel machines usually find the *optimal* linear separator—the one that has the largest **margin** between it and the positive examples on one side and the negative examples on the other. (See Figure 20.28.) It can be shown, using arguments from computational learning theory (Section 18.5), that this separator has desirable properties in terms of robust generalization to new examples.

Now, how do we find this separator? It turns out that this is a **quadratic programming** optimization problem. Suppose we have examples \mathbf{x}_i with classifications $y_i = \pm 1$ and we want to find an optimal separator in the input space; then the quadratic programming problem

¹⁴ The reader may notice that we could have used just f_1 and f_2 , but the 3D mapping illustrates the idea better.

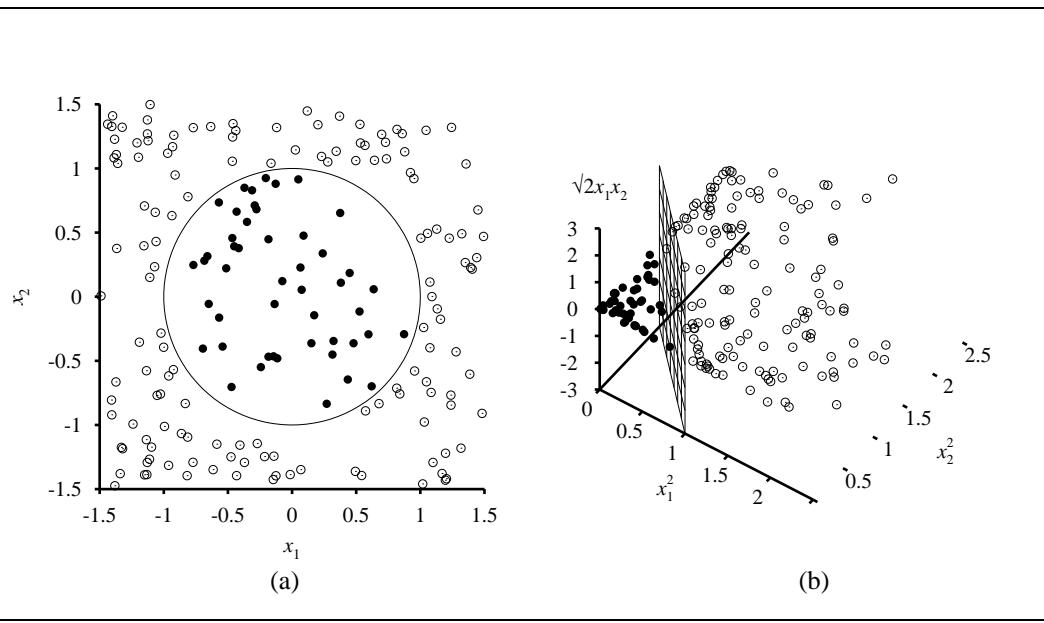


Figure 20.27 (a) A two-dimensional training with positive examples as black circles and negative examples as white circles. The true decision boundary, $x_1^2 + x_2^2 \leq 1$, is also shown. (b) The same data after mapping into a three-dimensional input space $(x_1^2, x_2^2, \sqrt{2}x_1x_2)$. The circular decision boundary in (a) becomes a linear decision boundary in three dimensions.

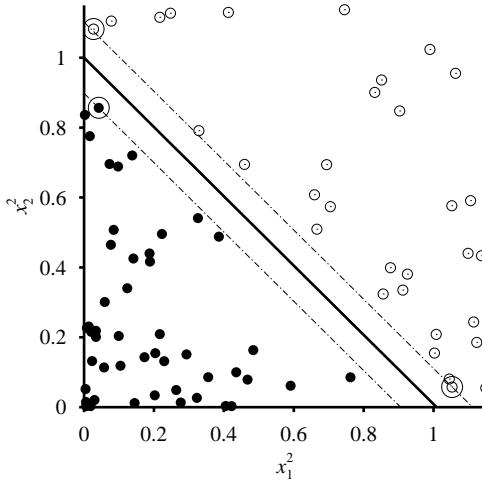


Figure 20.28 A close-up, projected onto the first two dimensions, of the optimal separator shown in Figure 20.27(b). The separator is shown as a heavy line, with the closest points—the **support vectors**—marked with circles. The **margin** is the separation between the positive and negative examples.

is to find values of the parameters α_i that maximize the expression

$$\sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j) \quad (20.17)$$

subject to the constraints $\alpha_i \geq 0$ and $\sum_i \alpha_i y_i = 0$. Although the derivation of this expression is not crucial to the story, it does have two important properties. First, the expression has a single global maximum that can be found efficiently. Second, *the data enter the expression only in the form of dot products of pairs of points*. This second property is also true of the equation for the separator itself; once the optimal α_i s have been calculated, it is

$$h(\mathbf{x}) = \text{sign} \left(\sum_i \alpha_i y_i (\mathbf{x} \cdot \mathbf{x}_i) \right). \quad (20.18)$$

SUPPORT VECTOR

A final important property of the optimal separator defined by this equation is that the weights α_i associated with each data point are *zero* except for those points closest to the separator—the so-called **support vectors**. (They are called this because they “hold up” the separating plane.) Because there are usually many fewer support vectors than data points, the effective number of parameters defining the optimal separator is usually much less than N .

Now, we would not usually expect to find a linear separator in the input space \mathbf{x} , but it is easy to see that we can find linear separators in the high-dimensional feature space $F(\mathbf{x})$ simply by replacing $\mathbf{x}_i \cdot \mathbf{x}_j$ in Equation (20.17) with $F(\mathbf{x}_i) \cdot F(\mathbf{x}_j)$. This by itself is not remarkable—replacing \mathbf{x} by $F(\mathbf{x})$ in *any* learning algorithm has the required effect—but the dot product has some special properties. It turns out that $F(\mathbf{x}_i) \cdot F(\mathbf{x}_j)$ can often be computed without first computing F for each point. In our three-dimensional feature space defined by Equation (20.16), a little bit of algebra shows that

$$F(\mathbf{x}_i) \cdot F(\mathbf{x}_j) = (\mathbf{x}_i \cdot \mathbf{x}_j)^2.$$

The expression $(\mathbf{x}_i \cdot \mathbf{x}_j)^2$ is called a **kernel function**, usually written as $K(\mathbf{x}_i, \mathbf{x}_j)$. In the kernel machine context, this means a function that can be applied to pairs of input data to evaluate dot products in some corresponding feature space. So, we can restate the claim at the beginning of this paragraph as follows: we can find linear separators in the high-dimensional feature space $F(\mathbf{x})$ simply by replacing $\mathbf{x}_i \cdot \mathbf{x}_j$ in Equation (20.17) with a kernel function $K(\mathbf{x}_i, \mathbf{x}_j)$. Thus, we can learn in the high-dimensional space but we compute only kernel functions rather than the full list of features for each data point.

MERCER'S THEOREM

POLYNOMIAL KERNEL



The next step, which should by now be obvious, is to see that there's nothing special about the kernel $K(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i \cdot \mathbf{x}_j)^2$. It corresponds to a particular higher-dimensional feature space, but other kernel functions correspond to other feature spaces. A venerable result in mathematics, **Mercer's theorem** (1909), tells us that any “reasonable”¹⁵ kernel function corresponds to *some* feature space. These feature spaces can be very large, even for innocuous-looking kernels. For example, the **polynomial kernel**, $K(\mathbf{x}_i, \mathbf{x}_j) = (1 + \mathbf{x}_i \cdot \mathbf{x}_j)^d$, corresponds to a feature space whose dimension is exponential in d . Using such kernels in Equation (20.17), then, *optimal linear separators can be found efficiently in feature spaces with billions (or, in some cases, infinitely many) dimensions*. The resulting linear separators,

¹⁵ Here, “reasonable” means that the matrix $\mathbf{K}_{ij} = K(\mathbf{x}_i, \mathbf{x}_j)$ is positive definite; see Appendix A.

when mapped back to the original input space, can correspond to arbitrarily wiggly, nonlinear boundaries between the positive and negative examples.

We mentioned in the preceding section that kernel machines excel at handwritten digit recognition; they are rapidly being adopted for other applications—especially those with many input features. As part of this process, many new kernels have been designed that work with strings, trees, and other non-numerical data types. It has also been observed that the kernel method can be applied not only with learning algorithms that find optimal linear separators, but also with any other algorithm that can be reformulated to work only with dot products of pairs of data points, as in Equations 20.17 and 20.18. Once this is done, the dot product is replaced by a kernel function and we have a **kernelized** version of the algorithm. This can be done easily for k -nearest-neighbor and perceptron learning, among others.

KERNELIZATION

20.7 CASE STUDY: HANDWRITTEN DIGIT RECOGNITION

Recognizing handwritten digits is an important problem with many applications, including automated sorting of mail by postal code, automated reading of checks and tax returns, and data entry for hand-held computers. It is an area where rapid progress has been made, in part because of better learning algorithms and in part because of the availability of better training sets. The United States National Institute of Science and Technology (**NIST**) has archived a database of 60,000 labeled digits, each $20 \times 20 = 400$ pixels with 8-bit grayscale values. It has become one of the standard benchmark problems for comparing new learning algorithms. Some example digits are shown in Figure 20.29.



Figure 20.29 Examples from the NIST database of handwritten digits. Top row: examples of digits 0–9 that are easy to identify. Bottom row: more difficult examples of the same digits.

Many different learning approaches have been tried. One of the first, and probably the simplest, is the **3-nearest-neighbor** classifier, which also has the advantage of requiring no training time. As a memory-based algorithm, however, it must store all 60,000 images, and its runtime performance is slow. It achieved a test error rate of 2.4%.

A **single-hidden-layer neural network** was designed for this problem with 400 input units (one per pixel) and 10 output units (one per class). Using cross-validation, it was found that roughly 300 hidden units gave the best performance. With full interconnections between layers, there were a total of 123,300 weights. This network achieved a 1.6% error rate.

A series of **specialized neural networks** called LeNet were devised to take advantage of the structure of the problem—that the input consists of pixels in a two-dimensional array, and that small changes in the position or slant of an image are unimportant. Each network had an input layer of 32×32 units, onto which the 20×20 pixels were centered so that each input unit is presented with a local neighborhood of pixels. This was followed by three layers of hidden units. Each layer consisted of several planes of $n \times n$ arrays, where n is smaller than the previous layer so that the network is down-sampling the input, and where the weights of every unit in a plane are constrained to be identical, so that the plane is acting as a feature detector: it can pick out a feature such as a long vertical line or a short semi-circular arc. The output layer had 10 units. Many versions of this architecture were tried; a representative one had hidden layers with 768, 192, and 30 units, respectively. The training set was augmented by applying affine transformations to the actual inputs: shifting, slightly rotating, and scaling the images. (Of course, the transformations have to be small, or else a 6 will be transformed into a 9!) The best error rate achieved by LeNet was 0.9%.

A **boosted neural network** combined three copies of the LeNet architecture, with the second one trained on a mix of patterns that the first one got 50% wrong, and the third one trained on patterns for which the first two disagreed. During testing, the three nets voted with their weights for each of the ten digits, and the scores are added to determine the winner. The test error rate was 0.7%.

A **support vector machine** (see Section 20.6) with 25,000 support vectors achieved an error rate of 1.1%. This is remarkable because the SVM technique, like the simple nearest-neighbor approach, required almost no thought or iterated experimentation on the part of the developer, yet it still came close to the performance of LeNet, which had had years of development. Indeed, the support vector machine makes no use of the structure of the problem, and would perform just as well if the pixels were presented in a permuted order.

VIRTUAL SUPPORT VECTOR MACHINE

A **virtual support vector machine** starts with a regular SVM and then improves it with a technique that is designed to take advantage of the structure of the problem. Instead of allowing products of all pixel pairs, this approach concentrates on kernels formed from pairs of nearby pixels. It also augments the training set with transformations of the examples, just as LeNet did. A virtual SVM achieved the best error rate recorded to date, 0.56%.

Shape matching is a technique from computer vision used to align corresponding parts of two different images of objects. (See Chapter 24.) The idea is to pick out a set of points in each of the two images, and then compute, for each point in the first image, which point in the second image it corresponds to. From this alignment, we then compute a transformation between the images. The transformation gives us a measure of the distance between the images. This distance measure is better motivated than just counting the number of differing pixels, and it turns out that a 3-nearest neighbor algorithm using this distance measure performs very well. Training on only 20,000 of the 60,000 digits, and using 100 sample points per image extracted from a Canny edge detector, a shape matching classifier achieved 0.63% test error.

Humans are estimated to have an error rate of about 0.2% on this problem. This figure is somewhat suspect because humans have not been tested as extensively as have machine learning algorithms. On a similar data set of digits from the United States Postal Service, human errors were at 2.5%.

The following figure summarizes the error rates, runtime performance, memory requirements, and amount of training time for the seven algorithms we have discussed. It also adds another measure, the percentage of digits that must be rejected to achieve 0.5% error. For example, if the SVM is allowed to reject 1.8% of the inputs—that is, pass them on for someone else to make the final judgment—then its error rate on the remaining 98.2% of the inputs is reduced from 1.1% to 0.5%.

The following table summarizes the error rate and some of the other characteristics of the seven techniques we have discussed.

	3 NN	300 Hidden	LeNet	Boosted LeNet	SVM	Virtual SVM	Shape Match
Error rate (pct.)	2.4	1.6	0.9	0.7	1.1	0.56	0.63
Runtime (millisec/digit)	1000	10	30	50	2000	200	
Memory requirements (Mbyte)	12	.49	.012	.21	11		
Training time (days)	0	7	14	30	10		
% rejected to reach 0.5% error	8.1	3.2	1.8	0.5	1.8		

20.8 SUMMARY

Statistical learning methods range from simple calculation of averages to the construction of complex models such as Bayesian networks and neural networks. They have applications throughout computer science, engineering, neurobiology, psychology, and physics. This chapter has presented some of the basic ideas and given a flavor of the mathematical underpinnings. The main points are as follows:

- **Bayesian learning** methods formulate learning as a form of probabilistic inference, using the observations to update a prior distribution over hypotheses. This approach provides a good way to implement Ockham’s razor, but quickly becomes intractable for complex hypothesis spaces.
- **Maximum a posteriori** (MAP) learning selects a single most likely hypothesis given the data. The hypothesis prior is still used and the method is often more tractable than full Bayesian learning.
- **Maximum likelihood** learning simply selects the hypothesis that maximizes the likelihood of the data; it is equivalent to MAP learning with a uniform prior. In simple cases such as linear regression and fully observable Bayesian networks, maximum likelihood solutions can be found easily in closed form. **Naive Bayes** learning is a particularly effective technique that scales well.
- When some variables are hidden, local maximum likelihood solutions can be found using the EM algorithm. Applications include clustering using mixtures of Gaussians, learning Bayesian networks, and learning hidden Markov models.

- Learning the structure of Bayesian networks is an example of **model selection**. This usually involves a discrete search in the space of structures. Some method is required for trading off model complexity against degree of fit.
- **Instance-based models** represent a distribution using the collection of training instances. Thus, the number of parameters grows with the training set. **Nearest-neighbor** methods look at the instances nearest to the point in question, whereas **kernel** methods form a distance-weighted combination of all the instances.
- **Neural networks** are complex nonlinear functions with many parameters. Their parameters can be learned from noisy data and they have been used for thousands of applications.
- A **perceptron** is a feed-forward neural network with no hidden units that can represent only **linearly separable** functions. If the data are linearly separable, a simple weight-update rule can be used to fit the data exactly.
- **Multilayer feed-forward** neural networks can represent any function, given enough units. The **back-propagation** algorithm implements a gradient descent in parameter space to minimize the output error.

Statistical learning continues to be a very active area of research. Enormous strides have been made in both theory and practice, to the point where it is possible to learn almost any model for which exact or approximate inference is feasible.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

The application of statistical learning techniques in AI was an active area of research in the early years (see Duda and Hart, 1973) but became separated from mainstream AI as the latter field concentrated on symbolic methods. It continued in various forms—some explicitly probabilistic, others not—in areas such as **pattern recognition** (Devroye *et al.*, 1996) and **information retrieval** (Salton and McGill, 1983). A resurgence of interest occurred shortly after the introduction of Bayesian network models in the late 1980s; at roughly the same time, a statistical view of neural network learning began to emerge. In the late 1990s, there was a noticeable convergence of interests in machine learning, statistics, and neural networks, centered on methods for creating large probabilistic models from data.

The naive Bayes model is one of the oldest and simplest forms of Bayesian network, dating back to the 1950s. Its origins were mentioned in the notes at the end of Chapter 13. is partially explained by Domingos and Pazzani (1997). A boosted form of naive Bayes learning won the first KDD Cup data mining competition (Elkan, 1997). Heckerman (1998) gives an excellent introduction to the general problem of Bayes net learning. Bayesian parameter learning with Dirichlet priors for Bayesian networks was discussed by Spiegelhalter *et al.* (1993). The BUGS software package (Gilks *et al.*, 1994) incorporates many of these ideas and provides a very powerful tool for formulating and learning complex probability models. The first algorithms for learning Bayes net structures used conditional independence tests (Pearl, 1988; Pearl and Verma, 1991). Spirtes *et al.* (1993) developed a comprehensive approach

and the TETRAD package for Bayes net learning using similar ideas. Algorithmic improvements since then led to a clear victory in the 2001 KDD Cup data mining competition for a Bayes net learning method (Cheng *et al.*, 2002). (The specific task here was a bioinformatics problem with 139,351 features!) A structure-learning approach based on maximizing likelihood was developed by Cooper and Herskovits (1992) and improved by Heckerman *et al.* (1994). Friedman and Goldszmidt (1996) pointed out the influence of the representation of local conditional distributions on the learned structure.

The general problem of learning probability models with hidden variables and missing data was addressed by the EM algorithm (Dempster *et al.*, 1977), which was abstracted from several existing methods including the Baum–Welch algorithm for HMM learning (Baum and Petrie, 1966). (Dempster himself views EM as a schema rather than an algorithm, since a good deal of mathematical work may be required before it can be applied to a new family of distributions.) EM is now one of the most widely used algorithms in science, and McLachlan and Krishnan (1997) devote an entire book to the algorithm and its properties. The specific problem of learning mixture models, including mixtures of Gaussians, is covered by Titterington *et al.* (1985). Within AI, the first successful system that used EM for mixture modelling was AUTOCLASS (Cheeseman *et al.*, 1988; Cheeseman and Stutz, 1996). AUTOCLASS has been applied to a number of real-world scientific classification tasks, including the discovery of new types of stars from spectral data (Goebel *et al.*, 1989) and new classes of proteins and introns in DNA/protein sequence databases (Hunter and States, 1992).

An EM algorithm for learning Bayes nets with hidden variables was developed by Lauritzen (1995). Gradient-based techniques have also proved effective for Bayes nets as well as dynamic Bayes nets (Russell *et al.*, 1995; Binder *et al.*, 1997a). The structural EM algorithm was developed by (Friedman, 1998). The ability to learn the structure of Bayesian networks is closely connected to the issue of recovering *causal* information from data. That is, is it possible to learn Bayes nets in such a way that the recovered network structure indicates real causal influences? For many years, statisticians avoided this question, believing that observational data (as opposed to data generated from experimental trials) could yield only correlational information—after all, any two variables that appear related might in fact be influenced by third, unknown causal factor rather than influencing each other directly. Pearl (2000) has presented convincing arguments to the contrary, showing that there are in fact many cases where causality can be ascertained and developing the **causal network** formalism to express causes and the effects of intervention as well as ordinary conditional probabilities.

CAUSAL NETWORK

Nearest-neighbor models date back at least to (Fix and Hodges, 1951) and have been a standard tool in statistics and pattern recognition ever since. Within AI, they were popularized by (Stanfill and Waltz, 1986), who investigated methods for adapting the distance metric to the data. Hastie and Tibshirani (1996) developed a way to localize the metric to each point in the space, depending on the distribution of data around that point. Efficient indexing schemes for finding nearest neighbors are studied within the algorithms community (see, e.g., Indyk, 2000). Kernel density estimation, also called **Parzen window** density estimation, was investigated initially by Rosenblatt (1956) and Parzen (1962). Since that time, a huge literature has developed investigating the properties of various estimators. Devroye (1987) gives a thorough introduction.

The literature on neural networks is rather too large (approximately 100,000 papers to date) to cover in detail. Cowan and Sharp (1988b, 1988a) survey the early history, beginning with the work of McCulloch and Pitts (1943). Norbert Wiener, a pioneer of cybernetics and control theory (Wiener, 1948), worked with McCulloch and Pitts and influenced a number of young researchers including Marvin Minsky, who may have been the first to develop a working neural network in hardware in 1951 (see Minsky and Papert, 1988, pp. ix–x). Meanwhile, in Britain, W. Ross Ashby (also a pioneer of cybernetics; see Ashby, 1940), Alan Turing, Grey Walter, and others formed the Ratio Club for “those who had Wiener’s ideas before Wiener’s book appeared.” Ashby’s *Design for a Brain* (1948, 1952) put forth the idea that intelligence could be created by the use of **homeostatic** devices containing appropriate feedback loops to achieve stable adaptive behavior. Turing (1948) wrote a research report titled *Intelligent Machinery* that begins with the sentence “I propose to investigate the question as to whether it is possible for machinery to show intelligent behaviour” and goes on to describe a recurrent neural network architecture he called “B-type unorganized machines” and an approach to training them. Unfortunately, the report went unpublished until 1969, and was all but ignored until recently.

Frank Rosenblatt (1957) invented the modern “perceptron” and proved the perceptron convergence theorem (1960), although it had been foreshadowed by purely mathematical work outside the context of neural networks (Agmon, 1954; Motzkin and Schoenberg, 1954). Some early work was also done on multilayer networks, including **Gamba perceptrons** (Gamba *et al.*, 1961) and **madalines** (Widrow, 1962). *Learning Machines* (Nilsson, 1965) covers much of this early work and more. The subsequent demise of early perceptron research efforts was hastened (or, the authors later claimed, merely explained) by the book *Perceptrons* (Minsky and Papert, 1969), which lamented the field’s lack of mathematical rigor. The book pointed out that single-layer perceptrons could represent only linearly separable concepts and noted the lack of effective learning algorithms for multilayer networks.

The papers in (Hinton and Anderson, 1981), based on a conference in San Diego in 1979, can be regarded as marking the renaissance of connectionism. The two-volume “PDP” (Parallel Distributed Processing) anthology (Rumelhart *et al.*, 1986a) and a short article in *Nature* (Rumelhart *et al.*, 1986b) attracted a great deal of attention—indeed, the number of papers on “neural networks” multiplied by a factor of 200 between 1980–84 and 1990–94. The analysis of neural networks using the physical theory of magnetic spin glasses (Amit *et al.*, 1985) tightened the links between statistical mechanics and neural network theory—providing not only useful mathematical insights but also *respectability*. The back-propagation technique had been invented quite early (Bryson and Ho, 1969) but it was rediscovered several times (Werbos, 1974; Parker, 1985).

Support vector machines were originated in the 1990s (Cortes and Vapnik, 1995) and are now the subject of a fast-growing literature, including textbooks such as Cristianini and Shawe-Taylor (2000). They have proven to be very popular and effective for tasks such as text categorization (Joachims, 2001), bioinformatics research (Brown *et al.*, 2000), and natural language processing, such as the handwritten digit recognition of DeCoste and Scholkopf (2002). A related technique that also uses the “kernel trick” to implicitly represent an exponential feature space is the voted perceptron (Collins and Duffy, 2002).

The probabilistic interpretation of neural networks has several sources, including Baum and Wilczek (1988) and Bridle (1990). The role of the sigmoid function is discussed by Jordan (1995). Bayesian parameter learning for neural networks was proposed by MacKay (1992) and is explored further by Neal (1996). The capacity of neural networks to represent functions was investigated by Cybenko (1988, 1989), who showed that two hidden layers are enough to represent any function and a single layer is enough to represent any *continuous* function. The “optimal brain damage” method for removing useless connections is by LeCun *et al.* (1989), and Sietsma and Dow (1988) show how to remove useless units. The tiling algorithm for growing larger structures is due to Mézard and Nadal (1989). LeCun *et al.* (1995) survey a number of algorithms for handwritten digit recognition. Improved error rates since then were reported by Belongie *et al.* (2002) for shape matching and DeCoste and Scholkopf (2002) for virtual support vectors.

The complexity of neural network learning has been investigated by researchers in computational learning theory. Early computational results were obtained by Judd (1990), who showed that the general problem of finding a set of weights consistent with a set of examples is NP-complete, even under very restrictive assumptions. Some of the first sample complexity results were obtained by Baum and Haussler (1989), who showed that the number of examples required for effective learning grows as roughly $W \log W$, where W is the number of weights.¹⁶ Since then, a much more sophisticated theory has been developed (Anthony and Bartlett, 1999), including the important result that the representational capacity of a network depends on the *size* of the weights as well as on their number.

The most popular kind of neural network that we did not cover is the **radial basis function**, or RBF, network. A radial basis function combines a weighted collection of kernels (usually Gaussians, of course) to do function approximation. RBF networks can be trained in two phases: first, an unsupervised clustering approach is used to train the parameters of the Gaussians—the means and variances—are trained, as in Section 20.3. In the second phase, the relative weights of the Gaussians are determined. This is a system of linear equations, which we know how to solve directly. Thus, both phases of RBF training have a nice benefit: the first phase is unsupervised, and thus does not require labelled training data, and the second phase, although supervised, is efficient. See Bishop (1995) for more details.

HOPFIELD NETWORKS

ASSOCIATIVE MEMORY

The **Recurrent networks**, in which units are linked in cycles, were mentioned in the chapter but not explored in depth. **Hopfield networks** (Hopfield, 1982) are probably the best-understood class of recurrent networks. They use *bidirectional* connections with *symmetric* weights (i.e., $W_{i,j} = W_{j,i}$), all of the units are both input and output units, the activation function g is the sign function, and the activation levels can only be ± 1 . A Hopfield network functions as an **associative memory**: after the network trains on a set of examples, a new stimulus will cause it to settle into an activation pattern corresponding to the example in the training set that *most closely resembles* the new stimulus. For example, if the training set consists of a set of photographs, and the new stimulus is a small piece of one of the photographs, then the network activation levels will reproduce the photograph from which the piece was

¹⁶ This approximately confirmed “Uncle Bernie’s rule.” The rule was named after Bernie Widrow, who recommended using roughly ten times as many examples as weights.

taken. Notice that the original photographs are not stored separately in the network; each weight is a partial encoding of all the photographs. One of the most interesting theoretical results is that Hopfield networks can reliably store up to $0.138N$ training examples, where N is the number of units in the network.

Boltzmann machines (Hinton and Sejnowski, 1983, 1986) also use symmetric weights, but include hidden units. In addition, they use a *stochastic* activation function, such that the probability of the output being 1 is some function of the total weighted input. Boltzmann machines therefore undergo state transitions that resemble a simulated annealing search (see Chapter 4) for the configuration that best approximates the training set. It turns out that Boltzmann machines are very closely related to a special case of Bayesian networks evaluated with a stochastic simulation algorithm. (See Section 14.5.)

The first application of the ideas underlying kernel machines was by Aizerman *et al.* (1964), but the full development of the theory, under the heading of support vector machines, is due to Vladimir Vapnik and colleagues (Boser *et al.*, 1992; Vapnik, 1998). Cristianini and Shawe-Taylor (2000) and Scholkopf and Smola (2002) provide rigorous introductions; a friendlier exposition appears in the *AI Magazine* article by Cristianini and Schölkopf (2002).

The material in this chapter brings together work from the fields of statistics, pattern recognition, and neural networks, so the story has been told many times in many ways. Good texts on Bayesian statistics include those by DeGroot (1970), Berger (1985), and Gelman *et al.* (1995). Hastie *et al.* (2001) provide an excellent introduction to statistical learning methods. For pattern classification, the classic text for many years has been Duda and Hart (1973), now updated (Duda *et al.*, 2001). For neural nets, Bishop (1995) and Ripley (1996) are the leading texts. The field of computational neuroscience is covered by Dayan and Abbott (2001). The most important conference on neural networks and related topics is the annual NIPS (Neural Information Processing Conference) conference, whose proceedings are published as the series *Advances in Neural Information Processing Systems*. Papers on learning Bayesian networks also appear in the *Uncertainty in AI* and *Machine Learning* conferences and in several statistics conferences. Journals specific to neural networks include *Neural Computation*, *Neural Networks*, and the *IEEE Transactions on Neural Networks*.

EXERCISES

20.1 The data used for Figure 20.1 can be viewed as being generated by h_5 . For each of the other four hypotheses, generate a data set of length 100 and plot the corresponding graphs for $P(h_i|d_1, \dots, d_m)$ and $P(D_{m+1} = \text{lime}|d_1, \dots, d_m)$. Comment on your results.

20.2 Repeat Exercise 20.1, this time plotting the values of $P(D_{m+1} = \text{lime}|h_{\text{MAP}})$ and $P(D_{m+1} = \text{lime}|h_{\text{ML}})$.

20.3 Suppose that Ann’s utilities for cherry and lime candies are c_A and ℓ_A , whereas Bob’s utilities are c_B and ℓ_B . (But once Ann has unwrapped a piece of candy, Bob won’t buy it.) Presumably, if Bob likes lime candies much more than Ann, it would be wise to sell for Ann

to sell her bag of candies once she is sufficiently sure of its lime content. On the other hand, if Ann unwraps too many candies in the process, the bag will be worth less. Discuss the problem of determining the optimal point at which to sell the bag. Determine the expected utility of the optimal procedure, given the prior distribution from Section 20.1.

20.4 Two statisticians go to the doctor and are both given the same prognosis: A 40% chance that the problem is the deadly disease A , and a 60% chance of the fatal disease B . Fortunately, there are anti- A and anti- B drugs that are inexpensive, 100% effective, and free of side-effects. The statisticians have the choice of taking one drug, both, or neither. What will the first statistician (an avid Bayesian) do? How about the second statistician, who always uses the maximum likelihood hypothesis?

The doctor does some research and discovers that disease B actually comes in two versions, dextro- B and levo- B , which are equally likely and equally treatable by the anti- B drug. Now that there are three hypotheses, what will the two statisticians do?

20.5 Explain how to apply the boosting method of Chapter 18 to naive Bayes learning. Test the performance of the resulting algorithm on the restaurant learning problem.

20.6 Consider m data points (x_j, y_j) , where the y_j s are generated from the x_j s according to the linear Gaussian model in Equation (20.5). Find the values of θ_1 , θ_2 , and σ that maximize the conditional log likelihood of the data.

20.7 Consider the noisy-OR model for fever described in Section 14.3. Explain how to apply maximum-likelihood learning to fit the parameters of such a model to a set of complete data. (*Hint:* use the chain rule for partial derivatives.)

20.8 This exercise investigates properties of the Beta distribution defined in Equation (20.6).

- a. By integrating over the range $[0, 1]$, show that the normalization constant for the distribution $\text{beta}[a, b]$ is given by $\alpha = \Gamma(a + b)/\Gamma(a)\Gamma(b)$ where $\Gamma(x)$ is the **Gamma function**, defined by $\Gamma(x + 1) = x \cdot \Gamma(x)$ and $\Gamma(1) = 1$. (For integer x , $\Gamma(x + 1) = x!$.)
- b. Show that the mean is $a/(a + b)$.
- c. Find the mode(s) (the most likely value(s) of θ).
- d. Describe the distribution $\text{beta}[\epsilon, \epsilon]$ for very small ϵ . What happens as such a distribution is updated?

GAMMA FUNCTION

20.9 Consider an arbitrary Bayesian network, a complete data set for that network, and the likelihood for the data set according to the network. Give a simple proof that the likelihood of the data cannot decrease if we add a new link to the network and recompute the maximum-likelihood parameter values.

20.10 Consider the application of EM to learn the parameters for the network in Figure 20.10(a), given the true parameters in Equation (20.7).

- a. Explain why the EM algorithm would not work if there were just two attributes in the model rather than three.
- b. Show the calculations for the first iteration of EM starting from Equation (20.8).

- c. What happens if we start with all the parameters set to the same value p ? (*Hint:* you may find it helpful to investigate this empirically before deriving the general result.)
- d. Write out an expression for the log likelihood of the tabulated candy data on page 729 in terms of the parameters, calculate the partial derivatives with respect to each parameter, and investigate the nature of the fixed point reached in part (c).

20.11 Construct by hand a neural network that computes the XOR function of two inputs. Make sure to specify what sort of units you are using.

20.12 Construct a support vector machine that computes the XOR function. It will be convenient to use values of 1 and -1 instead of 1 and 0 for the inputs and for the outputs. So an example looks like $([-1, 1], 1)$ or $([-1, -1], -1)$. It is typical to map an input \mathbf{x} into a space consisting of five dimensions, the two original dimensions x_1 and x_2 , and the three combination x_1^2 , x_2^2 and $x_1 x_2$. But for this exercise we will consider only the two dimensions x_1 and $x_1 x_2$. Draw the four input points in this space, and the maximal margin separator. What is the margin? Now draw the separating line back in the original Euclidean input space.

20.13 A simple perceptron cannot represent XOR (or, generally, the parity function of its inputs). Describe what happens to the weights of a four-input, step-function perceptron, beginning with all weights set to 0.1, as examples of the parity function arrive.

20.14 Recall from Chapter 18 that there are 2^{2^n} distinct Boolean functions of n inputs. How many of these are representable by a threshold perceptron?



20.15 Consider the following set of examples, each with six inputs and one target output:

I_1	1	1	1	1	1	1	1	0	0	0	0	0	0
I_2	0	0	0	1	1	0	0	1	1	0	1	0	1
I_3	1	1	1	0	1	0	0	1	1	0	0	0	1
I_4	0	1	0	0	1	0	0	1	0	1	1	1	0
I_5	0	0	1	1	0	1	1	0	1	1	0	0	1
I_6	0	0	0	1	0	1	0	1	1	0	1	1	0
T	1	1	1	1	1	1	0	1	0	0	0	0	0

- a. Run the perceptron learning rule on these data and show the final weights.
- b. Run the decision tree learning rule, and show the resulting decision tree.
- c. Comment on your results.

20.16 Starting from Equation (20.13), show that $\partial L / \partial W_j = Err \times a_j$.

20.17 Suppose you had a neural network with linear activation functions. That is, for each unit the output is some constant c times the weighted sum of the inputs.

- a. Assume that the network has one hidden layer. For a given assignment to the weights \mathbf{W} , write down equations for the value of the units in the output layer as a function of \mathbf{W} and the input layer \mathbf{I} , without any explicit mention to the output of the hidden layer. Show that there is a network with no hidden units that computes the same function.

- b. Repeat the calculation in part (a), this time for a network with any number of hidden layers. What can you conclude about linear activation functions?



20.18 Implement a data structure for layered, feed-forward neural networks, remembering to provide the information needed for both forward evaluation and backward propagation. Using this data structure, write a function NEURAL-NETWORK-OUTPUT that takes an example and a network and computes the appropriate output values.

20.19 Suppose that a training set contains only a single example, repeated 100 times. In 80 of the 100 cases, the single output value is 1; in the other 20, it is 0. What will a back-propagation network predict for this example, assuming that it has been trained and reaches a global optimum? (*Hint:* to find the global optimum, differentiate the error function and set to zero.)

20.20 The network in Figure 20.24 has four hidden nodes. This number was chosen somewhat arbitrarily. Run systematic experiments to measure the learning curves for networks with different numbers of hidden nodes. What is the optimal number? Would it be possible to use a cross-validation method to find the best network before the fact?

20.21 Consider the problem of separating N data points into positive and negative examples using a linear separator. Clearly, this can always be done for $N = 2$ points on a line of dimension $d = 1$, regardless of how the points are labelled or where they are located (unless the points are in the same place).

- a. Show that it can always be done for $N = 3$ points on a plane of dimension $d = 2$, unless they are collinear.
- b. Show that it cannot always be done for $N = 4$ points on a plane of dimension $d = 2$.
- c. Show that it can always be done for $N = 4$ points in a space of dimension $d = 3$, unless they are coplanar.
- d. Show that it cannot always be done for $N = 5$ points in a space of dimension $d = 3$.
- e. The ambitious student may wish to prove that N points in general position (but not $N + 1$) are linearly separable in a space of dimension $N - 1$. From this it follows that the **VC dimension** (see Chapter 18) of linear halfspaces in dimension $N - 1$ is N .

No Pizza for You: Value-based Plan Selection in BDI Agents

Stephen Cranefield

University of Otago

stephen.cranefield
@otago.ac.nz

Michael Winikoff

University of Otago

michael.winikoff
@otago.ac.nz

Virginia Dignum

TU Delft

M.V.Dignum@tudelft.nl

Frank Dignum

Utrecht University

F.P.M.Dignum@uu.nl

Abstract

Autonomous agents are increasingly required to be able to make moral decisions. In these situations, the agent should be able to reason about the ethical bases of the decision and explain its decision in terms of the moral values involved. This is of special importance when the agent is interacting with a user and should understand the value priorities of the user in order to provide adequate support. This paper presents a model of agent behavior that takes into account user preferences and moral values.

1 Introduction

Social assistive software and robots is an increasing area of research and development [Oishi *et al.*, 2010]. Such systems are envisioned as supporting their users in daily activities, such as medication monitoring and agenda reminders. The use of such systems, often involving vulnerable users, raises substantial ethical concerns: How is users' privacy protected? Which tasks should the system be allowed to perform and who can regulate and monitor this? Will artificial caregivers displace human caregivers, negatively affecting both client welfare and health-care provider jobs? How to ensure that the system is aligned with and able to uphold the moral, societal and legal values of the user and society? This paper focuses on this last question and proposes means to integrate values into the planning of agent activities.

Ethical decision making can be understood as action selection under conditions where principles, values, and social norms play a central role in determining which behavioral attitudes and responses are acceptable. However, the way agents choose between different possible courses of action ("plans"), is often left to the programmer of the agent. This may be done by statically prioritizing the plans by ordering them in a file or by using (implicit) criteria that are predetermined (and usually are utility-, resource- or time-optimizing).

Although this usually works well in applications where agents only have goals related to a particular type of situation, it does not transfer to applications where agents have several different tasks that are not directly related, e.g. an elderly companion robot or an e-health coach. In these applications different interactions might require different criteria to optimize and long-term criteria might differ from short-term

objectives. Thus a person can eat a cake on his birthday while trying to lose weight over a longer period. At first sight it may seem inconsistent, but there is a balance between enjoyment of a birthday and long-term health. However, it would be better if the person would get the cake by walking to the shop rather than going there by car (provided it is within walking distance). In other words, the health criterion not only plays a role in the eating decision, but also in the transport decision.

In this paper we present a *computational mechanism* for using values to select between (hierarchical) plans in a consistent manner. Using values has two advantages. First there exists an extensive literature on human values and their relations. It shows that people have a common base system of values where the difference between people lies in the priorities they give to the values. It also indicates that values are relatively stable over the life span of a person. Thus they can be used as an underlying stable mechanism for decision making. Note that we do not claim that every decision is explicitly based on some value. Many decisions are made based on norms and longer term goals. However these norms and goals are often chosen based on the value system. Thus these decisions are indirectly influenced by the values.

A second advantage of using values is the ability to explain decisions over different situations in a relatively simple manner. This is important to generate a level of trust in the system by a human user in that the user can maintain a model of the system and can predict its future actions based on that model. Having underlying values explaining a wide variety of decisions in different situations makes this much easier than having explicit rules for all possible situations not directly related.

In the rest of the paper we will first sketch some background literature, indicating the added value of our approach (Section 2). Section 3 then presents a scenario to illustrate the use of values in the deliberation of a companion agent that advises a person on healthy living. Section 4 describes how this scenario can be modeled in our framework, and Section 5 defines the computational mechanism for reasoning about values. Section 6 concludes the paper and outlines future work.

2 Literature

This paper proposes a novel approach to plan selection in reactive planning, in which societal, moral and legal values of

users guide the planning process. As such it is based on current work on value-sensitive design and on planning. Values (e.g. honesty, beauty, respect, environmental care, Self-Enhancement) are key drivers in human decision making (see e.g. [Rokeach, 1973; Schwartz, 2012]). As such, values can be seen as criteria to measure the difference between two situations or for comparing *alternative* plans. Values are abstract and context dependent, and therefore cannot easily be measured directly. For example, the value *wealth* can include assets other than money, but can be approximated by the amount of money someone owns. Miceli and Castelfranchi [1989] discuss in depth the consequences of this indirect use of values.

Values combine two core properties: (1) *Genericity*: values are generic and can be instantiated in a wide range of concrete situations, and therefore can be seen as a very abstract goal (e.g. eating well, exercising and avoiding stress all contribute to the value ‘health’). (2) *Comparison*: values allow comparison of different situations with respect to that value (e.g. according to the value ‘health’, salad is preferred over pizza). In this sense, values become metrics that measure the effects of actions in different dimensions.

For strengthening their decisions and covering a wider range of decisions, individuals tend to rely on the influence of multiple values (e.g. environmental care and wealth). However, for certain decisions, the values involved can lead to contradictory preferences. For example, cycling to work through the rain might be good for the environment, but will leave you soaked and giving a bad impression in an important meeting.

In order to handle these contradictions, value-systems *internally order* values. There are two dimensions along which this ordering takes place. First, there is an intrinsic opposition between different basic values. This intrinsic opposition is depicted by Schwartz as a circle in which the values are placed (see Figure 1, adapted from [Schwartz, 2012]). Values that are close together on the circle work in the same direction and values on opposite sides drive people in opposite directions. For example, “achievement” and “benevolence” are conflicting values. This means that generally trying to do something that is primarily good for one’s own benefit (Self-Enhancement) is not necessarily the best for society. For example, making more profit by paying low wages is good for the employer’s wealth but bad for the wealth of employees. However, opposition of values does not mean that one value excludes another value. It mainly means that they are in general “pulling” in different directions and a balance must be found. For example, if wages are too high the company might go bankrupt and no one profits anymore.

The second ordering is a personal preference one. Some values are given a relative *importance* over others. When evaluating a decision with conflicting values, alternatives that satisfy the most important values tend to be preferred (e.g. if health is more important than wealth then a person will buy healthy food even if it is more expensive than junk food). So, the importance of values for a person determines how the balance is struck between conflicting values.

Value-Sensitive Design (VSD) is a theoretically grounded approach to the design of technology that accounts for human

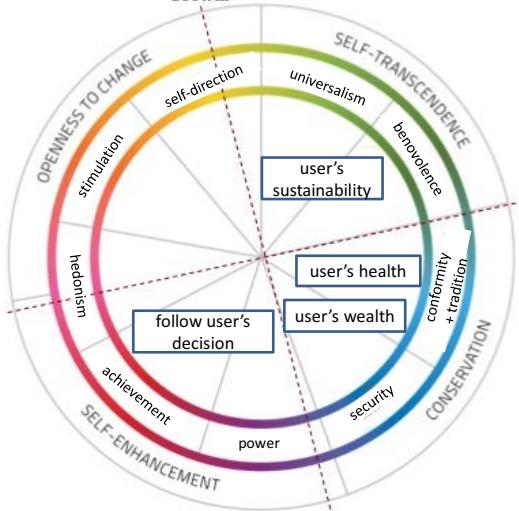


Figure 1: Schwartz’s value model annotated with scenario

values in a principled and comprehensive manner [Friedman *et al.*, 2013; van den Hoven, 2007]. A crucial step in VSD is the translation of values into design requirements. In this paper, we follow the *value hierarchy* approach proposed by van de Poel [2013], which links values, norms and design requirements or goals through a ‘*for the sake of*’ relationship. Formally, values can be linked to specific tasks or sub-plans in a plan tree, using a ‘*counts as*’ formalization, which enables reasoning about the motives to choose for a specific course of action. This is similar to work done in normative systems to link norms to agent actions [Grossi *et al.*, 2006] and implements the properties of values as described above.

Our proposal to link values to the plans of agents is similar to how Visser *et al.* [2016] add preferences to the plans of BDI (Belief-Desire-Intention) agents [Rao and Georgeff, 1995]. However, using values gives us a way to create consistency between decisions over different actions resulting in a generic approach that is still consistent with Visser *et al.* [2016].

Also related is work on integrating planning into BDI languages [Sardiña and Padgham, 2011; Sardiña *et al.*, 2006]. This work proposes the CANPLAN language. CANPLAN exploits the similarities between BDI languages and HTN planning to provide a construct $\text{Plan}(P)$ which does lookahead planning for the plan P . This work differs from our work in that they are doing full lookahead planning to find a course of action, whereas we are considering an under-constrained situation (with multiple options), and using the consequences of the available options to select from among them. Meneguzzi and de Silva [2015] survey other related work that incorporates planning into the BDI architecture.

Finally, Bordini *et al.* [2002] implemented one of Agent-Speak’s selection functions by integrating quantitative reasoning using TÆMS [Decker, 1996]. A key difference between our work and theirs is that they focused on intention selection, i.e. selecting *which* intention to execute next, whereas we are dealing with selecting *how* to achieve a given sub-goal.

3 Scenario

Given the growing elderly population in many countries, the development of health-care robots and virtual assistants is a significant area of research and, at least in Japan, a serious option for elderly care. The functionalities and requirements for these Elderly Care Artificial Systems (ECAS) are very diverse, but it is certain that ECAS will need to take decisions on behalf and for their users. In this paper, we explore the situation in which a extremely simplified ECAS should order and serve a meal to its user [McColl and Nejat, 2013].

In order to decide on the most suitable meal, the robot must consider the preferences of the user, the dietary prescriptions, the financial implications, the ease and speed of delivery, possibly the carbon footprint of the choice, and other issues. Besides contextual constraints (time, money, availability) this decision is guided by moral and societal values held by the user and his/her current priorities. An ECAS's highest value is to assist its user, which should be done in accordance with the values of the user. In this meal-assistance scenario, relevant high level user values are Self-Enhancement, Conservation and Self-Transcendence, using the Schwartz classification [Schwartz, 2012]. It should be noted that the values of Self-Enhancement and Self-Transcendence pull in different directions and need thus to be balanced carefully.

According to van de Poel [2013], the abstract values are translated into concrete values to govern the ECAS's actions, e.g. ‘follow user’s desires’ or ‘ensure user’s health’, which are finally linked to concrete system goals, e.g. ‘serve the desired meal’ or ‘serve a healthy meal’. Figure 2 depicts the value hierarchy, linking user values to concrete goals for the ECAS’s actions. If the desired meal is neither sustainable nor healthy a choice has to be made about which value to support most, and the priorities between values determine the outcome of the choice. In our approach, plans are selected using a multi-criteria optimisation (via a weighted sum). Here each criterion measures the extent to which a particular value is currently satisfied, given the plans for its child goals.

4 Model

In modeling the problem we take two aspects into account: the agent’s goals and plans; and the values and their relationships.

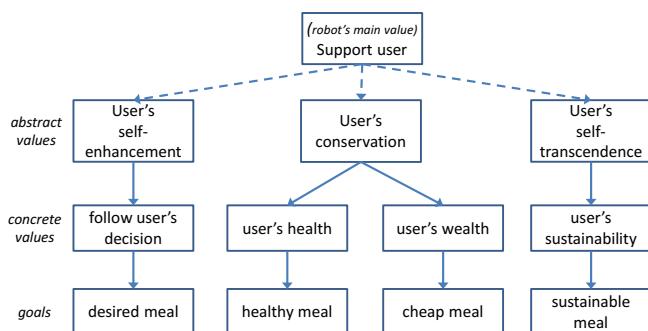


Figure 2: Value Tree

4.1 Goals and Plans

We begin with the agent’s goals and plans. We assume BDI-style plans (excluding cycles), where each plan has a goal that it achieves, a context condition that indicates in which situations the plan can be used, and a plan body. Following AgentSpeak(L) and Jason [Rao, 1996; Bordini *et al.*, 2007] we consider a plan body to be a sequence of steps (actions or sub-goals), but this restriction can be easily relaxed. Space precludes a detailed exposition of BDI languages.

The options for BDI agents to achieve their goals can be represented as a goal-plan tree, where a goal node has as children the plans that can be used to achieve it (an “or” relationship), and a plan node has as children its sub-goals (“and”, or more precisely “seq”). We extend our BDI language by annotating actions or plans with their effects on the Value State (defined below), similar to how Visser *et al.* [2016] extend goal-plan trees with preferences. For example, the plan `evil_pizza` (left side of Figure 3) is annotated “Desire: +20” indicating that it increases the Value State of Desire by 20.

Figure 3 shows a goal-plan tree for the scenario. The top-level goal has a single plan with three sub-goals: choosing a meal, preparing the meal, and consuming the meal. There are three choices: toast (which is highly unhealthy due to inadequate nutrition, but both sustainable and cheap), a frozen meal (most healthy), and pizza (most desired) with two possible providers: a local pizza company (within walking distance), and a multinational “evil” pizza company that is cheaper but less sustainable. Figure 3 elides the bindings and context conditions that are needed to constrain the plans chosen to be consistent with the selection made earlier when achieving `choose`. The annotations “+” and “-” before a variable’s name indicate whether the variable has values produced by the goal in question (“-”), or whether the goal in question uses values produced by earlier goals (“+”). The `weather` goal (bottom left) instantiates a variable (`W`) that depends on the environment. For such ‘query goals’ we assume we have a probabilistic model of the possible variable bindings, which appear as child nodes of the query goal.

4.2 Values

A *Value* is an abstract representation of a human driver (e.g. Self-Enhancement, Self-Conservation, Self-Transcendence). As we have explained in Section 2, values can be in conflict and also have a relative importance. These two aspects together determine how much time or effort a person (or in our case agent) will spend to promote the different values.

We start by determining a target for each value, $T(v)$. This is a number that is compared to the current Value State to determine the current need to advance that value. To reflect observations in human behavior [Schwartz, 2012], a design constraint in our current model is that conflicting values (that pull decisions in opposite directions) cannot both be important at the same time. For example, the annotations assigned to plans and the relative importance of goals (“salience”, defined below) should reflect that decisions related to Self-Enhancement can potentially be at odds with those related to Self-Transcendence, and therefore often cannot be realized

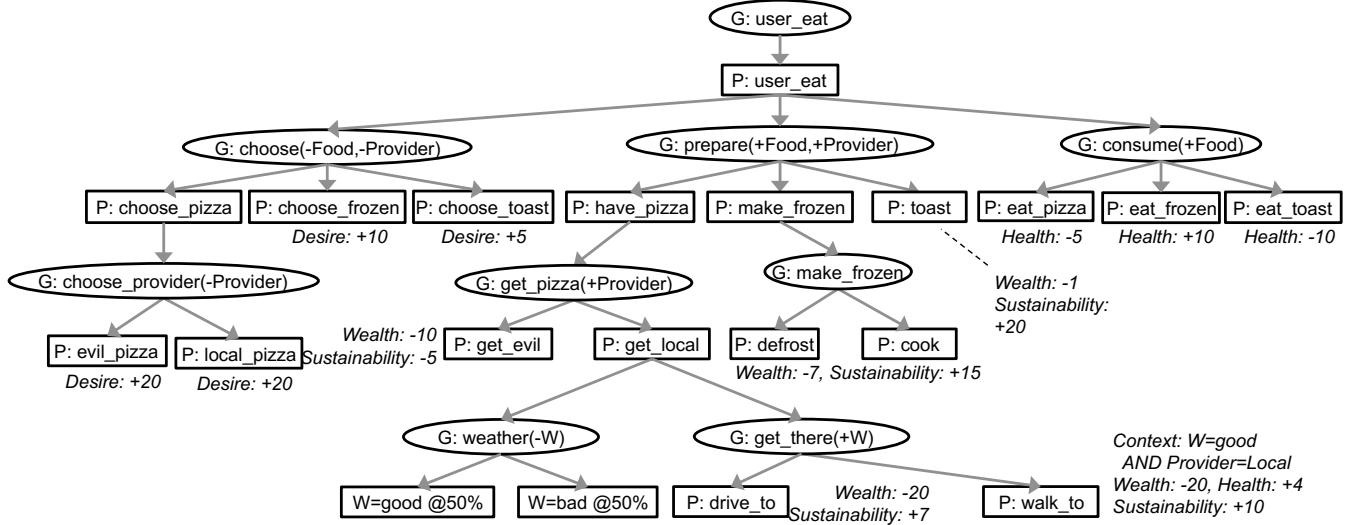


Figure 3: Goal-plan tree annotated with values

together. On the other hand, values that are very much aligned should have compatible importance. Finally, for simplicity's sake, we assume (for now) that the $T(v)$ remain constant during the life of the agent.

At each point in time, for a given value v , the agent also has a Value State, $S(v)$, that represents the current level of "satisfaction" for the value, i.e. the current level at which the value is experienced. For example, if the user has been deprived of coffee for a long time, then its hedonism Value State might be low, which will make the need to satisfy it more urgent. Formally, $S(v)$ assigns to each value type a number. The Value State is represented numerically on the same scale as $T(v)$, i.e. $S(v) < T(v)$ indicates that the Value State is below the target. We assume that $S(v)$ decays over time (the precise decay pattern needs to be specified, but our approach is agnostic). Decay represents the fact that if for some time nothing has been done to promote that value, its satisfaction will decrease. Decay is calculated at the 'concrete value' nodes of the value tree (see Figure 2) using the same decay function for all nodes. This is propagated up to the root in a very simple way: if all children of a node decay then the node itself also decays (using the same decay function).

Values also have a *current importance*, $CI(v)$, consisting of two components: the salience of the value in a situation $sal(v, g)$ and the difference between the current Value State and the target amount of that value ($T(v) - S(v)$) [Di Tosto and Dignum, 2013]. We assume the salience of a value to be given for each goal. We calculate $CI(v)$ as

$$CI(v) = sal(v, g) \times f(T(v) - S(v))$$

using some function $f(x)$ (cf. Section 5.1 for the actual implementation) to model that the importance of a value does not grow linearly with the distance of its satisfaction to its target, but can have some threshold or other non-linear shape. However, we retain generality by permitting an arbitrary user-specified function for the importance of a given value, as a function of the distance from the Value State to the target.

Priorities between values are thus not static, but can depend on the context. For example, if $sal(wealth, prepare)$ is very high, but the $S(v)$ for wealth is well above $T(v)$, then the current priority of the value "wealth" may be quite low.

4.3 Extending AgentSpeak

We now briefly explain how the AgentSpeak language is extended to accommodate reasoning about courses of action using values at runtime. Specifically we use the Jason language [Bordini *et al.*, 2007] (which extends Rao's original AgentSpeak(L)). Although we do make use of some of Jason's extensions, our work can also be easily adapted to apply to the original AgentSpeak, and to other BDI languages.

Inspired by Sardiña and Padgham [2011] and Sardiña *et al.* [2006] we introduce a construct " $\mathcal{VBR}(G)$ " denoting that sub-goal G should be achieved, but with the selection of choices to achieve it being guided by value-based reasoning.

At compile-time we pre-process this construct away¹, resulting in a collection of constraints, and a modified Jason program. Where the original program has $\mathcal{VBR}(G)$, the modified program invokes an external constraint solver to find a best course of action (in accordance with the values and their Value State), and it then uses the recommended course of action to guide the achievement of G .

We transform the agent program by firstly identifying the plans that are involved in achieving G , either directly because their trigger is $!G$, or indirectly (recursively) because their trigger is a goal that is a sub-goal of a plan that is used to achieve G . Each of these plans is then modified by: (i) adding an annotation², and (ii) adding an additional test to the context condition. The annotation is used to carry an additional argument capturing the *choices* generated by the value-based reasoning.

¹Doing this allows Jason to be used without modification, but it does mean that plans cannot be updated at run-time.

²This is a Jason construct that allows (e.g.) a goal to have additional information associated with it.

ated by the value-based reasoning³. The additional test in the context condition is that the choice specified by the value-based reasoning is this plan. Formally, we use the annotation *choice(Choice)* to capture, when the plan is called, the path through the goal-plan tree generated by the value-based reasoning. We define *Choice*[G] to be the numerical index of the plan chosen to achieve G. The additional context condition for the *i*th plan to achieve G is then a test that either *Choice* is not ground, or that it indicates the current plan, i.e. $(\neg \text{ground}(\text{Choice})) \vee \text{Choice}[G] = i$. For example, the Jason plan $+!g : c \leftarrow P$ (where $!g$ is the goal being handled, c the context condition, and P the plan body) is modified to the following (where P' is P where sub-goals have the annotation *choice(Choice)* added).

```
+!g[choice(Choice)]
: ((not .ground(Choice)) | Choice[g] = i) ∧ c
  ← P'.
```

Secondly, we generate the constraints from the goal-plan tree (which is itself derived from the program). Note that the constraints do not change, and hence can be generated at compile-time. We discuss this process below in Section 5.

Finally, we replace $\mathcal{VBR}(G)$ with the sequence of steps: *.callSolver(Choice)* and $!G[\text{choice}(\text{Choice})]$. The first step (implemented as an internal action) invokes an external solver to solve the constraints. The second step calls the sub-goal G , but with the output from the solver provided as an annotation. An exception is made for plans with an initial query goal (see Section 5.1). These are optimised using an expected value approach. A second optimisation is needed for the subgoals of such a plan after the query has been executed to instantiate its variable and before the plan is executed.

5 Process

The process for making value-based decisions has two steps. Firstly, we take the constraint problem that has been generated from the goal-plan tree (Section 5.1) and use a standard constraint solver to solve it, yielding a best course of action for the current situation. Secondly, we implement the selected course of action (by achieving⁴ the goal $!G[\text{choice}(\text{Choice})]$) in the modified Jason program. In Section 5.2 we analyse the scenario using this implementation.

5.1 Finding a best Course of Action

In order to determine a best course of action we generate constraints from the program, and then solve the constraints. We use constraints because, unlike [Thangarajah *et al.*, 2002; Visser *et al.*, 2016], we need to deal with dependencies between different parts of the tree.

Before defining the mapping we need to formally define goal-plan trees. A goal plan tree is represented by its root

³If using a BDI language other than Jason, then a copy of each plan can be added, extended by an additional argument.

⁴If the goal achievement fails, then failure handling will be used, e.g. for Jason using a pattern [Bordini *et al.*, 2007, Section 8.2]. One slight wrinkle is that we need to ensure that if $!G[\text{choice}(\text{Choice})]$ fails, that the recovery is not bound to the choices, i.e. use $!G$ in the recovery plan, not $!G[\text{choice}(\text{Choice})]$.

node. A node N comprises a name N^n , an optional annotation N^a that is a tuple of changes to the Value State, an optional context condition N^{cc} (a logical formula), the input (N^i) and output (N^o) variables (both sets of variables), an optional binding N^b of the form $\text{var} = \text{val}$ (allowing for multiple variables, i.e. var can be a tuple of variables and val a tuple of constants), a type N^t (either g or p for “Goal” or “Plan”), and a list of N^c child nodes $N^C = \{N_1^C, \dots, N_{N^c}^C\}$ (note that N^c , lower case “c”, is the *number* of children and N^C is the set of child nodes N_i^C).

We now define the mapping from a goal-plan tree to a constraint solving problem. For each node we declare a variable with the name of the node and type of tuple of Value State changes. We also declare variables that appear in any node’s N^i or N^o , and, for nodes that are goals, we declare a variable c_N^n (c for “chosen”) that is an array of N^c Booleans (0 or 1) constrained so that exactly one of them is true. The variable c_N^n represents which plan is chosen to achieve the goal corresponding to node N . Formally we define $d(N)$ to denote these declarations associated with a node N .

We generate the following constraints. For a plan node the value of the node is the sum of the node’s children. For a goal node we add a constraint for each child of the form $c_N^n[i] = 1 \Rightarrow N^n = (N_i^C)^n \wedge N^{cc} \wedge N^b$, i.e. if the choice is i , then the value of the node N is the value of its i th child N_i^C , and the context condition and binding of the i th child apply. Finally, if a node has an annotation, then the value of the node is simply that annotation (and the node’s type and children are ignored⁵). Formally:

$$c(N) = \begin{cases} N^n = N^a & \text{if } N^a \text{ is present} \\ N^n = \sum_{i=1}^{N^c} N_i^C & \text{else if } N^t = p \\ \sum_i c_N^n[i] = 1 & \text{otherwise} \\ \wedge \quad \wedge_{i=1}^{N^c} c_N^n[i] = 1 & \\ \Rightarrow (N^n = (N_i^C)^n \wedge N^{cc} \wedge N^b) \end{cases}$$

Plan nodes with an initial query goal⁶ are optimised using an expected value approach. Copies of the plan node’s subtree with the query goal removed are made for each binding, with distinct new node names N_i^n replacing each N^n in the original subtree. Distinct renamed copies of the query goal’s variable are made for each copied tree and set to the respective binding. Constraints are generated as shown above for each copy, and finally the following constraint is added: $N^n = \sum_i p_i N_i^n$, where the p_i are the binding probabilities.

We apply the functions $d(N)$ and $c(N)$ to all nodes and collect the results. This gives us the constraints and variables.

Figure 4 shows an extract of a goal-plan tree (which corresponds to a Jason program, not shown for space reasons), and the constraints that are generated (using our implementation) from the goal-plan tree. In the tree goal nodes are *italic* and a node of the form $N_{x=c}^\delta$ indicates that the plan labeled N is annotated with value change δ , and has a plan selection constraint or effect that constrains variable x to equal c (and $[x_1 = c_1, x_2 = c_2]$ constrains x_i to equal C_i). Here *food* values represent pizza (1), frozen food (2), and toast (3), while

⁵Therefore if a node has an annotation then the tree below it cannot contain any bindings.

⁶We do not currently handle more complex uses of query goals.

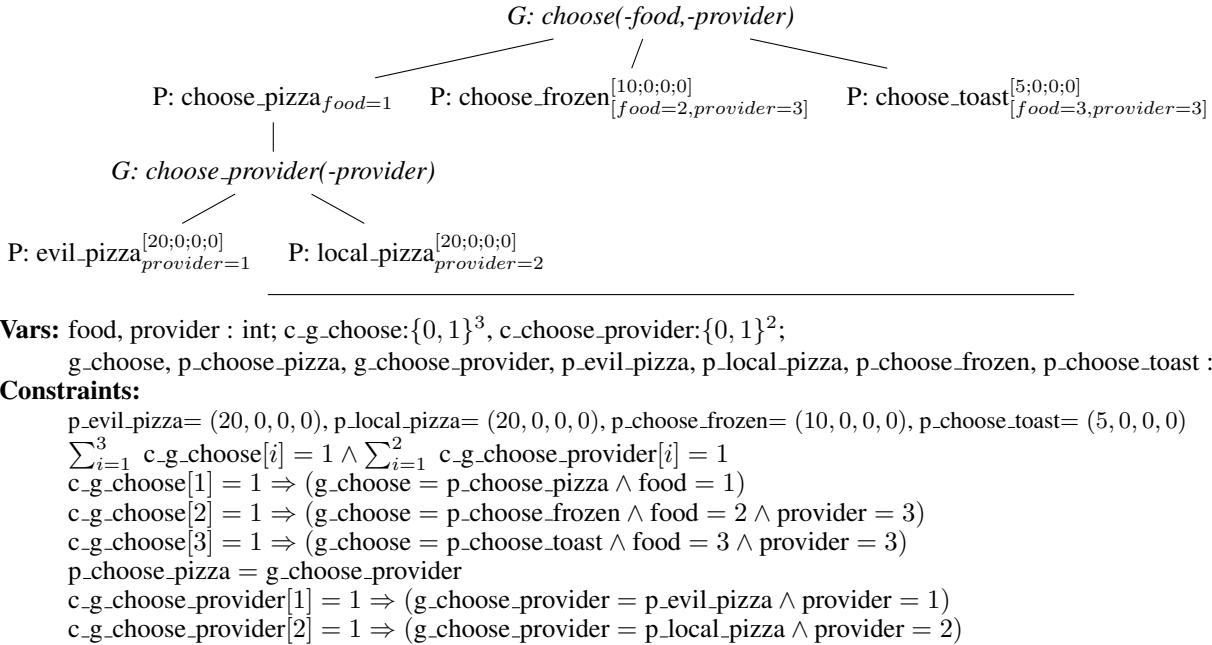


Figure 4: Example goal-plan tree and the constraints generated from it (“c_” is short for “chosen.”)

provider values represent evil pizza (1), local pizza (2), and home (3).

The objective function that we minimize, subject to these constraints, is $CI(v) = \text{sal}(v, g) \times f(T(v) - S(v))$, summed over the different values. We use the function $f(x) = (\max\{x, 0\})^2$ (although other functions could obviously be used). Thus, the constraints are solved while minimizing:

$$\sum_{v \in \mathcal{V}} \text{sal}(v, g) \times (\max\{T(v) - S(v), 0\})^2$$

where \mathcal{V} is the set of all value types. The output from the constraint solver includes the values for each of the choices c_{N^n} - this allows us to guide the selection of plans to follow the recommended course of action.

5.2 The Scenario Revisited

The goal-plan tree in Figure 3 has been encoded and our implementation of the mapping in Section 5.1 has been used to generate constraints in MATLAB using the YALMIP (yalmip.github.io) optimisation library. We use MOSEK (mosek.com), a state-of-the-art industrial optimiser, as the underlying solver. The internal representation of the problem has 194 variables and 370 constraints.

In a situation where all four values (desire, health, wealth and sustainability) have equal salience, their targets are all 100, and their Value States are (110, 50, 80, 20), the best choice (found by the constraint solver in a fraction of a second⁷), is to get and eat pizza from the local provider. The computed Value State change is (20, -3, -20, 8.5), where

⁷ 0.2324 seconds, obtained from YALMIP’s *solvetime* property and averaged over 10 runs on a 2.6GHz Intel Core i7 running Windows 7.

the health and sustainability values are expected values from either walking or driving, depending on the weather.

However, in other situations, different decisions are appropriate. For instance, in a situation where all four values have equal importance (i.e. equal CI), the best choice is a frozen meal (with Value State change (10, 10, -7, 15)). On the other hand, if wealth is somewhat more important (i.e. $CI(\text{wealth})$ is sufficiently greater than the other values’ CI), and sustainability not important at all, then the cost saving offered by evil pizza makes it the best choice (Value State change of (20, -5, -10, -5)). Finally, if wealth is the overriding criterion, then toast becomes the best choice (5, -10, -1, 20).

6 Conclusions

In this paper, we propose a value-based approach to plan selection, that takes into account societal and ethical values that influence decision-making. Using values as basis for deliberation supports both stability over time and the ability to explain decisions over different situations in a relatively simple and cohesive manner. We show the potential of this approach on a simple scenario of a Elderly Care Artificial System (ECAS) as an example of social assistive technology.

We have described a mechanism for BDI agents to make decisions using value-based reasoning. The mechanism uses an external constraint solver, and does not require changing the BDI language or its implementation.

Future work is needed on the evaluation of the scalability and broader applicability of the approach, on a mechanism to use values to provide explanations of an agent’s behaviour, and on consideration of multi-agent decision making.

References

- [Bordini *et al.*, 2002] Rafael H. Bordini, Ana L. C. Bazzan, Rafael de Oliveira Jannone, Daniel M. Basso, Rosa Maria Vicari, and Victor R. Lesser. Agent-Speak(XL): efficient intention selection in BDI agents via decision-theoretic task scheduling. In *The First International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS)*, pages 1294–1302. ACM, 2002. doi:10.1145/545056.545122.
- [Bordini *et al.*, 2007] Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*. Wiley, 2007. ISBN 0470029005. doi:10.1002/9780470061848
- [Decker, 1996] Keith Decker. TÆMS: A Framework for Environment Centered Analysis & Design of Coordination Mechanisms. In G. O'Hare and N. Jennings, editors, *Foundations of Distributed Artificial Intelligence*, chapter 16, pages 429–448. Wiley, 1996. <http://mas.cs.umass.edu/paper/159>.
- [Di Tosto and Dignum, 2013] Gennaro Di Tosto and Frank Dignum. Simulating Social Behaviour Implementing Agents Endowed with Values and Drives. In Francesca Giardini and Frédéric Amblard, editors, *International Workshop on Multi-Agent-Based Simulation XIII (MABS): Revised Selected Papers*, pages 1–12. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. doi:10.1007/978-3-642-38859-0_1.
- [Friedman *et al.*, 2013] Batya Friedman, Peter H. Kahn, Alan Borning, and Alina Huldtgren. Value sensitive design and information systems. In Neelke Doorn, Daan Schuurbiers, Ibo van de Poel, and Michael E. Gorman, editors, *Early engagement and new technologies: Opening up the laboratory*, pages 55–95. Springer Netherlands, Dordrecht, 2013. doi:10.1007/978-94-007-7844-3_4.
- [Grossi *et al.*, 2006] Davide Grossi, John-Jules C.H. Meyer, and Frank Dignum. Classificatory aspects of counts-as: An analysis in modal logic. *Journal of Logic and Computation*, 16(5):613–643, 2006. doi:10.1093/logcom/exl027.
- [McColl and Nejat, 2013] Derek McColl and Goldie Nejat. Meal-time with a socially assistive robot and older adults at a long-term care facility. *Journal of Human-Robot Interaction*, 2(1):152–171, 2013. doi:10.5898/JHRI.2.1.McColl.
- [Meneguzzi and de Silva, 2015] Felipe Meneguzzi and Lavindra de Silva. Planning in BDI agents: a survey of the integration of planning algorithms and agent reasoning. *The Knowledge Engineering Review*, 30(1):1–44, 2015. doi:10.1017/S0269888913000337.
- [Miceli and Castelfranchi, 1989] Maria Miceli and Christiano Castelfranchi. A Cognitive Approach to Values. *Journal for the Theory of Social Behaviour*, 19(2):169–193, 1989. doi:10.1111/j.1468-5914.1989.tb00143.x.
- [Oishi *et al.*, 2010] Meeko Mitsuko K. Oishi, Ian M. Mitchell, and H.F. Machiel Van der Loos. *Design and use of assistive technology: social, technical, ethical, and economic challenges*. Springer Science & Business Media, 2010. doi:10.1007/978-1-4419-7031-2
- [Rao and Georgeff, 1995] Anand S. Rao and Michael P. Georgeff. BDI agents: From theory to practice. In Victor R. Lesser and Les Gasser, editors, *Proceedings of the First International Conference on Multiagent Systems*, pages 312–319. The MIT Press, 1995.
- [Rao, 1996] Anand S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In Walter Van de Velde and John Perrame, editors, *Workshop on Modelling Autonomous Agents in a Multi-Agent World*, volume 1038 of *LNAI*, pages 42–55. Springer, 1996.
- [Rokeach, 1973] M. Rokeach. Rokeach Values Survey. In *The Nature of Human Values*. 1973.
- [Sardiña and Padgham, 2011] Sebastian Sardiña and Lin Padgham. A BDI agent programming language with failure handling, declarative goals, and planning. *Autonomous Agents and Multi-Agent Systems*, 23(1):18–70, 2011. doi:10.1007/s10458-010-9130-9.
- [Sardiña *et al.*, 2006] Sebastian Sardiña, Lavindra de Silva, and Lin Padgham. Hierarchical Planning in BDI Agent Programming Languages: A Formal Approach. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 1001–1008. ACM, 2006. doi:10.1145/1160633.1160813.
- [Schwartz, 2012] S.H. Schwartz. An Overview of the Schwartz Theory of Basic Values. *Online Readings in Psychology and Culture*, 2(1), 2012. doi:10.9707/2307-0919.1116.
- [Thangarajah *et al.*, 2002] John Thangarajah, Michael Winikoff, Lin Padgham, and Klaus Fischer. Avoiding resource conflicts in intelligent agents. In F. van Harmelen, editor, *Proceedings of the 15th European Conference on Artificial Intelligence (ECAI)*, pages 18–22. IOS Press, 2002.
- [van de Poel, 2013] Ibo van de Poel. Translating values into design requirements. In Diane P. Michelfelder, Natasha McCarthy, and David E. Goldberg, editors, *Philosophy and Engineering: Reflections on Practice, Principles and Process*, pages 253–266. Springer Netherlands, Dordrecht, 2013. doi:10.1007/978-94-007-7762-0_20.
- [van den Hoven, 2007] Jeroen van den Hoven. ICT and value sensitive design. In Philippe Goujon, Sylvain Lavelle, Penny Duquenoy, Kai Kimppa, and Véronique Laurent, editors, *The Information Society: Innovation, Legitimacy, Ethics and Democracy*, University of Namur, Belgium 22–23 May 2006, pages 67–72. Springer US, Boston, MA, 2007. doi:10.1007/978-0-387-72381-5_8.
- [Visser *et al.*, 2016] Simeon Visser, John Thangarajah, James Harland, and Frank Dignum. Preference-based reasoning in BDI agent systems. *Autonomous Agents and Multi-Agent Systems*, 30(2):291–330, 2016. doi:10.1007/s10458-015-9288-2.

A Primer on Neural Network Models for Natural Language Processing

Yoav Goldberg

Draft as of October 6, 2015.

The most up-to-date version of this manuscript is available at <http://www.cs.biu.ac.il/~yogo/nnlp.pdf>. Major updates will be published on arxiv periodically. I welcome any comments you may have regarding the content and presentation. If you spot a missing reference or have relevant work you'd like to see mentioned, do let me know. first.last@gmail.com

Abstract

Over the past few years, neural networks have re-emerged as powerful machine-learning models, yielding state-of-the-art results in fields such as image recognition and speech processing. More recently, neural network models started to be applied also to textual natural language signals, again with very promising results. This tutorial surveys neural network models from the perspective of natural language processing research, in an attempt to bring natural-language researchers up to speed with the neural techniques. The tutorial covers input encoding for natural language tasks, feed-forward networks, convolutional networks, recurrent networks and recursive networks, as well as the computation graph abstraction for automatic gradient computation.

1. Introduction

For a long time, core NLP techniques were dominated by machine-learning approaches that used linear models such as support vector machines or logistic regression, trained over very high dimensional yet very sparse feature vectors.

Recently, the field has seen some success in switching from such linear models over sparse inputs to non-linear neural-network models over dense inputs. While most of the neural network techniques are easy to apply, sometimes as almost drop-in replacements of the old linear classifiers, there is in many cases a strong barrier of entry. In this tutorial I attempt to provide NLP practitioners (as well as newcomers) with the basic background, jargon, tools and methodology that will allow them to understand the principles behind the neural network models and apply them to their own work. This tutorial is expected to be self-contained, while presenting the different approaches under a unified notation and framework. It repeats a lot of material which is available elsewhere. It also points to external sources for more advanced topics when appropriate.

This primer is not intended as a comprehensive resource for those that will go on and develop the next advances in neural-network machinery (though it may serve as a good entry point). Rather, it is aimed at those readers who are interested in taking the existing, useful technology and applying it in useful and creative ways to their favourite NLP problems. For more in-depth, general discussion of neural networks, the theory behind them, advanced

optimization methods and other advanced topics, the reader is referred to other existing resources. In particular, the book by Bengio et al (2015) is highly recommended.

Scope The focus is on applications of neural networks to language processing tasks. However, some subareas of language processing with neural networks were decidedly left out of scope of this tutorial. These include the vast literature of language modeling and acoustic modeling, the use of neural networks for machine translation, and multi-modal applications combining language and other signals such as images and videos (e.g. caption generation). Caching methods for efficient runtime performance, methods for efficient training with large output vocabularies and attention models are also not discussed. Word embeddings are discussed only to the extent that is needed to understand in order to use them as inputs for other models. Other unsupervised approaches, including autoencoders and recursive autoencoders, also fall out of scope. While some applications of neural networks for language modeling and machine translation are mentioned in the text, their treatment is by no means comprehensive.

A Note on Terminology The word “feature” is used to refer to a concrete, linguistic input such as a word, a suffix, or a part-of-speech tag. For example, in a first-order part-of-speech tagger, the features might be “current word, previous word, next word, previous part of speech”. The term “input vector” is used to refer to the actual input that is fed to the neural-network classifier. Similarly, “input vector entry” refers to a specific value of the input. This is in contrast to a lot of the neural networks literature in which the word “feature” is overloaded between the two uses, and is used primarily to refer to an input-vector entry.

Mathematical Notation I use bold upper case letters to represent matrices (\mathbf{X} , \mathbf{Y} , \mathbf{Z}), and bold lower-case letters to represent vectors (\mathbf{b}). When there are series of related matrices and vectors (for example, where each matrix corresponds to a different layer in the network), superscript indices are used (\mathbf{W}^1 , \mathbf{W}^2). For the rare cases in which we want indicate the power of a matrix or a vector, a pair of brackets is added around the item to be exponentiated: $(\mathbf{W})^2$, $(\mathbf{W}^3)^2$. Unless otherwise stated, vectors are assumed to be row vectors. We use $[\mathbf{v}_1; \mathbf{v}_2]$ to denote vector concatenation.

2. Neural Network Architectures

Neural networks are powerful learning models. We will discuss two kinds of neural network architectures, that can be mixed and matched – feed-forward networks and Recurrent / Recursive networks. Feed-forward networks include networks with fully connected layers, such as the multi-layer perceptron, as well as networks with convolutional and pooling layers. All of the networks act as classifiers, but each with different strengths.

Fully connected feed-forward neural networks (Section 4) are non-linear learners that can, for the most part, be used as a drop-in replacement wherever a linear learner is used. This includes binary and multiclass classification problems, as well as more complex structured prediction problems (Section 8). The non-linearity of the network, as well as the ability to easily integrate pre-trained word embeddings, often lead to superior classification accuracy. A series of works (Chen & Manning, 2014; Weiss, Alberti, Collins, & Petrov, 2015; Pei, Ge, & Chang, 2015; Durrett & Klein, 2015) managed to obtain improved syntactic parsing results by simply replacing the linear model of a parser with a fully connected feed-forward network. Straight-forward applications of a feed-forward network as a classifier replacement (usually coupled with the use of pre-trained word vectors) provide benefits also for CCG supertagging (Lewis & Steedman, 2014), dialog state tracking (Henderson, Thomson, & Young, 2013), pre-ordering for statistical machine translation (de Gispert, Iglesias, & Byrne, 2015) and language modeling (Bengio, Ducharme, Vincent, & Janvin, 2003; Vaswani, Zhao, Fossum, & Chiang, 2013). Iyyer et al (2015) demonstrate that multi-layer feed-forward networks can provide competitive results on sentiment classification and factoid question answering.

Networks with convolutional and pooling layers (Section 9) are useful for classification tasks in which we expect to find strong local clues regarding class membership, but these clues can appear in different places in the input. For example, in a document classification task, a single key phrase (or an ngram) can help in determining the topic of the document (Johnson & Zhang, 2015). We would like to learn that certain sequences of words are good indicators of the topic, and do not necessarily care where they appear in the document. Convolutional and pooling layers allow the model to learn to find such local indicators, regardless of their position. Convolutional and pooling architecture show promising results on many tasks, including document classification (Johnson & Zhang, 2015), short-text categorization (Wang, Xu, Xu, Liu, Zhang, Wang, & Hao, 2015a), sentiment classification (Kalchbrenner, Grefenstette, & Blunsom, 2014; Kim, 2014), relation type classification between entities (Zeng, Liu, Lai, Zhou, & Zhao, 2014; dos Santos, Xiang, & Zhou, 2015), event detection (Chen, Xu, Liu, Zeng, & Zhao, 2015; Nguyen & Grishman, 2015), paraphrase identification (Yin & Schütze, 2015) semantic role labeling (Collobert, Weston, Bottou, Karlen, Kavukcuoglu, & Kuksa, 2011), question answering (Dong, Wei, Zhou, & Xu, 2015), predicting box-office revenues of movies based on critic reviews (Bitvai & Cohn, 2015) modeling text interestingness (Gao, Pantel, Gamon, He, & Deng, 2014), and modeling the relation between character-sequences and part-of-speech tags (Santos & Zadrozny, 2014).

In natural language we often work with structured data of arbitrary sizes, such as sequences and trees. We would like to be able to capture regularities in such structures, or to model similarities between such structures. In many cases, this means encoding the structure as a fixed width vector, which we can then pass on to another statistical

learner for further processing. While convolutional and pooling architectures allow us to encode arbitrary large items as fixed size vectors capturing their most salient features, they do so by sacrificing most of the structural information. Recurrent (Section 10) and recursive (Section 12) architectures, on the other hand, allow us to work with sequences and trees while preserving a lot of the structural information. Recurrent networks (Elman, 1990) are designed to model sequences, while recursive networks (Goller & Küchler, 1996) are generalizations of recurrent networks that can handle trees. We will also discuss an extension of recurrent networks that allow them to model stacks (Dyer, Ballesteros, Ling, Matthews, & Smith, 2015; Watanabe & Sumita, 2015).

Recurrent models have been shown to produce very strong results for language modeling, including (Mikolov, Karafiat, Burget, Cernocky, & Khudanpur, 2010; Mikolov, Kombrink, Lukáš Burget, Černocky, & Khudanpur, 2011; Mikolov, 2012; Duh, Neubig, Sudoh, & Tsukada, 2013; Adel, Vu, & Schultz, 2013; Auli, Galley, Quirk, & Zweig, 2013; Auli & Gao, 2014); as well as for sequence tagging (Irsoy & Cardie, 2014; Xu, Auli, & Clark, 2015; Ling, Dyer, Black, Trancoso, Fernandez, Amir, Marujo, & Luis, 2015b), machine translation (Sundermeyer, Alkhouri, Wuebker, & Ney, 2014; Tamura, Watanabe, & Sumita, 2014; Sutskever, Vinyals, & Le, 2014; Cho, van Merriënboer, Gulcehre, Bahdanau, Bougares, Schwenk, & Bengio, 2014b), dependency parsing (Dyer et al., 2015; Watanabe & Sumita, 2015), sentiment analysis (Wang, Liu, SUN, Wang, & Wang, 2015b), noisy text normalization (Chrupala, 2014), dialog state tracking (Mrkšić, Ó Séaghdha, Thomson, Gasic, Su, Vandyke, Wen, & Young, 2015), response generation (Sordoni, Galley, Auli, Brockett, Ji, Mitchell, Nie, Gao, & Dolan, 2015), and modeling the relation between character sequences and part-of-speech tags (Ling et al., 2015b).

Recursive models were shown to produce state-of-the-art or near state-of-the-art results for constituency (Socher, Bauer, Manning, & Andrew Y., 2013) and dependency (Le & Zuidema, 2014; Zhu, Qiu, Chen, & Huang, 2015a) parse re-ranking, discourse parsing (Li, Li, & Hovy, 2014), semantic relation classification (Hashimoto, Miwa, Tsuruoka, & Chikayama, 2013; Liu, Wei, Li, Ji, Zhou, & WANG, 2015), political ideology detection based on parse trees (Iyyer, Enns, Boyd-Graber, & Resnik, 2014b), sentiment classification (Socher, Perelygin, Wu, Chuang, Manning, Ng, & Potts, 2013; Hermann & Blunsom, 2013), target-dependent sentiment classification (Dong, Wei, Tan, Tang, Zhou, & Xu, 2014) and question answering (Iyyer, Boyd-Graber, Claudino, Socher, & Daumé III, 2014a).

3. Feature Representation

Before discussing the network structure in more depth, it is important to pay attention to how features are represented. For now, we can think of a feed-forward neural network as a function $NN(\mathbf{x})$ that takes as input a d_{in} dimensional vector \mathbf{x} and produces a d_{out} dimensional output vector. The function is often used as a *classifier*, assigning the input \mathbf{x} a degree of membership in one or more of d_{out} classes. The function can be complex, and is almost always non-linear. Common structures of this function will be discussed in Section 4. Here, we focus on the input, \mathbf{x} . When dealing with natural language, the input \mathbf{x} encodes features such as words, part-of-speech tags or other linguistic information. Perhaps the biggest jump when moving from sparse-input linear models to neural-network based models is to stop representing each feature as a unique dimension (the so called one-hot representation) and representing them instead as dense vectors. That is, each core feature is *embedded* into a d dimensional space, and represented as a vector in that space.¹ The embeddings (the vector representation of each core feature) can then be trained like the other parameter of the function NN . Figure 1 shows the two approaches to feature representation.

The feature embeddings (the values of the vector entries for each feature) are treated as *model parameters* that need to be trained together with the other components of the network. Methods of training (or obtaining) the feature embeddings will be discussed later. For now, consider the feature embeddings as given.

The general structure for an NLP classification system based on a feed-forward neural network is thus:

1. Extract a set of core linguistic features f_1, \dots, f_k that are relevant for predicting the output class.
2. For each feature f_i of interest, retrieve the corresponding vector $v(f_i)$.
3. Combine the vectors (either by concatenation, summation or a combination of both) into an input vector \mathbf{x} .
4. Feed \mathbf{x} into a non-linear classifier (feed-forward neural network).

The biggest change in the input, then, is the move from sparse representations in which each feature is its own dimension, to a dense representation in which each feature is mapped to a vector. Another difference is that we extract only *core features* and not feature combinations. We will elaborate on both these changes briefly.

Dense Vectors vs. One-hot Representations What are the benefits of representing our features as vectors instead of as unique IDs? Should we always represent features as dense vectors? Let's consider the two kinds of representations:

One Hot Each feature is its own dimension.

- Dimensionality of one-hot vector is same as number of distinct features.

-
1. Different feature types may be embedded into different spaces. For example, one may represent word features using 100 dimensions, and part-of-speech features using 20 dimensions.

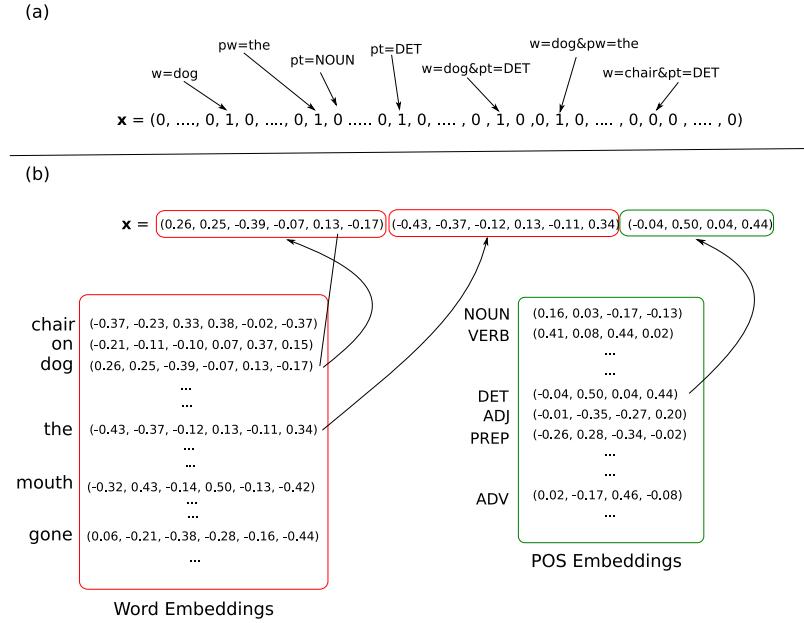


Figure 1: **Sparse vs. dense feature representations.** Two encodings of the information: *current word is “dog”*; *previous word is “the”*; *previous pos-tag is “DET”*. (a) Sparse feature vector. Each dimension represents a feature. Feature combinations receive their own dimensions. Feature values are binary. Dimensionality is very high. (b) Dense, embeddings-based feature vector. Each core feature is represented as a vector. Each feature corresponds to several input vector entries. No explicit encoding of feature combinations. Dimensionality is low. The feature-to-vector mappings come from an embedding table.

- Features are completely independent from one another. The feature “word is ‘dog’ ” is as dis-similar to “word is ‘thinking’ ” than it is to “word is ‘cat’ ”.

Dense Each feature is a d -dimensional vector.

- Dimensionality of vector is d .
- Similar features will have similar vectors – information is shared between similar features.

One benefit of using dense and low-dimensional vectors is computational: the majority of neural network toolkits do not play well with very high-dimensional, sparse vectors. However, this is just a technical obstacle, which can be resolved with some engineering effort.

The main benefit of the dense representations is in generalization power: if we believe some features may provide similar clues, it is worthwhile to provide a representation that is able to capture these similarities. For example, assume we have observed the word ‘dog’ many times during training, but only observed the word ‘cat’ a handful of times, or not at

all. If each of the words is associated with its own dimension, occurrences of ‘dog’ will not tell us anything about the occurrences of ‘cat’. However, in the dense vectors representation the learned vector for ‘dog’ may be similar to the learned vector from ‘cat’, allowing the model to share statistical strength between the two events. This argument assumes that “good” vectors are somehow given to us. Section 5 describes ways of obtaining such vector representations.

In cases where we have relatively few distinct features in the category, and we believe there are no correlations between the different features, we may use the one-hot representation. However, if we believe there are going to be correlations between the different features in the group (for example, for part-of-speech tags, we may believe that the different verb inflections VB and VBZ may behave similarly as far as our task is concerned) it may be worthwhile to let the network figure out the correlations and gain some statistical strength by sharing the parameters. It may be the case that under some circumstances, when the feature space is relatively small and the training data is plentiful, or when we do not wish to share statistical information between distinct words, there are gains to be made from using the one-hot representations. However, this is still an open research question, and there are no strong evidence to either side. The majority of work (pioneered by (Collobert & Weston, 2008; Collobert et al., 2011; Chen & Manning, 2014)) advocate the use of dense, trainable embedding vectors for all features. For work using neural network architecture with sparse vector encodings see (Johnson & Zhang, 2015).

Finally, it is important to note that representing features as dense vectors is an integral part of the neural network framework, and that consequentially the differences between using sparse and dense feature representations are subtler than they may appear at first. In fact, using sparse, one-hot vectors as input when training a neural network amounts to dedicating the first layer of the network to learning a dense embedding vector for each feature based on the training data. We touch on this in Section 4.4.

Variable Number of Features: Continuous Bag of Words Feed-forward networks assume a fixed dimensional input. This can easily accommodate the case of a feature-extraction function that extracts a fixed number of features: each feature is represented as a vector, and the vectors are concatenated. This way, each region of the resulting input vector corresponds to a different feature. However, in some cases the number of features is not known in advance (for example, in document classification it is common that each word in the sentence is a feature). We thus need to represent an unbounded number of features using a fixed size vector. One way of achieving this is through a so-called *continuous bag of words* (CBOW) representation (Mikolov, Chen, Corrado, & Dean, 2013). The CBOW is very similar to the traditional bag-of-words representation in which we discard order information, and works by either summing or averaging the embedding vectors of the corresponding features:²

2. Note that if the $v(f_i)$ s were one-hot vectors rather than dense feature representations, the *CBOW* and *WCBO* equations above would reduce to the traditional (weighted) bag-of-words representations, which is in turn equivalent to a sparse feature-vector representation in which each binary indicator feature corresponds to a unique “word”.

$$CBOW(f_1, \dots, f_k) = \frac{1}{k} \sum_{i=1}^k v(f_i)$$

A simple variation on the CBOW representation is weighted CBOW, in which different vectors receive different weights:

$$WCBOw(f_1, \dots, f_k) = \frac{1}{\sum_{i=1}^k a_i} \sum_{i=1}^k a_i v(f_i)$$

Here, each feature f_i has an associated weight a_i , indicating the relative importance of the feature. For example, in a document classification task, a feature f_i may correspond to a word in the document, and the associated weight a_i could be the word’s TF-IDF score.

Distance and Position Features The linear distance in between two words in a sentence may serve as an informative feature. For example, in an event extraction task³ we may be given a trigger word and a candidate argument word, and asked to predict if the argument word is indeed an argument of the trigger. The distance (or relative position) between the trigger and the argument is a strong signal for this prediction task. In the “traditional” NLP setup, distances are usually encoded by binning the distances into several groups (i.e. 1, 2, 3, 4, 5–10, 10+) and associating each bin with a one-hot vector. In a neural architecture, where the input vector is not composed of binary indicator features, it may seem natural to allocate a single input vector entry to the distance feature, where the numeric value of that entry is the distance. However, this approach is not taken in practice. Instead, distance features are encoded similarly to the other feature types: each bin is associated with a d -dimensional vector, and these distance-embedding vectors are then trained as regular parameters in the network (Zeng et al., 2014; dos Santos et al., 2015; Zhu et al., 2015a; Nguyen & Grishman, 2015).

Feature Combinations Note that the feature extraction stage in the neural-network settings deals only with extraction of *core* features. This is in contrast to the traditional linear-model-based NLP systems in which the feature designer had to manually specify not only the core features of interests but also interactions between them (e.g., introducing not only a feature stating “word is X” and a feature stating “tag is Y” but also combined feature stating “word is X and tag is Y” or sometimes even “word is X, tag is Y and previous word is Z”). The combination features are crucial in linear models because they introduce more dimensions to the input, transforming it into a space where the data-points are closer to being linearly separable. On the other hand, the space of possible combinations is very large, and the feature designer has to spend a lot of time coming up with an effective set of feature combinations. One of the promises of the non-linear neural network models is that one needs to define only the core features. The non-linearity of the classifier, as defined by the network structure, is expected to take care of finding the indicative feature combinations, alleviating the need for feature combination engineering.

3. The event extraction task involves identification of events from a predefined set of event types. For example identification of “purchase” events or “terror-attack” events. Each event type can be triggered by various triggering words (commonly verbs), and has several slots (arguments) that needs to be filled (i.e. who purchased? what was purchased? at what amount?).

Kernel methods (Shawe-Taylor & Cristianini, 2004), and in particular polynomial kernels (Kudo & Matsumoto, 2003), also allow the feature designer to specify only core features, leaving the feature combination aspect to the learning algorithm. In contrast to neural-network models, kernels methods are convex, admitting exact solutions to the optimization problem. However, the classification efficiency in kernel methods scales linearly with the size of the training data, making them too slow for most practical purposes, and not suitable for training with large datasets. On the other hand, neural network classification efficiency scales linearly with the size of the network, regardless of the training data size.

Dimensionality How many dimensions should we allocate for each feature? Unfortunately, there are no theoretical bounds or even established best-practices in this space. Clearly, the dimensionality should grow with the number of the members in the class (you probably want to assign more dimensions to word embeddings than to part-of-speech embeddings) but how much is enough? In current research, the dimensionality of word-embedding vectors range between about 50 to a few hundreds, and, in some extreme cases, thousands. Since the dimensionality of the vectors has a direct effect on memory requirements and processing time, a good rule of thumb would be to experiment with a few different sizes, and choose a good trade-off between speed and task accuracy.

Vector Sharing Consider a case where you have a few features that share the same vocabulary. For example, when assigning a part-of-speech to a given word, we may have a set of features considering the previous word, and a set of features considering the next word. When building the input to the classifier, we will concatenate the vector representation of the previous word to the vector representation of the next word. The classifier will then be able to distinguish the two different indicators, and treat them differently. But should the two features share the same vectors? Should the vector for “dog:previous-word” be the same as the vector of “dog:next-word”? Or should we assign them two distinct vectors? This, again, is mostly an empirical question. If you believe words behave differently when they appear in different positions (e.g., word X behaves like word Y when in the previous position, but X behaves like Z when in the next position) then it may be a good idea to use two different vocabularies and assign a different set of vectors for each feature type. However, if you believe the words behave similarly in both locations, then something may be gained by using a shared vocabulary for both feature types.

Network’s Output For multi-class classification problems with k classes, the network’s output is a k -dimensional vector in which every dimension represents the strength of a particular output class. That is, the output remains as in the traditional linear models – scalar scores to items in a discrete set. However, as we will see in Section 4, there is a $d \times k$ matrix associated with the output layer. The columns of this matrix can be thought of as d dimensional embeddings of the output classes. The vector similarities between the vector representations of the k classes indicate the model’s learned similarities between the output classes.

Historical Note Representing words as dense vectors for input to a neural network was introduced by Bengio et al (Bengio et al., 2003) in the context of neural language modeling. It was introduced to NLP tasks in the pioneering work of Collobert, Weston and colleagues

(2008, 2011). Using embeddings for representing not only words but arbitrary features was popularized following Chen and Manning (2014).

4. Feed-forward Neural Networks

A Brain-inspired metaphor As the name suggest, neural-networks are inspired by the brain’s computation mechanism, which consists of computation units called neurons. In the metaphor, a neuron is a computational unit that has scalar inputs and outputs. Each input has an associated weight. The neuron multiplies each input by its weight, and then sums⁴ them, applies a non-linear function to the result, and passes it to its output. The neurons are connected to each other, forming a network: the output of a neuron may feed into the inputs of one or more neurons. Such networks were shown to be very capable computational devices. If the weights are set correctly, a neural network with enough neurons and a non-linear activation function can approximate a very wide range of mathematical functions (we will be more precise about this later).

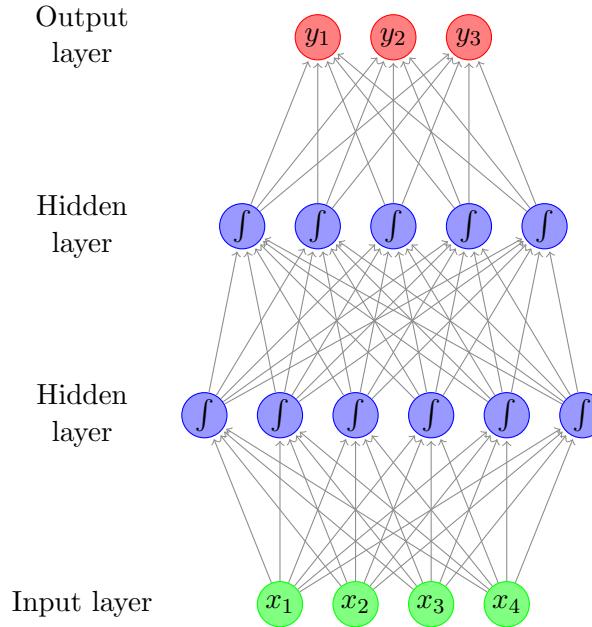


Figure 2: Feed-forward neural network with two hidden layers.

A typical feed-forward neural network may be drawn as in Figure 2. Each circle is a neuron, with incoming arrows being the neuron’s inputs and outgoing arrows being the neuron’s outputs. Each arrow carries a weight, reflecting its importance (not shown). Neurons are arranged in layers, reflecting the flow of information. The bottom layer has no incoming arrows, and is the input to the network. The top-most layer has no outgoing arrows, and is the output of the network. The other layers are considered “hidden”. The sigmoid shape inside the neurons in the middle layers represent a non-linear function (typically a $1/(1 + e^{-x})$) that is applied to the neuron’s value before passing it to the output. In the figure, each neuron is connected to all of the neurons in the next layer – this is called a *fully-connected layer* or an *affine layer*.

4. While summing is the most common operation, other functions, such as a max, are also possible

While the brain metaphor is sexy and intriguing, it is also distracting and cumbersome to manipulate mathematically. We therefore switch to using more concise mathematic notation. The values of each row of neurons in the network can be thought of as a vector. In Figure 2 the input layer is a 4 dimensional vector (\mathbf{x}), and the layer above it is a 6 dimensional vector (\mathbf{h}^1). The fully connected layer can be thought of as a linear transformation from 4 dimensions to 6 dimensions. A fully-connected layer implements a vector-matrix multiplication, $\mathbf{h} = \mathbf{x}\mathbf{W}$ where the weight of the connection from the i th neuron in the input row to the j th neuron in the output row is W_{ij} .⁵ The values of \mathbf{h} are then transformed by a non-linear function g that is applied to each value before being passed on to the next input. The whole computation from input to output can be written as: $(g(\mathbf{x}\mathbf{W}^1))\mathbf{W}^2$ where \mathbf{W}^1 are the weights of the first layer and \mathbf{W}^2 are the weights of the second one.

In Mathematical Notation From this point on, we will abandon the brain metaphor and describe networks exclusively in terms of vector-matrix operations.

The simplest neural network is the perceptron, which is a linear function of its inputs:

$$NN_{Perceptron}(\mathbf{x}) = \mathbf{x}\mathbf{W} + \mathbf{b}$$

$$\mathbf{x} \in \mathbb{R}^{d_{in}}, \quad \mathbf{W} \in \mathbb{R}^{d_{in} \times d_{out}}, \quad \mathbf{b} \in \mathbb{R}^{d_{out}}$$

\mathbf{W} is the weight matrix, and \mathbf{b} is a bias term.⁶ In order to go beyond linear functions, we introduce a non-linear hidden layer (the network in Figure 2 has two such layers), resulting in the 1-layer Multi Layer Perceptron (MLP1). A one-layer feed-forward neural network has the form:

$$NN_{MLP1}(\mathbf{x}) = g(\mathbf{x}\mathbf{W}^1 + \mathbf{b}^1)\mathbf{W}^2 + \mathbf{b}^2$$

$$\mathbf{x} \in \mathbb{R}^{d_{in}}, \quad \mathbf{W}^1 \in \mathbb{R}^{d_{in} \times d_1}, \quad \mathbf{b}^1 \in \mathbb{R}^{d_1}, \quad \mathbf{W}^2 \in \mathbb{R}^{d_1 \times d_2}, \quad \mathbf{b}^2 \in \mathbb{R}^{d_2}$$

Here \mathbf{W}^1 and \mathbf{b}^1 are a matrix and a bias term for the first linear transformation of the input, g is a non-linear function that is applied element-wise (also called a *non-linearity* or an *activation function*), and \mathbf{W}^2 and \mathbf{b}^2 are the matrix and bias term for a second linear transform.

Breaking it down, $\mathbf{x}\mathbf{W}^1 + \mathbf{b}^1$ is a linear transformation of the input \mathbf{x} from d_{in} dimensions to d_1 dimensions. g is then applied to each of the d_1 dimensions, and the matrix \mathbf{W}^2 together with bias vector \mathbf{b}^2 are then used to transform the result into the d_2 dimensional output vector. The non-linear activation function g has a crucial role in the network's ability to represent complex functions. Without the non-linearity in g , the neural network can only represent linear transformations of the input.⁷

We can add additional linear-transformations and non-linearities, resulting in a 2-layer MLP (the network in Figure 2 is of this form):

$$NN_{MLP2}(\mathbf{x}) = (g^2(g^1(\mathbf{x}\mathbf{W}^1 + \mathbf{b}^1)\mathbf{W}^2 + \mathbf{b}^2))\mathbf{W}^3$$

-
- 5. To see why this is the case, denote the weight of the i th input of the j th neuron in \mathbf{h} as w_{ij} . The value of h_j is then $h_j = \sum_{i=1}^4 x_i \cdot w_{ij}$.
 - 6. The network in figure 2 does not include bias terms. A bias term can be added to a layer by adding to it an additional neuron that does not have any incoming connections, whose value is always 1.
 - 7. To see why, consider that a sequence of linear transformations is still a linear transformation.

It is perhaps clearer to write deeper networks like this using intermediary variables:

$$\begin{aligned} NN_{MLP2}(\mathbf{x}) &= \mathbf{y} \\ \mathbf{h}^1 &= g^1(\mathbf{x}\mathbf{W}^1 + \mathbf{b}^1) \\ \mathbf{h}^2 &= g^2(\mathbf{h}^1\mathbf{W}^2 + \mathbf{b}^2) \\ \mathbf{y} &= \mathbf{h}^2\mathbf{W}^3 \end{aligned}$$

The vector resulting from each linear transform is referred to as a *layer*. The outer-most linear transform results in the *output layer* and the other linear transforms result in *hidden layers*. Each hidden layer is followed by a non-linear activation. In some cases, such as in the last layer of our example, the bias vectors are forced to 0 (“dropped”).

Layers resulting from linear transformations are often referred to as *fully connected*, or *affine*. Other types of architectures exist. In particular, image recognition problems benefit from *convolutional* and *pooling* layers. Such layers have uses also in language processing, and will be discussed in Section 9. Networks with more than one hidden layer are said to be *deep* networks, hence the name *deep learning*.

When describing a neural network, one should specify the *dimensions* of the layers and the input. A layer will expect a d_{in} dimensional vector as its input, and transform it into a d_{out} dimensional vector. The dimensionality of the layer is taken to be the dimensionality of its output. For a fully connected layer $l(\mathbf{x}) = \mathbf{x}\mathbf{W} + \mathbf{b}$ with input dimensionality d_{in} and output dimensionality d_{out} , the dimensions of \mathbf{x} is $1 \times d_{in}$, of \mathbf{W} is $d_{in} \times d_{out}$ and of \mathbf{b} is $1 \times d_{out}$.

The output of the network is a d_{out} dimensional vector. In case $d_{out} = 1$, the network’s output is a scalar. Such networks can be used for regression (or scoring) by considering the value of the output, or for binary classification by consulting the sign of the output. Networks with $d_{out} = k > 1$ can be used for k -class classification, by associating each dimension with a class, and looking for the dimension with maximal value. Similarly, if the output vector entries are positive and sum to one, the output can be interpreted as a distribution over class assignments (such output normalization is typically achieved by applying a softmax transformation on the output layer, see Section 4.3).

The matrices and the bias terms that define the linear transformations are the *parameters* of the network. It is common to refer to the collection of all parameters as θ . Together with the input, the parameters determine the network’s output. The training algorithm is responsible for setting their values such that the network’s predictions are correct. Training is discussed in Section 6.

4.1 Representation Power

In terms of representation power, it was shown by (Hornik, Stinchcombe, & White, 1989; Cybenko, 1989) that MLP1 is a universal approximator – it can approximate with any desired non-zero amount of error a family of functions⁸ that include all continuous functions

8. Specifically, a feed-forward network with linear output layer and at least one hidden layer with a “squashing” activation function can approximate any Borel measurable function from one finite dimensional space to another.

on a closed and bounded subset of \mathbb{R}^n , and any function mapping from any finite dimensional discrete space to another. This may suggest there is no reason to go beyond MLP1 to more complex architectures. However, the theoretical result does not state how large the hidden layer should be, nor does it say anything about the learnability of the neural network (it states that a representation exists, but does not say how easy or hard it is to set the parameters based on training data and a specific learning algorithm). It also does not guarantee that a training algorithm will find the *correct* function generating our training data. Since in practice we train neural networks on relatively small amounts of data, using a combination of the backpropagation algorithm and variants of stochastic gradient descent, and use hidden layers of relatively modest sizes (up to several thousands), there is benefit to be had in trying out more complex architectures than MLP1. In many cases, however, MLP1 does indeed provide very strong results. For further discussion on the representation power of feed-forward neural networks, see (Bengio et al., 2015, Section 6.5).

4.2 Common Non-linearities

The non-linearity g can take many forms. There is currently no good theory as to which non-linearity to apply in which conditions, and choosing the correct non-linearity for a given task is for the most part an empirical question. I will now go over the common non-linearities from the literature: the sigmoid, tanh, hard tanh and the rectified linear unit (ReLU). Some NLP researchers also experimented with other forms of non-linearities such as cube and tanh-cube.

Sigmoid The sigmoid activation function $\sigma(x) = 1/(1 + e^{-x})$ is an S-shaped function, transforming each value x into the range $[0, 1]$.

Hyperbolic tangent (tanh) The hyperbolic tangent $\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$ activation function is an S-shaped function, transforming the values x into the range $[-1, 1]$.

Hard tanh The hard-tanh activation function is an approximation of the *tanh* function which is faster to compute and take derivatives of:

$$\text{hardtanh}(x) = \begin{cases} -1 & x < -1 \\ 1 & x > 1 \\ x & \text{otherwise} \end{cases}$$

Rectifier (ReLU) The Rectifier activation function (Glorot, Bordes, & Bengio, 2011), also known as the rectified linear unit is a very simple activation function that is easy to work with and was shown many times to produce excellent results.⁹ The ReLU unit clips each value $x < 0$ at 0. Despite its simplicity, it performs well for many tasks, especially when combined with the dropout regularization technique (see Section 6.4).

9. The technical advantages of the ReLU over the sigmoid and tanh activation functions is that it does not involve expensive-to-compute functions, and more importantly that it does not saturate. The sigmoid and tanh activation are capped at 1, and the gradients at this region of the functions are near zero, driving the entire gradient near zero. The ReLU activation does not have this problem, making it especially suitable for networks with multiple layers, which are susceptible to the vanishing gradients problem when trained with the saturating units.

$$ReLU(x) = \max(0, x) = \begin{cases} 0 & x < 0 \\ x & \text{otherwise} \end{cases}$$

As a rule of thumb, ReLU units work better than tanh, and tanh works better than sigmoid.¹⁰

4.3 Output Transformations

In many cases, the output layer vector is also transformed. A common transformation is the *softmax*:

$$\mathbf{x} = x_1, \dots, x_k$$

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^k e^{x_j}}$$

The result is a vector of non-negative real numbers that sum to one, making it a discrete probability distribution over k possible outcomes.

The *softmax* output transformation is used when we are interested in modeling a probability distribution over the possible output classes. To be effective, it should be used in conjunction with a probabilistic training objective such as cross-entropy (see Section 4.5 below).

When the softmax transformation is applied to the output of a network without a hidden layer, the result is the well known multinomial logistic regression model, also known as a maximum-entropy classifier.

4.4 Embedding Layers

Up until now, the discussion ignored the source of \mathbf{x} , treating it as an arbitrary vector. In an NLP application, \mathbf{x} is usually composed of various embeddings vectors. We can be explicit about the source of \mathbf{x} , and include it in the network's definition. We introduce $c(\cdot)$, a function from core features to an input vector.

It is common for c to extract the embedding vector associated with each feature, and concatenate them:

10. In addition to these activation functions, recent works from the NLP community experiment with and reported success with other forms of non-linearities. The **Cube** activation function, $g(x) = (x)^3$, was suggested by (Chen & Manning, 2014), who found it to be more effective than other non-linearities in a feed-forward network that was used to predict the actions in a greedy transition-based dependency parser. The **tanh cube** activation function $g(x) = \tanh((x)^3 + x)$ was proposed by (Pei et al., 2015), who found it to be more effective than other non-linearities in a feed-forward network that was used as a component in a structured-prediction graph-based dependency parser.

The cube and tanh-cube activation functions are motivated by the desire to better capture interactions between different features. While these activation functions are reported to improve performance in certain situations, their general applicability is still to be determined.

$$\begin{aligned}
\mathbf{x} &= c(f_1, f_2, f_3) = [v(f_1); v(f_2); v(f_3)] \\
NN_{MLP1}(\mathbf{x}) &= NN_{MLP1}(c(f_1, f_2, f_3)) \\
&= NN_{MLP1}([v(f_1); v(f_2); v(f_3)]) \\
&= (g([v(f_1); v(f_2); v(f_3)] \mathbf{W}^1 + \mathbf{b}^1)) \mathbf{W}^2 + \mathbf{b}^2
\end{aligned}$$

Another common choice is for c to sum the embedding vectors (this assumes the embedding vectors all share the same dimensionality):

$$\begin{aligned}
\mathbf{x} &= c(f_1, f_2, f_3) = v(f_1) + v(f_2) + v(f_3) \\
NN_{MLP1}(\mathbf{x}) &= NN_{MLP1}(c(f_1, f_2, f_3)) \\
&= NN_{MLP1}(v(f_1) + v(f_2) + v(f_3)) \\
&= (g((v(f_1) + v(f_2) + v(f_3)) \mathbf{W}^1 + \mathbf{b}^1)) \mathbf{W}^2 + \mathbf{b}^2
\end{aligned}$$

The form of c is an essential part of the network’s design. In many papers, it is common to refer to c as part of the network, and likewise treat the word embeddings $v(f_i)$ as resulting from an “embedding layer” or “lookup layer”. Consider a vocabulary of $|V|$ words, each embedded as a d dimensional vector. The collection of vectors can then be thought of as a $|V| \times d$ embedding matrix \mathbf{E} in which each row corresponds to an embedded feature. Let \mathbf{f}_i be a $|V|$ -dimensional vector, which is all zeros except from one index, corresponding to the value of the i th feature, in which the value is 1 (this is called a one-hot vector). The multiplication $\mathbf{f}_i \mathbf{E}$ will then “select” the corresponding row of \mathbf{E} . Thus, $v(f_i)$ can be defined in terms of \mathbf{E} and \mathbf{f}_i :

$$v(f_i) = \mathbf{f}_i \mathbf{E}$$

And similarly:

$$CBOW(f_1, \dots, f_k) = \sum_{i=1}^k (\mathbf{f}_i \mathbf{E}) = \left(\sum_{i=1}^k \mathbf{f}_i \right) \mathbf{E}$$

The input to the network is then considered to be a collection of one-hot vectors. While this is elegant and well defined mathematically, an efficient implementation typically involves a hash-based data structure mapping features to their corresponding embedding vectors, without going through the one-hot representation.

In this tutorial, we take c to be separate from the network architecture: the network’s inputs are always dense real-valued input vectors, and c is applied before the input is passed the network, similar to a “feature function” in the familiar linear-models terminology. However, when training a network, the input vector \mathbf{x} does remember how it was constructed, and can propagate error gradients back to its component embedding vectors, as appropriate.

A note on notation When describing network layers that get concatenated vectors \mathbf{x} , \mathbf{y} and \mathbf{z} as input, some authors use explicit concatenation ($[\mathbf{x}; \mathbf{y}; \mathbf{z}] \mathbf{W} + \mathbf{b}$) while others use an affine transformation ($\mathbf{x}\mathbf{U} + \mathbf{y}\mathbf{V} + \mathbf{z}\mathbf{W} + \mathbf{b}$). If the weight matrices \mathbf{U} , \mathbf{V} , \mathbf{W} in the affine transformation are different than one another, the two notations are equivalent.

A note on sparse vs. dense features Consider a network which uses a “traditional” sparse representation for its input vectors, and no embedding layer. Assuming the set of all available features is V and we have k “on” features $f_1, \dots, f_k, f_i \in V$, the network’s input is:

$$\mathbf{x} = \sum_{i=1}^k \mathbf{f}_i \quad \mathbf{x} \in \mathbb{N}_+^{|V|}$$

and so the first layer (ignoring the non-linear activation) is:

$$\begin{aligned} \mathbf{x}\mathbf{W} + \mathbf{b} &= \left(\sum_{i=1}^k \mathbf{f}_i \right) \mathbf{W} \\ \mathbf{W} &\in \mathbb{R}^{|V| \times d}, \quad \mathbf{b} \in \mathbb{R}^d \end{aligned}$$

This layer selects rows of \mathbf{W} corresponding to the input features in \mathbf{x} and sums them, then adding a bias term. This is very similar to an embedding layer that produces a CBOW representation over the features, where the matrix \mathbf{W} acts as the embedding matrix. The main difference is the introduction of the bias vector \mathbf{b} , and the fact that the embedding layer typically does not undergo a non-linear activation but rather passed on directly to the first layer. Another difference is that this scenario forces each feature to receive a separate vector (row in \mathbf{W}) while the embedding layer provides more flexibility, allowing for example for the features “next word is dog” and “previous word is dog” to share the same vector. However, these differences are small and subtle. When it comes to multi-layer feed-forward networks, the difference between dense and sparse inputs is smaller than it may seem at first sight.

4.5 Loss Functions

When training a neural network (more on training in Section 6 below), much like when training a linear classifier, one defines a loss function $L(\hat{\mathbf{y}}, \mathbf{y})$, stating the loss of predicting $\hat{\mathbf{y}}$ when the true output is \mathbf{y} . The training objective is then to minimize the loss across the different training examples. The loss $L(\hat{\mathbf{y}}, \mathbf{y})$ assigns a numerical score (a scalar) for the network’s output $\hat{\mathbf{y}}$ given the true expected output \mathbf{y} .¹¹ The loss is always positive, and should be zero only for cases where the network’s output is correct.

The parameters of the network (the matrices \mathbf{W}^i , the biases \mathbf{b}^i and commonly the embeddings \mathbf{E}) are then set in order to minimize the loss L over the training examples (usually, it is the sum of the losses over the different training examples that is being minimized).

The loss can be an arbitrary function mapping two vectors to a scalar. For practical purposes of optimization, we restrict ourselves to functions for which we can easily compute gradients (or sub-gradients). In most cases, it is sufficient and advisable to rely on a common loss function rather than defining your own. For a detailed discussion on loss functions for neural networks see (LeCun, Chopra, Hadsell, Ranzato, & Huang, 2006; LeCun & Huang, 2005; Bengio et al., 2015). We now discuss some loss functions that are commonly used in neural networks for NLP.

11. In our notation, both the model’s output and the expected output are vectors, while in many cases it is more natural to think of the expected output as a scalar (class assignment). In such cases, \mathbf{y} is simply the corresponding one-hot vector.

Hinge (binary) For binary classification problems, the network’s output is a single scalar \hat{y} and the intended output y is in $\{+1, -1\}$. The classification rule is $sign(\hat{y})$, and a classification is considered correct if $y \cdot \hat{y} > 0$, meaning that y and \hat{y} share the same sign. The hinge loss, also known as margin loss or SVM loss, is defined as:

$$L_{hinge(binary)}(\hat{y}, y) = \max(0, 1 - y \cdot \hat{y})$$

The loss is 0 when y and \hat{y} share the same sign and $|\hat{y}| \geq 1$. Otherwise, the loss is linear. In other words, the binary hinge loss attempts to achieve a correct classification, with a margin of at least 1.

Hinge (multiclass) The hinge loss was extended to the multiclass setting by Crammer and Singer (2002). Let $\hat{\mathbf{y}} = \hat{y}_1, \dots, \hat{y}_n$ be the network’s output vector, and \mathbf{y} be the one-hot vector for the correct output class.

The classification rule is defined as selecting the class with the highest score:

$$\text{prediction} = \arg \max_i \hat{y}_i$$

Denote by $t = \arg \max_i y_i$ the correct class, and by $k = \arg \max_{i \neq t} \hat{y}_i$ the highest scoring class such that $k \neq t$. The multiclass hinge loss is defined as:

$$L_{hinge(multiclass)}(\hat{\mathbf{y}}, \mathbf{y}) = \max(0, 1 - (\hat{y}_t - \hat{y}_k))$$

The multiclass hinge loss attempts to score the correct class above all other classes with a margin of at least 1.

Both the binary and multiclass hinge losses are intended to be used with a linear output layer. The hinge losses are useful whenever we require a hard decision rule, and do not attempt to model class membership probability.

Log loss The log loss is a common variation of the hinge loss, which can be seen as a “soft” version of the hinge loss with an infinite margin (LeCun et al., 2006).

$$L_{log}(\hat{\mathbf{y}}, \mathbf{y}) = \log(1 + \exp(-(\hat{y}_t - \hat{y}_k)))$$

Categorical cross-entropy loss The categorical cross-entropy loss (also referred to as *negative log likelihood*) is used when a probabilistic interpretation of the scores is desired.

Let $\mathbf{y} = y_1, \dots, y_n$ be a vector representing the true multinomial distribution over the labels $1, \dots, n$, and let $\hat{\mathbf{y}} = \hat{y}_1, \dots, \hat{y}_n$ be the network’s output, which was transformed by the *softmax* activation function, and represent the class membership conditional distribution $\hat{y}_i = P(y = i | \mathbf{x})$. The categorical cross entropy loss measures the dissimilarity between the true label distribution \mathbf{y} and the predicted label distribution $\hat{\mathbf{y}}$, and is defined as cross entropy:

$$L_{cross-entropy}(\hat{\mathbf{y}}, \mathbf{y}) = - \sum_i y_i \log(\hat{y}_i)$$

For hard classification problems in which each training example has a single correct class assignment, \mathbf{y} is a one-hot vector representing the true class. In such cases, the cross entropy can be simplified to:

$$L_{\text{cross-entropy}(\text{hard classification})}(\hat{\mathbf{y}}, \mathbf{y}) = -\log(\hat{y}_t)$$

where t is the correct class assignment. This attempts to set the probability mass assigned to the correct class t to 1. Because the scores $\hat{\mathbf{y}}$ have been transformed using the *softmax* function and represent a conditional distribution, increasing the mass assigned to the correct class means decreasing the mass assigned to all the other classes.

The cross-entropy loss is very common in the neural networks literature, and produces a multi-class classifier which does not only predict the one-best class label but also predicts a distribution over the possible labels. When using the cross-entropy loss, it is assumed that the network's output is transformed using the *softmax* transformation.

Ranking losses In some settings, we are not given supervision in term of labels, but rather as pairs of correct and incorrect items \mathbf{x} and \mathbf{x}' , and our goal is to score correct items above incorrect ones. Such training situations arise when we have only positive examples, and generate negative examples by corrupting a positive example. A useful loss in such scenarios is the margin-based ranking loss, defined for a pair of correct and incorrect examples:

$$L_{\text{ranking}(\text{margin})}(\mathbf{x}, \mathbf{x}') = \max(0, 1 - (NN(\mathbf{x}) - NN(\mathbf{x}')))$$

where $NN(\mathbf{x})$ is the score assigned by the network for input vector \mathbf{x} . The objective is to score (rank) correct inputs over incorrect ones with a margin of at least 1.

A common variation is to use the log version of the ranking loss:

$$L_{\text{ranking}(\text{log})}(\mathbf{x}, \mathbf{x}') = \log(1 + \exp(-(NN(\mathbf{x}) - NN(\mathbf{x}'))))$$

Examples using the ranking hinge loss in language tasks include training with the auxiliary tasks used for deriving pre-trained word embeddings (see section 5), in which we are given a correct word sequence and a corrupted word sequence, and our goal is to score the correct sequence above the corrupt one (Collobert & Weston, 2008). Similarly, Van de Cruys (2014) used the ranking loss in a selectional-preferences task, in which the network was trained to rank correct verb-object pairs above incorrect, automatically derived ones, and (Weston, Bordes, Yakhnenko, & Usunier, 2013) trained a model to score correct (head,relation,trail) triplets above corrupted ones in an information-extraction setting. An example of using the ranking log loss can be found in (Gao et al., 2014). A variation of the ranking log loss allowing for a different margin for the negative and positive class is given in (dos Santos et al., 2015).

5. Word Embeddings

A main component of the neural-network approach is the use of embeddings – representing each feature as a vector in a low dimensional space. But where do the vectors come from? This section will survey the common approaches.

5.1 Random Initialization

When enough supervised training data is available, one can just treat the feature embeddings the same as the other model parameters: initialize the embedding vectors to random values, and let the network-training procedure tune them into “good” vectors.

Some care has to be taken in the way the random initialization is performed. The method used by the effective word2vec implementation (Mikolov et al., 2013; Mikolov, Sutskever, Chen, Corrado, & Dean, 2013) is to initialize the word vectors to uniformly sampled random numbers in the range $[-\frac{1}{2d}, \frac{1}{2d}]$ where d is the number of dimensions. Another option is to use *xavier initialization* (see Section 6.3) and initialize with uniformly sampled values from $[-\frac{\sqrt{6}}{\sqrt{d}}, \frac{\sqrt{6}}{\sqrt{d}}]$.

In practice, one will often use the random initialization approach to initialize the embedding vectors of commonly occurring features, such as part-of-speech tags or individual letters, while using some form of supervised or unsupervised pre-training to initialize the potentially rare features, such as features for individual words. The pre-trained vectors can then either be treated as fixed during the network training process, or, more commonly, treated like the randomly-initialized vectors and further tuned to the task at hand.

5.2 Supervised Task-specific Pre-training

If we are interested in task A, for which we only have a limited amount of labeled data (for example, syntactic parsing), but there is an auxiliary task B (say, part-of-speech tagging) for which we have much more labeled data, we may want to pre-train our word vectors so that they perform well as predictors for task B, and then use the trained vectors for training task A. In this way, we can utilize the larger amounts of labeled data we have for task B. When training task A we can either treat the pre-trained vectors as fixed, or tune them further for task A. Another option is to train jointly for both objectives, see Section 7 for more details.

5.3 Unsupervised Pre-training

The common case is that we do not have an auxiliary task with large enough amounts of annotated data (or maybe we want to help bootstrap the auxiliary task training with better vectors). In such cases, we resort to “unsupervised” methods, which can be trained on huge amounts of unannotated text.

The techniques for training the word vectors are essentially those of supervised learning, but instead of supervision for the task that we care about, we instead create practically

unlimited number of supervised training instances from raw text, hoping that the tasks that we created will match (or be close enough to) the final task we care about.¹²

The key idea behind the unsupervised approaches is that one would like the embedding vectors of “similar” words to have similar vectors. While word similarity is hard to define and is usually very task-dependent, the current approaches derive from the distributional hypothesis (Harris, 1954), stating that *words are similar if they appear in similar contexts*. The different methods all create supervised training instances in which the goal is to either predict the word from its context, or predict the context from the word.

An important benefit of training word embeddings on large amounts of unannotated data is that it provides vector representations for words that do not appear in the supervised training set. Ideally, the representations for these words will be similar to those of related words that do appear in the training set, allowing the model to generalize better on unseen events. It is thus desired that the similarity between word vectors learned by the unsupervised algorithm captures the same aspects of similarity that are useful for performing the intended task of the network.

Common unsupervised word-embedding algorithms include word2vec¹³ (Mikolov et al., 2013, 2013), GloVe (Pennington, Socher, & Manning, 2014) and the Collobert and Weston (2008, 2011) embeddings algorithm. These models are inspired by neural networks and are based on stochastic gradient training. However, they are deeply connected to another family of algorithms which evolved in the NLP and IR communities, and that are based on matrix factorization (see (Levy & Goldberg, 2014b; Levy et al., 2015) for a discussion).

Arguably, the choice of auxiliary problem (what is being predicted, based on what kind of context) affects the resulting vectors much more than the learning method that is being used to train them. We thus focus on the different choices of auxiliary problems that are available, and only skim over the details of the training methods. Several software packages for deriving word vectors are available, including word2vec¹⁴ and Gensim¹⁵ implementing the word2vec models with word-windows based contexts, word2vecf¹⁶ which is a modified version of word2vec allowing the use of arbitrary contexts, and GloVe¹⁷ implementing the GloVe model. Many pre-trained word vectors are also available for download on the web.

While beyond the scope of this tutorial, it is worth noting that the word embeddings derived by unsupervised training algorithms have a wide range of applications in NLP beyond using them for initializing the word-embeddings layer of a neural-network model.

5.4 Training Objectives

Given a word w and its context c , different algorithms formulate different auxiliary tasks. In all cases, each word is represented as a d -dimensional vector which is initialized to a random value. Training the model to perform the auxiliary tasks well will result in good

-
- 12. The interpretation of creating auxiliary problems from raw text is inspired by Ando and Zhang (Ando & Zhang, 2005a, 2005b).
 - 13. While often treated as a single algorithm, word2vec is actually a software package including various training objectives, optimization methods and other hyperparameters. See (Rong, 2014; Levy, Goldberg, & Dagan, 2015) for a discussion.
 - 14. <https://code.google.com/p/word2vec/>
 - 15. <https://radimrehurek.com/gensim/>
 - 16. <https://bitbucket.org/yoavgo/word2vecf>
 - 17. <http://nlp.stanford.edu/projects/glove/>

word embeddings for relating the words to the contexts, which in turn will result in the embedding vectors for similar words to be similar to each other.

Language-modeling inspired approaches such as those taken by (Mikolov et al., 2013; Mnih & Kavukcuoglu, 2013) as well as GloVe (Pennington et al., 2014) use auxiliary tasks in which the goal is to predict the word given its context. This is posed in a probabilistic setup, trying to model the conditional probability $P(w|c)$.

Other approaches reduce the problem to that of binary classification. In addition to the set D of observed word-context pairs, a set \bar{D} is created from random words and context pairings. The binary classification problem is then: does the given (w, c) pair come from D or not? The approaches differ in how the set \bar{D} is constructed, what is the structure of the classifier, and what is the objective being optimized. Collobert and Weston (2008, 2011) take a margin-based binary ranking approach, training a feed-forward neural network to score correct (w, c) pairs over incorrect ones. Mikolov et al (2013, 2014) take instead a probabilistic version, training a log-bilinear model to predict the probability $P((w, c) \in D|w, c)$ that the pair come from the corpus rather than the random sample.

5.5 The Choice of Contexts

In most cases, the contexts of a word are taken to be other words that appear in its surrounding, either in a short window around it, or within the same sentence, paragraph or document. In some cases the text is automatically parsed by a syntactic parser, and the contexts are derived from the syntactic neighbourhood induced by the automatic parse trees. Sometimes, the definitions of words and context change to include also parts of words, such as prefixes or suffixes.

Neural word embeddings originated from the world of language modeling, in which a network is trained to predict the next word based on a sequence of preceding words (Bengio et al., 2003). There, the text is used to create auxiliary tasks in which the aim is to predict a word based on a context the k previous words. While training for the language modeling auxiliary prediction problems indeed produce useful embeddings, this approach is needlessly restricted by the constraints of the language modeling task, in which one is allowed to look only at the previous words. If we do not care about language modeling but only about the resulting embeddings, we may do better by ignoring this constraint and taking the context to be a symmetric window around the focus word.

5.5.1 WINDOW APPROACH

The most common approach is a sliding window approach, in which auxiliary tasks are created by looking at a sequence of $2k + 1$ words. The middle word is called the *focus word* and the k words to each side are the *contexts*. Then, either a single task is created in which the goal is to predict the focus word based on all of the context words (represented either using CBOW (Mikolov et al., 2013) or vector concatenation (Collobert & Weston, 2008)), or $2k$ distinct tasks are created, each pairing the focus word with a different context word. The $2k$ tasks approach, popularized by (Mikolov et al., 2013) is referred to as a *skip-gram* model. Skip-gram based approaches are shown to be robust and efficient to train (Mikolov et al., 2013; Pennington et al., 2014), and often produce state of the art results.

Effect of Window Size The size of the sliding window has a strong effect on the resulting vector similarities. Larger windows tend to produce more topical similarities (i.e. “dog”, “bark” and “leash” will be grouped together, as well as “walked”, “run” and “walking”), while smaller windows tend to produce more functional and syntactic similarities (i.e. “Poodle”, “Pitbull”, “Rottweiler”, or “walking”, “running”, “approaching”).

Positional Windows When using the CBOW or skip-gram context representations, all the different context words within the window are treated equally. There is no distinction between context words that are close to the focus words and those that are farther from it, and likewise there is no distinction between context words that appear before the focus words to context words that appear after it. Such information can easily be factored in by using *positional contexts*: indicating for each context word also its relative position to the focus words (i.e. instead of the context word being “the” it becomes “the:+2”, indicating the word appears two positions to the right of the focus word). The use of positional context together with smaller windows tend to produce similarities that are more syntactic, with a strong tendency of grouping together words that share a part of speech, as well as being functionally similar in terms of their semantics. Positional vectors were shown by (Ling, Dyer, Black, & Trancoso, 2015a) to be more effective than window-based vectors when used to initialize networks for part-of-speech tagging and syntactic dependency parsing.

Variants Many variants on the window approach are possible. One may lemmatize words before learning, apply text normalization, filter too short or too long sentences, or remove capitalization (see, e.g., the pre-processing steps described in (dos Santos & Gatti, 2014)). One may sub-sample part of the corpus, skipping with some probability the creation of tasks from windows that have too common or too rare focus words. The window size may be dynamic, using a different window size at each turn. One may weigh the different positions in the window differently, focusing more on trying to predict correctly close word-context pairs than further away ones. Each of these choices will effect the resulting vectors. Some of these hyperparameters (and others) are discussed in (Levy et al., 2015).

5.5.2 SENTENCES, PARAGRAPHS OR DOCUMENTS

Using a skip-grams (or CBOW) approach, one can consider the contexts of a word to be all the other words that appear with it in the same sentence, paragraph or document. This is equivalent to using very large window sizes, and is expected to result in word vectors that capture topical similarity (words from the same topic, i.e. words that one would expect to appear in the same document, are likely to receive similar vectors).

5.5.3 SYNTACTIC WINDOW

Some work replace the linear context within a sentence with a syntactic one (Levy & Goldberg, 2014a; Bansal, Gimpel, & Livescu, 2014). The text is automatically parsed using a dependency parser, and the context of a word is taken to be the words that are in its proximity in the parse tree, together with the syntactic relation by which they are connected. Such approaches produce highly *functional* similarities, grouping together words than can fill the same role in a sentence (e.g. colors, names of schools, verbs of movement).

The grouping is also syntactic, grouping together words that share an inflection (Levy & Goldberg, 2014a).

5.5.4 MULTILINGUAL

Another option is using multilingual, translation based contexts (Hermann & Blunsom, 2014; Faruqui & Dyer, 2014). For example, given a large amount of sentence-aligned parallel text, one can run a bilingual alignment model such as the IBM model 1 or model 2 (i.e. using the GIZA++ software), and then use the produced alignments to derive word contexts. Here, the context of a word instance are the foreign language words that are aligned to it. Such alignments tend to result in synonym words receiving similar vectors. Some authors work instead on the sentence alignment level, without relying on word alignments. An appealing method is to mix a monolingual window-based approach with a multilingual approach, creating both kinds of auxiliary tasks. This is likely to produce vectors that are similar to the window-based approach, but reducing the somewhat undesired effect of the window-based approach in which antonyms (e.g. hot and cold, high and low) tend to receive similar vectors (Faruqui & Dyer, 2014).

5.5.5 CHARACTER-BASED AND SUB-WORD REPRESENTATIONS

An interesting line of work attempts to derive the vector representation of a word from the characters that compose it. Such approaches are likely to be particularly useful for tasks which are syntactic in nature, as the character patterns within words are strongly related to their syntactic function. These approaches also have the benefit of producing very small model sizes (only one vector for each character in the alphabet together with a handful of small matrices needs to be stored), and being able to provide an embedding vector for every word that may be encountered. dos Santos and Gatti (2014) and dos Santos and Zadrozny (2014) model the embedding of a word using a convolutional network (see Section 9) over the characters. Ling et al (2015b) model the embedding of a word using the concatenation of the final states of two RNN (LSTM) encoders (Section 10), one reading the characters from left to right, and the other from right to left. Both produce very strong results for part-of-speech tagging. The work of Ballesteros et al (2015) show that the two-LSTMs encoding of (Ling et al., 2015b) is beneficial also for representing words in dependency parsing of morphologically rich languages.

Deriving representations of words from the representations of their characters is motivated by the *unknown words problem* – what do you do when you encounter a word for which you do not have an embedding vector? Working on the level of characters alleviates this problem to a large extent, as the vocabulary of possible characters is much smaller than the vocabulary of possible words. However, working on the character level is very challenging, as the relationship between form (characters) and function (syntax, semantics) in language is quite loose. Restricting oneself to stay on the character level may be an unnecessarily hard constraint. Some researchers propose a middle-ground, in which a word is represented as a combination of a vector for the word itself with vectors of sub-word units that comprise it. The sub-word embeddings then help in sharing information between different words with similar forms, as well as allowing back-off to the subword level when the word is not observed. At the same time, the models are not forced to rely solely on

form when enough observations of the word are available. Botha and Blunsom (2014) suggest to model the embedding vector of a word as a sum of the word-specific vector if such vector is available, with vectors for the different morphological components that comprise it (the components are derived using Morfessor (Creutz & Lagus, 2007), an unsupervised morphological segmentation method). Gao et al (Gao et al., 2014) suggest using as core features not only the word form itself but also a unique feature (hence a unique embedding vector) for each of the letter-trigrams in the word.

6. Neural Network Training

Neural network training is done by trying to minimize a loss function over a training set, using a gradient-based method. Roughly speaking, all training methods work by repeatedly computing an estimate of the error over the dataset, computing the gradient with respect to the error, and then moving the parameters in the direction of the gradient. Models differ in how the error estimate is computed, and how “moving in the direction of the gradient” is defined. We describe the basic algorithm, *stochastic gradient descent* (SGD), and then briefly mention the other approaches with pointers for further reading. Gradient calculation is central to the approach. Gradients can be efficiently and automatically computed using reverse mode differentiation on a computation graph – a general algorithmic framework for automatically computing the gradient of any network and loss function.

6.1 Stochastic Gradient Training

The common approach for training neural networks is using the stochastic gradient descent (SGD) algorithm (Bottou, 2012; LeCun, Bottou, Orr, & Muller, 1998a) or a variant of it. SGD is a general optimization algorithm. It receives a function f parameterized by θ , a loss function, and desired input and output pairs. It then attempts to set the parameters θ such that the loss of f with respect to the training examples is small. The algorithm works as follows:

Algorithm 1 Online Stochastic Gradient Descent Training

- 1: **Input:** Function $f(\mathbf{x}; \theta)$ parameterized with parameters θ .
 - 2: **Input:** Training set of inputs $\mathbf{x}_1, \dots, \mathbf{x}_n$ and outputs $\mathbf{y}_1, \dots, \mathbf{y}_n$.
 - 3: **Input:** Loss function L .
 - 4: **while** stopping criteria not met **do**
 - 5: Sample a training example $\mathbf{x}_i, \mathbf{y}_i$
 - 6: Compute the loss $L(f(\mathbf{x}_i; \theta), \mathbf{y}_i)$
 - 7: $\hat{\mathbf{g}} \leftarrow$ gradients of $L(f(\mathbf{x}_i; \theta), \mathbf{y}_i)$ w.r.t θ
 - 8: $\theta \leftarrow \theta + \eta_k \hat{\mathbf{g}}$
 - 9: **return** θ
-

The goal of the algorithm is to set the parameters θ so as to minimize the total loss $\sum_{i=1}^n L(f(\mathbf{x}_i; \theta), \mathbf{y}_i)$ over the training set. It works by repeatedly sampling a training example and computing the gradient of the error on the example with respect to the parameters θ (line 7) – the input and expected output are assumed to be fixed, and the loss is treated as a function of the parameters θ . The parameters θ are then updated in the direction of the gradient, scaled by a learning rate η_k (line 8). For further discussion on setting the learning rate, see Section 6.3.

Note that the error calculated in line 6 is based on a single training example, and is thus just a rough estimate of the corpus-wide loss that we are aiming to minimize. The noise in the loss computation may result in inaccurate gradients. A common way of reducing this noise is to estimate the error and the gradients based on a sample of m examples. This gives rise to the *minibatch SGD* algorithm:

Algorithm 2 Minibatch Stochastic Gradient Descent Training

```
1: Input: Function  $f(\mathbf{x}; \theta)$  parameterized with parameters  $\theta$ .  
2: Input: Training set of inputs  $\mathbf{x}_1, \dots, \mathbf{x}_n$  and outputs  $\mathbf{y}_1, \dots, \mathbf{y}_n$ .  
3: Input: Loss function  $L$ .  
4: while stopping criteria not met do  
5:   Sample a minibatch of  $m$  examples  $\{(\mathbf{x}_1, \mathbf{y}_1), \dots, (\mathbf{x}_m, \mathbf{y}_m)\}$   
6:    $\hat{\mathbf{g}} \leftarrow 0$   
7:   for  $i = 1$  to  $m$  do  
8:     Compute the loss  $L(f(\mathbf{x}_i; \theta), \mathbf{y}_i)$   
9:      $\hat{\mathbf{g}} \leftarrow \hat{\mathbf{g}} + \text{gradients of } \frac{1}{m} L(f(\mathbf{x}_i; \theta), \mathbf{y}_i) \text{ w.r.t } \theta$   
10:     $\theta \leftarrow \theta + \eta_k \hat{\mathbf{g}}$   
11: return  $\theta$ 
```

In lines 6 – 9 the algorithm estimates the gradient of the corpus loss based on the minibatch. After the loop, $\hat{\mathbf{g}}$ contains the gradient estimate, and the parameters θ are updated toward $\hat{\mathbf{g}}$. The minibatch size can vary in size from $m = 1$ to $m = n$. Higher values provide better estimates of the corpus-wide gradients, while smaller values allow more updates and in turn faster convergence. Besides the improved accuracy of the gradients estimation, the minibatch algorithm provides opportunities for improved training efficiency. For modest sizes of m , some computing architectures (i.e. GPUs) allow an efficient parallel implementation of the computation in lines 6–9. With a small enough learning rate, SGD is guaranteed to converge to a global optimum if the function is convex. However, it can also be used to optimize non-convex functions such as neural-network. While there are no longer guarantees of finding a global optimum, the algorithm proved to be robust and performs well in practice.

When training a neural network, the parameterized function f is the neural network, and the parameters θ are the layer-transfer matrices, bias terms, embedding matrices and so on. The gradient computation is a key step in the SGD algorithm, as well as in all other neural network training algorithms. The question is, then, how to compute the gradients of the network’s error with respect to the parameters. Fortunately, there is an easy solution in the form of the *backpropagation algorithm* (Rumelhart, Hinton, & Williams, 1986; Lecun, Bottou, Bengio, & Haffner, 1998b). The backpropagation algorithm is a fancy name for methodologically computing the derivatives of a complex expression using the chain-rule, while caching intermediary results. More generally, the backpropagation algorithm is a special case of the reverse-mode automatic differentiation algorithm (Neidinger, 2010, Section 7), (Baydin, Pearlmutter, Radul, & Siskind, 2015; Bengio, 2012). The following section describes reverse mode automatic differentiation in the context of the *computation graph* abstraction.

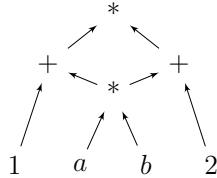
Beyond SGD While the SGD algorithm can and often does produce good results, more advanced algorithms are also available. The *SGD+Momentum* (Polyak, 1964) and *Nesterov Momentum* (Sutskever, Martens, Dahl, & Hinton, 2013) algorithms are variants of SGD in which previous gradients are accumulated and affect the current update. Adaptive learning rate algorithms including AdaGrad (Duchi, Hazan, & Singer, 2011), AdaDelta (Zeiler, 2012),

RMSProp (Tieleman & Hinton, 2012) and Adam (Kingma & Ba, 2014) are designed to select the learning rate for each minibatch, sometimes on a per-coordinate basis, potentially alleviating the need of fiddling with learning rate scheduling. For details of these algorithms, see the original papers or (Bengio et al., 2015, Sections 8.3, 8.4). As many neural-network software frameworks provide implementations of these algorithms, it is easy and sometimes worthwhile to try out different variants.

6.2 The Computation Graph Abstraction

While one can compute the gradients of the various parameters of a network by hand and implement them in code, this procedure is cumbersome and error prone. For most purposes, it is preferable to use automatic tools for gradient computation (Bengio, 2012). The computation-graph abstraction allows us to easily construct arbitrary networks, evaluate their predictions for given inputs (forward pass), and compute gradients for their parameters with respect to arbitrary scalar losses (backward pass).

A computation graph is a representation of an arbitrary mathematical computation as a graph. It is a directed acyclic graph (DAG) in which nodes correspond to mathematical operations or (bound) variables and edges correspond to the flow of intermediary values between the nodes. The graph structure defines the order of the computation in terms of the dependencies between the different components. The graph is a DAG and not a tree, as the result of one operation can be the input of several continuations. Consider for example a graph for the computation of $(a * b + 1) * (a * b + 2)$:



The computation of $a * b$ is shared. We restrict ourselves to the case where the computation graph is connected.

Since a neural network is essentially a mathematical expression, it can be represented as a computation graph.

For example, Figure 3a presents the computation graph for a 1-layer MLP with a softmax output transformation. In our notation, oval nodes represent mathematical operations or functions, and shaded rectangle nodes represent parameters (bound variables). Network inputs are treated as constants, and drawn without a surrounding node. Input and parameter nodes have no incoming arcs, and output nodes have no outgoing arcs. The output of each node is a matrix, the dimensionality of which is indicated above the node.

This graph is incomplete: without specifying the inputs, we cannot compute an output. Figure 3b shows a complete graph for an MLP that takes three words as inputs, and predicts the distribution over part-of-speech tags for the third word. This graph can be used for prediction, but not for training, as the output is a vector (not a scalar) and the graph does not take into account the correct answer or the loss term. Finally, the graph in 3c shows the computation graph for a specific training example, in which the inputs are the (embeddings

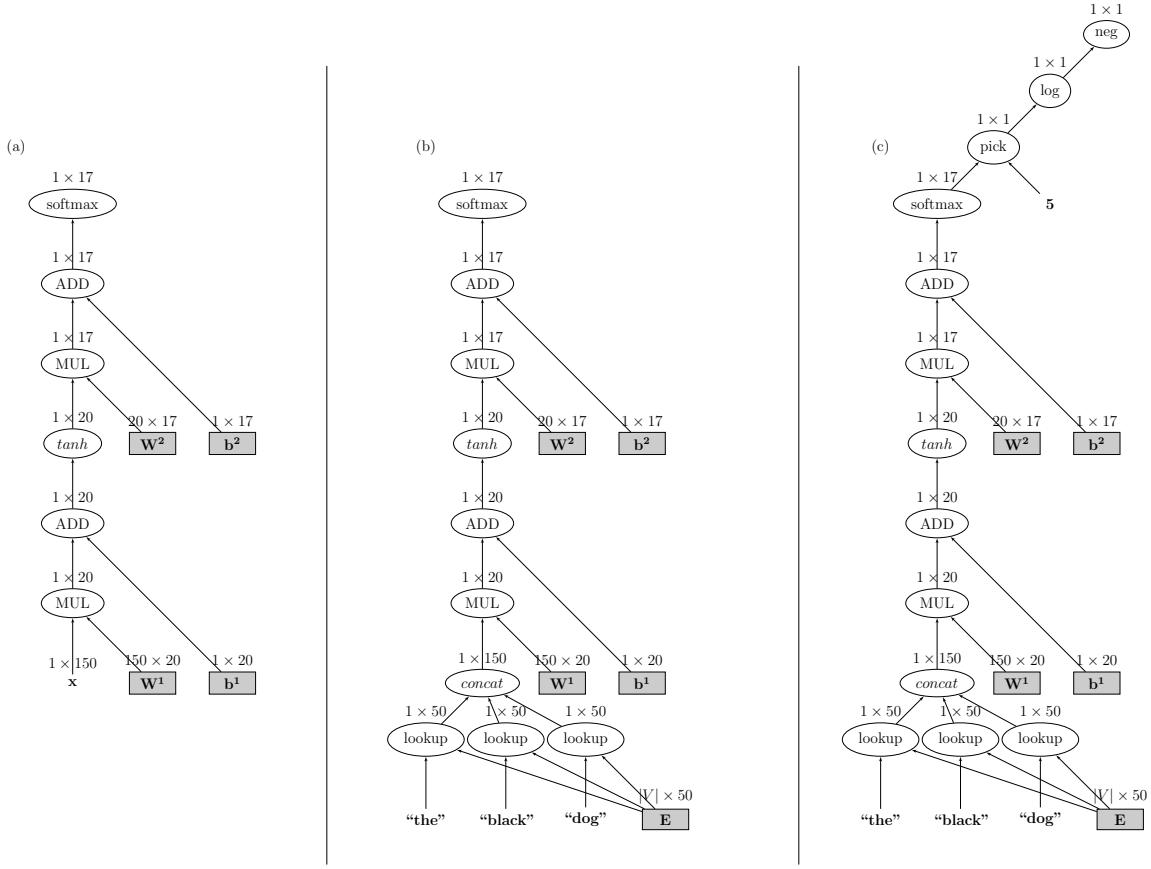


Figure 3: **Computation Graph for MLP1.** (a) Graph with unbound input. (b) Graph with concrete input. (c) Graph with concrete input, expected output, and loss node.

of) the words “the”, “black”, “dog”, and the expected output is “NOUN” (whose index is 5).

Once the graph is built, it is straightforward to run either a forward computation (compute the result of the computation) or a backward computation (computing the gradients), as we show below. Constructing the graphs may look daunting, but is actually very easy using dedicated software libraries and APIs.

Forward Computation The forward pass computes the outputs of the nodes in the graph. Since each node’s output depends only on itself and on its incoming edges, it is trivial to compute the outputs of all nodes by traversing the nodes in a topological order and computing the output of each node given the already computed outputs of its predecessors.

More formally, in a graph of N nodes, we associate each node with an index i according to their topological ordering. Let f_i be the function computed by node i (e.g. *multiplication*, *addition*, ...). Let $\pi(i)$ be the parent nodes of node i , and $\pi^{-1}(i) = \{j \mid i \in \pi(j)\}$ the children nodes of node i (these are the arguments of f_i). Denote by $v(i)$ the output of node

i , that is, the application of f_i to the output values of its arguments $\pi^{-1}(i)$. For variable and input nodes, f_i is a constant function and $\pi^{-1}(i)$ is empty. The Forward algorithm computes the values $v(i)$ for all $i \in [1, N]$.

Algorithm 3 Computation Graph Forward Pass

```

1: for  $i = 1$  to  $N$  do
2:   Let  $a_1, \dots, a_m = \pi^{-1}(i)$ 
3:    $v(i) \leftarrow f_i(v(a_1), \dots, v(a_m))$ 
```

Backward Computation (Derivatives, Backprop) The backward pass begins by designating a node N with scalar (1×1) output as a loss-node, and running forward computation up to that node. The backward computation will compute the gradients with respect to that node's value. Denote by $d(i)$ the quantity $\frac{\partial N}{\partial i}$. The backpropagation algorithm is used to compute the values $d(i)$ for all nodes i .

The backward pass fills a table $d(i)$ as follows:

Algorithm 4 Computation Graph Backward Pass (Backpropagation)

```

1:  $d(N) \leftarrow 1$ 
2: for  $i = N-1$  to  $1$  do
3:    $d(i) \leftarrow \sum_{j \in \pi(i)} d(j) \cdot \frac{\partial f_j}{\partial i}$ 
```

The quantity $\frac{\partial f_j}{\partial i}$ is the partial derivative of $f_j(\pi^{-1}(j))$ w.r.t the argument $i \in \pi^{-1}(j)$. This value depends on the function f_j and the values $v(a_1), \dots, v(a_m)$ (where $a_1, \dots, a_m = \pi^{-1}(j)$) of its arguments, which were computed in the forward pass.

Thus, in order to define a new kind of node, one need to define two methods: one for calculating the forward value $v(i)$ based on the nodes inputs, and the another for calculating $\frac{\partial f_i}{\partial x}$ for each $x \in \pi^{-1}(i)$.

For further information on automatic differentiation see (Neidinger, 2010, Section 7), (Baydin et al., 2015). For more in depth discussion of the backpropagation algorithm and computation graphs (also called flow graphs) see (Bengio et al., 2015, Section 6.4), (Lecun et al., 1998b; Bengio, 2012). For a popular yet technical presentation, see Chris Olah's description at <http://colah.github.io/posts/2015-08-Backprop/>.

Software Several software packages implement the computation-graph model, including Theano¹⁸, Chainer¹⁹, penne²⁰ and CNN/pyCNN²¹. All these packages support all the essential components (node types) for defining a wide range of neural network architectures, covering the structures described in this tutorial and more. Graph creation is made almost transparent by use of operator overloading. The framework defines a type for representing graph nodes (commonly called *expressions*), methods for constructing nodes for inputs and

18. <http://deeplearning.net/software/theano/>

19. <http://chainer.org>

20. <https://bitbucket.org/ndnlp/penne>

21. <https://github.com/clab/cnn>

parameters, and a set of functions and mathematical operations that take expressions as input and result in more complex expressions. For example, the python code for creating the computation graph from Figure (3c) using the pyCNN framework is:

```

from pycnn import *
# model initialization.
model = Model()
model.add_parameters("W1", (20,150))
model.add_parameters("b1", 20)
model.add_parameters("W2", (17,20))
model.add_parameters("b2", 17)
model.add_lookup_parameters("words", (100, 50))

# Building the computation graph:
renew_cg() # create a new graph.
# Wrap the model parameters as graph-nodes.
W1 = parameter(model["W1"])
b1 = parameter(model["b1"])
W2 = parameter(model["W2"])
b2 = parameter(model["b2"])
def get_index(x): return 1
# Generate the embeddings layer.
vthe = lookup(model["words"], get_index("the"))
vblack = lookup(model["words"], get_index("black"))
vdog = lookup(model["words"], get_index("dog"))

# Connect the leaf nodes into a complete graph.
x = concatenate([vthe, vblack, vdog])
output = softmax(W2*(tanh(W1*x)+b1)+b2)
loss = -log(pick(output, 5))

loss_value = loss.forward()
loss.backward() # the gradient is computed
                # and stored in the corresponding
                # parameters.

```

Most of the code involves various initializations: the first block defines model parameters that are shared between different computation graphs (recall that each graph corresponds to a specific training example). The second block turns the model parameters into the graph-node (Expression) types. The third block retrieves the Expressions for the embeddings of the input words. Finally, the fourth block is where the graph is created. Note how transparent the graph creation is – there is an almost a one-to-one correspondence between creating the graph and describing it mathematically. The last block shows a forward and backward pass. The other software frameworks follow similar patterns.

Theano involves an optimizing compiler for computation graphs, which is both a blessing and a curse. On the one hand, once compiled, large graphs can be run efficiently on either the CPU or a GPU, making it ideal for large graphs with a fixed structure, where only the inputs change between instances. However, the compilation step itself can be costly, and it makes the interface a bit cumbersome to work with. In contrast, the other packages focus on building large and dynamic computation graphs and executing them “on the fly” without a compilation step. While the execution speed may suffer with respect to Theano’s optimized version, these packages are especially convenient when working with the recurrent and

recursive networks described in Sections 10, 12 as well as in structured prediction settings as described in Section 8.

Implementation Recipe Using the computation graph abstraction, the pseudo-code for a network training algorithm is given in Algorithm 5.

Algorithm 5 Neural Network Training with Computation Graph Abstraction (using mini-batches of size 1)

```
1: Define network parameters.  
2: for iteration = 1 to N do  
3:   for Training example  $\mathbf{x}_i, \mathbf{y}_i$  in dataset do  
4:     loss_node  $\leftarrow$  build_computation_graph( $\mathbf{x}_i, \mathbf{y}_i$ , parameters)  
5:     loss_node.forward()  
6:     gradients  $\leftarrow$  loss_node().backward()  
7:     parameters  $\leftarrow$  update_parameters(parameters, gradients)  
8: return parameters.
```

Here, `build_computation_graph` is a user-defined function that builds the computation graph for the given input, output and network structure, returning a single loss node. `update_parameters` is an optimizer specific update rule. The recipe specifies that a new graph is created for each training example. This accommodates cases in which the network structure varies between training example, such as recurrent and recursive neural networks, to be discussed in Sections 10 – 12. For networks with fixed structures, such as an MLPs, it may be more efficient to create one base computation graph and vary only the inputs and expected outputs between examples.

Network Composition As long as the network’s output is a vector ($1 \times k$ matrix), it is trivial to compose networks by making the output of one network the input of another, creating arbitrary networks. The computation graph abstractions makes this ability explicit: a node in the computation graph can itself be a computation graph with a designated output node. One can then design arbitrarily deep and complex networks, and be able to easily evaluate and train them thanks to automatic forward and gradient computation. This makes it easy to define and train networks for structured outputs and multi-objective training, as we discuss in Section 7, as well as complex recurrent and recursive networks, as discussed in Sections 10–12.

6.3 Optimization Issues

Once the gradient computation is taken care of, the network is trained using SGD or another gradient-based optimization algorithm. The function being optimized is not convex, and for a long time training of neural networks was considered a “black art” which can only be done by selected few. Indeed, many parameters affect the optimization process, and care has to be taken to tune these parameters. While this tutorial is not intended as a comprehensive guide to successfully training neural networks, we do list here a few of the prominent issues. For further discussion on optimization techniques and algorithms for neural networks, refer to (Bengio et al., 2015, Chapter 8). For some theoretical discussion and analysis, refer

to (Glorot & Bengio, 2010). For various practical tips and recommendations, see (LeCun et al., 1998a; Bottou, 2012).

Initialization The non-convexity of the loss function means the optimization procedure may get stuck in a local minimum or a saddle point, and that starting from different initial points (e.g. different random values for the parameters) may result in different results. Thus, it is advised to run several restarts of the training starting at different random initializations, and choosing the best one based on a development set.²² The amount of variance in the results is different for different network formulations and datasets, and cannot be predicted in advance.

The magnitude of the random values has an important effect on the success of training. An effective scheme due to Glorot and Bengio (2010), called *xavier initialization* after Glorot’s first name, suggests initializing a weight matrix $\mathbf{W} \in \mathbb{R}^{d_{in} \times d_{out}}$ as:

$$\mathbf{W} \sim U \left[-\frac{\sqrt{6}}{\sqrt{d_{in} + d_{out}}}, +\frac{\sqrt{6}}{\sqrt{d_{in} + d_{out}}} \right]$$

where $U[a, b]$ is a uniformly sampled random value in the range $[a, b]$. This advice works well on many occasions, and is the preferred default initialization method by many.

Analysis by He et al (2015) suggests that when using ReLU non-linearities, the weights should be initialized by sampling from a zero-mean Gaussian distribution whose standard deviation is $\sqrt{\frac{2}{d_{in}}}$. This initialization was found by He et al to work better than xavier initialization in an image classification task, especially when deep networks were involved.

Vanishing and Exploding Gradients In deep networks, it is common for the error gradients to either vanish (become exceedingly close to 0) or explode (become exceedingly high) as they propagate back through the computation graph. The problem becomes more severe in deeper networks, and especially so in recursive and recurrent networks (Pascanu, Mikolov, & Bengio, 2012). Dealing with the vanishing gradients problem is still an open research question. Solutions include making the networks shallower, step-wise training (first train the first layers based on some auxiliary output signal, then fix them and train the upper layers of the complete network based on the real task signal), or specialized architectures that are designed to assist in gradient flow (e.g., the LSTM and GRU architectures for recurrent networks, discussed in Section 11). Dealing with the exploding gradients has a simple but very effective solution: clipping the gradients if their norm exceeds a given threshold. Let $\hat{\mathbf{g}}$ be the gradients of all parameters in the network, and $\|\hat{\mathbf{g}}\|$ be their L_2 norm. Pascanu et al (2012) suggest to set: $\hat{\mathbf{g}} \leftarrow \frac{\text{threshold}}{\|\hat{\mathbf{g}}\|} \hat{\mathbf{g}}$ if $\|\hat{\mathbf{g}}\| > \text{threshold}$.

Saturation and Dead Neurons Layers with *tanh* and *sigmoid* activations can become saturated – resulting in output values for that layer that are all close to one, the upper-limit of the activation function. Saturated neurons have very small gradients, and should be avoided. Layers with the ReLU activation cannot be saturated, but can “die” – most or all values are negative and thus clipped at zero for all inputs, resulting in a gradient of zero for that layer. If your network does not train well, it is advisable to monitor the network for saturated or dead layers. Saturated neurons are caused by too large values

22. When debugging, and for reproducibility of results, it is advised to used a fixed random seed.

entering the layer. This may be controlled for by changing the initialization, scaling the range of the input values, or changing the learning rate. Dead neurons are caused by all weights entering the layer being negative (for example this can happen after a large gradient update). Reducing the learning rate will help in this situation. For saturated layers, another option is to normalize the values in the saturated layer after the activation, i.e. instead of $g(\mathbf{h}) = \tanh(\mathbf{h})$ using $g(\mathbf{h}) = \frac{\tanh(\mathbf{h})}{\|\tanh(\mathbf{h})\|}$. Layer normalization is an effective measure for countering saturation, but is also expensive in terms of gradient computation.

Shuffling The order in which the training examples are presented to the network is important. The SGD formulation above specifies selecting a random example in each turn. In practice, most implementations go over the training example in order. It is advised to shuffle the training examples before each pass through the data.

Learning Rate Selection of the learning rate is important. Too large learning rates will prevent the network from converging on an effective solution. Too small learning rates will take very long time to converge. As a rule of thumb, one should experiment with a range of initial learning rates in range $[0, 1]$, e.g. 0.001, 0.01, 0.1, 1. Monitor the network's loss over time, and decrease the learning rate once the network seem to be stuck in a fixed region. *Learning rate scheduling* decrease the rate as a function of the number of observed minibatches. A common schedule is dividing the initial learning rate by the iteration number. Léon Bottou (2012) recommends using a learning rate of the form $\eta_t = \eta_0(1 + \eta_0\lambda t)^{-1}$ where η_0 is the initial learning rate, η_t is the learning rate to use on the t th training example, and λ is an additional hyperparameter. He further recommends determining a good value of η_0 based on a small sample of the data prior to running on the entire dataset.

Minibatches Parameter updates occur either every training example (minibatches of size 1) or every k training examples. Some problems benefit from training with larger minibatch sizes. In terms of the computation graph abstraction, one can create a computation graph for each of the k training examples, and then connecting the k loss nodes under an averaging node, whose output will be the loss of the minibatch. Large minibatched training can also be beneficial in terms of computation efficiency on specialized computing architectures such as GPUs. This is beyond the scope of this tutorial.

6.4 Regularization

Neural network models have many parameters, and overfitting can easily occur. Overfitting can be alleviated to some extent by *regularization*. A common regularization method is L_2 regularization, placing a squared penalty on parameters with large values by adding an additive $\frac{\lambda}{2}\|\theta\|^2$ term to the objective function to be minimized, where θ is the set of model parameters, $\|\cdot\|^2$ is the squared L_2 norm (sum of squares of the values), and λ is a hyperparameter controlling the amount of regularization.

A recently proposed alternative regularization method is *dropout* (Hinton, Srivastava, Krizhevsky, Sutskever, & Salakhutdinov, 2012). The dropout method is designed to prevent the network from learning to rely on specific weights. It works by randomly dropping (setting to 0) half of the neurons in the network (or in a specific layer) in each training example. Work by Wager et al (2013) establishes a strong connection between the dropout method

and L_2 regularization. Gal and Gharamani (2015) show that a multi-layer perceptron with dropout applied at every layer can be interpreted as Bayesian model averaging.

The dropout technique is one of the key factors contributing to very strong results of neural-network methods on image classification tasks (Krizhevsky, Sutskever, & Hinton, 2012), especially when combined with ReLU activation units (Dahl, Sainath, & Hinton, 2013). The dropout technique is effective also in NLP applications of neural networks.

7. Cascading and Multi-task Learning

The combination of online training methods with automatic gradient computations using the computation graph abstraction allows for an easy implementation of model cascading, parameter sharing and multi-task learning.

Model cascading is a powerful technique in which large networks are built by composing them out of smaller component networks. For example, we may have a feed-forward network for predicting the part of speech of a word based on its neighbouring words and/or the characters that compose it. In a pipeline approach, we would use this network for predicting parts of speech, and then feed the predictions as input features to neural network that does syntactic chunking or parsing. Instead, we could think of the hidden layers of this network as an encoding that captures the relevant information for predicting the part of speech. In a cascading approach, we take the hidden layers of this network and connect them (and not the part of speech prediction themselves) as the inputs for the syntactic network. We now have a larger network that takes as input sequences of words and characters, and outputs a syntactic structure. The computation graph abstraction allows us to easily propagate the error gradients from the syntactic task loss all the way back to the characters.

To combat the vanishing gradient problem of deep networks, as well as to make better use of available training material, the individual component network's parameters can be bootstrapped by training them separately on a relevant task, before plugging them in to the larger network for further tuning. For example, the part-of-speech predicting network can be trained to accurately predict parts-of-speech on a relatively large annotated corpus, before plugging its hidden layer into the syntactic parsing network for which less training data is available. In case the training data provide direct supervision for both tasks, we can make use of it during training by creating a network with two outputs, one for each task, computing a separate loss for each output, and then summing the losses into a single node from which we backpropagate the error gradients.

Model cascading is very common when using convolutional, recursive and recurrent neural networks, where, for example, a recurrent network is used to encode a sentence into a fixed sized vector, which is then used as the input of another network. The supervision signal of the recurrent network comes primarily from the upper network that consumes the recurrent network's output as its inputs.

Multi-task learning is used when we have related prediction tasks that do not necessarily feed into one another, but we do believe that information that is useful for one type of prediction can be useful also to some of the other tasks. For example, chunking, named entity recognition (NER) and language modeling are examples of synergistic tasks. Information for predicting chunk boundaries, named-entity boundaries and the next word in the sentence all rely on some shared underlying syntactic-semantic representation. Instead of training a separate network for each task, we can create a single network with several outputs. A common approach is to have a multi-layer feed-forward network, whose final hidden layer (or a concatenation of all hidden layers) is then passed to different output layers. This way, most of the parameters of the network are shared between the different tasks. Useful information learned from one task can then help to disambiguate other tasks. Again, the computation graph abstraction makes it very easy to construct such networks and compute

the gradients for them, by computing a separate loss for each available supervision signal, and then summing the losses into a single loss that is used for computing the gradients. In case we have several corpora, each with different kind of supervision signal (e.g. we have one corpus for NER and another for chunking), the training procedure will shuffle all of the available training example, performing gradient computation and updates with respect to a different loss in every turn. Multi-task learning in the context of language-processing is introduced and discussed in (Collobert et al., 2011).

8. Structured Output Prediction

Many problems in NLP involve structured outputs: cases where the desired output is not a class label or distribution over class labels, but a structured object such as a sequence, a tree or a graph. Canonical examples are sequence tagging (e.g. part-of-speech tagging) sequence segmentation (chunking, NER), and syntactic parsing. In this section, we discuss how feed-forward neural network models can be used for structured tasks. In later sections we discuss specialized neural network models for dealing with sequences (Section 10) and trees (Section 12).

8.1 Greedy Structured Prediction

The greedy approach to structured prediction is to decompose the structure prediction problem into a sequence of local prediction problems and training a classifier to perform each local decision. At test time, the trained classifier is used in a greedy manner. Examples of this approach are left-to-right tagging models (Giménez & Màrquez, 2004) and greedy transition-based parsing (Nivre, 2008). Such approaches are easily adapted to use neural networks by simply replacing the local classifier from a linear classifier such as an SVM or a logistic regression model to a neural network, as demonstrated in (Chen & Manning, 2014; Lewis & Steedman, 2014).

The greedy approaches suffer from error propagation, where mistakes in early decisions carry over and influence later decisions. The overall higher accuracy achievable with non-linear neural network classifiers helps in offsetting this problem to some extent. In addition, training techniques were proposed for mitigating the error propagation problem by either attempting to take easier predictions before harder ones (the easy-first approach (Goldberg & Elhadad, 2010)) or making training conditions more similar to testing conditions by exposing the training procedure to inputs that result from likely mistakes (Hal Daumé III, Langford, & Marcu, 2009; Goldberg & Nivre, 2013). These are effective also for training greedy neural network models, as demonstrated by Ma et al (Ma, Zhang, & Zhu, 2014) (easy-first tagger) and (?) (dynamic oracle training for greedy dependency parsing).

8.2 Search Based Structured Prediction

The common approach to predicting natural language structures is search based. For in-depth discussion of search-based structure prediction in NLP, see the book by Smith (Smith, 2011). The techniques can easily be adapted to use a neural-network. In the neural-networks literature, such models were discussed under the framework of *energy based learning* (LeCun et al., 2006, Section 7). They are presented here using setup and terminology familiar to the NLP community.

Search-based structured prediction is formulated as a search problem over possible structures:

$$predict(x) = \arg \max_{y \in \mathcal{Y}(x)} score(x, y)$$

where x is an input structure, y is an output over x (in a typical example x is a sentence and y is a tag-assignment or a parse-tree over the sentence), $\mathcal{Y}(x)$ is the set of all valid

structures over x , and we are looking for an output y that will maximize the score of the x, y pair.

The scoring function is defined as a linear model:

$$score(x, y) = \Phi(x, y) \cdot \mathbf{w}$$

where Φ is a feature extraction function and \mathbf{w} is a weight vector.

In order to make the search for the optimal y tractable, the structure y is decomposed into parts, and the feature function is defined in terms of the parts, where $\phi(p)$ is a part-local feature extraction function:

$$\Phi(x, y) = \sum_{p \in parts(x, y)} \phi(p)$$

Each part is scored separately, and the structure score is the sum of the component parts scores:

$$score(x, y) = \mathbf{w} \cdot \Phi(x, y) = \mathbf{w} \cdot \sum_{p \in y} \phi(p) = \sum_{p \in y} \mathbf{w} \cdot \phi(p) = \sum_{p \in y} score(p)$$

where $p \in y$ is a shorthand for $p \in parts(x, y)$. The decomposition of y into parts is such that there exists an inference algorithm that allows for efficient search for the best scoring structure given the scores of the individual parts.

One can now trivially replace the linear scoring function over parts with a neural-network:

$$score(x, y) = \sum_{p \in y} score(p) = \sum_{p \in y} NN(c(p))$$

where $c(p)$ maps the part p into a d_{in} dimensional vector.

In case of a one hidden-layer feed-forward network:

$$score(x, y) = \sum_{p \in y} NN_{MLP1}(c(p)) = \sum_{p \in y} (g(c(p)\mathbf{W}^1 + \mathbf{b}^1))\mathbf{w}$$

$c(p) \in \mathbb{R}^{d_{in}}$, $\mathbf{W}^1 \in \mathbb{R}^{d_{in} \times d_1}$, $\mathbf{b}^1 \in \mathbb{R}^{d_1}$, $\mathbf{w} \in \mathbb{R}^{d_1}$. A common objective in structured prediction is making the gold structure y score higher than any other structure y' , leading to the following (generalized perceptron) loss:

$$\max_{y'} score(x, y') - score(x, y)$$

In terms of implementation, this means: create a computation graph CG_p for each of the possible parts, and calculate its score. Then, run inference over the scored parts to find the best scoring structure y' . Connect the output nodes of the computation graphs corresponding to parts in the gold (predicted) structure y (y') into a summing node CG_y (CG'_y). Connect CG_y and CG'_y using a “minus” node, CG_l , and compute the gradients.

As argued in (LeCun et al., 2006, Section 5), the generalized perceptron loss may not be a good loss function when training structured prediction neural networks as it does not have a margin, and a margin-based hinge loss is preferred:

$$\max(0, m + \text{score}(x, y) - \max_{y' \neq y} \text{score}(x, y'))$$

It is trivial to modify the implementation above to work with the hinge loss.

Note that in both cases we lose the nice properties of the linear model. In particular, the model is no longer convex. This is to be expected, as even the simplest non-linear neural network is already non-convex. Nonetheless, we could still use standard neural-network optimization techniques to train the structured model.

Training and inference is slower, as we have to evaluate the neural network (and take gradients) $|\text{parts}(x, y)|$ times.

Structured prediction is a vast field and is beyond the scope of this tutorial, but loss functions, regularizers and methods described in, e.g., (Smith, 2011), such as cost-augmented decoding, can be easily applied or adapted to the neural-network framework.²³

Probabilistic objective (CRF) In a probabilistic framework (“CRF”), we treat each of the parts scores as a *clique potential* (see (Smith, 2011)) and define the score of each structure y to be:

$$\text{score}_{\text{CRF}}(x, y) = P(y|x) = \frac{\sum_{p \in y} e^{\text{score}(p)}}{\sum_{y' \in \mathcal{Y}(x)} \sum_{p \in y'} e^{\text{score}(p)}} = \frac{\sum_{p \in y} e^{NN(c(p))}}{\sum_{y' \in \mathcal{Y}(x)} \sum_{p \in y'} e^{NN(c(p))}}$$

The scoring function defines a conditional distribution $P(y|x)$, and we wish to set the parameters of the network such that corpus conditional log likelihood $\sum_{(x_i, y_i) \in \text{training}} \log P(y_i|x_i)$ is maximized.

The loss for a given training example (x, y) is then: $-\log \text{score}_{\text{CRF}}(x, y)$. Taking the gradient with respect to the loss is as involved as building the associated computation graph. The tricky part is the denominator (the *partition function*) which requires summing over the potentially exponentially many structures in \mathcal{Y} . However, for some problems, a dynamic programming algorithm exists for efficiently solving the summation in polynomial time. When such an algorithm exists, it can be adapted to also create a polynomial-size computation graph.

When an efficient enough algorithm for computing the partition function is not available, approximate methods can be used. For example, one may use beam search for inference, and for the partition function sum over the structures remaining in the beam instead of over the exponentially large $\mathcal{Y}(x)$.

A hinge based approach was used by Pei et al (2015) for arc-factored dependency parsing, and the probabilistic approach by Durrett and Klein (Durrett & Klein, 2015) for a CRF constituency parser. The approximate beam-based partition function was effectively used by Zhou et al (2015) in a transition based parser.

Reranking When searching over all possible structures is intractable, inefficient or hard to integrate into a model, reranking methods are often used. In the reranking framework (Charniak & Johnson, 2005; Collins & Koo, 2005) a base model is used to produce a

23. One should keep in mind that the resulting objectives are no longer convex, and so lack the formal guarantees and bounds associated with convex optimization problems. Similarly, the theory, learning bounds and guarantees associated with the algorithms do not automatically transfer to the neural versions.

list of the k -best scoring structures. A more complex model is then trained to score the candidates in the k -best list such that the best structure with respect to the gold one is scored highest. As the search is now performed over k items rather than over an exponential space, the complex model can condition on (extract features from) arbitrary aspects of the scored structure. Reranking methods are natural candidates for structured prediction using neural-network models, as they allow the modeler to focus on the feature extraction and network structure, while removing the need to integrate the neural network scoring into a decoder. Indeed, reranking methods are often used for experimenting with neural models that are not straightforward to integrate into a decoder, such as convolutional, recurrent and recursive networks, which will be discussed in later sections. Works using the reranking approach include (Socher et al., 2013; Auli et al., 2013; Le & Zuidema, 2014; Zhu et al., 2015a)

MEMM and hybrid approaches Other formulations are, of course, also possible. For example, an MEMM (McCallum, Freitag, & Pereira, 2000) can be trivially adapted to the neural network world by replacing the logistic regression (“Maximum Entropy”) component with an MLP.

Hybrid approaches between neural networks and linear models are also explored. In particular, Weiss et al (Weiss et al., 2015) report strong results for transition-based dependency parsing in a two-stage model. In the first stage, a static feed-forward neural network (MLP2) is trained to perform well on each of the individual decisions of the structured problem in isolation. In the second stage, the neural network model is held fixed, and the different layers (output as well as hidden layer vectors) for each input are then concatenated and used as the input features of a linear structured perceptron model (Collins, 2002) that is trained to perform beam-search for the best resulting structure. While it is not clear that such training regime is more effective than training a single structured-prediction neural network, the use of two simpler, isolated models allowed the researchers to perform a much more extensive hyper-parameter search (e.g. tuning layer sizes, activation functions, learning rates and so on) for each model than is feasible with more complicated networks.

9. Convolutional Layers

Sometimes we are interested in making predictions based on ordered sets of items (e.g. the sequence of words in a sentence, the sequence of sentences in a document and so on). Consider for example predicting the sentiment (positive, negative or neutral) of a sentence. Some of the sentence words are very informative of the sentiment, other words are less informative, and to a good approximation, an informative clue is informative regardless of its position in the sentence. We would like to feed all of the sentence words into a learner, and let the training process figure out the important clues. One possible solution is feeding a CBOW representation into a fully connected network such as an MLP. However, a downside of the CBOW approach is that it ignores the ordering information completely, assigning the sentences “it was not good, it was actually quite bad” and “it was not bad, it was actually quite good” the exact same representation. While the global position of the indicators “not good” and “not bad” does not matter for the classification task, the local ordering of the words (that the word “not” appears right before the word “bad”) is very important. A naive approach would suggest embedding word-pairs (bi-grams) rather than words, and building a CBOW over the embedded bigrams. While such architecture could be effective, it will result in huge embedding matrices, will not scale for longer n-grams, and will suffer from data sparsity problems as it does not share statistical strength between different n-grams (the embedding of “quite good” and “very good” are completely independent of one another, so if the learner saw only one of them during training, it will not be able to deduce anything about the other based on its component words). The convolution-and-pooling (also called convolutional neural networks, or CNNs) architecture is an elegant and robust solution to the this modeling problem. A convolutional neural network is designed to identify indicative local predictors in a large structure, and combine them to produce a fixed size vector representation of the structure, capturing these local aspects that are most informative for the prediction task at hand.

Convolution-and-pooling architectures (LeCun & Bengio, 1995) evolved in the neural networks vision community, where they showed great success as object detectors – recognizing an object from a predefined category (“cat”, “bicycles”) regardless of its position in the image (Krizhevsky et al., 2012). When applied to images, the architecture is using 2-dimensional (grid) convolutions. When applied to text, NLP we are mainly concerned with 1-d (sequence) convolutions. Convolutional networks were introduced to the NLP community in the pioneering work of Collobert, Weston and Colleagues (2011) who used them for semantic-role labeling, and later by Kalchbrenner et al (2014) and Kim (Kim, 2014) who used them for sentiment and question-type classification.

9.1 Basic Convolution + Pooling

The main idea behind a convolution and pooling architecture for language tasks is to apply a non-linear (learned) function over each instantiation of a k -word sliding window over the sentence. This function (also called “filter”) transforms a window of k words into a d dimensional vector that captures important properties of the words in the window (each dimension is sometimes referred to in the literature as a “channel”). Then, a “pooling” operation is used combine the vectors resulting from the different windows into a single d -dimensional vector, by taking the max or the average value observed in each of the d

channels over the different windows. The intention is to focus on the most important “features” in the sentence, regardless of their location. The d -dimensional vector is then fed further into a network that is used for prediction. The gradients that are propagated back from the network’s loss during the training process are used to tune the parameters of the filter function to highlight the aspects of the data that are important for the task the network is trained for. Intuitively, when the sliding window is run over a sequence, the filter function learns to identify informative k-grams.

More formally, consider a sequence of words $\mathbf{x} = x_1, \dots, x_n$, each with their corresponding d_{emb} dimensional word embedding $v(x_i)$. A 1d convolution layer²⁴ of width k works by moving a sliding window of size k over the sentence, and applying the same “filter” to each window in the sequence $(v(x_i); v(x_{i+1}), \dots; v(x_{i+k-1}))$. The filter function is usually a linear transformation followed by a non-linear activation function.

Let the concatenated vector of the i th window be $\mathbf{w}_i = v(x_i); v(x_{i+1}); v(x_{i+k-1})$, $\mathbf{w}_i \in \mathbb{R}^{kd_{emb}}$. Depending on whether we pad the sentence with $k - 1$ words to each side, we may get either $m = n - k + 1$ (*narrow convolution*) or $m = n + k + 1$ windows (*wide convolution*) (Kalchbrenner et al., 2014). The result of the convolution layer is m vectors $\mathbf{p}_1, \dots, \mathbf{p}_m$, $\mathbf{p}_i \in \mathbb{R}^{d_{conv}}$ where:

$$\mathbf{p}_i = g(\mathbf{w}_i \mathbf{W} + \mathbf{b})$$

g is a non-linear activation function that is applied element-wise, $\mathbf{W} \in \mathbb{R}^{k \cdot d_{emb} \times d_{conv}}$ and $\mathbf{b} \in \mathbb{R}^{d_{conv}}$ are parameters of the network. Each \mathbf{p}_i is a d_{conv} dimensional vector, encoding the information in \mathbf{w}_i . Ideally, each dimension captures a different kind of indicative information. The m vectors are then combined using a *max pooling layer*, resulting in a single d_{conv} dimensional vector \mathbf{c} .

$$c_j = \max_{1 \leq i \leq m} \mathbf{p}_i[j]$$

$\mathbf{p}_i[j]$ denotes the j th component of \mathbf{p}_i . The effect of the max-pooling operation is to get the most salient information across window positions. Ideally, each dimension will “specialize” in a particular sort of predictors, and max operation will pick on the most important predictor of each type.

Figure 4 provides an illustration of the process.

The resulting vector \mathbf{c} is a representation of the sentence in which each dimension reflects the most salient information with respect to some prediction task. \mathbf{c} is then fed into a downstream network layers, perhaps in parallel to other vectors, culminating in an output layer which is used for prediction. The training procedure of the network calculates the loss with respect to the prediction task, and the error gradients are propagated all the way back through the pooling and convolution layers, as well as the embedding layers.²⁵

-
- 24. 1d here refers to a convolution operating over 1-dimensional inputs such as sequences, as opposed to 2d convolutions which are applied to images.
 - 25. Besides being useful for prediction, a by-product of the training procedure is a set of parameters \mathbf{W}, \mathbf{B} and embeddings $v()$ that can be used in a convolution and pooling architecture to encode arbitrary length sentences into fixed-size vectors, such that sentences that share the same kind of predictive information will be close to each other.

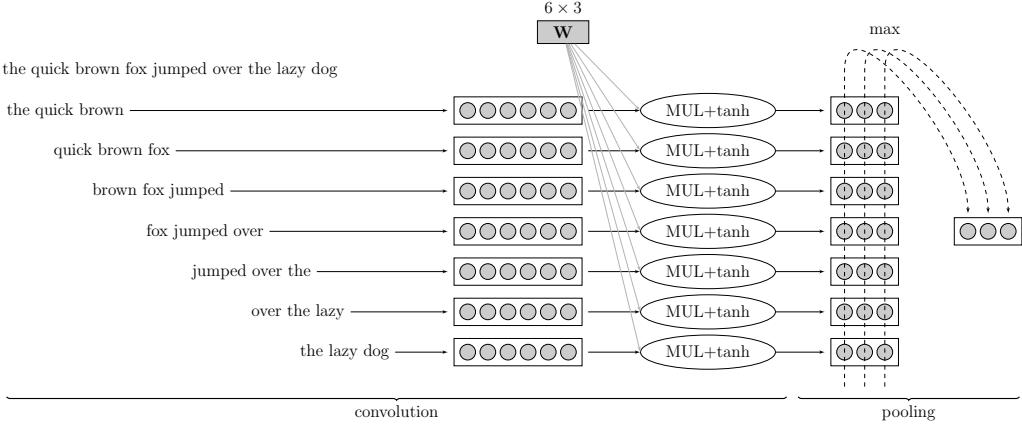


Figure 4: 1d convolution+pooling over the sentence “the quick brown fox jumped over the lazy dog”. This is a narrow convolution (no padding is added to the sentence) with a window size of 3. Each word is translated to a 2-dim embedding vector (not shown). The embedding vectors are then concatenated, resulting in 6-dim window representations. Each of the seven windows is transferred through a 6×3 filter (linear transformation followed by element-wise tanh), resulting in seven 3-dimensional filtered representations. Then, a max-pooling operation is applied, taking the max over each dimension, resulting in a final 3-dimensional pooled vector.

While max-pooling is the most common pooling operation in text applications, other pooling operations are also possible, the second most common operation being *average pooling*, taking the average value of each index instead of the max.

9.2 Dynamic, Hierarchical and k-max Pooling

Rather than performing a single pooling operation over the entire sequence, we may want to retain some positional information based on our domain understanding of the prediction problem at hand. To this end, we can split the vectors \mathbf{p}_i into ℓ distinct groups, apply the pooling separately on each group, and then concatenate the ℓ resulting d_{conv} vectors $\mathbf{c}_1, \dots, \mathbf{c}_\ell$. The division of the \mathbf{p}_i s into groups is performed based on domain knowledge. For example, we may conjecture that words appearing early in the sentence are more indicative than words appearing late. We can then split the sequence into ℓ equally sized regions, applying a separate max-pooling to each region. For example, Johnson and Zhang (Johnson & Zhang, 2014) found that when classifying documents into topics, it is useful to have 20 average-pooling regions, clearly separating the initial sentences (where the topic is usually introduced) from later ones, while for a sentiment classification task a single max-pooling operation over the entire sentence was optimal (suggesting that one or two very strong signals are enough to determine the sentiment, regardless of the position in the sentence).

Similarly, in a relation extraction kind of task we may be given two words and asked to determine the relation between them. We could argue that the words before the first word, the words after the second word, and the words between them provide three different kinds of information (Chen et al., 2015). We can thus split the \mathbf{p}_i vectors accordingly, pooling separately the windows resulting from each group.

Another variation is performing *hierarchical pooling*, in which we have a succession of convolution and pooling layers, where each stage applies a convolution to a sequence, pools every k neighboring vectors, performs a convolution on the resulting pooled sequence, applies another convolution and so on. This architecture allows sensitivity to increasingly larger structures.

Finally, (Kalchbrenner et al., 2014) introduced a *k-max pooling* operation, in which the top k values in each dimension are retained instead of only the best one, while preserving the order in which they appeared in the text. For example a, consider the following matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 9 & 6 & 5 \\ 2 & 3 & 1 \\ 7 & 8 & 1 \\ 3 & 4 & 1 \end{bmatrix}$$

A 1-max pooling over the column vectors will result in $[9 \ 8 \ 5]$, while a 2-max pooling will result in the following matrix: $\begin{bmatrix} 9 & 6 & 3 \\ 7 & 8 & 5 \end{bmatrix}$ whose rows will then be concatenated to $[9 \ 6 \ 3 \ 7 \ 8 \ 5]$

The k-max pooling operation makes it possible to pool the k most active indicators that may be a number of positions apart; it preserves the order of the features, but is insensitive to their specific positions. It can also discern more finely the number of times the feature is highly activated (Kalchbrenner et al., 2014).

9.3 Variations

Rather than a single convolutional layer, several convolutional layers may be applied in parallel. For example, we may have four different convolutional layers, each with a different window size in the range 2–5, capturing n-gram sequences of varying lengths. The result of each convolutional layer will then be pooled, and the resulting vectors concatenated and fed to further processing (Kim, 2014).

The convolutional architecture need not be restricted into the linear ordering of a sentence. For example, Ma et al (2015) generalize the convolution operation to work over syntactic dependency trees. There, each window is around a node in the syntactic tree, and the pooling is performed over the different nodes. Similarly, Liu et al (2015) apply a convolutional architecture on top of dependency paths extracted from dependency trees. Le and Zuidema (2015) propose to perform max pooling over vectors representing the different derivations leading to the same chart item in a chart parser.

10. Recurrent Neural Networks – Modeling Sequences and Stacks

When dealing with language data, it is very common to work with sequences, such as words (sequences of letters), sentences (sequences of words) and documents. We saw how feed-forward networks can accommodate arbitrary feature functions over sequences through the use of vector concatenation and vector addition (CBOW). In particular, the CBOW representations allows to encode arbitrary length sequences as fixed sized vectors. However, the CBOW representation is quite limited, and forces one to disregard the order of features. The convolutional networks also allow encoding a sequence into a fixed size vector. While representations derived from convolutional networks are an improvement above the CBOW representation as they offer some sensitivity to word order, their order sensitivity is restricted to mostly local patterns, and disregards the order of patterns that are far apart in the sequence.

Recurrent neural networks (RNNs) (Elman, 1990) allow representing arbitrarily sized structured inputs in a fixed-size vector, while paying attention to the structured properties of the input.

10.1 The RNN Abstraction

We use $\mathbf{x}_{i:j}$ to denote the sequence of vectors $\mathbf{x}_i, \dots, \mathbf{x}_j$. The RNN abstraction takes as input an ordered list of input vectors $\mathbf{x}_1, \dots, \mathbf{x}_n$ together with an initial *state vector* \mathbf{s}_0 , and returns an ordered list of state vectors $\mathbf{s}_1, \dots, \mathbf{s}_n$, as well as an ordered list of *output vectors* $\mathbf{y}_1, \dots, \mathbf{y}_n$. An output vector \mathbf{y}_i is a function of the corresponding state vector \mathbf{s}_i . The input vectors \mathbf{x}_i are presented to the RNN in a sequential fashion, and the state vector \mathbf{s}_i and output vector \mathbf{y}_i represent the state of the RNN after observing the inputs $\mathbf{x}_{1:i}$. The output vector \mathbf{y}_i is then used for further prediction. For example, a model for predicting the conditional probability of an event e given the sequence $\mathbf{m}_{1:i}$ can be defined as $p(e = j | \mathbf{x}_{1:i}) = \text{softmax}(\mathbf{y}_i \mathbf{W} + \mathbf{b})[j]$. The RNN model provides a framework for conditioning on the entire history $\mathbf{x}_1, \dots, \mathbf{x}_i$ without resorting to the Markov assumption which is traditionally used for modeling sequences. Indeed, RNN-based language models result in very good perplexity scores when compared to n-gram based models.

Mathematically, we have a recursively defined function R that takes as input a state vector \mathbf{s}_i and an input vector \mathbf{x}_{i+1} , and results in a new state vector \mathbf{s}_{i+1} . An additional function O is used to map a state vector \mathbf{s}_i to an output vector \mathbf{y}_i . When constructing an RNN, much like when constructing a feed-forward network, one has to specify the dimension of the inputs \mathbf{x}_i as well as the dimensions of the outputs \mathbf{y}_i . The dimensions of the states \mathbf{s}_i are a function of the output dimension.²⁶

26. While RNN architectures in which the state dimension is independent of the output dimension are possible, the current popular architectures, including the Simple RNN, the LSTM and the GRU do not follow this flexibility.

$$\begin{aligned}
RNN(\mathbf{s}_0, \mathbf{x}_{1:n}) &= \mathbf{s}_{1:n}, \mathbf{y}_{1:n} \\
\mathbf{s}_i &= R(\mathbf{s}_{i-1}, \mathbf{x}_i) \\
\mathbf{y}_i &= O(\mathbf{s}_i)
\end{aligned}$$

$$\mathbf{x}_i \in \mathbb{R}^{d_{in}}, \quad \mathbf{y}_i \in \mathbb{R}^{d_{out}}, \quad \mathbf{s}_i \in \mathbb{R}^{f(d_{out})}$$

The functions R and O are the same across the sequence positions, but the RNN keeps track of the states of computation through the state vector that is kept and being passed between invocations of R .

Graphically, the RNN has been traditionally presented as in Figure 5.

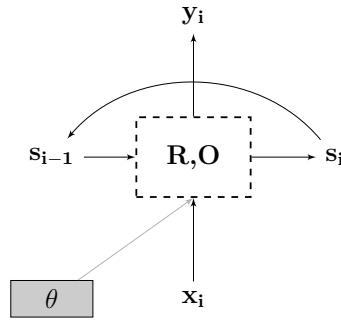


Figure 5: Graphical representation of an RNN (recursive).

This presentation follows the recursive definition, and is correct for arbitrary long sequences. However, for a finite sized input sequence (and all input sequences we deal with are finite) one can *unroll* the recursion, resulting in the structure in Figure 6.

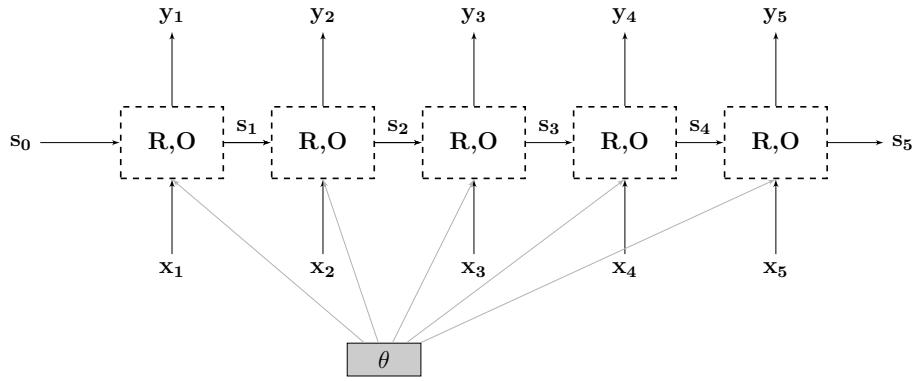


Figure 6: Graphical representation of an RNN (unrolled).

While not usually shown in the visualization, we include here the parameters θ in order to highlight the fact that the same parameters are shared across all time steps. Different

instantiations of R and O will result in different network structures, and will exhibit different properties in terms of their running times and their ability to be trained effectively using gradient-based methods. However, they all adhere to the same abstract interface. We will provide details of concrete instantiations of R and O – the Simple RNN, the LSTM and the GRU – in Section 11. Before that, let’s consider modeling with the RNN abstraction.

First, we note that the value of s_i is based on the entire input x_1, \dots, x_i . For example, by expanding the recursion for $i = 4$ we get:

$$\begin{aligned} s_4 &= R(s_3, x_4) \\ &= R(\overbrace{R(s_2, x_3)}^{s_3}, x_4) \\ &= R(R(\overbrace{R(s_1, x_2)}^{s_2}), x_3), x_4) \\ &= R(R(R(\overbrace{R(s_0, x_1)}^{s_1}), x_2), x_3), x_4) \end{aligned}$$

Thus, s_n (as well as y_n) could be thought of as *encoding* the entire input sequence.²⁷ Is the encoding useful? This depends on our definition of usefulness. The job of the network training is to set the parameters of R and O such that the state conveys useful information for the task we are trying to solve.

10.2 RNN Training

Viewed as in Figure 6 it is easy to see that an unrolled RNN is just a very deep neural network (or rather, a very large *computation graph* with somewhat complex nodes), in which the same parameters are shared across many parts of the computation. To train an RNN network, then, all we need to do is to create the unrolled computation graph for a given input sequence, add a loss node to the unrolled graph, and then use the backward (backpropagation) algorithm to compute the gradients with respect to that loss. This procedure is referred to in the RNN literature as *backpropagation through time*, or BPTT (Werbos, 1990).²⁸ There are various ways in which the supervision signal can be applied.

Acceptor One option is to base the supervision signal only on the final output vector, y_n . Viewed this way, the RNN is an *acceptor*. We observe the final state, and then decide

-
- 27. Note that, unless R is specifically designed against this, it is likely that the later elements of the input sequence have stronger effect on s_n than earlier ones.
 - 28. Variants of the BPTT algorithm include unrolling the RNN only for a fixed number of input symbols at each time: first unroll the RNN for inputs $x_{1:k}$, resulting in $s_{1:k}$. Compute a loss, and backpropagate the error through the network (k steps back). Then, unroll the inputs $x_{k+1:2k}$, this time using s_k as the initial state, and again backpropagate the error for k steps, and so on. This strategy is based on the observations that for the Simple-RNN variant, the gradients after k steps tend to vanish (for large enough k), and so omitting them is negligible. This procedure allows training of arbitrarily long sequences. For RNN variants such as the LSTM or the GRU that are designed specifically to mitigate the vanishing gradients problem, this fixed size unrolling is less motivated, yet it is still being used, for example when doing language modeling over a book without breaking it into sentences.

on an outcome.²⁹ For example, consider training an RNN to read the characters of a word one by one and then use the final state to predict the part-of-speech of that word (this is inspired by (Ling et al., 2015b)), an RNN that reads in a sentence and, based on the final state decides if it conveys positive or negative sentiment (this is inspired by (Wang et al., 2015b)) or an RNN that reads in a sequence of words and decides whether it is a valid noun-phrase. The loss in such cases is defined in terms of a function of $y_n = O(s_n)$, and the error gradients will backpropagate through the rest of the sequence (see Figure 7).³⁰ The loss can take any familiar form – cross entropy, hinge, margin, etc.

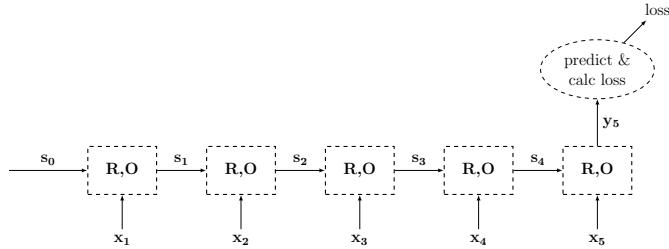


Figure 7: Acceptor RNN Training Graph.

Encoder Similar to the acceptor case, an encoder supervision uses only the final output vector, y_n . However, unlike the acceptor, where a prediction is made solely on the basis of the final vector, here the final vector is treated as an encoding of the information in the sequence, and is used as additional information together with other signals. For example, an extractive document summarization system may first run over the document with an RNN, resulting in a vector y_n summarizing the entire document. Then, y_n will be used together with other features in order to select the sentences to be included in the summarization.

Transducer Another option is to treat the RNN as a transducer, producing an output for each input it reads in. Modeled this way, we can compute a local loss signal $L_{local}(\hat{y}_i, y_i)$ for each of the outputs \hat{y}_i based on a true label y_i . The loss for unrolled sequence will then be: $L(\hat{y}_{1:n}, y_{1:n}) = \sum_{i=1}^n L_{local}(\hat{y}_i, y_i)$, or using another combination rather than a sum such as an average or a weighted average (see Figure 8). One example for such a transducer is a sequence tagger, in which we take $x_{1:n}$ to be feature representations for the n words of a sentence, and y_i as an input for predicting the tag assignment of word i based on words $1:i$. A CCG super-tagger based on such an architecture provides state-of-the art CCG super-tagging results (Xu et al., 2015).

A very natural use-case of the transduction setup is for language modeling, in which the sequence of words $x_{1:i}$ is used to predict a distribution over the $i+1$ th word. RNN based

-
- 29. The terminology is borrowed from Finite-State Acceptors. However, the RNN has a potentially infinite number of states, making it necessary to rely on a function other than a lookup table for mapping states to decisions.
 - 30. This kind of supervision signal may be hard to train for long sequences, especially so with the Simple-RNN, because of the vanishing gradients problem. It is also a generally hard learning task, as we do not tell the process on which parts of the input to focus.

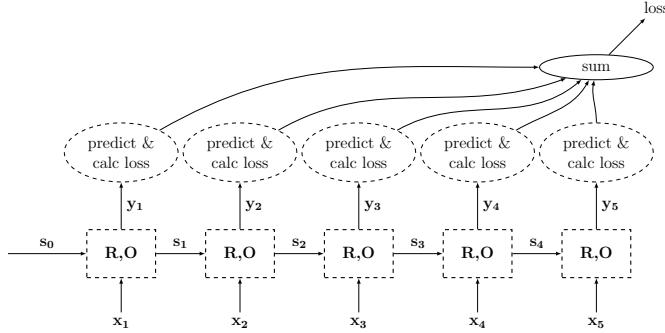


Figure 8: Transducer RNN Training Graph.

language models are shown to provide much better perplexities than traditional language models (Mikolov et al., 2010; Sundermeyer, Schlüter, & Ney, 2012; Mikolov, 2012).

Using RNNs as transducers allows us to relax the Markov assumption that is traditionally taken in language models and HMM taggers, and condition on the entire prediction history. The power of the ability to condition on arbitrarily long histories is demonstrated in generative character-level RNN models, in which a text is generated character by character, each character conditioning on the previous ones (Sutskever, Martens, & Hinton, 2011). The generated texts show sensitivity to properties that are not captured by n-gram language models, including line lengths and nested parenthesis balancing. For a good demonstration and analysis of the properties of RNN-based character level language models, see (Karpathy, Johnson, & Li, 2015).

Encoder - Decoder Finally, an important special case of the encoder scenario is the Encoder-Decoder framework (Cho, van Merriënboer, Bahdanau, & Bengio, 2014a; Sutskever et al., 2014). The RNN is used to encode the sequence into a vector representation y_n , and this vector representation is then used as auxiliary input to another RNN that is used as a decoder. For example, in a machine-translation setup the first RNN encodes the source sentence into a vector representation y_n , and then this state vector is fed into a separate (decoder) RNN that is trained to predict (using a transducer-like language modeling objective) the words of the target language sentence based on the previously predicted words as well as y_n . The supervision happens only for the decoder RNN, but the gradients are propagated all the way back to the encoder RNN (see Figure 9).

Such an approach was shown to be surprisingly effective for Machine Translation (Sutskever et al., 2014) using LSTM RNNs. In order for this technique to work, Sutskever et al found it effective to input the source sentence in reverse, such that x_n corresponds to the first word of the sentence. In this way, it is easier for the second RNN to establish the relation between the first word of the source sentence to the first word of the target sentence. Another use-case of the encoder-decoder framework is for sequence transduction. Here, in order to generate tags t_1, \dots, t_n , an encoder RNN is first used to encode the sentence $x_{1:n}$ into fixed sized vector. This vector is then fed as the initial state vector of another (transducer) RNN, which is used together with $x_{1:n}$ to predict the label t_i at each position i . This approach

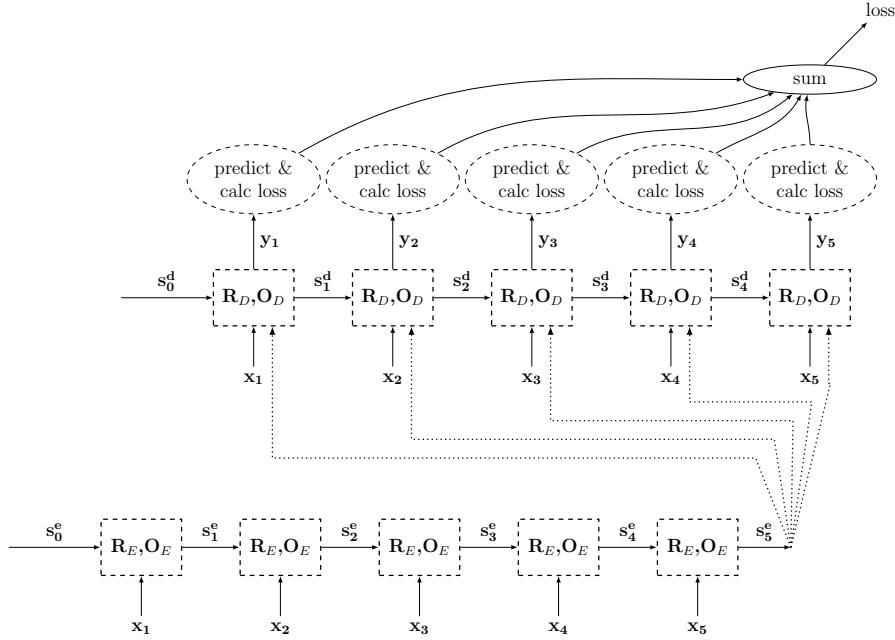


Figure 9: Encoder-Decoder RNN Training Graph.

was used in (Filippova, Alfonseca, Colmenares, Kaiser, & Vinyals, 2015) to model sentence compression by deletion.

10.3 Multi-layer (stacked) RNNs

RNNs can be stacked in layers, forming a grid (Hihi & Bengio, 1996). Consider k RNNs, RNN_1, \dots, RNN_k , where the j th RNN has states $s_{1:n}^j$ and outputs $y_{1:n}^j$. The input for the first RNN are $x_{1:n}$, while the input of the j th RNN ($j \geq 2$) are the outputs of the RNN below it, $y_{1:n}^{j-1}$. The output of the entire formation is the output of the last RNN, $y_{1:n}^k$. Such layered architectures are often called *deep RNNs*. A visual representation of a 3-layer RNN is given in Figure 10.

While it is not theoretically clear what is the additional power gained by the deeper architecture, it was observed empirically that deep RNNs work better than shallower ones on some tasks. In particular, Sutskever et al (2014) report that a 4-layers deep architecture was crucial in achieving good machine-translation performance in an encoder-decoder framework. Irsoy and Cardie (2014) also report improved results from moving from a one-layer BI-RNN to an architecture with several layers. Many other works report result using layered RNN architectures, but do not explicitly compare to 1-layer RNNs.

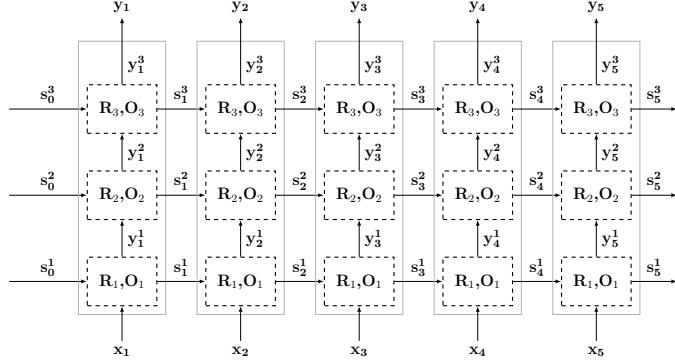


Figure 10: A 3-layer (“deep”) RNN architecture.

10.4 BI-RNN

A useful elaboration of an RNN is a *bidirectional-RNN* (BI-RNN) (Schuster & Paliwal, 1997; Graves, 2008).³¹ Consider the task of sequence tagging over a sentence x_1, \dots, x_n . An RNN allows us to compute a function of the i th word x_i based on the past – the words $x_{1:i}$ up to and including it. However, the following words $x_{i:n}$ may also be useful for prediction, as is evident by the common sliding-window approach in which the focus word is categorized based on a window of k words surrounding it. Much like the RNN relaxes the Markov assumption and allows looking arbitrarily back into the past, the BI-RNN relaxes the fixed window size assumption, allowing to look arbitrarily far at both the past and the future.

Consider an input sequence $\mathbf{x}_{1:n}$. The BI-RNN works by maintaining two separate states, \mathbf{s}_i^f and \mathbf{s}_i^b for each input position i . The *forward state* \mathbf{s}_i^f is based on $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_i$, while the *backward state* \mathbf{s}_i^b is based on $\mathbf{x}_n, \mathbf{x}_{n-1}, \dots, \mathbf{x}_i$. The forward and backward states are generated by two different RNNs. The first RNN (R^f, O^f) is fed the input sequence $\mathbf{x}_{1:n}$ as is, while the second RNN (R^b, O^b) is fed the input sequence in reverse. The state representation \mathbf{s}_i is then composed of both the forward and backward states.

The output at position i is based on the concatenation of the two output vectors $\mathbf{y}_i = [\mathbf{y}_i^f; \mathbf{y}_i^b] = [O^f(\mathbf{s}_i^f); O^b(\mathbf{s}_i^b)]$, taking into account both the past and the future. The vector \mathbf{y}_i can then be used directly for prediction, or fed as part of the input to a more complex network. While the two RNNs are run independently of each other, the error gradients at position i will flow both forward and backward through the two RNNs. A visual representation of the BI-RNN architecture is given in Figure 11.

The use of BI-RNNs for sequence tagging was introduced to the NLP community by Irsoy and Cardie (2014).

10.5 RNNs for Representing Stacks

Some algorithms in language processing, including those for transition-based parsing (Nivre, 2008), require performing feature extraction over a stack. Instead of being confined to

31. When used with a specific RNN architecture such as an LSTM, the model is called BI-LSTM.

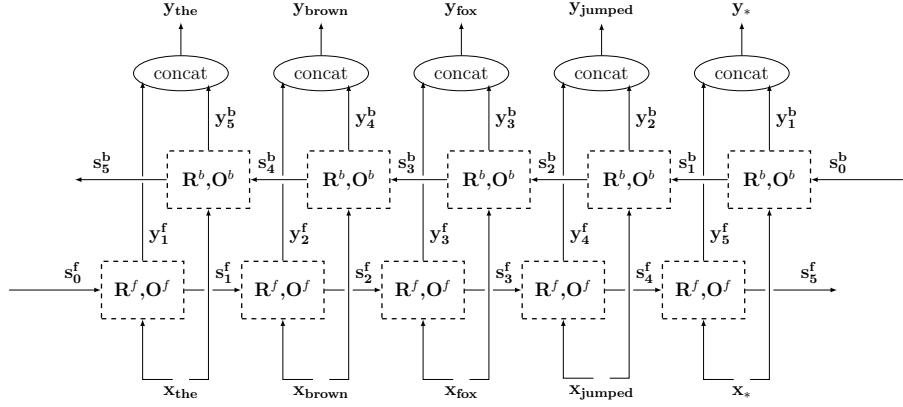


Figure 11: BI-RNN over the sentence “the brown fox jumped .”.

looking at the k top-most elements of the stack, the RNN framework can be used to provide a fixed-sized vector encoding of the entire stack.

The main intuition is that a stack is essentially a sequence, and so the stack state can be represented by taking the stack elements and feeding them in order into an RNN, resulting in a final encoding of the entire stack. In order to do this computation efficiently (without performing an $O(n)$ stack encoding operation each time the stack changes), the RNN state is maintained together with the stack state. If the stack was push-only, this would be trivial: whenever a new element x is pushed into the stack, the corresponding vector \mathbf{x} will be used together with the RNN state \mathbf{s}_i in order to obtain a new state \mathbf{s}_{i+1} . Dealing with pop operation is more challenging, but can be solved by using the persistent-stack data-structure (Okasaki, 1999; Goldberg, Zhao, & Huang, 2013). Persistent, or immutable, data-structures keep old versions of themselves intact when modified. The persistent stack construction represents a stack as a pointer to the head of a linked list. An empty stack is the empty list. The push operation appends an element to the list, returning the new head. The pop operation then returns the parent of the head, but keeping the original list intact. From the point of view of someone who held a pointer to the previous head, the stack did not change. A subsequent push operation will add a new child to the same node. Applying this procedure throughout the lifetime of the stack results in a tree, where the root is an empty stack and each path from a node to the root represents an intermediary stack state. Figure 12 provides an example of such a tree. The same process can be applied in the computation graph construction, creating an RNN with a tree structure instead of a chain structure. Backpropagating the error from a given node will then affect all the elements that participated in the stack when the node was created, in order. Figure 13 shows the computation graph for the stack-RNN corresponding to the last state in Figure 12. This modeling approach was proposed independently by Dyer et al and Watanabe et al (Dyer et al., 2015; Watanabe & Sumita, 2015) for transition-based dependency parsing.

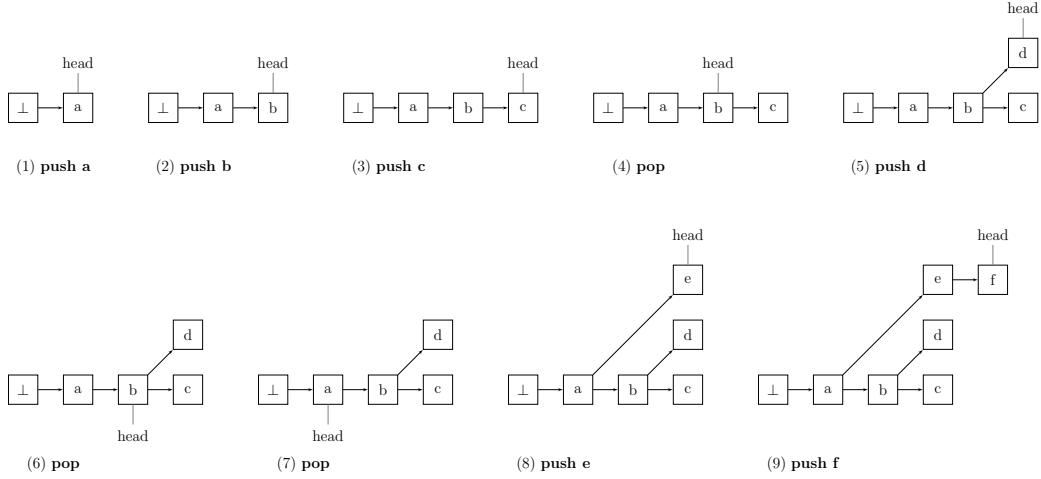


Figure 12: An immutable stack construction for the sequence of operations *push a; push b; push c; pop; push d; pop; pop; push e; push f.*

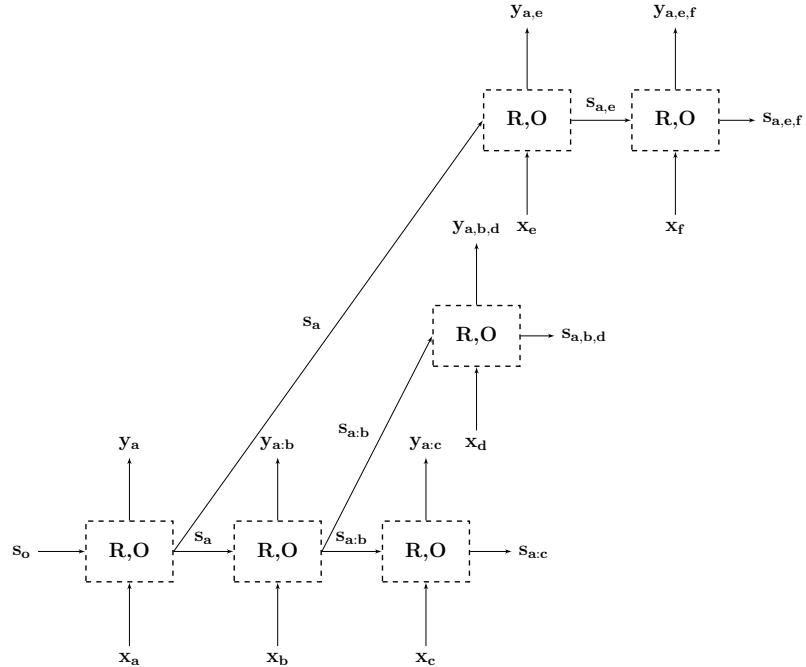


Figure 13: The stack-RNN corresponding to the final state in Figure 12.

11. Concrete RNN Architectures

We now turn to present three different instantiations of the abstract *RNN* architecture discussed in the previous section, providing concrete definitions of the functions R and O . These are the *Simple RNN* (SRNN), the *Long Short-Term Memory* (LSTM) and the *Gated Recurrent Unit* (GRU).

11.1 Simple RNN

The simplest RNN formulation, known as an Elman Network or Simple-RNN (S-RNN), was proposed by Elman (1990) and explored for use in language modeling by Mikolov (2012). The S-RNN takes the following form:

$$\mathbf{s}_i = R_{SRNN}(\mathbf{s}_{i-1}, \mathbf{x}_i) = g(\mathbf{x}_i \mathbf{W}^x + \mathbf{s}_{i-1} \mathbf{W}^s + \mathbf{b})$$
$$\mathbf{y}_i = O_{SRNN}(\mathbf{s}_i) = \mathbf{s}_i$$

$$\mathbf{s}_i, \mathbf{y}_i \in \mathbb{R}^{d_s}, \quad \mathbf{x}_i \in \mathbb{R}^{d_x}, \quad \mathbf{W}^x \in \mathbb{R}^{d_x \times d_s}, \quad \mathbf{W}^s \in \mathbb{R}^{d_s \times d_s}, \quad \mathbf{b} \in \mathbb{R}^{d_s}$$

That is, the state at position i is a linear combination of the input at position i and the previous state, passed through a non-linear activation (commonly tanh or ReLU). The output at position i is the same as the hidden state in that position.³²

In spite of its simplicity, the Simple RNN provides strong results for sequence tagging (Xu et al., 2015) as well as language modeling. For comprehensive discussion on using Simple RNNs for language modeling, see the PhD thesis by Mikolov (2012).

11.2 LSTM

The S-RNN is hard to train effectively because of the vanishing gradients problem. Error signals (gradients) in later steps in the sequence diminish in quickly in the back-propagation process, and do not reach earlier input signals, making it hard for the S-RNN to capture long-range dependencies. The Long Short-Term Memory (LSTM) architecture (Hochreiter & Schmidhuber, 1997) was designed to solve the vanishing gradients problem. The main idea behind the LSTM is to introduce as part of the state representation also “memory cells” (a vector) that can preserve gradients across time. Access to the memory cells is controlled by *gating components* – smooth mathematical functions that simulate logical gates. At each input state, a gate is used to decide how much of the new input should be written to the memory cell, and how much of the current content of the memory cell should be forgotten. Concretely, a gate $\mathbf{g} \in [0, 1]^n$ is a vector of values in the range $[0, 1]$ that is multiplied component-wise with another vector $\mathbf{v} \in \mathbb{R}^n$, and the result is then added to another vector. The values of \mathbf{g} are designed to be close to either 0 or 1, i.e. by using a sigmoid function. Indices in \mathbf{v} corresponding to near-one values in \mathbf{g} are allowed to pass, while those corresponding to near-zero values are blocked.

32. Some authors treat the output at position i as a more complicated function of the state. In our presentation, such further transformation of the output are not considered part of the RNN, but as separate computations that are applied to the RNNs output. The distinction between the state and the output are needed for the LSTM architecture, in which not all of the state is observed outside of the RNN.

Mathematically, the LSTM architecture is defined as:³³

$$\begin{aligned}
\mathbf{s}_j &= R_{LSTM}(\mathbf{s}_{j-1}, \mathbf{x}_j) = [\mathbf{c}_j; \mathbf{h}_j] \\
\mathbf{c}_j &= \mathbf{c}_{j-1} \odot \mathbf{f} + \mathbf{g} \odot \mathbf{i} \\
\mathbf{h}_j &= \tanh(\mathbf{c}_j) \odot \mathbf{o} \\
\mathbf{i} &= \sigma(\mathbf{x}_j \mathbf{W}^{\mathbf{x}\mathbf{i}} + \mathbf{h}_{j-1} \mathbf{W}^{\mathbf{h}\mathbf{i}}) \\
\mathbf{f} &= \sigma(\mathbf{x}_j \mathbf{W}^{\mathbf{x}\mathbf{f}} + \mathbf{h}_{j-1} \mathbf{W}^{\mathbf{h}\mathbf{f}}) \\
\mathbf{o} &= \sigma(\mathbf{x}_j \mathbf{W}^{\mathbf{x}\mathbf{o}} + \mathbf{h}_{j-1} \mathbf{W}^{\mathbf{h}\mathbf{o}}) \\
\mathbf{g} &= \tanh(\mathbf{x}_j \mathbf{W}^{\mathbf{x}\mathbf{g}} + \mathbf{h}_{j-1} \mathbf{W}^{\mathbf{h}\mathbf{g}})
\end{aligned}$$

$$\mathbf{y}_j = O_{LSTM}(\mathbf{s}_j) = \mathbf{h}_j$$

$$\mathbf{s}_j \in \mathbb{R}^{2 \cdot d_h}, \quad \mathbf{x}_i \in \mathbb{R}^{d_x}, \quad \mathbf{c}_j, \mathbf{h}_j, \mathbf{i}, \mathbf{f}, \mathbf{o}, \mathbf{g} \in \mathbb{R}^{d_h}, \quad \mathbf{W}^{\mathbf{x}\mathbf{o}} \in \mathbb{R}^{d_x \times d_h}, \quad \mathbf{W}^{\mathbf{h}\mathbf{o}} \in \mathbb{R}^{d_h \times d_h},$$

The symbol \odot is used to denote component-wise product. The state at time j is composed of two vectors, \mathbf{c}_j and \mathbf{h}_j , where \mathbf{c}_j is the memory component and \mathbf{h}_j is the output, or state, component. There are three gates, \mathbf{i} , \mathbf{f} and \mathbf{o} , controlling for input, forget and output. The gate values are computed based on linear combinations of the current input \mathbf{x}_j and the previous state \mathbf{h}_{j-1} , passed through a sigmoid activation function. An update candidate \mathbf{g} is computed as a linear combination of \mathbf{x}_j and \mathbf{h}_{j-1} , passed through a tanh activation function. The memory \mathbf{c}_j is then updated: the forget gate controls how much of the previous memory to keep ($\mathbf{c}_{j-1} \odot \mathbf{f}$), and the input gate controls how much of the proposed update to keep ($\mathbf{g} \odot \mathbf{i}$). Finally, the value of \mathbf{h}_j (which is also the output \mathbf{y}_j) is determined based on the content of the memory \mathbf{c}_j , passed through a tanh non-linearity and controlled by the output gate. The gating mechanisms allow for gradients related to the memory part \mathbf{c}_j to stay high across very long time ranges.

For further discussion on the LSTM architecture see the PhD thesis by Alex Graves (2008), as well as Chris Olah's description.³⁴ For an analysis of the behavior of an LSTM when used as a character-level language model, see (Karpathy et al., 2015).

LSTMs are currently the most successful type of RNN architecture, and they are responsible for many state-of-the-art sequence modeling results. The main competitor of the LSTM-RNN is the GRU, to be discussed next.

Practical Considerations When training LSTM networks, Jozefowicz et al (2015) strongly recommend to always initialize the bias term of the forget gate to be close to one. When applying dropout to an RNN with an LSTM, Zaremba et al (2014) found out that it is

33. There are many variants on the LSTM architecture presented here. For example, forget gates were not part of the original proposal in (Hochreiter & Schmidhuber, 1997), but are shown to be an important part of the architecture. Other variants include peephole connections and gate-tying. For an overview and comprehensive empirical comparison of various LSTM architectures see (Greff, Srivastava, Koutník, Steunebrink, & Schmidhuber, 2015).

34. <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

crucial to apply dropout only on the non-recurrent connection, i.e. only to apply it between layers and not between sequence positions.

11.3 GRU

The LSTM architecture is very effective, but also quite complicated. The complexity of the system makes it hard to analyze, and also computationally expensive to work with. The gated recurrent unit (GRU) was recently introduced by Cho et al (2014b) as an alternative to the LSTM. It was subsequently shown by Chung et al (2014) to perform comparably to the LSTM on several (non textual) datasets.

Like the LSTM, the GRU is also based on a gating mechanism, but with substantially fewer gates and without a separate memory component.

$$\begin{aligned} \mathbf{s}_j &= R_{GRU}(\mathbf{s}_{j-1}, \mathbf{x}_j) = (\mathbf{1} - \mathbf{z}) \odot \mathbf{s}_{j-1} + \mathbf{z} \odot \mathbf{h} \\ \mathbf{z} &= \sigma(\mathbf{x}_j \mathbf{W}^{xz} + \mathbf{h}_{j-1} \mathbf{W}^{hz}) \\ \mathbf{r} &= \sigma(\mathbf{x}_j \mathbf{W}^{xr} + \mathbf{h}_{j-1} \mathbf{W}^{hr}) \\ \mathbf{h} &= \tanh(\mathbf{x}_j \mathbf{W}^{xh} + (\mathbf{h}_{j-1} \odot \mathbf{r}) \mathbf{W}^{hg}) \\ \mathbf{y}_j &= O_{LSTM}(\mathbf{s}_j) = \mathbf{s}_j \end{aligned}$$

$$\mathbf{s}_j \in \mathbb{R}^{d_h}, \quad \mathbf{x}_i \in \mathbb{R}^{d_x}, \quad \mathbf{z}, \mathbf{r}, \mathbf{h} \in \mathbb{R}^{d_h}, \quad \mathbf{W}^{xo} \in \mathbb{R}^{d_x \times d_h}, \quad \mathbf{W}^{ho} \in \mathbb{R}^{d_h \times d_h},$$

One gate (\mathbf{r}) is used to control access to the previous state \mathbf{s}_{j-1} and compute a proposed update \mathbf{h} . The updated state \mathbf{s}_j (which also serves as the output \mathbf{y}_j) is then determined based on an interpolation of the previous state \mathbf{s}_{j-1} and the proposal \mathbf{h} , where the proportions of the interpolation are controlled using the gate \mathbf{z} .

The GRU was shown to be effective in language modeling and machine translation. However, the jury between the GRU, the LSTM and possible alternative RNN architectures is still out, and the subject is actively researched. For an empirical exploration of the GRU and the LSTM architectures, see (Jozefowicz et al., 2015).

11.4 Other Variants

The gated architectures of the LSTM and the GRU help in alleviating the vanishing gradients problem of the Simple RNN, and allow these RNNs to capture dependencies that span long time ranges. Some researchers explore simpler architectures than the LSTM and the GRU for achieving similar benefits.

Mikolov et al (2014) observed that the matrix multiplication $\mathbf{s}_{i-1} \mathbf{W}^s$ coupled with the nonlinearity g in the update rule R of the Simple RNN causes the state vector \mathbf{s}_i to undergo large changes at each time step, prohibiting it from remembering information over long time periods. They propose to split the state vector \mathbf{s}_i into a slow changing component \mathbf{c}_i (“context units”) and a fast changing component \mathbf{h}_i .³⁵ The slow changing component \mathbf{c}_i is

³⁵. We depart from the notation in (Mikolov et al., 2014) and reuse the symbols used in the LSTM description.

updated according to a linear interpolation of the input and the previous component: $\mathbf{c}_i = (1 - \alpha)\mathbf{x}_i\mathbf{W}^{\mathbf{x1}} + \alpha\mathbf{c}_{i-1}$, where $\alpha \in (0, 1)$. This update allows \mathbf{c}_i to accumulate the previous inputs. The fast changing component \mathbf{h}_i is updated similarly to the Simple RNN update rule, but changed to take \mathbf{c}_i into account as well:³⁶ $\mathbf{h}_i = \sigma(\mathbf{x}_i\mathbf{W}^{\mathbf{x2}} + \mathbf{h}_{i-1}\mathbf{W}^h + \mathbf{c}_i\mathbf{W}^c)$. Finally, the output \mathbf{y}_i is the concatenation of the slow and the fast changing parts of the state: $\mathbf{y}_i = [\mathbf{c}_i; \mathbf{h}_i]$. Mikolov et al demonstrate that this architecture provides competitive perplexities to the much more complex LSTM on language modeling tasks.

The approach of Mikolov et al can be interpreted as constraining the block of the matrix \mathbf{W}^s in the S-RNN corresponding to \mathbf{c}_i to be a multiply of the identity matrix (see Mikolov et al (2014) for the details). Le et al (Le, Jaitly, & Hinton, 2015) propose an even simpler approach: set the activation function of the S-RNN to a ReLU, and initialize the biases \mathbf{b} as zeroes and the matrix \mathbf{W}^s as the identify matrix. This causes an untrained RNN to copy the previous state to the current state, add the effect of the current input \mathbf{x}_i and set the negative values to zero. After setting this initial bias towards state copying, the training procedure allows \mathbf{W}^s to change freely. Le et al demonstrate that this simple modification makes the S-RNN comparable to an LSTM with the same number of parameters on several tasks, including language modeling.

36. The update rule diverges from the S-RNN update rule also by fixing the non-linearity to be a sigmoid function, and by not using a bias term. However, these changes are not discussed as central to the proposal.

12. Modeling Trees – Recursive Neural Networks

The RNN is very useful for modeling sequences. In language processing, it is often natural and desirable to work with tree structures. The trees can be syntactic trees, discourse trees, or even trees representing the sentiment expressed by various parts of a sentence (Socher et al., 2013). We may want to predict values based on specific tree nodes, predict values based on the root nodes, or assign a quality score to a complete tree or part of a tree. In other cases, we may not care about the tree structure directly but rather reason about spans in the sentence. In such cases, the tree is merely used as a backbone structure which help guide the encoding process of the sequence into a fixed size vector.

The *recursive neural network* (RecNN) abstraction (Pollack, 1990), popularized in NLP by Richard Socher and colleagues (Socher, Manning, & Ng, 2010; Socher, Lin, Ng, & Manning, 2011; Socher et al., 2013; Socher, 2014) is a generalization of the RNN from sequences to (binary) trees.³⁷

Much like the RNN encodes each sentence prefix as a state vector, the RecNN encodes each tree-node as a state vector in \mathbb{R}^d . We can then use these state vectors either to predict values of the corresponding nodes, assign quality values to each node, or as a semantic representation of the spans rooted at the nodes.

The main intuition behind the recursive neural networks is that each subtree is represented as a d dimensional vector, and the representation of a node p with children c_1 and c_2 is a function of the representation of the nodes: $\text{vec}(p) = f(\text{vec}(c_1), \text{vec}(c_2))$, where f is a composition function taking two d -dimensional vectors and returning a single d -dimensional vector. Much like the RNN state \mathbf{s}_i is used to encode the entire sequence $\mathbf{x}_1 : \mathbf{i}$, the RecNN state associated with a tree node p encodes the entire subtree rooted at p . See Figure 14 for an illustration.

12.1 Formal Definition

Consider a binary parse tree \mathcal{T} over an n -word sentence. As a reminder, an ordered, unlabeled tree over a string x_1, \dots, x_n can be represented as a unique set of triplets (i, k, j) , s.t. $i \leq k \leq j$. Each such triplet indicates that a node spanning words $x_{i:j}$ is parent of the nodes spanning $x_{i:k}$ and $x_{k+1:j}$. Triplets of the form (i, i, i) correspond to terminal symbols at the tree leaves (the words x_i). Moving from the unlabeled case to the labeled one, we can represent a tree as a set of 6-tuples $(A \rightarrow B, C, i, k, j)$, whereas i, k and j indicate the spans as before, and A, B and C are the node labels of the nodes spanning $x_{i:j}, x_{i:k}$ and $x_{k+1:j}$ respectively. Here, leaf nodes have the form $(A \rightarrow A, A, i, i, i)$, where A is a pre-terminal symbol. We refer to such tuples as *production rules*. For an example, consider the syntactic tree for the sentence “the boy saw her duck”.

37. While presented in terms of binary parse trees, the concepts easily transfer to general recursively-defined data structures, with the major technical challenge is the definition of an effective form for R , the combination function.

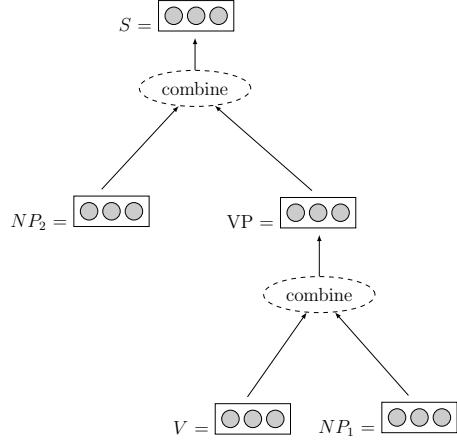
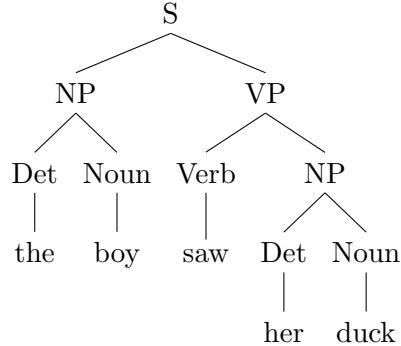


Figure 14: Illustration of a recursive neural network. The representations of V and NP_1 are combined to form the representation of VP . The representations of VP and NP_2 are then combined to form the representation of S .



Its corresponding unlabeled and labeled representations are :

Unlabeled	Labeled	Corresponding Span
(1,1,1)	(Det, Det, Det, 1, 1, 1)	$x_{1:1}$ the
(2,2,2)	(Noun, Noun, Noun, 2, 2, 2)	$x_{2:2}$ boy
(3,3,3)	(Verb, Verb, Verb, 3, 3, 3)	saw
(4,4,4)	(Det, Det, Det, 4, 4, 4)	her
(5,5,5)	(Noun, Noun, Noun, 5, 5, 5)	duck
(4,4,5)	(NP, Det, Noun, 4, 4, 5)	her duck
(3,3,5)	(VP, Verb, NP, 3, 3, 5)	saw her duck
(1,1,2)	(NP, Det, Noun, 1, 1, 2)	the boy
(1,2,5)	(S, NP, VP, 1, 2, 5)	the boy saw her duck

The set of production rules above can be uniquely converted to a set tree nodes $q_{i:j}^A$ (indicating a node with symbol A over the span $x_{i:j}$) by simply ignoring the elements

(B, C, k) in each production rule. We are now in position to define the Recursive Neural Network.

A Recursive Neural Network (RecNN) is a function that takes as input a parse tree over an n -word sentence x_1, \dots, x_n . Each of the sentence's words is represented as a d -dimensional vector \mathbf{x}_i , and the tree is represented as a set \mathcal{T} of production rules ($A \rightarrow B, C, i, j, k$). Denote the nodes of \mathcal{T} by $q_{i:j}^A$. The RecNN returns as output a corresponding set of *inside state vectors* $\mathbf{s}_{i:j}^A$, where each inside state vector $\mathbf{s}_{i:j}^A \in \mathbb{R}^d$ represents the corresponding tree node $q_{i:j}^A$, and encodes the entire structure rooted at that node. Like the sequence RNN, the tree shaped RecNN is defined recursively using a function R , where the inside vector of a given node is defined as a function of the inside vectors of its direct children.³⁸ Formally:

$$\begin{aligned} \text{RecNN}(x_1, \dots, x_n, \mathcal{T}) = & \{\mathbf{s}_{i:j}^A \in \mathbb{R}^d \mid q_{i:j}^A \in \mathcal{T}\} \\ \mathbf{s}_{i:i}^A = & v(x_i) \\ \mathbf{s}_{i:j}^A = & R(A, B, C, \mathbf{s}_{i:k}^B, \mathbf{s}_{k+1:j}^C) \quad q_{i:k}^B \in \mathcal{T}, \quad q_{k+1:j}^C \in \mathcal{T} \end{aligned}$$

The function R usually takes the form of a simple linear transformation, which may or may not be followed by a non-linear activation function g :

$$R(A, B, C, \mathbf{s}_{i:k}^B, \mathbf{s}_{k+1:j}^C) = g([\mathbf{s}_{i:k}^B; \mathbf{s}_{k+1:j}^C] \mathbf{W})$$

This formulation of R ignores the tree labels, using the same matrix $\mathbf{W} \in \mathbb{R}^{2d \times d}$ for all combinations. This may be a useful formulation in case the node labels do not exist (e.g. when the tree does not represent a syntactic structure with clearly defined labels) or when they are unreliable. However, if the labels are available, it is generally useful to include them in the composition function. One approach would be to introduce *label embeddings* $v(A)$ mapping each non-terminal symbol to a d_{nt} dimensional vector, and change R to include the embedded symbols in the combination function:

$$R(A, B, C, \mathbf{s}_{i:k}^B, \mathbf{s}_{k+1:j}^C) = g([\mathbf{s}_{i:k}^B; \mathbf{s}_{k+1:j}^C; v(A); v(B)] \mathbf{W})$$

(here, $\mathbf{W} \in \mathbb{R}^{2d+2d_{nt} \times d}$). Such approach is taken by (Qian, Tian, Huang, Liu, Zhu, & Zhu, 2015). An alternative approach, due to (Socher et al., 2013) is to untie the weights according to the non-terminals, using a different composition matrix for each B, C pair of symbols:³⁹

$$R(A, B, C, \mathbf{s}_{i:k}^B, \mathbf{s}_{k+1:j}^C) = g([\mathbf{s}_{i:k}^B; \mathbf{s}_{k+1:j}^C] \mathbf{W}^{BC})$$

-
38. Le and Zuidema (2014) extend the RecNN definition such that each node has, in addition to its inside state vector, also an *outside state vector* representing the entire structure around the subtree rooted at that node. Their formulation is based on the recursive computation of the classic inside-outside algorithm, and can be thought of as the BI-RNN counterpart of the tree RecNN. For details, see (Le & Zuidema, 2014).
39. While not explored in the literature, a trivial extension would condition the transformation matrix also on A .

This formulation is useful when the number of non-terminal symbols (or the number of possible symbol combinations) is relatively small, as is usually the case with phrase-structure parse trees. A similar model was also used by (Hashimoto et al., 2013) to encode subtrees in semantic-relation classification task.

12.2 Extensions and Variations

As all of the definitions of R above suffer from the vanishing gradients problem of the Simple RNN, several authors sought to replace it with functions inspired by the Long Short-Term Memory (LSTM) gated architecture, resulting in Tree-shaped LSTMs (Tai, Socher, & Manning, 2015; Zhu, Sobhani, & Guo, 2015b). The question of optimal tree representation is still very much an open research question, and the vast space of possible combination functions R is yet to be explored. Other proposed variants on tree-structured RNNs includes a *recursive matrix-vector model* (Socher, Huval, Manning, & Ng, 2012) and *recursive neural tensor network* (Socher et al., 2013). In the first variant, each word is represented as a combination of a vector and a matrix, where the vector defines the word’s static semantic content as before, while the matrix acts as a learned “operator” for the word, allowing more subtle semantic compositions than the addition and weighted averaging implied by the concatenation followed by linear transformation function. In the second variant, words are associated with vectors as usual, but the composition function becomes more expressive by basing it on tensor instead of matrix operations.

12.3 Training Recursive Neural Networks

The training procedure for a recursive neural network follows the same recipe as training other forms of networks: define a loss, spell out the computation graph, compute gradients using backpropagation⁴⁰, and train the parameters using SGD.

With regard to the loss function, similar to the sequence RNN one can associate a loss either with the root of the tree, with any given node, or with a set of nodes, in which case the individual node’s losses are combined, usually by summation. The loss function is based on the labeled training data which associates a label or other quantity with different tree nodes.

Additionally, one can treat the RecNN as an Encoder, whereas the inside-vector associated with a node is taken to be an encoding of the tree rooted at that node. The encoding can potentially be sensitive to arbitrary properties of the structure. The vector is then passed as input to another network.

For further discussion on recursive neural networks and their use in natural language tasks, refer to the PhD thesis of Richard Socher (2014).

40. Before the introduction of the computation graph abstraction, the specific backpropagation procedure for computing the gradients in a RecNN as defined above was referred to as the Back-propagation trough Structure (BPTS) algorithm (Goller & Küchler, 1996).

13. Conclusions

Neural networks are powerful learners, providing opportunities ranging from non-linear classification to non-Markovian modeling of sequences and trees. We hope that this exposition help NLP researchers to incorporate neural network models in their work and take advantage of their power.

References

- Adel, H., Vu, N. T., & Schultz, T. (2013). Combination of Recurrent Neural Networks and Factored Language Models for Code-Switching Language Modeling. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 206–211, Sofia, Bulgaria. Association for Computational Linguistics.
- Ando, R., & Zhang, T. (2005a). A High-Performance Semi-Supervised Learning Method for Text Chunking. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, pp. 1–9, Ann Arbor, Michigan. Association for Computational Linguistics.
- Ando, R. K., & Zhang, T. (2005b). A framework for learning predictive structures from multiple tasks and unlabeled data. *The Journal of Machine Learning Research*, 6, 1817–1853.
- Auli, M., Galley, M., Quirk, C., & Zweig, G. (2013). Joint Language and Translation Modeling with Recurrent Neural Networks. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pp. 1044–1054, Seattle, Washington, USA. Association for Computational Linguistics.
- Auli, M., & Gao, J. (2014). Decoder Integration and Expected BLEU Training for Recurrent Neural Network Language Models. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 136–142, Baltimore, Maryland. Association for Computational Linguistics.
- Ballesteros, M., Dyer, C., & Smith, N. A. (2015). Improved Transition-based Parsing by Modeling Characters instead of Words with LSTMs. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pp. 349–359, Lisbon, Portugal. Association for Computational Linguistics.
- Bansal, M., Gimpel, K., & Livescu, K. (2014). Tailoring Continuous Word Representations for Dependency Parsing. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 809–815, Baltimore, Maryland. Association for Computational Linguistics.
- Baydin, A. G., Pearlmutter, B. A., Radul, A. A., & Siskind, J. M. (2015). Automatic differentiation in machine learning: a survey. *arXiv:1502.05767 [cs]*.
- Bengio, Y. (2012). Practical recommendations for gradient-based training of deep architectures. *arXiv:1206.5533 [cs]*.
- Bengio, Y., Ducharme, R., Vincent, P., & Janvin, C. (2003). A Neural Probabilistic Language Model. *J. Mach. Learn. Res.*, 3, 1137–1155.

- Bengio, Y., Goodfellow, I. J., & Courville, A. (2015). Deep Learning. Book in preparation for MIT Press.
- Bitvai, Z., & Cohn, T. (2015). Non-Linear Text Regression with a Deep Convolutional Neural Network. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pp. 180–185, Beijing, China. Association for Computational Linguistics.
- Botha, J. A., & Blunsom, P. (2014). Compositional Morphology for Word Representations and Language Modelling. In *Proceedings of the 31st International Conference on Machine Learning (ICML)*, Beijing, China. *Award for best application paper*.
- Bottou, L. (2012). Stochastic gradient descent tricks. In *Neural Networks: Tricks of the Trade*, pp. 421–436. Springer.
- Charniak, E., & Johnson, M. (2005). Coarse-to-Fine n-Best Parsing and MaxEnt Discriminative Reranking. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, pp. 173–180, Ann Arbor, Michigan. Association for Computational Linguistics.
- Chen, D., & Manning, C. (2014). A Fast and Accurate Dependency Parser using Neural Networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 740–750, Doha, Qatar. Association for Computational Linguistics.
- Chen, Y., Xu, L., Liu, K., Zeng, D., & Zhao, J. (2015). Event Extraction via Dynamic Multi-Pooling Convolutional Neural Networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 167–176, Beijing, China. Association for Computational Linguistics.
- Cho, K., van Merriënboer, B., Bahdanau, D., & Bengio, Y. (2014a). On the Properties of Neural Machine Translation: Encoder–Decoder Approaches. In *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, pp. 103–111, Doha, Qatar. Association for Computational Linguistics.
- Cho, K., van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014b). Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1724–1734, Doha, Qatar. Association for Computational Linguistics.
- Chrupala, G. (2014). Normalizing tweets with edit scripts and recurrent neural embeddings. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 680–686, Baltimore, Maryland. Association for Computational Linguistics.
- Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *arXiv:1412.3555 [cs]*.
- Collins, M. (2002). Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with Perceptron Algorithms. In *Proceedings of the 2002 Confer-*

- ence on Empirical Methods in Natural Language Processing, pp. 1–8. Association for Computational Linguistics.
- Collins, M., & Koo, T. (2005). Discriminative Reranking for Natural Language Parsing. *Computational Linguistics*, 31(1), 25–70.
- Collobert, R., & Weston, J. (2008). A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pp. 160–167. ACM.
- Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., & Kuksa, P. (2011). Natural language processing (almost) from scratch. *The Journal of Machine Learning Research*, 12, 2493–2537.
- Crammer, K., & Singer, Y. (2002). On the algorithmic implementation of multiclass kernel-based vector machines. *The Journal of Machine Learning Research*, 2, 265–292.
- Creutz, M., & Lagus, K. (2007). Unsupervised Models for Morpheme Segmentation and Morphology Learning. *ACM Trans. Speech Lang. Process.*, 4(1), 3:1–3:34.
- Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4), 303–314.
- Dahl, G., Sainath, T., & Hinton, G. (2013). Improving deep neural networks for LVCSR using rectified linear units and dropout. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 8609–8613.
- de Gispert, A., Iglesias, G., & Byrne, B. (2015). Fast and Accurate Preordering for SMT using Neural Networks. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 1012–1017, Denver, Colorado. Association for Computational Linguistics.
- Dong, L., Wei, F., Tan, C., Tang, D., Zhou, M., & Xu, K. (2014). Adaptive Recursive Neural Network for Target-dependent Twitter Sentiment Classification. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 49–54, Baltimore, Maryland. Association for Computational Linguistics.
- Dong, L., Wei, F., Zhou, M., & Xu, K. (2015). Question Answering over Freebase with Multi-Column Convolutional Neural Networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 260–269, Beijing, China. Association for Computational Linguistics.
- dos Santos, C., & Gatti, M. (2014). Deep Convolutional Neural Networks for Sentiment Analysis of Short Texts. In *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*, pp. 69–78, Dublin, Ireland. Dublin City University and Association for Computational Linguistics.
- dos Santos, C., Xiang, B., & Zhou, B. (2015). Classifying Relations by Ranking with Convolutional Neural Networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 626–634, Beijing, China. Association for Computational Linguistics.

- Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12, 2121–2159.
- Duh, K., Neubig, G., Sudoh, K., & Tsukada, H. (2013). Adaptation Data Selection using Neural Language Models: Experiments in Machine Translation. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 678–683, Sofia, Bulgaria. Association for Computational Linguistics.
- Durrett, G., & Klein, D. (2015). Neural CRF Parsing. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 302–312, Beijing, China. Association for Computational Linguistics.
- Dyer, C., Ballesteros, M., Ling, W., Matthews, A., & Smith, N. A. (2015). Transition-Based Dependency Parsing with Stack Long Short-Term Memory. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 334–343, Beijing, China. Association for Computational Linguistics.
- Elman, J. L. (1990). Finding Structure in Time. *Cognitive Science*, 14(2), 179–211.
- Faruqui, M., & Dyer, C. (2014). Improving Vector Space Word Representations Using Multilingual Correlation. In *Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics*, pp. 462–471, Gothenburg, Sweden. Association for Computational Linguistics.
- Filippova, K., Alfonseca, E., Colmenares, C. A., Kaiser, L., & Vinyals, O. (2015). Sentence Compression by Deletion with LSTMs. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pp. 360–368, Lisbon, Portugal. Association for Computational Linguistics.
- Gal, Y., & Ghahramani, Z. (2015). Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning. *arXiv:1506.02142 [cs, stat]*.
- Gao, J., Pantel, P., Gamon, M., He, X., & Deng, L. (2014). Modeling Interestingness with Deep Neural Networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 2–13, Doha, Qatar. Association for Computational Linguistics.
- Giménez, J., & Màrquez, L. (2004). SVMTool: A general POS tagger generator based on Support Vector Machines. In *Proceedings of the 4th LREC*, Lisbon, Portugal.
- Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *International conference on artificial intelligence and statistics*, pp. 249–256.
- Glorot, X., Bordes, A., & Bengio, Y. (2011). Deep sparse rectifier neural networks. In *International Conference on Artificial Intelligence and Statistics*, pp. 315–323.
- Goldberg, Y., & Elhadad, M. (2010). An Efficient Algorithm for Easy-First Non-Directional Dependency Parsing. In *Human Language Technologies: The 2010 Annual Conference*

- of the North American Chapter of the Association for Computational Linguistics*, pp. 742–750, Los Angeles, California. Association for Computational Linguistics.
- Goldberg, Y., & Levy, O. (2014). word2vec Explained: deriving Mikolov et al.’s negative-sampling word-embedding method. *arXiv:1402.3722 [cs, stat]*.
- Goldberg, Y., & Nivre, J. (2013). Training Deterministic Parsers with Non-Deterministic Oracles. *Transactions of the Association for Computational Linguistics*, 1(0), 403–414.
- Goldberg, Y., Zhao, K., & Huang, L. (2013). Efficient Implementation of Beam-Search Incremental Parsers. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 628–633, Sofia, Bulgaria. Association for Computational Linguistics.
- Goller, C., & Küchler, A. (1996). Learning Task-Dependent Distributed Representations by Backpropagation Through Structure. In *In Proc. of the ICNN-96*, pp. 347–352. IEEE.
- Graves, A. (2008). *Supervised sequence labelling with recurrent neural networks*. Ph.D. thesis, Technische Universität München.
- Greff, K., Srivastava, R. K., Koutný, J., Steunebrink, B. R., & Schmidhuber, J. (2015). LSTM: A Search Space Odyssey. *arXiv:1503.04069 [cs]*.
- Hal Daumé III, Langford, J., & Marcu, D. (2009). Search-based Structured Prediction. *Machine Learning Journal (MLJ)*.
- Harris, Z. (1954). Distributional Structure. *Word*, 10(23), 146–162.
- Hashimoto, K., Miwa, M., Tsuruoka, Y., & Chikayama, T. (2013). Simple Customization of Recursive Neural Networks for Semantic Relation Classification. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pp. 1372–1376, Seattle, Washington, USA. Association for Computational Linguistics.
- He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *arXiv:1502.01852 [cs]*.
- Henderson, M., Thomson, B., & Young, S. (2013). Deep Neural Network Approach for the Dialog State Tracking Challenge. In *Proceedings of the SIGDIAL 2013 Conference*, pp. 467–471, Metz, France. Association for Computational Linguistics.
- Hermann, K. M., & Blunsom, P. (2013). The Role of Syntax in Vector Space Models of Compositional Semantics. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 894–904, Sofia, Bulgaria. Association for Computational Linguistics.
- Hermann, K. M., & Blunsom, P. (2014). Multilingual Models for Compositional Distributed Semantics. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 58–68, Baltimore, Maryland. Association for Computational Linguistics.
- Hihi, S. E., & Bengio, Y. (1996). Hierarchical Recurrent Neural Networks for Long-Term Dependencies. In Touretzky, D. S., Mozer, M. C., & Hasselmo, M. E. (Eds.), *Advances in Neural Information Processing Systems 8*, pp. 493–499. MIT Press.

- Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *arXiv:1207.0580 [cs]*.
- Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8), 1735–1780.
- Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5), 359–366.
- Irsay, O., & Cardie, C. (2014). Opinion Mining with Deep Recurrent Neural Networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 720–728, Doha, Qatar. Association for Computational Linguistics.
- Iyyer, M., Boyd-Graber, J., Claudino, L., Socher, R., & Daumé III, H. (2014a). A Neural Network for Factoid Question Answering over Paragraphs. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 633–644, Doha, Qatar. Association for Computational Linguistics.
- Iyyer, M., Enns, P., Boyd-Graber, J., & Resnik, P. (2014b). Political Ideology Detection Using Recursive Neural Networks. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1113–1122, Baltimore, Maryland. Association for Computational Linguistics.
- Iyyer, M., Manjunatha, V., Boyd-Graber, J., & Daumé III, H. (2015). Deep Unordered Composition Rivals Syntactic Methods for Text Classification. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 1681–1691, Beijing, China. Association for Computational Linguistics.
- Johnson, R., & Zhang, T. (2014). Effective Use of Word Order for Text Categorization with Convolutional Neural Networks. *arXiv:1412.1058 [cs, stat]*.
- Johnson, R., & Zhang, T. (2015). Effective Use of Word Order for Text Categorization with Convolutional Neural Networks. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 103–112, Denver, Colorado. Association for Computational Linguistics.
- Jozefowicz, R., Zaremba, W., & Sutskever, I. (2015). An Empirical Exploration of Recurrent Network Architectures. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pp. 2342–2350.
- Kalchbrenner, N., Grefenstette, E., & Blunsom, P. (2014). A Convolutional Neural Network for Modelling Sentences. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 655–665, Baltimore, Maryland. Association for Computational Linguistics.
- Karpathy, A., Johnson, J., & Li, F.-F. (2015). Visualizing and Understanding Recurrent Networks. *arXiv:1506.02078 [cs]*.

- Kim, Y. (2014). Convolutional Neural Networks for Sentence Classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1746–1751, Doha, Qatar. Association for Computational Linguistics.
- Kingma, D., & Ba, J. (2014). Adam: A Method for Stochastic Optimization. *arXiv:1412.6980 [cs]*.
- Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. In Pereira, F., Burges, C. J. C., Bottou, L., & Weinberger, K. Q. (Eds.), *Advances in Neural Information Processing Systems 25*, pp. 1097–1105. Curran Associates, Inc.
- Kudo, T., & Matsumoto, Y. (2003). Fast Methods for Kernel-based Text Analysis. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics - Volume 1*, ACL '03, pp. 24–31, Stroudsburg, PA, USA. Association for Computational Linguistics.
- Le, P., & Zuidema, W. (2014). The Inside-Outside Recursive Neural Network model for Dependency Parsing. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 729–739, Doha, Qatar. Association for Computational Linguistics.
- Le, P., & Zuidema, W. (2015). The Forest Convolutional Network: Compositional Distributional Semantics with a Neural Chart and without Binarization. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pp. 1155–1164, Lisbon, Portugal. Association for Computational Linguistics.
- Le, Q. V., Jaitly, N., & Hinton, G. E. (2015). A Simple Way to Initialize Recurrent Networks of Rectified Linear Units. *arXiv:1504.00941 [cs]*.
- LeCun, Y., & Bengio, Y. (1995). Convolutional Networks for Images, Speech, and Time-Series. In Arbib, M. A. (Ed.), *The Handbook of Brain Theory and Neural Networks*. MIT Press.
- LeCun, Y., Bottou, L., Orr, G., & Muller, K. (1998a). Efficient BackProp. In Orr, G., & K, M. (Eds.), *Neural Networks: Tricks of the trade*. Springer.
- Lecun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998b). Gradient Based Learning Applied to Pattern Recognition..
- LeCun, Y., Chopra, S., Hadsell, R., Ranzato, M., & Huang, F. (2006). A tutorial on energy-based learning. *Predicting structured data*, 1, 0.
- LeCun, Y., & Huang, F. (2005). Loss functions for discriminative training of energybased models.. AIStats.
- Levy, O., & Goldberg, Y. (2014a). Dependency-Based Word Embeddings. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 302–308, Baltimore, Maryland. Association for Computational Linguistics.
- Levy, O., & Goldberg, Y. (2014b). Neural Word Embedding as Implicit Matrix Factorization. In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N. D., & Weinberger,

- K. Q. (Eds.), *Advances in Neural Information Processing Systems 27*, pp. 2177–2185. Curran Associates, Inc.
- Levy, O., Goldberg, Y., & Dagan, I. (2015). Improving Distributional Similarity with Lessons Learned from Word Embeddings. *Transactions of the Association for Computational Linguistics*, 3(0), 211–225.
- Lewis, M., & Steedman, M. (2014). Improved CCG Parsing with Semi-supervised Supertagging. *Transactions of the Association for Computational Linguistics*, 2(0), 327–338.
- Li, J., Li, R., & Hovy, E. (2014). Recursive Deep Models for Discourse Parsing. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 2061–2069, Doha, Qatar. Association for Computational Linguistics.
- Ling, W., Dyer, C., Black, A. W., & Trancoso, I. (2015a). Two/Too Simple Adaptations of Word2Vec for Syntax Problems. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 1299–1304, Denver, Colorado. Association for Computational Linguistics.
- Ling, W., Dyer, C., Black, A. W., Trancoso, I., Fernandez, R., Amir, S., Marujo, L., & Luis, T. (2015b). Finding Function in Form: Compositional Character Models for Open Vocabulary Word Representation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pp. 1520–1530, Lisbon, Portugal. Association for Computational Linguistics.
- Liu, Y., Wei, F., Li, S., Ji, H., Zhou, M., & WANG, H. (2015). A Dependency-Based Neural Network for Relation Classification. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pp. 285–290, Beijing, China. Association for Computational Linguistics.
- Ma, J., Zhang, Y., & Zhu, J. (2014). Tagging The Web: Building A Robust Web Tagger with Neural Network. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 144–154, Baltimore, Maryland. Association for Computational Linguistics.
- Ma, M., Huang, L., Zhou, B., & Xiang, B. (2015). Dependency-based Convolutional Neural Networks for Sentence Embedding. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pp. 174–179, Beijing, China. Association for Computational Linguistics.
- McCallum, A., Freitag, D., & Pereira, F. C. (2000). Maximum Entropy Markov Models for Information Extraction and Segmentation.. In *ICML*, Vol. 17, pp. 591–598.
- Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. *arXiv:1301.3781 [cs]*.
- Mikolov, T., Joulin, A., Chopra, S., Mathieu, M., & Ranzato, M. (2014). Learning Longer Memory in Recurrent Neural Networks. *arXiv:1412.7753 [cs]*.

- Mikolov, T., Karafiat, M., Burget, L., Cernocky, J., & Khudanpur, S. (2010). Recurrent neural network based language model.. In *INTERSPEECH 2010, 11th Annual Conference of the International Speech Communication Association, Makuhari, Chiba, Japan, September 26-30, 2010*, pp. 1045–1048.
- Mikolov, T., Kombrink, S., Lukáš Burget, Černocky, J. H., & Khudanpur, S. (2011). Extensions of recurrent neural network language model. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pp. 5528–5531. IEEE.
- Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013). Distributed Representations of Words and Phrases and their Compositionality. In Burges, C. J. C., Bottou, L., Welling, M., Ghahramani, Z., & Weinberger, K. Q. (Eds.), *Advances in Neural Information Processing Systems 26*, pp. 3111–3119. Curran Associates, Inc.
- Mikolov, T. (2012). *Statistical language models based on neural networks*. Ph.D. thesis, Ph. D. thesis, Brno University of Technology.
- Mnih, A., & Kavukcuoglu, K. (2013). Learning word embeddings efficiently with noise-contrastive estimation. In Burges, C. J. C., Bottou, L., Welling, M., Ghahramani, Z., & Weinberger, K. Q. (Eds.), *Advances in Neural Information Processing Systems 26*, pp. 2265–2273. Curran Associates, Inc.
- Mrkšić, N., Ó Séaghdha, D., Thomson, B., Gasic, M., Su, P.-H., Vandyke, D., Wen, T.-H., & Young, S. (2015). Multi-domain Dialog State Tracking using Recurrent Neural Networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pp. 794–799, Beijing, China. Association for Computational Linguistics.
- Neidinger, R. (2010). Introduction to Automatic Differentiation and MATLAB Object-Oriented Programming. *SIAM Review*, 52(3), 545–563.
- Nguyen, T. H., & Grishman, R. (2015). Event Detection and Domain Adaptation with Convolutional Neural Networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pp. 365–371, Beijing, China. Association for Computational Linguistics.
- Nivre, J. (2008). Algorithms for Deterministic Incremental Dependency Parsing. *Computational Linguistics*, 34(4), 513–553.
- Okasaki, C. (1999). *Purely Functional Data Structures*. Cambridge University Press, Cambridge, U.K.; New York.
- Pascanu, R., Mikolov, T., & Bengio, Y. (2012). On the difficulty of training Recurrent Neural Networks. *arXiv:1211.5063 [cs]*.
- Pei, W., Ge, T., & Chang, B. (2015). An Effective Neural Network Model for Graph-based Dependency Parsing. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 313–322, Beijing, China. Association for Computational Linguistics.

- Pennington, J., Socher, R., & Manning, C. (2014). Glove: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1532–1543, Doha, Qatar. Association for Computational Linguistics.
- Pollack, J. B. (1990). Recursive Distributed Representations. *Artificial Intelligence*, 46, 77–105.
- Polyak, B. T. (1964). Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5), 1 – 17.
- Qian, Q., Tian, B., Huang, M., Liu, Y., Zhu, X., & Zhu, X. (2015). Learning Tag Embeddings and Tag-specific Composition Functions in Recursive Neural Network. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 1365–1374, Beijing, China. Association for Computational Linguistics.
- Rong, X. (2014). word2vec Parameter Learning Explained. *arXiv:1411.2738 [cs]*.
- Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088), 533–536.
- Santos, C. D., & Zadrozny, B. (2014). Learning Character-level Representations for Part-of-Speech Tagging.. pp. 1818–1826.
- Schuster, M., & Paliwal, K. K. (1997). Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11), 2673–2681.
- Shawe-Taylor, J., & Cristianini, N. (2004). *Kernel Methods for Pattern Analysis*. Cambridge University Press.
- Smith, N. A. (2011). *Linguistic Structure Prediction*. Synthesis Lectures on Human Language Technologies. Morgan and Claypool.
- Socher, R. (2014). *Recursive Deep Learning For Natural Language Processing and Computer Vision*. Ph.D. thesis, Stanford University.
- Socher, R., Bauer, J., Manning, C. D., & Andrew Y., N. (2013). Parsing with Compositional Vector Grammars. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 455–465, Sofia, Bulgaria. Association for Computational Linguistics.
- Socher, R., Huval, B., Manning, C. D., & Ng, A. Y. (2012). Semantic Compositionality through Recursive Matrix-Vector Spaces. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pp. 1201–1211, Jeju Island, Korea. Association for Computational Linguistics.
- Socher, R., Lin, C. C.-Y., Ng, A. Y., & Manning, C. D. (2011). Parsing Natural Scenes and Natural Language with Recursive Neural Networks. In Getoor, L., & Scheffer, T. (Eds.), *Proceedings of the 28th International Conference on Machine Learning, ICML 2011, Bellevue, Washington, USA, June 28 - July 2, 2011*, pp. 129–136. Omnipress.

- Socher, R., Manning, C., & Ng, A. (2010). Learning Continuous Phrase Representations and Syntactic Parsing with Recursive Neural Networks. In *Proceedings of the Deep Learning and Unsupervised Feature Learning Workshop of {NIPS} 2010*, pp. 1–9.
- Socher, R., Perelygin, A., Wu, J., Chuang, J., Manning, C. D., Ng, A., & Potts, C. (2013). Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pp. 1631–1642, Seattle, Washington, USA. Association for Computational Linguistics.
- Sordoni, A., Galley, M., Auli, M., Brockett, C., Ji, Y., Mitchell, M., Nie, J.-Y., Gao, J., & Dolan, B. (2015). A Neural Network Approach to Context-Sensitive Generation of Conversational Responses. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 196–205, Denver, Colorado. Association for Computational Linguistics.
- Sundermeyer, M., Alkhouri, T., Wuebker, J., & Ney, H. (2014). Translation Modeling with Bidirectional Recurrent Neural Networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 14–25, Doha, Qatar. Association for Computational Linguistics.
- Sundermeyer, M., Schlüter, R., & Ney, H. (2012). LSTM Neural Networks for Language Modeling.. In *INTERSPEECH*.
- Sutskever, I., Martens, J., Dahl, G., & Hinton, G. (2013). On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th international conference on machine learning (ICML-13)*, pp. 1139–1147.
- Sutskever, I., Martens, J., & Hinton, G. E. (2011). Generating text with recurrent neural networks. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pp. 1017–1024.
- Sutskever, I., Vinyals, O., & Le, Q. V. V. (2014). Sequence to Sequence Learning with Neural Networks. In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N. D., & Weinberger, K. Q. (Eds.), *Advances in Neural Information Processing Systems 27*, pp. 3104–3112. Curran Associates, Inc.
- Tai, K. S., Socher, R., & Manning, C. D. (2015). Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 1556–1566, Beijing, China. Association for Computational Linguistics.
- Tamura, A., Watanabe, T., & Sumita, E. (2014). Recurrent Neural Networks for Word Alignment Model. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1470–1480, Baltimore, Maryland. Association for Computational Linguistics.
- Tieleman, T., & Hinton, G. (2012). Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning.

- Van de Cruys, T. (2014). A Neural Network Approach to Selectional Preference Acquisition. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 26–35, Doha, Qatar. Association for Computational Linguistics.
- Vaswani, A., Zhao, Y., Fossum, V., & Chiang, D. (2013). Decoding with Large-Scale Neural Language Models Improves Translation. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pp. 1387–1392, Seattle, Washington, USA. Association for Computational Linguistics.
- Wager, S., Wang, S., & Liang, P. S. (2013). Dropout Training as Adaptive Regularization. In Burges, C. J. C., Bottou, L., Welling, M., Ghahramani, Z., & Weinberger, K. Q. (Eds.), *Advances in Neural Information Processing Systems 26*, pp. 351–359. Curran Associates, Inc.
- Wang, P., Xu, J., Xu, B., Liu, C., Zhang, H., Wang, F., & Hao, H. (2015a). Semantic Clustering and Convolutional Neural Network for Short Text Categorization. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pp. 352–357, Beijing, China. Association for Computational Linguistics.
- Wang, X., Liu, Y., SUN, C., Wang, B., & Wang, X. (2015b). Predicting Polarities of Tweets by Composing Word Embeddings with Long Short-Term Memory. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 1343–1353, Beijing, China. Association for Computational Linguistics.
- Watanabe, T., & Sumita, E. (2015). Transition-based Neural Constituent Parsing. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 1169–1179, Beijing, China. Association for Computational Linguistics.
- Weiss, D., Alberti, C., Collins, M., & Petrov, S. (2015). Structured Training for Neural Network Transition-Based Parsing. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 323–333, Beijing, China. Association for Computational Linguistics.
- Werbos, P. J. (1990). Backpropagation through time: What it does and how to do it.. *Proceedings of the IEEE*, 78(10), 1550 – 1560.
- Weston, J., Bordes, A., Yakhnenko, O., & Usunier, N. (2013). Connecting Language and Knowledge Bases with Embedding Models for Relation Extraction. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pp. 1366–1371, Seattle, Washington, USA. Association for Computational Linguistics.
- Xu, W., Auli, M., & Clark, S. (2015). CCG Supertagging with a Recurrent Neural Network. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pp. 250–255, Beijing, China. Association for Computational Linguistics.

- Yin, W., & Schütze, H. (2015). Convolutional Neural Network for Paraphrase Identification. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 901–911, Denver, Colorado. Association for Computational Linguistics.
- Zaremba, W., Sutskever, I., & Vinyals, O. (2014). Recurrent Neural Network Regularization. *arXiv:1409.2329 [cs]*.
- Zeiler, M. D. (2012). ADADELTA: An Adaptive Learning Rate Method. *arXiv:1212.5701 [cs]*.
- Zeng, D., Liu, K., Lai, S., Zhou, G., & Zhao, J. (2014). Relation Classification via Convolutional Deep Neural Network. In *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*, pp. 2335–2344, Dublin, Ireland. Dublin City University and Association for Computational Linguistics.
- Zhou, H., Zhang, Y., Huang, S., & Chen, J. (2015). A Neural Probabilistic Structured-Prediction Model for Transition-Based Dependency Parsing. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 1213–1222, Beijing, China. Association for Computational Linguistics.
- Zhu, C., Qiu, X., Chen, X., & Huang, X. (2015a). A Re-ranking Model for Dependency Parser with Recursive Convolutional Neural Network. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 1159–1168, Beijing, China. Association for Computational Linguistics.
- Zhu, X., Sobhani, P., & Guo, H. (2015b). Long Short-Term Memory Over Tree Structures. *arXiv:1503.04881 [cs]*.

BLACKOUT: SPEEDING UP RECURRENT NEURAL NETWORK LANGUAGE MODELS WITH VERY LARGE VOCABULARIES

Shihao Ji

Parallel Computing Lab, Intel
shihao.ji@intel.com

S. V. N. Vishwanathan

Univ. of California, Santa Cruz
vishy@ucsc.edu

Nadathur Satish, Michael J. Anderson & Pradeep Dubey

Parallel Computing Lab, Intel
{nadathur.rajagopalan.satish,michael.j.anderson,pradeep.dubey}@intel.com

ABSTRACT

We propose *BlackOut*, an approximation algorithm to efficiently train massive recurrent neural network language models (RNNLMs) with million word vocabularies. BlackOut is motivated by using a discriminative loss, and we describe a weighted sampling strategy which significantly reduces computation while improving stability, sample efficiency, and rate of convergence. One way to understand BlackOut is to view it as an extension of the DropOut strategy to the output layer, wherein we use a discriminative training loss and a weighted sampling scheme. We also establish close connections between BlackOut, importance sampling, and noise contrastive estimation (NCE). Our experiments, on the recently released one billion word language modeling benchmark, demonstrate scalability and accuracy of BlackOut; we outperform the state-of-the art, and achieve the lowest perplexity scores on this dataset. Moreover, unlike other established methods which typically require GPUs or CPU clusters, we show that a carefully implemented version of BlackOut requires only 1-10 days on a single machine to train a RNNLM with a million word vocabulary and billions of parameters on one billion words. Although we describe BlackOut in the context of RNNLM training, it can be used to any networks with large softmax output layers.

1 INTRODUCTION

Statistical language models are a crucial component of speech recognition, machine translation and information retrieval systems. In order to handle the data sparsity problem associated with traditional n -gram language models (LMs), neural network language models (NNLMs) (Bengio et al., 2001) represent the history context in a continuous vector space that can be learned towards error rate reduction by sharing data among similar contexts. Instead of using fixed number of words to represent context, recurrent neural network language models (RNNLMs) (Mikolov et al., 2010) use a recurrent hidden layer to represent longer and variable length histories. RNNLMs significantly outperform traditional n -gram LMs, and are therefore becoming an increasingly popular choice for practitioners (Mikolov et al., 2010; Sundermeyer et al., 2013; Devlin et al., 2014).

Consider a standard RNNLM, depicted in Figure 1. The network has an input layer x , a hidden layer s (also called context layer or state) with a recurrent connection to itself, and an output layer y . Typically, at time step t the network is fed as input $x_t \in \mathbb{R}^V$, where V denotes the vocabulary size, and $s_{t-1} \in \mathbb{R}^h$, the previous state. It produces a hidden state $s_t \in \mathbb{R}^h$, where h is the size of the hidden layer, which in turn is transformed to the output $y_t \in \mathbb{R}^V$. Different layers are fully connected, with the weight matrices denoted by $\Omega = \{W_{in}^{V \times h}, W_r^{h \times h}, W_{out}^{V \times h}\}$.

For language modeling applications, the input x_t is a sparse vector of a 1-of- V (or one-hot) encoding with the element corresponding to the input word w_{t-1} being 1 and the rest of components of x_t set to 0; the state of the network s_t is a dense vector, summarizing the history context $\{w_{t-1}, \dots, w_0\}$

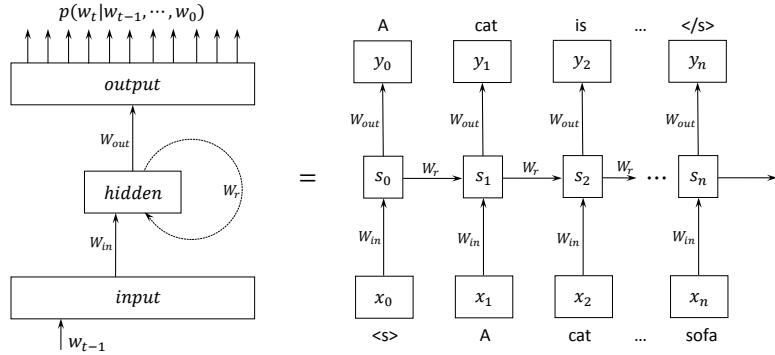


Figure 1: The network architecture of a standard RNNLM and its unrolled version for an example input sentence: $\langle s \rangle$ A cat is sitting on a sofa $\langle /s \rangle$.

preceding the output word w_t ; and the output y_t is a dense vector, with the i -th element denoting the probability of the next word being w_i , that is, $p(w_i | w_{t-1}, \dots, w_0)$, or more concisely, $p(w_i | s_t)$. The input to output transformation occurs via:

$$s_t = \sigma(W_{in}^T x_t + W_r s_{t-1}) \quad (1)$$

$$y_t = f(W_{out} s_t), \quad (2)$$

where $\sigma(v) = 1/(1 + \exp(-v))$ is the sigmoid activation function, and $f(\cdot)$ is the softmax function $f(u_i) := \exp(u_i) / \sum_{j=1}^V \exp(u_j)$.

One can immediately see that if x_t uses a 1-of- V encoding, then the computations in equation (1) are relatively inexpensive (typically h is of the order of a few thousand, and the computations are $\mathcal{O}(h^2)$), while the computations in equation (2) are expensive (typically V is of the order of a million, and the computations are $\mathcal{O}(Vh)$). Similarly, back propagating the gradients from the output layer to the hidden layer is expensive. Consequently, the training times for some of the largest models reported in literature are of the order of weeks (Mikolov et al., 2011; Williams et al., 2015).

In this paper, we ask the following question: Can we design an approximate training scheme for RNNLM which will improve on the state of the art models, while using significantly less computational resources? Towards this end, we propose *BlackOut* an approximation algorithm to efficiently train massive RNNLMs with million word vocabularies. *BlackOut* is motivated by using a discriminative loss, and we describe a weighted sampling strategy which significantly reduces computation while improving stability, sample efficiency, and rate of convergence. We also establish close connections between *BlackOut*, importance sampling, and noise contrastive estimation (NCE) (Gutmann & Hyvärinen, 2012; Mnih & Teh, 2012), and demonstrate that *BlackOut* mitigates some of the limitations of both previous methods. Our experiments, on the recently released one billion word language modeling benchmark (Chelba et al., 2014), demonstrate scalability and accuracy of *BlackOut*; we outperform the state-of-the-art, achieving the lowest perplexity scores on this dataset. Moreover, unlike other established methods which typically require GPUs or CPU clusters, we show that a carefully implemented version of *BlackOut* requires only 1-10 days on a single CPU machine to train a RNNLM with a million word vocabulary and billions of parameters on one billion words.

One way to understand *BlackOut* is to view it as an extension of the DropOut strategy (Srivastava et al., 2014) to the output layer, wherein we use a discriminative training loss and a weighted sampling scheme. The connection to DropOut is mainly from the way they operate in model training and model evaluation. Similar to DropOut, in *BlackOut* training a subset of output layer is sampled and trained at each training batch and when evaluating, the full network participates. Also, like DropOut, a regularization technique, our experiments show that the models trained by *BlackOut* are less prone to overfitting. A primary difference between them is that DropOut is routinely used at input and/or hidden layers of deep neural networks, while *BlackOut* only operates at output layer. We chose the name *BlackOut* in light of the similarities between our method and DropOut, and the complementary they offer to train deep neural networks.

2 BLACKOUT: A SAMPLING-BASED APPROXIMATION

We will primarily focus on estimation of the matrix W_{out} . To simplify notation, in the sequel we will use θ to denote W_{out} and θ_j to denote the j -th row of W_{out} . Moreover, let $\langle \cdot, \cdot \rangle$ denote the dot product between two vectors. Given these notations, one can rewrite equation (2) as

$$p_\theta(w_i|s) = \frac{\exp(\langle \theta_i, s \rangle)}{\sum_{j=1}^V \exp(\langle \theta_j, s \rangle)} \quad \forall i \in \{1, \dots, V\}. \quad (3)$$

RNNLMs with a softmax output layer are typically trained using cross-entropy as the loss function, which is equivalent to maximum likelihood (ML) estimation, that is, to find the model parameter θ which maximizes the log-likelihood of target word w_i , given a history context s :

$$J_{ml}^s(\theta) = \log p_\theta(w_i|s), \quad (4)$$

whose gradient is given by

$$\begin{aligned} \frac{\partial J_{ml}^s(\theta)}{\partial \theta} &= \frac{\partial}{\partial \theta} \langle \theta_i, s \rangle - \sum_{j=1}^V p_\theta(w_j|s) \frac{\partial}{\partial \theta} \langle \theta_j, s \rangle, \\ &= \frac{\partial}{\partial \theta} \langle \theta_i, s \rangle - \mathbb{E}_{p_\theta(w|s)} \left[\frac{\partial}{\partial \theta} \langle \theta_w, s \rangle \right]. \end{aligned} \quad (5)$$

The gradient of log-likelihood is expensive to evaluate because (1) the cost of computing $p_\theta(w_j|s)$ is $\mathcal{O}(Vh)$ and (2) the summation above takes time linear in the vocabulary size $\mathcal{O}(V)$.

To alleviate the computational bottleneck of computing the gradient (5), we propose to use the following *discriminative* objective function for training RNNLM:

$$J_{disc}^s(\theta) = \log \tilde{p}_\theta(w_i|s) + \sum_{j \in S_K} \log(1 - \tilde{p}_\theta(w_j|s)), \quad (6)$$

where S_K is a set of indices of K words drawn from the vocabulary, and $i \notin S_K$. Typically, K is a tiny fraction of V , and in our experiments we use $K \approx V/200$. To generate S_K we will sample K words from the vocabulary using an easy to sample distribution $Q(w)$, and set $q_j := \frac{1}{Q(w_j)}$ in order to compute

$$\tilde{p}_\theta(w_i|s) = \frac{q_i \exp(\langle \theta_i, s \rangle)}{q_i \exp(\langle \theta_i, s \rangle) + \sum_{j \in S_K} q_j \exp(\langle \theta_j, s \rangle)}. \quad (7)$$

Equation 6 is the cost function of a standard logistic regression classifier that discriminates one positive sample w_i from K negative samples $w_j, \forall j \in S_K$. The first term in (6) corresponds to the traditional maximum likelihood training, and the second term explicitly pushes down the probability of negative samples in addition to the implicit shrinkage enforced by the denominator of (7). In our experiments, we found the discriminative training (6) outperforms the maximum likelihood training (the first term of Eq. 6) in all the cases, with varying degree of accuracy improvement depending on K .

The weighted softmax function (7) can be considered as a stochastic version of the standard softmax (3) on a different base measure. While the standard softmax (3) uses a base measure which gives equal weights to all words, and has support over the entire vocabulary, the base measure used in (7) has support only on $K + 1$ words: the target word w_i and K samples from $Q(w)$. The noise portion of (7) has the motivation from the sampling scheme, and the q_i term for target word w_i is introduced mainly to balance the contributions from target word and noisy sample words.¹ Other justifications are discussed in Sec. 2.1 and Sec. 2.2, where we establish close connections between BlackOut, importance sampling, and noise contrastive estimation.

Due to the weighted sampling property of BlackOut, some words might be sampled multiple times according to the proposal distribution $Q(w)$, and thus their indices may appear multiple times in S_K . As w_i is the target word, which is assumed to be included in computing (7), we therefore set $i \notin S_K$ explicitly.

¹It's shown empirically in our experiments that setting $q_i = 1$ in (7) hurts the accuracy significantly.

Substituting (7) into (6) and letting $u_j = \langle \theta_j, s \rangle$ and $\tilde{p}_j = \tilde{p}_\theta(w_j|s)$, we have

$$J_{disc}^s(\theta) \propto u_i - (K+1) \log \sum_{k \in \{i\} \cup S_K} q_k \exp(u_k) + \sum_{j \in S_K} \log \left(\sum_{k \in \{i\} \cup S_K} q_k \exp(u_k) - q_j \exp(u_j) \right). \quad (8)$$

Then taking derivatives with respect to $u_j, \forall j \in \{i\} \cup S_K$, yields

$$\frac{\partial J_{disc}^s(\theta)}{\partial u_i} = 1 - \left(K+1 - \sum_{j \in S_K} \frac{1}{1-\tilde{p}_j} \right) \tilde{p}_i \quad (9)$$

$$\frac{\partial J_{disc}^s(\theta)}{\partial u_j} = - \left(K+1 - \sum_{k \in S_K \setminus \{j\}} \frac{1}{1-\tilde{p}_k} \right) \tilde{p}_j, \quad \text{for } j \in S_K. \quad (10)$$

By the chain rule of derivatives, we can propagate the errors backward to previous layers and compute the gradients with respect to the full model parameters Ω . In contrast to Eq. 5, Eqs. 9 and 10 are much cheaper to evaluate as (1) the cost of computing \tilde{p}_j is $\mathcal{O}(Kh)$ and (2) the summation takes $\mathcal{O}(K)$, hence roughly a V/K times of speed-up.

Next we turn our attention to the proposal distribution $Q(w)$. In the past, a uniform distribution or the unigram distribution have been advocated as promising candidates for sampling distributions (Bengio & Senécal, 2003; Jean et al., 2015; Bengio & Senécal, 2008; Mnih & Teh, 2012). As we will see in the experiments, neither one is suitable for a wide range of datasets, and we find that the power-raised unigram distribution of Mikolov et al. (2013) is very important in this context:

$$Q_\alpha(w) \propto p_{uni}^\alpha(w), \quad \alpha \in [0, 1]. \quad (11)$$

Note that $Q_\alpha(w)$ is a generalization of uniform distribution (when $\alpha = 0$) and unigram distribution (when $\alpha = 1$). The rationale behind our choice is that by tuning α , one can interpolate smoothly between sampling popular words, as advocated by the unigram distribution, and sampling all words equally. The best α is typically dataset and/or problem dependent; in our experiments, we use a holdout set to find the best value of α . It's worth noting that this sampling strategy has been used by Mikolov et al. (2013) in a similar context of word embedding, while here we explore its effect in the language modeling applications.

After BlackOut training, we evaluate the predictive performance of RNNLM by perplexity. To calculate perplexity, we explicitly normalize the output distribution by using the exact softmax function (3). This is similar to DropOut (Srivastava et al., 2014), wherein a subset of network is sampled and trained at each training batch and when evaluating, the full network participates.

2.1 CONNECTION TO IMPORTANCE SAMPLING

BlackOut has a close connection to importance sampling (IS). To see this, differentiating the logarithm of Eq. 7 with respect to model parameter θ , we have

$$\begin{aligned} \frac{\partial}{\partial \theta} \log \tilde{p}_\theta(w_i|s) &= \frac{\partial}{\partial \theta} \langle \theta_i, s \rangle - \frac{1}{\sum_{k \in \{i\} \cup S_K} q_k \exp(\langle \theta_k, s \rangle)} \sum_{j \in \{i\} \cup S_K} q_j \exp(\langle \theta_j, s \rangle) \frac{\partial}{\partial \theta} \langle \theta_j, s \rangle \\ &= \frac{\partial}{\partial \theta} \langle \theta_i, s \rangle - \mathbb{E}_{\tilde{p}_\theta(w|s)} \left[\frac{\partial}{\partial \theta} \langle \theta_w, s \rangle \right]. \end{aligned} \quad (12)$$

In contrast with Eq. 5, it shows that the weighted softmax function (7) corresponds to an IS-based estimator of the standard softmax (3) with a proposal distribution $Q(w)$.

Importance sampling has been applied to NNLMs with large output layers in previous works (Bengio & Senécal, 2003; 2008; Jean et al., 2015). However, either uniform distribution or unigram distribution is used for sampling and all aforementioned works exploit the maximum likelihood learning of model parameter θ . By contrast, BlackOut uses a discriminative training (6) and a power-raised unigram distribution $Q_\alpha(w)$ for sampling; these two changes are important to mitigate some of limitations of IS-based approaches. While an IS-based approach with a uniform proposal distribution

is very stable for training, it suffers from large bias due to the apparent divergence of the uniform distribution from the true data distribution $p_\theta(w|s)$. On the other hand, a unigram-based IS estimate can make learning unstable due to the high variance (Bengio & Senécal, 2003; 2008). Using a power-raised unigram distribution $Q_\alpha(w)$ entails a better trade-off between bias and variance, and thus strikes a better balance between these two extremes. In addition, as we will see from the experiments, the discriminative training of BlackOut speeds up the rate of convergence over the traditional maximum likelihood learning.

2.2 CONNECTION TO NOISE CONTRASTIVE ESTIMATION

The basic idea of NCE is to transform the density estimation problem to the problem of learning by comparison, e.g., estimating the parameters of a binary classifier that distinguishes samples from the data distribution p_d from samples generated by a known noise distribution p_n (Gutmann & Hyvärinen, 2012). In the language modeling setting, the data distribution p_d will be the distribution $p_\theta(w|s)$ of interest, and the noise distribution p_n is often chosen from the ones that are easy to sample from and possibly close to the true data distribution (so that the classification problem isn't trivial). While Mnih & Teh (2012) uses a context-independent (unigram) noise distribution $p_n(w)$, BlackOut can be formulated into the NCE framework by considering a context-dependent noise distribution $p_n(w|s)$, estimated from K samples drawn from $Q(w)$, by

$$p_n(w_i|s) = \frac{1}{K} \sum_{j \in S_K} \frac{q_j}{q_i} p_\theta(w_j|s), \quad (13)$$

which is a probability distribution function under the expectation that K samples are drawn from $Q(w)$: $S_K \sim Q(w)$ since $\mathbb{E}_{S_K \sim Q(w)}(p_n(w_i|s)) = Q(w_i)$ and $\mathbb{E}_{S_K \sim Q(w)}(\sum_{i=1}^V p_n(w_i|s)) = 1$ (See the proof in Appendix A).

Similar to Gutmann & Hyvärinen (2012), noise samples are assumed K times more frequent than data samples so that data points are generated from a mixture of two distributions: $\frac{1}{K+1}p_\theta(w|s)$ and $\frac{K}{K+1}p_n(w|s)$. Then the conditional probability of sample w_i being generated from the data distribution is

$$p_\theta(D = 1|w_i, s) = \frac{p_\theta(w_i|s)}{p_\theta(w_i|s) + Kp_n(w_i|s)}. \quad (14)$$

Inserting Eq. 13 into Eq. 14, we have

$$p_\theta(D = 1|w_i, s) = \frac{q_i \exp(\langle \theta_i, s \rangle)}{q_i \exp(\langle \theta_i, s \rangle) + \sum_{j \in S_K} q_j \exp(\langle \theta_j, s \rangle)}, \quad (15)$$

which is exactly the weighted softmax function defined in (7). Note that due to the noise distribution proposed in Eq. 13, the expensive denominator (or the partition function Z) of $p_\theta(w_j|s)$ is canceled out, while in Mnih & Teh (2012) the partition function Z is either treated as a free parameter to be learned or approximated by a constant. Mnih & Teh (2012) recommended to set $Z = 1.0$ in the NCE training. However, from our experiments, setting $Z = 1.0$ often leads to sub-optimal solutions² and different settings of Z sometimes incur numerical instability since the log-sum-exp trick³ can not be used there to shift the scores of the output layer to a range that is amenable to the exponential function. BlackOut does not have this hyper-parameter to tune and the log-sum-exp trick still works for the weighted softmax function (7). Due to the discriminative training of NCE and BlackOut, they share the same objective function (6).

We shall emphasize that according to the theory of NCE, the K samples should be sampled from the noise distribution $p_n(w|s)$. But in order to calculate $p_n(w|s)$, we need the K samples drawn from $Q(w)$ beforehand. As an approximation, we use the same K samples drawn from $Q(w)$ as the K samples from $p_n(w|s)$, and only use the expression of $p_n(w|s)$ in (13) to evaluate the noise density value required by Eq. 14. This approximation is accurate since $\mathbb{E}_{S_K \sim Q(w)}(p_n(w_i|s)) = Q(w_i)$ as proved in Appendix A, and we find empirically that it performs much better (with improved stability) than using a unigram noise distribution as in Mnih & Teh (2012).

²Similarly, Chen et al. (2015) reported that setting $\ln(Z) = 9$ gave them the best results.

³<https://en.wikipedia.org/wiki/LogSumExp>

2.3 RELATED WORK

Many approaches have been proposed to address the difficulty of training deep neural networks with large output spaces. In general, they can be categorized into four categories:

- *Hierarchical softmax* (Morin & Bengio, 2005; Mnih & Hinton, 2008) uses a hierarchical binary tree representation of the output layer with the V words as its leaves. It allows exponentially faster computation of word probabilities and their gradients, but the predictive performance of the resulting model is heavily dependent on the tree used, which is often constructed heuristically. Moreover, by relaxing the constraint of a binary structure, Le et al. (2011) introduces a structured output layer with an arbitrary tree structure constructed from word clustering. All these methods speed up both the model training and evaluation considerably.
- *Sampling-based approximations* select at random or heuristically a small subset of the output layer and estimate gradient only from those samples. The use of importance sampling in Bengio & Senécal (2003; 2008); Jean et al. (2015), and the use of NCE (Gutmann & Hyvärinen, 2012) in Mnih & Teh (2012) all fall under this category, so does the more recent use of Locality Sensitive Hashing (LSH) techniques (Shrivastava & Li, 2014; Vijayanarasimhan et al., 2014) to select a subset of good samples. BlackOut, with close connections to importance sampling and NCE, also falls in this category. All these approaches only speed up the model training, while the model evaluation still remains computationally challenging.
- *Selfnormalization* (Devlin et al., 2014) extends the cross-entropy loss function by explicitly encouraging the partition function of softmax to be as close to 1.0 as possible. Initially, this approach only speeds up the model evaluation and more recently it's extended to facilitate the training as well with some theoretical guarantees (Andreas & Klein, 2014; Andreas et al., 2015).
- *Exact gradient on limited loss functions* (Vincent et al., 2015) introduces an algorithmic approach to efficiently compute the exact loss, gradient update for the output weights in $\mathcal{O}(h^2)$ per training example instead of $\mathcal{O}(Vh)$. Unfortunately, it only applies to a limited family of loss functions that includes squared error and spherical softmax, while the standard softmax isn't included.

As discussed in the introduction, BlackOut also shares some similarity to DropOut (Srivastava et al., 2014). While DropOut is often applied to input and/or hidden layers of deep neural networks to avoid feature co-adaptation and overfitting by uniform sampling, BlackOut applies to a softmax output layer, uses a weighted sampling, and employs a discriminative training loss. We chose the name *BlackOut* in light of the similarities between our method and DropOut, and the complementary they offer to train deep neural networks.

3 IMPLEMENTATION AND FURTHER SPEED-UP

We implemented BlackOut on a standard machine with a dual-socket 28-core Intel® Xeon®⁴ Haswell CPU. To achieve high throughput, we train RNNLM with Back-Propagation Through Time (BPTT) (Rumelhart et al., 1988) with mini-batches (Chen et al., 2014). We use RMSProp (Hinton, 2012) for learning rate scheduling and gradient clipping (Bengio et al., 2013) to avoid the gradient explosion issue of recurrent networks. We use the latest Intel MKL library (version 11.3.0) for SGEMM calls, which has improved support for tall-skinny matrix-matrix multiplications, which consume about 80% of the run-time of RNNLMs.

It is expensive to access and update large models with billions of parameters. Fortunately, due to the 1-of- V encoding at input layer and the BlackOut sampling at output layer, the model update on W_{in} and W_{out} is sparse, i.e., only the model parameters corresponding to input/output words and the samples in S_K are updated at each training batch. However, subnet updates have to be done carefully due to the dependency within RMSProp updating procedure. We therefore propose an approximated RMSProp that enables an efficient subnet update and thus speeds up the algorithm even further. Details can be found in Appendix C.

⁴Intel and Xeon are trademarks of Intel Corporation in the U.S. and/or other countries.

4 EXPERIMENTS

In our experiments, we first compare BlackOut, NCE and exact softmax (without any approximation) using a small dataset. We then evaluate the performance of BlackOut on the recently released one billion word language modeling benchmark (Chelba et al., 2014) with a vocabulary size of up to one million. We compare the performance of BlackOut on a standard CPU machine versus the state-of-the-arts reported in the literature that are achieved on GPUs or on clusters of CPU nodes. Our implementation and scripts are open sourced at <https://github.com/IntelLabs/rnnlm>.

Corpus Models are trained and evaluated on two different corpora: a small dataset provided by the RNNLM Toolkit⁵, and the recently released one billion word language modeling benchmark⁶, which is perhaps the largest public dataset in language modeling. The small dataset has 10,000 training sentences, with 71,350 words in total and 3,720 unique words; and the test perplexity is evaluated on 1,000 test sentences. The one billion word benchmark was constructed from a monolingual/English corpora; after all necessary preprocessing including de-duplication, normalization and tokenization, 30,301,028 sentences (about 0.8 billion words) are randomly selected for training, 6,075 sentences are randomly selected for test and the remaining 300,613 sentences are reserved for future development and can be used as holdout set.

4.1 RESULTS ON SMALL DATASET

We evaluate BlackOut, NCE and exact softmax (without any approximation) on the small dataset described above. This small dataset is used so that we can train the standard RNNLM algorithm with exact softmax within a reasonable time frame and hence to provide a baseline of expected perplexity. There are many other techniques involved in the training, such as RMSProp for learning rate scheduling (Hinton, 2012), subnet update (Appendix C), and mini-batch splicing (Chen et al., 2014), etc., which can affect the perplexity significantly. For a fair comparison, we use the same tricks and settings for all the algorithms, and only evaluate the impact of the different approximations (or no approximation) on the softmax output layer. Moreover, there are a few hyper-parameters that have strong impact on the predictive performance, including α of the proposal distribution $Q_\alpha(w)$ for BlackOut and NCE, and additionally for NCE, the partition function Z . We pay an equal amount of effort to tune these hyper-parameters for BlackOut and NCE on the validation set as number of samples increases.

Figure 2 shows the perplexity reduction as a function of number of samples K under two different vocabulary settings: (a) a full vocabulary of 3,720 words, and (b) using the most frequent 2,065 words as vocabulary. The latter is a common approach used in practice to accelerate RNNLM computation by using RNNLM to predict only the most frequent words and handling the rest using an n -gram model (Schwenk & Gauvain, 2005). We will see similar vocabulary settings when we evaluate BlackOut on the large scale one billion word benchmark.

As can be seen, when the size of the samples increases, in general both BlackOut and NCE improve their prediction accuracy under the two vocabulary settings, and even with only 2 samples both algorithms still converge to reasonable solutions. BlackOut can utilize samples much more effectively than NCE as manifested by the significantly lower perplexities achieved by BlackOut, especially when number of samples is small; Given about 20-50 samples, BlackOut and NCE reach similar perplexities as the exact softmax, which is expensive to train as it requires to evaluate all the words in the vocabularies. When the vocabulary size is 2,065, BlackOut achieves even better perplexity than that of the exact softmax. This is possible since BlackOut does stochastic sampling at each training example and uses the full softmax output layer in prediction; this is similar to DropOut that is routinely used in input layer and/or hidden layers of deep neural networks (Srivastava et al., 2014). As in DropOut, BlackOut has the benefit of regularization and avoids feature co-adaption and is possibly less prone to overfitting. To verify this hypothesis, we evaluate the perplexities achieved on the training set for different algorithms and provide the results in Figure 5 at Appendix B. As can be seen, the exact softmax indeed overfits to the training set and reaches lower training perplexities than NCE and BlackOut.

⁵<http://www.rnnlm.org/>

⁶<https://code.google.com/p/1-billion-word-language-modeling-benchmark/>

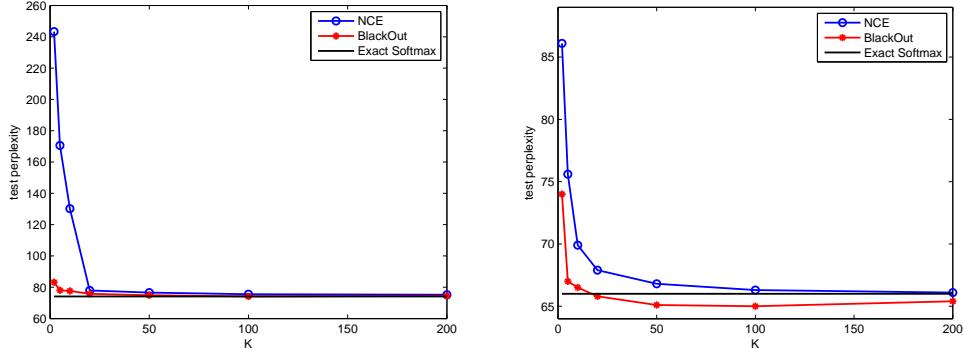


Figure 2: Test perplexity evolution as a function of number of samples K (a) with a full vocabulary of 3,720 words, and (b) with the most frequent 2,065 words in vocabulary. The experiments are executed on the RNNLMs with 16 hidden units.

Next we compare the convergence rates of BlackOut and NCE when training the RNNLMs with 16 hidden units for a full vocabulary of 3,720 words. Figures 3(a) and 3(b) plot the learning curves of BlackOut and NCE when 10 samples or 50 samples are used in training, respectively. The figure shows that BlackOut enjoys a much faster convergence rate than NCE, especially when number of samples is small (Figure 3(a)); but this advantage gets smaller when number of samples increases (Figure 3(b)). We also observed similar behavior when we evaluated BlackOut and NCE on the large scale one billion word benchmark.

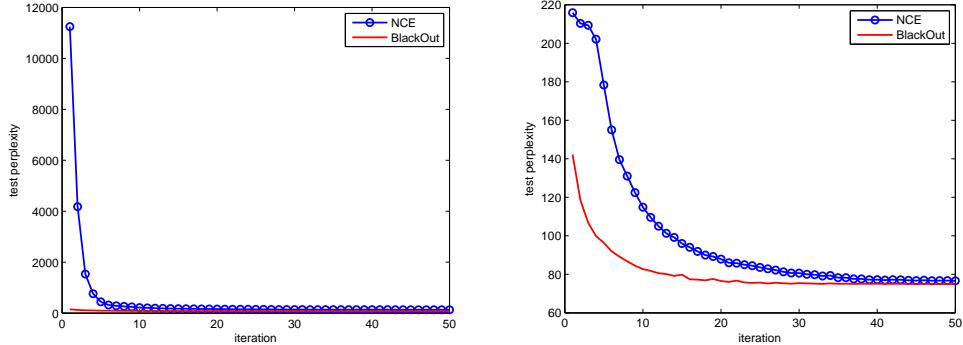


Figure 3: The learning curves of BlackOut and NCE when training the RNNLMs with 16 hidden units with (a) 10 samples, and (b) 50 samples.

4.2 RESULTS ON ONE BILLION WORD BENCHMARK

We follow the experiments from Williams et al. (2015) and Le et al. (2015) and compare the performance of BlackOut with the state-of-the-art results provided by them. While we evaluated BlackOut on a dual-socket 28-core Intel® Xeon® Haswell machine, Williams et al. (2015) implemented RNNLM with the NCE approximation on NVIDIA GTX Titan GPUs, and Le et al. (2015) executed an array of recurrent networks, including deep RNN and LSTM, without approximation on a CPU cluster. Besides the time-to-solution comparison, these published results enable us to cross-check the predictive performance of BlackOut with another implementation of NCE or with other competitive network architectures.

4.2.1 WHEN VOCABULARY SIZE IS 64K

Following the experiments in Williams et al. (2015), we evaluate the performance of BlackOut on a vocabulary of 64K most frequent words. This is similar to the scenario in Figure 2(b) where the

most frequent words are kept in vocabulary and the rest rare words are mapped to a special <unk> token. We first study the importance of α of the proposal distribution $Q_\alpha(w)$ and the discriminative training (6) as proposed in BlackOut. As we discussed in Sec. 2, when $\alpha = 0$, the proposal distribution $Q_\alpha(w)$ degenerates to a uniform distribution over all the words in the vocabulary, and when $\alpha = 1$, we recover the unigram distribution. Thus, we evaluate the impact of α in the range of $[0, 1]$. Figure 4(a) shows the evolution of test perplexity as a function of α for the RNNLMs with 256 hidden units. As can be seen, α has a significant impact on the prediction accuracy. The commonly used uniform distribution (when $\alpha = 0$) and unigram distribution (when $\alpha = 1$) often yield sub-optimal solutions. For the dataset and experiment considered, $\alpha = 0.4$ gives the best perplexity (consistent on holdout set and test set). We therefore use $\alpha = 0.4$ in the experiments that follow. The number of samples used is 500, which is about 0.8% of the vocabulary size.

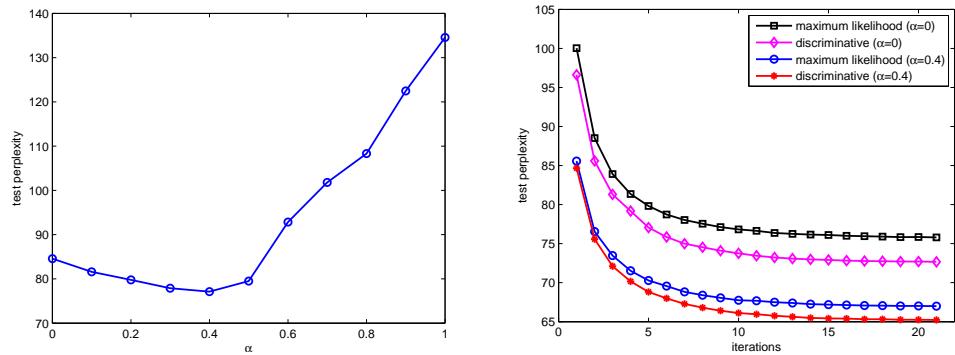


Figure 4: (a) The impact of α evaluated when 256 hidden units are used; (b) The learning curves of maximum likelihood and discriminative training when 512 hidden units are used.

Figure 4(b) demonstrates the impact of discriminative training (6) over the maximum likelihood training (the first term of Eq. 6) on the RNNLMs with 512 hidden units using two different α 's. In general, we observe 1-3 points of perplexity reduction due to discriminative training over traditional maximum likelihood training.

Finally, we evaluate the scalability of BlackOut when number of hidden units increases. As the dataset is large, we observed that the performance of RNNLM depends on the size of the hidden layer: they perform better as the size of the hidden layer gets larger. As a truncated 64K word vocabulary is used, we interpolate the RNNLM scores with a full size 5-gram to fill in rare word probabilities (Schwenk & Gauvain, 2005; Park et al., 2010). We report the interpolated perplexities BlackOut achieved and compare them with the results from Williams et al. (2015) in Table 1. As can be seen, BlackOut reaches lower perplexities than those reported in Williams et al. (2015) within comparable time frames (often 10%-40% faster). We achieved a perplexity of 42.0 when the hidden layer size is 4096. To the best of our knowledge, this is the lowest perplexity reported on this benchmark.

Table 1: Performance on the one billion word benchmark by interpolating RNNLM on a 64K word vocabulary with a full-size KN 5-gram LM.

Model	#Params [millions]	Test Perplexity		Time to Solution	
		Published ¹	BlackOut	Published ¹	BlackOut
KN 5-gram	1,748	66.95		45m	
RNN-128 + KN 5-gram	1,764	60.8	59.0	6h	9h
RNN-256 + KN 5-gram	1,781	57.3	55.1	16h	14h
RNN-512 + KN 5-gram	1,814	53.2	51.5	1d2h	1d
RNN-1024 + KN 5-gram	1,880	48.9	47.6	2d2h	1d14h
RNN-2048 + KN 5-gram	2,014	45.2	43.9	4d7h	2d15h
RNN-4096 + KN 5-gram	2,289	42.4	42.0	14d5h	10d

¹Data from Table 1 of Williams et al. (2015).

4.2.2 WHEN VOCABULARY SIZE IS 1M

In the final set of experiments, we evaluate the performance of BlackOut with a very large vocabulary of 1,000,000 words, and the results are provided in Table 2. This is the largest vocabulary used on this benchmark that we could find in existing literature. We consider the RNNLM with 1,024 hidden units (about 2 billion parameters) and 2,048 hidden units (about 4.1 billion parameters) and compare their test perplexities with the results from Le et al. (2015). We use 2,000 samples, 0.2% of the vocabulary size, for BlackOut training with $\alpha = 0.1$. Comparing to the experiments with the 64K word vocabulary, a much smaller α is used here since the sampling rate (0.2%) is much lower than that is used (0.8%) when the vocabulary size is 64K, and a smaller α strikes a better balance between sample coverage per training example and convergence rate. In contrast, NCE with the same setting converges very slowly (similar to Figure 3(a)) and couldn't reach a competitive perplexity within the time frame considered, and its results are not reported here.

As the standard RNN/LSTM algorithms (without approximation) are used in Le et al. (2015), a cluster of 32 CPU machines (at least 20 cores each) are used to train the models for about 60 hours. BlackOut enables us to train this large model using a single CPU machine for 175 hours. Since different model architectures are used in the experiments (deep RNN/LSTM vs. standard RNNLM), the direct comparison of test perplexity isn't very meaningful. However, this experiment demonstrates that even though our largest model is about 2-3 times larger than the models evaluated in Le et al. (2015), BlackOut, along with a few other optimization techniques, make this large scale learning problem still feasible on a single box machine without using GPUs or CPU clusters.

Table 2: Performance on the one billion word benchmark with a vocabulary of 1,000,000 words. Single model (RNN/LSTM-only) perplexities are reported; no interpolation is applied to any models.

	Model	Perplexity
Results from Le et al. (2015) 60 hours 32 machines	LSTM (512 units)	68.8
	IRNN (4 layers, 512 units)	69.4
	IRNN (1 layer, 1024 units + 512 linear units)	70.2
	RNN (4 layers, 512 tanh units)	71.8
	RNN (1 layer, 1024 tanh units + 512 linear units)	72.5
	RNN (1 layer, 1024 sigmoid units)	78.4
Our Results 175 hours, 1 machine	RNN (1 layer, 2048 sigmoid units)	68.3

Last, we collect all the state of the art results we are aware of on this benchmark and summarize them in Table 3. Since all the models are the interpolated ones, we interpolate our best RNN model⁷ from Table 2 with the KN 5-gram model and achieve a perplexity score of 47.3. Again, different papers provide their best models trained with different architectures and vocabulary settings. Hence, an absolutely fair comparison isn't possible. Regardless of these discrepancies, our models, within different groups of vocabulary settings, are very competitive in terms of prediction accuracy and model size.

Table 3: Comparison with the state of the art results reported on the one billion word benchmark.

Model	#Params [billions]	Test Perplexity
RNN-1024 (full vocab) + MaxEnt ¹	20	51.3
RNN-2048 (full vocab) + KN 5-gram ²	5.0	47.3
RNN-1024 (full vocab) + MaxEnt + 3 models ¹	42.9	43.8
RNN-4096 (64K vocab) + KN 5-gram ³	2.3	42.4
RNN-4096 (64K vocab) + KN 5-gram ²	2.3	42.0

¹Data from Chelba et al. (2014); ²Our results; ³Data from Williams et al. (2015).

⁷To be consistent with the benchmark in Chelba et al. (2014), we retrained it with the full-size vocabulary of about 0.8M words.

5 CONCLUSION

We proposed *BlackOut*, a sampling-based approximation, to train RNNLMs with very large vocabularies (e.g., 1 million). We established its connections to importance sampling and noise contrastive estimation (NCE), and demonstrated its stability, sample efficiency and rate of convergence on the recently released one billion word language modeling benchmark. We achieved the lowest reported perplexity on this benchmark without using GPUs or CPU clusters.

As for future extensions, our plans include exploring other proposal distributions $Q(w)$, and theoretical properties of the generalization property and sample complexity bounds for *BlackOut*. We will also investigate a multi-machine distributed implementation.

ACKNOWLEDGMENTS

We would like to thank Oriol Vinyals, Andriy Mnih and the anonymous reviewers for their excellent comments and suggestions, which helped improve the quality of this paper.

REFERENCES

- Andreas, Jacob and Klein, Dan. When and why are log-linear models self-normalizing? In *Proceedings of the Annual Meeting of the North American Chapter of the Association for Computational Linguistics*, 2014.
- Andreas, Jacob, Rabinovich, Maxim, Klein, Dan, and Jordan, Michael I. On the accuracy of self-normalized log-linear models. In *NIPS*, 2015.
- Bengio, Yoshua and Senécal, Jean-Sébastien. Quick training of probabilistic neural nets by importance sampling. In *AISTATS*, 2003.
- Bengio, Yoshua and Senécal, Jean-Sébastien. Adaptive importance sampling to accelerate training of a neural probabilistic language model. In *IEEE Transactions on Neural Networks*, volume 19, pp. 713–722, 2008.
- Bengio, Yoshua, Ducharme, Rjean, and Vincent, Pascal. A neural probabilistic language model. In *NIPS*, pp. 932–938, 2001.
- Bengio, Yoshua, Boulanger-Lewandowski, Nicolas, and Pascanu, Razvan. Advances in optimizing recurrent networks. In *ICASSP*, pp. 8624–8628, 2013.
- Chelba, Ciprian, Mikolov, Tomas, Schuster, Mike, Ge, Qi, Brants, Thorsten, Koehn, Philipp, and Robinson, Tony. One billion word benchmark for measuring progress in statistical language modeling. In *INTERSPEECH*, pp. 2635–2639, 2014.
- Chen, Xie, Wang, Yongqiang, Liu, Xunying, Gales, Mark JF, and Woodland, Philip C. Efficient gpu-based training of recurrent neural network language models using spliced sentence bunch. In *INTERSPEECH*, 2014.
- Chen, Xie, Liu, Xunying, Gales, Mark JF, and Woodland, Philip C. Recurrent neural network language model training with noise contrastive estimation for speech recognition. In *ICASSP*, 2015.
- Devlin, Jacob, Zbib, Rabih, Huang, Zhongqiang, Lamar, Thomas, Schwartz, Richard, and Makhoul, John. Fast and robust neural network joint models for statistical machine translation. In *ACL*, 2014.
- Gutmann, Michael U. and Hyvärinen, Aapo. Noise-contrastive estimation of unnormalized statistical models, with applications to natural image statistics. *JMLR*, 13:307–361, 2012.
- Hinton, Geoffrey. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

- Jean, Sbastien, Cho, Kyunghyun, Memisevic, Roland, and Bengio, Yoshua. On using very large target vocabulary for neural machine translation. In *ACL*, 2015.
- Le, Hai-Son, Oparin, Ilya, Allauzen, Alexandre, Gauvain, Jean-Luc, and Yvon, Fran ois. Structured output layer neural network language model. In *ICASSP*, pp. 5524–5527, 2011.
- Le, Quoc V., Jaitly, Navdeep, and Hinton, Geoffrey. A simple way to initialize recurrent networks of rectified linear units. *arXiv preprint arxiv:1504.00941*, 2015.
- Mikolov, Tomas, Karaf at, Martin, Burget, Luk s, Cernock , Jan, and Khudanpur, Sanjeev. Recurrent neural network based language model. In *INTERSPEECH*, pp. 1045–1048, 2010.
- Mikolov, Tomas, Deoras, Anoop, Povey, Dan, Burget, Lukar, and Cernocky, Jan Honza. Strategies for training large scale neural network language models. IEEE Automatic Speech Recognition and Understanding Workshop, 2011.
- Mikolov, Tomas, Sutskever, Ilya, Chen, Kai, Corrado, Greg, and Dean, Jeffrey. Distributed representations of words and phrases and their compositionality. In Burges, Chris, Bottou, Leon, Welling, Max, Ghahramani, Zoubin, and Weinberger, Kilian (eds.), *Advances in Neural Information Processing Systems 26*, 2013.
- Mnih, Andriy and Hinton, Geoffrey E. A scalable hierarchical distributed language model. In *NIPS*, volume 21, pp. 1081–1088, 2008.
- Mnih, Andriy and Teh, Yee Whye. A fast and simple algorithm for training neural probabilistic language models. In *Proceedings of the International Conference on Machine Learning*, 2012.
- Morin, Frederic and Bengio, Yoshua. Hierarchical probabilistic neural network language model. In *Proceedings of the international workshop on artificial intelligence and statistics*, pp. 246–252. Citeseer, 2005.
- Park, Junho, Liu, Xunying, Gales, Mark J. F., and Woodland, P. C. Improved neural network based language modelling and adaptation. In *Proc. ISCA Interspeech*, pp. 10411044, 2010.
- Rumelhart, David E., Hinton, Geoffrey E., and Williams, Ronald J. Neurocomputing: Foundations of research. chapter Learning Representations by Back-propagating Errors, pp. 696–699. MIT Press, Cambridge, MA, USA, 1988.
- Schwenk, Holger and Gauvain, Jean-Luc. Training neural network language models on very large corpora. In *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing*, pp. 201–208, 2005.
- Shrivastava, Anshumali and Li, Ping. Asymmetric lsh (alsh) for sublinear time maximum inner product search (mips). In *NIPS*, volume 27, pp. 2321–2329, 2014.
- Srivastava, Nitish, Hinton, Geoffrey, Krizhevsky, Alex, Sutskever, Ilya, and Salakhutdinov, Ruslan. Dropout: A simple way to prevent neural networks from overfitting. *JMLR*, 15:1929–1958, 2014.
- Sundermeyer, Martin, Oparin, Ilya, Gauvain, Jean-Luc, Freiberg, Ben, Schl uter, Ralf, and Ney, Hermann. Comparison of feedforward and recurrent neural network language models. In *ICASSP*, pp. 8430–8434, 2013.
- Vijayanarasimhan, Sudheendra, Shlens, Jonathon, Monga, Rajat, and Yagnik, Jay. Deep networks with large output spaces. *arXiv preprint arxiv:1412.7479*, 2014.
- Vincent, Pascal, de Brbisson, Alexandre, and Bouthillier, Xavier. Efficient exact gradient update for training deep networks with very large sparse targets. In *NIPS*, 2015.
- Williams, Will, Prasad, Niranjan, Mrva, David, Ash, Tom, and Robinson, Tony. Scaling recurrent neural network language models. In *ICASSP*, 2015.

BlackOut: Speeding up Recurrent Neural Network Language Models with Very Large Vocabularies (Supplementary Material)

A NOISE DISTRIBUTION $p_n(w_i|s)$

Theorem 1 *The noise distribution function $p_n(w_i|s)$ defined in Eq. 13 is a probability distribution function under the expectation that K samples in S_K are drawn from $Q(w)$ randomly, $S_K \sim Q(w)$, such that $\mathbb{E}_{S_K \sim Q(w)}(p_n(w_i|s)) = Q(w_i)$ and $\mathbb{E}_{S_K \sim Q(w)}(\sum_{i=1}^V p_n(w_i|s)) = 1$.*

Proof

$$\begin{aligned}
\mathbb{E}_{S_K \sim Q(w)}(p_n(w_i|s)) &= \mathbb{E}_{S_K \sim Q(w)} \left(\frac{1}{K} \sum_{j \in S_K} \frac{q_j}{q_i} p_\theta(w_j|s) \right) \\
&= \frac{Q(w_i)}{K} \mathbb{E}_{S_K \sim Q(w)} \left(\sum_{j \in S_K} \frac{p_\theta(w_j|s)}{Q(w_j)} \right) \\
&= \frac{Q(w_i)}{K} \sum_{w_k, \forall k \in S_K} \left(\prod_{k \in S_K} Q(w_k) \cdot \sum_{j \in S_K} \frac{p_\theta(w_j|s)}{Q(w_j)} \right) \\
&= \frac{Q(w_i)}{K} K \\
&= Q(w_i)
\end{aligned}$$

$$\mathbb{E}_{S_K \sim Q(w)} \left(\sum_{i=1}^V p_n(w_i|s) \right) = \sum_{i=1}^V \mathbb{E}_{S_K \sim Q(w)} (p_n(w_i|s)) = \sum_{i=1}^V (Q(w_i)) = 1$$
■

B PERPLEXITIES ON TRAINING SET

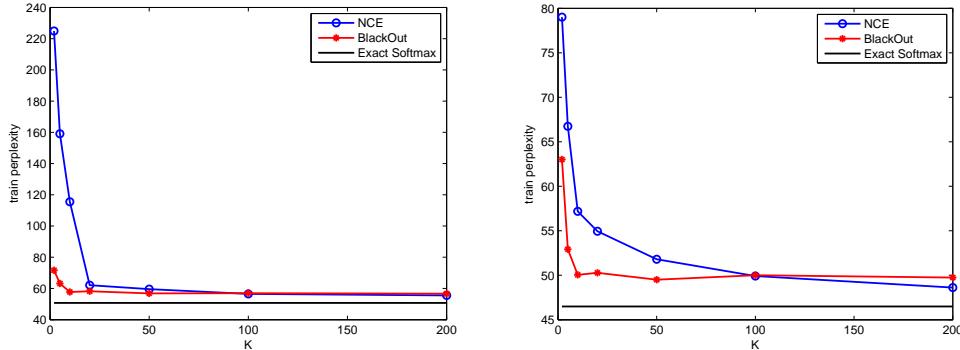


Figure 5: Training perplexity evolution as a function of number of samples K (a) with a full vocabulary of 3,720 words, and (b) with the most frequent 2,065 words in vocabulary. The experiments are executed on the RNNLMs with 16 hidden units.

C SUBNET UPDATE WITH APPROXIMATED RMSPROP

RMSProp (Hinton, 2012) is an adaptive learning rate method that has found much success in practice. Instead of using a single learning rate to all the model parameters in Ω , RMSProp dedicates a learning rate for each model parameter and normalizes the gradient by an exponential moving average of the magnitude of the gradient:

$$v_t = \beta v_{t-1} + (1 - \beta)(\nabla J)^2 \quad (16)$$

where $\beta \in (0, 1)$ denotes the decay rate. The model update at time step t is then given by

$$\theta_t = \theta_{t-1} + \epsilon \frac{\nabla J(\theta_{t-1})}{\sqrt{v_t + \lambda}} \quad (17)$$

where ϵ is the learning rate and λ is a damping factor, e.g., $\lambda = 10^{-6}$. While RMSProp is one of the most effective learning rate scheduling techniques, it requires a large amount of memory to store per-parameter v_t in addition to model parameter Ω and their gradients.

It is expensive to access and update large models with billions of parameters. Fortunately, due to the 1-of- V encoding at input layer and the BlackOut sampling at output layer, the model update on W_{in} and W_{out} is sparse, e.g., only the model parameters corresponding to input/output words and the samples in S_K are to be updated.⁸ For Eq. 16, however, even a model parameter is not involved in the current training, its v_t value still needs to be updated by $v_t = \beta v_{t-1}$ since its $(\nabla J)^2 = 0$. Ignoring this update has detrimental effect on the predictive performance; in our experiments, we observed 5 – 10 point perplexity loss if we ignore this update completely.

We resort to an approximation to $v_t = \beta v_{t-1}$. Given $p_u(w)$ is the probability of a word w being selected for update, the number of time steps elapsed when it is successfully selected follows a geometric distribution with a success rate $p_u(w)$, whose mean value is $1/p_u(w)$. Assume that an input/output word is selected according to the unigram distribution $p_{uni}(w)$ and the samples in S_K are drawn from $Q_\alpha(w)$, Eq. 16 can be approximated by

$$v_t \approx \beta^{1/p_u} v_{t-n} + (1 - \beta)(\nabla J)^2 \quad (18)$$

with

$$p_u(w) = \begin{cases} p_{uni}(w) \times B \times T & \text{for word } w \text{ at input layer} \\ p_{uni}(w) \times B \times T + Q_\alpha(w) \times K \times T & \text{for word } w \text{ at output layer,} \end{cases} \quad (19)$$

where B is the mini-batch size and T is the BPTT block size. Now we can only update the model parameters, typically a tiny fraction of Ω , that are *really* involved in the current training, and thus speed up the RNNLM training further.

⁸The parameter update on W_r is still dense, but its size is several orders of magnitude smaller than those of W_{in} and W_{out} .

Tosca: Operationalizing Commitments Over Information Protocols

Thomas C. King¹ and Akin Günay¹ and Amit K. Chopra¹ and Munindar P. Singh²

¹Lancaster University, Lancaster, LA1 4WA, United Kingdom

²North Carolina State University, Raleigh, NC 27695-8206, USA

{t.c.king, a.gunay, amit.chopra}@lancaster.ac.uk, singh@ncsu.edu

Abstract

The notion of *commitment* is widely studied as a high-level abstraction for modeling multiagent interaction. An important challenge is supporting flexible decentralized enactments of commitment specifications. In this paper, we combine recent advances on specifying commitments and *information protocols*. Specifically, we contribute Tosca, a technique for automatically synthesizing information protocols from commitment specifications. Our main result is that the synthesized protocols support *commitment alignment*, which is the idea that agents must make compatible inferences about their commitments despite decentralization.

1 Introduction

Commitments represent a high-level abstraction for modeling multiagent interaction [Singh, 1999]. The main idea behind commitment protocols is to specify the *meanings* of messages in terms of commitments [Pitt *et al.*, 2001; Yolum and Singh, 2002]. For example, to capture a purchase, one may specify that a *Quote* message means creating a commitment from the seller to the buyer to deliver an item in exchange for payment. In addition to meanings, a commitment protocol typically also specifies operational constraints such as message ordering and occurrence. Thus, for example, one would specify that the *Quote* message cannot occur before the *Request For Quote* message from the buyer to the seller. Intuitively, the motivation behind operational constraints is to rule out causally invalid protocol enactments.

A fundamental challenge in this line of work has been supporting *decentralized* enactments of commitment protocols, that is, in *shared nothing* settings where agents communicate *asynchronously*. Specifically, the only way for one agent to convey information to another is to send it a message. Supporting decentralized enactments in such settings is nontrivial because agents may observe messages in incompatible orders. Specifically, decentralization may lead to situations where agents deadlock (lack of *liveness*), observe inconsistent messages (lack of *safety*), or come to incompatible conclusions about commitments that hold between them (lack of *alignment* [Chopra and Singh, 2008; Chopra and Singh, 2009; Chopra and Singh, 2015b])—all three properties being crucial to interoperability.

Tosca addresses the challenge of decentralized enactments. It builds upon the conceptual observation that commitment specification and operational constraints are distinct concerns [Chopra and Singh, 2008; Baldoni *et al.*, 2013]. For simplicity and clarity, from here on, we reserve *protocol* to mean an operational protocol specifying messages and the operational constraints on their ordering and occurrence. Specifically, the question Tosca answers is: how can we operationalize commitment specifications over protocols such that liveness and safety are preserved, and alignment is guaranteed? Tosca’s contribution is a method for automatically synthesizing the appropriate protocol.

Tosca’s conceptual contribution is bringing three technical strands on interaction in multiagent systems together. One, BSPL [Singh, 2011], a declarative language for specifying protocols. BSPL protocols are known as *information protocols* because ordering and occurrence constraints fall out from more fundamental causality and integrity constraints on information in messages. A BSPL protocol can be checked for liveness and safety [Singh, 2012]. Two, Cupid [Chopra and Singh, 2015a], a declarative language for specifying commitments. The semantics of Cupid is in terms of commitment-oriented queries on a relational database. Thus we may imagine an agent that runs (for whatever purpose) commitment-oriented queries on its local database. Three, research on alignment [Chopra and Singh, 2015b] (C&S, for brevity), which is about mechanisms for ensuring that the parties to a commitment (the debtor and creditor) always progress toward states where they make mutually compatible local inferences about the commitment. Specifically, whenever the creditor infers a commitment as active from the messages it has observed, the debtor must as well infer it as active from its own observations.

Architecturally, Tosca brings the three strands together in the following manner. Each agent’s local database or *state* comprises the messages it would have sent or received following a BSPL protocol. Cupid enables inferring the states of the commitments an agent is party to from this database. However, because each agent carries out this inference on its own local state, it may turn out that agents are not aligned with respect to a commitment. Tosca gives a method for ensuring progress toward alignment. Specifically, given a BSPL protocol and a set of commitments defined over the messages in the protocol, it gives a method for synthesizing a BSPL protocol whose enactment guarantees progress toward

alignment. Furthermore, if the input protocol is live and safe, the synthesized protocol is live and safe as well.

Tosca goes beyond C&S in two ways. One, it addresses alignment for a more expressive language that includes deadlines, nested commitments, and a richer commitment lifecycle. Two, whereas C&S give algorithms for alignment, thereby constraining the implementation of agents, Tosca gives a purely interactive solution in terms of a protocol whose enactment would guarantee alignment.

2 Background

We now overview BSPL and Cupid, where for clarity we use *message* (as in BSPL) and *commitment* (as in Cupid) to mean instances, and *specification* to mean the respective specifications.

2.1 BSPL

BSPL is used to declaratively specify protocols without explicit control flow. By contrast, languages such as AUML [Huget and Odell, 2004] and RASA [Miller and Mcginnis, 2007] rely on explicitly specifying message ordering. Instead, BSPL protocols impose information causality constraints on each message m : what information m 's emission creates and what information the sending role must know before sending m . Thus, an implicit message ordering is imposed based solely on a protocol's explicit and declarative information causality specification.

Listing 1 demonstrates BSPL via the *Ordering* protocol. From here on, we describe such a protocol as an “input protocol” for Tosca, because it provides general message schemas for taking communicative actions (instantiating messages) and it is distinguished from a synthesized protocol for aligning a commitment.

The protocol *Ordering* has the roles M (merchant), C (customer), and S (shipper); and the parameters oID (order identifier), $item$, $price$, pID (pay identifier), rID (request identifier), and sID (ship identifier).

A *complete* enactment of *Ordering* comprises a tuple of bindings for all of its parameters. All parameters are adorned $\lceil \text{out} \rceil$ for the protocol as a whole, meaning that their values are bound by enacting the protocol. Parameter oID is annotated as a key for the other parameters. This means each oID binding corresponds to a distinct tuple of bindings for non key parameters and thus identifies *Ordering*'s enactment. For example, it is not possible for the merchant to send two quotes with key binding $oID = 1$ and different non key parameter bindings.

Ordering declares four message schemas (their placement is irrelevant). By convention, any key parameter of the protocol is a key parameter for any message in which it appears. The message schema *quote* on Line 4 is from the merchant to the customer. It has three parameters, whose values are bound by sending a quote due to being adorned $\lceil \text{out} \rceil$, namely oID , $item$, and $price$.

The message schema *pay* on Line 5 is from the customer to the merchant. It comprises one parameter adorned $\lceil \text{in} \rceil$, namely oID , which means that its value binding must be known via message emission or reception before a pay message is sent from the customer to the merchant. For example, the customer cannot send a *pay* message with $\lceil \text{in} \rceil$ parameter binding $oID = 1$ before receiving a *quote* message with the same binding.

Hence, the customer can only send a *pay* message after receiving a *quote* from the merchant with the same key value, based on the information (parameter) causality constraints.

Likewise, the message schema *requestShip* on Line 6 is from the merchant to the shipper and it has the $\lceil \text{in} \rceil$ parameter oID . Finally, *ship* on Line 7 has the oID parameter adorned $\lceil \text{in} \rceil$. Since the shipper can only know about oID 's binding by receiving a *requestShip* message from the merchant, *ship* can only be sent after being requested.

Listing 1: A BSPL protocol for placing and fulfilling orders.

```

1 Ordering {
2   roles M, C, S // Merchant, Customer, Shipper
3   parameters out oID key, out item, out price,
4       out pID, out rID, out sID
5   M → C: quote [out oID, out item, out price]
6   C → M: pay [in oID, out pID]
7   M → S: requestShip [in oID, out rID]
8   S → C: ship [in oID, out sID] }
```

2.2 Cupid

Cupid is a language for specifying commitments over an event database schema and inferring commitment states based on an event database state. In this paper, we only consider defining commitments over protocol message schemas, where commitments are inferred over messages.

We demonstrate Cupid's basic ideas with an example commitment in Listing 2 defined on top of the message schemas of Listing 1.

Listing 2: A specification in Cupid's surface syntax.

```

commitment Purchase M to C
  create quote
  detach pay [, quote + 10]
  discharge ship [, pay + 5]
```

A *Purchase* commitment from M (merchant) to C (customer) is *created* when a quote is made. The created commitment is uniquely identified by *quote*'s key (oID). *Purchase* is *detached* if a payment correlated to a quote occurs within ten time points of the quote (pay and quote both have the same key, oID). If the payment does not occur by the deadline, then the commitment is *expired* (failure to meet *detach*). The commitment is *discharged* if the (correlated) shipment occurs within five time points of the payment; if the shipment does not happen by the deadline, the commitment is *violated* (failure to meet *discharge*). Cupid treats such lifecycle events as first-class events, meaning that one commitment's lifecycle event may depend upon another's.

2.3 Separation of Concerns

Architecturally, Tosca uses BSPL to specify the *operational* layer interaction focusing on informational causality, as a separate concern from the interaction *requirements* specified in Cupid. For example, we could change Listing 1's *requestShip* message schema to be causally dependent on *pay*'s identifier (pID) before emission:

```

1 M → S: requestShip [in oID, in pID, out rID]
```

The modified information causality does not alter the fact that the *Purchase* commitment continues to be discharged by a *ship* message within five time points of the *pay* message.

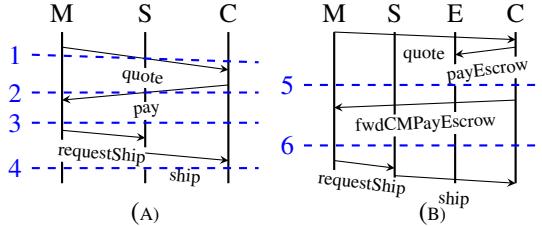


Figure 1: Protocol enactment for Merchant, Shipper, Escrow, and Customer roles.

The modification does alter when these messages *can* be sent. Conversely, changing the *Purchase* commitment in Listing 2’s discharge from *ship* to another message would not affect if and when *ship* can be sent. Tosca’s separation of concerns supports modularity: swapping out a protocol or modifying its information causality does not affect commitment level requirements, only when information (descriptively) *can* be created and thus when commitments can be met; changing a commitment does not affect if and when information can be created, only the (prescriptive) messaging *requirements* between parties.

3 Technical Motivation

We now demonstrate commitment operationalization protocols, which support realizing a commitment’s lifecycle and progression towards alignment via messaging. Specifically, given a commitment defined over an input protocol that potentially causes misalignment, we synthesize a commitment alignment protocol as output. A commitment alignment protocol includes the necessary message schemas for forwarding messages to the creditor and debtor in order to guarantee commitment alignment. Together, an input protocol and multiple commitment alignment protocols are composed to form a *commitment operationalization protocol*.

3.1 Commitments Guaranteed Alignment

The *Purchase* commitment in Listing 2 is already alignable for the create, detach, discharge, and expired lifecycle events by the input protocol, *Ordering*, in Listing 1. In Figure 1 (A), at time point 1 after the merchant sends the customer a *quote* but before the customer receives it, the debtor (merchant) infers that the *Purchase* commitment is created. Hence the commitment is already aligned (the debtor knows that they are committed) regardless of what the creditor (customer) knows.

When the customer emits *pay* before time point 2 they infer that the *Purchase* commitment is *detached*. Hence, *Purchase* becomes misaligned, since the creditor (customer) has a stronger expectation of the debtor (merchant) to discharge the commitment, which the debtor does not know. The misalignment is rectified at time point 3, after the merchant receives the *pay* message.

Subsequently, after the merchant requests the shipper to *ship*, the shipper sends a *ship* message to the customer. At time point 4, after receiving the *ship* message, the creditor (customer) infers that the commitment is discharged and hence does not have a stronger expectation of the debtor (merchant) than what the debtor knows about (alignment).

Alignment for create, detach, discharge, and expired lifecycle events is guaranteed, either because misalignment does not occur (the creditor does not infer stronger expectations of the debtor) or misalignment is rectified via message reception. However, if *ship* is received after five time points of payment, then the creditor (customer) infers violation whereas the debtor (merchant) cannot (permanent misalignment). Such misalignment requires message forwarding, (e.g., notifying the debtor of *ship*), which we will cover in the next section.

3.2 Commitments Requiring Forwarding

Suppose an escrow service is used instead of direct payment from the customer to the merchant. The *EscrowOrdering* input protocol in Listing 3 and the *EscrowPurchase* commitment in Listing 4 capture this situation.

Listing 3: An input protocol providing messaging for placing and carrying out orders using an escrow service.

```

1 EscrowOrdering {
2   roles E, M, C, S // Escrow, Merchant,
3   Customer, Shipper
4   parameters out oID key, out item, out price,
5   out pID, out rID, out sID, out tID
6   M → C: quote [out oID, out item, out price]
7   C → E: payEscrow [in oID, out pID]
8   M → S: requestShip [in oID, out rID]
9   S → C: ship [in oID, out sID]
10  E → M: payTransfer [in oID, in pID, out
tID] }
```

Listing 4: A commitment to capture escrow payment.

```

commitment EscrowPurchase M to C
create quote
detach payEscrow [, quote + 10]
discharge ship [, payEscrow + 5]
```

In this scenario, we need to introduce a message that forwards another message’s occurrence to an otherwise ignorant party. Listing 5 shows a protocol that introduces message forwarding in order to align the *EscrowPurchase* commitment (Listing 4) for the input protocol, *EscrowOrdering* in Listing 3. Specifically, by incorporating the message schema, *fwdCMPayEscrowID*, for *forwarding payEscrow* from the customer to the merchant. Each forwarding message schema has a distinct name mapped to the message being forwarded.

To exemplify, in Figure 1 (B) the customer sends *payEscrow* to the escrow. At time point 5 we have misalignment, because the customer (creditor) infers the *EscrowPurchase*’s detach and an expectation for the merchant to *ship* the goods. Yet the debtor (merchant) cannot know the customer’s expectation without notification. Misalignment is resolved at time point 6 once the customer forwards *payEscrow* to the merchant via *fwdCMPayEscrow*. Both roles know that the debtor (merchant) is expected to discharge the commitment (alignment).

Listing 5: A protocol for aligning the *EscrowPurchase* commitment in Listing 4.

```

1 EscrowPurchaseA1 {
2   roles C, M
3   parameters in oID key, in pID, out
fwdCMPayEscrowID
4 }
```

3.3 Nested Commitments

We now consider the case where one commitment's lifecycle event depends upon another's (nesting). The *EscrowTransfer* commitment given in Listing 6 is defined over the message schemas from the input protocol *EscrowOrdering* in Listing 3. The escrow service is committed to the merchant to transfer the customer's payment, once *EscrowPurchase* is discharged.

Listing 6: Escrow service's commitment to the merchant.

```
create payEscrow  
detach discharged(EscrowPurchase)  
discharge payTransfer [,  
    discharged(EscrowPurchase) +
```

EscrowTransfer is operationalized with the protocol in Listing 7. Focusing on the nested lifecycle event, the idea is that if *EscrowTransfer*'s creditor (merchant) infers its detach, then so should the debtor (escrow). *EscrowTransfer*'s detach is *EscrowPurchase*'s discharge. Hence we ensure that whenever a message contributes to *EscrowPurchase*'s discharge it can be forwarded to *EscrowTransfer*'s debtor (escrow).

Listing 7: A protocol for aligning the *EscrowTransfer* commitment in Listing 6.

```

1 EscrowTransferAI{
2   roles C, E, M //Customer, Escrow, Merchant
3   parameters in oID key, in item, in price,
4     in pID, in sID, out fwdMEQuoteID, out
5     fwdCMPayEscrowID, out fwdSEShipID,
6     out fwdMEShipID
7
7   M → E:fwdMEQuote[in oID, in item, in
8     price, out fwdMEQuoteID]
9   C → M:fwdCMPayEscrow[in oID, in pID, out
10    fwdCMPayEscrowID]
11  S → E:fwdSEShip[in oID, in sID, out
12    fwdSEShipID]
13  M → E:fwdMEShip[in oID, in sID, out
14    fwdMEShipID]

```

In Figure 2 at time point 7, the merchant infers *EscrowPurchase*'s discharge and consequently *EscrowTransfer*'s detach. Specifically, due to knowing about the messages contributing to *EscrowPurchase*'s discharge: *quote* (by sending it), *payEscrow* (via the forwarding message *fwdCMPayEscrow*), and *ship* (via the forwarding message *fwdSMSShip*). Yet the debtor (escrow) neither infers *EscrowPurchase*'s discharge nor consequently *EscrowTransfer*'s detach. Hence, the creditor (merchant) expects the debtor (escrow) to discharge *EscrowTransfer*, which the debtor is unaware of (misalignment).

Forwarding message schemas to the escrow support rectifying the misalignment. *EscrowPurchase*'s discharge is due to: *quote*, which can be forwarded to the escrow; *payEscrow*, which the escrow receives (hence no forwarding is required) and *ship*, which can be forwarded to the escrow. In Figure 1 at time point 8, after sending the forwarding messages, escrow knows about *EscrowPurchase*'s discharge and thus *EscrowTransfer*'s detach (alignment).

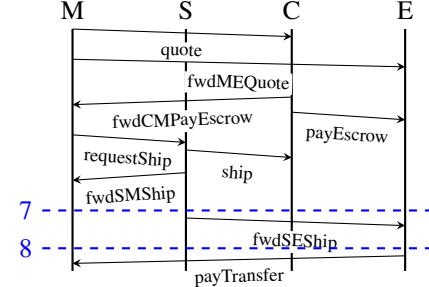


Figure 2: Protocol enactment for Merchant, Shipper, Escrow, and Customer roles where each message shares key values.

3.4 Compositionality

Commitment alignment protocols can be synthesized independently and then composed. In Listing 8, our preceding commitment alignment protocols are subprotocols of an overall operationalization protocol. If two commitment alignment protocols bind the same `out` parameter, it is due to the same message being sent and hence the binding is the same. For example `fwdCMPayEscrowID` is bound by `EscrowPurchaseA1` when a forward message `fwdCMPayEscrow` is sent if and only if `fwdCMPayEscrowID` is bound with the same value by `EscrowTransferA1` when the same `fwdCMPayEscrow` message is sent. Hence, independently constructed alignment protocols are composed together without contradictory parameter bindings during enactment.

Listing 8: An operationalization protocol composed from an input protocol and synthesized alignment protocols.

```

1 OperationalizationProtocol{
2 roles M, C, E, S
3 parameters out oid key, out item, out price,
   out pid, out sid, out rid, out tid,
   out fwdMEQuoteID, out fwdCMPayEscrowID,
   out fwdSEShipID, out fwdMEShipID
4
5 EscrowOrdering(M,
  C, E, S, out oid, out item, out price,
  out pid, out sid, out rid, out tid)
6 EscrowPurchaseAl(E, M, C, S,
  in oid, in pid, out fwdCMPayEscrowID)
7 EscrowTransferAl(C, E, M, S, in oid,
  in item, in price, in pid, in sid, out
  fwdMEQuoteID, out fwdCMPayEscrowID,
  out fwdSEShipID, out fwdMEShipID) }

```

3.5 Summary

Tosca synthesizes the alignment protocol for a commitment and an input protocol. The alignment protocol comprises forward- ing message schemas, supporting participants in aligning the commitment via messaging. Multiple commitment alignment protocols are composed together, without parameter interference, into an operationalization protocol for triggering com- mitment lifecycles and supporting alignment via messaging.

Table 1: Cupid’s grammar. Expr is create, detach, and discharge conditions.

Event	\rightarrow	Base LifeEvent
LifeEvent	\rightarrow	created(\mathcal{R} , \mathcal{R} , Expr, Expr, Expr) detached(\mathcal{R} , \mathcal{R} , Expr, Expr, Expr) discharged(\mathcal{R} , \mathcal{R} , Expr, Expr, Expr) expired(\mathcal{R} , \mathcal{R} , Expr, Expr, Expr) violated(\mathcal{R} , \mathcal{R} , Expr, Expr, Expr)
Expr	\rightarrow	Event[Time, Time] Expr \sqcap Expr Expr \sqcup Expr Expr \ominus Expr
Time	\rightarrow	Event + $\mathcal{T}^\dagger \mathcal{T}$
ComSpec	\rightarrow	c(\mathcal{R} , \mathcal{R} , Expr, Expr, Expr)

4 Synthesizing Protocols

4.1 Protocols

We adopt BSPL’s formal syntax from [Singh, 2012]. We use the following lists treated as sets: public roles \vec{x} , private roles \vec{y} , public parameters \vec{p} , $\lceil \text{key} \rceil$ parameters $\vec{k} \subseteq \vec{p}$, $\lceil \text{in} \rceil$ parameters $\vec{p}_I \subseteq \vec{p}$, $\lceil \text{out} \rceil$ parameters $\vec{p}_O \subseteq \vec{p}$, private parameters \vec{q} , and parameter bindings \vec{v} and \vec{w} . The set of all parameters is $\vec{p} = \vec{p}_I \cup \vec{p}_O$. The $\lceil \text{in} \rceil$ and $\lceil \text{out} \rceil$ parameters are mutually disjoint: $\vec{p}_I \cap \vec{p}_O = \emptyset$. A protocol’s references (i.e., subprotocols, including message schemas) are denoted by the set F .

Definition 1. A *protocol* is a tuple $P = \langle n, \vec{x}, \vec{y}, \vec{p}, \vec{k}, \vec{q}, F \rangle$ where n is a name. $\vec{x}, \vec{y}, \vec{p}, \vec{q}$ are as above, F is a finite set of f subprotocol references $F = \{F_1, \dots, F_f\}$, such that P includes each referenced sub protocol F_i ’s roles, and key and non key parameters ($\forall i : 1 \leq i \leq f \Rightarrow F_i = \langle n_i, \vec{x}_i, \vec{p}_i, \vec{k}_i \rangle$) where $\vec{x}_i \subseteq \vec{x} \cup \vec{y}$, $\vec{p}_i \subseteq \vec{p} \cup \vec{q}$, $\vec{k}_i = \vec{p}_i \cap \vec{k}$). An atomic protocol with two roles and no references is a message schema denoted as $\lceil s \mapsto r : m \vec{p}(\vec{k}) \rceil$.

Later, protocol enactment is defined over a Universe of Discourse (UoD) comprising roles and message schemas (for convenience we modify the original BSPL definition from a UoD comprising message *references*).

Definition 2. [Singh, 2012, Def. 12] The *UoD* of protocol P , $\text{UoD}(P) = \langle \mathcal{R}, \mathcal{M} \rangle$ consists of P ’s roles and message schemas including the message schemas of its referenced protocols recursively.

4.2 Commitments

A *commitment specification* is a finite string according to the corresponding representation in Table 1 over a message schema name set Base.

A commitment specification, $c(x, y, Cre, Det, Dis)$ from a debtor x to a creditor y is defined over BSPL protocol message schema references (Base events) used by an input protocol. Role names and time instants are sets \mathcal{R} and \mathcal{T} , respectively. Operators \sqcap , \sqcup , and \ominus are respectively conjunction, disjunction, and exception. In $E[l, r]$, $[l, r]$ is the time interval that $E[l, r]$ occurs within. We omit l and r when they are respectively 0 and ∞ .

4.3 Commitment Operationalization

Each commitment we wish to operationalize is rewritten into a commitment alignment protocol for an input protocol. A forwarding message schema has a unique name and $\lceil \text{out} \rceil$ parameter, to avoid conflicts with other message schemas.

Let \mathcal{N} be the set of unique message forwarding schema names disjoint from the message schema name set Base. Unique forwarding message schema names are obtained taking as input a message schema name from Base and a role, and then outputting a forwarding message schema name, using an assumed injective function $V : \text{Base} \times \mathcal{R} \rightarrow \mathcal{N}$. For example, $\text{fwdSEShip} = V(\text{ship}, E)$ is a unique name for forwarding *ship* from the shipper (S) to the escrow (E). The inverse injective function V^{-1} determines which message is being forwarded. An assumed injective parameter name function $\text{VID} : \mathcal{N} \rightarrow \mathcal{ID}$ from forwarding message names to identifier parameter names (\mathcal{ID}) produces a uniquely named $\lceil \text{out} \rceil$ parameter for each forwarding message schema (e.g., $\text{fwdSEShipID} = \text{VID}(\text{fwdSEShip})$).

A forwarding message schema is included in a commitment operationalization protocol when necessary to support commitment alignment. For example, if E is the commitment’s create condition, b the debtor and a the creditor. Then, the event formula E must be aligned between roles a and b such that if a knows E then b can learn of E via message forwarding.

Each commitment C is decomposed via rewrites into instructions of the form $\text{al}(a, E, b)$ stating a requirement for E to be aligned from a to b . If the formula E is non atomic, then it is further decomposed to eventually atomic alignment instructions, $\text{al}(a, m, b)$ on messages m . Atomic message alignment instructions are rewritten into the necessary message forwarding schema in the alignment protocol that we are constructing. We present the base case rewrites for alignment instructions operating on message schemas, then non atomic event formulae and finally commitments.

The presented rewrite rules are for an input protocol $P = \langle n, \vec{x}, \vec{y}, \vec{p}, \vec{k}, \vec{q}, F \rangle$ and its Universe of Discourse $\langle \mathcal{R}, \mathcal{M} \rangle = \text{UoD}(P)$, a commitment C and the commitment alignment protocol $P^C = \langle n^C, \vec{x}^C, \vec{y}^C, \vec{p}^C, \vec{k}^C, \vec{q}^C, F^C \rangle$ being constructed.

Rule R1 handles aligning an atomic message. It is conditional on: (1) An atomic message alignment instruction $\text{al}(a, m, b)$. (2) The commitment alignment protocol P^C being constructed. (3) A message schema in the input protocol P where a role s that is distinct from b instantiates the message m via emission to a role r distinct from b .

The rewrite result is: (4) A new message schema labeled m^{forw} acting to forward message schema m ’s instances, from the role s to the role b . (5) The commitment alignment protocol referencing m^{forw} . (6) The commitment alignment protocol including the forwarding message schema’s $\lceil \text{key} \rceil$ parameters, (7) $\lceil \text{in} \rceil$ and $\lceil \text{out} \rceil$ parameters, and roles.

$$\begin{aligned}
 & \text{al}(a, m, b), & (1) \\
 & P^C = \langle n^C, \vec{x}^C, \vec{y}^C, \vec{p}^C, \vec{k}^C, \vec{q}^C, F^C \rangle, & (2) \\
 & \lceil s \mapsto r : m \vec{p}(\vec{k}) \rceil \in \mathcal{M}, s \neq b, r \neq b & (3) \\
 & \lceil s \mapsto b : m^{\text{forw}} \vec{p}^{\text{forw}}(\vec{k}^{\text{forw}}) \rceil & (4) \\
 & F^C = F^C \cup \{m^{\text{forw}}\}, & (5) \\
 & \vec{k}^C = \vec{k}^C \cup \vec{k}, & (6) \\
 & p_I^C = p_I^C \cup p_I^{\text{forw}}, p_O^C = p_O^C \cup p_O^{\text{forw}}, \vec{x}^C = \vec{x} \cup \vec{x}^C & (7)
 \end{aligned} \tag{R1}$$

Where:

- The uniquely named forwarding message schema

forwards m to b : $m^{forw} = V(m, b)$.

- The forwarding message schema's parameters comprise: keys corresponding to m 's ($k^{forw} = k$); a unique \sqcap parameter to ensure that protocol enactment requires forwarding ($p_O^{forw} = \{VID(m^{forw})\}$); and \sqcup parameters matching m 's parameters ($p_I^{forw} = p$), meaning that m must be instantiated prior to it being forwarded.

Instructions to align messages from a to b occurring within a time window are reduced to atomic message alignment instructions. The message that occurs as well as any start or deadline messages are necessarily also aligned from a to b according to the rewrite Rule R4 (omitting cases for time windows without either a start time, a deadline, or both).

$$\frac{al(a,m[s \pm J,d \pm K],b)}{al(a,m,b) al(a,s,b) al(a,d,b)} \quad (R4)$$

To give an example, the *EscrowPurchase* commitment from the merchant to the customer in Listing 4 is detached when the customer pays the escrow within ten time points of a price quote. Hence we have an alignment instruction $al(C, payEscrow[, quote + 10], M)$ to ensure that when the creditor (customer) knows the detachment so does the debtor (merchant). The alignment instruction is reduced to $al(C, payEscrow, M)$ and $al(C, quote, M)$.

In the input protocol in Listing 3 *quote* is from the merchant to the customer, guaranteeing alignment, and so $al(C, quote, M)$ is not rewritten. However, *payEscrow* is not received or sent by the merchant and hence must be forwarded by the customer. The instruction $al(C, payEscrow, M)$ is rewritten to a message schema as in Listing 9.

Listing 9: A partial alignment protocol for the *EscrowPurchase* commitment in Listing 4

```

1 EscrowPurchaseA1{
2 roles C, M
3 parameters in oID key , in pID , out
   fwdCMPayEscrowID
4C  $\mapsto$  M: fwdCMPayEscrow [ in oID , in pID , out
   fwdCMPayEscrowID ] }
```

Both sides of a conjunct are aligned according to Rule R5. Likewise both sides of a disjunct are aligned via Rule R6, since we do not know which runtime messages will occur *a priori*.

Exceptions are handled by Rule R7. In order to guarantee that when a role a knows $L \ominus R$ then so does b , we must ensure that if a knows L then so can b via messaging. Yet, if b knows R then it will never know $L \ominus R$ to be true, even if a knows L , believes $L \ominus R$ is true and so forwards L to b . Thus we align L from a to b and R from b to a .

$$\frac{al(a,L \sqcap R,b)}{al(a,L,b) al(a,R,b)} \quad (R5) \quad \frac{al(a,L \sqcup R,b)}{al(a,L,b) al(a,R,b)} \quad (R6)$$

$$\frac{al(a,L \ominus R,b)}{al(a,L,b) al(b,R,a)} \quad (R7)$$

For example, a shipment commitment from the shipper to the merchant is discharged if: the item is shipped within five time points of the shipment being requested, except if the shipment is reported as damaged within

five time points of being received. Thus we have an alignment instruction from the shipper to the merchant: $al(S, ship[, requestShip + 5] \ominus reportDamage[, ship + 5], M)$. Alignment holds when: if the shipper knows $ship[, requestShip + 5]$ then so does the merchant and if the merchant knows the exception $reportDamage[, ship + 5]$ then so does the shipper. Hence the alignment instruction is rewritten to $al(S, ship[requestShip + 5], M)$ and $al(M, reportDamage, S)$.

Nested commitment lifecycle events occur when the corresponding lifecycle event for the referenced commitment $c(x,y,Created,Det,Dis)$ occurs. Hence, we rewrite nested lifecycle events to the conditions under which they occur according to Cupid's semantics with Rules R8 to R12.

$$\frac{al(a, created(x,y,Created,Det,Dis),b)}{al(a, Created, b)} \quad (R8)$$

$$\frac{al(a, detached(x,y,Created,Det,Dis),b)}{al(a, Created \sqcap Det, b)} \quad (R9)$$

$$\frac{al(a, discharged(x,y,Created,Det,Dis),b)}{al(a, (Created \sqcap Dis) \sqcup (Det \sqcap Dis), b)} \quad (R10)$$

$$\frac{al(a, expired(x,y,Created,Det,Dis),b)}{al(a, Created \ominus Det, b)} \quad (R11)$$

$$\frac{al(a, violated(x,y,Created,Det,Dis),b)}{al(a, (Created \sqcap Det) \ominus Disch, b)} \quad (R12)$$

The final rewrite is for commitments. A commitment is aligned when: if the creditor knows it is created, detached, or violated, then the debtor respectively knows it is created, detached, or violated; and if the debtor knows it is discharged or expired, then the creditor knows it is respectively discharged or expired. Owing to this asymmetry, we rewrite a commitment with Rule R13 to alignment instructions for each lifecycle event.

$$\frac{c(x,y,Created,Det,Dis)}{\begin{aligned} &al(c, created(x,y,Created,Det,Dis),d) \\ &al(c, detached(x,y,Created,Det,Dis),d) \\ &al(c, violated(x,y,Created,Det,Dis),d) \\ &al(d, discharged(x,y,Created,Det,Dis),c) \\ &al(d, expired(x,y,Created,Det,Dis),c) \end{aligned}} \quad (R13)$$

We now define a commitment alignment protocol.

Definition 3. A protocol P^C is a *commitment alignment protocol* for a commitment C and an input protocol P iff all possible applications of R1 to R13 are made to C , an empty version of P^C and P 's UoD $\langle \mathcal{R}, \mathcal{M} \rangle = \text{UoD}(P)$.

Each rule is a monotonic reduction on finite formulae. Hence:

Lemma 1. There exists a commitment operationalization protocol P^C for each commitment C and input protocol P .

An operationalization protocol is composed from the input protocol and commitment alignment protocols (omitting empty and redundant subprotocols).

Definition 4. Let $P = \langle n, \vec{x}, \vec{y}, \vec{p}, \vec{k}, \vec{q}, F \rangle$ be an input protocol and \mathbb{C} be a set of commitments defined over the message schema names and roles in P 's Universe of Discourse $\langle \mathcal{R}, \mathcal{M} \rangle =$

$\text{UoD}(P)$. Let each commitment $C \in \mathbb{C}$ have an alignment protocol $P^C = \langle n^C, \vec{x}^C, \vec{y}^C, \vec{p}^C, \vec{k}^C, \vec{q}^C, F^C \rangle$ for P that includes at least two roles ($|\mathcal{R}| \geq 2$). $P^{\mathbb{C}} = \langle n^{\mathbb{C}}, \vec{x}^{\mathbb{C}}, \vec{y}^{\mathbb{C}}, \vec{p}^{\mathbb{C}}, \vec{k}^{\mathbb{C}}, \vec{q}^{\mathbb{C}}, F^{\mathbb{C}} \rangle$ is an operationalization protocol for \mathbb{C} and P iff:

$P^{\mathbb{C}}$ references the input protocol and all commitment operationalization protocols: $F^{\mathbb{C}} = \{n\} \cup \bigcup_{C \in \mathbb{C}} \{n^C\}$.

$P^{\mathbb{C}}$'s roles and key parameters match the input protocol's: $\vec{x}^{\mathbb{C}} = \vec{x}$ and $\vec{k}^{\mathbb{C}} = \vec{k}$.

P 's $\lceil \text{out} \rceil$ parameters comprise the input protocol's and each commitment alignment protocol's $\lceil \text{out} \rceil$ parameters: $\vec{p}_O^{\mathbb{C}} = \{\vec{p}_O\} \cup \bigcup_{C \in \mathbb{C}} \{\vec{p}_O^C\}$.

4.4 Semantics

In BSPL, each message instance $m[s, r, \vec{p}, \vec{v}]$ denotes a sending role s , a recipient r , a parameter vector \vec{p} with a corresponding parameter binding value vector \vec{v} . A role's history denotes its sent and received messages in sequence. A history vector comprises each role's history where every received message must have been sent.

Definition 5. [Singh, 2012, Def. 5] A history of a role ρ , H_ρ , is given by a sequence of zero or more message instances $m_1 \circ m_2 \circ \dots$. Each m_i is of the form $m[s, r, \vec{p}, \vec{v}]$ where $\rho = s$ or $\rho = r$, and \circ means sequencing.

Definition 6. [Singh, 2012, Def. 7] We define a *history vector* for a UoD \mathcal{R}, \mathcal{M} , as $[H^1, \dots, H^{|\mathcal{R}|}]$, such that $\forall s, r : 1 \leq s, r \leq |\mathcal{R}| : H^s$ is a history s.t. $\forall m[s, r, \vec{p}, \vec{v}] \in H^r : m \in \mathcal{M}$ and $m[s, r, \vec{p}, \vec{v}] \in H^s$.

A history vector is viable if and only if sent and received messages bind values to parameters specified in each corresponding message schema, respecting values already determined by keys via known messages (for brevity, we omit Singh's [2012, Def. 8] definition). The set of all viable history vectors for a UoD (e.g., a protocol's roles and message schemas) is its Universe of Enactments.

Definition 7. Given a UoD $\langle \mathcal{R}, \mathcal{M} \rangle$, the *Universe of Enactments* (UoE) for that UoD, $\mathcal{U}_{\mathcal{R}, \mathcal{M}}$, is the set of viable history vectors [2012, Definition 8], each of which has exactly $|\mathcal{R}|$ dimensions and each of whose messages instantiates a schema in \mathcal{M} .

In Cupid [Chopra and Singh, 2015a], an agent's *model* maps from event schemas to event instances, representing the agent's local view of event occurrences. Here, we are dealing with messages and hence a model is simply a role's history albeit substituting each forwarding message with the message it forwards and timestamping each message with the time it became known (via emission, reception, or notification).

Definition 8. Let P be an input protocol and let Base be the message schema names for the message schemas in P 's UoD. Let $P^{\mathbb{C}}$ be an operationalization protocol for P . A *model* for a role a 's history H and $P^{\mathbb{C}}$ is a history M , where each message is in the model $m_i^M[s_i^M, r_i^M, \vec{p}^M, \vec{v}^M] \in M$ iff there is a corresponding original message $m_i^H[s_i^H, r_i^H, \vec{p}^H, \vec{v}^H] \in H$ meeting (a) or (b), and (c):

(a) The corresponding original message in H is a non forwarding message $m_i^H \in \text{Base}$ and the names match: $m_i^M = m_i^H$.

- (b) The corresponding original message m_i^H in H is a forwarding message and the message m_i^M in the model takes the name of the message being forwarded: $m_i^M = V^{-1}(m_i^H, r_i^H)$.
- (c) The message m_i^M contains all of the original message m_i^H in H 's non forwarding ID parameters and parameter bindings with an additional timestamp parameter and parameter binding: if $\exists \vec{p}_j^H \in \vec{p}^H = VID^{-1}(m_i^H)$ then $\vec{p}_i^M = (\vec{p}_i^H \setminus \vec{p}_j^H) \cup \{\text{time}\}$ and $\vec{v}_i^M = (\vec{v}_i^H \setminus \vec{v}_j^H) \cup \{t\}$, otherwise $\vec{p}_i^M = \vec{p}_j^H \cup \{\text{time}\}$ and $\vec{v}_i^M = \vec{v}_j^H \cup \{t\}$, where t is a timestamp.

We adopt Cupid's commitment semantics. For brevity, we only define the set of instances for event formula E (e.g., a lifecycle event) entailed by a model M : $\llbracket E \rrbracket_M$. If E is a non atomic event formula or a lifecycle event, then the result is a database operation on the messages that cause E to occur. For example, the set of all ship instances is denoted as $\llbracket \text{ship} \rrbracket$. Moreover, the set of tuples modeled by the formulae for shipment occurring within ten time points, $\text{ship}[0, 10]$, is returned by selecting all shipment events that occur between zero and ten time points ($\llbracket \text{ship}[0, 10] \rrbracket = \sigma_{0 \leq t < 10}(\llbracket \text{ship} \rrbracket)$). A database operation is inductively defined in Cupid for queries corresponding to each event formula type.

Definition 9. Let M be a model and E an event formula. $\llbracket E \rrbracket_M$ is the set of E 's instances (a set of tuples combining stored events) returned from the query for E on M according to [Chopra and Singh, 2015a, D₁–D₂₀].

5 Properties

An operationalization protocol retains an input protocol's message ordering.

Lemma 2. Let P be an input protocol, and $P^{\mathbb{C}}$ be a commitment operationalization protocol. If there is a history vector H in $\text{UoD}(P)$'s UoE then there exists a history vector $H^{\mathbb{C}}$ in $\text{UoD}(P^{\mathbb{C}})$'s UoE where for each history H^C in $H^{\mathbb{C}}$ the messages instantiating schemas in \mathcal{M} are included in the same order of the corresponding history H^i in H .

Proof sketch. We include messages from the input protocol's history to the operationalization protocol's corresponding history, if they instantiate a schema in the input protocol. We can always include received messages ([2012, Def. 6]), emitted messages can be included since they do not violate bindings and $\lceil \text{in} \rceil$ parameters are necessarily known by binding parameters in the operationalization protocol's message schemas. \square

Singh [2012] formalizes liveness and safety for BSPL, and gives verification techniques. A protocol is *safe* iff each history vector in the UoE preserves uniqueness for each binding. A protocol is *live* iff any enactment can progress to completion such that all parameters are bound.

Theorem 1. Let $P^{\mathbb{C}}$ be a commitment operationalization protocol for an input protocol P . If P is safe then $P^{\mathbb{C}}$ is safe. If P is live then $P^{\mathbb{C}}$ is live.

Proof sketch. Safety: Assume $P^{\mathbb{C}}$ is unsafe. Since $P^{\mathbb{C}}$ is an operationalization protocol, by Rule R1 (7) it does not

introduce ‘out’ parameters used by another message schema, hence by applying Lemma 2 P must be unsafe as well. Liveness: As for safety, relying on P^C not introducing ‘out’ parameters used by P . \square

C&S define alignment for active commitments. Definition 10 generalizes it to all states in Cupid’s commitment lifecycle. Specifically, if a creditor infers created, detached or violated of a commitment (thereby strengthening the expectation) from its history, then the debtor must as well (i.e., know what is expected of them). Conversely, if a debtor infers discharge or expired (hence weakening the expectation), the creditor must as well.

Definition 10. Let M^x and M^y be models. The history vector H is aligned with respect to $c(x,y,C,D,U)$ iff:

$$\begin{aligned} i \in [\![\text{created}(x,y,C,D,U)]\!]_{M^y} &\Rightarrow i \in [\![\text{created}(x,y,C,D,U)]\!]_{M^x} \\ i \in [\![\text{detached}(x,y,C,D,U)]\!]_{M^y} &\Rightarrow i \in [\![\text{detached}(x,y,C,D,U)]\!]_{M^x} \\ i \in [\![\text{violated}(x,y,C,D,U)]\!]_{M^y} &\Rightarrow i \in [\![\text{violated}(x,y,C,D,U)]\!]_{M^x} \\ i \in [\![\text{discharged}(x,y,C,D,U)]\!]_{M^x} &\Rightarrow i \in [\![\text{discharged}(x,y,C,D,U)]\!]_{M^y} \\ i \in [\![\text{expired}(x,y,C,D,U)]\!]_{M^x} &\Rightarrow i \in [\![\text{expired}(x,y,C,D,U)]\!]_{M^y} \end{aligned}$$

The idea that messages should happen in some time interval relies on a global clock. However, there is the potential for misalignment due to message delays and local clock skews, rather than which messages are emitted and received. Such problems are avoided by making the time intervals an order of magnitude larger than the maximum clock skew and message delays [Cranefield, 2005].

We assume that for a commitment $c(x,y,Cre,Det,Dis)$ being operationalized with respect to a history vector H if when x or y knows a message m and the counter-party receives m they will do so at a time to make the same inferences over Cre , Det , and Dis . Under this assumption, an operationalization protocol is sufficient to support alignment.

Theorem 2 states that a commitment operationalization protocol always makes it possible to rectify alignment via messaging.

Theorem 2. Let P^C be a protocol that operationalizes a set of commitments \mathbb{C} such that $c(x,y,Cre,Det,Dis) \in \mathbb{C}$. If history vector $H \in \mathcal{U}_{\mathcal{R},\mathcal{M}}$ is in P^C ’s UoE then there exists a longer H'' in P^C ’s UoE that is aligned with respect to $c(x,y,Cre,Det,Dis)$.

Proof sketch. If H is misaligned. Definition 10 and Cupid’s semantics for lifecycle events [Chopra and Singh, 2015a, D₁₅ to D₁₉] imply role s ’s model entails E . Base case: $E = m$. By R1 extend H to a history vector H'' in P^C ’s UoE (Definition 7) by inserting a notification m_i from m ’s sender to r in their respective histories. Inductive hypothesis: assume there exists a history H' in P^C ’s UoE extending H s.t. if $E = F \sqcap G$ or $E = F \sqcup G$ then r knows F and G , or for $E = F \ominus G$ r knows F and s knows G . Inductive step: Extend H' to H'' with an m_i via rules R1 to R12. By the time assumption and Cupid’s semantic definitions [Chopra and Singh, 2015a, Def. 16 to Def. 19] s and r are aligned. \square

6 Conclusions

Tosca addresses challenges of decentralized commitment enactment. Given a set of commitments defined over a protocol, it enables automatically synthesizing a new protocol that supports alignment, a form of commitment-level interoperability.

Furthermore, the synthesized protocol preserves liveness and safety, both of which are also necessary for interoperability. The new protocol may be thought of as a *fleshing out* of the input protocol. Tosca brings together several advances—in the specification of protocols, commitments, and ideas about interoperability—toward supporting decentralization.

Tosca establishes a separation of concerns between commitments and protocols. Protocols are specified in BSPL whereas commitments are specified in Cupid—languages developed independently of each other. Notably, in Cupid, commitments are defined over a database schema, not a protocol. We use the fact that BSPL specifications are interpreted over databases in layering Cupid specifications over BSPL specifications.

Decentralization is a theme of growing interest, (e.g., for norm compliance [Baldoni *et al.*, 2015] and monitoring [Bulling *et al.*, 2013]). Tosca’s architecture is distinct from shared memory (environment) approaches (e.g., [Omicini *et al.*, 2008]). Such approaches would benefit from Tosca in that they would also need a clear specification of interactions, both in terms of meanings and protocols, even if alignment itself would be trivial because of shared memory.

Other works treat commitments [Chesani *et al.*, 2013] and protocols [Yadav *et al.*, 2015] separately without studying their relationship. Günay *et al.* [Günay *et al.*, 2015] generate commitment specifications from requirements. We understand commitment specifications as requirements and synthesize operational protocol specifications.

Analogously Searle [1995, pp. 26–27] demarcates between constitutive rules, which make social actions possible by ascribing institutional (social) facts, and norms, which prescribe institutional facts. This separation is adopted for commitment protocols overlaying constitutive rules [Baldoni *et al.*, 2013]. Moreover, norms are both defined over institutional facts and interpreted at different levels of abstraction using constitutive rules within agents [Criado *et al.*, 2013] and legal institutions [King, 2016; King *et al.*, 2017]. We separate richer concerns: commitments, focusing on relational information, overlaying operational protocols, focusing on information causality.

Future directions should address limitations and further applications. Tosca maintains agent autonomy, making alignment possible but not regimented and hence limited by the extent to which autonomous agents communicate message notifications. We demonstrated Tosca on a business domain but it could just as well be applied to requirements in other domains that involve interaction. In particular, healthcare and government (e.g., national) contracts, studied in connection with Cupid are prime candidates for Tosca. Since Tosca provides support for messaging requirements via protocols, the extent to which it can be applied to agent development should be investigated (e.g., using communication abstractions supported in agent programming frameworks such as [Boissier *et al.*, 2013]).

Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments. King, Günay and Chopra were supported by the EPSRC grant EP/N027965/1 (Turtles). Singh thanks the US Department of Defense for partial support under the Science of Security Lablet.

References

- [Baldoni *et al.*, 2013] Matteo Baldoni, Cristina Baroglio, Elisa Marengo, and Viviana Patti. Constitutive and Regulative Specifications of Commitment Protocols: a Decoupled Approach. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 4(2), 2013.
- [Baldoni *et al.*, 2015] Matteo Baldoni, Cristina Baroglio, Amit K. Chopra, and Munindar P. Singh. Composing and Verifying Commitment-Based Multiagent Protocols. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 10–17. AAAI Press, 2015.
- [Boissier *et al.*, 2013] Olivier Boissier, Rafael H. Bordini, Jomi F. Hübner, Alessandro Ricci, and Andrea Santi. Multi-agent oriented programming with JaCaMo. *Science of Computer Programming*, 78(6):747–761, 2013.
- [Bulling *et al.*, 2013] Nils Bulling, Mehdi Dastani, and Max Knobbe. Monitoring norm violations in multi-agent systems. In *Proceedings of the 2013 International Conference on Autonomous Agents and Multi-agent Systems (AAMAS 2013)*, pages 491–498. International Foundation for Autonomous Agents and Multiagent Systems, 2013.
- [Chesani *et al.*, 2013] Federico Chesani, Paola Mello, Marco Montali, and Paolo Torroni. Representing and monitoring social commitments using the event calculus. *Autonomous Agents and Multiagent Systems*, 27(1):85–130, 2013.
- [Chopra and Singh, 2008] Amit K. Chopra and Munindar P. Singh. Constitutive interoperability. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 2*, pages 797–804, 2008.
- [Chopra and Singh, 2009] AK Chopra and MP Singh. Multiagent commitment alignment. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, pages 937–944, 2009.
- [Chopra and Singh, 2015a] Amit K. Chopra and Munindar P. Singh. Cupid: Commitments in Relational Algebra. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, pages 2052–2059, 2015.
- [Chopra and Singh, 2015b] Amit K. Chopra and Munindar P. Singh. Generalized Commitment Alignment. In *Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems*, pages 453–461, 2015.
- [Cranefield, 2005] Stephen Cranefield. A Rule Language for Modelling and Monitoring Social Expectations in Multi-Agent Systems. In *Coordination, Organization, Institutions and Norms in Multi-Agent Systems. Lecture Notes in Computer Science.*, volume 3913, pages 246–258. Springer, 2005.
- [Criado *et al.*, 2013] N Criado, E. Argente, P. Noriega, and V. Botti. Reasoning about constitutive norms in BDI agents. *Logic Journal of the IGPL*, 22(1):66–93, 2013.
- [Günay *et al.*, 2015] Akin Günay , Michael Winikoff, and Pinar Yolum. Dynamically generated commitment protocols in open systems. *Autonomous Agents and Multi-Agent Systems*, 29:192–229, 2015.
- [Huget and Odell, 2004] Marc-Philippe Huget and James Odell. Representing agent interaction protocols with agent UML. In *Proceedings of the 1st International Workshop on Agent-Oriented Software Engineering (AOSE 2000), Lecture Notes in Computer Science*, volume 3382, pages 16 – 30, 2004.
- [King *et al.*, 2017] Thomas C. King, Marina De Vos, Virginia Dignum, Catholijn M. Jonker, Tingting Li, Julian Padgett, and M. Birna van Riemsdijk. Automated multi-level governance compliance checking. *Autonomous Agents and Multi-Agent Systems*, 2017.
- [King, 2016] Thomas C. King. *Governing Governance: A Formal Framework for Analysing Institutional Design and Enactment Governance*. PhD thesis, Delft University of Technology, 2016.
- [Miller and Mcginnis, 2007] Tim Miller and Jarred Mcginnis. Amongst First-Class Protocols. In *Proceedings of the 8th International Workshop on Engineering Societies in the Agents World (ESAW 2007). Lecture Notes in Computer Science.*, volume 1957, pages 208 – 223, 2007.
- [Omicini *et al.*, 2008] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. Artifacts in the A&A meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 17(3):432–456, 2008.
- [Pitt *et al.*, 2001] Jeremy Pitt, Lloyd Kamara, and Alexander Artikis. Interaction patterns and observable commitments in a multi-agent trading scenario. In *Proceedings of the 5th International Conference on Autonomous Agents*, pages 481–488, 2001.
- [Searle, 1995] John R. Searle. *The Construction of Social Reality*. The Free Press, New York, 1995.
- [Singh, 1999] MP Singh. An ontology for commitments in multiagent systems: Toward a unification of normative concepts. *Artificial Intelligence and Law*, 7:97–113, 1999.
- [Singh, 2011] Munindar P. Singh. Information-driven interaction-oriented programming: BSPL, the blindingly simple protocol language. In *Proceedings of the 10th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, pages 491–498, 2011.
- [Singh, 2012] Munindar Singh. Semantics and Verification of Information-Based Protocols. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, pages 1149–1156, 2012.
- [Yadav *et al.*, 2015] Nitin Yadav, Michael Winikoff, and Lin Padgham. HAPN: Hierarchical Agent Protocol Notation. In *Proceedings of the International Workshop on Coordination, Organisation, Institutions and Norms in Multi-Agent Systems (COIN@IJCAI)*, 2015.
- [Yolum and Singh, 2002] Pinar Yolum and Munindar P. Singh. Flexible Protocol Specification and Execution: Applying Event Calculus Planning using Commitments. In *Proceedings of the first International Joint Conference on Autonomous Agents and Multiagent Systems: part 2*, pages 527–534, Bologna, Italy, 2002.

Deep learning

Yann LeCun^{1,2}, Yoshua Bengio³ & Geoffrey Hinton^{4,5}

Deep learning allows computational models that are composed of multiple processing layers to learn representations of data with multiple levels of abstraction. These methods have dramatically improved the state-of-the-art in speech recognition, visual object recognition, object detection and many other domains such as drug discovery and genomics. Deep learning discovers intricate structure in large data sets by using the backpropagation algorithm to indicate how a machine should change its internal parameters that are used to compute the representation in each layer from the representation in the previous layer. Deep convolutional nets have brought about breakthroughs in processing images, video, speech and audio, whereas recurrent nets have shone light on sequential data such as text and speech.

Machine-learning technology powers many aspects of modern society: from web searches to content filtering on social networks to recommendations on e-commerce websites, and it is increasingly present in consumer products such as cameras and smartphones. Machine-learning systems are used to identify objects in images, transcribe speech into text, match news items, posts or products with users' interests, and select relevant results of search. Increasingly, these applications make use of a class of techniques called deep learning.

Conventional machine-learning techniques were limited in their ability to process natural data in their raw form. For decades, constructing a pattern-recognition or machine-learning system required careful engineering and considerable domain expertise to design a feature extractor that transformed the raw data (such as the pixel values of an image) into a suitable internal representation or feature vector from which the learning subsystem, often a classifier, could detect or classify patterns in the input.

Representation learning is a set of methods that allows a machine to be fed with raw data and to automatically discover the representations needed for detection or classification. Deep-learning methods are representation-learning methods with multiple levels of representation, obtained by composing simple but non-linear modules that each transform the representation at one level (starting with the raw input) into a representation at a higher, slightly more abstract level. With the composition of enough such transformations, very complex functions can be learned. For classification tasks, higher layers of representation amplify aspects of the input that are important for discrimination and suppress irrelevant variations. An image, for example, comes in the form of an array of pixel values, and the learned features in the first layer of representation typically represent the presence or absence of edges at particular orientations and locations in the image. The second layer typically detects motifs by spotting particular arrangements of edges, regardless of small variations in the edge positions. The third layer may assemble motifs into larger combinations that correspond to parts of familiar objects, and subsequent layers would detect objects as combinations of these parts. The key aspect of deep learning is that these layers of features are not designed by human engineers: they are learned from data using a general-purpose learning procedure.

Deep learning is making major advances in solving problems that have resisted the best attempts of the artificial intelligence community for many years. It has turned out to be very good at discovering

intricate structures in high-dimensional data and is therefore applicable to many domains of science, business and government. In addition to beating records in image recognition^{1–4} and speech recognition^{5–7}, it has beaten other machine-learning techniques at predicting the activity of potential drug molecules⁸, analysing particle accelerator data^{9,10}, reconstructing brain circuits¹¹, and predicting the effects of mutations in non-coding DNA on gene expression and disease^{12,13}. Perhaps more surprisingly, deep learning has produced extremely promising results for various tasks in natural language understanding¹⁴, particularly topic classification, sentiment analysis, question answering¹⁵ and language translation^{16,17}.

We think that deep learning will have many more successes in the near future because it requires very little engineering by hand, so it can easily take advantage of increases in the amount of available computation and data. New learning algorithms and architectures that are currently being developed for deep neural networks will only accelerate this progress.

Supervised learning

The most common form of machine learning, deep or not, is supervised learning. Imagine that we want to build a system that can classify images as containing, say, a house, a car, a person or a pet. We first collect a large data set of images of houses, cars, people and pets, each labelled with its category. During training, the machine is shown an image and produces an output in the form of a vector of scores, one for each category. We want the desired category to have the highest score of all categories, but this is unlikely to happen before training. We compute an objective function that measures the error (or distance) between the output scores and the desired pattern of scores. The machine then modifies its internal adjustable parameters to reduce this error. These adjustable parameters, often called weights, are real numbers that can be seen as 'knobs' that define the input–output function of the machine. In a typical deep-learning system, there may be hundreds of millions of these adjustable weights, and hundreds of millions of labelled examples with which to train the machine.

To properly adjust the weight vector, the learning algorithm computes a gradient vector that, for each weight, indicates by what amount the error would increase or decrease if the weight were increased by a tiny amount. The weight vector is then adjusted in the opposite direction to the gradient vector.

The objective function, averaged over all the training examples, can

¹Facebook AI Research, 770 Broadway, New York, New York 10003 USA. ²New York University, 715 Broadway, New York, New York 10003, USA. ³Department of Computer Science and Operations Research Université de Montréal, Pavillon André-Aisenstadt, PO Box 6128 Centre-Ville STN Montréal, Quebec H3C 3J7, Canada. ⁴Google, 1600 Amphitheatre Parkway, Mountain View, California 94043, USA. ⁵Department of Computer Science, University of Toronto, 6 King's College Road, Toronto, Ontario M5S 3G4, Canada.

be seen as a kind of hilly landscape in the high-dimensional space of weight values. The negative gradient vector indicates the direction of steepest descent in this landscape, taking it closer to a minimum, where the output error is low on average.

In practice, most practitioners use a procedure called stochastic gradient descent (SGD). This consists of showing the input vector for a few examples, computing the outputs and the errors, computing the average gradient for those examples, and adjusting the weights accordingly. The process is repeated for many small sets of examples from the training set until the average of the objective function stops decreasing. It is called stochastic because each small set of examples gives a noisy estimate of the average gradient over all examples. This simple procedure usually finds a good set of weights surprisingly quickly when compared with far more elaborate optimization techniques¹⁸. After training, the performance of the system is measured on a different set of examples called a test set. This serves to test the generalization ability of the machine — its ability to produce sensible answers on new inputs that it has never seen during training.

Many of the current practical applications of machine learning use linear classifiers on top of hand-engineered features. A two-class linear classifier computes a weighted sum of the feature vector components. If the weighted sum is above a threshold, the input is classified as belonging to a particular category.

Since the 1960s we have known that linear classifiers can only carve their input space into very simple regions, namely half-spaces separated by a hyperplane¹⁹. But problems such as image and speech recognition require the input–output function to be insensitive to irrelevant variations of the input, such as variations in position, orientation or illumination of an object, or variations in the pitch or accent of speech, while being very sensitive to particular minute variations (for example, the difference between a white wolf and a breed of wolf-like white dog called a Samoyed). At the pixel level, images of two Samoyeds in different poses and in different environments may be very different from each other, whereas two images of a Samoyed and a wolf in the same position and on similar backgrounds may be very similar to each other. A linear classifier, or any other ‘shallow’ classifier operating on

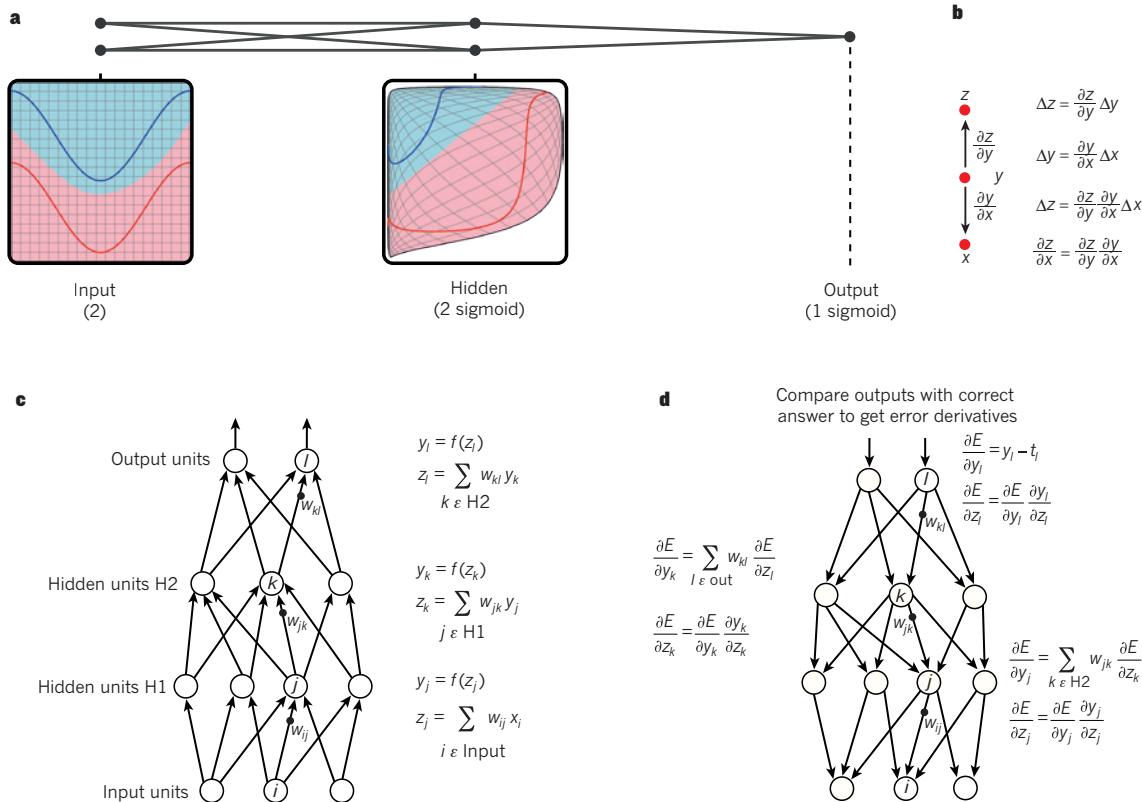


Figure 1 | Multilayer neural networks and backpropagation. **a**, A multilayer neural network (shown by the connected dots) can distort the input space to make the classes of data (examples of which are on the red and blue lines) linearly separable. Note how a regular grid (shown on the left) in input space is also transformed (shown in the middle panel) by hidden units. This is an illustrative example with only two input units, two hidden units and one output unit, but the networks used for object recognition or natural language processing contain tens or hundreds of thousands of units. Reproduced with permission from C. Olah (<http://colah.github.io/>). **b**, The chain rule of derivatives tells us how two small effects (that of a small change of x on y , and that of y on z) are composed. A small change Δx in x gets transformed first into a small change Δy in y by getting multiplied by $\partial y / \partial x$ (that is, the definition of partial derivative). Similarly, the change Δy creates a change Δz in z . Substituting one equation into the other gives the chain rule of derivatives — how Δx gets turned into Δz through multiplication by the product of $\partial y / \partial x$ and $\partial z / \partial y$. It also works when x , y and z are vectors (and the derivatives are Jacobian matrices). **c**, The equations used for computing the forward pass in a neural net with two hidden layers and one output layer, each constituting a module through

which one can backpropagate gradients. At each layer, we first compute the total input z to each unit, which is a weighted sum of the outputs of the units in the layer below. Then a non-linear function $f(\cdot)$ is applied to z to get the output of the unit. For simplicity, we have omitted bias terms. The non-linear functions used in neural networks include the rectified linear unit (ReLU) $f(z) = \max(0, z)$, commonly used in recent years, as well as the more conventional sigmoids, such as the hyperbolic tangent, $f(z) = (\exp(z) - \exp(-z)) / (\exp(z) + \exp(-z))$ and logistic function logistic, $f(z) = 1 / (1 + \exp(-z))$. **d**, The equations used for computing the backward pass. At each hidden layer we compute the error derivative with respect to the output of each unit, which is a weighted sum of the error derivatives with respect to the total inputs to the units in the layer above. We then convert the error derivative with respect to the input by multiplying it by the gradient of $f(z)$. At the output layer, the error derivative with respect to the output of a unit is computed by differentiating the cost function. This gives $y_l - t_l$ if the cost function for unit l is $0.5(y_l - t_l)^2$, where t_l is the target value. Once the $\partial E / \partial z_k$ is known, the error-derivative for the weight w_{jk} on the connection from unit j in the layer below is just $y_j \partial E / \partial z_k$.

Samoyed (16); Papillon (5.7); Pomeranian (2.7); Arctic fox (1.0); Eskimo dog (0.6); white wolf (0.4); Siberian husky (0.4)

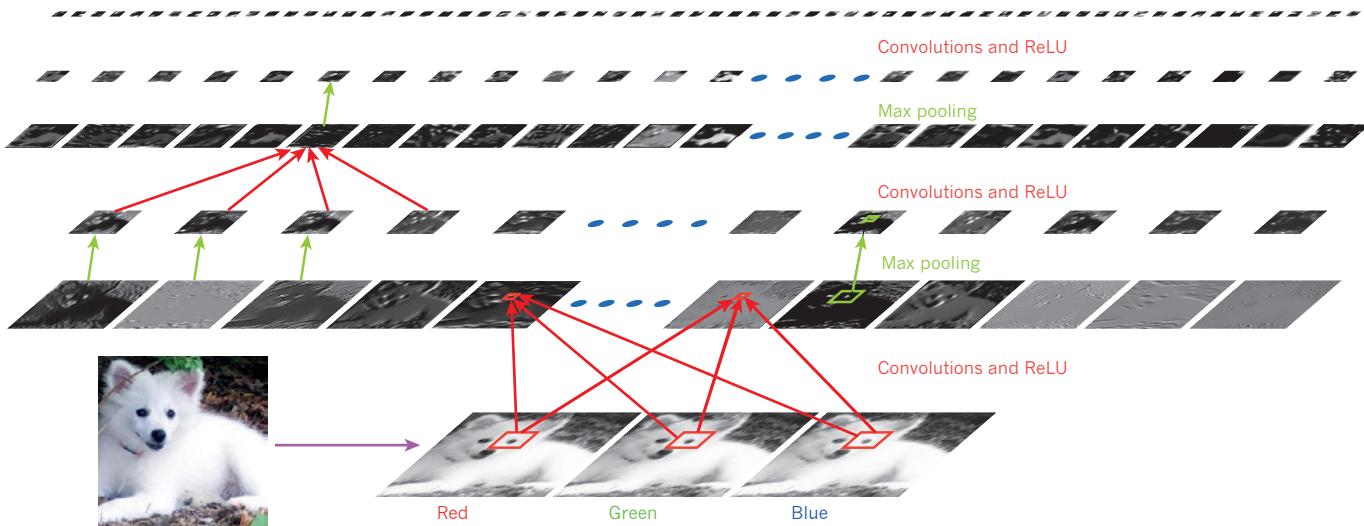


Figure 2 | Inside a convolutional network. The outputs (not the filters) of each layer (horizontally) of a typical convolutional network architecture applied to the image of a Samoyed dog (bottom left; and RGB (red, green, blue) inputs, bottom right). Each rectangular image is a feature map

raw pixels could not possibly distinguish the latter two, while putting the former two in the same category. This is why shallow classifiers require a good feature extractor that solves the selectivity–invariance dilemma — one that produces representations that are selective to the aspects of the image that are important for discrimination, but that are invariant to irrelevant aspects such as the pose of the animal. To make classifiers more powerful, one can use generic non-linear features, as with kernel methods²⁰, but generic features such as those arising with the Gaussian kernel do not allow the learner to generalize well far from the training examples²¹. The conventional option is to hand design good feature extractors, which requires a considerable amount of engineering skill and domain expertise. But this can all be avoided if good features can be learned automatically using a general-purpose learning procedure. This is the key advantage of deep learning.

A deep-learning architecture is a multilayer stack of simple modules, all (or most) of which are subject to learning, and many of which compute non-linear input–output mappings. Each module in the stack transforms its input to increase both the selectivity and the invariance of the representation. With multiple non-linear layers, say a depth of 5 to 20, a system can implement extremely intricate functions of its inputs that are simultaneously sensitive to minute details — distinguishing Samoyeds from white wolves — and insensitive to large irrelevant variations such as the background, pose, lighting and surrounding objects.

Backpropagation to train multilayer architectures

From the earliest days of pattern recognition^{22,23}, the aim of researchers has been to replace hand-engineered features with trainable multilayer networks, but despite its simplicity, the solution was not widely understood until the mid 1980s. As it turns out, multilayer architectures can be trained by simple stochastic gradient descent. As long as the modules are relatively smooth functions of their inputs and of their internal weights, one can compute gradients using the backpropagation procedure. The idea that this could be done, and that it worked, was discovered independently by several different groups during the 1970s and 1980s^{24–27}.

The backpropagation procedure to compute the gradient of an objective function with respect to the weights of a multilayer stack of modules is nothing more than a practical application of the chain

corresponding to the output for one of the learned features, detected at each of the image positions. Information flows bottom up, with lower-level features acting as oriented edge detectors, and a score is computed for each image class in output. ReLU, rectified linear unit.

rule for derivatives. The key insight is that the derivative (or gradient) of the objective with respect to the input of a module can be computed by working backwards from the gradient with respect to the output of that module (or the input of the subsequent module) (Fig. 1). The backpropagation equation can be applied repeatedly to propagate gradients through all modules, starting from the output at the top (where the network produces its prediction) all the way to the bottom (where the external input is fed). Once these gradients have been computed, it is straightforward to compute the gradients with respect to the weights of each module.

Many applications of deep learning use feedforward neural network architectures (Fig. 1), which learn to map a fixed-size input (for example, an image) to a fixed-size output (for example, a probability for each of several categories). To go from one layer to the next, a set of units compute a weighted sum of their inputs from the previous layer and pass the result through a non-linear function. At present, the most popular non-linear function is the rectified linear unit (ReLU), which is simply the half-wave rectifier $f(z) = \max(z, 0)$. In past decades, neural nets used smoother non-linearities, such as $\tanh(z)$ or $1/(1 + \exp(-z))$, but the ReLU typically learns much faster in networks with many layers, allowing training of a deep supervised network without unsupervised pre-training²⁸. Units that are not in the input or output layer are conventionally called hidden units. The hidden layers can be seen as distorting the input in a non-linear way so that categories become linearly separable by the last layer (Fig. 1).

In the late 1990s, neural nets and backpropagation were largely forsaken by the machine-learning community and ignored by the computer-vision and speech-recognition communities. It was widely thought that learning useful, multistage, feature extractors with little prior knowledge was infeasible. In particular, it was commonly thought that simple gradient descent would get trapped in poor local minima — weight configurations for which no small change would reduce the average error.

In practice, poor local minima are rarely a problem with large networks. Regardless of the initial conditions, the system nearly always reaches solutions of very similar quality. Recent theoretical and empirical results strongly suggest that local minima are not a serious issue in general. Instead, the landscape is packed with a combinatorially large number of saddle points where the gradient is zero, and the surface curves up in most dimensions and curves down in the

remainder^{29,30}. The analysis seems to show that saddle points with only a few downward curving directions are present in very large numbers, but almost all of them have very similar values of the objective function. Hence, it does not much matter which of these saddle points the algorithm gets stuck at.

Interest in deep feedforward networks was revived around 2006 (refs 31–34) by a group of researchers brought together by the Canadian Institute for Advanced Research (CIFAR). The researchers introduced unsupervised learning procedures that could create layers of feature detectors without requiring labelled data. The objective in learning each layer of feature detectors was to be able to reconstruct or model the activities of feature detectors (or raw inputs) in the layer below. By ‘pre-training’ several layers of progressively more complex feature detectors using this reconstruction objective, the weights of a deep network could be initialized to sensible values. A final layer of output units could then be added to the top of the network and the whole deep system could be fine-tuned using standard backpropagation^{33–35}. This worked remarkably well for recognizing handwritten digits or for detecting pedestrians, especially when the amount of labelled data was very limited³⁶.

The first major application of this pre-training approach was in speech recognition, and it was made possible by the advent of fast graphics processing units (GPUs) that were convenient to program³⁷ and allowed researchers to train networks 10 or 20 times faster. In 2009, the approach was used to map short temporal windows of coefficients extracted from a sound wave to a set of probabilities for the various fragments of speech that might be represented by the frame in the centre of the window. It achieved record-breaking results on a standard speech recognition benchmark that used a small vocabulary³⁸ and was quickly developed to give record-breaking results on a large vocabulary task³⁹. By 2012, versions of the deep net from 2009 were being developed by many of the major speech groups⁶ and were already being deployed in Android phones. For smaller data sets, unsupervised pre-training helps to prevent overfitting⁴⁰, leading to significantly better generalization when the number of labelled examples is small, or in a transfer setting where we have lots of examples for some ‘source’ tasks but very few for some ‘target’ tasks. Once deep learning had been rehabilitated, it turned out that the pre-training stage was only needed for small data sets.

There was, however, one particular type of deep, feedforward network that was much easier to train and generalized much better than networks with full connectivity between adjacent layers. This was the convolutional neural network (ConvNet)^{41,42}. It achieved many practical successes during the period when neural networks were out of favour and it has recently been widely adopted by the computer-vision community.

Convolutional neural networks

ConvNets are designed to process data that come in the form of multiple arrays, for example a colour image composed of three 2D arrays containing pixel intensities in the three colour channels. Many data modalities are in the form of multiple arrays: 1D for signals and sequences, including language; 2D for images or audio spectrograms; and 3D for video or volumetric images. There are four key ideas behind ConvNets that take advantage of the properties of natural signals: local connections, shared weights, pooling and the use of many layers.

The architecture of a typical ConvNet (Fig. 2) is structured as a series of stages. The first few stages are composed of two types of layers: convolutional layers and pooling layers. Units in a convolutional layer are organized in feature maps, within which each unit is connected to local patches in the feature maps of the previous layer through a set of weights called a filter bank. The result of this local weighted sum is then passed through a non-linearity such as a ReLU. All units in a feature map share the same filter bank. Different feature maps in a layer use different filter banks. The reason for

this architecture is twofold. First, in array data such as images, local groups of values are often highly correlated, forming distinctive local motifs that are easily detected. Second, the local statistics of images and other signals are invariant to location. In other words, if a motif can appear in one part of the image, it could appear anywhere, hence the idea of units at different locations sharing the same weights and detecting the same pattern in different parts of the array. Mathematically, the filtering operation performed by a feature map is a discrete convolution, hence the name.

Although the role of the convolutional layer is to detect local conjunctions of features from the previous layer, the role of the pooling layer is to merge semantically similar features into one. Because the relative positions of the features forming a motif can vary somewhat, reliably detecting the motif can be done by coarse-graining the position of each feature. A typical pooling unit computes the maximum of a local patch of units in one feature map (or in a few feature maps). Neighbouring pooling units take input from patches that are shifted by more than one row or column, thereby reducing the dimension of the representation and creating an invariance to small shifts and distortions. Two or three stages of convolution, non-linearity and pooling are stacked, followed by more convolutional and fully-connected layers. Backpropagating gradients through a ConvNet is as simple as through a regular deep network, allowing all the weights in all the filter banks to be trained.

Deep neural networks exploit the property that many natural signals are compositional hierarchies, in which higher-level features are obtained by composing lower-level ones. In images, local combinations of edges form motifs, motifs assemble into parts, and parts form objects. Similar hierarchies exist in speech and text from sounds to phones, phonemes, syllables, words and sentences. The pooling allows representations to vary very little when elements in the previous layer vary in position and appearance.

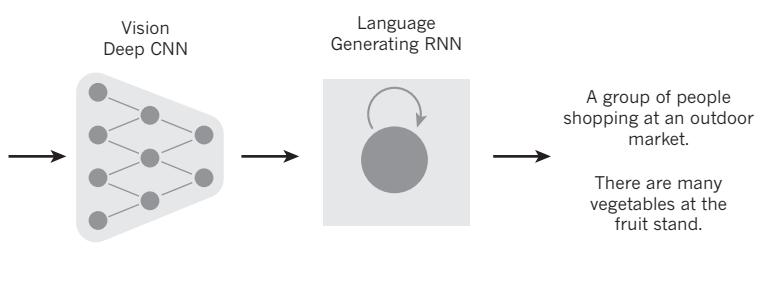
The convolutional and pooling layers in ConvNets are directly inspired by the classic notions of simple cells and complex cells in visual neuroscience⁴³, and the overall architecture is reminiscent of the LGN–V1–V2–V4–IT hierarchy in the visual cortex ventral pathway⁴⁴. When ConvNet models and monkeys are shown the same picture, the activations of high-level units in the ConvNet explains half of the variance of random sets of 160 neurons in the monkey’s inferotemporal cortex⁴⁵. ConvNets have their roots in the neocognitron⁴⁶, the architecture of which was somewhat similar, but did not have an end-to-end supervised-learning algorithm such as backpropagation. A primitive 1D ConvNet called a time-delay neural net was used for the recognition of phonemes and simple words^{47,48}.

There have been numerous applications of convolutional networks going back to the early 1990s, starting with time-delay neural networks for speech recognition⁴⁷ and document reading⁴². The document reading system used a ConvNet trained jointly with a probabilistic model that implemented language constraints. By the late 1990s this system was reading over 10% of all the cheques in the United States. A number of ConvNet-based optical character recognition and handwriting recognition systems were later deployed by Microsoft⁴⁹. ConvNets were also experimented with in the early 1990s for object detection in natural images, including faces and hands^{50,51}, and for face recognition⁵².

Image understanding with deep convolutional networks

Since the early 2000s, ConvNets have been applied with great success to the detection, segmentation and recognition of objects and regions in images. These were all tasks in which labelled data was relatively abundant, such as traffic sign recognition⁵³, the segmentation of biological images⁵⁴ particularly for connectomics⁵⁵, and the detection of faces, text, pedestrians and human bodies in natural images^{36,50,51,56–58}. A major recent practical success of ConvNets is face recognition⁵⁹.

Importantly, images can be labelled at the pixel level, which will have applications in technology, including autonomous mobile robots and



A woman is throwing a **frisbee** in a park.



A **dog** is standing on a hardwood floor.



A **stop** sign is on a road with a mountain in the background



A little **girl** sitting on a bed with a **teddy bear**.



A group of **people** sitting on a boat in the water.



A **giraffe** standing in a forest with **trees** in the background.

Figure 3 | From image to text. Captions generated by a recurrent neural network (RNN) taking, as extra input, the representation extracted by a deep convolutional neural network (CNN) from a test image, with the RNN trained to ‘translate’ high-level representations of images into captions (top). Reproduced

self-driving cars^{60,61}. Companies such as Mobileye and NVIDIA are using such ConvNet-based methods in their upcoming vision systems for cars. Other applications gaining importance involve natural language understanding¹⁴ and speech recognition⁷.

Despite these successes, ConvNets were largely forsaken by the mainstream computer-vision and machine-learning communities until the ImageNet competition in 2012. When deep convolutional networks were applied to a data set of about a million images from the web that contained 1,000 different classes, they achieved spectacular results, almost halving the error rates of the best competing approaches¹. This success came from the efficient use of GPUs, ReLUs, a new regularization technique called dropout⁶², and techniques to generate more training examples by deforming the existing ones. This success has brought about a revolution in computer vision; ConvNets are now the dominant approach for almost all recognition and detection tasks^{4,58,59,63–65} and approach human performance on some tasks. A recent stunning demonstration combines ConvNets and recurrent net modules for the generation of image captions (Fig. 3).

Recent ConvNet architectures have 10 to 20 layers of ReLUs, hundreds of millions of weights, and billions of connections between units. Whereas training such large networks could have taken weeks only two years ago, progress in hardware, software and algorithm parallelization have reduced training times to a few hours.

The performance of ConvNet-based vision systems has caused most major technology companies, including Google, Facebook,

with permission from ref. 102. When the RNN is given the ability to focus its attention on a different location in the input image (middle and bottom; the lighter patches were given more attention) as it generates each word (bold), we found⁶⁶ that it exploits this to achieve better ‘translation’ of images into captions.

Microsoft, IBM, Yahoo!, Twitter and Adobe, as well as a quickly growing number of start-ups to initiate research and development projects and to deploy ConvNet-based image understanding products and services.

ConvNets are easily amenable to efficient hardware implementations in chips or field-programmable gate arrays^{66,67}. A number of companies such as NVIDIA, Mobileye, Intel, Qualcomm and Samsung are developing ConvNet chips to enable real-time vision applications in smartphones, cameras, robots and self-driving cars.

Distributed representations and language processing

Deep-learning theory shows that deep nets have two different exponential advantages over classic learning algorithms that do not use distributed representations²¹. Both of these advantages arise from the power of composition and depend on the underlying data-generating distribution having an appropriate componential structure⁴⁰. First, learning distributed representations enable generalization to new combinations of the values of learned features beyond those seen during training (for example, 2^n combinations are possible with n binary features)^{68,69}. Second, composing layers of representation in a deep net brings the potential for another exponential advantage⁷⁰ (exponential in the depth).

The hidden layers of a multilayer neural network learn to represent the network’s inputs in a way that makes it easy to predict the target outputs. This is nicely demonstrated by training a multilayer neural network to predict the next word in a sequence from a local

context of earlier words⁷¹. Each word in the context is presented to the network as a one-of-N vector, that is, one component has a value of 1 and the rest are 0. In the first layer, each word creates a different pattern of activations, or word vectors (Fig. 4). In a language model, the other layers of the network learn to convert the input word vectors into an output word vector for the predicted next word, which can be used to predict the probability for any word in the vocabulary to appear as the next word. The network learns word vectors that contain many active components each of which can be interpreted as a separate feature of the word, as was first demonstrated²⁷ in the context of learning distributed representations for symbols. These semantic features were not explicitly present in the input. They were discovered by the learning procedure as a good way of factorizing the structured relationships between the input and output symbols into multiple ‘micro-rules’. Learning word vectors turned out to also work very well when the word sequences come from a large corpus of real text and the individual micro-rules are unreliable⁷¹. When trained to predict the next word in a news story, for example, the learned word vectors for Tuesday and Wednesday are very similar, as are the word vectors for Sweden and Norway. Such representations are called distributed representations because their elements (the features) are not mutually exclusive and their many configurations correspond to the variations seen in the observed data. These word vectors are composed of learned features that were not determined ahead of time by experts, but automatically discovered by the neural network. Vector representations of words learned from text are now very widely used in natural language applications^{14,17,72–76}.

The issue of representation lies at the heart of the debate between the logic-inspired and the neural-network-inspired paradigms for cognition. In the logic-inspired paradigm, an instance of a symbol is something for which the only property is that it is either identical or non-identical to other symbol instances. It has no internal structure that is relevant to its use; and to reason with symbols, they must be bound to the variables in judiciously chosen rules of inference. By contrast, neural networks just use big activity vectors, big weight matrices and scalar non-linearities to perform the type of fast ‘intuitive’ inference that underpins effortless commonsense reasoning.

Before the introduction of neural language models⁷¹, the standard approach to statistical modelling of language did not exploit distributed representations: it was based on counting frequencies of occurrences of short symbol sequences of length up to N (called N-grams). The number of possible N-grams is on the order of V^N , where V is the vocabulary size, so taking into account a context of more than a

handful of words would require very large training corpora. N-grams treat each word as an atomic unit, so they cannot generalize across semantically related sequences of words, whereas neural language models can because they associate each word with a vector of real valued features, and semantically related words end up close to each other in that vector space (Fig. 4).

Recurrent neural networks

When backpropagation was first introduced, its most exciting use was for training recurrent neural networks (RNNs). For tasks that involve sequential inputs, such as speech and language, it is often better to use RNNs (Fig. 5). RNNs process an input sequence one element at a time, maintaining in their hidden units a ‘state vector’ that implicitly contains information about the history of all the past elements of the sequence. When we consider the outputs of the hidden units at different discrete time steps as if they were the outputs of different neurons in a deep multilayer network (Fig. 5, right), it becomes clear how we can apply backpropagation to train RNNs.

RNNs are very powerful dynamic systems, but training them has proved to be problematic because the backpropagated gradients either grow or shrink at each time step, so over many time steps they typically explode or vanish^{77,78}.

Thanks to advances in their architecture^{79,80} and ways of training them^{81,82}, RNNs have been found to be very good at predicting the next character in the text⁸³ or the next word in a sequence⁷⁵, but they can also be used for more complex tasks. For example, after reading an English sentence one word at a time, an English ‘encoder’ network can be trained so that the final state vector of its hidden units is a good representation of the thought expressed by the sentence. This thought vector can then be used as the initial hidden state of (or as extra input to) a jointly trained French ‘decoder’ network, which outputs a probability distribution for the first word of the French translation. If a particular first word is chosen from this distribution and provided as input to the decoder network it will then output a probability distribution for the second word of the translation and so on until a full stop is chosen^{17,72,76}. Overall, this process generates sequences of French words according to a probability distribution that depends on the English sentence. This rather naive way of performing machine translation has quickly become competitive with the state-of-the-art, and this raises serious doubts about whether understanding a sentence requires anything like the internal symbolic expressions that are manipulated by using inference rules. It is more compatible with the view that everyday reasoning involves many simultaneous analogies

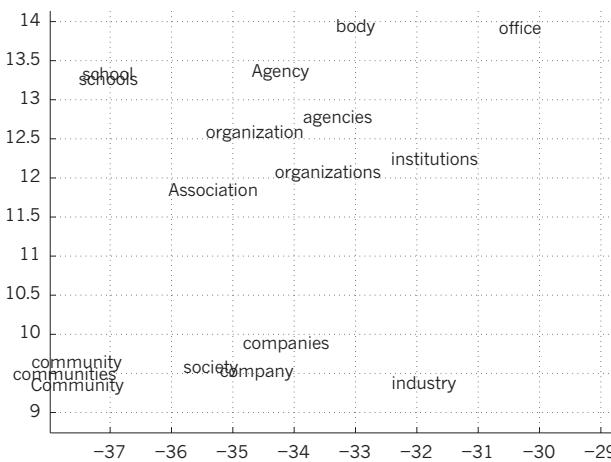


Figure 4 | Visualizing the learned word vectors. On the left is an illustration of word representations learned for modelling language, non-linearly projected to 2D for visualization using the t-SNE algorithm¹⁰³. On the right is a 2D representation of phrases learned by an English-to-French encoder-decoder recurrent neural network⁷⁵. One can observe that semantically similar words

or sequences of words are mapped to nearby representations. The distributed representations of words are obtained by using backpropagation to jointly learn a representation for each word and a function that predicts a target quantity such as the next word in a sequence (for language modelling) or a whole sequence of translated words (for machine translation)^{18,75}.

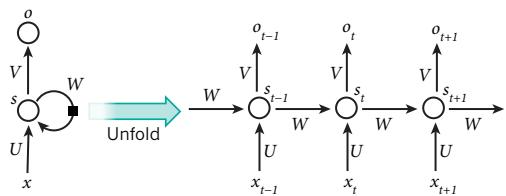


Figure 5 | A recurrent neural network and the unfolding in time of the computation involved in its forward computation. The artificial neurons (for example, hidden units grouped under node s with values s_t at time t) get inputs from other neurons at previous time steps (this is represented with the black square, representing a delay of one time step, on the left). In this way, a recurrent neural network can map an input sequence with elements x_t into an output sequence with elements o_t , with each o_t depending on all the previous x_i (for $t' \leq t$). The same parameters (matrices U, V, W) are used at each time step. Many other architectures are possible, including a variant in which the network can generate a sequence of outputs (for example, words), each of which is used as inputs for the next time step. The backpropagation algorithm (Fig. 1) can be directly applied to the computational graph of the unfolded network on the right, to compute the derivative of a total error (for example, the log-probability of generating the right sequence of outputs) with respect to all the states s , and all the parameters.

that each contribute plausibility to a conclusion^{84,85}.

Instead of translating the meaning of a French sentence into an English sentence, one can learn to ‘translate’ the meaning of an image into an English sentence (Fig. 3). The encoder here is a deep ConvNet that converts the pixels into an activity vector in its last hidden layer. The decoder is an RNN similar to the ones used for machine translation and neural language modelling. There has been a surge of interest in such systems recently (see examples mentioned in ref. 86).

RNNs, once unfolded in time (Fig. 5), can be seen as very deep feedforward networks in which all the layers share the same weights. Although their main purpose is to learn long-term dependencies, theoretical and empirical evidence shows that it is difficult to learn to store information for very long⁷⁸.

To correct for that, one idea is to augment the network with an explicit memory. The first proposal of this kind is the long short-term memory (LSTM) networks that use special hidden units, the natural behaviour of which is to remember inputs for a long time⁷⁹. A special unit called the memory cell acts like an accumulator or a gated leaky neuron: it has a connection to itself at the next time step that has a weight of one, so it copies its own real-valued state and accumulates the external signal, but this self-connection is multiplicatively gated by another unit that learns to decide when to clear the content of the memory.

LSTM networks have subsequently proved to be more effective than conventional RNNs, especially when they have several layers for each time step⁸⁷, enabling an entire speech recognition system that goes all the way from acoustics to the sequence of characters in the transcription. LSTM networks or related forms of gated units are also currently used for the encoder and decoder networks that perform so well at machine translation^{17,72,76}.

Over the past year, several authors have made different proposals to augment RNNs with a memory module. Proposals include the Neural Turing Machine in which the network is augmented by a ‘tape-like’ memory that the RNN can choose to read from or write to⁸⁸, and memory networks, in which a regular network is augmented by a kind of associative memory⁸⁹. Memory networks have yielded excellent performance on standard question-answering benchmarks. The memory is used to remember the story about which the network is later asked to answer questions.

Beyond simple memorization, neural Turing machines and memory networks are being used for tasks that would normally require reasoning and symbol manipulation. Neural Turing machines can be taught ‘algorithms’. Among other things, they can learn to output

a sorted list of symbols when their input consists of an unsorted sequence in which each symbol is accompanied by a real value that indicates its priority in the list⁸⁸. Memory networks can be trained to keep track of the state of the world in a setting similar to a text adventure game and after reading a story, they can answer questions that require complex inference⁹⁰. In one test example, the network is shown a 15-sentence version of the *The Lord of the Rings* and correctly answers questions such as “where is Frodo now?”⁸⁹.

The future of deep learning

Unsupervised learning^{91–98} had a catalytic effect in reviving interest in deep learning, but has since been overshadowed by the successes of purely supervised learning. Although we have not focused on it in this Review, we expect unsupervised learning to become far more important in the longer term. Human and animal learning is largely unsupervised: we discover the structure of the world by observing it, not by being told the name of every object.

Human vision is an active process that sequentially samples the optic array in an intelligent, task-specific way using a small, high-resolution fovea with a large, low-resolution surround. We expect much of the future progress in vision to come from systems that are trained end-to-end and combine ConvNets with RNNs that use reinforcement learning to decide where to look. Systems combining deep learning and reinforcement learning are in their infancy, but they already outperform passive vision systems⁹⁹ at classification tasks and produce impressive results in learning to play many different video games¹⁰⁰.

Natural language understanding is another area in which deep learning is poised to make a large impact over the next few years. We expect systems that use RNNs to understand sentences or whole documents will become much better when they learn strategies for selectively attending to one part at a time^{76,86}.

Ultimately, major progress in artificial intelligence will come about through systems that combine representation learning with complex reasoning. Although deep learning and simple reasoning have been used for speech and handwriting recognition for a long time, new paradigms are needed to replace rule-based manipulation of symbolic expressions by operations on large vectors¹⁰¹. ■

Received 25 February; accepted 1 May 2015.

- Krizhevsky, A., Sutskever, I. & Hinton, G. ImageNet classification with deep convolutional neural networks. In *Proc. Advances in Neural Information Processing Systems 25* 1090–1098 (2012). **This report was a breakthrough that used convolutional nets to almost halve the error rate for object recognition, and precipitated the rapid adoption of deep learning by the computer vision community.**
- Farabet, C., Couprie, C., Najman, L. & LeCun, Y. Learning hierarchical features for scene labeling. *IEEE Trans. Pattern Anal. Mach. Intell.* **35**, 1915–1929 (2013).
- Tompson, J., Jain, A., LeCun, Y. & Bregler, C. Joint training of a convolutional network and a graphical model for human pose estimation. In *Proc. Advances in Neural Information Processing Systems 27* 1799–1807 (2014).
- Szegedy, C. et al. Going deeper with convolutions. Preprint at <http://arxiv.org/abs/1409.4842> (2014).
- Mikolov, T., Deoras, A., Povey, D., Burget, L. & Cernocky, J. Strategies for training large scale neural network language models. In *Proc. Automatic Speech Recognition and Understanding* 196–201 (2011).
- Hinton, G. et al. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal Processing Magazine* **29**, 82–97 (2012). **This joint paper from the major speech recognition laboratories, summarizing the breakthrough achieved with deep learning on the task of phonetic classification for automatic speech recognition, was the first major industrial application of deep learning.**
- Sainath, T., Mohamed, A.-R., Kingsbury, B. & Ramabhadran, B. Deep convolutional neural networks for LVCSR. In *Proc. Acoustics, Speech and Signal Processing* 8614–8618 (2013).
- Ma, J., Sheridan, R. P., Liaw, A., Dahl, G. E. & Svetnik, V. Deep neural nets as a method for quantitative structure-activity relationships. *J. Chem. Inf. Model.* **55**, 263–274 (2015).
- Ciodaro, T., Deva, D., de Seixas, J. & Damazio, D. Online particle detection with neural networks based on topological calorimetry information. *J. Phys. Conf. Series* **368**, 012030 (2012).
- Kaggle. Higgs boson machine learning challenge. Kaggle <https://www.kaggle.com/c/higgs-boson> (2014).
- Helmaedter, M. et al. Connectomic reconstruction of the inner plexiform layer in the mouse retina. *Nature* **500**, 168–174 (2013).

12. Leung, M. K., Xiong, H. Y., Lee, L. J. & Frey, B. J. Deep learning of the tissue-regulated splicing code. *Bioinformatics* **30**, i121–i129 (2014).
13. Xiong, H. Y. et al. The human splicing code reveals new insights into the genetic determinants of disease. *Science* **347**, 6218 (2015).
14. Collobert, R., et al. Natural language processing (almost) from scratch. *J. Mach. Learn. Res.* **12**, 2493–2537 (2011).
15. Bordes, A., Chopra, S. & Weston, J. Question answering with subgraph embeddings. In *Proc. Empirical Methods in Natural Language Processing* <http://arxiv.org/abs/1406.3676v3> (2014).
16. Jean, S., Cho, K., Memisevic, R. & Bengio, Y. On using very large target vocabulary for neural machine translation. In *Proc. ACL-IJCNLP* <http://arxiv.org/abs/1412.2007> (2015).
17. Sutskever, I., Vinyals, O. & Le, Q. V. Sequence to sequence learning with neural networks. In *Proc. Advances in Neural Information Processing Systems* **27** 3104–3112 (2014). **This paper showed state-of-the-art machine translation results with the architecture introduced in ref. 72, with a recurrent network trained to read a sentence in one language, produce a semantic representation of its meaning, and generate a translation in another language.**
18. Bottou, L. & Bousquet, O. The tradeoffs of large scale learning. In *Proc. Advances in Neural Information Processing Systems* **20** 161–168 (2007).
19. Duda, R. O. & Hart, P. E. *Pattern Classification and Scene Analysis* (Wiley, 1973).
20. Schölkopf, B. & Smola, A. *Learning with Kernels* (MIT Press, 2002).
21. Bengio, Y., Delalleau, O. & Le Roux, N. The curse of highly variable functions for local kernel machines. In *Proc. Advances in Neural Information Processing Systems* **18** 107–114 (2005).
22. Selfridge, O. G. Pandemonium: a paradigm for learning in mechanisation of thought processes. In *Proc. Symposium on Mechanisation of Thought Processes* 513–526 (1958).
23. Rosenblatt, F. *The Perceptron — A Perceiving and Recognizing Automaton*. Tech. Rep. 85-460-1 (Cornell Aeronautical Laboratory, 1957).
24. Werbos, P. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard Univ. (1974).
25. Parker, D. B. *Learning Logic Report TR-47* (MIT Press, 1985).
26. LeCun, Y. Une procédure d'apprentissage pour Réseau à seuil assymétrique in *Cognitiva 85: à la Frontière de l'Intelligence Artificielle, des Sciences de la Connaissance et des Neurosciences* [in French] 599–604 (1985).
27. Rumelhart, D. E., Hinton, G. E. & Williams, R. J. Learning representations by back-propagating errors. *Nature* **323**, 533–536 (1986).
28. Glorot, X., Bordes, A. & Bengio, Y. Deep sparse rectifier neural networks. In *Proc. 14th International Conference on Artificial Intelligence and Statistics* 315–323 (2011). **This paper showed that supervised training of very deep neural networks is much faster if the hidden layers are composed of ReLU.**
29. Dauphin, Y. et al. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Proc. Advances in Neural Information Processing Systems* **27** 2933–2941 (2014).
30. Choromanska, A., Henaff, M., Mathieu, M., Arous, G. B. & LeCun, Y. The loss surface of multilayer networks. In *Proc. Conference on AI and Statistics* <http://arxiv.org/abs/1412.0233> (2014).
31. Hinton, G. E. What kind of graphical model is the brain? In *Proc. 19th International Joint Conference on Artificial intelligence* 1765–1775 (2005).
32. Hinton, G. E., Osindero, S. & Teh, Y.-W. A fast learning algorithm for deep belief nets. *Neural Comput.* **18**, 1527–1554 (2006). **This paper introduced a novel and effective way of training very deep neural networks by pre-training one hidden layer at a time using the unsupervised learning procedure for restricted Boltzmann machines.**
33. Bengio, Y., Lamblin, P., Popovici, D. & Larochelle, H. Greedy layer-wise training of deep networks. In *Proc. Advances in Neural Information Processing Systems* **19** 153–160 (2006). **This report demonstrated that the unsupervised pre-training method introduced in ref. 32 significantly improves performance on test data and generalizes the method to other unsupervised representation-learning techniques, such as auto-encoders.**
34. Ranzato, M., Poultney, C., Chopra, S. & LeCun, Y. Efficient learning of sparse representations with an energy-based model. In *Proc. Advances in Neural Information Processing Systems* **19** 1137–1144 (2006).
35. Hinton, G. E. & Salakhutdinov, R. Reducing the dimensionality of data with neural networks. *Science* **313**, 504–507 (2006).
36. Sermanet, P., Kavukcuoglu, K., Chintala, S. & LeCun, Y. Pedestrian detection with unsupervised multi-stage feature learning. In *Proc. International Conference on Computer Vision and Pattern Recognition* <http://arxiv.org/abs/1212.0142> (2013).
37. Raina, R., Madhavan, A. & Ng, A. Y. Large-scale deep unsupervised learning using graphics processors. In *Proc. 26th Annual International Conference on Machine Learning* 873–880 (2009).
38. Mohamed, A.-R., Dahl, G. E. & Hinton, G. Acoustic modeling using deep belief networks. *IEEE Trans. Audio Speech Lang. Process.* **20**, 14–22 (2012).
39. Dahl, G. E., Yu, D., Deng, L. & Acero, A. Context-dependent pre-trained deep neural networks for large vocabulary speech recognition. *IEEE Trans. Audio Speech Lang. Process.* **20**, 33–42 (2012).
40. Bengio, Y., Courville, A. & Vincent, P. Representation learning: a review and new perspectives. *IEEE Trans. Pattern Anal. Machine Intell.* **35**, 1798–1828 (2013). **This paper introduced neural language models, which learn to convert a word symbol into a word vector or word embedding composed of learned semantic features in order to predict the next word in a sequence.**
41. LeCun, Y. et al. Handwritten digit recognition with a back-propagation network. In *Proc. Advances in Neural Information Processing Systems* 396–404 (1990). **This is the first paper on convolutional networks trained by backpropagation for the task of classifying low-resolution images of handwritten digits.**
42. LeCun, Y., Bottou, L., Bengio, Y. & Haffner, P. Gradient-based learning applied to document recognition. *Proc. IEEE* **86**, 2278–2324 (1998). **This overview paper on the principles of end-to-end training of modular systems such as deep neural networks using gradient-based optimization showed how neural networks (and in particular convolutional nets) can be combined with search or inference mechanisms to model complex outputs that are interdependent, such as sequences of characters associated with the content of a document.**
43. Hubel, D. H. & Wiesel, T. N. Receptive fields, binocular interaction, and functional architecture in the cat's visual cortex. *J. Physiol.* **160**, 106–154 (1962).
44. Fellman, D. J. & Essen, D. C. V. Distributed hierarchical processing in the primate cerebral cortex. *Cereb. Cortex* **1**, 1–47 (1991).
45. Cadieu, C. F. et al. Deep neural networks rival the representation of primate it cortex for core visual object recognition. *PLoS Comp. Biol.* **10**, e1003963 (2014).
46. Fukushima, K. & Miyake, S. Neocognitron: a new algorithm for pattern recognition tolerant of deformations and shifts in position. *Pattern Recognition* **15**, 455–469 (1982).
47. Waibel, A., Hanazawa, T., Hinton, G. E., Shikano, K. & Lang, K. Phoneme recognition using time-delay neural networks. *IEEE Trans. Acoustics Speech Signal Process.* **37**, 328–339 (1989).
48. Bottou, L., Fogelman-Soulie, F., Blanchet, P. & Lienard, J. Experiments with time delay networks and dynamic time warping for speaker independent isolated digit recognition. In *Proc. EuroSpeech* 89 537–540 (1989).
49. Simard, D., Steinhaus, P. Y. & Platt, J. C. Best practices for convolutional neural networks. In *Proc. Document Analysis and Recognition* 958–963 (2003).
50. Vaillant, R., Monrocq, C. & LeCun, Y. Original approach for the localisation of objects in images. In *Proc. Vision, Image, and Signal Processing* **141**, 245–250 (1994).
51. Nowlan, S. & Platt, J. In *Neural Information Processing Systems* 901–908 (1995).
52. Lawrence, S., Giles, C. L., Tsoi, A. C. & Back, A. D. Face recognition: a convolutional neural-network approach. *IEEE Trans. Neural Networks* **8**, 98–113 (1997).
53. Ciresan, D., Meier, U., Masci, J. & Schmidhuber, J. Multi-column deep neural network for traffic sign classification. *Neural Networks* **32**, 333–338 (2012).
54. Ning, F. et al. Toward automatic phenotyping of developing embryos from videos. *IEEE Trans. Image Process.* **14**, 1360–1371 (2005).
55. Turaga, S. C. et al. Convolutional networks can learn to generate affinity graphs for image segmentation. *Neural Comput.* **22**, 511–538 (2010).
56. Garcia, C. & Delakis, M. Convolutional face finder: a neural architecture for fast and robust face detection. *IEEE Trans. Pattern Anal. Machine Intell.* **26**, 1408–1423 (2004).
57. Osadchy, M., LeCun, Y. & Miller, M. Synergistic face detection and pose estimation with energy-based models. *J. Mach. Learn. Res.* **8**, 1197–1215 (2007).
58. Tompson, J., Goroshin, R. R., Jain, A., LeCun, Y. Y. & Bregler, C. C. Efficient object localization using convolutional networks. In *Proc. Conference on Computer Vision and Pattern Recognition* <http://arxiv.org/abs/1411.4280> (2014).
59. Taigman, Y., Yang, M., Ranzato, M. & Wolf, L. Deepface: closing the gap to human-level performance in face verification. In *Proc. Conference on Computer Vision and Pattern Recognition* 1701–1708 (2014).
60. Hadsell, R. et al. Learning long-range vision for autonomous off-road driving. *J. Field Robot.* **26**, 120–144 (2009).
61. Farabet, C., Courville, C., Najman, L. & LeCun, Y. Scene parsing with multiscale feature learning, purity trees, and optimal covers. In *Proc. International Conference on Machine Learning* <http://arxiv.org/abs/1202.2160> (2012).
62. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. & Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *J. Machine Learning Res.* **15**, 1929–1958 (2014).
63. Sermanet, P. et al. Overfeat: integrated recognition, localization and detection using convolutional networks. In *Proc. International Conference on Learning Representations* <http://arxiv.org/abs/1312.6229> (2014).
64. Girshick, R., Donahue, J., Darrell, T. & Malik, J. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proc. Conference on Computer Vision and Pattern Recognition* 580–587 (2014).
65. Simonyan, K. & Zisserman, A. Very deep convolutional networks for large-scale image recognition. In *Proc. International Conference on Learning Representations* <http://arxiv.org/abs/1409.1556> (2014).
66. Boser, B., Sackinger, E., Bromley, J., LeCun, Y. & Jackel, L. An analog neural network processor with programmable topology. *J. Solid State Circuits* **26**, 2017–2025 (1991).
67. Farabet, C. et al. Large-scale FPGA-based convolutional networks. In *Scaling up Machine Learning: Parallel and Distributed Approaches* (eds Bekkerman, R., Bilenko, M. & Langford, J.) 399–419 (Cambridge Univ. Press, 2011).
68. Bengio, Y. *Learning Deep Architectures for AI* (Now, 2009).
69. Montufar, G. & Morton, J. When does a mixture of products contain a product of mixtures? *J. Discrete Math.* **29**, 321–347 (2014).
70. Montufar, G. F., Pascanu, R., Cho, K. & Bengio, Y. On the number of linear regions of deep neural networks. In *Proc. Advances in Neural Information Processing Systems* **27** 2924–2932 (2014).
71. Bengio, Y., Ducharme, R. & Vincent, P. A neural probabilistic language model. In *Proc. Advances in Neural Information Processing Systems* **13** 932–938 (2001). **This paper introduced neural language models, which learn to convert a word symbol into a word vector or word embedding composed of learned semantic features in order to predict the next word in a sequence.**
72. Cho, K. et al. Learning phrase representations using RNN encoder-decoder

- for statistical machine translation. In *Proc. Conference on Empirical Methods in Natural Language Processing* 1724–1734 (2014).
73. Schwenk, H. Continuous space language models. *Computer Speech Lang.* **21**, 492–518 (2007).
 74. Socher, R., Lin, C.-Y., Manning, C. & Ng, A. Y. Parsing natural scenes and natural language with recursive neural networks. In *Proc. International Conference on Machine Learning* 129–136 (2011).
 75. Mikolov, T., Sutskever, I., Chen, K., Corrado, G. & Dean, J. Distributed representations of words and phrases and their compositionality. In *Proc. Advances in Neural Information Processing Systems* 26 3111–3119 (2013).
 76. Bahdanau, D., Cho, K. & Bengio, Y. Neural machine translation by jointly learning to align and translate. In *Proc. International Conference on Learning Representations* <http://arxiv.org/abs/1409.0473> (2015).
 77. Hochreiter, S. Untersuchungen zu dynamischen neuronalen Netzen [in German] Diploma thesis, T.U. Münich (1991).
 78. Bengio, Y., Simard, P. & Frasconi, P. Learning long-term dependencies with gradient descent is difficult. *IEEE Trans. Neural Networks* **5**, 157–166 (1994).
 79. Hochreiter, S. & Schmidhuber, J. Long short-term memory. *Neural Comput.* **9**, 1735–1780 (1997).
 - This paper introduced LSTM recurrent networks, which have become a crucial ingredient in recent advances with recurrent networks because they are good at learning long-range dependencies.
 80. ElHilli, S. & Bengio, Y. Hierarchical recurrent neural networks for long-term dependencies. In *Proc. Advances in Neural Information Processing Systems* 8 <http://papers.nips.cc/paper/1102-hierarchical-recurrent-neural-networks-for-long-term-dependencies> (1995).
 81. Sutskever, I. *Training Recurrent Neural Networks*. PhD thesis, Univ. Toronto (2012).
 82. Pascanu, R., Mikolov, T. & Bengio, Y. On the difficulty of training recurrent neural networks. In *Proc. 30th International Conference on Machine Learning* 1310–1318 (2013).
 83. Sutskever, I., Martens, J. & Hinton, G. E. Generating text with recurrent neural networks. In *Proc. 28th International Conference on Machine Learning* 1017–1024 (2011).
 84. Lakoff, G. & Johnson, M. *Metaphors We Live By* (Univ. Chicago Press, 2008).
 85. Rogers, T. T. & McClelland, J. L. *Semantic Cognition: A Parallel Distributed Processing Approach* (MIT Press, 2004).
 86. Xu, K. et al. Show, attend and tell: Neural image caption generation with visual attention. In *Proc. International Conference on Learning Representations* <http://arxiv.org/abs/1502.03044> (2015).
 87. Graves, A., Mohamed, A.-R. & Hinton, G. Speech recognition with deep recurrent neural networks. In *Proc. International Conference on Acoustics, Speech and Signal Processing* 6645–6649 (2013).
 88. Graves, A., Wayne, G. & Danihelka, I. Neural Turing machines. <http://arxiv.org/abs/1410.5401> (2014).
 89. Weston, J., Chopra, S. & Bordes, A. Memory networks. <http://arxiv.org/abs/1410.3916> (2014).
 90. Weston, J., Bordes, A., Chopra, S. & Mikolov, T. Towards AI-complete question answering: a set of prerequisite toy tasks. <http://arxiv.org/abs/1502.05698> (2015).
 91. Hinton, G. E., Dayan, P., Frey, B. J. & Neal, R. M. The wake-sleep algorithm for unsupervised neural networks. *Science* **268**, 1558–1161 (1995).
 92. Salakhutdinov, R. & Hinton, G. Deep Boltzmann machines. In *Proc. International Conference on Artificial Intelligence and Statistics* 448–455 (2009).
 93. Vincent, P., Larochelle, H., Bengio, Y. & Manzagol, P.-A. Extracting and composing robust features with denoising autoencoders. In *Proc. 25th International Conference on Machine Learning* 1096–1103 (2008).
 94. Kavukcuoglu, K. et al. Learning convolutional feature hierarchies for visual recognition. In *Proc. Advances in Neural Information Processing Systems* 23 1090–1098 (2010).
 95. Gregor, K. & LeCun, Y. Learning fast approximations of sparse coding. In *Proc. International Conference on Machine Learning* 399–406 (2010).
 96. Ranzato, M., Mnih, V., Susskind, J. M. & Hinton, G. E. Modeling natural images using gated MRFs. *IEEE Trans. Pattern Anal. Machine Intell.* **35**, 2206–2222 (2013).
 97. Bengio, Y., Thibodeau-Laufer, E., Alain, G. & Yosinski, J. Deep generative stochastic networks trainable by backprop. In *Proc. 31st International Conference on Machine Learning* 226–234 (2014).
 98. Kingma, D., Rezende, D., Mohamed, S. & Welling, M. Semi-supervised learning with deep generative models. In *Proc. Advances in Neural Information Processing Systems* 27 3581–3589 (2014).
 99. Ba, J., Mnih, V. & Kavukcuoglu, K. Multiple object recognition with visual attention. In *Proc. International Conference on Learning Representations* <http://arxiv.org/abs/1412.7755> (2014).
 100. Mnih, V. et al. Human-level control through deep reinforcement learning. *Nature* **518**, 529–533 (2015).
 101. Bottou, L. From machine learning to machine reasoning. *Mach. Learn.* **94**, 133–149 (2014).
 102. Vinyals, O., Toshev, A., Bengio, S. & Erhan, D. Show and tell: a neural image caption generator. In *Proc. International Conference on Machine Learning* <http://arxiv.org/abs/1502.03044> (2014).
 103. van der Maaten, L. & Hinton, G. E. Visualizing data using t-SNE. *J. Mach. Learn. Research* **9**, 2579–2605 (2008).

Acknowledgements The authors would like to thank the Natural Sciences and Engineering Research Council of Canada, the Canadian Institute For Advanced Research (CIFAR), the National Science Foundation and Office of Naval Research for support. Y.L. and Y.B. are CIFAR fellows.

Author Information Reprints and permissions information is available at www.nature.com/reprints. The authors declare no competing financial interests. Readers are welcome to comment on the online version of this paper at go.nature.com/7cjbaa. Correspondence should be addressed to Y.L. (yann@cs.nyu.edu).

word2vec Parameter Learning Explained

Xin Rong
ronxin@umich.edu

Abstract

The word2vec model and application by Mikolov et al. have attracted a great amount of attention in recent two years. The vector representations of words learned by word2vec models have been shown to carry semantic meanings and are useful in various NLP tasks. As an increasing number of researchers would like to experiment with word2vec or similar techniques, I notice that there lacks a material that comprehensively explains the parameter learning process of word embedding models in details, thus preventing researchers that are non-experts in neural networks from understanding the working mechanism of such models.

This note provides detailed derivations and explanations of the parameter update equations of the word2vec models, including the original continuous bag-of-word (CBOW) and skip-gram (SG) models, as well as advanced optimization techniques, including hierarchical softmax and negative sampling. Intuitive interpretations of the gradient equations are also provided alongside mathematical derivations.

In the appendix, a review on the basics of neuron networks and backpropagation is provided. I also created an interactive demo, wevi, to facilitate the intuitive understanding of the model.¹

1 Continuous Bag-of-Word Model

1.1 One-word context

We start from the simplest version of the continuous bag-of-word model (CBOW) introduced in Mikolov et al. (2013a). We assume that there is only one word considered per context, which means the model will predict one target word given one context word, which is like a bigram model. For readers who are new to neural networks, it is recommended that one go through Appendix A for a quick review of the important concepts and terminologies before proceeding further.

Figure 1 shows the network model under the simplified context definition². In our setting, the vocabulary size is V , and the hidden layer size is N . The units on adjacent

¹An online interactive demo is available at: <http://bit.ly/wevi-online>.

²In Figures 1, 2, 3, and the rest of this note, \mathbf{W}' is not the transpose of \mathbf{W} , but a different matrix instead.

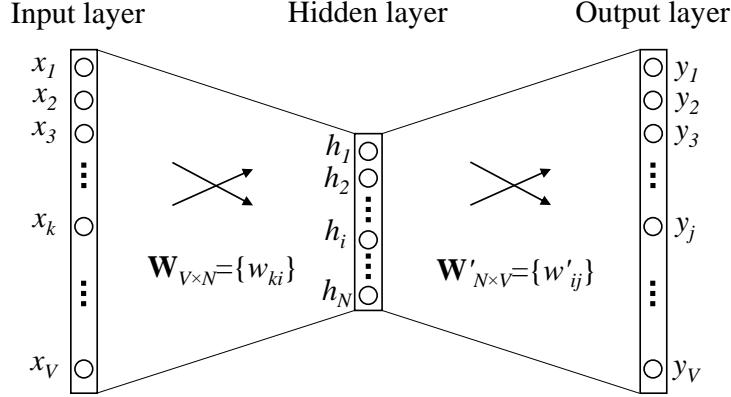


Figure 1: A simple CBOW model with only one word in the context

layers are fully connected. The input is a one-hot encoded vector, which means for a given input context word, only one out of V units, $\{x_1, \dots, x_V\}$, will be 1, and all other units are 0.

The weights between the input layer and the output layer can be represented by a $V \times N$ matrix \mathbf{W} . Each row of \mathbf{W} is the N -dimension vector representation \mathbf{v}_w of the associated word of the input layer. Formally, row i of \mathbf{W} is \mathbf{v}_w^T . Given a context (a word), assuming $x_k = 1$ and $x_{k'} = 0$ for $k' \neq k$, we have

$$\mathbf{h} = \mathbf{W}^T \mathbf{x} = \mathbf{W}_{(k,\cdot)}^T := \mathbf{v}_{w_I}^T, \quad (1)$$

which is essentially copying the k -th row of \mathbf{W} to \mathbf{h} . \mathbf{v}_{w_I} is the vector representation of the input word w_I . This implies that the link (activation) function of the hidden layer units is simply *linear* (i.e., directly passing its weighted sum of inputs to the next layer).

From the hidden layer to the output layer, there is a different weight matrix $\mathbf{W}' = \{w'_{ij}\}$, which is an $N \times V$ matrix. Using these weights, we can compute a score u_j for each word in the vocabulary,

$$u_j = \mathbf{v}_{w_j}^T \mathbf{h}, \quad (2)$$

where \mathbf{v}_{w_j} is the j -th column of the matrix \mathbf{W}' . Then we can use softmax, a log-linear classification model, to obtain the posterior distribution of words, which is a multinomial distribution.

$$p(w_j | w_I) = y_j = \frac{\exp(u_j)}{\sum_{j'=1}^V \exp(u_{j'})}, \quad (3)$$

where y_j is the output of the j -th unit in the output layer. Substituting (1) and (2) into

(3), we obtain

$$p(w_j|w_I) = \frac{\exp\left(\mathbf{v}'_{w_j}^T \mathbf{v}_{w_I}\right)}{\sum_{j'=1}^V \exp\left(\mathbf{v}'_{w_{j'}}^T \mathbf{v}_{w_I}\right)} \quad (4)$$

Note that \mathbf{v}_w and \mathbf{v}'_w are two representations of the word w . \mathbf{v}_w comes from rows of \mathbf{W} , which is the input→hidden weight matrix, and \mathbf{v}'_w comes from columns of \mathbf{W}' , which is the hidden→output matrix. In subsequent analysis, we call \mathbf{v}_w as the “**input vector**”, and \mathbf{v}'_w as the “**output vector**” of the word w .

Update equation for hidden→output weights

Let us now derive the weight update equation for this model. Although the actual computation is impractical (explained below), we are doing the derivation to gain insights on this original model with no tricks applied. For a review of basics of backpropagation, see Appendix A.

The training objective (for one training sample) is to maximize (4), the conditional probability of observing the actual output word w_O (denote its index in the output layer as j^*) given the input context word w_I with regard to the weights.

$$\max p(w_O|w_I) = \max y_{j^*} \quad (5)$$

$$= \max \log y_{j^*} \quad (6)$$

$$= u_{j^*} - \log \sum_{j'=1}^V \exp(u_{j'}) := -E, \quad (7)$$

where $E = -\log p(w_O|w_I)$ is our loss function (we want to minimize E), and j^* is the index of the actual output word in the output layer. Note that this loss function can be understood as a special case of the cross-entropy measurement between two probabilistic distributions.

Let us now derive the update equation of the weights between hidden and output layers. Take the derivative of E with regard to j -th unit's net input u_j , we obtain

$$\frac{\partial E}{\partial u_j} = y_j - t_j := e_j \quad (8)$$

where $t_j = \mathbb{1}(j = j^*)$, i.e., t_j will only be 1 when the j -th unit is the actual output word, otherwise $t_j = 0$. Note that this derivative is simply the prediction error e_j of the output layer.

Next we take the derivative on w'_{ij} to obtain the gradient on the hidden→output weights.

$$\frac{\partial E}{\partial w'_{ij}} = \frac{\partial E}{\partial u_j} \cdot \frac{\partial u_j}{\partial w'_{ij}} = e_j \cdot h_i \quad (9)$$

Therefore, using stochastic gradient descent, we obtain the weight updating equation for hidden→output weights:

$$w'_{ij}^{(\text{new})} = w'_{ij}^{(\text{old})} - \eta \cdot e_j \cdot h_i. \quad (10)$$

or

$$\mathbf{v}'_{w_j}^{(\text{new})} = \mathbf{v}'_{w_j}^{(\text{old})} - \eta \cdot e_j \cdot \mathbf{h} \quad \text{for } j = 1, 2, \dots, V. \quad (11)$$

where $\eta > 0$ is the learning rate, $e_j = y_j - t_j$, and h_i is the i -th unit in the hidden layer; \mathbf{v}'_{w_j} is the output vector of w_j . Note that this update equation implies that we have to go through every possible word in the vocabulary, check its output probability y_j , and compare y_j with its expected output t_j (either 0 or 1). If $y_j > t_j$ (“overestimating”), then we subtract a proportion of the hidden vector \mathbf{h} (i.e., \mathbf{v}_{w_I}) from \mathbf{v}'_{w_j} , thus making \mathbf{v}'_{w_j} farther away from \mathbf{v}_{w_I} ; if $y_j < t_j$ (“underestimating”, which is true only if $t_j = 1$, i.e., $w_j = w_O$), we add some \mathbf{h} to \mathbf{v}'_{w_O} , thus making \mathbf{v}'_{w_O} closer³ to \mathbf{v}_{w_I} . If y_j is very close to t_j , then according to the update equation, very little change will be made to the weights. Note, again, that \mathbf{v}_w (input vector) and \mathbf{v}'_w (output vector) are two different vector representations of the word w .

Update equation for input→hidden weights

Having obtained the update equations for \mathbf{W}' , we can now move on to \mathbf{W} . We take the derivative of E on the output of the hidden layer, obtaining

$$\frac{\partial E}{\partial h_i} = \sum_{j=1}^V \frac{\partial E}{\partial u_j} \cdot \frac{\partial u_j}{\partial h_i} = \sum_{j=1}^V e_j \cdot w'_{ij} := \text{EH}_i \quad (12)$$

where h_i is the output of the i -th unit of the hidden layer; u_j is defined in (2), the net input of the j -th unit in the output layer; and $e_j = y_j - t_j$ is the prediction error of the j -th word in the output layer. EH, an N -dim vector, is the sum of the output vectors of all words in the vocabulary, weighted by their prediction error.

Next we should take the derivative of E on \mathbf{W} . First, recall that the hidden layer performs a linear computation on the values from the input layer. Expanding the vector notation in (1) we get

$$h_i = \sum_{k=1}^V x_k \cdot w_{ki} \quad (13)$$

Now we can take the derivative of E with regard to each element of \mathbf{W} , obtaining

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial E}{\partial h_i} \cdot \frac{\partial h_i}{\partial w_{ki}} = \text{EH}_i \cdot x_k \quad (14)$$

³Here when I say “closer” or “farther”, I meant using the inner product instead of Euclidean as the distance measurement.

This is equivalent to the tensor product of \mathbf{x} and \mathbf{EH} , i.e.,

$$\frac{\partial E}{\partial \mathbf{W}} = \mathbf{x} \otimes \mathbf{EH} = \mathbf{x}\mathbf{EH}^T \quad (15)$$

from which we obtain a $V \times N$ matrix. Since only one component of \mathbf{x} is non-zero, only one row of $\frac{\partial E}{\partial \mathbf{W}}$ is non-zero, and the value of that row is \mathbf{EH}^T , an N -dim vector. We obtain the update equation of \mathbf{W} as

$$\mathbf{v}_{w_I}^{(\text{new})} = \mathbf{v}_{w_I}^{(\text{old})} - \eta \mathbf{EH}^T \quad (16)$$

where \mathbf{v}_{w_I} is a row of \mathbf{W} , the “input vector” of the only context word, and is the only row of \mathbf{W} whose derivative is non-zero. All the other rows of \mathbf{W} will remain unchanged after this iteration, because their derivatives are zero.

Intuitively, since vector \mathbf{EH} is the sum of output vectors of all words in vocabulary weighted by their prediction error $e_j = y_j - t_j$, we can understand (16) as adding a portion of every output vector in vocabulary to the input vector of the context word. If, in the output layer, the probability of a word w_j being the output word is overestimated ($y_j > t_j$), then the input vector of the context word w_I will tend to move farther away from the output vector of w_j ; conversely if the probability of w_j being the output word is underestimated ($y_j < t_j$), then the input vector w_I will tend to move closer to the output vector of w_j ; if the probability of w_j is fairly accurately predicted, then it will have little effect on the movement of the input vector of w_I . The movement of the input vector of w_I is determined by the prediction error of all vectors in the vocabulary; the larger the prediction error, the more significant effects a word will exert on the movement on the input vector of the context word.

As we iteratively update the model parameters by going through context-target word pairs generated from a training corpus, the effects on the vectors will accumulate. We can imagine that the output vector of a word w is “dragged” back-and-forth by the input vectors of w 's co-occurring neighbors, as if there are physical strings between the vector of w and the vectors of its neighbors. Similarly, an input vector can also be considered as being dragged by many output vectors. This interpretation can remind us of gravity, or force-directed graph layout. The equilibrium length of each imaginary string is related to the strength of cooccurrence between the associated pair of words, as well as the learning rate. After many iterations, the relative positions of the input and output vectors will eventually stabilize.

1.2 Multi-word context

Figure 2 shows the CBOW model with a multi-word context setting. When computing the hidden layer output, instead of directly copying the input vector of the input context word, the CBOW model takes the average of the vectors of the input context words, and

use the product of the input→hidden weight matrix and the average vector as the output.

$$\mathbf{h} = \frac{1}{C} \mathbf{W}^T (\mathbf{x}_1 + \mathbf{x}_2 + \cdots + \mathbf{x}_C) \quad (17)$$

$$= \frac{1}{C} (\mathbf{v}_{w_1} + \mathbf{v}_{w_2} + \cdots + \mathbf{v}_{w_C})^T \quad (18)$$

where C is the number of words in the context, w_1, \dots, w_C are the words in the context, and \mathbf{v}_w is the input vector of a word w . The loss function is

$$E = -\log p(w_O | w_{I,1}, \dots, w_{I,C}) \quad (19)$$

$$= -u_{j^*} + \log \sum_{j'=1}^V \exp(u_{j'}) \quad (20)$$

$$= -\mathbf{v}'_{w_O}^T \cdot \mathbf{h} + \log \sum_{j'=1}^V \exp(\mathbf{v}'_{w_j}^T \cdot \mathbf{h}) \quad (21)$$

which is the same as (7), the objective of the one-word-context model, except that \mathbf{h} is different, as defined in (18) instead of (1).

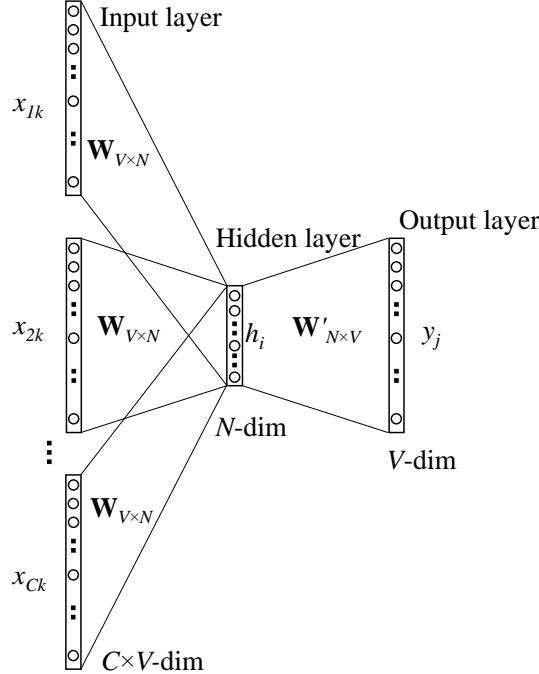


Figure 2: Continuous bag-of-word model

The update equation for the hidden→output weights stay the same as that for the one-word-context model (11). We copy it here:

$$\mathbf{v}'_{w_j}^{(\text{new})} = \mathbf{v}'_{w_j}^{(\text{old})} - \eta \cdot e_j \cdot \mathbf{h} \quad \text{for } j = 1, 2, \dots, V. \quad (22)$$

Note that we need to apply this to every element of the hidden→output weight matrix for each training instance.

The update equation for input→hidden weights is similar to (16), except that now we need to apply the following equation for every word $w_{I,c}$ in the context:

$$\mathbf{v}_{w_{I,c}}^{(\text{new})} = \mathbf{v}_{w_{I,c}}^{(\text{old})} - \frac{1}{C} \cdot \eta \cdot \mathbf{E}\mathbf{H}^T \quad \text{for } c = 1, 2, \dots, C. \quad (23)$$

where $\mathbf{v}_{w_{I,c}}$ is the input vector of the c -th word in the input context; η is a positive learning rate; and $\mathbf{E}\mathbf{H} = \frac{\partial E}{\partial h_i}$ is given by (12). The intuitive understanding of this update equation is the same as that for (16).

2 Skip-Gram Model

The skip-gram model is introduced in Mikolov et al. (2013a,b). Figure 3 shows the skip-gram model. It is the opposite of the CBOW model. The target word is now at the input layer, and the context words are on the output layer.

We still use \mathbf{v}_{w_I} to denote the input vector of the only word on the input layer, and thus we have the same definition of the hidden-layer outputs \mathbf{h} as in (1), which means \mathbf{h} is simply copying (and transposing) a row of the input→hidden weight matrix, \mathbf{W} , associated with the input word w_I . We copy the definition of \mathbf{h} below:

$$\mathbf{h} = \mathbf{W}_{(k,\cdot)}^T := \mathbf{v}_{w_I}^T, \quad (24)$$

On the output layer, instead of outputting one multinomial distribution, we are outputting C multinomial distributions. Each output is computed using the same hidden→output matrix:

$$p(w_{c,j} = w_{O,c} | w_I) = y_{c,j} = \frac{\exp(u_{c,j})}{\sum_{j'=1}^V \exp(u_{j'})} \quad (25)$$

where $w_{c,j}$ is the j -th word on the c -th panel of the output layer; $w_{O,c}$ is the actual c -th word in the output context words; w_I is the only input word; $y_{c,j}$ is the output of the j -th unit on the c -th panel of the output layer; $u_{c,j}$ is the net input of the j -th unit on the c -th panel of the output layer. Because the output layer panels share the same weights, thus

$$u_{c,j} = u_j = \mathbf{v}'_{w_j}^T \cdot \mathbf{h}, \quad \text{for } c = 1, 2, \dots, C \quad (26)$$

where \mathbf{v}'_{w_j} is the output vector of the j -th word in the vocabulary, w_j , and also \mathbf{v}'_{w_j} is taken from a column of the hidden→output weight matrix, \mathbf{W}' .

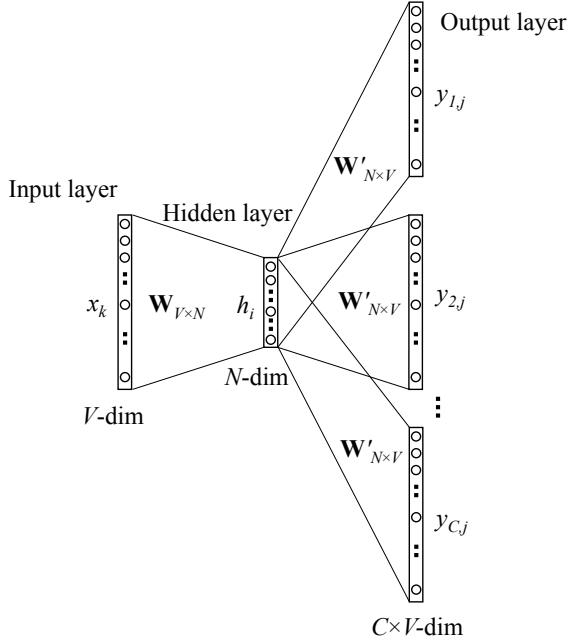


Figure 3: The skip-gram model.

The derivation of parameter update equations is not so different from the one-word-context model. The loss function is changed to

$$E = -\log p(w_{O,1}, w_{O,2}, \dots, w_{O,C} | w_I) \quad (27)$$

$$= -\log \prod_{c=1}^C \frac{\exp(u_{c,j_c^*})}{\sum_{j'=1}^V \exp(u_{j'})} \quad (28)$$

$$= -\sum_{c=1}^C u_{j_c^*} + C \cdot \log \sum_{j'=1}^V \exp(u_{j'}) \quad (29)$$

where j_c^* is the index of the actual c -th output context word in the vocabulary.

We take the derivative of E with regard to the net input of every unit on every panel of the output layer, $u_{c,j}$ and obtain

$$\frac{\partial E}{\partial u_{c,j}} = y_{c,j} - t_{c,j} := e_{c,j} \quad (30)$$

which is the prediction error on the unit, the same as in (8). For notation simplicity, we define a V -dimensional vector $\text{EI} = \{\text{EI}_1, \dots, \text{EI}_V\}$ as the sum of prediction errors over all

context words:

$$\text{EI}_j = \sum_{c=1}^C e_{c,j} \quad (31)$$

Next, we take the derivative of E with regard to the hidden→output matrix \mathbf{W}' , and obtain

$$\frac{\partial E}{\partial w'_{ij}} = \sum_{c=1}^C \frac{\partial E}{\partial u_{c,j}} \cdot \frac{\partial u_{c,j}}{\partial w'_{ij}} = \text{EI}_j \cdot h_i \quad (32)$$

Thus we obtain the update equation for the hidden→output matrix \mathbf{W}' ,

$$w'_{ij}^{(\text{new})} = w'_{ij}^{(\text{old})} - \eta \cdot \text{EI}_j \cdot h_i \quad (33)$$

or

$$\mathbf{v}'_{w_j}^{(\text{new})} = \mathbf{v}'_{w_j}^{(\text{old})} - \eta \cdot \text{EI}_j \cdot \mathbf{h} \quad \text{for } j = 1, 2, \dots, V. \quad (34)$$

The intuitive understanding of this update equation is the same as that for (11), except that the prediction error is summed across all context words in the output layer. Note that we need to apply this update equation for every element of the hidden→output matrix for each training instance.

The derivation of the update equation for the input→hidden matrix is identical to (12) to (16), except taking into account that the prediction error e_j is replaced with EI_j . We directly give the update equation:

$$\mathbf{v}_{w_I}^{(\text{new})} = \mathbf{v}_{w_I}^{(\text{old})} - \eta \cdot \text{EH}^T \quad (35)$$

where EH is an N -dim vector, each component of which is defined as

$$\text{EH}_i = \sum_{j=1}^V \text{EI}_j \cdot w'_{ij}. \quad (36)$$

The intuitive understanding of (35) is the same as that for (16).

3 Optimizing Computational Efficiency

So far the models we have discussed (“bigram” model, CBOW and skip-gram) are both in their original forms, without any efficiency optimization tricks being applied.

For all these models, there exist two vector representations for each word in the vocabulary: the input vector \mathbf{v}_w , and the output vector \mathbf{v}'_w . Learning the input vectors is cheap; but learning the output vectors is very expensive. From the update equations (22) and (33), we can find that, in order to update \mathbf{v}'_w , for each training instance, we have to iterate through every word w_j in the vocabulary, compute their net input u_j , probability

prediction y_j (or $y_{c,j}$ for skip-gram), their prediction error e_j (or EI_j for skip-gram), and finally use their prediction error to update their output vector \mathbf{v}'_j .

Doing such computations for all words for every training instance is very expensive, making it impractical to scale up to large vocabularies or large training corpora. To solve this problem, an intuition is to limit the number of output vectors that must be updated per training instance. One elegant approach to achieving this is hierarchical softmax; another approach is through sampling, which will be discussed in the next section.

Both tricks optimize only the computation of the updates for output vectors. In our derivations, we care about three values: (1) E , the new objective function; (2) $\frac{\partial E}{\partial \mathbf{v}'_w}$, the new update equation for the output vectors; and (3) $\frac{\partial E}{\partial \mathbf{h}}$, the weighted sum of predictions errors to be backpropagated for updating input vectors.

3.1 Hierarchical Softmax

Hierarchical softmax is an efficient way of computing softmax (Morin and Bengio, 2005; Mnih and Hinton, 2009). The model uses a binary tree to represent all words in the vocabulary. The V words must be leaf units of the tree. It can be proved that there are $V - 1$ inner units. For each leaf unit, there exists a unique path from the root to the unit; and this path is used to estimate the probability of the word represented by the leaf unit. See Figure 4 for an example tree.

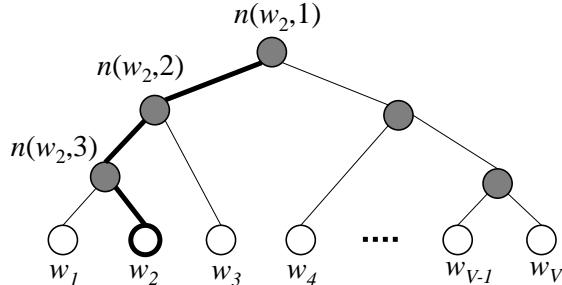


Figure 4: An example binary tree for the hierarchical softmax model. The white units are words in the vocabulary, and the dark units are inner units. An example path from root to w_2 is highlighted. In the example shown, the length of the path $L(w_2) = 4$. $n(w, j)$ means the j -th unit on the path from root to the word w .

In the hierarchical softmax model, there is no output vector representation for words. Instead, each of the $V - 1$ inner units has an output vector $\mathbf{v}'_{n(w,j)}$. And the probability of a word being the output word is defined as

$$p(w = w_O) = \prod_{j=1}^{L(w)-1} \sigma \left(\llbracket n(w, j+1) = \text{ch}(n(w, j)) \rrbracket \cdot \mathbf{v}'_{n(w,j)}^T \mathbf{h} \right) \quad (37)$$

where $\text{ch}(n)$ is the left child of unit n ; $\mathbf{v}'_{n(w,j)}$ is the vector representation (“output vector”) of the inner unit $n(w,j)$; \mathbf{h} is the output value of the hidden layer (in the skip-gram model $\mathbf{h} = \mathbf{v}_{w_I}$; and in CBOW, $\mathbf{h} = \frac{1}{C} \sum_{c=1}^C \mathbf{v}_{w_c}$); $\llbracket x \rrbracket$ is a special function defined as

$$\llbracket x \rrbracket = \begin{cases} 1 & \text{if } x \text{ is true;} \\ -1 & \text{otherwise.} \end{cases} \quad (38)$$

Let us intuitively understand the equation by going through an example. Looking at Figure 4, suppose we want to compute the probability that w_2 being the output word. We define this probability as the probability of a random walk starting from the root ending at the leaf unit in question. At each inner unit (including the root unit), we need to assign the probabilities of going left and going right.⁴ We define the probability of going left at an inner unit n to be

$$p(n, \text{left}) = \sigma \left(\mathbf{v}'_n^T \cdot \mathbf{h} \right) \quad (39)$$

which is determined by both the vector representation of the inner unit, and the hidden layer’s output value (which is then determined by the vector representation of the input word(s)). Apparently the probability of going right at unit n is

$$p(n, \text{right}) = 1 - \sigma \left(\mathbf{v}'_n^T \cdot \mathbf{h} \right) = \sigma \left(-\mathbf{v}'_n^T \cdot \mathbf{h} \right) \quad (40)$$

Following the path from the root to w_2 in Figure 4, we can compute the probability of w_2 being the output word as

$$p(w_2 = w_O) = p(n(w_2, 1), \text{left}) \cdot p(n(w_2, 2), \text{left}) \cdot p(n(w_2, 3), \text{right}) \quad (41)$$

$$= \sigma \left(\mathbf{v}'_{n(w_2,1)}^T \mathbf{h} \right) \cdot \sigma \left(\mathbf{v}'_{n(w_2,2)}^T \mathbf{h} \right) \cdot \sigma \left(-\mathbf{v}'_{n(w_2,3)}^T \mathbf{h} \right) \quad (42)$$

which is exactly what is given by (37). It should not be hard to verify that

$$\sum_{i=1}^V p(w_i = w_O) = 1 \quad (43)$$

making the hierarchical softmax a well defined multinomial distribution among all words.

Now let us derive the parameter update equation for the vector representations of the inner units. For simplicity, we look at the one-word context model first. Extending the update equations to CBOW and skip-gram models is easy.

For the simplicity of notation, we define the following shortenizations without introducing ambiguity:

$$\llbracket \cdot \rrbracket := \llbracket n(w, j + 1) = \text{ch}(n(w, j)) \rrbracket \quad (44)$$

⁴While an inner unit of a binary tree may not always have both children, a binary Huffman tree’s inner units always do. Although theoretically one can use many different types of trees for hierarchical softmax, word2vec uses a binary Huffman tree for fast training.

$$\mathbf{v}'_j := \mathbf{v}'_{n_{w,j}} \quad (45)$$

For a training instance, the error function is defined as

$$E = -\log p(w = w_O | w_I) = -\sum_{j=1}^{L(w)-1} \log \sigma(\|\cdot\| \mathbf{v}'_j^T \mathbf{h}) \quad (46)$$

We take the derivative of E with regard to $\mathbf{v}'_j \mathbf{h}$, obtaining

$$\frac{\partial E}{\partial \mathbf{v}'_j \mathbf{h}} = (\sigma(\|\cdot\| \mathbf{v}'_j^T \mathbf{h}) - 1) \|\cdot\| \quad (47)$$

$$= \begin{cases} \sigma(\mathbf{v}'_j^T \mathbf{h}) - 1 & (\|\cdot\| = 1) \\ \sigma(\mathbf{v}'_j^T \mathbf{h}) & (\|\cdot\| = -1) \end{cases} \quad (48)$$

$$= \sigma(\mathbf{v}'_j^T \mathbf{h}) - t_j \quad (49)$$

where $t_j = 1$ if $\|\cdot\| = 1$ and $t_j = 0$ otherwise.

Next we take the derivative of E with regard to the vector representation of the inner unit $n(w, j)$ and obtain

$$\frac{\partial E}{\partial \mathbf{v}'_j} = \frac{\partial E}{\partial \mathbf{v}'_j \mathbf{h}} \cdot \frac{\partial \mathbf{v}'_j \mathbf{h}}{\partial \mathbf{v}'_j} = (\sigma(\mathbf{v}'_j^T \mathbf{h}) - t_j) \cdot \mathbf{h} \quad (50)$$

which results in the following update equation:

$$\mathbf{v}'_j^{(\text{new})} = \mathbf{v}'_j^{(\text{old})} - \eta (\sigma(\mathbf{v}'_j^T \mathbf{h}) - t_j) \cdot \mathbf{h} \quad (51)$$

which should be applied to $j = 1, 2, \dots, L(w) - 1$. We can understand $\sigma(\mathbf{v}'_j^T \mathbf{h}) - t_j$ as the prediction error for the inner unit $n(w, j)$. The “task” for each inner unit is to predict whether it should follow the left child or the right child in the random walk. $t_j = 1$ means the ground truth is to follow the left child; $t_j = 0$ means it should follow the right child. $\sigma(\mathbf{v}'_j^T \mathbf{h})$ is the prediction result. For a training instance, if the prediction of the inner unit is very close to the ground truth, then its vector representation \mathbf{v}'_j will move very little; otherwise \mathbf{v}'_j will move in an appropriate direction by moving (either closer or farther away⁵ from \mathbf{h}) so as to reduce the prediction error for this instance. This update equation can be used for both CBOW and the skip-gram model. When used for the skip-gram model, we need to repeat this update procedure for each of the C words in the output context.

⁵Again, the distance measurement is inner product.

In order to backpropagate the error to learn input→hidden weights, we take the derivative of E with regard to the output of the hidden layer and obtain

$$\frac{\partial E}{\partial \mathbf{h}} = \sum_{j=1}^{L(w)-1} \frac{\partial E}{\partial \mathbf{v}'_j \mathbf{h}} \cdot \frac{\partial \mathbf{v}'_j \mathbf{h}}{\partial \mathbf{h}} \quad (52)$$

$$= \sum_{j=1}^{L(w)-1} (\sigma(\mathbf{v}'_j^T \mathbf{h}) - t_j) \cdot \mathbf{v}'_j \quad (53)$$

$$:= \text{EH} \quad (54)$$

which can be directly substituted into (23) to obtain the update equation for the input vectors of CBOW. For the skip-gram model, we need to calculate a EH value for each word in the skip-gram context, and plug the sum of the EH values into (35) to obtain the update equation for the input vector.

From the update equations, we can see that the computational complexity per training instance per context word is reduced from $O(V)$ to $O(\log(V))$, which is a big improvement in speed. We still have roughly the same number of parameters ($V-1$ vectors for inner-units compared to originally V output vectors for words).

3.2 Negative Sampling

The idea of negative sampling is more straightforward than hierarchical softmax: in order to deal with the difficulty of having too many output vectors that need to be updated per iteration, we only update a sample of them.

Apparently the output word (i.e., the ground truth, or positive sample) should be kept in our sample and gets updated, and we need to sample a few words as negative samples (hence “negative sampling”). A probabilistic distribution is needed for the sampling process, and it can be arbitrarily chosen. We call this distribution the noise distribution, and denote it as $P_n(w)$. One can determine a good distribution empirically.⁶

In word2vec, instead of using a form of negative sampling that produces a well-defined posterior multinomial distribution, the authors argue that the following simplified training objective is capable of producing high-quality word embeddings:⁷

$$E = -\log \sigma(\mathbf{v}'_{w_O}^T \mathbf{h}) - \sum_{w_j \in \mathcal{W}_{\text{neg}}} \log \sigma(-\mathbf{v}'_{w_j}^T \mathbf{h}) \quad (55)$$

where w_O is the output word (i.e., the positive sample), and \mathbf{v}'_{w_O} is its output vector; \mathbf{h} is the output value of the hidden layer: $\mathbf{h} = \frac{1}{C} \sum_{c=1}^C \mathbf{v}_{w_c}$ in the CBOW model and $\mathbf{h} = \mathbf{v}_{w_I}$

⁶As described in (Mikolov et al., 2013b), word2vec uses a unigram distribution raised to the $\frac{3}{4}$ th power for the best quality of results.

⁷Goldberg and Levy (2014) provide a theoretical analysis on the reason of using this objective function.

in the skip-gram model; $\mathcal{W}_{\text{neg}} = \{w_j | j = 1, \dots, K\}$ is the set of words that are sampled based on $P_n(w)$, i.e., negative samples.

To obtain the update equations of the word vectors under negative sampling, we first take the derivative of E with regard to the net input of the output unit w_j :

$$\frac{\partial E}{\partial \mathbf{v}'_{w_j}^T \mathbf{h}} = \begin{cases} \sigma(\mathbf{v}'_{w_j}^T \mathbf{h}) - 1 & \text{if } w_j = w_O \\ \sigma(\mathbf{v}'_{w_j}^T \mathbf{h}) & \text{if } w_j \in \mathcal{W}_{\text{neg}} \end{cases} \quad (56)$$

$$= \sigma(\mathbf{v}'_{w_j}^T \mathbf{h}) - t_j \quad (57)$$

where t_j is the “label” of word w_j . $t = 1$ when w_j is a positive sample; $t = 0$ otherwise. Next we take the derivative of E with regard to the output vector of the word w_j ,

$$\frac{\partial E}{\partial \mathbf{v}'_{w_j}} = \frac{\partial E}{\partial \mathbf{v}'_{w_j}^T \mathbf{h}} \cdot \frac{\partial \mathbf{v}'_{w_j}^T \mathbf{h}}{\partial \mathbf{v}'_{w_j}} = (\sigma(\mathbf{v}'_{w_j}^T \mathbf{h}) - t_j) \mathbf{h} \quad (58)$$

which results in the following update equation for its output vector:

$$\mathbf{v}'_{w_j}^{(\text{new})} = \mathbf{v}'_{w_j}^{(\text{old})} - \eta (\sigma(\mathbf{v}'_{w_j}^T \mathbf{h}) - t_j) \mathbf{h} \quad (59)$$

which only needs to be applied to $w_j \in \{w_O\} \cup \mathcal{W}_{\text{neg}}$ instead of every word in the vocabulary. This shows why we may save a significant amount of computational effort per iteration.

The intuitive understanding of the above update equation should be the same as that of (11). This equation can be used for both CBOW and the skip-gram model. For the skip-gram model, we apply this equation for one context word at a time.

To backpropagate the error to the hidden layer and thus update the input vectors of words, we need to take the derivative of E with regard to the hidden layer’s output, obtaining

$$\frac{\partial E}{\partial \mathbf{h}} = \sum_{w_j \in \{w_O\} \cup \mathcal{W}_{\text{neg}}} \frac{\partial E}{\partial \mathbf{v}'_{w_j}^T \mathbf{h}} \cdot \frac{\partial \mathbf{v}'_{w_j}^T \mathbf{h}}{\partial \mathbf{h}} \quad (60)$$

$$= \sum_{w_j \in \{w_O\} \cup \mathcal{W}_{\text{neg}}} (\sigma(\mathbf{v}'_{w_j}^T \mathbf{h}) - t_j) \mathbf{v}'_{w_j} := \text{EH} \quad (61)$$

By plugging EH into (23) we obtain the update equation for the input vectors of the CBOW model. For the skip-gram model, we need to calculate a EH value for each word in the skip-gram context, and plug the sum of the EH values into (35) to obtain the update equation for the input vector.

Acknowledgement

The author would like to thank Eytan Adar, Qiaozhu Mei, Jian Tang, Dragomir Radev, Daniel Pressel, Thomas Dean, Sudeep Gandhe, Peter Lau, Luheng He, Tomas Mikolov, Hao Jiang, and Oded Shmueli for discussions on the topic and/or improving the writing of the note.

References

- Goldberg, Y. and Levy, O. (2014). word2vec explained: deriving mikolov et al.’s negative-sampling word-embedding method. *arXiv:1402.3722 [cs, stat]*. arXiv: 1402.3722.
- Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013a). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.
- Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. (2013b). Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems*, pages 3111–3119.
- Mnih, A. and Hinton, G. E. (2009). A scalable hierarchical distributed language model. In Koller, D., Schuurmans, D., Bengio, Y., and Bottou, L., editors, *Advances in Neural Information Processing Systems 21*, pages 1081–1088. Curran Associates, Inc.
- Morin, F. and Bengio, Y. (2005). Hierarchical probabilistic neural network language model. In *AISTATS*, volume 5, pages 246–252. Citeseer.

A Back Propagation Basics

A.1 Learning Algorithms for a Single Unit

Figure 5 shows an artificial neuron (unit). $\{x_1, \dots, x_K\}$ are input values; $\{w_1, \dots, w_K\}$ are weights; y is a scalar output; and f is the link function (also called activation/decision/transfer function).

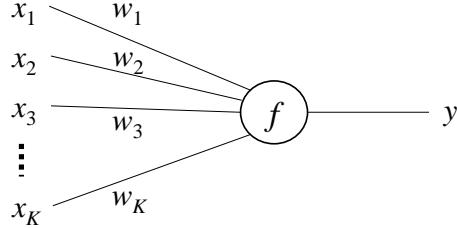


Figure 5: An artificial neuron

The unit works in the following way:

$$y = f(u), \quad (62)$$

where u is a scalar number, which is the net input (or “new input”) of the neuron. u is defined as

$$u = \sum_{i=0}^K w_i x_i. \quad (63)$$

Using vector notation, we can write

$$u = \mathbf{w}^T \mathbf{x} \quad (64)$$

Note that here we ignore the bias term in u . To include a bias term, one can simply add an input dimension (e.g., x_0) that is constant 1.

Apparently, different link functions result in distinct behaviors of the neuron. We discuss two example choices of link functions here.

The first example choice of $f(u)$ is the **unit step function** (aka **Heaviside step function**):

$$f(u) = \begin{cases} 1 & \text{if } u > 0 \\ 0 & \text{otherwise} \end{cases} \quad (65)$$

A neuron with this link function is called a perceptron. The learning algorithm for a perceptron is the perceptron algorithm. Its update equation is defined as:

$$\mathbf{w}^{(\text{new})} = \mathbf{w}^{(\text{old})} - \eta \cdot (y - t) \cdot \mathbf{x} \quad (66)$$

where t is the label (gold standard) and η is the learning rate ($\eta > 0$). Note that a perceptron is a linear classifier, which means its description capacity can be very limited. If we want to fit more complex functions, we need to use a non-linear model.

The second example choice of $f(u)$ is the **logistic function** (a most common kind of **sigmoid function**), defined as

$$\sigma(u) = \frac{1}{1 + e^{-u}} \quad (67)$$

The logistic function has two primary good properties: (1) the output y is always between 0 and 1, and (2) unlike a unit step function, $\sigma(u)$ is smooth and differentiable, making the derivation of update equation very easy.

Note that $\sigma(u)$ also has the following two properties that can be very convenient and will be used in our subsequent derivations:

$$\sigma(-u) = 1 - \sigma(u) \quad (68)$$

$$\frac{d\sigma(u)}{du} = \sigma(u)\sigma(-u) \quad (69)$$

We use stochastic gradient descent as the learning algorithm of this model. In order to derive the update equation, we need to define the error function, i.e., the training objective. The following objective function seems to be convenient:

$$E = \frac{1}{2}(t - y)^2 \quad (70)$$

We take the derivative of E with regard to w_i ,

$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial u} \cdot \frac{\partial u}{\partial w_i} \quad (71)$$

$$= (y - t) \cdot y(1 - y) \cdot x_i \quad (72)$$

where $\frac{\partial y}{\partial u} = y(1 - y)$ because $y = f(u) = \sigma(u)$, and (68) and (69). Once we have the derivative, we can apply stochastic gradient descent:

$$\mathbf{w}^{(\text{new})} = \mathbf{w}^{(\text{old})} - \eta \cdot (y - t) \cdot y(1 - y) \cdot \mathbf{x}. \quad (73)$$

A.2 Back-propagation with Multi-Layer Network

Figure 6 shows a multi-layer neural network with an input layer $\{x_k\} = \{x_1, \dots, x_K\}$, a hidden layer $\{h_i\} = \{h_1, \dots, h_N\}$, and an output layer $\{y_j\} = \{y_1, \dots, y_M\}$. For clarity we use k, i, j as the subscript for input, hidden, and output layer units respectively. We use u_i and u'_j to denote the net input of hidden layer units and output layer units respectively.

We want to derive the update equation for learning the weights w_{ki} between the input and hidden layers, and w'_{ij} between the hidden and output layers. We assume that all the

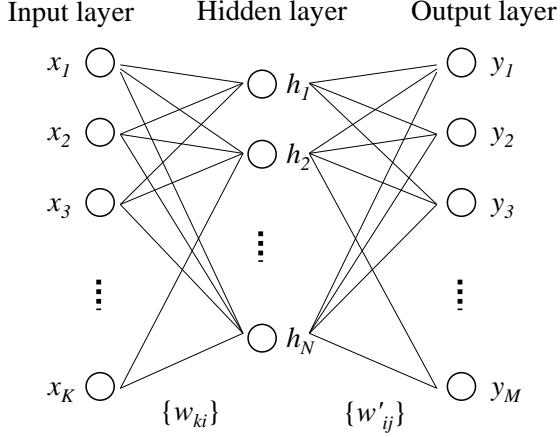


Figure 6: A multi-layer neural network with one hidden layer

computation units (i.e., units in the hidden layer and the output layer) use the logistic function $\sigma(u)$ as the link function. Therefore, for a unit h_i in the hidden layer, its output is defined as

$$h_i = \sigma(u_i) = \sigma\left(\sum_{k=1}^K w_{ki}x_k\right). \quad (74)$$

Similarly, for a unit y_j in the output layer, its output is defined as

$$y_j = \sigma(u'_j) = \sigma\left(\sum_{i=1}^N w'_{ij}h_i\right). \quad (75)$$

We use the squared sum error function given by

$$E(\mathbf{x}, \mathbf{t}, \mathbf{W}, \mathbf{W}') = \frac{1}{2} \sum_{j=1}^M (y_j - t_j)^2, \quad (76)$$

where $\mathbf{W} = \{w_{ki}\}$, a $K \times N$ weight matrix (input-hidden), and $\mathbf{W}' = \{w'_{ij}\}$, a $N \times M$ weight matrix (hidden-output). $\mathbf{t} = \{t_1, \dots, t_M\}$, a M -dimension vector, which is the gold-standard labels of output.

To obtain the update equations for w_{ki} and w'_{ij} , we simply need to take the derivative of the error function E with regard to the weights respectively. To make the derivation straightforward, we do start computing the derivative for the right-most layer (i.e., the output layer), and then move left. For each layer, we split the computation into three steps, computing the derivative of the error with regard to the output, net input, and weight respectively. This process is shown below.

We start with the output layer. The first step is to compute the derivative of the error w.r.t. the output:

$$\frac{\partial E}{\partial y_j} = y_j - t_j. \quad (77)$$

The second step is to compute the derivative of the error with regard to the net input of the output layer. Note that when taking derivatives with regard to something, we need to keep everything else fixed. Also note that this value is very important because it will be reused multiple times in subsequent computations. We denote it as EI'_j for simplicity.

$$\frac{\partial E}{\partial u'_j} = \frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial u'_j} = (y_j - t_j) \cdot y_j(1 - y_j) := \text{EI}'_j \quad (78)$$

The third step is to compute the derivative of the error with regard to the weight between the hidden layer and the output layer.

$$\frac{\partial E}{\partial w'_{ij}} = \frac{\partial E}{\partial u'_j} \cdot \frac{\partial u'_j}{\partial w'_{ij}} = \text{EI}'_j \cdot h_i \quad (79)$$

So far, we have obtained the update equation for weights between the hidden layer and the output layer.

$$w'_{ij}^{(\text{new})} = w'_{ij}^{(\text{old})} - \eta \cdot \frac{\partial E}{\partial w'_{ij}} \quad (80)$$

$$= w'_{ij}^{(\text{old})} - \eta \cdot \text{EI}'_j \cdot h_i. \quad (81)$$

where $\eta > 0$ is the learning rate.

We can repeat the same three steps to obtain the update equation for weights of the previous layer, which is essentially the idea of back propagation.

We repeat the first step and compute the derivative of the error with regard to the output of the hidden layer. Note that the output of the hidden layer is related to all units in the output layer.

$$\frac{\partial E}{\partial h_i} = \sum_{j=1}^M \frac{\partial E}{\partial u'_j} \frac{\partial u'_j}{\partial h_i} = \sum_{j=1}^M \text{EI}'_j \cdot w'_{ij}. \quad (82)$$

Then we repeat the second step above to compute the derivative of the error with regard to the net input of the hidden layer. This value is again very important, and we denote it as EI_i .

$$\frac{\partial E}{\partial u_i} = \frac{\partial E}{\partial h_i} \cdot \frac{\partial h_i}{\partial u_i} = \sum_{j=1}^M \text{EI}'_j \cdot w'_{ij} \cdot h_i(1 - h_i) := \text{EI}_i \quad (83)$$

Next we repeat the third step above to compute the derivative of the error with regard to the weights between the input layer and the hidden layer.

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial E}{\partial u_i} \cdot \frac{\partial u_i}{\partial w_{ki}} = \text{EI}_i \cdot x_k, \quad (84)$$

Finally, we can obtain the update equation for weights between the input layer and the hidden layer.

$$w_{ki}^{(\text{new})} = w_{ki}^{(\text{old})} - \eta \cdot EI_i \cdot x_k. \quad (85)$$

From the above example, we can see that the intermediate results (EI'_j) when computing the derivatives for one layer can be reused for the previous layer. Imagine there were another layer prior to the input layer, then EI_i can also be reused to continue computing the chain of derivatives efficiently. Compare Equations (78) and (83), we may find that in (83), the factor $\sum_{j=1}^M EI'_j w'_{ij}$ is just like the “error” of the hidden layer unit h_i . We may interpret this term as the error “back-propagated” from the next layer, and this propagation may go back further if the network has more hidden layers.

B wevi: Word Embedding Visual Inspector

An interactive visual interface, wevi (word embedding visual inspector), is available online to demonstrate the working mechanism of the models described in this paper. See Figure 7 for a screenshot of wevi.

The demo allows the user to visually examine the movement of input vectors and output vectors as each training instance is consumed. The training process can be also run in batch mode (e.g., consuming 500 training instances in a row), which can reveal the emergence of patterns in the weight matrices and the corresponding word vectors. Principal component analysis (PCA) is employed to visualize the “high”-dimensional vectors in a 2D scatter plot. The demo supports both CBOW and skip-gram models.

After training the model, the user can manually *activate* one or multiple input-layer units, and inspect which hidden-layer units and output-layer units become active. The user can also customize training data, hidden layer size, and learning rate. Several preset training datasets are provided, which can generate different results that seem interesting, such as using a toy vocabulary to reproduce the famous word analogy: *king - queen = man - woman*.

It is hoped that by interacting with this demo one can quickly gain insights of the working mechanism of the model. The system is available at <http://bit.ly/wevi-online>. The source code is available at <http://github.com/ronxin/wevi>.

wevi: word embedding visual inspector

[Everything you need to know about this tool](#) - [Source code](#)

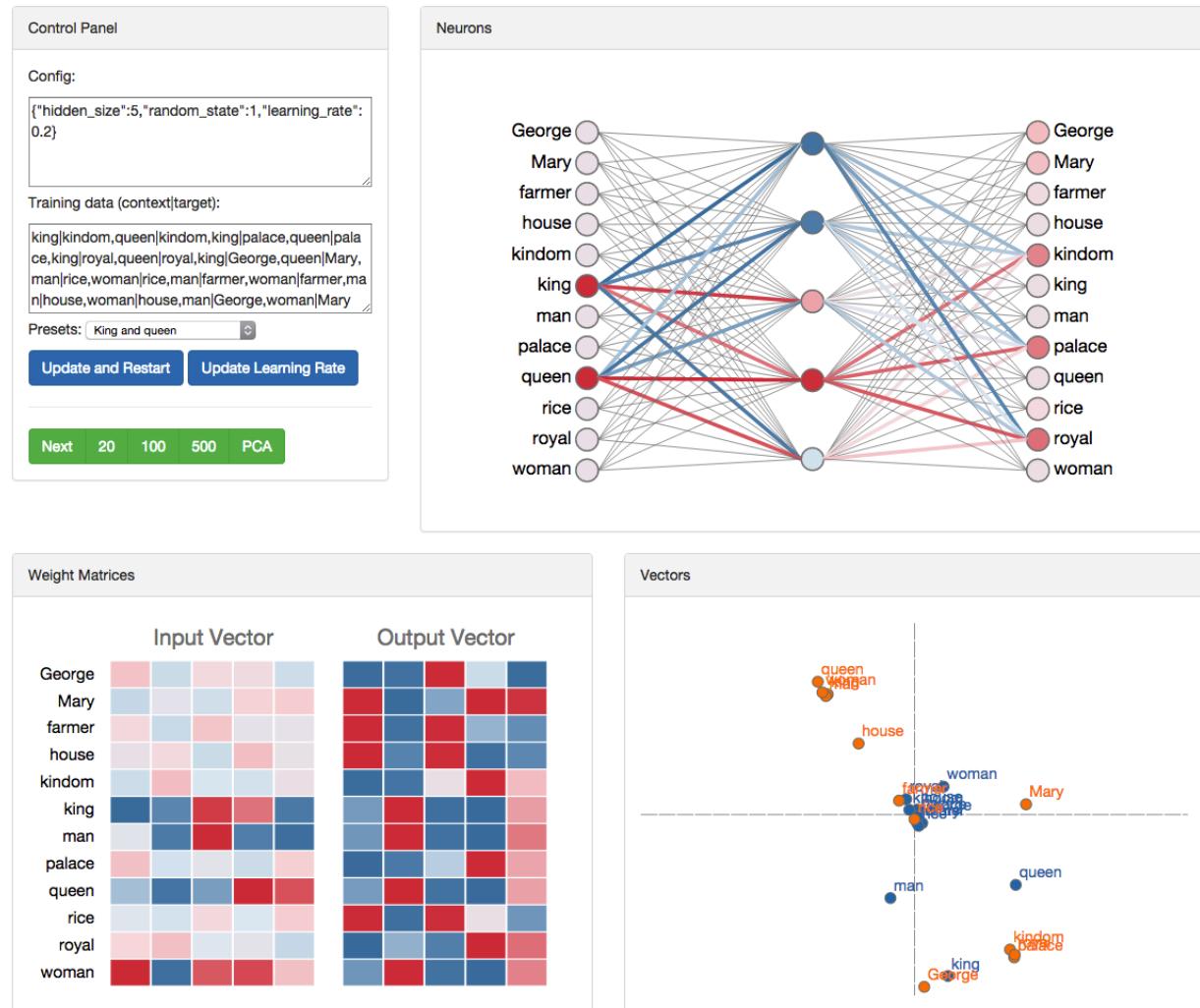


Figure 7: wevi screenshot (<http://bit.ly/wevi-online>)

Dispute Resolution Using Argumentation-Based Mediation

Tomas Trescak¹, Carles Sierra², Simeon Simoff¹, and
Ramon Lopez de Mantaras²

¹ School of Computing, Engineering and Mathematics, University of Western Sydney,
Australia

t.trescak@uws.edu.au, s.simoff@uws.edu.au

² Artificial Intelligence Research Institute, CSIC, Barcelona, Spain
sierra@iiia.csic.es, mantaras@iiia.csic.es

Abstract. Mediation is a process, in which both parties agree to resolve their dispute by negotiating over alternative solutions presented by a mediator. In order to construct such solutions, mediation brings more information and knowledge, and, if possible, resources to the negotiation table. The contribution of this paper is the automated mediation machinery which does that. It presents an argumentation-based mediation approach that extends the logic-based approach to argumentation-based negotiation involving BDI agents. The paper describes the mediation algorithm. For comparison it illustrates the method with a case study used in an earlier work. It demonstrates how the computational mediator can deal with realistic situations in which the negotiating agents would otherwise fail due to lack of knowledge and/or resources.

1 Introduction and Motivation

Dispute resolution is a complex process, depending on the will of involved parties to reach consensus, when they are satisfied with the result of negotiation, which allows them to partially or completely fulfil their goals with the available resources. In many cases, such negotiation depends on searching for alternative solutions, which requires an extensive knowledge about the disputed matter for sound argumentation. Such information may not be available to the negotiating parties and negotiation fails. Mediation, a less radical alternative to arbitration, can assist both parties to come to a mutual agreement. This paper presents an argumentation-based mediation system that builds on previous works in the field of argumentation-based negotiation. It is an extension of the work presented in [1] and focuses on problems where negotiation stalled and had no solution. In [1] agents contain all the knowledge and resources needed to resolve their dispute - a relatively strong assumption in the context of real world negotiations. Agents present arguments, which their opponent can either accept, reject, or they can negotiate on a possible solution. As mentioned earlier, lacking knowledge or resources may lead to an unsuccessful negotiation. In many cases, such knowledge or even alternative resources may be available, but agents are not aware of them.

Our extension proposes a role of a trust-worthy mediator that possesses extensive knowledge about possible solutions of mediation cases, which it can adapt to the current case. *Mediator also has access to various resources that may help to resolve the dispute.* Using this knowledge and resources, as well as knowledge and resources obtained from agents, the mediator creates alternative solutions, which become subject to further negotiation. Furthermore, mediator is guaranteed to be neutral and considered trust-worthy by all interested parties.

In the next section, we summarise related work in the field of automatic mediation and argumentation-based negotiation. In Section 3, we recall the agent architecture proposed by Parsons et al. [1] and extend it with the notion of resources for the purposes of the mediation system. Section 4 presents our mediation algorithm. In Section 5, we revisit the home improvement agents example from [1] and apply our mediation process. Section 6 concludes this work.

2 Previous Work

Computational mediation has recognized the role of the mediator as a problem solver. The MEDIATOR [2] focused on case-based reasoning as a single-step for finding a solution to a dispute resolution problem. The mediation process was reduced to a one-step case-based inference, aimed at selecting an abstract “mediation plan”. The work did not consider the value of the actual dialog with the mediated parties. The PERSUADER [3] deployed mechanisms for problem restructuring that operated over the goals and the relationships between the goals within the game theory paradigm, applied to labor management disputes. To some extent this work is a precursor of another game-theoretic approach to mediation, presented in [4] and the interest-based negotiation approach in [5]. Notable are recent game-theoretic computational mediators AutoMed [6] and AniMed [7] for multi-issue bilateral negotiation under time constraints. They operate within known solution space, offering either specific complete solutions (AutoMed) or incremental partial solutions (AniMed). Similar to the mediator proposed in the ‘curious negotiator’ [8], both mediators monitor negotiations and intervene when there is a conflict between negotiators. The Family_Winner [9] manipulative mediator aimed at modifying the initial preferences of the parties in order to converge to a feasible and mutually acceptable solution. This line of works incorporated “fairness” in the mediation strategies [10].

In real settings information only about negotiation issues is not sufficient to derive the outcome preferences [11]. An exploratory study [12] of a multiple (three) issue negotiation setting suggests the need for developing integrative (rather than position-based) negotiation processes which take into account information about the motivational orientation of negotiating parties. Incorporation of information beyond negotiation issues has been the focus of a series of works related to information-based agency [13, 14, 15]. Value-based argumentation frameworks [16], interest-based negotiation [5] and interest-based reasoning [11] considers the treatment of any kind of motivational information that leads to a preference in negotiation and decision making.

In this paper we propose a new mechanism for automatic mediation using *argumentation-based negotiation* (ABN) as a principal framework for mediation. ABN systems evolved from classical argumentation systems, bringing power to agents to resolve potential dispute deadlocks by persuasion of agents in their beliefs and finding common acceptance grounds by negotiation [17, 1, 18, 19]. ABN is performed by exchanging arguments, which represent a stance of an agent related to the negotiated subject and constructed from beliefs of the agent. Such a stance can support another argument of the agent, explain why a given offer is rejected, or provide conditions upon which the offer would be accepted. Disputing parties can modify their offer or present a counter-offer, based on the information extracted from the argument. Arguments can be used to attack [20] other arguments, supporting or justifying the original offer. With certain level of trust between negotiating agents, arguments serve as knowledge exchange carriers [1] - here we use such mechanisms to exchange information between negotiating parties and the mediator. The decision of whether to trust the negotiating party or not is a part of the *strategy* of an agent. Different strategies are proposed in [21, 22, 23]. Apart from the strategy, essential are the *reasoning mechanisms* and *negotiation protocols*. Relevant to this work are logic frameworks that use argumentation as the key mechanism for reasoning [24, 25, 26, 27]. *Negotiation protocols*, which specify the negotiation procedures include either finite-state machines [1], or functions based on the previously executed actions [28]. The reader is referred to [29] for the recent state-of-the-art in ABN frameworks.

Our ABN framework for mediation allows us to seamlessly design and execute realistic mediation process, which utilises the power of argumentation, using agent logics and a negotiation procedure to search for the common agreement space. We have decided to extend the ABN framework in [1], due to the clarity of its logics. In the next section we recall the necessary aspects of the work in [1]. We describe the agent architecture in the ABN systems and define the components that we reuse in our work. Our agents *reason* using argumentation, based on a domain dependent theory specified in a first-order logic. Within the theory, we encode *agent strategies*, by defining their planning steps. Apart from agent theories, *strategy* is defined also in *bridge rules*, explained further in the text. We do not explore a custom protocol, therefore we adopt the one from [1].

3 Agent Architecture

The ABN system presented in [1] is concerned with BDI agents in a multi-context framework, which allows distinct theoretical components to be defined, interrelated and easily transformed to executable components. The authors use different contexts to represent different components of an agent architecture, and specify the interactions between them by means of the bridge rules between contexts. We recall briefly the components of the agent architecture within the ABN system in [1] and add a new “resources” component for mediation purposes.

Units are structural entities representing the main components of the architecture. There are four units within a multi-context BDI agent, namely: the Communication unit, and units for each of the Beliefs, Desires and Intentions.

Bridge rules connect units, which specify internal agent architecture by determining their relationship. Three well-established sets of relationships for BDI agents have been identified in [30]: *strong realism*, *realism* and *weak realism*. In this work, we consider strongly realist agents.

Logics is represented by declarative languages, each with a set of axioms and a number of rules of inference. Each unit has a single logic associated with it. For each of the mentioned B, D, I, C units, we use classical first-order logic, with special predicates B, D and I related to their units. These predicates allow to omit the temporal logic CTL modalities as proposed in [30].

Theories are sets of formulae written in the logic associated with a unit. For each of the four units, we provide domain dependent information, specified as logical expressions in the language of each unit.

Bridge rules are rules of inference which relate formulae in different units. Following are bridge rules for strongly realist BDI agents:

$$\begin{array}{ll} I : I(\alpha) \Rightarrow D : D([\alpha]) & B : \neg B(\alpha) \Rightarrow D : \neg D([\alpha]) \\ D : \neg D(\alpha) \Rightarrow I : \neg I([\alpha]) & C : done(e) \Rightarrow B : B([done(e)]) \\ D : D(\alpha) \Rightarrow B : B([\alpha]) & I : I([does(e)]) \Rightarrow C : does(e) \end{array}$$

Resources are our extension of the contextual architecture of strongly realist BDI agents. Each agent can possess a set of resources R^v with a specific importance value for its owner. This value may determine the order in which agents are willing to give up their resources during the mediation process. We define a value function $v : \$ \rightarrow \mathbb{R}$, which for each resource ϕ specifies a value $\vartheta \in \langle 0, 1 \rangle$, $v(\phi) = \vartheta$. Set R^v is ordered according to function v .

Units, *logics* and *bridge rules* are static components of the mediation system. All participants have to agree on them before the mediation process starts. *Theories* and *resources* are dynamic components, they change during the mediation process depending on the current state of negotiation.

4 Mediation Algorithm

In a mediation process both parties try to resolve their dispute by negotiating over alternative solutions presented by a mediator. Such solutions are constructed, using available knowledge and resources. Agent knowledge is considered private and is not shared with the other negotiating party. Resources to obtain alternative solutions may have a high value for their owners or be entirely missing. Thus, we propose that the role of the mediator is to obtain enough knowledge and resources to be able to construct a new solution. The mediator presents a possible solution to agents (in the form of an *argument*), which they either approve, or reject (*attack*). For the purposes of this paper we follow Dong's notion of attack [20]. Parties can negotiate over a possible solution to come to a mutual agreement. Below we formally define the foundations of our algorithm.

Definition 1. Δ is a set of formulae in language L . An argument is a pair (Φ, ω) , $\Phi \subseteq \Delta$ and $\omega \in \Delta$ such that: (1) $\Phi \not\models \perp$; (2) $\Phi \vdash \omega$; and (3) Φ is a minimal subset of Δ satisfying 2.

A *mediation game* is executed in one or more rounds, during which both mediator and agents perform various actions in order to resolve the dispute. Algorithm 1 describes our proposal of the mediation game. In the beginning of each round, agents α and β have an opportunity to present new knowledge to the mediator μ . This new knowledge is helping their case, or helping to resolve the dispute. Agents can either present knowledge in the form of formulas from their theory or new resources. Resources can be presented in ascending order of importance, one resource in each round or altogether, depending on the strategy of agents. The mediator obtains knowledge by executing function $\Gamma_i^\mu \leftarrow GetKnowledge(i)$, where $i \in \{\alpha, \beta\}$. The mediator incorporates knowledge Γ_i^μ into theory Γ_μ , obtaining Γ'_μ . Please note, that the belief revision operator \oplus is responsible for automatic elimination of conflicting beliefs from the theory. Belief revision operator uses argumentation techniques to find the minimal set of non-conflicting arguments. Using the knowledge in Γ'_μ , the mediator tries to construct a new *solution* by executing the *CreateSolution*(Γ'_μ) function. If the *solution* does not exist and agents did not present new knowledge in this round, mediation fails. Therefore, it is of utmost importance that agents try to introduce knowledge in each round. In the next step, the possible outcomes are:

- When both agents accept the *solution*, mediation finishes with success.
- When both agents reject the *solution*, mediator adds the incorrect solution $\neg solution$ and the explanation of the rejection $\Gamma_i^{\mu'}$ from both agents to its knowledge Γ'_μ and starts a new mediation round.
- When only one agent rejects the solution, a new negotiation process is initiated, where agents try to come to a mutual agreement (e.g. partial division of a specific item) resulting to *solution'*. If this negotiation is successful, the mediator records *solution'* as a new solution and finishes mediation with a success. If the negotiation fails, the mediator adds the explanation of the failure $\Gamma_i^{\mu''}$ and the failed *solution* to Γ'_μ and starts a new mediation round.

The mediation process continues till a resolution is obtained, or fails, when no new solution can be obtained, and no new knowledge can be presented. In the next section, we revisit the example of home improvement agents from [1] and apply the mediation algorithm.

5 Case Study: Revisiting Home Improvement Agents

In this example, agent α is trying to hang a picture on the wall. Agent α knows that to hang a picture it needs a nail and a hammer. Agent α only has a screw, and a hammer, but it knows that agent β owns a nail. Agent β is trying to hang a mirror on the wall. β knows that it needs a nail and a hammer to hang the mirror, but β currently possesses only a nail, and also knows that α has a hammer. Mediator μ owns a screwdriver and knows that a mirror can be hung using a screw and a screwdriver.

The difference with the example in [1] is that mediator owns the knowledge and resource needed to resolve the dispute μ and not the agents. This reflects

Input : Agents α , β and the mediator μ . Γ_α , Γ_β and Γ_μ denote the knowledge of α , β and μ , while Γ_α^μ and Γ_β^μ denote the knowledge presented to the mediator μ respectively by α and β . \oplus is a *belief revision operator*

Output: Resolution of the dispute, or \perp if solution does not exists.

```

1 repeat
2    $\Gamma_\alpha^\mu \leftarrow \text{GetKnowledge}(\alpha);$            // Theory and resources from  $\alpha$ 
3    $\Gamma_\beta^\mu \leftarrow \text{GetKnowledge}(\beta);$            // Theory and resources from  $\beta$ 
4    $\Gamma'_\mu \leftarrow \Gamma_\mu \oplus (\Gamma_\alpha^\mu \cup \Gamma_\beta^\mu);$ 
5    $\text{solution} \leftarrow \text{CreateSolution}(\Gamma'_\mu);$ 
6   if  $\text{solution} = \perp$  and  $\Gamma_\mu = \Gamma'_\mu$  then
7     | return  $\perp;$                                 // Missing new knowledge and no solution
8   end
9   if  $\text{solution} \neq \perp$  then
10    |  $\langle \text{result}_\alpha, \Gamma_\alpha^{\mu'} \rangle \leftarrow \text{Propose}(\mu, \alpha, \text{solution})$ 
11    |  $\langle \text{result}_\beta, \Gamma_\beta^{\mu'} \rangle \leftarrow \text{Propose}(\mu, \beta, \text{solution})$ 
12    |  $\Gamma'_\mu \leftarrow \Gamma'_\mu \oplus (\Gamma_\alpha^{\mu'} \cup \Gamma_\beta^{\mu'})$ 
13    | if  $\neg \text{result}_\alpha$  and  $\neg \text{result}_\beta$  then
14      |   |  $\Gamma'_\mu \leftarrow \Gamma'_\mu \oplus \neg \text{solution};$ 
15      |   |  $\text{solution} \leftarrow \perp;$ 
16    | else if  $\neg \text{result}_\alpha$  or  $\neg \text{result}_\beta$  then
17      |   |  $\langle \text{solution}', \Gamma_\alpha^{\mu''}, \Gamma_\beta^{\mu''} \rangle \leftarrow \text{Negotiate}(\text{solution}, \alpha, \beta);$ 
18      |   |  $\Gamma'_\mu \leftarrow \Gamma'_\mu \oplus (\Gamma_\alpha^{\mu''} \cup \Gamma_\beta^{\mu''})$ 
19      |   | if  $\neg \text{solution}'$  then
20        |     |  $\Gamma'_\mu \leftarrow \Gamma'_\mu \oplus (\neg \text{solution} \cup \neg \text{solution}')$ 
21        |     |  $\text{solution} \leftarrow \perp;$ 
22      |   | else
23        |     |  $\text{solution} \leftarrow \text{solution}'$ 
24      |   | end
25    | end
26  | end
27  |  $\Gamma_\mu \leftarrow \Gamma'_\mu;$ 
28 until  $\text{solution} \neq \emptyset;$ 
29 return  $\text{solution}$ 

```

Algorithm 1: Mediation algorithm

the reality, when clients seek advice of an expert to resolve their problem. As mentioned in the Section 3, agents α and β are strongly realist BDI agents using related bridge rules and predicate logic. We now define all the dynamic parts of the mediation system, i.e. domain specific agent theory and bridge rules¹.

¹ We adopt following notation: $A.^*$ is the theory introduced by the agent α , $B.^*$ is the theory of the agent β , $M.^*$ is the mediator's theory, $G.^*$ is the general theory and $R.^*$ are bridge rules

5.1 Agent Theories

What follows, is the private theory Γ_α of the agent α , whose intention is to hang a picture:

$$I : I_\alpha(\text{Can}(\alpha, \text{hang_picture})) \quad (\text{A.1})$$

$$B : B_\alpha(\text{Have}(\alpha, \text{picture})) \quad (\text{A.2})$$

$$B : B_\alpha(\text{Have}(\alpha, \text{screw})) \quad (\text{A.3})$$

$$B : B_\alpha(\text{Have}(\alpha, \text{hammer})) \quad (\text{A.4})$$

$$B : B_\alpha(\text{Have}(\beta, \text{nail})) \quad (\text{A.5})$$

$$B : B_\alpha(\text{Have}(X, \text{hammer}) \wedge \text{Have}(X, \text{nail}) \wedge \text{Have}(X, \text{picture}) \rightarrow \text{Can}(X, \text{hangPicture})) \quad (\text{A.6})$$

Please note, that agent α , contrarily to the example in [1], no longer knows that a mirror can be hung with a screw and a screwdriver. What follows, is the private theory Γ_β of agent β , whose intention is to hang a mirror.

$$I : I_\beta(\text{Can}(\beta, \text{hangMirror})) \quad (\text{B.1})$$

$$B : B_\beta(\text{Have}(\beta, \text{mirror})) \quad (\text{B.2})$$

$$B : B_\beta(\text{Have}(\beta, \text{nail})) \quad (\text{B.3})$$

$$B : B_\beta(\text{Have}(X, \text{hammer}) \wedge \text{Have}(X, \text{nail}) \wedge \text{Have}(X, \text{mirror}) \rightarrow \text{Can}(X, \text{hangMirror})) \quad (\text{B.4})$$

Following is the theory Γ_μ of the mediator μ , related to the home improvement agents case (please note, that mediator's knowledge can consist of many other beliefs, for example learned from other mediation cases):

$$B : B_\mu(\text{Have}(\mu, \text{screwdriver})) \quad (\text{M.1})$$

$$B : B_\mu(\text{Have}(X, \text{screw}) \wedge \text{Have}(X, \text{screwdriver}) \wedge \text{Have}(X, \text{mirror}) \rightarrow \text{Can}(X, \text{hang_mirror})). \quad (\text{M.2})$$

$$B : B_\mu(\text{Have}(X, \text{hammer}) \wedge \text{Have}(X, \text{nail}) \wedge \text{Have}(X, \text{mirror}) \rightarrow \text{Can}(X, \text{hangMirror})) \quad (\text{M.3})$$

We adopt the following theories from [1] with actions that integrate different models reflecting real world processes such as change of ownership, and processes that model decisions and planning of actions. In what follows $i \in \{\alpha, \beta\}$.

Ownership. When an agent (X) gives up artifact (Z) to (Y), (Y) becomes its new owner:

$$B : B_i(\text{Have}(X, Z) \wedge \text{Give}(X, Y, Z) \rightarrow \text{Have}(Y, Z)) \quad (\text{G.1})$$

Reduction. If there is a way to achieve an intention, an agent adopts the intention to achieve its preconditions:

$$B : B_i(I_j(Q)) \wedge B_i(P_1 \wedge \dots \wedge P_k \wedge \dots \wedge P_n \rightarrow Q) \quad (\text{G.2})$$

$$\wedge \neg B_i(R_1 \wedge \dots \wedge R_m \rightarrow Q) \rightarrow B_i(I_j(P_l))$$

Generosity Mediator μ is willing to give up any resource Q

$$B : B_\mu(\text{Have}(\mu, Q)) \rightarrow \neg I_\mu(\text{Have}(\mu, Q)). \quad (\text{G.3})$$

Unicity. When an agent (X) gives an artifact (Z) away, (X) longer owns it:

$$B : B_i(\text{Have}(X, Z) \wedge \text{Give}(X, Y, Z) \rightarrow \neg \text{Have}(X, Z)) \quad (\text{G.4})$$

Benevolence. When agent i does not need (Z) and is asked for it by X , i will give Z up:

$$B : B_i(\text{Have}(i, Z) \wedge \neg I_i(\text{Have}, i, Z) \wedge \text{Ask}(X, i, \text{Give}(i, X, Z)) \rightarrow I_i(\text{Give}(i, X, Z))) \quad (\text{G.5})$$

Parsimony. If an agent believes that it does not intend to do something, it does not believe that it will intend to achieve the preconditions (i.e. the means) to achieve it:

$$B : B_i(\neg I_i(Q)) \wedge B_i(P_1 \wedge \dots \wedge P_j \wedge \dots \wedge P_n \rightarrow Q) \rightarrow \neg B_i(I_i(P_j)) \quad (\text{G.6})$$

Unique choice. If there are two ways of achieving an intention, only one is intended. Note that we use ∇ to denote exclusive or.

$$\begin{aligned} B : & B_i(I_i(Q)) \wedge B_i(P_1 \wedge \dots \wedge P_j \wedge \dots \wedge P_n \rightarrow Q) \\ & \wedge B_i(R_1 \wedge \dots \wedge R_n \rightarrow Q) \rightarrow \\ & B_i(I_i(P_1 \wedge \dots \wedge P_n)) \nabla B_i(I_i(R_1 \wedge \dots \wedge R_n)) \end{aligned} \quad (\text{G.7})$$

A theory that contains free variables (e.g. X) is considered the *general theory*, while a theory with bound variables (e.g. α or β) is considered the *case theory*. The mediator stores only the *general theory* for its reuse with future cases. In addition, an agent's theory contains rules of inference, such as modus ponens, modus tollens and particularization.

5.2 Bridge Rules

What follows, is a set of domain dependent bridge rules that link inter-agent communication and the agent's internal states.

Advice. When the mediator μ believes that it knows about possible intention I_X of X it *tells* it to X . Also, when mediator μ knows something (ϕ) that can help to achieve intention φ of agent X , mediator *tells* it to X .

$$B_\mu(I_X(\varphi)) \Rightarrow \text{Tell}(\mu, X, B_\mu(I_X(\varphi))) \quad (\text{R.1})$$

$$B_\mu(I_X(\varphi)) \wedge B_\mu(\phi \rightarrow I_X(\varphi)) \Rightarrow \text{Tell}(\mu, X, B_\mu(\phi \rightarrow I_X(\varphi))) \quad (\text{R.2})$$

Trust in mediator When an agent (i) is told of a belief of mediator (μ), it accepts that belief:

$$C : \text{Tell}(\mu, i, B_\mu(\varphi)) \Rightarrow B : B_i(\varphi). \quad (\text{R.3})$$

Request. When agent (i) needs (Z) from agent (X), it asks for it:

$$I : I_i(\text{Give}(X, i, Z)) \Rightarrow C : \text{Ask}(i, X, \text{Give}(X, i, Z)). \quad (\text{R.4})$$

Accept Request. When agent (i) asks something (Z) from agent (X), and it is not in intention of (X) to have (Z), it is given to i :

$$C : \text{Ask}(i, X, \text{Give}(X, i, Z)) \wedge \neg I_X(\text{Have}(X, Z)) \Rightarrow I_i(\text{Give}(X, i, Z)). \quad (\text{R.5})$$

5.3 Resources

In Section 3, we have introduced the notion of importance of resources, which defines the order in which agents are giving up their resources during the mediation process. The picture and the hammer depend on the successful accomplishment of agent's α goal and have an importance value of 1. Agent β owns a mirror and a nail, both with importance 1. All other resources have importance 0.

5.4 Argumentation System

Our automatic mediation system uses the ABN system, proposed in [1], which is based on the one proposed in [24]. The system constructs a series of logical steps (arguments) for and against propositions of interest and as such may be seen as an extension of classical logic. In classical logic, an argument is a sequence of inferences leading to a true conclusion. It is summarized by the schema $\Gamma \vdash (\varphi, G)$, where Γ is the set of formulae available for building arguments, \vdash is a suitable consequence relation, φ is the proposition for which the argument is made, and G indicates the set of formulae used to infer φ , with $G \subseteq \Gamma$.

5.5 Mediation

In this section, we follow Algorithm 1 and explain how we can resolve the home improvement agent dispute using automatic mediation. In comparison to Parsons et al. [1], our agents do not possess all the knowledge and resources to resolve their dispute; thus the classical argumentation fails. The mediation algorithm runs in rounds and finishes with:

1. Success, when both agents accept the solution proposed by the mediator.
2. Failure, when the mediator can not create a new solution and no new knowledge or resources are presented in two consecutive rounds.

The algorithm starts with the mediator gathering information about the dispute from both agents (function *GetKnowledge*). In the first round, agents α and β state their goals, which become part of the mediator's beliefs B_μ :

$$B : B_\mu(I_\alpha(\text{Can}(\alpha, \text{hang-picture}))) \quad (M.4)$$

$$B : B_\mu(I_\beta(\text{Can}(\beta, \text{hangMirror}))) \quad (M.5)$$

With this new theory, the mediator tries to construct a new solution, and it fails. Therefore, in the next round, agents have to present more knowledge or resources. Failing to do so would lead to failure of the mediation process. To speed things up, we assume that agents presented all the necessary knowledge and resources in this single step, although this process can last several rounds depending on the strategy of an agent. For example, if a “cautious” agent owns more than one resource, it chooses to give up the resource with the lowest importance.

$$B: B_\mu(\text{Have}(\alpha, \text{picture})) \quad (M.6) \quad B: B_\mu(\text{Have}(\beta, \text{nail})) \quad (M.9)$$

$$B: B_\mu(\text{Have}(\alpha, \text{screw})) \quad (M.7) \quad B: B_\mu(\text{Have}(\beta, \text{mirror})) \quad (M.10)$$

$$B: B_\mu(\text{Have}(\alpha, \text{hammer})) \quad (M.8)$$

With this new information, the mediator is finally able to construct the **solution** to the dispute consisting of three different arguments. With the following two arguments, mediator proposes agent β to hang the mirror using the screw and the screwdriver (M.2), and screw can be obtained from the agent α and the screwdriver obtained from the mediator itself (Please note, that this

knowledge is part of the support for the presented arguments). The first argument is: $(I_\beta(Give(\alpha, \beta, screw)), P'_\beta)$, where P'_β is²:

$$\{(M.2), (M.5), (G.2)\} \vdash_{pt, mp} B_\mu(I_\beta(Have(\beta, screw))) \quad (M.11)$$

$$\{(M.7), (G.1)\} \vdash_{mp} B_\mu(Give(\alpha, Y, screw) \rightarrow Have(Y, screw)) \quad (M.12)$$

$$\{(M.11), (M.12), (G.2)\} \vdash_{pt, mp} B_\mu(I_\beta(Have(\beta, screw))) \quad (M.13)$$

$$\{(M.13)\} \vdash_{R.1} Tell(\mu, \beta, I_\beta(Have(\beta, screw))) \quad (M.14)$$

$$\{(M.14)\} \vdash_{R.3} I_\beta(Have(\beta, screw)) \quad (M.15)$$

The second argument is: $(I_\beta(Give(\mu, \beta, screwdriver)), P''_\beta)$, where P''_β is

$$\{(M.2), (M.5), (G.2)\} \vdash_{pt, mp} B_\mu(I_\beta(Have(\beta, screwdriver))) \quad (M.16)$$

$$\{(M.1), (G.1)\} \vdash_{mp} B_\mu(Give(\mu, Y, screwdriver) \rightarrow Have(Y, screwdriver)) \quad (M.17)$$

$$\{(M.16), (M.17), (G.2)\} \vdash_{pt, mp} B_\mu(I_\beta(Have(\beta, screwdriver))) \quad (M.18)$$

$$\{(M.18)\} \vdash_{R.1} Tell(\mu, \beta, I_\beta(Have(\beta, screwdriver))) \quad (M.19)$$

$$\{(M.19)\} \vdash_{R.3} I_\beta(Have(\beta, screwdriver)) \quad (M.20)$$

These two arguments represent advices to β on how it can achieve its goal (B.1) that was communicated to mediator μ as (M.5). Using bridge rule (R.4) β converts this into the following actions:

$$\{M.15\} \vdash_{R.4} Ask(\beta, \alpha, Give(\alpha, \beta, screw)).$$

$$\{M.20\} \vdash_{R.4} Ask(\beta, \mu, Give(\mu, \beta, screwdriver)).$$

When both α and μ receive this request, they convert this into *accept request* action using bridge rule (R.5). Mediator accepts this request due to the *generosity* theory (G.3), which defines that it is not an intention of mediator to own anything. Agent β cannot find a counter-argument that would reject this request (it does not need the nail) and accepts it. With the screw, the screwdriver, the mirror and knowledge on how to hang the mirror using these tools, β can fulfil its goal, and it no longer needs the nail. Therefore, the following argument that solves the goal of α is also accepted: $(I_\alpha(Give(\beta, \alpha, nail)), P_\alpha)$, where P_α is:

$$\{(M.3), (M.4), (G.2)\} \vdash_{mp} B_\mu(I_\alpha(Have(\alpha, nail))) \quad (M.21)$$

$$\{(M.9), (G.1)\} \vdash_{mp} B_\mu(Give(\beta, Y, nail) \rightarrow Have(Y, nail)) \quad (M.22)$$

$$\{(M.21), (M.22), (G.2)\} \vdash_{pt, mp} B_\mu(I_\alpha(Have(\alpha, nail))) \quad (M.23)$$

$$\{(M.23)\} \vdash_{R.1} Tell(\mu, \alpha, B_\mu(I_\alpha(Have(\alpha, nail)))) \quad (M.18)$$

$$\{(M.18)\} \vdash_{R.3} I_\alpha(Have(\alpha, nail)) \quad (M.19)$$

we convert this into action, using the bridge rule R.1 into:

$$\{M.19\} \vdash_{R.1} Ask(\alpha, \beta, Give(\beta, \alpha, nail)).$$

When agent β receives this request, β can accept it by the bridge rule (R.5). This is only possible because of the previous two arguments, when an alternative plan to hang the mirror was presented to β , otherwise β would not be willing to give up the nail needed for his plan. Agent β can now decide between two plans using (G.7); therefore it decides to *give* α the nail and both agents were able to fulfil their goals (we assume that β does not want to sabotage the mediation).

² mp stands for *modus ponens* and pt stands for *particularization*

6 Conclusion

Mediation brings more information and knowledge to the negotiation table, hence, an automated mediator would need the machinery that could do that. Addressing this issue in an automated setting, we have presented an ABN approach that extends the logic-based approach to ABN involving BDI agents, presented in [1]. We have introduced a mediator in the multiagent architecture, which has extensive knowledge concerning mediation cases and access to resources. Using both, knowledge and resources, the mediator proposes solutions that become the subject of further negotiation when the agents in conflict cannot solve the dispute by themselves. We have described our mediation algorithm and illustrated it with the same case study introduced in [1]. The presence of a mediator in ABN allows to deal with realistic situations when negotiation is stalled. In this work we assumed that the agents and the mediator operate within the same ontology, describing the negotiation domain. In real settings, the negotiators may interpret the same term differently. In order to avoid this, mediation will require the initial alignment of the ontologies with which all parties operate.

References

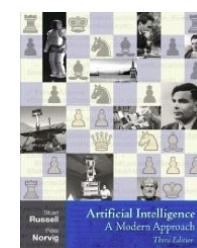
- [1] Parsons, S., Sierra, C., Jennings, N.: Agents that reason and negotiate by arguing. *Journal of Logic and computation* **8**(3) (1998) 261–292
- [2] Koldner, J.L., Simpson, R.L.: The mediator: Analysis of an early case-based problem solver. *Cognitive Science* **13**(4) (1989) 507–549
- [3] Sycara, K.P.: Problem restructuring in negotiation. *Management Science* **37**(10) (1991) 1248–1268
- [4] Wilkenfeld, J., Kraus, S., Santmire, T.E., Fraim, C.K.: The role of mediation in conflict management: Conditions for successful resolution. In: *Multiple Paths to Knowledge in International Relations*. Lexington Books (2004)
- [5] Rahwan, I., Pasquier, P., Sonenberg, L., Dignum, F.: A formal analysis of interest-based negotiation. *Annals of Mathematics and Artificial Intelligence* **55**(3-4) (2009) 253–276
- [6] Chalamish, M., Kraus, S.: AutoMed: an automated mediator for multi-issue bilateral negotiations. *Autonomous Agents and Multi-Agent Systems* **24** (2012) 536–564
- [7] Lin, R., Gev, Y., Kraus, S.: Bridging the gap: Face-to-face negotiations with automated mediator. *IEEE Intelligent Systems* **26**(6) (2011) 40–47
- [8] Simoff, S.J., Debenham, J.: Curious negotiator. In Klusch, M., Ossowski, S., Shehory, O., eds.: *Proceedings of the Int. Conference on Cooperative Information Agents, CIA-2002*, Springer, Heidelberg (2002)
- [9] Bellucci, E., Zeleznykow, J.: Developing negotiation decision support systems that support mediators: case study of the Family_Winner system. *Artificial Intelligence and Law* **13**(2) (2005) 233–271
- [10] Abrahams, B., Bellucci, E., Zeleznykow, J.: Incorporating fairness into development of an integrated multi-agent online dispute resolution environment. *Group Decision and Negotiation* **21** (2012) 3–28
- [11] Visser, W., Hindriks, K.V., Jonker, C.M.: Interest-based preference reasoning. In: *Proceedings of International Conference on Agents and Artificial Intelligence ICAART2011*. (2011) 79–88

- [12] Schei, V., Rognes, J.K.: Knowing me, knowing you: Own orientation and information about the opponent's orientation in negotiation. *International Journal of Conflict Management* **14**(1) (2003) 43–60
- [13] Debenham, J.: Bargaining with information. In Jennings, N.R., Sierra, C., Sonnenberg, L., Tambe, M., eds.: *Proceedings Third International Conference on Autonomous Agents and Multi Agent Systems AAMAS-2004*, ACM Press, New York (2004) 664–671
- [14] Debenham, J.K., Simoff, S.: Negotiating intelligently. In Bramer, M., Coenen, F., Tuson, A., eds.: *Proceedings 26th International Conference on Innovative Techniques and Applications of Artificial Intelligence*, Cambridge, UK (2006) 159–172
- [15] Sierra, C., Debenham, J.: Information-based agency. In: *Proceedings of Twentieth International Joint Conference on Artificial Intelligence IJCAI-07*, Hyderabad, India (2007) 1513–1518
- [16] Bench-Capon, T.J.M.: Persuasion in practical argument using value-based argumentation frameworks. *Journal of Logic and Computation* **13**(3) (2003) 429–448
- [17] Sycara, K.P.: Persuasive argumentation in negotiation. *Theory and decision* **28**(3) (1990) 203–242
- [18] Kakas, A., Moraitis, P.: Adaptive agent negotiation via argumentation. In: *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, ACM (2006) 384–391
- [19] Rahwan, I., Ramchurn, S.D., Jennings, N.R., Mcburney, P., Parsons, S., Sonnenberg, L.: Argumentation-based negotiation. *The Knowledge Engineering Review* **18**(04) (2003) 343–375
- [20] Dung, P.M.: On the acceptability of arguments and its fundamental role in non-monotonic reasoning, logic programming and n-person games. *Artificial Intelligence* **77** (1995) 321–358
- [21] Hadidi, N., Dimopoulos, Y., Moraitis, P.: Argumentative alternating offers. In: *Argumentation in Multi-Agent Systems*. Springer (2011) 105–122
- [22] Phan Minh Dung. Phan Minh Thang, F.T.: Towards argumentation-based contract negotiation. *Computational Models of Argument: Proceedings of Comma 2008* **172** (2008) 134
- [23] Dijkstra, P., Prakken, H., de Vey Mestdagh, K.: An implementation of norm-based agent negotiation. In: *Proceedings of the 11th international conference on Artificial intelligence and law*, ACM (2007) 167–175
- [24] Krause, P., Ambler, S., Elvang-Goransson, M., Fox, J.: A logic of argumentation for reasoning under uncertainty. *Computational Intelligence* **11**(1) (1995) 113–131
- [25] Prakken, H., Sartor, G.: Argument-based extended logic programming with de-feasible priorities. *Journal of applied non-classical logics* **7**(1-2) (1997) 25–75
- [26] Dung, P.M., Kowalski, R.A., Toni, F.: Dialectic proof procedures for assumption-based, admissible argumentation. *Artificial Intelligence* **170**(2) (2006) 114–159
- [27] Oren, N., Norman, T.J., Preece, A.: Subjective logic and arguing with evidence. *Artificial Intelligence* **171**(10) (2007) 838–854
- [28] Amgoud, L., Dimopoulos, Y., Moraitis, P.: A unified and general framework for argumentation-based negotiation. In: *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, ACM (2007) 963–970
- [29] Rahwan, I., Simari, G., eds.: *Argumentation in Artificial Intelligence*. Springer-Verlag (2009)
- [30] Rao, A.S., Georgeff, M.P.: Formal models and decision procedures for multi-agent systems. *Technical Report Technical Note 61* (1995)

Decision Making

- Rational preferences
- Utilities
- Money
- Multiattribute utilities
- Decision networks
- Value of information

Material from
Russell & Norvig,
chapter 16



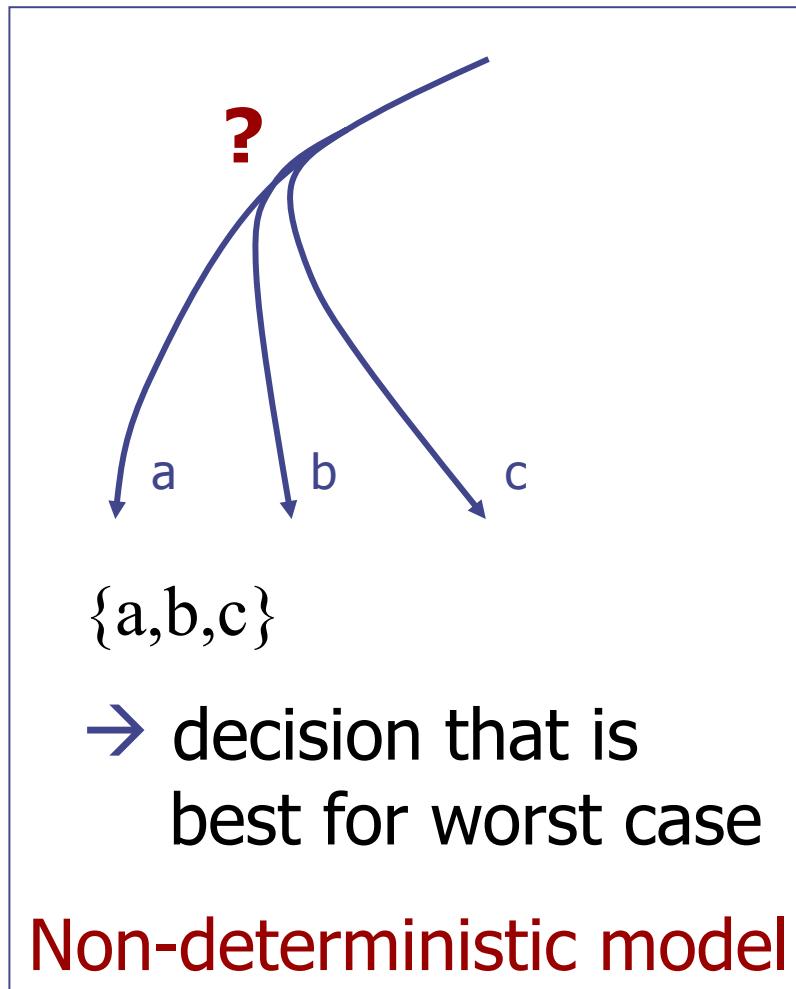
Many slides taken from
Russell & Norvig's slides
**Artificial Intelligence:
A Modern Approach**

Some based on Slides by
Lise Getoor, Jean-Claude
Latombe and Daphne Koller

Decision Making under Uncertainty

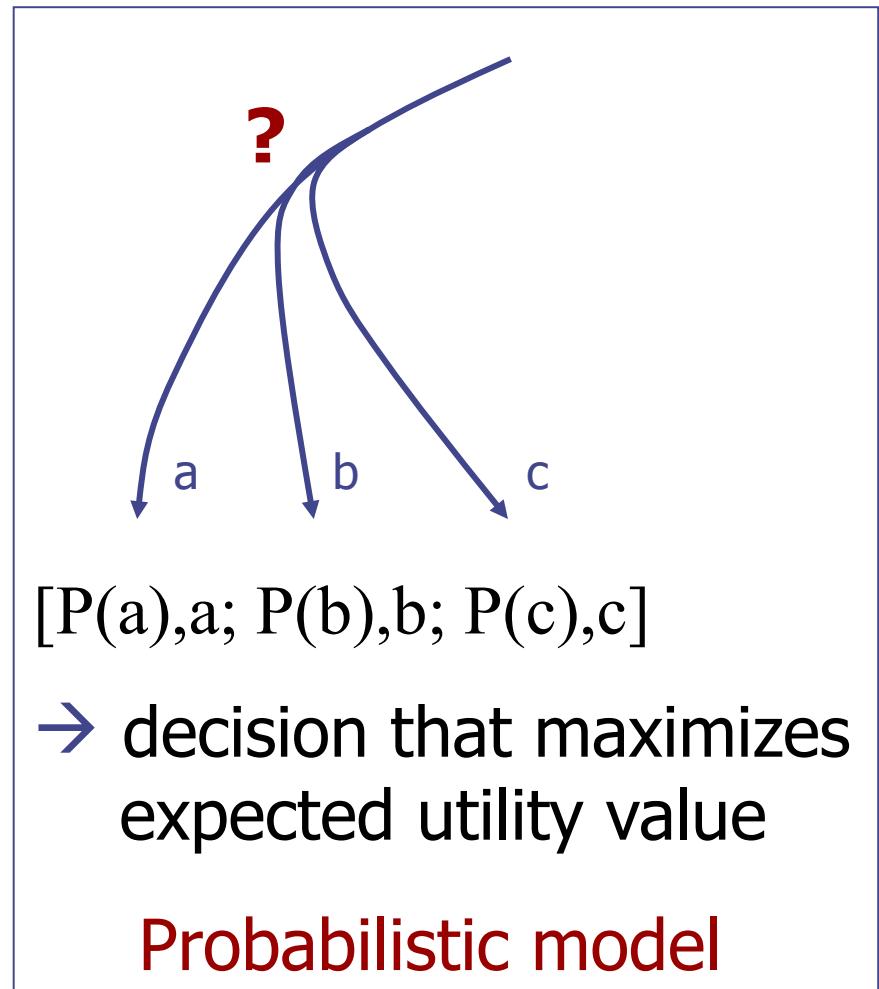
- Many environments are uncertain in the sense that it is not clear what state an action will lead to
 - **Uncertainty:** Some states may be likely, others may be unlikely
 - **Utility:** Some states may be desirable, others may be undesirable
- Still, an agent has to make a decision which action to choose
→ **Decision Theory** is concerned with making rational decisions in such scenarios

Non-Deterministic vs. Probabilistic Uncertainty



Non-deterministic model

~ Adversarial search

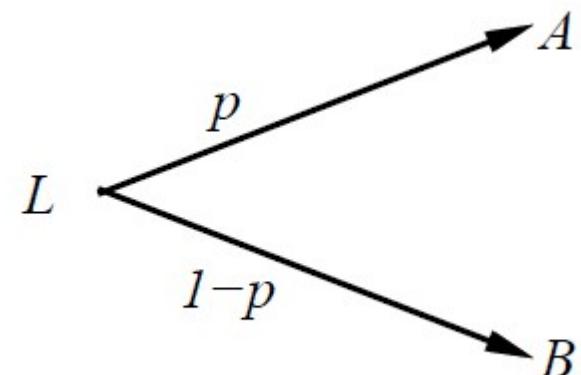


Probabilistic model

Lotteries and Preferences

- In the following, we call such probabilistic events **lotteries**
 - A lottery consists of a set of events (**prizes**) with their **probabilities**

Lottery $L = [p, A; (1 - p), B]$



- Preferences:**
 - An agent likes certain prizes better than others
 - An agent therefore also likes certain lotteries better than others

Notation:

$A \succ B$ A preferred to B

$A \sim B$ indifference between A and B

$A \gtrsim B$ B not preferred to A

Preferences and Rational Behavior

- Preferences between prizes may, in principle, be arbitrary
- For example, preferences may be cyclic

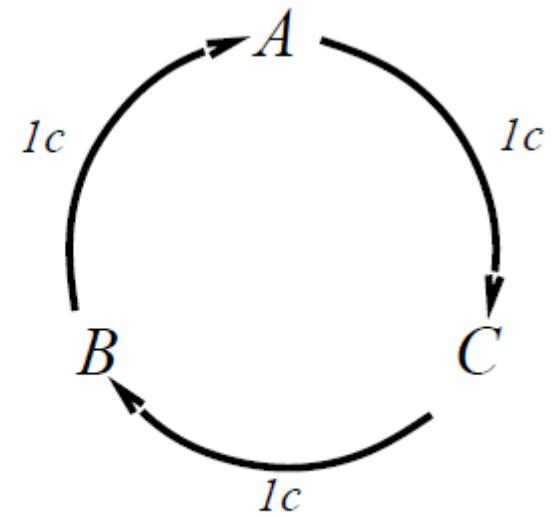
$$A \succ B, B \succ C, C \succ A$$

- However, cyclic preferences lead to irrational behavior:

If $B \succ C$, then an agent who has C would pay (say) 1 cent to get B

If $A \succ B$, then an agent who has B would pay (say) 1 cent to get A

If $C \succ A$, then an agent who has A would pay (say) 1 cent to get C



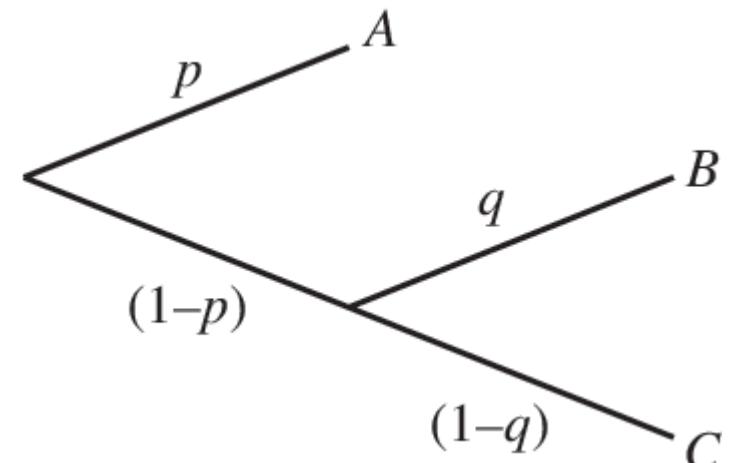
→ Eventually the agent will give away all its money

Preferences and Rational Behavior

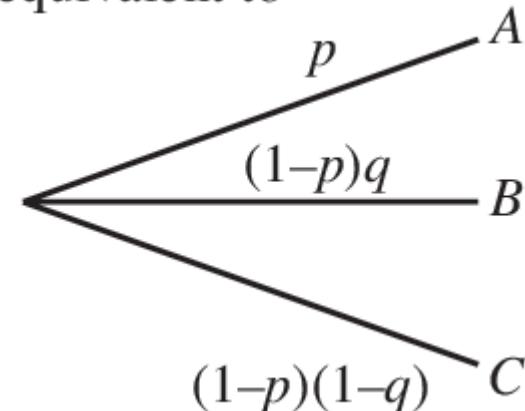
- Another property that should be obeyed is that lotteries are decomposable
- Therefore, no rational agent should have a preference between the two equivalent formulations

$$[p, A; 1-p, [q, B; 1-q, C]] \sim [p, A; (1-p)q, B; (1-p)(1-q), C]$$

- Decomposability



is equivalent to



- Such properties be formulated as **constraints** on preferences

Other Constraints for Rational Behavior

- Together with Decomposability, these constraints define a rational behavior:

Orderability

$$(A \succ B) \vee (B \succ A) \vee (A \sim B)$$

Transitivity

$$\cdot \quad (A \succ B) \wedge (B \succ C) \Rightarrow (A \succ C)$$

Continuity

$$A \succ B \succ C \Rightarrow \exists p \ [p, A; 1 - p, C] \sim B$$

Substitutability

$$A \sim B \Rightarrow [p, A; 1 - p, C] \sim [p, B; 1 - p, C]$$

Monotonicity

$$A \succ B \Rightarrow (p \geq q \Leftrightarrow [p, A; 1 - p, B] \succsim [q, A; 1 - q, B])$$

Utility functions

- A natural way for measuring how desirable certain prizes are is using a **utility function** U
 - A utility function assigns a numerical value to each prize
- Utility function naturally lead to preferences

$$A > B \Leftrightarrow U(A) > U(B)$$

- The Expected Utility of an Event is the expected value of the utility function in a lottery

$$EU(X) = \sum_{x \in X} P(x) \cdot U(x)$$

- A utility function in a deterministic environment (no lotteries) is also called a **value function**

Maximizing Utilities

- It has been shown that acting according to rational preferences corresponds to maximizing a utility function U

Theorem (Ramsey, 1931; von Neumann and Morgenstern, 1944):

Given preferences satisfying the constraints

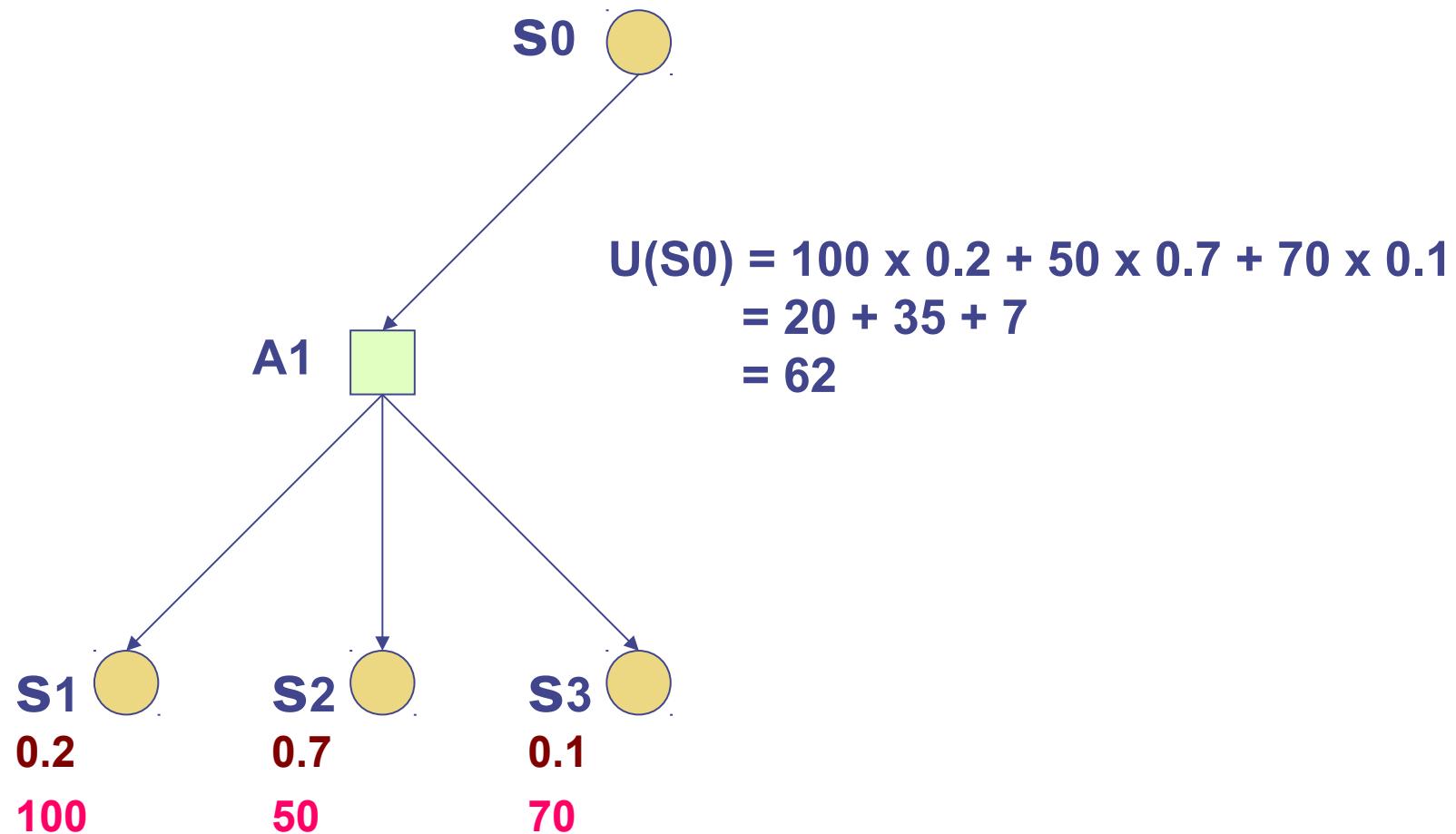
there exists a real-valued function U such that

$$U(A) \geq U(B) \Leftrightarrow A \succsim B$$

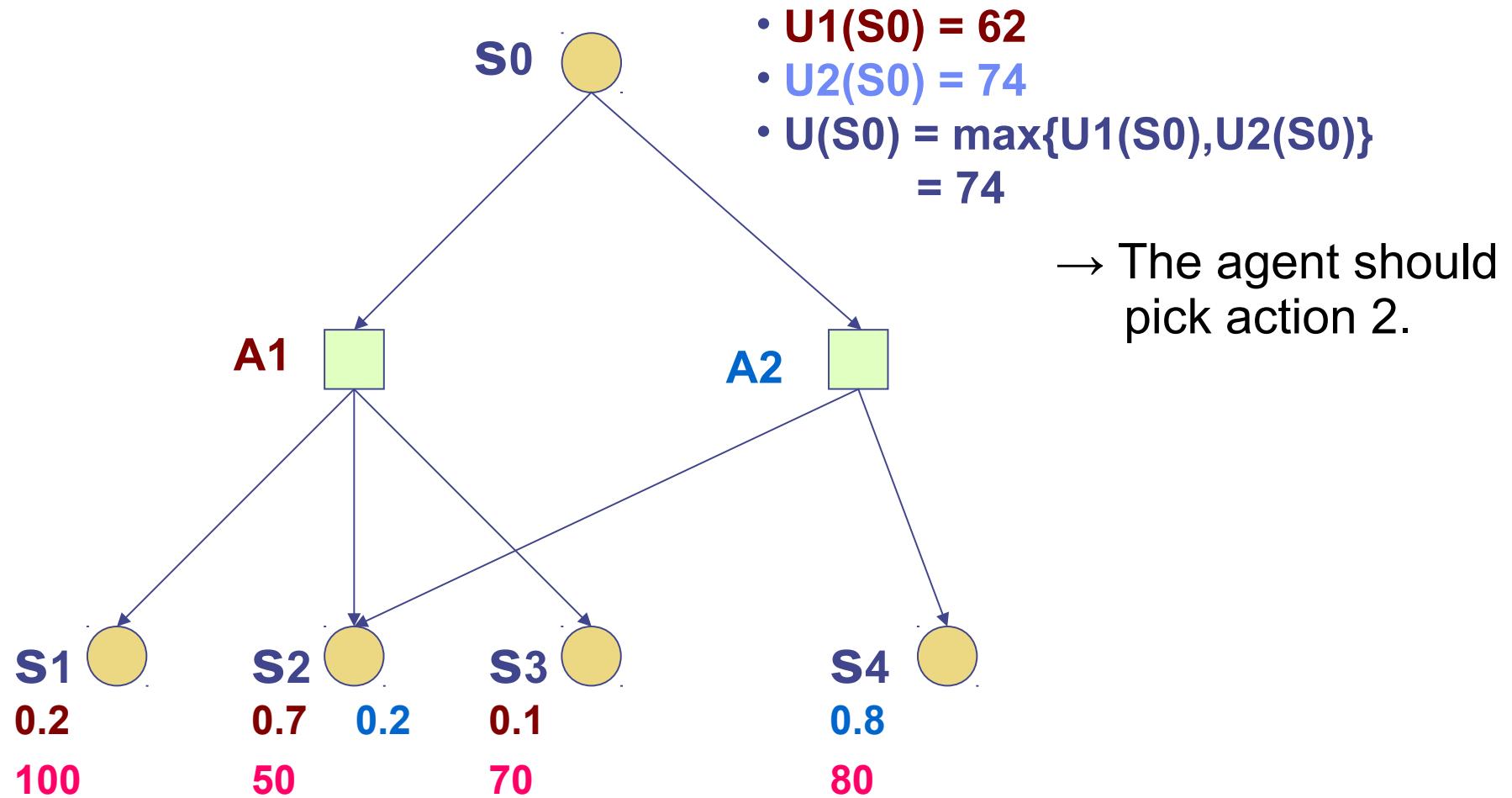
$$U([p_1, S_1; \dots; p_n, S_n]) = \sum_i p_i U(S_i)$$

- Maximizing Expected Utility (MEU) principle**
 - An agent acts rationally if it selects the action that promises the highest expected utility
- Note that an agent may act rationally without knowing U or the probabilities!
 - e.g., according to pre-compiled look-up tables for optimal actions

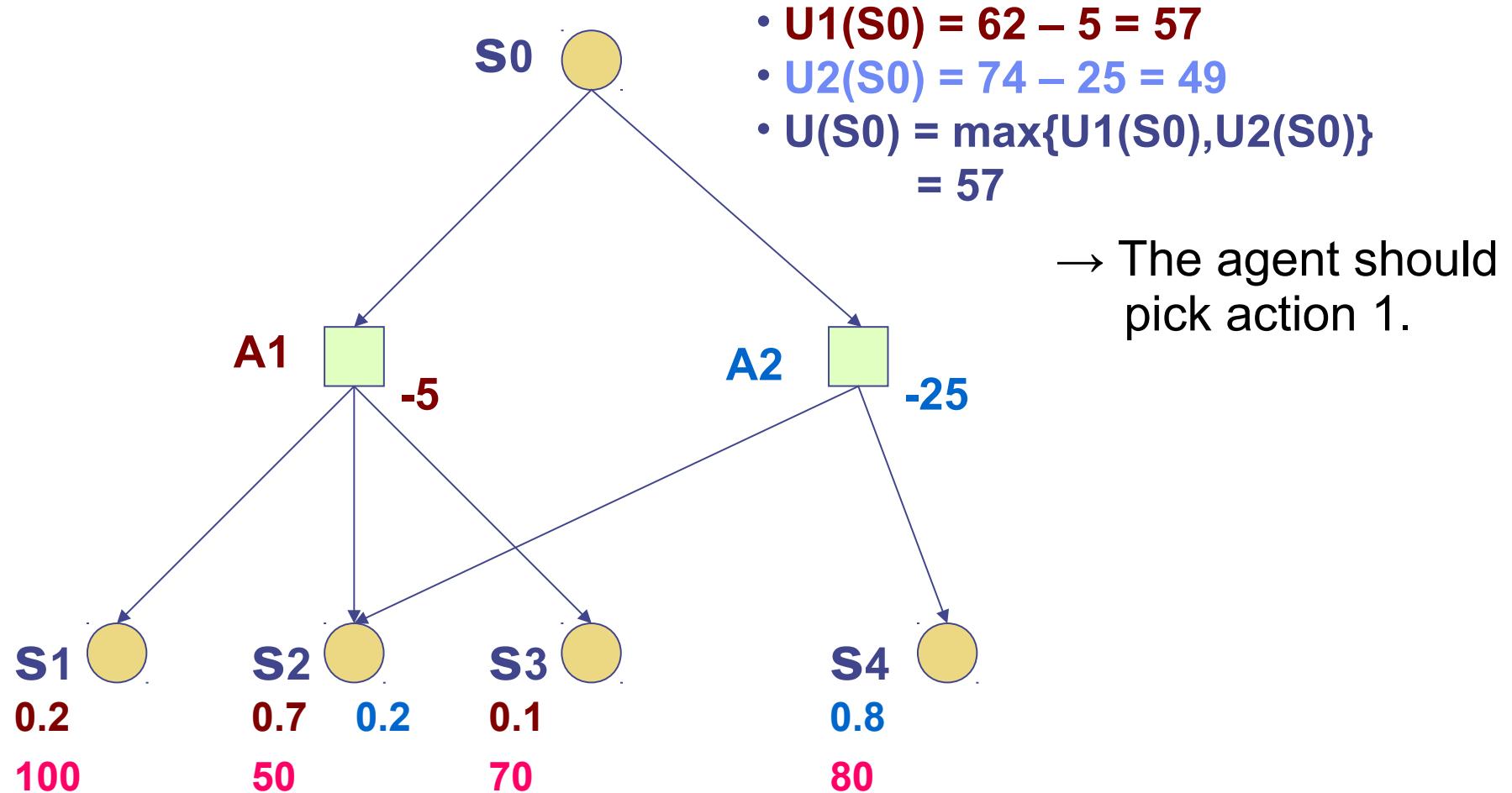
Example: Expected Utility of an Action



Example: Choice between 2 Actions



Example: Adding Action Costs



MEU Principle

- A **rational agent** should choose the action that maximizes agent's expected utility
- This is the basis of the field of **decision theory**
- The MEU principle provides a **normative criterion** for rational choice of action

Do we now have a working definition of rational behavior?
And therefore solved AI?

Not quite...

- Must have **complete** model of:
 - Actions
 - Utilities
 - States
- Even if you have a complete model, will be computationally **intractable**
- In fact, a truly rational agent takes into account the utility of reasoning as well – **bounded rationality**
- Nevertheless, great progress has been made in this area recently, and we are able to solve much more complex decision-theoretic problems than ever before

Decision Theory vs. Reinforcement Learning

- Simple decision-making techniques are good for selecting the best action in simple scenarios
- → Reinforcement Learning is concerned with selecting the optimal action in Sequential Decision Problems
 - Problems where a sequence of actions has to be taken until a goal is reached.

How to measure Utility?

An obvious idea: Money

- However, Money is not the same as utility

Example:

- If you just earned 1,000,000\$, are you willing to bet them on a double-or-nothing coin flip?
- How about triple or nothing?

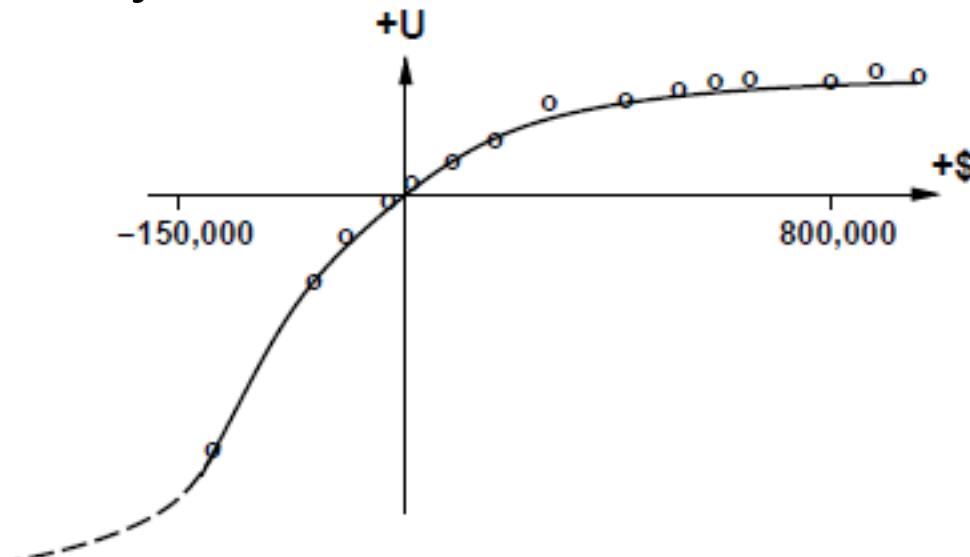
$$U(1,000,000) > EU([0.5, 0; 0.5, 3,000,000])?$$

$$U(1,000,000) > 0.5 \cdot U(0) + 0.5 \cdot U(3,000,000)?$$

Most people would grab a million and run, although the expected value of the lottery is 1.5 million

The Utility of Money

- Grayson (1960) found that the utility of money is almost exactly proportional to its logarithm
- One way to measure it:
 - Which is the amount of money for which your behavior between „grab the money“ changes to „play the lottery“?
 - Obviously, this also depends on the person i
 - if you already have 50 million, you are more likely to gamble...
- Utility of money for a certain Mr. Beard:



Risk-Averse vs. Risk-Seeking

- People like Mr. Beard are **risk-averse**
 - Prefer to have the expected monetary value of the lottery ($EMV(L)$) handed over than to play the lottery L

$$U(L) < U(S_{EMV(L)})$$

- Other people are **risk-seeking**
 - Prefer the thrill of a wager over secure money

$$U(L) > U(S_{EMV(L)})$$

- For **risk-neutral** people, the Utility function is the identity

$$U(L) = U(S_{EMV(L)})$$

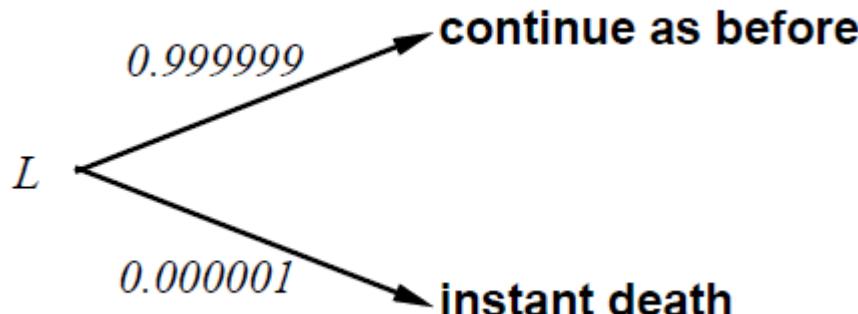
- The difference $U(L) - U(S_{EMV(L)})$ is called the **insurance premium**. This is the business model of insurances

General Approach to Assessing Utilities

- Find probability p so that the expected value of a lottery between two extremes corresponds the value of the prize A
 - compare a given state A to a standard lottery L_p that has “best possible prize” u_{\top} with probability p
 - “worst possible catastrophe” u_{\perp} with probability $(1 - p)$
 - adjust lottery probability p until $A \sim L_p$
- Normalized utility scales interpolate $u_{\top} = 1.0$, $u_{\perp} = 0.0$
 - Normalization does not change the behavior of an agent, because (positive) linear transformations $U'(S) = k_1 + k_2 \cdot U(S)$ leave the ordering of actions unchanged
 - If there are no lotteries, any monotonic transformation leaves the preference ordering of actions unchanged

Other Units of Measurements for Utilities

- In particular for medicine and safety-critical environments, other proposals have been made (and used)
- Micromorts:
 - A **micromort** is the lottery of dying with a probability of one in a million
 - It has been established that a micromort is worth about \$50.
 - Does not mean that you kill yourself for \$50,000,000 (we have already seen that utility functions are not linear)
 - Used in safety-critical scenarios, car insurance, ...
 - Quality-Adjusted Life Year (**QALY**)
 - A year in good health, used in medical applications



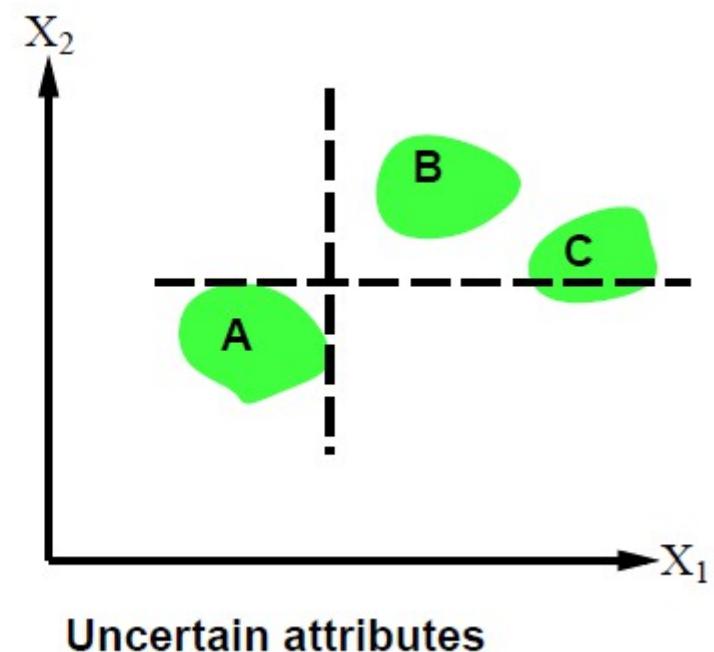
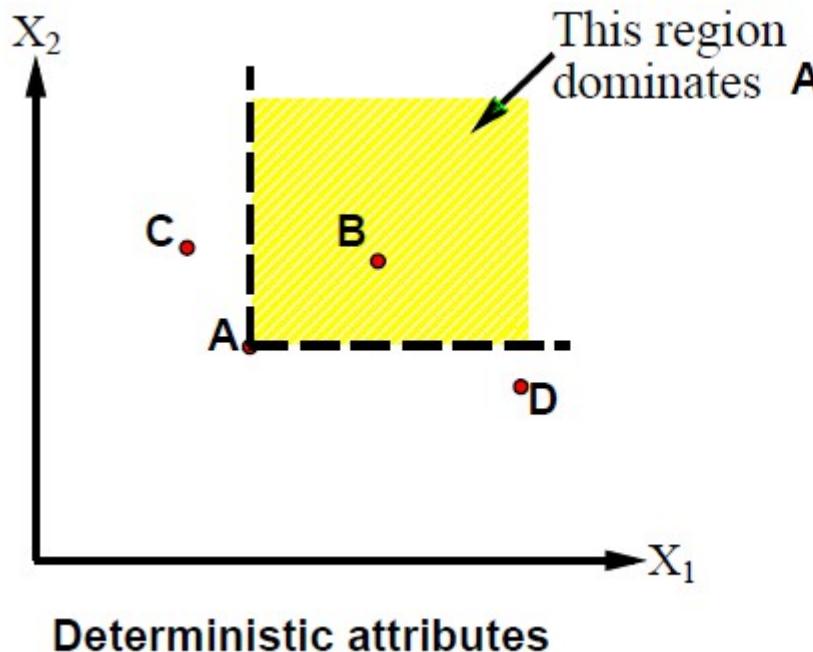
Multi-Attribute Utilities

- Often, the utility does not depend on a single value but on multiple values simultaneously
- Example: Utility of a car depends on
 - Safety
 - Horse-Power
 - Fuel Consumption
 - Size
 - Price
- How can we reason in this case?
 - It is often hard to define a function that maps multiple dimensions X_i to a single utility value $U(X_1, X_2, \dots, X_n)$
→ **Dominance** is a useful concept in such cases

Strict Dominance

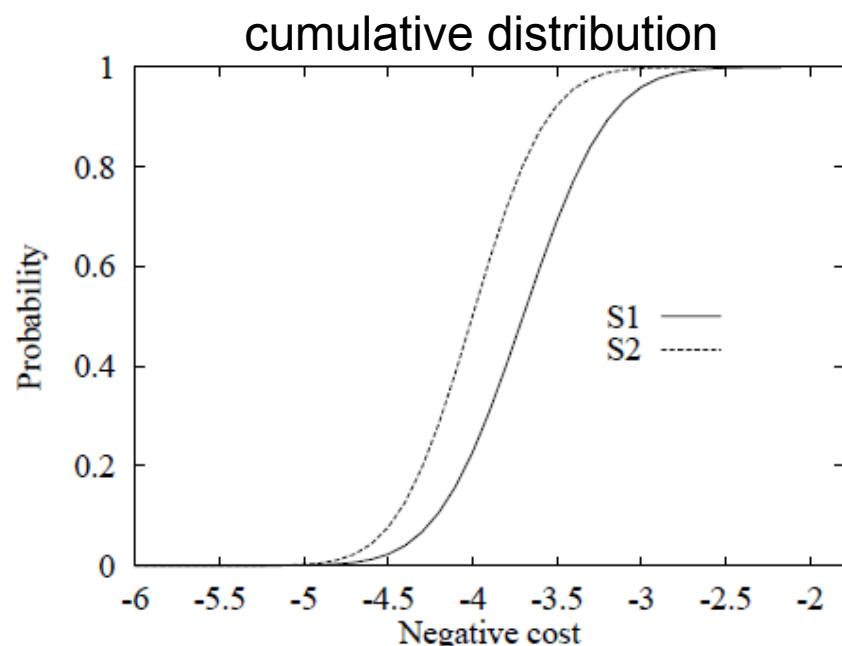
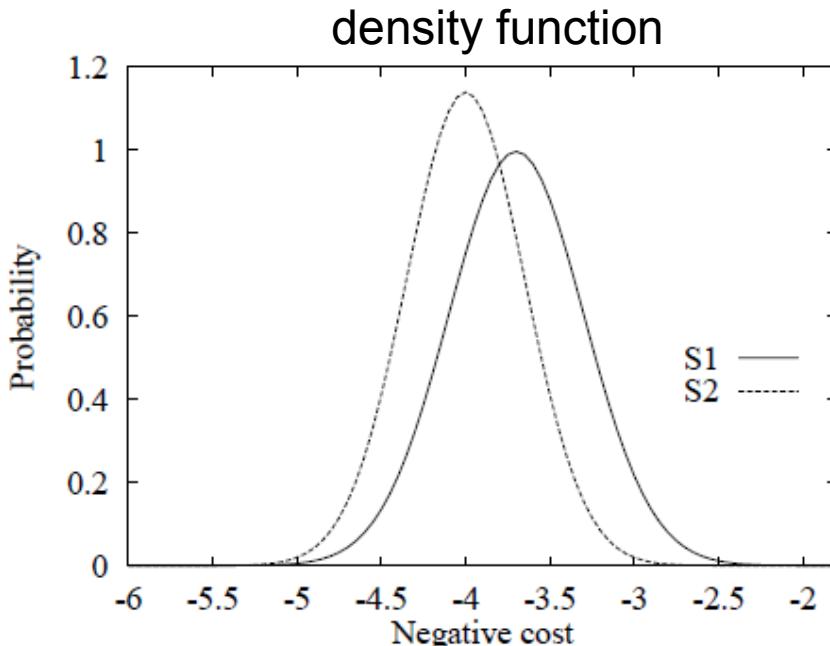
- Scenario A is better than Scenario B if it is better along all dimensions
- Example:
 - 2 dimensions, in both dimensions higher is better (utility grows monotonically with the value)

$$A \gtrsim B \Leftrightarrow U(X_A, Y_A) \geq U(X_B, Y_B) \Leftrightarrow (X_A \geq X_B) \wedge (Y_A \geq Y_B)$$



Stochastic Dominance

- Strict dominance rarely occurs in practice
 - The car that is better in horse-power is rarely also better in fuel consumption and price
- Stochastic dominance:
 - A utility distribution p_1 dominates utility distribution p_2 if the probability of having a utility less or equal a given threshold (cumulative probability) is always lower for p_1 than for p_2



Stochastic Dominance

- If the utility $U(x)$ of action A_1 on attribute X has probability $p_1(x)$ and $U(x)$ occurs with probability $p_2(x)$ for A_2 then

A_1 stochastically dominates A_2 iff

$$\forall x \int_{-\infty}^x p_1(x') dx' \leq \forall x \int_{-\infty}^x p_2(x') dx'$$

- If U is monotonic in x , then A_1 with outcome distribution p_1 stochastically dominates A_2 with outcome distribution p_2 :

$$\int_{-\infty}^{\infty} p_1(x)U(x)dx \geq \int_{-\infty}^{\infty} p_2(x)U(x)dx$$

- because high utility values have a higher probability in p_1
- Extension for Multiple attributes:
 - If there is stochastic dominance along all attributes, then action A_1 dominates A_2

Assessing Stochastic Dominance

- It may seem that stochastic dominance is a concept that is hard to grasp and hard to measure
- But actually it is often quite intuitive and can be established without knowing the exact distribution using qualitative reasoning
- Examples:
 - Construction costs for large building will increase with the distance from the city
 - For higher costs, the probability of such costs are larger for a site further away from the city than for a site that is closer to the city
 - Degree of injury increases with collision speed

Preference (In-)Dependence

- As with probability distribution, it may be hard to establish the utility for all possible value combinations of a multi-attribute utility function $U(X_1, X_2, \dots, X_n)$
- Again, we can simplify things by introducing a notion of dependency
 - Attribute X_1 is preference-independent of attribute X_2 , if knowing X_1 does not influence our preference in X_2
- Examples:
 - Drink preferences depend on the choice of the main course
 - For meat, red wine is preferred over white wine
 - For fish, white wine is preferred over red wine
 - Table preferences do not depend on the choice of the main course
 - A quite table is always preferred, no matter what is ordered

Mutual Preference Independence

- A set of variables is mutually preferentially independent if each subset of variables is preferentially independent of its complement
 - Can be established by checking only attribute pairs (Leontief, 1947)
- If variables are mutually preferentially independent, the value function can be decomposed

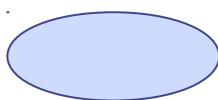
Theorem (Debreu, 1960): mutual P.I. $\Rightarrow \exists$ additive value function:

$$V(S) = \sum_i V_i(X_i(S))$$

-
- **Note:**
 - This only holds for deterministic environments (value functions). For stochastic environments (utility functions), establishing utility-independence is more complex

Decision Networks

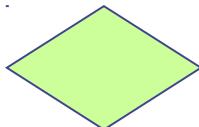
- Extend BNs to handle actions and utilities and enable rational decision making



- Chance nodes: random variables, as in BNs



- Decision nodes: actions that decision maker can take



- Utility/value nodes: the utility of the outcome state.

- Use BN inference methods to solve

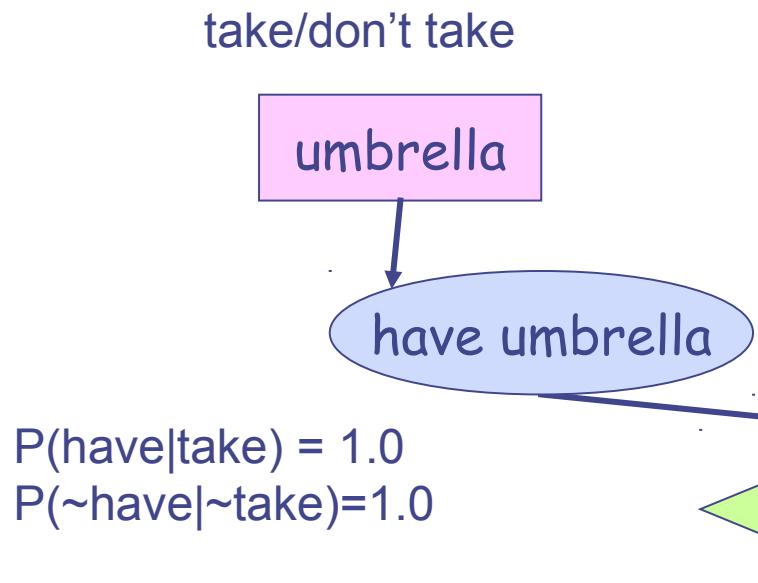
For each value of action node

compute expected value of utility node given action, evidence

Return MEU action

Example: Umbrella Network

- Should I take an umbrella to increase my happiness?



$U(\text{have}, \text{rain}) = -25$
 $U(\text{have}, \sim\text{rain}) = 0$
 $U(\sim\text{have}, \text{rain}) = -100$
 $U(\sim\text{have}, \sim\text{rain}) = 100$

$P(\text{rain}) = 0.4$

f	w	$p(f w)$
sunny	rain	0.3
	rainy	0.7
rainy	no rain	0.8
	no rain	0.2

Evaluating Decision Networks

- Set the evidence variables for current state
- For each possible value of the decision node:
 - Set decision node to that value
 - Calculate the posterior probability of the parent nodes of the utility node, using BN inference
 - Calculate the resulting utility for action
- Return the action with the highest utility
- In the Umbrella example:

$$EU(\text{take}) = 0.4 \times -25 + 0.6 \times 0 = -10$$

$$EU(\neg\text{take}) = 0.4 \times -100 + 0.6 \times 100 = +20$$

- My expected utility is higher if I don't take the umbrella
- But note that we did not take the weather report into account!

Value of Information

- Decision Networks allow to measure the value of information

Example: buying oil drilling rights

Two blocks A and B , exactly one has oil, worth k

Prior probabilities 0.5 each, mutually exclusive

Current price of each block is $k/2$

“Consultant” offers accurate survey of A . Fair price?

Solution: compute expected value of information

= expected value of best action given the information

minus expected value of best action without information

Survey may say “oil in A” or “no oil in A”, **prob. 0.5 each (given!)**

= $[0.5 \times$ value of “buy A” given “oil in A”

$+ 0.5 \times$ value of “buy B” given “no oil in A”]

$- 0$

$= (0.5 \times k/2) + (0.5 \times k/2) - 0 = k/2$

Value of Perfect Information (VPI)

- General Idea:
 - Compute the Expected Utility of an action without the evidence
 - Compute the Expected Utility of the action over all possible outcomes of the evidence
 - The difference is the value of knowing the evidence.

- More formally

- current evidence E , α is best of actions A

$$EU(\alpha | E) = \max_A \sum_i U(\text{Result}_i(A)) \cdot P(\text{Result}_i(A) | Do(A), E)$$

- after obtaining new evidence E_j

$$EU(\alpha | E, E_j) = \max_A \sum_i U(\text{Result}_i(A)) \cdot P(\text{Result}_i(A) | Do(A), E, E_j)$$

- Difference between expected value over all possible outcomes e_{jk} of E_j and the expected value without E_j

$$VPI_E(E_j) = \left(\sum_k P(E_j = e_{jk} | E) \cdot EU(\alpha_{e_{jk}} | E, E_j = e_{jk}) \right) - EU(\alpha | E)$$

Properties of VPI

Nonnegative—in expectation, not post hoc

$$\forall j, E \ VPI_E(E_j) \geq 0$$

Nonadditive—consider, e.g., obtaining E_j twice

$$VPI_E(E_j, E_k) \neq VPI_E(E_j) + VPI_E(E_k)$$

Order-independent

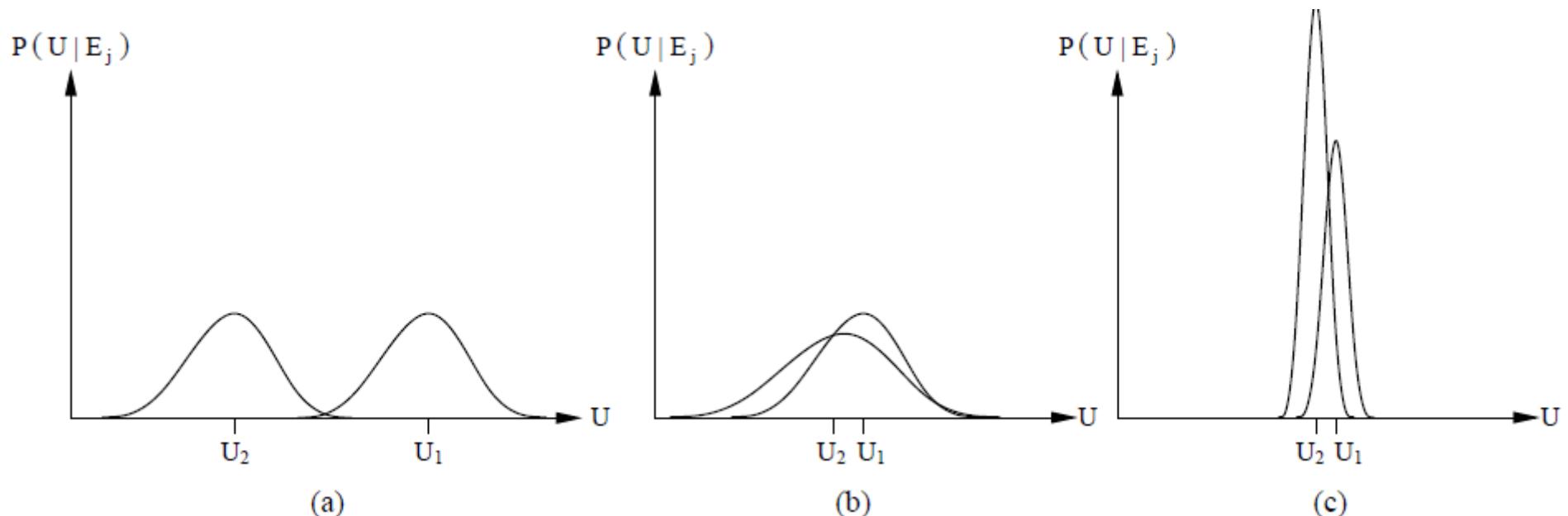
$$VPI_E(E_j, E_k) = VPI_E(E_j) + VPI_{E,E_j}(E_k) = VPI_E(E_k) + VPI_{E,E_k}(E_j)$$

Note: when more than one piece of evidence can be gathered, maximizing VPI for each to select one is not always optimal

⇒ evidence-gathering becomes a **sequential** decision problem

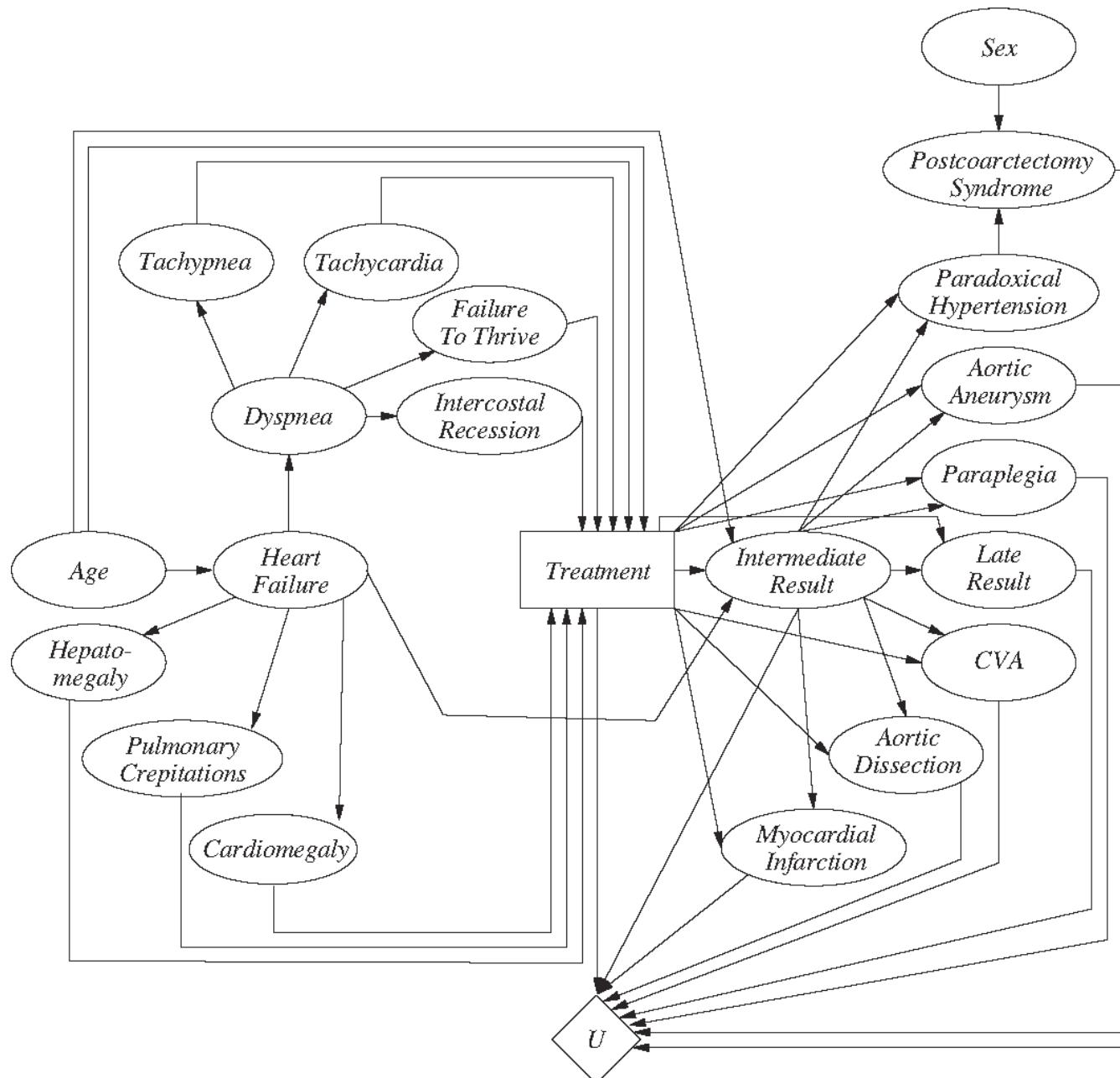
Qualitative Behaviors

- The value of information depends on the distribution of the new utility values in dependence of their old estimates



- a) Choice is obvious, information worth little
- b) Choice is nonobvious, information worth a lot
- c) Choice is nonobvious, information worth little

Real-World Decision Network



Learning from Observations

Russell & Norvig - AIMA2e

Ioan Alfred Letia

<http://cs-gw.utcluj.ro/~letia>

Outline

- Forms of learning
- Inductive learning
- Learning decision trees
- Ensemble learning
- Why learning works?

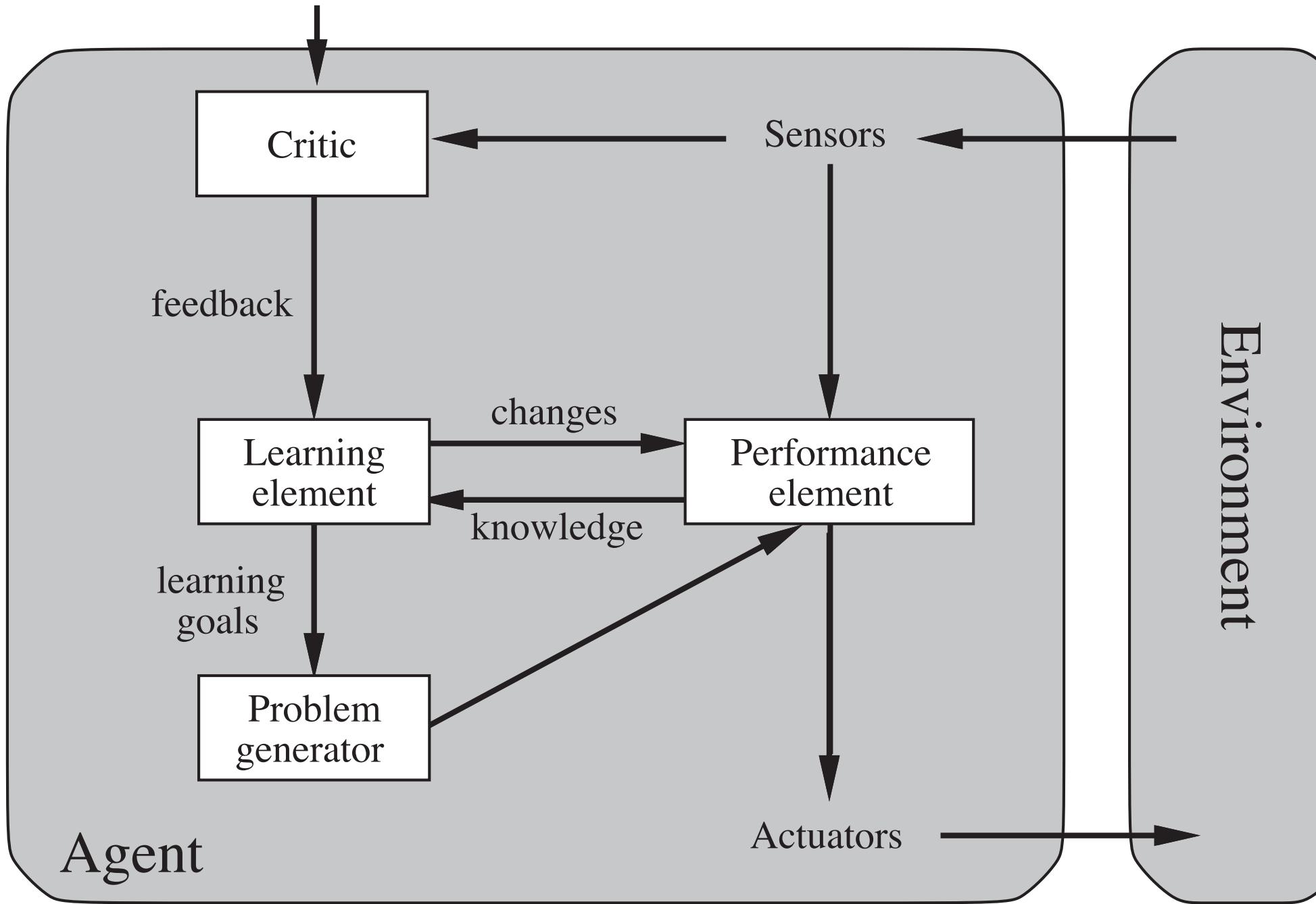
Forms of Learning

Design of learning element affected by

- Components of the performance element:
 1. Direct mapping from conditions on the current state to actions
 2. Means to infer relevant properties of the world from percepts
 3. Information about the way the world evolves and about the results of possible agent actions
 4. Utility information indicating desirability of states
 5. Action-value information indicating desirability
 6. Goals that describe classes of states
- Available feedback: **supervised learning, unsupervised learning, reinforcement learning**
- Representation of the learned information
- Prior knowledge

Learning Agents

Performance standard



Inductive Learning (a.k.a. Science)

- Simplest form: learn a function from examples (**tabula rasa**)
- f is the target function

O	O	X
	X	
X		

, +1

- An example is a pair $x, f(x)$, e.g.,
- Problem: find a(n) hypothesis h such that $h \approx f$ given a training set of examples
- **This is a highly simplified model of real learning**
 - Ignores prior knowledge
 - Assumes a deterministic, observable “environment”
 - Assumes examples are **given**
 - Assumes that the agent **wants** to learn f —why?

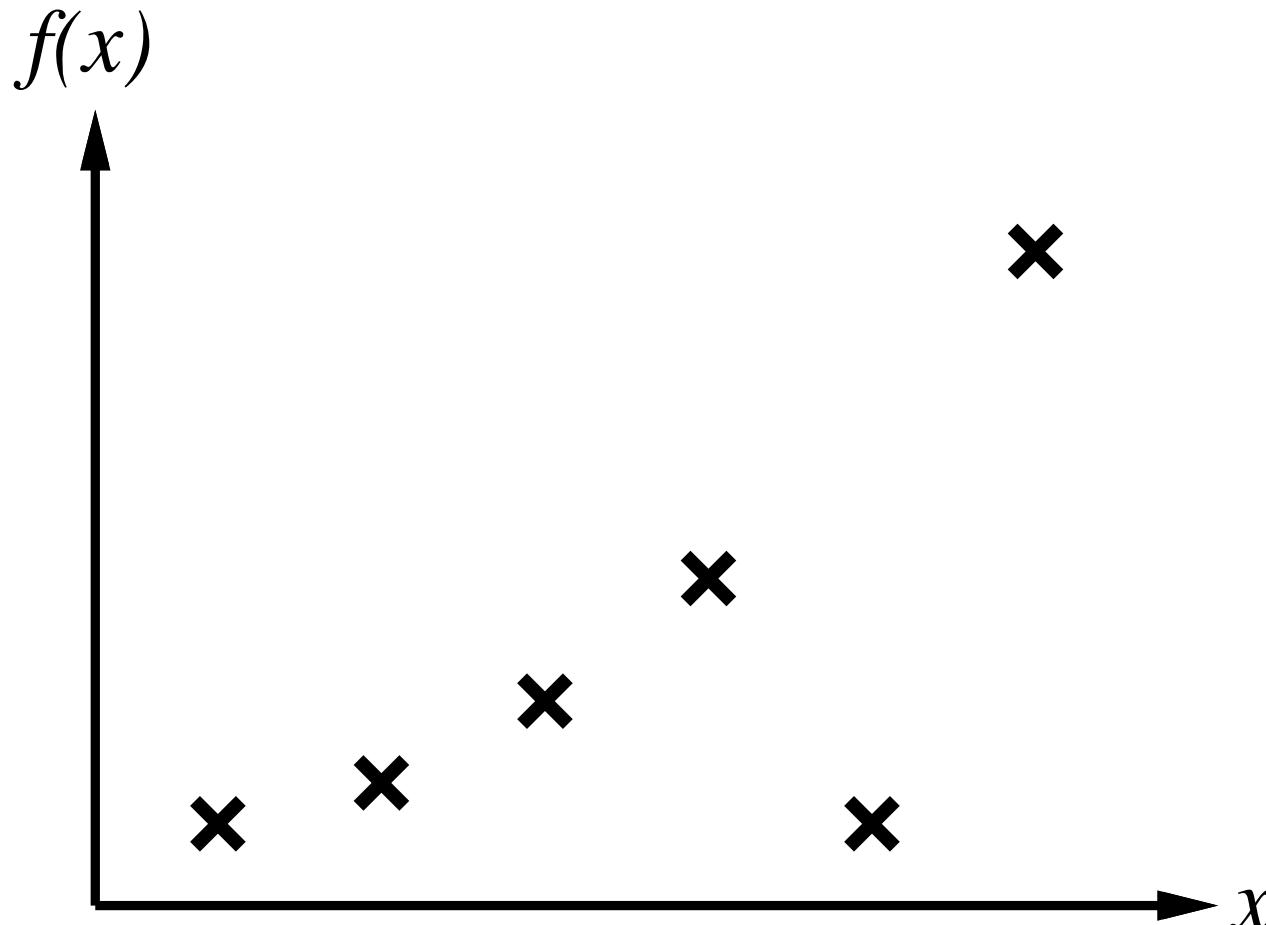
Learning Element

Performance element	Component	Representation	Feedback
Alpha–beta search	Eval. fn.	Weighted linear function	Win/loss
Logical agent	Transition model	Successor–state axioms	Outcome
Utility–based agent	Transition model	Dynamic Bayes net	Outcome
Simple reflex agent	Percept–action fn	Neural net	Correct action

- Design of learning element is dictated by
 - what type of performance element is used
 - which functional component is to be learned
 - how that functional component is represented
 - what kind of feedback is available
- Example scenarios
- Supervised learning: correct answers for each instance
- Reinforcement learning: occasional rewards

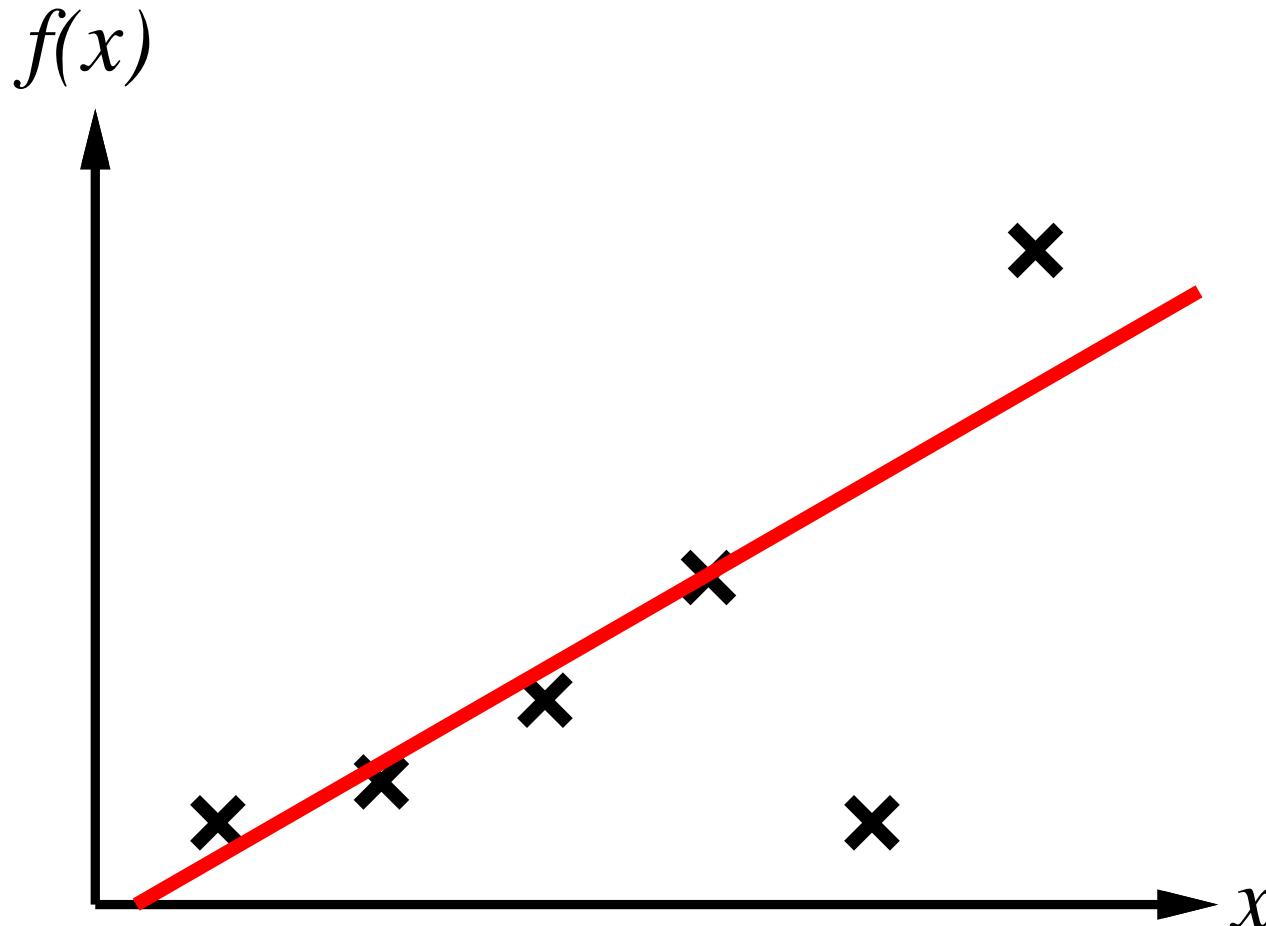
Inductive Learning Method

- Construct/adjust h to agree with f on training set (h is consistent if it agrees with f on all examples)
- E.g., curve fitting:



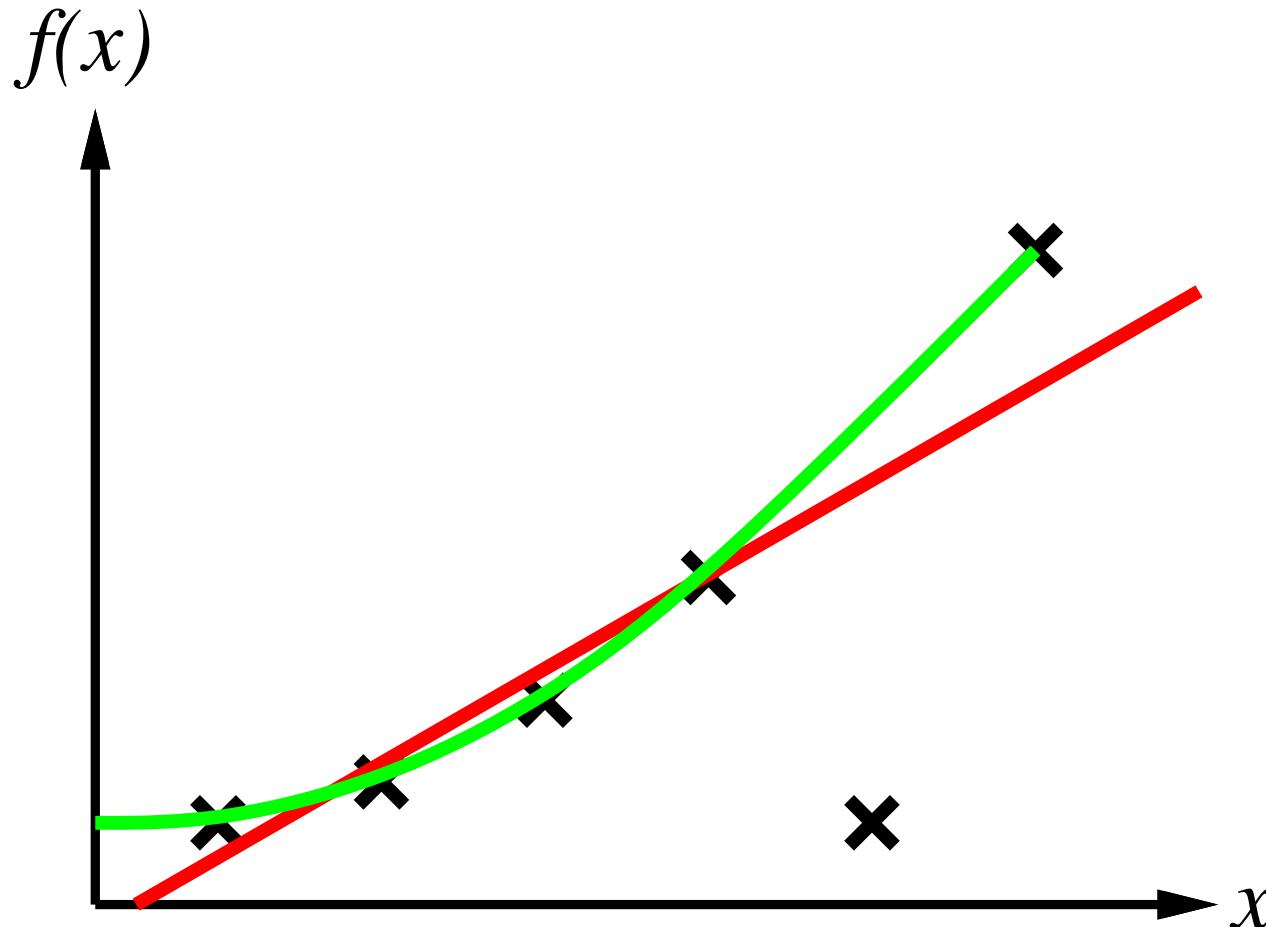
Inductive Learning Method

- Construct/adjust h to agree with f on training set (h is consistent if it agrees with f on all examples)
- E.g., curve fitting:



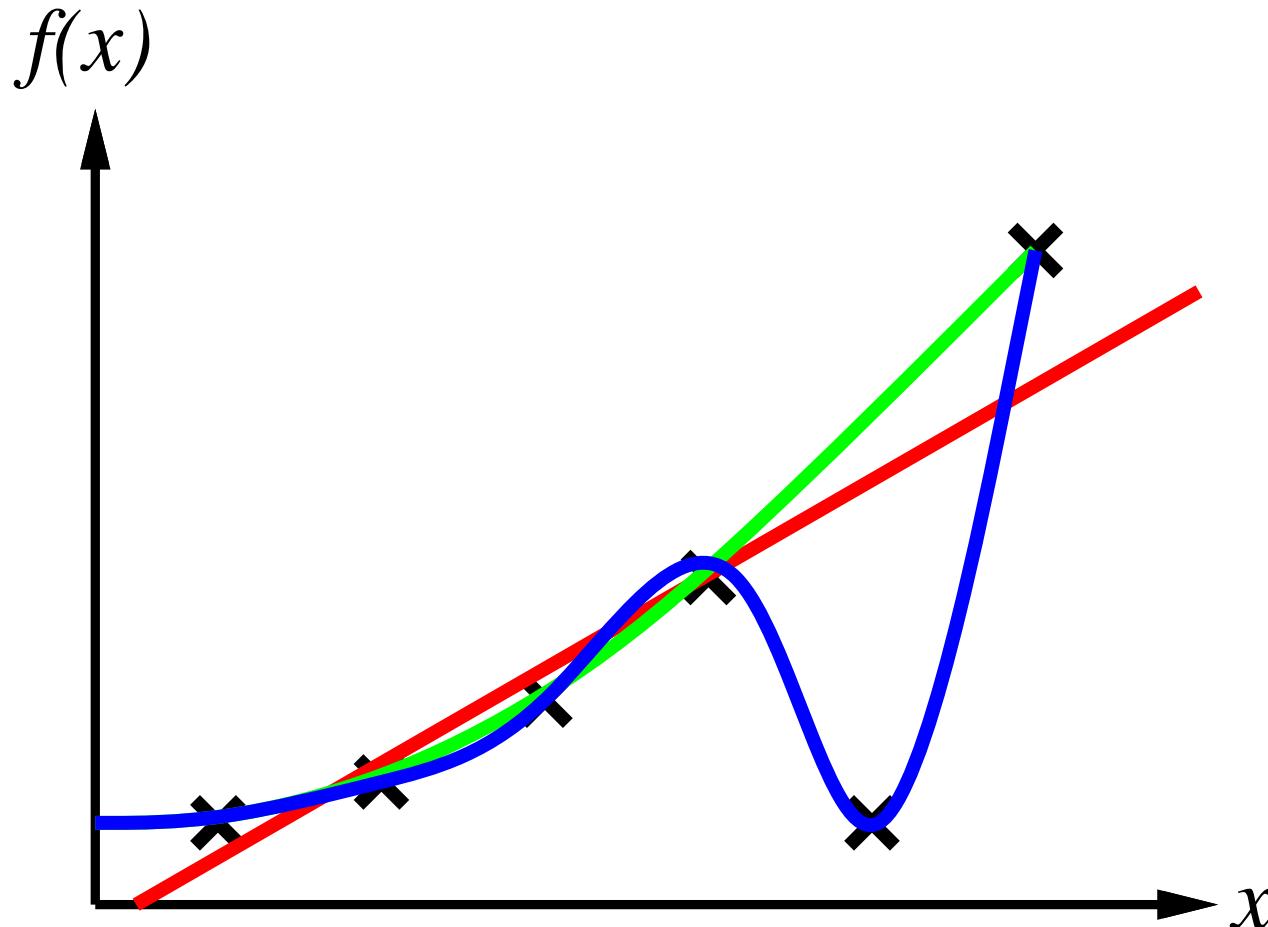
Inductive Learning Method

- Construct/adjust h to agree with f on training set (h is consistent if it agrees with f on all examples)
- E.g., curve fitting:



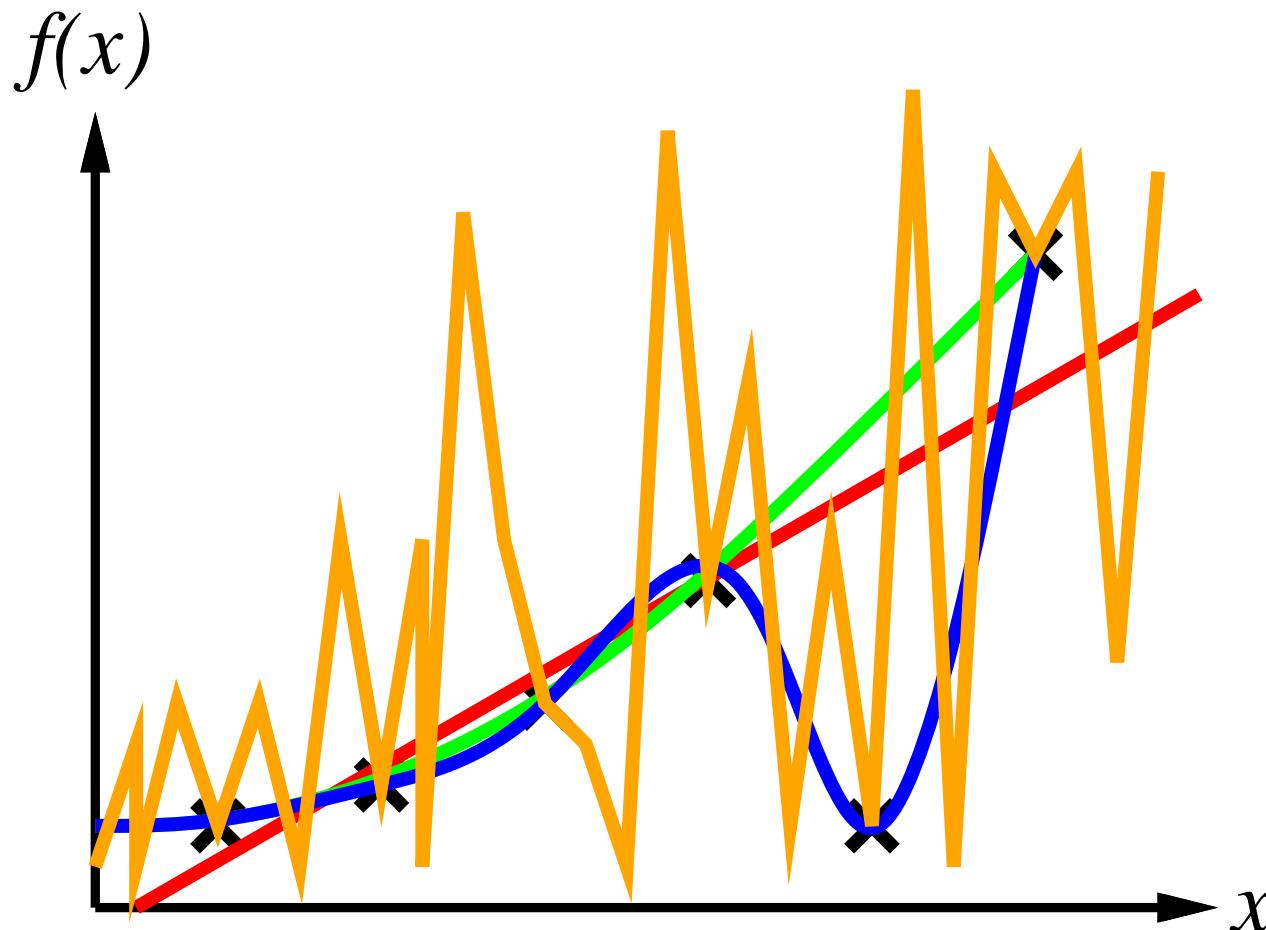
Inductive Learning Method

- Construct/adjust h to agree with f on training set (h is consistent if it agrees with f on all examples)
- E.g., curve fitting:



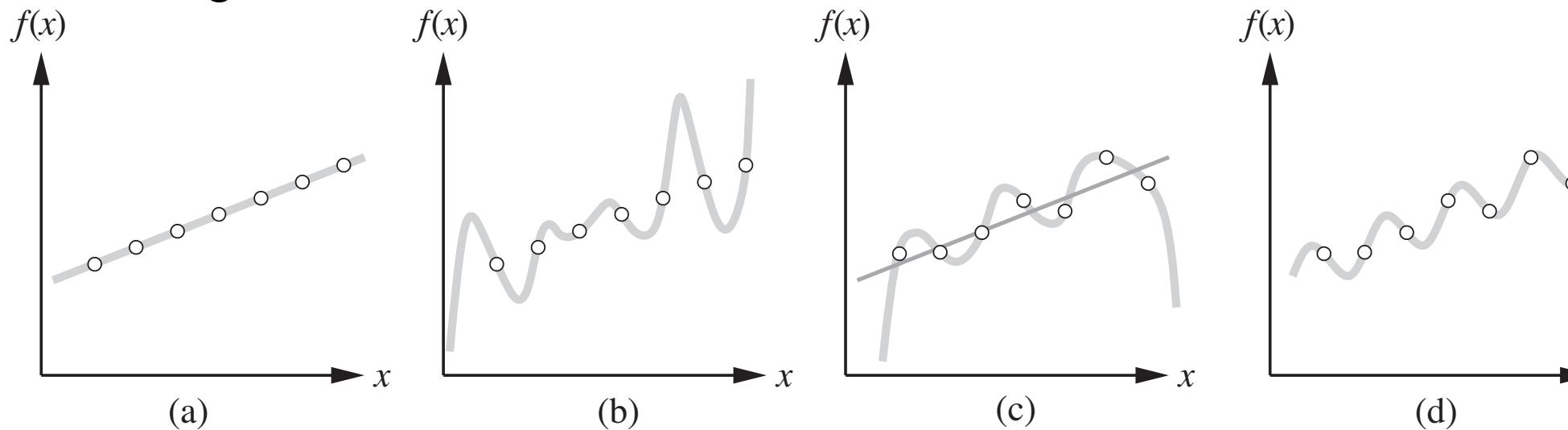
Inductive Learning Method

- Construct/adjust h to agree with f on training set (h is consistent if it agrees with f on all examples)
- E.g., curve fitting:



Inductive Learning

- Construct/adjust h to agree with f on training set (h is consistent if it agrees with f on all examples)
- Curve fitting: hypothesis space H – polynomials of degree at most k



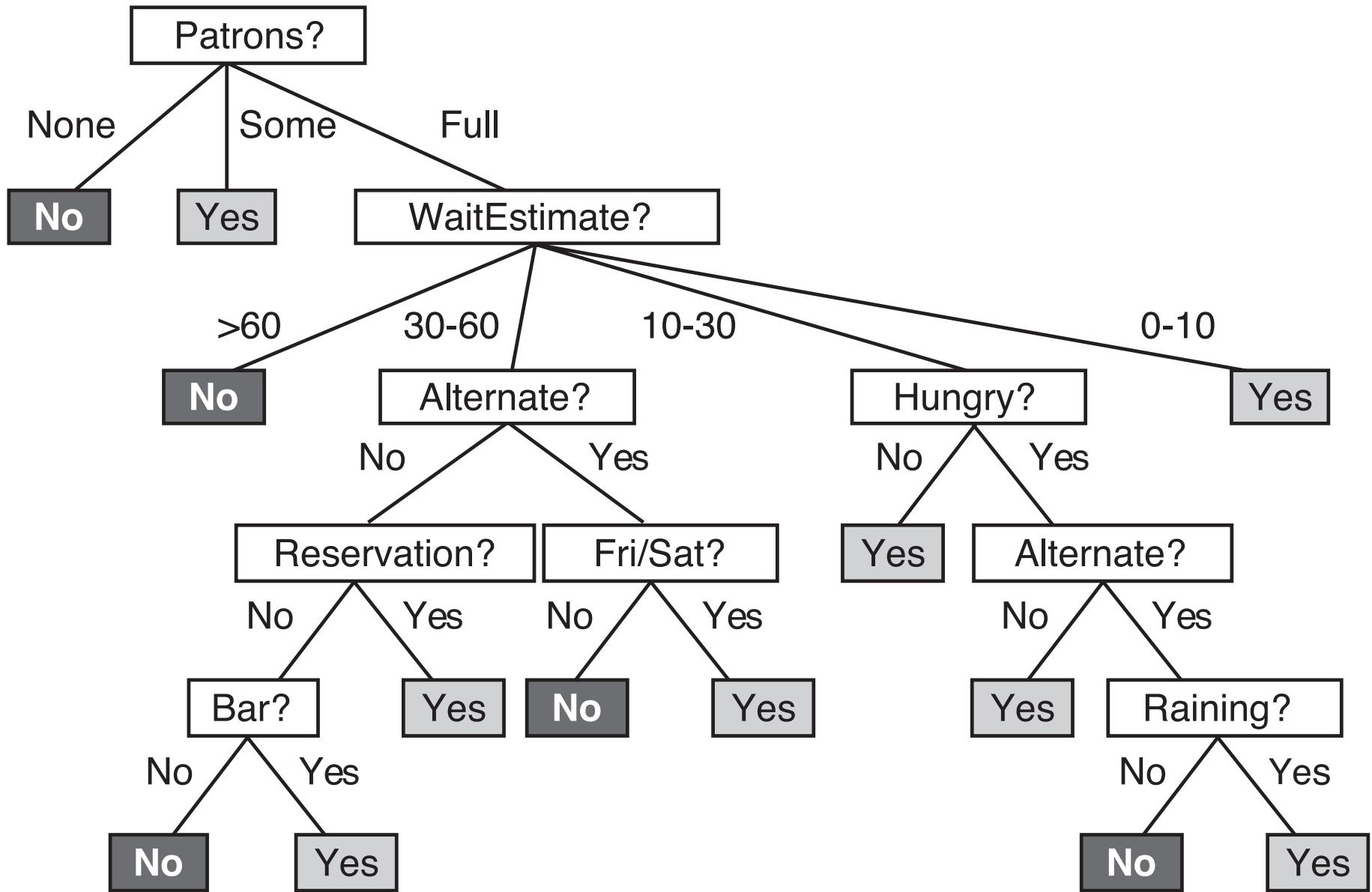
- A good hypothesis will **generalize well**
- How to choose from among multiple consistent hypotheses?
- **Ockham's razor:** prefer the simplest hypothesis consistent with the data

Learning Decision Trees

Properties or *attributes* available to describe examples in the domain of waiting for a restaurant table:
goal predicate *WillWait*

- **Alternate**: whether there is a suitable alternative
- **Bar**: whether the restaurant has a comfortable bar
- **Fri/Sat**: true on Fridays and Saturdays
- **Hungry**: whether we are hungry
- **Patrons**: how many people are in the restaurant
- **Price**: restaurant's price range
- **Raining**: whether it is raining outside
- **Reservation**: whether we made a reservation
- **Type**: the kind of restaurant
- **WaitEstimate**: the wait estimated by the host

Decision Trees



Expressiveness of Decision Trees

- Any particular decision tree hypothesis for the *WillWait* predicate can be seen as an assertion of the form
$$\forall s \text{ } WillWait}(s) \Leftrightarrow (P_1(s) \vee P_2(s) \dots \vee P_n(s))$$
where each condition $P_i(s)$ is a conjunction of tests
- We cannot use decision trees to represent tests that refer to two or more different objects
$$\exists r_2 \text{ } Nearby(r_2, r) \vee Price(r, p) \vee Price(r_2, p_2) \vee Cheaper(p_2, p)$$
- Decision trees can express any function of the input attributes.
- For Boolean functions, truth table row \rightarrow path to leaf
- Trivially, \exists a consistent decision tree for any training set with one path to leaf for each example (unless f nondeterministic in x) but it probably won't generalize to new examples
- Prefer to find more **compact** decision trees

Hypothesis Spaces

- How many distinct decision trees with n Boolean attributes??
-

Hypothesis Spaces

- How many distinct decision trees with n Boolean attributes??
- = number of Boolean functions

Hypothesis Spaces

- How many distinct decision trees with n Boolean attributes??
- = number of Boolean functions
- = number of distinct truth tables with 2^n rows

Hypothesis Spaces

- How many distinct decision trees with n Boolean attributes??
- = number of Boolean functions
- = number of distinct truth tables with 2^n rows = 2^{2^n}

Hypothesis Spaces

- How many distinct decision trees with n Boolean attributes??
- = number of Boolean functions
- = number of distinct truth tables with 2^n rows = 2^{2^n}
- E.g., with 6 Boolean attributes, there are
18,446,744,073,709,551,616 trees

Hypothesis Spaces

- How many distinct decision trees with n Boolean attributes??
- = number of Boolean functions
- = number of distinct truth tables with 2^n rows = 2^{2^n}
- E.g., with 6 Boolean attributes, there are
18,446,744,073,709,551,616 trees
- How many purely conjunctive hypotheses ($\text{Hungry} \wedge \neg \text{Rain}$)??

Hypothesis Spaces

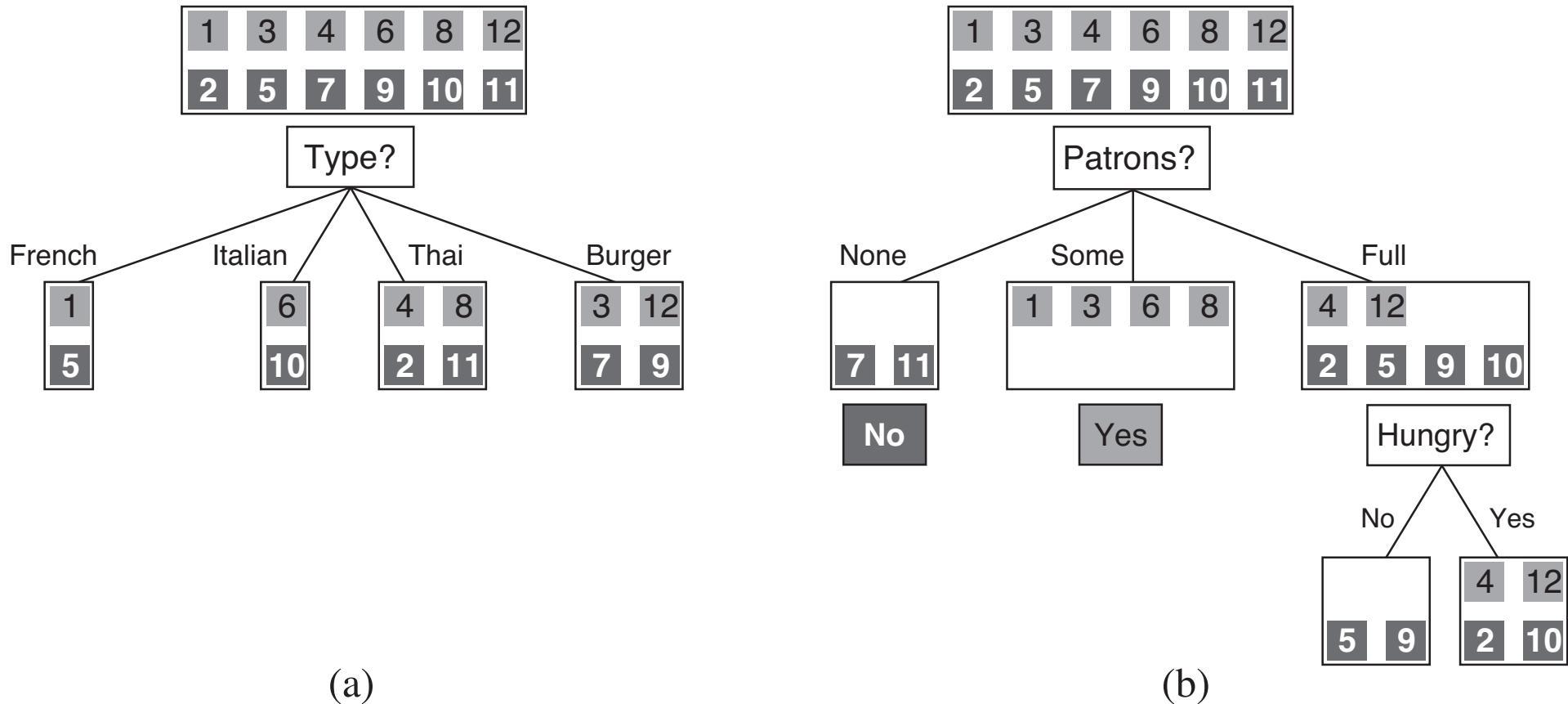
- How many distinct decision trees with n Boolean attributes??
- = number of Boolean functions
- = number of distinct truth tables with 2^n rows = 2^{2^n}
- E.g., with 6 Boolean attributes, there are
18,446,744,073,709,551,616 trees
- How many purely conjunctive hypotheses ($\text{Hungry} \wedge \neg \text{Rain}$)??
- Each attribute can be in (positive), in (negative), or out
 $\Rightarrow 3^n$ distinct conjunctive hypotheses
- More expressive hypothesis space
 - increases chance that target function can be expressed
 - increases number of hypotheses consistent w/ training set
 \Rightarrow may get worse predictions

Inducing Decision Trees

- Examples described by attribute values (Boolean, discrete, continuous, etc.)
E.g., situations where I will/won't wait for a table
- Classification of examples is positive (T) or negative (F)

example	Attributes										Target <i>WillWait</i>
	<i>Alt</i>	<i>Bar</i>	<i>Fri</i>	<i>Hun</i>	<i>Pat</i>	<i>Price</i>	<i>Rain</i>	<i>Res</i>	<i>Type</i>	<i>Est</i>	
X_1	T	F	F	T	Some	\$\$\$	F	T	French	0–10	T
X_2	T	F	F	T	Full	\$	F	F	Thai	30–60	F
X_3	F	T	F	F	Some	\$	F	F	Burger	0–10	T
X_4	T	F	T	T	Full	\$	F	F	Thai	10–30	T
X_5	T	F	T	F	Full	\$\$\$	F	T	French	>60	F
X_6	F	T	F	T	Some	\$\$	T	T	Italian	0–10	T
X_7	F	T	F	F	None	\$	T	F	Burger	0–10	F
X_8	F	F	F	T	Some	\$\$	T	T	Thai	0–10	T
X_9	F	T	T	F	Full	\$	T	F	Burger	>60	F
X_{10}	T	T	T	T	Full	\$\$\$	F	T	Italian	10–30	F
X_{11}	F	F	F	F	None	\$	F	F	Thai	0–10	F
X_{12}	T	T	T	T	Full	\$	F	F	Burger	30–60	T

Choosing an Attribute



- Idea: a good attribute splits the examples into subsets that are (ideally) “all positive” or “all negative”
- Patrons?* is a better choice—gives **information** about the classification

Decision Tree Learning

function DECISION-TREE-LEARNING(*examples, attributes, default*) **returns** a decision tree

inputs: examples, set of examples

attributes, set of attributes

default, default value for the goal predicate

if `examples` **is empty then return** `default`

else if all examples have the same classification then return the classification

else if *attributes* is empty **then return** MAJORITY-VALUE(*examples*)

else

best \leftarrow CHOOSE-ATTRIBUTE(*attributes*, *examples*)

tree \leftarrow a new decision tree with root test *best*

for each value v_i of *best* do

`examplesi` \leftarrow {elements of `examples` with `best` = v_i }

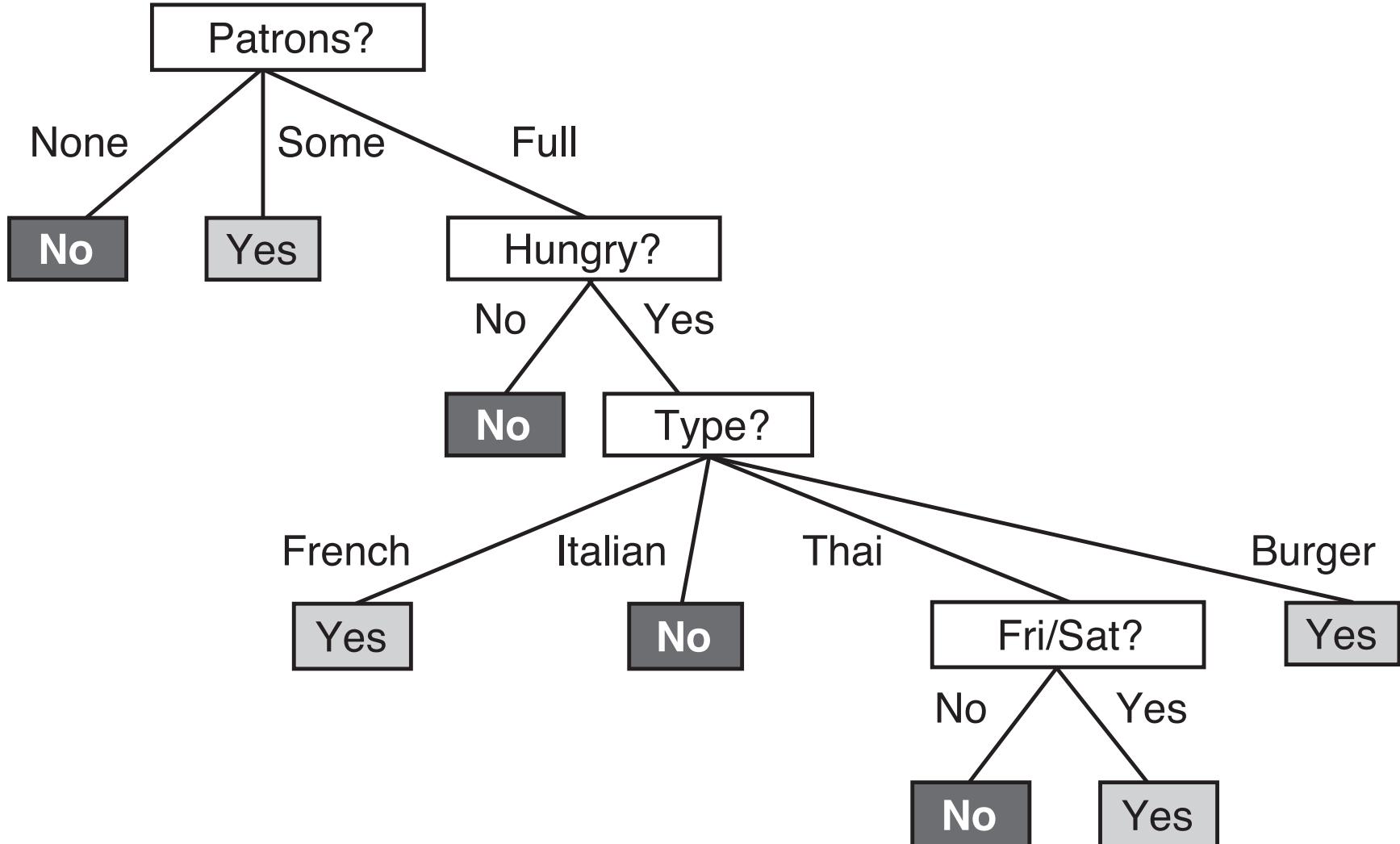
$\text{subtree} \leftarrow \text{DECISION-TREE-LEARNING}(\text{examples}_i, \text{attributes} - \text{best}, \text{MAJORITY-VALUE}(\text{examples}))$

add a branch to *tree* with label v_i and subtree *subtree*

end

return tree

Induced Restaurant Tree



- Did not include tests for *Reservation* and *Raining* because it can classify the examples without them
- It has not seen a case where the wait is 0–10 minutes but the restaurant is full \Rightarrow error

Choosing Attribute Tests

- Information answers questions
- The more clueless I am about the answer initially, the more information is contained in the answer
- Information content I of the actual answer when possible answers v_i have probabilities $P(v_i)$ is

$$I(P(v_1), \dots, P(v_n)) = \sum_{i=1}^n -P(v_i) \log_2 P(v_i)$$

- For the tossing of a fair coin

$$I\left(\frac{1}{2}, \frac{1}{2}\right) = -\frac{1}{2} \log_2 \frac{1}{2} - \frac{1}{2} \log_2 \frac{1}{2} = 1 \text{ bit}$$

- If the coin is loaded to give 99% heads, we get
 $I(1/100, 99/100) = 0.08$ bits
- Training set contains p positive and n negative examples; estimate of information content in answer

$$I\left(\frac{p}{p+n}, \frac{n}{p+n}\right) = -\frac{p}{p+n} \log_2 \frac{p}{p+n} - \frac{n}{p+n} \log_2 \frac{n}{p+n}$$

- For 12 restaurant examples, $p=n=6$ so we need 1 bit

Information

- Any attribute divides the training set E into subsets E_i , each of which have p_i positive and n_i negative examples
- After testing attribute A we will need

$$Remainder(A) = \sum_{i=1}^v \frac{p_i + n_i}{p + n} I\left(\frac{p_i}{p_i + n_i}, \frac{n_i}{p_i + n_i}\right)$$

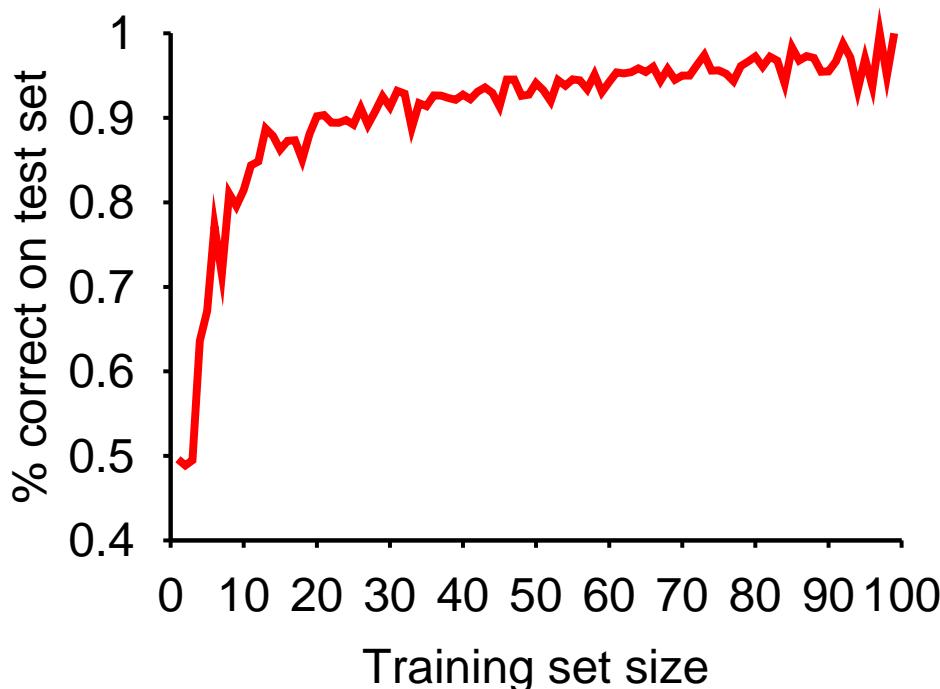
bits of information to classify the example

$$Gain(A) = I\left(\frac{p}{p+n}, \frac{n}{p+n}\right) - Remainder(A)$$

$$Gain(Patrons) = 1 - \left[\frac{2}{12} I(0, 1) + \frac{4}{12} I(1, 0) + \frac{6}{12} I\left(\frac{2}{6}, \frac{4}{6}\right) \right] \approx 0.541 bits$$

$$Gain(Type) = 1 - \left[\frac{2}{12} I\left(\frac{1}{2}, \frac{1}{2}\right) + \frac{2}{12} I\left(\frac{1}{2}, \frac{1}{2}\right) + \frac{4}{12} I\left(\frac{2}{4}, \frac{2}{4}\right) + \frac{4}{12} I\left(\frac{2}{4}, \frac{2}{4}\right) \right] = 0$$

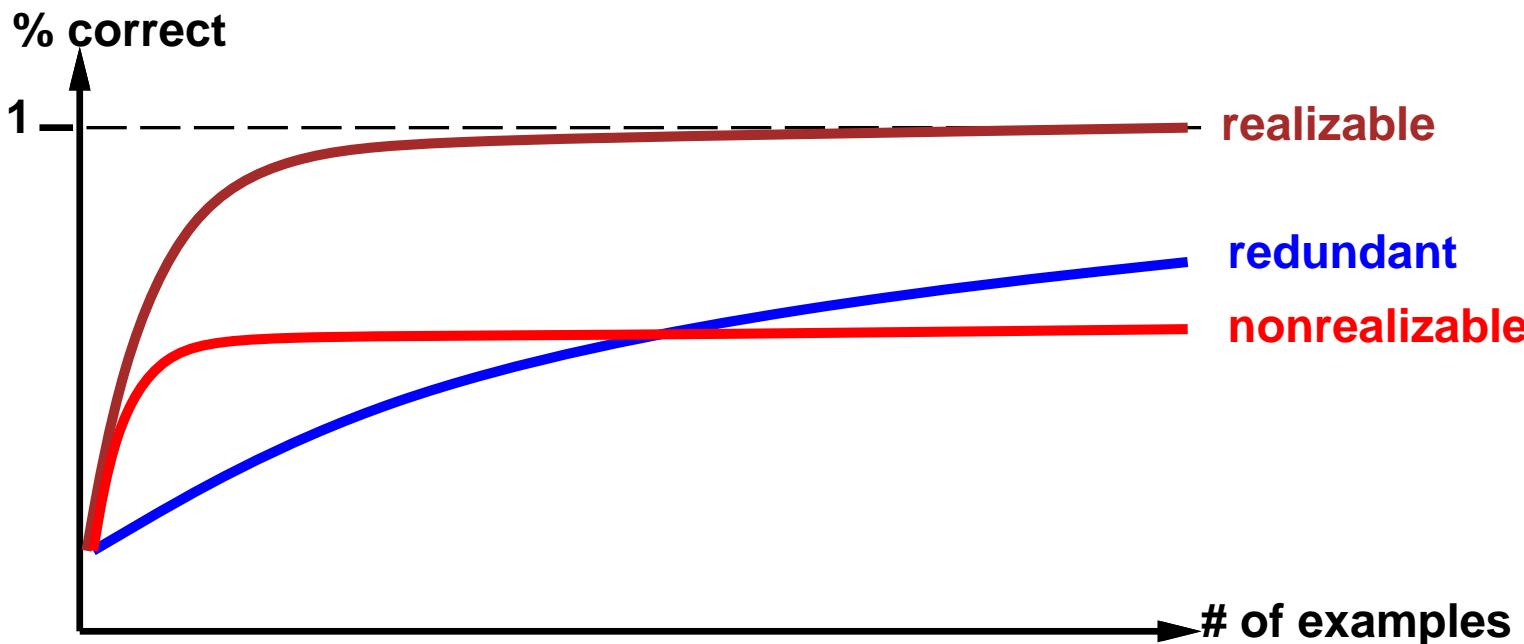
Assessing Performance of Learning



- How do we know that $h \approx f$? (Hume's **Problem of Induction**)
 1. Use theorems of computational/statistical learning theory
 2. Try h on a new test set of examples (use **same distribution over example space** as training set)
- **Learning curve** = % correct on test set as a function of training set size

Assessing Performance of Learning 2

- Learning curve depends on
 - realizable (can express target function) vs. non-realizable
 - non-realizability can be due to missing attributes or restricted hypothesis class (e.g., thresholded linear function)
 - redundant expressiveness (e.g., loads of irrelevant attributes)



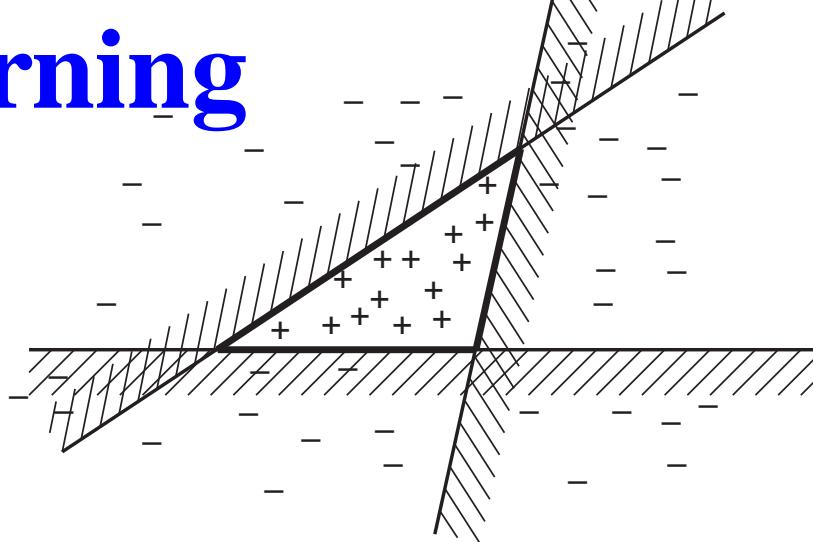
Noise and Overfitting

- Overfitting: whenever there is a large set of possible hypotheses, one has to be careful not to use the resulting freedom to find meaningless "regularity"
- Decision tree pruning: prevents recursive splitting on attributes that are not clearly relevant, even when the data at that node in the tree is not uniformly classified
- Null hypothesis: assumption \Rightarrow no underlying pattern
- Expected: $\hat{p}_i = p \times \frac{p_i + n_i}{p+n}$ $\hat{n}_i = n \times \frac{p_i + n_i}{p+n}$
- Convenient measure of the total deviation:

$$D = \sum_{i=1}^v \frac{(p_i - \hat{p}_i)^2}{\hat{p}_i} + \frac{(n_i - \hat{n}_i)^2}{\hat{n}_i}$$

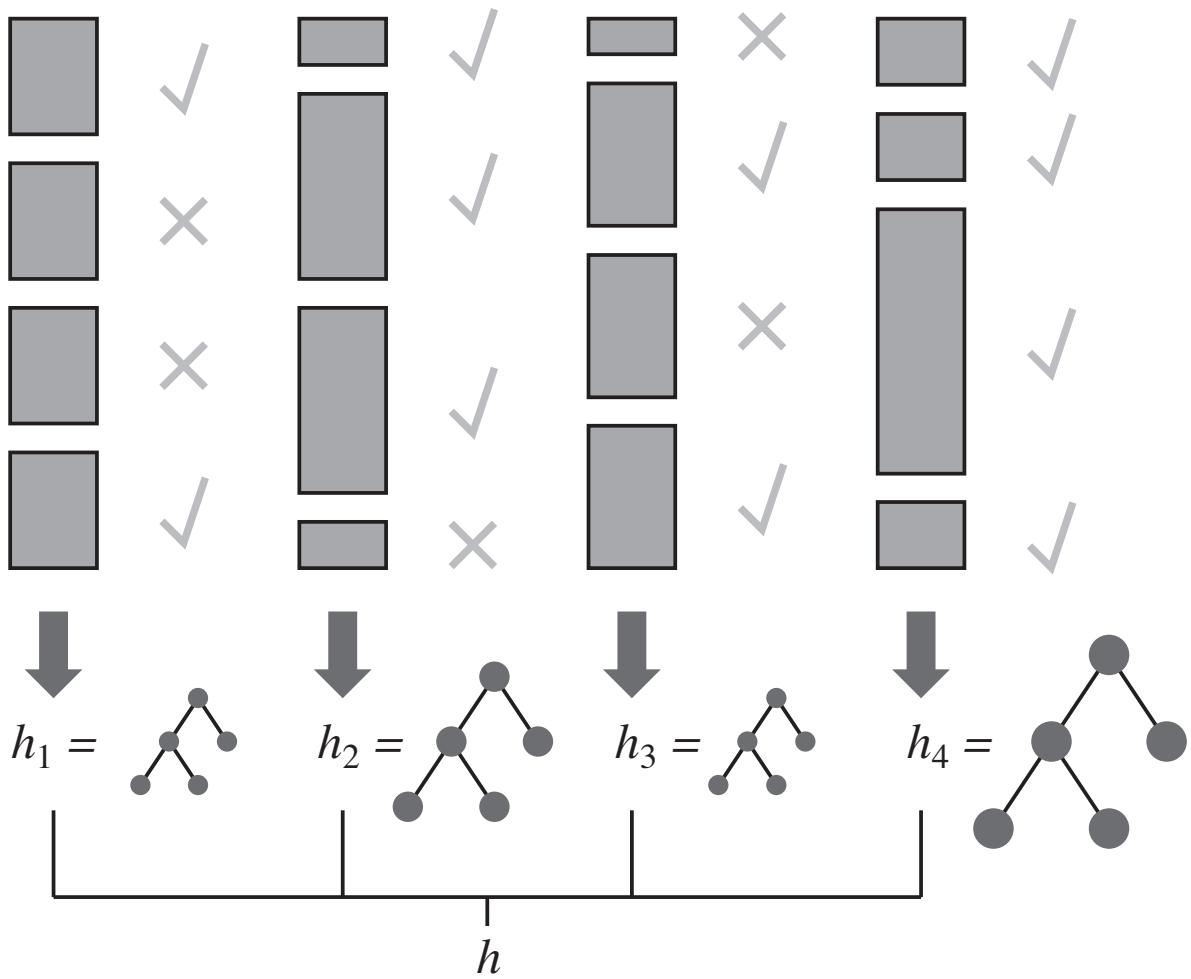
- χ^2 pruning: uses χ tables to calculate if the attribute is really irrelevant
- Cross-validation: estimates prediction of unseen data

Ensemble Learning



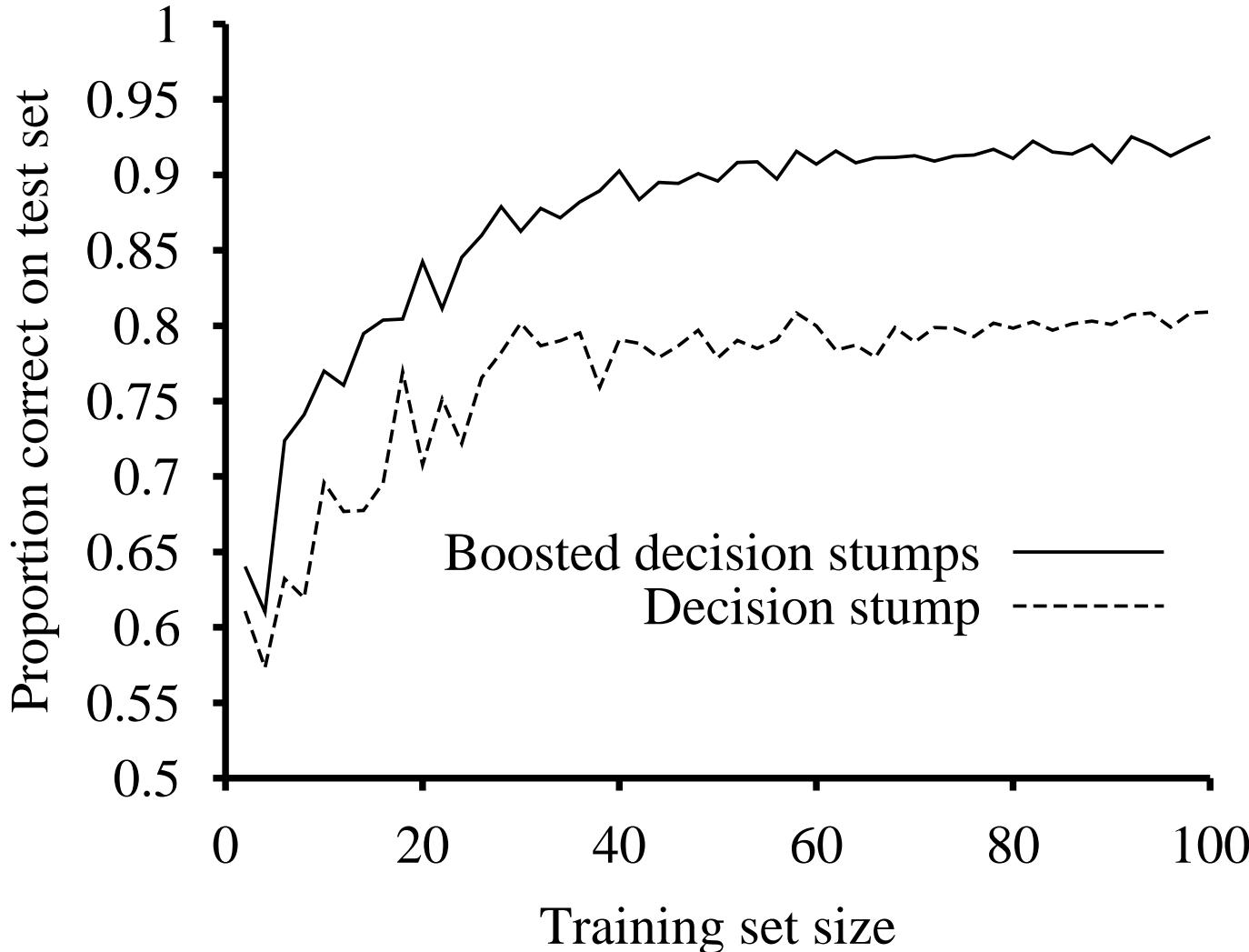
- Ensemble of $M = 5$ hypotheses – combine their predictions using simple majority voting: at least 3 of the 5 hypotheses have to missclassify
- If each hypothesis h_i has error p and errors made by each hypothesis are independent then for p small the probability of a large number of misclassifications occurring is minuscule
- If original hypothesis space allows for simple and efficient learning then the ensemble method provides a way to learn a much more expressive class of hypotheses without incurring much additional computational complexity

Boosting



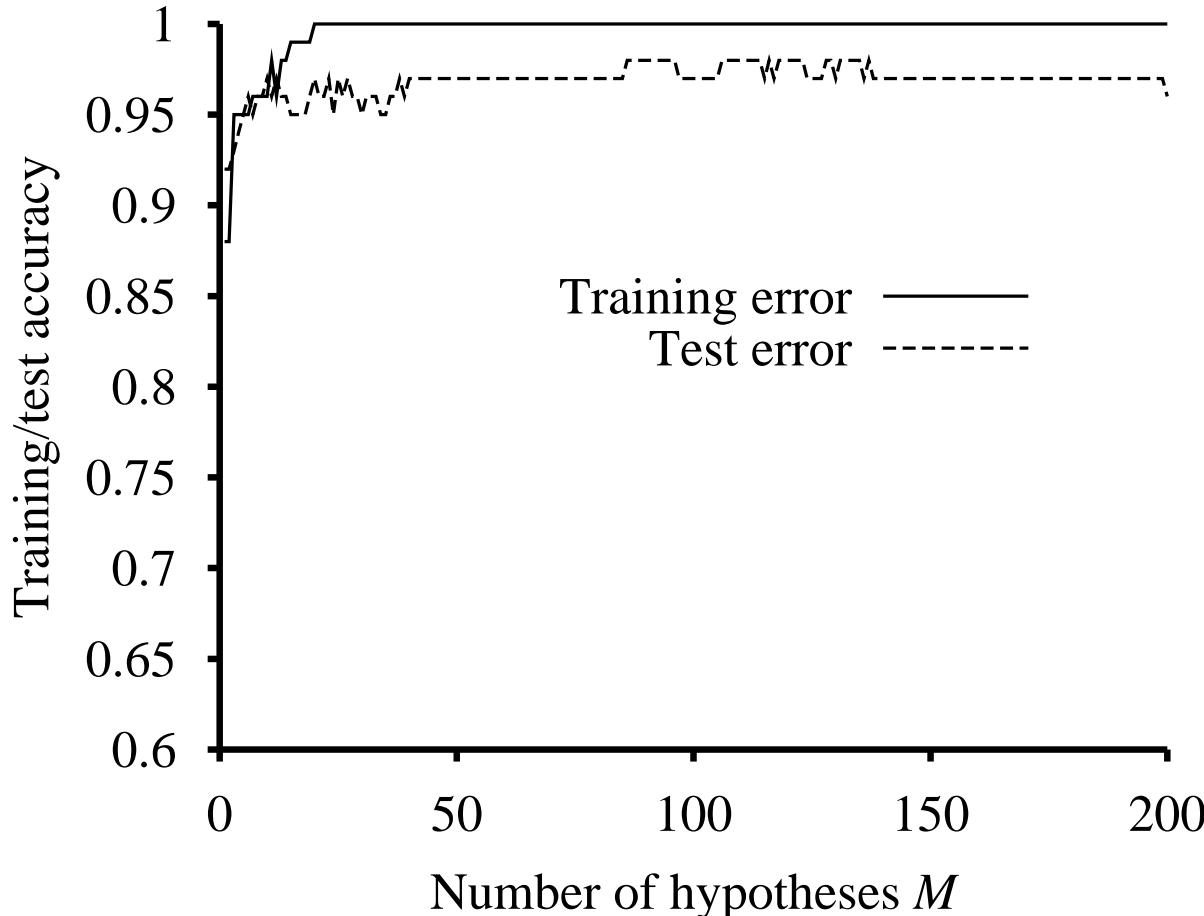
- **Weighted training set:** each example has an associated weight $w_j \geq 0$: the higher the weight of an example, the higher is the importance attached to it during learning
- Starts with $w_j = 1$ for all examples $\Rightarrow h_1$
- Increase weight of misclassified examples, decrease weight of correctly classified

Decision Stumps



- Performance with $M=5$ on the restaurant data
- Unboosted decision stumps are not very effective, reaching prediction of 81% on 100 examples
- Boosting reaches 93% after 100 examples

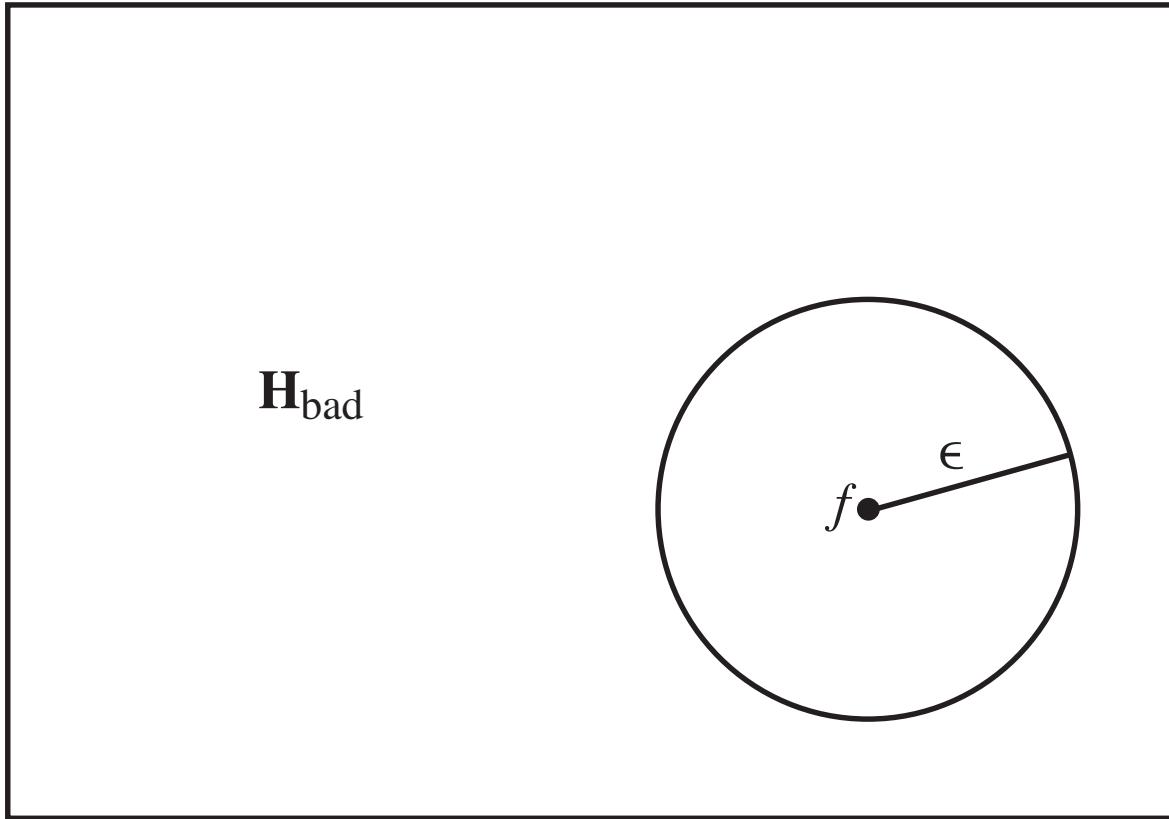
Performance – Ensemble Size



- Error reaches zero when M is 20; a weighted-majority combination of 20 decision stumps suffices to fit 100 ex.
- As more stumps are added to the ensemble, error remains at zero
- The test set performance continues to increase long after the training set error has reached zero

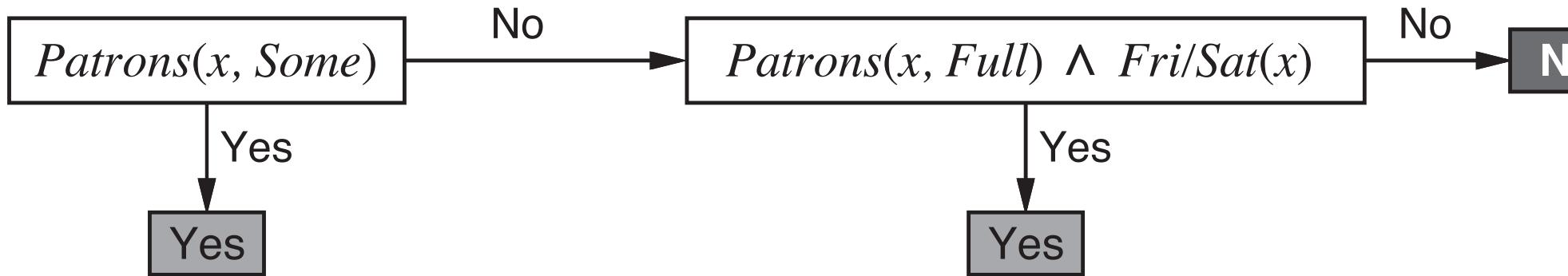
Why Learning Works

H



- $P(h_b \text{ agrees with the examples}) \leq (1 - \epsilon)^N$
- $P(\mathcal{H}_{bad} \text{ contains consistent hypothesis}) \leq |\mathcal{H}_{bad}|(1 - \epsilon)^N \leq |\mathcal{H}|(1 - \epsilon)^N$
- $|\mathcal{H}|(1 - \epsilon)^N \leq \delta \quad N \geq \frac{1}{\epsilon}(\ln \frac{1}{\delta} + \ln |\mathcal{H}|)$

Learning Decision Lists

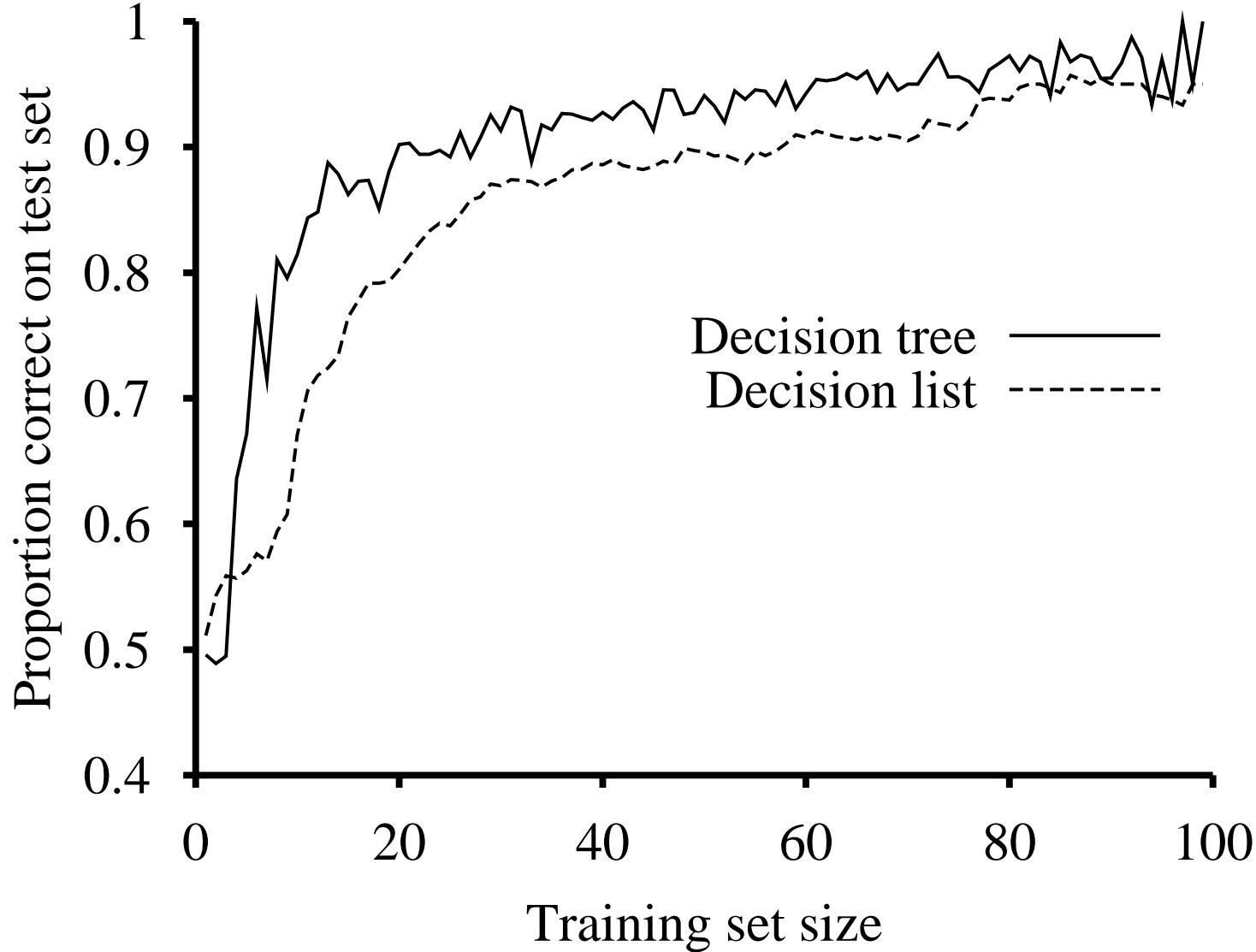


- $\forall x \text{ WillWait}(x) \Leftrightarrow \text{Patrons}(x, \text{Some}) \vee (\text{Patrons}(x, \text{Full}) \wedge \text{Fri/Sat}(x))$
- $|\text{k-DL}(n)| \leq 3^{|\text{Conj}(n, k)|} |\text{Conj}(n, k)|!$
- $|\text{Conj}(n, k)| = O(n^k)$
- $|\text{k-DL}(n)| = 2^{O(n^k \log_2(n^k))}$
- $N \geq \frac{1}{\epsilon} (\ln \frac{1}{\delta} + O(n^k \log_2(n^k)))$
- Any algorithm that returns a consistent decision list will PAC-learn a k-DL function in a reasonable number of examples, for small k.

Decision List Learning

```
function DECISION-LIST-LEARNING(examples) returns a decision list, No or failure
  if examples is empty then return the value No
  t ← a test that matches a nonempty subset  $\text{examples}_t$  of examples
    such that the members of  $\text{examples}_t$  are all positive or all negative
  if there is no such t then return failure
  if the examples in  $\text{examples}_t$  are positive then o ← Yes
  else o ← No
  return a decision list with initial test t and outcome o and remaining elements
  given by DECISION-LIST-LEARNING(examples –  $\text{examples}_t$ )
```

Learning Curve for DLL, DTL



- Selection of next test to add to the decision list: prefer small tests that match large sets of uniformly classified examples

Summary

- Supervised learning: when available feedback provides the correct value for the examples
- Learning a discrete-valued function: classification
- Learning a continuous function: regression
- Inductive learning involves finding a consistent hypothesis that agrees with the examples
- Ockham's razor suggests choosing the simplest consistent hypothesis
- Decision trees can represent all Boolean functions
- The information gain heuristic provides an efficient method for finding a simple, consistent decision tree
- Ensemble methods such as boosting often perform better than individual methods
- Computational learning theory analyzes the computational complexity of inductive learning

Exercise 18

Color	Barks	Size	Intel	Sheltie?
black	Y	S	H	P
yellow	N	L	M	N
yellow	N	L	L	N
white	Y	M	H	P
black	N	M	H	N
brown	Y	S	H	P
brown	Y	S	L	N
black	N	M	M	N

1. What score would the *information gain* formula assign to each feature? **Explain all your work.**
2. Which feature would be chosen as the root?
3. Show the remaining steps, if any, that the decision tree learning algorithm would perform. **Explain all your work.**

Exercise 18a

Show how AdaBoost works on the restaurant problem with M=3 and decision trees with just one test at the root.

	<i>A</i>	<i>B</i>	<i>F</i>	<i>H</i>	<i>Pat</i>	<i>Price</i>	<i>Rn</i>	<i>Rs</i>	<i>Type</i>	<i>Est</i>	<i>WW</i>
X_1	T	F	F	T	Some	\$\$\$	F	T	Fren	0–10	T
X_2	T	F	F	T	Full	\$	F	F	Thai	30–60	F
X_3	F	T	F	F	Some	\$	F	F	Burg	0–10	T
X_4	T	F	T	T	Full	\$	F	F	Thai	10–30	T
X_5	T	F	T	F	Full	\$\$\$	F	T	Fren	>60	F
X_6	F	T	F	T	Some	\$\$	T	T	Ital	0–10	T
X_7	F	T	F	F	None	\$	T	F	Burg	0–10	F
X_8	F	F	F	T	Some	\$\$	T	T	Thai	0–10	T
X_9	F	T	T	F	Full	\$	T	F	Burg	>60	F
X_{10}	T	T	T	T	Full	\$\$\$	F	T	Ital	10–30	F
X_{11}	F	F	F	F	None	\$	F	F	Thai	0–10	F
X_{12}	T	T	T	T	Full	\$	F	F	Burg	30–60	T

Knowledge in Learning

Russell & Norvig - AIMA2e

Ioan Alfred Letia

<http://cs-gw.utcluj.ro/~letia>

Outline

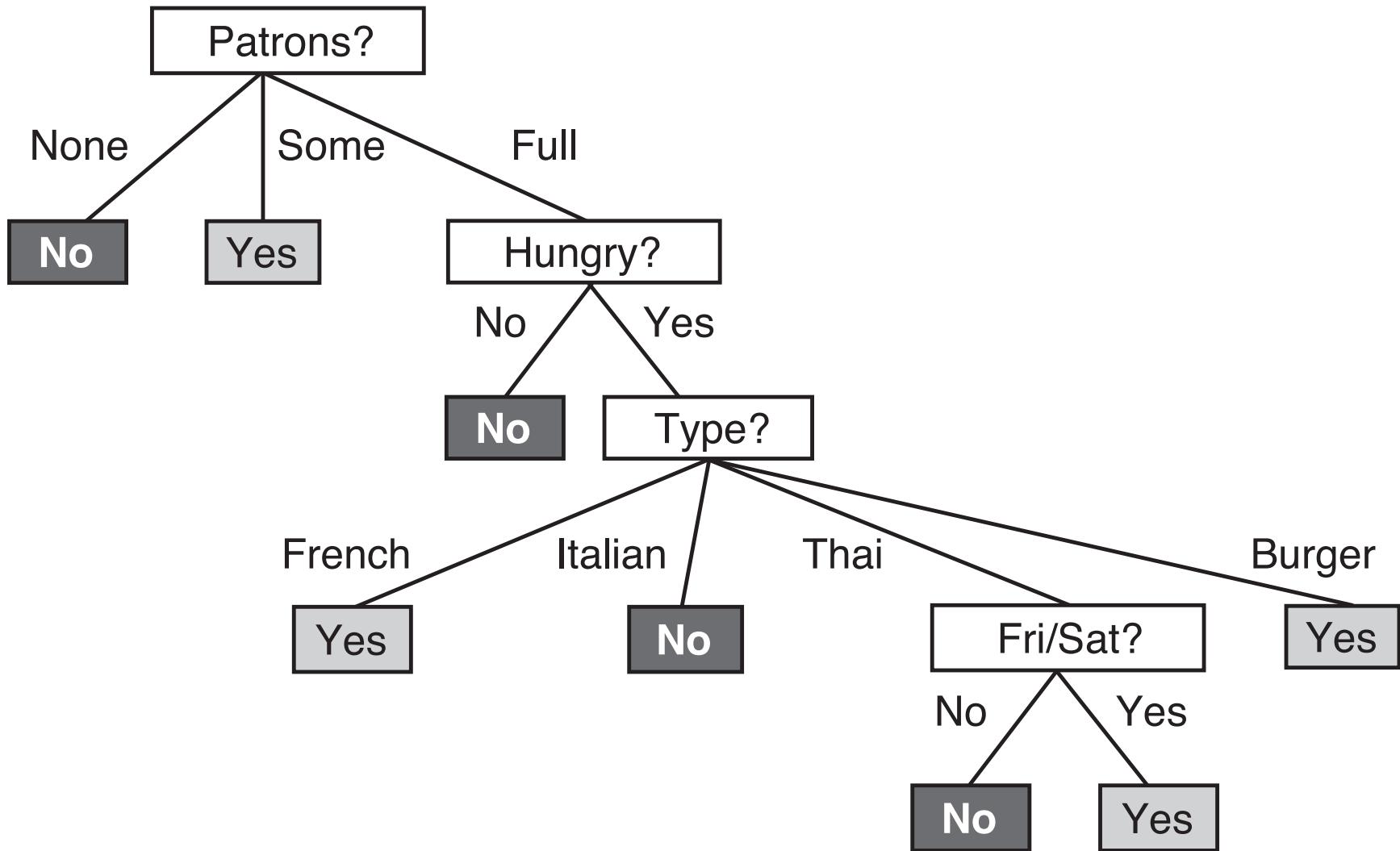
- A logical formulation of learning
- Knowledge in learning
- Explanation-based learning
- Learning using relevance information
- Inductive logic programming

Examples – Restaurant Domain

- Examples described by attribute values (Boolean, discrete, continuous, etc.)
E.g., situations where I will/won't wait for a table
- Classification of examples is positive (T) or negative (F)

example	Attributes										Target <i>WillWait</i>
	<i>Alt</i>	<i>Bar</i>	<i>Fri</i>	<i>Hun</i>	<i>Pat</i>	<i>Price</i>	<i>Rain</i>	<i>Res</i>	<i>Type</i>	<i>Est</i>	
X_1	T	F	F	T	Some	\$\$\$	F	T	French	0–10	T
X_2	T	F	F	T	Full	\$	F	F	Thai	30–60	F
X_3	F	T	F	F	Some	\$	F	F	Burger	0–10	T
X_4	T	F	T	T	Full	\$	F	F	Thai	10–30	T
X_5	T	F	T	F	Full	\$\$\$	F	T	French	>60	F
X_6	F	T	F	T	Some	\$\$	T	T	Italian	0–10	T
X_7	F	T	F	F	None	\$	T	F	Burger	0–10	F
X_8	F	F	F	T	Some	\$\$	T	T	Thai	0–10	T
X_9	F	T	T	F	Full	\$	T	F	Burger	>60	F
X_{10}	T	T	T	T	Full	\$\$\$	F	T	Italian	10–30	F
X_{11}	F	F	F	F	None	\$	F	F	Thai	0–10	F
X_{12}	T	T	T	T	Full	\$	F	F	Burger	30–60	T

Induced Restaurant Tree



Logical Formulation of Learning

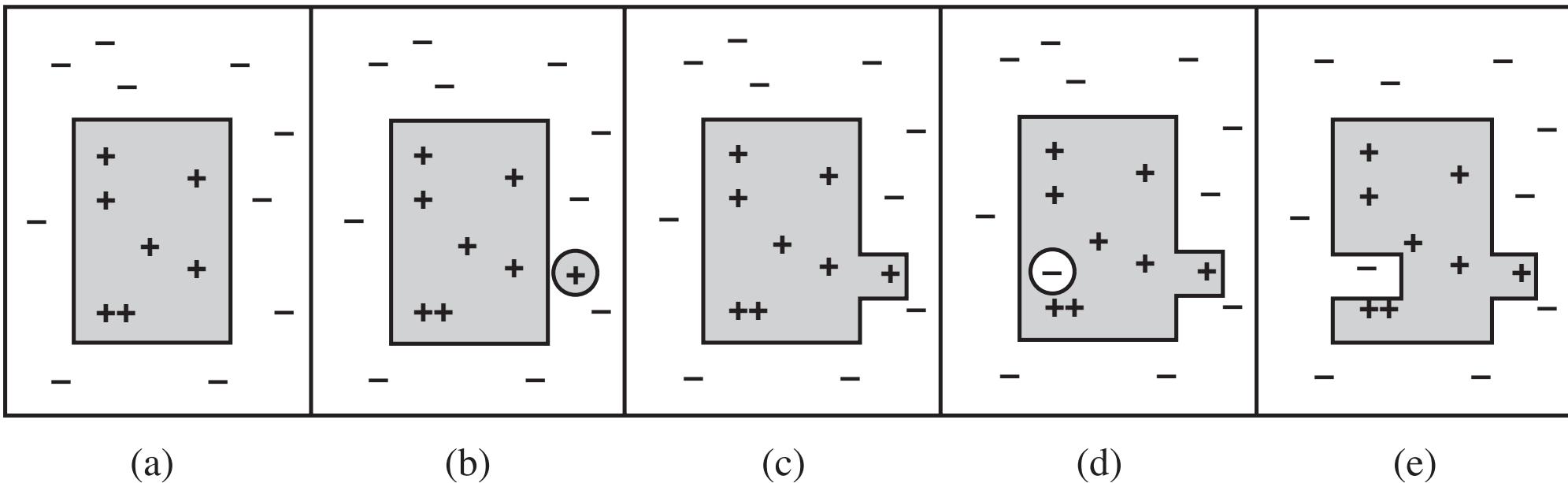
- $\text{Alternate}(X_1) \wedge \neg \text{Bar}(X_1) \wedge \neg \text{Fri/Sat}(X_1) \wedge \text{Hungry}(r)$
 $\wedge \text{Patrons}(X_1, \text{Some}) \wedge \dots$
- $\text{WillWait}(X_1)$
-

$$\forall r \text{ WillWait}(r) \Leftrightarrow \begin{aligned} & \text{Patrons}(r, \text{Some}) \\ & \vee \text{Patrons}(r, \text{Full}) \wedge \text{Hungry}(r) \wedge \text{Type}(r, \text{Fre}) \\ & \vee \text{Patrons}(r, \text{Full}) \wedge \text{Hungry}(r) \wedge \text{Type}(r, \text{Tai}) \\ & \quad \wedge \text{Fri/Sat}(r) \\ & \vee \text{Patrons}(r, \text{Full}) \wedge \text{Hungry}(r) \wedge \text{Type}(r, \text{Bur}) \end{aligned}$$

- Each hypothesis predicts that a certain set of examples, the **extension** of the predicate, will be examples of the goal predicate
- Hypothesis space $\mathbf{H} \{H_1, \dots, H_n\}$
- The learning algorithm believes the sentence
 $H_1 \vee H_2 \vee H_3 \dots \vee H_n$

Current Best Hypothesis Search

- Idea: maintain a single hypothesis, and adjust it as new examples arrive in order to maintain consistency
- (b) **false negative**: extension of the hypothesis must be increased to include the example \Rightarrow generalization
- (d) **false positive**: extension of the hypothesis must be decreased to exclude the example \Rightarrow specialization



Current Best Hypothesis

function CURRENT-BEST-LEARNING(*examples*) **returns** a hypothesis

$H \leftarrow$ any hypothesis consistent with the first example in *examples*

for each remaining example in *examples* **do**

if *e* is false positive for H **then**

$H \leftarrow$ **choose** a specialization of H consistent with *examples*

else if *e* is false negative for H **then**

$H \leftarrow$ **choose** a generalization of H consistent with *examples*

if no consistent specialization/generalization can be found **then fail**

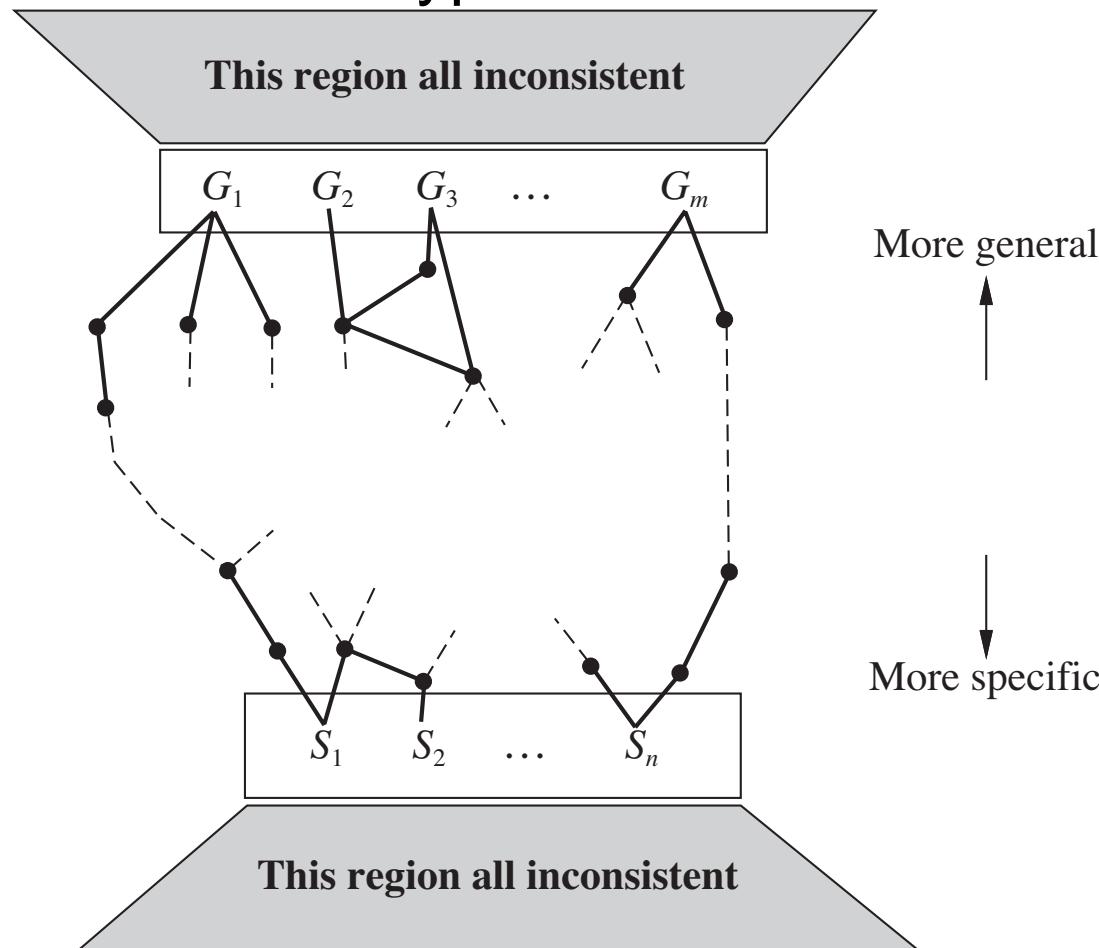
end

return H

- The algorithm is described nondeterministically, because at any point, several possible specializations or generalizations may be applicable
- The choices that are made will not necessarily lead to the simplest hypothesis, and may lead to an unrecoverable situation where no simple modification of the hypothesis is consistent with all of the data
- With a large number of instances and a large space, however, some difficulties arise

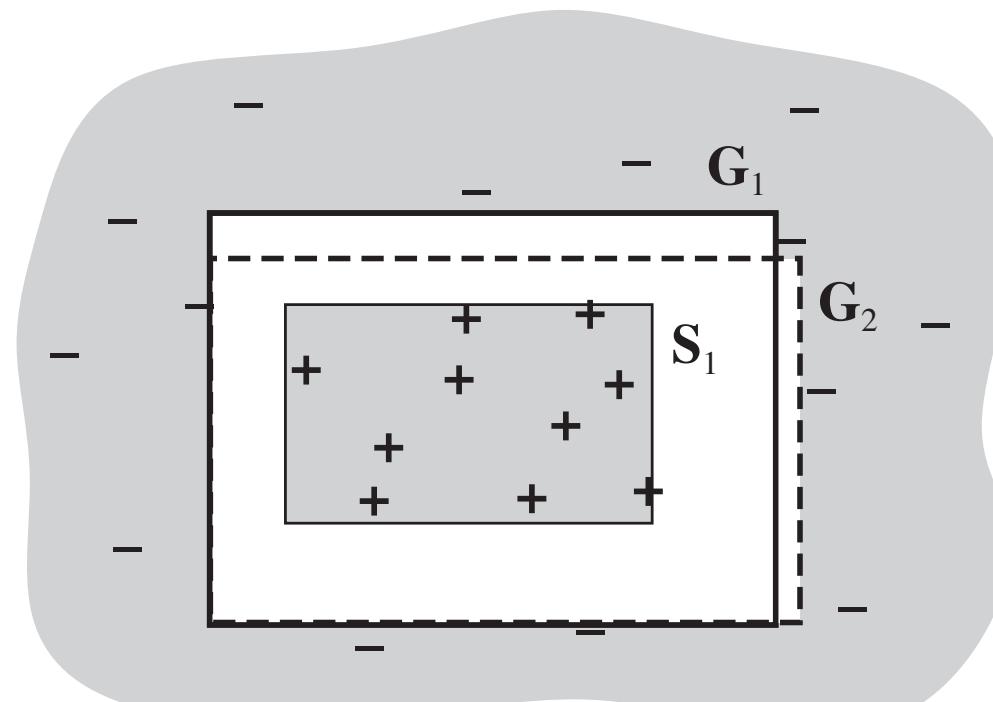
Version Space

- Every consistent hypothesis is more specific than some member of the G-set and more general than some member of the S-set
- Every hypothesis more specific than some member of the G-set and more general than some member of the S-set is a consistent hypothesis



Version Space Update

- False positive for S_i : S_i too general, no consistent specializations of $S_i \Rightarrow$ throw it out of the S-set
- False negative for S_i : S_i too specific \Rightarrow replace it by all its immediate generalizations
- False positive for S_i : S_i too general \Rightarrow replace it by all its immediate specializations
- False negative for S_i : S_i too specific, no consistent generalizations of $S_i \Rightarrow$ throw it out of the G-set



Version Space Learning

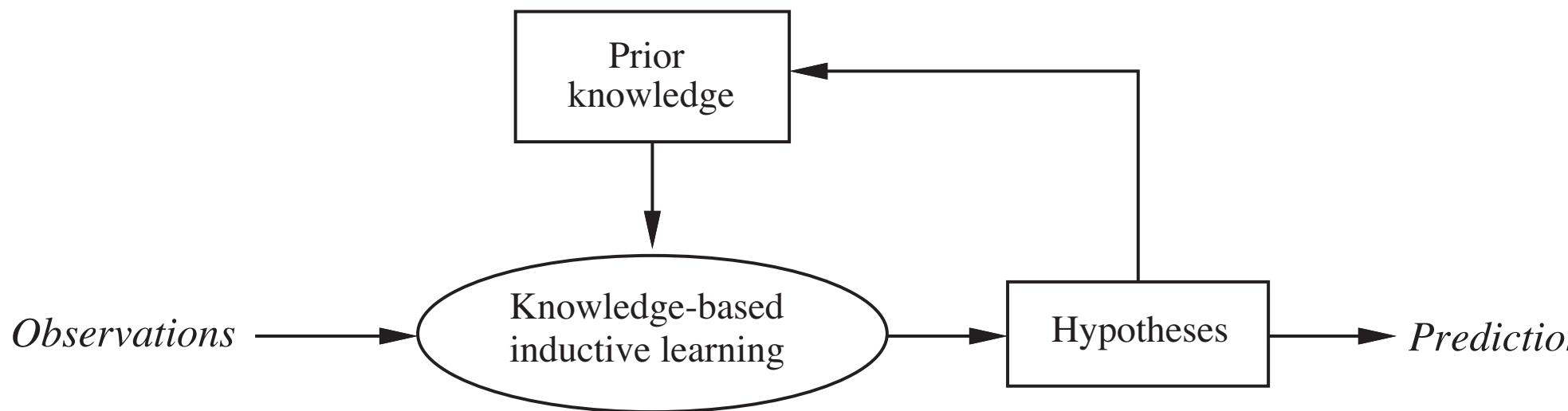
```
function VERSION-SPACE-LEARNING(examples) returns a version space
  local variables:  $V$ , the version space: the set of all hypotheses
     $V \leftarrow$  the set of all hypotheses
    for each example  $e$  in examples do
      if  $V$  is not empty then  $V \leftarrow$  VERSION-SPACE-UPDATE( $V, e$ )
    end
    return  $V$ 
```

```
function VERSION-SPACE-UPDATE( $V, e$ ) returns an updated version space
   $V \leftarrow \{h \in V : h \text{ is consistent with } e\}$ 
```

- If the domain contains noise or insufficient attributes for exact classification, the version space will collapse
- With unlimited disjunction in hypothesis space, the S-set always contains a single most-specific hypothesis, the disjunction of descriptions of positive examples seen

Knowledge in Learning

- A hypothesis that **explains the observations** must satisfy Hypothesis \wedge Descriptions \models Classifications entailment constraint
- The use of background knowledge allows much faster learning than one might expect from a pure induction program



Some General Schemes

- EBL explanation based learning

Hypothesis \wedge Descriptions \models Classifications

Background \models Hypothesis

- RBL relevance-based learning

Hypothesis \wedge Descriptions \models Classifications

Background \wedge Descriptions \wedge Classifications \models Hypothesis

- KBIL knowledge-based inductive learning

Background \wedge Hypothesis \wedge Descriptions \models Classifications

- ILP inductive logic programming

- the effective hypothesis space size is reduced
- the size of the hypothesis required to construct an explanation for the observations can be much reduced

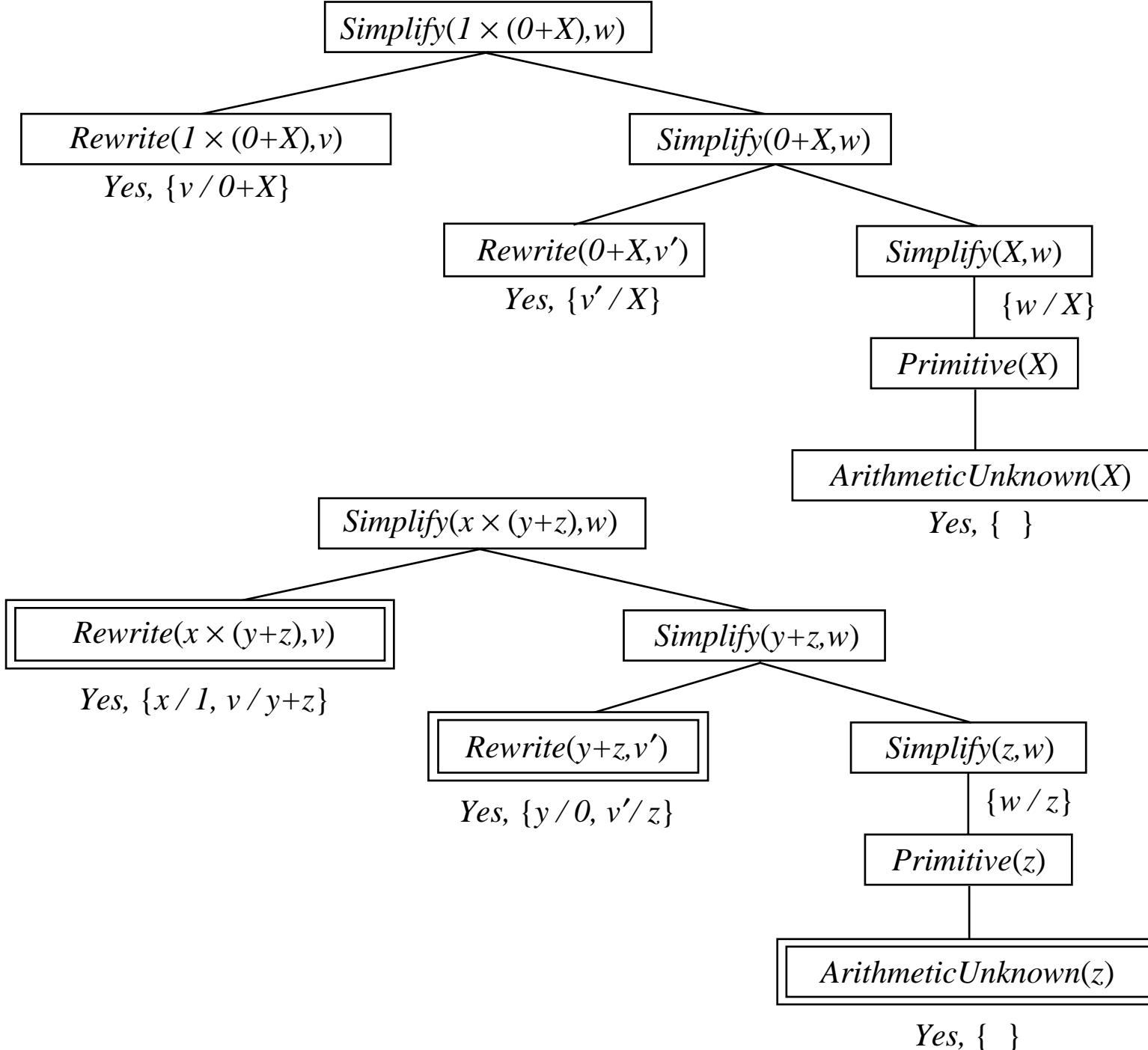
Explanation-Based Learning

- memoization speed up programs by saving the results
- $\text{ArithmetcUnknown}(u) \Rightarrow \text{Derivative}(u^2, u) = 2u$
- Alfred North Whitehead, co-author with Bertrand Russell of **Principia Mathematica**:
"Civilization advances by extending the number of important operations that we can do without thinking about them"
- $\text{Rewrite}(u, v) \wedge \text{Simplify}(v, w) \Rightarrow \text{Simplify}(u, w)$
 $\text{Primitive}(u) \Rightarrow \text{Simplify}(u, u)$
 $\text{ArithmetcUnknown}(u) \Rightarrow \text{Primitive}(u)$
 $\text{Number}(u) \Rightarrow \text{Primitive}(u)$
 $\text{Rewrite}(1 \times u, u)$
 $\text{Rewrite}(0 + u, u)$
...

General Rules from Examples

- $\text{Rewrite}(1 \times (0+z), 0+z) \wedge \text{Rewrite}(0+z, z) \wedge \text{ArithmeticUnknown}(z) \Rightarrow \text{Simplify}(1 \times (0+z), z)$
- $\text{ArithmeticUnknown}(z) \Rightarrow \text{Simplify}(1 \times (0+z), z)$
- Basic EBL process:
 - Given an example, construct a proof that the goal predicate applies to the example using the available background knowledge
 - In parallel, construct a generalized proof tree for the variabilized goal using the same inference step as in the original proof
 - Construct a new rule whose left-hand side consists of the leaves of the proof tree, and whose right-hand side is the variabilized goal (after applying the necessary bindings from the generalized proof)
 - Drop any conditions that are true regardless of the values of the variables in the goal

Proof Trees for Simplification



Improving Efficiency

- $\text{Primitive}(z) \Rightarrow \text{Simplify}(1 \times (0+z), z)$
- $\text{Simplify}(y+z, w) \Rightarrow \text{Simplify}(1 \times (y+z), w)$
- Analysis of efficiency gains from EBL:
 - Adding large numbers of rules to a knowledge base can slow down the reasoning process, because the inference mechanism must still check those rules even in cases where they do not yield a solution
 - The derived rules must offer significant increases in speed for the cases that they do cover
 - Derived rules should also be as general as possible, so that they apply to the largest possible set of cases
- By generalizing from past example problems, EBL makes the knowledge base more efficient for the kind of problems that it is reasonable to expect

RBL – Using Relevance Information

- $\forall x,y,n,l \text{ Nationality}(x,n) \wedge \text{Nationality}(y,n) \wedge \text{Language}(x,l)$
 $\Rightarrow \text{Language}(y,l)$
- $\text{Nationality}(\text{Fernando},\text{Brazil}) \wedge$
 $\text{Language}(\text{Fernando},\text{Portuguese})$
- $\forall x \text{ Nationality}(x,\text{Brazil}) \Rightarrow \text{Language}(x,\text{Portuguese})$
- functional dependencies determinations
- $\text{Nationality}(x,n) \succ \text{Language}(x,l)$
- $\text{Material}(x,m) \wedge \text{Temperature}(x,t) \succ \text{Conductance}(x,\rho)$
 $\text{Material}(x,m) \wedge \text{Temperature}(x,t) \succ \text{Density}(x,d)$

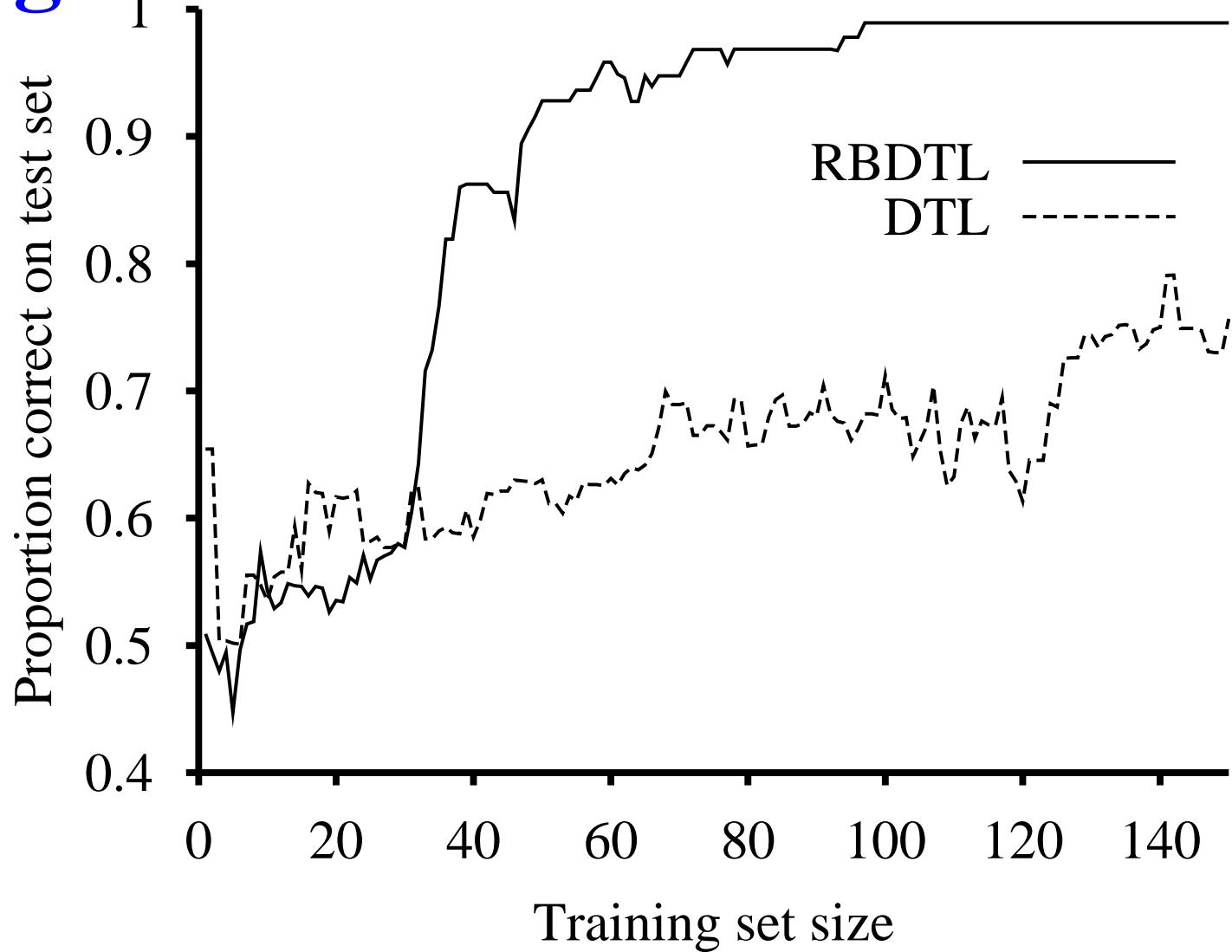
```
function MINIMAL-CONSISTENT-DET( $E, A$ ) returns a determination
  inputs:  $E$ , a set of examples
           $A$ , a set of attributes, of size  $n$ 

  for  $i \leftarrow 0, \dots, n$  do
    for each subset  $A_i$  of  $A$  of size  $i$  do
      if CONSISTENT-DET?( $A_i, E$ ) then return  $A_i$ 
    end
  end
```

```
function CONSISTENT-DET?( $A, E$ ) returns a truth-value
  inputs:  $A$ , a set of attributes
           $E$ , a set of examples
  local variables:  $H$ , a hash table

  for each example  $e$  in  $E$  do
    if some example in  $H$  has the same values as  $e$  for the attributes  $A$ 
      but a different classification then return False
    store the class of  $e$  in  $H$ , indexed by the values for attributes  $A$  of the example
  end
  return True
```

Using Relevance Information



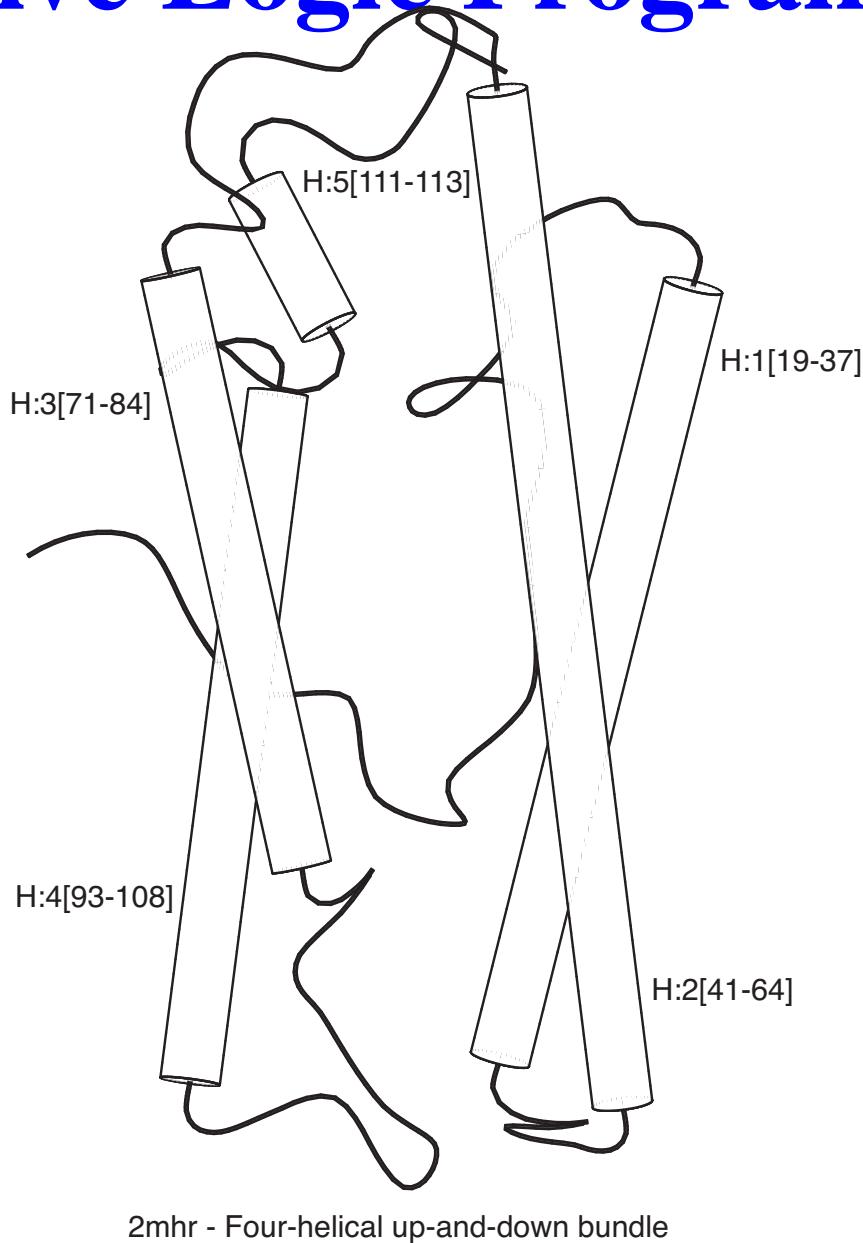
- Relevance-based decision-tree function $\text{RBDTL}(E, A, v)$ returns a decision tree $\text{DECISION-TREE-LEARNING}(E, \text{MINIMAL-CONSISTENT-DET}(E, A), v)$

Exercise 19

Illustrate how the algorithm MINIMAL-CONSISTENT-DET works on the conductance measurements below to obtain the minimal consistent determination.

Sample	Mass	Temperature	Material	Size	Conductance
S1	12	26	Copper	3	0.59
S2	12	100	Copper	3	0.57
S3	24	26	Copper	6	0.59
S4	12	26	Lead	2	0.05
S5	12	100	Lead	2	0.04
S6	24	26	Lead	4	0.05

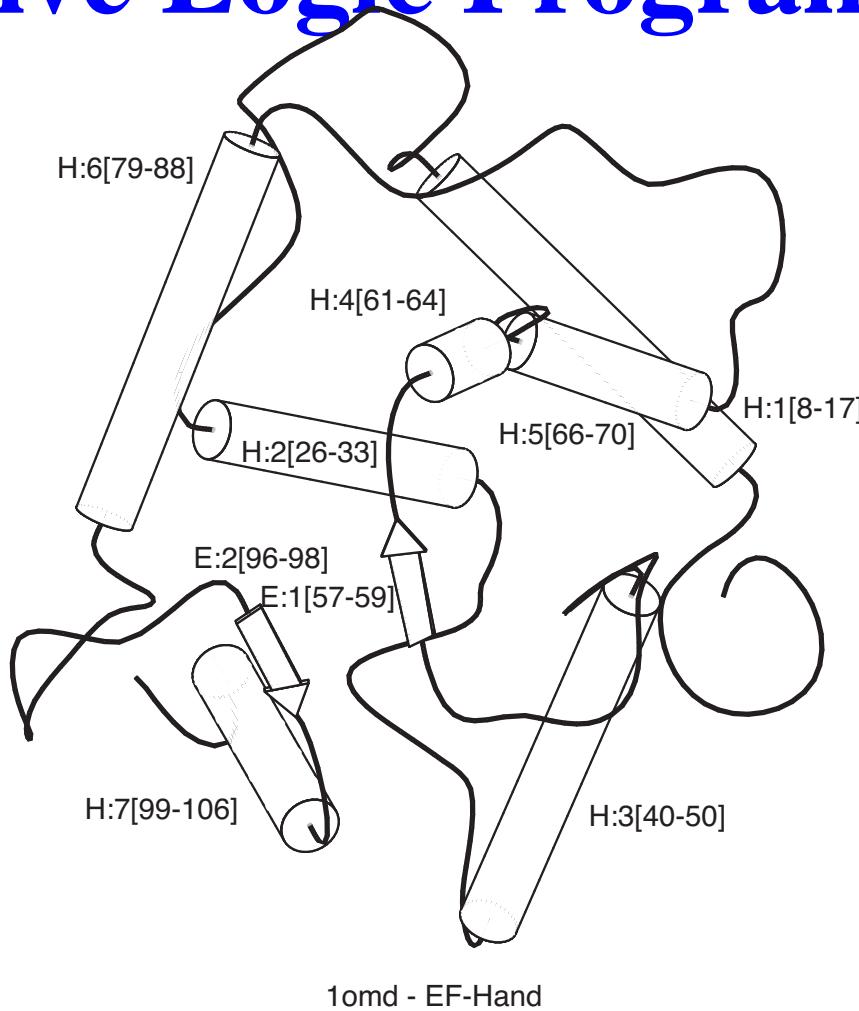
Inductive Logic Programming



2mhr - Four-helical up-and-down bundle

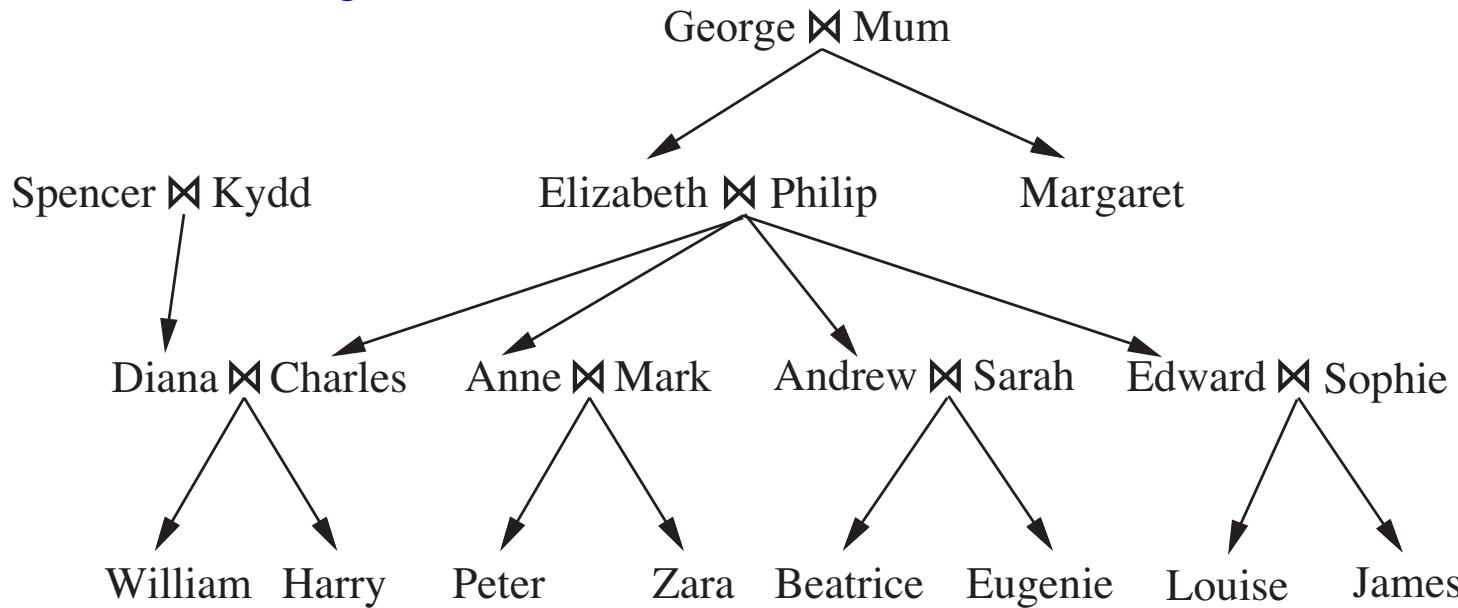
- positive examples of the "four-helical up-and-down bundle" concept in the domain of protein folding

Inductive Logic Programming



- negative examples of the "four-helical up-and-down bundle" concept in the domain of protein folding
- a rule that cannot be learned, or even represented, by an attribute-based mechanism such as we saw in previous chapters

Family



- Father(Philip, Charles) Father(Philip, Anne) ...
- Mother(Mum, Margaret) Mother(Mum, Elizabeth) ...
- Married(Diana, Charles) Married(Elixabeth, Philip) ...
- Male (Philip) Male(Charles) ...
- Female (Beatrice) Female (Margaret) ...

ILP Example

- Grandparent(Mum,Charles)
Grandparent(Elizabeth,Beatrice) ...
 \neg Grandparent(Mum,Harry)
 \neg Grandparent(Spencer,Peter)
- $\text{Grandparent}(x,y) \Leftrightarrow [\exists z \text{ Mother}(x,z) \wedge \text{Mother}(z,y)]$
 $\vee [\exists z \text{ Mother}(x,z) \wedge \text{Father}(z,y)]$
 $\vee [\exists z \text{ Father}(x,z) \wedge \text{Mother}(z,y)]$
 $\vee [\exists z \text{ Father}(x,z) \wedge \text{Father}(z,y)]$
- Grandparent(<Mum,Charles>) ...
- FirstElementIsMotherOfElizabeth(<Mum,Charles>)
- Attribute-based learning algorithms are incapable of learning relational predicates
- Parent(x,y) $\Leftrightarrow [\text{Mother}(x,y) \vee \text{Father}(x,y)]$
- $\text{Grandparent}(x,y) \Leftrightarrow [\exists z \text{ Parent}(x,z) \wedge \text{Parent}(z,y)]$

Top-Down Learning Methods

- FOIL (Quinlan, 1990)
- Trying to learn the definition of $\text{Grandfather}(x,y)$
- Positive examples:
 $\langle \text{George}, \text{Anne} \rangle, \langle \text{Philip}, \text{Peter} \rangle, \langle \text{Spencer}, \text{Harry} \rangle, \dots$
- Negative examples:
 $\langle \text{George}, \text{Elizabeth} \rangle, \langle \text{Harry}, \text{Zara} \rangle, \langle \text{Charles}, \text{Philip} \rangle, \dots$
- $\Rightarrow \text{Grandfather}(x,y)$
- Potential additions:
 $\text{Father}(x,y) \Rightarrow \text{Grandfather}(x,y)$
 $\text{Parent}(x,z) \Rightarrow \text{Grandfather}(x,y)$
 $\text{Father}(x,z) \Rightarrow \text{Grandfather}(x,y)$
- $\text{Father}(x,z) \wedge \text{Parent}(z,y) \Rightarrow \text{Grandfather}(x,y)$

FOIL

- **NEW-LITERALS** takes a clause and constructs all possible "useful" literals that could be added to the clause
- Example: $\text{Father}(x,z) \Rightarrow \text{Grandfather}(x,y)$
- Kinds of literals that can be added:
 - Literals using predicates: the literal can be negated or unnegated, any existing predicate can be used, and the arguments must all be variables
 - Equality and inequality literals: these relate variables already appearing in the clause
 - Arithmetic comparisons: when dealing with functions of continuous variables, literals such as $x > y$ and $y \leq z$ can be added
- **CHOOSE-LITERAL** uses a heuristic somewhat similar to information gain to decide which literal to add

FOIL

function FOIL(*examples, target*) **returns** a set of Horn clauses

inputs: *examples*, set of examples

target, a literal for the goal predicate

local variables: *clauses*, set of clauses, initially empty

while *examples* contains positive examples **do**

clause \leftarrow NEW-CLAUSE(*examples, target*)

 remove examples covered by *clause* from *examples*

 add *clause* to *clauses*

return *clauses*

function NEW-CLAUSE(*examples, target*) **returns** a Horn clause

local variables: *clause*, a clause with *target* as head and an empty body

l, a literal to be added to the clause

extended-examples, a set of examples with values for new variables

extended-examples \leftarrow *examples*

while *extended-examples* contains negative examples **do**

l \leftarrow CHOOSE-LITERAL(NEW-LITERALS(*clause*), *extended-examples*)

 append *l* to the body of *clause*

extended-examples \leftarrow set of examples created by applying EXTEND-EXAMPLES to *clause* and *extended-examples*

add *clause* to *clauses*

return *clauses*

FOIL

function NEW-CLAUSE(*examples*, *target*) **returns** a Horn clause

local variables: *clause*, a clause with *target* as head and an empty body

l, a literal to be added to the clause

extended-examples, a set of examples with values for new variables

extended-examples \leftarrow *examples*

while *extended-examples* contains negative examples **do**

l \leftarrow CHOOSE-LITERAL(NEW-LITERALS(*clause*), *extended-examples*)

append *l* to the body of *clause*

extended-examples \leftarrow set of examples created by applying EXTEND-EXAMPLE
to each example in *extended-examples*

return *clause*

function EXTEND-EXAMPLE(*example*, *literal*) **returns**

if *example* satisfies *literal*

then return the set of examples created by extending *example* with
each possible constant value for each new variable in *literal*

else return the empty set

Inverse Resolution

$\neg \text{Parent}(x,z) \vee \neg \text{Parent}(z,y) \vee \text{Grandparent}(x,y)$

$\text{Parent}(\text{George}, \text{Elizabeth})$

$\{x/\text{George}, z/\text{Elizabeth}\}$

$\neg \text{Parent}(\text{Elizabeth},y) \vee \text{Grandparent}(\text{George},y)$

$\text{Parent}(\text{Elizabeth}, \text{Anne})$

$\{y/\text{Anne}\}$

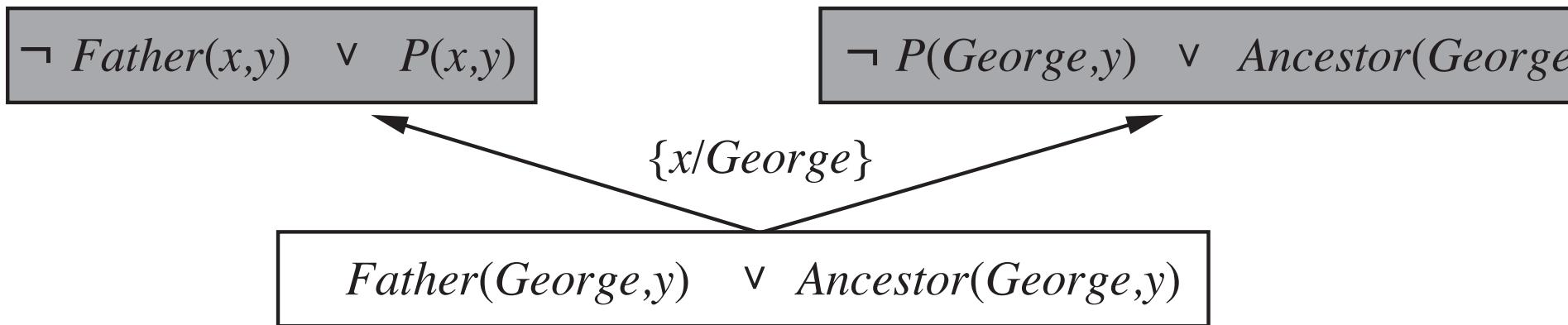
$\text{Grandparent}(\text{George}, \text{Anne})$

$\neg \text{Grandparent}(\text{George}, \text{Anne})$



- An inverse resolution step takes a resolvent C and produces two clauses C_1 and C_2 , such that C is the result of resolving C_1 and C_2 ; or takes C and C_1 and produces a suitable C_2
- Each inverse resolution step is nondeterministic, because for any C and C_1 , there can be several or even an infinite number of clauses C_2 that satisfy the requirement that when resolved with C_1 it generates C

Making Discoveries



- An inverse resolution procedure that inverts a complete resolution strategy is, in principle, a complete algorithm for learning first-order theories
- If some unknown **hypothesis** generates a set of examples, then an inverse resolution procedure can generate **hypothesis** from the examples
- In several applications, ILP techniques have outperformed knowledge-free methods

Summary

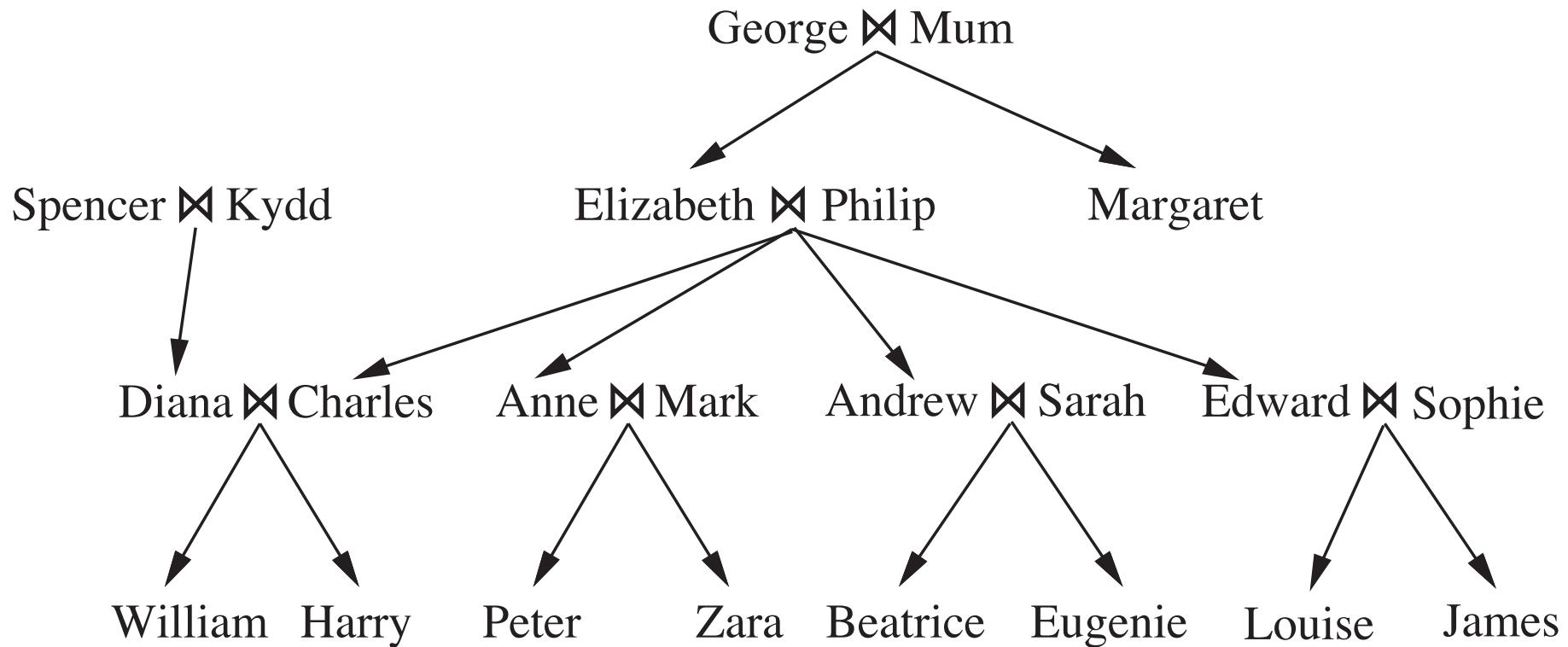
- Use of prior knowledge leads to cumulative learning
- Understanding logical roles expressed by entailment constraints helps define learning techniques
- Explanation-based learning extracts general rules by explaining the examples and generalizing the explanation
- Relevance-based learning uses prior knowledge in the form of determinations to identify relevant attributes
- Knowledge-based inductive learning finds inductive hypotheses that explain sets of observations with the help of background knowledge
- Inductive logic programming techniques perform KBIL on knowledge that is expressed in first-order logic

Exercise 19.4

Fill in the missing values for the clauses C_1 or C_2 (or both) in the following sets of clauses, given that C is the resolvent of C_1 and C_2 .

- $C = \text{True} \Rightarrow P(A,B)$, $C_1 = P(x,y) \Rightarrow Q(x,y)$, $C_2 = ??$
- $C = \text{True} \Rightarrow P(A,B)$, $C_1 = ??$, $C_2 = ??$
- $C = P(x,y) \Rightarrow P(x,f(y))$, $C_1 = ??$, $C_2 = ??$

Exercise 19.7



Using the data from the family tree, or a subset thereof, apply the FOIL algorithm to learn a definition for the **Ancestor** predicate.

Dynamic Protocols for Open Agent Systems

Alexander Artikis^{1,2}

¹Institute of Informatics & Telecommunications, NCSR “Demokritos”, Athens, 15310, Greece

²Electrical & Electronic Engineering Department, Imperial College London, SW7 2BT, UK

a.artikis@acm.org

ABSTRACT

Multi-agent systems where the members are developed by parties with competing interests, and where there is no access to a member's internal state, are often classified as 'open'. The specification of protocols for open agent systems of this sort is largely seen as a design-time activity. Moreover, there is no support for run-time specification modification. Due to environmental, social, or other conditions, however, it is often required to revise the specification during the protocol execution. To address this requirement, we present an infrastructure for 'dynamic' protocol specifications, that is, specifications that may be modified at run-time by agents. The infrastructure consists of well-defined procedures for proposing a modification of the 'rules of the game' as well as decision-making over and enactment of proposed modifications. We evaluate proposals for rule modification by modelling dynamic specifications as metric spaces. Furthermore, we constrain the enactment of proposals that do not meet the evaluation criteria. We illustrate our infrastructure by presenting a dynamic specification of a resource-sharing protocol, and an execution of this protocol in which the participating agents modify the protocol specification.

Categories and Subject Descriptors

I.2.11 [Artificial Intelligence]: Distributed Artificial Intelligence

General Terms

Design

Keywords

organised adaptation, norm, action language

1. INTRODUCTION

A particular kind of Multi-Agent System (MAS) is one where the member agents are developed by different parties, and where there is no access to an agent's internal state. In this kind of MAS it cannot be assumed that all agents will behave according to the system specification because the agents act on behalf of parties with competing interests, and thus may inadvertently fail to, or even deliberately choose not to, conform to the system specification in order to

Cite as: Multi-Agent Programming Languages, Alexander Artikis, *Proc. of 8th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2009)*, Decker, Sichman, Sierra and Castelfranchi (eds.), May, 10–15, 2009, Budapest, Hungary, pp. 97–104

Copyright © 2009, International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org), All rights reserved.

achieve their individual goals. Two examples of this type of MAS are Virtual Organisations and electronic marketplaces. MAS of this type are often classified as 'open'.

Open MAS can be viewed as instances of *normative systems* [11]. A feature of this type of system is that actuality, what is the case, and ideality, what ought to be the case, do not necessarily coincide. Therefore, it is essential to specify what is permitted, prohibited, and obligatory, and perhaps other more complex normative relations that may exist between the agents. Among these relations, considerable emphasis has been placed on the representation of *institutional power* [12] — a standard feature of any normative system whereby designated agents, when acting in specified roles, are empowered by an institution to create specific relations or states of affairs (such as when an agent is empowered by an institution to award a contract and thereby create a bundle of normative relations between the contracting parties).

Several approaches have been proposed in the literature for the specification of protocols for open MAS. The majority of these approaches offer 'static' specifications, that is, there is no support for run-time specification modification. In some open MAS, however, environmental, social or other conditions may favour, or even require, specifications modifiable during the protocol execution. Consider, for instance, the case of a malfunction of a large number of sensors in a sensor network, or the case of manipulation of a voting procedure due to strategic voting, or when an organisation conducts its business in an inefficient manner. Therefore, we present in this paper an infrastructure for 'dynamic' protocol specifications, that is, specifications that are developed at design-time but may be modified at run-time by agents. The presented infrastructure is an extension of the framework for static specifications of [3], and is motivated by 'dynamic argument systems' [5] — argument systems in which, at any point in the disputation, agents may start a meta level debate, that is, the rules of order become the current point of discussion, with the intention of altering these rules.

Our infrastructure for dynamic specifications allows agents to alter the rules of a protocol P during the protocol execution. P is considered an 'object' protocol; at any point in time during the execution of the object protocol the participants may start a 'meta' protocol in order to decide whether the object protocol rules should be modified. Moreover, the participants of the meta protocol may initiate a meta-meta protocol to decide whether to modify the rules of the meta protocol, or they may initiate a meta-meta-meta protocol to modify the rules of the meta-meta protocol, and so on.

We employ a resource-sharing protocol to illustrate our

Table 1: Main Predicates of the Event Calculus.

Predicate	Meaning
<code>happens(Act, T)</code>	Action <i>Act</i> occurs at time <i>T</i>
<code>initially(F = V)</code>	The value of fluent <i>F</i> is <i>V</i> at time 0
<code>holdsAt(F = V, T)</code>	The value of fluent <i>F</i> is <i>V</i> at time <i>T</i>
<code>initiates(Act, F = V, T)</code>	The occurrence of action <i>Act</i> at time <i>T</i> initiates a period of time for which the value of fluent <i>F</i> is <i>V</i>
<code>terminates(Act, F = V, T)</code>	The occurrence of action <i>Act</i> at time <i>T</i> terminates a period of time for which the value of fluent <i>F</i> is <i>V</i>

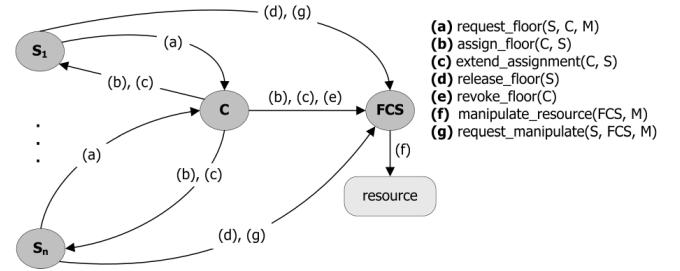
infrastructure for dynamic specifications: the object protocol concerns resource-sharing while the meta protocols are voting protocols. In other words, at any time during a resource-sharing procedure the agents may vote to change the rules that govern the management of resources. The resource-sharing protocol was chosen for the sake of providing a concrete example. In general, the object protocol may be any protocol for open MAS, such as a protocol for coordination or e-commerce; similarly a meta protocol can be any procedure for decision-making over rule modification (argumentation, negotiation, and so on).

The remainder of this paper is organised as follows. First, we briefly review the Event Calculus, the action language that we employ to formalise protocol specifications. Second, we review a static specification of a resource-sharing protocol. Third, we present a dynamic specification of the resource-sharing protocol and an infrastructure for modifying the protocol specification at run-time. Fourth, we present an execution of the protocol, demonstrating how the agents may alter the protocol specification. Finally, we compare our work to research on dynamic specifications, and discuss further research.

2. THE EVENT CALCULUS

The Event Calculus (EC), introduced by Kowalski and Sergot [14], is a formalism for representing and reasoning about actions or events and their effects in a logic programming framework. In this section we briefly describe the version of the EC that we employ. EC is based on a many-sorted first-order predicate calculus. For the version used here, the underlying time model is linear and it may include real numbers or integers. Where *F* is a *fluent* (a property that is allowed to have different values at different points in time), the term *F = V* denotes that fluent *F* has value *V*. Boolean fluents are a special case in which the possible values are *true* and *false*. Informally, *F = V* holds at a particular time-point if *F = V* has been *initiated* by an action at some earlier time-point, and not *terminated* by another action in the meantime.

An *action description* in EC includes axioms that define, among other things, the action occurrences (with the use of `happens` predicates), the effects of actions (with the use of `initiates` and `terminates` predicates), and the values of the fluents (with the use of `initially` and `holdsAt` predicates). Table 1 summarises the main EC predicates. Variables (starting with an upper-case letter) are assumed to be universally


Figure 1: A chaired floor control protocol.

quantified unless otherwise indicated. Predicates, function symbols and constants start with a lower-case letter.

The following sections present a logic programming implementation of an EC action description expressing an infrastructure for a dynamic resource-sharing protocol.

3. A RESOURCE-SHARING PROTOCOL

We present a specification of a resource-sharing or *floor control* protocol in the style of [3]. In the field of Computer-Supported Co-operative Work the term *floor control* denotes a service guaranteeing that at any given moment only a designated set of users (subjects) may simultaneously work on the same objects (shared resources), thus, creating a temporary exclusivity for access on such resources. We present a ‘chair-designated’ Floor Control Protocol (cFCP), that is, a distinguished participant is the arbiter over the usage of a specific resource. For simplicity we assume a single resource.

The roles of cFCP are summarised below:

- *Floor Control Server (FCS)*, the role of the only participant physically manipulating the shared resource.
- *Subject (S)*, the role of designated participants requesting the floor from the chair, releasing the floor, and requesting from the FCS to manipulate the resource.
- *Chair (C)*, the role of the participant assigning the floor for a particular time period to a subject, extending the time allocated for the floor, and revoking the floor from the subject holding it.

Figure 1 provides an informal description of the possible interactions between the agents occupying the roles of cFCP. More details about these interactions will be given presently.

It has been argued [12] that the specifications of protocols for open MAS should explicitly represent the concept of institutional power, that is, the characteristic feature of organisations/institutions — legal, formal, or informal — whereby designated agents, often when acting in specific roles, are empowered, by the institution, to create or modify facts of special significance in that institution — *institutional facts* — usually by performing a specified kind of act. Searle [17], for example, has distinguished between *brute facts* and institutional facts. Being in physical possession of an object is an example of a brute fact (it can be observed); being the owner of that object is an institutional fact. The cFCP specification explicitly represents the concept of institutional power; moreover, there is a distinction between institutional power, physical capability, permission and obligation. Table 2 presents the conditions in which a cFCP participant has the physical capability, institutional power, permission and obligation to perform an action (to save space Table 2 displays only three protocol actions).

According to the cFCP specification all protocol actions

Table 2: cFCP Specification.

Action	Capability	Power	Permission	Obligation
<i>request_floor(S, C, M)</i>	\top	$f_requested(S) = \text{false}$	\top	\perp
<i>assign_floor(C, S)</i>	\top	$status = \text{free},$ $best_candidate = S$	$status = \text{free},$ $best_candidate = S$	$status = \text{free},$ $best_candidate = S$
<i>revoke_floor(C)</i>	\top	$status = \text{granted}(S, Tg)$ $CurrentTime \geq Tg,$ $best_candidate \neq S$	$status = \text{granted}(S, Tg),$ $CurrentTime \geq Tg,$ $best_candidate \neq S$	$status = \text{granted}(S, Tg),$ $CurrentTime \geq Tg,$ $best_candidate = S', S \neq S'$

are physically possible at any time. In other examples the specification of physical capability could have been different.

In this example, a subject S is empowered to request the floor from the chair C when S has no pending requests:

$$\begin{aligned} \text{holdsAt}(\text{pow}(S, \text{request_floor}(S, C, M)) = \text{true}, T) \leftarrow \\ \text{holdsAt}(\text{role_of}(S) = \text{subject}, T), \\ \text{holdsAt}(\text{role_of}(C) = \text{chair}, T), \\ \text{holdsAt}(f_requested(S) = \text{false}, T) \end{aligned} \quad (1)$$

The variable M in request_floor represents the requested type of resource manipulation. The pow fluent expresses institutional power, the $\text{role_of}(Ag)$ fluent expresses the roles an agent Ag occupies, while the $f_requested(Ag)$ fluent is true if Ag has pending requests for the floor. To avoid clutter, in Table 2 we do not display the role_of fluents and assume that S denotes an agent occupying the role of subject and C denotes an agent occupying the role of chair.

Having specified the institutional power to request the floor, it is now possible to define the effects of this action: a request for the floor is eligible to be serviced if and only if it is issued by an agent with the institutional power to request the floor. Requests for the floor issued by agents without the necessary institutional power are ignored. Due to space limitations, we do not present here the EC axioms expressing the effects of protocol actions (we show only one such axiom below).

We chose to specify that a subject is always permitted to exercise its power to request the floor. Moreover, a subject S is permitted to request the floor even if S is not empowered to do so (see Table 2). In the latter case a request for the floor will be ignored by the chair (since S was not empowered to request the floor) but S will not be *sanctioned* since it was not forbidden to issue the request. In general, an agent is sanctioned when performing a forbidden action or not complying with an obligation. To save space, we do not present a specification of sanctions for cFCP. Finally, a subject is never obliged to request the floor.

The chair's power to assign the floor is defined as follows:

$$\begin{aligned} \text{holdsAt}(\text{pow}(C, \text{assign_floor}(C, S)) = \text{true}, T) \leftarrow \\ \text{holdsAt}(\text{role_of}(C) = \text{chair}, T), \\ \text{holdsAt}(status = \text{free}, T), \\ \text{holdsAt}(best_candidate = S, T) \end{aligned} \quad (2)$$

The chair C is empowered to assign the floor to S if the floor is free, and S is the best candidate for the floor. The $status$ fluent expresses the status of the floor; it can be either *free* or $\text{granted}(S, Tg)$, that is, granted to some subject S until time Tg . The $best_candidate$ fluent denotes the best candidate for the floor. The definition of this fluent is application-specific. For instance, the best candidate could

be the one with the earliest request, that with the most ‘urgent’ request (however ‘urgent’ may be defined), and so on. There is no difficulty in expressing such definitions in the formalism employed here. Indeed, the availability of the full power of logic programming is one of the main attractions of employing EC as the temporal formalism.

The effects of assigning the floor are expressed as follows:

$$\begin{aligned} \text{initiates}(\text{Act}, \text{status} = \text{granted}(S, T+60), T) \leftarrow \\ \text{Act} = \text{assign_floor}(C, S), \\ \text{holdsAt}(\text{pow}(C, \text{Act}) = \text{true}, T) \end{aligned} \quad (3)$$

The result of exercising the power to assign the floor to S at time T is that the floor becomes granted to S until $T+60$. In other versions of the specification (as we shall see later) the floor could be allocated for a longer/shorter period.

The conditions in which the chair is permitted and obliged to assign the floor are the same as the conditions in which the chair is empowered to assign the floor (see Table 2).

Similarly we specify the power, permission and obligation to perform the remaining protocol actions, and the effects of these actions. For instance, the chair's power to revoke the floor is defined as follows:

$$\begin{aligned} \text{holdsAt}(\text{pow}(C, \text{revoke_floor}(C)) = \text{true}, T) \leftarrow \\ \text{holdsAt}(\text{role_of}(C) = \text{chair}, T), \\ \text{holdsAt}(status = \text{granted}(S, Tg), T), T \geq Tg, \\ \text{not holdsAt}(best_candidate = S, T) \end{aligned} \quad (4)$$

The chair C is empowered to revoke the floor if the floor is granted to a subject S until time Tg , the current time is greater or equal to Tg , and S is not the best candidate for the floor. ‘not’ represents negation by failure.

The specification of the power, permission or obligation to revoke the floor, assign the floor, or perform some other protocol action, should include a deadline stating the time by which the action of revoking the floor, assigning the floor, etc, should be performed. There is no particular difficulty in including deadlines in the formalisation but it lengthens the presentation and is omitted here for simplicity. Example formalisations of deadlines may be found in [3].

4. A DYNAMIC RESOURCE-SHARING PROTOCOL

Being motivated by Brewka [5], we present an infrastructure that allows agents to modify (a subset of) the rules of a protocol at run-time. Regarding our running example, we consider the resource-sharing protocol as an ‘object’ protocol; at any point in time during the execution of the object protocol the participants may start a ‘meta’ protocol in order to potentially modify the object protocol rules — for instance, replace an existing rule-set with a new one. The

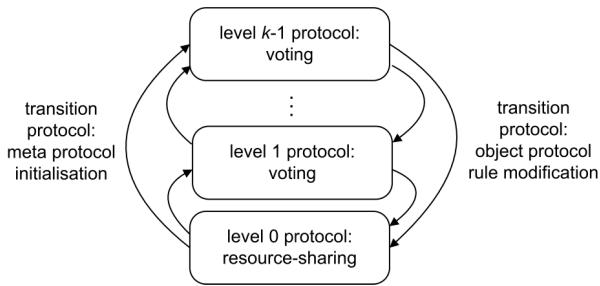


Figure 2: A k -level Infrastructure for Dynamic Specifications.

meta protocol may be any protocol for decision-making over rule modification. For the sake of presenting a concrete example, we chose a voting procedure as a meta protocol, that is, the meta protocol participants take a vote on a proposed modification of the object protocol rules. The participants of the meta protocol may initiate a meta-meta protocol to modify the rules of the meta protocol, or they may initiate a meta-meta-meta protocol to modify the rules of the meta-meta protocol, and so on. For simplicity, in this example *all* meta protocols are voting procedures (in other systems each meta protocol may be a different decision-making procedure). In general, in a k -level infrastructure, level 0 corresponds to the main (resource-sharing, in this example) protocol while a protocol of level n , $0 < n \leq k-1$ (voting, in this example), is created, by the protocol participants of a level m , $0 \leq m < n$, in order to decide whether to modify the protocol rules of level $n-1$. The infrastructure for dynamic (resource-sharing) specifications is displayed in Figure 2.

Apart from object and meta protocols, the infrastructure for dynamic specifications includes ‘transition’ protocols — see Figure 2 — that is, procedures that express, among other things, the conditions in which an agent may successfully initiate a meta protocol (for instance, only the members of the board of directors may successfully initiate a meta protocol in some organisations), the roles that each meta protocol participant will occupy, and the ways in which an object protocol is modified as a result of the meta protocol interactions. The components of the infrastructure for dynamic specifications, level 0 protocol, level n protocol ($n > 0$), and transition protocol, are discussed in the following sections.

4.1 Level 0

For illustration purposes we chose a resource-sharing protocol — the specification of which was presented in Section 3 — as a level 0 protocol. A protocol specification consists of the core rules that are always part of the specification, and the *Degrees of Freedom (DoF)*, that is, the specification components that may be modified at run-time. Consider, for instance, the definition of the best candidate for the floor as a DoF: the participants of the resource-sharing protocol may decide, at run-time, to change the way the best candidate is computed. To provide a simple example, consider the following rules:

$$\begin{aligned} \text{holdsAt}(\text{best_candidate} = S, T) \leftarrow \\ \text{holdsAt}(\text{active}(bc) = \text{fcfs}, T), \\ \text{holdsAt}(\text{requests} = \text{List}, T), \\ \text{first}(\text{List}, (S, Tr, M)) \end{aligned} \quad (5)$$

$$\begin{aligned} \text{holdsAt}(\text{best_candidate} = S, T) \leftarrow \\ \text{holdsAt}(\text{active}(bc) = \text{random}, T), \\ \text{holdsAt}(\text{requests} = \text{List}, T) \\ \text{random}(\text{List}, (S, Tr, M)) \end{aligned} \quad (6)$$

The above two rules provide simple, alternative specifications of what constitutes the best candidate for the floor, that is, they are two possible ‘values’ of the best candidate DoF. According to rule (5) the best candidate is the subject with the earliest pending request for the floor, while according to rule (6) the best candidate is chosen randomly from the list of subjects that have pending requests. The *requests* fluent provides a list of the pending requests for the floor, each request represented as a triple $(\text{subject}, \text{request-time}, \text{manipulation-type})$, while *first* and *random* are suitably chosen atemporal predicates returning, respectively, the first and a random element of a list of requests (this list is sorted by request time). The *active(DoF)* fluent denotes the current or ‘active’ value of a DoF. For instance, $\text{active}(bc) = \text{fcfs}$ states that the best candidate (bc) is determined on a first-come, first-served (*fcfs*) basis whereas $\text{active}(bc) = \text{random}$ states that the best candidate is chosen randomly. Rules (5) and (6) are replaceable in the sense that the participants of the resource-sharing protocol may activate/deactivate one of them at run-time (that is, change the value of the corresponding *active* fluent) by means of a meta protocol.

The resource-sharing protocol may have other DoF, such as the specification of the permission and the obligation to perform an action. The classification of a specification component as a DoF is application-specific.

4.2 Level n

To provide a concrete example we chose a three-level infrastructure for dynamic specifications. Moreover, both levels 1 and 2 are voting procedures, such as that presented in [16]. Due to space limitations we do not present here a specification of a voting procedure. Briefly, we assume a simple procedure including a set of voters casting their votes, ‘for’ or ‘against’ a particular motion, which would be in this example a proposed modification of the rules of level $n-1$, and a chair counting the votes and declaring the motion carried or not carried, based on the *standing rules* of the voting procedure — simple majority, two-thirds majority, etc.

Our infrastructure allows for the modification of the rules of all protocol levels apart from the top one. Consequently, we define DoF for all protocol levels apart from the top one. For the protocol of level 1 — voting — we chose to set as a DoF the standing rules of the voting procedure. In other words, a level 2 protocol may be initiated in order to decide whether level 1 voting should become, say, simple majority instead of two-thirds majority. Note that the voting procedures of levels 1 and 2 may not always have the same set of rules. For example, at a particular time-point level 2 voting may require a simple majority whereas level 1 voting may require a two-thirds majority (as mentioned above, the standing rules of level 1 constitute a DoF and thus the specification of this part of the protocol may change at run-time).

Each protocol level, including level 0, has its own state; for instance, at a particular time-point agent Ag_1 may have the institutional power to vote in level 1 but have no powers in levels 0 and 2, Ag_2 may occupy the role of subject in level 0 and the role of voter in level 1 (role-assignment in a meta protocol will be discussed presently), etc. In order to distinguish between the states of different protocol levels, we

add a parameter in the representation of actions and fluents of all protocol specifications, expressing the protocol level PL , as follows: $role_of(Ag, PL)$ expresses the set of roles Ag occupies in level PL , $active(DoF, PL)$ expresses the value of DoF in level PL , etc.

4.3 Transition Protocol

In order to modify the protocol rules of level m , $m \geq 0$ (for example, to change the value of the best candidate DoF from *fefs* to *random*), that is, in order to start a protocol of level $m+1$, the participants of level m need to follow a ‘transition’ protocol — see Figure 2. The infrastructure for dynamic specifications presented in this paper requires two types of transition protocol: one leading from the resource-sharing protocol to a voting one (level 0 to level 1 or 2), and one leading from one voting protocol to the other (level 1 to level 2). We will only present the first type of transition protocol; the latter type of protocol may be specified in a similar manner. An example transition protocol leading from resource-sharing to voting is the following: a subject of the resource-sharing protocol proposes a modification of the rules of this protocol (or of the rules of level 1). If the subject is empowered to make such a proposal, then the modification is directly accepted, without the execution of a meta protocol, provided that another subject seconds the proposal, and no other subject objects to that proposal. If the proposal is not seconded then it is ignored. If the proposal is seconded and there is an objection then an argumentation procedure commences, the topic of which is the proposed rule modification, the proponent of the topic is the subject that made the proposal, and the opponent is the subject that objected to the proposal. The argumentation procedure is followed by a meta protocol (level 1 or 2) in which a vote is taken on the proposed rule modification.

In this example transition protocol we have specified the power to propose a rule-set replacement as follows:

$$\begin{aligned} \text{holdsAt}(\text{pow}(Ag, \text{propose}(Ag, P)) = \text{true}, T) \leftarrow \\ P = \text{replace}(DoF, OldVal, NewVal, PL), \\ \text{holdsAt}(\text{role_of}(Ag, 0) = \text{subject}, T), \\ \text{holdsAt}(\text{protocol}(PL+1) = \text{idle}, T), \\ \text{holdsAt}(\text{active}(DoF, PL) = OldVal, T) \end{aligned} \quad (7)$$

An agent Ag is empowered to propose to change the value of DoF in protocol level PL from $OldVal$ to $NewVal$ if:

- Ag occupies the role of subject in level 0. In this example, the chair is not empowered to propose a modification of the protocol rules.
- There is no protocol taking place in level $PL+1$. A protocol for modifying the rules of level PL , that is, a protocol of level $PL+1$, may commence only if there is no other protocol of level $PL+1$ taking place.
- The value of DoF in level PL is $OldVal$.

A proposal for rule modification needs to be seconded by a subject having the institutional power to second the proposal, in order to be directly enacted, or initiate an argumentation procedure, that is followed by a voting procedure. We chose to specify the power to second a proposal for rule modification as follows:

$$\begin{aligned} \text{holdsAt}(\text{pow}(Ag, \text{second}(Ag, P)) = \text{true}, T) \leftarrow \\ \text{holdsAt}(\text{role_of}(Ag, 0) = \text{subject}, T), \\ \text{holdsAt}(\text{proposal}(Ag', P) = \text{true}, T), Ag \neq Ag' \end{aligned} \quad (8)$$

Ag is empowered to second any proposal for rule modification P made by some other subject. The proposal fluent

records proposals made by empowered agents.

In other examples the specification of the power to propose a rule modification, or second such a proposal, could have different, or additional conditions further constraining how a protocol specification may change at run-time. For instance, in some systems an agent would not have the power to propose a rule modification, or second a proposal, that would create some type of protocol inconsistency. In Section 4.4 we present a way of constraining the process of run-time specification modification. Other ways of achieving that, such as the one described above, could have been formalised (see Section 6).

We have specified that any subject is empowered to object to a proposal for rule modification. Exercising the power to object to a proposal for rule modification initiates an argumentation procedure, the topic of which is the proposed modification. To save space we do present here a specification of an argumentation procedure; see [2] for an example formalisation of such a procedure.

The completion of the argumentation taking place in the context of a transition protocol initiates a meta protocol. The latter protocol is a voting procedure concerning a proposed rule modification. The agents participating in this procedure and the roles they occupy are determined by the transition protocol that results in the voting procedure. We chose to specify that the chair of the resource-sharing protocol becomes the chair of the voting procedure. Furthermore, the agents occupying the role of voter, thus having the power to vote, are the subjects that have not been sanctioned for exhibiting ‘anti-social’ behaviour, that is, performing forbidden actions or not complying with obligations:

$$\begin{aligned} \text{holdsAt}(\text{role_of}(Ag, PL) = \text{voter}, T) \leftarrow \\ \text{holdsAt}(\text{role_of}(Ag, 0) = \text{subject}, T), \\ \text{holdsAt}(\text{protocol}(PL) = \text{executing}, T), PL > 0, \\ \text{holdsAt}(\text{sanctions}(Ag) = \text{false}, T) \end{aligned} \quad (9)$$

The value of $\text{protocol}(PL)$ becomes *executing*, in the case where $PL > 0$, at the end of the argumentation of the transition protocol that leads to level PL . We chose not to relativise the fluent recording sanctions to a protocol level. Therefore, the *sanctions* fluent records ‘anti-social’ behaviour exhibited at any protocol level. Depending on the employed treatment of sanctions, ‘anti-social’ behaviour may be permanently recorded, thus depriving a subject of participating in a meta level, or it may be temporarily recorded, thus enabling subjects to participate in a meta level after a specified period has elapsed from the performance of forbidden actions or non-compliance with obligations.

The fact that an agent may successfully start a protocol of level $m+1$ by proposing a modification of the protocol rules of level m , does not necessarily imply that the rules of level m will be modified. It is only if the motion of level $m+1$, that is, the proposed rule modification in level m , is carried that the rules of level m will be modified. Consider the following rule expressing the outcome of a voting procedure of level $m+1$ that takes place in order to change the value of DoF in level m from $OldVal$ to $NewVal$:

$$\begin{aligned} \text{initiates}(Act, \text{active}(DoF, PL) = NewVal, T) \leftarrow \\ Act = \text{declare}(C, \text{Motion}, \text{carried}, PL+1), \\ \text{Motion} = \text{replace}(DoF, OldVal, NewVal, PL), \\ \text{holdsAt}(\text{pow}(C, Act) = \text{true}, T) \end{aligned} \quad (10)$$

Exercising the power to declare the motion carried in level $PL+1$ results in setting the value of DoF in level PL to $NewVal$, if the motion was the replacement of $OldVal$ with $NewVal$. If the chair of the voting procedure did not declare the motion carried, or was not empowered to make the declaration, then the value of DoF would not have changed in level PL . To save space we do not present here the specification of the power to make a declaration.

4.4 Constraining the Process of Run-time Specification Modification

As already mentioned, all protocol levels apart from the top one have DoF and, therefore, their specification may be modified at run-time. In this section we present a way of evaluating a proposal for protocol modification. Moreover, we present a way of constraining the enactment of proposals that do not meet the evaluation criteria.

A protocol specification with l DoF creates an l -dimensional specification space where each dimension corresponds to a DoF . A point in the l -dimensional specification space, or *specification point*, represents a complete protocol specification, and is denoted by an l -tuple where each element of the tuple expresses a value of a DoF . Consider, for example, the resource sharing protocol with three DoF : the specification of the best candidate, the power to request the floor, and the floor allocation time. The specification of these three protocol features may change at run-time — for instance, the best candidate may be determined randomly, on a first-come, first-served basis, priority may be given to subjects requesting a particular manipulation type, or to subjects that have not been sanctioned. The specification of the power to request the floor may dictate that a subject is empowered to request the floor if it has no pending requests, or it may be that a subject is always empowered to request the floor. Finally, in this example, the floor may be allocated for 60, 120 or 180 time-points. In the resource-sharing example with these three DoF , a specification point s is, for instance:

$$s = (fcfs, \text{anytime}, 120)$$

According to the above specification point, the best candidate is determined on first-come, first-served basis (*fcfs*), a subject is empowered to request the floor at any time (*anytime*), and the floor is allocated for 120 time-points.

We evaluate a proposal for protocol modification by modelling a dynamic protocol specification as a *metric space* [6]. More precisely, given the set of specification points of a protocol, we define a ‘desired’ specification point, and compute the ‘distance’ between the desired point and the specification point that would be reached if the proposal for protocol modification was accepted (‘resulting’ point). We constrain the process of run-time protocol modification by forbidding agents to propose a modification in a way that the resulting specification point is ‘far’ from the desired point. In what follows we describe when a specification point is considered ‘desired’, the way we compute the distance between two specification points, and the way we forbid agents to propose protocol modifications.

A desired specification point is a tuple of the desired DoF values. A desired specification point for the resource-sharing protocol could be one where the best candidate is determined on a first-come, first-served basis, the power to request the floor depends on whether a subject has pending requests, and the floor allocation time is 60 time-points.

We follow two steps to compute the distance between two specification points s and s' , each represented as an l -tuple, where each element of the tuple expresses a DoF value. First, we transform s and s' to l -tuples of non-negative integers qs and qs' respectively. To achieve that we define an application-specific function v , that ‘ranks’ each DoF value, that is, associates every DoF value with a non-negative integer. The i -th element of qs , qs_i , has the value of $v(s_i)$, where s_i is the i -th element of s (respectively, the i -th element of qs' , qs'_i , has the value of $v(s'_i)$, where s'_i is the i -th element of s'). Second, we employ a *metric* (or *distance function*), such as the Manhattan metric, to compute the distance between qs and qs' (the choice of a metric is application-specific — see [6] for a list of metrics). Depending on the employed metric, we may add weights on the DoF — for instance, we may require that the computation of the distance between qs and qs' is primarily based on the best candidate DoF rather than the other two DoF . The distance between s and s' is the distance between qs and qs' .

We constrain run-time protocol modification as follows:

$$\begin{aligned} \text{holdsAt}(\text{per}(Ag, \text{propose}(Ag, P)) = \text{true}, T) \leftarrow \\ \text{holdsAt}(\text{pow}(Ag, \text{propose}(Ag, P)) = \text{true}, T), \\ P = \text{replace}(DoF, OldVal, NewVal, PL), \\ \text{holdsAt}(\text{dofValuesExcept}(DoF, PL) = List, T), \\ \text{holdsAt}(\text{distance}(List \cup [(DoF, NewVal)], PL) = Dist, T), \\ \text{holdsAt}(\text{threshold}(PL) = TDist, T), \\ Dist < TDist \end{aligned} \tag{11}$$

The $\text{per}(Ag, Act)$ fluent expresses whether Ag is permitted to perform Act . The $\text{dofValuesExcept}(DoF, PL)$ fluent returns a list of the degrees of freedom of level PL , except DoF , and their current values. The distance fluent calculates the distance between a given specification point (the first argument of the fluent) and the desired specification point in level PL . distance includes a logic programming implementation of a chosen metric. The threshold fluent returns the maximum distance that a specification point should have from the desired one in level PL . Different protocol levels may have different threshold values, different metrics and different desired specification points. According to axiom (11), an agent is permitted to propose a rule modification if it is empowered to propose a rule modification, and the distance between the resulting specification point and the desired one is less than a specified threshold. In this example, an agent is forbidden to propose a rule modification if these conditions are not satisfied.

Similarly we may express the permission to second a proposal for rule modification or the obligation to object to such a proposal. To further constrain the process of run-time specification modification, when the current specification point is too ‘far’ from the desired one (say twice the distance given by threshold), we could impose the obligation to propose a rule modification in a way that the resulting specification point is ‘close’ to the desired one. Due to space limitations we do not present here such an obligation.

What constitutes a ‘desired’ specification point often depends on environmental, social or other conditions. Consequently, we need to allow for the possibility that a desired specification point may change during a protocol execution. Similarly, we need to allow for the possibility that the metric with which the distance between two specification points is computed, and the threshold distance, may change at run-time. Changing the desired specification point, metric and

Table 3: Narrative of Events.

Time	Action
0	<i>request_floor(s₁, c, cpu, 0)</i>
5	<i>request_floor(s₂, c, cpu, 0)</i>
6	<i>request_floor(s₃, c, cpu, 0)</i>
8	<i>request_floor(s₅, c, cpu, 0)</i>
14	<i>propose(s₃, replace(sr, 3/4m, sm, 1))</i>
16	<i>object(s₁, replace(sr, 3/4m, sm, 1))</i>
17	<i>second(s₅, replace(sr, 3/4m, sm, 1))</i>
	<transition protocol argumentation>
28	<i>vote([s₂, s₃, s₄, s₅, s₆], for, 2)</i>
30	<i>vote(s₁, against, 2)</i>
31	<i>declare(c, carried, 2)</i>
35	<i>propose(s₅, replace(bc, fcfs, random, 0))</i>
36	<i>second(s₃, replace(bc, fcfs, random, 0))</i>
39	<i>object(s₂, replace(bc, fcfs, random, 0))</i>
	<transition protocol argumentation>
51	<i>vote([s₃, s₄, s₆], for, 1)</i>
53	<i>vote([s₁, s₂], against, 1)</i>
54	<i>declare(c, carried, 1)</i>
58	<i>assign_floor(c, s₃, 0)</i>

threshold distance, may be achieved in a manner similar to run-time rule modification.

5. ANIMATION

We illustrate our infrastructure for dynamic specifications by animating an example run of the dynamic resource-sharing protocol. A part of the narrative of events of this run is displayed in Table 3. The events of transition protocols (*propose*, *second*, *object*), level 1 and level 2 protocols (*vote* and *declare*) are indented. The last argument of a level 0, 1 and 2 event indicates the protocol level in which the event took place. Recall that the resource-sharing protocol (level 0) includes three DoF (see Section 4.4). Initially, the best candidate is determined on a first-come, first-served basis (see rule (5)), the power to request the floor depends on whether a subject has pending requests, and the floor allocation time is 60 time-points. Level 1 voting has a single DoF: the standing rules (*sr*). Initially, a 75% majority ($3/4m$) is required. In this example run, the initial specification points of levels 0 and 1 happen to coincide with the initial desired specification points of these levels. Level 2 voting does not have a DoF; moreover, a 75% majority is required. The Euclidean metric is used in levels 0 and 1.

Given that our infrastructure is expressed as a logic program, we may query our implementation to determine the state current at each time and protocol level. We may calculate, for instance, which roles each agent occupies, what powers, permissions, obligations each agent has, etc.

The present example includes 7 agents, a chair *c* and 6 subjects *s₁–s₆*. In the beginning of the run-time activities *s₁*, *s₂*, *s₃* and *s₅* exercise their power to request the floor, all of them requiring CPU cycles (see third argument of *request_floor*). *s₃* and *s₅* aim to change the best candidate DoF from *fcfs* to *random* because in this way they may acquire access to the resource faster. Before attempting to change the value of the best candidate DoF, *s₃* and *s₅* attempt to change the standing rules of level 1 voting from 75% majority to simple majority (*sm*), so that less votes would be required in level 1 when they propose to change the value of best candidate DoF. The proposal for chang-

ing the standing rules of level 1 takes place at time-point 14. At that time *s₃* is empowered to make the proposal (see rule (7)), and permitted to exercise its power (see rule (11)) because the distance, in level 1, between the resulting specification point and the desired one is less than the specified threshold. Effectively the specified threshold distance allows agents to change level 1 standing rules to simple majority. *s₃*'s proposal is followed by an objection, a secondment, and an argumentation. Then level 2 voting commences; the motion is the proposal for modifying the standing rules of level 1. *s₂–s₆* vote for the motion while *s₁* votes against it. At time-point 31 the motion of level 2 is declared carried (recall that level 2 requires 75% majority) and thus the standing rules of level 1 change to simple majority (see rule (10)). To avoid clutter in Table 3 we omit the motion in *declare*.

s₅ proposes at time-point 35 to change value of the best candidate DoF to *random*. *s₅* is empowered to make the proposal at that time. However, *s₅* is not permitted to exercise its power because the distance, in level 0, between the resulting specification point and the desired one is greater than the specified threshold. Consequently, *s₅* is sanctioned for performing a forbidden action and thus cannot participate in level 1 to vote (see rule (9)). *s₃*, *s₄* and *s₆* vote for the motion of level 1 — the proposal for changing the best candidate DoF of level 0 — while *s₁* and *s₂* vote against it. At time 54 the motion of level 1 is declared carried (recall that level 1 now requires a simple majority) and thus the best candidate in level 0 is chosen randomly from the list of subjects having pending requests (see rule (6)). At time 58 *c* exercises its power (see rule (2)) to assign the floor to *s₃*.

6. DISCUSSION

We presented a significant extension of [3, 1]: we developed an infrastructure for dynamic protocol specifications for open MAS, that is, specifications that are developed at design-time but may be modified at run-time by agents. Moreover, we modelled dynamic specifications as metric spaces in order to evaluate proposals for specification modification and constrain the enactment of proposals that do not meet the evaluation criteria. Any protocol for open MAS may be in level 0 of our infrastructure whereas any protocol for decision-making over specification modification may be in level *n*, *n*>0. The transition protocols linking the different protocol levels can be as complex/simple as required by the application in question.

The use of metric spaces for modelling dynamic specifications was proposed in [13]. In this line of work a protocol is evaluated by computing the distance between the specification point current at each time and the desired point. Our work differs in two ways from [13]. First, we developed an infrastructure (meta protocols, transition protocols) allowing for run-time specification point change. Second, we constrained the process of run-time specification point change by requiring that the specification point current at each time is ‘close’ to the desired one.

Bou and colleagues [4] have presented a mechanism for the run-time modification of the norms of an ‘electronic institution’. These researchers have proposed a ‘normative transition function’ that maps a set of norms (and goals) into a new set of norms. The ‘institutional agents’, representing the institution, are observing the members’ interactions in order to learn the normative transition function, so that they (the institutional agents) will directly enact the norms en-

abling the achievement of the ‘institutional goals’ in a given scenario. Unlike Bou and colleagues, we do not necessarily rely on designated agents to modify norms. We presented an infrastructure with which any agent may (attempt to) adapt a protocol specification. This does not exclude the possibility, however, that, in some applications, specific agents are given the institutional power to directly modify a protocol specification.

Chopra and Singh [7] have presented a way of adapting protocols according to context, or the preferences of agents in a given context. They formalise protocols and ‘transformers’, that is, additions/enhancements to an existing protocol specification that handle some aspect of context or preference. Depending on the context or preference, a protocol specification is complemented, at *design-time*, by the appropriate transformer thus resulting in a new specification. Unlike Chopra and Singh, we are concerned here with the *run-time* adaptation of a protocol specification and, therefore, we developed an infrastructure to achieve that.

Chopra and Singh, and Bou and colleagues, express protocols in terms of ‘commitments’ (here the term ‘commitment’ refers to a form of directed obligation). It is difficult to see how an interaction protocol for open MAS can be specified simply in terms of commitments. At the very least, a specification of institutional power is also required.

The Organisational Model for Adaptive Computational Systems (OMACS) [8] is another approach for dynamic MAS. OMACS concentrates mainly on re-organisation from the perspective of a functional assignment — the assignment of goals and roles to agents — whereas we emphasise, like Bou and colleagues, a different perspective, aimed at achieving a mapping from norms to norms, that is, the rules which regulate, among other things, the process of performing such a functional assignment. In general, we have been concerned with a particular aspect of ‘adaptation’: the run-time modification of the ‘rules of the game’ of normative systems. Clearly there are other aspects of adaptive/dynamic systems such as the run-time alteration of the (trading and other) relationships between agents, the members of a system, the assignment of roles to agents (as done in OMACS, for example), and the goals of a system. [15, 10, 19] and the papers in [9] are but a few examples of studies of adaptive systems.

Our approach for evaluating proposals for specification modification, that is, modelling dynamic specifications as metric spaces, is computationally efficient — computing the distance between two specification points may be performed in real-time even when there is a large number of DoF — and flexible — agents may change at run-time the desired specification point or the metric. There are several applications, however, where additional constraints are required on the process of run-time specification modification. For example, it may be required that a protocol specification always satisfies a set of properties (‘norm consistency’, for instance). In this case agents may not be empowered, or permitted, to propose a modification resulting in a specification that does not satisfy the desirable properties. We are currently investigating formalisms (such as that presented in [18]), supported by software implementations allowing for proving properties and assimilating narratives of events in an efficient manner.

7. REFERENCES

- [1] A. Artikis, D. Kaponis, and J. Pitt. Dynamic specifications of norm-governed systems. In V. Dignum, editor, *Multi-Agent Systems: Semantics and Dynamics of Organizational Models*. IGI, 2009.
- [2] A. Artikis, M. Sergot, and J. Pitt. An executable specification of a formal argumentation protocol. *Artificial Intelligence*, 171(10–15):776–804, 2007.
- [3] A. Artikis, M. Sergot, and J. Pitt. Specifying norm-governed computational societies. *ACM Transactions on Computational Logic*, 10(1), 2009.
- [4] E. Bou, M. López-Sánchez, and J. Rodriguez-Aguilar. Using case-based reasoning in autonomic electronic institutions. In *Proceedings of COIN Workshop*, LNCS 4870, pages 125–138. Springer, 2008.
- [5] G. Brewka. Dynamic argument systems: a formal model of argumentation processes based on situation calculus. *Journal of Logic and Computation*, 11(2):257–282, 2001.
- [6] V. Bryant. *Metric Spaces*. Cambridge University Press, 1985.
- [7] A. Chopra and M. Singh. Contextualizing commitment protocols. In *Proceedings of AAMAS*, pages 1345–1352. ACM, 2006.
- [8] S. DeLoach, W. Oyenar, and E. Matson. A capabilities-based model for adaptive organizations. *Autonomous Agents and Multi-Agent Systems*, 16(1):13–56, 2008.
- [9] V. Dignum, editor. *Multi-Agent Systems: Semantics and Dynamics of Organizational Models*. IGI, 2009.
- [10] M. Hoogendoorn, C. Jonker, M. Schut, and J. Treur. Modeling centralized organization of organizational change. *Computational & Mathematical Organization Theory*, 13(2):147–184, 2007.
- [11] A. Jones and M. Sergot. On the characterisation of law and computer systems: the normative systems perspective. In *Deontic Logic in Computer Science*, pages 275–307. J. Wiley and Sons, 1993.
- [12] A. Jones and M. Sergot. A formal characterisation of institutionalised power. *Journal of the IGPL*, 4(3):429–445, 1996.
- [13] D. Kaponis and J. Pitt. Dynamic specifications in norm-governed open computational societies. In *Proceedings of ESAW Workshop*, LNCS 4457, pages 265–283. Springer, 2007.
- [14] R. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67–96, 1986.
- [15] C. Martin and K. S. Barber. Adaptive decision-making frameworks for dynamic multi-agent organizational change. *Autonomous Agents and Multi-Agent Systems*, 13(3):391–428, 2006.
- [16] J. Pitt, L. Kamara, M. Sergot, and A. Artikis. Voting in multi-agent systems. *Computer Journal*, 49(2):156–170, 2006.
- [17] J. Searle. What is a speech act? In A. Martinich, editor, *Philosophy of Language*, pages 130–140. Oxford University Press, third edition, 1996.
- [18] M. Sergot. Action and agency in norm-governed multi-agent systems. In *Proceedings of ESAW VIII*, LNAI 4995, pages 1–54. Springer, 2008.
- [19] Y. Shoham and M. Tennenholtz. On the emergence of social conventions: modeling, analysis and simulations. *Artificial Intelligence*, 94(1–2):139–166, 1997.

6

Association Analysis: Basic Concepts and Algorithms

Many business enterprises accumulate large quantities of data from their day-to-day operations. For example, huge amounts of customer purchase data are collected daily at the checkout counters of grocery stores. Table 6.1 illustrates an example of such data, commonly known as **market basket transactions**. Each row in this table corresponds to a transaction, which contains a unique identifier labeled *TID* and a set of items bought by a given customer. Retailers are interested in analyzing the data to learn about the purchasing behavior of their customers. Such valuable information can be used to support a variety of business-related applications such as marketing promotions, inventory management, and customer relationship management.

This chapter presents a methodology known as **association analysis**, which is useful for discovering interesting relationships hidden in large data sets. The uncovered relationships can be represented in the form of **associa-**

Table 6.1. An example of market basket transactions.

<i>TID</i>	Items
1	{Bread, Milk}
2	{Bread, Diapers, Beer, Eggs}
3	{Milk, Diapers, Beer, Cola}
4	{Bread, Milk, Diapers, Beer}
5	{Bread, Milk, Diapers, Cola}

tion rules or sets of frequent items. For example, the following rule can be extracted from the data set shown in Table 6.1:

$$\{\text{Diapers}\} \longrightarrow \{\text{Beer}\}.$$

The rule suggests that a strong relationship exists between the sale of diapers and beer because many customers who buy diapers also buy beer. Retailers can use this type of rules to help them identify new opportunities for cross-selling their products to the customers.

Besides market basket data, association analysis is also applicable to other application domains such as bioinformatics, medical diagnosis, Web mining, and scientific data analysis. In the analysis of Earth science data, for example, the association patterns may reveal interesting connections among the ocean, land, and atmospheric processes. Such information may help Earth scientists develop a better understanding of how the different elements of the Earth system interact with each other. Even though the techniques presented here are generally applicable to a wider variety of data sets, for illustrative purposes, our discussion will focus mainly on market basket data.

There are two key issues that need to be addressed when applying association analysis to market basket data. First, discovering patterns from a large transaction data set can be computationally expensive. Second, some of the discovered patterns are potentially spurious because they may happen simply by chance. The remainder of this chapter is organized around these two issues. The first part of the chapter is devoted to explaining the basic concepts of association analysis and the algorithms used to efficiently mine such patterns. The second part of the chapter deals with the issue of evaluating the discovered patterns in order to prevent the generation of spurious results.

6.1 Problem Definition

This section reviews the basic terminology used in association analysis and presents a formal description of the task.

Binary Representation Market basket data can be represented in a binary format as shown in Table 6.2, where each row corresponds to a transaction and each column corresponds to an item. An item can be treated as a binary variable whose value is one if the item is present in a transaction and zero otherwise. Because the presence of an item in a transaction is often considered more important than its absence, an item is an **asymmetric** binary variable.

Table 6.2. A binary 0/1 representation of market basket data.

TID	Bread	Milk	Diapers	Beer	Eggs	Cola
1	1	1	0	0	0	0
2	1	0	1	1	1	0
3	0	1	1	1	0	1
4	1	1	1	1	0	0
5	1	1	1	0	0	1

This representation is perhaps a very simplistic view of real market basket data because it ignores certain important aspects of the data such as the quantity of items sold or the price paid to purchase them. Methods for handling such non-binary data will be explained in Chapter 7.

Itemset and Support Count Let $I = \{i_1, i_2, \dots, i_d\}$ be the set of all items in a market basket data and $T = \{t_1, t_2, \dots, t_N\}$ be the set of all transactions. Each transaction t_i contains a subset of items chosen from I . In association analysis, a collection of zero or more items is termed an itemset. If an itemset contains k items, it is called a k -itemset. For instance, **{Beer, Diapers, Milk}** is an example of a 3-itemset. The null (or empty) set is an itemset that does not contain any items.

The transaction width is defined as the number of items present in a transaction. A transaction t_j is said to contain an itemset X if X is a subset of t_j . For example, the second transaction shown in Table 6.2 contains the itemset **{Bread, Diapers}** but not **{Bread, Milk}**. An important property of an itemset is its support count, which refers to the number of transactions that contain a particular itemset. Mathematically, the support count, $\sigma(X)$, for an itemset X can be stated as follows:

$$\sigma(X) = |\{t_i | X \subseteq t_i, t_i \in T\}|,$$

where the symbol $|\cdot|$ denote the number of elements in a set. In the data set shown in Table 6.2, the support count for **{Beer, Diapers, Milk}** is equal to two because there are only two transactions that contain all three items.

Association Rule An association rule is an implication expression of the form $X \rightarrow Y$, where X and Y are disjoint itemsets, i.e., $X \cap Y = \emptyset$. The strength of an association rule can be measured in terms of its **support** and **confidence**. Support determines how often a rule is applicable to a given

data set, while confidence determines how frequently items in Y appear in transactions that contain X . The formal definitions of these metrics are

$$\text{Support, } s(X \rightarrow Y) = \frac{\sigma(X \cup Y)}{N}; \quad (6.1)$$

$$\text{Confidence, } c(X \rightarrow Y) = \frac{\sigma(X \cup Y)}{\sigma(X)}. \quad (6.2)$$

Example 6.1. Consider the rule $\{\text{Milk, Diapers}\} \rightarrow \{\text{Beer}\}$. Since the support count for $\{\text{Milk, Diapers, Beer}\}$ is 2 and the total number of transactions is 5, the rule's support is $2/5 = 0.4$. The rule's confidence is obtained by dividing the support count for $\{\text{Milk, Diapers, Beer}\}$ by the support count for $\{\text{Milk, Diapers}\}$. Since there are 3 transactions that contain milk and diapers, the confidence for this rule is $2/3 = 0.67$. ■

Why Use Support and Confidence? Support is an important measure because a rule that has very low support may occur simply by chance. A low support rule is also likely to be uninteresting from a business perspective because it may not be profitable to promote items that customers seldom buy together (with the exception of the situation described in Section 6.8). For these reasons, support is often used to eliminate uninteresting rules. As will be shown in Section 6.2.1, support also has a desirable property that can be exploited for the efficient discovery of association rules.

Confidence, on the other hand, measures the reliability of the inference made by a rule. For a given rule $X \rightarrow Y$, the higher the confidence, the more likely it is for Y to be present in transactions that contain X . Confidence also provides an estimate of the conditional probability of Y given X .

Association analysis results should be interpreted with caution. The inference made by an association rule does not necessarily imply causality. Instead, it suggests a strong co-occurrence relationship between items in the antecedent and consequent of the rule. Causality, on the other hand, requires knowledge about the causal and effect attributes in the data and typically involves relationships occurring over time (e.g., ozone depletion leads to global warming).

Formulation of Association Rule Mining Problem The association rule mining problem can be formally stated as follows:

Definition 6.1 (Association Rule Discovery). Given a set of transactions T , find all the rules having support $\geq \text{minsup}$ and confidence $\geq \text{minconf}$, where minsup and minconf are the corresponding support and confidence thresholds.

A brute-force approach for mining association rules is to compute the support and confidence for every possible rule. This approach is prohibitively expensive because there are exponentially many rules that can be extracted from a data set. More specifically, the total number of possible rules extracted from a data set that contains d items is

$$R = 3^d - 2^{d+1} + 1. \quad (6.3)$$

The proof for this equation is left as an exercise to the readers (see Exercise 5 on page 405). Even for the small data set shown in Table 6.1, this approach requires us to compute the support and confidence for $3^6 - 2^7 + 1 = 602$ rules. More than 80% of the rules are discarded after applying $\text{minsup} = 20\%$ and $\text{minconf} = 50\%$, thus making most of the computations become wasted. To avoid performing needless computations, it would be useful to prune the rules early without having to compute their support and confidence values.

An initial step toward improving the performance of association rule mining algorithms is to decouple the support and confidence requirements. From Equation 6.2, notice that the support of a rule $X \rightarrow Y$ depends only on the support of its corresponding itemset, $X \cup Y$. For example, the following rules have identical support because they involve items from the same itemset, $\{\text{Beer, Diapers, Milk}\}$:

$$\begin{aligned} \{\text{Beer, Diapers}\} &\rightarrow \{\text{Milk}\}, & \{\text{Beer, Milk}\} &\rightarrow \{\text{Diapers}\}, \\ \{\text{Diapers, Milk}\} &\rightarrow \{\text{Beer}\}, & \{\text{Beer}\} &\rightarrow \{\text{Diapers, Milk}\}, \\ \{\text{Milk}\} &\rightarrow \{\text{Beer, Diapers}\}, & \{\text{Diapers}\} &\rightarrow \{\text{Beer, Milk}\}. \end{aligned}$$

If the itemset is infrequent, then all six candidate rules can be pruned immediately without our having to compute their confidence values.

Therefore, a common strategy adopted by many association rule mining algorithms is to decompose the problem into two major subtasks:

1. **Frequent Itemset Generation**, whose objective is to find all the itemsets that satisfy the minsup threshold. These itemsets are called frequent itemsets.
2. **Rule Generation**, whose objective is to extract all the high-confidence rules from the frequent itemsets found in the previous step. These rules are called strong rules.

The computational requirements for frequent itemset generation are generally more expensive than those of rule generation. Efficient techniques for generating frequent itemsets and association rules are discussed in Sections 6.2 and 6.3, respectively.

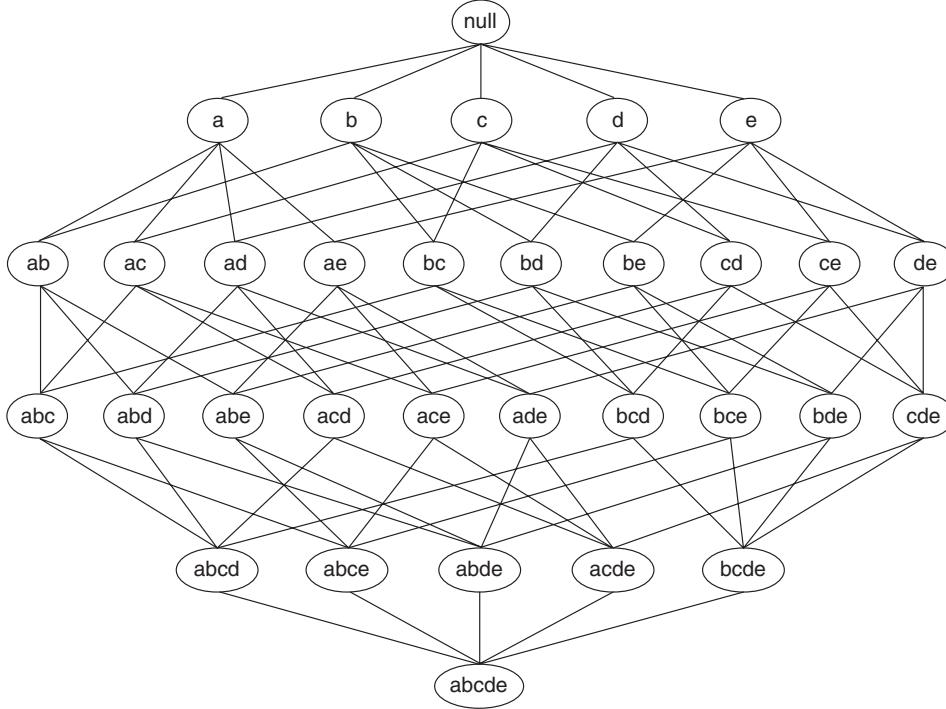


Figure 6.1. An itemset lattice.

6.2 Frequent Itemset Generation

A lattice structure can be used to enumerate the list of all possible itemsets. Figure 6.1 shows an itemset lattice for $I = \{a, b, c, d, e\}$. In general, a data set that contains k items can potentially generate up to $2^k - 1$ frequent itemsets, excluding the null set. Because k can be very large in many practical applications, the search space of itemsets that need to be explored is exponentially large.

A brute-force approach for finding frequent itemsets is to determine the support count for every **candidate itemset** in the lattice structure. To do this, we need to compare each candidate against every transaction, an operation that is shown in Figure 6.2. If the candidate is contained in a transaction, its support count will be incremented. For example, the support for {Bread, Milk} is incremented three times because the itemset is contained in transactions 1, 4, and 5. Such an approach can be very expensive because it requires $O(NMw)$ comparisons, where N is the number of transactions, $M = 2^k - 1$ is the number of candidate itemsets, and w is the maximum transaction width.

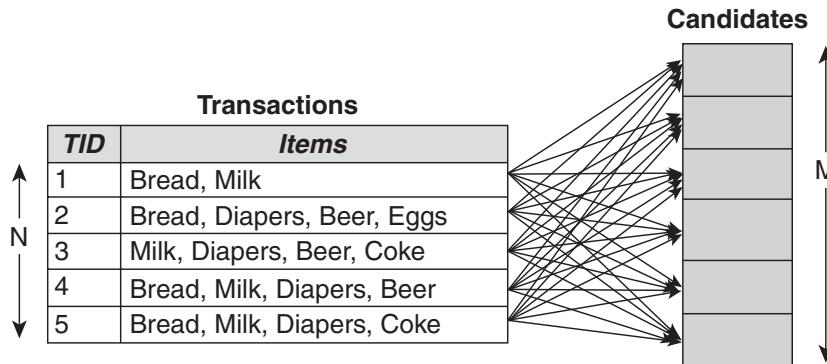


Figure 6.2. Counting the support of candidate itemsets.

There are several ways to reduce the computational complexity of frequent itemset generation.

1. **Reduce the number of candidate itemsets (M).** The *Apriori* principle, described in the next section, is an effective way to eliminate some of the candidate itemsets without counting their support values.
2. **Reduce the number of comparisons.** Instead of matching each candidate itemset against every transaction, we can reduce the number of comparisons by using more advanced data structures, either to store the candidate itemsets or to compress the data set. We will discuss these strategies in Sections 6.2.4 and 6.6.

6.2.1 The *Apriori* Principle

This section describes how the support measure helps to reduce the number of candidate itemsets explored during frequent itemset generation. The use of support for pruning candidate itemsets is guided by the following principle.

Theorem 6.1 (Apriori Principle). *If an itemset is frequent, then all of its subsets must also be frequent.*

To illustrate the idea behind the *Apriori* principle, consider the itemset lattice shown in Figure 6.3. Suppose $\{c, d, e\}$ is a frequent itemset. Clearly, any transaction that contains $\{c, d, e\}$ must also contain its subsets, $\{c, d\}$, $\{c, e\}$, $\{d, e\}$, $\{c\}$, $\{d\}$, and $\{e\}$. As a result, if $\{c, d, e\}$ is frequent, then all subsets of $\{c, d, e\}$ (i.e., the shaded itemsets in this figure) must also be frequent.

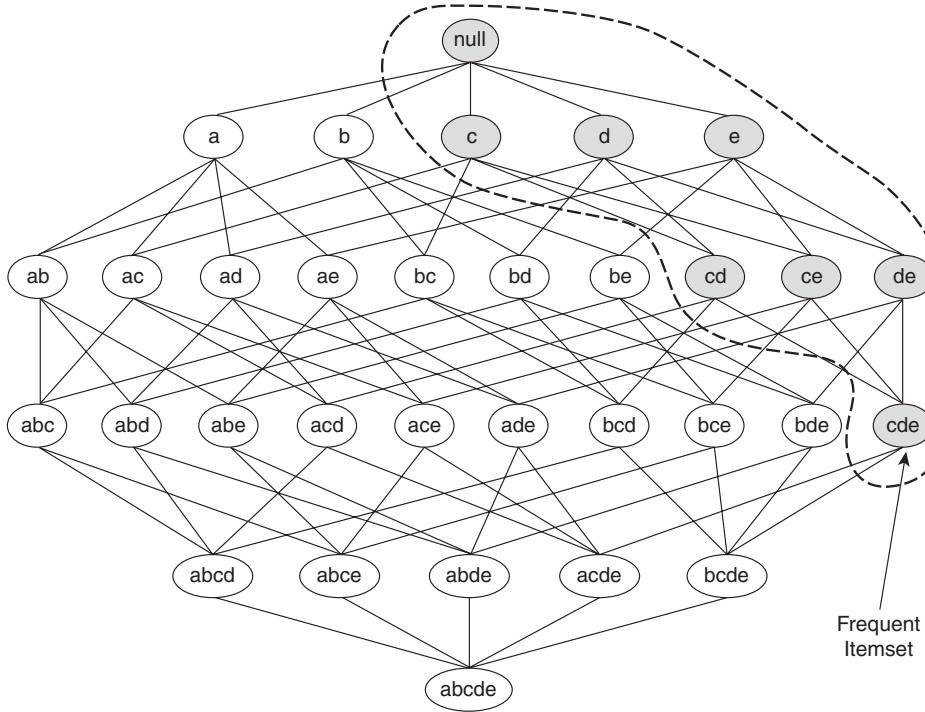


Figure 6.3. An illustration of the *Apriori* principle. If $\{c, d, e\}$ is frequent, then all subsets of this itemset are frequent.

Conversely, if an itemset such as $\{a, b\}$ is infrequent, then all of its supersets must be infrequent too. As illustrated in Figure 6.4, the entire subgraph containing the supersets of $\{a, b\}$ can be pruned immediately once $\{a, b\}$ is found to be infrequent. This strategy of trimming the exponential search space based on the support measure is known as **support-based pruning**. Such a pruning strategy is made possible by a key property of the support measure, namely, that the support for an itemset never exceeds the support for its subsets. This property is also known as the **anti-monotone** property of the support measure.

Definition 6.2 (Monotonicity Property). Let I be a set of items, and $J = 2^I$ be the power set of I . A measure f is monotone (or upward closed) if

$$\forall X, Y \in J : (X \subseteq Y) \longrightarrow f(X) \leq f(Y),$$

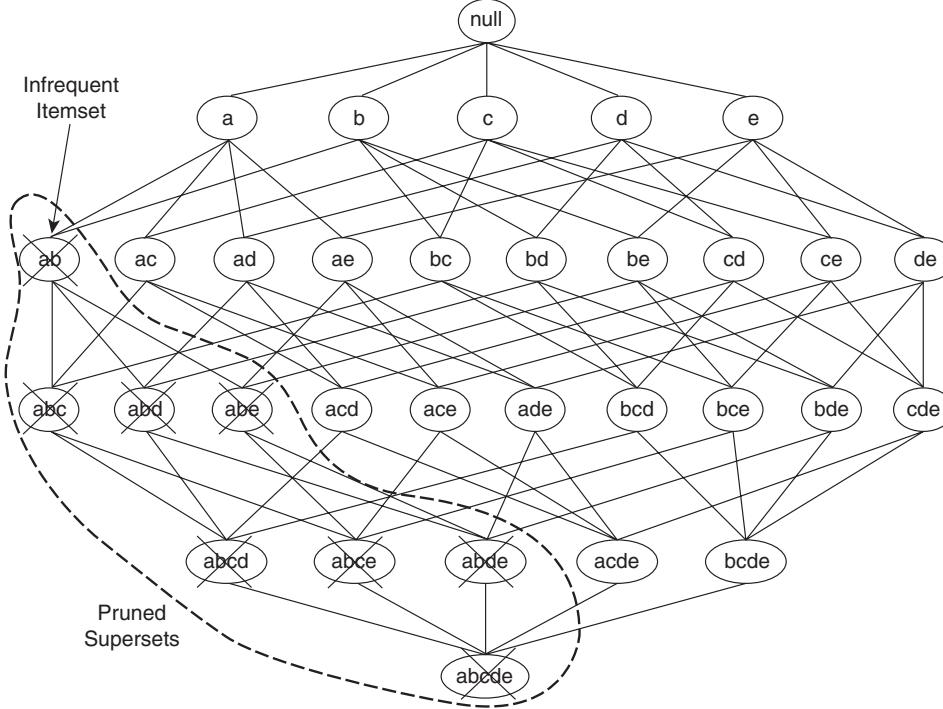


Figure 6.4. An illustration of support-based pruning. If $\{a, b\}$ is infrequent, then all supersets of $\{a, b\}$ are infrequent.

which means that if X is a subset of Y , then $f(X)$ must not exceed $f(Y)$. On the other hand, f is anti-monotone (or downward closed) if

$$\forall X, Y \in J : (X \subseteq Y) \longrightarrow f(Y) \leq f(X),$$

which means that if X is a subset of Y , then $f(Y)$ must not exceed $f(X)$.

Any measure that possesses an anti-monotone property can be incorporated directly into the mining algorithm to effectively prune the exponential search space of candidate itemsets, as will be shown in the next section.

6.2.2 Frequent Itemset Generation in the *Apriori* Algorithm

Apriori is the first association rule mining algorithm that pioneered the use of support-based pruning to systematically control the exponential growth of candidate itemsets. Figure 6.5 provides a high-level illustration of the frequent itemset generation part of the *Apriori* algorithm for the transactions shown in

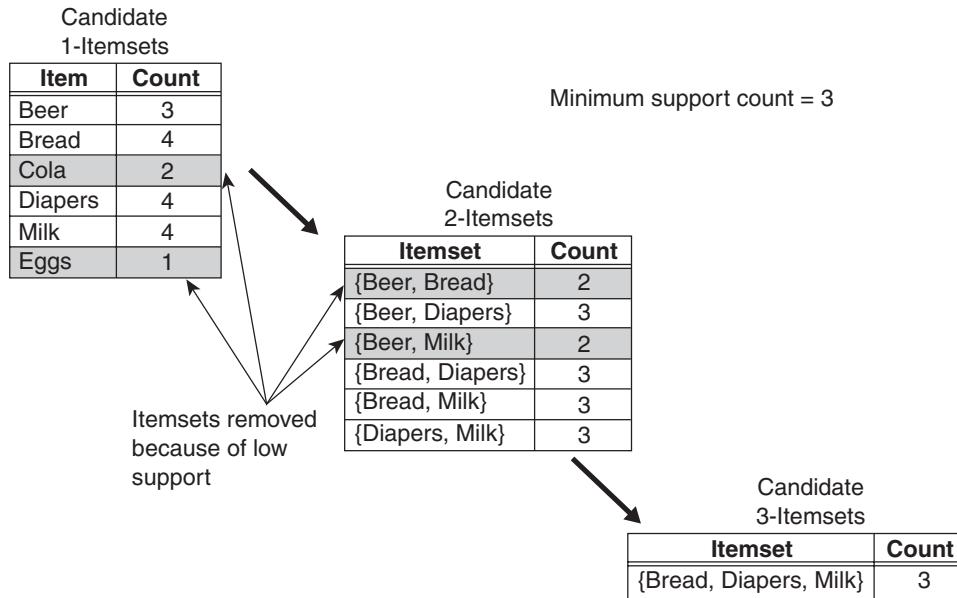


Figure 6.5. Illustration of frequent itemset generation using the *Apriori* algorithm.

Table 6.1. We assume that the support threshold is 60%, which is equivalent to a minimum support count equal to 3.

Initially, every item is considered as a candidate 1-itemset. After counting their supports, the candidate itemsets $\{\text{Cola}\}$ and $\{\text{Eggs}\}$ are discarded because they appear in fewer than three transactions. In the next iteration, candidate 2-itemsets are generated using only the frequent 1-itemsets because the *Apriori* principle ensures that all supersets of the infrequent 1-itemsets must be infrequent. Because there are only four frequent 1-itemsets, the number of candidate 2-itemsets generated by the algorithm is $\binom{4}{2} = 6$. Two of these six candidates, $\{\text{Beer}, \text{Bread}\}$ and $\{\text{Beer}, \text{Milk}\}$, are subsequently found to be infrequent after computing their support values. The remaining four candidates are frequent, and thus will be used to generate candidate 3-itemsets. Without support-based pruning, there are $\binom{6}{3} = 20$ candidate 3-itemsets that can be formed using the six items given in this example. With the *Apriori* principle, we only need to keep candidate 3-itemsets whose subsets are frequent. The only candidate that has this property is $\{\text{Bread}, \text{Diapers}, \text{Milk}\}$.

The effectiveness of the *Apriori* pruning strategy can be shown by counting the number of candidate itemsets generated. A brute-force strategy of

enumerating all itemsets (up to size 3) as candidates will produce

$$\binom{6}{1} + \binom{6}{2} + \binom{6}{3} = 6 + 15 + 20 = 41$$

candidates. With the *Apriori* principle, this number decreases to

$$\binom{6}{1} + \binom{4}{2} + 1 = 6 + 6 + 1 = 13$$

candidates, which represents a 68% reduction in the number of candidate itemsets even in this simple example.

The pseudocode for the frequent itemset generation part of the *Apriori* algorithm is shown in Algorithm 6.1. Let C_k denote the set of candidate k -itemsets and F_k denote the set of frequent k -itemsets:

- The algorithm initially makes a single pass over the data set to determine the support of each item. Upon completion of this step, the set of all frequent 1-itemsets, F_1 , will be known (steps 1 and 2).
- Next, the algorithm will iteratively generate new candidate k -itemsets using the frequent $(k - 1)$ -itemsets found in the previous iteration (step 5). Candidate generation is implemented using a function called `apriori-gen`, which is described in Section 6.2.3.

Algorithm 6.1 Frequent itemset generation of the *Apriori* algorithm.

```

1:  $k = 1$ .
2:  $F_k = \{ i \mid i \in I \wedge \sigma(\{i\}) \geq N \times \text{minsup} \}$ . {Find all frequent 1-itemsets}
3: repeat
4:    $k = k + 1$ .
5:    $C_k = \text{apriori-gen}(F_{k-1})$ . {Generate candidate itemsets}
6:   for each transaction  $t \in T$  do
7:      $C_t = \text{subset}(C_k, t)$ . {Identify all candidates that belong to  $t$ }
8:     for each candidate itemset  $c \in C_t$  do
9:        $\sigma(c) = \sigma(c) + 1$ . {Increment support count}
10:    end for
11:   end for
12:    $F_k = \{ c \mid c \in C_k \wedge \sigma(c) \geq N \times \text{minsup} \}$ . {Extract the frequent  $k$ -itemsets}
13: until  $F_k = \emptyset$ 
14: Result =  $\bigcup F_k$ .

```

- To count the support of the candidates, the algorithm needs to make an additional pass over the data set (steps 6–10). The subset function is used to determine all the candidate itemsets in C_k that are contained in each transaction t . The implementation of this function is described in Section 6.2.4.
- After counting their supports, the algorithm eliminates all candidate itemsets whose support counts are less than minsup (step 12).
- The algorithm terminates when there are no new frequent itemsets generated, i.e., $F_k = \emptyset$ (step 13).

The frequent itemset generation part of the *Apriori* algorithm has two important characteristics. First, it is a **level-wise** algorithm; i.e., it traverses the itemset lattice one level at a time, from frequent 1-itemsets to the maximum size of frequent itemsets. Second, it employs a **generate-and-test** strategy for finding frequent itemsets. At each iteration, new candidate itemsets are generated from the frequent itemsets found in the previous iteration. The support for each candidate is then counted and tested against the minsup threshold. The total number of iterations needed by the algorithm is $k_{\max} + 1$, where k_{\max} is the maximum size of the frequent itemsets.

6.2.3 Candidate Generation and Pruning

The apriori-gen function shown in Step 5 of Algorithm 6.1 generates candidate itemsets by performing the following two operations:

1. **Candidate Generation.** This operation generates new candidate k -itemsets based on the frequent $(k - 1)$ -itemsets found in the previous iteration.
2. **Candidate Pruning.** This operation eliminates some of the candidate k -itemsets using the support-based pruning strategy.

To illustrate the candidate pruning operation, consider a candidate k -itemset, $X = \{i_1, i_2, \dots, i_k\}$. The algorithm must determine whether all of its proper subsets, $X - \{i_j\}$ ($\forall j = 1, 2, \dots, k$), are frequent. If one of them is infrequent, then X is immediately pruned. This approach can effectively reduce the number of candidate itemsets considered during support counting. The complexity of this operation is $O(k)$ for each candidate k -itemset. However, as will be shown later, we do not have to examine all k subsets of a given candidate itemset. If m of the k subsets were used to generate a candidate, we only need to check the remaining $k - m$ subsets during candidate pruning.

In principle, there are many ways to generate candidate itemsets. The following is a list of requirements for an effective candidate generation procedure:

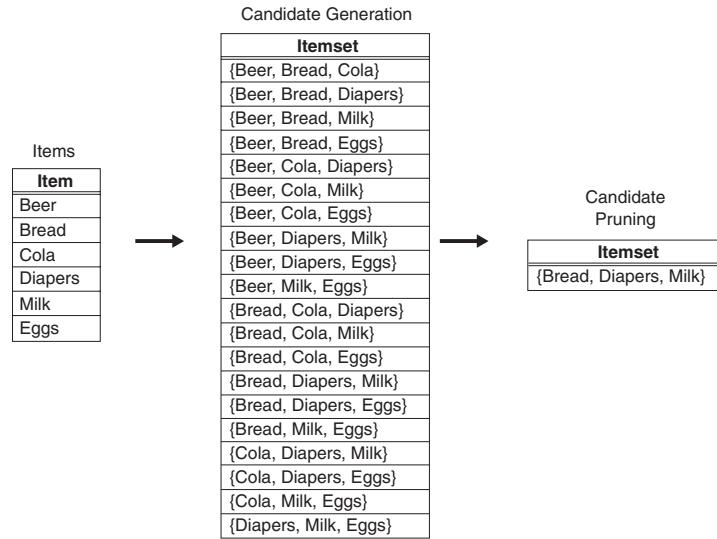
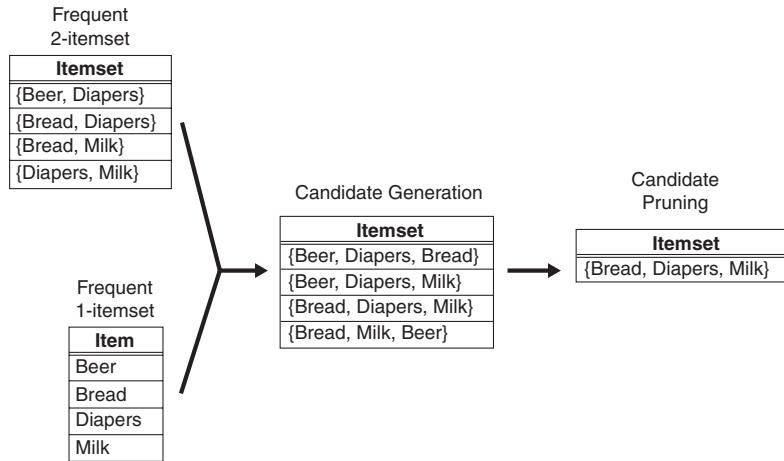
1. It should avoid generating too many unnecessary candidates. A candidate itemset is unnecessary if at least one of its subsets is infrequent. Such a candidate is guaranteed to be infrequent according to the anti-monotone property of support.
2. It must ensure that the candidate set is complete, i.e., no frequent itemsets are left out by the candidate generation procedure. To ensure completeness, the set of candidate itemsets must subsume the set of all frequent itemsets, i.e., $\forall k : F_k \subseteq C_k$.
3. It should not generate the same candidate itemset more than once. For example, the candidate itemset $\{a, b, c, d\}$ can be generated in many ways—by merging $\{a, b, c\}$ with $\{d\}$, $\{b, d\}$ with $\{a, c\}$, $\{c\}$ with $\{a, b, d\}$, etc. Generation of duplicate candidates leads to wasted computations and thus should be avoided for efficiency reasons.

Next, we will briefly describe several candidate generation procedures, including the one used by the `apriori-gen` function.

Brute-Force Method The brute-force method considers every k -itemset as a potential candidate and then applies the candidate pruning step to remove any unnecessary candidates (see Figure 6.6). The number of candidate itemsets generated at level k is equal to $\binom{d}{k}$, where d is the total number of items. Although candidate generation is rather trivial, candidate pruning becomes extremely expensive because a large number of itemsets must be examined. Given that the amount of computations needed for each candidate is $O(k)$, the overall complexity of this method is $O(\sum_{k=1}^d k \times \binom{d}{k}) = O(d \cdot 2^{d-1})$.

$F_{k-1} \times F_1$ Method An alternative method for candidate generation is to extend each frequent $(k - 1)$ -itemset with other frequent items. Figure 6.7 illustrates how a frequent 2-itemset such as $\{\text{Beer}, \text{Diapers}\}$ can be augmented with a frequent item such as `Bread` to produce a candidate 3-itemset $\{\text{Beer}, \text{Diapers}, \text{Bread}\}$. This method will produce $O(|F_{k-1}| \times |F_1|)$ candidate k -itemsets, where $|F_j|$ is the number of frequent j -itemsets. The overall complexity of this step is $O(\sum_k k |F_{k-1}| |F_1|)$.

The procedure is complete because every frequent k -itemset is composed of a frequent $(k - 1)$ -itemset and a frequent 1-itemset. Therefore, all frequent k -itemsets are part of the candidate k -itemsets generated by this procedure.

**Figure 6.6.** A brute-force method for generating candidate 3-itemsets.**Figure 6.7.** Generating and pruning candidate k -itemsets by merging a frequent $(k - 1)$ -itemset with a frequent item. Note that some of the candidates are unnecessary because their subsets are infrequent.

This approach, however, does not prevent the same candidate itemset from being generated more than once. For instance, {Bread, Diapers, Milk} can be generated by merging {Bread, Diapers} with {Milk}, {Bread, Milk} with {Diapers}, or {Diapers, Milk} with {Bread}. One way to avoid generating

duplicate candidates is by ensuring that the items in each frequent itemset are kept sorted in their lexicographic order. Each frequent $(k-1)$ -itemset X is then extended with frequent items that are lexicographically larger than the items in X . For example, the itemset $\{\text{Bread}, \text{Diapers}\}$ can be augmented with $\{\text{Milk}\}$ since Milk is lexicographically larger than Bread and Diapers . However, we should not augment $\{\text{Diapers}, \text{Milk}\}$ with $\{\text{Bread}\}$ nor $\{\text{Bread}, \text{Milk}\}$ with $\{\text{Diapers}\}$ because they violate the lexicographic ordering condition.

While this procedure is a substantial improvement over the brute-force method, it can still produce a large number of unnecessary candidates. For example, the candidate itemset obtained by merging $\{\text{Beer}, \text{Diapers}\}$ with $\{\text{Milk}\}$ is unnecessary because one of its subsets, $\{\text{Beer}, \text{Milk}\}$, is infrequent. There are several heuristics available to reduce the number of unnecessary candidates. For example, note that, for every candidate k -itemset that survives the pruning step, every item in the candidate must be contained in at least $k - 1$ of the frequent $(k - 1)$ -itemsets. Otherwise, the candidate is guaranteed to be infrequent. For example, $\{\text{Beer}, \text{Diapers}, \text{Milk}\}$ is a viable candidate 3-itemset only if every item in the candidate, including Beer , is contained in at least two frequent 2-itemsets. Since there is only one frequent 2-itemset containing Beer , all candidate itemsets involving Beer must be infrequent.

$\mathbf{F}_{k-1} \times \mathbf{F}_{k-1}$ Method The candidate generation procedure in the apriori-gen function merges a pair of frequent $(k - 1)$ -itemsets only if their first $k - 2$ items are identical. Let $A = \{a_1, a_2, \dots, a_{k-1}\}$ and $B = \{b_1, b_2, \dots, b_{k-1}\}$ be a pair of frequent $(k - 1)$ -itemsets. A and B are merged if they satisfy the following conditions:

$$a_i = b_i \text{ (for } i = 1, 2, \dots, k - 2\text{) and } a_{k-1} \neq b_{k-1}.$$

In Figure 6.8, the frequent itemsets $\{\text{Bread}, \text{Diapers}\}$ and $\{\text{Bread}, \text{Milk}\}$ are merged to form a candidate 3-itemset $\{\text{Bread}, \text{Diapers}, \text{Milk}\}$. The algorithm does not have to merge $\{\text{Beer}, \text{Diapers}\}$ with $\{\text{Diapers}, \text{Milk}\}$ because the first item in both itemsets is different. Indeed, if $\{\text{Beer}, \text{Diapers}, \text{Milk}\}$ is a viable candidate, it would have been obtained by merging $\{\text{Beer}, \text{Diapers}\}$ with $\{\text{Beer}, \text{Milk}\}$ instead. This example illustrates both the completeness of the candidate generation procedure and the advantages of using lexicographic ordering to prevent duplicate candidates. However, because each candidate is obtained by merging a pair of frequent $(k - 1)$ -itemsets, an additional candidate pruning step is needed to ensure that the remaining $k - 2$ subsets of the candidate are frequent.

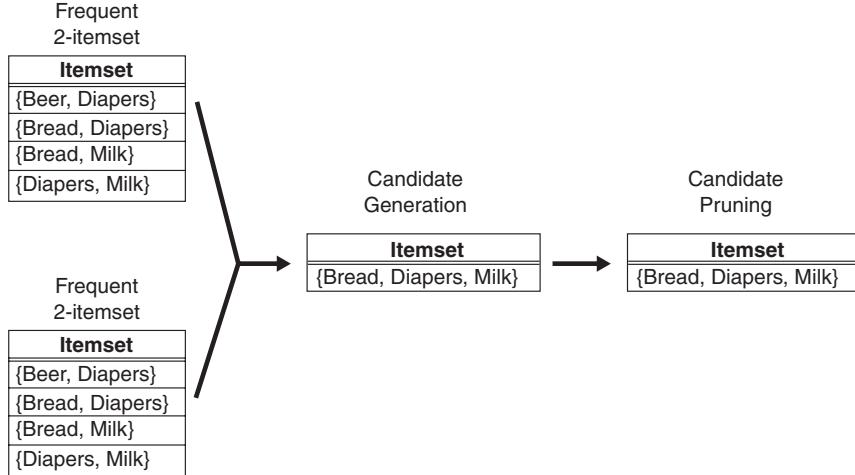


Figure 6.8. Generating and pruning candidate k -itemsets by merging pairs of frequent $(k-1)$ -itemsets.

6.2.4 Support Counting

Support counting is the process of determining the frequency of occurrence for every candidate itemset that survives the candidate pruning step of the apriori-gen function. Support counting is implemented in steps 6 through 11 of Algorithm 6.1. One approach for doing this is to compare each transaction against every candidate itemset (see Figure 6.2) and to update the support counts of candidates contained in the transaction. This approach is computationally expensive, especially when the numbers of transactions and candidate itemsets are large.

An alternative approach is to enumerate the itemsets contained in each transaction and use them to update the support counts of their respective candidate itemsets. To illustrate, consider a transaction t that contains five items, $\{1, 2, 3, 5, 6\}$. There are $\binom{5}{3} = 10$ itemsets of size 3 contained in this transaction. Some of the itemsets may correspond to the candidate 3-itemsets under investigation, in which case, their support counts are incremented. Other subsets of t that do not correspond to any candidates can be ignored.

Figure 6.9 shows a systematic way for enumerating the 3-itemsets contained in t . Assuming that each itemset keeps its items in increasing lexicographic order, an itemset can be enumerated by specifying the smallest item first, followed by the larger items. For instance, given $t = \{1, 2, 3, 5, 6\}$, all the 3-itemsets contained in t must begin with item 1, 2, or 3. It is not possible to construct a 3-itemset that begins with items 5 or 6 because there are only two

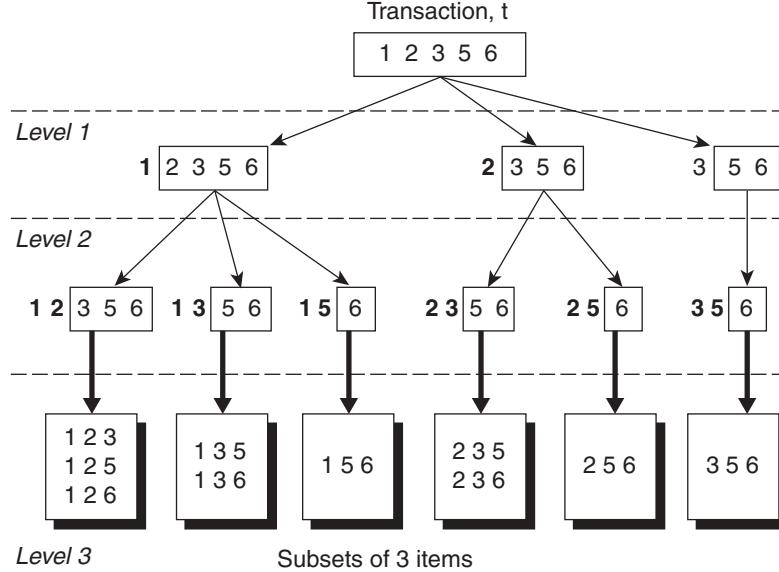


Figure 6.9. Enumerating subsets of three items from a transaction t .

items in t whose labels are greater than or equal to 5. The number of ways to specify the first item of a 3-itemset contained in t is illustrated by the Level 1 prefix structures depicted in Figure 6.9. For instance, $1 \boxed{2 3 5 6}$ represents a 3-itemset that begins with item 1, followed by two more items chosen from the set $\{2, 3, 5, 6\}$.

After fixing the first item, the prefix structures at Level 2 represent the number of ways to select the second item. For example, $1 2 \boxed{3 5 6}$ corresponds to itemsets that begin with prefix $\{1 2\}$ and are followed by items 3, 5, or 6. Finally, the prefix structures at Level 3 represent the complete set of 3-itemsets contained in t . For example, the 3-itemsets that begin with prefix $\{1 2\}$ are $\{1, 2, 3\}$, $\{1, 2, 5\}$, and $\{1, 2, 6\}$, while those that begin with prefix $\{2 3\}$ are $\{2, 3, 5\}$ and $\{2, 3, 6\}$.

The prefix structures shown in Figure 6.9 demonstrate how itemsets contained in a transaction can be systematically enumerated, i.e., by specifying their items one by one, from the leftmost item to the rightmost item. We still have to determine whether each enumerated 3-itemset corresponds to an existing candidate itemset. If it matches one of the candidates, then the support count of the corresponding candidate is incremented. In the next section, we illustrate how this matching operation can be performed efficiently using a hash tree structure.

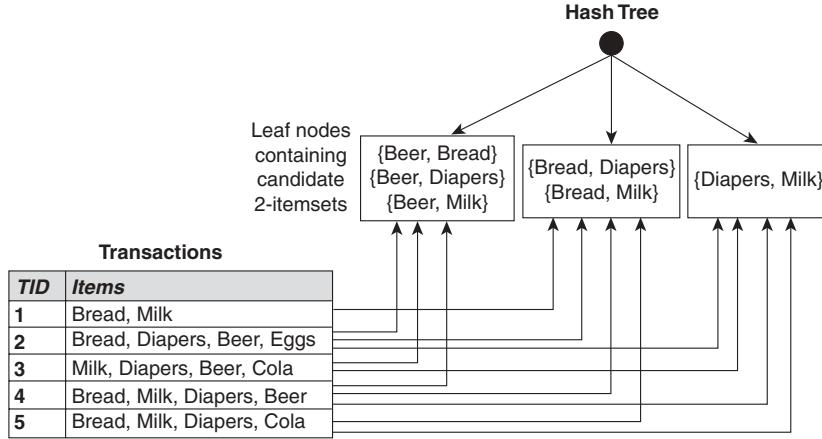


Figure 6.10. Counting the support of itemsets using hash structure.

Support Counting Using a Hash Tree

In the *Apriori* algorithm, candidate itemsets are partitioned into different buckets and stored in a hash tree. During support counting, itemsets contained in each transaction are also hashed into their appropriate buckets. That way, instead of comparing each itemset in the transaction with every candidate itemset, it is matched only against candidate itemsets that belong to the same bucket, as shown in Figure 6.10.

Figure 6.11 shows an example of a hash tree structure. Each internal node of the tree uses the following hash function, $h(p) = p \bmod 3$, to determine which branch of the current node should be followed next. For example, items 1, 4, and 7 are hashed to the same branch (i.e., the leftmost branch) because they have the same remainder after dividing the number by 3. All candidate itemsets are stored at the leaf nodes of the hash tree. The hash tree shown in Figure 6.11 contains 15 candidate 3-itemsets, distributed across 9 leaf nodes.

Consider a transaction, $t = \{1, 2, 3, 5, 6\}$. To update the support counts of the candidate itemsets, the hash tree must be traversed in such a way that all the leaf nodes containing candidate 3-itemsets belonging to t must be visited at least once. Recall that the 3-itemsets contained in t must begin with items 1, 2, or 3, as indicated by the Level 1 prefix structures shown in Figure 6.9. Therefore, at the root node of the hash tree, the items 1, 2, and 3 of the transaction are hashed separately. Item 1 is hashed to the left child of the root node, item 2 is hashed to the middle child, and item 3 is hashed to the right child. At the next level of the tree, the transaction is hashed on the second

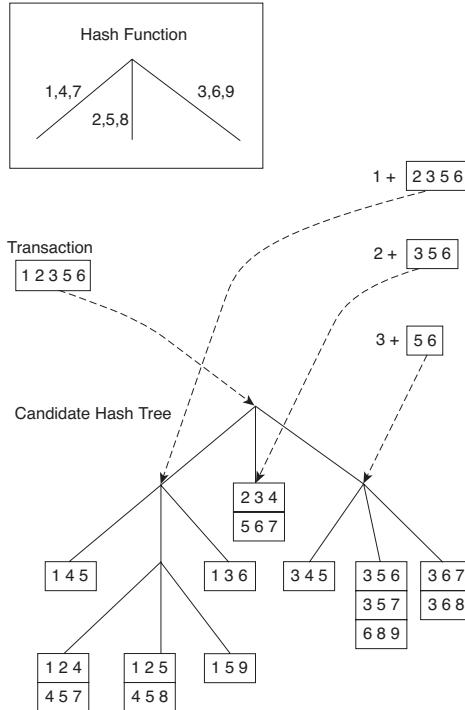


Figure 6.11. Hashing a transaction at the root node of a hash tree.

item listed in the Level 2 structures shown in Figure 6.9. For example, after hashing on item 1 at the root node, items 2, 3, and 5 of the transaction are hashed. Items 2 and 5 are hashed to the middle child, while item 3 is hashed to the right child, as shown in Figure 6.12. This process continues until the leaf nodes of the hash tree are reached. The candidate itemsets stored at the visited leaf nodes are compared against the transaction. If a candidate is a subset of the transaction, its support count is incremented. In this example, 5 out of the 9 leaf nodes are visited and 9 out of the 15 itemsets are compared against the transaction.

6.2.5 Computational Complexity

The computational complexity of the *Apriori* algorithm can be affected by the following factors.

Support Threshold Lowering the support threshold often results in more itemsets being declared as frequent. This has an adverse effect on the com-

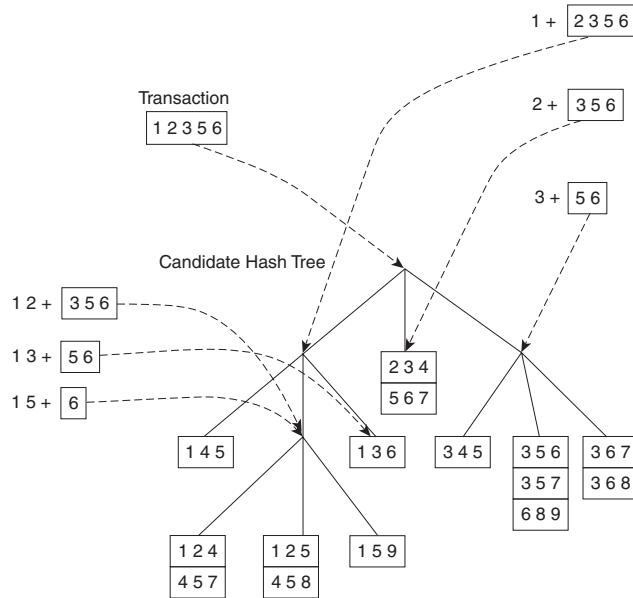


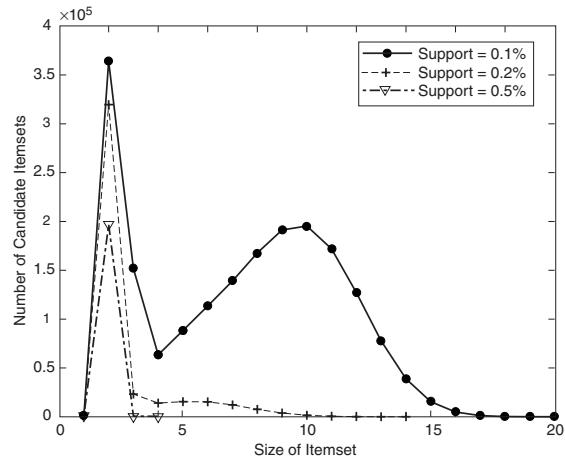
Figure 6.12. Subset operation on the leftmost subtree of the root of a candidate hash tree.

putational complexity of the algorithm because more candidate itemsets must be generated and counted, as shown in Figure 6.13. The maximum size of frequent itemsets also tends to increase with lower support thresholds. As the maximum size of the frequent itemsets increases, the algorithm will need to make more passes over the data set.

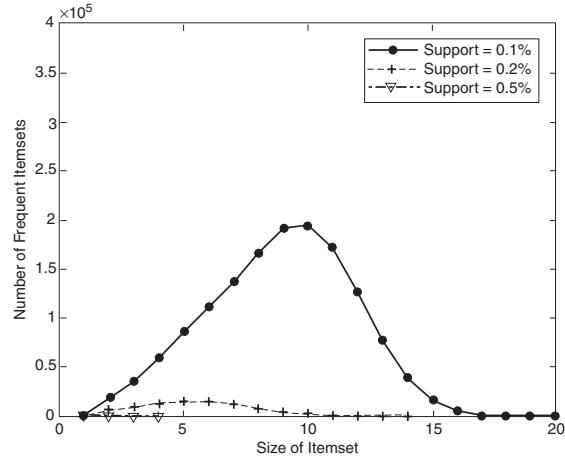
Number of Items (Dimensionality) As the number of items increases, more space will be needed to store the support counts of items. If the number of frequent items also grows with the dimensionality of the data, the computation and I/O costs will increase because of the larger number of candidate itemsets generated by the algorithm.

Number of Transactions Since the *Apriori* algorithm makes repeated passes over the data set, its run time increases with a larger number of transactions.

Average Transaction Width For dense data sets, the average transaction width can be very large. This affects the complexity of the *Apriori* algorithm in two ways. First, the maximum size of frequent itemsets tends to increase as the



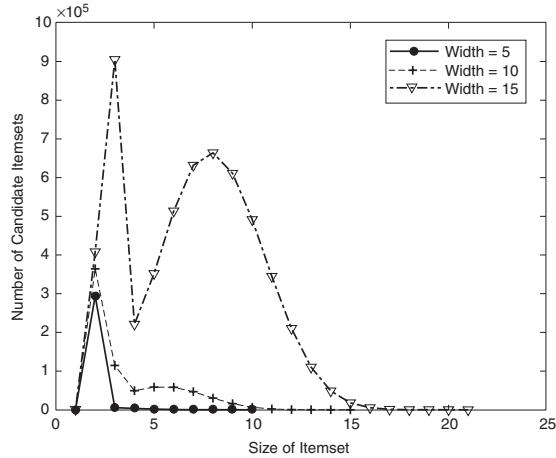
(a) Number of candidate itemsets.



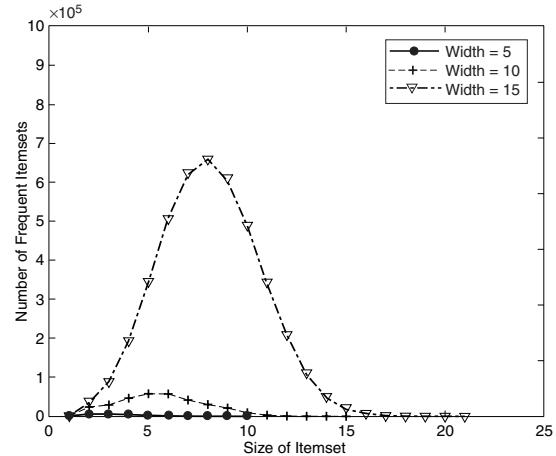
(b) Number of frequent itemsets.

Figure 6.13. Effect of support threshold on the number of candidate and frequent itemsets.

average transaction width increases. As a result, more candidate itemsets must be examined during candidate generation and support counting, as illustrated in Figure 6.14. Second, as the transaction width increases, more itemsets



(a) Number of candidate itemsets.



(b) Number of Frequent Itemsets.

Figure 6.14. Effect of average transaction width on the number of candidate and frequent itemsets.

are contained in the transaction. This will increase the number of hash tree traversals performed during support counting.

A detailed analysis of the time complexity for the *Apriori* algorithm is presented next.

Generation of frequent 1-itemsets For each transaction, we need to update the support count for every item present in the transaction. Assuming that w is the average transaction width, this operation requires $O(Nw)$ time, where N is the total number of transactions.

Candidate generation To generate candidate k -itemsets, pairs of frequent $(k - 1)$ -itemsets are merged to determine whether they have at least $k - 2$ items in common. Each merging operation requires at most $k - 2$ equality comparisons. In the best-case scenario, every merging step produces a viable candidate k -itemset. In the worst-case scenario, the algorithm must merge every pair of frequent $(k - 1)$ -itemsets found in the previous iteration. Therefore, the overall cost of merging frequent itemsets is

$$\sum_{k=2}^w (k - 2)|C_k| < \text{Cost of merging} < \sum_{k=2}^w (k - 2)|F_{k-1}|^2.$$

A hash tree is also constructed during candidate generation to store the candidate itemsets. Because the maximum depth of the tree is k , the cost for populating the hash tree with candidate itemsets is $O(\sum_{k=2}^w k|C_k|)$. During candidate pruning, we need to verify that the $k - 2$ subsets of every candidate k -itemset are frequent. Since the cost for looking up a candidate in a hash tree is $O(k)$, the candidate pruning step requires $O(\sum_{k=2}^w k(k - 2)|C_k|)$ time.

Support counting Each transaction of length $|t|$ produces $\binom{|t|}{k}$ itemsets of size k . This is also the effective number of hash tree traversals performed for each transaction. The cost for support counting is $O(N \sum_k \binom{w}{k} \alpha_k)$, where w is the maximum transaction width and α_k is the cost for updating the support count of a candidate k -itemset in the hash tree.

6.3 Rule Generation

This section describes how to extract association rules efficiently from a given frequent itemset. Each frequent k -itemset, Y , can produce up to $2^k - 2$ association rules, ignoring rules that have empty antecedents or consequents ($\emptyset \rightarrow Y$ or $Y \rightarrow \emptyset$). An association rule can be extracted by partitioning the itemset Y into two non-empty subsets, X and $Y - X$, such that $X \rightarrow Y - X$ satisfies the confidence threshold. Note that all such rules must have already met the support threshold because they are generated from a frequent itemset.

Example 6.2. Let $X = \{1, 2, 3\}$ be a frequent itemset. There are six candidate association rules that can be generated from X : $\{1, 2\} \rightarrow \{3\}$, $\{1, 3\} \rightarrow \{2\}$, $\{2, 3\} \rightarrow \{1\}$, $\{1\} \rightarrow \{2, 3\}$, $\{2\} \rightarrow \{1, 3\}$, and $\{3\} \rightarrow \{1, 2\}$. As each of their support is identical to the support for X , the rules must satisfy the support threshold. ■

Computing the confidence of an association rule does not require additional scans of the transaction data set. Consider the rule $\{1, 2\} \rightarrow \{3\}$, which is generated from the frequent itemset $X = \{1, 2, 3\}$. The confidence for this rule is $\sigma(\{1, 2, 3\})/\sigma(\{1, 2\})$. Because $\{1, 2, 3\}$ is frequent, the anti-monotone property of support ensures that $\{1, 2\}$ must be frequent, too. Since the support counts for both itemsets were already found during frequent itemset generation, there is no need to read the entire data set again.

6.3.1 Confidence-Based Pruning

Unlike the support measure, confidence does not have any monotone property. For example, the confidence for $X \rightarrow Y$ can be larger, smaller, or equal to the confidence for another rule $\tilde{X} \rightarrow \tilde{Y}$, where $\tilde{X} \subseteq X$ and $\tilde{Y} \subseteq Y$ (see Exercise 3 on page 405). Nevertheless, if we compare rules generated from the same frequent itemset Y , the following theorem holds for the confidence measure.

Theorem 6.2. *If a rule $X \rightarrow Y - X$ does not satisfy the confidence threshold, then any rule $X' \rightarrow Y - X'$, where X' is a subset of X , must not satisfy the confidence threshold as well.*

To prove this theorem, consider the following two rules: $X' \rightarrow Y - X'$ and $X \rightarrow Y - X$, where $X' \subset X$. The confidence of the rules are $\sigma(Y)/\sigma(X')$ and $\sigma(Y)/\sigma(X)$, respectively. Since X' is a subset of X , $\sigma(X') \geq \sigma(X)$. Therefore, the former rule cannot have a higher confidence than the latter rule.

6.3.2 Rule Generation in *Apriori* Algorithm

The *Apriori* algorithm uses a level-wise approach for generating association rules, where each level corresponds to the number of items that belong to the rule consequent. Initially, all the high-confidence rules that have only one item in the rule consequent are extracted. These rules are then used to generate new candidate rules. For example, if $\{acd\} \rightarrow \{b\}$ and $\{abd\} \rightarrow \{c\}$ are high-confidence rules, then the candidate rule $\{ad\} \rightarrow \{bc\}$ is generated by merging the consequents of both rules. Figure 6.15 shows a lattice structure for the association rules generated from the frequent itemset $\{a, b, c, d\}$. If any node in the lattice has low confidence, then according to Theorem 6.2, the

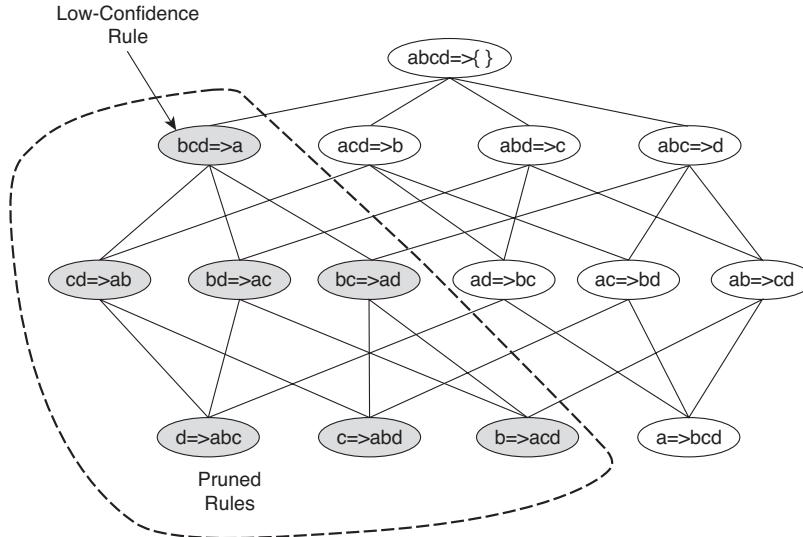


Figure 6.15. Pruning of association rules using the confidence measure.

entire subgraph spanned by the node can be pruned immediately. Suppose the confidence for $\{bcd\} \rightarrow \{a\}$ is low. All the rules containing item a in its consequent, including $\{cd\} \rightarrow \{ab\}$, $\{bd\} \rightarrow \{ac\}$, $\{bc\} \rightarrow \{ad\}$, and $\{d\} \rightarrow \{abc\}$ can be discarded.

A pseudocode for the rule generation step is shown in Algorithms 6.2 and 6.3. Note the similarity between the `ap-genrules` procedure given in Algorithm 6.3 and the frequent itemset generation procedure given in Algorithm 6.1. The only difference is that, in rule generation, we do not have to make additional passes over the data set to compute the confidence of the candidate rules. Instead, we determine the confidence of each rule by using the support counts computed during frequent itemset generation.

Algorithm 6.2 Rule generation of the *Apriori* algorithm.

```

1: for each frequent  $k$ -itemset  $f_k$ ,  $k \geq 2$  do
2:    $H_1 = \{i \mid i \in f_k\}$  {1-item consequents of the rule.}
3:   call ap-genrules( $f_k, H_1.$ )
4: end for
```

Algorithm 6.3 Procedure ap-genrules(f_k, H_m).

```

1:  $k = |f_k|$  {size of frequent itemset.}
2:  $m = |H_m|$  {size of rule consequent.}
3: if  $k > m + 1$  then
4:    $H_{m+1} = \text{apriori-gen}(H_m)$ .
5:   for each  $h_{m+1} \in H_{m+1}$  do
6:      $\text{conf} = \sigma(f_k)/\sigma(f_k - h_{m+1})$ .
7:     if  $\text{conf} \geq \text{minconf}$  then
8:       output the rule  $(f_k - h_{m+1}) \rightarrow h_{m+1}$ .
9:     else
10:      delete  $h_{m+1}$  from  $H_{m+1}$ .
11:    end if
12:   end for
13:   call ap-genrules( $f_k, H_{m+1}$ .)
14: end if

```

6.3.3 An Example: Congressional Voting Records

This section demonstrates the results of applying association analysis to the voting records of members of the United States House of Representatives. The data is obtained from the 1984 Congressional Voting Records Database, which is available at the UCI machine learning data repository. Each transaction contains information about the party affiliation for a representative along with his or her voting record on 16 key issues. There are 435 transactions and 34 items in the data set. The set of items are listed in Table 6.3.

The *Apriori* algorithm is then applied to the data set with $\text{minsup} = 30\%$ and $\text{minconf} = 90\%$. Some of the high-confidence rules extracted by the algorithm are shown in Table 6.4. The first two rules suggest that most of the members who voted yes for aid to El Salvador and no for budget resolution and MX missile are Republicans; while those who voted no for aid to El Salvador and yes for budget resolution and MX missile are Democrats. These high-confidence rules show the key issues that divide members from both political parties. If minconf is reduced, we may find rules that contain issues that cut across the party lines. For example, with $\text{minconf} = 40\%$, the rules suggest that corporation cutbacks is an issue that receives almost equal number of votes from both parties—52.3% of the members who voted no are Republicans, while the remaining 47.7% of them who voted no are Democrats.

Table 6.3. List of binary attributes from the 1984 United States Congressional Voting Records. Source: The UCI machine learning repository.

1. Republican	18. aid to Nicaragua = no
2. Democrat	19. MX-missile = yes
3. handicapped-infants = yes	20. MX-missile = no
4. handicapped-infants = no	21. immigration = yes
5. water project cost sharing = yes	22. immigration = no
6. water project cost sharing = no	23. synfuel corporation cutback = yes
7. budget-resolution = yes	24. synfuel corporation cutback = no
8. budget-resolution = no	25. education spending = yes
9. physician fee freeze = yes	26. education spending = no
10. physician fee freeze = no	27. right-to-sue = yes
11. aid to El Salvador = yes	28. right-to-sue = no
12. aid to El Salvador = no	29. crime = yes
13. religious groups in schools = yes	30. crime = no
14. religious groups in schools = no	31. duty-free-exports = yes
15. anti-satellite test ban = yes	32. duty-free-exports = no
16. anti-satellite test ban = no	33. export administration act = yes
17. aid to Nicaragua = yes	34. export administration act = no

Table 6.4. Association rules extracted from the 1984 United States Congressional Voting Records.

Association Rule	Confidence
{budget resolution = no, MX-missile=no, aid to El Salvador = yes } → {Republican}	91.0%
{budget resolution = yes, MX-missile=yes, aid to El Salvador = no } → {Democrat}	97.5%
{crime = yes, right-to-sue = yes, physician fee freeze = yes} → {Republican}	93.5%
{crime = no, right-to-sue = no, physician fee freeze = no} → {Democrat}	100%

6.4 Compact Representation of Frequent Itemsets

In practice, the number of frequent itemsets produced from a transaction data set can be very large. It is useful to identify a small representative set of itemsets from which all other frequent itemsets can be derived. Two such representations are presented in this section in the form of maximal and closed frequent itemsets.

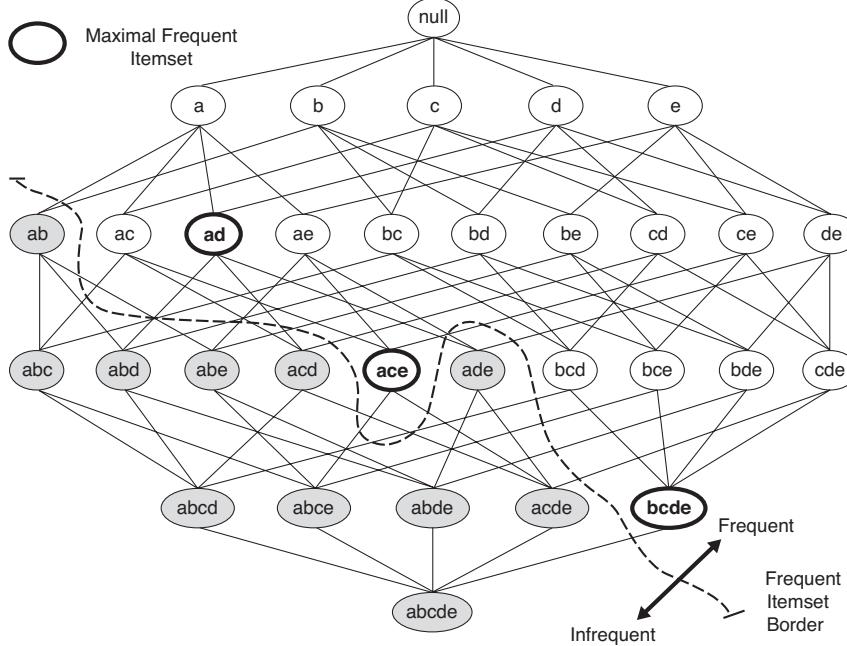


Figure 6.16. Maximal frequent itemset.

6.4.1 Maximal Frequent Itemsets

Definition 6.3 (Maximal Frequent Itemset). A maximal frequent itemset is defined as a frequent itemset for which none of its immediate supersets are frequent.

To illustrate this concept, consider the itemset lattice shown in Figure 6.16. The itemsets in the lattice are divided into two groups: those that are frequent and those that are infrequent. A frequent itemset border, which is represented by a dashed line, is also illustrated in the diagram. Every itemset located above the border is frequent, while those located below the border (the shaded nodes) are infrequent. Among the itemsets residing near the border, $\{a, d\}$, $\{a, c, e\}$, and $\{b, c, d, e\}$ are considered to be maximal frequent itemsets because their immediate supersets are infrequent. An itemset such as $\{a, d\}$ is maximal frequent because all of its immediate supersets, $\{a, b, d\}$, $\{a, c, d\}$, and $\{a, d, e\}$, are infrequent. In contrast, $\{a, c\}$ is non-maximal because one of its immediate supersets, $\{a, c, e\}$, is frequent.

Maximal frequent itemsets effectively provide a compact representation of frequent itemsets. In other words, they form the smallest set of itemsets from

which all frequent itemsets can be derived. For example, the frequent itemsets shown in Figure 6.16 can be divided into two groups:

- Frequent itemsets that begin with item a and that may contain items c , d , or e . This group includes itemsets such as $\{a\}$, $\{a, c\}$, $\{a, d\}$, $\{a, e\}$, and $\{a, c, e\}$.
- Frequent itemsets that begin with items b , c , d , or e . This group includes itemsets such as $\{b\}$, $\{b, c\}$, $\{c, d\}$, $\{b, c, d, e\}$, etc.

Frequent itemsets that belong in the first group are subsets of either $\{a, c, e\}$ or $\{a, d\}$, while those that belong in the second group are subsets of $\{b, c, d, e\}$. Hence, the maximal frequent itemsets $\{a, c, e\}$, $\{a, d\}$, and $\{b, c, d, e\}$ provide a compact representation of the frequent itemsets shown in Figure 6.16.

Maximal frequent itemsets provide a valuable representation for data sets that can produce very long, frequent itemsets, as there are exponentially many frequent itemsets in such data. Nevertheless, this approach is practical only if an efficient algorithm exists to explicitly find the maximal frequent itemsets without having to enumerate all their subsets. We briefly describe one such approach in Section 6.5.

Despite providing a compact representation, maximal frequent itemsets do not contain the support information of their subsets. For example, the support of the maximal frequent itemsets $\{a, c, e\}$, $\{a, d\}$, and $\{b, c, d, e\}$ do not provide any hint about the support of their subsets. An additional pass over the data set is therefore needed to determine the support counts of the non-maximal frequent itemsets. In some cases, it might be desirable to have a minimal representation of frequent itemsets that preserves the support information. We illustrate such a representation in the next section.

6.4.2 Closed Frequent Itemsets

Closed itemsets provide a minimal representation of itemsets without losing their support information. A formal definition of a closed itemset is presented below.

Definition 6.4 (Closed Itemset). An itemset X is closed if none of its immediate supersets has exactly the same support count as X .

Put another way, X is not closed if at least one of its immediate supersets has the same support count as X . Examples of closed itemsets are shown in Figure 6.17. To better illustrate the support count of each itemset, we have associated each node (itemset) in the lattice with a list of its corresponding

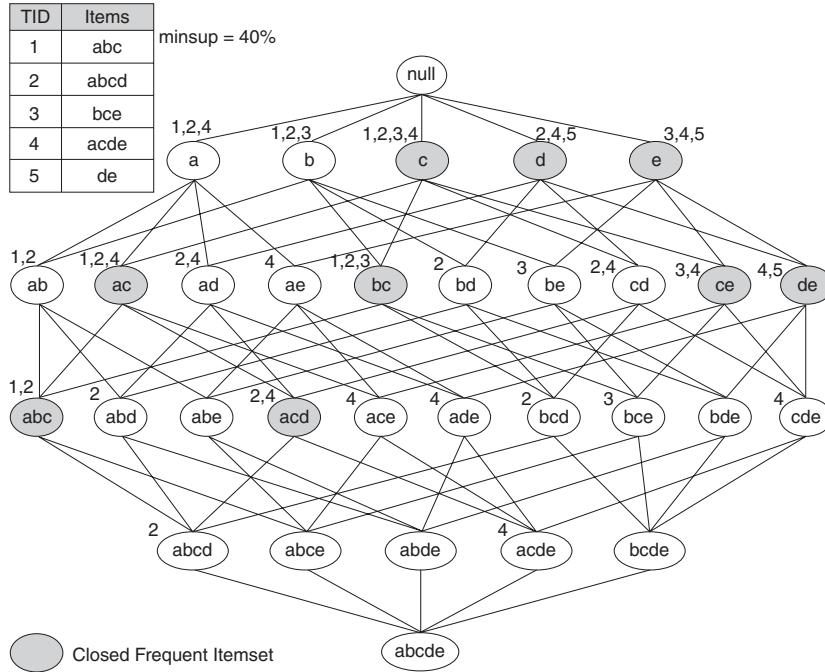


Figure 6.17. An example of the closed frequent itemsets (with minimum support count equal to 40%).

transaction IDs. For example, since the node $\{b, c\}$ is associated with transaction IDs 1, 2, and 3, its support count is equal to three. From the transactions given in this diagram, notice that every transaction that contains b also contains c . Consequently, the support for $\{b\}$ is identical to $\{b, c\}$ and $\{b\}$ should not be considered a closed itemset. Similarly, since c occurs in every transaction that contains both a and d , the itemset $\{a, d\}$ is not closed. On the other hand, $\{b, c\}$ is a closed itemset because it does not have the same support count as any of its supersets.

Definition 6.5 (Closed Frequent Itemset). An itemset is a closed frequent itemset if it is closed and its support is greater than or equal to $minsup$.

In the previous example, assuming that the support threshold is 40%, $\{b, c\}$ is a closed frequent itemset because its support is 60%. The rest of the closed frequent itemsets are indicated by the shaded nodes.

Algorithms are available to explicitly extract closed frequent itemsets from a given data set. Interested readers may refer to the bibliographic notes at the end of this chapter for further discussions of these algorithms. We can use the closed frequent itemsets to determine the support counts for the non-closed

Algorithm 6.4 Support counting using closed frequent itemsets.

```

1: Let  $C$  denote the set of closed frequent itemsets
2: Let  $k_{\max}$  denote the maximum size of closed frequent itemsets
3:  $F_{k_{\max}} = \{f | f \in C, |f| = k_{\max}\}$  {Find all frequent itemsets of size  $k_{\max}$ .}
4: for  $k = k_{\max} - 1$  downto 1 do
5:    $F_k = \{f | f \subset F_{k+1}, |f| = k\}$  {Find all frequent itemsets of size  $k$ .}
6:   for each  $f \in F_k$  do
7:     if  $f \notin C$  then
8:        $f.support = \max\{f'.support | f' \in F_{k+1}, f \subset f'\}$ 
9:     end if
10:   end for
11: end for

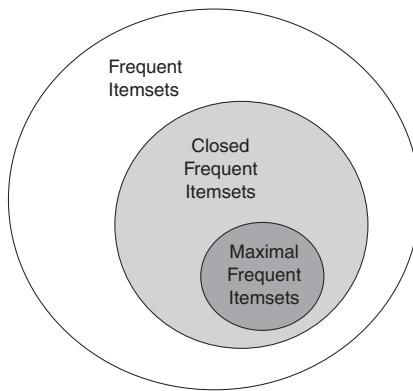
```

frequent itemsets. For example, consider the frequent itemset $\{a, d\}$ shown in Figure 6.17. Because the itemset is not closed, its support count must be identical to one of its immediate supersets. The key is to determine which superset (among $\{a, b, d\}$, $\{a, c, d\}$, or $\{a, d, e\}$) has exactly the same support count as $\{a, d\}$. The *Apriori* principle states that any transaction that contains the superset of $\{a, d\}$ must also contain $\{a, d\}$. However, any transaction that contains $\{a, d\}$ does not have to contain the supersets of $\{a, d\}$. For this reason, the support for $\{a, d\}$ must be equal to the largest support among its supersets. Since $\{a, c, d\}$ has a larger support than both $\{a, b, d\}$ and $\{a, d, e\}$, the support for $\{a, d\}$ must be identical to the support for $\{a, c, d\}$. Using this methodology, an algorithm can be developed to compute the support for the non-closed frequent itemsets. The pseudocode for this algorithm is shown in Algorithm 6.4. The algorithm proceeds in a specific-to-general fashion, i.e., from the largest to the smallest frequent itemsets. This is because, in order to find the support for a non-closed frequent itemset, the support for all of its supersets must be known.

To illustrate the advantage of using closed frequent itemsets, consider the data set shown in Table 6.5, which contains ten transactions and fifteen items. The items can be divided into three groups: (1) Group A , which contains items a_1 through a_5 ; (2) Group B , which contains items b_1 through b_5 ; and (3) Group C , which contains items c_1 through c_5 . Note that items within each group are perfectly associated with each other and they do not appear with items from another group. Assuming the support threshold is 20%, the total number of frequent itemsets is $3 \times (2^5 - 1) = 93$. However, there are only three closed frequent itemsets in the data: ($\{a_1, a_2, a_3, a_4, a_5\}$, $\{b_1, b_2, b_3, b_4, b_5\}$, and $\{c_1, c_2, c_3, c_4, c_5\}$). It is often sufficient to present only the closed frequent itemsets to the analysts instead of the entire set of frequent itemsets.

Table 6.5. A transaction data set for mining closed itemsets.

TID	a_1	a_2	a_3	a_4	a_5	b_1	b_2	b_3	b_4	b_5	c_1	c_2	c_3	c_4	c_5
1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
2	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
3	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0
5	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0
6	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
8	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
9	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
10	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1

**Figure 6.18.** Relationships among frequent, maximal frequent, and closed frequent itemsets.

Closed frequent itemsets are useful for removing some of the redundant association rules. An association rule $X \rightarrow Y$ is redundant if there exists another rule $X' \rightarrow Y'$, where X is a subset of X' and Y is a subset of Y' , such that the support and confidence for both rules are identical. In the example shown in Figure 6.17, $\{b\}$ is not a closed frequent itemset while $\{b, c\}$ is closed. The association rule $\{b\} \rightarrow \{d, e\}$ is therefore redundant because it has the same support and confidence as $\{b, c\} \rightarrow \{d, e\}$. Such redundant rules are not generated if closed frequent itemsets are used for rule generation.

Finally, note that all maximal frequent itemsets are closed because none of the maximal frequent itemsets can have the same support count as their immediate supersets. The relationships among frequent, maximal frequent, and closed frequent itemsets are shown in Figure 6.18.

6.5 Alternative Methods for Generating Frequent Itemsets

Apriori is one of the earliest algorithms to have successfully addressed the combinatorial explosion of frequent itemset generation. It achieves this by applying the *Apriori* principle to prune the exponential search space. Despite its significant performance improvement, the algorithm still incurs considerable I/O overhead since it requires making several passes over the transaction data set. In addition, as noted in Section 6.2.5, the performance of the *Apriori* algorithm may degrade significantly for dense data sets because of the increasing width of transactions. Several alternative methods have been developed to overcome these limitations and improve upon the efficiency of the *Apriori* algorithm. The following is a high-level description of these methods.

Traversal of Itemset Lattice A search for frequent itemsets can be conceptually viewed as a traversal on the itemset lattice shown in Figure 6.1. The search strategy employed by an algorithm dictates how the lattice structure is traversed during the frequent itemset generation process. Some search strategies are better than others, depending on the configuration of frequent itemsets in the lattice. An overview of these strategies is presented next.

- **General-to-Specific versus Specific-to-General:** The *Apriori* algorithm uses a general-to-specific search strategy, where pairs of frequent $(k-1)$ -itemsets are merged to obtain candidate k -itemsets. This general-to-specific search strategy is effective, provided the maximum length of a frequent itemset is not too long. The configuration of frequent itemsets that works best with this strategy is shown in Figure 6.19(a), where the darker nodes represent infrequent itemsets. Alternatively, a specific-to-general search strategy looks for more specific frequent itemsets first, before finding the more general frequent itemsets. This strategy is useful to discover maximal frequent itemsets in dense transactions, where the frequent itemset border is located near the bottom of the lattice, as shown in Figure 6.19(b). The *Apriori* principle can be applied to prune all subsets of maximal frequent itemsets. Specifically, if a candidate k -itemset is maximal frequent, we do not have to examine any of its subsets of size $k-1$. However, if the candidate k -itemset is infrequent, we need to check all of its $k-1$ subsets in the next iteration. Another approach is to combine both general-to-specific and specific-to-general search strategies. This bidirectional approach requires more space to

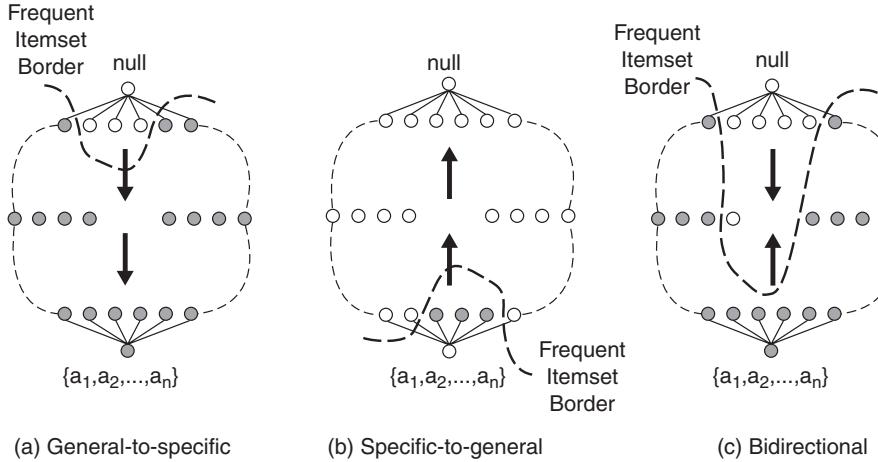


Figure 6.19. General-to-specific, specific-to-general, and bidirectional search.

store the candidate itemsets, but it can help to rapidly identify the frequent itemset border, given the configuration shown in Figure 6.19(c).

- **Equivalence Classes:** Another way to envision the traversal is to first partition the lattice into disjoint groups of nodes (or equivalence classes). A frequent itemset generation algorithm searches for frequent itemsets within a particular equivalence class first before moving to another equivalence class. As an example, the level-wise strategy used in the *Apriori* algorithm can be considered to be partitioning the lattice on the basis of itemset sizes; i.e., the algorithm discovers all frequent 1-itemsets first before proceeding to larger-sized itemsets. Equivalence classes can also be defined according to the prefix or suffix labels of an itemset. In this case, two itemsets belong to the same equivalence class if they share a common prefix or suffix of length k . In the prefix-based approach, the algorithm can search for frequent itemsets starting with the prefix a before looking for those starting with prefixes b , c , and so on. Both prefix-based and suffix-based equivalence classes can be demonstrated using the tree-like structure shown in Figure 6.20.
- **Breadth-First versus Depth-First:** The *Apriori* algorithm traverses the lattice in a breadth-first manner, as shown in Figure 6.21(a). It first discovers all the frequent 1-itemsets, followed by the frequent 2-itemsets, and so on, until no new frequent itemsets are generated. The itemset

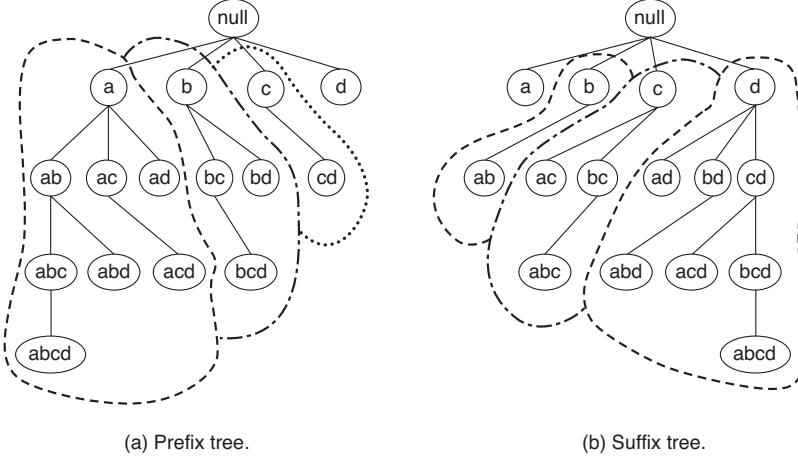


Figure 6.20. Equivalence classes based on the prefix and suffix labels of itemsets.

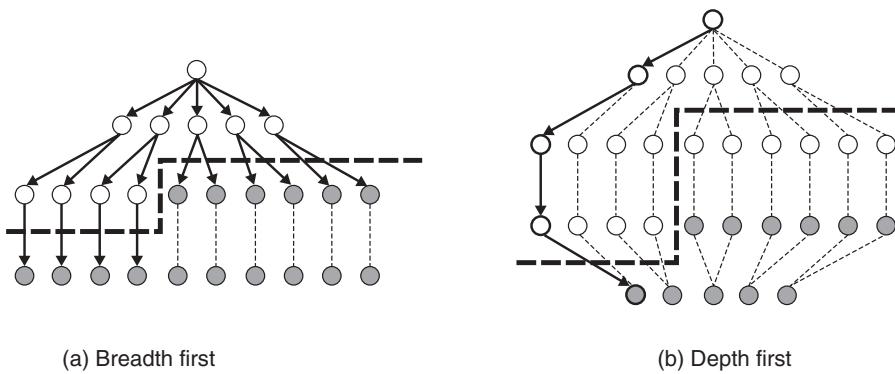


Figure 6.21. Breadth-first and depth-first traversals.

lattice can also be traversed in a depth-first manner, as shown in Figures 6.21(b) and 6.22. The algorithm can start from, say, node a in Figure 6.22, and count its support to determine whether it is frequent. If so, the algorithm progressively expands the next level of nodes, i.e., ab , abc , and so on, until an infrequent node is reached, say, $abcd$. It then backtracks to another branch, say, $abce$, and continues the search from there.

The depth-first approach is often used by algorithms designed to find maximal frequent itemsets. This approach allows the frequent itemset border to be detected more quickly than using a breadth-first approach. Once a maximal frequent itemset is found, substantial pruning can be

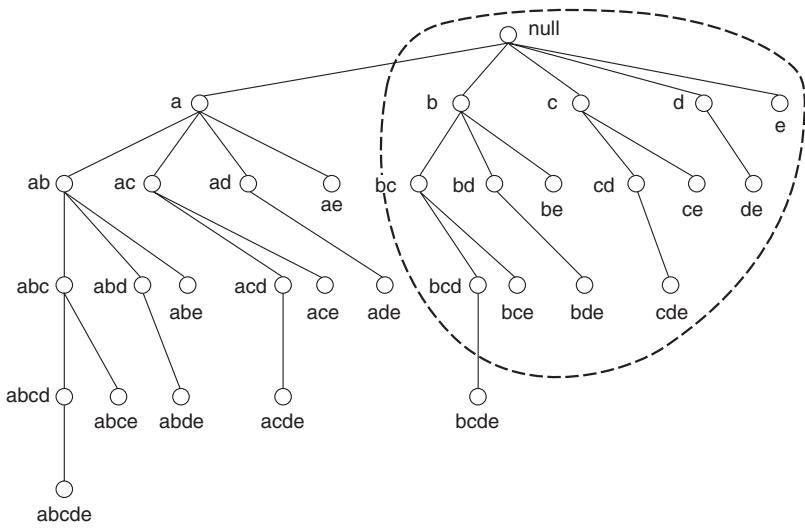


Figure 6.22. Generating candidate itemsets using the depth-first approach.

performed on its subsets. For example, if the node $bcde$ shown in Figure 6.22 is maximal frequent, then the algorithm does not have to visit the subtrees rooted at bd , be , c , d , and e because they will not contain any maximal frequent itemsets. However, if abc is maximal frequent, only the nodes such as ac and bc are not maximal frequent (but the subtrees of ac and bc may still contain maximal frequent itemsets). The depth-first approach also allows a different kind of pruning based on the support of itemsets. For example, suppose the support for $\{a, b, c\}$ is identical to the support for $\{a, b\}$. The subtrees rooted at abd and abe can be skipped because they are guaranteed not to have any maximal frequent itemsets. The proof of this is left as an exercise to the readers.

Representation of Transaction Data Set There are many ways to represent a transaction data set. The choice of representation can affect the I/O costs incurred when computing the support of candidate itemsets. Figure 6.23 shows two different ways of representing market basket transactions. The representation on the left is called a **horizontal** data layout, which is adopted by many association rule mining algorithms, including *Apriori*. Another possibility is to store the list of transaction identifiers (TID-list) associated with each item. Such a representation is known as the **vertical** data layout. The support for each candidate itemset is obtained by intersecting the TID-lists of its subset items. The length of the TID-lists shrinks as we progress to larger

Horizontal Data Layout		Vertical Data Layout				
TID	Items	a	b	c	d	e
1	a,b,e	1	1	2	2	1
2	b,c,d	4	2	3	4	3
3	c,e	5	5	4	5	6
4	a,c,d	6	7	8	9	
5	a,b,c,d	7	8	9		
6	a,e	8	10			
7	a,b	9				
8	a,b,c					
9	a,c,d					
10	b					

Figure 6.23. Horizontal and vertical data format.

sized itemsets. However, one problem with this approach is that the initial set of TID-lists may be too large to fit into main memory, thus requiring more sophisticated techniques to compress the TID-lists. We describe another effective approach to represent the data in the next section.

6.6 FP-Growth Algorithm

This section presents an alternative algorithm called **FP-growth** that takes a radically different approach to discovering frequent itemsets. The algorithm does not subscribe to the generate-and-test paradigm of *Apriori*. Instead, it encodes the data set using a compact data structure called an **FP-tree** and extracts frequent itemsets directly from this structure. The details of this approach are presented next.

6.6.1 FP-Tree Representation

An FP-tree is a compressed representation of the input data. It is constructed by reading the data set one transaction at a time and mapping each transaction onto a path in the FP-tree. As different transactions can have several items in common, their paths may overlap. The more the paths overlap with one another, the more compression we can achieve using the FP-tree structure. If the size of the FP-tree is small enough to fit into main memory, this will allow us to extract frequent itemsets directly from the structure in memory instead of making repeated passes over the data stored on disk.

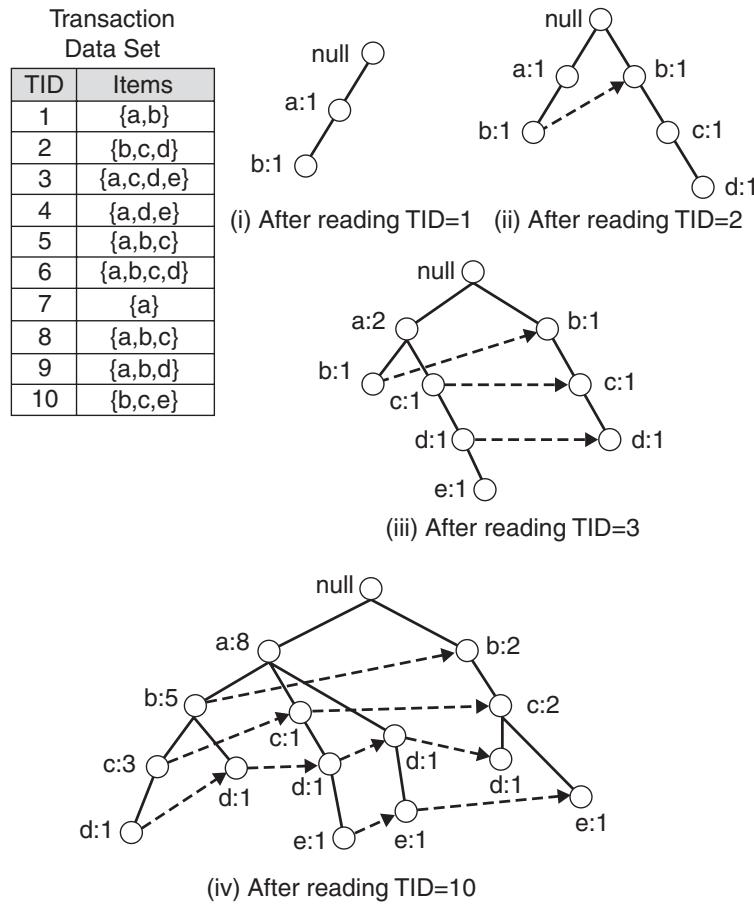


Figure 6.24. Construction of an FP-tree.

Figure 6.24 shows a data set that contains ten transactions and five items. The structures of the FP-tree after reading the first three transactions are also depicted in the diagram. Each node in the tree contains the label of an item along with a counter that shows the number of transactions mapped onto the given path. Initially, the FP-tree contains only the root node represented by the `null` symbol. The FP-tree is subsequently extended in the following way:

1. The data set is scanned once to determine the support count of each item. Infrequent items are discarded, while the frequent items are sorted in decreasing support counts. For the data set shown in Figure 6.24, *a* is the most frequent item, followed by *b*, *c*, *d*, and *e*.

2. The algorithm makes a second pass over the data to construct the FP-tree. After reading the first transaction, $\{a, b\}$, the nodes labeled as a and b are created. A path is then formed from $\text{null} \rightarrow a \rightarrow b$ to encode the transaction. Every node along the path has a frequency count of 1.
3. After reading the second transaction, $\{b, c, d\}$, a new set of nodes is created for items b , c , and d . A path is then formed to represent the transaction by connecting the nodes $\text{null} \rightarrow b \rightarrow c \rightarrow d$. Every node along this path also has a frequency count equal to one. Although the first two transactions have an item in common, which is b , their paths are disjoint because the transactions do not share a common prefix.
4. The third transaction, $\{a, c, d, e\}$, shares a common prefix item (which is a) with the first transaction. As a result, the path for the third transaction, $\text{null} \rightarrow a \rightarrow c \rightarrow d \rightarrow e$, overlaps with the path for the first transaction, $\text{null} \rightarrow a \rightarrow b$. Because of their overlapping path, the frequency count for node a is incremented to two, while the frequency counts for the newly created nodes, c , d , and e , are equal to one.
5. This process continues until every transaction has been mapped onto one of the paths given in the FP-tree. The resulting FP-tree after reading all the transactions is shown at the bottom of Figure 6.24.

The size of an FP-tree is typically smaller than the size of the uncompressed data because many transactions in market basket data often share a few items in common. In the best-case scenario, where all the transactions have the same set of items, the FP-tree contains only a single branch of nodes. The worst-case scenario happens when every transaction has a unique set of items. As none of the transactions have any items in common, the size of the FP-tree is effectively the same as the size of the original data. However, the physical storage requirement for the FP-tree is higher because it requires additional space to store pointers between nodes and counters for each item.

The size of an FP-tree also depends on how the items are ordered. If the ordering scheme in the preceding example is reversed, i.e., from lowest to highest support item, the resulting FP-tree is shown in Figure 6.25. The tree appears to be denser because the branching factor at the root node has increased from 2 to 5 and the number of nodes containing the high support items such as a and b has increased from 3 to 12. Nevertheless, ordering by decreasing support counts does not always lead to the smallest tree. For example, suppose we augment the data set given in Figure 6.24 with 100 transactions that contain $\{e\}$, 80 transactions that contain $\{d\}$, 60 transactions

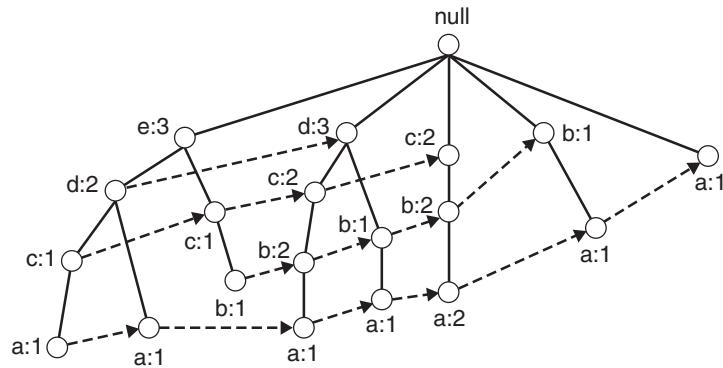


Figure 6.25. An FP-tree representation for the data set shown in Figure 6.24 with a different item ordering scheme.

that contain $\{c\}$, and 40 transactions that contain $\{b\}$. Item e is now most frequent, followed by d , c , b , and a . With the augmented transactions, ordering by decreasing support counts will result in an FP-tree similar to Figure 6.25, while a scheme based on increasing support counts produces a smaller FP-tree similar to Figure 6.24(iv).

An FP-tree also contains a list of pointers connecting between nodes that have the same items. These pointers, represented as dashed lines in Figures 6.24 and 6.25, help to facilitate the rapid access of individual items in the tree. We explain how to use the FP-tree and its corresponding pointers for frequent itemset generation in the next section.

6.6.2 Frequent Itemset Generation in FP-Growth Algorithm

FP-growth is an algorithm that generates frequent itemsets from an FP-tree by exploring the tree in a bottom-up fashion. Given the example tree shown in Figure 6.24, the algorithm looks for frequent itemsets ending in e first, followed by d , c , b , and finally, a . This bottom-up strategy for finding frequent itemsets ending with a particular item is equivalent to the suffix-based approach described in Section 6.5. Since every transaction is mapped onto a path in the FP-tree, we can derive the frequent itemsets ending with a particular item, say, e , by examining only the paths containing node e . These paths can be accessed rapidly using the pointers associated with node e . The extracted paths are shown in Figure 6.26(a). The details on how to process the paths to obtain frequent itemsets will be explained later.

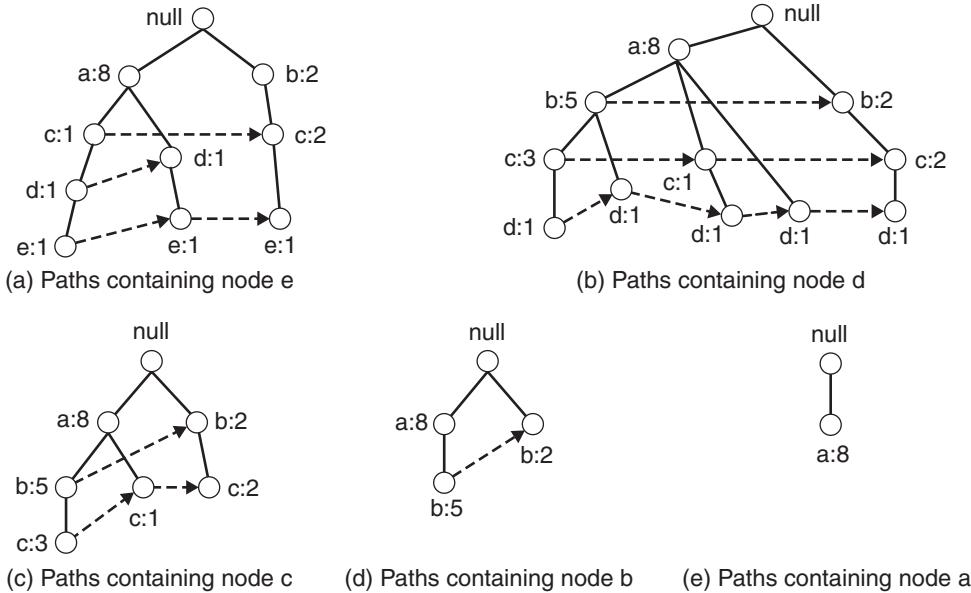


Figure 6.26. Decomposing the frequent itemset generation problem into multiple subproblems, where each subproblem involves finding frequent itemsets ending in e , d , c , b , and a .

Table 6.6. The list of frequent itemsets ordered by their corresponding suffixes.

Suffix	Frequent Itemsets
e	$\{e\}$, $\{d,e\}$, $\{a,d,e\}$, $\{c,e\}$, $\{a,e\}$
d	$\{d\}$, $\{c,d\}$, $\{b,c,d\}$, $\{a,c,d\}$, $\{b,d\}$, $\{a,b,d\}$, $\{a,d\}$
c	$\{c\}$, $\{b,c\}$, $\{a,b,c\}$, $\{a,c\}$
b	$\{b\}$, $\{a,b\}$
a	$\{a\}$

After finding the frequent itemsets ending in e , the algorithm proceeds to look for frequent itemsets ending in d by processing the paths associated with node d . The corresponding paths are shown in Figure 6.26(b). This process continues until all the paths associated with nodes c , b , and finally a , are processed. The paths for these items are shown in Figures 6.26(c), (d), and (e), while their corresponding frequent itemsets are summarized in Table 6.6.

FP-growth finds all the frequent itemsets ending with a particular suffix by employing a divide-and-conquer strategy to split the problem into smaller subproblems. For example, suppose we are interested in finding all frequent

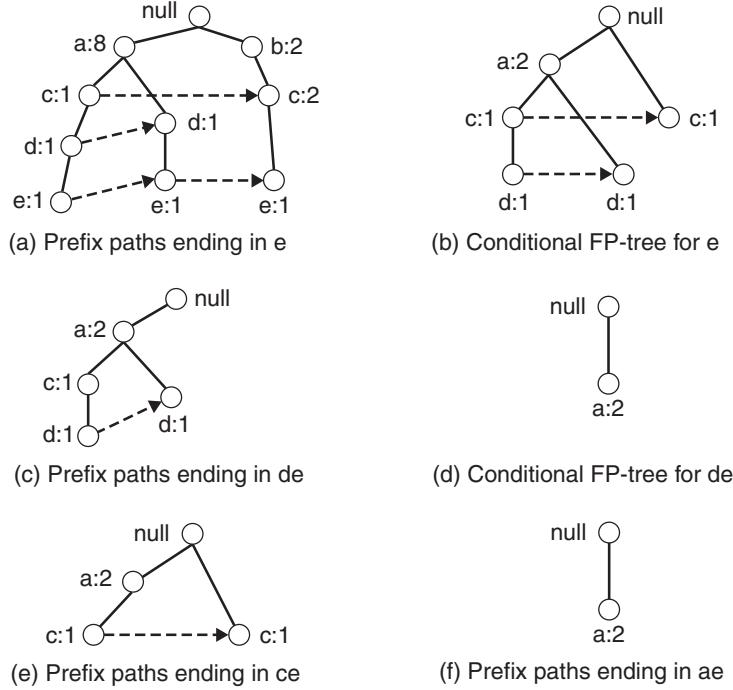


Figure 6.27. Example of applying the FP-growth algorithm to find frequent itemsets ending in e .

itemsets ending in e . To do this, we must first check whether the itemset $\{e\}$ itself is frequent. If it is frequent, we consider the subproblem of finding frequent itemsets ending in de , followed by ce , be , and ae . In turn, each of these subproblems are further decomposed into smaller subproblems. By merging the solutions obtained from the subproblems, all the frequent itemsets ending in e can be found. This divide-and-conquer approach is the key strategy employed by the FP-growth algorithm.

For a more concrete example on how to solve the subproblems, consider the task of finding frequent itemsets ending with e .

1. The first step is to gather all the paths containing node e . These initial paths are called **prefix paths** and are shown in Figure 6.27(a).
2. From the prefix paths shown in Figure 6.27(a), the support count for e is obtained by adding the support counts associated with node e . Assuming that the minimum support count is 2, $\{e\}$ is declared a frequent itemset because its support count is 3.

3. Because $\{e\}$ is frequent, the algorithm has to solve the subproblems of finding frequent itemsets ending in de , ce , be , and ae . Before solving these subproblems, it must first convert the prefix paths into a **conditional FP-tree**, which is structurally similar to an FP-tree, except it is used to find frequent itemsets ending with a particular suffix. A conditional FP-tree is obtained in the following way:
 - (a) First, the support counts along the prefix paths must be updated because some of the counts include transactions that do not contain item e . For example, the rightmost path shown in Figure 6.27(a), $\text{null} \rightarrow b:2 \rightarrow c:2 \rightarrow e:1$, includes a transaction $\{b, c\}$ that does not contain item e . The counts along the prefix path must therefore be adjusted to 1 to reflect the actual number of transactions containing $\{b, c, e\}$.
 - (b) The prefix paths are truncated by removing the nodes for e . These nodes can be removed because the support counts along the prefix paths have been updated to reflect only transactions that contain e and the subproblems of finding frequent itemsets ending in de , ce , be , and ae no longer need information about node e .
 - (c) After updating the support counts along the prefix paths, some of the items may no longer be frequent. For example, the node b appears only once and has a support count equal to 1, which means that there is only one transaction that contains both b and e . Item b can be safely ignored from subsequent analysis because all itemsets ending in be must be infrequent.

The conditional FP-tree for e is shown in Figure 6.27(b). The tree looks different than the original prefix paths because the frequency counts have been updated and the nodes b and e have been eliminated.

4. FP-growth uses the conditional FP-tree for e to solve the subproblems of finding frequent itemsets ending in de , ce , and ae . To find the frequent itemsets ending in de , the prefix paths for d are gathered from the conditional FP-tree for e (Figure 6.27(c)). By adding the frequency counts associated with node d , we obtain the support count for $\{d, e\}$. Since the support count is equal to 2, $\{d, e\}$ is declared a frequent itemset. Next, the algorithm constructs the conditional FP-tree for de using the approach described in step 3. After updating the support counts and removing the infrequent item c , the conditional FP-tree for de is shown in Figure 6.27(d). Since the conditional FP-tree contains only one item,

a , whose support is equal to $minsup$, the algorithm extracts the frequent itemset $\{a, d, e\}$ and moves on to the next subproblem, which is to generate frequent itemsets ending in ce . After processing the prefix paths for c , only $\{c, e\}$ is found to be frequent. The algorithm proceeds to solve the next subprogram and found $\{a, e\}$ to be the only frequent itemset remaining.

This example illustrates the divide-and-conquer approach used in the FP-growth algorithm. At each recursive step, a conditional FP-tree is constructed by updating the frequency counts along the prefix paths and removing all infrequent items. Because the subproblems are disjoint, FP-growth will not generate any duplicate itemsets. In addition, the counts associated with the nodes allow the algorithm to perform support counting while generating the common suffix itemsets.

FP-growth is an interesting algorithm because it illustrates how a compact representation of the transaction data set helps to efficiently generate frequent itemsets. In addition, for certain transaction data sets, FP-growth outperforms the standard *Apriori* algorithm by several orders of magnitude. The run-time performance of FP-growth depends on the **compaction factor** of the data set. If the resulting conditional FP-trees are very bushy (in the worst case, a full prefix tree), then the performance of the algorithm degrades significantly because it has to generate a large number of subproblems and merge the results returned by each subproblem.

6.7 Evaluation of Association Patterns

Association analysis algorithms have the potential to generate a large number of patterns. For example, although the data set shown in Table 6.1 contains only six items, it can produce up to hundreds of association rules at certain support and confidence thresholds. As the size and dimensionality of real commercial databases can be very large, we could easily end up with thousands or even millions of patterns, many of which might not be interesting. Sifting through the patterns to identify the most interesting ones is not a trivial task because “one person’s trash might be another person’s treasure.” It is therefore important to establish a set of well-accepted criteria for evaluating the quality of association patterns.

The first set of criteria can be established through statistical arguments. Patterns that involve a set of mutually independent items or cover very few transactions are considered uninteresting because they may capture spurious relationships in the data. Such patterns can be eliminated by applying an

objective interestingness measure that uses statistics derived from data to determine whether a pattern is interesting. Examples of objective interestingness measures include support, confidence, and correlation.

The second set of criteria can be established through subjective arguments. A pattern is considered subjectively uninteresting unless it reveals unexpected information about the data or provides useful knowledge that can lead to profitable actions. For example, the rule $\{Butter\} \rightarrow \{Bread\}$ may not be interesting, despite having high support and confidence values, because the relationship represented by the rule may seem rather obvious. On the other hand, the rule $\{Diapers\} \rightarrow \{Beer\}$ is interesting because the relationship is quite unexpected and may suggest a new cross-selling opportunity for retailers. Incorporating subjective knowledge into pattern evaluation is a difficult task because it requires a considerable amount of prior information from the domain experts.

The following are some of the approaches for incorporating subjective knowledge into the pattern discovery task.

Visualization This approach requires a user-friendly environment to keep the human user in the loop. It also allows the domain experts to interact with the data mining system by interpreting and verifying the discovered patterns.

Template-based approach This approach allows the users to constrain the type of patterns extracted by the mining algorithm. Instead of reporting all the extracted rules, only rules that satisfy a user-specified template are returned to the users.

Subjective interestingness measure A subjective measure can be defined based on domain information such as concept hierarchy (to be discussed in Section 7.3) or profit margin of items. The measure can then be used to filter patterns that are obvious and non-actionable.

Readers interested in subjective interestingness measures may refer to resources listed in the bibliography at the end of this chapter.

6.7.1 Objective Measures of Interestingness

An objective measure is a data-driven approach for evaluating the quality of association patterns. It is domain-independent and requires minimal input from the users, other than to specify a threshold for filtering low-quality patterns. An objective measure is usually computed based on the frequency

Table 6.7. A 2-way contingency table for variables A and B .

	B	\bar{B}	
A	f_{11}	f_{10}	f_{1+}
\bar{A}	f_{01}	f_{00}	f_{0+}
	f_{+1}	f_{+0}	N

counts tabulated in a **contingency table**. Table 6.7 shows an example of a contingency table for a pair of binary variables, A and B . We use the notation \bar{A} (\bar{B}) to indicate that A (B) is absent from a transaction. Each entry f_{ij} in this 2×2 table denotes a frequency count. For example, f_{11} is the number of times A and B appear together in the same transaction, while f_{01} is the number of transactions that contain B but not A . The row sum f_{1+} represents the support count for A , while the column sum f_{+1} represents the support count for B . Finally, even though our discussion focuses mainly on asymmetric binary variables, note that contingency tables are also applicable to other attribute types such as symmetric binary, nominal, and ordinal variables.

Limitations of the Support-Confidence Framework Existing association rule mining formulation relies on the support and confidence measures to eliminate uninteresting patterns. The drawback of support was previously described in Section 6.8, in which many potentially interesting patterns involving low support items might be eliminated by the support threshold. The drawback of confidence is more subtle and is best demonstrated with the following example.

Example 6.3. Suppose we are interested in analyzing the relationship between people who drink tea and coffee. We may gather information about the beverage preferences among a group of people and summarize their responses into a table such as the one shown in Table 6.8.

Table 6.8. Beverage preferences among a group of 1000 people.

	<i>Coffee</i>	$\bar{\text{Coffee}}$	
<i>Tea</i>	150	50	200
$\bar{\text{Tea}}$	650	150	800
	800	200	1000

The information given in this table can be used to evaluate the association rule $\{Tea\} \rightarrow \{Coffee\}$. At first glance, it may appear that people who drink tea also tend to drink coffee because the rule's support (15%) and confidence (75%) values are reasonably high. This argument would have been acceptable except that the fraction of people who drink coffee, regardless of whether they drink tea, is 80%, while the fraction of tea drinkers who drink coffee is only 75%. Thus knowing that a person is a tea drinker actually decreases her probability of being a coffee drinker from 80% to 75%! The rule $\{Tea\} \rightarrow \{Coffee\}$ is therefore misleading despite its high confidence value. ■

The pitfall of confidence can be traced to the fact that the measure ignores the support of the itemset in the rule consequent. Indeed, if the support of coffee drinkers is taken into account, we would not be surprised to find that many of the people who drink tea also drink coffee. What is more surprising is that the fraction of tea drinkers who drink coffee is actually less than the overall fraction of people who drink coffee, which points to an inverse relationship between tea drinkers and coffee drinkers.

Because of the limitations in the support-confidence framework, various objective measures have been used to evaluate the quality of association patterns. Below, we provide a brief description of these measures and explain some of their strengths and limitations.

Interest Factor The tea-coffee example shows that high-confidence rules can sometimes be misleading because the confidence measure ignores the support of the itemset appearing in the rule consequent. One way to address this problem is by applying a metric known as **lift**:

$$Lift = \frac{c(A \rightarrow B)}{s(B)}, \quad (6.4)$$

which computes the ratio between the rule's confidence and the support of the itemset in the rule consequent. For binary variables, lift is equivalent to another objective measure called **interest factor**, which is defined as follows:

$$I(A, B) = \frac{s(A, B)}{s(A) \times s(B)} = \frac{Nf_{11}}{f_{1+}f_{+1}}. \quad (6.5)$$

Interest factor compares the frequency of a pattern against a baseline frequency computed under the statistical independence assumption. The baseline frequency for a pair of mutually independent variables is

$$\frac{f_{11}}{N} = \frac{f_{1+}}{N} \times \frac{f_{+1}}{N}, \quad \text{or equivalently, } f_{11} = \frac{f_{1+}f_{+1}}{N}. \quad (6.6)$$

Table 6.9. Contingency tables for the word pairs $\{p,q\}$ and $\{r,s\}$.

	p	\bar{p}	
q	880	50	930
\bar{q}	50	20	70
	930	70	1000

	r	\bar{r}	
s	20	50	70
\bar{s}	50	880	930
	70	930	1000

This equation follows from the standard approach of using simple fractions as estimates for probabilities. The fraction f_{11}/N is an estimate for the joint probability $P(A, B)$, while f_{1+}/N and f_{+1}/N are the estimates for $P(A)$ and $P(B)$, respectively. If A and B are statistically independent, then $P(A, B) = P(A) \times P(B)$, thus leading to the formula shown in Equation 6.6. Using Equations 6.5 and 6.6, we can interpret the measure as follows:

$$I(A, B) \begin{cases} = 1, & \text{if } A \text{ and } B \text{ are independent;} \\ > 1, & \text{if } A \text{ and } B \text{ are positively correlated;} \\ < 1, & \text{if } A \text{ and } B \text{ are negatively correlated.} \end{cases} \quad (6.7)$$

For the tea-coffee example shown in Table 6.8, $I = \frac{0.15}{0.2 \times 0.8} = 0.9375$, thus suggesting a slight negative correlation between tea drinkers and coffee drinkers.

Limitations of Interest Factor We illustrate the limitation of interest factor with an example from the text mining domain. In the text domain, it is reasonable to assume that the association between a pair of words depends on the number of documents that contain both words. For example, because of their stronger association, we expect the words `data` and `mining` to appear together more frequently than the words `compiler` and `mining` in a collection of computer science articles.

Table 6.9 shows the frequency of occurrences between two pairs of words, $\{p, q\}$ and $\{r, s\}$. Using the formula given in Equation 6.5, the interest factor for $\{p, q\}$ is 1.02 and for $\{r, s\}$ is 4.08. These results are somewhat troubling for the following reasons. Although p and q appear together in 88% of the documents, their interest factor is close to 1, which is the value when p and q are statistically independent. On the other hand, the interest factor for $\{r, s\}$ is higher than $\{p, q\}$ even though r and s seldom appear together in the same document. Confidence is perhaps the better choice in this situation because it considers the association between p and q (94.6%) to be much stronger than that between r and s (28.6%).

Correlation Analysis Correlation analysis is a statistical-based technique for analyzing relationships between a pair of variables. For continuous variables, correlation is defined using Pearson's correlation coefficient (see Equation 2.10 on page 77). For binary variables, correlation can be measured using the ϕ -coefficient, which is defined as

$$\phi = \frac{f_{11}f_{00} - f_{01}f_{10}}{\sqrt{f_{1+}f_{+1}f_{0+}f_{+0}}}. \quad (6.8)$$

The value of correlation ranges from -1 (perfect negative correlation) to $+1$ (perfect positive correlation). If the variables are statistically independent, then $\phi = 0$. For example, the correlation between the tea and coffee drinkers given in Table 6.8 is -0.0625 .

Limitations of Correlation Analysis The drawback of using correlation can be seen from the word association example given in Table 6.9. Although the words p and q appear together more often than r and s , their ϕ -coefficients are identical, i.e., $\phi(p, q) = \phi(r, s) = 0.232$. This is because the ϕ -coefficient gives equal importance to both co-presence and co-absence of items in a transaction. It is therefore more suitable for analyzing symmetric binary variables. Another limitation of this measure is that it does not remain invariant when there are proportional changes to the sample size. This issue will be discussed in greater detail when we describe the properties of objective measures on page 377.

IS Measure IS is an alternative measure that has been proposed for handling asymmetric binary variables. The measure is defined as follows:

$$IS(A, B) = \sqrt{I(A, B) \times s(A, B)} = \frac{s(A, B)}{\sqrt{s(A)s(B)}}. \quad (6.9)$$

Note that IS is large when the interest factor and support of the pattern are large. For example, the value of IS for the word pairs $\{p, q\}$ and $\{r, s\}$ shown in Table 6.9 are 0.946 and 0.286, respectively. Contrary to the results given by interest factor and the ϕ -coefficient, the IS measure suggests that the association between $\{p, q\}$ is stronger than $\{r, s\}$, which agrees with what we expect from word associations in documents.

It is possible to show that IS is mathematically equivalent to the cosine measure for binary variables (see Equation 2.7 on page 75). In this regard, we

Table 6.10. Example of a contingency table for items p and q .

	q	\bar{q}	
p	800	100	900
\bar{p}	100	0	100
	900	100	1000

consider \mathbf{A} and \mathbf{B} as a pair of bit vectors, $\mathbf{A} \bullet \mathbf{B} = s(A, B)$ the dot product between the vectors, and $|\mathbf{A}| = \sqrt{s(A)}$ the magnitude of vector \mathbf{A} . Therefore:

$$IS(A, B) = \frac{s(A, B)}{\sqrt{s(A) \times s(B)}} = \frac{\mathbf{A} \bullet \mathbf{B}}{|\mathbf{A}| \times |\mathbf{B}|} = \text{cosine}(\mathbf{A}, \mathbf{B}). \quad (6.10)$$

The IS measure can also be expressed as the geometric mean between the confidence of association rules extracted from a pair of binary variables:

$$IS(A, B) = \sqrt{\frac{s(A, B)}{s(A)} \times \frac{s(A, B)}{s(B)}} = \sqrt{c(A \rightarrow B) \times c(B \rightarrow A)}. \quad (6.11)$$

Because the geometric mean between any two numbers is always closer to the smaller number, the IS value of an itemset $\{p, q\}$ is low whenever one of its rules, $p \rightarrow q$ or $q \rightarrow p$, has low confidence.

Limitations of IS Measure The IS value for a pair of independent itemsets, A and B , is

$$IS_{\text{indep}}(A, B) = \frac{s(A, B)}{\sqrt{s(A) \times s(B)}} = \frac{s(A) \times s(B)}{\sqrt{s(A) \times s(B)}} = \sqrt{s(A) \times s(B)}.$$

Since the value depends on $s(A)$ and $s(B)$, IS shares a similar problem as the confidence measure—that the value of the measure can be quite large, even for uncorrelated and negatively correlated patterns. For example, despite the large IS value between items p and q given in Table 6.10 (0.889), it is still less than the expected value when the items are statistically independent ($IS_{\text{indep}} = 0.9$).

Alternative Objective Interestingness Measures

Besides the measures we have described so far, there are other alternative measures proposed for analyzing relationships between pairs of binary variables. These measures can be divided into two categories, **symmetric** and **asymmetric** measures. A measure M is symmetric if $M(A \rightarrow B) = M(B \rightarrow A)$. For example, interest factor is a symmetric measure because its value is identical for the rules $A \rightarrow B$ and $B \rightarrow A$. In contrast, confidence is an asymmetric measure since the confidence for $A \rightarrow B$ and $B \rightarrow A$ may not be the same. Symmetric measures are generally used for evaluating itemsets, while asymmetric measures are more suitable for analyzing association rules. Tables 6.11 and 6.12 provide the definitions for some of these measures in terms of the frequency counts of a 2×2 contingency table.

Consistency among Objective Measures

Given the wide variety of measures available, it is reasonable to question whether the measures can produce similar ordering results when applied to a set of association patterns. If the measures are consistent, then we can choose any one of them as our evaluation metric. Otherwise, it is important to understand what their differences are in order to determine which measure is more suitable for analyzing certain types of patterns.

Table 6.11. Examples of symmetric objective measures for the itemset $\{A, B\}$.

Measure (Symbol)	Definition
Correlation (ϕ)	$\frac{Nf_{11}-f_{1+}f_{+1}}{\sqrt{f_{1+}f_{+1}f_{0+}f_{+0}}}$
Odds ratio (α)	$(f_{11}f_{00})/(f_{10}f_{01})$
Kappa (κ)	$\frac{Nf_{11}+Nf_{00}-f_{1+}f_{+1}-f_{0+}f_{+0}}{N^2-f_{1+}f_{+1}-f_{0+}f_{+0}}$
Interest (I)	$(Nf_{11})/(f_{1+}f_{+1})$
Cosine (IS)	$(f_{11})/(\sqrt{f_{1+}f_{+1}})$
Piatetsky-Shapiro (PS)	$\frac{f_{11}}{N} - \frac{f_{1+}f_{+1}}{N^2}$
Collective strength (S)	$\frac{f_{11}+f_{00}}{f_{1+}f_{+1}+f_{0+}f_{+0}} \times \frac{N-f_{1+}f_{+1}-f_{0+}f_{+0}}{N-f_{11}-f_{00}}$
Jaccard (ζ)	$f_{11}/(f_{1+} + f_{+1} - f_{11})$
All-confidence (h)	$\min \left[\frac{f_{11}}{f_{1+}}, \frac{f_{11}}{f_{+1}} \right]$

Table 6.12. Examples of asymmetric objective measures for the rule $A \rightarrow B$.

Measure (Symbol)	Definition
Goodman-Kruskal (λ)	$(\sum_j \max_k f_{jk} - \max_k f_{+k}) / (N - \max_k f_{+k})$
Mutual Information (M)	$(\sum_i \sum_j \frac{f_{ij}}{N} \log \frac{Nf_{ij}}{f_{i+}f_{+j}}) / (-\sum_i \frac{f_{i+}}{N} \log \frac{f_{i+}}{N})$
J-Measure (J)	$\frac{f_{11}}{N} \log \frac{Nf_{11}}{f_{1+}f_{+1}} + \frac{f_{10}}{N} \log \frac{Nf_{10}}{f_{1+}f_{+0}}$
Gini index (G)	$\frac{f_{1+}}{N} \times (\frac{f_{11}}{f_{1+}})^2 + (\frac{f_{10}}{f_{1+}})^2] - (\frac{f_{+1}}{N})^2$ $+ \frac{f_{0+}}{N} \times [(\frac{f_{01}}{f_{0+}})^2 + (\frac{f_{00}}{f_{0+}})^2] - (\frac{f_{+0}}{N})^2$
Laplace (L)	$(f_{11} + 1) / (f_{1+} + 2)$
Conviction (V)	$(f_{1+}f_{+0}) / (Nf_{10})$
Certainty factor (F)	$(\frac{f_{11}}{f_{1+}} - \frac{f_{+1}}{N}) / (1 - \frac{f_{+1}}{N})$
Added Value (AV)	$\frac{f_{11}}{f_{1+}} - \frac{f_{+1}}{N}$

Table 6.13. Example of contingency tables.

Example	f_{11}	f_{10}	f_{01}	f_{00}
E_1	8123	83	424	1370
E_2	8330	2	622	1046
E_3	3954	3080	5	2961
E_4	2886	1363	1320	4431
E_5	1500	2000	500	6000
E_6	4000	2000	1000	3000
E_7	9481	298	127	94
E_8	4000	2000	2000	2000
E_9	7450	2483	4	63
E_{10}	61	2483	4	7452

Suppose the symmetric and asymmetric measures are applied to rank the ten contingency tables shown in Table 6.13. These contingency tables are chosen to illustrate the differences among the existing measures. The ordering produced by these measures are shown in Tables 6.14 and 6.15, respectively (with 1 as the most interesting and 10 as the least interesting table). Although some of the measures appear to be consistent with each other, there are certain measures that produce quite different ordering results. For example, the rankings given by the ϕ -coefficient agree with those provided by κ and collective strength, but are somewhat different than the rankings produced by interest

Table 6.14. Rankings of contingency tables using the symmetric measures given in Table 6.11.

	ϕ	α	κ	I	IS	PS	S	ζ	h
E_1	1	3	1	6	2	2	1	2	2
E_2	2	1	2	7	3	5	2	3	3
E_3	3	2	4	4	5	1	3	6	8
E_4	4	8	3	3	7	3	4	7	5
E_5	5	7	6	2	9	6	6	9	9
E_6	6	9	5	5	6	4	5	5	7
E_7	7	6	7	9	1	8	7	1	1
E_8	8	10	8	8	8	7	8	8	7
E_9	9	4	9	10	4	9	9	4	4
E_{10}	10	5	10	1	10	10	10	10	10

Table 6.15. Rankings of contingency tables using the asymmetric measures given in Table 6.12.

	λ	M	J	G	L	V	F	AV
E_1	1	1	1	1	4	2	2	5
E_2	2	2	2	3	5	1	1	6
E_3	5	3	5	2	2	6	6	4
E_4	4	6	3	4	9	3	3	1
E_5	9	7	4	6	8	5	5	2
E_6	3	8	6	5	7	4	4	3
E_7	7	5	9	8	3	7	7	9
E_8	8	9	7	7	10	8	8	7
E_9	6	4	10	9	1	9	9	10
E_{10}	10	10	8	10	6	10	10	8

factor and odds ratio. Furthermore, a contingency table such as E_{10} is ranked lowest according to the ϕ -coefficient, but highest according to interest factor.

Properties of Objective Measures

The results shown in Table 6.14 suggest that a significant number of the measures provide conflicting information about the quality of a pattern. To understand their differences, we need to examine the properties of these measures.

Inversion Property Consider the bit vectors shown in Figure 6.28. The 0/1 bit in each column vector indicates whether a transaction (row) contains a particular item (column). For example, the vector \mathbf{A} indicates that item a

A	B	C	D	E	F
1	0	0	1	0	0
0	0	1	1	1	0
0	0	1	1	1	0
0	0	1	1	1	0
0	1	1	0	1	1
0	0	1	1	1	0
0	0	1	1	1	0
0	0	1	1	1	0
1	0	0	1	0	0

(a)

(b)

(c)

Figure 6.28. Effect of the inversion operation. The vectors **C** and **E** are inversions of vector **A**, while the vector **D** is an inversion of vectors **B** and **F**.

belongs to the first and last transactions, whereas the vector **B** indicates that item *b* is contained only in the fifth transaction. The vectors **C** and **E** are in fact related to the vector **A**—their bits have been inverted from 0’s (absence) to 1’s (presence), and vice versa. Similarly, **D** is related to vectors **B** and **F** by inverting their bits. The process of flipping a bit vector is called **inversion**. If a measure is invariant under the inversion operation, then its value for the vector pair (**C, D**) should be identical to its value for (**A, B**). The inversion property of a measure can be tested as follows.

Definition 6.6 (Inversion Property). An objective measure *M* is invariant under the inversion operation if its value remains the same when exchanging the frequency counts f_{11} with f_{00} and f_{10} with f_{01} .

Among the measures that remain invariant under this operation include the ϕ -coefficient, odds ratio, κ , and collective strength. These measures may not be suitable for analyzing asymmetric binary data. For example, the ϕ -coefficient between **C** and **D** is identical to the ϕ -coefficient between **A** and **B**, even though items *c* and *d* appear together more frequently than *a* and *b*. Furthermore, the ϕ -coefficient between **C** and **D** is less than that between **E** and **F** even though items *e* and *f* appear together only once! We had previously raised this issue when discussing the limitations of the ϕ -coefficient on page 375. For asymmetric binary data, measures that do not remain invariant under the inversion operation are preferred. Some of the non-invariant measures include interest factor, *IS*, *PS*, and the Jaccard coefficient.

Null Addition Property Suppose we are interested in analyzing the relationship between a pair of words, such as `data` and `mining`, in a set of documents. If a collection of articles about ice fishing is added to the data set, should the association between `data` and `mining` be affected? This process of adding unrelated data (in this case, documents) to a given data set is known as the **null addition** operation.

Definition 6.7 (Null Addition Property). An objective measure M is invariant under the null addition operation if it is not affected by increasing f_{00} , while all other frequencies in the contingency table stay the same.

For applications such as document analysis or market basket analysis, the measure is expected to remain invariant under the null addition operation. Otherwise, the relationship between words may disappear simply by adding enough documents that do not contain both words! Examples of measures that satisfy this property include cosine (IS) and Jaccard (ξ) measures, while those that violate this property include interest factor, PS , odds ratio, and the ϕ -coefficient.

Scaling Property Table 6.16 shows the contingency tables for gender and the grades achieved by students enrolled in a particular course in 1993 and 2004. The data in these tables showed that the number of male students has doubled since 1993, while the number of female students has increased by a factor of 3. However, the male students in 2004 are not performing any better than those in 1993 because the ratio of male students who achieve a high grade to those who achieve a low grade is still the same, i.e., 3:4. Similarly, the female students in 2004 are performing no better than those in 1993. The association between grade and gender is expected to remain unchanged despite changes in the sampling distribution.

Table 6.16. The grade-gender example.

	Male	Female	
High	30	20	50
Low	40	10	50
	70	30	100

(a) Sample data from 1993.

	Male	Female	
High	60	60	120
Low	80	30	110
	140	90	230

(b) Sample data from 2004.

Table 6.17. Properties of symmetric measures.

Symbol	Measure	Inversion	Null Addition	Scaling
ϕ	ϕ -coefficient	Yes	No	No
α	odds ratio	Yes	No	Yes
κ	Cohen's	Yes	No	No
I	Interest	No	No	No
IS	Cosine	No	Yes	No
PS	Piatetsky-Shapiro's	Yes	No	No
S	Collective strength	Yes	No	No
ζ	Jaccard	No	Yes	No
h	All-confidence	No	No	No
s	Support	No	No	No

Definition 6.8 (Scaling Invariance Property). An objective measure M is invariant under the row/column scaling operation if $M(T) = M(T')$, where T is a contingency table with frequency counts $[f_{11}; f_{10}; f_{01}; f_{00}]$, T' is a contingency table with scaled frequency counts $[k_1 k_3 f_{11}; k_2 k_3 f_{10}; k_1 k_4 f_{01}; k_2 k_4 f_{00}]$, and k_1, k_2, k_3, k_4 are positive constants.

From Table 6.17, notice that only the odds ratio (α) is invariant under the row and column scaling operations. All other measures such as the ϕ -coefficient, κ , IS , interest factor, and collective strength (S) change their values when the rows and columns of the contingency table are rescaled. Although we do not discuss the properties of asymmetric measures (such as confidence, J-measure, Gini index, and conviction), it is clear that such measures do not preserve their values under inversion and row/column scaling operations, but are invariant under the null addition operation.

6.7.2 Measures beyond Pairs of Binary Variables

The measures shown in Tables 6.11 and 6.12 are defined for pairs of binary variables (e.g., 2-itemsets or association rules). However, many of them, such as support and all-confidence, are also applicable to larger-sized itemsets. Other measures, such as interest factor, IS , PS , and Jaccard coefficient, can be extended to more than two variables using the frequency tables tabulated in a multidimensional contingency table. An example of a three-dimensional contingency table for a , b , and c is shown in Table 6.18. Each entry f_{ijk} in this table represents the number of transactions that contain a particular combination of items a , b , and c . For example, f_{101} is the number of transactions that contain a and c , but not b . On the other hand, a marginal frequency

Table 6.18. Example of a three-dimensional contingency table.

c	b	\bar{b}		\bar{c}	b	\bar{b}	
a	f_{111}	f_{101}	f_{1+1}	a	f_{110}	f_{100}	f_{1+0}
\bar{a}	f_{011}	f_{001}	f_{0+1}	\bar{a}	f_{010}	f_{000}	f_{0+0}
	f_{+11}	f_{+01}	f_{++1}		f_{+10}	f_{+00}	f_{++0}

such as f_{1+1} is the number of transactions that contain a and c , irrespective of whether b is present in the transaction.

Given a k -itemset $\{i_1, i_2, \dots, i_k\}$, the condition for statistical independence can be stated as follows:

$$f_{i_1 i_2 \dots i_k} = \frac{f_{i_1+ \dots +} \times f_{+i_2+ \dots +} \times \dots \times f_{++ \dots i_k}}{N^{k-1}}. \quad (6.12)$$

With this definition, we can extend objective measures such as interest factor and PS , which are based on deviations from statistical independence, to more than two variables:

$$\begin{aligned} I &= \frac{N^{k-1} \times f_{i_1 i_2 \dots i_k}}{f_{i_1+ \dots +} \times f_{+i_2+ \dots +} \times \dots \times f_{++ \dots i_k}} \\ PS &= \frac{f_{i_1 i_2 \dots i_k}}{N^k} - \frac{f_{i_1+ \dots +} \times f_{+i_2+ \dots +} \times \dots \times f_{++ \dots i_k}}{N^{k-1}} \end{aligned}$$

Another approach is to define the objective measure as the maximum, minimum, or average value for the associations between pairs of items in a pattern. For example, given a k -itemset $X = \{i_1, i_2, \dots, i_k\}$, we may define the ϕ -coefficient for X as the average ϕ -coefficient between every pair of items (i_p, i_q) in X . However, because the measure considers only pairwise associations, it may not capture all the underlying relationships within a pattern.

Analysis of multidimensional contingency tables is more complicated because of the presence of partial associations in the data. For example, some associations may appear or disappear when conditioned upon the value of certain variables. This problem is known as **Simpson's paradox** and is described in the next section. More sophisticated statistical techniques are available to analyze such relationships, e.g., loglinear models, but these techniques are beyond the scope of this book.

Table 6.19. A two-way contingency table between the sale of high-definition television and exercise machine.

Buy HDTV	Buy Exercise Machine		
	Yes	No	
Yes	99	81	180
No	54	66	120
	153	147	300

Table 6.20. Example of a three-way contingency table.

Customer Group	Buy HDTV	Buy Exercise Machine		Total
		Yes	No	
College Students	Yes	1	9	10
	No	4	30	34
Working Adult	Yes	98	72	170
	No	50	36	86

6.7.3 Simpson's Paradox

It is important to exercise caution when interpreting the association between variables because the observed relationship may be influenced by the presence of other confounding factors, i.e., hidden variables that are not included in the analysis. In some cases, the hidden variables may cause the observed relationship between a pair of variables to disappear or reverse its direction, a phenomenon that is known as Simpson's paradox. We illustrate the nature of this paradox with the following example.

Consider the relationship between the sale of high-definition television (HDTV) and exercise machine, as shown in Table 6.19. The rule $\{\text{HDTV}=\text{Yes}\} \rightarrow \{\text{Exercise machine}=\text{Yes}\}$ has a confidence of $99/180 = 55\%$ and the rule $\{\text{HDTV}=\text{No}\} \rightarrow \{\text{Exercise machine}=\text{Yes}\}$ has a confidence of $54/120 = 45\%$. Together, these rules suggest that customers who buy high-definition televisions are more likely to buy exercise machines than those who do not buy high-definition televisions.

However, a deeper analysis reveals that the sales of these items depend on whether the customer is a college student or a working adult. Table 6.20 summarizes the relationship between the sale of HDTVs and exercise machines among college students and working adults. Notice that the support counts given in the table for college students and working adults sum up to the frequencies shown in Table 6.19. Furthermore, there are more working adults

than college students who buy these items. For college students:

$$\begin{aligned} c(\{\text{HDTV=Yes}\} \rightarrow \{\text{Exercise machine=Yes}\}) &= 1/10 = 10\%, \\ c(\{\text{HDTV=No}\} \rightarrow \{\text{Exercise machine=Yes}\}) &= 4/34 = 11.8\%, \end{aligned}$$

while for working adults:

$$\begin{aligned} c(\{\text{HDTV=Yes}\} \rightarrow \{\text{Exercise machine=Yes}\}) &= 98/170 = 57.7\%, \\ c(\{\text{HDTV=No}\} \rightarrow \{\text{Exercise machine=Yes}\}) &= 50/86 = 58.1\%. \end{aligned}$$

The rules suggest that, for each group, customers who do not buy high-definition televisions are more likely to buy exercise machines, which contradict the previous conclusion when data from the two customer groups are pooled together. Even if alternative measures such as correlation, odds ratio, or interest are applied, we still find that the sale of HDTV and exercise machine is positively correlated in the combined data but is negatively correlated in the stratified data (see Exercise 20 on page 414). The reversal in the direction of association is known as Simpson's paradox.

The paradox can be explained in the following way. Notice that most customers who buy HDTVs are working adults. Working adults are also the largest group of customers who buy exercise machines. Because nearly 85% of the customers are working adults, the observed relationship between HDTV and exercise machine turns out to be stronger in the combined data than what it would have been if the data is stratified. This can also be illustrated mathematically as follows. Suppose

$$a/b < c/d \text{ and } p/q < r/s,$$

where a/b and p/q may represent the confidence of the rule $A \rightarrow B$ in two different strata, while c/d and r/s may represent the confidence of the rule $\bar{A} \rightarrow B$ in the two strata. When the data is pooled together, the confidence values of the rules in the combined data are $(a+p)/(b+q)$ and $(c+r)/(d+s)$, respectively. Simpson's paradox occurs when

$$\frac{a+p}{b+q} > \frac{c+r}{d+s},$$

thus leading to the wrong conclusion about the relationship between the variables. The lesson here is that proper stratification is needed to avoid generating spurious patterns resulting from Simpson's paradox. For example, market

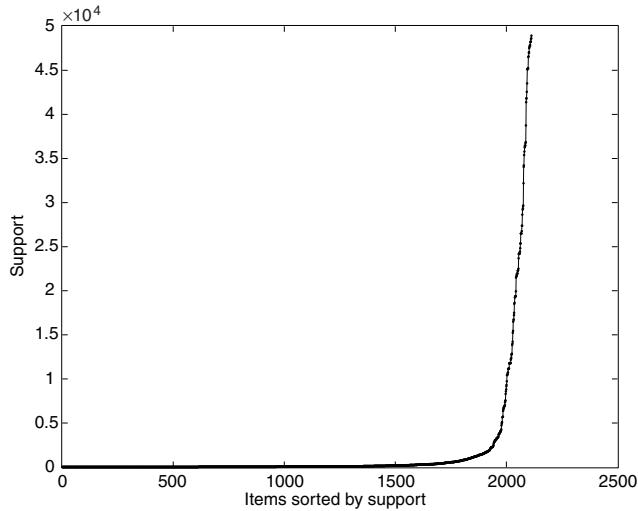


Figure 6.29. Support distribution of items in the census data set.

basket data from a major supermarket chain should be stratified according to store locations, while medical records from various patients should be stratified according to confounding factors such as age and gender.

6.8 Effect of Skewed Support Distribution

The performances of many association analysis algorithms are influenced by properties of their input data. For example, the computational complexity of the *Apriori* algorithm depends on properties such as the number of items in the data and average transaction width. This section examines another important property that has significant influence on the performance of association analysis algorithms as well as the quality of extracted patterns. More specifically, we focus on data sets with skewed support distributions, where most of the items have relatively low to moderate frequencies, but a small number of them have very high frequencies.

An example of a real data set that exhibits such a distribution is shown in Figure 6.29. The data, taken from the PUMS (Public Use Microdata Sample) census data, contains 49,046 records and 2113 asymmetric binary variables. We shall treat the asymmetric binary variables as items and records as transactions in the remainder of this section. While more than 80% of the items have support less than 1%, a handful of them have support greater than 90%.

Table 6.21. Grouping the items in the census data set based on their support values.

Group	G_1	G_2	G_3
Support	< 1%	1% – 90%	> 90%
Number of Items	1735	358	20

To illustrate the effect of skewed support distribution on frequent itemset mining, we divide the items into three groups, G_1 , G_2 , and G_3 , according to their support levels. The number of items that belong to each group is shown in Table 6.21.

Choosing the right support threshold for mining this data set can be quite tricky. If we set the threshold too high (e.g., 20%), then we may miss many interesting patterns involving the low support items from G_1 . In market basket analysis, such low support items may correspond to expensive products (such as jewelry) that are seldom bought by customers, but whose patterns are still interesting to retailers. Conversely, when the threshold is set too low, it becomes difficult to find the association patterns due to the following reasons. First, the computational and memory requirements of existing association analysis algorithms increase considerably with low support thresholds. Second, the number of extracted patterns also increases substantially with low support thresholds. Third, we may extract many spurious patterns that relate a high-frequency item such as milk to a low-frequency item such as caviar. Such patterns, which are called **cross-support** patterns, are likely to be spurious because their correlations tend to be weak. For example, at a support threshold equal to 0.05%, there are 18,847 frequent pairs involving items from G_1 and G_3 . Out of these, 93% of them are cross-support patterns; i.e., the patterns contain items from both G_1 and G_3 . The maximum correlation obtained from the cross-support patterns is 0.029, which is much lower than the maximum correlation obtained from frequent patterns involving items from the same group (which is as high as 1.0). Similar statement can be made about many other interestingness measures discussed in the previous section. This example shows that a large number of weakly correlated cross-support patterns can be generated when the support threshold is sufficiently low. Before presenting a methodology for eliminating such patterns, we formally define the concept of cross-support patterns.

Definition 6.9 (Cross-Support Pattern). A cross-support pattern is an itemset $X = \{i_1, i_2, \dots, i_k\}$ whose support ratio

$$r(X) = \frac{\min [s(i_1), s(i_2), \dots, s(i_k)]}{\max [s(i_1), s(i_2), \dots, s(i_k)]}, \quad (6.13)$$

is less than a user-specified threshold h_c .

Example 6.4. Suppose the support for milk is 70%, while the support for sugar is 10% and caviar is 0.04%. Given $h_c = 0.01$, the frequent itemset {milk, sugar, caviar} is a cross-support pattern because its support ratio is

$$r = \frac{\min [0.7, 0.1, 0.0004]}{\max [0.7, 0.1, 0.0004]} = \frac{0.0004}{0.7} = 0.00058 < 0.01.$$

■

Existing measures such as support and confidence may not be sufficient to eliminate cross-support patterns, as illustrated by the data set shown in Figure 6.30. Assuming that $h_c = 0.3$, the itemsets $\{p, q\}$, $\{p, r\}$, and $\{p, q, r\}$ are cross-support patterns because their support ratios, which are equal to 0.2, are less than the threshold h_c . Although we can apply a high support threshold, say, 20%, to eliminate the cross-support patterns, this may come at the expense of discarding other interesting patterns such as the strongly correlated itemset, $\{q, r\}$ that has support equal to 16.7%.

Confidence pruning also does not help because the confidence of the rules extracted from cross-support patterns can be very high. For example, the confidence for $\{q\} \rightarrow \{p\}$ is 80% even though $\{p, q\}$ is a cross-support pattern. The fact that the cross-support pattern can produce a high-confidence rule should not come as a surprise because one of its items (p) appears very frequently in the data. Therefore, p is expected to appear in many of the transactions that contain q . Meanwhile, the rule $\{q\} \rightarrow \{r\}$ also has high confidence even though $\{q, r\}$ is not a cross-support pattern. This example demonstrates the difficulty of using the confidence measure to distinguish between rules extracted from cross-support and non-cross-support patterns.

Returning to the previous example, notice that the rule $\{p\} \rightarrow \{q\}$ has very low confidence because most of the transactions that contain p do not contain q . In contrast, the rule $\{r\} \rightarrow \{q\}$, which is derived from the pattern $\{q, r\}$, has very high confidence. This observation suggests that cross-support patterns can be detected by examining the lowest confidence rule that can be extracted from a given itemset. The proof of this statement can be understood as follows.

p	q	r
0	1	1
1	1	1
1	1	1
1	1	1
1	1	1
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
0	0	0
0	0	0
0	0	0
0	0	0

Figure 6.30. A transaction data set containing three items, p , q , and r , where p is a high support item and q and r are low support items.

1. Recall the following anti-monotone property of confidence:

$$\text{conf}(\{i_1 i_2\} \rightarrow \{i_3, i_4, \dots, i_k\}) \leq \text{conf}(\{i_1 i_2 i_3\} \rightarrow \{i_4, i_5, \dots, i_k\}).$$

This property suggests that confidence never increases as we shift more items from the left- to the right-hand side of an association rule. Because of this property, the lowest confidence rule extracted from a frequent itemset contains only one item on its left-hand side. We denote the set of all rules with only one item on its left-hand side as R_1 .

2. Given a frequent itemset $\{i_1, i_2, \dots, i_k\}$, the rule

$$\{i_j\} \rightarrow \{i_1, i_2, \dots, i_{j-1}, i_{j+1}, \dots, i_k\}$$

has the lowest confidence in R_1 if $s(i_j) = \max[s(i_1), s(i_2), \dots, s(i_k)]$. This follows directly from the definition of confidence as the ratio between the rule's support and the support of the rule antecedent.

3. Summarizing the previous points, the lowest confidence attainable from a frequent itemset $\{i_1, i_2, \dots, i_k\}$ is

$$\frac{s(\{i_1, i_2, \dots, i_k\})}{\max[s(i_1), s(i_2), \dots, s(i_k)]}.$$

This expression is also known as the **h-confidence** or **all-confidence** measure. Because of the anti-monotone property of support, the numerator of the h-confidence measure is bounded by the minimum support of any item that appears in the frequent itemset. In other words, the h-confidence of an itemset $X = \{i_1, i_2, \dots, i_k\}$ must not exceed the following expression:

$$\text{h-confidence}(X) \leq \frac{\min[s(i_1), s(i_2), \dots, s(i_k)]}{\max[s(i_1), s(i_2), \dots, s(i_k)]}.$$

Note the equivalence between the upper bound of h-confidence and the support ratio (r) given in Equation 6.13. Because the support ratio for a cross-support pattern is always less than h_c , the h-confidence of the pattern is also guaranteed to be less than h_c .

Therefore, cross-support patterns can be eliminated by ensuring that the h-confidence values for the patterns exceed h_c . As a final note, it is worth mentioning that the advantages of using h-confidence go beyond eliminating cross-support patterns. The measure is also anti-monotone, i.e.,

$$\text{h-confidence}(\{i_1, i_2, \dots, i_k\}) \geq \text{h-confidence}(\{i_1, i_2, \dots, i_{k+1}\}),$$

and thus can be incorporated directly into the mining algorithm. Furthermore, h-confidence ensures that the items contained in an itemset are strongly associated with each other. For example, suppose the h-confidence of an itemset X is 80%. If one of the items in X is present in a transaction, there is at least an 80% chance that the rest of the items in X also belong to the same transaction. Such strongly associated patterns are called **hyperclique patterns**.

6.9 Bibliographic Notes

The association rule mining task was first introduced by Agrawal et al. in [228, 229] to discover interesting relationships among items in market basket

transactions. Since its inception, extensive studies have been conducted to address the various conceptual, implementation, and application issues pertaining to the association analysis task. A summary of the various research activities in this area is shown in Figure 6.31.

Conceptual Issues

Research in conceptual issues is focused primarily on (1) developing a framework to describe the theoretical underpinnings of association analysis, (2) extending the formulation to handle new types of patterns, and (3) extending the formulation to incorporate attribute types beyond asymmetric binary data.

Following the pioneering work by Agrawal et al., there has been a vast amount of research on developing a theory for the association analysis problem. In [254], Gunopoulos et al. showed a relation between the problem of finding maximal frequent itemsets and the hypergraph transversal problem. An upper bound on the complexity of association analysis task was also derived. Zaki et al. [334, 336] and Pasquier et al. [294] have applied formal concept analysis to study the frequent itemset generation problem. The work by Zaki et al. have subsequently led them to introduce the notion of closed frequent itemsets [336]. Friedman et al. have studied the association analysis problem in the context of **bump hunting** in multidimensional space [252]. More specifically, they consider frequent itemset generation as the task of finding high probability density regions in multidimensional space.

Over the years, new types of patterns have been defined, such as profile association rules [225], cyclic association rules [290], fuzzy association rules [273], exception rules [316], negative association rules [238, 304], weighted association rules [240, 300], dependence rules [308], peculiar rules[340], inter-transaction association rules [250, 323], and partial classification rules [231, 285]. Other types of patterns include closed itemsets [294, 336], maximal itemsets [234], hyperclique patterns [330], support envelopes [314], emerging patterns [246], and contrast sets [233]. Association analysis has also been successfully applied to sequential [230, 312], spatial [266], and graph-based [268, 274, 293, 331, 335] data. The concept of cross-support pattern was first introduced by Hui et al. in [330]. An efficient algorithm (called Hyperclique Miner) that automatically eliminates cross-support patterns was also proposed by the authors.

Substantial research has been conducted to extend the original association rule formulation to nominal [311], ordinal [281], interval [284], and ratio [253, 255, 311, 325, 339] attributes. One of the key issues is how to define the support measure for these attributes. A methodology was proposed by Steinbach et

al. [315] to extend the traditional notion of support to more general patterns and attribute types.

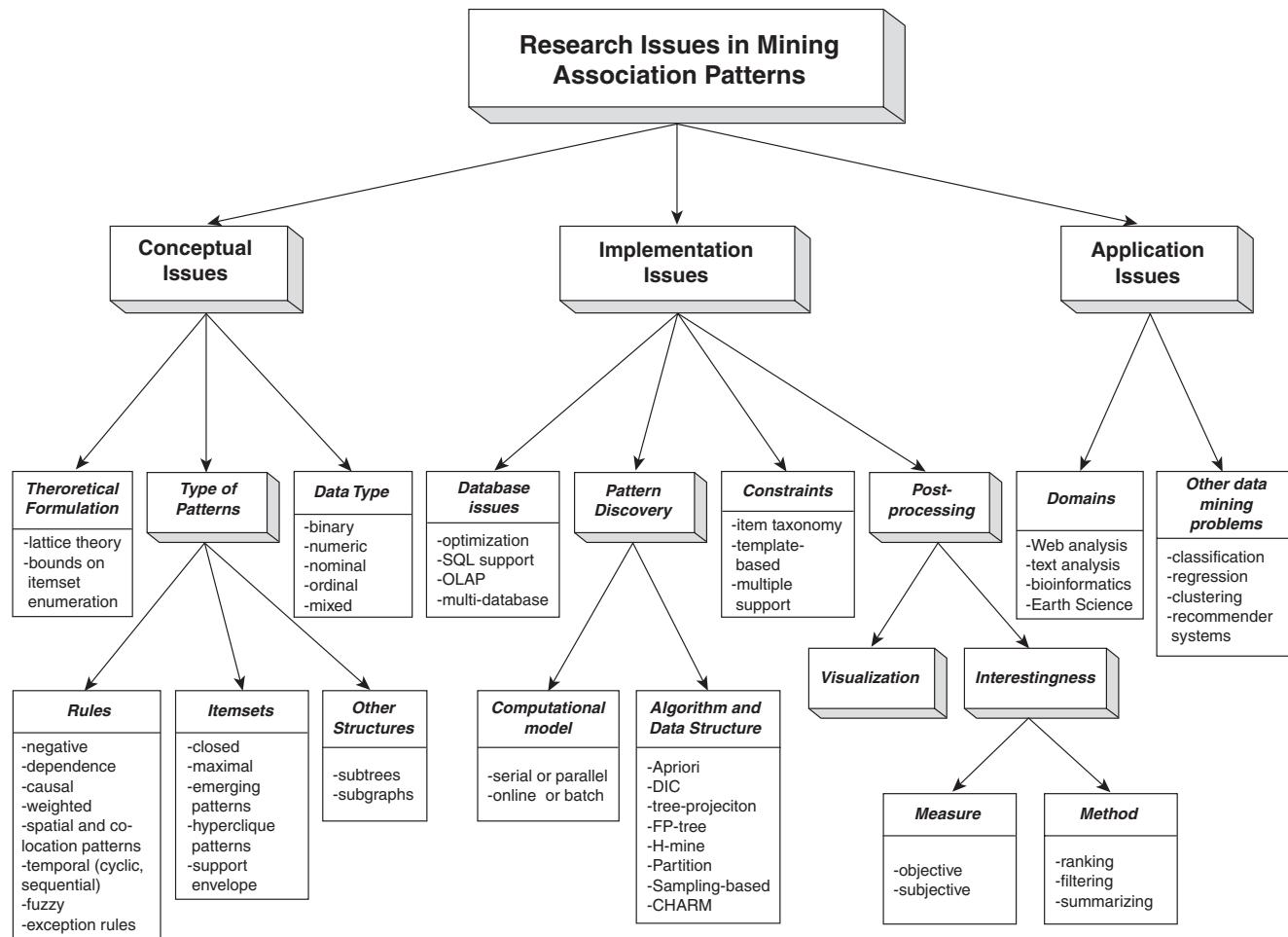


Figure 6.31. A summary of the various research activities in association analysis.

Implementation Issues

Research activities in this area revolve around (1) integrating the mining capability into existing database technology, (2) developing efficient and scalable mining algorithms, (3) handling user-specified or domain-specific constraints, and (4) post-processing the extracted patterns.

There are several advantages to integrating association analysis into existing database technology. First, it can make use of the indexing and query processing capabilities of the database system. Second, it can also exploit the DBMS support for scalability, check-pointing, and parallelization [301]. The SETM algorithm developed by Houtsma et al. [265] was one of the earliest algorithms to support association rule discovery via SQL queries. Since then, numerous methods have been developed to provide capabilities for mining association rules in database systems. For example, the DMQL [258] and M-SQL [267] query languages extend the basic SQL with new operators for mining association rules. The Mine Rule operator [283] is an expressive SQL operator that can handle both clustered attributes and item hierarchies. Tsur et al. [322] developed a generate-and-test approach called **query flocks** for mining association rules. A distributed OLAP-based infrastructure was developed by Chen et al. [241] for mining multilevel association rules.

Dunkel and Soparkar [248] investigated the time and storage complexity of the *Apriori* algorithm. The FP-growth algorithm was developed by Han et al. in [259]. Other algorithms for mining frequent itemsets include the DHP (dynamic hashing and pruning) algorithm proposed by Park et al. [292] and the Partition algorithm developed by Savasere et al [303]. A sampling-based frequent itemset generation algorithm was proposed by Toivonen [320]. The algorithm requires only a single pass over the data, but it can produce more candidate itemsets than necessary. The Dynamic Itemset Counting (DIC) algorithm [239] makes only 1.5 passes over the data and generates less candidate itemsets than the sampling-based algorithm. Other notable algorithms include the tree-projection algorithm [223] and H-Mine [295]. Survey articles on frequent itemset generation algorithms can be found in [226, 262]. A repository of data sets and algorithms is available at the Frequent Itemset Mining Implementations (FIMI) repository (<http://fimi.cs.helsinki.fi>). Parallel algorithms for mining association patterns have been developed by various authors [224, 256, 287, 306, 337]. A survey of such algorithms can be found in [333]. Online and incremental versions of association rule mining algorithms had also been proposed by Hidber [260] and Cheung et al. [242].

Srikant et al. [313] have considered the problem of mining association rules in the presence of boolean constraints such as the following:

$$(\text{Cookies} \wedge \text{Milk}) \vee (\text{descendents}(\text{Cookies}) \wedge \neg\text{ancestors}(\text{Wheat Bread}))$$

Given such a constraint, the algorithm looks for rules that contain both cookies and milk, or rules that contain the descendent items of cookies but not ancestor items of wheat bread. Singh et al. [310] and Ng et al. [288] had also developed alternative techniques for constrained-based association rule mining. Constraints can also be imposed on the support for different itemsets. This problem was investigated by Wang et al. [324], Liu et al. in [279], and Seno et al. [305].

One potential problem with association analysis is the large number of patterns that can be generated by current algorithms. To overcome this problem, methods to rank, summarize, and filter patterns have been developed. Toivonen et al. [321] proposed the idea of eliminating redundant rules using **structural rule covers** and to group the remaining rules using clustering. Liu et al. [280] applied the statistical chi-square test to prune spurious patterns and summarized the remaining patterns using a subset of the patterns called **direction setting rules**. The use of objective measures to filter patterns has been investigated by many authors, including Brin et al. [238], Bayardo and Agrawal [235], Aggarwal and Yu [227], and DuMouchel and Pregibon[247]. The properties for many of these measures were analyzed by Piatetsky-Shapiro [297], Kamber and Singhal [270], Hilderman and Hamilton [261], and Tan et al. [318]. The grade-gender example used to highlight the importance of the row and column scaling invariance property was heavily influenced by the discussion given in [286] by Mosteller. Meanwhile, the tea-coffee example illustrating the limitation of confidence was motivated by an example given in [238] by Brin et al. Because of the limitation of confidence, Brin et al. [238] had proposed the idea of using interest factor as a measure of interestingness. The all-confidence measure was proposed by Omiecinski [289]. Xiong et al. [330] introduced the cross-support property and showed that the all-confidence measure can be used to eliminate cross-support patterns. A key difficulty in using alternative objective measures besides support is their lack of a monotonicity property, which makes it difficult to incorporate the measures directly into the mining algorithms. Xiong et al. [328] have proposed an efficient method for mining correlations by introducing an upper bound function to the ϕ -coefficient. Although the measure is non-monotone, it has an upper bound expression that can be exploited for the efficient mining of strongly correlated itempairs.

Fabris and Freitas [249] have proposed a method for discovering interesting associations by detecting the occurrences of Simpson's paradox [309]. Megiddo and Srikant [282] described an approach for validating the extracted

patterns using hypothesis testing methods. A resampling-based technique was also developed to avoid generating spurious patterns because of the multiple comparison problem. Bolton et al. [237] have applied the Benjamini-Hochberg [236] and Bonferroni correction methods to adjust the p-values of discovered patterns in market basket data. Alternative methods for handling the multiple comparison problem were suggested by Webb [326] and Zhang et al. [338].

Application of subjective measures to association analysis has been investigated by many authors. Silberschatz and Tuzhilin [307] presented two principles in which a rule can be considered interesting from a subjective point of view. The concept of unexpected condition rules was introduced by Liu et al. in [277]. Cooley et al. [243] analyzed the idea of combining soft belief sets using the Dempster-Shafer theory and applied this approach to identify contradictory and novel association patterns in Web data. Alternative approaches include using Bayesian networks [269] and neighborhood-based information [245] to identify subjectively interesting patterns.

Visualization also helps the user to quickly grasp the underlying structure of the discovered patterns. Many commercial data mining tools display the complete set of rules (which satisfy both support and confidence threshold criteria) as a two-dimensional plot, with each axis corresponding to the antecedent or consequent itemsets of the rule. Hofmann et al. [263] proposed using Mosaic plots and Double Decker plots to visualize association rules. This approach can visualize not only a particular rule, but also the overall contingency table between itemsets in the antecedent and consequent parts of the rule. Nevertheless, this technique assumes that the rule consequent consists of only a single attribute.

Application Issues

Association analysis has been applied to a variety of application domains such as Web mining [296, 317], document analysis [264], telecommunication alarm diagnosis [271], network intrusion detection [232, 244, 275], and bioinformatics [302, 327]. Applications of association and correlation pattern analysis to Earth Science studies have been investigated in [298, 299, 319].

Association patterns have also been applied to other learning problems such as classification [276, 278], regression [291], and clustering [257, 329, 332]. A comparison between classification and association rule mining was made by Freitas in his position paper [251]. The use of association patterns for clustering has been studied by many authors including Han et al. [257], Kosters et al. [272], Yang et al. [332] and Xiong et al. [329].

Bibliography

- [223] R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad. A Tree Projection Algorithm for Generation of Frequent Itemsets. *Journal of Parallel and Distributed Computing (Special Issue on High Performance Data Mining)*, 61(3):350–371, 2001.
- [224] R. C. Agarwal and J. C. Shafer. Parallel Mining of Association Rules. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):962–969, March 1998.
- [225] C. C. Aggarwal, Z. Sun, and P. S. Yu. Online Generation of Profile Association Rules. In *Proc. of the 4th Intl. Conf. on Knowledge Discovery and Data Mining*, pages 129–133, New York, NY, August 1996.
- [226] C. C. Aggarwal and P. S. Yu. Mining Large Itemsets for Association Rules. *Data Engineering Bulletin*, 21(1):23–31, March 1998.
- [227] C. C. Aggarwal and P. S. Yu. Mining Associations with the Collective Strength Approach. *IEEE Trans. on Knowledge and Data Engineering*, 13(6):863–873, January/February 2001.
- [228] R. Agrawal, T. Imielinski, and A. Swami. Database mining: A performance perspective. *IEEE Transactions on Knowledge and Data Engineering*, 5:914–925, 1993.
- [229] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. ACM SIGMOD Intl. Conf. Management of Data*, pages 207–216, Washington, DC, 1993.
- [230] R. Agrawal and R. Srikant. Mining Sequential Patterns. In *Proc. of Intl. Conf. on Data Engineering*, pages 3–14, Taipei, Taiwan, 1995.
- [231] K. Ali, S. Manganaris, and R. Srikant. Partial Classification using Association Rules. In *Proc. of the 3rd Intl. Conf. on Knowledge Discovery and Data Mining*, pages 115–118, Newport Beach, CA, August 1997.
- [232] D. Barbará, J. Couto, S. Jajodia, and N. Wu. ADAM: A Testbed for Exploring the Use of Data Mining in Intrusion Detection. *SIGMOD Record*, 30(4):15–24, 2001.
- [233] S. D. Bay and M. Pazzani. Detecting Group Differences: Mining Contrast Sets. *Data Mining and Knowledge Discovery*, 5(3):213–246, 2001.
- [234] R. Bayardo. Efficiently Mining Long Patterns from Databases. In *Proc. of 1998 ACM-SIGMOD Intl. Conf. on Management of Data*, pages 85–93, Seattle, WA, June 1998.
- [235] R. Bayardo and R. Agrawal. Mining the Most Interesting Rules. In *Proc. of the 5th Intl. Conf. on Knowledge Discovery and Data Mining*, pages 145–153, San Diego, CA, August 1999.
- [236] Y. Benjamini and Y. Hochberg. Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *Journal Royal Statistical Society B*, 57(1):289–300, 1995.
- [237] R. J. Bolton, D. J. Hand, and N. M. Adams. Determining Hit Rate in Pattern Search. In *Proc. of the ESF Exploratory Workshop on Pattern Detection and Discovery in Data Mining*, pages 36–48, London, UK, September 2002.
- [238] S. Brin, R. Motwani, and C. Silverstein. Beyond market baskets: Generalizing association rules to correlations. In *Proc. ACM SIGMOD Intl. Conf. Management of Data*, pages 265–276, Tucson, AZ, 1997.
- [239] S. Brin, R. Motwani, J. Ullman, and S. Tsur. Dynamic Itemset Counting and Implication Rules for market basket data. In *Proc. of 1997 ACM-SIGMOD Intl. Conf. on Management of Data*, pages 255–264, Tucson, AZ, June 1997.
- [240] C. H. Cai, A. Fu, C. H. Cheng, and W. W. Kwong. Mining Association Rules with Weighted Items. In *Proc. of IEEE Intl. Database Engineering and Applications Symp.*, pages 68–77, Cardiff, Wales, 1998.

- [241] Q. Chen, U. Dayal, and M. Hsu. A Distributed OLAP infrastructure for E-Commerce. In *Proc. of the 4th IFCIS Intl. Conf. on Cooperative Information Systems*, pages 209–220, Edinburgh, Scotland, 1999.
- [242] D. C. Cheung, S. D. Lee, and B. Kao. A General Incremental Technique for Maintaining Discovered Association Rules. In *Proc. of the 5th Intl. Conf. on Database Systems for Advanced Applications*, pages 185–194, Melbourne, Australia, 1997.
- [243] R. Cooley, P. N. Tan, and J. Srivastava. Discovery of Interesting Usage Patterns from Web Data. In M. Spiliopoulou and B. Masand, editors, *Advances in Web Usage Analysis and User Profiling*, volume 1836, pages 163–182. Lecture Notes in Computer Science, 2000.
- [244] P. Dokas, L. Ertöz, V. Kumar, A. Lazarevic, J. Srivastava, and P. N. Tan. Data Mining for Network Intrusion Detection. In *Proc. NSF Workshop on Next Generation Data Mining*, Baltimore, MD, 2002.
- [245] G. Dong and J. Li. Interestingness of discovered association rules in terms of neighborhood-based unexpectedness. In *Proc. of the 2nd Pacific-Asia Conf. on Knowledge Discovery and Data Mining*, pages 72–86, Melbourne, Australia, April 1998.
- [246] G. Dong and J. Li. Efficient Mining of Emerging Patterns: Discovering Trends and Differences. In *Proc. of the 5th Intl. Conf. on Knowledge Discovery and Data Mining*, pages 43–52, San Diego, CA, August 1999.
- [247] W. DuMouchel and D. Pregibon. Empirical Bayes Screening for Multi-Item Associations. In *Proc. of the 7th Intl. Conf. on Knowledge Discovery and Data Mining*, pages 67–76, San Francisco, CA, August 2001.
- [248] B. Dunkel and N. Soparkar. Data Organization and Access for Efficient Data Mining. In *Proc. of the 15th Intl. Conf. on Data Engineering*, pages 522–529, Sydney, Australia, March 1999.
- [249] C. C. Fabris and A. A. Freitas. Discovering surprising patterns by detecting occurrences of Simpson’s paradox. In *Proc. of the 19th SGES Intl. Conf. on Knowledge-Based Systems and Applied Artificial Intelligence*, pages 148–160, Cambridge, UK, December 1999.
- [250] L. Feng, H. J. Lu, J. X. Yu, and J. Han. Mining inter-transaction associations with templates. In *Proc. of the 8th Intl. Conf. on Information and Knowledge Management*, pages 225–233, Kansas City, Missouri, Nov 1999.
- [251] A. A. Freitas. Understanding the crucial differences between classification and discovery of association rules—a position paper. *SIGKDD Explorations*, 2(1):65–69, 2000.
- [252] J. H. Friedman and N. I. Fisher. Bump hunting in high-dimensional data. *Statistics and Computing*, 9(2):123–143, April 1999.
- [253] T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama. Mining Optimized Association Rules for Numeric Attributes. In *Proc. of the 15th Symp. on Principles of Database Systems*, pages 182–191, Montreal, Canada, June 1996.
- [254] D. Gunopulos, R. Kharden, H. Mannila, and H. Toivonen. Data Mining, Hypergraph Transversals, and Machine Learning. In *Proc. of the 16th Symp. on Principles of Database Systems*, pages 209–216, Tucson, AZ, May 1997.
- [255] E.-H. Han, G. Karypis, and V. Kumar. Min-Apriori: An Algorithm for Finding Association Rules in Data with Continuous Attributes. <http://www.cs.umn.edu/~han>, 1997.
- [256] E.-H. Han, G. Karypis, and V. Kumar. Scalable Parallel Data Mining for Association Rules. In *Proc. of 1997 ACM-SIGMOD Intl. Conf. on Management of Data*, pages 277–288, Tucson, AZ, May 1997.

- [257] E.-H. Han, G. Karypis, V. Kumar, and B. Mobasher. Clustering Based on Association Rule Hypergraphs. In *Proc. of the 1997 ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, Tucson, AZ, 1997.
- [258] J. Han, Y. Fu, K. Koperski, W. Wang, and O. R. Zaïane. DMQL: A data mining query language for relational databases. In *Proc. of the 1996 ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, Montreal, Canada, June 1996.
- [259] J. Han, J. Pei, and Y. Yin. Mining Frequent Patterns without Candidate Generation. In *Proc. ACM-SIGMOD Int. Conf. on Management of Data (SIGMOD'00)*, pages 1–12, Dallas, TX, May 2000.
- [260] C. Hidber. Online Association Rule Mining. In *Proc. of 1999 ACM-SIGMOD Intl. Conf. on Management of Data*, pages 145–156, Philadelphia, PA, 1999.
- [261] R. J. Hilderman and H. J. Hamilton. *Knowledge Discovery and Measures of Interest*. Kluwer Academic Publishers, 2001.
- [262] J. Hipp, U. Guntzer, and G. Nakhaeizadeh. Algorithms for Association Rule Mining—A General Survey. *SigKDD Explorations*, 2(1):58–64, June 2000.
- [263] H. Hofmann, A. P. J. M. Siebes, and A. F. X. Wilhelm. Visualizing Association Rules with Interactive Mosaic Plots. In *Proc. of the 6th Intl. Conf. on Knowledge Discovery and Data Mining*, pages 227–235, Boston, MA, August 2000.
- [264] J. D. Holt and S. M. Chung. Efficient Mining of Association Rules in Text Databases. In *Proc. of the 8th Intl. Conf. on Information and Knowledge Management*, pages 234–242, Kansas City, Missouri, 1999.
- [265] M. Houtsma and A. Swami. Set-oriented Mining for Association Rules in Relational Databases. In *Proc. of the 11th Intl. Conf. on Data Engineering*, pages 25–33, Taipei, Taiwan, 1995.
- [266] Y. Huang, S. Shekhar, and H. Xiong. Discovering Co-location Patterns from Spatial Datasets: A General Approach. *IEEE Trans. on Knowledge and Data Engineering*, 16(12):1472–1485, December 2004.
- [267] T. Imielinski, A. Virmani, and A. Abdulghani. DataMine: Application Programming Interface and Query Language for Database Mining. In *Proc. of the 2nd Intl. Conf. on Knowledge Discovery and Data Mining*, pages 256–262, Portland, Oregon, 1996.
- [268] A. Inokuchi, T. Washio, and H. Motoda. An Apriori-based Algorithm for Mining Frequent Substructures from Graph Data. In *Proc. of the 4th European Conf. of Principles and Practice of Knowledge Discovery in Databases*, pages 13–23, Lyon, France, 2000.
- [269] S. Jaroszewicz and D. Simovici. Interestingness of Frequent Itemsets Using Bayesian Networks as Background Knowledge. In *Proc. of the 10th Intl. Conf. on Knowledge Discovery and Data Mining*, pages 178–186, Seattle, WA, August 2004.
- [270] M. Kamber and R. Shinghal. Evaluating the Interestingness of Characteristic Rules. In *Proc. of the 2nd Intl. Conf. on Knowledge Discovery and Data Mining*, pages 263–266, Portland, Oregon, 1996.
- [271] M. Klemettinen. *A Knowledge Discovery Methodology for Telecommunication Network Alarm Databases*. PhD thesis, University of Helsinki, 1999.
- [272] W. A. Kosters, E. Marchiori, and A. Oerlemans. Mining Clusters with Association Rules. In *The 3rd Symp. on Intelligent Data Analysis (IDA99)*, pages 39–50, Amsterdam, August 1999.
- [273] C. M. Kuok, A. Fu, and M. H. Wong. Mining Fuzzy Association Rules in Databases. *ACM SIGMOD Record*, 27(1):41–46, March 1998.

- [274] M. Kuramochi and G. Karypis. Frequent Subgraph Discovery. In *Proc. of the 2001 IEEE Intl. Conf. on Data Mining*, pages 313–320, San Jose, CA, November 2001.
- [275] W. Lee, S. J. Stolfo, and K. W. Mok. Adaptive Intrusion Detection: A Data Mining Approach. *Artificial Intelligence Review*, 14(6):533–567, 2000.
- [276] W. Li, J. Han, and J. Pei. CMAR: Accurate and Efficient Classification Based on Multiple Class-association Rules. In *Proc. of the 2001 IEEE Intl. Conf. on Data Mining*, pages 369–376, San Jose, CA, 2001.
- [277] B. Liu, W. Hsu, and S. Chen. Using General Impressions to Analyze Discovered Classification Rules. In *Proc. of the 3rd Intl. Conf. on Knowledge Discovery and Data Mining*, pages 31–36, Newport Beach, CA, August 1997.
- [278] B. Liu, W. Hsu, and Y. Ma. Integrating Classification and Association Rule Mining. In *Proc. of the 4th Intl. Conf. on Knowledge Discovery and Data Mining*, pages 80–86, New York, NY, August 1998.
- [279] B. Liu, W. Hsu, and Y. Ma. Mining association rules with multiple minimum supports. In *Proc. of the 5th Intl. Conf. on Knowledge Discovery and Data Mining*, pages 125–134, San Diego, CA, August 1999.
- [280] B. Liu, W. Hsu, and Y. Ma. Pruning and Summarizing the Discovered Associations. In *Proc. of the 5th Intl. Conf. on Knowledge Discovery and Data Mining*, pages 125–134, San Diego, CA, August 1999.
- [281] A. Marcus, J. I. Maletic, and K.-I. Lin. Ordinal association rules for error identification in data sets. In *Proc. of the 10th Intl. Conf. on Information and Knowledge Management*, pages 589–591, Atlanta, GA, October 2001.
- [282] N. Megiddo and R. Srikant. Discovering Predictive Association Rules. In *Proc. of the 4th Intl. Conf. on Knowledge Discovery and Data Mining*, pages 274–278, New York, August 1998.
- [283] R. Meo, G. Psaila, and S. Ceri. A New SQL-like Operator for Mining Association Rules. In *Proc. of the 22nd VLDB Conf.*, pages 122–133, Bombay, India, 1996.
- [284] R. J. Miller and Y. Yang. Association Rules over Interval Data. In *Proc. of 1997 ACM-SIGMOD Intl. Conf. on Management of Data*, pages 452–461, Tucson, AZ, May 1997.
- [285] Y. Morimoto, T. Fukuda, H. Matsuzawa, T. Tokuyama, and K. Yoda. Algorithms for mining association rules for binary segmentations of huge categorical databases. In *Proc. of the 24th VLDB Conf.*, pages 380–391, New York, August 1998.
- [286] F. Mosteller. Association and Estimation in Contingency Tables. *Journal of the American Statistical Association*, 63:1–28, 1968.
- [287] A. Mueller. Fast sequential and parallel algorithms for association rule mining: A comparison. Technical Report CS-TR-3515, University of Maryland, August 1995.
- [288] R. T. Ng, L. V. S. Lakshmanan, J. Han, and A. Pang. Exploratory Mining and Pruning Optimizations of Constrained Association Rules. In *Proc. of 1998 ACM-SIGMOD Intl. Conf. on Management of Data*, pages 13–24, Seattle, WA, June 1998.
- [289] E. Omiecinski. Alternative Interest Measures for Mining Associations in Databases. *IEEE Trans. on Knowledge and Data Engineering*, 15(1):57–69, January/February 2003.
- [290] B. Ozden, S. Ramaswamy, and A. Silberschatz. Cyclic Association Rules. In *Proc. of the 14th Intl. Conf. on Data Eng.*, pages 412–421, Orlando, FL, February 1998.
- [291] A. Ozgur, P. N. Tan, and V. Kumar. RBA: An Integrated Framework for Regression based on Association Rules. In *Proc. of the SIAM Intl. Conf. on Data Mining*, pages 210–221, Orlando, FL, April 2004.

- [292] J. S. Park, M.-S. Chen, and P. S. Yu. An effective hash-based algorithm for mining association rules. *SIGMOD Record*, 25(2):175–186, 1995.
- [293] S. Parthasarathy and M. Coatney. Efficient Discovery of Common Substructures in Macromolecules. In *Proc. of the 2002 IEEE Intl. Conf. on Data Mining*, pages 362–369, Maebashi City, Japan, December 2002.
- [294] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *Proc. of the 7th Intl. Conf. on Database Theory (ICDT'99)*, pages 398–416, Jerusalem, Israel, January 1999.
- [295] J. Pei, J. Han, H. J. Lu, S. Nishio, and S. Tang. H-Mine: Hyper-Structure Mining of Frequent Patterns in Large Databases. In *Proc. of the 2001 IEEE Intl. Conf. on Data Mining*, pages 441–448, San Jose, CA, November 2001.
- [296] J. Pei, J. Han, B. Mortazavi-Asl, and H. Zhu. Mining Access Patterns Efficiently from Web Logs. In *Proc. of the 4th Pacific-Asia Conf. on Knowledge Discovery and Data Mining*, pages 396–407, Kyoto, Japan, April 2000.
- [297] G. Piatetsky-Shapiro. Discovery, Analysis and Presentation of Strong Rules. In G. Piatetsky-Shapiro and W. Frawley, editors, *Knowledge Discovery in Databases*, pages 229–248. MIT Press, Cambridge, MA, 1991.
- [298] C. Potter, S. Klooster, M. Steinbach, P. N. Tan, V. Kumar, S. Shekhar, and C. Carvalho. Understanding Global Teleconnections of Climate to Regional Model Estimates of Amazon Ecosystem Carbon Fluxes. *Global Change Biology*, 10(5):693–703, 2004.
- [299] C. Potter, S. Klooster, M. Steinbach, P. N. Tan, V. Kumar, S. Shekhar, R. Myneni, and R. Nemani. Global Teleconnections of Ocean Climate to Terrestrial Carbon Flux. *J. Geophysical Research*, 108(D17), 2003.
- [300] G. D. Ramkumar, S. Ranka, and S. Tsur. Weighted Association Rules: Model and Algorithm. <http://www.cs.ucla.edu/~czdemo/tsur/>, 1997.
- [301] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating Mining with Relational Database Systems: Alternatives and Implications. In *Proc. of 1998 ACM-SIGMOD Intl. Conf. on Management of Data*, pages 343–354, Seattle, WA, 1998.
- [302] K. Satou, G. Shibayama, T. Ono, Y. Yamamura, E. Furuichi, S. Kuhara, and T. Takagi. Finding Association Rules on Heterogeneous Genome Data. In *Proc. of the Pacific Symp. on Biocomputing*, pages 397–408, Hawaii, January 1997.
- [303] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc. of the 21st Int. Conf. on Very Large Databases (VLDB'95)*, pages 432–444, Zurich, Switzerland, September 1995.
- [304] A. Savasere, E. Omiecinski, and S. Navathe. Mining for Strong Negative Associations in a Large Database of Customer Transactions. In *Proc. of the 14th Intl. Conf. on Data Engineering*, pages 494–502, Orlando, Florida, February 1998.
- [305] M. Seno and G. Karypis. LPMiner: An Algorithm for Finding Frequent Itemsets Using Length-Decreasing Support Constraint. In *Proc. of the 2001 IEEE Intl. Conf. on Data Mining*, pages 505–512, San Jose, CA, November 2001.
- [306] T. Shintani and M. Kitsuregawa. Hash based parallel algorithms for mining association rules. In *Proc. of the 4th Intl. Conf. on Parallel and Distributed Info. Systems*, pages 19–30, Miami Beach, FL, December 1996.
- [307] A. Silberschatz and A. Tuzhilin. What makes patterns interesting in knowledge discovery systems. *IEEE Trans. on Knowledge and Data Engineering*, 8(6):970–974, 1996.
- [308] C. Silverstein, S. Brin, and R. Motwani. Beyond market baskets: Generalizing association rules to dependence rules. *Data Mining and Knowledge Discovery*, 2(1):39–68, 1998.

- [309] E.-H. Simpson. The Interpretation of Interaction in Contingency Tables. *Journal of the Royal Statistical Society, B*(13):238–241, 1951.
- [310] L. Singh, B. Chen, R. Haight, and P. Scheuermann. An Algorithm for Constrained Association Rule Mining in Semi-structured Data. In *Proc. of the 3rd Pacific-Asia Conf. on Knowledge Discovery and Data Mining*, pages 148–158, Beijing, China, April 1999.
- [311] R. Srikant and R. Agrawal. Mining Quantitative Association Rules in Large Relational Tables. In *Proc. of 1996 ACM-SIGMOD Intl. Conf. on Management of Data*, pages 1–12, Montreal, Canada, 1996.
- [312] R. Srikant and R. Agrawal. Mining Sequential Patterns: Generalizations and Performance Improvements. In *Proc. of the 5th Intl. Conf. on Extending Database Technology (EDBT'96)*, pages 18–32, Avignon, France, 1996.
- [313] R. Srikant, Q. Vu, and R. Agrawal. Mining Association Rules with Item Constraints. In *Proc. of the 3rd Intl. Conf. on Knowledge Discovery and Data Mining*, pages 67–73, Newport Beach, CA, August 1997.
- [314] M. Steinbach, P. N. Tan, and V. Kumar. Support Envelopes: A Technique for Exploring the Structure of Association Patterns. In *Proc. of the 10th Intl. Conf. on Knowledge Discovery and Data Mining*, pages 296–305, Seattle, WA, August 2004.
- [315] M. Steinbach, P. N. Tan, H. Xiong, and V. Kumar. Extending the Notion of Support. In *Proc. of the 10th Intl. Conf. on Knowledge Discovery and Data Mining*, pages 689–694, Seattle, WA, August 2004.
- [316] E. Suzuki. Autonomous Discovery of Reliable Exception Rules. In *Proc. of the 3rd Intl. Conf. on Knowledge Discovery and Data Mining*, pages 259–262, Newport Beach, CA, August 1997.
- [317] P. N. Tan and V. Kumar. Mining Association Patterns in Web Usage Data. In *Proc. of the Intl. Conf. on Advances in Infrastructure for e-Business, e-Education, e-Science and e-Medicine on the Internet*, L’Aquila, Italy, January 2002.
- [318] P. N. Tan, V. Kumar, and J. Srivastava. Selecting the Right Interestingness Measure for Association Patterns. In *Proc. of the 8th Intl. Conf. on Knowledge Discovery and Data Mining*, pages 32–41, Edmonton, Canada, July 2002.
- [319] P. N. Tan, M. Steinbach, V. Kumar, S. Klooster, C. Potter, and A. Torregrossa. Finding Spatio-Temporal Patterns in Earth Science Data. In *KDD 2001 Workshop on Temporal Data Mining*, San Francisco, CA, 2001.
- [320] H. Toivonen. Sampling Large Databases for Association Rules. In *Proc. of the 22nd VLDB Conf.*, pages 134–145, Bombay, India, 1996.
- [321] H. Toivonen, M. Klemettinen, P. Ronkainen, K. Hatonen, and H. Mannila. Pruning and Grouping Discovered Association Rules. In *ECML-95 Workshop on Statistics, Machine Learning and Knowledge Discovery in Databases*, pages 47 – 52, Heraklion, Greece, April 1995.
- [322] S. Tsur, J. Ullman, S. Abiteboul, C. Clifton, R. Motwani, S. Nestorov, and A. Rosenthal. Query Flocks: A Generalization of Association Rule Mining. In *Proc. of 1998 ACM-SIGMOD Intl. Conf. on Management of Data*, pages 1–12, Seattle, WA, June 1998.
- [323] A. Tung, H. J. Lu, J. Han, and L. Feng. Breaking the Barrier of Transactions: Mining Inter-Transaction Association Rules. In *Proc. of the 5th Intl. Conf. on Knowledge Discovery and Data Mining*, pages 297–301, San Diego, CA, August 1999.
- [324] K. Wang, Y. He, and J. Han. Mining Frequent Itemsets Using Support Constraints. In *Proc. of the 26th VLDB Conf.*, pages 43–52, Cairo, Egypt, September 2000.

- [325] K. Wang, S. H. Tay, and B. Liu. Interestingness-Based Interval Merger for Numeric Association Rules. In *Proc. of the 4th Intl. Conf. on Knowledge Discovery and Data Mining*, pages 121–128, New York, NY, August 1998.
- [326] G. I. Webb. Preliminary investigations into statistically valid exploratory rule discovery. In *Proc. of the Australasian Data Mining Workshop (AusDM03)*, Canberra, Australia, December 2003.
- [327] H. Xiong, X. He, C. Ding, Y. Zhang, V. Kumar, and S. R. Holbrook. Identification of Functional Modules in Protein Complexes via Hyperclique Pattern Discovery. In *Proc. of the Pacific Symposium on Biocomputing, (PSB 2005)*, Maui, January 2005.
- [328] H. Xiong, S. Shekhar, P. N. Tan, and V. Kumar. Exploiting a Support-based Upper Bound of Pearson’s Correlation Coefficient for Efficiently Identifying Strongly Correlated Pairs. In *Proc. of the 10th Intl. Conf. on Knowledge Discovery and Data Mining*, pages 334–343, Seattle, WA, August 2004.
- [329] H. Xiong, M. Steinbach, P. N. Tan, and V. Kumar. HICAP: Hierarchical Clustering with Pattern Preservation. In *Proc. of the SIAM Intl. Conf. on Data Mining*, pages 279–290, Orlando, FL, April 2004.
- [330] H. Xiong, P. N. Tan, and V. Kumar. Mining Strong Affinity Association Patterns in Data Sets with Skewed Support Distribution. In *Proc. of the 2003 IEEE Intl. Conf. on Data Mining*, pages 387–394, Melbourne, FL, 2003.
- [331] X. Yan and J. Han. gSpan: Graph-based Substructure Pattern Mining. In *Proc. of the 2002 IEEE Intl. Conf. on Data Mining*, pages 721–724, Maebashi City, Japan, December 2002.
- [332] C. Yang, U. M. Fayyad, and P. S. Bradley. Efficient discovery of error-tolerant frequent itemsets in high dimensions. In *Proc. of the 7th Intl. Conf. on Knowledge Discovery and Data Mining*, pages 194–203, San Francisco, CA, August 2001.
- [333] M. J. Zaki. Parallel and Distributed Association Mining: A Survey. *IEEE Concurrency, special issue on Parallel Mechanisms for Data Mining*, 7(4):14–25, December 1999.
- [334] M. J. Zaki. Generating Non-Redundant Association Rules. In *Proc. of the 6th Intl. Conf. on Knowledge Discovery and Data Mining*, pages 34–43, Boston, MA, August 2000.
- [335] M. J. Zaki. Efficiently mining frequent trees in a forest. In *Proc. of the 8th Intl. Conf. on Knowledge Discovery and Data Mining*, pages 71–80, Edmonton, Canada, July 2002.
- [336] M. J. Zaki and M. Orihara. Theoretical foundations of association rules. In *Proc. of the 1998 ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, Seattle, WA, June 1998.
- [337] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New Algorithms for Fast Discovery of Association Rules. In *Proc. of the 3rd Intl. Conf. on Knowledge Discovery and Data Mining*, pages 283–286, Newport Beach, CA, August 1997.
- [338] H. Zhang, B. Padmanabhan, and A. Tuzhilin. On the Discovery of Significant Statistical Quantitative Rules. In *Proc. of the 10th Intl. Conf. on Knowledge Discovery and Data Mining*, pages 374–383, Seattle, WA, August 2004.
- [339] Z. Zhang, Y. Lu, and B. Zhang. An Effective Partitioning-Combining Algorithm for Discovering Quantitative Association Rules. In *Proc. of the 1st Pacific-Asia Conf. on Knowledge Discovery and Data Mining*, Singapore, 1997.
- [340] N. Zhong, Y. Y. Yao, and S. Ohsuga. Peculiarity Oriented Multi-database Mining. In *Proc. of the 3rd European Conf. of Principles and Practice of Knowledge Discovery in Databases*, pages 136–146, Prague, Czech Republic, 1999.

6.10 Exercises

1. For each of the following questions, provide an example of an association rule from the market basket domain that satisfies the following conditions. Also, describe whether such rules are subjectively interesting.
 - (a) A rule that has high support and high confidence.
 - (b) A rule that has reasonably high support but low confidence.
 - (c) A rule that has low support and low confidence.
 - (d) A rule that has low support and high confidence.
2. Consider the data set shown in Table 6.22.

Table 6.22. Example of market basket transactions.

Customer ID	Transaction ID	Items Bought
1	0001	{a, d, e}
1	0024	{a, b, c, e}
2	0012	{a, b, d, e}
2	0031	{a, c, d, e}
3	0015	{b, c, e}
3	0022	{b, d, e}
4	0029	{c, d}
4	0040	{a, b, c}
5	0033	{a, d, e}
5	0038	{a, b, e}

- (a) Compute the support for itemsets $\{e\}$, $\{b, d\}$, and $\{b, d, e\}$ by treating each transaction ID as a market basket.
- (b) Use the results in part (a) to compute the confidence for the association rules $\{b, d\} \rightarrow \{e\}$ and $\{e\} \rightarrow \{b, d\}$. Is confidence a symmetric measure?
- (c) Repeat part (a) by treating each customer ID as a market basket. Each item should be treated as a binary variable (1 if an item appears in at least one transaction bought by the customer, and 0 otherwise.)
- (d) Use the results in part (c) to compute the confidence for the association rules $\{b, d\} \rightarrow \{e\}$ and $\{e\} \rightarrow \{b, d\}$.
- (e) Suppose s_1 and c_1 are the support and confidence values of an association rule r when treating each transaction ID as a market basket. Also, let s_2 and c_2 be the support and confidence values of r when treating each customer ID as a market basket. Discuss whether there are any relationships between s_1 and s_2 or c_1 and c_2 .

3. (a) What is the confidence for the rules $\emptyset \rightarrow A$ and $A \rightarrow \emptyset$?
- (b) Let c_1 , c_2 , and c_3 be the confidence values of the rules $\{p\} \rightarrow \{q\}$, $\{p\} \rightarrow \{q, r\}$, and $\{p, r\} \rightarrow \{q\}$, respectively. If we assume that c_1 , c_2 , and c_3 have different values, what are the possible relationships that may exist among c_1 , c_2 , and c_3 ? Which rule has the lowest confidence?
- (c) Repeat the analysis in part (b) assuming that the rules have identical support. Which rule has the highest confidence?
- (d) Transitivity: Suppose the confidence of the rules $A \rightarrow B$ and $B \rightarrow C$ are larger than some threshold, minconf . Is it possible that $A \rightarrow C$ has a confidence less than minconf ?
4. For each of the following measures, determine whether it is monotone, anti-monotone, or non-monotone (i.e., neither monotone nor anti-monotone).

Example: Support, $s = \frac{\sigma(X)}{|T|}$ is anti-monotone because $s(X) \geq s(Y)$ whenever $X \subset Y$.

- (a) A characteristic rule is a rule of the form $\{p\} \rightarrow \{q_1, q_2, \dots, q_n\}$, where the rule antecedent contains only a single item. An itemset of size k can produce up to k characteristic rules. Let ζ be the minimum confidence of all characteristic rules generated from a given itemset:

$$\zeta(\{p_1, p_2, \dots, p_k\}) = \min [c(\{p_1\} \rightarrow \{p_2, p_3, \dots, p_k\}), \dots, c(\{p_k\} \rightarrow \{p_1, p_2, \dots, p_{k-1}\})]$$

Is ζ monotone, anti-monotone, or non-monotone?

- (b) A discriminant rule is a rule of the form $\{p_1, p_2, \dots, p_n\} \rightarrow \{q\}$, where the rule consequent contains only a single item. An itemset of size k can produce up to k discriminant rules. Let η be the minimum confidence of all discriminant rules generated from a given itemset:

$$\eta(\{p_1, p_2, \dots, p_k\}) = \min [c(\{p_2, p_3, \dots, p_k\} \rightarrow \{p_1\}), \dots, c(\{p_1, p_2, \dots, p_{k-1}\} \rightarrow \{p_k\})]$$

Is η monotone, anti-monotone, or non-monotone?

- (c) Repeat the analysis in parts (a) and (b) by replacing the min function with a max function.
5. Prove Equation 6.3. (Hint: First, count the number of ways to create an itemset that forms the left hand side of the rule. Next, for each size k itemset selected for the left-hand side, count the number of ways to choose the remaining $d - k$ items to form the right-hand side of the rule.)

Table 6.23. Market basket transactions.

Transaction ID	Items Bought
1	{Milk, Beer, Diapers}
2	{Bread, Butter, Milk}
3	{Milk, Diapers, Cookies}
4	{Bread, Butter, Cookies}
5	{Beer, Cookies, Diapers}
6	{Milk, Diapers, Bread, Butter}
7	{Bread, Butter, Diapers}
8	{Beer, Diapers}
9	{Milk, Diapers, Bread, Butter}
10	{Beer, Cookies}

6. Consider the market basket transactions shown in Table 6.23.
- What is the maximum number of association rules that can be extracted from this data (including rules that have zero support)?
 - What is the maximum size of frequent itemsets that can be extracted (assuming $minsup > 0$)?
 - Write an expression for the maximum number of size-3 itemsets that can be derived from this data set.
 - Find an itemset (of size 2 or larger) that has the largest support.
 - Find a pair of items, a and b , such that the rules $\{a\} \rightarrow \{b\}$ and $\{b\} \rightarrow \{a\}$ have the same confidence.
7. Consider the following set of frequent 3-itemsets:
- $$\{1, 2, 3\}, \{1, 2, 4\}, \{1, 2, 5\}, \{1, 3, 4\}, \{1, 3, 5\}, \{2, 3, 4\}, \{2, 3, 5\}, \{3, 4, 5\}.$$
- Assume that there are only five items in the data set.
- List all candidate 4-itemsets obtained by a candidate generation procedure using the $F_{k-1} \times F_1$ merging strategy.
 - List all candidate 4-itemsets obtained by the candidate generation procedure in *Apriori*.
 - List all candidate 4-itemsets that survive the candidate pruning step of the *Apriori* algorithm.
8. The *Apriori* algorithm uses a generate-and-count strategy for deriving frequent itemsets. Candidate itemsets of size $k + 1$ are created by joining a pair of frequent itemsets of size k (this is known as the candidate generation step). A candidate is discarded if any one of its subsets is found to be infrequent during the candidate pruning step. Suppose the *Apriori* algorithm is applied to the

Table 6.24. Example of market basket transactions.

Transaction ID	Items Bought
1	$\{a, b, d, e\}$
2	$\{b, c, d\}$
3	$\{a, b, d, e\}$
4	$\{a, c, d, e\}$
5	$\{b, c, d, e\}$
6	$\{b, d, e\}$
7	$\{c, d\}$
8	$\{a, b, c\}$
9	$\{a, d, e\}$
10	$\{b, d\}$

data set shown in Table 6.24 with $minsup = 30\%$, i.e., any itemset occurring in less than 3 transactions is considered to be infrequent.

- (a) Draw an itemset lattice representing the data set given in Table 6.24. Label each node in the lattice with the following letter(s):
- **N:** If the itemset is not considered to be a candidate itemset by the *Apriori* algorithm. There are two reasons for an itemset not to be considered as a candidate itemset: (1) it is not generated at all during the candidate generation step, or (2) it is generated during the candidate generation step but is subsequently removed during the candidate pruning step because one of its subsets is found to be infrequent.
 - **F:** If the candidate itemset is found to be frequent by the *Apriori* algorithm.
 - **I:** If the candidate itemset is found to be infrequent after support counting.
- (b) What is the percentage of frequent itemsets (with respect to all itemsets in the lattice)?
- (c) What is the pruning ratio of the *Apriori* algorithm on this data set? (Pruning ratio is defined as the percentage of itemsets not considered to be a candidate because (1) they are not generated during candidate generation or (2) they are pruned during the candidate pruning step.)
- (d) What is the false alarm rate (i.e, percentage of candidate itemsets that are found to be infrequent after performing support counting)?
9. The *Apriori* algorithm uses a hash tree data structure to efficiently count the support of candidate itemsets. Consider the hash tree for candidate 3-itemsets shown in Figure 6.32.

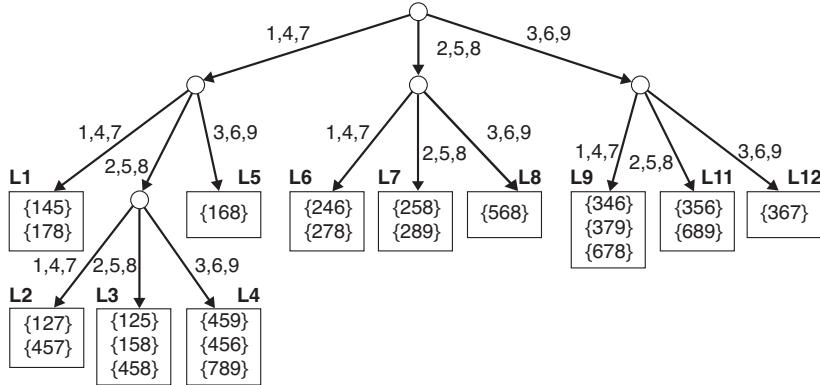


Figure 6.32. An example of a hash tree structure.

- (a) Given a transaction that contains items $\{1, 3, 4, 5, 8\}$, which of the hash tree leaf nodes will be visited when finding the candidates of the transaction?
- (b) Use the visited leaf nodes in part (b) to determine the candidate itemsets that are contained in the transaction $\{1, 3, 4, 5, 8\}$.
10. Consider the following set of candidate 3-itemsets:
 $\{1, 2, 3\}, \{1, 2, 6\}, \{1, 3, 4\}, \{2, 3, 4\}, \{2, 4, 5\}, \{3, 4, 6\}, \{4, 5, 6\}$
- (a) Construct a hash tree for the above candidate 3-itemsets. Assume the tree uses a hash function where all odd-numbered items are hashed to the left child of a node, while the even-numbered items are hashed to the right child. A candidate k -itemset is inserted into the tree by hashing on each successive item in the candidate and then following the appropriate branch of the tree according to the hash value. Once a leaf node is reached, the candidate is inserted based on one of the following conditions:
- Condition 1:** If the depth of the leaf node is equal to k (the root is assumed to be at depth 0), then the candidate is inserted regardless of the number of itemsets already stored at the node.
- Condition 2:** If the depth of the leaf node is less than k , then the candidate can be inserted as long as the number of itemsets stored at the node is less than $maxsize$. Assume $maxsize = 2$ for this question.
- Condition 3:** If the depth of the leaf node is less than k and the number of itemsets stored at the node is equal to $maxsize$, then the leaf node is converted into an internal node. New leaf nodes are created as children of the old leaf node. Candidate itemsets previously stored

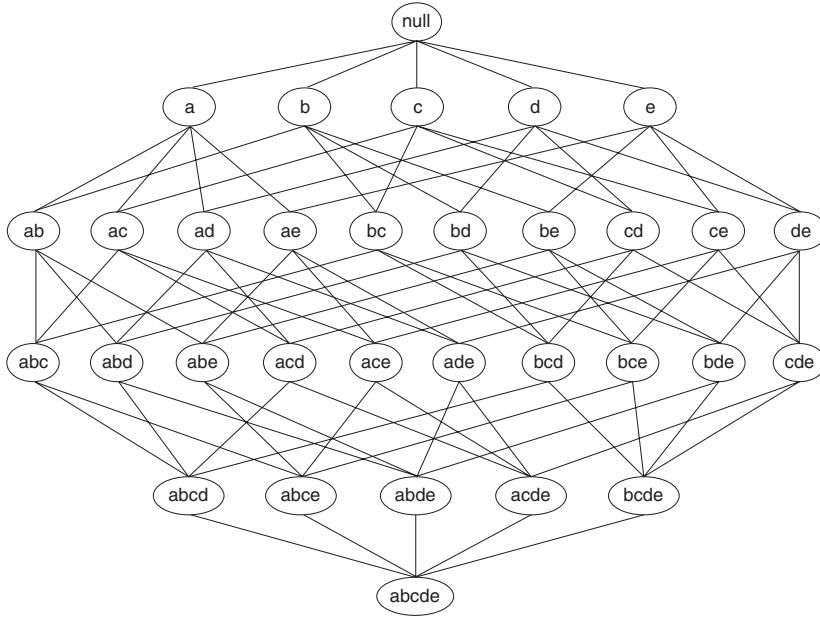


Figure 6.33. An itemset lattice

in the old leaf node are distributed to the children based on their hash values. The new candidate is also hashed to its appropriate leaf node.

- (b) How many leaf nodes are there in the candidate hash tree? How many internal nodes are there?
 - (c) Consider a transaction that contains the following items: $\{1, 2, 3, 5, 6\}$. Using the hash tree constructed in part (a), which leaf nodes will be checked against the transaction? What are the candidate 3-itemsets contained in the transaction?
11. Given the lattice structure shown in Figure 6.33 and the transactions given in Table 6.24, label each node with the following letter(s):

- M if the node is a maximal frequent itemset,
- C if it is a closed frequent itemset,
- N if it is frequent but neither maximal nor closed, and
- I if it is infrequent.

Assume that the support threshold is equal to 30%.

12. The original association rule mining formulation uses the support and confidence measures to prune uninteresting rules.

410 Chapter 6 Association Analysis

- (a) Draw a contingency table for each of the following rules using the transactions shown in Table 6.25.

Table 6.25. Example of market basket transactions.

Transaction ID	Items Bought
1	{a, b, d, e}
2	{b, c, d}
3	{a, b, d, e}
4	{a, c, d, e}
5	{b, c, d, e}
6	{b, d, e}
7	{c, d}
8	{a, b, c}
9	{a, d, e}
10	{b, d}

Rules: $\{b\} \rightarrow \{c\}$, $\{a\} \rightarrow \{d\}$, $\{b\} \rightarrow \{d\}$, $\{e\} \rightarrow \{c\}$, $\{c\} \rightarrow \{a\}$.

- (b) Use the contingency tables in part (a) to compute and rank the rules in decreasing order according to the following measures.

i. Support.

ii. Confidence.

iii. Interest($X \rightarrow Y$) = $\frac{P(X,Y)}{P(X)} P(Y)$.

iv. IS($X \rightarrow Y$) = $\frac{P(X,Y)}{\sqrt{P(X)P(Y)}}$.

v. Klosgen($X \rightarrow Y$) = $\sqrt{P(X, \overline{Y})} \times (P(Y|X) - P(Y))$, where $P(Y|X) = \frac{P(X,Y)}{P(X)}$.

vi. Odds ratio($X \rightarrow Y$) = $\frac{P(X,Y)P(\overline{X}, \overline{Y})}{P(X, \overline{Y})P(\overline{X}, Y)}$.

13. Given the rankings you had obtained in Exercise 12, compute the correlation between the rankings of confidence and the other five measures. Which measure is most highly correlated with confidence? Which measure is least correlated with confidence?
14. Answer the following questions using the data sets shown in Figure 6.34. Note that each data set contains 1000 items and 10,000 transactions. Dark cells indicate the presence of items and white cells indicate the absence of items. We will apply the *Apriori* algorithm to extract frequent itemsets with $minsup = 10\%$ (i.e., itemsets must be contained in at least 1000 transactions)?

- (a) Which data set(s) will produce the most number of frequent itemsets?

- (b) Which data set(s) will produce the fewest number of frequent itemsets?
 - (c) Which data set(s) will produce the longest frequent itemset?
 - (d) Which data set(s) will produce frequent itemsets with highest maximum support?
 - (e) Which data set(s) will produce frequent itemsets containing items with wide-varying support levels (i.e., items with mixed support, ranging from less than 20% to more than 70%).
15. (a) Prove that the ϕ coefficient is equal to 1 if and only if $f_{11} = f_{1+} = f_{+1}$.
- (b) Show that if A and B are independent, then $P(A, B) \times P(A, \bar{B}) = P(A, \bar{B}) \times P(\bar{A}, B)$.
- (c) Show that Yule's Q and Y coefficients
- $$Q = \left[\frac{f_{11}f_{00} - f_{10}f_{01}}{f_{11}f_{00} + f_{10}f_{01}} \right]$$
- $$Y = \left[\frac{\sqrt{f_{11}f_{00}} - \sqrt{f_{10}f_{01}}}{\sqrt{f_{11}f_{00}} + \sqrt{f_{10}f_{01}}} \right]$$
- are normalized versions of the odds ratio.
- (d) Write a simplified expression for the value of each measure shown in Tables 6.11 and 6.12 when the variables are statistically independent.
16. Consider the interestingness measure, $M = \frac{P(B|A) - P(B)}{1 - P(B)}$, for an association rule $A \rightarrow B$.
- (a) What is the range of this measure? When does the measure attain its maximum and minimum values?
 - (b) How does M behave when $P(A, B)$ is increased while $P(A)$ and $P(B)$ remain unchanged?
 - (c) How does M behave when $P(A)$ is increased while $P(A, B)$ and $P(B)$ remain unchanged?
 - (d) How does M behave when $P(B)$ is increased while $P(A, B)$ and $P(A)$ remain unchanged?
 - (e) Is the measure symmetric under variable permutation?
 - (f) What is the value of the measure when A and B are statistically independent?
 - (g) Is the measure null-invariant?
 - (h) Does the measure remain invariant under row or column scaling operations?
 - (i) How does the measure behave under the inversion operation?

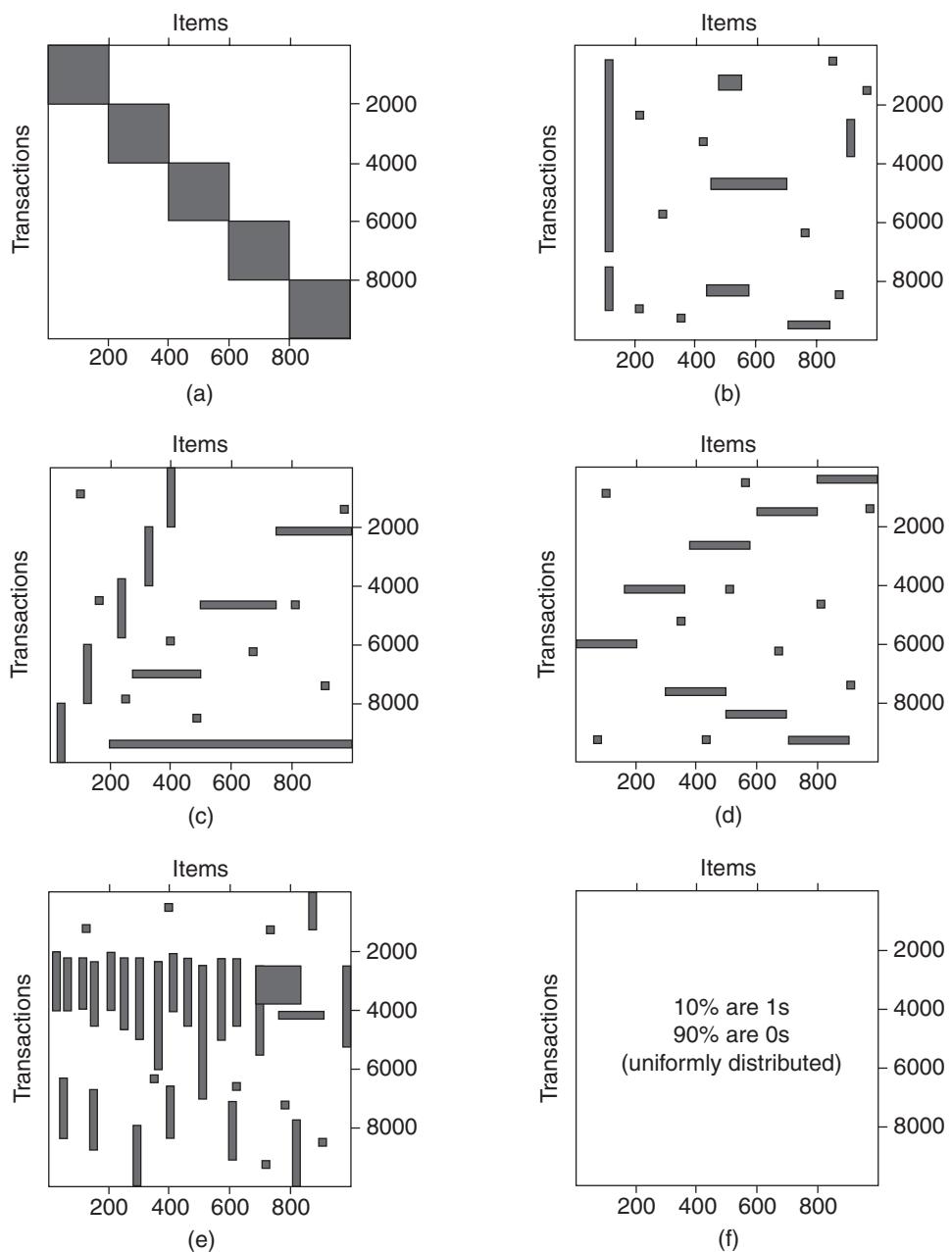


Figure 6.34. Figures for Exercise 14.

17. Suppose we have market basket data consisting of 100 transactions and 20 items. If the support for item a is 25%, the support for item b is 90% and the support for itemset $\{a, b\}$ is 20%. Let the support and confidence thresholds be 10% and 60%, respectively.
- Compute the confidence of the association rule $\{a\} \rightarrow \{b\}$. Is the rule interesting according to the confidence measure?
 - Compute the interest measure for the association pattern $\{a, b\}$. Describe the nature of the relationship between item a and item b in terms of the interest measure.
 - What conclusions can you draw from the results of parts (a) and (b)?
 - Prove that if the confidence of the rule $\{a\} \rightarrow \{b\}$ is less than the support of $\{b\}$, then:
 - $c(\{\bar{a}\} \rightarrow \{b\}) > c(\{\bar{a}\} \rightarrow \{b\})$,
 - $c(\{\bar{a}\} \rightarrow \{b\}) > s(\{b\})$,
 where $c(\cdot)$ denote the rule confidence and $s(\cdot)$ denote the support of an itemset.
18. Table 6.26 shows a $2 \times 2 \times 2$ contingency table for the binary variables A and B at different values of the control variable C .

Table 6.26. A Contingency Table.

		A		
		1	0	
C = 0	B	1	0	15
		0	15	30
C = 1	B	1	5	0
		0	0	15

- Compute the ϕ coefficient for A and B when $C = 0$, $C = 1$, and $C = 0$ or 1. Note that $\phi(\{A, B\}) = \frac{P(A, B) - P(A)P(B)}{\sqrt{P(A)P(B)(1-P(A))(1-P(B))}}$.
 - What conclusions can you draw from the above result?
19. Consider the contingency tables shown in Table 6.27.
- For table I, compute support, the interest measure, and the ϕ correlation coefficient for the association pattern $\{A, B\}$. Also, compute the confidence of rules $A \rightarrow B$ and $B \rightarrow A$.

Table 6.27. Contingency tables for Exercise 19.

	B	\bar{B}		B	\bar{B}
A	9	1	A	89	1
\bar{A}	1	89	\bar{A}	1	9

(a) Table I. (b) Table II.

- (b) For table II, compute support, the interest measure, and the ϕ correlation coefficient for the association pattern $\{A, B\}$. Also, compute the confidence of rules $A \rightarrow B$ and $B \rightarrow A$.
- (c) What conclusions can you draw from the results of (a) and (b)?
20. Consider the relationship between customers who buy high-definition televisions and exercise machines as shown in Tables 6.19 and 6.20.
- (a) Compute the odds ratios for both tables.
 - (b) Compute the ϕ -coefficient for both tables.
 - (c) Compute the interest factor for both tables.

For each of the measures given above, describe how the direction of association changes when data is pooled together instead of being stratified.

Data Mining Association Analysis: Basic Concepts and Algorithms

Lecture Notes for Chapter 6

Introduction to Data Mining

by

Tan, Steinbach, Kumar

Association Rule Mining

- Given a set of transactions, find rules that will predict the occurrence of an item based on the occurrences of other items in the transaction

Market-Basket transactions

TID	Items
1	Bread, Milk
2	Bread, Diaper, Beer, Eggs
3	Milk, Diaper, Beer, Coke
4	Bread, Milk, Diaper, Beer
5	Bread, Milk, Diaper, Coke

Example of Association Rules

$\{\text{Diaper}\} \rightarrow \{\text{Beer}\}$,
 $\{\text{Milk, Bread}\} \rightarrow \{\text{Eggs, Coke}\}$,
 $\{\text{Beer, Bread}\} \rightarrow \{\text{Milk}\}$,

Implication means co-occurrence,
not causality!

Definition: Frequent Itemset

● Itemset

- A collection of one or more items
 - ◆ Example: {Milk, Bread, Diaper}

- k-itemset

- ◆ An itemset that contains k items

● Support count (σ)

- Frequency of occurrence of an itemset
- E.g. $\sigma(\{\text{Milk, Bread, Diaper}\}) = 2$

● Support

- Fraction of transactions that contain an itemset
- E.g. $s(\{\text{Milk, Bread, Diaper}\}) = 2/5$

● Frequent Itemset

- An itemset whose support is greater than or equal to a $minsup$ threshold

TID	Items
1	Bread, Milk
2	Bread, Diaper, Beer, Eggs
3	Milk, Diaper, Beer, Coke
4	Bread, Milk, Diaper, Beer
5	Bread, Milk, Diaper, Coke

Definition: Association Rule

● Association Rule

- An implication expression of the form $X \rightarrow Y$, where X and Y are itemsets
- Example:
 $\{\text{Milk, Diaper}\} \rightarrow \{\text{Beer}\}$

TID	Items
1	Bread, Milk
2	Bread, Diaper, Beer, Eggs
3	Milk, Diaper, Beer, Coke
4	Bread, Milk, Diaper, Beer
5	Bread, Milk, Diaper, Coke

● Rule Evaluation Metrics

- Support (s)
 - ◆ Fraction of transactions that contain both X and Y
- Confidence (c)
 - ◆ Measures how often items in Y appear in transactions that contain X

Example:

$$\{\text{Milk, Diaper}\} \Rightarrow \text{Beer}$$

$$s = \frac{\sigma(\{\text{Milk, Diaper, Beer}\})}{|\text{T}|} = \frac{2}{5} = 0.4$$

$$c = \frac{\sigma(\{\text{Milk, Diaper, Beer}\})}{\sigma(\{\text{Milk, Diaper}\})} = \frac{2}{3} = 0.67$$

Association Rule Mining Task

- Given a set of transactions T, the goal of association rule mining is to find all rules having
 - support $\geq m_{insup}$ threshold
 - confidence $\geq m_{inconf}$ threshold
- Brute-force approach:
 - List all possible association rules
 - Compute the support and confidence for each rule
 - Prune rules that fail the m_{insup} and m_{inconf} thresholds

⇒ Computationally prohibitive!

Mining Association Rules

Example of Rules:

TID	Items
1	Bread, Milk
2	Bread, Diaper, Beer, Eggs
3	Milk, Diaper, Beer, Coke
4	Bread, Milk, Diaper, Beer
5	Bread, Milk, Diaper, Coke

$\{\text{Milk}, \text{Diaper}\} \rightarrow \{\text{Beer}\}$ (s=0.4, c=0.67)
 $\{\text{Milk}, \text{Beer}\} \rightarrow \{\text{Diaper}\}$ (s=0.4, c=1.0)
 $\{\text{Diaper}, \text{Beer}\} \rightarrow \{\text{Milk}\}$ (s=0.4, c=0.67)
 $\{\text{Beer}\} \rightarrow \{\text{Milk}, \text{Diaper}\}$ (s=0.4, c=0.67)
 $\{\text{Diaper}\} \rightarrow \{\text{Milk}, \text{Beer}\}$ (s=0.4, c=0.5)
 $\{\text{Milk}\} \rightarrow \{\text{Diaper}, \text{Beer}\}$ (s=0.4, c=0.5)

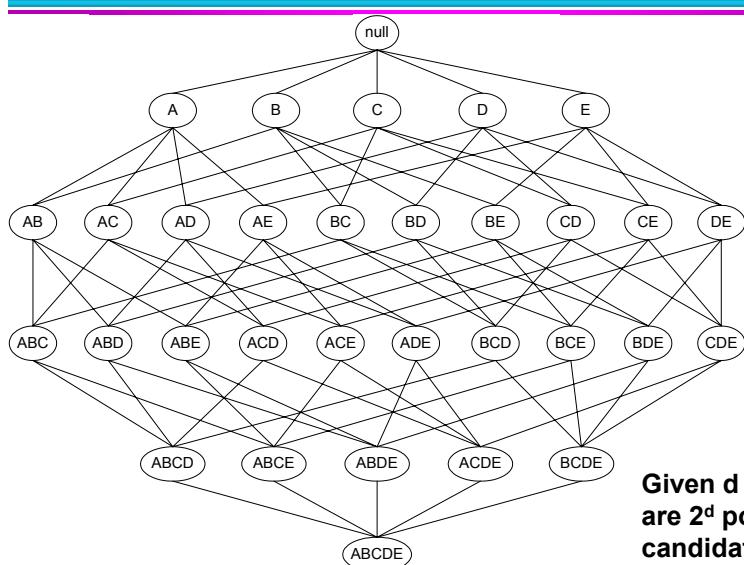
Observations:

- All the above rules are binary partitions of the same itemset:
 {Milk, Diaper, Beer}
- Rules originating from the same itemset have identical support but can have different confidence
- Thus, we may decouple the support and confidence requirements

Mining Association Rules

- Two-step approach:
 1. Frequent Itemset Generation
 - Generate all itemsets whose support $\geq \text{minsup}$
 2. Rule Generation
 - Generate high confidence rules from each frequent itemset, where each rule is a binary partitioning of a frequent itemset
- Frequent itemset generation is still computationally expensive

Frequent Itemset Generation

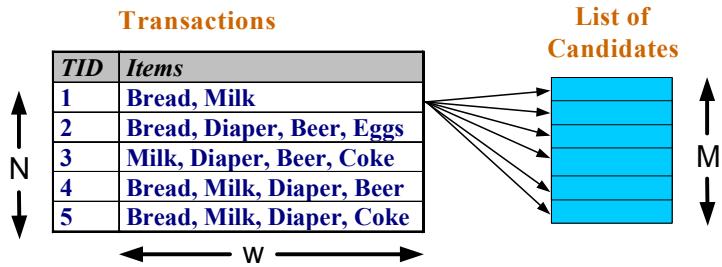


Given d items, there
are 2^d possible
candidate itemsets

Frequent Itemset Generation

- Brute-force approach:

- Each itemset in the lattice is a **candidate** frequent itemset
- Count the support of each candidate by scanning the database

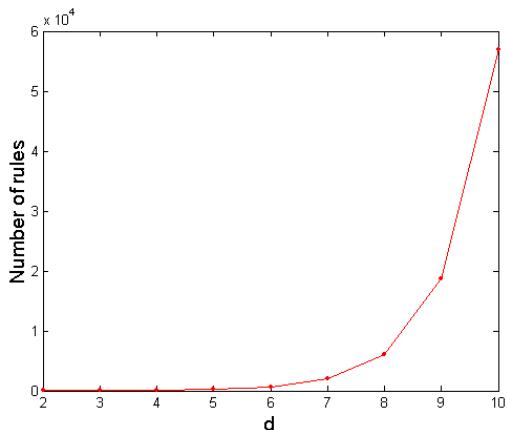


- Match each transaction against every candidate
- Complexity $\sim O(NMw)$ => **Expensive since $M = 2^d$!!!**

Computational Complexity

- Given d unique items:

- Total number of itemsets = 2^d
- Total number of possible association rules:



$$R = \sum_{k=1}^{d-1} \left[\binom{d}{k} \times \sum_{j=1}^{d-k} \binom{d-k}{j} \right] \\ = 3^d - 2^{d+1} + 1$$

If $d=6$, $R = 602$ rules

Frequent Itemset Generation Strategies

- Reduce the **number of candidates** (M)
 - Complete search: $M=2^d$
 - Use pruning techniques to reduce M
- Reduce the **number of transactions** (N)
 - Reduce size of N as the size of itemset increases
 - Used by DHP and vertical-based mining algorithms
- Reduce the **number of comparisons** (NM)
 - Use efficient data structures to store the candidates or transactions
 - No need to match every candidate against every transaction

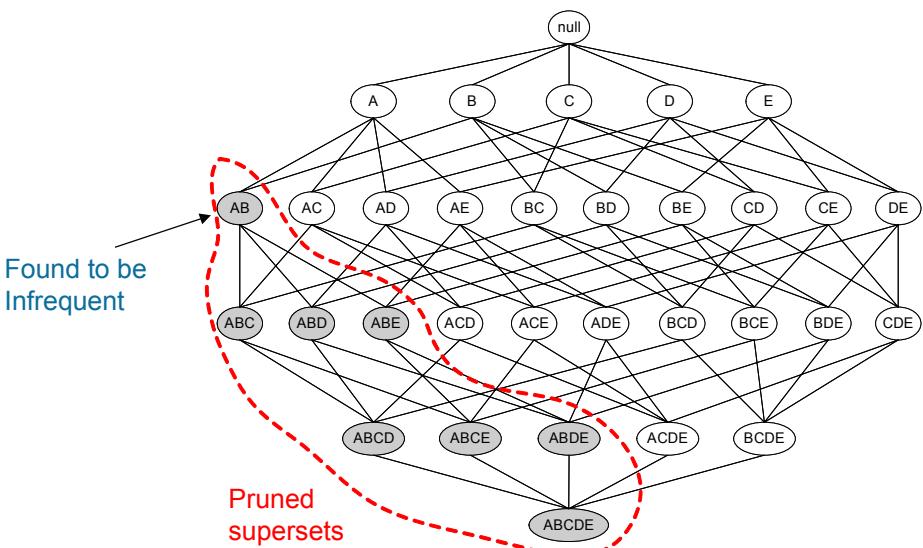
Reducing Number of Candidates

- **Apriori principle:**
 - If an itemset is frequent, then all of its subsets must also be frequent
- Apriori principle holds due to the following property of the support measure:

$$\forall X, Y : (X \subseteq Y) \Rightarrow s(X) \geq s(Y)$$

- Support of an itemset never exceeds the support of its subsets
- This is known as the **anti-monotone** property of support

Illustrating Apriori Principle



Illustrating Apriori Principle

Item	Count
Bread	4
Coke	2
Milk	4
Beer	3
Diaper	4
Eggs	1

Items (1-itemsets)



Itemset	Count
{Bread,Milk}	3
{Bread,Beer}	2
{Bread,Diaper}	3
{Milk,Beer}	2
{Milk,Diaper}	3
{Beer,Diaper}	3

Pairs (2-itemsets)

(No need to generate candidates involving Coke or Eggs)

Minimum Support = 3



Triplets (3-itemsets)

Itemset	Count
{Bread,Milk,Diaper}	3

If every subset is considered,
 ${}^6C_1 + {}^6C_2 + {}^6C_3 = 41$
With support-based pruning,
 $6 + 6 + 1 = 13$



Apriori Algorithm

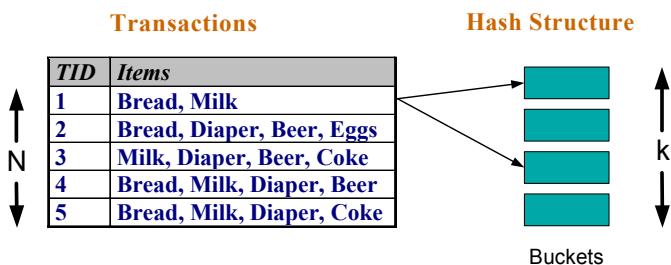
- Method:

- Let k=1
- Generate frequent itemsets of length 1
- Repeat until no new frequent itemsets are identified
 - Generate length (k+1) candidate itemsets from length k frequent itemsets
 - Prune candidate itemsets containing subsets of length k that are infrequent
 - Count the support of each candidate by scanning the DB
 - Eliminate candidates that are infrequent, leaving only those that are frequent

Reducing Number of Comparisons

- Candidate counting:

- Scan the database of transactions to determine the support of each candidate itemset
- To reduce the number of comparisons, store the candidates in a hash structure
 - Instead of matching each transaction against every candidate, match it against candidates contained in the hashed buckets



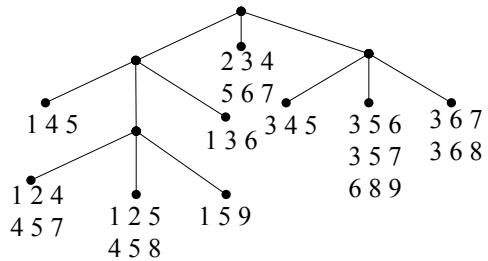
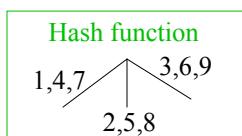
Generate Hash Tree

Suppose you have 15 candidate itemsets of length 3:

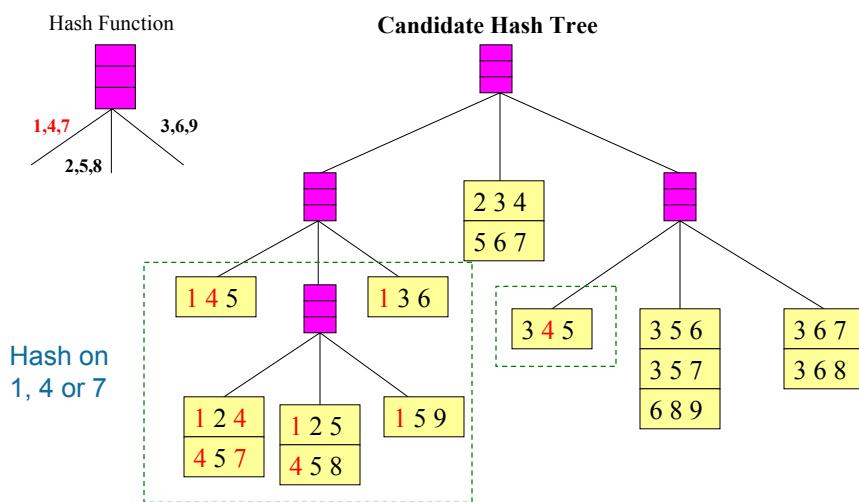
{1 4 5}, {1 2 4}, {4 5 7}, {1 2 5}, {4 5 8}, {1 5 9}, {1 3 6}, {2 3 4}, {5 6 7}, {3 4 5},
{3 5 6}, {3 5 7}, {6 8 9}, {3 6 7}, {3 6 8}

You need:

- Hash function
- Max leaf size: max number of itemsets stored in a leaf node (if number of candidate itemsets exceeds max leaf size, split the node)

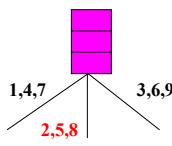


Association Rule Discovery: Hash tree

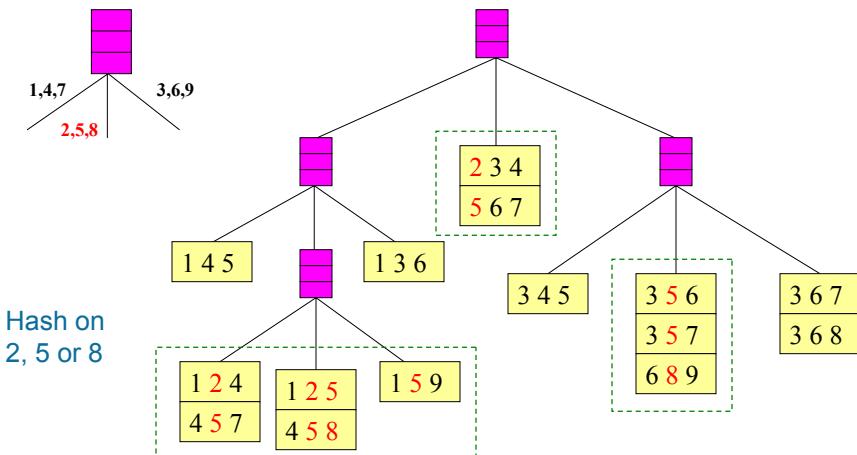


Association Rule Discovery: Hash tree

Hash Function



Candidate Hash Tree



Hash on
2, 5 or 8

© Tan, Steinbach, Kumar

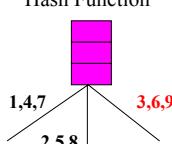
Introduction to Data Mining

4/18/2004

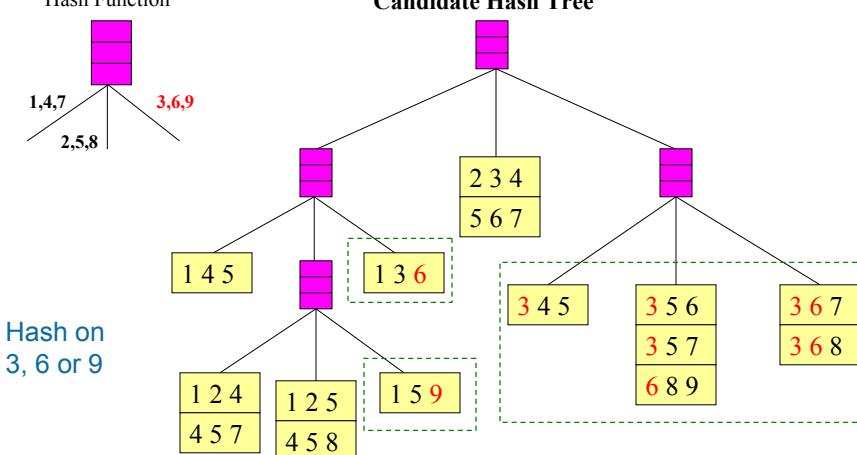
19

Association Rule Discovery: Hash tree

Hash Function



Candidate Hash Tree



Hash on
3, 6 or 9

© Tan, Steinbach, Kumar

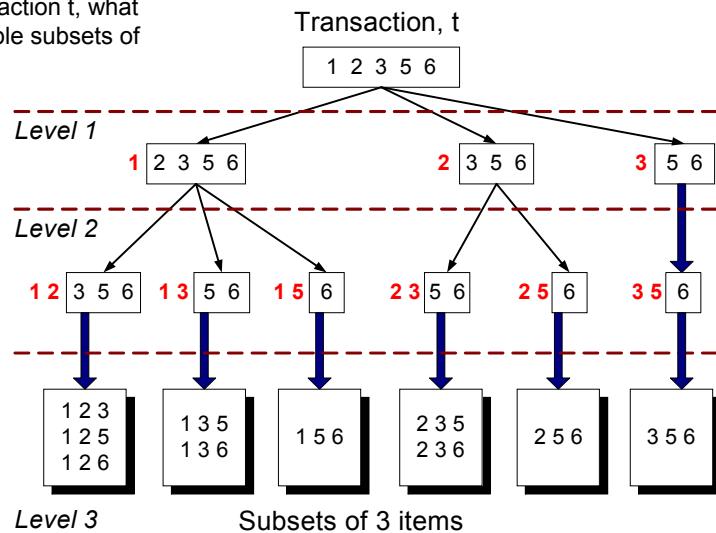
Introduction to Data Mining

4/18/2004

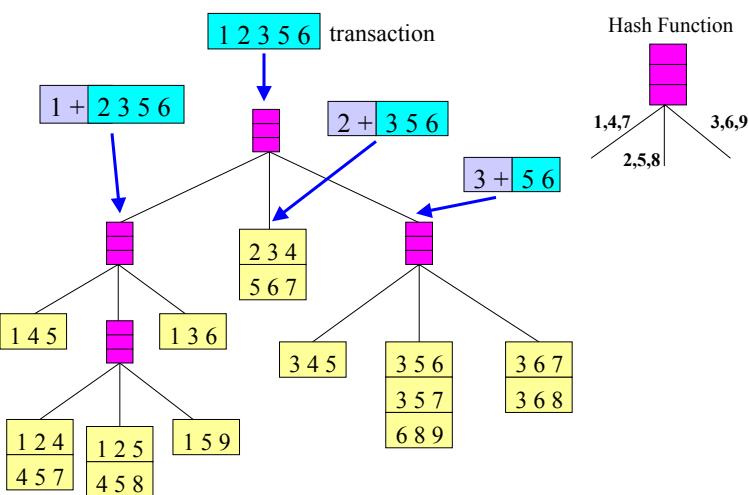
20

Subset Operation

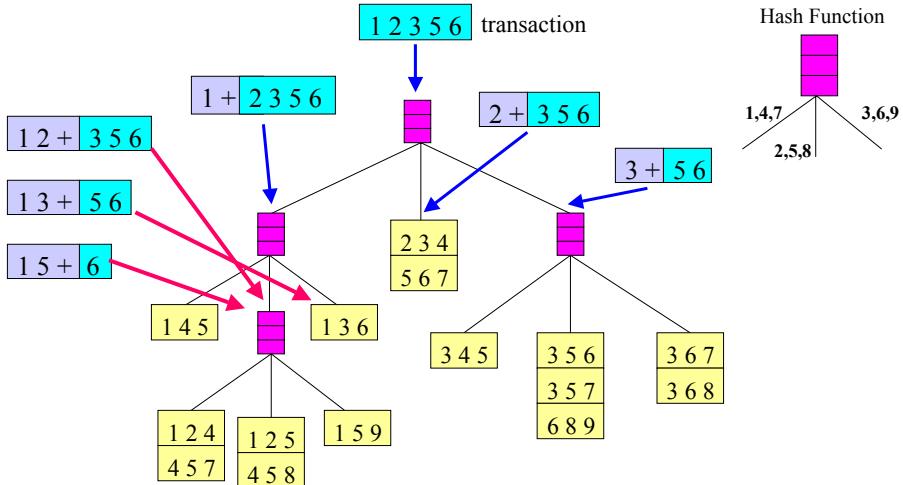
Given a transaction t , what are the possible subsets of size 3?



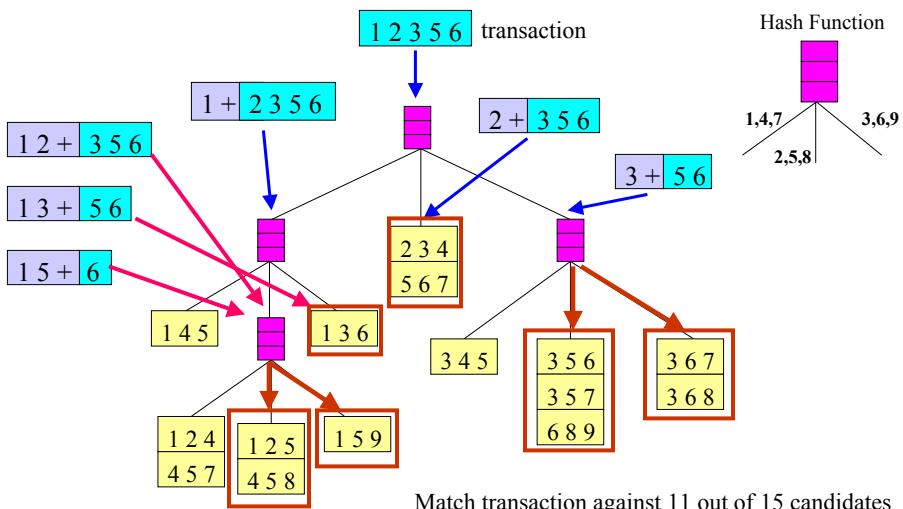
Subset Operation Using Hash Tree



Subset Operation Using Hash Tree



Subset Operation Using Hash Tree



Factors Affecting Complexity

- Choice of minimum support threshold
 - lowering support threshold results in more frequent itemsets
 - this may increase number of candidates and max length of frequent itemsets
- Dimensionality (number of items) of the data set
 - more space is needed to store support count of each item
 - if number of frequent items also increases, both computation and I/O costs may also increase
- Size of database
 - since Apriori makes multiple passes, run time of algorithm may increase with number of transactions
- Average transaction width
 - transaction width increases with denser data sets
 - This may increase max length of frequent itemsets and traversals of hash tree (number of subsets in a transaction increases with its width)

Compact Representation of Frequent Itemsets

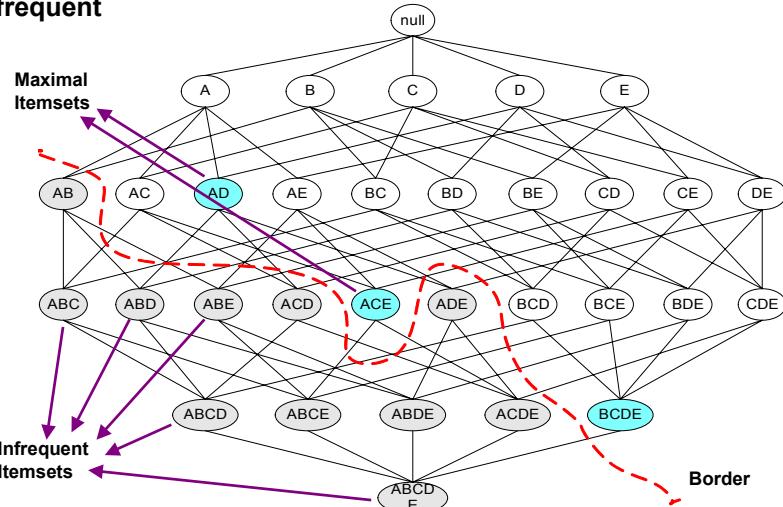
- Some itemsets are redundant because they have identical support as their supersets

TID	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
2	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
3	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
4	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
5	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
6	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0		
7	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0		
8	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0		
9	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0		
10	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0		
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1		
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1		
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1		
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1		
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1		

- Number of frequent itemsets = $3 \times \sum_{k=1}^{10} \binom{10}{k}$
- Need a compact representation

Maximal Frequent Itemset

An itemset is maximal frequent if none of its immediate supersets is frequent



Closed Itemset

- An itemset is closed if none of its immediate supersets has the same support as the itemset

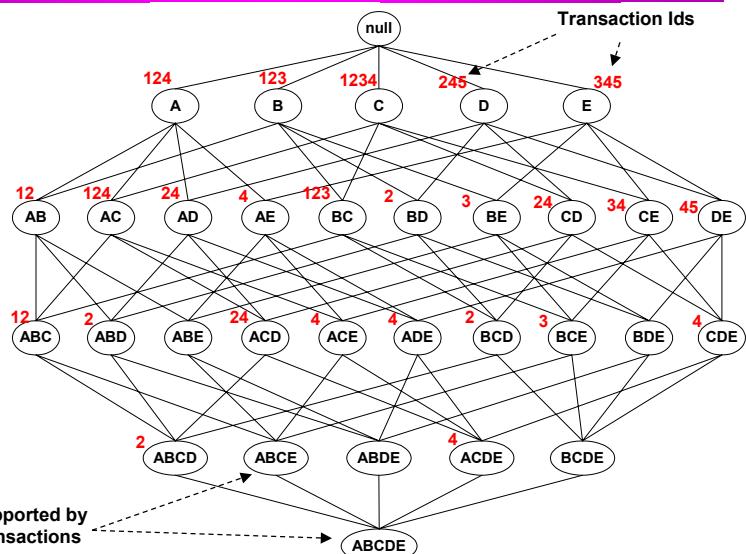
TID	Items
1	{A,B}
2	{B,C,D}
3	{A,B,C,D}
4	{A,B,D}
5	{A,B,C,D}

Itemset	Support
{A}	4
{B}	5
{C}	3
{D}	4
{A,B}	4
{A,C}	2
{A,D}	3
{B,C}	3
{B,D}	4
{C,D}	3

Itemset	Support
{A,B,C}	2
{A,B,D}	3
{A,C,D}	2
{B,C,D}	3
{A,B,C,D}	2

Maximal vs Closed Itemsets

TID	Items
1	ABC
2	ABCD
3	BCE
4	ACDE
5	DE



© Tan, Steinbach, Kumar

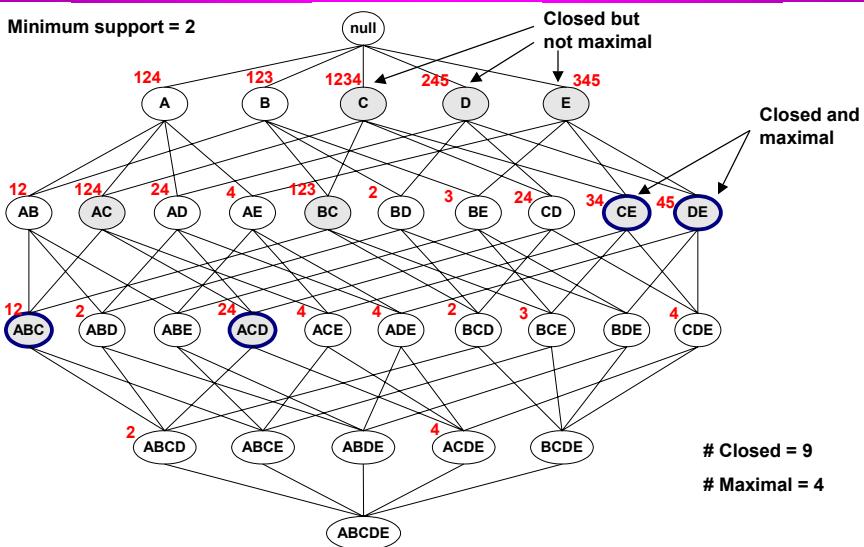
Introduction to Data Mining

4/18/2004

29

Maximal vs Closed Frequent Itemsets

Minimum support = 2



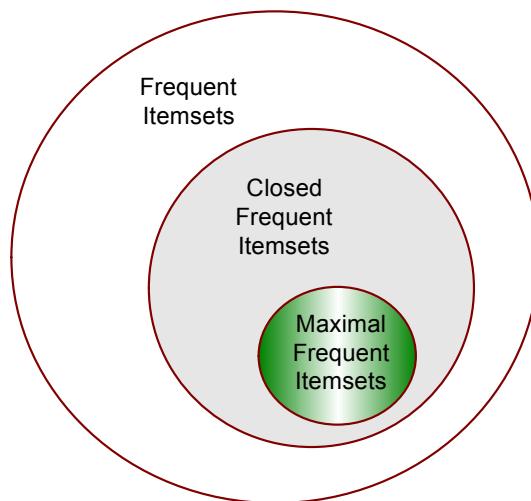
© Tan, Steinbach, Kumar

Introduction to Data Mining

4/18/2004

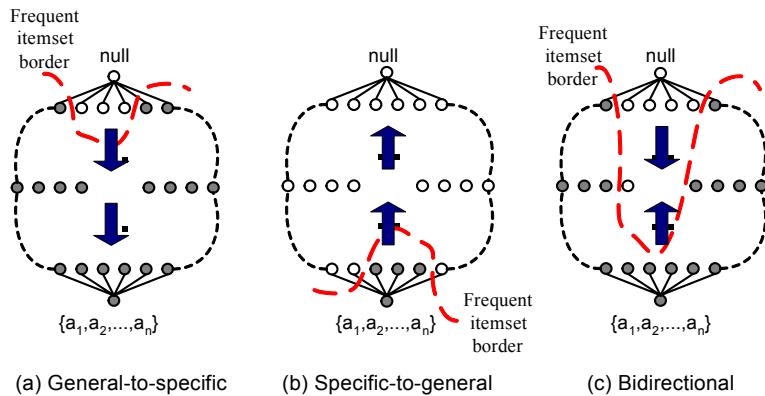
30

Maximal vs Closed Itemsets



Alternative Methods for Frequent Itemset Generation

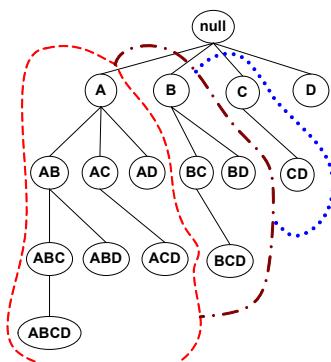
- Traversal of Itemset Lattice
 - General-to-specific vs Specific-to-general



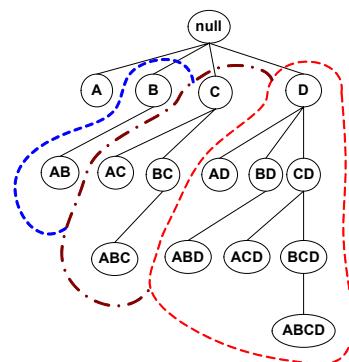
Alternative Methods for Frequent Itemset Generation

- Traversal of Itemset Lattice

- Equivalent Classes



(a) Prefix tree

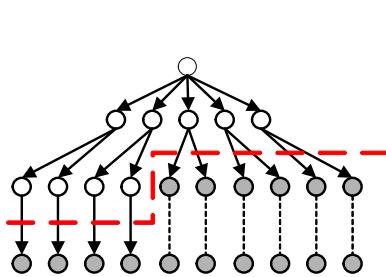


(b) Suffix tree

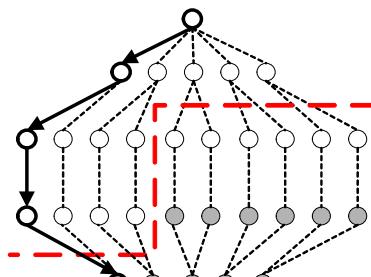
Alternative Methods for Frequent Itemset Generation

- Traversal of Itemset Lattice

- Breadth-first vs Depth-first



(a) Breadth first



(b) Depth first

Alternative Methods for Frequent Itemset Generation

- Representation of Database

- horizontal vs vertical data layout

Horizontal
Data Layout

TID	Items
1	A,B,E
2	B,C,D
3	C,E
4	A,C,D
5	A,B,C,D
6	A,E
7	A,B
8	A,B,C
9	A,C,D
10	B

Vertical Data Layout

A	B	C	D	E
1	1	2	2	1
4	2	3	4	3
5	5	4	5	6
6	7	8	9	
7	8	9		
8	10			
9				

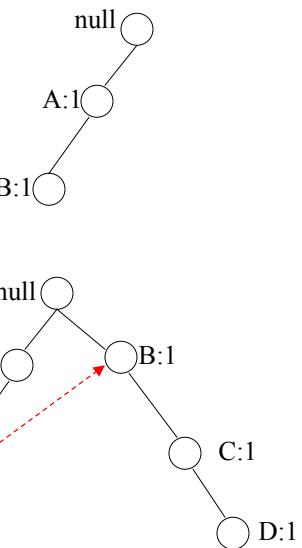
FP-growth Algorithm

- Use a compressed representation of the database using an **FP-tree**
- Once an FP-tree has been constructed, it uses a recursive divide-and-conquer approach to mine the frequent itemsets

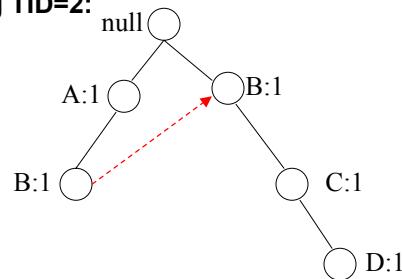
FP-tree construction

After reading TID=1:

TID	Items
1	{A,B}
2	{B,C,D}
3	{A,C,D,E}
4	{A,D,E}
5	{A,B,C}
6	{A,B,C,D}
7	{B,C}
8	{A,B,C}
9	{A,B,D}
10	{B,C,E}



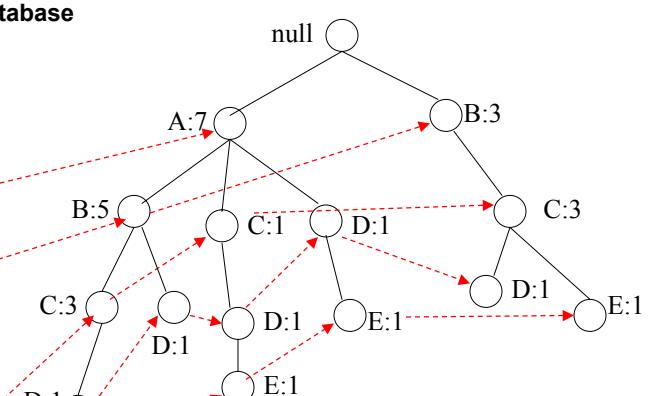
After reading TID=2:



FP-Tree Construction

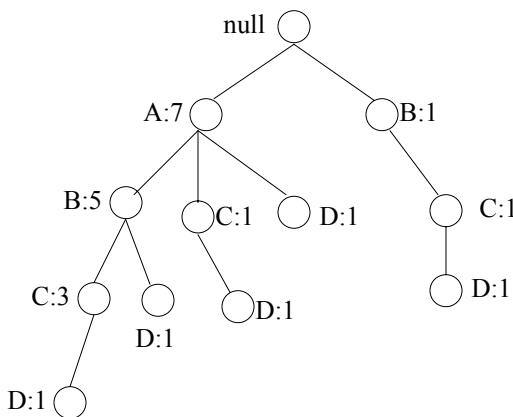
Transaction Database

TID	Items
1	{A,B}
2	{B,C,D}
3	{A,C,D,E}
4	{A,D,E}
5	{A,B,C}
6	{A,B,C,D}
7	{B,C}
8	{A,B,C}
9	{A,B,D}
10	{B,C,E}



Pointers are used to assist frequent itemset generation

FP-growth



Conditional Pattern base
for D:

$$P = \{(A:1, B:1, C:1), (A:1, B:1), (A:1, C:1), (A:1), (B:1, C:1)\}$$

Recursively apply FP-
growth on P

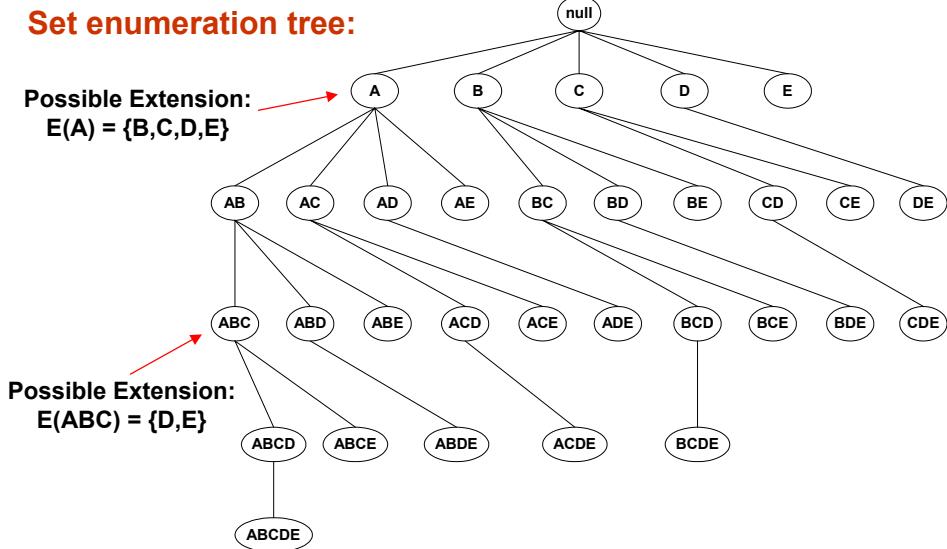
Frequent Itemsets found
(with sup > 1):

AD, BD, CD, ACD, BCD

Tree Projection

Set enumeration tree:

Possible Extension: $E(A) = \{B, C, D, E\}$



Possible Extension:
 $E(ABC) = \{D, E\}$

Tree Projection

- Items are listed in lexicographic order
- Each node P stores the following information:
 - Itemset for node P
 - List of possible lexicographic extensions of P: $E(P)$
 - Pointer to projected database of its ancestor node
 - Bitvector containing information about which transactions in the projected database contain the itemset

Projected Database

Original Database:

TID	Items
1	{A,B}
2	{B,C,D}
3	{A,C,D,E}
4	{A,D,E}
5	{A,B,C}
6	{A,B,C,D}
7	{B,C}
8	{A,B,C}
9	{A,B,D}
10	{B,C,E}

Projected Database
for node A:

TID	Items
1	{B}
2	{}
3	{C,D,E}
4	{D,E}
5	{B,C}
6	{B,C,D}
7	{}
8	{B,C}
9	{B,D}
10	{}

For each transaction T, projected transaction at node A is $T \cap E(A)$

ECLAT

- For each item, store a list of transaction ids (tids)

Horizontal
Data Layout

TID	Items
1	A,B,E
2	B,C,D
3	C,E
4	A,C,D
5	A,B,C,D
6	A,E
7	A,B
8	A,B,C
9	A,C,D
10	B

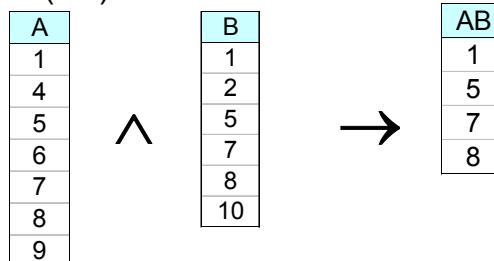
Vertical Data Layout

A	B	C	D	E
1	1	2	2	1
4	2	3	4	3
5	5	4	5	6
6	7	8	9	
7	8	9		
8	10			
9				

TID-list

ECLAT

- Determine support of any k-itemset by intersecting tid-lists of two of its (k-1) subsets.



- 3 traversal approaches:
 - top-down, bottom-up and hybrid
- Advantage: very fast support counting
- Disadvantage: intermediate tid-lists may become too large for memory

Rule Generation

- Given a frequent itemset L , find all non-empty subsets $f \subset L$ such that $f \rightarrow L - f$ satisfies the minimum confidence requirement

- If $\{A, B, C, D\}$ is a frequent itemset, candidate rules:

$$\begin{array}{llll} ABC \rightarrow D, & ABD \rightarrow C, & ACD \rightarrow B, & BCD \rightarrow A, \\ A \rightarrow BCD, & B \rightarrow ACD, & C \rightarrow ABD, & D \rightarrow ABC \\ AB \rightarrow CD, & AC \rightarrow BD, & AD \rightarrow BC, & BC \rightarrow AD, \\ BD \rightarrow AC, & CD \rightarrow AB, & & \end{array}$$

- If $|L| = k$, then there are $2^k - 2$ candidate association rules (ignoring $L \rightarrow \emptyset$ and $\emptyset \rightarrow L$)

Rule Generation

- How to efficiently generate rules from frequent itemsets?

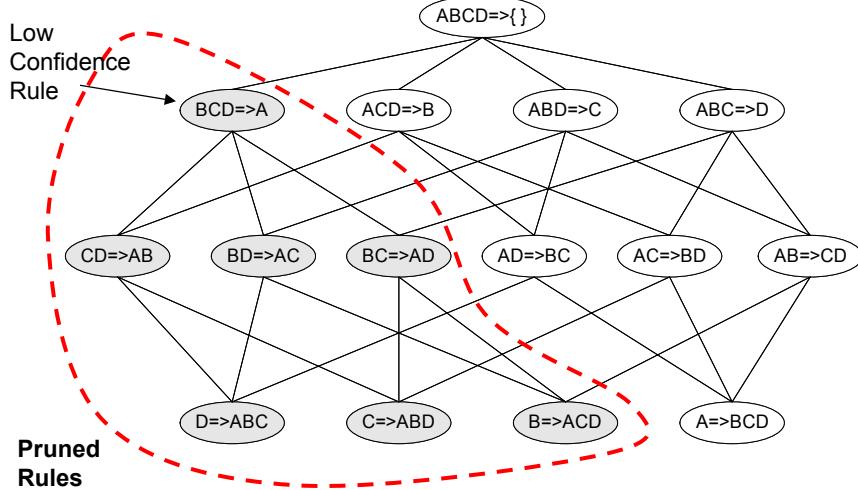
- In general, confidence does not have an anti-monotone property
 $c(ABC \rightarrow D)$ can be larger or smaller than $c(AB \rightarrow D)$
- But confidence of rules generated from the same itemset has an anti-monotone property
- e.g., $L = \{A, B, C, D\}$:

$$c(ABC \rightarrow D) \geq c(AB \rightarrow CD) \geq c(A \rightarrow BCD)$$

- Confidence is anti-monotone w.r.t. number of items on the RHS of the rule

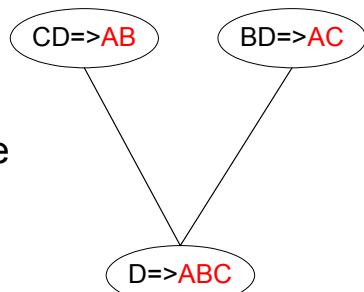
Rule Generation for Apriori Algorithm

Lattice of rules



Rule Generation for Apriori Algorithm

- Candidate rule is generated by merging two rules that share the same prefix in the rule consequent



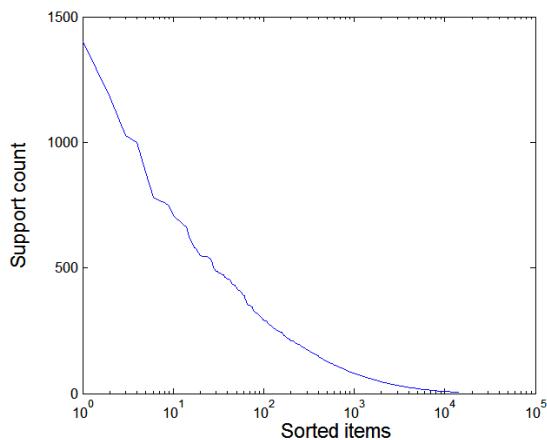
- $\text{join}(CD \Rightarrow AB, BD \Rightarrow AC)$ would produce the candidate rule $D \Rightarrow ABC$

- Prune rule $D \Rightarrow ABC$ if its subset $AD \Rightarrow BC$ does not have high confidence

Effect of Support Distribution

- Many real data sets have skewed support distribution

Support distribution of a retail data set



Effect of Support Distribution

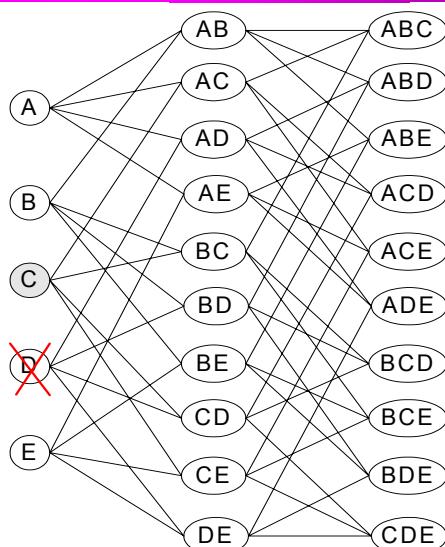
- How to set the appropriate *minsup* threshold?
 - If *minsup* is set too high, we could miss itemsets involving interesting rare items (e.g., expensive products)
 - If *minsup* is set too low, it is computationally expensive and the number of itemsets is very large
- Using a single minimum support threshold may not be effective

Multiple Minimum Support

- How to apply multiple minimum supports?
 - MS(i): minimum support for item i
 - e.g.: $MS(\text{Milk})=5\%$, $MS(\text{Coke}) = 3\%$,
 $MS(\text{Broccoli})=0.1\%$, $MS(\text{Salmon})=0.5\%$
 - $MS(\{\text{Milk, Broccoli}\}) = \min (MS(\text{Milk}), MS(\text{Broccoli})) = 0.1\%$
- Challenge: Support is no longer anti-monotone
 - Suppose: $\text{Support}(\text{Milk, Coke}) = 1.5\%$ and
 $\text{Support}(\text{Milk, Coke, Broccoli}) = 0.5\%$
 - $\{\text{Milk, Coke}\}$ is infrequent but $\{\text{Milk, Coke, Broccoli}\}$ is frequent

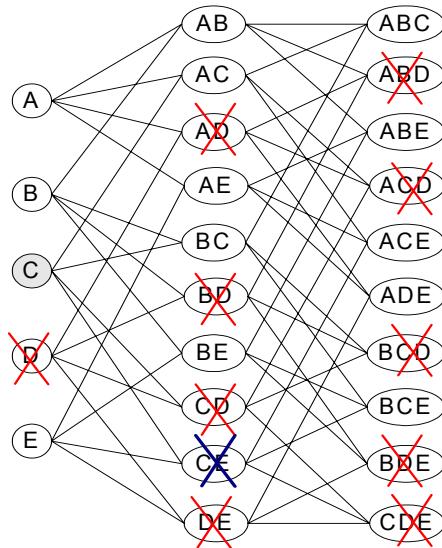
Multiple Minimum Support

Item	MS(I)	Sup(I)
A	0.10%	0.25%
B	0.20%	0.26%
C	0.30%	0.29%
D	0.50%	0.05%
E	3%	4.20%



Multiple Minimum Support

Item	MS(I)	Sup(I)
A	0.10%	0.25%
B	0.20%	0.26%
C	0.30%	0.29%
D	0.50%	0.05%
E	3%	4.20%



Multiple Minimum Support (Liu 1999)

- Order the items according to their minimum support (in ascending order)
 - e.g.: $MS(\text{Milk})=5\%$, $MS(\text{Coke}) = 3\%$,
 $MS(\text{Broccoli})=0.1\%$, $MS(\text{Salmon})=0.5\%$
 - Ordering: Broccoli, Salmon, Coke, Milk
- Need to modify Apriori such that:
 - L_1 : set of frequent items
 - F_1 : set of items whose support is $\geq MS(1)$ where $MS(1)$ is $\min_i(MS(i))$
 - C_2 : candidate itemsets of size 2 is generated from F_1 instead of L_1

Multiple Minimum Support (Liu 1999)

- Modifications to Apriori:

- In traditional Apriori,
 - ◆ A candidate $(k+1)$ -itemset is generated by merging two frequent itemsets of size k
 - ◆ The candidate is pruned if it contains any infrequent subsets of size k
 - Pruning step has to be modified:
 - ◆ Prune only if subset contains the first item
 - ◆ e.g.: Candidate={Broccoli, Coke, Milk} (ordered according to minimum support)
 - ◆ {Broccoli, Coke} and {Broccoli, Milk} are frequent but {Coke, Milk} is infrequent
 - Candidate is not pruned because {Coke,Milk} does not contain the first item, i.e., Broccoli.

Pattern Evaluation

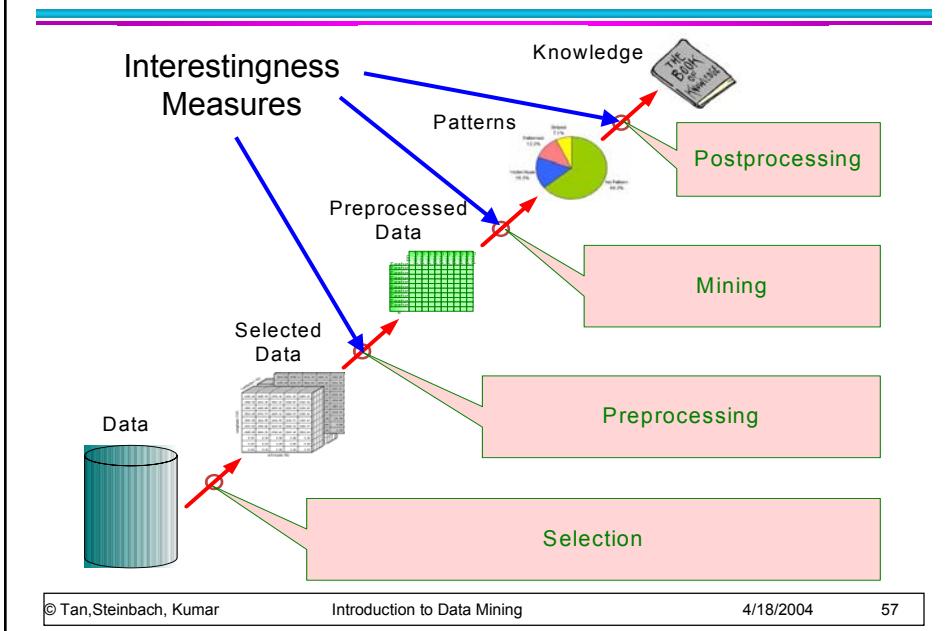
- Association rule algorithms tend to produce too many rules

- many of them are uninteresting or redundant
 - Redundant if $\{A,B,C\} \rightarrow \{D\}$ and $\{A,B\} \rightarrow \{D\}$ have same support & confidence

- Interestingness measures can be used to prune/rank the derived patterns

- In the original formulation of association rules, support & confidence are the only measures used

Application of Interestingness Measure



Computing Interestingness Measure

- Given a rule $X \rightarrow Y$, information needed to compute rule interestingness can be obtained from a contingency table

Contingency table for $X \rightarrow Y$

	Y	\bar{Y}	
X	f_{11}	f_{10}	f_{1+}
\bar{X}	f_{01}	f_{00}	f_{0+}
	f_{+1}	f_{+0}	$ T $

f_{11} : support of X and Y
 f_{10} : support of X and \bar{Y}
 f_{01} : support of \bar{X} and Y
 f_{00} : support of \bar{X} and \bar{Y}

Used to define various measures

- support, confidence, lift, Gini, J-measure, etc.

Drawback of Confidence

	Coffee	$\overline{\text{Coffee}}$	
Tea	15	5	20
$\overline{\text{Tea}}$	75	5	80
	90	10	100

Association Rule: Tea \rightarrow Coffee

$$\text{Confidence} = P(\text{Coffee}|\text{Tea}) = 0.75$$

$$\text{but } P(\text{Coffee}) = 0.9$$

\Rightarrow Although confidence is high, rule is misleading

$$\Rightarrow P(\text{Coffee}|\overline{\text{Tea}}) = 0.9375$$

Statistical Independence

- Population of 1000 students
 - 600 students know how to swim (S)
 - 700 students know how to bike (B)
 - 420 students know how to swim and bike (S,B)
 - $P(S \wedge B) = 420/1000 = 0.42$
 - $P(S) \times P(B) = 0.6 \times 0.7 = 0.42$
 - $P(S \wedge B) = P(S) \times P(B) \Rightarrow$ Statistical independence
 - $P(S \wedge B) > P(S) \times P(B) \Rightarrow$ Positively correlated
 - $P(S \wedge B) < P(S) \times P(B) \Rightarrow$ Negatively correlated

Statistical-based Measures

- Measures that take into account statistical dependence

$$Lift = \frac{P(Y|X)}{P(Y)}$$

$$Interest = \frac{P(X,Y)}{P(X)P(Y)}$$

$$PS = P(X,Y) - P(X)P(Y)$$

$$\phi-coefficient = \frac{P(X,Y) - P(X)P(Y)}{\sqrt{P(X)[1-P(X)]P(Y)[1-P(Y)]}}$$

Example: Lift/Interest

	Coffee	$\bar{\text{Coffee}}$	
Tea	15	5	20
$\bar{\text{Tea}}$	75	5	80
	90	10	100

Association Rule: Tea \rightarrow Coffee

Confidence= $P(\text{Coffee}|\text{Tea}) = 0.75$

but $P(\text{Coffee}) = 0.9$

$\Rightarrow Lift = 0.75/0.9 = 0.8333 (< 1, \text{ therefore is negatively associated})$

Drawback of Lift & Interest

	Y	\bar{Y}	
X	10	0	10
\bar{X}	0	90	90
	10	90	100

$$Lift = \frac{0.1}{(0.1)(0.1)} = 10$$

	Y	\bar{Y}	
X	90	0	90
\bar{X}	0	10	10
	90	10	100

$$Lift = \frac{0.9}{(0.9)(0.9)} = 1.11$$

Statistical independence:

If $P(X,Y) = P(X)P(Y)$ \Rightarrow Lift = 1

#	Measure	Formula
1	ϕ -coefficient	$\frac{P(A,B) - P(A)P(B)}{\sqrt{P(A)P(B)(1-P(A))(1-P(B))}}$
2	Goodman-Kruskal's (λ)	$\sum_j \frac{\max_i P(A_j, B_k) + \sum_k \max_j P(A_j, B_k) - \max_j P(A_j) - \max_k P(B_k)}{2 - \max_j P(A_j) - \max_k P(B_k)}$
3	Odds ratio (α)	$\frac{P(A,B)P(\bar{A},\bar{B})}{P(\bar{A},B)P(A,\bar{B})}$
4	Yule's Q	$\frac{P(A,B)P(\bar{A},\bar{B}) + P(A,\bar{B})P(\bar{A},B)}{P(A,B)P(\bar{A},\bar{B}) + P(A,\bar{B})P(\bar{A},B)} = \frac{\alpha - 1}{\alpha + 1}$
5	Yule's Y	$\sqrt{\frac{P(A,B)P(\bar{A},\bar{B})}{P(A,B)P(\bar{A},\bar{B})}} - \sqrt{\frac{P(A,B)P(\bar{A},B)}{P(A,B)P(\bar{A},B)}} = \frac{\sqrt{\alpha} - 1}{\sqrt{\alpha} + 1}$
6	Kappa (κ)	$\frac{\sqrt{P(A,B)P(\bar{A},\bar{B}) + P(A,\bar{B})P(\bar{A},B)}}{1 - P(A)P(\bar{B}) - P(\bar{A})P(B)}$
7	Mutual Information (M)	$\sum_i \sum_j P(A_i, B_j) \log \frac{P(A_i, B_j)}{P(A_i)P(B_j)}$
8	J-Measure (J)	$\max \left(P(A, B) \log \left(\frac{P(B A)}{P(B)} \right) + P(\bar{A} B) \log \left(\frac{P(\bar{B} A)}{P(\bar{B})} \right), P(A, \bar{B}) \log \left(\frac{P(\bar{A} B)}{P(\bar{A})} \right) + P(\bar{A} B) \log \left(\frac{P(\bar{A} B)}{P(\bar{A})} \right) \right)$
9	Gini index (G)	$\max \left(P(A)[P(B A)^2 + P(\bar{B} A)^2] + P(\bar{A})[P(B \bar{A})^2 + P(\bar{B} \bar{A})^2], P(B)[P(A B)^2 + P(\bar{A} B)^2] + P(\bar{B})[P(A \bar{B})^2 + P(\bar{A} \bar{B})^2] - P(A)^2 - P(\bar{B})^2, P(B)[P(A B)^2 + P(\bar{A} B)^2] + P(\bar{B})[P(A \bar{B})^2 + P(\bar{A} \bar{B})^2] - P(A)^2 - P(\bar{A})^2 \right)$
10	Support (s)	$P(A, B)$
11	Confidence (c)	$\max(P(B A), P(A B))$
12	Laplace (L)	$\max \left(\frac{NP(A,B)+1}{NP(A)+2}, \frac{NP(A,B)+1}{NP(B)+2} \right)$
13	Conviction (V)	$\max \left(\frac{P(A)P(\bar{B})}{P(AB)}, \frac{P(B)P(\bar{A})}{P(B\bar{A})} \right)$
14	Interest (I)	$\frac{P(A,B)}{P(A)P(\bar{B})}$
15	cosine (IS)	$\frac{P(A,B)}{\sqrt{P(A)P(B)}}$
16	Piatetsky-Shapiro's (PS)	$P(A, B) - P(A)P(B)$
17	Certainty factor (F)	$\max \left(\frac{P(B A) - P(B)}{1 - P(B)}, \frac{P(A B) - P(A)}{1 - P(A)} \right)$
18	Added Value (AV)	$\max(P(B A) - P(B), P(A B) - P(A))$
19	Collective strength (S)	$\frac{P(A,B) + P(\bar{A},\bar{B})}{P(A)P(\bar{B}) + P(\bar{A})P(B)} \times \frac{1 - P(A)P(B) - P(\bar{A})P(\bar{B})}{1 - P(A,B) - P(\bar{A}\bar{B})}$
20	Jaccard (ζ)	$\frac{P(A) + P(\bar{B}) - P(A,\bar{B})}{P(A) + P(\bar{B})}$
21	Klosgen (K)	$\sqrt{P(A,B) \max(P(B A) - P(B), P(A B) - P(A))}$

There are lots of measures proposed in the literature

Some measures are good for certain applications, but not for others

What criteria should we use to determine whether a measure is good or bad?

What about Apriori-style support based pruning? How does it affect these measures?

Properties of A Good Measure

● Piatetsky-Shapiro:

3 properties a good measure M must satisfy:

- $M(A,B) = 0$ if A and B are statistically independent
- $M(A,B)$ increase monotonically with $P(A,B)$ when $P(A)$ and $P(B)$ remain unchanged
- $M(A,B)$ decreases monotonically with $P(A)$ [or $P(B)$] when $P(A,B)$ and $P(B)$ [or $P(A)$] remain unchanged

Comparing Different Measures

10 examples of contingency tables:

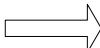
Example	f_{11}	f_{10}	f_{01}	f_{00}
E1	8123	83	424	1370
E2	8330	2	622	1046
E3	9481	94	127	298
E4	3954	3080	5	2961
E5	2886	1363	1320	4431
E6	1500	2000	500	6000
E7	4000	2000	1000	3000
E8	4000	2000	2000	2000
E9	1720	7121	5	1154
E10	61	2483	4	7452

Rankings of contingency tables using various measures:

#	ϕ	λ	α	Q	Y	κ	M	J	G	s	c	L	V	I	IS	PS	F	AV	S	ζ	K
E1	1	1	3	3	3	1	2	2	1	3	5	5	4	6	2	2	4	6	1	2	5
E2	2	2	1	1	1	2	1	3	2	2	1	1	1	8	3	5	1	8	2	3	6
E3	3	3	4	4	4	3	3	8	7	1	4	4	6	10	1	8	6	10	3	1	10
E4	4	7	2	2	2	5	4	1	3	6	2	2	2	4	4	1	2	3	4	5	1
E5	5	4	8	8	8	4	7	5	4	7	9	9	9	3	6	3	9	4	5	6	3
E6	6	6	7	7	7	7	6	4	6	9	8	8	7	2	8	6	7	2	7	8	2
E7	7	5	9	9	9	6	8	6	5	4	7	7	8	5	5	4	8	5	6	4	4
E8	8	9	10	10	8	10	10	8	4	10	10	10	9	7	7	10	9	8	7	9	9
E9	9	9	5	5	5	9	9	7	9	8	3	3	3	7	9	9	3	7	9	9	8
E10	10	8	6	6	6	10	5	9	10	10	6	6	5	1	10	10	5	1	10	10	7

Property under Variable Permutation

	B	\bar{B}
A	p	q
\bar{A}	r	s



	A	\bar{A}
B	p	r
\bar{B}	q	s

Does $M(A,B) = M(B,A)$?

Symmetric measures:

- ◆ support, lift, collective strength, cosine, Jaccard, etc

Asymmetric measures:

- ◆ confidence, conviction, Laplace, J-measure, etc

Property under Row/Column Scaling

Grade-Gender Example (Mosteller, 1968):

	Male	Female	
High	2	3	5
Low	1	4	5
	3	7	10

	Male	Female	
High	4	30	34
Low	2	40	42
	6	70	76

↓
2x ↓
10x

Mosteller:

Underlying association should be independent of the relative number of male and female students in the samples

Property under Inversion Operation

(a)

C	D
0	1
1	1
1	1
1	1
1	0
1	1
1	1
1	1
1	1
0	1

(b)

E	F
0	0
1	0
1	0
1	0
1	1
1	0
1	0
1	0
1	0
0	0

(c)

Example: ϕ -Coefficient

- ϕ -coefficient is analogous to correlation coefficient for continuous variables

	Y	\bar{Y}	
X	60	10	70
\bar{X}	10	20	30
	70	30	100

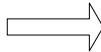
	Y	\bar{Y}	
X	20	10	30
\bar{X}	10	60	70
	30	70	100

$$\phi = \frac{0.6 - 0.7 \times 0.7}{\sqrt{0.7 \times 0.3 \times 0.7 \times 0.3}} = 0.5238$$

ϕ Coefficient is the same for both tables

Property under Null Addition

	B	\bar{B}
A	p	q
\bar{A}	r	s



	B	\bar{B}
A	p	q
\bar{A}	r	$s + k$

Invariant measures:

- ◆ support, cosine, Jaccard, etc

Non-invariant measures:

- ◆ correlation, Gini, mutual information, odds ratio, etc

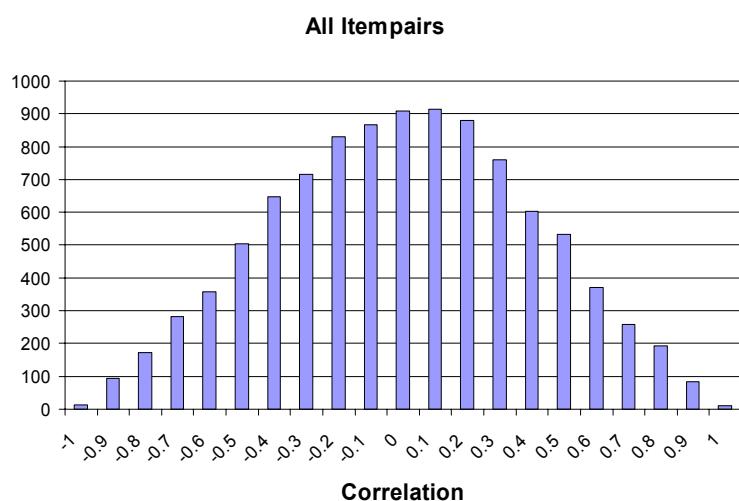
Different Measures have Different Properties

Symbol	Measure	Range	P1	P2	P3	O1	O2	O3	O3'	O4
ϕ	Correlation	-1 ... 0 ... 1	Yes	Yes	Yes	Yes	No	Yes	Yes	No
λ	Lambda	0 ... 1	Yes	No	No	Yes	No	No*	Yes	No
α	Odds ratio	0 ... 1 ... ∞	Yes*	Yes	Yes	Yes	Yes	Yes*	Yes	No
Q	Yule's Q	-1 ... 0 ... 1	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
Y	Yule's Y	-1 ... 0 ... 1	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
κ	Cohen's	-1 ... 0 ... 1	Yes	Yes	Yes	Yes	No	No	Yes	No
M	Mutual Information	0 ... 1	Yes	Yes	Yes	Yes	No	No*	Yes	No
J	J-Measure	0 ... 1	Yes	No	No	No	No	No	No	No
G	Gini Index	0 ... 1	Yes	No	No	No	No	No*	Yes	No
S	Support	0 ... 1	No	Yes	No	Yes	No	No	No	No
C	Confidence	0 ... 1	No	Yes	No	Yes	No	No	No	Yes
L	Laplace	0 ... 1	No	Yes	No	Yes	No	No	No	No
V	Conviction	0.5 ... 1 ... ∞	No	Yes	No	Yes**	No	No	Yes	No
I	Interest	0 ... 1 ... ∞	Yes*	Yes	Yes	Yes	No	No	No	No
IS	IS (cosine)	0 .. 1	No	Yes	Yes	Yes	No	No	No	Yes
PS	Piatetsky-Shapiro's	-0.25 ... 0 ... 0.25	Yes	Yes	Yes	Yes	No	Yes	Yes	No
F	Certainty factor	-1 ... 0 ... 1	Yes	Yes	Yes	No	No	No	Yes	No
AV	Added value	0.5 ... 1 ... 1	Yes	Yes	Yes	No	No	No	No	No
S	Collective strength	0 ... 1 ... ∞	No	Yes	Yes	Yes	No	Yes*	Yes	No
ζ	Jaccard	0 .. 1	No	Yes	Yes	Yes	No	No	No	Yes
K	Klosgen's	$\left(\frac{2}{\sqrt{3}} - 1\right) \left(2 - \sqrt{3} - \frac{1}{\sqrt{3}}\right) \dots 0 \dots \frac{2}{3\sqrt{3}}$	Yes	Yes	Yes	No	No	No	No	No

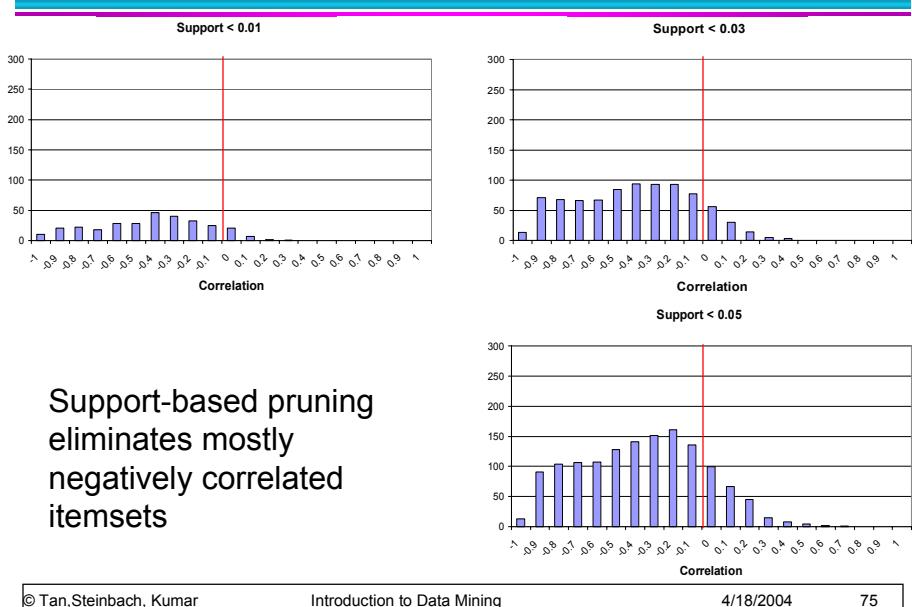
Support-based Pruning

- Most of the association rule mining algorithms use support measure to prune rules and itemsets
- Study effect of support pruning on correlation of itemsets
 - Generate 10000 random contingency tables
 - Compute support and pairwise correlation for each table
 - Apply support-based pruning and examine the tables that are removed

Effect of Support-based Pruning



Effect of Support-based Pruning

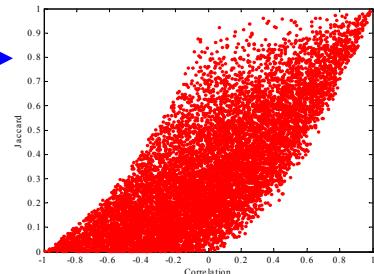
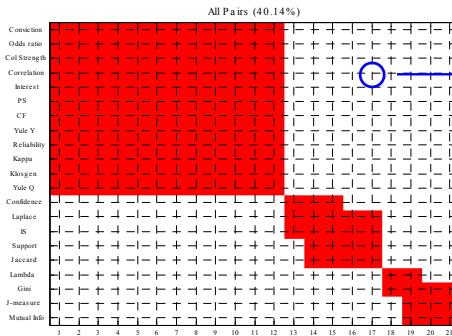


Effect of Support-based Pruning

- Investigate how support-based pruning affects other measures
- Steps:
 - Generate 10000 contingency tables
 - Rank each table according to the different measures
 - Compute the pair-wise correlation between the measures

Effect of Support-based Pruning

- Without Support Pruning (All Pairs)

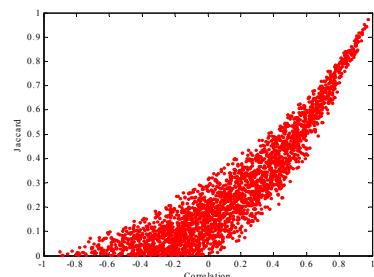
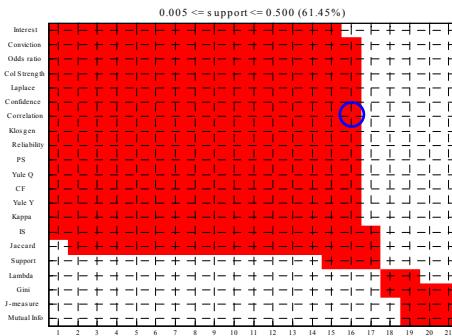


Scatter Plot between Correlation & Jaccard Measure

- Red cells indicate correlation between the pair of measures > 0.85
- 40.14% pairs have correlation > 0.85

Effect of Support-based Pruning

- 0.5% ≤ support ≤ 50%

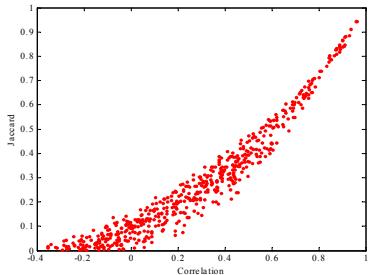
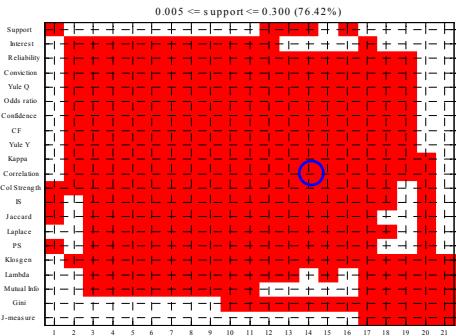


Scatter Plot between Correlation & Jaccard Measure:

- 61.45% pairs have correlation > 0.85

Effect of Support-based Pruning

- ◆ $0.5\% \leq \text{support} \leq 30\%$



Scatter Plot between Correlation & Jaccard Measure

- ◆ 76.42% pairs have correlation > 0.85

Subjective Interestingness Measure

● Objective measure:

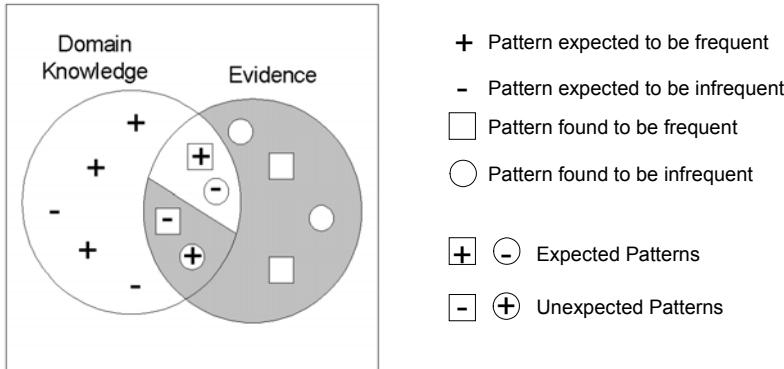
- Rank patterns based on statistics computed from data
- e.g., 21 measures of association (support, confidence, Laplace, Gini, mutual information, Jaccard, etc).

● Subjective measure:

- Rank patterns according to user's interpretation
 - ◆ A pattern is subjectively interesting if it contradicts the expectation of a user (Silberschatz & Tuzhilin)
 - ◆ A pattern is subjectively interesting if it is actionable (Silberschatz & Tuzhilin)

Interestingness via Unexpectedness

- Need to model expectation of users (domain knowledge)



- Need to combine expectation of users with evidence from data (i.e., extracted patterns)

Interestingness via Unexpectedness

- Web Data (Cooley et al 2001)

- Domain knowledge in the form of site structure
- Given an itemset $F = \{X_1, X_2, \dots, X_k\}$ (X_i : Web pages)
 - L: number of links connecting the pages
 - Ifactor = $L / (k \times k-1)$
 - Cfactor = 1 (if graph is connected), 0 (disconnected graph)
- Structure evidence = cfactor \times Ifactor

$$\text{Usage evidence} = \frac{P(X_1 \cap X_2 \cap \dots \cap X_k)}{P(X_1 \cup X_2 \cup \dots \cup X_k)}$$

- Use Dempster-Shafer theory to combine domain knowledge and evidence from data