output as a weighted sum of inputs passed through a threshold function, $y = \mathbb{I}(\sum_i w_i x_i > \theta)$, for some threshold θ. This is similar to a sigmoidal activation function. Frank Rosenblatt invented the perceptron learning algorithm in 1957, which is a way to estimate the parameters of a McCulloch-Pitts neuron (see Section 8.5.4 for details). A very similar model called the **adaline** (for adaptive linear element) was invented in 1960 by Widrow and Hoff.

In 1969, Minsky and Papert (Minsky and Papert 1969) published a famous book called "Perceptrons" in which they showed that such linear models, with no hidden layers, were very limited in their power, since they cannot classify data that is not linearly separable. This considerably reduced interest in the field.

In 1986, Rumelhart, Hinton and Williams (Rumelhart et al. 1986) discovered the backpropagation algorithm (see Section 16.5.4), which allows one to fit models with hidden layers. (The backpropagation algorithm was originally discovered in (Bryson and Ho 1969), and independently in (Werbos 1974); however, it was (Rumelhart et al. 1986) that brought the algorithm to people's attention.) This spawned a decade of intense interest in these models.

In 1987, Sejnowski and Rosenberg (Sejnowski and Rosenberg 1987) created the famous **NETtalk** system, that learned a mapping from English words to phonetic symbols which could be fed into a speech synthesizer. An audio demo of the system as it learns over time can be found at `http://www.cnl.salk.edu/ParallelNetsPronounce/nettalk.mp3`. The systems starts by "babbling" and then gradually learns to pronounce English words. NETtalk learned a **distributed representation** (via its hidden layer) of various sounds, and its success spawned a big debate in psychology between **connectionism**, based on neural networks, and **computationalism**, based on syntactic rules. This debate lives on to some extent in the machine learning community, where there are still arguments about whether learning is best performed using low-level, "neural-like" representations, or using more structured models.

In 1989, Yann Le Cun and others (LeCun et al. 1989) created the famous LeNet system described in Section 16.5.1.

In 1992, the support vector machine (see Section 14.5) was invented (Boser et al. 1992). SVMs provide similar prediction accuracy to neural networks while being considerably easier to train (since they use a convex objective function). This spawned a decade of interest in kernel methods in general.[7] Note, however, that SVMs do not use adaptive basis functions, so they require a fair amount of human expertise to design the right kernel function.

In 2002, Geoff Hinton invented the contrastive divergence training procedure (Hinton 2002), which provided a way, for the first time, to learn deep networks, by training one layer at a time in an unsupervised fashion (see Section 27.7.2.4 for details). This in turn has spawned renewed interest in neural networks over the last few years (see Chapter 28).

16.5.4 The backpropagation algorithm

Unlike a GLM, the NLL of an MLP is a non-convex function of its parameters. Nevertheless, we can find a locally optimal ML or MAP estimate using standard gradient-based optimization methods. Since MLPs have lots of parameters, they are often trained on very large data sets.

7. It became part of the folklore during the 1990s that to get published in the top machine learning conference known as NIPS, which stands for "neural information processing systems", it was important to ensure your paper did not contain the word "neural network"!

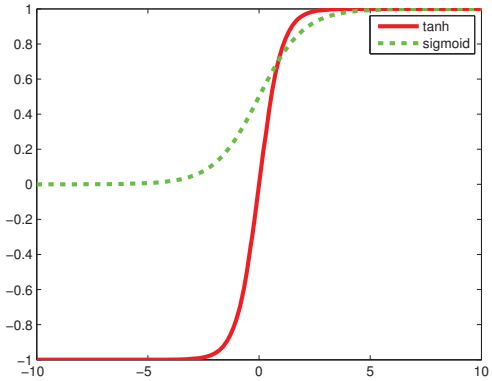


Figure 16.16 Two possible activation functions. $\tanh$ maps $\mathbb{R}$ to $[-1, +1]$ and is the preferred nonlinearity for the hidden nodes. sigm maps $\mathbb{R}$ to $[0, 1]$ and is the preferred nonlinearity for binary nodes at the output layer. Figure generated by `tanhPlot`.

Consequently it is common to use first-order online methods, such as stochastic gradient descent (Section 8.5.2), whereas GLMs are usually fit with IRLS, which is a second-order offline method.

We now discuss how to compute the gradient vector of the NLL by applying the chain rule of calculus. The resulting algorithm is known as **backpropagation**, for reasons that will become apparent.

For notational simplicity, we shall assume a model with just one hidden layer. It is helpful to distinguish the pre- and post-synaptic values of a neuron, that is, before and after we apply the nonlinearity. Let $\mathbf{x}_n$ be the n'th input, $\mathbf{a}_n = \mathbf{V}\mathbf{x}_n$ be the pre-synaptic hidden layer, and $\mathbf{z}_n = g(\mathbf{a}_n)$ be the post-synaptic hidden layer, where g is some **transfer function**. We typically use $g(a) = \text{sigm}(a)$, but we may also use $g(a) = \tanh(a)$: see Figure 16.16 for a comparison. (When the input to sigm or $\tanh$ is a vector, we assume it is applied component-wise.)

We now convert this hidden layer to the output layer as follows. Let $\mathbf{b}_n = \mathbf{W}\mathbf{z}_n$ be the pre-synaptic output layer, and $\hat{\mathbf{y}}_n = h(\mathbf{b}_n)$ be the post-synaptic output layer, where h is another nonlinearity, corresponding to the canonical link for the GLM. (We reserve the notation $\mathbf{y}_n$, without the hat, for the output corresponding to the n'th training case.) For a regression model, we use $h(\mathbf{b}) = \mathbf{b}$; for binary classifcation, we use $h(\mathbf{b}) = [\text{sigm}(b_1), \ldots, \text{sigm}(b_c)]$; for multi-class classification, we use $h(\mathbf{b}) = \mathcal{S}(\mathbf{b})$.

We can write the overall model as follows:

$$\mathbf{x}_n \xrightarrow{\mathbf{V}} \mathbf{a}_n \xrightarrow{g} \mathbf{z}_n \xrightarrow{\mathbf{W}} \mathbf{b}_n \xrightarrow{h} \hat{\mathbf{y}}_n \tag{16.65}$$

The parameters of the model are $\boldsymbol{\theta} = (\mathbf{V}, \mathbf{W})$, the first and second layer weight matrices. Offset or bias terms can be accomodated by clamping an element of $\mathbf{x}_n$ and $\mathbf{z}_n$ to 1.[8]

8. In the regression setting, we can easily estimate the variance of the output noise using the empirical variance of the residual errors, $\hat{\sigma}^2 = \frac{1}{N}||\hat{\mathbf{y}}(\hat{\boldsymbol{\theta}}) - \mathbf{y}||^2$, after training is complete. There will be one value of σ^2 for each output node, if we are performing multi-target regression, as we usually assume.

In the regression case, with K outputs, the NLL is given by the squared error:

$$J(\boldsymbol{\theta}) = -\sum_n \sum_k (\hat{y}_{nk}(\boldsymbol{\theta}) - y_{nk})^2 \tag{16.66}$$

In the classification case, with K classes, the NLL is given by the cross entropy

$$J(\boldsymbol{\theta}) = -\sum_n \sum_k y_{nk} \log \hat{y}_{nk}(\boldsymbol{\theta}) \tag{16.67}$$

Our task is to compute $\nabla_{\boldsymbol{\theta}} J$. We will derive this for each n separately; the overall gradient is obtained by summing over n, although often we just use a mini-batch (see Section 8.5.2).

Let us start by considering the output layer weights. We have

$$\nabla_{\mathbf{w}_k} J_n = \frac{\partial J_n}{\partial b_{nk}} \nabla_{\mathbf{w}_k} b_{nk} = \frac{\partial J_n}{\partial b_{nk}} \mathbf{z}_n \tag{16.68}$$

since $b_{nk} = \mathbf{w}_k^T \mathbf{z}_n$. Assuming h is the canonical link function for the output GLM, then Equation 9.91 tells us that

$$\frac{\partial J_n}{\partial b_{nk}} \triangleq \delta_{nk}^w = (\hat{y}_{nk} - y_{nk}) \tag{16.69}$$

which is the error signal. So the overall gradient is

$$\nabla_{\mathbf{w}_k} J_n = \delta_{nk}^w \mathbf{z}_n \tag{16.70}$$

which is the pre-synaptic input to the output layer, namely $\mathbf{z}_n$, times the error signal, namely δ_{nk}^w.

For the input layer weights, we have

$$\nabla_{\mathbf{v}_j} J_n = \frac{\partial J_n}{\partial a_{nj}} \nabla_{\mathbf{v}_j} a_{nj} \triangleq \delta_{nj}^v \mathbf{x}_n \tag{16.71}$$

where we exploited the fact that $a_{nj} = \mathbf{v}_j^T \mathbf{x}_n$. All that remains is to compute the first level error signal δ_{nj}^v. We have

$$\delta_{nj}^v = \frac{\partial J_n}{\partial a_{nj}} = \sum_{k=1}^K \frac{\partial J_n}{\partial b_{nk}} \frac{\partial b_{nk}}{\partial a_{nj}} = \sum_{k=1}^K \delta_{nk}^w \frac{\partial b_{nk}}{\partial a_{nj}} \tag{16.72}$$

Now

$$b_{nk} = \sum_j w_{kj} g(a_{nj}) \tag{16.73}$$

so

$$\frac{\partial b_{nk}}{\partial a_{nj}} = w_{kj} g'(a_{nj}) \tag{16.74}$$

where $g'(a) = \frac{d}{da} g(a)$. For tanh units, $g'(a) = \frac{d}{da} \tanh(a) = 1 - \tanh^2(a) = \text{sech}^2(a)$, and for sigmoid units, $g'(a) = \frac{d}{da} \sigma(a) = \sigma(a)(1 - \sigma(a))$. Hence

$$\delta_{nj}^v = \sum_{k=1}^K \delta_{nk}^w w_{kj} g'(a_{nj}) \tag{16.75}$$

Thus the layer 1 errors can be computed by passing the layer 2 errors back through the $\mathbf{W}$ matrix; hence the term "backpropagation". The key property is that we can compute the gradients locally: each node only needs to know about its immediate neighbors. This is supposed to make the algorithm "neurally plausible", although this interpretation is somewhat controversial.

Putting it all together, we can compute all the gradients as follows: we first perform a forwards pass to compute $\mathbf{a}_n$, $\mathbf{z}_n$, $\mathbf{b}_n$ and $\hat{\mathbf{y}}_n$. We then compute the error for the output layer, $\boldsymbol{\delta}_n^{(2)} = \hat{\mathbf{y}}_n - \mathbf{y}_n$, which we pass backwards through $\mathbf{W}$ using Equation 16.75 to compute the error for the hidden layer, $\boldsymbol{\delta}_n^{(1)}$. We then compute the overall gradient as follows:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \sum_n [\boldsymbol{\delta}_n^v \mathbf{x}_n,\ \boldsymbol{\delta}_n^w \mathbf{z}_n] \tag{16.76}$$

16.5.5 Identifiability

It is easy to see that the parameters of a neural network are not identifiable. For example, we can change the sign of the weights going into one of the hidden units, so long as we change the sign of all the weights going out of it; these effects cancel, since tanh is an odd function, so $\tanh(-a) = -\tanh(a)$. There will be H such sign flip symmetries, leading to 2^H equivalent settings of the parameters. Similarly, we can change the identity of the hidden units without affecting the likelihood. There are $H!$ such permutations. The total number of equivalent parameter settings (with the same likelihood) is therefore $H!2^H$.

In addition, there may be local minima due to the non-convexity of the NLL. This can be a more serious problem, although with enough data, these local optima are often quite "shallow", and simple stochastic optimization methods can avoid them. In addition, it is common to perform multiple restarts, and to pick the best solution, or to average over the resulting predictions. (It does not make sense to average the parameters themselves, since they are not identifiable.)

16.5.6 Regularization

As usual, the MLE can overfit, especially if the number of nodes is large. A simple way to prevent this is called **early stopping**, which means stopping the training procedure when the error on the validation set first starts to increase. This method works because we usually initialize from small random weights, so the model is initially simple (since the $\tanh$ and sigm functions are nearly linear near the origin). As training progresses, the weights become larger, and the model becomes nonlinear. Eventually it will overfit.

Another way to prevent overfitting, that is more in keeping with the approaches used elsewhere in this book, is to impose a prior on the parameters, and then use MAP estimation. It is standard to use a $\mathcal{N}(0, \alpha^{-1}\mathbf{I})$ prior (equivalent to ℓ_2 regularization), where α is the precision (strength) of the prior. In the neural networks literature, this is called **weight decay**, since it encourages small weights, and hence simpler models. The penalized NLL objective becomes

$$J(\boldsymbol{\theta}) = -\sum_{n=1}^{N} \log p(y_n|\mathbf{x}_n, \boldsymbol{\theta}) + \frac{\alpha}{2}[\sum_{ij} v_{ij}^2 + \sum_{jk} w_{jk}^2] \tag{16.77}$$

(Note that we don't penalize the bias terms.) The gradient of the modified objective becomes

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = [\sum_n \boldsymbol{\delta}_n^v \mathbf{x}_n + \alpha \mathbf{v},\ \sum_n \boldsymbol{\delta}_n^w \mathbf{z}_n + \alpha \mathbf{w}] \tag{16.78}$$

as in Section 8.3.6.

If the regularization is sufficiently strong, it does not matter if we have too many hidden units (apart from wasted computation). Hence it is advisable to set H to be as large as you can afford (say 10–100), and then to choose an appropriate regularizer. We can set the α parameter by cross validation or empirical Bayes (see Section 16.5.7.5).

As with ridge regression, it is good practice to standardize the inputs to zero mean and unit variance, so that the spherical Gaussian prior makes sense.

16.5.6.1 Consistent Gaussian priors *

One can show (MacKay 1992) that using the same regularization parameter for both the first and second layer weights results in the lack of a certain desirable invariance property. In particular, suppose we linearly scale and shift the inputs and/or outputs to a neural network regression model. Then we would like the model to learn to predict the same function, by suitably scaling its internal weights and bias terms. However, the amount of scaling needed by the first and second layer weights to compensate for a change in the inputs and/or outputs is not the same. Therefore we need to use a different regularization strength for the first and second layer. Fortunately, this is easy to do — we just use the following prior:

$$p(\boldsymbol{\theta}) = \mathcal{N}(\mathbf{W}|\mathbf{0}, \frac{1}{\alpha_w}\mathbf{I})\mathcal{N}(\mathbf{V}|\mathbf{0}, \frac{1}{\alpha_v}\mathbf{I})\mathcal{N}(\mathbf{b}|\mathbf{0}, \frac{1}{\alpha_b}\mathbf{I})\mathcal{N}(\mathbf{c}|\mathbf{0}, \frac{1}{\alpha_c}\mathbf{I}) \tag{16.79}$$

where $\mathbf{b}$ and $\mathbf{c}$ are the bias terms.[9]

To get a feeling for the effect of these hyper-parameters, we can sample MLP parameters from this prior and plot the resulting random functions. Figure 16.17 shows some examples. Decreasing α_v allows the first layer weights to get bigger, making the sigmoid-like shape of the functions steeper. Decreasing α_b allows the first layer biases to get bigger, which allows the center of the sigmoid to shift left and right more. Decreasing α_w allows the second layer weights to get bigger, making the functions more "wiggly" (greater sensitivity to change in the input, and hence larger dynamic range). And decreasing α_c allows the second layer biases to get bigger, allowing the mean level of the function to move up and down more. (In Chapter 15, we will see an easier way to define priors over functions.)

16.5.6.2 Weight pruning

Since there are many weights in a neural network, it is often helpful to encourage sparsity. Various ad-hoc methods for doing this, with names such as "optimal brain damage", were devised in the 1990s; see e.g., (Bishop 1995) for details.

9. Since we are regularizing the output bias terms, it is helpful, in the case of regression, to normalize the target responses in the training set to zero mean, to be consistent with the fact that the prior on the output bias has zero mean.

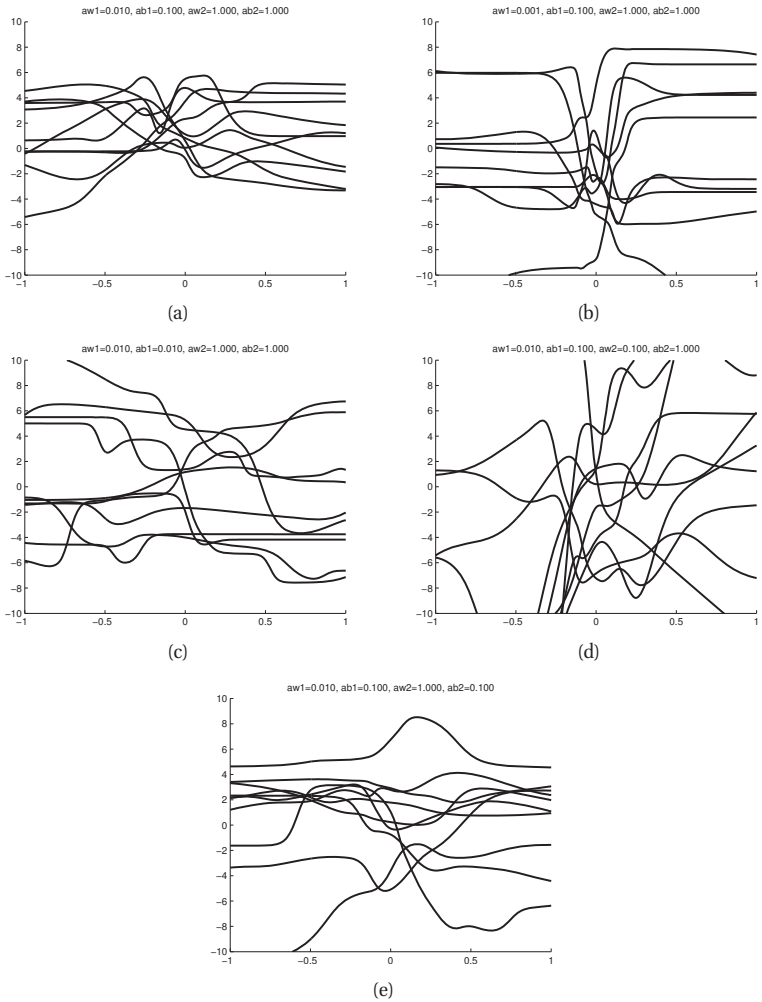
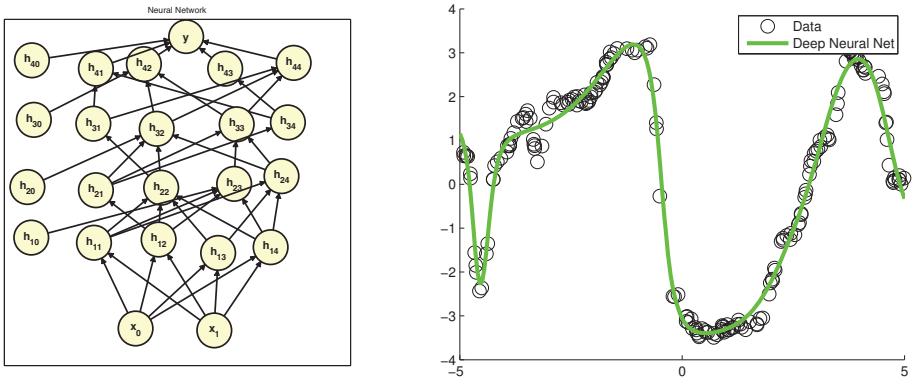


Figure 16.17 The effects of changing the hyper-parameters on an MLP. (a) Default parameter values $\alpha_v = 0.01$, $\alpha_b = 0.1$, $\alpha_w = 1$, $\alpha_c = 1$. (b) Decreasing α_v by factor of 10. (c) Decreasing α_b by factor of 10. (d) Decreasing α_w by factor of 10. (e) Decreasing α_c by factor of 10. Figure generated by `mlpPriorsDemo`.



(a) (b)

Figure 16.18 (a) A deep but sparse neural network. The connections are pruned using ℓ_1 regularization. At each level, nodes numbered 0 are clamped to 1, so their outgoing weights correspond to the offset/bias terms. (b) Predictions made by the model on the training set. Figure generated by `sparseNnetDemo`, written by Mark Schmidt.

However, we can also use the more principled sparsity-promoting techniques we discussed in Chapter 13. One approach is to use an ℓ_1 regularizer. See Figure 16.18 for an example. Another approach is to use ARD; this is discussed in more detail in Section 16.5.7.5.

16.5.6.3 Soft weight sharing*

Another way to regularize the parameters is to encourage similar weights to share statistical strength. But how do we know which parameters to group together? We can learn this, by using a mixture model. That is, we model $p(\boldsymbol{\theta})$ as a mixture of (diagonal) Gaussians. Parameters that are assigned to the same cluster will share the same mean and variance and thus will have similar values (assuming the variance for that cluster is low). This is called **soft weight sharing** (Nowlan and Hinton 1992). In practice, this technique is not widely used. See e.g., (Bishop 2006a, p271) if you want to know the details.

16.5.6.4 Semi-supervised embedding *

An interesting way to regularize "deep" feedforward neural networks is to encourage the hidden layers to assign similar objects to similar representations. This is useful because it is often easy to obtain "side" information consisting of sets of pairs of similar and dissimilar objects. For example, in a video classification task, neighboring frames can be deemed similar, but frames that are distant in time can be deemed dis-similar (Mobahi et al. 2009). Note that this can be done without collecting any labels.

Let $S_{ij} = 1$ if examples i and j are similar, and $S_{ij} = 0$ otherwise. Let $f(\mathbf{x}_i)$ be some embedding of item $\mathbf{x}_i$, e.g., $f(\mathbf{x}_i) = \mathbf{z}(\mathbf{x}_i, \boldsymbol{\theta})$, where $\mathbf{z}$ is the hidden layer of a neural network. Now define a loss function $L(f(\mathbf{x}_i), f(\mathbf{x}_j), S_{ij})$ that depends on the embedding of two objects,

and the observed similarity measure. For example, we might want to force similar objects to have similar embeddings, and to force the embeddings of dissimilar objects to be a minimal distance apart:

$$L(\mathbf{f}_i, \mathbf{f}_j, S_{ij}) = \begin{cases} ||\mathbf{f}_i - \mathbf{f}_j||^2 & \text{if } S_{ij} = 1 \\ \max(0, m - ||\mathbf{f}_i - \mathbf{f}_j||^2) & \text{if } S_{ij} = 0 \end{cases} \tag{16.80}$$

where m is some minimal margin. We can now define an augmented loss function for training the neural network:

$$\sum_{i \in \mathcal{L}} \text{NLL}(f(\mathbf{x}_i), y_i) + \lambda \sum_{i,j \in \mathcal{U}} L(f(\mathbf{x}_i), f(\mathbf{x}_j), S_{ij}) \tag{16.81}$$

where $\mathcal{L}$ is the labeled training set, $\mathcal{U}$ is the unlabeled training set, and $\lambda \geq 0$ is some tradeoff parameter. This is called **semi-supervised embedding** (Weston et al. 2008).

Such an objective can be easily optimized by stochastic gradient descent. At each iteration, pick a random labeled training example, $(\mathbf{x}_n, y_n)$, and take a gradient step to optimize $\text{NLL}(f(\mathbf{x}_i), y_i)$. Then pick a random pair of similar unlabeled examples $\mathbf{x}_i$, $\mathbf{x}_j$ (these can sometimes be generated on the fly rather than stored in advance), and make a gradient step to optimize $\lambda L(f(\mathbf{x}_i), f(\mathbf{x}_j), 1)$, Finally, pick a random unlabeled example $\mathbf{x}_k$, which with high probability is dissimilar to $\mathbf{x}_i$, and make a gradient step to optimize $\lambda L(f(\mathbf{x}_i), f(\mathbf{x}_k), 0)$.

Note that this technique is effective because it can leverage massive amounts of data. In a related approach, (Collobert and Weston 2008) trained a neural network to distinguish valid English sentences from invalid ones. This was done by taking all 631 million words from English Wikipedia (`en.wikipedia.org`), and then creating windows of length 11 containing neighboring words. This constitutes the positive examples. To create negative examples, the middle word of each window was replaced by a random English word (this is likely to be an "invalid" sentence — either grammatically and/or semantically — with high probability). This neural network was then trained over the course of 1 week, and its latent representation was then used as the input to a supervised semantic role labeling task, for which very little labeled training data is available. (See also (Ando and Zhang 2005) for related work.)

16.5.7 Bayesian inference *

Although MAP estimation is a succesful way to reduce overfitting, there are still some good reasons to want to adopt a fully Bayesian approach to "fitting" neural networks:

- Integrating out the parameters instead of optimizing them is a much stronger form of regularization than MAP estimation.
- We can use Bayesian model selection to determine things like the hyper-parameter settings and the number of hidden units. This is likely to be much faster than cross validation, especially if we have many hyper-parameters (e.g., as in ARD).
- Modelling uncertainty in the parameters will induce uncertainty in our predictive distributions, which is important for certain problems such as active learning and risk-averse decision making.

- We can use online inference methods, such as the extended Kalman filter, to do online learning (Haykin 2001).

One can adopt a variety of approximate Bayesian inference techniques in this context. In this section, we discuss the Laplace approximation, first suggested in (MacKay 1992, 1995b). One can also use hybrid Monte Carlo (Neal 1996), or variational Bayes (Hinton and Camp 1993; Barber and Bishop 1998).

16.5.7.1 Parameter posterior for regression

We start by considering regression, following the presentation of (Bishop 2006a, sec 5.7), which summarizes the work of (MacKay 1992, 1995b). We will use a prior of the form $p(\mathbf{w}) = \mathcal{N}(\mathbf{w}|\mathbf{0}, (1/\alpha)\mathbf{I})$, where $\mathbf{w}$ represents all the weights combined. We will denote the precision of the noise by $\beta = 1/\sigma^2$.

The posterior can be approximated as follows:

$$p(\mathbf{w}|\mathcal{D}, \alpha, \beta) \propto \exp(-E(\mathbf{w})) \tag{16.82}$$

$$E(\mathbf{w}) \triangleq \beta E_D(\mathbf{w}) + \alpha E_W(\mathbf{w}) \tag{16.83}$$

$$E_D(\mathbf{w}) \triangleq \frac{1}{2}\sum_{n=1}^{N}(y_n - f(\mathbf{x}_n, \mathbf{w}))^2 \tag{16.84}$$

$$E_W(\mathbf{w}) \triangleq \frac{1}{2}\mathbf{w}^T\mathbf{w} \tag{16.85}$$

where E_D is the data error, E_W is the prior error, and E is the overall error (negative log prior plus log likelihood). Now let us make a second-order Taylor series approximation of $E(\mathbf{w})$ around its minimum (the MAP estimate)

$$E(\mathbf{w}) \approx E(\mathbf{w}_{MP}) + \frac{1}{2}(\mathbf{w} - \mathbf{w}_{MP})^T\mathbf{A}(\mathbf{w} - \mathbf{w}_{MP}) \tag{16.86}$$

where $\mathbf{A}$ is the Hessian of E:

$$\mathbf{A} = \nabla\nabla E(\mathbf{w}_{MP}) = \beta\mathbf{H} + \alpha\mathbf{I} \tag{16.87}$$

where $\mathbf{H} = \nabla\nabla E_D(\mathbf{w}_{MP})$ is the Hessian of the data error. This can be computed exactly in $O(d^2)$ time, where d is the number of parameters, using a variant of backpropagation (see (Bishop 2006a, sec 5.4) for details). Alternatively, if we use a quasi-Newton method to find the mode, we can use its internally computed (low-rank) approximation to $\mathbf{H}$. (Note that diagonal approximations of $\mathbf{H}$ are usually very inaccurate.) In either case, using this quadratic approximation, the posterior becomes Gaussian:

$$p(\mathbf{w}|\alpha, \beta, \mathcal{D}) \approx \mathcal{N}(\mathbf{w}|\mathbf{w}_{MP}, \mathbf{A}^{-1}) \tag{16.88}$$

16.5.7.2 Parameter posterior for classification

The classification case is the same as the regression case, except $\beta = 1$ and E_D is a cross-entropy error of the form

$$E_D(\mathbf{w}) \triangleq \sum_{n=1}^{N} [y_n \ln f(\mathbf{x}_n, \mathbf{w}) + (1 - y_n) \ln f(\mathbf{x}_n, \mathbf{w})] \tag{16.89}$$

$$\tag{16.90}$$

16.5.7.3 Predictive posterior for regression

The posterior predictive density is given by

$$p(y|\mathbf{x}, \mathcal{D}, \alpha, \beta) = \int \mathcal{N}(y|f(\mathbf{x}, \mathbf{w}), 1/\beta)\mathcal{N}(\mathbf{w}|\mathbf{w}_{MP}, \mathbf{A}^{-1})d\mathbf{w} \tag{16.91}$$

This is not analytically tractable because of the nonlinearity of $f(\mathbf{x}, \mathbf{w})$. Let us therefore construct a first-order Taylor series approximation around the mode:

$$f(\mathbf{x}, \mathbf{w}) \approx f(\mathbf{x}, \mathbf{w}_{MP}) + \mathbf{g}^T(\mathbf{w} - \mathbf{w}_{MP}) \tag{16.92}$$

where

$$\mathbf{g} = \nabla_{\mathbf{w}} f(\mathbf{x}, \mathbf{w})|_{\mathbf{w}=\mathbf{w}_{MP}} \tag{16.93}$$

We now have a linear-Gaussian model with a Gaussian prior on the weights. From Equation 4.126 we have

$$p(y|\mathbf{x}, \mathcal{D}, \alpha, \beta) \approx \mathcal{N}(y|f(\mathbf{x}, \mathbf{w}_{MP}), \sigma^2(\mathbf{x})) \tag{16.94}$$

where the predictive variance depends on the input $\mathbf{x}$ as follows:

$$\sigma^2(\mathbf{x}) = \beta^{-1} + \mathbf{g}^T \mathbf{A}^{-1} \mathbf{g} \tag{16.95}$$

The error bars will be larger in regions of input space where we have little training data. See Figure 16.19 for an example.

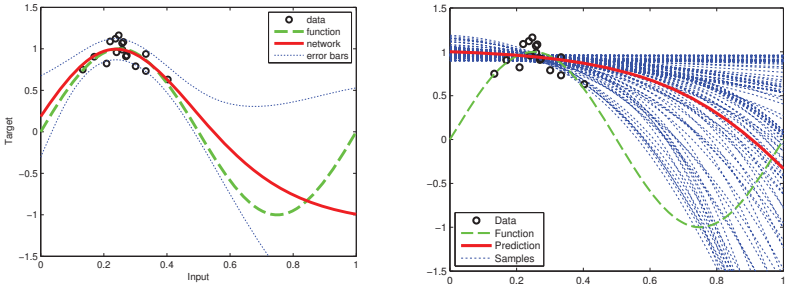
16.5.7.4 Predictive posterior for classification

In this section, we discuss how to approximate $p(y|\mathbf{x}, \mathcal{D})$ in the case of binary classification. The situation is similar to the case of logistic regression, discussed in Section 8.4.4, except in addition the posterior predictive mean is a non-linear function of $\mathbf{w}$. Specifically, we have $\mu = \mathbb{E}[y|\mathbf{x}, \mathbf{w}] = \text{sigm}(a(\mathbf{x}, \mathbf{w}))$, where $a(\mathbf{x}, \mathbf{w})$ is the pre-synaptic output of the final layer. Let us make a linear approximation to this:

$$a(\mathbf{x}, \mathbf{w}) \approx a_{MP}(\mathbf{x}) + \mathbf{g}^T(\mathbf{w} - \mathbf{w}_{MP}) \tag{16.96}$$

where $a_{MP}(\mathbf{x}) = a(\mathbf{x}, \mathbf{w}_{MP})$ and $\mathbf{g} = \nabla_{\mathbf{x}} a(\mathbf{x}, \mathbf{w}_{MP})$ can be found by a modified version of backpropagation. Clearly

$$p(a|\mathbf{x}, \mathcal{D}) \approx \mathcal{N}(a(\mathbf{x}, \mathbf{w}_{MP}), \mathbf{g}(\mathbf{x})^T \mathbf{A}^{-1} \mathbf{g}(\mathbf{x})) \tag{16.97}$$



(a) (b)

Figure 16.19 The posterior predictive density for an MLP with 3 hidden nodes, trained on 16 data points. The dashed green line is the true function. (a) Result of using a Laplace approximation, after performing empirical Bayes to optimize the hyperparameters. The solid red line is the posterior mean prediction, and the dotted blue lines are 1 standard deviation above and below the mean. Figure generated by `mlpRegEvidenceDemo`. (b) Result of using hybrid Monte Carlo, using the same trained hyperparameters as in (a). The solid red line is the posterior mean prediction, and the dotted blue lines are samples from the posterior predictive. Figure generated by `mlpRegHmcDemo`, written by Ian Nabney.

Hence the posterior predictive for the output is

$$p(y=1|\mathbf{x},\mathcal{D}) = \int \text{sigm}(a)p(a|\mathbf{x},\mathcal{D})da \approx \text{sigm}(\kappa(\sigma_a^2)\mathbf{b}^T\mathbf{w}_{MP}) \tag{16.98}$$

where κ is defined by Equation 8.70, which we repeat here for convenience:

$$\kappa(\sigma^2) \triangleq (1+\pi\sigma^2/8)^{-\frac{1}{2}} \tag{16.99}$$

Of course, a simpler (and potentially more accurate) alternative to this is to draw a few samples from the Gaussian posterior and to approximate the posterior predictive using Monte Carlo.

In either case, the effect of taking uncertainty of the parameters into account, as in Section 8.4.4, is to "moderate" the confidence of the output; the decision boundary itself is unaffected, however.

16.5.7.5 ARD for neural networks

Once we have made the Laplace approximation to the posterior, we can optimize the marginal likelihood wrt the hyper-parameters $\boldsymbol{\alpha}$ using the same fixed-point equations as in Section 13.7.4.2. Typically we use one hyper-parameter for the weight vector leaving each node, to achieve an effect similar to group lasso (Section 13.5.1). That is, the prior has the form

$$p(\boldsymbol{\theta}) = \prod_{i=1}^{D} \mathcal{N}(\mathbf{v}_{:,i}|\mathbf{0}, \frac{1}{\alpha_{v,i}}\mathbf{I}) \prod_{j=1}^{H} \mathcal{N}(\mathbf{w}_{:,j}|\mathbf{0}, \frac{1}{\alpha_{w,j}}\mathbf{I}) \tag{16.100}$$

If we find $\alpha_{v,i} = \infty$, then input feature i is irrelevant, and its weight vector $\mathbf{v}_{:,i}$ is pruned out. Similarly, if we find $\alpha_{w,j} = \infty$, then hidden feature j is irrelevant. This is known as automatic

relevancy determination or ARD, which was discussed in detail in Section 13.7. Applying this to neural networks gives us an efficient means of variable selection in non-linear models.

The software package NETLAB contains a simple example of ARD applied to a neural network, called `demard`. This demo creates some data according to a nonlinear regression function $f(x_1, x_2, x_3) = \sin(2\pi x_1) + \epsilon$, where x_2 is a noisy copy of x_1. We see that x_2 and x_3 are irrelevant for predicting the target. However, x_2 is correlated with x_1, which is relevant. Using ARD, the final hyper-parameters are as follows:

$$\boldsymbol{\alpha} = [0.2, \quad 21.4, \quad 249001.8] \tag{16.101}$$

This clearly indicates that feature 3 is irrelevant, feature 2 is only weakly relevant, and feature 1 is very relevant.

16.6 Ensemble learning

Ensemble learning refers to learning a weighted combination of base models of the form

$$f(y|\mathbf{x}, \boldsymbol{\pi}) = \sum_{m \in \mathcal{M}} w_m f_m(y|\mathbf{x}) \tag{16.102}$$

where the w_m are tunable parameters. Ensemble learning is sometimes called a **committee method**, since each base model f_m gets a weighted "vote".

Clearly ensemble learning is closely related to learning adaptive-basis function models. In fact, one can argue that a neural net is an ensemble method, where f_m represents the m'th hidden unit, and w_m are the output layer weights. Also, we can think of boosting as kind of ensemble learning, where the weights on the base models are determined sequentially. Below we describe some other forms of ensemble learning.

16.6.1 Stacking

An obvious way to estimate the weights in Equation 16.102 is to use

$$\hat{\mathbf{w}} = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{i=1}^{N} L(y_i, \sum_{m=1}^{M} w_m f_m(\mathbf{x})) \tag{16.103}$$

However, this will result in overfitting, with w_m being large for the most complex model. A simple solution to this is to use cross-validation. In particular, we can use the LOOCV estimate

$$\hat{\mathbf{w}} = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{i=1}^{N} L(y_i, \sum_{m=1}^{M} w_m \hat{f}_m^{-i}(\mathbf{x})) \tag{16.104}$$

where $\hat{f}_m^{-i}(\mathbf{x})$ is the predictor obtained by training on data excluding $(\mathbf{x}_i, y_i)$. This is known as **stacking**, which stands for "stacked generalization" (Wolpert 1992). This technique is more robust to the case where the "true" model is not in the model class than standard BMA (Clarke 2003). This approach was used by the Netflix team known as "The Ensemble", which tied the submission of the winning team (BellKor's Pragmatic Chaos) in terms of accuracy (Sill et al. 2009). Stacking has also been used for problems such as image segmentation and labeling.

Class	C_1	C_2	C_3	C_4	C_5	C_6	$\cdots$	C_{15}
0	1	1	0	0	0	0	$\cdots$	1
1	0	0	1	1	1	1	$\cdots$	0
					$\vdots$			
9	0	1	1	1	0	0	$\cdots$	0

Table 16.2 Part of a 15-bit error-correcting output code for a 10-class problem. Each row defines a two-class problem. Based on Table 16.1 of (Hastie et al. 2009).

16.6.2 Error-correcting output codes

An interesting form of ensemble learning is known as **error-correcting output codes** or **ECOC** (Dietterich and Bakiri 1995), which can be used in the context of multi-class classification. The idea is that we are trying to decode a symbol (namely the class label) which has C possible states. We could use a bit vector of length $B = \lceil \log_2 C \rceil$ to encode the class label, and train B separate binary classifiers to predict each bit. However, by using more bits, and by designing the codewords to have maximal Hamming distance from each other, we get a method that is more resistant to individual bit-flipping errors (misclassification). For example, in Table 16.2, we use $B = 15$ bits to encode a $C = 10$ class problem. The minimum Hamming distance between any pair of rows is 7. The decoding rule is

$$\hat{c}(\mathbf{x}) = \min_c \sum_{b=1}^{B} |C_{cb} - \hat{p}_b(\mathbf{x})| \tag{16.105}$$

where C_{cb} is the b'th bit of the codeword for class c. (James and Hastie 1998) showed that a random code worked just as well as the optimal code: both methods work by averaging the results of multiple classifiers, thereby reducing variance.

16.6.3 Ensemble learning is not equivalent to Bayes model averaging

In Section 5.3, we discussed Bayesian model selection. An alternative to picking the best model, and then using this to make predictions, is to make a weighted average of the predictions made by each model, i.e., we compute

$$p(y|\mathbf{x}, \mathcal{D}) = \sum_{m \in \mathcal{M}} p(y|\mathbf{x}, m, \mathcal{D}) p(m|\mathcal{D}) \tag{16.106}$$

This is called **Bayes model averaging** (BMA), and can sometimes give better performance than using any single model (Hoeting et al. 1999). Of course, averaging over all models is typically computationally infeasible (analytical integration is obviously not possible in a discrete space, although one can sometimes use dynamic programming to perform the computation exactly, e.g., (Meila and Jaakkola 2006)). A simple approximation is to sample a few models from the posterior. An even simpler approximation (and the one most widely used in practice) is to just use the MAP model.

It is important to note that BMA is not equivalent to ensemble learning (Minka 2000c). This latter technique corresponds to enlarging the model space, by defining a single new model

MODEL	1ST	2ND	3RD	4TH	5TH	6TH	7TH	8TH	9TH	10TH
BST-DT	0.580	0.228	0.160	0.023	0.009	0.000	0.000	0.000	0.000	0.000
RF	0.390	0.525	0.084	0.001	0.000	0.000	0.000	0.000	0.000	0.000
BAG-DT	0.030	0.232	0.571	0.150	0.017	0.000	0.000	0.000	0.000	0.000
SVM	0.000	0.008	0.148	0.574	0.240	0.029	0.001	0.000	0.000	0.000
ANN	0.000	0.007	0.035	0.230	0.606	0.122	0.000	0.000	0.000	0.000
KNN	0.000	0.000	0.000	0.009	0.114	0.592	0.245	0.038	0.002	0.000
BST-STMP	0.000	0.000	0.002	0.013	0.014	0.257	0.710	0.004	0.000	0.000
DT	0.000	0.000	0.000	0.000	0.000	0.000	0.004	0.616	0.291	0.089
LOGREG	0.000	0.000	0.000	0.000	0.000	0.000	0.040	0.312	0.423	0.225
NB	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.030	0.284	0.686

Table 16.3 Fraction of time each method achieved a specified rank, when sorting by mean performance across 11 datasets and 8 metrics. Based on Table 4 of (Caruana and Niculescu-Mizil 2006). Used with kind permission of Alexandru Niculescu-Mizil.

which is a convex combination of base models, as follows:

$$p(y|\mathbf{x}, \boldsymbol{\pi}) = \sum_{m \in \mathcal{M}} \pi_m p(y|\mathbf{x}, m) \tag{16.107}$$

In principle, we can now perform Bayesian inference to compute $p(\boldsymbol{\pi}|\mathcal{D})$; we then make predictions using $p(y|\mathbf{x}, \mathcal{D}) = \int p(y|\mathbf{x}, \boldsymbol{\pi}) p(\boldsymbol{\pi}|\mathcal{D}) d\boldsymbol{\pi}$. However, it is much more common to use point estimation methods for $\boldsymbol{\pi}$, as we saw above.

16.7 Experimental comparison

We have described many different methods for classification and regression. Which one should you use? That depends on which inductive bias you think is most appropriate for your domain. Usually this is hard to assess, so it is common to just try several different methods, and see how they perform empirically. Below we summarize two such comparisons that were carefully conducted (although the data sets that were used are relatively small). See the website `mlcomp.org` for a distributed way to perform large scale comparisons of this kind. Of course, we must always remember the no free lunch theorem (Section 1.4.9), which tells us that there is no universally best learning method.

16.7.1 Low-dimensional features

In 2006, Rich Caruana and Alex Niculescu-Mizil (Caruana and Niculescu-Mizil 2006) conducted a very extensive experimental comparison of 10 different binary classification methods, on 11 different data sets. The 11 data sets all had 5000 training cases, and had test sets containing $\sim 10,000$ examples on average. The number of features ranged from 9 to 200, so this is much lower dimensional than the NIPS 2003 feature selection challenge. 5-fold cross validation was used to assess average test error. (This is separate from any internal CV a method may need to use for model selection.)

The methods they compared are as follows (listed in roughly decreasing order of performance, as assessed by Table 16.3):

- BST-DT: boosted decision trees
- RF: random forest
- BAG-DT: bagged decision trees
- SVM: support vector machine
- ANN: artificial neural network
- KNN: K-nearest neighbors
- BST-STMP: boosted stumps
- DT: decision tree
- LOGREG: logistic regression
- NB: naive Bayes

They used 8 different performance measures, which can be divided into three groups. Threshold metrics just require a point estimate as output. These include accuracy, F-score (Section 5.7.2.3), etc. Ordering/ ranking metrics measure how well positive cases are ordered before the negative cases. These include area under the ROC curve (Section 5.7.2.1), average precision, and the precision/recall break even point. Finally, the probability metrics included cross-entropy (log-loss) and squared error, $(y - \hat{p})^2$. Methods such as SVMs that do not produce calibrated probabilities were post-processed using Platt's logistic regression trick (Section 14.5.2.3), or using isotonic regression. Performance measures were standardized to a 0:1 scale so they could be compared.

Obviously the results vary by dataset and by metric. Therefore just averaging the performance does not necessarily give reliable conclusions. However, one can perform a bootstrap analysis, which shows how robust the conclusions are to such changes. The results are shown in Table 16.3. We see that most of the time, boosted decision trees are the best method, followed by random forests, bagged decision trees, SVMs and neural networks. However, the following methods all did relatively poorly: KNN, stumps, single decision trees, logistic regression and naive Bayes.

These results are generally consistent with conventional wisdom of practioners in the field. Of course, the conclusions may change if there the features are high dimensional and/ or there are lots of irrelevant features (as in Section 16.7.2), or if there is lots of noise, etc.

16.7.2 High-dimensional features

In 2003, the NIPS conference ran a competition where the goal was to solve binary classification problems with large numbers of (mostly irrelevant) features, given small training sets. (This was called a "feature selection" challenge, but performance was measured in terms of predictive accuracy, not in terms of the ability to select features.) The five datasets that were used are summarized in Table 16.4. The term **probe** refers to artifical variables that were added to the problem to make it harder. These have no predictive power, but are correlated with the original features.

Results of the competition are discussed in (Guyon et al. 2006). The overall winner was an approach based on Bayesian neural networks (Neal and Zhang 2006). In a follow-up study

Dataset	Domain	Type	D	% probes	N_{train}	N_{val}	N_{test}
Aracene	Mass spectrometry	Dense	10,000	30	100	100	700
Dexter	Text classification	Sparse	20,000	50	300	300	2000
Dorothea	Drug discovery	Sparse	100,000	50	800	350	800
Gisette	Digit recognition	Dense	5000	30	6000	1000	6500
Madelon	Artificial	Dense	500	96	2000	600	1800

Table 16.4 Summary of the data used in the NIPS 2003 "feature selection" challenge. For the Dorothea datasets, the features are binary. For the others, the features are real-valued.

	Screened features		ARD	
Method	Avg rank	Avg time	Avg rank	Avg time
HMC MLP	1.5	384 (138)	1.6	600 (186)
Boosted MLP	3.8	9.4 (8.6)	2.2	35.6 (33.5)
Bagged MLP	3.6	3.5 (1.1)	4.0	6.4 (4.4)
Boosted trees	3.4	3.03 (2.5)	4.0	34.1 (32.4)
Random forests	2.7	1.9 (1.7)	3.2	11.2 (9.3)

Table 16.5 Performance of different methods on the NIPS 2003 "feature selection" challenge. (HMC stands for hybrid Monte Carlo; see Section 24.5.4.) We report the average rank (lower is better) across the 5 datasets. We also report the average training time in minutes (standard error in brackets). The MCMC and bagged MLPs use two hidden layers of 20 and 8 units. The boosted MLPs use one hidden layer with 2 or 4 hidden units. The boosted trees used depths between 2 and 9, and shrinkage between 0.001 and 0.1. Each tree was trained on 80% of the data chosen at random at each step (so-called **stochastic gradient boosting**). From Table 11.3 of (Hastie et al. 2009).

(Johnson 2009), Bayesian neural nets (MLPs with 2 hidden layers) were compared to several other methods based on bagging and boosting. Note that all of these methods are quite similar: in each case, the prediction has the form

$$\hat{f}(\mathbf{x}_*) = \sum_{m=1}^{M} w_m \mathbb{E}\left[y|\mathbf{x}_*, \boldsymbol{\theta}_m\right] \tag{16.108}$$

The Bayesian MLP was fit by MCMC (hybrid Monte Carlo), so we set $w_m = 1/M$ and set $\boldsymbol{\theta}_m$ to a draw from the posterior. In bagging, we set $w_m = 1/M$ and $\boldsymbol{\theta}_m$ is estimated by fitting the model to a bootstrap sample from the data. In boosting, we set $w_m = 1$ and the $\boldsymbol{\theta}_m$ are estimated sequentially.

To improve computational and statistical performance, some feature selection was performed. Two methods were considered: simple uni-variate screening using T-tests, and a method based on MLP+ARD. Results of this follow-up study are shown in Table 16.5. We see that Bayesian MLPs are again the winner. In second place are either random forests or boosted MLPs, depending on the preprocessing. However, it is not clear how statistically significant these differences are, since the test sets are relatively small.

In terms of training time, we see that MCMC is much slower than the other methods. It would be interesting to see how well deterministic Bayesian inference (e.g., Laplace approximation) would perform. (Obviously it will be much faster, but the question is: how much would one lose

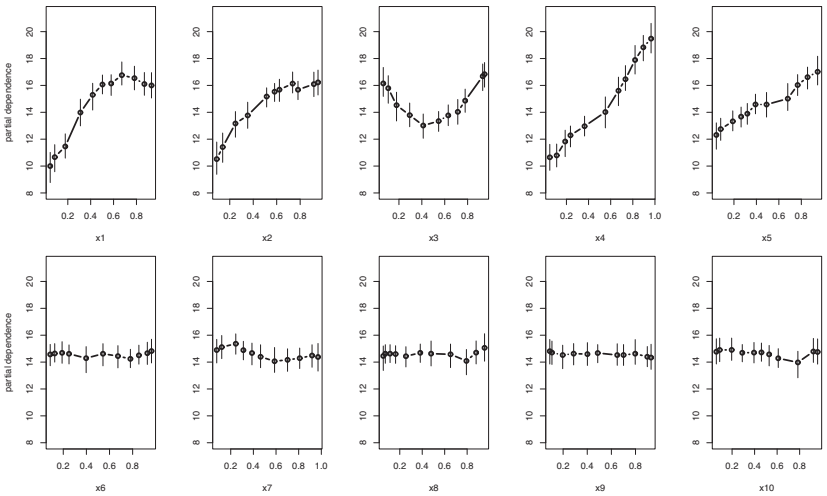


Figure 16.20 Partial dependence plots for the 10 predictors in Friedman's synthetic 5-dimensional regression problem. Source: Figure 4 of (Chipman et al. 2010) . Used with kind permission of Hugh Chipman.

in statistical performance?)

16.8 Interpreting black-box models

Linear models are popular in part because they are easy to interpet. However, they often are poor predictors, which makes them a poor proxy for "nature's mechanism". Thus any conclusions about the importance of particular variables should only be based on models that have good predictive accuracy (Breiman 2001b). (Interestingly, many standard statistical tests of "goodness of fit" do not test the predictive accuracy of a model.)

In this chapter, we studied **black-box** models, which do have good predictive accuracy. Unfortunately, they are hard to interpret directly. Fortunately, there are various heuristics we can use to "probe" such models, in order to assess which input variables are the most important.

As a simple example, consider the following non-linear function, first proposed (Friedman 1991) to illustrate the power of MARS:

$$f(\mathbf{x}) = 10\sin(\pi x_1 x_2) + 20(x_3 - 0.5)^2 + 10x_4 + 5x_5 + \epsilon \tag{16.109}$$

where $\epsilon \sim \mathcal{N}(0,1)$. We see that the output is a complex function of the inputs. By augmenting the $\mathbf{x}$ vector with additional irrelevant random variables, all drawn uniform on $[0,1]$, we can create a challenging feature selection problem. In the experiments below, we add 5 extra dummy variables.

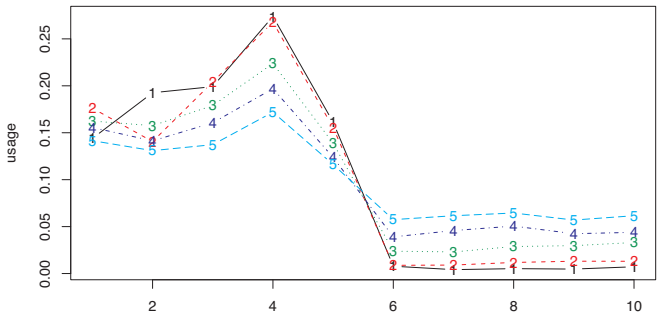


Figure 16.21 Average usage of each variable in a BART model fit to data where only the first 5 features are relevant. The different coloured lines correspond to different numbers of trees in the ensemble. Source: Figure 5 of (Chipman et al. 2010) . Used with kind permission of Hugh Chipman.

One useful way to measure the effect of a set s of variables on the output is to compute a **partial dependence plot** (Friedman 2001). This is a plot of $f(\mathbf{x}_s)$ vs $\mathbf{x}_s$, where $f(\mathbf{x}_s)$ is defined as the response to $\mathbf{x}_s$ with the other predictors averaged out:

$$f(\mathbf{x}_s) = \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x}_s, \mathbf{x}_{i,-s}) \tag{16.110}$$

Figure 16.20 shows an example where we use sets corresponding to each single variable. The data was generated from Equation 16.109, with 5 irrelevant variables added. We then fit a BART model (Section 16.2.5) and computed the partial dependence plots. We see that the predicted response is invariant for $s \in \{6, \ldots, 10\}$, indicating that these variables are (marginally) irrelevant. The response is roughly linear in x_4 and x_5, and roughly quadratic in x_3. (The error bars are obtained by computing empirical quantiles of $f(\mathbf{x}, \boldsymbol{\theta})$ based on posterior samples of $\boldsymbol{\theta}$; alternatively, we can use bootstrap.)

Another very useful summary computes the **relative importance of predictor variables**. This can be thought of as a nonlinear, or even "model free", way of performing variable selection, although the technique is restricted to ensembles of trees. The basic idea, originally proposed in (Breiman et al. 1984), is to count how often variable j is used as a node in any of the trees. In particular, let $v_j = \frac{1}{M}\sum_{m=1}^{M} \mathbb{I}(j \in T_m)$ be the proportion of all splitting rules that use x_j, where T_m is the m'th tree. If we can sample the posterior of trees, $p(T_{1:M}|\mathcal{D})$, we can easily compute the posterior for v_j. Alternatively, we can use bootstrap.

Figure 16.21 gives an example, using BART. We see that the five relevant variables are chosen much more than the five irrelevant variables. As we increase the number M of trees, all the variables are more likely to be chosen, reducing the sensitivity of this method, but for small M, the method is farily diagnostic.

Exercises

Exercise 16.1 Nonlinear regression for inverse dynamics

In this question, we fit a model which can predict what torques a robot needs to apply in order to make its arm reach a desired point in space. The data was collected from a SARCOS robot arm with 7 degrees of freedom. The input vector $\mathbf{x} \in \mathbb{R}^{21}$ encodes the desired position, velocity and accelaration of the 7 joints. The output vector $\mathbf{y} \in \mathbb{R}^7$ encodes the torques that should be applied to the joints to reach that point. The mapping from $\mathbf{x}$ to $\mathbf{y}$ is highly nonlinear.

We have $N = 48,933$ training points and $N_{test} = 4,449$ testing points. For simplicity, we following standard practice and focus on just predicting a scalar output, namely the torque for the first joint.

Download the data from `http://www.gaussianprocess.org/gpml`. Standardize the inputs so they have zero mean and unit variance on the training set, and center the outputs so they have zero mean on the training set. Apply the corresponding transformations to the test data. Below we will describe various models which you should fit to this transformed data. Then make predictions and compute the standardized mean squared error on the test set as follows:

$$SMSE = \frac{\frac{1}{N_{test}} \sum_{i=1}^{N_{test}} (y_i - \hat{y}_i)^2}{\sigma^2} \tag{16.111}$$

where $\sigma^2 = \frac{1}{N_{train}} \sum_{i=1}^{Ntrain} (y_i - \overline{y})^2$ is the variance of the output computed on the training set.

a. The first method you should try is standard linear regression. Turn in your numbers and code. (According to (Rasmussen and Williams 2006, p24), you should be able to achieve a SMSE of 0.075 using this method.)

b. Now try running K-means clustering (using cross validation to pick K). Then fit an RBF network to the data, using the $\boldsymbol{\mu}_k$ estimated by K-means. Use CV to estimate the RBF bandwidth. What SMSE do you get? Turn in your numbers and code. (According to (Rasmussen and Williams 2006, p24), Gaussian process regression can get an SMSE of 0.011, so the goal is to get close to that.)

c. Now try fitting a feedforward neural network. Use CV to pick the number of hidden units and the strength of the ℓ_2 regularizer. What SMSE do you get? Turn in your numbers and code.

17 *Markov and hidden Markov models*

17.1 Introduction

In this chapter, we discuss probabilistic models for sequences of observations, $X_1, \ldots, X_T$, of arbitrary length T. Such models have applications in computational biology, natural language processing, time series forecasting, etc. We focus on the case where we the observations occur at discrete "time steps", although "time" may also refer to locations within a sequence.

17.2 Markov models

Recall from Section 10.2.2 that the basic idea behind a Markov chain is to assume that X_t captures all the relevant information for predicting the future (i.e., we assume it is a sufficient statistic). If we assume discrete time steps, we can write the joint distribution as follows:

$$p(X_{1:T}) = p(X_1)p(X_2|X_1)p(X_3|X_2)\ldots = p(X_1)\prod_{t=2}^{T} p(X_t|X_{t-1}) \tag{17.1}$$

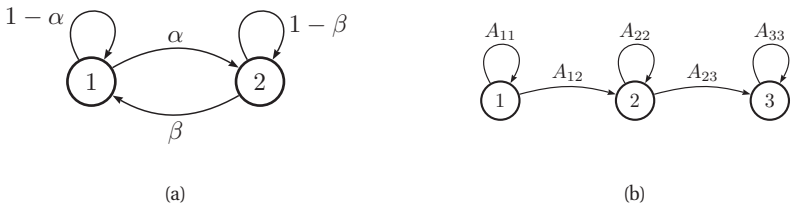
This is called a **Markov chain** or **Markov model**.

If we assume the transition function $p(X_t|X_{t-1})$ is independent of time, then the chain is called **homogeneous**, **stationary**, or **time-invariant**. This is an example of **parameter tying**, since the same parameter is shared by multiple variables. This assumption allows us to model an arbitrary number of variables using a fixed number of parameters; such models are called **stochastic processes**.

If we assume that the observed variables are discrete, so $X_t \in \{1, \ldots, K\}$, this is called a discrete-state or finite-state Markov chain. We will make this assumption throughout the rest of this section.

17.2.1 Transition matrix

When X_t is discrete, so $X_t \in \{1, \ldots, K\}$, the conditional distribution $p(X_t|X_{t-1})$ can be written as a $K \times K$ matrix, known as the **transition matrix A**, where $A_{ij} = p(X_t = j|X_{t-1} = i)$ is the probability of going from state i to state j. Each row of the matrix sums to one, $\sum_j A_{ij} = 1$, so this is called a **stochastic matrix**.



(a) (b)

Figure 17.1 State transition diagrams for some simple Markov chains. Left: a 2-state chain. Right: a 3-state left-to-right chain.

A stationary, finite-state Markov chain is equivalent to a **stochastic automaton**. It is common to visualize such automata by drawing a directed graph, where nodes represent states and arrows represent legal transitions, i.e., non-zero elements of $\mathbf{A}$. This is known as a **state transition diagram**. The weights associated with the arcs are the probabilities. For example, the following 2-state chain

$$\mathbf{A} = \begin{pmatrix} 1-\alpha & \alpha \\ \beta & 1-\beta \end{pmatrix} \tag{17.2}$$

is illustrated in Figure 17.1(left). The following 3-state chain

$$\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & 0 \\ 0 & A_{22} & A_{23} \\ 0 & 0 & 1 \end{pmatrix} \tag{17.3}$$

is illustrated in Figure 17.1(right). This is called a **left-to-right transition matrix**, and is commonly used in speech recognition (Section 17.6.2).

The A_{ij} element of the transition matrix specifies the probability of getting from i to j in one step. The n-step transition matrix $\mathbf{A}(n)$ is defined as

$$A_{ij}(n) \triangleq p(X_{t+n} = j | X_t = i) \tag{17.4}$$

which is the probability of getting from i to j in exactly n steps. Obviously $\mathbf{A}(1) = \mathbf{A}$. The **Chapman-Kolmogorov** equations state that

$$A_{ij}(m+n) = \sum_{k=1}^{K} A_{ik}(m) A_{kj}(n) \tag{17.5}$$

In words, the probability of getting from i to j in $m+n$ steps is just the probability of getting from i to k in m steps, and then from k to j in n steps, summed up over all k. We can write the above as a matrix multiplication

$$\mathbf{A}(m+n) = \mathbf{A}(m)\mathbf{A}(n) \tag{17.6}$$

Hence

$$\mathbf{A}(n) = \mathbf{A}\ \mathbf{A}(n-1) = \mathbf{A}\ \mathbf{A}\ \mathbf{A}(n-2) = \cdots = \mathbf{A}^n \tag{17.7}$$

Thus we can simulate multiple steps of a Markov chain by "powering up" the transition matrix.

```
SAYS IT'S NOT IN THE CARDS LEGENDARY RECONNAISSANCE BY ROLLIE
DEMOCRACIES UNSUSTAINABLE COULD STRIKE REDLINING VISITS TO PROFIT
BOOKING WAIT HERE AT MADISON SQUARE GARDEN COUNTY COURTHOUSE WHERE HE
HAD BEEN DONE IN THREE ALREADY IN ANY WAY IN WHICH A TEACHER
```

Table 17.1 Example output from an 4-gram word model, trained using backoff smoothing on the Broadcast News corpus. The first 4 words are specified by hand, the model generates the 5th word, and then the results are fed back into the model. Source: `http://www.fit.vutbr.cz/~imikolov/rnnlm/gen-4gram.txt` .

17.2.2 Application: Language modeling

One important application of Markov models is to make statistical **language models**, which are probability distributions over sequences of words. We define the state space to be all the words in English (or some other language). The marginal probabilities $p(X_t = k)$ are called **unigram statistics**. If we use a first-order Markov model, then $p(X_t = k|X_{t-1} = j)$ is called a **bigram model**. If we use a second-order Markov model, then $p(X_t = k|X_{t-1} = j, X_{t-2} = i)$ is called a **trigram model**. And so on. In general these are called **n-gram models**. For example, Figure 17.2 shows 1-gram and 2-grams counts for the *letters* $\{a, \ldots, z, -\}$ (where - represents space) estimated from Darwin's *On The Origin Of Species*.

Language models can be used for several things, such as the following:

- **Sentence completion** A language model can predict the next word given the previous words in a sentence. This can be used to reduce the amount of typing required, which is particularly important for disabled users (see e.g., David Mackay's Dasher system[1]), or uses of mobile devices.
- **Data compression** Any density model can be used to define an encoding scheme, by assigning short codewords to more probable strings. The more accurate the predictive model, the fewer the number of bits it requires to store the data.
- **Text classification** Any density model can be used as a class-conditional density and hence turned into a (generative) classifier. Note that using a 0-gram class-conditional density (i.e., only unigram statistics) would be equivalent to a naive Bayes classifier (see Section 3.5).
- **Automatic essay writing** One can sample from $p(x_{1:t})$ to generate artificial text. This is one way of assessing the quality of the model. In Table 17.1, we give an example of text generated from a 4-gram model, trained on a corpus with 400 million words. ((Tomas et al. 2011) describes a much better language model, based on recurrent neural networks, which generates much more semantically plausible text.)

1. `http://www.inference.phy.cam.ac.uk/dasher/`

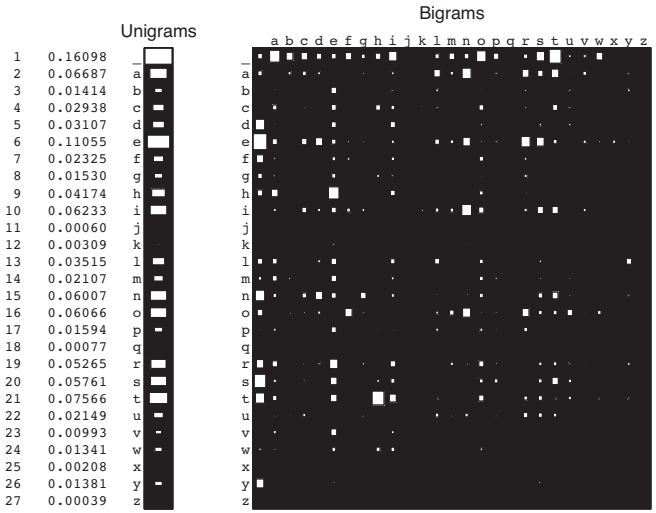


Figure 17.2 Unigram and bigram counts from Darwin's *On The Origin Of Species*. The 2D picture on the right is a Hinton diagram of the joint distribution. The size of the white squares is proportional to the value of the entry in the corresponding vector/ matrix. Based on (MacKay 2003, p22). Figure generated by `ngramPlot`.

17.2.2.1 MLE for Markov language models

We now discuss a simple way to estimate the transition matrix from training data. The probability of any particular sequence of length T is given by

$$p(x_{1:T}|\boldsymbol{\theta}) = \pi(x_1)A(x_1, x_2)\dots A(x_{T-1}, x_T) \tag{17.8}$$

$$= \prod_{j=1}^{K}(\pi_j)^{\mathbb{I}(x_1=j)} \prod_{t=2}^{T}\prod_{j=1}^{K}\prod_{k=1}^{K}(A_{jk})^{\mathbb{I}(x_t=k, x_{t-1}=j)} \tag{17.9}$$

Hence the log-likelihood of a set of sequences $\mathcal{D} = (\mathbf{x}_1, \dots, \mathbf{x}_N)$, where $\mathbf{x}_i = (x_{i1}, \dots, x_{i,T_i})$ is a sequence of length T_i, is given by

$$\log p(\mathcal{D}|\boldsymbol{\theta}) = \sum_{i=1}^{N}\log p(\mathbf{x}_i|\boldsymbol{\theta}) = \sum_j N_j^1 \log \pi_j + \sum_j\sum_k N_{jk}\log A_{jk} \tag{17.10}$$

where we define the following counts:

$$N_j^1 \triangleq \sum_{i=1}^{N}\mathbb{I}(x_{i1}=j), \quad N_{jk} \triangleq \sum_{i=1}^{N}\sum_{t=1}^{T_i-1}\mathbb{I}(x_{i,t}=j, x_{i,t+1}=k) \tag{17.11}$$

Hence we can write the MLE as the normalized counts:

$$\hat{\pi}_j = \frac{N_j^1}{\sum_j N_j^1}, \quad \hat{A}_{jk} = \frac{N_{jk}}{\sum_k N_{jk}} \tag{17.12}$$

These results can be extended in a straightforward way to higher order Markov models. However, the problem of zero-counts becomes very acute whenever the number of states K, and/or the order of the chain, n, is large. An n-gram models has $O(K^n)$ parameters. If we have $K \sim 50,000$ words in our vocabulary, then a bi-gram model will have about 2.5 billion free parameters, corresponding to all possible word pairs. It is very unlikely we will see all of these in our training data. However, we do not want to predict that a particular word string is totally impossible just because we happen not to have seen it in our training text — that would be a severe form of overfitting.[2]

A simple solution to this is to use add-one smoothing, where we simply add one to all the empirical counts before normalizing. The Bayesian justification for this is given in Section 3.3.4.1. However add-one smoothing assumes all n-grams are equally likely, which is not very realistic. A more sophisticated Bayesian approach is discussed in Section 17.2.2.2.

An alternative to using smart priors is to gather lots and lots of data. For example, Google has fit n-gram models (for $n = 1 : 5$) based on one trillion words extracted from the web. Their data, which is over 100GB when uncompressed, is publically available.[3] An example of their data, for a set of 4-grams, is shown below.

```
serve as the incoming 92
serve as the incubator 99
serve as the independent 794
serve as the index 223
serve as the indication 72
serve as the indicator 120
serve as the indicators 45
serve as the indispensable 111
serve as the indispensible 40
serve as the individual 234
...
```

Although such an approach, based on "brute force and ignorance", can be successful, it is rather unsatisfying, since it is clear that this is not how humans learn (see e.g., (Tenenbaum and Xu 2000)). A more refined Bayesian approach, that needs much less data, is described in Section 17.2.2.2.

17.2.2.2 Empirical Bayes version of deleted interpolation

A common heuristic used to fix the sparse data problem is called **deleted interpolation** (Chen and Goodman 1996). This defines the transition matrix as a convex combination of the bigram

2. A famous example of an improbable, but syntactically valid, English word string, due to Noam Chomsky, is "colourless green ideas sleep furiously". We would not want our model to predict that this string is impossible. Even ungrammatical constructs should be allowed by our model with a certain probability, since people frequently violate grammatical rules, especially in spoken language.

3. See `http://googleresearch.blogspot.com/2006/08/all-our-n-gram-are-belong-to-you.html` for details.

frequencies $f_{jk} = N_{jk}/N_j$ and the unigram frequencies $f_k = N_k/N$:

$$A_{jk} = (1 - \lambda) f_{jk} + \lambda f_k \tag{17.13}$$

The term λ is usually set by cross validation. There is also a closely related technique called **backoff smoothing**; the idea is that if f_{jk} is too small, we "back off" to a more reliable estimate, namely f_k.

We will now show that the deleted interpolation heuristic is an approximation to the predictions made by a simple hierarchical Bayesian model. Our presentation follows (McKay and Peto 1995). First, let us use an independent Dirichlet prior on each row of the transition matrix:

$$\mathbf{A}_j \sim \text{Dir}(\alpha_0 m_1, \ldots, \alpha_0 m_K) = \text{Dir}(\alpha_0 \mathbf{m}) = \text{Dir}(\boldsymbol{\alpha}) \tag{17.14}$$

where $\mathbf{A}_j$ is row j of the transition matrix, $\mathbf{m}$ is the prior mean (satisfying $\sum_k m_k = 1$) and α_0 is the prior strength. We will use the same prior for each row: see Figure 17.3.

The posterior is given by $\mathbf{A}_j \sim \text{Dir}(\boldsymbol{\alpha} + \mathbf{N}_j)$, where $\mathbf{N}_j = (N_{j1}, \ldots, N_{jK})$ is the vector that records the number of times we have transitioned out of state j to each of the other states. From Equation 3.51, the posterior predictive density is

$$p(X_{t+1} = k | X_t = j, \mathcal{D}) = \overline{A}_{jk} = \frac{N_{jk} + \alpha m_k}{N_j + \alpha_0} = \frac{f_{jk} N_j + \alpha m_k}{N_j + \alpha_0} = (1 - \lambda_j) f_{jk} + \lambda_j m_k \tag{17.15}$$

where $\overline{A}_{jk} = \mathbb{E}[A_{jk} | \mathcal{D}, \boldsymbol{\alpha}]$ and

$$\lambda_j = \frac{\alpha}{N_j + \alpha_0} \tag{17.16}$$

This is very similar to Equation 17.13 but not identical. The main difference is that the Bayesian model uses a context-dependent weight λ_j to combine m_k with the empirical frequency f_{jk}, rather than a fixed weight λ. This is like *adaptive* deleted interpolation. Furthermore, rather than backing off to the empirical marginal frequencies f_k, we back off to the model parameter m_k.

The only remaining question is: what values should we use for α and $\mathbf{m}$? Let's use empirical Bayes. Since we assume each row of the transition matrix is a priori independent given $\boldsymbol{\alpha}$, the marginal likelihood for our Markov model is found by applying Equation 5.24 to each row:

$$p(\mathcal{D} | \boldsymbol{\alpha}) = \prod_j \frac{B(\mathbf{N}_j + \boldsymbol{\alpha})}{B(\boldsymbol{\alpha})} \tag{17.17}$$

where $\mathbf{N}_j = (N_{j1}, \ldots, N_{jK})$ are the counts for leaving state j and $B(\boldsymbol{\alpha})$ is the generalized beta function.

We can fit this using the methods discussed in (Minka 2000e). However, we can also use the following approximation (McKay and Peto 1995, p12):

$$m_k \propto |\{j : N_{jk} > 0\}| \tag{17.18}$$

This says that the prior probability of word k is given by the number of different contexts in which it occurs, rather than the number of times it occurs. To justify the reasonableness of this result, Mackay and Peto (McKay and Peto 1995) give the following example.

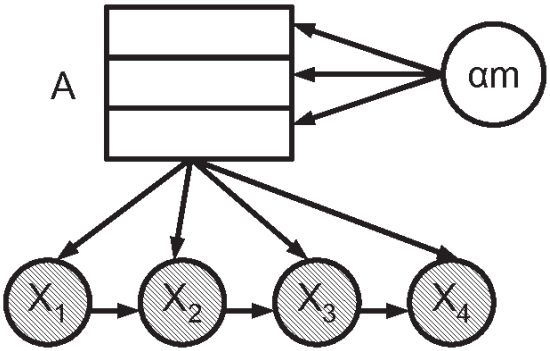


Figure 17.3 A Markov chain in which we put a different Dirichlet prior on every row of the transition matrix **A**, but the hyperparameters of the Dirichlet are shared.

```
Imagine, you see, that the language, you see, has, you see, a
frequently occuring couplet 'you see', you see, in which the second
word of the couplet, see, follows the first word, you, with very high
probability, you see. Then the marginal statistics, you see, are going
to become hugely dominated, you see, by the words you and see, with
equal frequency, you see.
```

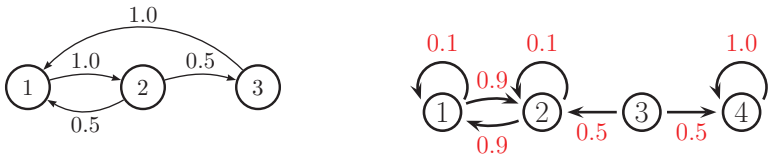
If we use the standard smoothing formula, Equation 17.13, then P(you|novel) and P(see|novel), for some novel context word not seen before, would turn out to be the same, since the marginal frequencies of 'you' and 'see' are the same (11 times each). However, this seems unreasonable. 'You' appears in many contexts, so P(you|novel) should be high, but 'see' only follows 'you', so P(see|novel) should be low. If we use the Bayesian formula Equation 17.15, we will get this effect for free, since we back off to m_k not f_k, and m_k will be large for 'you' and small for 'see' by Equation 17.18.

Unfortunately, although elegant, this Bayesian model does not beat the state-of-the-art language model, known as **interpolated Kneser-Ney** (Kneser and Ney 1995; Chen and Goodman 1998). However, in (Teh 2006), it was shown how one can build a non-parametric Bayesian model which outperforms interpolated Kneser-Ney, by using variable-length contexts. In (Wood et al. 2009), this method was extended to create the "sequence memoizer", which is currently (2010) the best-performing language model.[4]

17.2.2.3 Handling out-of-vocabulary words

While the above smoothing methods handle the case where the counts are small or even zero, none of them deal with the case where the test set may contain a completely novel word. In particular, they all assume that the words in the vocabulary (i.e., the state space of X_t) is fixed and known (typically it is the set of unique words in the training data, or in some dictionary).

4. Interestingly, these non-parametric methods are based on posterior inference using MCMC (Section 24.1) and/or particle filtering (Section 23.5), rather than optimization methods such as EB. Despite this, they are quite efficient.



(a) (b)

Figure 17.4 Some Markov chains. (a) A 3-state aperiodic chain. (b) A reducible 4-state chain.

Even if all $\overline{A}_{jk}$'s are non-zero, none of these models will predict a novel word outside of this set, and hence will assign zero probability to a test sentence with an unfamiliar word. (Unfamiliar words are bound to occur, because the set of words is an open class. For example, the set of proper nouns (names of people and places) is unbounded.)

A standard heuristic to solve this problem is to replace all novel words with the special symbol **unk**, which stands for "unknown". A certain amount of probability mass is held aside for this event.

A more principled solution would be to use a Dirichlet process, which can generate a countably infinite state space, as the amount of data increases (see Section 25.2.2). If all novel words are "accepted" as genuine words, then the system has no predictive power, since any misspelling will be considered a new word. So the novel word has to be seen frequently enough to warrant being added to the vocabulary. See e.g., (Friedman and Singer 1999; Griffiths and Tenenbaum 2001) for details.

17.2.3 Stationary distribution of a Markov chain *

We have been focussing on Markov models as a way of defining joint probability distributions over sequences. However, we can also interpret them as stochastic dynamical systems, where we "hop" from one state to another at each time step. In this case, we are often interested in the long term distribution over states, which is known as the **stationary distribution** of the chain. In this section, we discuss some of the relevant theory. Later we will consider two important applications: Google's PageRank algorithm for ranking web pages (Section 17.2.4), and the MCMC algorithm for generating samples from hard-to-normalize probability distributions (Chapter 24).

17.2.3.1 What is a stationary distribution?

Let $A_{ij} = p(X_t = j|X_{t-1} = i)$ be the one-step transition matrix, and let $\pi_t(j) = p(X_t = j)$ be the probability of being in state j at time t. It is conventional in this context to assume that $\boldsymbol{\pi}$ is a *row* vector. If we have an initial distribution over states of $\boldsymbol{\pi}_0$, then at time 1 we have

$$\pi_1(j) = \sum_i \pi_0(i) A_{ij} \tag{17.19}$$

or, in matrix notation,

$$\boldsymbol{\pi}_1 = \boldsymbol{\pi}_0 \mathbf{A} \tag{17.20}$$

We can imagine iterating these equations. If we ever reach a stage where

$$\boldsymbol{\pi} = \boldsymbol{\pi}\mathbf{A} \tag{17.21}$$

then we say we have reached the **stationary distribution** (also called the **invariant distribution** or **equilibrium distribution**). Once we enter the stationary distribution, we will never leave.

For example, consider the chain in Figure 17.4(a). To find its stationary distribution, we write

$$\begin{pmatrix}\pi_1 & \pi_2 & \pi_3\end{pmatrix} = \begin{pmatrix}\pi_1 & \pi_2 & \pi_3\end{pmatrix} \begin{pmatrix} 1-A_{12}-A_{13} & A_{12} & A_{13} \\ A_{21} & 1-A_{21}-A_{23} & A_{23} \\ A_{31} & A_{32} & 1-A_{31}-A_{32} \end{pmatrix} \tag{17.22}$$

so

$$\pi_1 = \pi_1(1 - A_{12} - A_{12}) + \pi_2 A_{21} + \pi_3 A_{31} \tag{17.23}$$

or

$$\pi_1(A_{12} + A_{13}) = \pi_2 A_{21} + \pi_3 A_{31} \tag{17.24}$$

In general, we have

$$\pi_i \sum_{j \neq i} A_{ij} = \sum_{j \neq i} \pi_j A_{ji} \tag{17.25}$$

In other words, the probability of being in state i times the net flow out of state i must equal the probability of being in each other state j times the net flow from that state into i. These are called the **global balance equations**. We can then solve these equations, subject to the constraint that $\sum_j \pi_j = 1$.

17.2.3.2 Computing the stationary distribution

To find the stationary distribution, we can just solve the eigenvector equation $\mathbf{A}^T\mathbf{v} = \mathbf{v}$, and then to set $\boldsymbol{\pi} = \mathbf{v}^T$, where $\mathbf{v}$ is an eigenvector with eigenvalue 1. (We can be sure such an eigenvector exists, since $\mathbf{A}$ is a row-stochastic matrix, so $\mathbf{A1} = \mathbf{1}$; also recall that the eigenvalues of $\mathbf{A}$ and $\mathbf{A}^T$ are the same.) Of course, since eigenvectors are unique only up to constants of proportionality, we must normalize $\mathbf{v}$ at the end to ensure it sums to one.

Note, however, that the eigenvectors are only guaranteed to be real-valued if the matrix is positive, $A_{ij} > 0$ (and hence $A_{ij} < 1$, due to the sum-to-one constraint). A more general approach, which can handle chains where some transition probabilities are 0 or 1 (such as Figure 17.4(a)), is as follows (Resnick 1992, p138). We have K constraints from $\boldsymbol{\pi}(\mathbf{I} - \mathbf{A}) = \mathbf{0}_{K\times 1}$ and 1 constraint from $\boldsymbol{\pi}\mathbf{1}_{K\times 1} = 0$. Since we only have K unknowns, this is overconstrained. So let us replace any column (e.g., the last) of $\mathbf{I} - \mathbf{A}$ with $\mathbf{1}$, to get a new matrix, call it $\mathbf{M}$. Next we define $\mathbf{r} = [0, 0, \ldots, 1]$, where the 1 in the last position corresponds to the column of all 1s in $\mathbf{M}$. We then solve $\boldsymbol{\pi}\mathbf{M} = \mathbf{r}$. For example, for a 3 state chain we have to solve this linear system:

$$\begin{pmatrix}\pi_1 & \pi_2 & \pi_3\end{pmatrix} \begin{pmatrix} 1-A_{11} & -A_{12} & 1 \\ -A_{21} & 1-A_{22} & 1 \\ -A_{31} & -A_{32} & 1 \end{pmatrix} = \begin{pmatrix}0 & 0 & 1\end{pmatrix} \tag{17.26}$$

For the chain in Figure 17.4(a) we find $\boldsymbol{\pi} = [0.4, 0.4, 0.2]$. We can easily verify this is correct, since $\boldsymbol{\pi} = \boldsymbol{\pi}\mathbf{A}$. See `mcStatDist` for some Matlab code.

Unfortunately, not all chains have a stationary distribution. as we explain below.

17.2.3.3 When does a stationary distribution exist? *

Consider the 4-state chain in Figure 17.4(b). If we start in state 4, we will stay there forever, since 4 is an **absorbing state**. Thus $\boldsymbol{\pi} = (0, 0, 0, 1)$ is one possible stationary distribution. However, if we start in 1 or 2, we will oscillate between those two states for ever. So $\boldsymbol{\pi} = (0.5, 0.5, 0, 0)$ is another possible stationary distribution. If we start in state 3, we could end up in either of the above stationary distributions.

We see from this example that a necessary condition to have a unique stationary distribution is that the state transition diagram be a singly connected component, i.e., we can get from any state to any other state. Such chains are called **irreducible**.

Now consider the 2-state chain in Figure 17.1(a). This is irreducible provided $\alpha, \beta > 0$. Suppose $\alpha = \beta = 0.9$. It is clear by symmetry that this chain will spend 50% of its time in each state. Thus $\boldsymbol{\pi} = (0.5, 0.5)$. But now suppose $\alpha = \beta = 1$. In this case, the chain will oscillate between the two states, but the long-term distribution on states depends on where you start from. If we start in state 1, then on every odd time step (1,3,5,...) we will be in state 1; but if we start in state 2, then on every odd time step we will be in state 2.

This example motivates the following definition. Let us say that a chain has a **limiting distribution** if $\pi_j = \lim_{n\to\infty} A_{ij}^n$ exists and is independent of i, for all j. If this holds, then the long-run distribution over states will be independent of the starting state:

$$P(X_t = j) = \sum_i P(X_0 = i) A_{ij}(t) \to \pi_j \text{ as } t \to \infty \tag{17.27}$$

Let us now characterize when a limiting distribution exists. Define the **period** of state i to be

$$d(i) = \text{gcd}\{t : A_{ii}(t) > 0\} \tag{17.28}$$

where gcd stands for **greatest common divisor**, i.e., the largest integer that divides all the members of the set. For example, in Figure 17.4(a), we have $d(1) = d(2) = gcd(2, 3, 4, 6, ...) = 1$ and $d(3) = gcd(3, 5, 6, ...) = 1$. We say a state i is **aperiodic** if $d(i) = 1$. (A sufficient condition to ensure this is if state i has a self-loop, but this is not a necessary condition.) We say a chain is aperiodic if all its states are aperiodic. One can show the following important result:

Theorem 17.2.1. *Every irreducible (singly connected), aperiodic finite state Markov chain has a limiting distribution, which is equal to $\boldsymbol{\pi}$, its unique stationary distribution.*

A special case of this result says that every regular finite state chain has a unique stationary distribution, where a **regular** chain is one whose transition matrix satisfies $A_{ij}^n > 0$ for some integer n and all i, j, i.e., it is possible to get from any state to any other state in n steps. Consequently, after n steps, the chain could be in any state, no matter where it started. One can show that sufficient conditions to ensure regularity are that the chain be irreducible (singly connected) and that every state have a self-transition.

To handle the case of Markov chains whose state-space is not finite (e.g, the countable set of all integers, or all the uncountable set of all reals), we need to generalize some of the earlier

definitions. Since the details are rather technical, we just briefly state the main results without proof. See e.g., (Grimmett and Stirzaker 1992) for details.

For a stationary distribution to exist, we require irreducibility (singly connected) and aperiodicity, as before. But we also require that each state is **recurrent**. (A chain in which all states are recurrent is called a recurrent chain.) Recurrent means that you will return to that state with probability 1. As a simple example of a non-recurrent state (i.e., a **transient** state), consider Figure 17.4(b): states 3 is transient because one immediately leaves it and either spins around state 4 forever, or oscillates between states 1 and 2 forever. There is no way to return to state 3.

It is clear that any finite-state irreducible chain is recurrent, since you can always get back to where you started from. But now consider an example with an infinite state space. Suppose we perform a random walk on the integers, $\mathcal{X} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$. Let $A_{i,i+1} = p$ be the probability of moving right, and $A_{i,i-1} = 1 - p$ be the probability of moving left. Suppose we start at $X_1 = 0$. If $p > 0.5$, we will shoot off to $+\infty$; we are not guaranteed to return. Similarly, if $p < 0.5$, we will shoot off to $-\infty$. So in both cases, the chain is not recurrent, even though it is irreducible.

It should be intuitively obvious that we require all states to be recurrent for a stationary distribution to exist. However, this is not sufficient. To see this, consider the random walk on the integers again, and suppose $p = 0.5$. In this case, we can return to the origin an infinite number of times, so the chain is recurrent. However, it takes infinitely long to do so. This prohibits it from having a stationary distribution. The intuitive reason is that the distribution keeps spreading out over a larger and larger set of the integers, and never converges to a stationary distribution. More formally, we define a state to be **non-null recurrent** if the expected time to return to this state is finite. A chain in which all states are non-null is called a non-null chain.

For brevity, we we say that a state is **ergodic** if it is aperiodic, recurrent and non-null, and we say a chain is ergodic if all its states are ergodic.

We can now state our main theorem:

Theorem 17.2.2. *Every irreducible (singly connected), ergodic Markov chain has a limiting distribution, which is equal to $\boldsymbol{\pi}$, its unique stationary distribution.*

This generalizes Theorem 17.2.1, since for irreducible finite-state chains, all states are recurrent and non-null.

17.2.3.4 Detailed balance

Establishing ergodicity can be difficult. We now give an alternative condition that is easier to verify.

We say that a Markov chain $\mathbf{A}$ is **time reversible** if there exists a distribution $\boldsymbol{\pi}$ such that

$$\pi_i A_{ij} = \pi_j A_{ji} \tag{17.29}$$

These are called the **detailed balance equations**. This says that the flow from i to j must equal the flow from j to i, weighted by the appropriate source probabilities.

We have the following important result.

Theorem 17.2.3. *If a Markov chain with transition matrix $\mathbf{A}$ is regular and satisfies detailed balance wrt distribution $\boldsymbol{\pi}$, then $\boldsymbol{\pi}$ is a stationary distribution of the chain.*

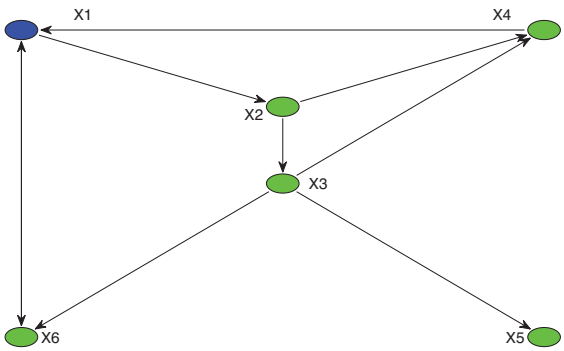


Figure 17.5 A very small world wide web. Figure generated by `pagerankDemo`, written by Tim Davis.

Proof. To see this, note that

$$\sum_i \pi_i A_{ij} = \sum_i \pi_j A_{ji} = \pi_j \sum_i A_{ji} = \pi_j \tag{17.30}$$

and hence $\boldsymbol{\pi} = \mathbf{A}\boldsymbol{\pi}$. □

Note that this condition is sufficient but not necessary (see Figure 17.4(a) for an example of a chain with a stationary distribution which does not satisfy detailed balance).

In Section 24.1, we will discuss Markov chain Monte Carlo or MCMC methods. These take as input a desired distribution $\boldsymbol{\pi}$ and construct a transition matrix (or in general, a transition **kernel**) $\mathbf{A}$ which satisfies detailed balance wrt $\boldsymbol{\pi}$. Thus by sampling states from such a chain, we will eventually enter the stationary distribution, and will visit states with probabilities given by $\boldsymbol{\pi}$.

17.2.4 Application: Google's PageRank algorithm for web page ranking *

The results in Section 17.2.3 form the theoretical underpinnings to Google's **PageRank** algorithm, which is used for information retrieval on the world-wide web. We sketch the basic idea below; see (Byran and Leise 2006) for a more detailed explanation.

We will treat the web as a giant directed graph, where nodes represent web pages (documents) and edges represent hyper-links.[5] We then perform a process called **web crawling**. We start at a few designated root nodes, such as `dmoz.org`, the home of the Open Directory Project, and then follows the links, storing all the pages that we encounter, until we run out of time.

Next, all of the words in each web page are entered into a data structure called an **inverted index**. That is, for each word, we store a list of the documents where this word occurs. (In practice, we store a list of hash codes representing the URLs.) At test time, when a user enters

5. In 2008, Google said it had indexed 1 trillion (10^{12}) unique URLs. If we assume there are about 10 URLs per page (on average), this means there were about 100 billion unique web pages. Estimates for 2010 are about 121 billion unique web pages. Source: `thenextweb.com/shareables/2011/01/11/infographic-how-big-is-the-internet`.

a query, we can just look up all the documents containing each word, and intersect these lists (since queries are defined by a conjunction of search terms). We can get a refined search by storing the location of each word in each document. We can then test if the words in a document occur in the same order as in the query.

Let us give an example, from `http://en.wikipedia.org/wiki/Inverted_index`. We have 3 documents, T_0 = "it is what it is", T_1 = "what is it" and T_2 = "it is a banana". Then we can create the following inverted index, where each pair represents a document and word location:

```
"a":      {(2, 2)}
"banana": {(2, 3)}
"is":     {(0, 1), (0, 4), (1, 1), (2, 1)}
"it":     {(0, 0), (0, 3), (1, 2), (2, 0)}
"what":   {(0, 2), (1, 0)}
```

For example, we see that the word "what" occurs at location 2 (counting from 0) in document 0, and location 0 in document 1. Suppose we search for "what is it". If we ignore word order, we retrieve the following documents:

$$\{T_0, T_1\} \cap \{T_0, T_1, T_2\} \cap \{T_0, T_1, T_2\} = \{T_0, T_1\} \tag{17.31}$$

If we require that the word order matches, only document T_1 would be returned. More generally, we can allow out-of-order matches, but can give "bonus points" to documents whose word order matches the query's word order, or to other features, such as if the words occur in the title of a document. We can then return the matching documents in decreasing order of their score/relevance. This is called document **ranking**.

So far, we have described the standard process of information retrieval. But the link structure of the web provides an additional source of information. The basic idea is that some web pages are more authoritative than others, so these should be ranked higher (assuming they match the query). A web page is an authority if it is linked to by many other pages. But to protect against the effect of so-called **link farms**, which are dummy pages which just link to a given site to boost its apparent relevance, we will weight each incoming link by the source's authority. Thus we get the following recursive definition for the authoritativeness of page j, also called its **PageRank**:

$$\pi_j = \sum_i A_{ij} \pi_i \tag{17.32}$$

where A_{ij} is the probability of following a link from i to j. We recognize Equation 17.32 as the stationary distribution of a Markov chain.

In the simplest setting, we define $A_{i.}$ as a uniform distribution over all states that i is connected to. However, to ensure the distribution is unique, we need to make the chain into a regular chain. This can be done by allowing each state i to jump to any other state (including itself) with some small probability. This effectively makes the transition matrix aperiodic and fully connected (although the adjacency matrix G_{ij} of the web itself is highly sparse).

We discuss efficient methods for computing the leading eigenvector of this giant matrix below. But first, let us give an example of the PageRank algorithm. Consider the small web in Figure 17.5.

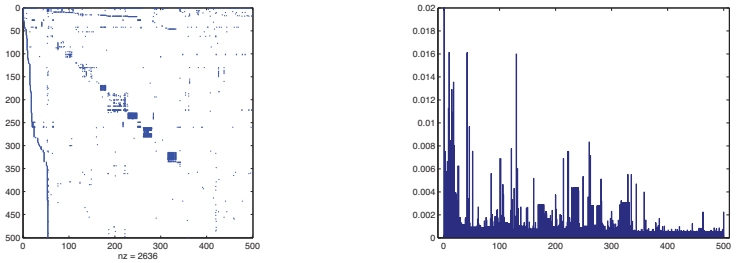


Figure 17.6 (a) Web graph of 500 sites rooted at `www.harvard.edu`. (b) Corresponding page rank vector. Figure generated by `pagerankDemoPmtk`, Based on code by Cleve Moler (Moler 2004).

We find that the stationary distribution is

$$\boldsymbol{\pi} = (0.3209, 0.1706, 0.1065, 0.1368, 0.0643, 0.2008) \tag{17.33}$$

So a random surfer will visit site 1 about 32% of the time. We see that node 1 has a higher PageRank than nodes 4 or 6, even though they all have the same number of in-links. This is because being linked to from an influential nodehelps increase your PageRank score more than being linked to by a less influential node.

As a slightly larger example, Figure 17.6(a) shows a web graph, derived from the root of `harvard.edu`. Figure 17.6(b) shows the corresponding PageRank vector.

17.2.4.1 Efficiently computing the PageRank vector

Let $G_{ij} = 1$ iff there is a link from j to i. Now imagine performing a random walk on this graph, where at every time step, with probability $p = 0.85$ you follow one of the outlinks uniformly at random, and with probability $1 - p$ you jump to a random node, again chosen uniformly at random. If there are no outlinks, you just jump to a random page. (These random jumps, including self-transitions, ensure the chain is irreducible (singly connected) and regular. Hence we can solve for its unique stationary distribution using eigenvector methods.) This defines the following transition matrix:

$$M_{ij} = \begin{cases} pG_{ij}/c_j + \delta & \text{if } c_j \neq 0 \\ 1/n & \text{if } c_j = 0 \end{cases} \tag{17.34}$$

where n is the number of nodes, $\delta = (1-p)/n$ is the probability of jumping from one page to another without following a link and $c_j = \sum_i G_{ij}$ represents the out-degree of page j. (If $n = 4 \cdot 10^9$ and $p = 0.85$, then $\delta = 3.75 \cdot 10^{-11}$.) Here $\mathbf{M}$ is a stochastic matrix in which *columns* sum to one. Note that $\mathbf{M} = \mathbf{A}^T$ in our earlier notation.

We can represent the transition matrix compactly as follows. Define the diagonal matrix $\mathbf{D}$ with entries

$$d_{jj} = \begin{cases} 1/c_j & \text{if } c_j \neq 0 \\ 0 & \text{if } c_j = 0 \end{cases} \tag{17.35}$$

Define the vector $\mathbf{z}$ with components

$$z_j = \begin{cases} \delta & \text{if } c_j \neq 0 \\ 1/n & \text{if } c_j = 0 \end{cases} \tag{17.36}$$

Then we can rewrite Equation 17.34 as follows:

$$\mathbf{M} = p\mathbf{G}\mathbf{D} + \mathbf{1}\mathbf{z}^T \tag{17.37}$$

The matrix $\mathbf{M}$ is not sparse, but it is a rank one modification of a sparse matrix. Most of the elements of $\mathbf{M}$ are equal to the small constant δ. Obviously these do not need to be stored explicitly.

Our goal is to solve $\mathbf{v} = \mathbf{M}\mathbf{v}$, where $\mathbf{v} = \boldsymbol{\pi}^T$. One efficient method to find the leading eigenvector of a large matrix is known as the **power method**. This simply consists of repeated matrix-vector multiplication, followed by normalization:

$$\mathbf{v} \propto \mathbf{M}\mathbf{v} = p\mathbf{G}\mathbf{D}\mathbf{v} + \mathbf{1}\mathbf{z}^T\mathbf{v} \tag{17.38}$$

It is possible to implement the power method without using any matrix multiplications, by simply sampling from the transition matrix and counting how often you visit each state. This is essentially a Monte Carlo approximation to the sum implied by $\mathbf{v} = \mathbf{M}\mathbf{v}$. Applying this to the data in Figure 17.6(a) yields the stationary distribution in Figure 17.6(b). This took 13 iterations to converge, starting from a uniform distribution. (See also the function `pagerankDemo`, by Tim Davis, for an animation of the algorithm in action, applied to the small web example.) To handle changing web structure, we can re-run this algorithm every day or every week, starting $\mathbf{v}$ off at the old distribution (Langville and Meyer 2006).

For details on how to perform this Monte Carlo power method in a parallel distributed computing environment, see e.g., (Rajaraman and Ullman 2010).

17.2.4.2 Web spam

PageRank is not foolproof. For example, consider the strategy adopted by JC Penney, a department store in the USA. During the Christmas season of 2010, it planted many links to its home page on 1000s of irrelevant web pages, thus increasing its ranking on Google's search engine (Segal 2011). Even though each of these source pages has low PageRank, there were so many of them that their effect added up. Businesses call this **search engine optimization**; Google calls it **web spam**. When Google was notified of this scam (by the *New York Times*), it manually downweighted JC Penney, since such behavior violates Google's code of conduct. The result was that JC Penney dropped from rank 1 to rank 65, essentially making it disappear from view. Automatically detecting such scams relies on various techniques which are beyond the scope of this chapter.

17.3 Hidden Markov models

As we mentioned in Section 10.2.2, a **hidden Markov model** or **HMM** consists of a discrete-time, discrete-state Markov chain, with hidden states $z_t \in \{1, \ldots, K\}$, plus an **observation** model

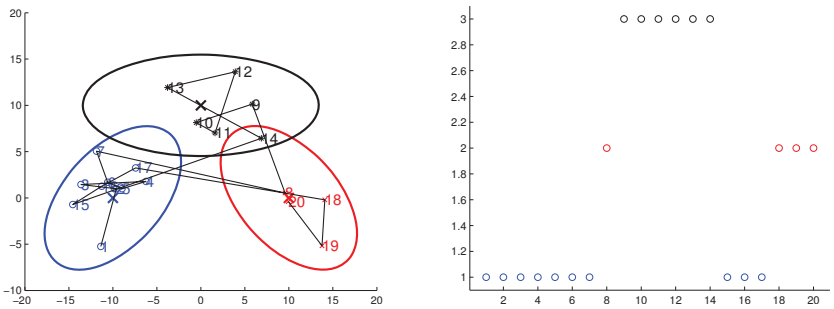


Figure 17.7 (a) Some 2d data sampled from a 3 state HMM. Each state emits from a 2d Gaussian. (b) The hidden state sequence. Based on Figure 13.8 of (Bishop 2006b). Figure generated by `hmmLillypadDemo`.

$p(\mathbf{x}_t|z_t)$. The corresponding joint distribution has the form

$$p(\mathbf{z}_{1:T}, \mathbf{x}_{1:T}) = p(\mathbf{z}_{1:T})p(\mathbf{x}_{1:T}|\mathbf{z}_{1:T}) = \left[p(z_1)\prod_{t=2}^{T} p(z_t|z_{t-1})\right]\left[\prod_{t=1}^{T} p(\mathbf{x}_t|z_t)\right] \tag{17.39}$$

The observations in an HMM can be discrete or continuous. If they are discrete, it is common for the observation model to be an observation matrix:

$$p(\mathbf{x}_t = l|z_t = k, \boldsymbol{\theta}) = B(k, l) \tag{17.40}$$

If the observations are continuous, it is common for the observation model to be a conditional Gaussian:

$$p(\mathbf{x}_t|z_t = k, \boldsymbol{\theta}) = \mathcal{N}(\mathbf{x}_t|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \tag{17.41}$$

Figure 17.7 shows an example where we have 3 states, each of which emits a different Gaussian. The resulting model is similar to a Gaussian mixture model, except the cluster membership has Markovian dynamics. (Indeed, HMMs are sometimes called **Markov switching models** (Fruhwirth-Schnatter 2007).) We see that we tend to get multiple observations in the same location, and then a sudden jump to a new cluster.

17.3.1 Applications of HMMs

HMMs can be used as black-box density models on sequences. They have the advantage over Markov models in that they can represent long-range dependencies between observations, mediated via the latent variables. In particular, note that they do not assume the Markov property holds for the observations themselves. Such black-box models are useful for time-series prediction (Fraser 2008). They can also be used to define class-conditional densities inside a generative classifier.

However, it is more common to imbue the hidden states with some desired meaning, and to then try to estimate the hidden states from the observations, i.e., to compute $p(z_t|\mathbf{x}_{1:t})$ if we are

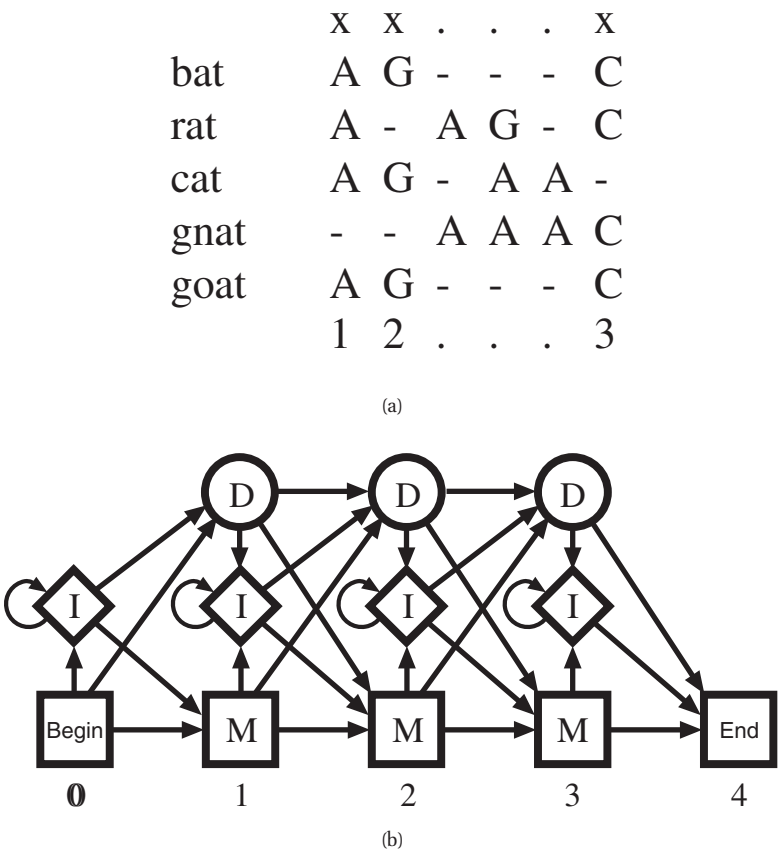


Figure 17.8 (a) Some DNA sequences. (b) State transition diagram for a profile HMM. Source: Figure 5.7 of (Durbin et al. 1998). Used with kind permission of Richard Durbin.

in an online scenario, or $p(z_t|\mathbf{x}_{1:T})$ if we are in an offline scenario (see Section 17.4.1 for further discussion of the differences between these two approaches). Below we give some examples of applications which use HMMs in this way:

- **Automatic speech recognition**. Here $\mathbf{x}_t$ represents features extracted from the speech signal, and z_t represents the word that is being spoken. The transition model $p(z_t|z_{t-1})$ represents the language model, and the observation model $p(\mathbf{x}_t|z_t)$ represents the acoustic model. See e.g., (Jelinek 1997; Jurafsky and Martin 2008) for details.
- **Activity recognition**. Here $\mathbf{x}_t$ represents features extracted from a video frame, and z_t is the class of activity the person is engaged in (e.g., running, walking, sitting, etc.) See e.g., (Szeliski 2010) for details.
- **Part of speech tagging**. Here x_t represents a word, and z_t represents its **part of speech** (noun, verb, adjective, etc.) See Section 19.6.2.1 for more information on POS tagging and

related tasks.

- **Gene finding**. Here x_t represents the DNA nucleotides (A,C,G,T), and z_t represents whether we are inside a gene-coding region or not. See e.g., (Schweikerta et al. 2009) for details.
- **Protein sequence alignment**. Here x_t represents an amino acid, and z_t represents whether this matches the latent **consensus sequence** at this location. This model is called a **profile HMM** and is illustrated in Figure 17.8. The HMM has 3 states, called match, insert and delete. If z_t is a match state, then x_t is equal to the t'th value of the consensus. If z_t is an insert state, then x_t is generated from a uniform distribution that is unrelated to the consensus sequence. If z_t is a delete state, then $x_t = -$. In this way, we can generate noisy copies of the consensus sequence of different lengths. In Figure 17.8(a), the consensus is "AGC", and we see various versions of this below. A path through the state transition diagram, shown in Figure 17.8(b), specifies how to align a sequence to the consensus, e.g., for the gnat, the most probable path is D, D, I, I, I, M. This means we delete the A and G parts of the consensus sequence, we insert 3 A's, and then we match the final C. We can estimate the model parameters by counting the number of such transitions, and the number of emissions from each kind of state, as shown in Figure 17.8(c). See Section 17.5 for more information on training an HMM, and (Durbin et al. 1998) for details on profile HMMs.

Note that for some of these tasks, conditional random fields, which are essentially discriminative versions of HMMs, may be more suitable; see Chapter 19 for details.

17.4 Inference in HMMs

We now discuss how to infer the hidden state sequence of an HMM, assuming the parameters are known. Exactly the same algorithms apply to other chain-structured graphical models, such as chain CRFs (see Section 19.6.1). In Chapter 20, we generalize these methods to arbitrary graphs. And in Section 17.5.2, we show how we can use the output of inference in the context of parameter estimation.

17.4.1 Types of inference problems for temporal models

There are several different kinds of inferential tasks for an HMM (and SSM in general). To illustrate the differences, we will consider an example called the **occasionally dishonest casino**, from (Durbin et al. 1998). In this model, $x_t \in \{1, 2, \ldots, 6\}$ represents which dice face shows up, and z_t represents the identity of the dice that is being used. Most of the time the casino uses a fair dice, $z = 1$, but occasionally it switches to a loaded dice, $z = 2$, for a short period. If $z = 1$ the observation distribution is a uniform multinoulli over the symbols $\{1, \ldots, 6\}$. If $z = 2$, the observation distribution is skewed towards face 6 (see Figure 17.9). If we sample from this model, we may observe data such as the following:

Listing 17.1 Example output of `casinoDemo`

```
Rolls:    664153216162115234653214356634261655234232315142464156663246
Die:      LLLLLLLLLLLLLLFFFFFFLLLLLLLLLLLLLLFFFFFFFFFFFFFFFFFFLLLLLLL
```

Here "rolls" refers to the observed symbol and "die" refers to the hidden state (L is loaded and F is fair). Thus we see that the model generates a sequence of symbols, but the statistics of the

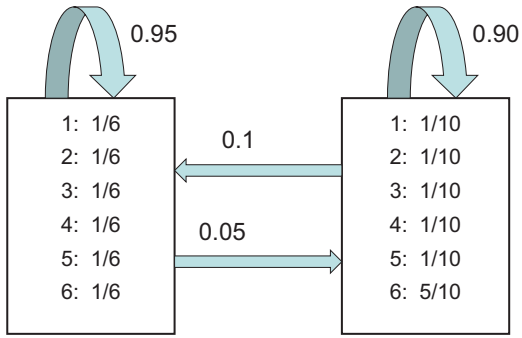
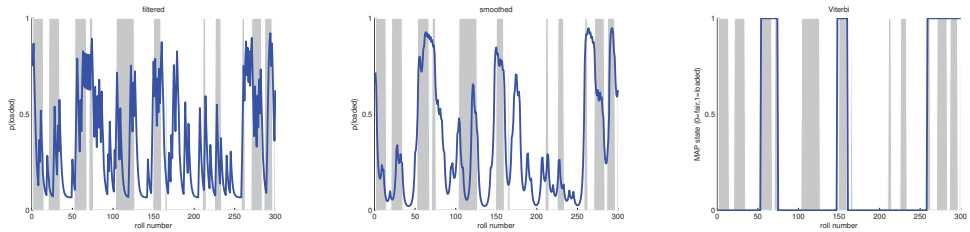


Figure 17.9 An HMM for the occasionally dishonest casino. The blue arrows visualize the state transition diagram $\mathbf{A}$. Based on (Durbin et al. 1998, p54).



(a) (b) (c)

Figure 17.10 Inference in the dishonest casino. Vertical gray bars denote the samples that we generated using a loaded die. (a) Filtered estimate of probability of using a loaded dice. (b) Smoothed estimates. (c) MAP trajectory. Figure generated by `casinoDemo`.

distribution changes abruptly every now and then. In a typical application, we just see the rolls and want to infer which dice is being used. But there are different kinds of inference, which we summarize below.

- **Filtering** means to compute the **belief state** $p(z_t|\mathbf{x}_{1:t})$ online, or recursively, as the data streams in. This is called "filtering" because it reduces the noise more than simply estimating the hidden state using just the current estimate, $p(z_t|\mathbf{x}_t)$. We will see below that we can perform filtering by simply applying Bayes rule in a sequential fashion. See Figure 17.10(a) for an example.
- **Smoothing** means to compute $p(z_t|\mathbf{x}_{1:T})$ offline, given all the evidence. See Figure 17.10(b) for an example. By conditioning on past and future data, our uncertainty will be significantly reduced. To understand this intuitively, consider a detective trying to figure out who committed a crime. As he moves through the crime scene, his uncertainty is high until he finds the key clue; then he has an "aha" moment, his uncertainty is reduced, and all the previously confusing observations are, in **hindsight**, easy to explain.

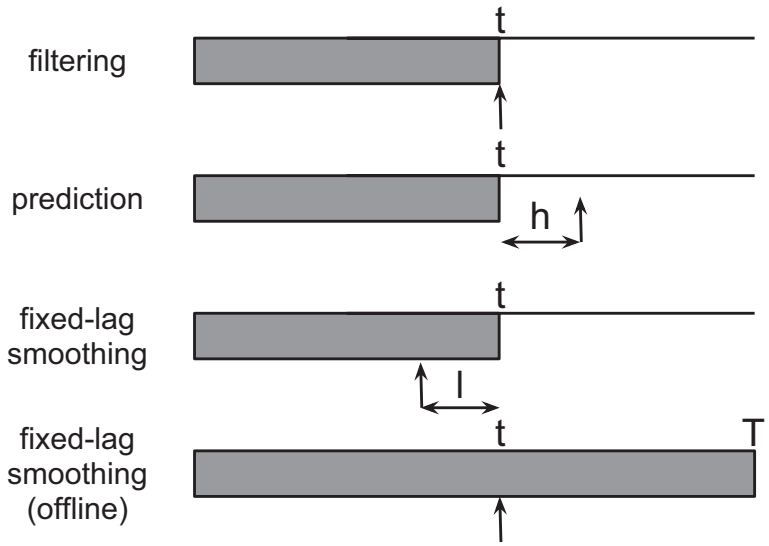


Figure 17.11 The main kinds of inference for state-space models. The shaded region is the interval for which we have data. The arrow represents the time step at which we want to perform inference. t is the current time, T is the sequence length, ℓ is the lag and h is the prediction horizon. See text for details.

- **Fixed lag smoothing** is an interesting compromise between online and offline estimation; it involves computing $p(z_{t-\ell}|\mathbf{x}_{1:t})$, where $\ell > 0$ is called the lag. This gives better performance than filtering, but incurs a slight delay. By changing the size of the lag, one can trade off accuracy vs delay.
- **Prediction** Instead of predicting the past given the future, as in fixed lag smoothing, we might want to predict the future given the past, i.e., to compute $p(z_{t+h}|\mathbf{x}_{1:t})$, where $h > 0$ is called the prediction **horizon**. For example, suppose $h = 2$; then we have

$$p(z_{t+2}|\mathbf{x}_{1:t}) = \sum_{z_{t+1}} \sum_{z_t} p(z_{t+2}|z_{t+1}) p(z_{t+1}|z_t) p(z_t|\mathbf{x}_{1:t}) \tag{17.42}$$

  It is straightforward to perform this computation: we just power up the transition matrix and apply it to the current belief state. The quantity $p(z_{t+h}|\mathbf{x}_{1:t})$ is a prediction about future hidden states; it can be converted into a prediction about future observations using

$$p(\mathbf{x}_{t+h}|\mathbf{x}_{1:t}) = \sum_{z_{t+h}} p(\mathbf{x}_{t+h}|z_{t+h}) p(z_{t+h}|\mathbf{x}_{1:t}) \tag{17.43}$$

  This is the posterior predictive density, and can be used for time-series forecasting (see (Fraser 2008) for details). See Figure 17.11 for a sketch of the relationship between filtering, smoothing, and prediction.
- **MAP estimation** This means computing $\arg\max_{\mathbf{z}_{1:T}} p(\mathbf{z}_{1:T}|\mathbf{x}_{1:T})$, which is a most probable state sequence. In the context of HMMs, this is known as **Viterbi decoding** (see

Section 17.4.4). Figure 17.10 illustrates the difference between filtering, smoothing and MAP decoding for the occasionally dishonest casino HMM. We see that the smoothed (offline) estimate is indeed smoother than the filtered (online) estimate. If we threshold the estimates at 0.5 and compare to the true sequence, we find that the filtered method makes 71 errors out of 300, and the smoothed method makes 49/300; the MAP path makes 60/300 errors. It is not surprising that smoothing makes fewer errors than Viterbi, since the optimal way to minimize bit-error rate is to threshold the posterior marginals (see Section 5.7.1.1). Nevertheless, for some applications, we may prefer the Viterbi decoding, as we discuss in Section 17.4.4.

- **Posterior samples** If there is more than one plausible interpretation of the data, it can be useful to sample from the posterior, $\mathbf{z}_{1:T} \sim p(\mathbf{z}_{1:T}|\mathbf{x}_{1:T})$. These sample paths contain much more information than the sequence of marginals computed by smoothing.
- **Probability of the evidence** We can compute the **probability of the evidence**, $p(\mathbf{x}_{1:T})$, by summing up over all hidden paths, $p(\mathbf{x}_{1:T}) = \sum_{\mathbf{z}_{1:T}} p(\mathbf{z}_{1:T}, \mathbf{x}_{1:T})$. This can be used to classify sequences (e.g., if the HMM is used as a class conditional density), for model-based clustering, for anomaly detection, etc.

17.4.2 The forwards algorithm

We now describe how to recursively compute the filtered marginals, $p(z_t|\mathbf{x}_{1:t})$ in an HMM.

The algorithm has two steps. First comes the prediction step, in which we compute the **one-step-ahead predictive density**; this acts as the new prior for time t:

$$p(z_t = j|\mathbf{x}_{1:t-1}) = \sum_i p(z_t = j|z_{t-1} = i)p(z_{t-1} = i|\mathbf{x}_{1:t-1}) \tag{17.44}$$

Next comes the update step, in which we absorb the observed data from time t using Bayes rule:

$$\alpha_t(j) \triangleq p(z_t = j|\mathbf{x}_{1:t}) = p(z_t = j|\mathbf{x}_t, \mathbf{x}_{1:t-1}) \tag{17.45}$$

$$= \frac{1}{Z_t} p(\mathbf{x}_t|z_t = j, \cancel{\mathbf{x}_{1:t-1}}) p(z_t = j|\mathbf{x}_{1:t-1}) \tag{17.46}$$

where the normalization constant is given by

$$Z_t \triangleq p(\mathbf{x}_t|\mathbf{x}_{1:t-1}) = \sum_j p(z_t = j|\mathbf{x}_{1:t-1}) p(\mathbf{x}_t|z_t = j) \tag{17.47}$$

This process is known as the **predict-update cycle**. The distribution $p(z_t|\mathbf{x}_{1:t})$ is called the (filtered) **belief state** at time t, and is a vector of K numbers, often denoted by $\boldsymbol{\alpha}_t$. In matrix-vector notation, we can write the update in the following simple form:

$$\boldsymbol{\alpha}_t \propto \boldsymbol{\psi}_t \odot (\boldsymbol{\Psi}^T \boldsymbol{\alpha}_{t-1}) \tag{17.48}$$

where $\psi_t(j) = p(\mathbf{x}_t|z_t = j)$ is the local evidence at time t, $\Psi(i, j) = p(z_t = j|z_{t-1} = i)$ is the transition matrix, and $\mathbf{u} \odot \mathbf{v}$ is the **Hadamard product**, representing elementwise vector multiplication. See Algorithm 6 for the pseudo-code, and `hmmFilter` for some Matlab code.

In addition to computing the hidden states, we can use this algorithm to compute the log probability of the evidence:

$$\log p(\mathbf{x}_{1:T}|\boldsymbol{\theta}) = \sum_{t=1}^{T} \log p(\mathbf{x}_t|\mathbf{x}_{1:t-1}) = \sum_{t=1}^{T} \log Z_t \tag{17.49}$$

(We need to work in the log domain to avoid numerical underflow.)

Algorithm 17.1: Forwards algorithm

Input: Transition matrices $\psi(i,j) = p(z_t = j|z_{t-1} = i)$, local evidence vectors $\psi_t(j) = p(\mathbf{x}_t|z_t = j)$, initial state distribution $\pi(j) = p(z_1 = j)$;
$[\alpha_1, Z_1] = \text{normalize}(\boldsymbol{\psi}_1 \odot \boldsymbol{\pi})$;
for $t = 2 : T$ **do**
  $[\alpha_t, Z_t] = \text{normalize}(\boldsymbol{\psi}_t \odot (\boldsymbol{\Psi}^T \boldsymbol{\alpha}_{t-1}))$;
Return $\boldsymbol{\alpha}_{1:T}$ and $\log p(\mathbf{y}_{1:T}) = \sum_t \log Z_t$;
Subroutine: $[\mathbf{v}, Z] = \text{normalize}(\mathbf{u}) : Z = \sum_j u_j; \quad v_j = u_j/Z$;

17.4.3 The forwards-backwards algorithm

In Section 17.4.2, we explained how to compute the filtered marginals $p(z_t = j|\mathbf{x}_{1:t})$ using online inference. We now discuss how to compute the smoothed marginals, $p(z_t = j|\mathbf{x}_{1:T})$, using offline inference.

17.4.3.1 Basic idea

The key decomposition relies on the fact that we can break the chain into two parts, the past and the future, by conditioning on z_t:

$$p(z_t = j|\mathbf{x}_{1:T}) \propto p(z_t = j, \mathbf{x}_{t+1:T}|\mathbf{x}_{1:t}) \propto p(z_t = j|\mathbf{x}_{1:t}) p(\mathbf{x}_{t+1:T}|z_t = j, \cancel{\mathbf{x}_{1:t}}) \tag{17.50}$$

Let $\alpha_t(j) \triangleq p(z_t = j|\mathbf{x}_{1:t})$ be the filtered belief state as before. Also, define

$$\beta_t(j) \triangleq p(\mathbf{x}_{t+1:T}|z_t = j) \tag{17.51}$$

as the conditional likelihood of future evidence given that the hidden state at time t is j. (Note that this is not a probability distribution over states, since it does not need to satisfy $\sum_j \beta_t(j) = 1$.) Finally, define

$$\gamma_t(j) \triangleq p(z_t = j|\mathbf{x}_{1:T}) \tag{17.52}$$

as the desired smoothed posterior marginal. From Equation 17.50, we have

$$\gamma_t(j) \propto \alpha_t(j)\beta_t(j) \tag{17.53}$$

We have already described how to recursively compute the α's in a left-to-right fashion in Section 17.4.2. We now describe how to recursively compute the β's in a right-to-left fashion. If we have already computed β_t, we can compute β_{t-1} as follows:

$$\begin{aligned}
\beta_{t-1}(i) &= p(\mathbf{x}_{t:T}|z_{t-1}=i) && (17.54)\\
&= \sum_j p(z_t=j, \mathbf{x}_t, \mathbf{x}_{t+1:T}|z_{t-1}=i) && (17.55)\\
&= \sum_j p(\mathbf{x}_{t+1:T}|z_t=j, \cancel{z_{t-1}=i}, \cancel{\mathbf{x}_t}) p(z_t=j, \mathbf{x}_t|z_{t-1}=i) && (17.56)\\
&= \sum_j p(\mathbf{x}_{t+1:T}|z_t=j) p(\mathbf{x}_t|z_t=j, \cancel{z_{t-1}=i}) p(z_t=j|z_{t-1}=i) && (17.57)\\
&= \sum_j \beta_t(j)\psi_t(j)\psi(i,j) && (17.58)
\end{aligned}$$

We can write the resulting equation in matrix-vector form as

$$\boldsymbol{\beta}_{t-1} = \boldsymbol{\Psi}(\boldsymbol{\psi}_t \odot \boldsymbol{\beta}_t) \tag{17.59}$$

The base case is

$$\beta_T(i) = p(\mathbf{x}_{T+1:T}|z_T=i) = p(\emptyset|z_T=i) = 1 \tag{17.60}$$

which is the probability of a non-event.

Having computed the forwards and backwards messages, we can combine them to compute $\gamma_t(j) \propto \alpha_t(j)\beta_t(j)$. The overall algorithm is known as the **forwards-backwards algorithm**. The pseudo code is very similar to the forwards case; see `hmmFwdBack` for an implementation.

We can think of this algorithm as passing "messages" from left to right, and then from right to left, and then combining them at each node. We will generalize this intuition in Section 20.2, when we discuss belief propagation.

17.4.3.2 Two-slice smoothed marginals

When we estimate the parameters of the transition matrix using EM (see Section 17.5), we will need to compute the expected number of transitions from state i to state j:

$$N_{ij} = \sum_{t=1}^{T-1} \mathbb{E}\left[\mathbb{I}(z_t=i, z_{t+1}=j)|\mathbf{x}_{1:T}\right] = \sum_{t=1}^{T-1} p(z_t=i, z_{t+1}=j|\mathbf{x}_{1:T}) \tag{17.61}$$

The term $p(z_t=i, z_{t+1}=j|\mathbf{x}_{1:T})$ is called a (smoothed) **two-slice marginal**, and can be computed as follows

$$\begin{aligned}
\xi_{t,t+1}(i,j) &\triangleq p(z_t=i, z_{t+1}=j|\mathbf{x}_{1:T}) && (17.62)\\
&\propto p(z_t|\mathbf{x}_{1:t}) p(z_{t+1}|z_t, \mathbf{x}_{t+1:T}) && (17.63)\\
&\propto p(z_t|\mathbf{x}_{1:t}) p(\mathbf{x}_{t+1:T}|z_t, z_{t+1}) p(z_{t+1}|z_t) && (17.64)\\
&\propto p(z_t|\mathbf{x}_{1:t}) p(\mathbf{x}_{t+1}|z_{t+1}) p(\mathbf{x}_{t+2:T}|z_{t+1}) p(z_{t+1}|z_t) && (17.65)\\
&= \alpha_t(i)\phi_{t+1}(j)\beta_{t+1}(j)\psi(i,j) && (17.66)
\end{aligned}$$

In matrix-vector form, we have

$$\boldsymbol{\xi}_{t,t+1} \propto \boldsymbol{\Psi} \odot (\boldsymbol{\alpha}_t (\boldsymbol{\phi}_{t+1} \odot \boldsymbol{\beta}_{t+1})^T) \tag{17.67}$$

For another interpretation of these equations, see Section 20.2.4.3.

17.4.3.3 Time and space complexity

It is clear that a straightforward implementation of FB takes $O(K^2T)$ time, since we must perform a $K \times K$ matrix multiplication at each step. For some applications, such as speech recognition, K is very large, so the $O(K^2)$ term becomes prohibitive. Fortunately, if the transition matrix is sparse, we can reduce this substantially. For example, in a left-to-right transition matrix, the algorithm takes $O(TK)$ time.

In some cases, we can exploit special properties of the state space, even if the transition matrix is not sparse. In particular, suppose the states represent a discretization of an underlying continuous state-space, and the transition matrix has the form $\psi(i,j) \propto \exp(-\sigma^2|\mathbf{z}_i - \mathbf{z}_j|)$, where $\mathbf{z}_i$ is the continuous vector represented by state i. Then one can implement the forwards-backwards algorithm in $O(TK \log K)$ time. This is very useful for models with large state spaces. See Section 22.2.6.1 for details.

In some cases, the bottleneck is memory, not time. The expected sufficient statistics needed by EM are $\sum_t \xi_{t-1,t}(i,j)$; this takes constant space (independent of T); however, to compute them, we need $O(KT)$ working space, since we must store α_t for $t = 1, \ldots, T$ until we do the backwards pass. It is possible to devise a simple divide-and-conquer algorithm that reduces the space complexity from $O(KT)$ to $O(K \log T)$ at the cost of increasing the running time from $O(K^2T)$ to $O(K^2T \log T)$: see (Binder et al. 1997; Zweig and Padmanabhan 2000) for details.

17.4.4 The Viterbi algorithm

The **Viterbi** algorithm (Viterbi 1967) can be used to compute the most probable sequence of states in a chain-structured graphical model, i.e., it can compute

$$\mathbf{z}^* = \arg\max_{\mathbf{z}_{1:T}} p(\mathbf{z}_{1:T}|\mathbf{x}_{1:T}) \tag{17.68}$$

This is equivalent to computing a shortest path through the **trellis diagram** in Figure 17.12, where the nodes are possible states at each time step, and the node and edge weights are log probabilities. That is, the weight of a path $z_1, z_2, \ldots, z_T$ is given by

$$\log \pi_1(z_1) + \log \phi_1(z_1) + \sum_{t=2}^{T} [\log \psi(z_{t-1}, z_t) + \log \phi_t(z_t)] \tag{17.69}$$

17.4.4.1 MAP vs MPE

Before discussing how the algorithm works, let us make one important remark: the *(jointly) most probable sequence of states is not necessarily the same as the sequence of (marginally) most probable states*. The former is given by Equation 17.68, and is what Viterbi computes, whereas the latter is given by the maximizer of the posterior marginals or **MPM**:

$$\hat{\mathbf{z}} = (\arg\max_{z_1} p(z_1|\mathbf{x}_{1:T}), \ldots, \arg\max_{z_T} p(z_T|\mathbf{x}_{1:T})) \tag{17.70}$$

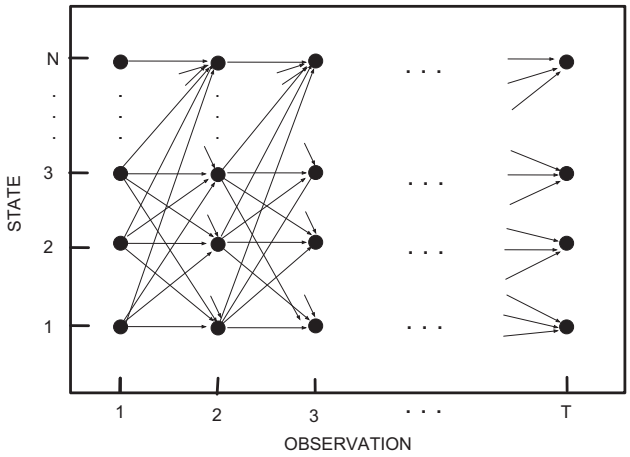


Figure 17.12 The trellis of states vs time for a Markov chain. Based on (Rabiner 1989).

As a simple example of the difference, consider a chain with two time steps, defining the following joint:

	$X_1 = 0$	$X_1 = 1$	
$X_2 = 0$	0.04	0.3	0.34
$X_2 = 1$	0.36	0.3	0.66
	0.4	0.6	

The joint MAP estimate is $(0, 1)$, whereas the sequence of marginal MPMs is $(1, 1)$.

The advantage of the joint MAP estimate is that is is always globally consistent. For example, suppose we are performing speech recognition and someones says "recognize speech". This could be mis-heard as "wreck a nice beach". Locally it may appear that "beach" is the most probable interpretation of that particular window of sound, but when we add the requirement that the data be explained by a single linguistically plausible path, this interpretation becomes less likely.

On the other hand, the MPM estimates can be more robust (Marroquin et al. 1987). To see why, note that in Viterbi, when we estimate z_t, we "max out" the other variables:

$$z_t^* = \arg\max_{z_t} \max_{\mathbf{z}_{1:t-1}, \mathbf{z}_{t+1:T}} p(\mathbf{z}_{1:t-1}, z_t, \mathbf{z}_{t+1:T} | \mathbf{x}_{1:T}) \tag{17.71}$$

whereas we when we use forwards-backwards, we sum out the other variables:

$$p(z_t | \mathbf{x}_{1:T}) = \sum_{\mathbf{z}_{1:t-1}, \mathbf{z}_{t+1:T}} p(\mathbf{z}_{1:t-1}, z_t, \mathbf{z}_{t+1:T} | \mathbf{x}_{1:T}) \tag{17.72}$$

This makes the MPM in Equation 17.70 more robust, since we estimate each node averaging over its neighbors, rather than conditioning on a specific value of its neighbors.[6]

6. In general, we may want to mix max and sum. For example, consider a joint distribution where we observe

17.4.4.2 Details of the algorithm

It is tempting to think that we can implement Viterbi by just replacing the sum-operator in forwards-backwards with a max-operator. The former is called the **sum-product**, and the latter the **max-product** algorithm. If there is a unique mode, running max-product and then computing using Equation 17.70 will give the same result as using Equation 17.68 (Weiss and Freeman 2001b), but in general, it can lead to incorrect results if there are multiple equally probably joint assignments. The reasons is that each node breaks ties independently and hence may do so in a manner that is inconsistent with its neighbors. The Viterbi algorithm is therefore not quite as simple as replacing sum with max. In particular, the forwards pass does use max-product, but the backwards pass uses a **traceback** procedure to recover the most probable path through the trellis of states. Essentially, once z_t picks its most probable state, the previous nodes condition on this event, and therefore they will break ties consistently.

In more detail, define

$$\delta_t(j) \triangleq \max_{z_1,\ldots,z_{t-1}} p(\mathbf{z}_{1:t-1}, z_t = j|\mathbf{x}_{1:t}) \tag{17.73}$$

This is the probability of ending up in state j at time t, given that we take the most probable path. The key insight is that the most probable path to state j at time t must consist of the most probable path to some other state i at time $t-1$, followed by a transition from i to j. Hence

$$\delta_t(j) = \max_i \delta_{t-1}(i)\psi(i,j)\phi_t(j) \tag{17.74}$$

We also keep track of the most likely previous state, for each possible state that we end up in:

$$a_t(j) = \operatorname*{argmax}_i \delta_{t-1}(i)\psi(i,j)\phi_t(j) \tag{17.75}$$

That is, $a_t(j)$ tells us the most likely previous state on the most probable path to $z_t = j$. We initialize by setting

$$\delta_1(j) = \pi_j \phi_1(j) \tag{17.76}$$

and we terminate by computing the most probable final state z_T^*:

$$z_T^* = \arg\max_i \delta_T(i) \tag{17.77}$$

We can then compute the most probable sequence of states using **traceback**:

$$z_t^* = a_{t+1}(z_{t+1}^*) \tag{17.78}$$

As usual, we have to worry about numerical underflow. We are free to normalize the $\boldsymbol{\delta}_t$ terms at each step; this will not affect the maximum. However, unlike the forwards-backwards case,

---

v and we want to query q; let n be the remaining nuisance variables. We define the MAP estimate as $\mathbf{x}_q^* = \arg\max_{\mathbf{x}_q} \sum_{\mathbf{x}_n} p(\mathbf{x}_q, \mathbf{x}_n|\mathbf{x}_v)$, where we max over $\mathbf{x}_q$ and sum over $\mathbf{x}_n$. By contrast, we define the **MPE** or most probable explanation as $(\mathbf{x}_q^*, \mathbf{x}_n^*) = \arg\max_{\mathbf{x}_q,\mathbf{x}_n} p(\mathbf{x}_q, \mathbf{x}_n|\mathbf{x}_v)$, where we max over both $\mathbf{x}_q$ and $\mathbf{x}_n$. This terminology is due to (Pearl 1988), although it is not widely used outside the Bayes net literatire. Obviously MAP=MPE if $n = \emptyset$. However, if $n \neq \emptyset$, then summing out the nuisance variables can give different results than maxing them out. Summing out nuisance variables is more sensible, but computationally harder, because of the need to combine max and sum operations (Lerner and Parr 2001).

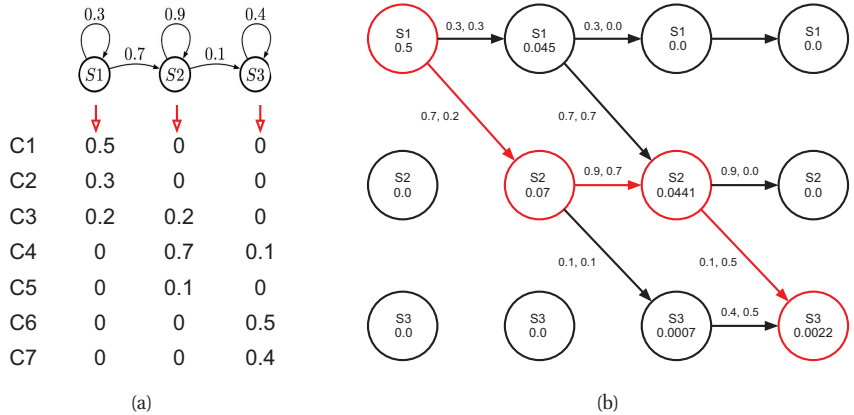


Figure 17.13 Illustration of Viterbi decoding in a simple HMM for speech recognition. (a) A 3-state HMM for a single phone. We are visualizing the state transition diagram. We assume the observations have been vector quantized into 7 possible symbols, $C_1, \ldots, C_7$. Each state z_1, z_2, z_3 has a different distribution over these symbols. Based on Figure 15.20 of (Russell and Norvig 2002). (b) Illustration of the Viterbi algorithm applied to this model, with data sequence $C1, C3, C4, C6$. The columns represent time, and the rows represent states. An arrow from state i at $t-1$ to state j at t is annotated with two numbers: the first is the probability of the $i \to j$ transition, and the second is the probability of generating observation $\mathbf{x}_t$ from state j. The bold lines/ circles represent the most probable sequence of states. Based on Figure 24.27 of (Russell and Norvig 1995).

we can also easily work in the log domain. The key difference is that $\log \max = \max \log$, whereas $\log \sum \neq \sum \log$. Hence we can use

$$\log \delta_t(j) \triangleq \max_{\mathbf{z}_{1:t-1}} \log p(\mathbf{z}_{1:t-1}, z_t = j | \mathbf{x}_{1:t}) \tag{17.79}$$

$$= \max_i \log \delta_{t-1}(i) + \log \psi(i,j) + \log \phi_t(j) \tag{17.80}$$

In the case of Gaussian observation models, this can result in a significant (constant factor) speedup, since computing $\log p(\mathbf{x}_t|z_t)$ can be much faster than computing $p(\mathbf{x}_t|z_t)$ for a high-dimensional Gaussian. This is one reason why the Viterbi algorithm is widely used in the E step of EM (Section 17.5.2) when training large speech recognition systems based on HMMs.

17.4.4.3 Example

Figure 17.13 gives a worked example of the Viterbi algorithm, based on (Russell et al. 1995). Suppose we observe the discrete sequence of observations $\mathbf{x}_{1:4} = (C_1, C_3, C_4, C_6)$, representing codebook entries in a vector-quantized version of a speech signal. The model starts in state z_1. The probability of generating C_1 in z_1 is 0.5, so we have $\delta_1(1) = 0.5$, and $\delta_1(i) = 0$ for all other states. Next we can self-transition to z_1 with probability 0.3, or transition to z_2 with proabability 0.7. If we end up in z_1, the probability of generating C_3 is 0.3; if we end up in z_2,

the probability of generating C_3 is 0.2. Hence we have

$$\delta_2(1) = \delta_1(1)\psi(1,1)\phi_2(1) = 0.5 \cdot 0.3 \cdot 0.3 = 0.045 \quad (17.81)$$

$$\delta_2(2) = \delta_1(1)\psi(1,2)\phi_2(2) = 0.5 \cdot 0.7 \cdot 0.2 = 0.07 \quad (17.82)$$

Thus state 2 is more probable at $t = 2$; see the second column of Figure 17.13(b). In time step 3, we see that there are two paths into z_2, from z_1 and from z_2. The bold arrow indicates that the latter is more probable. Hence this is the only one we have to remember. The algorithm continues in this way until we have reached the end of the sequence. One we have reached the end, we can follow the black arrows back to recover the MAP path (which is 1,2,2,3).

17.4.4.4 Time and space complexity

The time complexity of Viterbi is clearly $O(K^2T)$ in general, and the space complexity is $O(KT)$, both the same as forwards-backwards. If the transition matrix has the form $\psi(i,j) \propto \exp(-\sigma^2||\mathbf{z}_i - \mathbf{z}_j||^2)$, where $\mathbf{z}_i$ is the continuous vector represented by state i, we can implement Viterbi in $O(TK)$ time, instead of $O(TK \log K)$ needed by forwards-backwards. See Section 22.2.6.1 for details.

17.4.4.5 N-best list

The Viterbi algorithm returns one of the most probable paths. It can be extended to return the top N paths (Schwarz and Chow 1990; Nilsson and Goldberger 2001). This is called the **N-best list**. Once can then use a discriminative method to rerank the paths based on global features derived from the fully observed state sequence (as well as the visible features). This technique is widely used in speech recognition. For example, consider the sentence "recognize speech". It is possible that the most probable interpretation by the system of this acoustic signal is "wreck a nice speech", or maybe "wreck a nice beach". Maybe the correct interpretation is much lower down on the list. However, by using a re-ranking system, we may be able to improve the score of the correct interpretation based on a more global context.

One problem with the N-best list is that often the top N paths are very similar to each other, rather than representing qualitatively different interpretations of the data. Instead we might want to generate a more diverse set of paths to more accurately represent posterior uncertainty. One way to do this is to sample paths from the posterior, as we discuss below. For some other ways to generate diverse MAP estimates, see e.g., (Yadollahpour et al. 2011; Kulesza and Taskar 2011).

17.4.5 Forwards filtering, backwards sampling

It is often useful to sample paths from the posterior:

$$\mathbf{z}_{1:T}^s \sim p(\mathbf{z}_{1:T}|\mathbf{x}_{1:T}) \quad (17.83)$$

We can do this is as follow: run forwards backwards, to compute the two-slice smoothed posteriors, $p(z_{t-1,t}|\mathbf{x}_{1:T})$; next compute the conditionals $p(z_t|z_{t-1}, \mathbf{x}_{1:T})$ by normalizing; sample from the initial pair of states, $z_{1,2}^* \sim p(z_{1,2}|\mathbf{x}_{1:T})$; finally, recursively sample $z_t^* \sim p(z_t|z_{t-1}^*, \mathbf{x}_{1:T})$.

Note that the above solution requires a forwards-backwards pass, and then an additional forwards sampling pass. An alternative is to do the forwards pass, and then perform sampling

in the backwards pass. The key insight into how to do this is that we can write the joint from right to left using

$$p(\mathbf{z}_{1:T}|\mathbf{x}_{1:T}) = p(z_T|\mathbf{x}_{1:T}) \prod_{t=T-1}^{1} p(z_t|z_{t+1}, \mathbf{x}_{1:T}) \tag{17.84}$$

We can then sample z_t given future sampled states using

$$z_t^s \sim p(z_t|z_{t+1:T}, \mathbf{x}_{1:T}) = p(z_t|z_{t+1}, \cancel{\mathbf{z}_{t+2:T}}, \mathbf{x}_{1:t}, \cancel{\mathbf{x}_{t+1:T}}) = p(z_t|z_{t+1}^s, \mathbf{x}_{1:t}) \tag{17.85}$$

The sampling distribution is given by

$$\begin{aligned} p(z_t = i|z_{t+1} = j, \mathbf{x}_{1:t}) &= p(z_t|z_{t+1}, \mathbf{x}_{1:t}, \cancel{\mathbf{x}_{t+1}}) && (17.86)\\ &= \frac{p(z_{t+1}, z_t|\mathbf{x}_{1:t+1})}{p(z_{t+1}|\mathbf{x}_{1:t+1})} && (17.87)\\ &\propto \frac{p(\mathbf{x}_{t+1}|z_{t+1}, \cancel{z_t}, \cancel{\mathbf{x}_{1:t}})p(z_{t+1}, z_t|\mathbf{x}_{1:t})}{p(z_{t+1}|\mathbf{x}_{1:t+1})} && (17.88)\\ &= \frac{p(\mathbf{x}_{t+1}|z_{t+1})p(z_{t+1}|z_t, \cancel{\mathbf{x}_{1:t}})p(z_t|\mathbf{x}_{1:t})}{p(z_{t+1}|\mathbf{x}_{1:t+1})} && (17.89)\\ &= \frac{\phi_{t+1}(j)\psi(i,j)\alpha_t(i)}{\alpha_{t+1}(j)} && (17.90) \end{aligned}$$

The base case is

$$z_T^s \sim p(z_T = i|\mathbf{x}_{1:T}) = \alpha_T(i) \tag{17.91}$$

This algorithm forms the basis of blocked-Gibbs sampling methods for parameter inference, as we will see below.

17.5 Learning for HMMs

We now discuss how to estimate the parameters $\boldsymbol{\theta} = (\boldsymbol{\pi}, \mathbf{A}, \mathbf{B})$, where $\pi(i) = p(z_1 = i)$ is the initial state distribution, $A(i,j) = p(z_t = j|z_{t-1} = i)$ is the transition matrix, and $\mathbf{B}$ are the parameters of the class-conditional densities $p(\mathbf{x}_t|z_t = j)$. We first consider the case where $\mathbf{z}_{1:T}$ is observed in the training set, and then the harder case where $\mathbf{z}_{1:T}$ is hidden.

17.5.1 Training with fully observed data

If we observe the hidden state sequences, we can compute the MLEs for $\mathbf{A}$ and $\boldsymbol{\pi}$ exactly as in Section 17.2.2.1. If we use a conjugate prior, we can also easily compute the posterior.

The details on how to estimate $\mathbf{B}$ depend on the form of the observation model. The situation is identical to fitting a generative classifier. For example, if each state has a multinoulli distribution associated with it, with parameters $B_{jl} = p(X_t = l|z_t = j)$, where $l \in \{1, \ldots, L\}$ represents the observed symbol, the MLE is given by

$$\hat{B}_{jl} = \frac{N_{jl}^X}{N_j}, \quad N_{jl}^X \triangleq \sum_{i=1}^{N} \sum_{t=1}^{T_i} \mathbb{I}(z_{i,t} = j, x_{i,t} = l) \tag{17.92}$$

This result is quite intuitive: we simply add up the number of times we are in state j and we see a symbol l, and divide by the number of times we are in state j.

Similarly, if each state has a Gaussian distribution associated with it, we have (from Section 4.2.4) the following MLEs:

$$\hat{\boldsymbol{\mu}}_k = \frac{\overline{\mathbf{x}}_k}{N_k}, \;\; \hat{\boldsymbol{\Sigma}}_k = \frac{(\overline{\mathbf{xx}})_k^T - N_k \hat{\boldsymbol{\mu}}_k \hat{\boldsymbol{\mu}}_k^T}{N_k} \tag{17.93}$$

where the sufficient statistics are given by

$$\overline{\mathbf{x}}_k \triangleq \sum_{i=1}^{N} \sum_{t=1}^{T_i} \mathbb{I}(z_{i,t} = k) \mathbf{x}_{i,t} \tag{17.94}$$

$$(\overline{\mathbf{xx}})_k^T \triangleq \sum_{i=1}^{N} \sum_{t=1}^{T_i} \mathbb{I}(z_{i,t} = k) \mathbf{x}_{i,t} \mathbf{x}_{i,t}^T \tag{17.95}$$

Analogous results can be derived for other kinds of distributions. One can also easily extend all of these results to compute MAP estimates, or even full posteriors over the parameters.

17.5.2 EM for HMMs (the Baum-Welch algorithm)

If the z_t variables are not observed, we are in a situation analogous to fitting a mixture model. The most common approach is to use the EM algorithm to find the MLE or MAP parameters, although of course one could use other gradient-based methods (see e.g., (Baldi and Chauvin 1994)). In this Section, we derive the EM algorithm. When applied to HMMs, this is also known as the **Baum-Welch** algorithm (Baum et al. 1970).

17.5.2.1 E step

It is straightforward to show that the expected complete data log likelihood is given by

$$Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{old}) = \sum_{k=1}^{K} \mathbb{E}\left[N_k^1\right] \log \pi_k + \sum_{j=1}^{K} \sum_{k=1}^{K} \mathbb{E}\left[N_{jk}\right] \log A_{jk} \tag{17.96}$$

$$+ \sum_{i=1}^{N} \sum_{t=1}^{T_i} \sum_{k=1}^{K} p(z_t = k | \mathbf{x}_i, \boldsymbol{\theta}^{old}) \log p(\mathbf{x}_{i,t} | \boldsymbol{\phi}_k) \tag{17.97}$$

where the expected counts are given by

$$\mathbb{E}\left[N_k^1\right] = \sum_{i=1}^{N} p(z_{i1} = k | \mathbf{x}_i, \boldsymbol{\theta}^{old}) \tag{17.98}$$

$$\mathbb{E}\left[N_{jk}\right] = \sum_{i=1}^{N} \sum_{t=2}^{T_i} p(z_{i,t-1} = j, z_{i,t} = k | \mathbf{x}_i, \boldsymbol{\theta}^{old}) \tag{17.99}$$

$$\mathbb{E}\left[N_j\right] = \sum_{i=1}^{N} \sum_{t=1}^{T_i} p(z_{i,t} = j | \mathbf{x}_i, \boldsymbol{\theta}^{old}) \tag{17.100}$$

These expected sufficient statistics can be computed by running the forwards-backwards algorithm on each sequence. In particular, this algorithm computes the following smoothed node and edge marginals:

$$\gamma_{i,t}(j) \triangleq p(z_t = j|\mathbf{x}_{i,1:T_i}, \boldsymbol{\theta}) \tag{17.101}$$

$$\xi_{i,t}(j,k) \triangleq p(z_{t-1} = j, z_t = k|\mathbf{x}_{i,1:T_i}, \boldsymbol{\theta}) \tag{17.102}$$

17.5.2.2 M step

Based on Section 11.3, we have that the M step for $\mathbf{A}$ and $\boldsymbol{\pi}$ is to just normalize the expected counts:

$$\hat{A}_{jk} = \frac{\mathbb{E}\left[N_{jk}\right]}{\sum_{k'} \mathbb{E}\left[N_{jk'}\right]}, \; \hat{\pi}_k = \frac{\mathbb{E}\left[N_k^1\right]}{N} \tag{17.103}$$

This result is quite intuitive: we simply add up the expected number of transitions from j to k, and divide by the expected number of times we transition from j to anything else.

For a multinoulli observation model, the expected sufficient statistics are

$$\mathbb{E}\left[M_{jl}\right] = \sum_{i=1}^{N}\sum_{t=1}^{T_i} \gamma_{i,t}(j)\mathbb{I}(x_{i,t} = l) = \sum_{i=1}^{N}\sum_{t:x_{i,t}=l} \gamma_{i,t}(j) \tag{17.104}$$

The M step has the form

$$\hat{B}_{jl} = \frac{\mathbb{E}\left[M_{jl}\right]}{\mathbb{E}\left[N_j\right]} \tag{17.105}$$

This result is quite intuitive: we simply add up the expected number of times we are in state j and we see a symbol l, and divide by the expected number of times we are in state j.

For a Gaussian observation model, the expected sufficient statistics are given by

$$\mathbb{E}\left[\overline{\mathbf{x}}_k\right] = \sum_{i=1}^{N}\sum_{t=1}^{T_i} \gamma_{i,t}(k)\mathbf{x}_{i,t} \tag{17.106}$$

$$\mathbb{E}\left[(\overline{\mathbf{x}\mathbf{x}})_k^T\right] = \sum_{i=1}^{N}\sum_{t=1}^{T_i} \gamma_{i,t}(k)\mathbf{x}_{i,t}\mathbf{x}_{i,t}^T \tag{17.107}$$

The M step becomes

$$\hat{\boldsymbol{\mu}}_k = \frac{\mathbb{E}\left[\overline{\mathbf{x}}_k\right]}{\mathbb{E}\left[N_k\right]}, \; \hat{\boldsymbol{\Sigma}}_k = \frac{\mathbb{E}\left[(\overline{\mathbf{x}\mathbf{x}})_k^T\right] - \mathbb{E}\left[N_k\right]\hat{\boldsymbol{\mu}}_k\hat{\boldsymbol{\mu}}_k^T}{\mathbb{E}\left[N_k\right]} \tag{17.108}$$

This can (and should) be regularized in the same way we regularize GMMs.

17.5.2.3 Initialization

As usual with EM, we must take care to ensure that we initialize the parameters carefully, to minimize the chance of getting stuck in poor local optima. There are several ways to do this, such as

- Use some fully labeled data to initialize the parameters.
- Initially ignore the Markov dependencies, and estimate the observation parameters using the standard mixture model estimation methods, such as K-means or EM.
- Randomly initialize the parameters, use multiple restarts, and pick the best solution.

Techniques such as deterministic annealing (Ueda and Nakano 1998; Rao and Rose 2001) can help mitigate the effect of local minima. Also, just as K-means is often used to initialize EM for GMMs, so it is common to initialize EM for HMMs using **Viterbi training**, which means approximating the posterior over paths with the single most probable path. (This is not necessarily a good idea, since initially the parameters are often poorly estimated, so the Viterbi path will be fairly arbitrary. A safer option is to start training using forwards-backwards, and to switch to Viterbi near convergence.)

17.5.3 Bayesian methods for "fitting" HMMs *

EM returns a MAP estimate of the parameters. In this section, we briefly discuss some methods for Bayesian parameter estimation in HMMs. (These methods rely on material that we will cover later in the book.)

One approach is to use variational Bayes EM (VBEM), which we discuss in general terms in Section 21.6. The details for the HMM case can be found in (MacKay 1997; Beal 2003), but the basic idea is this: The E step uses forwards-backwards, but where (roughly speaking) we plug in the posterior mean parameters instead of the MAP estimates. The M step updates the parameters of the conjugate posteriors, instead of updating the parameters themselves.

An alternative to VBEM is to use MCMC. A particularly appealing algorithm is block Gibbs sampling, which we discuss in general terms in Section 24.2.8. The details for the HMM case can be found in (Fruhwirth-Schnatter 2007), but the basic idea is this: we sample $\mathbf{z}_{1:T}$ given the data and parameters using forwards-filtering, backwards-sampling, and we then sample the parameters from their posteriors, conditional on the sampled latent paths. This is simple to implement, but one does need to take care of unidentifiability (label switching), just as with mixture models (see Section 11.3.1).

17.5.4 Discriminative training

Sometimes HMMs are used as the class conditional density inside a generative classifier. In this case, $p(\mathbf{x}|y=c,\boldsymbol{\theta})$ can be computed using the forwards algorithm. We can easily maximize the joint likelihood $\prod_{i=1}^{N} p(\mathbf{x}_i, y_i|\boldsymbol{\theta})$ by using EM (or some other method) to fit the HMM for each class-conditional density separately.

However, we might like to find the parameters that maximize the conditional likelihood

$$\prod_{i=1}^{N} p(y_i|\mathbf{x}_i,\boldsymbol{\theta}) = \prod_i \frac{p(y_i|\boldsymbol{\theta})p(\mathbf{x}_i|y_i,\boldsymbol{\theta})}{\sum_c p(y_i=c|\boldsymbol{\theta})p(\mathbf{x}_i|c,\boldsymbol{\theta})} \tag{17.109}$$

This is more expensive than maximizing the joint likelihood, since the denominator couples all C class-conditional HMMs together. Furthermore, EM can no longer be used, and one must resort

to generic gradient based methods. Nevertheless, discriminative training can result in improved accuracies. The standard practice in speech recognition is to initially train the generative models separately using EM, and then to fine tune them discriminatively (Jelinek 1997).

17.5.5 Model selection

In HMMs, the two main model selection issues are: how many states, and what topology to use for the state transition diagram. We discuss both of these issues below.

17.5.5.1 Choosing the number of hidden states

Choosing the number of hidden states K in an HMM is analogous to the problem of choosing the number of mixture components. Here are some possible solutions:

- Use grid-search over a range of K's, using as an objective function cross-validated likelihood, the BIC score, or a variational lower bound to the log-marginal likelihood.
- Use reversible jump MCMC. See (Fruhwirth-Schnatter 2007) for details. Note that this is very slow and is not widely used.
- Use variational Bayes to "extinguish" unwanted components, by analogy to the GMM case discussed in Section 21.6.1.6. See (MacKay 1997; Beal 2003) for details.
- Use an "infinite HMM", which is based on the hierarchical Dirichlet process. See e.g., (Beal et al. 2002; Teh et al. 2006) for details.

17.5.5.2 Structure learning

The term **structure learning** in the context of HMMs refers to learning a sparse transition matrix. That is, we want to learn the structure of the state transition diagram, not the structure of the graphical model (which is fixed). A large number of heuristic methods have been proposed. Most alternate between parameter estimation and some kind of heuristic **split merge** method (see e.g., (Stolcke and Omohundro 1992)).

Alternatively, one can pose the problem as MAP estimation using a **minimum entropy prior**, of the form

$$p(\mathbf{A}_{i,:}) \propto \exp(-\mathbb{H}(\mathbf{A}_{i,:})) \tag{17.110}$$

This prior prefers states whose outgoing distribution is nearly deterministic, and hence has low entropy (Brand 1999). The corresponding M step cannot be solved in closed form, but numerical methods can be used. The trouble with this is that we might prune out all incoming transitions to a state, creating isolated "islands" in state-space. The infinite HMM presents an interesting alternative to these methods. See e.g., (Beal et al. 2002; Teh et al. 2006) for details.

17.6 Generalizations of HMMs

Many variants of the basic HMM model have been proposed. We briefly discuss some of them below.

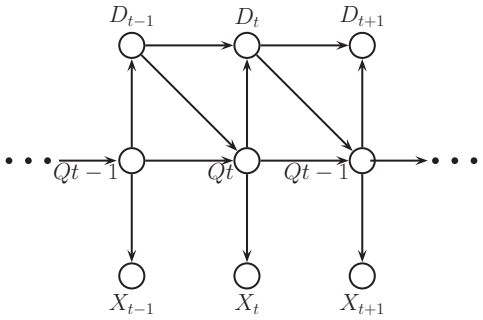


Figure 17.14 Encoding a hidden semi-Markov model as a DGM. D_t are deterministic duration counters.

17.6.1 Variable duration (semi-Markov) HMMs

In a standard HMM, the probability we remain in state i for exactly d steps is

$$p(t_i = d) = (1 - A_{ii})A_{ii}^d \propto \exp(d \log A_{ii}) \tag{17.111}$$

where A_{ii} is the self-loop probability. This is called the **geometric distribution**. However, this kind of exponentially decaying function of d is sometimes unrealistic.

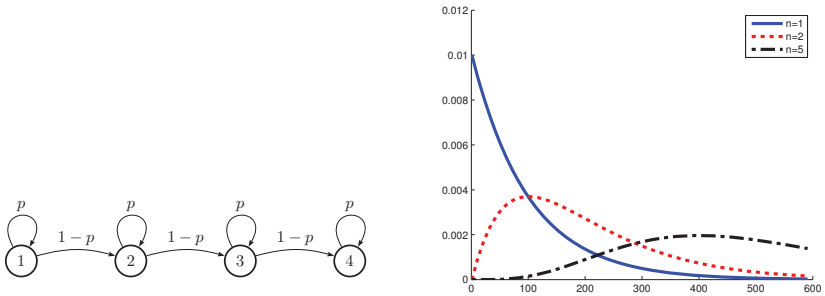
To allow for more general durations, one can use a **semi-Markov model**. It is called semi-Markov because to predict the next state, it is not sufficient to condition on the past state: we also need to know how long we've been in that state. When the state space is not observed directly, the result is called a **hidden semi-Markov model** (**HSMM**), a **variable duration HMM**, or an **explicit duration HMM**.

HSMMs are widely used in many gene finding programs, since the length distribution of exons and introns is not geometric (see e.g., (Schweikerta et al. 2009)), and in some chip-Seq data analysis programs (see e.g., (Kuan et al. 2009)).

HSMMs are useful not only because they can model the waiting time of each state more accurately, but also because they can model the distribution of a whole batch of observations at once, instead of assuming all observations are conditionally iid. That is, they can use likelihood models of the form $p(\mathbf{x}_{t:t+l}|z_t = k, d_t = l)$, which generate l correlated observations if the duration in state k is for l time steps. This is useful for modeling data that is piecewise linear, or shows other local trends (Ostendorf et al. 1996).

17.6.1.1 HSMM as augmented HMMs

One way to represent a HSMM is to use the graphical model shown in Figure 17.14. (In this figure, we have assumed the observations are iid within each state, but this is not required, as mentioned above.) The $D_t \in \{0, 1, \ldots, D\}$ node is a state duration counter, where D is the maximum duration of any state. When we first enter state j, we sample D_t from the duration distribution for that state, $D_t \sim p_j(\cdot)$. Thereafer, D_t deterministically counts down



(a) (b)

Figure 17.15 (a) A Markov chain with $n = 4$ repeated states and self loops. (b) The resulting distribution over sequence lengths, for $p = 0.99$ and various n. Figure generated by `hmmSelfLoopDist`.

until $D_t = 0$. While $D_t > 0$, the state z_t is not allowed to change. When $D_t = 0$, we make a stochastic transition to a new state.

More precisely, we define the CPDs as follows:

$$p(D_t = d'|D_{t-1} = d, z_t = j) = \begin{cases} p_j(d') & \text{if } d = 0 \\ 1 & \text{if } d' = d - 1 \text{ and } d \geq 1 \\ 0 & \text{otherwise} \end{cases} \tag{17.112}$$

$$p(z_t = k|z_{t-1} = j, D_{t-1} = d) = \begin{cases} 1 & \text{if } d > 0 \text{ and } j = k \\ A_{jk} & \text{if } d = 0 \\ 0 & \text{otherwise} \end{cases} \tag{17.113}$$

Note that $p_j(d)$ could be represented as a table (a non-parametric approach) or as some kind of parametric distribution, such as a Gamma distribution. If $p_j(d)$ is a geometric distribution, this emulates a standard HMM.

One can perform inference in this model by defining a mega-variable $Y_t = (D_t, z_t)$. However, this is rather inefficient, since D_t is deterministic. It is possible to marginalize D_t out, and derive special purpose inference procedures. See (Guedon 2003; Yu and Kobayashi 2006) for details. Unfortunately, all these methods take $O(TK^2D)$ time, where T is the sequence length, K is the number of states, and D is the maximum duration of any state.

17.6.1.2 Approximations to semi-Markov models

A more efficient, but less flexible, way to model non-geometric waiting times is to replace each state with n new states, each with the same emission probabilities as the original state. For example, consider the model in Figure 17.15(a). Obviously the smallest sequence this can generate is of length $n = 4$. Any path of length d through the model has probability $p^{d-n}(1-p)^n$; multiplying by the number of possible paths we find that the total probability of a path of length d is

$$p(d) = \binom{d-1}{n-1} p^{d-n}(1-p)^n \tag{17.114}$$

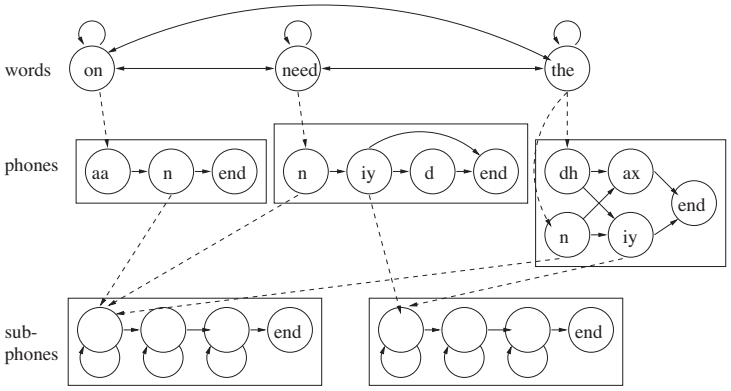


Figure 17.16 An example of an HHMM for an ASR system which can recognize 3 words. The top level represents bigram word probabilities. The middle level represents the phonetic spelling of each word. The bottom level represents the subphones of each phone. (It is traditional to represent a phone as a 3 state HMM, representing the beginning, middle and end.) Based on Figure 7.5 of (Jurafsky and Martin 2000).

This is equivalent to the negative binomial distribution. By adjusting n and the self-loop probabilities p of each state, we can model a wide range of waiting times: see Figure 17.15(b).

Let E be the number of expansions of each state needed to approximate $p_j(d)$. Forwards-backwards on this model takes $O(T(KE)F_{in})$ time, where F_{in} is the average number of predecessor states, compared to $O(TK(F_{in}+D))$ for the HSMM. For typical speech recognition applications, $F_{in} \sim 3$, $D \sim 50$, $K \sim 10^6$, $T \sim 10^5$. (Similar figures apply to problems such as gene finding, which also often uses HSMMs.) Since $F_{in} + D \gg EF_{in}$, the expanded state method is much faster than an HSMM. See (Johnson 2005) for details.

17.6.2 Hierarchical HMMs

A **hierarchical HMM** (HHMM) (Fine et al. 1998) is an extension of the HMM that is designed to model domains with hierarchical structure. Figure 17.16 gives an example of an HHMM used in automatic speech recognition. The phone and subphone models can be "called" from different higher level contexts. We can always "flatten" an HHMM to a regular HMM, but a factored representation is often easier to interpret, and allows for more efficient inference and model fitting.

HHMMs have been used in many application domains, e.g., speech recognition (Bilmes 2001), gene finding (Hu et al. 2000), plan recognition (Bui et al. 2002), monitoring transportation patterns (Liao et al. 2007), indoor robot localization (Theocharous et al. 2004), etc. HHMMs are less expressive than stochastic context free grammars (SCFGs), since they only allow hierarchies of bounded depth, but they support more efficient inference. In particular, inference in SCFGs (using the inside outside algorithm, (Jurafsky and Martin 2008)) takes $O(T^3)$ whereas inference in an HHMM takes $O(T)$ time (Murphy and Paskin 2001).

We can represent an HHMM as a directed graphical model as shown in Figure 17.17. Q_t^ℓ represents the state at time t and level ℓ. A state transition at level ℓ is only "allowed" if the

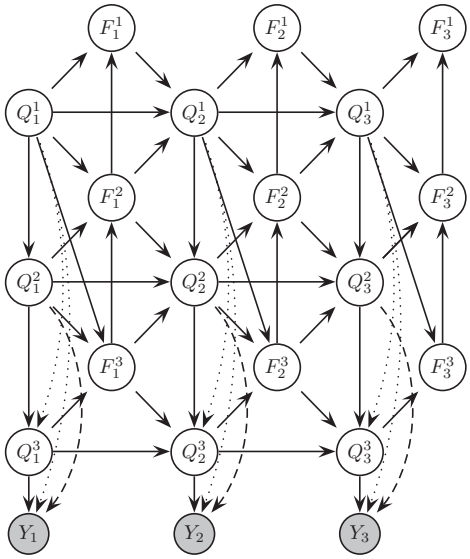


Figure 17.17 An HHMM represented as a DGM. Q_t^ℓ is the state at time t, level ℓ; $F_t^\ell = 1$ if the HMM at level ℓ has finished (entered its exit state), otherwise $F_t^\ell = 0$. Shaded nodes are observed; the remaining nodes are hidden. We may optionally clamp $F_T^\ell = 1$, where T is the length of the observation sequence, to ensure all models have finished by the end of the sequence. Source: Figure 2 of (Murphy and Paskin 2001).

chain at the level below has "finished", as determined by the $F_t^{\ell-1}$ node. (The chain below finishes when it chooses to enter its end state.) This mechanism ensures that higher level chains evolve more slowly than lower level chains, i.e., lower levels are nested within higher levels.

A variable duration HMM can be thought of as a special case of an HHMM, where the top level is a deterministic counter, and the bottom level is a regular HMM, which can only change states once the counter has "timed out". See (Murphy and Paskin 2001) for further details.

17.6.3 Input-output HMMs

It is straightforward to extend an HMM to handle inputs, as shown in Figure 17.18(a). This defines a conditional density model for sequences of the form

$$p(\mathbf{y}_{1:T}, \mathbf{z}_{1:T}|\mathbf{u}_{1:T}, \boldsymbol{\theta}) \tag{17.115}$$

where $\mathbf{u}_t$ is the input at time t; this is sometimes called a control signal. If the inputs and outputs are continuous, a typical parameterization would be

$$p(z_t|\mathbf{x}_t, z_{t-1} = i, \boldsymbol{\theta}) = \mathrm{Cat}(z_t|\mathcal{S}(\mathbf{W}_i\mathbf{u}_t)) \tag{17.116}$$

$$p(\mathbf{y}_t|\mathbf{x}_t, z_t = j, \boldsymbol{\theta}) = \mathcal{N}(\mathbf{y}_t|\mathbf{V}_j\mathbf{u}_t, \boldsymbol{\Sigma}_j) \tag{17.117}$$

Thus the transition matrix is a logistic regression model whose parameters depend on the previous state. The observation model is a Gaussian whose parameters depend on the current

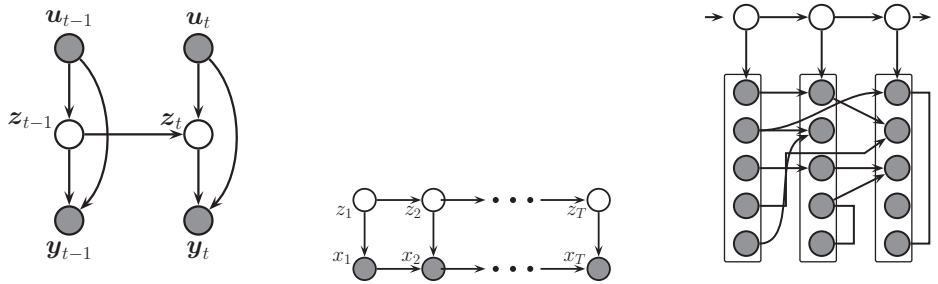


Figure 17.18 (a) Input-output HMM. (b) First-order auto-regressive HMM. (c) A second-order buried Markov model. Depending on the value of the hidden variables, the effective graph structure between the components of the observed variables (i.e., the non-zero elements of the regression matrix and the precision matrix) can change, although this is not shown.

state. The whole model can be thought of as a hidden version of a maximum entropy Markov model (Section 19.6.1).

Conditional on the inputs $\mathbf{u}_{1:T}$ and the parameters $\boldsymbol{\theta}$, one can apply the standard forwards-backwards algorithm to estimate the hidden states. It is also straightforward to derive an EM algorithm to estimate the parameters (see (Bengio and Frasconi 1996) for details).

17.6.4 Auto-regressive and buried HMMs

The standard HMM assumes the observations are conditionally independent given the hidden state. In practice this is often not the case. However, it is straightforward to have direct arcs from $\mathbf{x}_{t-1}$ to $\mathbf{x}_t$ as well as from z_t to $\mathbf{x}_t$, as in Figure 17.18(b). This is known as an **auto-regressive HMM**, or a **regime switching Markov model**. For continuous data, the observation model becomes

$$p(\mathbf{x}_t|\mathbf{x}_{t-1}, z_t = j, \boldsymbol{\theta}) = \mathcal{N}(\mathbf{x}_t|\mathbf{W}_j\mathbf{x}_{t-1} + \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j) \tag{17.118}$$

This is a linear regression model, where the parameters are chosen according to the current hidden state. We can also consider higher-order extensions, where we condition on the last L observations:

$$p(\mathbf{x}_t|\mathbf{x}_{t-L:t-1}, z_t = j, \boldsymbol{\theta}) = \mathcal{N}(\mathbf{x}_t|\sum_{\ell=1}^{L}\mathbf{W}_{j,\ell}\mathbf{x}_{t-\ell} + \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j) \tag{17.119}$$

Such models are widely used in econometrics (Hamilton 1990). Similar models can be defined for discrete observations.

The AR-HMM essentially combines two Markov chains, one on the hidden variables, to capture long range dependencies, and one on the observed variables, to capture short range dependencies (Berchtold 1999). Since the X nodes are observed, the connections between them only

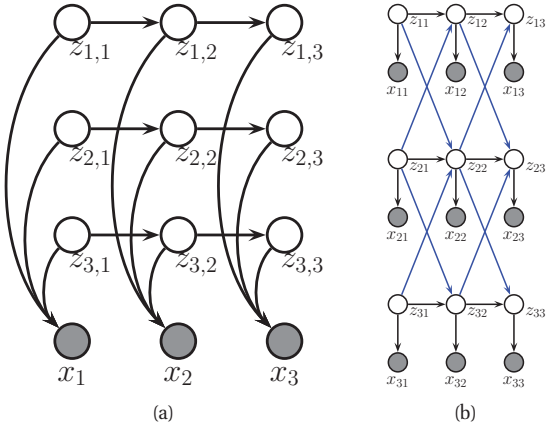


Figure 17.19 (a) A factorial HMM with 3 chains. (b) A coupled HMM with 3 chains.

change the computation of the local evidence; inference can still be performed using the standard forwards-backwards algorithm. Parameter estimation using EM is also straightforward: the E step is unchanged, as is the M step for the transition matrix. If we assume scalar observations for notational simplicty, the M step involves minimizing

$$\sum_t \mathbb{E}\left[\frac{1}{\sigma^2(s_t)}(y_t - \mathbf{y}_{t-L:t-1}^T \mathbf{w}(s_t))^2 + \log \sigma^2(s_t)\right] \tag{17.120}$$

Focussing on the $\mathbf{w}$ terms, we see that this requires solving K weighted least squares problems:

$$J(\mathbf{w}_{1:K}) = \sum_j \sum_t \frac{\gamma_t(j)}{\sigma^2(j)}(y_t - \mathbf{y}_{t-L:t-1}^T \mathbf{w}_j)^2 \tag{17.121}$$

where $\gamma_t(j) = p(z_t = k|\mathbf{x}_{1:T})$ is the smoothed posterior marginal. This is a weighted linear regression problem, where the design matrix has a Toeplitz form. This subproblem can be solved efficiently using the Levinson-Durbin method (Durbin and Koopman 2001).

Buried Markov models generalize AR-HMMs by allowing the dependency structure between the observable nodes to change based on the hidden state, as in Figure 17.18(c). Such a model is called a dynamic Bayesian **multi net**, since it is a mixture of different networks. In the linear-Gaussian setting, we can change the structure of the of $\mathbf{x}_{t-1} \rightarrow \mathbf{x}_t$ arcs by using sparse regression matrices, $\mathbf{W}_j$, and we can change the structure of the connections within the components of $\mathbf{x}_t$ by using sparse Gaussian graphical models, either directed or undirected. See (Bilmes 2000) for details.

17.6.5 Factorial HMM

An HMM represents the hidden state using a single discrete random variable $z_t \in \{1, \ldots, K\}$. To represent 10 bits of information would require $K = 2^{10} = 1024$ states. By contrast, consider a **distributed representation** of the hidden state, where each $z_{c,t} \in \{0, 1\}$ represents the c'th

bit of the t'th hidden state. Now we can represent 10 bits using just 10 binary variables, as illustrated in Figure 17.19(a). This model is called a **factorial HMM** (Ghahramani and Jordan 1997). The hope is that this kind of model could capture different aspects of a signal, e.g., one chain would represent speaking style, another the words that are being spoken.

Unfortunately, conditioned on $\mathbf{x}_t$, all the hidden variables are correlated (due to explaining away the common observed child $\mathbf{x}_t$). This make exact state estimation intractable. However, we can derive efficient approximate inference algorithms, as we discuss in Section 21.4.1.

17.6.6 Coupled HMM and the influence model

If we have multiple related data streams, we can use a **coupled HMM** (Brand 1996), as illustrated in Figure 17.19(b). This is a series of HMMs where the state transitions depend on the states of neighboring chains. That is, we represent the joint conditional distribution as

$$p(\mathbf{z}_t|\mathbf{z}_{t-1}) = \prod_c p(z_{ct}|\mathbf{z}_{t-1}) \tag{17.122}$$

$$p(z_{ct}|\mathbf{z}_{t-1}) = p(z_{ct}|z_{c,t-1},\ z_{c-1,t-1},\ z_{c+1,t-1}) \tag{17.123}$$

This has been used for various tasks, such as **audio-visual speech recognition** (Nefian et al. 2002) and modeling freeway traffic flows (Kwon and Murphy 2000).

The trouble with the above model is that it requires $O(CK^4)$ parameters to specify, if there are C chains with K states per chain, because each state depends on its own past plus the past of its two neighbors. There is a closely related model, known as the **influence model** (Asavathiratham 2000), which uses fewer parameters. It models the joint conditional distribution as

$$p(z_{ct}|\mathbf{z}_{t-1}) = \sum_{c'=1}^{C} \alpha_{c,c'} p(z_{ct}|z_{c',t-1}) \tag{17.124}$$

where $\sum_{c'} \alpha_{c,c'} = 1$ for each c. That is, we use a convex combination of pairwise transition matrices. The $\alpha_{c,c'}$ parameter specifies how much influence chain c has on chain c'. This model only takes $O(C^2 + CK^2)$ parameters to specify. Furthermore, it allows each chain to be influenced by all the other chains, not just its nearest neighbors. (Hence the corresponding graphical model is similar to Figure 17.19(b), except that each node has incoming edges from all the previous nodes.) This has been used for various tasks, such as modeling conversational interactions between people (Basu et al. 2001).

Unfortunately, inference in both of these models takes $O(T(K^C)^2)$ time, since all the chains become fully correlated even if the interaction graph is sparse. Various approximate inference methods can be applied, as we discuss later.

17.6.7 Dynamic Bayesian networks (DBNs)

A **dynamic Bayesian network** is just a way to represent a stochastic process using a directed graphical model.[7] Note that the network is not dynamic (the structure and parameters are fixed),

7. The acronym **DBN** can stand for either "dynamic Bayesian network" or "deep belief network" (Section 28.1) depending on the context. Geoff Hinton (who invented the term "deep belief network") has suggested the acronyms **DyBN** and **DeeBN** to avoid this ambiguity.

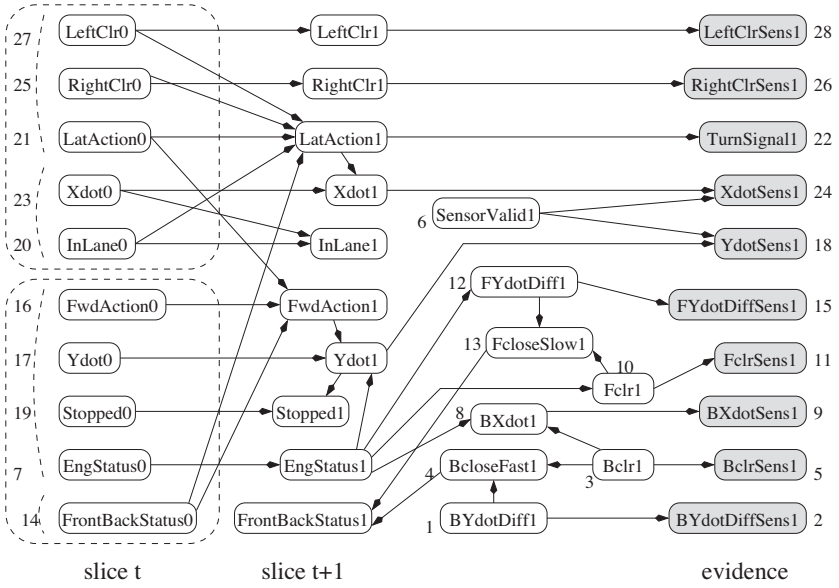


Figure 17.20 The BATnet DBN. The transient nodes are only shown for the second slice, to minimize clutter. The dotted lines can be ignored. Used with kind permission of Daphne Koller.

rather it is a network representation of a dynamical system. All of the HMM variants we have seen above could be considered to be DBNs. However, we prefer to reserve the term "DBN" for graph structures that are more "irregular" and problem-specific. An example is shown in Figure 17.20, which is a DBN designed to monitor the state of a simulated autonomous car known as the "Bayesian Automated Taxi", or "BATmobile" (Forbes et al. 1995).

Defining DBNs is straightforward: you just need to specify the structure of the first time-slice, the structure between two time-slices, and the form of the CPDs. Learning is also easy. The main problem is that exact inference can be computationally expensive, because all the hidden variables become correlated over time (this is known as **entanglement** — see e.g., (Koller and Friedman 2009, Sec. 15.2.4) for details). Thus a sparse graph does not necessarily result in tractable exact inference. However, later we will see algorithms that can exploit the graph structure for efficient approximate inference.

Exercises

Exercise 17.1 Derivation of Q function for HMM

Derive Equation 17.97.

Exercise 17.2 Two filter approach to smoothing in HMMs

Assuming that $\Pi_t(i) = p(S_t = i) > 0$ for all i and t, derive a recursive algorithm for updating $r_t(i) = p(S_t = i|\mathbf{x}_{t+1:T})$. Hint: it should be very similar to the standard forwards algorithm, but using a time-reversed transition matrix. Then show how to compute the posterior marginals $\gamma_t(i) = p(S_t = i|\mathbf{x}_{1:T})$

from the backwards filtered messages $r_t(i)$, the forwards filtered messages $\alpha_t(i)$, and the stationary distribution $\Pi_t(i)$.

Exercise 17.3 EM for for HMMs with mixture of Gaussian observations

Consider an HMM where the observation model has the form

$$p(\mathbf{x}_t|z_t = j, \boldsymbol{\theta}) = \sum_k w_{jk} \mathcal{N}(\mathbf{x}_t|\mu_{jk}, \boldsymbol{\Sigma}_{jk}) \tag{17.125}$$

- Draw the DGM.
- Derive the E step.
- Derive the M step.

Exercise 17.4 EM for for HMMs with tied mixtures

In many applications, it is common that the observations are high-dimensional vectors (e.g., in speech recognition, $\mathbf{x}_t$ is often a vector of cepstral coefficients and their derivatives, so $\mathbf{x}_t \in \mathbb{R}^{39}$), so estimating a full covariance matrix for KM values (where M is the number of mixture components per hidden state), as in Exercise 17.3, requires a lot of data. An alternative is to use just M Gaussians, rather than MK Gaussians, and to let the state influence the mixing weights but not the means and covariances. This is called a **semi-continuous HMM** or **tied-mixture HMM**.

- Draw the corresponding graphical model.
- Derive the E step.
- Derive the M step.

18 State space models

18.1 Introduction

A **state space model** or **SSM** is just like an HMM, except the hidden states are continuous. The model can be written in the following generic form:

$$\mathbf{z}_t = g(\mathbf{u}_t, \mathbf{z}_{t-1}, \boldsymbol{\epsilon}_t) \tag{18.1}$$

$$\mathbf{y}_t = h(\mathbf{z}_t, \mathbf{u}_t, \boldsymbol{\delta}_t) \tag{18.2}$$

where $\mathbf{z}_t$ is the hidden state, $\mathbf{u}_t$ is an optional input or control signal, $\mathbf{y}_t$ is the observation, g is the **transition model**, h is the **observation model**, $\boldsymbol{\epsilon}_t$ is the system noise at time t, and $\boldsymbol{\delta}_t$ is the observation noise at time t. We assume that all parameters of the model, $\boldsymbol{\theta}$, are known; if not, they can be included into the hidden state, as we discuss below.

One of the primary goals in using SSMs is to recursively estimate the belief state, $p(\mathbf{z}_t|\mathbf{y}_{1:t}, \mathbf{u}_{1:t}, \boldsymbol{\theta})$. (Note: we will often drop the conditioning on $\mathbf{u}$ and $\boldsymbol{\theta}$ for brevity.) We will discuss algorithms for this later in this chapter. We will also discuss how to convert our beliefs about the hidden state into predictions about future observables by computing the posterior predictive $p(\mathbf{y}_{t+1}|\mathbf{y}_{1:t})$.

An important special case of an SSM is where all the CPDs are linear-Gaussian. In other words, we assume

- The transition model is a linear function

$$\mathbf{z}_t = \mathbf{A}_t \mathbf{z}_{t-1} + \mathbf{B}_t \mathbf{u}_t + \boldsymbol{\epsilon}_t \tag{18.3}$$

- The observation model is a linear function

$$\mathbf{y}_t = \mathbf{C}_t \mathbf{z}_t + \mathbf{D}_t \mathbf{u}_t + \boldsymbol{\delta}_t \tag{18.4}$$

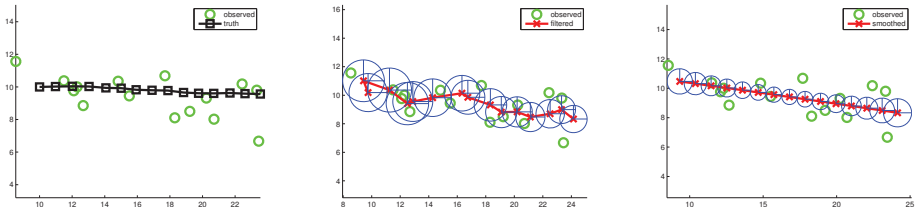
- The system noise is Gaussian

$$\boldsymbol{\epsilon}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{Q}_t) \tag{18.5}$$

- The observation noise is Gaussian

$$\boldsymbol{\delta}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{R}_t) \tag{18.6}$$

This model is called a **linear-Gaussian SSM** (**LG-SSM**) or a **linear dynamical system** (**LDS**). If the parameters $\boldsymbol{\theta}_t = (\mathbf{A}_t, \mathbf{B}_t, \mathbf{C}_t, \mathbf{D}_t, \mathbf{Q}_t, \mathbf{R}_t)$ are independent of time, the model is called **stationary**.



(a) (b) (c)

Figure 18.1 Illustration of Kalman filtering and smoothing. (a) Observations (green cirles) are generated by an object moving to the right (true location denoted by black squares). (b) Filtered estimated is shown by dotted red line. Red cross is the posterior mean, blue circles are 95% confidence ellipses derived from the posterior covariance. For clarity, we only plot the ellipses every other time step. (c) Same as (b), but using offline Kalman smoothing. Figure generated by `kalmanTrackingDemo`.

The LG-SSM is important because it supports exact inference, as we will see. In particular, if the initial belief state is Gaussian, $p(\mathbf{z}_1) = \mathcal{N}(\boldsymbol{\mu}_{1|0}, \boldsymbol{\Sigma}_{1|0})$, then all subsequent belief states will also be Gaussian; we will denote them by $p(\mathbf{z}_t|\mathbf{y}_{1:t}) = \mathcal{N}(\boldsymbol{\mu}_{t|t}, \boldsymbol{\Sigma}_{t|t})$. (The notation $\boldsymbol{\mu}_{t|\tau}$ denotes $\mathbb{E}[\mathbf{z}_t|\mathbf{y}_{1:\tau}]$, and similarly for $\boldsymbol{\Sigma}_{t|t}$; thus $\boldsymbol{\mu}_{t|0}$ denotes the prior for $\mathbf{z}_1$ before we have seen any data. For brevity we will denote the posterior belief states using $\boldsymbol{\mu}_{t|t} = \boldsymbol{\mu}_t$ and $\boldsymbol{\Sigma}_{t|t} = \boldsymbol{\Sigma}_t$.) We can compute these quantities efficiently using the celebrated Kalman filter, as we show in Section 18.3.1. But before discussing algorithms, we discuss some important applications.

18.2 Applications of SSMs

SSMs have many applications, some of which we discuss in the sections below. We mostly focus on LG-SSMs, for simplicity, although non-linear and/or non-Gaussian SSMs are even more widely used.

18.2.1 SSMs for object tracking

One of the earliest applications of Kalman filtering was for tracking objects, such as airplanes and missiles, from noisy measurements, such as radar. Here we give a simplified example to illustrate the key ideas. Consider an object moving in a 2D plane. Let z_{1t} and z_{2t} be the horizontal and vertical locations of the object, and $\dot{z}_{1t}$ and $\dot{z}_{2t}$ be the corresponding velocity. We can represent this as a state vector $\mathbf{z}_t \in \mathbb{R}^4$ as follows:

$$\mathbf{z}_t^T = \begin{pmatrix} z_{1t} & z_{2t} & \dot{z}_{1t} & \dot{z}_{2t} \end{pmatrix}. \tag{18.7}$$

Let us assume that the object is moving at constant velocity, but is "perturbed" by random Gaussian noise (e.g., due to the wind). Thus we can model the system dynamics as follows:

$$\mathbf{z}_t = \mathbf{A}_t \mathbf{z}_{t-1} + \boldsymbol{\epsilon}_t \tag{18.8}$$

$$\begin{pmatrix} z_{1t} \\ z_{2t} \\ \dot{z}_{1t} \\ \dot{z}_{2t} \end{pmatrix} = \begin{pmatrix} 1 & 0 & \Delta & 0 \\ 0 & 1 & 0 & \Delta \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} z_{1,t-1} \\ z_{2,t-1} \\ \dot{z}_{1,t-1} \\ \dot{z}_{2,t-1} \end{pmatrix} + \begin{pmatrix} \epsilon_{1t} \\ \epsilon_{2t} \\ \epsilon_{3t} \\ \epsilon_{4t} \end{pmatrix} \tag{18.9}$$

where $\boldsymbol{\epsilon}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{Q})$ is the system noise, and Δ is the **sampling period**. This says that the new location $z_{j,t}$ is the old location $z_{j,t-1}$ plus Δ times the old velocity $\dot{z}_{j,t-1}$, plus random noise, ϵ_{jt}, for $j = 1 : 2$. Also, the new velocity $\dot{z}_{j,t}$ is the old velocity $\dot{z}_{j,t-1}$ plus random noise, ϵ_{jt}, for $j = 3 : 4$. This is called a **random accelerations model**, since the object moves according to Newton's laws, but is subject to random changes in velocity.

Now suppose that we can observe the location of the object but not its velocity. Let $\mathbf{y}_t \in \mathbb{R}^2$ represent our observation, which we assume is subject to Gaussian noise. We can model this as follows:

$$\mathbf{y}_t = \mathbf{C}_t \mathbf{z}_t + \boldsymbol{\delta}_t \tag{18.10}$$

$$\begin{pmatrix} y_{1t} \\ y_{2t} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} z_{1t} \\ z_{2t} \\ \dot{z}_{1t} \\ \dot{z}_{2t} \end{pmatrix} + \begin{pmatrix} \delta_{1t} \\ \delta_{2t} \\ \delta_{3t} \\ \delta_{4t} \end{pmatrix} \tag{18.11}$$

where $\boldsymbol{\delta}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{R})$ is the measurement noise.

Finally, we need to specify our initial (prior) beliefs about the state of the object, $p(\mathbf{z}_1)$. We will assume this is a Gaussian, $p(\mathbf{z}_1) = \mathcal{N}(\mathbf{z}_1 | \boldsymbol{\mu}_{1|0}, \boldsymbol{\Sigma}_{1|0})$. We can represent prior ignorance by making $\boldsymbol{\Sigma}_{1|0}$ suitably "broad", e.g., $\boldsymbol{\Sigma}_{1|0} = \infty \mathbf{I}$. We have now fully specified the model and can perform sequential Bayesian updating to compute $p(\mathbf{z}_t | \mathbf{y}_{1:t})$ using an algorithm known as the Kalman filter, to be described in Section 18.3.1.

Figure 18.1(a) gives an example. The object moves to the right and generates an observation at each time step (think of "blips" on a radar screen). We observe these blips and filter out the noise by using the Kalman filter. At every step, we have $p(\mathbf{z}_t | \mathbf{y}_{1:t})$, from which we can compute $p(z_{1t}, z_{2t} | \mathbf{y}_{1:t})$ by marginalizing out the dimensions corresponding to the velocities. (This is easy to do since the posterior is Gaussian.) Our "best guess" about the location of the object is the posterior mean, $E[\mathbf{z}_t | \mathbf{y}_{1:t}]$, denoted as a red cross in Figure 18.1(b). Our uncertainty associated with this is represented as an ellipse, which contains 95% of the probability mass. We see that our uncertainty goes down over time, as the effects of the initial uncertainty get "washed out". We also see that the estimated trajectory has "filtered out" some of the noise. To obtain the much smoother plot in Figure 18.1(c), we need to use the Kalman smoother, which computes $p(\mathbf{z}_t | \mathbf{y}_{1:T})$; this depends on "future" as well as "past" data, as discussed in Section 18.3.2.

18.2.2 Robotic SLAM

Consider a robot moving around an unknown 2d world. It needs to learn a map and keep track of its location within that map. This problem is known as **simultaneous localization and**

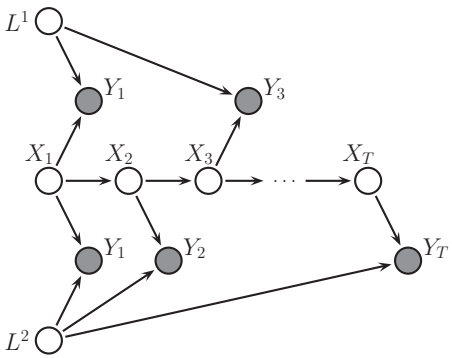


Figure 18.2 Illustration of graphical model underlying SLAM. L^i is the fixed location of landmark i, $\mathbf{x}_t$ is the location of the robot, and $\mathbf{y}_t$ is the observation. In this trace, the robot sees landmarks 1 and 2 at time step 1, then just landmark 2, then just landmark 1, etc. Based on Figure 15.A.3 of (Koller and Friedman 2009).

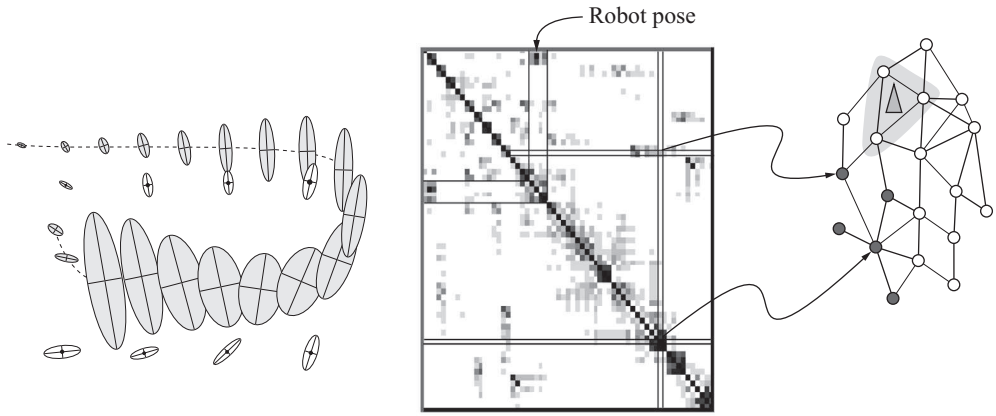


Figure 18.3 Illustration of the SLAM problem. (a) A robot starts at the top left and moves clockwise in a circle back to where it started. We see how the posterior uncertainty about the robot's location increases and then decreases as it returns to a familar location, closing the loop. If we performed smoothing, this new information would propagate backwards in time to disambiguate the entire trajectory. (b) We show the precision matrix, representing sparse correlations between the landmarks, and between the landmarks and the robot's position (pose). This sparse precision matrix can be visualized as a Gaussian graphical model, as shown. Source: Figure 15.A.3 of (Koller and Friedman 2009) . Used with kind permission of Daphne Koller.

mapping, or **SLAM** for short, and is widely used in mobile robotics, as well as other applications such as indoor navigation using cellphones (since GPS does not work inside buildings).

Let us assume we can represent the map as the 2d locations of a fixed set of K landmarks, denote them by $L^1, \ldots, L^K$ (each is a vector in $\mathbb{R}^2$). For simplicity, we will assume these are uniquely identifiable. Let $\mathbf{x}_t$ represent the unknown location of the robot at time t. We define the state space to be $\mathbf{z}_t = (\mathbf{x}_t, \mathbf{L}^{1:K})$; we assume the landmarks are static, so their motion model is a constant, and they have no system noise. If $\mathbf{y}_t$ measures the distance from $\mathbf{x}_t$ to the set of closest landmarks, then the robot can update its estimate of the landmark locations based on what it sees. Figure 18.2 shows the corresponding graphical model for the case where $K = 2$, and where on the first step it sees landmarks 1 and 2, then just landmark 2, then just landmark 1, etc.

If we assume the observation model $p(\mathbf{y}_t|\mathbf{z}_t, \mathbf{L})$ is linear-Gaussian, and we use a Gaussian motion model for $p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t)$, we can use a Kalman filter to maintain our belief state about the location of the robot and the location of the landmarks (Smith and Cheeseman 1986; Choset and Nagatani 2001).

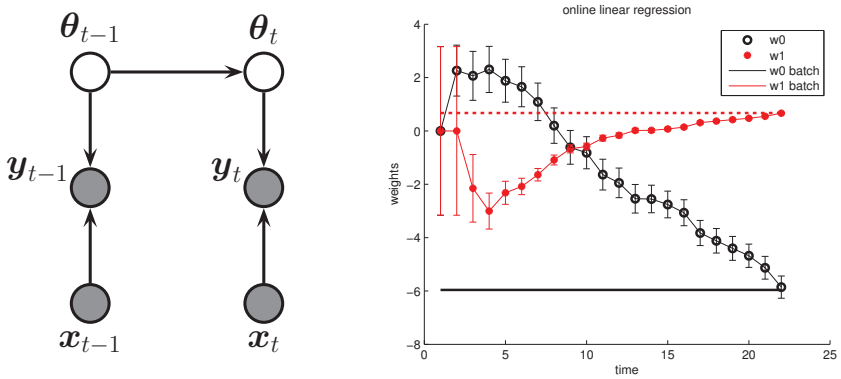
Over time, the uncertainty in the robot's location will increase, due to wheel slippage etc., but when the robot returns to a familiar location, its uncertainty will decrease again. This is called **closing the loop**, and is illustrated in Figure 18.3(a), where we see the uncertainty ellipses, representing $\mathrm{cov}\left[\mathbf{x}_t|\mathbf{y}_{1:t}, \mathbf{u}_{1:t}\right]$, grow and then shrink. (Note that in this section, we assume that a human is joysticking the robot through the environment, so $\mathbf{u}_{1:t}$ is given as input, i.e., we do not address the decision-theoretic issue of choosing where to explore.)

Since the belief state is Gaussian, we can visualize the posterior covariance matrix $\mathbf{\Sigma}_t$. Actually, it is more interesting to visualize the posterior precision matrix, $\mathbf{\Lambda}_t = \mathbf{\Sigma}_t^{-1}$, since that is fairly sparse, as shown in Figure 18.3(b). The reason for this is that zeros in the precision matrix correspond to absent edges in the corresponding undirected Gaussian graphical model (see Section 19.4.4). Initially all the landmarks are uncorrelated (assuming we have a diagonal prior on $\mathbf{L}$), so the GGM is a disconnected graph, and $\mathbf{\Lambda}_t$ is diagonal. However, as the robot moves about, it will induce correlation between nearby landmarks. Intuitively this is because the robot is estimating its position based on distance to the landmarks, but the landmarks' locations are being estimated based on the robot's position, so they all become inter-dependent. This can be seen more clearly from the graphical model in Figure 18.2: it is clear that L^1 and L^2 are not d-separated by $\mathbf{y}_{1:t}$, because there is a path between them via the unknown sequence of $\mathbf{x}_{1:t}$ nodes. As a consequence of the precision matrix becoming denser, exact inference takes $O(K^3)$ time. (This is an example of the entanglement problem for inference in DBNs.) This prevents the method from being applied to large maps.

There are two main solutions to this problem. The first is to notice that the correlation pattern moves along with the location of the robot (see Figure 18.3(b)). The remaining correlations become weaker over time. Consequently we can dynamically "prune out" weak edges from the GGM using a technique called the thin junction tree filter (Paskin 2003) (junction trees are explained in Section 20.4).

A second approach is to notice that, conditional on knowing the robot's path, $\mathbf{x}_{1:t}$, the landmark locations are independent. That is, $p(\mathbf{L}|\mathbf{x}_{1:t}, \mathbf{y}_{1:t}) = \prod_{k=1}^{K} p(\mathbf{L}^k|\mathbf{x}_{1:t}, \mathbf{y}_{1:t})$. This forms the basis of a method known as FastSLAM, which combines Kalman filtering and particle filtering, as discussed in Section 23.6.3.

(Thrun et al. 2006) provides a more detailed account of SLAM and mobile robotics.



(a) (b)

Figure 18.4 (a) A dynamic generalization of linear regression. (b) Illustration of the recursive least squares algorithm applied to the model $p(y|\mathbf{x}, \boldsymbol{\theta}) = \mathcal{N}(y|w_0 + w_1 x, \sigma^2)$. We plot the marginal posterior of w_0 and w_1 vs number of data points. (Error bars represent $\mathbb{E}[w_j|y_{1:t}] \pm \sqrt{\mathrm{var}[w_j|y_{1:t}]}$.) After seeing all the data, we converge to the offline ML (least squares) solution, represented by the horizontal lines. Figure generated by `linregOnlineDemoKalman`.

18.2.3 Online parameter learning using recursive least squares

We can perform online Bayesian inference for the parameters of various statistical models using SSMs. In this section, we focus on linear regression; in Section 18.5.3.2, we discuss logistic regression.

The basic idea is to let the hidden state represent the regression parameters, and to let the (time-varying) observation model represent the current data vector. In more detail, define the prior to be $p(\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{\theta}|\boldsymbol{\theta}_0, \boldsymbol{\Sigma}_0)$. (If we want to do online ML estimation, we can just set $\boldsymbol{\Sigma}_0 = \infty\mathbf{I}$.) Let the hidden state be $\mathbf{z}_t = \boldsymbol{\theta}$; if we assume the regression parameters do not change, we can set $\mathbf{A}_t = \mathbf{I}$ and $\mathbf{Q}_t = 0\mathbf{I}$, so

$$p(\boldsymbol{\theta}_t|\boldsymbol{\theta}_{t-1}) = \mathcal{N}(\boldsymbol{\theta}_t|\boldsymbol{\theta}_{t-1}, 0\mathbf{I}) = \delta_{\boldsymbol{\theta}_{t-1}}(\boldsymbol{\theta}_t) \tag{18.12}$$

(If we do let the parameters change over time, we get a so-called **dynamic linear model** (Harvey 1990; West and Harrison 1997; Petris et al. 2009).) Let $\mathbf{C}_t = \mathbf{x}_t^T$, and $\mathbf{R}_t = \sigma^2$, so the (non-stationary) observation model has the form

$$\mathcal{N}(\mathbf{y}_t|\mathbf{C}_t\mathbf{z}_t, \mathbf{R}_t) = \mathcal{N}(\mathbf{y}_t|\mathbf{x}_t^T\boldsymbol{\theta}_t, \sigma^2) \tag{18.13}$$

Applying the Kalman filter to this model provides a way to update our posterior beliefs about the parameters as the data streams in. This is known as the **recursive least squares** or **RLS** algorithm.

We can derive an explicit form for the updates as follows. In Section 18.3.1, we show that the Kalman update for the posterior mean has the form

$$\boldsymbol{\mu}_t = \mathbf{A}_t\boldsymbol{\mu}_{t-1} + \mathbf{K}_t(\mathbf{y}_t - \mathbf{C}_t\mathbf{A}_t\boldsymbol{\mu}_{t-1}) \tag{18.14}$$

where $\mathbf{K}_t$ is known as the Kalman gain matrix. Based on Equation 18.39, one can show that $\mathbf{K}_t = \boldsymbol{\Sigma}_t \mathbf{C}_t^T \mathbf{R}_t^{-1}$. In this context, we have $\mathbf{K}_t = \boldsymbol{\Sigma}_t \mathbf{x}_t / \sigma^2$. Hence the update for the parameters becomes

$$\hat{\boldsymbol{\theta}}_t = \hat{\boldsymbol{\theta}}_{t-1} + \frac{1}{\sigma^2} \boldsymbol{\Sigma}_{t|t} (y_t - \mathbf{x}_t^T \hat{\boldsymbol{\theta}}_{t-1}) \mathbf{x}_t \tag{18.15}$$

If we approximate $\frac{1}{\sigma^2}\boldsymbol{\Sigma}_{t|t-1}$ with $\eta_t \mathbf{I}$, we recover the **least mean squares** or **LMS** algorithm, discussed in Section 8.5.3. In LMS, we need to specify how to adapt the update parameter η_t to ensure convergence to the MLE. Furthermore, the algorithm may take multiple passes through the data. By contrast, the RLS algorithm automatically performs step-size adaptation, and converges to the optimal posterior in one pass over the data. See Figure 18.4 for an example.

18.2.4 SSM for time series forecasting *

SSMs are very well suited for time-series forecasting, as we explain below. We focus on the case of scalar (one dimensional) time series, for simplicity. Our presentation is based on (Varian 2011). See also (Aoki 1987; Harvey 1990; West and Harrison 1997; Durbin and Koopman 2001; Petris et al. 2009; Prado and West 2010) for good books on this topic.

At first sight, it might not be apparent why SSMs are useful, since the goal in forecasting is to predict future visible variables, not to estimate hidden states of some system. Indeed, most classical methods for time series forecasting are just functions of the form $\hat{y}_{t+1} = f(\mathbf{y}_{1:t}, \boldsymbol{\theta})$, where hidden variables play no role (see Section 18.2.4.4). The idea in the state-space approach to time series is to create a generative model of the data in terms of latent processes, which capture different aspects of the signal. We can then integrate out the hidden variables to compute the posterior predictive of the visibles.

Since the model is linear-Gaussian, we can just add these processes together to explain the observed data. This is called a **structural time series** model. Below we explain some of the basic building blocks.

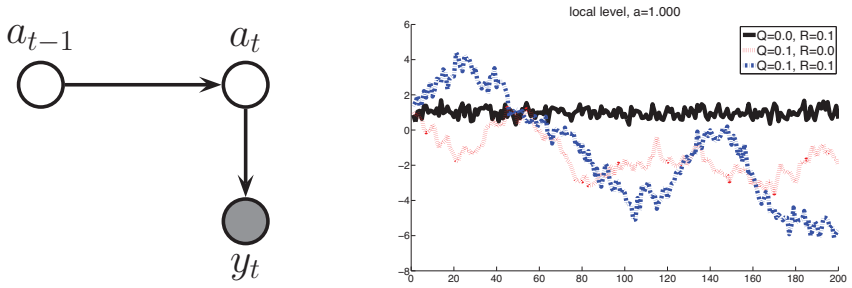
18.2.4.1 Local level model

The simplest latent process is known as the **local level model**, which has the form

$$y_t = a_t + \epsilon_t^y, \quad \epsilon_t^y \sim \mathcal{N}(0, R) \tag{18.16}$$

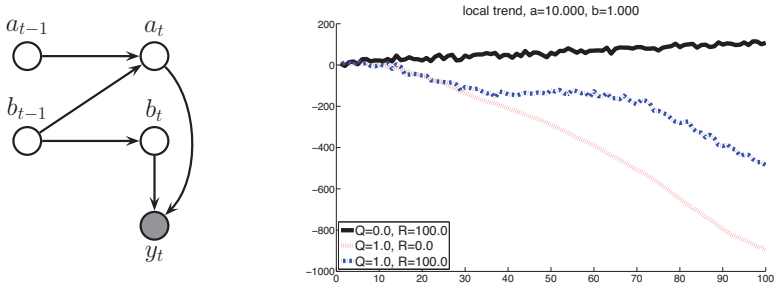
$$a_t = a_{t-1} + \epsilon_t^a, \quad \epsilon_t^a \sim \mathcal{N}(0, Q) \tag{18.17}$$

where the hidden state is just $\mathbf{z}_t = a_t$. This model asserts that the observed data $y_t \in \mathbb{R}$ is equal to some unknown level term $a_t \in \mathbb{R}$, plus observation noise with variance R. In addition, the level a_t evolves over time subject to system noise with variance Q. See Figure 18.5 for some examples.



(a) (b)

Figure 18.5 (a) Local level model. (b) Sample output, for $a_0 = 10$. Black solid line: $Q = 0$, $R = 1$ (deterministic system, noisy observations). Red dotted line: $Q = 0.1$, $R = 0$ (noisy system, deterministic observation). Blue dot-dash line: $Q = 0.1$, $R = 1$ (noisy system and observations). Figure generated by `ssmTimeSeriesSimple`.



(a) (b)

Figure 18.6 (a) Local Trend. (b) Sample output, for $a_0 = 10$, $b_0 = 1$. Color code as in Figure 18.5. Figure generated by `ssmTimeSeriesSimple`.

18.2.4.2 Local linear trend

Many time series exhibit linear trends upwards or downwards, at least locally. We can model this by letting the level a_t change by an amount b_t at each step as follows:

$$y_t = a_t + \epsilon_t^y, \quad \epsilon_t^y \sim \mathcal{N}(0, R) \tag{18.18}$$

$$a_t = a_{t-1} + b_{t-1} + \epsilon_t^a, \quad \epsilon_t^a \sim \mathcal{N}(0, Q_a) \tag{18.19}$$

$$b_t = b_{t-1} + \epsilon_t^b, \quad \epsilon_t^b \sim \mathcal{N}(0, Q_b) \tag{18.20}$$

See Figure 18.6(a). We can write this in standard form by defining $\mathbf{z}_t = (a_t, b_t)$ and

$$\mathbf{A} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}, \ \mathbf{C} = \begin{pmatrix} 1 & 0 \end{pmatrix}, \ \mathbf{Q} = \begin{pmatrix} Q_a & 0 \\ 0 & Q_b \end{pmatrix} \tag{18.21}$$

When $Q_b = 0$, we have $b_t = b_0$, which is some constant defining the slope of the line. If in addition we have $Q_a = 0$, we have $a_t = a_{t-1} + b_0 t$. Unrolling this, we have $a_t = a_0 + b_0 t$, and

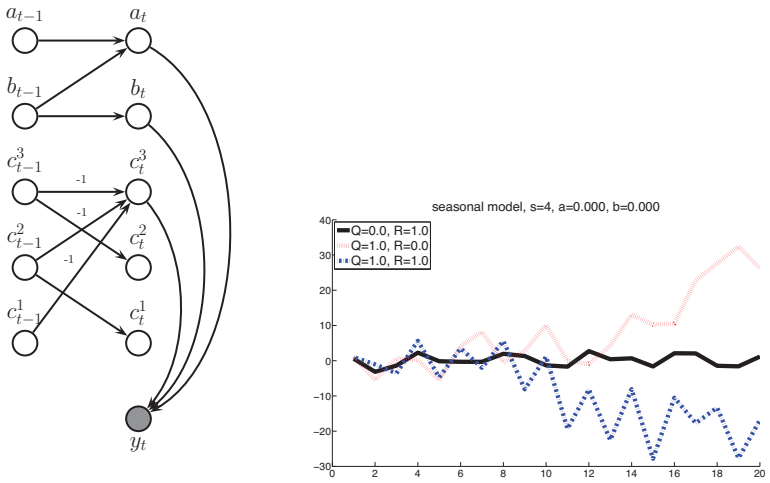


Figure 18.7 (a) Seasonal model. (b) Sample output, for $a_0 = b_0 = 0$, $\mathbf{c}_0 = (1, 1, 1)$, with a period of 4. Color code as in Figure 18.5. Figure generated by `ssmTimeSeriesSimple`.

hence $\mathbb{E}[y_t|\mathbf{y}_{1:t-1}] = a_0 + tb_0$. This is thus a generalization of the classic constant linear trend model, an example of which is shown in the black line of Figure 18.6(b).

18.2.4.3 Seasonality

Many time series fluctuate periodically, as illustrated in Figure 18.7(b). This can be modeled by adding a latent process consisting of a series offset terms, c_t, which sum to zero (on average) over a complete cycle of S steps:

$$c_t = -\sum_{s=1}^{S-1} c_{t-s} + \epsilon_t^c, \;\; \epsilon_t^c \sim \mathcal{N}(0, Q_c) \tag{18.22}$$

See Figure 18.7(a) for the graphical model for the case $S = 4$ (we only need 3 seasonal variable because of the sum-to-zero constraint). Writing this in standard LG-SSM form is left to Exercise 18.2.

18.2.4.4 ARMA models *

The classical approach to time-series forecasting is based on **ARMA** models. "ARMA" stands for auto-regressive moving-average, and refers to a model of the form

$$x_t = \sum_{i=1}^{p} \alpha_i x_{t-i} + \sum_{j=1}^{q} \beta_j w_{t-j} + v_t \tag{18.23}$$

where $v_t, w_t \sim \mathcal{N}(0, 1)$ are independent Gaussian noise terms. If $q = 0$, we have a pure AR model, where $x_t \perp x_i | x_{t-1:t-p}$, for $i < t - p$. For example, if $p = 1$, we have the AR(1) model

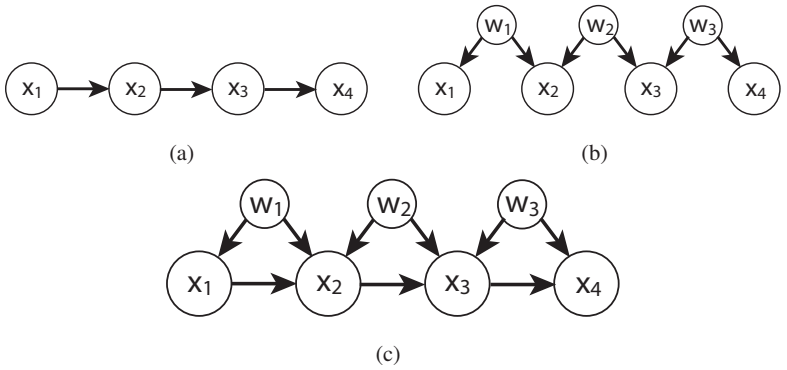


Figure 18.8 (a) An AR(1) model. (b) An MA(1) model represented as a bi-directed graph. (c) An ARMA(1,1) model. Source: Figure 5.14 of (Choi 2011). Used with kind permission of Myung Choi.

shown in Figure 18.8(a). (The v_t nodes are implicit in the Gaussian CPD for x_t.) This is just a first-order Markov chain. If $p = 0$, we have a pure MA model, where $x_t \perp x_i$, for $i < t - q$. For example, if $q = 1$, we have the MA(1) model shown in Figure 18.8(b). Here the w_t nodes are hidden common causes, which induces dependencies between adjacent time steps. This models short-range correlation. If $p = q = 1$, we get the ARMA(1,1) model shown in Figure 18.8(c), which captures correlation at short and long time scales.

It turns out that ARMA models can be represented as SSMs, as explained in (Aoki 1987; Harvey 1990; West and Harrison 1997; Durbin and Koopman 2001; Petris et al. 2009; Prado and West 2010). However, the structural approach to time series is often easier to understand than the ARMA approach. In addition, it allows the parameters to evolve over time, which makes the models more adaptive to non-stationarity.

18.3 Inference in LG-SSM

In this section, we discuss exact inference in LG-SSM models. We first consider the online case, which is analogous to the forwards algorithm for HMMs. We then consider the offline case, which is analogous to the forwards-backwards algorithm for HMMs.

18.3.1 The Kalman filtering algorithm

The **Kalman filter** is an algorithm for exact Bayesian filtering for linear-Gaussian state space models. We will represent the marginal posterior at time t by

$$p(\mathbf{z}_t|\mathbf{y}_{1:t}, \mathbf{u}_{1:t}) = \mathcal{N}(\mathbf{z}_t|\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t) \tag{18.24}$$

Since everything is Gaussian, we can perform the prediction and update steps in closed form, as we explain below. The resulting algorithm is the Gaussian analog of the HMM filter in Section 17.4.2.

18.3.1.1 Prediction step

The prediction step is straightforward to derive:

$$p(\mathbf{z}_t|\mathbf{y}_{1:t-1},\mathbf{u}_{1:t}) = \int \mathcal{N}(\mathbf{z}_t|\mathbf{A}_t\mathbf{z}_{t-1}+\mathbf{B}_t\mathbf{u}_t,\mathbf{Q}_t)\mathcal{N}(\mathbf{z}_{t-1}|\boldsymbol{\mu}_{t-1},\boldsymbol{\Sigma}_{t-1})d\mathbf{z}_{t-1} \quad (18.25)$$

$$= \mathcal{N}(\mathbf{z}_t|\boldsymbol{\mu}_{t|t-1},\boldsymbol{\Sigma}_{t|t-1}) \quad (18.26)$$

$$\boldsymbol{\mu}_{t|t-1} \triangleq \mathbf{A}_t\boldsymbol{\mu}_{t-1}+\mathbf{B}_t\mathbf{u}_t \quad (18.27)$$

$$\boldsymbol{\Sigma}_{t|t-1} \triangleq \mathbf{A}_t\boldsymbol{\Sigma}_{t-1}\mathbf{A}_t^T+\mathbf{Q}_t \quad (18.28)$$

18.3.1.2 Measurement step

The measurement step can be computed using Bayes rule, as follows

$$p(\mathbf{z}_t|\mathbf{y}_t,\mathbf{y}_{1:t-1},\mathbf{u}_{1:t}) \propto p(\mathbf{y}_t|\mathbf{z}_t,\mathbf{u}_t)p(\mathbf{z}_t|\mathbf{y}_{1:t-1},\mathbf{u}_{1:t}) \quad (18.29)$$

In Section 18.3.1.6, we show that this is given by

$$p(\mathbf{z}_t|\mathbf{y}_{1:t},\mathbf{u}_t) = \mathcal{N}(\mathbf{z}_t|\boldsymbol{\mu}_t,\boldsymbol{\Sigma}_t) \quad (18.30)$$

$$\boldsymbol{\mu}_t = \boldsymbol{\mu}_{t|t-1}+\mathbf{K}_t\mathbf{r}_t \quad (18.31)$$

$$\boldsymbol{\Sigma}_t = (\mathbf{I}-\mathbf{K}_t\mathbf{C}_t)\boldsymbol{\Sigma}_{t|t-1} \quad (18.32)$$

where $\mathbf{r}_t$ is the **residual** or **innovation**, given by the difference between our predicted observation and the actual observation:

$$\mathbf{r}_t \triangleq \mathbf{y}_t-\hat{\mathbf{y}}_t \quad (18.33)$$

$$\hat{\mathbf{y}}_t \triangleq \mathbb{E}\left[\mathbf{y}_t|\mathbf{y}_{1:t-1},\mathbf{u}_{1:t}\right] = \mathbf{C}_t\boldsymbol{\mu}_{t|t-1}+\mathbf{D}_t\mathbf{u}_t \quad (18.34)$$

and $\mathbf{K}_t$ is the **Kalman gain matrix**, given by

$$\mathbf{K}_t \triangleq \boldsymbol{\Sigma}_{t|t-1}\mathbf{C}_t^T\mathbf{S}_t^{-1} \quad (18.35)$$

where

$$\mathbf{S}_t \triangleq \mathrm{cov}\left[\mathbf{r}_t|\mathbf{y}_{1:t-1},\mathbf{u}_{1:t}\right] \quad (18.36)$$

$$= \mathbb{E}\left[(\mathbf{C}_t\mathbf{z}_t+\boldsymbol{\delta}_t-\hat{\mathbf{y}}_t)(\mathbf{C}_t\mathbf{z}_t+\boldsymbol{\delta}_t-\hat{\mathbf{y}}_t)^T|\mathbf{y}_{1:t-1},\mathbf{u}_{1:t}\right] \quad (18.37)$$

$$= \mathbf{C}_t\boldsymbol{\Sigma}_{t|t-1}\mathbf{C}_t^T+\mathbf{R}_t \quad (18.38)$$

where $\boldsymbol{\delta}_t \sim \mathcal{N}(\mathbf{0},\mathbf{R}_t)$ is an observation noise term which is independent of all other noise sources. Note that by using the matrix inversion lemma, the Kalman gain matrix can also be written as

$$\mathbf{K}_t = \boldsymbol{\Sigma}_{t|t-1}\mathbf{C}^T(\mathbf{C}\boldsymbol{\Sigma}_{t|t-1}\mathbf{C}^T+\mathbf{R})^{-1} = (\boldsymbol{\Sigma}_{t|t-1}^{-1}+\mathbf{C}^T\mathbf{R}\mathbf{C})^{-1}\mathbf{C}^T\mathbf{R}^{-1} \quad (18.39)$$

We now have all the quantities we need to implement the algorithm; see `kalmanFilter` for some Matlab code.

Let us try to make sense of these equations. In particular, consider the equation for the mean update: $\boldsymbol{\mu}_t = \boldsymbol{\mu}_{t|t-1}+\mathbf{K}_t\mathbf{r}_t$. This says that the new mean is the old mean plus a

correction factor, which is $\mathbf{K}_t$ times the error signal $\mathbf{r}_t$. The amount of weight placed on the error signal depends on the Kalman gain matrix. If $\mathbf{C}_t = \mathbf{I}$, then $\mathbf{K}_t = \boldsymbol{\Sigma}_{t|t-1}\mathbf{S}_t^{-1}$, which is the ratio between the covariance of the prior (from the dynamic model) and the covariance of the measurement error. If we have a strong prior and/or very noisy sensors, $|\mathbf{K}_t|$ will be small, and we will place little weight on the correction term. Conversely, if we have a weak prior and/or high precision sensors, then $|\mathbf{K}_t|$ will be large, and we will place a lot of weight on the correction term.

18.3.1.3 Marginal likelihood

As a byproduct of the algorithm, we can also compute the log-likelihood of the sequence using

$$\log p(\mathbf{y}_{1:T}|\mathbf{u}_{1:T}) = \sum_t \log p(\mathbf{y}_t|\mathbf{y}_{1:t-1}, \mathbf{u}_{1:t}) \tag{18.40}$$

where

$$p(\mathbf{y}_t|\mathbf{y}_{1:t-1}, \mathbf{u}_{1:t}) = \mathcal{N}(\mathbf{y}_t|\mathbf{C}_t\boldsymbol{\mu}_{t|t-1}, \mathbf{S}_t) \tag{18.41}$$

18.3.1.4 Posterior predictive

The one-step-ahead posterior predictive density for the observations can be computed as follows

$$p(\mathbf{y}_t|\mathbf{y}_{1:t-1}, \mathbf{u}_{1:t}) = \int \mathcal{N}(\mathbf{y}_t|\mathbf{C}\mathbf{z}_t, \mathbf{R})\mathcal{N}(\mathbf{z}_t|\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1})d\mathbf{z}_t \tag{18.42}$$

$$= \mathcal{N}(\mathbf{y}_t|\mathbf{C}\boldsymbol{\mu}_{t|t-1}, \mathbf{C}\boldsymbol{\Sigma}_{t|t-1}\mathbf{C}^T + \mathbf{R}) \tag{18.43}$$

This is useful for time series forecasting.

18.3.1.5 Computational issues

There are two dominant costs in the Kalman filter: the matrix inversion to compute the Kalman gain matrix, $\mathbf{K}_t$, which takes $O(|\mathbf{y}_t|^3)$ time; and the matrix-matrix multiply to compute $\boldsymbol{\Sigma}_t$, which takes $O(|\mathbf{z}_t|^2)$ time. In some applications (e.g., robotic mapping), we have $|\mathbf{z}_t| \gg |\mathbf{y}_t|$, so the latter cost dominates. However, in such cases, we can sometimes use sparse approximations (see (Thrun et al. 2006)).

In cases where $|\mathbf{y}_t| \gg |\mathbf{z}_t|$, we can precompute $\mathbf{K}_t$, since, suprisingly, it does not depend on the actual observations $\mathbf{y}_{1:t}$ (an unusual property that is specific to linear Gaussian systems). The iterative equations for updating $\boldsymbol{\Sigma}_t$ are called the **Ricatti equations**, and for time invariant systems (i.e., where $\boldsymbol{\theta}_t = \boldsymbol{\theta}$), they converge to a fixed point. This steady state solution can then be used instead of using a time-specific gain matrix.

In practice, more sophisticated implementations of the Kalman filter should be used, for reasons of numerical stability. One approach is the **information filter**, which recursively updates the canonical parameters of the Gaussian, $\boldsymbol{\Lambda}_t = \boldsymbol{\Sigma}_t^{-1}$ and $\boldsymbol{\eta}_t = \boldsymbol{\Lambda}_t\boldsymbol{\mu}_t$, instead of the moment parameters. Another approach is the **square root filter**, which works with the Cholesky decomposition or the $\mathbf{U}_t\mathbf{D}_t\mathbf{U}_t$ decomposition of $\boldsymbol{\Sigma}_t$. This is much more numerically stable than directly updating $\boldsymbol{\Sigma}_t$. Further details can be found at `http://www.cs.unc.edu/~welch/kalman/` and in various books, such as (Simon 2006).

18.3.1.6 Derivation *

We now derive the Kalman filter equations. For notational simplicity, we will ignore the input terms $\mathbf{u}_{1:t}$. From Bayes rule for Gaussians (Equation 4.125), we have that the posterior precision is given by

$$\boldsymbol{\Sigma}_t^{-1} = \boldsymbol{\Sigma}_{t|t-1}^{-1} + \mathbf{C}_t^T \mathbf{R}_t^{-1} \mathbf{C}_t \tag{18.44}$$

From the matrix inversion lemma (Equation 4.106) we can rewrite this as

$$\boldsymbol{\Sigma}_t = \boldsymbol{\Sigma}_{t|t-1} - \boldsymbol{\Sigma}_{t|t-1}\mathbf{C}_t^T(\mathbf{R}_t + \mathbf{C}_t\boldsymbol{\Sigma}_{t|t-1}\mathbf{C}_t^T)^{-1}\mathbf{C}_t\boldsymbol{\Sigma}_{t|t-1} \tag{18.45}$$

$$= (\mathbf{I} - \mathbf{K}_t\mathbf{C}_t)\boldsymbol{\Sigma}_{t|t-1} \tag{18.46}$$

From Bayes rule for Gaussians (Equation 4.125), the posterior mean is given by

$$\boldsymbol{\mu}_t = \boldsymbol{\Sigma}_t\mathbf{C}_t\mathbf{R}_t^{-1}\mathbf{y}_t + \boldsymbol{\Sigma}_t\boldsymbol{\Sigma}_{t|t-1}^{-1}\boldsymbol{\mu}_{t|t-1} \tag{18.47}$$

We will now massage this into the form stated earlier. Applying the second matrix inversion lemma (Equation 4.107) to the first term of Equation 18.47 we have

$$\boldsymbol{\Sigma}_t\mathbf{C}_t\mathbf{R}_t^{-1}\mathbf{y}_t = (\boldsymbol{\Sigma}_{t|t-1}^{-1} + \mathbf{C}_t^T\mathbf{R}_t^{-1}\mathbf{C}_t)^{-1}\mathbf{C}_t\mathbf{R}_t^{-1}\mathbf{y}_t \tag{18.48}$$

$$= \boldsymbol{\Sigma}_{t|t-1}\mathbf{C}_t^T(\mathbf{R}_t + \mathbf{C}_t\boldsymbol{\Sigma}_{t|t-1}\mathbf{C}_t^T)^{-1}\mathbf{y}_t = \mathbf{K}_t\mathbf{y}_t \tag{18.49}$$

Now applying the matrix inversion lemma (Equation 4.106) to the second term of Equation 18.47 we have

$$\boldsymbol{\Sigma}_t\boldsymbol{\Sigma}_{t|t-1}^{-1}\boldsymbol{\mu}_{t|t-1} \tag{18.50}$$

$$= (\boldsymbol{\Sigma}_{t|t-1}^{-1} + \mathbf{C}_t^T\mathbf{R}_t^{-1}\mathbf{C}_t)^{-1}\boldsymbol{\Sigma}_{t|t-1}^{-1}\boldsymbol{\mu}_{t|t-1} \tag{18.51}$$

$$= \left[\boldsymbol{\Sigma}_{t|t-1} - \boldsymbol{\Sigma}_{t|t-1}\mathbf{C}^T(\mathbf{R}_t + \mathbf{C}_t^T\boldsymbol{\Sigma}_{t|t-1}\mathbf{C}_t^T)\mathbf{C}_t\boldsymbol{\Sigma}_{t|t-1}\right]\boldsymbol{\Sigma}_{t|t-1}^{-1}\boldsymbol{\mu}_{t|t-1} \tag{18.52}$$

$$= (\boldsymbol{\Sigma}_{t|t-1} - \mathbf{K}_t\mathbf{C}_t^T\boldsymbol{\Sigma}_{t|t-1})\boldsymbol{\Sigma}_{t|t-1}^{-1}\boldsymbol{\mu}_{t|t-1} \tag{18.53}$$

$$= \boldsymbol{\mu}_{t|t-1} - \mathbf{K}_t\mathbf{C}_t^T\boldsymbol{\mu}_{t|t-1} \tag{18.54}$$

Putting the two together we get

$$\boldsymbol{\mu}_t = \boldsymbol{\mu}_{t|t-1} + \mathbf{K}_t(\mathbf{y}_t - \mathbf{C}_t\boldsymbol{\mu}_{t|t-1}) \tag{18.55}$$

18.3.2 The Kalman smoothing algorithm

In Section 18.3.1, we described the Kalman filter, which sequentially computes $p(\mathbf{z}_t|\mathbf{y}_{1:t})$ for each t. This is useful for online inference problems, such as tracking. However, in an offline setting, we can wait until all the data has arrived, and then compute $p(\mathbf{z}_t|\mathbf{y}_{1:T})$. By conditioning on past and future data, our uncertainty will be significantly reduced. This is illustrated in Figure 18.1(c), where we see that the posterior covariance ellipsoids are smaller for the smoothed trajectory than for the filtered trajectory. (The ellipsoids are larger at the beginning and end of the trajectory, since states near the boundary do not have as many useful neighbors from which to borrow information.)

We now explain how to compute the smoothed estimates, using an algorithm called the **RTS smoother**, named after its inventors, Rauch, Tung and Striebel (Rauch et al. 1965). It is also known as the **Kalman smoothing** algorithm. The algorithm is analogous to the forwards-backwards algorithm for HMMs, although there are some small differences which we discuss below.

18.3.2.1 Algorithm

Kalman filtering can be regarded as message passing on a graph, from left to right. When the messages have reached the end of the graph, we have successfully computed $p(\mathbf{z}_T|\mathbf{y}_{1:T})$. Now we work backwards, from right to left, sending information from the future back to the past, and them combining the two information sources. The question is: how do we compute these backwards equations? We first give the equations, then the derivation.

We have

$$
\begin{aligned}
p(\mathbf{z}_t|\mathbf{y}_{1:T}) &= \mathcal{N}(\boldsymbol{\mu}_{t|T}, \boldsymbol{\Sigma}_{t|T}) && (18.56)\\
\boldsymbol{\mu}_{t|T} &= \boldsymbol{\mu}_{t|t} + \mathbf{J}_t(\boldsymbol{\mu}_{t+1|T} - \boldsymbol{\mu}_{t+1|t}) && (18.57)\\
\boldsymbol{\Sigma}_{t|T} &= \boldsymbol{\Sigma}_{t|t} + \mathbf{J}_t(\boldsymbol{\Sigma}_{t+1|T} - \boldsymbol{\Sigma}_{t+1|t})\mathbf{J}_t^T && (18.58)\\
\mathbf{J}_t &\triangleq \boldsymbol{\Sigma}_{t|t}\mathbf{A}_{t+1}^T\boldsymbol{\Sigma}_{t+1|t}^{-1} && (18.59)
\end{aligned}
$$

where $\mathbf{J}_t$ is the backwards Kalman gain matrix. The algorithm can be initialized from $\boldsymbol{\mu}_{T|T}$ and $\boldsymbol{\Sigma}_{T|T}$ from the Kalman filter. Note that this backwards pass does not need access to the data, that is, it does not need $\mathbf{y}_{1:T}$. This allows us to "throw away" potentially high dimensional observation vectors, and just keep the filtered belief states, which usually requires less memory.

18.3.2.2 Derivation *

We now derive the Kalman smoother, following the presentation of (Jordan 2007, sec 15.7).

The key idea is to leverage the Markov property, which says that $\mathbf{z}_t$ is independent of future data, $\mathbf{y}_{t+1:T}$, as long as $\mathbf{z}_{t+1}$ is known. Of course, $\mathbf{z}_{t+1}$ is not known, but we have a distribution over it. So we condition on $\mathbf{z}_{t+1}$ and then integrate it out, as follows.

$$
\begin{aligned}
p(\mathbf{z}_t|\mathbf{y}_{1:T}) &= \int p(\mathbf{z}_t|\mathbf{y}_{1:T}, \mathbf{z}_{t+1})p(\mathbf{z}_{t+1}|\mathbf{y}_{1:T})d\mathbf{z}_{t+1} && (18.60)\\
&= \int p(\mathbf{z}_t|\mathbf{y}_{1:t}, \cancel{\mathbf{y}_{t+1:T}}, \mathbf{z}_{t+1})p(\mathbf{z}_{t+1}|\mathbf{y}_{1:T})d\mathbf{z}_{t+1} && (18.61)
\end{aligned}
$$

By induction, assume we have already computed the smoothed distribution for $t+1$:

$$p(\mathbf{z}_{t+1}|\mathbf{y}_{1:T}) = \mathcal{N}(\mathbf{z}_{t+1}|\boldsymbol{\mu}_{t+1|T}, \boldsymbol{\Sigma}_{t+1|T}) \quad (18.62)$$

The question is: how do we perform the integration?

First, we compute the filtered two-slice distribution $p(\mathbf{z}_t, \mathbf{z}_{t+1}|\mathbf{y}_{1:t})$ as follows:

$$p(\mathbf{z}_t, \mathbf{z}_{t+1}|\mathbf{y}_{1:t}) = \mathcal{N}\left(\begin{pmatrix}\mathbf{z}_t\\ \mathbf{z}_{t+1}\end{pmatrix} | \begin{pmatrix}\boldsymbol{\mu}_{t|t}\\ \boldsymbol{\mu}_{t+1|t}\end{pmatrix} \begin{pmatrix}\boldsymbol{\Sigma}_{t|t} & \boldsymbol{\Sigma}_{t|t}\mathbf{A}_{t+1}^T\\ \mathbf{A}_{t+1}\boldsymbol{\Sigma}_{t|t} & \boldsymbol{\Sigma}_{t+1|t}\end{pmatrix}\right) \quad (18.63)$$

Now we use Gaussian conditioning to compute $p(\mathbf{z}_t|\mathbf{z}_{t+1}, \mathbf{y}_{1:t})$ as follows:

$$p(\mathbf{z}_t|\mathbf{z}_{t+1}, \mathbf{y}_{1:t}) = \mathcal{N}(\mathbf{z}_t|\boldsymbol{\mu}_{t|t} + \mathbf{J}_t(\mathbf{z}_{t+1} - \boldsymbol{\mu}_{t+1|t}), \boldsymbol{\Sigma}_{t|t} - \mathbf{J}_t\boldsymbol{\Sigma}_{t+1|t}\mathbf{J}_t^T) \tag{18.64}$$

We can compute the smoothed distribution for t using the rules of iterated expectation and iterated covariance. First, the mean:

$$\begin{aligned}
\boldsymbol{\mu}_{t|T} &= \mathbb{E}\left[\mathbb{E}\left[\mathbf{z}_t|\mathbf{z}_{t+1}, \mathbf{y}_{1:T}\right]\middle|\mathbf{y}_{1:T}\right] && (18.65)\\
&= \mathbb{E}\left[\mathbb{E}\left[\mathbf{z}_t|\mathbf{z}_{t+1}, \mathbf{y}_{1:t}\right]\middle|\mathbf{y}_{1:T}\right] && (18.66)\\
&= \mathbb{E}\left[\boldsymbol{\mu}_{t|t} + \mathbf{J}_t(\mathbf{z}_{t+1} - \boldsymbol{\mu}_{t+1|t})\middle|\mathbf{y}_{1:T}\right] && (18.67)\\
&= \boldsymbol{\mu}_{t|t} + \mathbf{J}_t(\boldsymbol{\mu}_{t+1|T} - \boldsymbol{\mu}_{t+1|t}) && (18.68)
\end{aligned}$$

Now the covariance:

$$\begin{aligned}
\boldsymbol{\Sigma}_{t|T} &= \mathrm{cov}\left[\mathbb{E}\left[\mathbf{z}_t|\mathbf{z}_{t+1}, \mathbf{y}_{1:T}\right]\middle|\mathbf{y}_{1:T}\right] + \mathbb{E}\left[\mathrm{cov}\left[\mathbf{z}_t|\mathbf{z}_{t+1}, \mathbf{y}_{1:T}\right]\middle|\mathbf{y}_{1:T}\right] && (18.69)\\
&= \mathrm{cov}\left[\mathbb{E}\left[\mathbf{z}_t|\mathbf{z}_{t+1}, \mathbf{y}_{1:t}\right]\middle|\mathbf{y}_{1:T}\right] + \mathbb{E}\left[\mathrm{cov}\left[\mathbf{z}_t|\mathbf{z}_{t+1}, \mathbf{y}_{1:t}\right]\middle|\mathbf{y}_{1:T}\right] && (18.70)\\
&= \mathrm{cov}\left[\boldsymbol{\mu}_{t|t} + \mathbf{J}_t(\mathbf{z}_{t+1} - \boldsymbol{\mu}_{t+1|t})|\mathbf{y}_{1:T}\right] + \mathbb{E}\left[\boldsymbol{\Sigma}_{t|t} - \mathbf{J}_t\boldsymbol{\Sigma}_{t+1|t}\mathbf{J}_t^T|\mathbf{y}_{1:T}\right] && (18.71)\\
&= \mathbf{J}_t\mathrm{cov}\left[\mathbf{z}_{t+1} - \boldsymbol{\mu}_{t+1|t}|\mathbf{y}_{1:T}\right]\mathbf{J}_t^T + \boldsymbol{\Sigma}_{t|t} - \mathbf{J}_t\boldsymbol{\Sigma}_{t+1|t}\mathbf{J}_t^T && (18.72)\\
&= \mathbf{J}_t\boldsymbol{\Sigma}_{t+1|T}\mathbf{J}_t^T + \boldsymbol{\Sigma}_{t|t} - \mathbf{J}_t\boldsymbol{\Sigma}_{t+1|t}\mathbf{J}_t^T && (18.73)\\
&= \boldsymbol{\Sigma}_{t|t} + \mathbf{J}_t(\boldsymbol{\Sigma}_{t+1|T} - \boldsymbol{\Sigma}_{t+1|t})\mathbf{J}_t^T && (18.74)
\end{aligned}$$

The algorithm can be initialized from $\boldsymbol{\mu}_{T|T}$ and $\boldsymbol{\Sigma}_{T|T}$ from the last step of the filtering algorithm.

18.3.2.3 Comparison to the forwards-backwards algorithm for HMMs *

Note that in both the forwards and backwards passes for LDS, we always worked with normalized distributions, either conditioned on the past data or conditioned on all the data. Furthermore, the backwards pass depends on the results of the forwards pass. This is different from the usual presentation of forwards-backwards for HMMs, where the backwards pass can be computed independently of the forwards pass (see Section 17.4.3).

It turns out that we can rewrite the Kalman smoother in a modified form which makes it more similar to forwards-backwards for HMMs. In particular, we have

$$\begin{aligned}
p(\mathbf{z}_t|\mathbf{y}_{1:T}) &= \int p(\mathbf{z}_t|\mathbf{y}_{1:t}, \mathbf{z}_{t+1})p(\mathbf{z}_{t+1}|\mathbf{y}_{1:T})d\mathbf{z}_{t+1} && (18.75)\\
&= \int p(\mathbf{z}_t, \mathbf{z}_{t+1}|\mathbf{y}_{1:t})\frac{p(\mathbf{z}_{t+1}|\mathbf{y}_{1:T})}{p(\mathbf{z}_{t+1}|\mathbf{y}_{1:t})}d\mathbf{z}_{t+1} && (18.76)
\end{aligned}$$

Now

$$p(\mathbf{z}_{t+1}|\mathbf{y}_{1:T}) = \frac{p(\mathbf{y}_{t+1:T}|\mathbf{z}_{t+1}, \cancel{\mathbf{y}_{1:t}})p(\mathbf{z}_{t+1}|\mathbf{y}_{1:t})}{p(\mathbf{y}_{t+1:T}|\mathbf{y}_{1:t})} \tag{18.77}$$

so

$$\frac{p(\mathbf{z}_{t+1}|\mathbf{y}_{1:T})}{p(\mathbf{z}_{t+1}|\mathbf{y}_{1:t})} = \frac{p(\mathbf{z}_{t+1}|\mathbf{y}_{1:t})p(\mathbf{y}_{t+1:T}|\mathbf{z}_{t+1})}{p(\mathbf{z}_{t+1}|\mathbf{y}_{1:t})p(\mathbf{y}_{t+1:T}|\mathbf{y}_{1:t})} \propto p(\mathbf{y}_{t+1:T}|\mathbf{z}_{t+1}) \tag{18.78}$$

which is the conditional likelihood of the future data. This backwards message can be computed independently of the forwards message. However, this approach has several disadvantages: (1) it needs access to the original observation sequence; (2) the backwards message is a likelihood, not a posterior, so it need not to integrate to 1 over $\mathbf{z}_t$ – in fact, it may not always be possible to represent $p(\mathbf{y}_{t+1:T}|\mathbf{z}_{t+1})$ as a Gaussian with positive definite covariance (this problem does not arise in discrete state-spaces, as used in HMMs); (3) when exact inference is not possible, it makes more sense to try to approximate the smoothed distribution rather than the backwards likelihood term (see Section 22.5).

There is yet another variant, known as **two-filter smoothing**, whereby we compute $p(\mathbf{z}_t|\mathbf{y}_{1:t})$ in the forwards pass as usual, and the filtered posterior $p(\mathbf{z}_t|\mathbf{y}_{t+1:T})$ in the backwards pass. These can then be easily combined to compute $p(\mathbf{z}_t|\mathbf{y}_{1:T})$. See (Kitagawa 2004; Briers et al. 2010) for details.

18.4 Learning for LG-SSM

In this section, we briefly discuss how to estimate the parameters of an LG-SSM. In the control theory community, this is known as **systems identification** (Ljung 1987).

When using SSMs for time series forecasting, and also in some physical state estimation problems, the observation matrix $\mathbf{C}$ and the transition matrix $\mathbf{A}$ are both known and fixed, by definition of the model. In such cases, all that needs to be learned are the noise covariances $\mathbf{Q}$ and $\mathbf{R}$. (The initial state estimate $\boldsymbol{\mu}_0$ is often less important, since it will get "washed away" by the data after a few time steps. This can be encouraged by setting the initial state covariance to be large, representing a weak prior.) Although we can estimate $\mathbf{Q}$ and $\mathbf{R}$ offline, using the methods described below, it is also possible to derive a recursive procedure to exactly compute the posterior $p(\mathbf{z}_t, \mathbf{R}, \mathbf{Q}|\mathbf{y}_{1:t})$, which has the form of a Normal-inverse-Wishart; see (West and Harrison 1997; Prado and West 2010) for details.

18.4.1 Identifiability and numerical stability

In the more general setting, where the hidden states have no pre-specified meaning, we need to learn $\mathbf{A}$ and $\mathbf{C}$. However, in this case we can set $\mathbf{Q} = \mathbf{I}$ without loss of generality, since an arbitrary noise covariance can be modeled by appropriately modifying $\mathbf{A}$. Also, by analogy with factor analysis, we can require $\mathbf{R}$ to be diagonal without loss of generality. Doing this reduces the number of free parameters and improves numerical stability.

Another constraint that is useful to impose is on the eigenvalues of the dynamics matrix $\mathbf{A}$. To see why this is important, consider the case of no system noise. In this case, the hidden state at time t is given by

$$\mathbf{z}_t = \mathbf{A}^t \mathbf{z}_1 = \mathbf{U}\boldsymbol{\Lambda}^t \mathbf{U}^{-1} \mathbf{z}_1 \tag{18.79}$$

where $\mathbf{U}$ is the matrix of eigenvectors for $\mathbf{A}$, and $\boldsymbol{\Lambda} = \mathrm{diag}(\lambda_i)$ contains the eigenvalues. If any $\lambda_i > 1$, then for large t, $\mathbf{z}_t$ will blow up in magnitude. Consequently, to ensure stability, it is useful to require that all the eigenvalues are less than 1 (Siddiqi et al. 2007). Of course, if all the eigenvalues are less than 1, then $\mathbb{E}[\mathbf{z}_t] = \mathbf{0}$ for large t, so the state will return to the origin. Fortunately, when we add noise, the state become non-zero, so the model does not degenerate.

Below we discuss how to estimate the parameters. However, for simplicity of presentation, we do not impose any of the constraints mentioned above.

18.4.2 Training with fully observed data

If we observe the hidden state sequences, we can fit the model by computing the MLEs (or even the full posteriors) for the parameters by solving a multivariate linear regression problem for $\mathbf{z}_{t-1} \to \mathbf{z}_t$ and for $\mathbf{z}_t \to \mathbf{y}_t$. That is, we can estimate $\mathbf{A}$ by solving the least squares problem $J(\mathbf{A}) = \sum_{t=1}^{2}(\mathbf{z}_t - \mathbf{A}\mathbf{z}_{t-1})^2$, and similarly for $\mathbf{C}$. We can estimate the system noise covariance $\mathbf{Q}$ from the residuals in predicting $\mathbf{z}_t$ from $\mathbf{z}_{t-1}$, and estimate the observation noise covariance $\mathbf{R}$ from the residuals in predicting $\mathbf{y}_t$ from $\mathbf{z}_t$.

18.4.3 EM for LG-SSM

If we only observe the output sequence, we can compute ML or MAP estimates of the parameters using EM. The method is conceptually quite similar to the Baum-Welch algorithm for HMMs (Section 17.5), except we use Kalman smoothing instead of forwards-backwards in the E step, and use different calculations in the M step. We leave the details to Exercise 18.1.

18.4.4 Subspace methods

EM does not always give satisfactory results, because it is sensitive to the initial parameter estimates. One way to avoid this is to use a different approach known as a **subspace method** (Overschee and Moor 1996; Katayama 2005).

To understand this approach, let us initially assume there is no observation noise and no system noise. In this case, we have $\mathbf{z}_t = \mathbf{A}\mathbf{z}_{t-1}$ and $\mathbf{y}_t = \mathbf{C}\mathbf{z}_t$, and hence $\mathbf{y}_t = \mathbf{C}\mathbf{A}^{t-1}\mathbf{z}_1$. Consequently all the observations must be generated from a $\dim(\mathbf{z}_t)$-dimensional linear manifold or subspace. We can identify this subspace using PCA (see the above references for details). Once we have an estimate of the $\mathbf{z}_t$'s, we can fit the model as if it were fully observed. We can either use these estimates in their own right, or use them to initialize EM.

18.4.5 Bayesian methods for "fitting" LG-SSMs

There are various offline Bayesian alternatives to the EM algorithm, including variational Bayes EM (Beal 2003; Barber and Chiappa 2007) and blocked Gibbs sampling (Carter and Kohn 1994; Cappe et al. 2005; Fruhwirth-Schnatter 2007). The Bayesian approach can also be used to perform online learning, as we discussed in Section 18.2.3. Unfortunately, once we add the SSM parameters to the state space, the model is generally no longer linear Gaussian. Consequently we must use some of the approximate online inference methods to be discussed below.

18.5 Approximate online inference for non-linear, non-Gaussian SSMs

In Section 18.3.1, we discussed how to perform exact online inference for LG-SSMs. However, many models are non linear. For example, most moving objects do not move in straight lines. And even if they did, if we assume the parameters of the model are unknown and add them

to the state space, the model becomes nonlinear. Furthermore, non-Gaussian noise is also very common, e.g., due to outliers, or when inferring parameters for GLMs instead of just linear regression. For these more general models, we need to use approximate inference.

The approximate inference algorithms we discuss below approximate the posterior by a Gaussian. In general, if $Y = f(X)$, where X has a Gaussian distribution and f is a non-linear function, there are two main ways to approximate $p(Y)$ by a Gaussian. The first is to use a first-order approximation of f. The second is to use the exact f, but to project $f(X)$ onto the space of Gaussians by moment matching. We discuss each of these methods in turn. (See also Section 23.5, where we discuss particle filtering, which is a stochastic algorithm for approximate online inference, which uses a non-parametric approximation to the posterior, which is often more accurate but slower to compute.)

18.5.1 Extended Kalman filter (EKF)

In this section, we focus on non-linear models, but we assume the noise is Gaussian. That is, we consider models of the form

$$\mathbf{z}_t = g(\mathbf{u}_t, \mathbf{z}_{t-1}) + \mathcal{N}(\mathbf{0}, \mathbf{Q}_t) \tag{18.80}$$

$$\mathbf{y}_t = h(\mathbf{z}_t) + \mathcal{N}(\mathbf{0}, \mathbf{R}_t) \tag{18.81}$$

where the transition model g and the observation model h are nonlinear but differentiable functions. Furthermore, we focus on the case where we approximate the posterior by a single Gaussian. (The simplest way to handle more general posteriors (e.g., multi-modal, discrete, etc). is to use particle filtering, which we discuss in Section 23.5.)

The **extended Kalman filter** or **EKF** can be applied to nonlinear Gaussian dynamical systems of this form. The basic idea is to linearize g and h about the previous state estimate using a first order Taylor series expansion, and then to apply the standard Kalman filter equations. (The noise variance in the equations ($\mathbf{Q}$ and $\mathbf{R}$) is not changed, i.e., the additional error due to linearization is not modeled.) Thus we approximate the stationary non-linear dynamical system with a non-stationary linear dynamical system.

The intuition behind the approach is shown in Figure 18.9, which shows what happens when we pass a Gaussian distribution $p(x)$, shown on the bottom right, through a nonlinear function $y = g(x)$, shown on the top right. The resulting distribution (approximated by Monte Carlo) is shown in the shaded gray area in the top left corner. The best Gaussian approximation to this, computed from $\mathbb{E}[g(x)]$ and $\mathrm{var}[g(x)]$ by Monte Carlo, is shown by the solid black line. The EKF approximates this Gaussian as follows: it linearizes the g function at the current mode, μ, and then passes the Gaussian distribution $p(x)$ through this linearized function. In this example, the result is quite a good approximation to the first and second moments of $p(y)$, for much less cost than an MC approximation.

In more detail, the method works as follows. We approximate the measurement model using

$$p(\mathbf{y}_t|\mathbf{z}_t) \approx \mathcal{N}(\mathbf{y}_t|\mathbf{h}(\boldsymbol{\mu}_{t|t-1}) + \mathbf{H}_t(\mathbf{y}_t - \boldsymbol{\mu}_{t|t-1}), \mathbf{R}_t) \tag{18.82}$$

where $\mathbf{H}_t$ is the Jacobian matrix of $\mathbf{h}$ evaluated at the prior mode:

$$H_{ij} \triangleq \frac{\partial h_i(\mathbf{z})}{\partial z_j} \tag{18.83}$$

$$\mathbf{H}_t \triangleq \mathbf{H}|_{\mathbf{z}=\boldsymbol{\mu}_{t|t-1}} \tag{18.84}$$

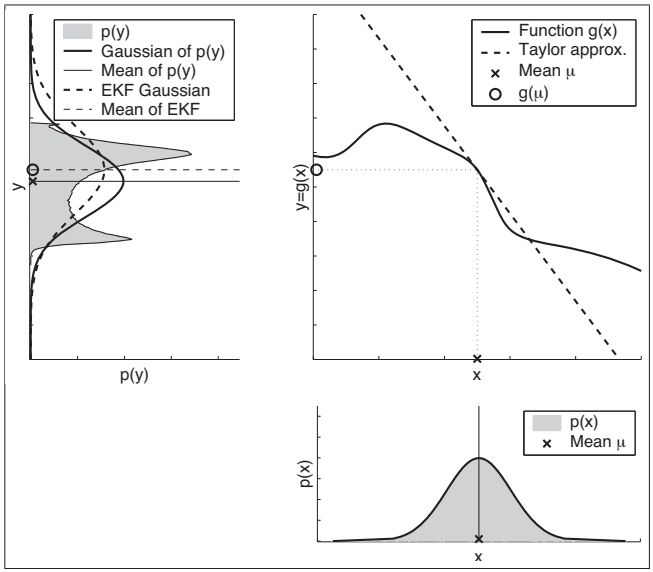


Figure 18.9 Nonlinear transformation of a Gaussian random variable. The prior $p(x)$ is shown on the bottom right. The function $y = g(x)$ is shown on the top right. The transformed distribution $p(y)$ is shown in the top left. A linear function induces a Gaussian distribution, but a non-linear function induces a complex distribution. The solid line is the best Gaussian approximation to this; the dotted line is the EKF approximation to this. Source: Figure 3.4 of (Thrun et al. 2006). Used with kind permission of Sebastian Thrun.

Similarly, we approximate the system model using

$$p(\mathbf{z}_t|\mathbf{z}_{t-1}, \mathbf{u}_t) \approx \mathcal{N}(\mathbf{z}_t|\mathbf{g}(\mathbf{u}_t, \boldsymbol{\mu}_{t-1}) + \mathbf{G}_t(\mathbf{z}_{t-1} - \boldsymbol{\mu}_{t-1}), \mathbf{Q}_t) \tag{18.85}$$

where

$$G_{ij}(\mathbf{u}) \triangleq \frac{\partial g_i(\mathbf{u}, \mathbf{z})}{\partial z_j} \tag{18.86}$$

$$\mathbf{G}_t \triangleq \mathbf{G}(\mathbf{u}_t)|_{\mathbf{z}=\boldsymbol{\mu}_{t-1}} \tag{18.87}$$

so $\mathbf{G}$ is the Jacobian matrix of $\mathbf{g}$ evaluated at the prior mode.

Given this, we can then apply the Kalman filter to compute the posterior as follows:

$$\boldsymbol{\mu}_{t|t-1} = \mathbf{g}(\mathbf{u}_t, \boldsymbol{\mu}_{t-1}) \tag{18.88}$$

$$\mathbf{V}_{t|t-1} = \mathbf{G}_t\mathbf{V}_{t-1}\mathbf{G}_t^T + \mathbf{Q}_t \tag{18.89}$$

$$\mathbf{K}_t = \mathbf{V}_{t|t-1}\mathbf{H}_t^T(\mathbf{H}_t\mathbf{V}_{t|t-1}\mathbf{H}_t^T + \mathbf{R}_t)^{-1} \tag{18.90}$$

$$\boldsymbol{\mu}_t = \boldsymbol{\mu}_{t|t-1} + \mathbf{K}_t(\mathbf{y}_t - \mathbf{h}(\boldsymbol{\mu}_{t|t-1})) \tag{18.91}$$

$$\mathbf{V}_t = (\mathbf{I} - \mathbf{K}_t\mathbf{H}_t)\mathbf{V}_{t|t-1} \tag{18.92}$$

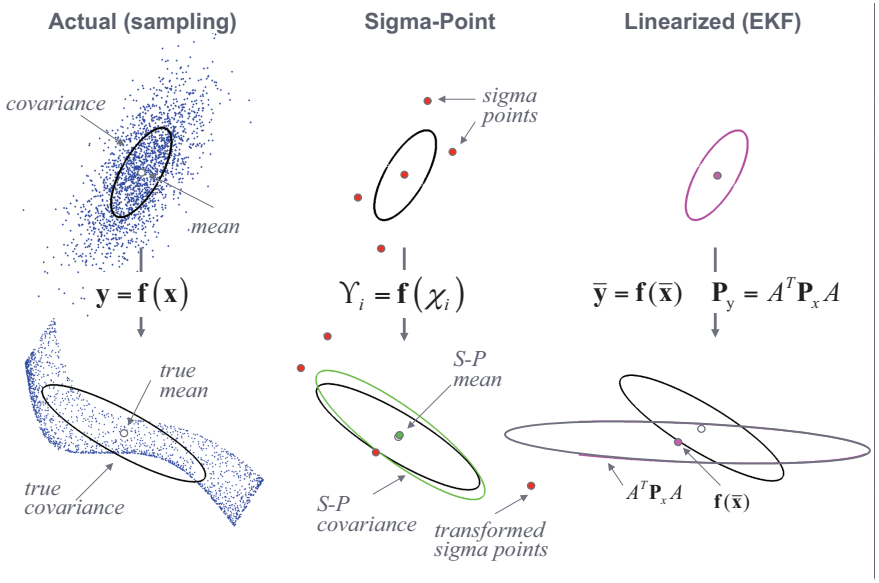


Figure 18.10 An example of the unscented transform in two dimensions. Source: (Wan and der Merwe 2001). Used with kind permission of Eric Wan.

We see that the only difference from the regular Kalman filter is that, when we compute the state prediction, we use $\mathbf{g}(\mathbf{u}_t, \boldsymbol{\mu}_{t-1})$ instead of $\mathbf{A}_t \boldsymbol{\mu}_{t-1} + \mathbf{B}_t \mathbf{u}_t$, and when we compute the measurement update we use $\mathbf{h}(\boldsymbol{\mu}_{t|t-1})$ instead of $\mathbf{C}_t \boldsymbol{\mu}_{t|t-1}$.

It is possible to improve performance by repeatedly re-linearizing the equations around $\boldsymbol{\mu}_t$ instead of $\boldsymbol{\mu}_{t|t-1}$; this is called the **iterated EKF**, and yields better results, although it is of course slower.

There are two cases when the EKF works poorly. The first is when the prior covariance is large. In this case, the prior distribution is broad, so we end up sending a lot of probability mass through different parts of the function that are far from the mean, where the function has been linearized. The other setting where the EKF works poorly is when the function is highly nonlinear near the current mean. In Section 18.5.2, we will discuss an algorithm called the UKF which works better than the EKF in both of these settings.

18.5.2 Unscented Kalman filter (UKF)

The **unscented Kalman filter** (**UKF**) is a better version of the EKF (Julier and Uhlmann 1997). (Apparently it is so-called because it "doesn't stink"!) The key intuition is this: it is easier to approximate a Gaussian than to approximate a function. So instead of performing a linear approximation to the function, and passing a Gaussian through it, instead pass a deterministically chosen set of points, known as **sigma points**, through the function, and fit a Gaussian to the resulting transformed points. This is known as the **unscented transform**, and is sketched in Figure 18.10. (We explain this figure in detail below.)

The UKF basically uses the unscented transform twice, once to approximate passing through the system model $\mathbf{g}$, and once to approximate passing through the measurement model $\mathbf{h}$. We give the details below. Note that the UKF and EKF both perform $O(d^3)$ operations per time step where d is the size of the latent state-space. However, the UKF is accurate to at least second order, whereas the EKF is only a first order approximation (although both the EKF and UKF can be extended to capture higher order terms). Furthermore, the unscented transform does not require the analytic evaluation of any derivatives or Jacobians (a so-called **derivative free filter**), making it simpler to implement and more widely applicable.

18.5.2.1 The unscented transform

Before explaining the UKF, we first explain the unscented transform. Assume $p(\mathbf{x}) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$, and consider estimating $p(\mathbf{y})$, where $\mathbf{y} = \mathbf{f}(\mathbf{x})$ for some nonlinear function $\mathbf{f}$. The unscented transform does this as follows. First we create a set of $2d + 1$ sigma points $\mathbf{x}_i$, given by

$$\mathbf{x} = \left(\boldsymbol{\mu}, \{\boldsymbol{\mu} + (\sqrt{(d+\lambda)\boldsymbol{\Sigma}})_{:i}\}_{i=1}^{d}, \{\boldsymbol{\mu} - (\sqrt{(d+\lambda)\boldsymbol{\Sigma}})_{:i}\}_{i=1}^{d}\right) \tag{18.93}$$

where $\lambda = \alpha^2(d + \kappa) - d$ is a scaling parameter to be specified below, and the notation $\mathbf{M}_{:i}$ means the i'th column of matrix $\mathbf{M}$.

These sigma points are propagated through the nonlinear function to yield $\mathbf{y}_i = f(\mathbf{x}_i)$, and the mean and covariance for $\mathbf{y}$ is computed as follows:

$$\boldsymbol{\mu}_y = \sum_{i=0}^{2d} w_m^i \mathbf{y}_i \tag{18.94}$$

$$\boldsymbol{\Sigma}_y = \sum_{i=0}^{2d} w_c^i (\mathbf{y}_i - \boldsymbol{\mu}_y)(\mathbf{y}_i - \boldsymbol{\mu}_y)^T \tag{18.95}$$

where the w's are weighting terms, given by

$$w_m^i = \frac{\lambda}{d+\lambda} \tag{18.96}$$

$$w_c^i = \frac{\lambda}{d+\lambda} + (1 - \alpha^2 + \beta) \tag{18.97}$$

$$w_m^i = w_c^i = \frac{1}{2(d+\lambda)} \tag{18.98}$$

See Figure 18.10 for an illustration.

In general, the optimal values of α, β and κ are problem dependent, but when $d = 1$, they are $\alpha = 1$, $\beta = 0$, $\kappa = 2$. Thus in the 1d case, $\lambda = 2$, so the 3 sigma points are μ, $\mu + \sqrt{3}\sigma$ and $\mu - \sqrt{3}\sigma$.

18.5.2.2 The UKF algorithm

The UKF algorithm is simply two applications of the unscented tranform, one to compute $p(\mathbf{z}_t|\mathbf{y}_{1:t-1}, \mathbf{u}_{1:t})$ and the other to compute $p(\mathbf{z}_t|\mathbf{y}_{1:t}, \mathbf{u}_{1:t})$. We give the details below.

The first step is to approximate the predictive density $p(\mathbf{z}_t|\mathbf{y}_{1:t-1}, \mathbf{u}_{1:t}) \approx \mathcal{N}(\mathbf{z}_t|\overline{\boldsymbol{\mu}}_t, \overline{\boldsymbol{\Sigma}}_t)$ by passing the old belief state $\mathcal{N}(\mathbf{z}_{t-1}|\boldsymbol{\mu}_{t-1}, \boldsymbol{\Sigma}_{t-1})$ through the system model $\mathbf{g}$ as follows:

$$\mathbf{z}_{t-1}^0 = \left(\boldsymbol{\mu}_{t-1}, \{\boldsymbol{\mu}_{t-1} + \gamma(\sqrt{\boldsymbol{\Sigma}_{t-1}})_{:i}\}_{i=1}^d, \{\boldsymbol{\mu}_{t-1} - \gamma(\sqrt{\boldsymbol{\Sigma}_{t-1}})_{:i}\}_{i=1}^d\right) \tag{18.99}$$

$$\overline{\mathbf{z}}_t^{*i} = \mathbf{g}(\mathbf{u}_t, \mathbf{z}_{t-1}^{0i}) \tag{18.100}$$

$$\overline{\boldsymbol{\mu}}_t = \sum_{i=0}^{2d} w_m^i \overline{\mathbf{z}}_t^{*i} \tag{18.101}$$

$$\overline{\boldsymbol{\Sigma}}_t = \sum_{i=0}^{2d} w_c^i (\overline{\mathbf{z}}_t^{*i} - \overline{\boldsymbol{\mu}}_t)(\overline{\mathbf{z}}_t^{*i} - \overline{\boldsymbol{\mu}}_t) + \mathbf{Q}_t \tag{18.102}$$

where $\gamma = \sqrt{d + \lambda}$.

The second step is to approximate the likelihood $p(\mathbf{y}_t|\mathbf{z}_t) \approx \mathcal{N}(\mathbf{y}_t|\hat{\mathbf{y}}_t, \mathbf{S}_t)$ by passing the prior $\mathcal{N}(\mathbf{z}_t|\overline{\boldsymbol{\mu}}_t, \overline{\boldsymbol{\Sigma}}_t)$ through the observation model $\mathbf{h}$:

$$\overline{\mathbf{z}}_t^0 = \left(\overline{\boldsymbol{\mu}}_t, \{\overline{\boldsymbol{\mu}}_t + \gamma(\sqrt{\overline{\boldsymbol{\Sigma}}_t})_{:i}\}_{i=1}^d, \{\overline{\boldsymbol{\mu}}_t - \gamma(\sqrt{\overline{\boldsymbol{\Sigma}}_t})_{:i}\}_{i=1}^d\right) \tag{18.103}$$

$$\overline{\mathbf{y}}_t^{*i} = \mathbf{h}(\overline{\mathbf{z}}_t^{0i}) \tag{18.104}$$

$$\hat{\mathbf{y}}_t = \sum_{i=0}^{2d} w_m^i \overline{\mathbf{y}}_t^{*i} \tag{18.105}$$

$$\mathbf{S}_t = \sum_{i=0}^{2d} w_c^i (\overline{\mathbf{y}}_t^{*i} - \hat{\mathbf{y}}_t)(\overline{\mathbf{y}}_t^{*i} - \hat{\mathbf{y}}_t)^T + \mathbf{R}_t \tag{18.106}$$

Finally, we use Bayes rule for Gaussians to get the posterior $p(\mathbf{z}_t|\mathbf{y}_{1:t}, \mathbf{u}_{1:t}) \approx \mathcal{N}(\mathbf{z}_t|\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t)$:

$$\overline{\boldsymbol{\Sigma}}_t^{z,y} = \sum_{i=0}^{2d} w_c^i (\overline{\mathbf{z}}_t^{*i} - \overline{\boldsymbol{\mu}}_t)(\overline{\mathbf{y}}_t^{*i} - \hat{\mathbf{y}}_t)^T \tag{18.107}$$

$$\mathbf{K}_t = \overline{\boldsymbol{\Sigma}}_t^{z,y} \mathbf{S}_t^{-1} \tag{18.108}$$

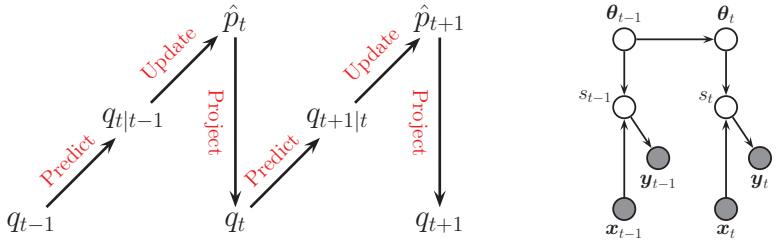
$$\boldsymbol{\mu}_t = \overline{\boldsymbol{\mu}}_t + \mathbf{K}_t(\mathbf{y}_t - \hat{\mathbf{y}}_t) \tag{18.109}$$

$$\boldsymbol{\Sigma}_t = \overline{\boldsymbol{\Sigma}}_t - \mathbf{K}_t \mathbf{S}_t \mathbf{K}_t^T \tag{18.110}$$

18.5.3 Assumed density filtering (ADF)

In this section, we discuss inference where we perform an exact update step, but then approximate the posterior by a distribution of a certain convenient form, such as a Gaussian. More precisely, let the unknowns that we want to infer be denoted by $\boldsymbol{\theta}_t$. Suppose that $\mathcal{Q}$ is a set of tractable distributions, e.g., Gaussians with a diagonal covariance matrix, or a product of discrete distributions. Suppose that we have an approximate prior $q_{t-1}(\boldsymbol{\theta}_{t-1}) \approx p(\boldsymbol{\theta}_{t-1}|\mathbf{y}_{1:t-1})$, where $q_{t-1} \in \mathcal{Q}$. We can update this with the new measurement to get the approximate posterior

$$\hat{p}(\boldsymbol{\theta}_t) = \frac{1}{Z_t} p(\mathbf{y}_t|\boldsymbol{\theta}_t) q_{t|t-1}(\boldsymbol{\theta}_t) \tag{18.111}$$



(a) (b)

Figure 18.11 (a) Illustration of the predict-update-project cycle of assumed density filtering. (b) A dynamical logistic regression model. Compare to Figure 18.4(a).

where

$$Z_t = \int p(\mathbf{y}_t|\boldsymbol{\theta}_t) q_{t|t-1}(\boldsymbol{\theta}_t) d\boldsymbol{\theta}_t \tag{18.112}$$

is the normalization constant and

$$q_{t|t-1}(\boldsymbol{\theta}_t) = \int p(\boldsymbol{\theta}_t|\boldsymbol{\theta}_{t-1}) q_{t-1}(\boldsymbol{\theta}_{t-1}) d\boldsymbol{\theta}_{t-1} \tag{18.113}$$

is the one step ahead predictive distribution. If the prior is from a suitably restricted family, this one-step update process is usually tractable. However, we often find that the resulting posterior is no longer in our tractable family, $\hat{p}(\boldsymbol{\theta}_t) \notin \mathcal{Q}$. So after updating we seek the best tractable approximation by computing

$$q(\boldsymbol{\theta}_t) = \underset{q \in \mathcal{Q}}{\operatorname{argmin}} \, \mathbb{KL}\left(\hat{p}(\boldsymbol{\theta}_t) || q(\boldsymbol{\theta}_t)\right) \tag{18.114}$$

This minimizes the the Kullback-Leibler divergence (Section 2.8.2) from the approximation $q(\boldsymbol{\theta}_t)$ to the "exact" posterior $\hat{p}(\boldsymbol{\theta}_t)$, and can be thought of as projecting $\hat{p}$ onto the space of tractable distributions. The whole algorithm consists of **predict-update-project** cycles. This is known as **assumed density filtering** or **ADF** (Maybeck 1979). See Figure 18.11(a) for a sketch.

If q is in the exponential family, one can show that this KL minimization can be done by **moment matching**. We give some examples of this below.

18.5.3.1 Boyen-Koller algorithm for online inference in DBNs

If we are performing inference in a discrete-state dynamic Bayes net (Section 17.6.7), where θ_{tj} is the j'th hidden variable at time t, then the exact posterior $p(\boldsymbol{\theta}_t)$ becomes intractable to compute because of the entanglement problem. Suppose we use a fully factored approximation of the form $q(\boldsymbol{\theta}_t) = \prod_{j=1}^{D} \mathrm{Cat}(\theta_{t,j}|\boldsymbol{\pi}_{t,j})$, where $\pi_{tjk} = q(\theta_{t,j} = k)$ is the probability variable j is in state k, and D is the number of variables. In this case, the moment matching operation becomes

$$\pi_{tjk} = \hat{p}(\theta_{t,j} = k) \tag{18.115}$$

This can be computed by performing a predict-update step using the factored prior, and then computing the posterior marginals. This is known as the **Boyen-Koller** algorithm, named after the authors of (Boyen and Koller 1998), who demonstrated that the error incurred by this series of repeated approximations remains bounded (under certain assumptions about the stochasticity of the system).

18.5.3.2 Gaussian approximation for online inference in GLMs

Now suppose $q(\boldsymbol{\theta}_t) = \prod_{j=1}^{D} \mathcal{N}(\theta_{t,j}|\mu_{t,j}, \tau_{t,j})$, where $\tau_{t,j}$ is the variance. Then the optimal parameters of the tractable approximation to the posterior are

$$\mu_{t,j} = \mathbb{E}_{\hat{p}}\left[\theta_{t,j}\right], \ \tau_{t,j} = \mathrm{var}_{\hat{p}}\left[\theta_{t,j}\right] \tag{18.116}$$

This method can be used to do online inference for the parameters of many statistical models. For example, theTrueSkill system, used in Microsoft's Xbox to rank players over time, uses this form of approximation (Herbrich et al. 2007). We can also apply this method to simpler models, such as GLM, which have the advantage that the posterior is log-concave. Below we explain how to do this for binary logistic regression, following the presentation of (Zoeter 2007).

The model has the form

$$p(y_t|\mathbf{x}_t, \boldsymbol{\theta}_t) = \mathrm{Ber}(y_t|\mathrm{sigm}(\mathbf{x}_t^T\boldsymbol{\theta}_t)) \tag{18.117}$$

$$p(\boldsymbol{\theta}_t|\boldsymbol{\theta}_{t-1}) = \mathcal{N}(\boldsymbol{\theta}_t|\boldsymbol{\theta}_{t-1}, \sigma^2\mathbf{I}) \tag{18.118}$$

where σ^2 is some process noise which allows the parameters to change slowly over time. (This can be set to 0, as in the recursive least squares method (Section 18.2.3), if desired.) We will assume $q_{t-1}(\boldsymbol{\theta}_{t-1}) = \prod_j \mathcal{N}(\theta_{t-1,j}|\mu_{t-1,j}, \tau_{t-1,j})$ is the tractable prior. We can compute the one-step-ahead predictive density $q_{t|t-1}(\boldsymbol{\theta}_t)$ using the standard linear-Gaussian update. So now we concentrate on the measurement update step.

Define the deterministic quantity $s_t = \boldsymbol{\theta}_t^T\mathbf{x}_t$, as shown in Figure 18.11(b). If $q_{t|t-1}(\boldsymbol{\theta}_t) = \prod_j \mathcal{N}(\theta_{t,j}|\mu_{t|t-1,j}, \tau_{t|t-1,j})$, then we can compute the predictive distribution for s_t as follows:

$$q_{t|t-1}(s_t) = \mathcal{N}(s_t|m_{t|t-1}, v_{t|t-1}) \tag{18.119}$$

$$m_{t|t-1} = \sum_j x_{t,j}\mu_{t|t-1,j} \tag{18.120}$$

$$v_{t|t-1} = \sum_j x_{t,j}^2\tau_{t|t-1,j} \tag{18.121}$$

The posterior for s_t is given by

$$q_t(s_t) = \mathcal{N}(s_t|m_t, v_t) \tag{18.122}$$

$$m_t = \int s_t \frac{1}{Z_t} p(y_t|s_t) q_{t|t-1}(s_t) ds_t \tag{18.123}$$

$$v_t = \int s_t^2 \frac{1}{Z_t} p(y_t|s_t) q_{t|t-1}(s_t) ds_t - m_t^2 \tag{18.124}$$

$$Z_t = \int p(y_t|s_t) q_{t|t-1}(s_t) ds_t \tag{18.125}$$

where $p(y_t|s_t) = \text{Ber}(y_t|s_t)$. These integrals are one dimensional, and so can be computed using Gaussian quadrature (see (Zoeter 2007) for details). This is the same as one step of the UKF algorithm.

Having inferred $q(s_t)$, we need to compute $q(\boldsymbol{\theta}|s_t)$. This can be done as follows. Define δ_m as the change in the mean of s_t and δ_v as the change in the variance:

$$m_t = m_{t|t-1} + \delta_m, \; v_t = v_{t|t-1} + \delta_v \tag{18.126}$$

Then one can show that the new factored posterior over the model parameters is given by

$$q(\theta_{t,j}) = \mathcal{N}(\theta_{t,j}|\mu_{t,j}, \tau_{t,j}) \tag{18.127}$$

$$\mu_{t,j} = \mu_{t|t-1,j} + a_j \delta_m \tag{18.128}$$

$$\tau_{t,j} = \tau_{t|t-1,j} + a_j^2 \delta_v \tag{18.129}$$

$$a_j \triangleq \frac{x_{t,j}\tau_{t|t-1,j}}{\sum_{j'} x_{t,j'}^2 \tau_{t|t-1,j}^2} \tag{18.130}$$

Thus we see that the parameters which correspond to inputs with larger magnitude (big $|x_{t,j}|$) or larger uncertainty (big $\tau_{t|t-1,j}$) get updated most, which makes intuitive sense.

In (Opper 1998) a version of this algorithm is derived using a probit likelihood (see Section 9.4). In this case, the measurement update can be done in closed form, without the need for numerical integration. In either case, the algorithm only takes $O(D)$ operations per time step, so it can be applied to models with large numbers of parameters. And since it is an online algorithm, it can also handle massive datasets. For example (Zhang et al. 2010) use a version of this algorithm to fit a multi-class classifier online to very large datasets. They beat alternative (non Bayesian) online learning algorithms, and sometimes even outperform state of the art batch (offline) learning methods such as SVMs (described in Section 14.5).

18.6 Hybrid discrete/continuous SSMs

Many systems contain both discrete and continuous hidden variables; these are known as **hybrid systems**. For example, the discrete variables may indicate whether a measurement sensor is faulty or not, or which "regime" the system is in. We will see some other examples below.

A special case of a hybrid system is when we combine an HMM and an LG-SSM. This is called a **switching linear dynamical system** (SLDS), a **jump Markov linear system** (JMLS), or a **switching state space model** (SSSM). More precisely, we have a discrete latent variable, $q_t \in \{1, \ldots, K\}$, a continuous latent variable, $\mathbf{z}_t \in \mathbb{R}^L$, an continuous observed response $\mathbf{y}_t \in \mathbb{R}^D$ and an optional continuous observed input or control $\mathbf{u}_t \in \mathbb{R}^U$. We then assume that the continuous variables have linear Gaussian CPDs, conditional on the discrete states:

$$p(q_t = k|q_{t-1} = j, \boldsymbol{\theta}) = A_{ij} \tag{18.131}$$

$$p(\mathbf{z}_t|\mathbf{z}_{t-1}, q_t = k, \mathbf{u}_t, \boldsymbol{\theta}) = \mathcal{N}(\mathbf{z}_t|\mathbf{A}_k\mathbf{z}_{t-1} + \mathbf{B}_k\mathbf{u}_t, \mathbf{Q}_k) \tag{18.132}$$

$$p(\mathbf{y}_t|\mathbf{z}_t, q_t = k, \mathbf{u}_t, \boldsymbol{\theta}) = \mathcal{N}(\mathbf{y}_t|\mathbf{C}_k\mathbf{z}_t + \mathbf{D}_k\mathbf{u}_t, \mathbf{R}_k) \tag{18.133}$$

See Figure 18.12(a) for the DGM representation.

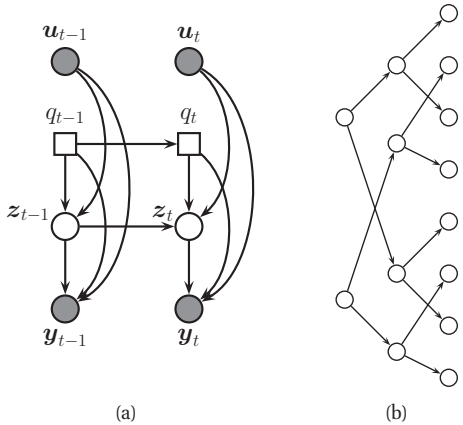


Figure 18.12 A switching linear dynamical system. (a) Squares represent discrete nodes, circles represent continuous nodes. (b) Illustration of how the number of modes in the belief state grows exponentially over time. We assume there are two binary states.

18.6.1 Inference

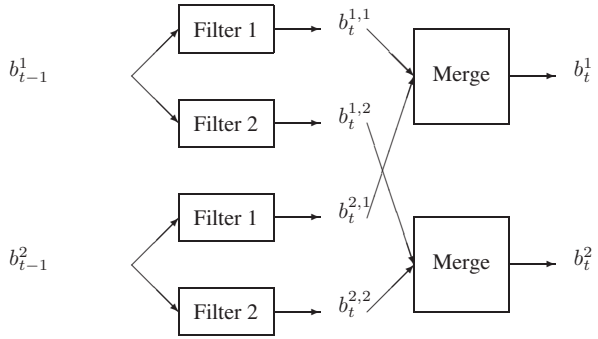
Unfortunately inference (i.e., state estimation) in hybrid models, including the switching LG-SSM model, is intractable. To see why, suppose q_t is binary, but that only the dynamics $\mathbf{A}$ depend on q_t, not the observation matrix. Our initial belief state will be a mixture of 2 Gaussians, corresponding to $p(\mathbf{z}_1|\mathbf{y}_1, q_1 = 1)$ and $p(\mathbf{z}_1|\mathbf{y}_1, q_1 = 2)$. The one-step-ahead predictive density will be a mixture of 4 Gaussians $p(\mathbf{z}_2|\mathbf{y}_1, q_1 = 1, q_2 = 1)$, $p(\mathbf{z}_2|\mathbf{y}_1, q_1 = 1, q_2 = 2)$, $p(\mathbf{z}_2|\mathbf{y}_1, q_1 = 2, q_2 = 1)$, and $p(\mathbf{z}_2|\mathbf{y}_1, q_1 = 2, q_2 = 2)$, obtained by passing each of the prior modes through the 2 possible transition models. The belief state at step 2 will also be a mixture of 4 Gaussians, obtained by updating each of the above distributions with $\mathbf{y}_2$. At step 3, the belief state will be a mixture of 8 Gaussians. And so on. So we see there is an exponential explosion in the number of modes (see Figure 18.12(b)).

Various approximate inference methods have been proposed for this model, such as the following:

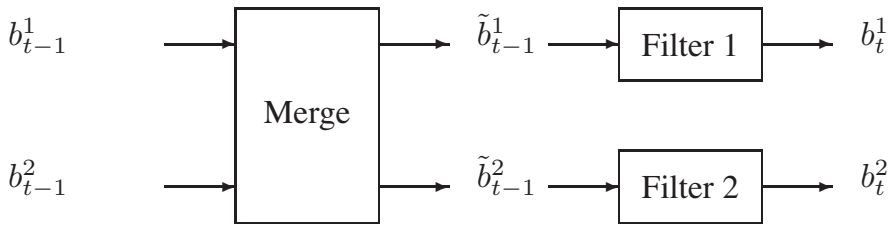
- Prune off low probability trajectories in the discrete tree; this is the basis of **multiple hypothesis tracking** (Bar-Shalom and Fortmann 1988; Bar-Shalom and Li 1993).
- Use Monte Carlo. Essentially we just sample discrete trajectories, and apply an analytical filter to the continuous variables conditional on a trajectory. See Section 23.6 for details.
- Use ADF, where we approximate the exponentially large mixture of Gaussians with a smaller mixture of Gaussians. See Section 18.6.1.1 for details.

18.6.1.1 A Gaussian sum filter for switching SSMs

A **Gaussian sum filter** (Sorenson and Alspach 1971) approximates the belief state at each step by a mixture of K Gaussians. This can be implemented by running K Kalman filters in



(a)



(b)

Figure 18.13 ADF for a switching linear dynamical system. (a) GPB2 method. (b) IMM method. See text for details.

parallel. This is particularly well suited to switching SSMs. We now describe one version of this algorithm, known as the "second order **generalized pseudo Bayes filter**" (GPB2) (Bar-Shalom and Fortmann 1988). We assume that the prior belief state b_{t-1} is a mixture of K Gaussians, one per discrete state:

$$b^i_{t-1} \triangleq p(\mathbf{z}_{t-1}, q_{t-1} = i|\mathbf{y}_{1:t-1}) = \pi_{t-1,i}\mathcal{N}(\mathbf{z}_{t-1}|\boldsymbol{\mu}_{t-1,i}, \boldsymbol{\Sigma}_{t-1,i}) \tag{18.134}$$

We then pass this through the K different linear models to get

$$b^{ij}_t \triangleq p(\mathbf{z}_t, q_{t-1} = i, q_t = j|\mathbf{y}_{1:t}) = \pi_{tij}\mathcal{N}(\mathbf{z}_t|\boldsymbol{\mu}_{t,ij}, \boldsymbol{\Sigma}_{t,ij}) \tag{18.135}$$

where $\pi_{tij} = \pi_{t-1,i}p(q_t = j|q_{t-1} = i)$. Finally, for each value of j, we collapse the K Gaussian mixtures down to a single mixture to give

$$b^j_t \triangleq p(\mathbf{z}_t, q_t = j|\mathbf{y}_{1:t}) = \pi_{tj}\mathcal{N}(\mathbf{z}_t|\boldsymbol{\mu}_{t,j}, \boldsymbol{\Sigma}_{t,j}) \tag{18.136}$$

See Figure 18.13(a) for a sketch.

The optimal way to approximate a mixture of Gaussians with a single Gaussian is given by $q = \arg\min_q \mathbb{KL}(q||p)$, where $p(\mathbf{z}) = \sum_k \pi_k \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$ and $q(\mathbf{z}) = \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$. This can be solved by moment matching, that is,

$$\boldsymbol{\mu} = \mathbb{E}[\mathbf{z}] = \sum_k \pi_k \boldsymbol{\mu}_k \tag{18.137}$$

$$\boldsymbol{\Sigma} = \mathrm{cov}[\mathbf{z}] = \sum_k \pi_k \left(\boldsymbol{\Sigma}_k + (\boldsymbol{\mu}_k - \boldsymbol{\mu})(\boldsymbol{\mu}_k - \boldsymbol{\mu})^T\right) \tag{18.138}$$

In the graphical model literature, this is called **weak marginalization** (Lauritzen 1992), since it preserves the first two moments. Applying these equations to our model, we can go from b_t^{ij} to b_t^j as follows (where we drop the t subscript for brevity):

$$\pi_j = \sum_i \pi_{ij} \tag{18.139}$$

$$\pi_{j|i} = \frac{\pi_{ij}}{\sum_{j'} \pi_{ij'}} \tag{18.140}$$

$$\boldsymbol{\mu}_j = \sum_i \pi_{j|i} \boldsymbol{\mu}_{ij} \tag{18.141}$$

$$\boldsymbol{\Sigma}_j = \sum_i \pi_{j|i} \left(\boldsymbol{\Sigma}_{ij} + (\boldsymbol{\mu}_{ij} - \boldsymbol{\mu}_j)(\boldsymbol{\mu}_{ij} - \boldsymbol{\mu}_j)^T\right) \tag{18.142}$$

This algorithm requires running K^2 filters at each step. A cheaper alternative is to represent the belief state by a single Gaussian, marginalizing over the discrete switch at each step. This is a straightforward application of ADF. An offline extension to this method, called **expectation correction**, is described in (Barber 2006; Mesot and Barber 2009).

Another heuristic approach, known as **interactive multiple models** or **IMM** (Bar-Shalom and Fortmann 1988), can be obtained by first collapsing the prior to a single Gaussian (by moment matching), and then updating it using K different Kalman filters, one per value of q_t. See Figure 18.13(b) for a sketch.

18.6.2 Application: data association and multi-target tracking

Suppose we are tracking K objects, such as airplanes, and at time t, we observe K' detection events, e.g., "blips" on a radar screen. We can have $K' < K$ due to occlusion or missed detections. We can have $K' > K$ due to clutter or false alarms. Or we can have $K' = K$. In any case, we need to figure out the **correspondence** between the K' detections $\mathbf{y}_{tk}$ and the K objects $\mathbf{z}_{tj}$. This is called the problem of **data association**, and it arises in many application domains.

Figure 18.14 gives an example in which we are tracking $K = 2$ objects. At each time step, q_t is the unknown mapping which specifies which objects caused which observations. It specifies the "wiring diagram" for time slice t. The standard way to solve this problem is to compute a weight which measures the "compatibility" between object j and measurement k, typically based on how close k is to where the model thinks j should be (the so-called **nearest neighbor data association** heuristic). This gives us a $K \times K'$ weight matrix. We can make this into a

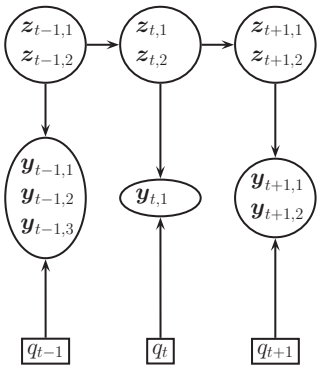


Figure 18.14 A model for tracking two objects in the presence of data-assocation ambiguity. We observe 3, 1 and 2 detections in the first three time steps.

square matrix of size $N \times N$, where $N = \max(K, K')$, by adding dummy background objects, which can explain all the false alarms, and adding dummy observations, which can explain all the missed detections. We can then compute the maximal weight bipartite matching using the **Hungarian algorithm**, which takes $O(N^3)$ time (see e.g., (Burkard et al. 2009)). Conditional on this, we can perform a Kalman filter update, where objects that are assigned to dummy observations do not perform a measurement update.

An extension of this method, to handle a variable and/or unknown number of objects, is known as **multi-target tracking**. This requires dealing with a variable-sized state space. There are many ways to do this, but perhaps the simplest and most robust methods are based on sequential Monte Carlo (e.g., (Ristic et al. 2004)) or MCMC (e.g., (Khan et al. 2006; Oh et al. 2009)).

18.6.3 Application: fault diagnosis

Consider the model in Figure 18.15(a). This represents an industrial plant consisting of various tanks of liquid, interconnected by pipes. In this example, we just have two tanks, for simplicity. We want to estimate the pressure inside each tank, based on a noisy measurement of the flow into and out of each tank. However, the measurement devices can sometimes fail. Furthermore, pipes can burst or get blocked; we call this a "resistance failure". This model is widely used as a benchmark in the **fault diagnosis** community (Mosterman and Biswas 1999).

We can create a probabilistic model of the system as shown in Figure 18.15(b). The square nodes represent discrete variables, such as measurement failures and resistance failures. The remaining variables are continuous. A variety of approximate inference algorithms can be applied to this model. See (Koller and Lerner 2001) for one approach, based on Rao-Blackwellized particle filtering (which is explained in Section 23.6).

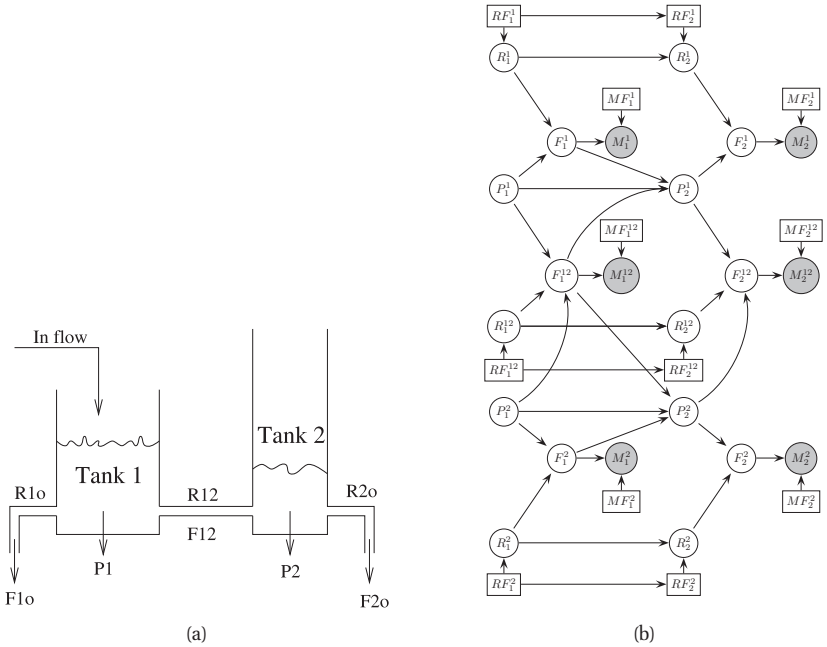


Figure 18.15 (a) The two-tank system. The goal is to infer when pipes are blocked or have burst, or sensors have broken, from (noisy) observations of the flow out of tank 1, $F1o$, out of tank 2, $F2o$, or between tanks 1 and 2, $F12$. $R1o$ is a hidden variable representing the resistance of the pipe out of tank 1, $P1$ is a hidden variable representing the pressure in tank 1, etc. Source: Figure 11 of (Koller and Lerner 2001) . Used with kind permission of Daphne Koller. (b) Dynamic Bayes net representation of the two-tank system. Discrete nodes are squares, continuous nodes are circles. Abbreviations: R = resistance, P = pressure, F = flow, M = measurement, RF = resistance failure, MF = measurement failure. Based on Figure 12 of (Koller and Lerner 2001).

18.6.4 Application: econometric forecasting

The switching LG-SSM model is widely used in **econometric forecasting**, where it is called a **regime switching** model. For example, we can combine two linear trend models (see Section 18.2.4.2), one in which $b_t > 0$ reflects a growing economy, and one in which $b_t < 0$ reflects a shrinking economy. See (West and Harrison 1997) for further details.

Exercises

Exercise 18.1 Derivation of EM for LG-SSM

Derive the E and M steps for computing a (locally optimal) MLE for an LG-SSM model. Hint: the results are in (Ghahramani and Hinton 1996b); your task is to derive these results.

Exercise 18.2 Seasonal LG-SSM model in standard form

Write the seasonal model in Figure 18.7(a) as an LG-SSM. Define the matrices $\mathbf{A}$, $\mathbf{C}$, $\mathbf{Q}$ and $\mathbf{R}$.

19 *Undirected graphical models (Markov random fields)*

19.1 Introduction

In Chapter 10, we discussed directed graphical models (DGMs), commonly known as Bayes nets. However, for some domains, being forced to choose a direction for the edges, as required by a DGM, is rather awkward. For example, consider modeling an image. We might suppose that the intensity values of neighboring pixels are correlated. We can create a DAG model with a 2d lattice topology as shown in Figure 19.1(a). This is known as a **causal MRF** or a **Markov mesh** (Abend et al. 1965). However, its conditional independence properties are rather unnatural. In particular, the Markov blanket (defined in Section 10.5) of the node X_8 in the middle is the other colored nodes (3, 4, 7, 9, 12 and 13) rather than just its 4 nearest neighbors as one might expect.

An alternative is to use an **undirected graphical model** (**UGM**), also called a **Markov random field** (**MRF**) or **Markov network**. These do not require us to specify edge orientations, and are much more natural for some problems such as image analysis and spatial statistics. For example, an undirected 2d lattice is shown in Figure 19.1(b); now the Markov blanket of each node is just its nearest neighbors, as we show in Section 19.2.

Roughly speaking, the main advantages of UGMs over DGMs are: (1) they are symmetric and therefore more "natural" for certain domains, such as spatial or relational data; and (2) discriminativel UGMs (aka conditional random fields, or CRFs), which define conditional densities of the form $p(\mathbf{y}|\mathbf{x})$, work better than discriminative DGMs, for reasons we explain in Section 19.6.1. The main disadvantages of UGMs compared to DGMs are: (1) the parameters are less interpretable and less modular, for reasons we explain in Section 19.3; and (2) parameter estimation is computationally more expensive, for reasons we explain in Section 19.5. See (Domke et al. 2008) for an empirical comparison of the two approaches for an image processing task.

19.2 Conditional independence properties of UGMs

19.2.1 Key properties

UGMs define CI relationships via simple graph separation as follows: for sets of nodes A, B, and C, we say $\mathbf{x}_A \perp_G \mathbf{x}_B|\mathbf{x}_C$ iff C separates A from B in the graph G. This means that, when we remove all the nodes in C, if there are no paths connecting any node in A to any node in B, then the CI property holds. This is called the **global Markov property** for UGMs. For example, in Figure 19.2(b), we have that $\{1, 2\} \perp \{6, 7\}|\{3, 4, 5\}$.

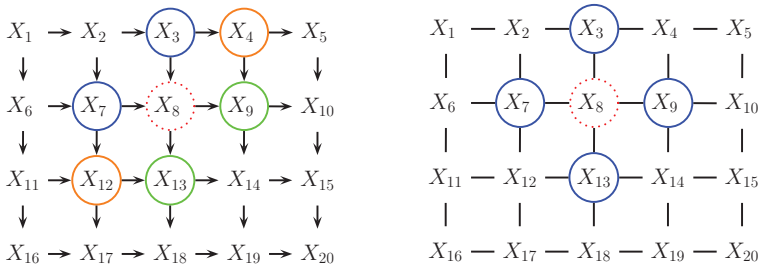


Figure 19.1 (a) A 2d lattice represented as a DAG. The dotted red node X_8 is independent of all other nodes (black) given its Markov blanket, which include its parents (blue), children (green) and co-parents (orange). (b) The same model represented as a UGM. The red node X_8 is independent of the other black nodes given its neighbors (blue nodes).

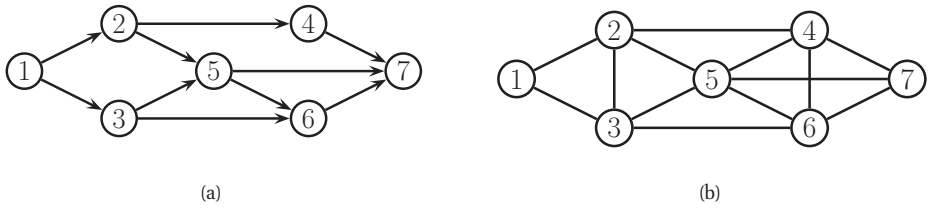


Figure 19.2 (a) A DGM. (b) Its moralized version, represented as a UGM.

The set of nodes that renders a node t conditionally independent of all the other nodes in the graph is called t's **Markov blanket**; we will denote this by $\text{mb}(t)$. Formally, the Markov blanket satisfies the following property:

$$t \perp \mathcal{V} \setminus \text{cl}(t) | \text{mb}(t) \tag{19.1}$$

where $\text{cl}(t) \triangleq \text{mb}(t) \cup \{t\}$ is the **closure** of node t. One can show that, in a UGM, a node's Markov blanket is its set of immediate neighbors. This is called the **undirected local Markov property**. For example, in Figure 19.2(b), we have $\text{mb}(5) = \{2, 3, 4, 6\}$.

From the local Markov property, we can easily see that two nodes are conditionally independent given the rest if there is no direct edge between them. This is called the **pairwise Markov property**. In symbols, this is written as

$$s \perp t | \mathcal{V} \setminus \{s, t\} \iff G_{st} = 0 \tag{19.2}$$

Using the three Markov properties we have discussed, we can derive the following CI properties (amongst others) from the UGM in Figure 19.2(b):

- **Pairwise** $1 \perp 7 | \text{rest}$
- **Local** $1 \perp \text{rest} | 2, 3$

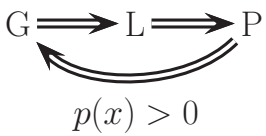


Figure 19.3 Relationship between Markov properties of UGMs.

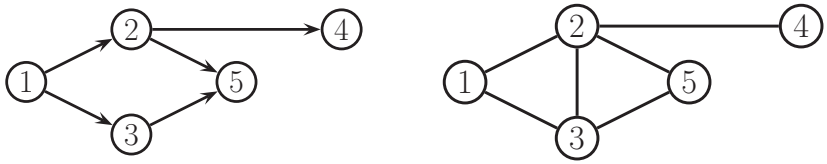


Figure 19.4 (a) The ancestral graph induced by the DAG in Figure 19.2(a) wrt $U = \{2, 4, 5\}$. (b) The moralized version of (a).

- **Global** $1, 2 \perp 6, 7 | 3, 4, 5$

It is obvious that global Markov implies local Markov which implies pairwise Markov. What is less obvious, but nevertheless true (assuming $p(\mathbf{x}) > 0$ for all $\mathbf{x}$, i.e., that p is a positive density), is that pairwise implies global, and hence that all these Markov properties are the same, as illustrated in Figure 19.3 (see e.g., (Koller and Friedman 2009, p119) for a proof).[1] The importance of this result is that it is usually easier to empirically assess pairwise conditional independence; such pairwise CI statements can be used to construct a graph from which global CI statements can be extracted.

19.2.2 An undirected alternative to d-separation

We have seen that determinining CI relationships in UGMs is much easier than in DGMs, because we do not have to worry about the directionality of the edges. In this section, we show how to determine CI relationships for a DGM using a UGM.

It is tempting to simply convert the DGM to a UGM by dropping the orientation of the edges, but this is clearly incorrect, since a v-structure $A \rightarrow B \leftarrow C$ has quite different CI properties than the corresponding undirected chain $A - B - C$. The latter graph incorrectly states that $A \perp C | B$. To avoid such incorrect CI statements, we can add edges between the "unmarried" parents A and C, and then drop the arrows from the edges, forming (in this case) a fully connected undirected graph. This process is called **moralization**. Figure 19.2(b) gives a larger

1. The restriction to positive densities arises because deterministic constraints can result in independencies present in the distribution that are not explicitly represented in the graph. See e.g., (Koller and Friedman 2009, p120) for some examples. Distributions with non-graphical CI properties are said to be **unfaithful** to the graph, so $I(p) \neq I(G)$.

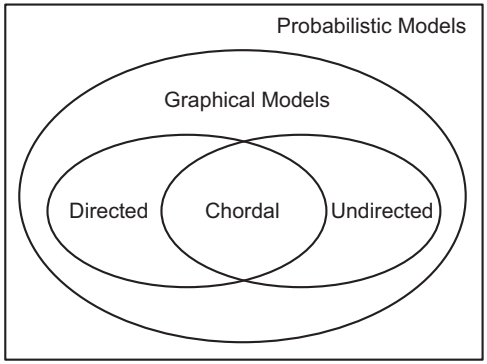


Figure 19.5 DGMs and UGMs can perfectly represent different sets of distributions. Some distributions can be perfectly represented by either DGMs or UGMs; the corresponding graph must be chordal.

example of moralization: we interconnect 2 and 3, since they have a common child 5, and we interconnect 4, 5 and 6, since they have a common child 7.

Unfortunately, moralization loses some CI information, and therefore we cannot use the moralized UGM to determine CI properties of the DGM. For example, in Figure 19.2(a), using d-separation, we see that $4 \perp 5|2$. Adding a moralization arc $4 - 5$ would lose this fact (see Figure 19.2(b)). However, notice that the 4-5 moralization edge, due to the common child 7, is not needed if we do not observe 7 or any of its descendants. This suggests the following approach to determining if $A \perp B|C$. First we form the **ancestral graph** of DAG G with respect to $U = A \cup B \cup C$. This means we remove all nodes from G that are not in U or are not ancestors of U. We then moralize this ancestral graph, and apply the simple graph separation rules for UGMs. For example, in Figure 19.4(a), we show the ancestral graph for Figure 19.2(a) using $U = \{2, 4, 5\}$. In Figure 19.4(b), we show the moralized version of this graph. It is clear that we now correctly conclude that $4 \perp 5|2$.

19.2.3 Comparing directed and undirected graphical models

Which model has more "expressive power", a DGM or a UGM? To formalize this question, recall that we say that G is an I-map of a distribution p if $I(G) \subseteq I(p)$. Now define G to be **perfect map** of p if $I(G) = I(p)$, in other words, the graph can represent all (and only) the CI properties of the distribution. It turns out that DGMs and UGMs are perfect maps for different sets of distributions (see Figure 19.5). In this sense, neither is more powerful than the other as a representation language.

As an example of some CI relationships that can be perfectly modeled by a DGM but not a UGM, consider a v-structure $A \rightarrow C \leftarrow B$. This asserts that $A \perp B$, and $A \not\perp B|C$. If we drop the arrows, we get $A - C - B$, which asserts $A \perp B|C$ and $A \not\perp B$, which is incorrect. In fact, there is no UGM that can precisely represent all and only the two CI statements encoded by a v-structure. In general, CI properties in UGMs are monotonic, in the following sense: if $A \perp B|C$, then $A \perp B|(C \cup D)$. But in DGMs, CI properties can be non-monotonic, since conditioning

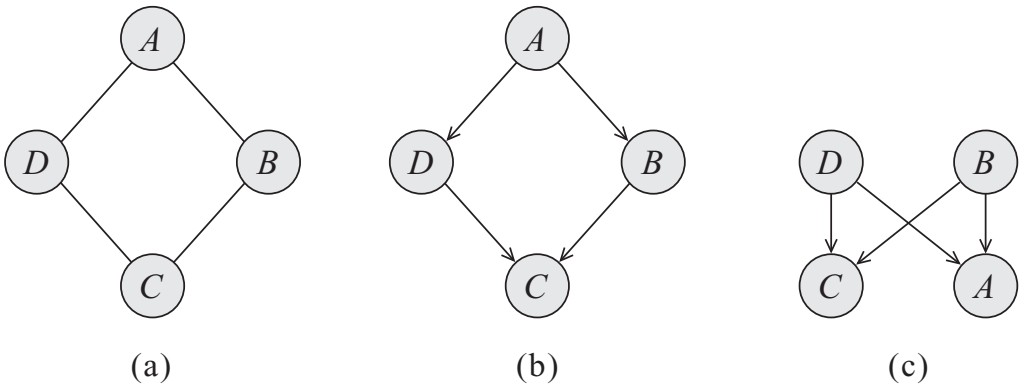


Figure 19.6 A UGM and two failed attempts to represent it as a DGM. Source: Figure 3.10 of (Koller and Friedman 2009). Used with kind permission of Daphne Koller.

on extra variables can eliminate conditional independencies due to explaining away.

As an example of some CI relationships that can be perfectly modeled by a UGM but not a DGM, consider the 4-cycle shown in Figure 19.6(a). One attempt to model this with a DGM is shown in Figure 19.6(b). This correctly asserts that $A \perp C|B, D$. However, it incorrectly asserts that $B \perp D|A$. Figure 19.6(c) is another incorrect DGM: it correctly encodes $A \perp C|B, D$, but incorrectly encodes $B \perp D$. In fact there is no DGM that can precisely represent all and only the CI statements encoded by this UGM.

Some distributions can be perfectly modeled by either a DGM or a UGM; the resulting graphs are called **decomposable** or **chordal**. Roughly speaking, this means the following: if we collapse together all the variables in each maximal clique, to make "mega-variables", the resulting graph will be a tree. Of course, if the graph is already a tree (which includes chains as a special case), it will be chordal. See Section 20.4.1 for further details.

19.3 Parameterization of MRFs

Although the CI properties of UGM are simpler and more natural than for DGMs, representing the joint distribution for a UGM is less natural than for a DGM, as we see below.

19.3.1 The Hammersley-Clifford theorem

Since there is no topological ordering associated with an undirected graph, we can't use the chain rule to represent $p(\mathbf{y})$. So instead of associating CPDs with each node, we associate **potential function**s or **factor**s with each maximal clique in the graph. We will denote the potential function for clique c by $\psi_c(\mathbf{y}_c|\boldsymbol{\theta}_c)$. A potential function can be any non-negative function of its arguments. The joint distribution is then defined to be proportional to the product of clique potentials. Rather surprisingly, one can show that any positive distribution whose CI properties can be represented by a UGM can be represented in this way. We state this result more formally below.

Theorem 19.3.1 (**Hammersley-Clifford**). *A positive distribution $p(\mathbf{y}) > 0$ satisfies the CI properties of an undirected graph G iff p can be represented as a product of factors, one per maximal clique, i.e.,*

$$p(\mathbf{y}|\boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})} \prod_{c \in \mathcal{C}} \psi_c(\mathbf{y}_c|\boldsymbol{\theta}_c) \tag{19.3}$$

where $\mathcal{C}$ is the set of all the (maximal) cliques of G, and $Z(\boldsymbol{\theta})$ is the **partition function** *given by*

$$Z(\boldsymbol{\theta}) \triangleq \sum_{\mathbf{x}} \prod_{c \in \mathcal{C}} \psi_c(\mathbf{y}_c|\boldsymbol{\theta}_c) \tag{19.4}$$

Note that the partition function is what ensures the overall distribution sums to 1.[2]

The proof was never published, but can be found in e.g., (Koller and Friedman 2009).

For example, consider the MRF in Figure 10.1(b). If p satisfies the CI properties of this graph then we can write p as follows:

$$p(\mathbf{y}|\boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})} \psi_{123}(y_1, y_2, y_3)\psi_{234}(y_2, y_3, y_4)\psi_{35}(y_3, y_5) \tag{19.5}$$

where

$$Z = \sum_{\mathbf{y}} \psi_{123}(y_1, y_2, y_3)\psi_{234}(y_2, y_3, y_4)\psi_{35}(y_3, y_5) \tag{19.6}$$

There is a deep connection between UGMs and statistical physics. In particular, there is a model known as the **Gibbs distribution**, which can be written as follows:

$$p(\mathbf{y}|\boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})} \exp(-\sum_{c} E(\mathbf{y}_c|\boldsymbol{\theta}_c)) \tag{19.7}$$

where $E(\mathbf{y}_c) > 0$ is the energy associated with the variables in clique c. We can convert this to a UGM by defining

$$\psi_c(\mathbf{y}_c|\boldsymbol{\theta}_c) = \exp(-E(\mathbf{y}_c|\boldsymbol{\theta}_c)) \tag{19.8}$$

We see that high probability states correspond to low energy configurations. Models of this form are known as **energy based models**, and are commonly used in physics and biochemistry, as well as some branches of machine learning (LeCun et al. 2006).

Note that we are free to restrict the parameterization to the edges of the graph, rather than the maximal cliques. This is called a **pairwise MRF**. In Figure 10.1(b), we get

$$p(\mathbf{y}|\boldsymbol{\theta}) \propto \psi_{12}(y_1, y_2)\psi_{13}(y_1, y_3)\psi_{23}(y_2, y_3)\psi_{24}(y_2, y_4)\psi_{34}(y_3, y_4)\psi_{35}(y_3, y_5) \tag{19.9}$$

$$\propto \prod_{s \sim t} \psi_{st}(y_s, y_t) \tag{19.10}$$

This form is widely used due to its simplicity, although it is not as general.

2. The partition function is denoted by Z because of the German word *zustandssumme*, which means "sum over states". This reflects the fact that a lot of pioneering working in statistical physics was done by Germans.

19.3.2 Representing potential functions

If the variables are discrete, we can represent the potential or energy functions as tables of (non-negative) numbers, just as we did with CPTs. However, the potentials are not probabilities. Rather, they represent the relative "compatibility" between the different assignments to the potential. We will see some examples of this below.

A more general approach is to define the log potentials as a linear function of the parameters:

$$\log \psi_c(\mathbf{y}_c) \triangleq \boldsymbol{\phi}_c(\mathbf{y}_c)^T \boldsymbol{\theta}_c \tag{19.11}$$

where $\boldsymbol{\phi}_c(\mathbf{x}_c)$ is a feature vector derived from the values of the variables $\mathbf{y}_c$. The resulting log probability has the form

$$\log p(\mathbf{y}|\boldsymbol{\theta}) = \sum_c \boldsymbol{\phi}_c(\mathbf{y}_c)^T \boldsymbol{\theta}_c - Z(\boldsymbol{\theta}) \tag{19.12}$$

This is also known as a **maximum entropy** or a **log-linear** model.

For example, consider a pairwise MRF, where for each edge, we associate a feature vector of length K^2 as follows:

$$\boldsymbol{\phi}_{st}(y_s, y_t) = [\ldots, \mathbb{I}(y_s = j, y_t = k), \ldots] \tag{19.13}$$

If we have a weight for each feature, we can convert this into a $K \times K$ potential function as follows:

$$\psi_{st}(y_s = j, y_t = k) = \exp([\boldsymbol{\theta}_{st}^T \boldsymbol{\phi}_{st}]_{jk}) = \exp(\theta_{st}(j, k)) \tag{19.14}$$

So we see that we can easily represent tabular potentials using a log-linear form. But the log-linear form is more general.

To see why this is useful, suppose we are interested in making a probabilistic model of English spelling. Since certain letter combinations occur together quite frequently (e.g., "ing"), we will need higher order factors to capture this. Suppose we limit ourselves to letter trigrams. A tabular potential still has $26^3 = 17,576$ parameters in it. However, most of these triples will never occur.

An alternative approach is to define indicator functions that look for certain "special" triples, such as "ing", "qu-", etc. Then we can define the potential on each trigram as follows:

$$\psi(y_{t-1}, y_t, y_{t+1}) = \exp(\sum_k \theta_k \phi_k(y_{t-1}, y_t, y_{t+1})) \tag{19.15}$$

where k indexes the different features, corresponding to "ing", "qu-", etc., and ϕ_k is the corresponding binary **feature function**. By tying the parameters across locations, we can define the probability of a word of any length using

$$p(\mathbf{y}|\boldsymbol{\theta}) \propto \exp(\sum_t \sum_k \theta_k \phi_k(y_{t-1}, y_t, y_{t+1})) \tag{19.16}$$

This raises the question of where these feature functions come from. In many applications, they are created by hand to reflect domain knowledge (we will see examples later), but it is also possible to learn them from data, as we discuss in Section 19.5.6.

19.4 Examples of MRFs

In this section, we show how several popular probability models can be conveniently expressed as UGMs.

19.4.1 Ising model

The **Ising model** is an example of an MRF that arose from statistical physics.[3] It was originally used for modeling the behavior of magnets. In particular, let $y_s \in \{-1, +1\}$ represent the spin of an atom, which can either be spin down or up. In some magnets, called **ferro-magnets**, neighboring spins tend to line up in the same direction, whereas in other kinds of magnets, called **anti-ferromagnets**, the spins "want" to be different from their neighbors.

We can model this as an MRF as follows. We create a graph in the form of a 2D or 3D lattice, and connect neighboring variables, as in Figure 19.1(b). We then define the following pairwise clique potential:

$$\psi_{st}(y_s, y_t) = \begin{pmatrix} e^{w_{st}} & e^{-w_{st}} \\ e^{-w_{st}} & e^{w_{st}} \end{pmatrix} \tag{19.17}$$

Here w_{st} is the coupling strength between nodes s and t. If two nodes are not connected in the graph, we set $w_{st} = 0$. We assume that the weight matrix $\mathbf{W}$ is symmetric, so $w_{st} = w_{ts}$. Often we assume all edges have the same strength, so $w_{st} = J$ (assuming $w_{st} \neq 0$).

If all the weights are positive, $J > 0$, then neighboring spins are likely to be in the same state; this can be used to model ferromagnets, and is an example of an **associative Markov network**. If the weights are sufficiently strong, the corresponding probability distribution will have two modes, corresponding to the all +1's state and the all -1's state. These are called the **ground states** of the system.

If all of the weights are negative, $J < 0$, then the spins want to be different from their neighbors; this can be used to model an anti-ferromagnet, and results in a **frustrated system**, in which not all the constraints can be satisfied at the same time. The corresponding probability distribution will have multiple modes. Interestingly, computing the partition function $Z(J)$ can be done in polynomial time for associative Markov networks, but is NP-hard in general (Cipra 2000).

There is an interesting analogy between Ising models and Gaussian graphical models. First, assuming $y_t \in \{-1, +1\}$, we can write the unnormalized log probability of an Ising model as follows:

$$\log \tilde{p}(\mathbf{y}) = -\sum_{s \sim t} y_s w_{st} y_t = -\frac{1}{2}\mathbf{y}^T \mathbf{W} \mathbf{y} \tag{19.18}$$

(The factor of $\frac{1}{2}$ arises because we sum each edge twice.) If $w_{st} = J > 0$, we get a low energy (and hence high probability) if neighboring states agree.

Sometimes there is an **external field**, which is an energy term which is added to each spin. This can be modelled using a local energy term of the form $-\mathbf{b}^T \mathbf{y}$, where $\mathbf{b}$ is sometimes called

3. Ernst Ising was a German-American physicist, 1900–1998.

a **bias term**. The modified distribution is given by

$$\log \tilde{p}(\mathbf{y}) = \sum_{s \sim t} w_{st} y_s y_t + \sum_s b_s y_s = \frac{1}{2}\mathbf{y}^T \mathbf{W}\mathbf{y} + \mathbf{b}^T \mathbf{y} \tag{19.19}$$

where $\boldsymbol{\theta} = (\mathbf{W}, \mathbf{b})$.

If we define $\boldsymbol{\mu} \triangleq -\frac{1}{2}\boldsymbol{\Sigma}^{-1}\mathbf{b}$, $\boldsymbol{\Sigma}^{-1} = -\mathbf{W}$, and $c \triangleq \frac{1}{2}\boldsymbol{\mu}^T \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}$, we can rewrite this in a form that looks similar to a Gaussian:

$$\tilde{p}(\mathbf{y}) \propto \exp(-\frac{1}{2}(\mathbf{y} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\mathbf{y} - \boldsymbol{\mu}) + c) \tag{19.20}$$

One very important difference is that, in the case of Gaussians, the normalization constant, $Z = |2\pi\boldsymbol{\Sigma}|$, requires the computation of a matrix determinant, which can be computed in $O(D^3)$ time, whereas in the case of the Ising model, the normalization constant requires summing over all 2^D bit vectors; this is equivalent to computing the matrix permanent, which is NP-hard in general (Jerrum et al. 2004).

19.4.2 Hopfield networks

A **Hopfield network** (Hopfield 1982) is a fully connected Ising model with a symmetric weight matrix, $\mathbf{W} = \mathbf{W}^T$. These weights, plus the bias terms $\mathbf{b}$, can be learned from training data using (approximate) maximum likelihood, as described in Section 19.5.[4]

The main application of Hopfield networks is as an **associative memory** or **content addressable memory**. The idea is this: suppose we train on a set of fully observed bit vectors, corresponding to patterns we want to memorize. Then, at test time, we present a partial pattern to the network. We would like to estimate the missing variables; this is called **pattern completion**. See Figure 19.7 for an example. This can be thought of as retrieving an example from memory based on a piece of the example itself, hence the term "associative memory".

Since exact inference is intractable in this model, it is standard to use a coordinate descent algorithm known as **iterative conditional modes** (ICM), which just sets each node to its most likely (lowest energy) state, given all its neighbors. The full conditional can be shown to be

$$p(y_s = 1|\mathbf{y}_{-s}, \boldsymbol{\theta}) = \text{sigm}(\mathbf{w}_{s,:}^T \mathbf{y}_{-s} + b_s) \tag{19.21}$$

Picking the most probable state amounts to using the rule $y_s^* = 1$ if $\sum_t w_{st} y_t > b_s$ and using $y_s^* = 0$ otherwise. (Much better inference algorithms will be discussed later in this book.)

Since inference is deterministic, it is also possible to interpret this model as a **recurrent neural network**. (This is quite different from the feedforward neural nets studied in Section 16.5; they are univariate conditional density models of the form $p(y|\mathbf{x}, \boldsymbol{\theta})$ which can only be used for supervised learning.) See Hertz et al. (1991) for further details on Hopfield networks.

A **Boltzmann machine** generalizes the Hopfield / Ising model by including some hidden nodes, which makes the model representationally more powerful. Inference in such models often uses Gibbs sampling, which is a stochastic version of ICM (see Section 24.2 for details).

4. ML estimation works much better than the outer product rule proposed in in (Hopfield 1982), because it not only lowers the energy of the observed patterns, but it also raises the energy of the non-observed patterns, in order to make the distribution sum to one (Hillar et al. 2012).

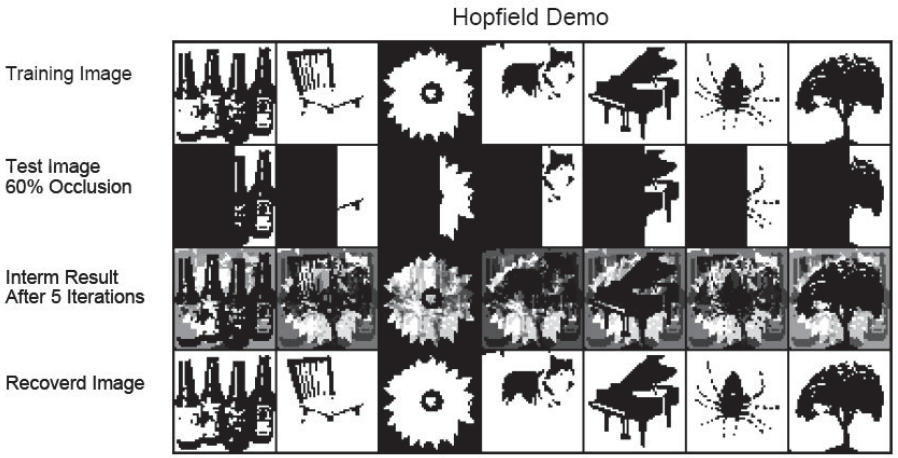
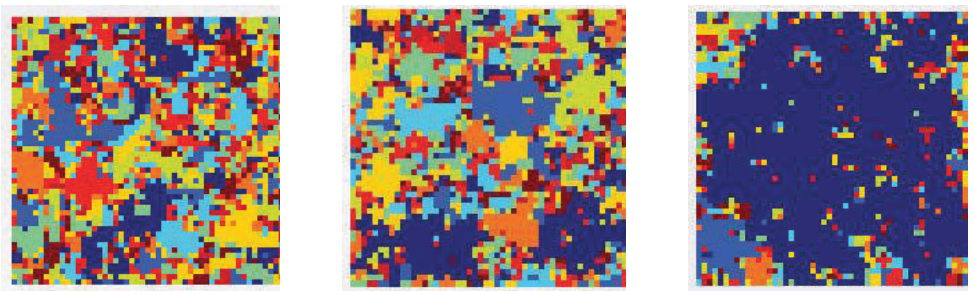


Figure 19.7 Examples of how an associative memory can reconstruct images. These are binary images of size 50×50 pixels. Top: training images. Row 2: partially visible test images. Row 3: estimate after 5 iterations. Bottom: final state estimate. Based on Figure 2.1 of Hertz et al. (1991). Figure generated by `hopfieldDemo`.



(a) (b) (c)

Figure 19.8 Visualizing a sample from a 10-state Potts model of size 128×128 for different association strengths: (a) $J = 1.42$, (b) $J = 1.44$, (c) $J = 1.46$. The regions are labeled according to size: blue is largest, red is smallest. Used with kind permission of Erik Sudderth. See `gibbsDemoIsing` for Matlab code to produce a similar plot for the Ising model.

However, we could equally well apply Gibbs to a Hopfield net and ICM to a Boltzmann machine: the inference algorithm is not part of the model definition. See Section 27.7 for further details on Boltzmann machines.

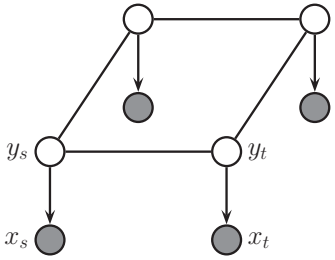


Figure 19.9 A grid-structured MRF with local evidence nodes.

19.4.3 Potts model

It is easy to generalize the Ising model to multiple discrete states, $y_t \in \{1, 2, \ldots, K\}$. It is common to use a potential function of the following form:

$$\psi_{st}(y_s, y_t) = \begin{pmatrix} e^J & 0 & 0 \\ 0 & e^J & 0 \\ 0 & 0 & e^J \end{pmatrix} \tag{19.22}$$

This is called the **Potts model**.[5] If $J > 0$, then neighboring nodes are encouraged to have the same label. Some samples from this model are shown in Figure 19.8. We see that for $J > 1.44$, large clusters occur, for $J < 1.44$, many small clusters occur, and at the **critical value** of $K = 1.44$, there is a mix of small and large clusters. This rapid change in behavior as we vary a parameter of the system is called a **phase transition**, and has been widely studied in the physics community. An analogous phenomenon occurs in the Ising model; see (MacKay 2003, ch 31) for details.

The Potts model can be used as a prior for **image segmentation**, since it says that neighboring pixels are likely to have the same discrete label and hence belong to the same segment. We can combine this prior with a likelihood term as follows:

$$p(\mathbf{y}, \mathbf{x}|\boldsymbol{\theta}) = p(\mathbf{y}|J) \prod_t p(x_t|y_t, \boldsymbol{\theta}) = \left[\frac{1}{Z(J)} \prod_{s \sim t} \psi(y_s, y_t; J) \right] \prod_t p(x_t|y_t, \boldsymbol{\theta}) \tag{19.23}$$

where $p(x_t|y_t = k, \boldsymbol{\theta})$ is the probability of observing pixel x_t given that the corresponding segment belongs to class k. This observation model can be modeled using a Gaussian or a non-parametric density. (Note that we label the hidden nodes y_t and the observed nodes x_t, to be compatible with Section 19.6.)

The corresponding graphical model is a mix of undirected and directed edges, as shown in Figure 19.9. The undirected 2d lattice represents the prior $p(\mathbf{y})$; in addition, there are directed edge from each y_t to its corresponding x_t, representing the **local evidence**. Technically speaking, this combination of an undirected and directed graph is called a **chain graph**. However,

5. Renfrey Potts was an Australian mathematician, 1925–2005.

since the x_t nodes are observed, they can be "absorbed" into the model, thus leaving behind an undirected "backbone".

This model is a 2d analog of an HMM, and could be called a **partially observed MRF**. As in an HMM, the goal is to perform posterior inference, i.e., to compute (some function of) $p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta})$. Unfortunately, the 2d case is provably much harder than the 1d case, and we must resort to approximate methods, as we discuss in later chapters.

Although the Potts prior is adequate for regularizing supervised learning problems, it is not sufficiently accurate to perform image segmentation in an unsupervised way, since the segments produced by this model do not accurately represent the kinds of segments one sees in natural images (Morris et al. 1996).[6] For the unsupervised case, one needs to use more sophisticated priors, such as the truncated Gaussian process prior of (Sudderth and Jordan 2008).

19.4.4 Gaussian MRFs

An undirected GGM, also called a **Gaussian MRF** (see e.g., (Rue and Held 2005)), is a pairwise MRF of the following form:

$$p(\mathbf{y}|\boldsymbol{\theta}) \propto \prod_{s \sim t} \psi_{st}(y_s, y_t) \prod_t \psi_t(y_t) \tag{19.24}$$

$$\psi_{st}(y_s, y_t) = \exp(-\frac{1}{2} y_s \Lambda_{st} y_t) \tag{19.25}$$

$$\psi_t(y_t) = \exp(-\frac{1}{2}\Lambda_{tt} y_t^2 + \eta_t y_t) \tag{19.26}$$

(Note that we could easily absorb the node potentials ψ_t into the edge potentials, but we have kept them separate for clarity.) The joint distribution can be written as follows:

$$p(\mathbf{y}|\boldsymbol{\theta}) \propto \exp[\boldsymbol{\eta}^T \mathbf{y} - \frac{1}{2}\mathbf{y}^T \boldsymbol{\Lambda} \mathbf{y}] \tag{19.27}$$

We recognize this as a multivariate Gaussian written in **information form** where $\boldsymbol{\Lambda} = \boldsymbol{\Sigma}^{-1}$ and $\boldsymbol{\eta} = \boldsymbol{\Lambda}\boldsymbol{\mu}$.

If $\Lambda_{st} = 0$, then there is no pairwise term connecting s and t, so by the factorization theorem (Theorem 2.2.1), we conclude that

$$y_s \perp y_t | \mathbf{y}_{-(st)} \iff \Lambda_{st} = 0 \tag{19.28}$$

The zero entries in $\boldsymbol{\Lambda}$ are called **structural zeros**, since they represent the absent edges in the graph. Thus undirected GGMs correspond to sparse precision matrices, a fact which we exploit in Section 26.7.2 to efficiently learn the structure of the graph.

19.4.4.1 Comparing Gaussian DGMs and UGMs *

In Section 10.2.5, we saw that directed GGMs correspond to sparse regression matrices, and hence sparse Cholesky factorizations of covariance matrices, whereas undirected GGMs correspond to

6. An influential paper (Geman and Geman 1984), which introduced the idea of a Gibbs sampler (Section 24.2), proposed using the Potts model as a prior for image segmentation, but the results in their paper are misleading because they did not run their Gibbs sampler for long enough. See Figure 24.10 for a vivid illustration of this point.

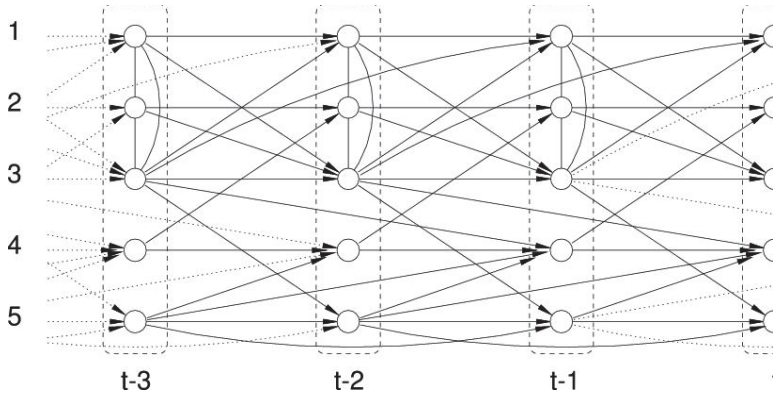


Figure 19.10 A VAR(2) process represented as a dynamic chain graph. Source: (Dahlhaus and Eichler 2000). Used with kind permission of Rainer Dahlhaus and Oxford University Press.

sparse precision matrices. The advantage of the DAG formulation is that we can make the regression weights $\mathbf{W}$, and hence $\boldsymbol{\Sigma}$, be conditional on covariate information (Pourahmadi 2004), without worrying about positive definite constraints. The disadavantage of the DAG formulation is its dependence on the order, although in certain domains, such as time series, there is already a natural ordering of the variables.

It is actually possible to combine both representations, resulting in a Gaussian chain graph. For example, consider a a discrete-time, second-order Markov chain in which the states are continuous, $\mathbf{y}_t \in \mathbb{R}^D$. The transition function can be represented as a (vector-valued) linear-Gaussian CPD:

$$p(\mathbf{y}_t|\mathbf{y}_{t-1}, \mathbf{y}_{t-2}, \boldsymbol{\theta}) = \mathcal{N}(\mathbf{y}_t|\mathbf{A}_1\mathbf{y}_{t-1} + \mathbf{A}_2\mathbf{y}_{t-2}, \boldsymbol{\Sigma}) \tag{19.29}$$

This is called **vector auto-regressive** or **VAR** process of order 2. Such models are widely used in econometrics for time-series forecasting.

The time series aspect is most naturally modeled using a DGM. However, if $\boldsymbol{\Sigma}^{-1}$ is sparse, then the correlation amongst the components within a time slice is most naturally modeled using a UGM. For example, suppose we have

$$\mathbf{A}_1 = \begin{pmatrix} \frac{3}{5} & 0 & \frac{1}{5} & 0 & 0 \\ 0 & \frac{3}{5} & 0 & -\frac{1}{5} & 0 \\ \frac{2}{5} & \frac{1}{3} & \frac{3}{5} & 0 & 0 \\ 0 & 0 & 0 & -\frac{1}{2} & \frac{1}{5} \\ 0 & 0 & \frac{1}{5} & 0 & \frac{2}{5} \end{pmatrix}, \; \mathbf{A}_2 = \begin{pmatrix} 0 & 0 & -\frac{1}{5} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{5} & 0 & \frac{1}{3} \\ 0 & 0 & 0 & 0 & -\frac{1}{5} \end{pmatrix} \tag{19.30}$$

and

$$\boldsymbol{\Sigma} = \begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} & 0 & 0 \\ \frac{1}{2} & 1 & -\frac{1}{3} & 0 & 0 \\ \frac{1}{3} & -\frac{1}{3} & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}, \; \boldsymbol{\Sigma}^{-1} = \begin{pmatrix} 2.13 & -1.47 & -1.2 & 0 & 0 \\ -1.47 & 2.13 & 1.2 & 0 & 0 \\ -1.2 & 1.2 & 1.8 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{19.31}$$

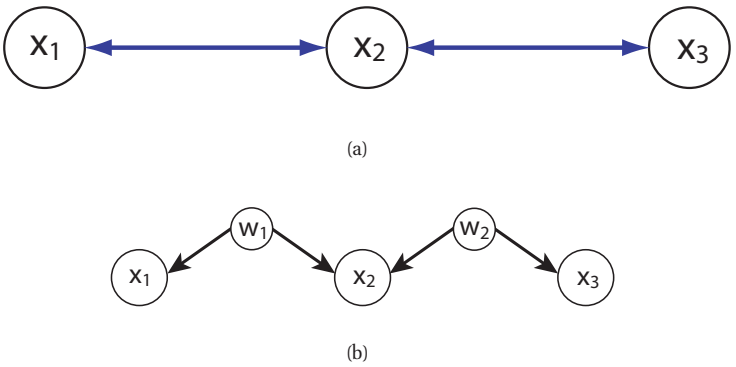


Figure 19.11 (a) A bi-directed graph. (b) The equivalent DAG. Here the w nodes are latent confounders. Based on Figures 5.12-5.13 of (Choi 2011). Used with kind permission of Myung Choi.

The resulting graphical model is illustrated in Figure 19.10. Zeros in the transition matrices $\mathbf{A}_1$ and $\mathbf{A}_2$ correspond to absent directed arcs from $\mathbf{y}_{t-1}$ and $\mathbf{y}_{t-2}$ into $\mathbf{y}_t$. Zeros in the precision matrix $\mathbf{\Sigma}^{-1}$ correspond to absent undirected arcs between nodes in $\mathbf{y}_t$.

Sometimes we have a sparse covariance matrix rather than a sparse precision matrix. This can be represented using a **bi-directed graph**, where each edge has arrows in both directions, as in Figure 19.11(a). Here nodes that are not connected are unconditionally independent. For example in Figure 19.11(a) we see that $Y_1 \perp Y_3$. In the Gaussian case, this means $\Sigma_{1,3} = \Sigma_{3,1} = 0$. (A graph representing a sparse covariance matrix is called a **covariance graph**.) By contrast, if this were an undirected model, we would have that $Y_1 \perp Y_3|Y_2$, and $\Lambda_{1,3} = \Lambda_{3,1} = 0$, where $\mathbf{\Lambda} = \mathbf{\Sigma}^{-1}$.

A bidirected graph can be converted to a DAG with latent variables, where each bidirected edge is replaced with a hidden variable representing a hidden common cause, or **confounder**, as illustrated in Figure 19.11(b). The relevant CI properties can then be determined using d-separation.

We can combine bidirected and directed edges to get a **directed mixed graphical model**. This is useful for representing a variety of models, such as ARMA models (Section 18.2.4.4), structural equation models (Section 26.5.5), etc.

19.4.5 Markov logic networks *

In Section 10.2.2, we saw how we could "unroll" Markov models and HMMs for an arbitrary number of time steps in order to model variable-length sequences. Similarly, in Section 19.4.1, we saw how we could expand a lattice UGM to model images of any size. What about more complex domains, where we have a variable number of objects and relationships between them? Creating models for such scenarios is often done using **first-order logic** (see e.g., (Russell and Norvig 2010)). For example, consider the sentences "Smoking causes cancer" and "If two people are friends, and one smokes, then so does the other". We can write these sentences in first-order

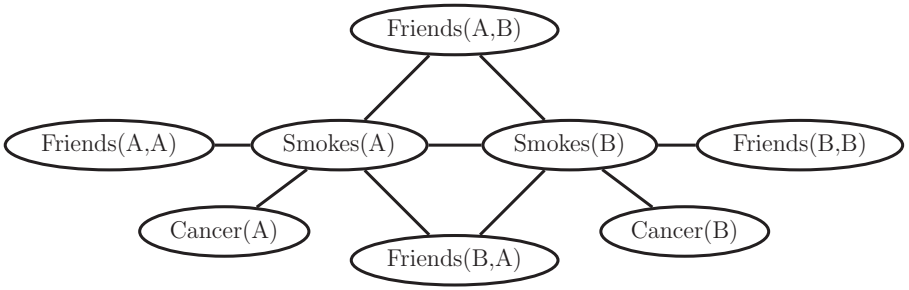


Figure 19.12 An example of a ground Markov logic network represented as a pairwise MRF for 2 people. Based on Figure 2.1 from (Domingos and Lowd 2009). Used with kind permission of Pedro Domingos.

logic as follows:

$$\forall x.Sm(x) \implies Ca(x) \tag{19.32}$$

$$\forall x.\forall y.Fr(x,y) \wedge Sm(x) \implies Sm(y) \tag{19.33}$$

where Sm and Ca are predicates, and Fr is a relation.[7]

Of course, such rules are not always true. Indeed, this brittleness is the main reason why logical approaches to AI are no longer widely used, at least not in their pure form. There have been a variety of attempts to combine first order logic with probability theory, an area known as **statistical relational AI** or **probabilistic relational modeling** (Kersting et al. 2011). One simple approach is to take logical rules and attach weights (known as **certainty factors**) to them, and then to interpret them as conditional probability distributions. For example, we might say $p(Ca(x) = 1|Sm(x) = 1) = 0.9$. Unfortunately, the rule does not say what to predict if $Sm(x) = 0$. Furthermore, combining CPDs in this way is not guaranteed to define a consistent joint distribution, because the resulting graph may not be a DAG.

An alternative approach is to treat these rules as a way of defining potential functions in an unrolled UGM. The result is known as a **Markov logic network** (Domingos and Lowd 2009). To specify the network, we first rewrite all the rules in **conjunctive normal form** (CNF), also known as **clausal form**. In this case, we get

$$\neg Sm(x) \vee Ca(x) \tag{19.34}$$

$$\neg Fr(x,y) \vee \neg Sm(x) \vee Sm(y) \tag{19.35}$$

The first clause can be read as "Either x does not smoke or he has cancer", which is logically equivalent to Equation 19.32. (Note that in a clause, any unbound variable, such as x, is assumed to be universally quantified.)

7. A predicate is just a function of one argument, known as an object, that evaluates to true or false, depending on whether the property holds or not of that object. A (logical) relation is just a function of two or more arguments (objects) that evaluates to true or false, depending on whether the relationship holds between that set of objects or not.

Inference in first-order logic is only semi-decidable, so it is common to use a restricted subset. A common approach (as used in Prolog) is to restrict the language to **Horn clauses**, which are clauses that contain at most one positive literal. Essentially this means the model is a series of if-then rules, where the right hand side of the rules (the "then" part, or consequence) has only a single term.

Once we have encoded our **knowledge base** as a set of clauses, we can attach weights to each one; these weights are the parameter of the model, and they define the clique potentials as follows:

$$\psi_c(\mathbf{x}_c) = \exp(w_c \phi_c(\mathbf{x}_c)) \tag{19.36}$$

where $\phi_c(\mathbf{x}_c)$ is a logical expression which evaluates clause c applied to the variables $\mathbf{x}_c$, and w_c is the weight we attach to this clause. Roughly speaking, the weight of a clause specifies the probability of a world in which this clause is satsified relative to a world in which it is not satisfied.

Now suppose there are two objects (people) in the world, Anna and Bob, which we will denote by **constant symbols** A and B. We can make a **ground network** from the above clauses by creating binary random variables S_x, C_x, and $F_{x,y}$ for $x, y \in \{A, B\}$, and then "wiring these up" according to the clauses above. The result is the UGM in Figure 19.12 with 8 binary nodes. Note that we have not encoded the fact that Fr is a symmetric relation, so $Fr(A, B)$ and $Fr(B, A)$ might have different values. Similarly, we have the "degenerate" nodes $Fr(A, A)$ and $Fr(B, B)$, since we did not enforce $x \neq y$ in Equation 19.33. (If we add such constraints, then the model compiler, which generates the ground network, could avoid creating redundant nodes.)

In summary, we can think of MLNs as a convenient way of specifying a UGM **template**, that can get unrolled to handle data of arbitrary size. There are several other ways to define relational probabilistic models; see e.g., (Koller and Friedman 2009; Kersting et al. 2011) for details. In some cases, there is uncertainty about the number or existence of objects or relations (the so-called **open universe** problem). Section 18.6.2 gives a concrete example in the context of multi-object tracking. See e.g., (Russell and Norvig 2010; Kersting et al. 2011) and references therein for further details.

19.5 Learning

In this section, we discuss how to perform ML and MAP parameter estimation for MRFs. We will see that this is quite computationally expensive. For this reason, it is rare to perform Bayesian inference for the parameters of MRFs (although see (Qi et al. 2005)).

19.5.1 Training maxent models using gradient methods

Consider an MRF in log-linear form:

$$p(\mathbf{y}|\boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})} \exp\left(\sum_c \boldsymbol{\theta}_c^T \boldsymbol{\phi}_c(\mathbf{y})\right) \tag{19.37}$$

where c indexes the cliques. The scaled log-likelihood is given by

$$\ell(\boldsymbol{\theta}) \triangleq \frac{1}{N}\sum_i \log p(\mathbf{y}_i|\boldsymbol{\theta}) = \frac{1}{N}\sum_i \left[\sum_c \boldsymbol{\theta}_c^T \boldsymbol{\phi}_c(\mathbf{y}_i) - \log Z(\boldsymbol{\theta})\right] \tag{19.38}$$

Since MRFs are in the exponential family, we know that this function is convex in $\boldsymbol{\theta}$ (see Section 9.2.3), so it has a unique global maximum which we can find using gradient-based optimizers. In particular, the derivative for the weights of a particular clique, c, is given by

$$\frac{\partial \ell}{\partial \boldsymbol{\theta}_c} = \frac{1}{N}\sum_i \left[\boldsymbol{\phi}_c(\mathbf{y}_i) - \frac{\partial}{\partial \boldsymbol{\theta}_c}\log Z(\boldsymbol{\theta})\right] \tag{19.39}$$

Exercise 19.1 asks you to show that the derivative of the log partition function wrt $\boldsymbol{\theta}_c$ is the expectation of the c'th feature under the model, i.e.,

$$\frac{\partial \log Z(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_c} = \mathbb{E}\left[\boldsymbol{\phi}_c(\mathbf{y})|\boldsymbol{\theta}\right] = \sum_{\mathbf{y}} \boldsymbol{\phi}_c(\mathbf{y}) p(\mathbf{y}|\boldsymbol{\theta}) \tag{19.40}$$

Hence the gradient of the log likelihood is

$$\frac{\partial \ell}{\partial \boldsymbol{\theta}_c} = \left[\frac{1}{N}\sum_i \boldsymbol{\phi}_c(\mathbf{y}_i)\right] - \mathbb{E}\left[\boldsymbol{\phi}_c(\mathbf{y})\right] \tag{19.41}$$

In the first term, we fix $\mathbf{y}$ to its observed values; this is sometimes called the **clamped term**. In the second term, $\mathbf{y}$ is free; this is sometimes called the **unclamped term** or **contrastive term**. Note that computing the unclamped term requires inference in the model, and this must be done once per gradient step. This makes UGM training much slower than DGM training.

The gradient of the log likelihood can be rewritten as the expected feature vector according to the empirical distribution minus the model's expectation of the feature vector:

$$\frac{\partial \ell}{\partial \boldsymbol{\theta}_c} = \mathbb{E}_{p_{\text{emp}}}\left[\boldsymbol{\phi}_c(\mathbf{y})\right] - \mathbb{E}_{p(\cdot|\boldsymbol{\theta})}\left[\boldsymbol{\phi}_c(\mathbf{y})\right] \tag{19.42}$$

At the optimum, the gradient will be zero, so the empirical distribution of the features will match the model's predictions:

$$\mathbb{E}_{p_{\text{emp}}}\left[\boldsymbol{\phi}_c(\mathbf{y})\right] = \mathbb{E}_{p(\cdot|\boldsymbol{\theta})}\left[\boldsymbol{\phi}_c(\mathbf{y})\right] \tag{19.43}$$

This is called **moment matching**. This observation motivates a different optimization algorithm which we discuss in Section 19.5.7.

19.5.2 Training partially observed maxent models

Suppose we have missing data and/or hidden variables in our model. In general, we can represent such models as follows:

$$p(\mathbf{y}, \mathbf{h}|\boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})}\exp(\sum_c \boldsymbol{\theta}_c^T \boldsymbol{\phi}_c(\mathbf{h}, \mathbf{y})) \tag{19.44}$$

The log likelihood has the form

$$\ell(\boldsymbol{\theta}) = \frac{1}{N}\sum_i \log\left(\sum_{\mathbf{h}_i} p(\mathbf{y}_i, \mathbf{h}_i|\boldsymbol{\theta})\right) = \frac{1}{N}\sum_i \log\left(\frac{1}{Z(\boldsymbol{\theta})}\sum_{\mathbf{h}_i} \tilde{p}(\mathbf{y}_i, \mathbf{h}_i|\boldsymbol{\theta})\right) \tag{19.45}$$

where

$$\tilde{p}(\mathbf{y}, \mathbf{h}|\boldsymbol{\theta}) \triangleq \exp\left(\sum_c \boldsymbol{\theta}_c^T \boldsymbol{\phi}_c(\mathbf{h}, \mathbf{y})\right) \tag{19.46}$$

is the unnormalized distribution. The term $\sum_{\mathbf{h}_i} \tilde{p}(\mathbf{y}_i, \mathbf{h}_i|\boldsymbol{\theta})$ is the same as the partition function for the whole model, except that $\mathbf{y}$ is fixed at $\mathbf{y}_i$. Hence the gradient is just the expected features where we clamp $\mathbf{y}_i$, but average over $\mathbf{h}$:

$$\frac{\partial}{\partial \boldsymbol{\theta}_c} \log\left(\sum_{\mathbf{h}_i} \tilde{p}(\mathbf{y}_i, \mathbf{h}_i|\boldsymbol{\theta})\right) = \mathbb{E}\left[\boldsymbol{\phi}_c(\mathbf{h}, \mathbf{y}_i)|\boldsymbol{\theta}\right] \tag{19.47}$$

So the overall gradient is given by

$$\frac{\partial \ell}{\partial \boldsymbol{\theta}_c} = \frac{1}{N}\sum_i \left\{\mathbb{E}\left[\boldsymbol{\phi}_c(\mathbf{h}, \mathbf{y}_i)|\boldsymbol{\theta}\right] - \mathbb{E}\left[\boldsymbol{\phi}_c(\mathbf{h}, \mathbf{y})|\boldsymbol{\theta}\right]\right\} \tag{19.48}$$

The first set of expectations are computed by "clamping" the visible nodes to their observed values, and the second set are computed by letting the visible nodes be free. In both cases, we marginalize over $\mathbf{h}_i$.

An alternative approach is to use generalized EM, where we use gradient methods in the M step. See (Koller and Friedman 2009, p956) for details.

19.5.3 Approximate methods for computing the MLEs of MRFs

When fitting a UGM there is (in general) no closed form solution for the ML or the MAP estimate of the parameters, so we need to use gradient-based optimizers. This gradient requires inference. In models where inference is intractable, learning also becomes intractable. This has motivated various computationally faster alternatives to ML/MAP estimation, which we list in Table 19.1. We dicsuss some of these alternatives below, and defer others to later sections.

19.5.4 Pseudo likelihood

One alternative to MLE is to maximize the **pseudo likelihood** (Besag 1975), defined as follows:

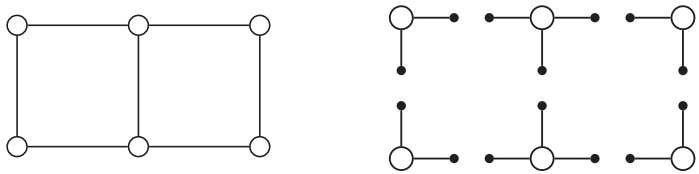
$$\ell_{PL}(\boldsymbol{\theta}) \triangleq \sum_{\mathbf{y}}\sum_{d=1}^{D} p_{\text{emp}}(\mathbf{y} \log p(y_d|\mathbf{y}_{-d}) = \frac{1}{N}\sum_{i=1}^{N}\sum_{d=1}^{D} \log p(y_{id}|\mathbf{y}_{i,-d}, \boldsymbol{\theta}) \tag{19.49}$$

That is, we optimize the product of the full conditionals, also known as the **composite likelihood** (Lindsay 1988), Compare this to the objective for maximum likelihood:

$$\ell_{ML}(\boldsymbol{\theta}) = \sum_{\mathbf{y},\mathbf{x}} p_{\text{emp}}(\mathbf{y} \log p(\mathbf{y}|\boldsymbol{\theta}) = \sum_{i=1}^{N} \log p(\mathbf{y}_i|\boldsymbol{\theta}) \tag{19.50}$$

Method	Restriction	Exact MLE?	Section
Closed form	Only Chordal MRF	Exact	Section 19.5.7.4
IPF	Only Tabular / Gaussian MRF	Exact	Section 19.5.7
Gradient-based optimization	Low tree width	Exact	Section 19.5.1
Max-margin training	Only CRFs	N/A	Section 19.7
Pseudo-likelihood	No hidden variables	Approximate	Section 19.5.4
Stochastic ML	-	Exact (up to MC error)	Section 19.5.5
Contrastive divergence	-	Approximate	Section 27.7.2.4
Minimum probability flow	Can integrate out the hiddens	Approximate	Sohl-Dickstein et al. (2011)

Table 19.1 Some methods that can be used to compute approximate ML/ MAP parameter estimates for MRFs/ CRFs. Low tree-width means that, in order for the method to be efficient, the graph must "tree-like"; see Section 20.5 for details.



(a) (b)

Figure 19.13 (a) A small 2d lattice. (b) The representation used by pseudo likelihood. Solid nodes are observed neighbors. Based on Figure 2.2 of (Carbonetto 2003).

In the case of Gaussian MRFs, PL is equivalent to ML (Besag 1975), but this is not true in general (Liang and Jordan 2008).

The PL approach is illustrated in Figure 19.13 for a 2d grid. We learn to predict each node, given all of its neighbors. This objective is generally fast to compute since each full conditional $p(y_{id}|\mathbf{y}_{i,-d}, \boldsymbol{\theta})$ only requires summing over the states of a single node, y_{id}, in order to compute the local normalization constant. The PL approach is similar to fitting each full conditional separately (which is the method used to train dependency networks, discussed in Section 26.2.2), except that the parameters are tied between adjacent nodes.

One problem with PL is that it is hard to apply to models with hidden variables (Parise and Welling 2005). Another more subtle problem is that each node assumes that its neighbors have known values. If node $t \in \mathrm{nbr}(s)$ is a perfect predictor for node s, then s will learn to rely completely on node t, even at the expense of ignoring other potentially useful information, such as its local evidence.

However, experiments in (Parise and Welling 2005; Hoefling and Tibshirani 2009) suggest that PL works as well as exact ML for fully observed Ising models, and of course PL is *much* faster.

19.5.5 Stochastic maximum likelihood

Recall that the gradient of the log-likelihood for a fully observed MRF is given by

$$\nabla_{\boldsymbol{\theta}} \ell(\boldsymbol{\theta}) = \frac{1}{N} \sum_i [\boldsymbol{\phi}(\mathbf{y}_i) - \mathbb{E}[\boldsymbol{\phi}(\mathbf{y})]] \tag{19.51}$$

The gradient for a partially observed MRF is similar. In both cases, we can approximate the model expectations using Monte Carlo sampling. We can combine this with stochastic gradient descent (Section 8.5.2), which takes samples from the empirical distribution. Pseudocode for the resulting method is shown in Algorithm 3.

Algorithm 19.1: Stochastic maximum likelihood for fitting an MRF

Initialize weights $\boldsymbol{\theta}$ randomly;
$k = 0$, $\eta = 1$;
for *each epoch* **do**
  for *each minibatch of size* B **do**
    for *each sample* $s = 1 : S$ **do**
      Sample $\mathbf{y}^{s,k} \sim p(\mathbf{y}|\boldsymbol{\theta}_k)$;
    $\hat{E}(\boldsymbol{\phi}(\mathbf{y})) = \frac{1}{S}\sum_{s=1}^{S} \boldsymbol{\phi}(\mathbf{y}^{s,k})$;
    for *each training case* i *in minibatch* **do**
      $\mathbf{g}_{ik} = \boldsymbol{\phi}(\mathbf{y}_i) - \hat{E}(\boldsymbol{\phi}(\mathbf{y}))$;
    $\mathbf{g}_k = \frac{1}{B}\sum_{i \in B} \mathbf{g}_{ik}$;
    $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta \mathbf{g}_k$;
    $k = k + 1$;
    Decrease step size η;

Typically we use MCMC to generate the samples. Of course, running MCMC to convergence at each step of the inner loop would be extremely slow. Fortunately, it was shown in (Younes 1989) that we can start the MCMC chain at its previous value, and just take a few steps. In otherwords, we sample $\mathbf{y}^{s,k}$ by initializing the MCMC chain at $\mathbf{y}^{s,k-1}$, and then run for a few iterations. This is valid since $p(\mathbf{y}|\boldsymbol{\theta}^k)$ is likely to be close to $p(\mathbf{y}|\boldsymbol{\theta}^{k-1})$, since we only changed the parameters a small amount. We call this algorithm **stochastic maximum likelihood** or **SML**. (There is a closely related algorithm called persistent contrastive divergence which we discuss in Section 27.7.2.5.)

19.5.6 Feature induction for maxent models *

MRFs require a good set of features. One unsupervised way to learn such features, known as **feature induction**, is to start with a base set of features, and then to continually create new feature combinations out of old ones, greedily adding the best ones to the model. This approach was first proposed in (Pietra et al. 1997; Zhu et al. 1997), and was later extended to the CRF case in (McCallum 2003).

To illustrate the basic idea, we present an example from (Pietra et al. 1997), which described how to build unconditional probabilistic models to represent English spelling. Initially the model has no features, which represents the uniform distribution. The algorithm starts by choosing to add the feature

$$\phi_1(\mathbf{y}) = \sum_t \mathbb{I}(y_t \in \{a, \ldots, z\}) \tag{19.52}$$

which checks if any letter is lower case or not. After the feature is added, the parameters are (re)-fit by maximum likelihood. For this feature, it turns out that $\hat{\theta}_1 = 1.944$, which means that a word with a lowercase letter in any position is about $e^{1.944} \approx 7$ times more likely than the same word without a lowercase letter in that position. Some samples from this model, generated using (annealed) Gibbs sampling (Section 24.2), are shown below.[8]

```
m, r, xevo, ijjiir, b, to, jz, gsr, wq, vf, x, ga, msmGh, pcp, d, oziVlal,
hzagh, yzop, io, advzmxnv, ijv_bolft, x, emx, kayerf, mlj, rawzyb, jp, ag,
ctdnnnbg, wgdw, t, kguv, cy, spxcq, uzflbbf, dxtkkn, cxwx, jpd, ztzh, lv,
zhpkvnu, l^, r, qee, nynrx, atze4n, ik, se, w, lrh, hp+, yrqyka'h, zcngotcnx,
igcump, zjcjs, lqpWiqu, cefmfhc, o, lb, fdcY, tzby, yopxmvk, by, fz,, t, govyccm,
ijyiduwfzo, 6xr, duh, ejv, pk, pjw, l, fl, w
```

The second feature added by the algorithm checks if two adjacent characters are lower case:

$$\phi_2(\mathbf{y}) = \sum_{s \sim t} \mathbb{I}(y_s \in \{a, \ldots, z\}, y_t \in \{a, \ldots, z\}) \tag{19.53}$$

Now the model has the form

$$p(\mathbf{y}) = \frac{1}{Z} \exp(\theta_1 \phi_1(\mathbf{y}) + \theta_2 \phi_2(\mathbf{y})) \tag{19.54}$$

Continuing in this way, the algorithm adds features for the strings `s>` and `ing>`, where $>$ represents the end of word, and for various regular expressions such as `[0-9]`, etc. Some samples from the model with 1000 features, generated using (annealed) Gibbs sampling, are shown below.

```
was, reaser, in, there, to, will, ,, was, by, homes, thing, be, reloverated,
ther, which, conists, at, fores, anditing, with, Mr., proveral, the, ,, ***,
on't, prolling, prothere, ,, mento, at, yaou, 1, chestraing, for, have, to,
intrally, of, qut, ., best, compers, ***, cluseliment, uster, of, is, deveral,
this, thise, of, offect, inatever, thifer, constranded, stater, vill, in, thase,
in, youse, menttering, and, ., of, in, verate, of, to
```

This approach of feature learning can be thought of as a form of graphical model structure learning (Chapter 26), except it is more fine-grained: we add features that are useful, regardless of the resulting graph structure. However, the resulting graphs can become densely connected, which makes inference (and hence parameter estimation) intractable.

19.5.7 Iterative proportional fitting (IPF) *

Consider a pairwise MRF where the potentials are represented as tables, with one parameter per variable setting. We can represent this in log-linear form using

$$\psi_{st}(y_s, y_t) = \exp\left(\boldsymbol{\theta}_{st}^T [\mathbb{I}(y_s = 1, y_t = 1), \ldots, \mathbb{I}(y_s = K, y_t = K)]\right) \tag{19.55}$$

and similarly for $\psi_t(y_t)$. Thus the feature vectors are just indicator functions.

8. We thank John Lafferty for sharing this example.

From Equation 19.43, we have that, at the maximum of the likelihood, the empirical expectation of the features equals the model's expectation:

$$\mathbb{E}_{p_{\text{emp}}}\left[\mathbb{I}(y_s = j, y_t = k)\right] = \mathbb{E}_{p(\cdot|\boldsymbol{\theta})}\left[\mathbb{I}(y_s = j, y_t = k)\right] \tag{19.56}$$

$$p_{\text{emp}}(y_s = j, y_t = k) = p(y_s = j, y_t = k|\boldsymbol{\theta}) \tag{19.57}$$

where p_{emp} is the empirical probability:

$$p_{\text{emp}}(y_s = j, y_t = k) = \frac{N_{st,jk}}{N} = \frac{\sum_{n=1}^{N} \mathbb{I}(y_{ns} = j, y_{nt} = k)}{N} \tag{19.58}$$

For a general graph, the condition that must hold at the optimum is

$$p_{\text{emp}}(\mathbf{y}_c) = p(\mathbf{y}_c|\boldsymbol{\theta}) \tag{19.59}$$

For a special family of graphs known as decomposable graphs (defined in Section 20.4.1), one can show that $p(\mathbf{y}_c|\boldsymbol{\theta}) = \psi_c(\mathbf{y}_c)$. However, even if the graph is not decomposable, we can imagine trying to enforce this condition. This suggests an iterative coordinate ascent scheme where at each step we compute

$$\psi_c^{t+1}(\mathbf{y}_c) = \psi_c^t(\mathbf{y}_c) \times \frac{p_{\text{emp}}(\mathbf{y}_c)}{p(\mathbf{y}_c|\psi^t)} \tag{19.60}$$

where the multiplication is elementwise. This is known as **iterative proportional fitting** or **IPF** (Fienberg 1970; Bishop et al. 1975). See Algorithm 7 for the pseudocode.

Algorithm 19.2: Iterative Proportional Fitting algorithm for tabular MRFs

```
Initialize ψ_c = 1 for c = 1 : C;
repeat
    for c = 1 : C do
        p_c = p(y_c|ψ);
        p̂_c = p_emp(y_c);
        ψ_c = ψ_c * p̂_c / p_c ;
until converged;
```

19.5.7.1 Example

Let us consider a simple example from `http://en.wikipedia.org/wiki/Iterative_proportional_fitting`. We have two binary variables, Y_1 and Y_2, where $Y_{n1} = 1$ if man n is left handed, and $Y_{n1} = 0$ otherwise; similarly, $Y_{n2} = 1$ if woman n is left handed, and $Y_{n2} = 0$ otherwise. We can summarize the data using the following 2×2 contingency table:

	right-handed	left-handed	Total
male	43	9	52
female	44	4	48
Total	87	13	100

Suppose we want to fit a disconnected graphical model containing nodes Y_1 and Y_2 but with no edge between them. That is, we want to find vectors $\boldsymbol{\psi}_1$ and $\boldsymbol{\psi}_2$ such that $\mathbf{M} \triangleq \boldsymbol{\psi}_1 \boldsymbol{\psi}_2^T \approx \mathbf{C}$, where $\mathbf{M}$ are the model's expected counts, and $\mathbf{C}$ are the empirical counts. By moment matching, we find that the row and column sums of the model must exactly match the row and column sums of the data. One possible solution is to use $\boldsymbol{\psi}_1 = [0.5200, 0.4800]$ and $\boldsymbol{\psi}_2 = [87, 13]$. Below we show the model's predictions, $\mathbf{M} = \boldsymbol{\psi}_1 \boldsymbol{\psi}_2^T$.

	right-handed	left-handed	Total
male	45.24	6.76	52
female	41.76	6.24	48
Total	87	13	100

It is easy to see that this matches the required constraints. See `IPFdemo2x2` for some Matlab code that computes these numbers. This method is easily to generalized to arbitrary graphs.

19.5.7.2 Speed of IPF

IPF is a fixed point algorithm for enforcing the moment matching constraints and is guaranteed to converge to the global optimum (Bishop et al. 1975). The number of iterations depends on the form of the model. If the graph is decomposable, then IPF converges in a single iteration, but in general, IPF may require many iterations.

It is clear that the dominant cost of IPF is computing the required marginals under the model. Efficient methods, such as the junction tree algorithm (Section 20.4), can be used, resulting in something called **efficient IPF** (Jirousek and Preucil 1995).

Nevertheless, coordinate descent can be slow. An alternative method is to update all the parameters at once, by simply following the gradient of the likelihood. This gradient approach has the further significant advantage that it works for models in which the clique potentials may not be fully parameterized, i.e., the features may not consist of all possible indicators for each clique, but instead can be arbitrary. Although it is possible to adapt IPF to this setting of general features, resulting in a method known as **iterative scaling**, in practice the gradient method is much faster (Malouf 2002; Minka 2003).

19.5.7.3 Generalizations of IPF

We can use IPF to fit Gaussian graphical models: instead of working with empirical counts, we work with empirical means and covariances (Speed and Kiiveri 1986). It is also possible to create a Bayesian IPF algorithm for sampling from the posterior of the model's parameters (see e.g., (Dobra and Massam 2010)).

19.5.7.4 IPF for decomposable graphical models

There is a special family of undirected graphical models known as decomposable graphical models. This is formally defined in Section 20.4.1, but the basic idea is that it contains graphs which are "tree-like". Such graphs can be represented by UGMs or DGMs without any loss of information.

In the case of decomposable graphical models, IPF converges in one iteration. In fact, the

MLE has a closed form solution (Lauritzen 1996). In particular, for tabular potentials we have

$$\hat{\psi}_c(\mathbf{y}_c = k) = \frac{\sum_{i=1}^N \mathbb{I}(\mathbf{y}_{i,c} = k)}{N} \tag{19.61}$$

and for Gaussian potentials, we have

$$\hat{\boldsymbol{\mu}}_c = \frac{\sum_{i=1}^N \mathbf{y}_{ic}}{N}, \; \hat{\boldsymbol{\Sigma}}_c = \frac{\sum_i (\mathbf{y}_{ic} - \hat{\boldsymbol{\mu}}_c)(\mathbf{x}_{ic} - \hat{\boldsymbol{\mu}}_c)^T}{N} \tag{19.62}$$

By using conjugate priors, we can also easily compute the full posterior over the model parameters in the decomposable case, just as we did in the DGM case. See (Lauritzen 1996) for details.

19.6 Conditional random fields (CRFs)

A **conditional random field** or **CRF** (Lafferty et al. 2001), sometimes a **discriminative random field** (Kumar and Hebert 2003), is just a version of an MRF where all the clique potentials are conditioned on input features:

$$p(\mathbf{y}|\mathbf{x}, \mathbf{w}) = \frac{1}{Z(\mathbf{x}, \mathbf{w})} \prod_c \psi_c(\mathbf{y}_c|\mathbf{x}, \mathbf{w}) \tag{19.63}$$

A CRF can be thought of as a **structured output** extension of logistic regression. We will usually assume a log-linear representation of the potentials:

$$\psi_c(\mathbf{y}_c|\mathbf{x}, \mathbf{w}) = \exp(\mathbf{w}_c^T \boldsymbol{\phi}(\mathbf{x}, \mathbf{y}_c)) \tag{19.64}$$

where $\boldsymbol{\phi}(\mathbf{x}, \mathbf{y}_c)$ is a feature vector derived from the global inputs $\mathbf{x}$ and the local set of labels $\mathbf{y}_c$. We will give some examples below which will make this notation clearer.

The advantage of a CRF over an MRF is analogous to the advantage of a discriminative classifier over a generative classifier (see Section 8.6), namely, we don't need to "waste resources" modeling things that we always observe. Instead we can focus our attention on modeling what we care about, namely the distribution of labels given the data.

Another important advantage of CRFs is that we can make the potentials (or factors) of the model be data-dependent. For example, in image processing applications, we may "turn off" the label smoothing between two neighboring nodes s and t if there is an observed discontinuity in the image intensity between pixels s and t. Similarly, in natural language processing problems, we can make the latent labels depend on global properties of the sentence, such as which language it is written in. It is hard to incorporate global features into generative models.

The disadvantage of CRFs over MRFs is that they require labeled training data, and they are slower to train, as we explain in Section 19.6.3. This is analogous to the strengths and weaknesses of logistic regression vs naive Bayes, discussed in Section 8.6.

19.6.1 Chain-structured CRFs, MEMMs and the label-bias problem

The most widely used kind of CRF uses a chain-structured graph to model correlation amongst neighboring labels. Such models are useful for a variety of sequence labeling tasks (see Section 19.6.2).

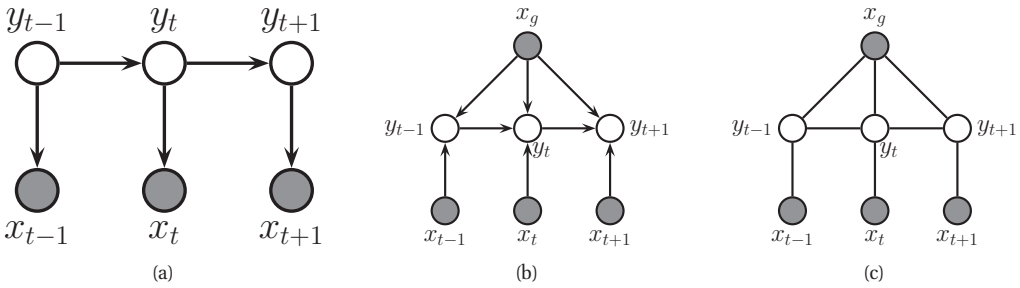


Figure 19.14 Various models for sequential data. (a) A generative directed HMM. (b) A discriminative directed MEMM. (c) A discriminative undirected CRF.

Traditionally, HMMs (discussed in detail in Chapter 17) have been used for such tasks. These are joint density models of the form

$$p(\mathbf{x}, \mathbf{y}|\mathbf{w}) = \prod_{t=1}^{T} p(y_t|y_{t-1}, \mathbf{w}) p(\mathbf{x}_t|y_t, \mathbf{w}) \tag{19.65}$$

where we have dropped the initial $p(y_1)$ term for simplicity. See Figure 19.14(a). If we observe both $\mathbf{x}_t$ and y_t for all t, it is very easy to train such models, using techniques described in Section 17.5.1.

An HMM requires specifying a generative observation model, $p(\mathbf{x}_t|y_t, \mathbf{w})$, which can be difficult. Furthemore, each $\mathbf{x}_t$ is required to be local, since it is hard to define a generative model for the whole stream of observations, $\mathbf{x} = \mathbf{x}_{1:T}$.

An obvious way to make a discriminative version of an HMM is to "reverse the arrows" from y_t to $\mathbf{x}_t$, as in Figure 19.14(b). This defines a directed discriminative model of the form

$$p(\mathbf{y}|\mathbf{x}, \mathbf{w}) = \prod_{t} p(y_t|y_{t-1}, \mathbf{x}, \mathbf{w}) \tag{19.66}$$

where $\mathbf{x} = (\mathbf{x}_{1:T}, \mathbf{x}_g)$, $\mathbf{x}_g$ are global features, and $\mathbf{x}_t$ are features specific to node t. (This partition into local and global is not necessary, but helps when comparing to HMMs.) This is called a **maximum entropy Markov model** or **MEMM** (McCallum et al. 2000; Kakade et al. 2002).

An MEMM is simply a Markov chain in which the state transition probabilities are conditioned on the input features. (It is therefore a special case of an input-output HMM, discussed in Section 17.6.3.) This seems like the natural generalization of logistic regression to the structured-output setting, but it suffers from a subtle problem known (rather obscurely) as the **label bias** problem (Lafferty et al. 2001). The problem is that local features at time t do not influence states prior to time t. This follows by examining the DAG, which shows that $\mathbf{x}_t$ is d-separated from y_{t-1} (and all earlier time points) by the v-structure at y_t, which is a hidden child, thus blocking the information flow.

To understand what this means in practice, consider the part of speech (POS) tagging task. Suppose we see the word "banks"; this could be a verb (as in "he banks at BoA"), or a noun (as in "the river banks were overflowing"). Locally the POS tag for the word is ambiguous. However,

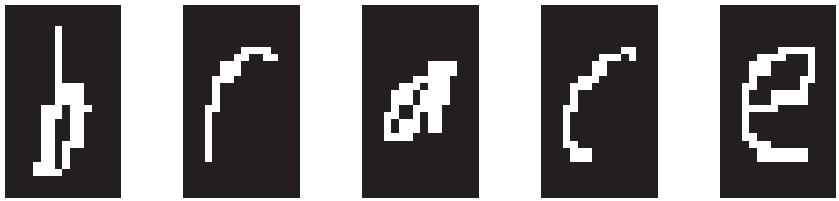


Figure 19.15 Example of handwritten letter recognition. In the word 'brace', the 'r' and the 'c' look very similar, but can be disambiguated using context. Source: (Taskar et al. 2003) . Used with kind permission of Ben Taskar.

suppose that later in the sentence, we see the word "fishing"; this gives us enough context to infer that the sense of "banks" is "river banks". However, in an MEMM (unlike in an HMM and CRF), the "fishing" evidence will not flow backwards, so we will not be able to disambiguate "banks".

Now consider a chain-structured CRF. This model has the form

$$p(\mathbf{y}|\mathbf{x},\mathbf{w}) = \frac{1}{Z(\mathbf{x},\mathbf{w})} \prod_{t=1}^{T} \psi(y_t|\mathbf{x},\mathbf{w}) \prod_{t=1}^{T-1} \psi(y_t, y_{t+1}|\mathbf{x},\mathbf{w}) \tag{19.67}$$

From the graph in Figure 19.14(c), we see that the label bias problem no longer exists, since y_t does not block the information from $\mathbf{x}_t$ from reaching other $y_{t'}$ nodes.

The label bias problem in MEMMs occurs because directed models are **locally normalized**, meaning each CPD sums to 1. By contrast, MRFs and CRFs are **globally normalized**, which means that local factors do not need to sum to 1, since the partition function Z, which sums over all joint configurations, will ensure the model defines a valid distribution. However, this solution comes at a price: we do not get a valid probability distribution over $\mathbf{y}$ until we have seen the whole sentence, since only then can we normalize over all configurations. Consequently, CRFs are not as useful as DGMs (whether discriminative or generative) for online or real-time inference. Furthermore, the fact that Z depends on all the nodes, and hence all their parameters, makes CRFs much slower to train than DGMs, as we will see in Section 19.6.3.

19.6.2 Applications of CRFs

CRFs have been applied to many interesting problems; we give a representative sample below. These applications illustrate several useful modeling tricks, and will also provide motivation for some of the inference techniques we will discuss in Chapter 20.

19.6.2.1 Handwriting recognition

A natural application of CRFs is to classify hand-written digit strings, as illustrated in Figure 19.15. The key observation is that locally a letter may be ambiguous, but by depending on the (unknown) labels of one's neighbors, it is possible to use context to reduce the error rate. Note that the node potential, $\psi_t(y_t|\mathbf{x}_t)$, is often taken to be a probabilistic discriminative classifier,

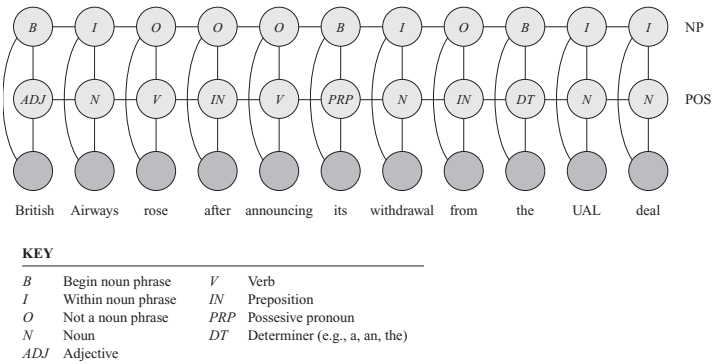


Figure 19.16 A CRF for joint POS tagging and NP segmentation. Source: Figure 4.E.1 of (Koller and Friedman 2009). Used with kind permission of Daphne Koller.

such as a neural network or RVM, that is trained on isolated letters, and the edge potentials, $\psi_{st}(y_s, y_t)$, are often taken to be a language bigram model. Later we will discuss how to train all the potentials jointly.

19.6.2.2 Noun phrase chunking

One common NLP task is **noun phrase chunking**, which refers to the task of segmenting a sentence into its distinct noun phrases (NPs). This is a simple example of a technique known as **shallow parsing**.

In more detail, we tag each word in the sentence with B (meaning beginning of a new NP), I (meaning inside a NP), or O (meaning outside an NP). This is called **BIO** notation. For example, in the following sentence, the NPs are marked with brackets:

```
   B      I       O     O      O        B    I        O    B   I   I
(British Airways) rose after announcing (its withdrawl) from (the UAI deal)
```

(We need the B symbol so that we can distinguish `I I`, meaning two words within a single NP, from `B B`, meaning two separate NPs.)

A standard approach to this problem would first convert the string of words into a string of POS tags, and then convert the POS tags to a string of BIOs. However, such a **pipeline** method can propagate errors. A more robust approach is to build a joint probabilistic model of the form $p(\text{NP}_{1:T}, \text{POS}_{1:T}|\text{words}_{1:T})$. One way to do this is to use the CRF in Figure 19.16. The connections between adjacent labels encode the probability of transitioning between the B, I and O states, and can enforce constraints such as the fact that B must preceed I. The features are usually hand engineered and include things like: does this word begin with a capital letter, is this word followed by a full stop, is this word a noun, etc. Typically there are $\sim 1,000 - 10,000$ features per node.

The number of features has minimal impact on the inference time, since the features are observed and do not need to be summed over. (There is a small increase in the cost of

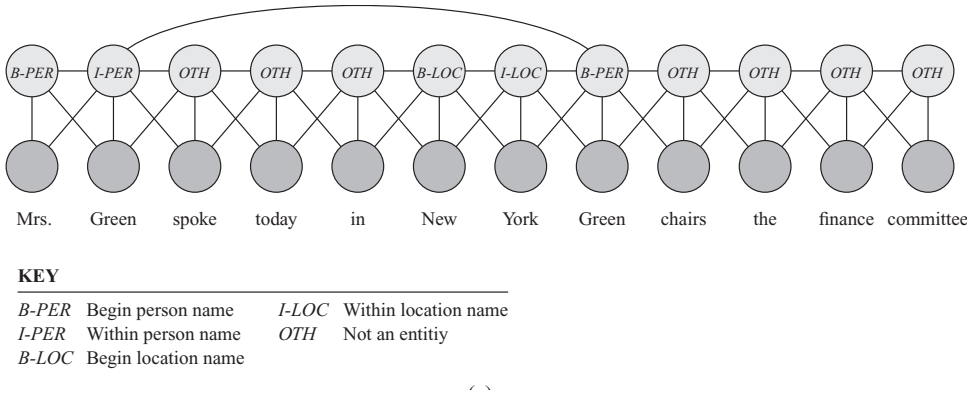


Figure 19.17 A skip-chain CRF for named entity recognition. Source: Figure 4.E.1 of (Koller and Friedman 2009). Used with kind permission of Daphne Koller.

evaluating potential functions with many features, but this is usually negligible; if not, one can use ℓ_1 regularization to prune out irrelevant features.) However, the graph structure can have a dramatic effect on inference time. The model in Figure 19.16 is tractable, since it is essentially a "fat chain", so we can use the forwards-backwards algorithm (Section 17.4.3) for exact inference in $O(T|\text{POS}|^2|\text{NP}|^2)$ time, where $|\text{POS}|$ is the number of POS tags, and $|\text{NP}|$ is the number of NP tags. However, the seemingly similar graph in Figure 19.17, to be explained below, is computationally intractable.

19.6.2.3 Named entity recognition

A task that is related to NP chunking is **named entity extraction**. Instead of just segmenting out noun phrases, we can segment out phrases to do with people and locations. Similar techniques are used to automatically populate your calendar from your email messages; this is called **information extraction**.

A simple approach to this is to use a chain-structured CRF, but to expand the state space from BIO to B-Per, I-Per, B-Loc, I-Loc, and Other. However, sometimes it is ambiguous whether a word is a person, location, or something else. (Proper nouns are particularly difficult to deal with because they belong to an **open class**, that is, there is an unbounded number of possible names, unlike the set of nouns and verbs, which is large but essentially fixed.) We can get better performance by considering long-range correlations between words. For example, we might add a link between all occurrences of the same word, and force the word to have the same tag in each occurence. (The same technique can also be helpful for resolving the identity of pronouns.) This is known as a **skip-chain CRF**. See Figure 19.17 for an illustration.

We see that the graph structure itself changes depending on the input, which is an additional advantage of CRFs over generative models. Unfortunately, inference in this model is generally more expensive than in a simple chain with local connections, for reasons explained in Section 20.5.

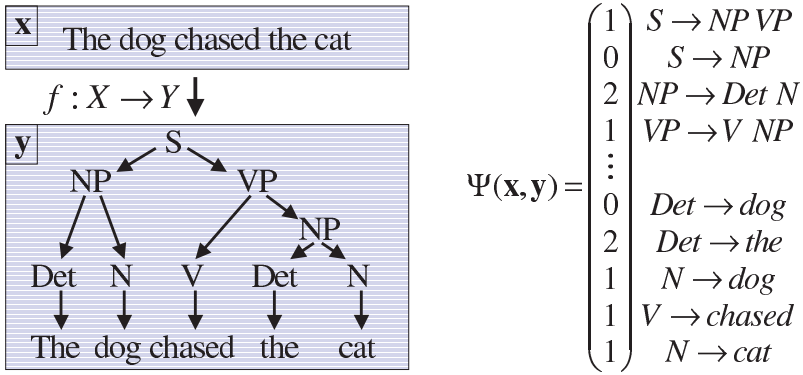


Figure 19.18 Illustration of a simple parse tree based on a context free grammar in Chomsky normal form. The feature vector $\boldsymbol{\phi}(\mathbf{x}, \mathbf{y}) = \Psi(\mathbf{x}, \mathbf{y})$ counts the number of times each production rule was used. Source: Figure 5.2 of (Altun et al. 2006) . Used with kind permission of Yasemin Altun.

19.6.2.4 Natural language parsing

A generalization of chain-structured models for language is to use probabilistic grammars. In particular, a probabilistic **context free grammar** or **PCFG** is a set of re-write or production rules of the form $\sigma \rightarrow \sigma'\sigma''$ or $\sigma \rightarrow x$, where $\sigma, \sigma', \sigma'' \in \Sigma$ are non-terminals (analogous to parts of speech), and $x \in \mathcal{X}$ are terminals, i.e., words. See Figure 19.18 for an example. Each such rule has an associated probability. The resulting model defines a probability distribution over sequences of words. We can compute the probability of observing a particular sequence $\mathbf{x} = x_1 \ldots x_T$ by summing over all trees that generate it. This can be done in $O(T^3)$ time using the **inside-outside algorithm**; see e.g., (Jurafsky and Martin 2008; Manning and Schuetze 1999) for details.

PCFGs are generative models. It is possible to make discriminative versions which encode the probability of a labeled tree, $\mathbf{y}$, given a sequence of words, $\mathbf{x}$, by using a CRF of the form $p(\mathbf{y}|\mathbf{x}) \propto \exp(\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}, \mathbf{y}))$. For example, we might define $\boldsymbol{\phi}(\mathbf{x}, \mathbf{y})$ to count the number of times each production rule was used (which is analogous to the number of state transitions in a chain-structured model). See e.g., (Taskar et al. 2004) for details.

19.6.2.5 Hierarchical classification

Suppose we are performing multi-class classification, where we have a **label taxonomy**, which groups the classes into a hierarchy. We can encode the position of y within this hierarchy by defining a binary vector $\boldsymbol{\phi}(y)$, where we turn on the bit for component y and for all its children. This can be combined with input features $\boldsymbol{\phi}(\mathbf{x})$ using a tensor product, $\boldsymbol{\phi}(\mathbf{x}, y) = \boldsymbol{\phi}(\mathbf{x}) \otimes \boldsymbol{\phi}(y)$. See Figure 19.19 for an example.

This method is widely used for text classification, where manually constructed taxnomies (such as the Open Directory Project at `www.dmoz.org`) are quite common. The benefit is that information can be shared between the parameters for nearby categories, enabling generalization across classes.

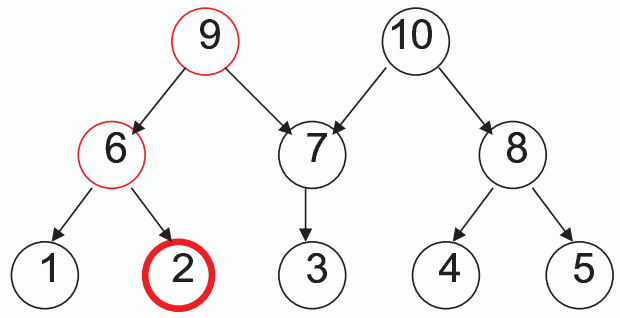


Figure 19.19 Illustration of a simple label taxonomy, and how it can be used to compute a distributed representation for the label for class 2. In this figure, $\boldsymbol{\phi}(\mathbf{x}) = \mathbf{x}$, $\boldsymbol{\phi}(y = 2) = \Lambda(2)$, $\boldsymbol{\phi}(\mathbf{x}, y)$ is denoted by $\Psi(\mathbf{x}, 2)$, and $\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}, y)$ is denoted by $\langle \mathbf{w}, \Psi(\mathbf{x}, 2) \rangle$. Source: Figure 5.1 of (Altun et al. 2006) . Used with kind permission of Yasemin Altun.

19.6.2.6 Protein side-chain prediction

An interesting analog to the skip-chain model arises in the problem of predicting the structure of protein side chains. Each residue in the side chain has 4 dihedral angles, which are usually discretized into 3 values called rotamers. The goal is to predict this discrete sequence of angles, $\mathbf{y}$, from the discrete sequence of amino acids, $\mathbf{x}$.

We can define an energy function $E(\mathbf{x}, \mathbf{y})$, where we include various pairwise interaction terms between nearby residues (elements of the $\mathbf{y}$ vector). This energy is usually defined as a weighted sum of individual energy terms, $E(\mathbf{x}, \mathbf{y}|\mathbf{w}) = \sum_{j=1}^{D} \theta_j E_j(\mathbf{x}, \mathbf{y})$, where the E_j are energy contribution due to various electrostatic charges, hydrogen bonding potentials, etc, and $\mathbf{w}$ are the parameters of the model. See (Yanover et al. 2007) for details.

Given the model, we can compute the most probable side chain configuration using $\mathbf{y}^* = \operatorname{argmin} E(\mathbf{x}, \mathbf{y}|\mathbf{w})$. In general, this problem is NP-hard, depending on the nature of the graph induced by the E_j terms, due to long-range connections between the variables. Nevertheless, some special cases can be efficiently handled, using methods discussed in Section 22.6.

19.6.2.7 Stereo vision

Low-level vision problems are problems where the input is an image (or set of images), and the output is a processed version of the image. In such cases, it is common to use 2d lattice-structured models; the models are similar to Figure 19.9, except that the features can be global, and are not generated by the model. We will assume a pairwise CRF.

A classic low-level vision problem is **dense stereo reconstruction**, where the goal is to estimate the depth of every pixel given two images taken from slightly different angles. In this section (based on (Sudderth and Freeman 2008)), we give a sketch of how a simple CRF can be used to solve this task. See e.g., (Sun et al. 2003) for a more sophisticated model.

By using some standard preprocessing techniques, one can convert depth estimation into a

problem of estimating the **disparity** y_s between the pixel at location (i_s, j_s) in the left image and the corresponding pixel at location $(i_s + y_s, j_s)$ in the right image. We typically assume that corresponding pixels have similar intensity, so we define a local node potential of the form

$$\psi_s(y_s|\mathbf{x}) \propto \exp\left\{-\frac{1}{2\sigma^2}\left(x_L(i_s, j_s) - x_R(i_s + y_s, j_s)\right)^2\right\} \tag{19.68}$$

where x_L is the left image and x_R is the right image. This equation can be generalized to model the intensity of small windows around each location. In highly textured regions, it is usually possible to find the corresponding patch using cross correlation, but in regions of low texture, there will be considerable ambiguity about the correct value of y_s.

We can easily add a Gaussian prior on the edges of the MRF that encodes the assumption that neighboring disparities y_s, y_t should be similar, as follows:

$$\psi_{st}(y_s, y_t) \propto \exp\left(-\frac{1}{2\gamma^2}(y_s - y_t)^2\right) \tag{19.69}$$

The resulting model is a Gaussian CRF.

However, using Gaussian edge-potentials will oversmooth the estimate, since this prior fails to account for the occasional large changes in disparity that occur between neighboring pixels which are on different sides of an occlusion boundary. One gets much better results using a **truncated Gaussian potential** of the form

$$\psi_{st}(y_s, y_t) \propto \exp\left\{-\frac{1}{2\gamma^2}\min\left((y_s - y_t)^2, \delta_0^2\right)\right\} \tag{19.70}$$

where γ encodes the expected smoothness, and δ_0 encodes the maximum penalty that will be imposed if disparities are significantly different. This is called a **discontinuity preserving** potential; note that such penalties are not convex. The local evidence potential can be made robust in a similar way, in order to handle outliers due to specularities, occlusions, etc.

Figure 19.20 illustrates the difference between these two forms of prior. On the top left is an image from the standard Middlebury stereo benchmark dataset (Scharstein and Szeliski 2002). On the bottom left is the corresponding true disparity values. The remaining columns represent the estimated disparity after 0, 1 and an "infinite" number of rounds of loopy belief propagation (see Section 22.2), where by "infinite" we mean the results at convergence. The top row shows the results using a Gaussian edge potential, and the bottom row shows the results using the truncated potential. The latter is clearly better.

Unfortunately, performing inference with real-valued variables is computationally difficult, unless the model is jointly Gaussian. Consequently, it is common to discretize the variables. (For example, Figure 19.20(bottom) used 50 states.) The edge potentials still have the form given in Equation 19.69. The resulting model is called a **metric CRF**, since the potentials form a metric. [9] Inference in metric CRFs is more efficient than in CRFs where the discrete labels have no natural ordering, as we explain in Section 22.6.3.3. See Section 22.6.4 for a comparison of various approximate inference methods applied to low-level CRFs, and see (Blake et al. 2011; Prince 2012) for more details on probabilistic models for computer vision.

9. A function f is said to be a **metric** if it satisfies the following three properties: Reflexivity: $f(a, b) = 0$ iff $a = b$; Symmetry: $f(a, b) = f(b, a)$; and Triangle inequality: $f(a, b) + f(b, c) \geq f(a, c)$. If f satisfies only the first two properties, it is called a **semi-metric**.

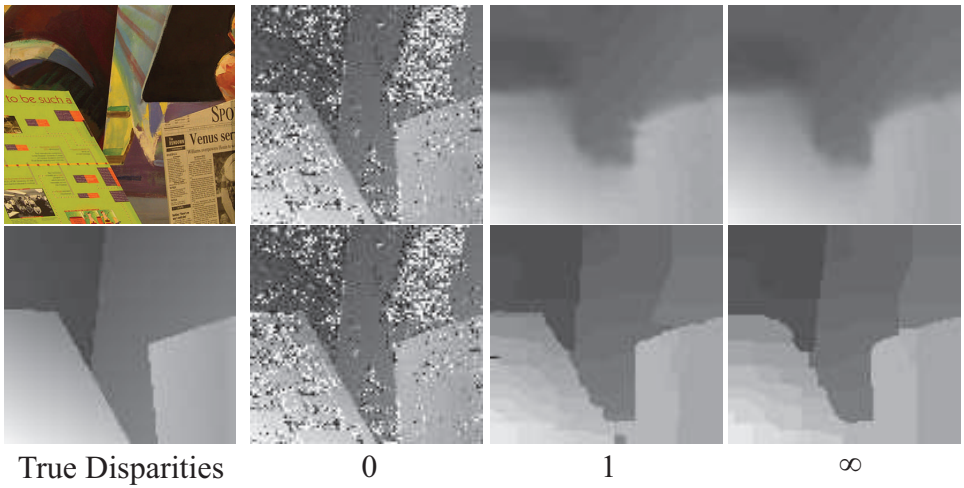


Figure 19.20 Illustration of belief propagation for stereo depth estimation. Left column: image and true disparities. Remaining columns: initial estimate, estimate after 1 iteration, and estimate at convergence. Top row: Gaussian edge potentials. Bottom row: robust edge potentials. Source: Figure 4 of (Sudderth and Freeman 2008). Used with kind permission of Erik Sudderth.

19.6.3 CRF training

We can modify the gradient based optimization of MRFs described in Section 19.5.1 to the CRF case in a straightforward way. In particular, the scaled log-likelihood becomes

$$\ell(\mathbf{w}) \triangleq \frac{1}{N}\sum_i \log p(\mathbf{y}_i|\mathbf{x}_i,\mathbf{w}) = \frac{1}{N}\sum_i \left[\sum_c \mathbf{w}_c^T \boldsymbol{\phi}_c(\mathbf{y}_i,\mathbf{x}_i) - \log Z(\mathbf{w},\mathbf{x}_i)\right] \tag{19.71}$$

and the gradient becomes

$$\frac{\partial \ell}{\partial \mathbf{w}_c} = \frac{1}{N}\sum_i \left[\boldsymbol{\phi}_c(\mathbf{y}_i,\mathbf{x}_i) - \frac{\partial}{\partial \mathbf{w}_c}\log Z(\mathbf{w},\mathbf{x}_i)\right] \tag{19.72}$$

$$= \frac{1}{N}\sum_i \left[\boldsymbol{\phi}_c(\mathbf{y}_i,\mathbf{x}_i) - \mathbb{E}\left[\boldsymbol{\phi}_c(\mathbf{y},\mathbf{x}_i)\right]\right] \tag{19.73}$$

Note that we now have to perform inference for every single training case inside each gradient step, which is $O(N)$ times slower than the MRF case. This is because the partition function depends on the inputs $\mathbf{x}_i$.

In most applications of CRFs (and some applications of MRFs), the size of the graph structure can vary. Hence we need to use parameter tying to ensure we can define a distribution of arbitrary size. In the pairwise case, we can write the model as follows:

$$p(\mathbf{y}|\mathbf{x},\mathbf{w}) = \frac{1}{Z(\mathbf{w},\mathbf{x})}\exp\left(\mathbf{w}^T\boldsymbol{\phi}(\mathbf{y},\mathbf{x})\right) \tag{19.74}$$

where $\mathbf{w} = [\mathbf{w}_n, \mathbf{w}_e]$ are the node and edge parameters, and

$$\boldsymbol{\phi}(\mathbf{y}, \mathbf{x}) \triangleq [\sum_t \boldsymbol{\phi}_t(y_t, \mathbf{x}), \sum_{s \sim t} \boldsymbol{\phi}_{st}(y_s, y_t, \mathbf{x})] \tag{19.75}$$

are the summed node and edge features (these are the sufficient statistics). The gradient expression is easily modified to handle this case.

In practice, it is important to use a prior/ regularization to prevent overfitting. If we use a Gaussian prior, the new objective becomes

$$\ell'(\mathbf{w}) \triangleq \frac{1}{N} \sum_i \log p(\mathbf{y}_i | \mathbf{x}_i, \mathbf{w}) - \lambda ||\mathbf{w}||_2^2 \tag{19.76}$$

It is simple to modify the gradient expression.

Alternatively, we can use ℓ_1 regularization. For example, we could use ℓ_1 for the edge weights $\mathbf{w}_e$ to learn a sparse graph structure, and ℓ_2 for the node weights $\mathbf{w}_n$, as in (Schmidt et al. 2008). In other words, the objective becomes

$$\ell'(\mathbf{w}) \triangleq \frac{1}{N} \sum_i \log p(\mathbf{y}_i | \mathbf{x}_i, \mathbf{w}) - \lambda_1 ||\mathbf{w}_e||_1 - \lambda_2 ||\mathbf{w}_n||_2^2 \tag{19.77}$$

Unfortunately, the optimization algorithms are more complicated when we use ℓ_1 (see Section 13.4), although the problem is still convex.

To handle large datasets, we can use stochastic gradient descent (SGD), as described in Section 8.5.2.

It is possible (and useful) to define CRFs with hidden variables, for example to allow for an unknown alignment between the visible features and the hidden labels (see e.g., (Schnitzspan et al. 2010)). In this case, the objective function is no longer convex. Nevertheless, we can find a locally optimal ML or MAP parameter estimate using EM and/ or gradient methods.

19.7 Structural SVMs

We have seen that training a CRF requires inference, in order to compute the expected sufficient statistics needed to evaluate the gradient. For certain models, computing a joint MAP estimate of the states is provably simpler than computing marginals, as we discuss in Section 22.6. In this section, we discuss a way to train structured output classifiers that that leverages the existence of fast MAP solvers. (To avoid confusion with MAP estimation of parameters, we will often refer to MAP estimation of states as **decoding**.) These methods are known as **structural support vector machines** or **SSVMs** (Tsochantaridis et al. 2005). (There is also a very similar class of methods known as **max margin Markov networks** or **M3nets** (Taskar et al. 2003); see Section 19.7.2 for a discussion of the differences.)

19.7.1 SSVMs: a probabilistic view

In this book, we have mostly concentrated on fitting models using MAP parameter estimation, i.e., by minimizing functions of the form

$$R_{MAP}(\mathbf{w}) = -\log p(\mathbf{w}) - \sum_{i=1}^{N} \log p(\mathbf{y}_i | \mathbf{x}_i, \mathbf{w}) \tag{19.78}$$

However, at test time, we pick the label so as to minimize the posterior expected loss (defined in Section 5.7):

$$\hat{\mathbf{y}}(\mathbf{x}|\mathbf{w}) = \operatorname*{argmin}_{\hat{\mathbf{y}}} \sum_{\mathbf{y}} L(\hat{\mathbf{y}}, \mathbf{y}) p(\mathbf{y}|\mathbf{x}, \mathbf{w}) \tag{19.79}$$

where $L(\mathbf{y}^*, \hat{\mathbf{y}})$ is the loss we incur when we estimate $\hat{\mathbf{y}}$ but the truth is $\mathbf{y}^*$. It therefore seems reasonable to take the loss function into account when performing parameter estimation.[10] So, following (Yuille and He 2011), let us instead minimized the posterior expected loss on the training set:

$$R_{EL}(\mathbf{w}) \triangleq -\log p(\mathbf{w}) + \sum_{i=1}^{N} \log \left[\sum_{\mathbf{y}} L(\mathbf{y}_i, \mathbf{y}) p(\mathbf{y}|\mathbf{x}_i, \mathbf{w}) \right] \tag{19.80}$$

In the special case of 0-1 loss, $L(\mathbf{y}_i, \mathbf{y}) = 1 - \delta_{\mathbf{y}, \mathbf{y}_i}$, this reduces to R_{MAP}.

We will assume that we can write our model in the following form:

$$p(\mathbf{y}|\mathbf{x}, \mathbf{w}) = \frac{\exp(\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}, \mathbf{y}))}{Z(\mathbf{x}, \mathbf{w})} \tag{19.81}$$

$$p(\mathbf{w}) = \frac{\exp(-E(\mathbf{w}))}{Z} \tag{19.82}$$

where $Z(\mathbf{x}, \mathbf{w}) = \sum_{\mathbf{y}} \exp(\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}, \mathbf{y}))$. Also, let us define $L(\mathbf{y}_i, \mathbf{y}) = \exp \tilde{L}(\mathbf{y}_i, \mathbf{y})$. With this, we can rewrite our objective as follows:

$$R_{EL}(\mathbf{w}) = -\log p(\mathbf{w}) + \sum_i \log \left[\sum_{\mathbf{y}} \exp \tilde{L}(\mathbf{y}_i, \mathbf{y}) \frac{\exp(\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}, \mathbf{y}))}{Z(\mathbf{x}, \mathbf{w})} \right] \tag{19.83}$$

$$= E(\mathbf{w}) + \sum_i -\log Z(\mathbf{x}_i, \mathbf{w}) + \log \sum_{\mathbf{y}} \exp \left(\tilde{L}(v y_i, \mathbf{y}) + \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_i, \mathbf{y}) \right) \tag{19.84}$$

We will now consider various bounds in order to simplify this objective. First note that for any function $f(\mathbf{y})$ we have

$$\max_{\mathbf{y} \in \mathcal{Y}} f(\mathbf{y}) \leq \log \sum_{\mathbf{y} \in \mathcal{Y}} \exp[f(\mathbf{y})] \leq \log \left[|\mathcal{Y}| \exp \left(\max_{\mathbf{y}} f(\mathbf{y}) \right) \right] = \log |\mathcal{Y}| + \max_{\mathbf{y}} f(\mathbf{y}) \tag{19.85}$$

For example, suppose $\mathcal{Y} = \{0, 1, 2\}$ and $f(y) = y$. Then we have

$$2 = \log[\exp(2)] \leq \log[\exp(0) + \exp(1) + \exp(2)] \leq \log[3 \times \exp(2)] = \log(3) + 2 \tag{19.86}$$

We can ignore the $\log |\mathcal{Y}|$ term, which is independent of $\mathbf{y}$, and treat $\max_{\mathbf{y} \in \mathcal{Y}} f(\mathbf{y})$ as both a lower and upper bound. Hence we see that

$$R_{EL}(\mathbf{w}) \sim E(\mathbf{w}) + \sum_{i=1}^{N} \left[\max_{\mathbf{y}} \left\{ \tilde{L}(\mathbf{y}_i, \mathbf{y}) + \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_i, \mathbf{y}) \right\} - \max_{\mathbf{y}} \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_i, \mathbf{y}) \right] \tag{19.87}$$

10. Note that this violates the fundamental Bayesian distinction between inference and decision making. However, performing these tasks separately will only result in an optimal decision if we can compute the exact posterior. In most cases, this is intractable, so we need to perform **loss-calibrated inference** (Lacoste-Julien et al. 2011). In this section, we just perform loss-calibrated MAP parameter estimation, which is computationally simpler. (See also (Stoyanov et al. 2011).)

where $x \sim y$ means $c_1 + x \leq y + c_2$ for some constants c_1, c_2. Unfortunately, this objective is not convex in $\mathbf{w}$. However, we can devise a convex upper bound by exploiting the following looser lower bound on the log-sum-exp function:

$$f(\mathbf{y}') \leq \log \sum_{\mathbf{y}} \exp[f(\mathbf{y})] \tag{19.88}$$

for any $\mathbf{y}' \in \mathcal{Y}$. Applying this equation to our earlier example, for $f(y) = y$ and $y' = 1$, we get $1 = \log[\exp(1)] \leq \log[\exp(0) + \exp(1) + \exp(2)]$. And applying this bound to R_{EL} we get

$$R_{EL}(\mathbf{w}) \quad \leq \quad E(\mathbf{w}) + \sum_{i=1}^{N} \left[\max_{\mathbf{y}} \left\{ \tilde{L}(\mathbf{y}_i, \mathbf{y}) + \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_i, \mathbf{y}) \right\} - \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_i, \mathbf{y}_i) \right] \tag{19.89}$$

If we set $E(\mathbf{w}) = -\frac{1}{2C}||\mathbf{w}||_2^2$ (corresponding to a spherical Gaussian prior), we get

$$R_{SSVM}(\mathbf{w}) \quad \triangleq \quad \frac{1}{2}||\mathbf{w}||^2 + C \sum_{i=1}^{N} \left[\max_{\mathbf{y}} \left\{ \tilde{L}(\mathbf{y}_i, \mathbf{y}) + \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_i, \mathbf{y}) \right\} - \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_i, \mathbf{y}_i) \right] \tag{19.90}$$

This is the same objective as used in the SSVM approach of (Tsochantaridis et al. 2005).

In the special case that $\mathcal{Y} = \{-1, +1\}$ $L(y^*, y) = 1 - \delta_{y,y^*}$, and $\boldsymbol{\phi}(\mathbf{x}, y) = \frac{1}{2} y \mathbf{x}$, this criterion reduces to the following (by considering the two cases that $y = y_i$ and $y \neq y_i$):

$$R_{SVM}(\mathbf{w}) \quad \triangleq \quad \frac{1}{2}||\mathbf{w}||^2 + C \sum_{i=1}^{N} \left[\max\{0, 1 - y_i \mathbf{w}^T \mathbf{x}_i\} \right] \tag{19.91}$$

which is the standard binary SVM objective (see Equation 14.57).

So we see that the SSVM criterion can be seen as optimizing an upper bound on the Bayesian objective, a result first shown in (Yuille and He 2011). This bound will be tight (and hence the approximation will be a good one) when $||\mathbf{w}||$ is large, since in that case, $p(\mathbf{y}|\mathbf{x}, \mathbf{w})$ will concentrate its mass on $\text{argmax}_{\mathbf{y}} p(\mathbf{y}|\mathbf{x}, \mathbf{w})$. Unfortunately, a large $||\mathbf{w}||$ corresponds to a model that is likely to overfit, so it is unlikely that we will be working in this regime (because we will tune the strength of the regularizer to avoid this situation). An alternative justification for the SVM criterion is that it focusses effort on fitting parameters that affect the decision boundary. This is a better use of computational resources than fitting the full distribution, especially when the model is wrong.

19.7.2 SSVMs: a non-probabilistic view

We now present SSVMs in a more traditional (non-probabilistic) way, following (Tsochantaridis et al. 2005). The resulting objective will be the same as the one above. However, this derivation will set the stage for the algorithms we discuss below.

Let $f(\mathbf{x}; \mathbf{w}) = \text{argmax}_{\mathbf{y} \in \mathcal{Y}} \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}, \mathbf{y})$ be the prediction function. We can obtain zero loss on the training set using this predictor if

$$\forall i. \max_{\mathbf{y} \in \mathcal{Y} \setminus \mathbf{y}_i} \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_i, \mathbf{y}) \leq \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_i, \mathbf{y}_i) \tag{19.92}$$

Each one of these nonlinear inequalities can be equivalently replaced by $|\mathcal{Y}| - 1$ linear inequalities, resulting in a total of $N|\mathcal{Y}| - N$ linear constraints of the following form:

$$\forall i. \forall \mathbf{y} \in \mathcal{Y} \setminus \mathbf{y}_i. \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_i, \mathbf{y}_i) - \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_i, \mathbf{y}) \geq 0 \tag{19.93}$$

For brevity, we introduce the notation

$$\boldsymbol{\delta}_i(\mathbf{y}) \triangleq \boldsymbol{\phi}(\mathbf{x}_i, \mathbf{y}_i) - \boldsymbol{\phi}(\mathbf{x}_i, \mathbf{y}) \tag{19.94}$$

so we can rewrite these constraints as $\mathbf{w}^T \boldsymbol{\delta}_i(\mathbf{y}) \geq 0$.

If we can achieve zero loss, there will typically be multiple solution vectors $\mathbf{w}$. We pick the one that maximizes the margin, defined as

$$\gamma \triangleq \min_i f(\mathbf{x}, \mathbf{y}_i; \mathbf{w}) - \max_{\mathbf{y}' \in \mathcal{Y} \setminus \mathbf{y}} f(\mathbf{x}, \mathbf{y}'; \mathbf{w}) \tag{19.95}$$

Since the margin can be made arbitrarily large by rescaling $\mathbf{w}$, we fix its norm to be 1, resulting in the optimization problem

$$\max_{\gamma, \mathbf{w}: ||\mathbf{w}||=1} \quad \text{s.t.} \quad \forall i. \forall \mathbf{y} \in \mathcal{Y} \setminus \mathbf{y}_i. \ \ \mathbf{w}^T \boldsymbol{\delta}_i(\mathbf{y}) \geq \gamma \tag{19.96}$$

Equivalently, we can write

$$\min_{\mathbf{w}} \frac{1}{2} ||\mathbf{w}||^2 \quad \text{s.t.} \quad \forall i. \forall \mathbf{y} \in \mathcal{Y} \setminus \mathbf{y}_i. \ \ \mathbf{w}^T \boldsymbol{\delta}_i(\mathbf{y}) \geq 1 \tag{19.97}$$

To allow for the case where zero loss cannot be achieved (equivalent to the data being inseparable in the case of binary classification), we relax the constraints by introducing slack terms ξ_i, one per data case. This yields

$$\min_{\mathbf{w}, \boldsymbol{\xi}} \frac{1}{2} ||\mathbf{w}||^2 + C \sum_{i=1}^{N} \xi_i \quad \text{s.t.} \quad \forall i. \forall \mathbf{y} \in \mathcal{Y} \setminus \mathbf{y}_i. \ \ \mathbf{w}^T \boldsymbol{\delta}_i(\mathbf{y}) \geq 1 - \xi_i, \xi_i \geq 0 \tag{19.98}$$

In the case of structured outputs, we don't want to treat all constraint violations equally. For example, in a segmentation problem, getting one position wrong should be punished less than getting many positions wrong. One way to achieve this is to divide the slack variable by the size of the loss (this is called **slack re-scaling**). This yields

$$\min_{\mathbf{w}, \boldsymbol{\xi}} \frac{1}{2} ||\mathbf{w}||^2 + C \sum_{i=1}^{N} \xi_i \quad \text{s.t.} \quad \forall i. \forall \mathbf{y} \in \mathcal{Y} \setminus \mathbf{y}_i. \ \ \mathbf{w}^T \boldsymbol{\delta}_i(\mathbf{y}) \geq 1 - \frac{\xi_i}{L(\mathbf{y}_i, \mathbf{y})}, \xi_i \geq 0 \tag{19.99}$$

Alternatively, we can define the margin to be proportional to the loss (this is called **margin re-rescaling**). This yields

$$\min_{\mathbf{w}, \boldsymbol{\xi}} \frac{1}{2} ||\mathbf{w}||^2 + C \sum_{i=1}^{N} \xi_i \quad \text{s.t.} \quad \forall i. \forall \mathbf{y} \in \mathcal{Y} \setminus \mathbf{y}_i. \ \ \mathbf{w}^T \boldsymbol{\delta}_i(\mathbf{y}) \geq L(\mathbf{y}_i, \mathbf{y}) - \xi_i, \ \ \xi_i \geq 0 \tag{19.100}$$

(In fact, we can write $\forall \mathbf{y} \in \mathcal{Y}$ instead of $\forall \mathbf{y} \in \mathcal{Y} \setminus \mathbf{y}_i$, since if $\mathbf{y} = \mathbf{y}_i$, then $\mathbf{w}^T \boldsymbol{\delta}_i(\mathbf{y}) = 0$ and $\xi_i = 0$. By using the simpler notation, which doesn't exclude $\mathbf{y}_i$, we add an extra but redundant constraint.) This latter approach is used in M3nets.

For future reference, note that we can solve for the ξ_i^* terms as follows:

$$\xi_i^*(\mathbf{w}) = \max\{0, \max_{\mathbf{y}}(L(\mathbf{y}_i, \mathbf{y}) - \mathbf{w}^T\boldsymbol{\delta}_i))\} = \max_{\mathbf{y}}(L(\mathbf{y}_i, \mathbf{y}) - \mathbf{w}^T\boldsymbol{\delta}_i)) \tag{19.101}$$

Substituting in, and dropping the constraints, we get the following equivalent problem:

$$\min_{\mathbf{w}} \frac{1}{2}||\mathbf{w}||^2 + C\sum_i \max_{\mathbf{y}} \left\{L(\mathbf{y}_i, \mathbf{y}) + \mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}_i, \mathbf{y})\right\} - \mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}_i, \mathbf{y}_i) \tag{19.102}$$

19.7.2.1 Empirical risk minimization

Let us pause and consider whether the above objective is reasonable. Recall that in the frequentist approach to machine learning (Section 6.5), the goal is to minimize the regularized empirical risk, defined by

$$\mathcal{R}(\mathbf{w}) + \frac{C}{N}\sum_{i=1}^{N} L(\mathbf{y}_i, f(\mathbf{x}_i, \mathbf{w})) \tag{19.103}$$

where $\mathcal{R}(\mathbf{w})$ is the regularizer, and $f(\mathbf{x}_i, \mathbf{w}) = \text{argmax}_{\mathbf{y}}\, \mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}_i, \mathbf{y}) = \hat{\mathbf{y}}_i$ is the prediction. Since this objective is hard to optimize, because the loss is not differentiable, we will construct a convex upper bound instead.

We can show that

$$\mathcal{R}(\mathbf{w}) + \frac{C}{N}\sum_i \max_{\mathbf{y}}(L(\mathbf{y}_i, \mathbf{y}) - \mathbf{w}^T\boldsymbol{\delta}_i)) \tag{19.104}$$

is such a convex upper bound. To see this, note that

$$L(\mathbf{y}_i, f(\mathbf{x}_i, \mathbf{w})) \leq L(\mathbf{y}_i, f(\mathbf{x}_i, \mathbf{w})) - \mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}_i, \mathbf{y}_i) + \mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}_i, \hat{\mathbf{y}}_i) \tag{19.105}$$

$$\leq \max_{\mathbf{y}} L(\mathbf{y}_i, \mathbf{y}) - \mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}_i, \mathbf{y}_i) + \mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}_i, \mathbf{y}) \tag{19.106}$$

Using this bound and $\mathcal{R}(\mathbf{w}) = \frac{1}{2}||\mathbf{w}||^2$ yields Equation 19.102.

19.7.2.2 Computational issues

Although the above objectives are simple quadratic programs (QP), they have $O(N|\mathcal{Y}|)$ constraints. This is intractable, since $\mathcal{Y}$ is usually exponentially large. In the case of the margin rescaling formulation, it is possible to reduce the exponential number of constraints to a polynomial number, provided the loss function and the feature vector decompose according to a graphical model. This is the approach used in M3nets (Taskar et al. 2003).

An alternative approach is to work directly with the exponentially sized QP. This allows for the use of more general loss functions. There are several possible methods to make this feasible. One is to use cutting plane methods. Another is to use stochastic subgradient methods. We discuss both of these below.

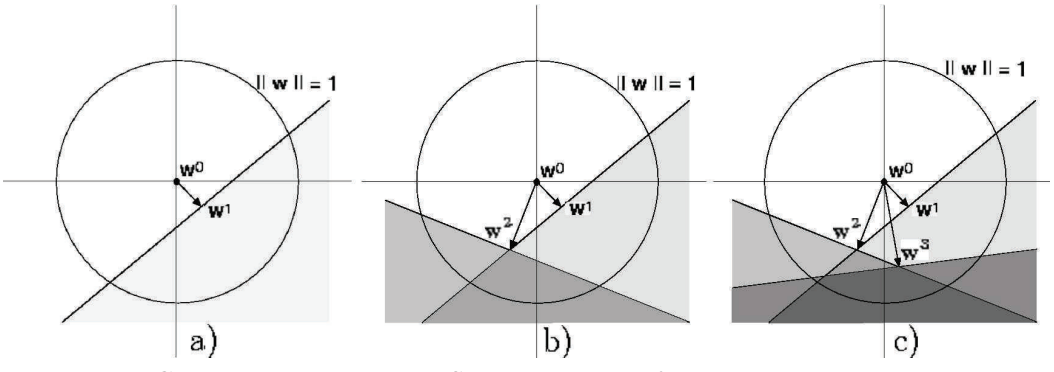


Figure 19.21 Illustration of the cutting plane algorithm in 2d. We start with the estimate $\mathbf{w} = \mathbf{w}_0 = \mathbf{0}$. (a) We add the first constraint; the shaded region is the new feasible set. The new minimum norm solution is $\mathbf{w}_1$. (b) We add another constraint; the dark shaded region is the new feasible set. (c) We add a third constraint. Source: Figure 5.3 of (Altun et al. 2006) . Used with kind permission of Yasemin Altun.

19.7.3 Cutting plane methods for fitting SSVMs

In this section, we discuss an efficient algorithm for fitting SSVMs due to (Joachims et al. 2009). This method can handle general loss functions, and is implemented in the popular **SVMstruct** package[11]. The method is based on the **cutting plane** method from convex optimization (Kelley 1960).

The basic idea is as follows. We start with an initial guess $\mathbf{w}$ and no constraints. At each iteration, we then do the following: for each example i, we find the "most violated" constraint involving $\mathbf{x}_i$ and $\hat{\mathbf{y}}_i$. If the loss-augmented margin violation exceeds the current value of ξ_i by more than ϵ, we add $\hat{\mathbf{y}}_i$ to the working set of constraints for this training case, $\mathcal{W}_i$, and then solve the resulting new QP to find the new $\mathbf{w}, \boldsymbol{\xi}$. See Figure 19.21 for a sketch, and Algorithm 11 for the pseudo code. (Since at each step we only add one new constraint, we can warm-start the QP solver.) We can can easily modify the algorithm to optimize the slack rescaling version by replacing the expression $L(\mathbf{y}_i, \mathbf{y}) - \mathbf{w}^T \boldsymbol{\delta}_i(\hat{\mathbf{y}}_i)$ with $L(\mathbf{y}_i, \mathbf{y})(1 - \mathbf{w}^T \boldsymbol{\delta}_i(\hat{\mathbf{y}}_i))$.

The key to the efficiency of this method is that only polynomially many constraints need to be added, and as soon as they are, the exponential number of other constraints are guaranteed to also be satisfied to within a tolerance of ϵ (see (Tsochantaridis et al. 2005) for the proof).

19.7.3.1 Loss-augmented decoding

The other key to efficiency is the ability to find the most violated constraint in line 5 of the algorithm, i.e., to compute

$$\operatorname*{argmax}_{\mathbf{y} \in \mathcal{Y}} L(\mathbf{y}_i, \mathbf{y}) - \mathbf{w}^T \boldsymbol{\delta}_i(\mathbf{y}) = \operatorname*{argmax}_{\mathbf{y} \in \mathcal{Y}} L(\mathbf{y}_i, \mathbf{y}) + \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_i, \mathbf{y}) \tag{19.107}$$

11. `http://svmlight.joachims.org/svm_struct.html`

Algorithm 19.3: Cutting plane algorithm for SSVMs (margin rescaling, N-slack version)

Input $\mathcal{D} = \{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_N, \mathbf{y}_n)\}$, C, ϵ ;
$\mathcal{W}_i = \emptyset$, $\xi_i = 0$ for $i = 1 : N$;
repeat
  for $i = 1 : N$ **do**
    $\hat{\mathbf{y}}_i = \text{argmax}_{\hat{\mathbf{y}}_i \in \mathcal{Y}} L(\mathbf{y}_i, \mathbf{y}) - \mathbf{w}^T \boldsymbol{\delta}_i(\hat{\mathbf{y}}_i)$;
    if $L(\mathbf{y}_i, \mathbf{y}) - \mathbf{w}^T \boldsymbol{\delta}_i(\hat{\mathbf{y}}_i) > \xi_i + \epsilon$ **then**
      $\mathcal{W}_i = \mathcal{W}_i \cup \{\hat{\mathbf{y}}_i\}$;
      $(\mathbf{w}, \boldsymbol{\xi}) = \text{argmin}_{\mathbf{w}, \boldsymbol{\xi} \geq \mathbf{0}} \frac{1}{2}||\mathbf{w}||_2^2 + C \sum_{i=1}^N \xi_i$;
        s.t. $\forall i = 1 : N, \forall \mathbf{y}' \in \mathcal{W}_i : \mathbf{w}^T \boldsymbol{\delta}_i(\hat{\mathbf{y}}_i) \geq L(\mathbf{y}_i, \mathbf{y}') - \xi_i$;
until *no* $\mathcal{W}_i$ *has changed*;
Return $(\mathbf{w}, \boldsymbol{\xi})$

We call this process **loss-augmented decoding**. (In (Joachims et al. 2009), this procedure is called the **separation oracle**.) If the loss function has an additive decomposition of the same form as the features, then we can fold the loss into the weight vector, i.e., we can find a new set of parameters $\mathbf{w}'$ such that $(\mathbf{w}')^T \boldsymbol{\delta}_i(\mathbf{y}) = \mathbf{w}^T \boldsymbol{\delta}_i(\mathbf{y})$. We can then use a standard decoding algorithm, such as Viterbi, on the model $p(\mathbf{y}|\mathbf{x}, \mathbf{w}')$.

In the special case of 0-1 loss, the optimum will either be the best solution, $\text{argmax}_{\mathbf{y}} \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_i, \mathbf{y})$, with a value of of $0 - \mathbf{w}^T \delta_i(\hat{\mathbf{y}})$, or it will be the second best solution, i.e.,

$$\tilde{\mathbf{y}} = \underset{\mathbf{y} \neq \hat{\mathbf{y}}}{\text{argmax}}\, \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_i, \mathbf{y}) \tag{19.108}$$

which achieves an overall value of $1 - \mathbf{w}^T \delta_i(\tilde{\mathbf{y}})$. For chain structured CRFs, we can use the Viterbi algorithm to do decoding; the second best path will differ from the best path in a single position, which can be obtained by changing the variable whose max marginal is closest to its decision boundary to its second best value. We can generalize this (with a bit more work) to find the N-best list (Schwarz and Chow 1990; Nilsson and Goldberger 2001).

For Hamming loss, $L(\mathbf{y}^*, \mathbf{y}) = \sum_t \mathbb{I}(y_t^* \neq y_t)$, and for the F1 score (defined in Section 5.7.2.3), we can devise a dynamic programming algorithm to compute Equation 19.107. See (Altun et al. 2006) for details. Other models and loss function combinations will require different methods.

19.7.3.2 A linear time algorithm

Although the above algorithm takes polynomial time, we can do better, and devise an algorithm that runs in *linear* time, assuming we use a linear kernel (i.e., we work with the original features $\boldsymbol{\phi}(\mathbf{x}, \mathbf{y})$ and do not apply the kernel trick). The basic idea, as explained in (Joachims et al. 2009), is to have a single slack variable, ξ, instead of N, but to use $|\mathcal{Y}|^N$ constraints, instead of

just $N|\mathcal{Y}|$. Specifically, we optimize the following (assuming the margin rescaling formulation):

$$\min_{\mathbf{w},\xi\geq 0} \quad \frac{1}{2}||\mathbf{w}||_2^2 + C\xi$$
$$\text{s.t.} \quad \forall(\overline{\mathbf{y}}_1,\ldots,\overline{\mathbf{y}}_N) \in \mathcal{Y}^N : \frac{1}{N}\mathbf{w}^T \sum_{i=1}^{N} \boldsymbol{\delta}_i(\overline{\mathbf{y}}_i) \geq \frac{1}{N}\sum_{i=1}^{N} L(\mathbf{y}_i,\overline{\mathbf{y}}_i) - \xi \quad (19.109)$$

Compare this to the original version, which was

$$\min_{\mathbf{w},\boldsymbol{\xi}\geq 0} \frac{1}{2}||\mathbf{w}||_2^2 + \frac{C}{N}\xi \quad \text{s.t.} \quad \forall i = 1:N, \forall \mathbf{y} \in \mathcal{Y} : \mathbf{w}^T\boldsymbol{\delta}_i(\mathbf{y}) \geq L(\mathbf{y}_i,\overline{\mathbf{y}}_i) - \xi_i \quad (19.110)$$

One can show that any solution $\mathbf{w}^*$ of Equation 19.109 is also a solution of Equation 19.110 and vice versa, with $\xi^* = \frac{1}{N}\xi_i^*$.

Algorithm 19.4: Cutting plane algorithm for SSVMs (margin rescaling, 1-slack version)

1 Input $\mathcal{D} = \{(\mathbf{x}_1,\mathbf{y}_1),\ldots,(\mathbf{x}_N,\mathbf{y}_n)\}$, C, ϵ ;
2 $\mathcal{W} = \emptyset$;
3 **repeat**
4 $\quad(\mathbf{w},\xi) = \text{argmin}_{\mathbf{w},\xi\geq 0}\ \frac{1}{2}||\mathbf{w}||_2^2 + C\sum_{i=1}^{N}\xi$;
5 $\quad\quad$ s.t. $\forall(\overline{\mathbf{y}}_1,\ldots,\overline{\mathbf{y}}_N) \in \mathcal{W} : \frac{1}{N}\mathbf{w}^T\sum_{i=1}^{N}\boldsymbol{\delta}_i(\overline{\mathbf{y}}_i) \geq \frac{1}{N}\sum_{i=1}^{N} L(\mathbf{y}_i,\overline{\mathbf{y}}_i) - \xi$;
6 $\quad$ **for** $i = 1:N$ **do**
7 $\quad\quad\hat{\mathbf{y}}_i = \text{argmax}_{\hat{\mathbf{y}}_i\in\mathcal{Y}}\, L(\mathbf{y}_i,\hat{\mathbf{y}}_i) + \mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}_i,\hat{\mathbf{y}}_i)$
8 $\quad\mathcal{W} = \mathcal{W}\cup\{(\hat{\mathbf{y}}_1,\ldots,\hat{\mathbf{y}}_N)\}$;
9 **until** $\frac{1}{N}\sum_{i=1}^{N} L(\mathbf{y}_i,\hat{\mathbf{y}}_i) - \frac{1}{N}\mathbf{w}^T\sum_{i=1}^{N}\boldsymbol{\delta}_i(\hat{\mathbf{y}}_i) \leq \xi + \epsilon$;
10 Return $(\mathbf{w},\xi)$

We can optimize Equation 19.109 using the cutting plane algorithm in Algorithm 10. (This is what is implemented in SVMstruct.) The inner QP in line 4 can be solved in $O(N)$ time using the method of (Joachims 2006). In line 7 we make N calls to the loss-augmented decoder. Finally, it can be shown that the number of iterations is a constant independent on N. Thus the overall running time is linear.

19.7.4 Online algorithms for fitting SSVMs

Although the cutting plane algorithm can be made to run in time linear in the number of data points, that can still be slow if we have a large dataset. In such cases, it is preferable to use online learning. We briefly mention a few possible algorithms below.

19.7.4.1 The structured perceptron algorithm

A very simple algorithm for fitting SSVMs is the **structured perceptron algorithm** (Collins 2002). This method is an extension of the regular perceptron algorithm of Section 8.5.4. At each

step, we compute $\hat{\mathbf{y}} = \text{argmax}\, p(\mathbf{y}|\mathbf{x})$ (e.g., using the Viterbi algorithm) for the current training sample $\mathbf{x}$. If $\hat{\mathbf{y}} = \mathbf{y}$, we do nothing, otherwise we update the weight vector using

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \boldsymbol{\phi}(\mathbf{y}, \mathbf{x}) - \boldsymbol{\phi}(\hat{\mathbf{y}}, \mathbf{x}) \tag{19.111}$$

To get good performance, it is necessary to average the parameters over the last few updates (see Section 8.5.2 for details), rather than using the most recent value.

19.7.4.2 Stochastic subgradient descent

The disadvantage of the structured perceptron algorithm is that it implicitly assumes 0-1 loss, and it does not enforce any kind of margin. An alternative approach is to perform stochastic subgradient descent. A specific instance of this the **Pegasos** algorithm (Shalev-Shwartz et al. 2007), which stands for "primal estimated sub-gradient solver for SVM". Pegasos was designed for binary SVMs, but can be extended to SSVMS.

Let us start by considering the objective function:

$$f(\mathbf{w}) = \sum_{i=1}^{N} \max_{\hat{\mathbf{y}}_i} \left[L(\mathbf{y}_i, \hat{\mathbf{y}}_i) + \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_i, \hat{\mathbf{y}}_i) \right] - \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_i, \mathbf{y}_i) + \lambda ||\mathbf{w}||^2 \tag{19.112}$$

Letting $\hat{\mathbf{y}}_i$ be the argmax of this max. Then the subgradient of this objective function is

$$g(\mathbf{w}) = \sum_{i=1}^{N} \boldsymbol{\phi}(\mathbf{x}_i, \hat{\mathbf{y}}_i) - \boldsymbol{\phi}(\mathbf{x}_i, \mathbf{y}_i) + 2\lambda \mathbf{w} \tag{19.113}$$

In stochastic subgradient descent, we approximate this gradient with a single term, i, and then perform an update:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \eta_k g_i(\mathbf{w}_k) = \mathbf{w}_k - \eta_k [\boldsymbol{\phi}(\mathbf{x}_i, \hat{\mathbf{y}}_i) - \boldsymbol{\phi}(\mathbf{x}_i, \mathbf{y}_i) + (2/N)\lambda \mathbf{w}] \tag{19.114}$$

where η_k is the step size parameter, which should satisfy the Robbins-Monro conditions (Section 8.5.2.1). (Notice that the perceptron algorithm is just a special case where $\lambda = 0$ and $\eta_k = 1$.) To ensure that $\mathbf{w}$ has unit norm, we can project it onto the ℓ_2 ball after each update.

19.7.5 Latent structural SVMs

In many applications of interest, we have latent or hidden variables $\mathbf{h}$. For example, in object detections problems, we may be told that the image contains an object, so $y = 1$, but we may not know where it is. The location of the object, or its pose, can be considered a hidden variable. Or in machine translation, we may know the source text $\mathbf{x}$ (say English) and the target text $\mathbf{y}$ (say French), but we typically do not know the alignment between the words.

We will extend our model as follows, to get a latent CRF:

$$p(\mathbf{y}, \mathbf{h}|\mathbf{x}, \mathbf{w}) = \frac{\exp(\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}, \mathbf{y}, \mathbf{h}))}{Z(\mathbf{x}, \mathbf{w})} \tag{19.115}$$

$$Z(\mathbf{x}, \mathbf{w}) = \sum_{\mathbf{y}, \mathbf{h}} \exp(\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}, \mathbf{y}, \mathbf{h})) \tag{19.116}$$

In addition, we introduce the loss function $L(\mathbf{y}^*, \mathbf{y}, \mathbf{h})$; this measures the loss when the "action" that we take is to predict $\mathbf{y}$ using latent variables $\mathbf{h}$. We could just use $L(\mathbf{y}^*, \mathbf{y})$ as before, since $\mathbf{h}$ is usually a nuisance variable and not of direct interest. However, $\mathbf{h}$ can sometimes play a useful role in defining a loss function.[12]

Given the loss function, we define our objective as

$$R_{EL}(\mathbf{w}) = -\log p(\mathbf{w}) + \sum_i \log \left[\sum_{\mathbf{y},\mathbf{h}} \exp \tilde{L}(\mathbf{y}_i, \mathbf{y}, \mathbf{h}) \frac{\exp(\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}, \mathbf{y}, \mathbf{h}))}{Z(\mathbf{x}, \mathbf{w})} \right] \tag{19.117}$$

Using the same loose lower bound as before, we get

$$\begin{aligned} R_{EL}(\mathbf{w}) &\leq E(\mathbf{w}) + \sum_{i=1}^{N} \max_{\mathbf{y},\mathbf{h}} \left\{ \tilde{L}(\mathbf{y}_i, \mathbf{y}, \mathbf{h}) + \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_i, \mathbf{y}, \mathbf{h}) \right\} \\ &\quad - \sum_{i=1}^{N} \max_{\mathbf{h}} \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_i, \mathbf{y}_i, \mathbf{h}) \end{aligned} \tag{19.118}$$

If we set $E(\mathbf{w}) = -\frac{1}{2C}||\mathbf{w}||_2^2$, we get the same objective as is optimized in **latent SVMs** (Yu and Joachims 2009).

Unfortunately, this objective is no longer convex. However, it is a difference of convex functions, and hence can be solved efficiently using the **CCCP** or **concave-convex procedure** (Yuille and Rangarajan 2003). This is a method for minimizing functions of the form $f(\mathbf{w}) - g(\mathbf{w})$, where f and g are convex. The method alternates between finding a linear upper bound u on $-g$, and then minimizing the convex function $f(\mathbf{w}) + u(\mathbf{w})$; see Algorithm 6 for the pseudocode. CCCP is guaranteed to decrease the objective at every iteration, and to converge to a local minimum or a saddle point.

Algorithm 19.5: Concave-Convex Procedure (CCCP)

```
Set t = 0 and initialize w_0 ;
repeat
    Find hyperplane v_t such that -g(w) ≤ -g(w_t) + (w - w_t)^T v_t for all w ;
    Solve w_{t+1} = argmin_w f(w) + w^T v_t ;
    Set t = t + 1
until converged;
```

When applied to latent SSVMs, CCCP is very similar to (hard) EM. In the "E step", we compute

12. For example, consider the problem of learning to classify a set of documents as relevant or not to a query. That is, given n documents $\mathbf{x}_1, \ldots, \mathbf{x}_n$ for a single query q, we want to produce a labeling $y_j \in \{-1, +1\}$, representing whether document j is relevant to q or not. Suppose our goal is to maximize the precision at k, which is a metric widely used in ranking (see Section 9.7.4). We will introduce a latent variable for each document h_j representing its degree of relevance. This corresponds to a latent total ordering, that has to be consistent with the observed partial ordering $\mathbf{y}$. Given this, we can define the following loss function: $L(\mathbf{y}, \hat{\mathbf{y}}, \hat{\mathbf{h}}) = \min\{1, \frac{n(\mathbf{y})}{k}\} - \frac{1}{k}\sum_{j=1}^{k} \mathbb{I}(y_{h_j} = 1)$, where $n(\mathbf{y})$ is the total number of relevant documents. This loss is essentially just 1 minus the precision@k, except we replace 1 with $n(\mathbf{y})/k$ so that the loss will have a minimum of zero. See (Yu and Joachims 2009) for details.

the linear upper bound by setting $\mathbf{v}_t = -C \sum_{i=1}^{N} \boldsymbol{\phi}(\mathbf{x}_i, \mathbf{y}_i, \mathbf{h}_i^*)$, where

$$\mathbf{h}_i = \underset{\mathbf{h}}{\operatorname{argmax}}\, \mathbf{w}_t^T \boldsymbol{\phi}(\mathbf{x}_i, \mathbf{y}_i, \mathbf{h}) \tag{19.119}$$

In the "M step", we estimate $\mathbf{w}$ using techniques for solving fully visible SSVMs. Specifically, we minimize

$$\frac{1}{2}||\mathbf{w}||_2 + C \sum_{i=1}^{N} \max_{\mathbf{y},\mathbf{h}} \left\{ L(\mathbf{y}_i, \mathbf{y}, \mathbf{h}) + \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_i, \mathbf{y}, \mathbf{h}) \right\} - C \sum_{i=1}^{N} \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_i, \mathbf{y}_i, \mathbf{h}_i^*) \tag{19.120}$$

Exercises

Exercise 19.1 Derivative of the log partition function

Derive Equation 19.40.

Exercise 19.2 CI properties of Gaussian graphical models

(Source: Jordan.)

In this question, we study the relationship between sparse matrices and sparse graphs for Gaussian graphical models. Consider a multivariate Gaussian $\mathcal{N}(x|\mu, \Sigma)$ in 3 dimensions. Suppose $\mu = (0, 0, 0)^T$ throughout.

Recall that for jointly Gaussian random variables, we know that X_i and X_j are independent iff they are uncorrelated, ie. $\Sigma_{ij} = 0$. (This is not true in general, or even if X_i and X_j are Gaussian but not jointly Gaussian.) Also, X_i is conditionally independent of X_j given all the other variables iff $\Sigma_{ij}^{-1} = 0$.

a. Suppose

$$\Sigma = \begin{pmatrix} 0.75 & 0.5 & 0.25 \\ 0.5 & 1.0 & 0.5 \\ 0.25 & 0.5 & 0.75 \end{pmatrix}$$

Are there any marginal independencies amongst X_1, X_2 and X_3? What about conditional independencies? Hint: compute Σ^{-1} and expand out $x^T \Sigma^{-1} x$: which pairwise terms $x_i x_j$ are missing? Draw an undirected graphical model that captures as many of these independence statements (marginal and conditional) as possible, but does not make any false independence assertions.

b. Suppose

$$\Sigma = \begin{pmatrix} 2 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 2 \end{pmatrix}$$

Are there any marginal independencies amongst X_1, X_2 and X_3? Are there any conditional independencies amongst X_1, X_2 and X_3? Draw an undirected graphical model that captures as many of these independence statements (marginal and conditional) as possible, but does not make any false independence assertions.

c. Now suppose the distribution on X can be represented by the following DAG:

$$X_1 \rightarrow X_2 \rightarrow X_3$$

Let the CPDs be as follows:

$$P(X_1) = \mathcal{N}(X_1; 0, 1),\ P(X_2|x_1) = \mathcal{N}(X_2; x_1, 1),\ P(X_3|x_2) = \mathcal{N}(X_3; x_2, 1) \tag{19.121}$$

Multiply these 3 CPDs together and complete the square (Bishop p101) to find the corresponding joint distribution $\mathcal{N}(X_{1:3}|\mu, \Sigma)$. (You may find it easier to solve for Σ^{-1} rather than Σ.)

d. For the DAG model in the previous question: Are there any marginal independencies amongst X_1, X_2 and X_3? What about conditional independencies? Draw an undirected graphical model that captures as many of these independence statements as possible, but does not make any false independence assertions (either marginal or conditional).

Exercise 19.3 Independencies in Gaussian graphical models

(Source: MacKay.)

a. Consider the DAG $X1 \leftarrow X2 \rightarrow X3$. Assume that all the CPDs are linear-Gaussian. Which of the following matrices *could* be the covariance matrix?

$$A = \begin{pmatrix} 9 & 3 & 1 \\ 3 & 9 & 3 \\ 1 & 3 & 9 \end{pmatrix}, B = \begin{pmatrix} 8 & -3 & 1 \\ -3 & 9 & -3 \\ 1 & -3 & 8 \end{pmatrix}, C = \begin{pmatrix} 9 & 3 & 0 \\ 3 & 9 & 3 \\ 0 & 3 & 9 \end{pmatrix}, D = \begin{pmatrix} 9 & -3 & 0 \\ -3 & 10 & -3 \\ 0 & -3 & 9 \end{pmatrix} \quad (19.122)$$

b. Which of the above matrices could be inverse covariance matrix?

c. Consider the DAG $X1 \rightarrow X2 \leftarrow X3$. Assume that all the CPDs are linear-Gaussian. Which of the above matrices could be the covariance matrix?

d. Which of the above matrices could be the inverse covariance matrix?

e. Let three variables x_1, x_2, x_4 have covariance matrix $\mathbf{\Sigma}_{(1:3)}$ and precision matrix $\mathbf{\Omega}_{(1:3)} = \mathbf{\Sigma}_{(1:3)}^{-1}$ as follows

$$\mathbf{\Sigma}_{(1:3)} = \begin{pmatrix} 1 & 0.5 & 0 \\ 0.5 & 1 & 0.5 \\ 0 & 0.5 & 1 \end{pmatrix}, \mathbf{\Omega}_{(1:3)} = \begin{pmatrix} 1.5 & -1 & 0.5 \\ -1 & 2 & -1 \\ 0.5 & -1 & 1.5 \end{pmatrix} \quad (19.123)$$

Now focus on x_1 and x_2. Which of the following statements about their covariance matrix $\mathbf{\Sigma}_{(1:2)}$ and precision matrix $\mathbf{\Omega}_{(1:2)}$ are true?

$$A : \mathbf{\Sigma}_{(1:2)} = \begin{pmatrix} 1 & 0.5 \\ 0.5 & 1 \end{pmatrix}, \; B : \mathbf{\Omega}_{(1:2)} = \begin{pmatrix} 1.5 & -1 \\ -1 & 2 \end{pmatrix} \quad (19.124)$$

Exercise 19.4 Cost of training MRFs and CRFs

(Source: Koller.) Consider the process of gradient-ascent training for a log-linear model with k features, given a data set with N training instances. Assume for simplicity that the cost of computing a single feature over a single instance in our data set is constant, as is the cost of computing the expected value of each feature once we compute a marginal over the variables in its scope. Assume that it takes c time to compute all the marginals for each data case. Also, assume that we need r iterations for the gradient process to converge.

- Using this notation, what is the time required to train an MRF in big-O notation?
- Using this notation, what is the time required to train a CRF in big-O notation?

Exercise 19.5 Full conditional in an Ising model

Consider an Ising model

$$p(x_1, \ldots, x_n|\boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})} \prod_{<ij>} \exp(J_{ij} x_i x_j) \prod_{i=1}^{n} \exp(h_i x_i) \quad (19.125)$$

where $< ij >$ denotes all unique pairs (i.e., all edges), $J_{ij} \in \mathbb{R}$ is the coupling strength (weight) on edge $i - j$, $h_i \in \mathbb{R}$ is the local evidence (bias term), and $\boldsymbol{\theta} = (\mathbf{J}, \mathbf{h})$ are all the parameters.

If $x_i \in \{0, 1\}$, derive an expression for the full conditional

$$p(x_i = 1|\mathbf{x}_{-i}, \boldsymbol{\theta}) = p(x_i = 1|\mathbf{x}_{nb_i}, \boldsymbol{\theta}) \tag{19.126}$$

where $\mathbf{x}_{-i}$ are all nodes except i, and nb_i are the neighbors of i in the graph. Hint: you answer should use the sigmoid/ logistic function $\sigma(z) = 1/(1 + e^{-z})$. Now suppose $x_i \in \{-1, +1\}$. Derive a related expression for $p(x_i|\mathbf{x}_{-i}, \boldsymbol{\theta})$ in this case. (This result can be used when applying Gibbs sampling to the model.)

20 *Exact inference for graphical models*

20.1 Introduction

In Section 17.4.3, we discussed the forwards-backwards algorithm, which can exactly compute the posterior marginals $p(x_t|\mathbf{v}, \boldsymbol{\theta})$ in any chain-structured graphical model, where $\mathbf{x}$ are the hidden variables (assumed discrete) and $\mathbf{v}$ are the visible variables. This algorithm can be modified to compute the posterior mode and posterior samples. A similar algorithm for linear-Gaussian chains, known as the Kalman smoother, was discussed in Section 18.3.2. Our goal in this chapter is to generalize these exact inference algorithms to arbitrary graphs. The resulting methods apply to both directed and undirected graphical models. We will describe a variety of algorithms, but we omit their derivations for brevity. See e.g., (Darwiche 2009; Koller and Friedman 2009) for a detailed exposition of exact inference techniques for discrete directed graphical models.

20.2 Belief propagation for trees

In this section, we generalize the forwards-backwards algorithm from chains to trees. The resulting algorithm is known as **belief propagation** (**BP**) (Pearl 1988), or the **sum-product algorithm**.

20.2.1 Serial protocol

We initially assume (for notational simplicity) that the model is a pairwise MRF (or CRF), i.e.,

$$p(\mathbf{x}|\mathbf{v}) = \frac{1}{Z(\mathbf{v})} \prod_{s \in \mathcal{V}} \psi_s(x_s) \prod_{(s,t) \in \mathcal{E}} \psi_{s,t}(x_s, x_t) \tag{20.1}$$

where ψ_s is the local evidence for node s, and ψ_{st} is the potential for edge $s - t$. We will consider the case of models with higher order cliques (such as directed trees) later on.

One way to implement BP for undirected trees is as follows. Pick an arbitrary node and call it the root, r. Now orient all edges away from r (intuitively, we can imagine "picking up the graph" at node r and letting all the edges "dangle" down). This gives us a well-defined notion of parent and child. Now we send messages up from the leaves to the root (the **collect evidence** phase) and then back down from the root (the **distribute evidence** phase), in a manner analogous to forwards-backwards on chains.

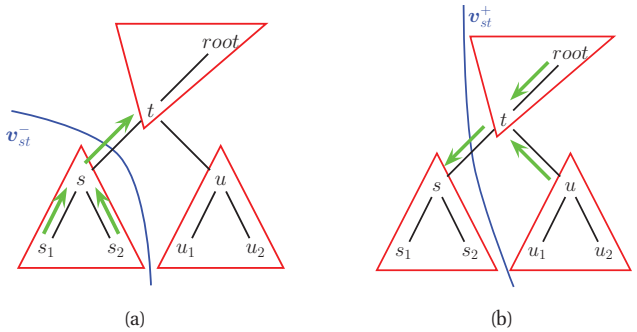


Figure 20.1 Message passing on a tree. (a) Collect-to-root phase. (b) Distribute-from-root phase.

To explain the process in more detail, consider the example in Figure 20.1. Suppose we want to compute the belief state at node t. We will initially condition the belief only on evidence that is at or below t in the graph, i.e., we want to compute $\text{bel}_t^-(x_t) \triangleq p(x_t|\mathbf{v}_t^-)$. We will call this a "bottom-up belief state". Suppose, by induction, that we have computed "messages" from t's two children, summarizing what they think t should know about the evidence in their subtrees, i.e., we have computed $m_{s\to t}^-(x_t) = p(x_t|\mathbf{v}_{st}^-)$, where $\mathbf{v}_{st}^-$ is all the evidence on the downstream side of the $s-t$ edge (see Figure 20.1(a)), and similarly we have computed $m_{u\to t}(x_t)$. Then we can compute the bottom-up belief state at t as follows:

$$\text{bel}_t^-(x_t) \triangleq p(x_t|\mathbf{v}_t^-) = \frac{1}{Z_t}\psi_t(x_t) \prod_{c\in \text{ch}(t)} m_{c\to t}^-(x_t) \tag{20.2}$$

where $\psi_t(x_t) \propto p(x_t|\mathbf{v}_t)$ is the local evidence for node t, and Z_t is the local normalization constant. In words, we multiply all the incoming messages from our children, as well as the incoming message from our local evidence, and then normalize.

We have explained how to compute the bottom-up belief states from the bottom-up messages. How do we compute the messages themselves? Consider computing $m_{s\to t}^-(x_t)$, where s is one of t's children. Assume, by recursion, that we have computed $\text{bel}_s^-(x_s) = p(x_s|\mathbf{v}_s^-)$. Then we can compute the message as follows:

$$m_{s\to t}^-(x_t) = \sum_{x_s} \psi_{st}(x_s, x_t)\text{bel}_s^-(x_s) \tag{20.3}$$

Essentially we convert beliefs about x_s into beliefs about x_t by using the edge potential ψ_{st}.

We continue in this way up the tree until we reach the root. Once at the root, we have "seen" all the evidence in the tree, so we can compute our local belief state at the root using

$$\text{bel}_r(x_r) \triangleq p(x_r|\mathbf{v}) = p(x_t|\mathbf{v}_r^-) \propto \psi_r(x_r) \prod_{c\in \text{ch}(r)} m_{c\to r}^-(x_r) \tag{20.4}$$

This completes the end of the upwards pass, which is analogous to the forwards pass in an HMM. As a "side effect", we can compute the probability of the evidence by collecting the

normalization constants:

$$p(\mathbf{v}) = \prod_t Z_t \tag{20.5}$$

We can now pass messages down from the root. For example, consider node s, with parent t, as shown in Figure 20.1(b). To compute the belief state for s, we need to combine the bottom-up belief for s together with a top-down message from t, which summarizes all the information in the rest of the graph, $m^+_{t\to s}(x_s) \triangleq p(x_t|\mathbf{v}^+_{st})$, where $\mathbf{v}^+_{st}$ is all the evidence on the upstream (root) side of the $s-t$ edge, as shown in Figure 20.1(b). We then have

$$\text{bel}_s(x_s) \triangleq p(x_s|\mathbf{v}) \propto \text{bel}^-_s(x_s) \prod_{t\in \text{pa}(s)} m^+_{t\to s}(x_t) \tag{20.6}$$

How do we compute these downward messages? For example, consider the message from t to s. Suppose t's parent is r, and t's children are s and u, as shown in Figure 20.1(b). We want to include in $m^+_{t\to s}$ all the information that t has received, except for the information that s sent it:

$$m^+_{t\to s}(x_s) \triangleq p(x_s|\mathbf{v}^+_{st}) = \sum_{x_t} \psi_{st}(x_s, x_t) \frac{\text{bel}_t(x_t)}{m^-_{s\to t}(x_t)} \tag{20.7}$$

Rather than dividing out the message sent up to t, we can plug in the equation of bel_t to get

$$m^+_{t\to s}(x_s) = \sum_{x_t} \psi_{st}(x_s, x_t)\psi_t(x_t) \prod_{c\in \text{ch}(t), c\neq s} m^-_{c\to t}(x_t) \prod_{p\in \text{pa}(t)} m^+_{p\to t}(x_t) \tag{20.8}$$

In other words, we multiply together all the messages coming into t from all nodes except for the recipient s, combine together, and then pass through the edge potential ψ_{st}. In the case of a chain, t only has one child s and one parent p, so the above simplifies to

$$m^+_{t\to s}(x_s) = \sum_{x_t} \psi_{st}(x_s, x_t)\psi_t(x_t) m^+_{p\to t}(x_t) \tag{20.9}$$

The version of BP in which we use division is called **belief updating**, and the version in which we multiply all-but-one of the messages is called **sum-product**. The belief updating version is analogous to how we formulated the Kalman smoother in Section 18.3.2: the top-down messages depend on the bottom-up messages. This means they can be interpreted as conditional posterior probabilities. The sum-product version is analogous to how we formulated the backwards algorithm in Section 17.4.3: the top-down messages are completely independent of the bottom-up messages, which means they can only be interpreted as conditional likelihoods. See Section 18.3.2.3 for a more detailed discussion of this subtle difference.

20.2.2 Parallel protocol

So far, we have presented a serial version of the algorithm, in which we send messages up to the root and back. This is the optimal approach for a tree, and is a natural extension of forwards-backwards on chains. However, as a prelude to handling general graphs with loops, we now consider a parallel version of BP. This gives equivalent results to the serial version but is less efficient when implemented on a serial machine.

The basic idea is that all nodes receive messages from their neighbors in parallel, they then updates their belief states, and finally they send new messages back out to their neighbors. This process repeats until convergence. This kind of computing architecture is called a **systolic array**, due to its resemblance to a beating heart.

More precisely, we initialize all messages to the all 1's vector. Then, in parallel, each node absorbs messages from all its neighbors using

$$\text{bel}_s(x_s) \propto \psi_s(x_s) \prod_{t \in \text{nbr}_s} m_{t \to s}(x_s) \tag{20.10}$$

Then, in parallel, each node sends messages to each of its neighbors:

$$m_{s \to t}(x_t) = \sum_{x_s} \left(\psi_s(x_s) \psi_{st}(x_s, x_t) \prod_{u \in \text{nbr}_s \setminus t} m_{u \to s}(x_s) \right) \tag{20.11}$$

The $m_{s \to t}$ message is computed by multiplying together all incoming messages, except the one sent by the recipient, and then passing through the ψ_{st} potential.

At iteration T of the algorithm, $\text{bel}_s(x_s)$ represents the posterior belief of x_s conditioned on the evidence that is T steps away in the graph. After $D(G)$ steps, where $D(G)$ is the **diameter** of the graph (the largest distance between any two pairs of nodes), every node has obtained information from all the other nodes. Its local belief state is then the correct posterior marginal. Since the diameter of a tree is at most $|\mathcal{V}| - 1$, the algorithm converges in a linear number of steps.

We can actually derive the up-down version of the algorithm by imposing the condition that a node can only send a message once it has received messages from all its other neighbors. This means we must start with the leaf nodes, which only have one neighbor. The messages then propagate up to the root and back. We can also update the nodes in a random order. The only requirement is that each node get updated $D(G)$ times. This is just enough time for information to spread throughout the whole tree.

Similar parallel, distributed algorithms for solving linear systems of equations are discussed in (Bertsekas 1997). In particular, the Gauss-Seidel algorithm is analogous to the serial up-down version of BP, and the Jacobi algorithm is analogous to the parallel version of BP.

20.2.3 Gaussian BP *

Now consider the case where $p(\mathbf{x}|\mathbf{v})$ is jointly Gaussian, so it can be represented as a Gaussian pairwise MRF, as in Section 19.4.4. We now present the belief propagation algorithm for this class of models, follow the presentation of (Bickson 2009) (see also (Malioutov et al. 2006)). We will assume the following node and edge potentials:

$$\psi_t(x_t) = \exp(-\frac{1}{2} A_{tt} x_t^2 + b_t x_t) \tag{20.12}$$

$$\psi_{st}(x_s, x_t) = \exp(-\frac{1}{2} x_s A_{st} x_t) \tag{20.13}$$

so the overall model has the form

$$p(\mathbf{x}|\mathbf{v}) \propto \exp(-\frac{1}{2} \mathbf{x}^T \mathbf{A} \mathbf{x} + \mathbf{b}^T \mathbf{x}) \tag{20.14}$$

This is the information form of the MVN (see Exercise 9.2), where $\mathbf{A}$ is the precision matrix. Note that by completing the square, the local evidence can be rewritten as a Gaussian:

$$\psi_t(x_t) \propto \mathcal{N}(b_t/A_{tt}, A_{tt}^{-1}) \triangleq \mathcal{N}(m_t, \ell_t^{-1}) \tag{20.15}$$

Below we describe how to use BP to compute the posterior node marginals,

$$p(x_t|\mathbf{v}) = \mathcal{N}(\mu_t, \lambda_t^{-1}) \tag{20.16}$$

If the graph is a tree, the method is exact. If the graph is loopy, the posterior means may still be exact, but the posterior variances are often too small (Weiss and Freeman 1999).

Although the precision matrix $\mathbf{A}$ is often sparse, computing the posterior mean requires inverting it, since $\boldsymbol{\mu} = \mathbf{A}^{-1}\mathbf{b}$. BP provides a way to exploit graph structure to perform this computation in $O(D)$ time instead of $O(D^3)$. This is related to various methods from linear algebra, as discussed in (Bickson 2009).

Since the model is jointly Gaussian, all marginals and all messages will be Gaussian. The key operations we need are to multiply together two Gaussian factors, and to marginalize out a variable from a joint Gaussian factor.

For multiplication, we can use the fact that the product of two Gaussians is Gaussian:

$$\mathcal{N}(x|\mu_1, \lambda_1^{-1}) \times \mathcal{N}(x|\mu_2, \lambda_2^{-1}) = C\mathcal{N}(x|\mu, \lambda^{-1}) \tag{20.17}$$

$$\lambda = \lambda_1 + \lambda_2 \tag{20.18}$$

$$\mu = \lambda^{-1}(\mu_1\lambda_1 + \mu_2\lambda_2) \tag{20.19}$$

where

$$C = \sqrt{\frac{\lambda}{\lambda_1\lambda_2}} \exp\left(\frac{1}{2}(\lambda_1\mu_1^2(\lambda^{-1}\lambda_1 - 1) + \lambda_2\mu_2^2(\lambda^{-1}\lambda_2 - 1) + 2\lambda^{-1}\lambda_1\lambda_2\mu_1\mu_2)\right) \tag{20.20}$$

See Exercise 20.2 for the proof.

For marginalization, we have the following result:

$$\int \exp(-ax^2 + bx)dx = \sqrt{\pi/a}\exp(b^2/4a) \tag{20.21}$$

which follows from the normalization constant of a Gaussian (Exercise 2.11).

We now have all the pieces we need. In particular, let the message $m_{s\to t}(x_t)$ be a Gaussian with mean μ_{st} and precision λ_{st}. From Equation 20.10, the belief at node s is given by the product of incoming messages times the local evidence (Equation 20.15) and hence

$$\text{bel}_s(x_s) = \psi_s(x_s) \prod_{t\in\text{nbr}(s)} m_{ts}(x_s) = \mathcal{N}(x_s|\mu_s, \lambda_s^{-1}) \tag{20.22}$$

$$\lambda_s = \ell_s + \sum_{t\in\text{nbr}(s)} \lambda_{ts} \tag{20.23}$$

$$\mu_s = \lambda_s^{-1}\left(\ell_s m_s + \sum_{t\in\text{nbr}(s)} \lambda_{ts}\mu_{ts}\right) \tag{20.24}$$

To compute the messages themselves, we use Equation 20.11, which is given by

$$m_{s\to t}(x_t) = \int_{x_s} \left(\psi_{st}(x_s, x_t)\psi_s(x_s) \prod_{u \in \text{nbr}_s \setminus t} m_{u\to s}(x_s) \right) dx_s \tag{20.25}$$

$$= \int_{x_s} \psi_{st}(x_s, x_t) f_{s\setminus t}(x_s) dx_s \tag{20.26}$$

where $f_{s\setminus t}(x_s)$ is the product of the local evidence and all incoming messages excluding the message from t:

$$f_{s\setminus t}(x_s) \triangleq \psi_s(x_s) \prod_{u \in \text{nbr}_s \setminus t} m_{u\to s}(x_s) \tag{20.27}$$

$$= \mathcal{N}(x_s | \mu_{s\setminus t}, \lambda_{s\setminus t}^{-1}) \tag{20.28}$$

$$\lambda_{s\setminus t} \triangleq \ell_s + \sum_{u \in \text{nbr}(s)\setminus t} \lambda_{us} \tag{20.29}$$

$$\mu_{s\setminus t} \triangleq \lambda_{s\setminus t}^{-1} \left(\ell_s m_s + \sum_{u \in \text{nbr}(s)\setminus t} \lambda_{us}\mu_{us} \right) \tag{20.30}$$

Returning to Equation 20.26 we have

$$m_{s\to t}(x_t) = \int_{x_s} \underbrace{\exp(-x_s A_{st} x_t)}_{\psi_{st}(x_s,x_t)} \underbrace{\exp(-\lambda_{s\setminus t}/2(x_s - \mu_{s\setminus t})^2)}_{f_{s\setminus t}(x_s)} dx_s \tag{20.31}$$

$$= \int_{x_s} \exp\left((-\lambda_{s\setminus t} x_s^2/2) + (\lambda_{s\setminus t}\mu_{s\setminus t} - A_{st}x_t)x_s \right) dx_s + \text{const} \tag{20.32}$$

$$\propto \exp\left((\lambda_{s\setminus t}\mu_{s\setminus t} - A_{st}x_t)^2/(2\lambda_{s\setminus t}) \right) \tag{20.33}$$

$$\propto \mathcal{N}(\mu_{st}, \lambda_{st}^{-1}) \tag{20.34}$$

$$\lambda_{st} = A_{st}^2/\lambda_{s\setminus t} \tag{20.35}$$

$$\mu_{st} = A_{st}\mu_{s\setminus t}/\lambda_{st} \tag{20.36}$$

One can generalize these equations to the case where each node is a vector, and the messages become small MVNs instead of scalar Gaussians (Alag and Agogino 1996). If we apply the resulting algorithm to a linear dynamical system, we recover the Kalman smoothing algorithm of Section 18.3.2.

To perform message passing in models with non-Gaussian potentials, one can use sampling methods to approximate the relevant integrals. This is called **non-parametric BP** (Sudderth et al. 2003; Isard 2003; Sudderth et al. 2010).

20.2.4 Other BP variants *

In this section, we briefly discuss several variants of the main algorithm.

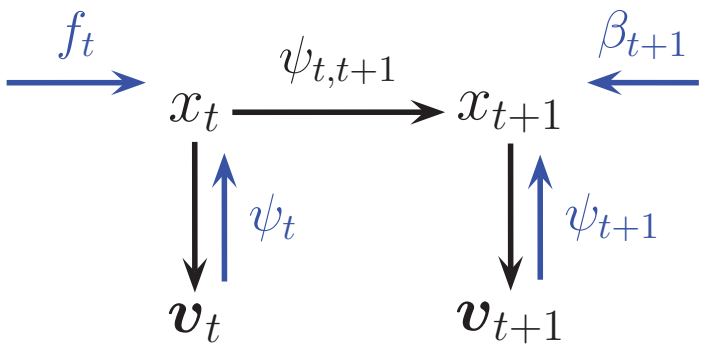


Figure 20.2 Illustration of how to compute the two-slice distribution for an HMM. The ψ_t and ψ_{t+1} terms are the local evidence messages from the visible nodes $\mathbf{v}_t$, $\mathbf{v}_{t+1}$ to the hidde nodes x_t, x_{t+1} respectively; f_t is the forwards message from x_{t-1} and β_{t+1} is the backwards message from x_{t+2}.

20.2.4.1 Max-product algorithm

It is possible to devise a **max-product** version of the BP algorithm, by replacing the $\sum$ operator with the $\max$ operator. We can then compute the local MAP marginal of each node. However, if there are ties, this might not be globally consistent, as discussed in Section 17.4.4. Fortunately, we can generalize the Viterbi algorithm to trees, where we use max and argmax in the collect-to-root phase, and perform traceback in the distribute-from-root phase. See (Dawid 1992) for details.

20.2.4.2 Sampling from a tree

It is possible to draw samples from a tree structured model by generalizing the forwards filtering / backwards sampling algorithm discussed in Section 17.4.5. See (Dawid 1992) for details.

20.2.4.3 Computing posteriors on sets of variables

In Section 17.4.3.2, we explained how to compute the "two-slice" distribution $\xi_{t,t+1}(i,j) = p(x_t = i, x_{t+1} = j|\mathbf{v})$ in an HMM, namely by using

$$\xi_{t,t+1}(i,j) \quad = \quad \alpha_t(i)\psi_{t+1}(j)\beta_{t+1}(j)\psi_{t,t+1}(i,j) \tag{20.37}$$

Since $\alpha_t(i) \propto \psi_t(i)f_t(i)$, where $f_t = p(x_t|\mathbf{v}_{1:t-1})$ is the forwards message, we can think of this as sending messages f_t and ψ_t into x_t, β_{t+1} and ϕ_{t+1} into x_{t+1}, and then combining them with the $\boldsymbol{\Psi}$ matrix, as shown in Figure 20.2. This is like treating x_t and x_{t+1} as a single "mega node", and then multiplying all the incoming messages as well as all the local factors (here, $\psi_{t,t+1}$).

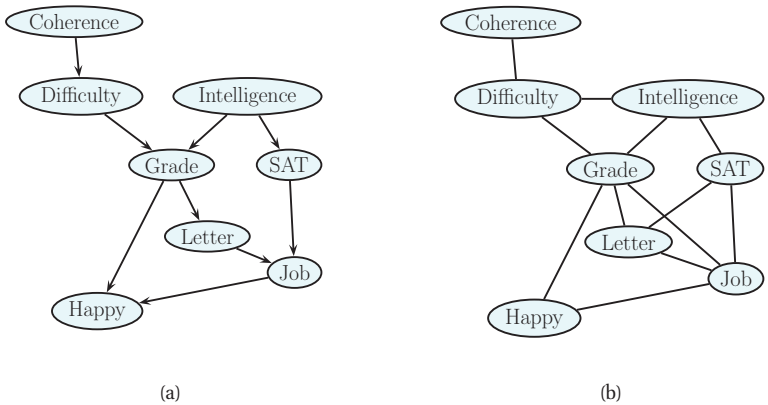


Figure 20.3 Left: The "student" DGM. Right: the equivalent UGM. We add moralization arcs D-I, G-J and L-S. Based on Figure 9.8 of (Koller and Friedman 2009).

20.3 The variable elimination algorithm

We have seen how to use BP to compute exact marginals on chains and trees. In this section, we discuss an algorithm to compute $p(\mathbf{x}_q|\mathbf{x}_v)$ for any kind of graph.

We will explain the algorithm by example. Consider the DGM in Figure 20.3(a). This model, from (Koller and Friedman 2009), is a hypothetical model relating various variables pertaining to a typical student. The corresponding joint has the following form:

$$P(C, D, I, G, S, L, J, H) \tag{20.38}$$

$$= P(C)P(D|C)P(I)P(G|I, D)P(S|I)P(L|G)P(J|L, S)P(H|G, J) \tag{20.39}$$

Note that the forms of the CPDs do not matter, since all our calculations will be symbolic. However, for illustration purposes, we will assume all variables are binary.

Before proceeding, we convert our model to undirected form. This is not required, but it makes for a more unified presentation, since the resulting method can then be applied to both DGMs and UGMs (and, as we will see in Section 20.3.1, to a variety of other problems that have nothing to do with graphical models). Since the computational complexity of inference in DGMs and UGMs is, generally speaking, the same, nothing is lost in this transformation from a computational point of view.[1]

To convert the DGM to a UGM, we simply define a potential or factor for every CPD, yielding

$$p(C, D, I, G, S, L, J, H) \tag{20.40}$$

$$= \psi_C(C)\psi_D(D, C)\psi_I(I)\psi_G(G, I, D)\psi_S(S, I)\psi_L(L, G)\psi_J(J, L, S)\psi_H(H, G, J) \tag{20.41}$$

1. There are a few "tricks" one can exploit in the directed case that cannot easily be exploited in the undirected case. One important example is **barren node removal**. To explain this, consider a naive Bayes classifier, as in Figure 10.2. Suppose we want to infer y and we observe x_1 and x_2, but not x_3 and x_4. It is clear that we can safely remove x_3 and x_4, since $\sum_{x_3} p(x_3|y) = 1$, and similarly for x_4. In general, once we have removed hidden leaves, we can apply this process recursively. Since potential functions do not necessary sum to one, we cannot use this trick in the undirected case. See (Koller and Friedman 2009) for a variety of other speedup tricks.

Since all the potentials are **locally normalized**, since they are CPDs, there is no need for a global normalization constant, so $Z = 1$. The corresponding undirected graph is shown in Figure 20.3(b). Note that it has more edges than the DAG. In particular, any "unmarried" nodes that share a child must get "married", by adding an edge between them; this process is known as **moralization**. Only then can the arrows be dropped. In this example, we added D-I, G-J, and L-S moralization arcs. The reason this operation is required is to ensure that the CI properties of the UGM match those of the DGM, as explained in Section 19.2.2. It also ensures there is a clique that can "store" the CPDs of each family.

Now suppose we want to compute $p(J = 1)$, the marginal probability that a person will get a job. Since we have 8 binary variables, we could simply enumerate over all possible assignments to all the variables (except for J), adding up the probability of each joint instantiation:

$$p(J) = \sum_L \sum_S \sum_G \sum_H \sum_I \sum_D \sum_C p(C, D, I, G, S, L, J, H) \tag{20.42}$$

However, this would take $O(2^7)$ time. We can be smarter by **pushing sums inside products**. This is the key idea behind the **variable elimination** algorithm (Zhang and Poole 1996), also called **bucket elimination** (Dechter 1996), or, in the context of genetic pedigree trees, the **peeling algorithm** (Cannings et al. 1978). In our example, we get

$$\begin{aligned} p(J) &= \sum_{L,S,G,H,I,D,C} p(C, D, I, G, S, L, J, H) \\ &= \sum_{L,S,G,H,I,D,C} \psi_C(C)\psi_D(D, C)\psi_I(I)\psi_G(G, I, D)\psi_S(S, I)\psi_L(L, G) \\ &\quad \times \psi_J(J, L, S)\psi_H(H, G, J) \\ &= \sum_{L,S} \psi_J(J, L, S) \sum_G \psi_L(L, G) \sum_H \psi_H(H, G, J) \sum_I \psi_S(S, I)\psi_I(I) \\ &\quad \times \sum_D \psi_G(G, I, D) \sum_C \psi_C(C)\psi_D(D, C) \end{aligned}$$

We now evaluate this expression, working right to left as shown in Table 20.1. First we multiply together all the terms in the scope of the $\sum_C$ operator to create the temporary factor

$$\tau'_1(C, D) = \psi_C(C)\psi_D(D, C) \tag{20.43}$$

Then we marginalize out C to get the new factor

$$\tau_1(D) = \sum_C \tau'_1(C, D) \tag{20.44}$$

Next we multiply together all the terms in the scope of the $\sum_D$ operator and then marginalize out to create

$$\tau'_2(G, I, D) = \psi_G(G, I, D)\tau_1(D) \tag{20.45}$$

$$\tau_2(G, I) = \sum_D \tau'_2(G, I, D) \tag{20.46}$$

$$\sum_L \sum_S \psi_J(J,L,S) \sum_G \psi_L(L,G) \sum_H \psi_H(H,G,J) \sum_I \psi_S(S,I)\psi_I(I) \sum_D \psi_G(G,I,D) \underbrace{\sum_C \psi_C(C)\psi_D(D,C)}_{\tau_1(D)}$$

$$\sum_L \sum_S \psi_J(J,L,S) \sum_G \psi_L(L,G) \sum_H \psi_H(H,G,J) \sum_I \psi_S(S,I)\psi_I(I) \underbrace{\sum_D \psi_G(G,I,D)\tau_1(D)}_{\tau_2(G,I)}$$

$$\sum_L \sum_S \psi_J(J,L,S) \sum_G \psi_L(L,G) \sum_H \psi_H(H,G,J) \underbrace{\sum_I \psi_S(S,I)\psi_I(I)\tau_2(G,I)}_{\tau_3(G,S)}$$

$$\sum_L \sum_S \psi_J(J,L,S) \sum_G \psi_L(L,G) \underbrace{\sum_H \psi_H(H,G,J)}_{\tau_4(G,J)} \tau_3(G,S)$$

$$\sum_L \sum_S \psi_J(J,L,S) \underbrace{\sum_G \psi_L(L,G)\tau_4(G,J)\tau_3(G,S)}_{\tau_5(J,L,S)}$$

$$\sum_L \underbrace{\sum_S \psi_J(J,L,S)\tau_5(J,L,S)}_{\tau_6(J,L)}$$

$$\underbrace{\sum_L \tau_6(J,L)}_{\tau_7(J)}$$

Table 20.1 Eliminating variables from Figure 20.3 in the order C, D, I, H, G, S, L to compute $P(J)$.

Next we multiply together all the terms in the scope of the $\sum_I$ operator and then marginalize out to create

$$\tau'_3(G,I,S) = \psi_S(S,I)\psi_I(I)\tau_2(G,I) \quad (20.47)$$

$$\tau_3(G,S) = \sum_I \tau'_3(G,I,S) \quad (20.48)$$

And so on.

The above technique can be used to compute any marginal of interest, such as $p(J)$ or $p(J,H)$. To compute a conditional, we can take a ratio of two marginals, where the visible variables have been clamped to their known values (and hence don't need to be summed over). For example,

$$p(J=j|I=1,H=0) = \frac{p(J=j,I=1,H=0)}{\sum_{j'} p(J=j',I=1,H=0)} \quad (20.49)$$

In general, we can write

$$p(\mathbf{x}_q|\mathbf{x}_v) = \frac{p(\mathbf{x}_q,\mathbf{x}_v)}{p(\mathbf{x}_v)} = \frac{\sum_{\mathbf{x}_h} p(\mathbf{x}_h,\mathbf{x}_q,\mathbf{x}_v)}{\sum_{\mathbf{x}_h}\sum_{\mathbf{x}'_q} p(\mathbf{x}_h,\mathbf{x}'_q,\mathbf{x}_v)} \quad (20.50)$$

The normalization constant in the denominator, $p(\mathbf{x}_v)$, is called the **probability of the evidence**.

See `variableElimination` for a simple Matlab implementation of this algorithm, which works for arbitrary graphs, and arbitrary discrete factors. But before you go too crazy, please read Section 20.3.2, which points out that VE can be exponentially slow in the worst case.

20.3.1 The generalized distributive law *

Abstractly, VE can be thought of as computing the following expression:

$$p(\mathbf{x}_q|\mathbf{x}_v) \propto \sum_{\mathbf{x}} \prod_c \psi_c(\mathbf{x}_c) \tag{20.51}$$

It is understood that the visible variables $\mathbf{x}_v$ are clamped, and not summed over. VE uses **non-serial dynamic programming** (Bertele and Brioschi 1972), caching intermediate results to avoid redundant computation.

However, there are other tasks we might like to solve for any given graphical model. For example, we might want the MAP estimate:

$$\mathbf{x}^* = \operatorname*{argmax}_{\mathbf{x}} \prod_c \psi_c(\mathbf{x}_c) \tag{20.52}$$

Fortunately, essentially the same algorithm can also be used to solve this task: we just replace sum with max. (We also need a **traceback** step, which actually recovers the argmax, as opposed to just the value of $\max$; these details are explained in Section 17.4.4.)

In general, VE can be applied to any **commutative semi-ring**. This is a set K, together with two binary operations called "+" and "$\times$", which satisfy the following three axioms:

1. The operation "+" is associative and commutative, and there is an additive identity element called "0" such that $k + 0 = k$ for all $k \in K$.
2. The operation "$\times$" is associative and commutative, and there is a multiplicative identity element called "1" such that $k \times 1 = k$ for all $k \in K$.
3. The **distributive law** holds, i.e.,

$$(a \times b) + (a \times c) = a \times (b + c) \tag{20.53}$$

for all triples (a, b, c) from K.

This framework covers an extremely wide range of important applications, including constraint satisfaction problems (Bistarelli et al. 1997; Dechter 2003), the fast Fourier transform (Aji and McEliece 2000), etc. See Table 20.2 for some examples.

20.3.2 Computational complexity of VE

The running time of VE is clearly exponential in the size of the largest factor, since we have sum over all of the corresponding variables. Some of the factors come from the original model (and are thus unavoidable), but new factors are created in the process of summing out. For example,

Domain	+	×	Name
$[0, \infty)$	$(+, 0)$	$(\times, 1)$	sum-product
$[0, \infty)$	$(\max, 0)$	$(\times, 1)$	max-product
$(-\infty, \infty]$	$(\min, \infty)$	$(+, 0)$	min-sum
$\{T, F\}$	$(\vee, F)$	$(\wedge, T)$	Boolean satisfiability

Table 20.2 Some commutative semirings.

$$\sum_D \sum_C \psi_D(D,C) \sum_H \sum_L \sum_S \psi_J(J,L,S) \sum_I \psi_I(I)\psi_S(S,I) \underbrace{\sum_G \psi_G(G,I,D)\psi_L(L,)\psi_H(H,G,J)}_{\tau_1(I,D,L,J,H)}$$

$$\sum_D \sum_C \psi_D(D,C) \sum_H \sum_L \sum_S \psi_J(J,L,S) \underbrace{\sum_I \psi_I(I)\psi_S(S,I)\tau_1(I,D,L,J,H)}_{\tau_2(D,L,S,J,H)}$$

$$\sum_D \sum_C \psi_D(D,C) \sum_H \sum_L \underbrace{\sum_S \psi_J(J,L,S)\tau_2(D,L,S,J,H)}_{\tau_3(D,L,J,H)}$$

$$\sum_D \sum_C \psi_D(D,C) \sum_H \underbrace{\sum_L \tau_3(D,L,J,H)}_{\tau_4(D,J,H)}$$

$$\sum_D \sum_C \psi_D(D,C) \underbrace{\sum_H \tau_4(D,J,H)}_{\tau_5(D,J)}$$

$$\sum_D \underbrace{\sum_C \psi_D(D,C)\tau_5(D,J)}_{\tau_6(D,J)}$$

$$\underbrace{\sum_D \tau_6(D,J)}_{\tau_7(J)}$$

Table 20.3 Eliminating variables from Figure 20.3 in the order G, I, S, L, H, C, D.

in Equation 20.47, we created a factor involving G, I and S; but these nodes were not originally present together in any factor.

The order in which we perform the summation is known as the **elimination order**. This can have a large impact on the size of the intermediate factors that are created. For example, consider the ordering in Table 20.1: the largest created factor (beyond the original ones in the model) has size 3, corresponding to $\tau_5(J, L, S)$. Now consider the ordering in Table 20.3: now the largest factors are $\tau_1(I, D, L, J, H)$ and $\tau_2(D, L, S, J, H)$, which are much bigger.

We can determine the size of the largest factor graphically, without worrying about the actual numerical values of the factors. When we eliminate a variable X_t, we connect it to all variables

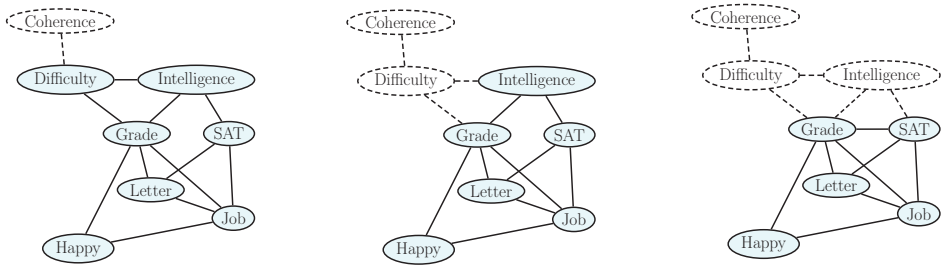


Figure 20.4 Example of the elimination process, in the order C, D, I, etc. When we eliminate I (figure c), we add a fill-in edge between G and S, since they are not connected. Based on Figure 9.10 of (Koller and Friedman 2009).

that share a factor with X_t (to reflect the new temporary factor τ'_t). The edges created by this process are called **fill-in edges**. For example, Figure 20.4 shows the fill-in edges introduced when we eliminate in the order $C, D, I, \ldots$. The first two steps do not introduce any fill-ins, but when we eliminate I, we connect G and S, since they co-occur in Equation 20.48.

Let $G(\prec)$ be the (undirected) graph induced by applying variable elimination to G using elimination ordering $\prec$. The temporary factors generated by VE correspond to maximal cliques in the graph $G(\prec)$. For example, with ordering (C, D, I, H, G, S, L), the maximal cliques are as follows:

$$\{C, D\}, \{D, I, G\}, \{G, L, S, J\}, \{G, J, H\}, \{G, I, S\} \tag{20.54}$$

It is clear that the time complexity of VE is

$$\prod_{c \in \mathcal{C}(G(\prec))} K^{|c|} \tag{20.55}$$

where $\mathcal{C}$ are the cliques that are created, $|c|$ is the size of the clique c, and we assume for notational simplicity that all the variables have K states each.

Let us define the **induced width** of a graph given elimination ordering $\prec$, denoted $w(\prec)$, as the size of the largest factor (i.e., the largest clique in the induced graph) minus 1. Then it is easy to see that the complexity of VE with ordering $\prec$ is $O(K^{w(\prec)+1})$.

Obviously we would like to minimize the running time, and hence the induced width. Let us define the **treewidth** of a graph as the minimal induced width.

$$w \triangleq \min_{\prec} \max_{c \in G(\prec)} |c| - 1 \tag{20.56}$$

Then clearly the best possible running time for VE is $O(DK^{w+1})$. Unfortunately, one can show that for arbitrary graphs, finding an elimination ordering $\prec$ that minimizes $w(\prec)$ is NP-hard (Arnborg et al. 1987). In practice greedy search techniques are used to find reasonable orderings (Kjaerulff 1990), although people have tried other heuristic methods for discrete optimization,