Chapter 19

Approximate inference

TODO: somewhere in this chapter, point out that variational inference implicitly defines a recurrent net, stochastic approximate inference implicitly defines a stochastic recurrent net

Misplaced TODO: discussion of the different directions of the KL divergence, and the effects on ignoring / preserving modes

Many probabilistic models are difficult to train because it is difficult to perform inference in them. In the context of deep learning, we usually have a set of visible variables $\boldsymbol{v}$ and a set of latent variables $\boldsymbol{h}$. The challenge of inference usually refers to the difficult problem of computing $p(\boldsymbol{h} \mid \boldsymbol{v})$ or taking expectations with respect to it. Such operations are often necessary for tasks like maximum likelihood learning.

Many simple graphical models with only one hidden layer, such as restricted Boltzmann machines and probabilistic PCA are defined in a way that makes inference operations like computing $p(\boldsymbol{h} \mid \boldsymbol{v})$ or taking expectations with respect to it simple. Unfortunately, most graphical models with multiple layers of hidden variables, such as deep belief networks and deep Boltzmann machines have intractable posterior distributions. Exact inference requires an exponential amount of time in these models. Even some models with only a single layer, such as sparse coding, have this problem.

Intractable inference problems usually arise from interactions between latent variables in a structured graphical model. See Fig. 19.1 for some examples. These interactions may be due to direct interactions in undirected models or "explaining away" interactions between mutual ancestors of the same visible unit in directed models.

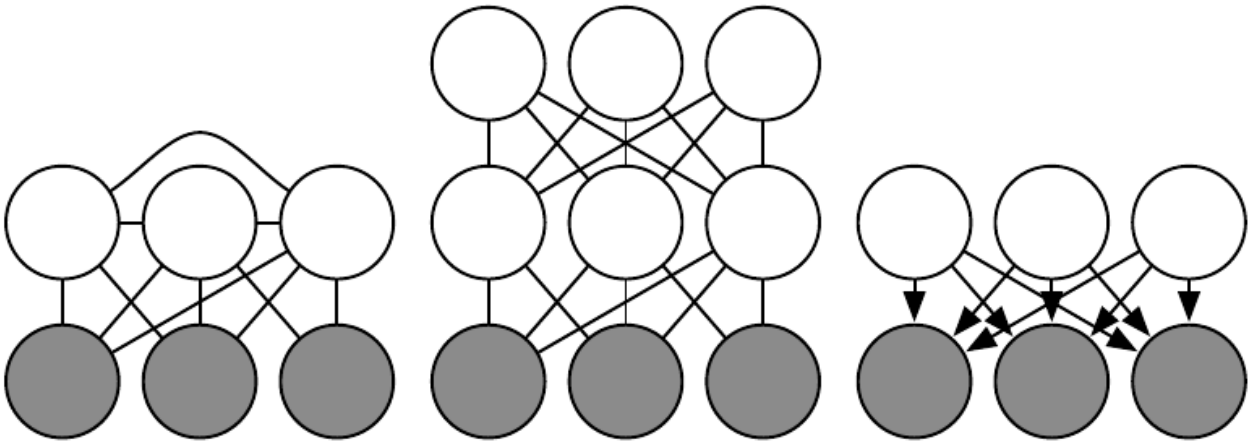

Figure 19.1: Intractable inference problems are usually the result of interactions between latent variables in a structured graphical model. These can be due to direct edges, or due to paths that are activated when the child of a V-structure is observed. Left) A semi-restricted Boltzmann machine with connections between hidden units. These direct connections between latent variables make the posterior distribution complicated. Center) A deep Boltzmann machine, organized into layers of variables without intra-layer connections, still has an intractable posterior distribution due to the connections between layers. Right) This directed model has interactions between latent variables when the visible variables are observed, because ever two latent variables are co-parents. Note that it is still possible to have these graph structures yet have tractable inference. For example, probabilistic PCA has the graph structure shown in the right, yet simple inference due to special properties of the specific conditional distributions it uses (linear-Gaussian conditionals with mutually orthogonal basis vectors). MISPLACED TODO– make sure probabilistic PCA is at least defined somewhere in the book

19.1 Inference as Optimization

Many approaches to confronting the problem of difficult inference make use of the observation that exact inference can be described as an optimization problem.

Specifically, assume we have a probabilistic model consisting of observed variables $\boldsymbol{v}$ and latent variables $\boldsymbol{h}$. We would like to compute the log probability of the observed data, $\log p(\boldsymbol{v}; \boldsymbol{\theta})$. Sometimes it is too difficult to compute $\log p(\boldsymbol{v}; \boldsymbol{\theta})$ if it is costly to marginalize out $\boldsymbol{h}$. Instead, we can compute a lower bound on it. This bound is called the *evidence lower bound* (ELBO). Other names for this lower bound include the negative *variational free energy* and the negative *Helmholtz free energy*. Specifically, this lower bound is defined as TODO– figure out why I was making q be bm in some but not all places

$$\mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, \boldsymbol{q}) = \log p(\boldsymbol{v}; \boldsymbol{\theta}) - D_{\mathrm{KL}}(q(\boldsymbol{h}) \| p(\boldsymbol{h} \mid \boldsymbol{v}; \boldsymbol{\theta}))$$

where q is an arbitrary probability distribution over $\boldsymbol{h}$.

TODO: the below equations framebust It is straightforward to see that this is a lower bound on $\log p(\boldsymbol{v})$:

$$\begin{aligned}
\ln p(\boldsymbol{v}) &= \ln p(\boldsymbol{v}) + \sum_{\boldsymbol{h}} q(\boldsymbol{h} \mid \boldsymbol{v}) \ln \left(\frac{p(\boldsymbol{v}, \boldsymbol{h})}{q(\boldsymbol{h} \mid \boldsymbol{v})} \right) - \sum_{\boldsymbol{h}} q(\boldsymbol{h} \mid \boldsymbol{v}) \ln \left(\frac{p(\boldsymbol{v}, \boldsymbol{h})}{q(\boldsymbol{h} \mid \boldsymbol{v})} \right) \\
&= \sum_{\boldsymbol{h}} q(\boldsymbol{h} \mid \boldsymbol{v}) \ln \left(\frac{p(\boldsymbol{v}, \boldsymbol{h})}{q(\boldsymbol{h} \mid \boldsymbol{v})} \right) - \sum_{\boldsymbol{h}} q(\boldsymbol{h} \mid \boldsymbol{v}) \ln \left(\frac{p(\boldsymbol{v}, \boldsymbol{h})}{q(\boldsymbol{h} \mid \boldsymbol{v})} \right) + \sum_{\boldsymbol{h}} q(\boldsymbol{h} \mid \boldsymbol{v}) \log p(\boldsymbol{v}) \\
&= \sum_{\boldsymbol{h}} q(\boldsymbol{h} \mid \boldsymbol{v}) \ln \left(\frac{p(\boldsymbol{v}, \boldsymbol{h})}{q(\boldsymbol{h} \mid \boldsymbol{v})} \right) - \sum_{\boldsymbol{h}} q(\boldsymbol{h} \mid \boldsymbol{v}) \left[\ln \left(\frac{p(\boldsymbol{v}, \boldsymbol{h})}{q(\boldsymbol{h} \mid \boldsymbol{v})} \right) - \ln p(\boldsymbol{v}) \right] \\
&= \sum_{\boldsymbol{h}} q(\boldsymbol{h} \mid \boldsymbol{v}) \ln \left(\frac{p(\boldsymbol{v}, \boldsymbol{h})}{q(\boldsymbol{h} \mid \boldsymbol{v})} \right) - \sum_{\boldsymbol{h}} q(\boldsymbol{h} \mid \boldsymbol{v}) \ln \left(\frac{p(\boldsymbol{v}, \boldsymbol{h})}{p(\boldsymbol{v}) q(\boldsymbol{h} \mid \boldsymbol{v})} \right) \\
&= \sum_{\boldsymbol{h}} q(\boldsymbol{h} \mid \boldsymbol{v}) \ln \left(\frac{p(\boldsymbol{v}, \boldsymbol{h})}{q(\boldsymbol{h} \mid \boldsymbol{v})} \right) - \sum_{\boldsymbol{h}} q(\boldsymbol{h} \mid \boldsymbol{v}) \ln \left(\frac{p(\boldsymbol{h} \mid \boldsymbol{v})}{q(\boldsymbol{h} \mid \boldsymbol{v})} \right) \\
&= \mathcal{L}(q) + \mathrm{KL}(q \| p)
\end{aligned}$$

Because the difference $\log p(\boldsymbol{v})$ and $\mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, \boldsymbol{q})$ is given by the KL-divergence and because the KL-divergence is always non-negative, we can see that $\mathcal{L}$ always has at most the same value as the desired log probability, and is equal to it if and only if q is the same distribution as $p(\boldsymbol{h} \mid \boldsymbol{v})$.

Surprisingly, $\mathcal{L}$ can be considerably easier to compute for some distributions q. Simple algebra shows that we can rearrange $\mathcal{L}$ into a much more convenient form:

$$\mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, \boldsymbol{q}) = \log p(\boldsymbol{v}; \boldsymbol{\theta}) - D^{\mathrm{KL}}(q(\boldsymbol{h}) \| p(\boldsymbol{h} \mid \boldsymbol{v}; \boldsymbol{\theta}))$$

$$= \log p(\boldsymbol{v}; \boldsymbol{\theta}) - \mathbb{E}_{\boldsymbol{h} \sim q} \frac{\log q(\boldsymbol{h})}{\log p(\boldsymbol{h} \mid \boldsymbol{v})}$$

$$= \log p(\boldsymbol{v}; \boldsymbol{\theta}) - \mathbb{E}_{\boldsymbol{h} \sim q} \frac{\log q(\boldsymbol{h})}{\log \frac{p(\boldsymbol{h}, \boldsymbol{v})}{p(\boldsymbol{v})}}$$

$$= \log p(\boldsymbol{v}; \boldsymbol{\theta}) - \mathbb{E}_{\boldsymbol{h} \sim q} \left[\log q(\boldsymbol{h}) - \log p(\boldsymbol{h}, \boldsymbol{v}) + \log p(\boldsymbol{v}) \right]$$

$$= -\mathbb{E}_{\boldsymbol{h} \sim q} \left[\log q(\boldsymbol{h}) - \log p(\boldsymbol{h}, \boldsymbol{v}) \right].$$

This yields the more canonical definition of the evidence lower bound,

$$\mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, \boldsymbol{q}) = \mathbb{E}_{\boldsymbol{h} \sim q} \left[\log p(\boldsymbol{h}, \boldsymbol{v}) \right] + H(q). \tag{19.1}$$

The first term of $\mathcal{L}$ is known as the *energy term*. The second term is known as the *entropy term*. For an appropriate choice of q, both terms can be easy to compute. The only question is how close to $p(\boldsymbol{h} \mid \boldsymbol{v})$ the distribution q will be. This determines how good of an approximation $\mathcal{L}$ will be for $\log p(\boldsymbol{v})$.

We can thus think of inference as the procedure for finding the q that maximizes $\mathcal{L}$. Exact inference maximizes $\mathcal{L}$ perfectly. Throughout this chapter, we will show how many forms of approximate inference are possible. No matter what choice of q we use, $\mathcal{L}$ will give us a lower bound on the likelihood. We can get tighter or looser bounds that are cheaper or more expensive to compute depending on how we choose to approach this optimization problem. We can obtain a poorly matched q but reduce the computational cost by using an imperfect optimization procedure, or by using a perfect optimization procedure over a restricted family of q distributions.

19.2 Expectation Maximization

Expectation maximization (EM) is a popular training algorithm for models with latent variables. It consists of alternating between two steps until convergence:

- The *E-step* (Expectation step): Set $q(\boldsymbol{h}^{(i)}) = p(\boldsymbol{h}^{(i)} \mid \boldsymbol{v}^(i); \boldsymbol{\theta})$ for all indices i of the training examples $\boldsymbol{v}^{(i)}$ we want to train on (both batch and minibatch variants are valid). By this we mean q is defined in terms of the *current* value of $\boldsymbol{\theta}$; if we vary θ then $p(\boldsymbol{h} \mid \boldsymbol{v}; \boldsymbol{\theta})$ will change but $q(\boldsymbol{h})$ will not.
- The M-step (Maximization step): Completely or partially maximize $\sum_i \mathcal{L}(\boldsymbol{v}^(i), \boldsymbol{\theta}, q)$ with respect to $\boldsymbol{\theta}$ using your optimization algorithm of choice.

This can be viewed as a coordinate ascent algorithm to maximize $\mathcal{L}$. On one step, we maximize $\mathcal{L}$ with respect to q, and on the other, we maximize $\mathcal{L}$ with respect to $\boldsymbol{\theta}$.

Stochastic gradient ascent on latent variable models can be seen as a special case of the EM algorithm where the M step consists of taking a single gradient step. Other variants of the EM algorithm can make much larger steps. For some model families, the M step can even be performed analytically, jumping all the way to the optimal solution given the current q.

Even though the E-step involves exact inference, we can think of the EM algorithm as using approximate inference in some sense. Specifically, the M-step assumes that the same value of q can be used for all values of $\boldsymbol{\theta}$. This will introduce a gap between $\mathcal{L}$ and the true $\log p(v)$ as the M-step moves further and further. Fortunately, the E-step reduces the gap to zero again as we enter the loop for the next time.

The EM algorithm is a workhorse of classical machine learning, and it can be considered to be used in deep learning in the sense that stochastic gradient ascent can be seen as EM with a very simple and small M step. However, because $\mathcal{L}$ can not be analytically maximized for many interesting deep models, the more general EM framework as a whole is typically not explored in the deep learning research community.

TODO–cite the emview paper

19.3 MAP Inference: Sparse Coding as a Probabilistic Model

TODO synch up with other sections on sparse coding

Many versions of sparse coding can be cast as probabilistic models. For example, suppose we encode visible data $\boldsymbol{v} \in \mathbb{R}^n$ with latent variables $\boldsymbol{h} \in \mathbb{R}^m$. We can use a prior to encourage our latent code variables to be sparse:

$$p(h) = TODO.$$

We can define the visible units to be Gaussian with an affine transformation from the code to the mean of the Gaussian:

$$\boldsymbol{v} \sim \mathcal{N}(\boldsymbol{v} \mid \boldsymbol{\mu} + \boldsymbol{W}\boldsymbol{h}, \boldsymbol{\beta}^{-1})$$

where $\boldsymbol{\beta}$ is a diagonal precision matrix to maintain tractability.

Computing $p(\boldsymbol{h} \mid \boldsymbol{v})$ is difficult. TODO explain why

One operation that we can do is perform *maximum a posteriori* (MAP) inference, which means solving the following optimization problem:

$$\boldsymbol{h}^* = \arg\max p(\boldsymbol{h} \mid \boldsymbol{v}).$$

This yields the familiar optimization problem

TODO synch with other sparse coding sections, make sure the other sections talk about using gradient descent, feature sign, ISTA, etc.

This shows that the popular feature extraction strategy for sparse coding can be justified as having a probabilistic interpretation–it may be MAP inference in this probabilistic model (there are other probabilistic models that yield the same optimization problem, so we cannot positively identify this specific model from the feature extraction process).

Excitingly, MAP inference of $\boldsymbol{h}$ given $\boldsymbol{v}$ also has an interpretation in terms of maximizing the evidence lower bound. Specifically, MAP inference maximizes $\mathcal{L}$ with respect to q under the constraint that q take the form form of a Dirac distribution. During learning of sparse coding, we alternate between using convex optimization to extract the codes, and using convex optimization to update $\boldsymbol{W}$ to achieve the optimal reconstruction given the codes. This turns out to be equivalent to maximizing $\mathcal{L}$ with respect to $\boldsymbol{\theta}$ for the q that was obtained from MAP inference. The learning algorithm can be thought of as EM restricted to using a Dirac posterior. In other words, rather than performing learning exactly using standard inference, we learn to maximize a bound on the true likelihood, using exact MAP inference.

19.4 Sequence Modeling with Graphical Models

This section regards probabilistic approaches to sequential data modeling which have often been viewed as in competition with RNNs, although RNNs can be seen as a particular form of *dynamic Bayesian networks*[1], as directed graphical models with deterministic latent variables[2].

19.4.1 Efficient Inference by Dynamic Programming

Many temporal modeling approaches can be cast in the following framework, which also includes hybrids of neural networks with HMMs and conditional random fields (CRFs), first introduced in Bottou *et al.* (1997); LeCun *et al.* (1998c) and later developed and applied with great success in Graves *et al.* (2006); Graves (2012) with the Connectionist Temporal Classification (CTC) approach, as well as in Do and Artières (2010) and other more recent work (Farabet *et al.*, 2013b;

[1] Dynamic Bayesian networks or dynamic probabilistic networks are directed graphical models for sequential data, with shared parameters across time (Dean and Kanazawa, 1989; Kanazawa *et al.*, 1995)

[2] Latent variables are random variables that are not directly observed, although they can depend on some that are, and here the dependency is so strong that the latent variables are functions of observed variables

Deng *et al.*, 2014). These ideas have been rediscovered in a simplified form (limiting the input-output relationship to a linear one) as CRFs (Lafferty *et al.*, 2001), i.e., undirected graphical models whose parameters are linear functions of input variables. In section 19.5 we consider in more detail the neural network hybrids and the "graph transformer" generalizations of the ideas presented below.

All these approaches (with or without neural nets in the middle) concern the case where we have an input sequence (discrete or continuous-valued) $\{\boldsymbol{x}_t\}$ and a symbolic output sequence $\{y_t\}$ (typically of the same length, although shorter output sequences can be handled by introducing "empty string" values in the output). Generalizations to non-sequential output structure have been introduced more recently (e.g. to condition the Markov Random Fields sometimes used to model structural dependencies in images (Stewart *et al.*, 2007)), at the loss of exact inference (the dynamic programming methods described below).

Optionally, one also considers a latent variable sequence $\{s_t\}$ that is also discrete and inference needs to be done over $\{s_t\}$, either via marginalization (summing over all possible values of the state sequence) or maximization, i.e., picking exactly or approximately the so-called MAP sequence, the one with the largest probability, given the input. If the state variables s_t and the target variables y_t have a 1-D Markov structure to their dependency, then computing likelihood, partition function and MAP values can all be done efficiently by exploiting dynamic programming to factorize the computation. On the other hand, if the state or output sequence dependencies are captured by an RNN, then there is no finite-order Markov property and no efficient and exact inference is generally possible. However, many reasonable approximations have been used in the past, such as with variants of the beam search algorithm (Lowerre, 1976). The idea of beam search is that one maintains a set of promising candidate paths that end at some time step t. For each additional time step, one considers extensions to $t+1$ of each of these paths and then prunes those with the worse overall cumulative score (up to $t+1$). The beam size is the number of candidates that are kept. See Section 19.5.1 for more details on beam search.

The application of the principle of dynamic programming in these setups is the same as what is used in the Forward-Backward algorithm (detailed more around Eq. 19.5), for graphical models and HMMs (detailed more in Section 19.4.2) and the Viterbi algorithm detailed below (Eq. 19.7). For both of these algorithms, we are trying to sum (Forward-Backward algorithm) or maximize (Viterbi algorithm) over paths the probability or score associated with each path.

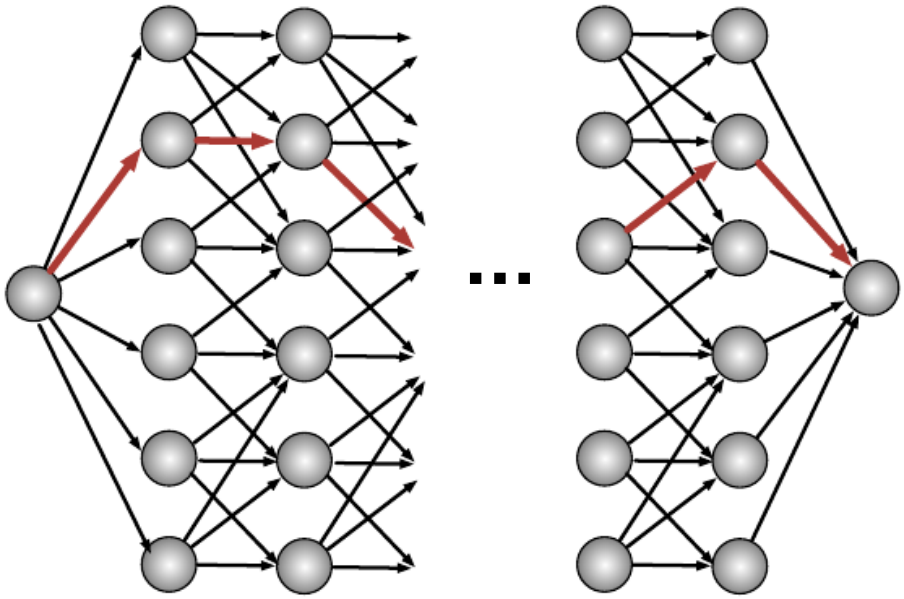

Figure 19.2: Example of a temporally structured output graph, as can be found in CRFs, HMMs and neural net hybrids. Each node corresponds to a particular **value** of an output random variable at a particular point in the output sequence (contrast with a graphical model representation, where each node corresponds to a random variable). A path from the source node to the sink node (e.g. red bold arrows) corresponds to an interpretation of the input as a sequence of output labels. The dynamic programming recursions that are used for computing likelihood (or conditional likelihood) or performing MAP inference (finding the best path) involve sums or maximizations over sub-paths ending at one of the particular interior nodes.

Let $\mathcal{G}$ be a directed acyclic graph whose paths correspond to the sequences that can be selected (for MAP) or summed over (marginalized for computing a likelihood), as illustrated in Fig. 19.2. In the above example, let z_t represent the choice variable (e.g., s_t and y_t in the above example), and each arc with score a corresponds to a particular value of z_t in its Markov context. In the language of undirected graphical models, if a is the score associated with an arc from the node for $z_{t-1} = j$ to the one for $z_t = i$, then a is minus the energy of a term of the energy function associated with the event $1_{z_{t-1}=j,zt=i}$ and the associated information from the input $\boldsymbol{x}$ (e.g. some value of x_t).

Hidden Markov models are based on the notion of Markov chain, which is covered in much more detail in Section 14.1. A Markov chain is a sequence of random variables $\mathrm{z}_1, \ldots \mathrm{z}_T$. For our purposes the main property of a Markov chain chain of *order 1* is that the current value of z_t contains enough information about the previous values $z_1, \ldots z_{t-1}$ in order to predict the distribution of the next random variable, z_{t+1}. In our context, we can make the z's conditioned on

some $\boldsymbol{x}$. The order 1 Markov property then means that

$$P(z_t \mid z_{t-1}, z_{t-2}, \ldots, z_1, \boldsymbol{x}) = P(z_t \mid z_{t-1}, \boldsymbol{x}),$$

where $\boldsymbol{x}$ is the conditioning information (the input sequence). When we consider a path in that space, i.e. a sequence of values, we draw a graph with a node for each discrete value of z_t, and if it is possible to transition from $z_{t-1} = j$ to $z_t = i$ we draw an arc between these two nodes. Hence, the total number of nodes in the graph would be equal to Tn (the length of the sequence times the number of values of z_t). The number of arcs in the graph could be up to Tn^2. This extreme case occurs if every value of z_t can follow every value of z_{t-1}. In practice the connectivity is often much smaller because not all transitions are typically feasible. A score a is computed for each arc (which may include some component that only depends on the source or only on the destination node), as a function of the conditioning information $\boldsymbol{x}$. The inference or marginalization problems involve performing the following computations.

For the **marginalization** task, we want to compute the sum over all complete paths (e.g. from source to sink) of the product along the path of the exponentiated scores associated with the arcs on that path:

$$m(\mathcal{G}) = \sum_{\text{path} \in \mathcal{G}} \prod_{a \in \text{path}} e^a \tag{19.2}$$

where the product is over all the arcs on a path (with score a), and the sum is over all the paths associated with complete sequences (from beginning to end of a sequence). $m(\mathcal{G})$ may correspond to a likelihood, numerator or denominator of a probability. For example,

$$P(\{z_t\} \in \mathbb{Y} \mid \boldsymbol{x}) = \frac{m(\mathcal{G}_{\mathbb{Y}})}{m(\mathcal{G})} \tag{19.3}$$

where $\mathcal{G}_{\mathbb{Y}}$ is the subgraph of $\mathcal{G}$ which is restricted to sequences that are compatible with some target answer $\mathbb{Y}$.

For the **inference** task, we want to compute

$$\begin{aligned} \pi(\mathcal{G}) &= \arg\max_{\text{path} \in \mathcal{G}} \prod_{a \in \text{path}} e^a = \arg\max_{\text{path} \in \mathcal{G}} \sum_{a \in \text{path}} a \\ v(\mathcal{G}) &= \max_{\text{path} \in \mathcal{G}} \sum_{a \in \text{path}} a \end{aligned}$$

where $\pi(\mathcal{G})$ is the most probable path and $v(\mathcal{G})$ is its log-score or value, and again the set of paths considered includes all of those starting at the beginning and ending at the end the sequence.

The principle of dynamic programming is to recursively compute intermediate quantities that can be reused efficiently so as to avoid actually going through an exponential number of computations, e.g., though the exponential number of paths to consider in the above sums or maxima. Note how it is already at play in the underlying efficiency of back-propagation (or back-propagation through time), where gradients w.r.t. intermediate layers or time steps or nodes in a flow graph can be computed based on previously computed gradients (for later layers, time steps or nodes). Here it can be achieved by considering to restrictions of the graph to those paths that end at a node n, which we denote $\mathcal{G}^n$. $\mathcal{G}^n_Y$ indicates the additional restriction to subsequences that are compatible with the target sequence Y, i.e., with the beginning of the sequence Y.

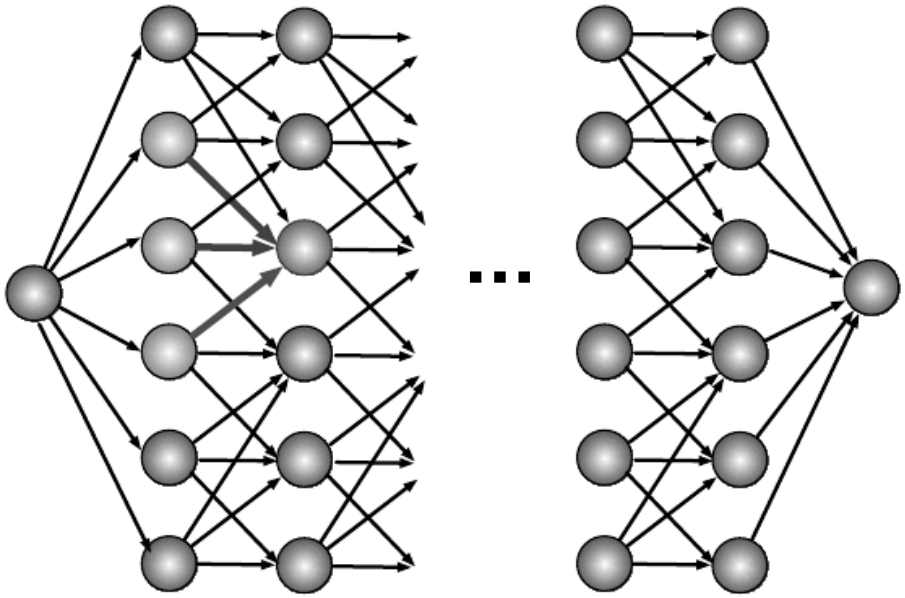

Figure 19.3: Illustration of the recursive computation taking place for inference or marginalization by dynamic programming. See Fig. 19.2. These recursions involve sums or maximizations over sub-paths ending at one of the particular interior nodes (red in the figure), each time only requiring to look up previously computed values at the predecessor nodes (green).

We can thus perform marginalization efficiently as follows, using a generalization of the Forward-Backward algorithm for HMMs. The process is illustrated in Fig. 19.3. This is a generalization of the so-called Forward-Backward algorithm for HMMs.

$$m(\mathcal{G}) = \sum_{n \in \text{final}(\mathcal{G})} m(\mathcal{G}^n) \tag{19.4}$$

where final($\mathcal{G}$) is the set of final nodes in the graph $\mathcal{G}$. We can recursively compute

the node-restricted sum via the identity

$$m(G^n) = \sum_{n' \in \text{pred}(n)} m(\mathcal{G}^{n'}) e^{a_{n',n}} \tag{19.5}$$

where $\text{pred}(n)$ is the set of predecessors of node n in the graph and $a_{m,n}$ is the log-score associated with the arc from m to n. It is easy to see that expanding the above recursion recovers the result of Eq. 19.2.

Similarly, we can perform efficient MAP inference (also known as Viterbi decoding) as follows.

$$v(\mathcal{G}) = \max_{n \in \text{final}(\mathcal{G})} v(G^n) \tag{19.6}$$

and

$$v(\mathcal{G}^n) = \max_{m \in \text{pred}(n)} v(\mathcal{G}^m) + a_{m,n}\,. \tag{19.7}$$

To obtain the corresponding path, it is enough to keep track of the argmax associated with each of the above maximizations and trace back $\pi(\mathcal{G})$ starting from the nodes in $\text{final}(\mathcal{G})$. For example, the last element of $\pi(\mathcal{G})$ is

$$n^* \leftarrow \arg\max_{n \in \text{final}(\mathcal{G})} v(\mathcal{G}^n)$$

and (recursively) the argmax node before n^* along the selected path is a new n^*,

$$n^* \leftarrow \arg\max_{m \in \text{pred}(n^*)} v(\mathcal{G}^m) + a_{m,n^*}\,,$$

etc. Keeping track of these n^* along the way gives the selected path. Proving that these recursive computations yield the desired results is straightforward and left as an exercise.

19.4.2 HMMs

Hidden Markov Models (HMMs) are probabilistic models of sequences that were introduced in the 60's (Baum and Petrie, 1966) along with the E-M algorithm (Section 19.2). They are very commonly used to model sequential structure, in particular having been since the mid 80's and until recently the technological core of speech recognition systems (Rabiner and Juang, 1986; Rabiner, 1989). Just like RNNs, HMMs are dynamic Bayes nets (Koller and Friedman, 2009), i.e., the same parameters and graphical model structure are used for every time step. Compared to RNNs, what is particular to HMMs is that the latent variable associated with each time step (called the *state*) is discrete, with a separate set of

parameters associated with each state value. We consider here the most common form of HMM, in which the Markov chain is of order 1, i.e., the state s_t at time t, given the previous states, only depends on the previous state s_{t-1}:

$$P(s_t \mid s_{t-1}, s_{t-2}, \ldots, s_1) = P(s_t \mid s_{t-1}),$$

which we call the *transition or state-to-state distribution*. Generalizing to higher-order Markov chains is straightforward: for example, order-2 Markov chains can be mapped to order-1 Markov chains by considering as order-1 "states" all the pairs $(s_t = i, s_{t-1} = j)$.

Given the state value, a generative probabilistic model of the visible variable $\boldsymbol{x}_t$ is defined, that specifies how each observation $\boldsymbol{x}_t$ in a sequence $(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T)$ can be generated, via a model $P(\boldsymbol{x}_t \mid s_t)$. Two kinds of parameters are distinguished: those that define the transition distribution, which can be given by a matrix

$$A_{ij} = P(s_t = i \mid s_{t-1} = j),$$

and those that define the output model $P(\boldsymbol{x}_t \mid s_t)$. For example, if the data are discrete and $\boldsymbol{x}_t$ is a symbol x_t, then another matrix can be used to define the output (or emission) model:

$$B_{ki} = P(x_t = k \mid s_t = i).$$

Another common parametrization for $P(\boldsymbol{x}_t \mid s_t = i)$, in the case of continuous vector-valued $\boldsymbol{x}_t$, is the Gaussian mixture model, where we have a different mixture (with its own means, covariances and component probabilities) for each state $s_t = i$. Alternatively, the means and covariances (or just variances) can be shared across states, and only the component probabilities are state-specific.

The overall likelihood of an observed sequence is thus

$$P(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T) = \sum_{s_1, s_2, \ldots, s_T} \prod_t P(\boldsymbol{x}_t \mid s_t) P(s_t \mid s_{t-1}). \tag{19.8}$$

In the language established earlier in Section 19.4.1, we have a graph $\mathcal{G}$ with one node n per time step t and state value i, i.e., for $s_t = i$, and one arc between each node n (for $\mathbf{1}_{s_t=i}$) and its predecessors m for $\mathbf{1}_{s_{t-1}=j}$ (when the transition probability is non-zero, i.e., $P(s_t = i \mid s_{t-1} = j) \neq 0$). Following Eq. 19.8, the log-score $a_{m,n}$ for the transition between m and n would then be

$$a_{m,n} = \log P(x_t \mid s_t = i) + \log P(s_t = i \mid s_{t-1} = j).$$

As explained in Section 19.4.1, this view gives us a dynamic programming algorithm for computing the likelihood (or the conditional likelihood given some

constraints on the set of allowed paths), called the forward-backward or sum-product algorithm, in time $O(kNT)$ where T is the sequence length, N is the number of states and k the average in-degree of each node.

Although the likelihood is tractable and could be maximized by a gradient-based optimization method, HMMs are typically trained by the E-M algorithm (Section 19.2), which has been shown to converge rapidly (approaching the rate of Newton-like methods) in some conditions (if we view the HMM as a big mixture, then the condition is for the final mixture components to be well-separated, i.e., have little overlap) (Xu and Jordan, 1996).

At test time, the sequence of states that maximizes the joint likelihood

$$P(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T, s_1, s_2, \ldots, s_T)$$

can also be obtained using a dynamic programming algorithm (called the Viterbi algorithm). This is a form of *inference* (see Section 13.5) that is called MAP (Maximum A Posteriori) inference because we want to find the most probable value of the unobserved state variables given the observed inputs. Using the same definitions as above (from Section 19.4.1) for the nodes and log-score of the graph $\mathcal{G}$ in which we search for the optimal path, the Viterbi algorithm corresponds to the recursion defined by Eq. 19.7.

If the HMM is structured in such a way that states have a meaning associated with labels of interest, then from the MAP sequence one can read off the associated labels. When the number of states is very large (which happens for example with large-vocabulary speech recognition based on n-gram language models), even the efficient Viterbi algorithm becomes too expensive. In such cases only approximate search is feasible. A common family of search algorithms for HMMs is the *beam search* algorithm (Lowerre, 1976) (Section 19.5.1).

More details about speech recognition are given in Section 12.3. An HMM can be used to associate a sequence of labels $(y_1, y_2, \ldots, y_N)$ with the input $(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T)$, where the output sequence is typically shorter than the input sequence, i.e., $N < T$. Knowledge of $(y_1, y_2, \ldots, y_N)$ constrains the set of compatible state sequences $(s_1, s_2, \ldots, s_T)$, and the generative conditional likelihood

$$P(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T \mid y_1, y_2, \ldots, y_N) = \sum_{s_1, s_2, \ldots, s_T \in \mathcal{S}(y_1, y_2, \ldots, y_N)} \prod_t P(\boldsymbol{x}_t \mid s_t) P(s_t \mid s_{t-1}). \tag{19.9}$$

can be computed using the same forward-backward technique. This enables us to maximize its logarithm during training, as discussed above.

Various discriminative alternatives to the generative likelihood of Eq. 19.9 have been proposed (Brown, 1987; Bahl *et al.*, 1987; Nadas *et al.*, 1988; Juang and Katagiri, 1992; Bengio *et al.*, 1992a; Bengio, 1993; Leprieur and Haffner, 1995; Bengio, 1999a), the simplest of which is simply $P(y_1, y_2, \ldots, y_N \mid \boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T)$,

which is obtained from Eq. 19.9 by Bayes rule, i.e., involving a normalization over all sequences, i.e., the unconstrained likelihood of Eq. 19.8:

$$P(y_1, y_2 \ldots, y_N \mid \boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T) = \frac{P(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T \mid y_1, y_2, \ldots, y_N) P(y_1, y_2, \ldots, y_N)}{P(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T)}.$$

Both the numerator and denominator can be formulated in the framework of the previous section (Eqs. 19.3-19.5), where for the numerator we merge (add) the log-scores coming from the structured output output model $P(y_1, y_2, \ldots, y_N)$ and from the input likelihood model $P(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T \mid y_1, y_2, \ldots, y_N)$. Again, each node of the graph corresponds to a state of the HMM at a particular time step t (which may or may not emit the next output symbol y_i), associated with an input vector $\boldsymbol{x}_t$. Instead of making the relationship to the input the result of a simple parametric form (Gaussian or multinomial, typically), the scores can be computed by a neural network (or any other parametrized differential function). This gives rise to discriminative hybrids of search or graphical models with neural networks, discussed below, Section 19.5.

19.4.3 CRFs

Whereas HMMs are typically trained to maximize the probability of an input sequence $\mathbf{x}$ given a target sequence $\mathbf{y}$ and correspond to a directed graphical model, Conditional Random Fields (CRFs) (Lafferty *et al.*, 2001) are *undirected* graphical models that are trained to maximize the joint probability of the target variables, given input variables, $P(\mathbf{y} \mid \mathbf{x})$. CRFs are special cases of the graph transformer model introduced in Bottou *et al.* (1997); LeCun *et al.* (1998c), where neural nets are replaced by affine transformations and there is a single graph involved. A *graph transformer* is an computational module that transforms a weighted (with a scalar on each arc) directed acyclic graph into another one. Examples of graph transformers are illustrated in Fig. 19.4. In this context, graph transformers can be seen as transforming the set of weights in their input graph into a set of weights on their output graph, typically so that we can compute derivatives of the output graph weights with respect to input graph weights, i.e., we can back-propagated costs through the graph transformer. Section 19.5 below provides more discussion and examples.

Many applications of CRFs involve sequences and the discussion here will be focused on this type of application, although applications to images (e.g. for image segmentation) are also common. Compared to other graphical models, another characteristic of CRFs is that there are no latent variables. The general equation for the probability distribution modeled by a CRF is basically the same as for fully visible (not latent variable) undirected graphical models, also known as Markov Random Fields (MRFs, see Section 13.2.2), *except* that the "potentials"

(terms of the energy function) are parametrized functions of the input variables, and the likelihood of interest is the posterior probability $P(\mathbf{y} \mid \mathbf{x})$.

As in many other MRFs, CRFs often have a particular connectivity structure in their graph, which allows one to perform learning or inference more efficiently. In particular, when dealing with sequences, the energy function typically only has terms that relate neighboring elements of the sequence of target variables. For example, the target variables could form a homogeneous[3] Markov chain of order k (given the input variables). A typical linear CRF example with binary outputs would have the following structure:

$$P(\mathbf{y} = \boldsymbol{y} \mid \mathbf{x}) = \frac{1}{Z} \exp\left(\sum_t y_t (b + \sum_j w_i x_{tj}) + \sum_{i=1}^{k} y_t y_{t-i} (u_i + \sum_j v_{ij} x_{tj}) \right) \tag{19.10}$$

where Z is the normalization constant, which is the sum over all $\boldsymbol{y}$ sequences of the numerator. In that case, the score marginalization framework of Section 19.4.1 and coming from Bottou *et al.* (1997); LeCun *et al.* (1998c) can be applied by making terms in the above exponential correspond to scores associated with nodes t of a graph $\mathcal{G}$. If there were more than two output classes, more nodes per time step would be required but the principle would remain the same. A more general formulation for Markov chains of order d is the following:

$$P(\mathbf{y} = \boldsymbol{y} \mid \mathbf{x}) = \frac{1}{Z} \exp\left(\sum_t \sum_{d'=0}^{d} f_{d'}(y_t, y_{t-1}, \ldots, y_{t-d'}, x_t) \right) \tag{19.11}$$

where $f_{d'}$ computes a potential of the energy function, a parametrized function of both the past target values (up to $y_{t-d'}$) and of the current input value x_t. For example, as discussed below $f_{d'}$ could be the output of an arbitrary parametrized computation, such as a neural network.

Although Z looks intractable, because of the Markov property of the model (order 1, in the example), it is again possible to exploit dynamic programming to compute Z efficiently, as per Eqs. 19.3-19.5). Again, the idea is to compute the sub-sum for sequences of length $t \leq T$ (where T is the length of a target sequence $\boldsymbol{y}$), ending in each of the possible state values at t, e.g., $y_t = 1$ and $y_t = 0$ in the above example. For higher order Markov chains (say order d instead of 1) and a larger number of state values (say N instead of 2), the required sub-sums to keep track of are for each element in the cross-product of $d-1$ state values, i.e., N^{d-1}. For each of these elements, the new sub-sums for sequences of length $t+1$ (for each of the N values at $t+1$ and corresponding $N^{\max(0,d-2)}$ past values for the

[3] meaning that the same parameters are used for every time step

past $d-2$ time steps) can be obtained by only considering the sub-sums for the N^{d-1} joint state values for the last $d-1$ time steps before $t+1$.

Following Eq. 19.7, the same kind of decomposition can be performed to efficiently find the MAP configuration of y's given $\boldsymbol{x}$, where instead of products (sums inside the exponential) and sums (for the outer sum of these exponentials, over different paths) we respectively have sums (corresponding to adding the sums inside the exponential) and maxima (across the different competing "previous-state" choices).

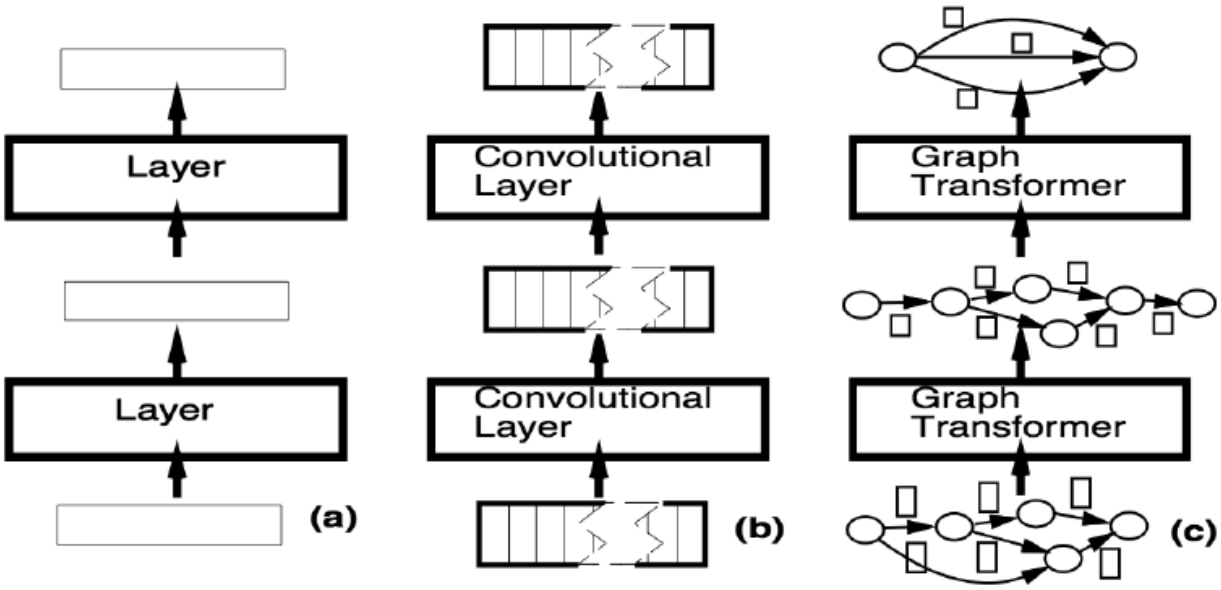

Figure 19.4: Illustration of the stacking of graph transformers (right, c) as a generalization of the stacking of convolutional layers (middle, b) or of regular feedforward layers that transform fixed-size vectors (left, a). Figure reproduced with permission from the authors of Bottou *et al.* (1997). Quoting from that paper, (c) shows that "multilayer graph transformer networks are composed of trainable modules that operate on and produce graphs whose arcs carry numerical information".

19.5 Combining Neural Networks and Search

The idea of combining neural networks with HMMs or related search or alignment-based components (such as graph transformers) for speech and handwriting recognition dates from the early days of research on multi-layer neural networks (Bourlard and Wellekens, 1990; Bottou *et al.*, 1990; Bengio, 1991; Bottou, 1991; Haffner *et al.*, 1991; Bengio *et al.*, 1992a; Matan *et al.*, 1992; Bourlard and Morgan, 1993; Bengio *et al.*, 1995; Bengio and Frasconi, 1996; Baldi and Brunak, 1998) – and see more references in Bengio (1999b). See also 12.5 for combining recurrent and

other deep learners with generative models such as CRFs, GSNs or RBMs.

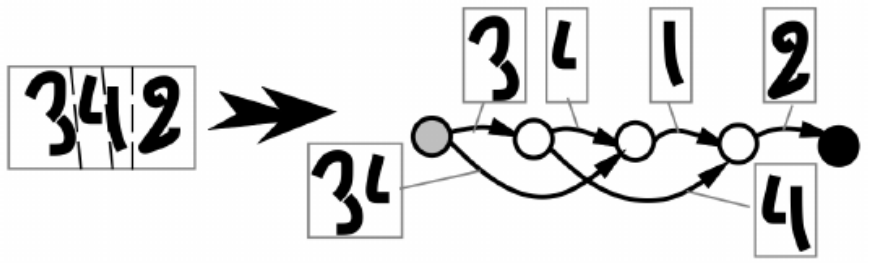

Figure 19.5: Illustration of the input and output of a simple graph transformer that maps a singleton graph corresponding to an input image to a graph representing hypothesized segmentation hypotheses. Reproduced with permission from the authors of Bottou *et al.* (1997).

The principle of efficient marginalization and inference for temporally structured outputs by exploiting dynamic programming (Sec. 19.4.1) can be applied not just when the log-scores of Eqs. 19.2 and 19.4 are parameters or linear functions of the input. This principle can also applied when the log-scores are *learned non-linear functions* of the input, including functions represented by a neural network. Bottou *et al.* (1997) and LeCun *et al.* (1998c) introduced this idea and the powerful idea of *learned graph transformers*, illustrated in Fig. 19.4. In this context, a graph transformer is a machine that can *map a directed acyclic graph* $\mathcal{G}_{\text{in}}$ *to another graph* $\mathcal{G}_{\text{out}}$. Both input and output graphs have paths that represent hypotheses about the observed data.

For example, a segmentation graph transformer takes a singleton input graph (the image $\boldsymbol{x}$) and outputs a graph representing segmentation hypotheses (regarding sequences of segments that could each contain a character in the image). This process is illustrated in Fig. 19.5. Such a graph transformer could be used as one layer of a *graph transformer network* for handwriting recognition or document analysis for reading amounts on checks, as illustrated respectively in Fig.s 19.6 and 19.7.

For example, after the segmentation graph transformer, a recognition graph transformer could expand each node of the segmentation graph into a subgraph whose arcs correspond to different interpretations of the segment (which character is present in the segment?). Then, a dictionary graph transformer takes the recognition graph and expands it further by considering only the sequences of characters that are compatible with sequences of words in the language of interest. Finally, a language-model graph transformer expands sequences of word hypotheses so as to include multiple words in the state (context) and weigh the arcs according to the language model next-word log-probabilities.

Each of these transformations is parametrized and takes real-valued scores

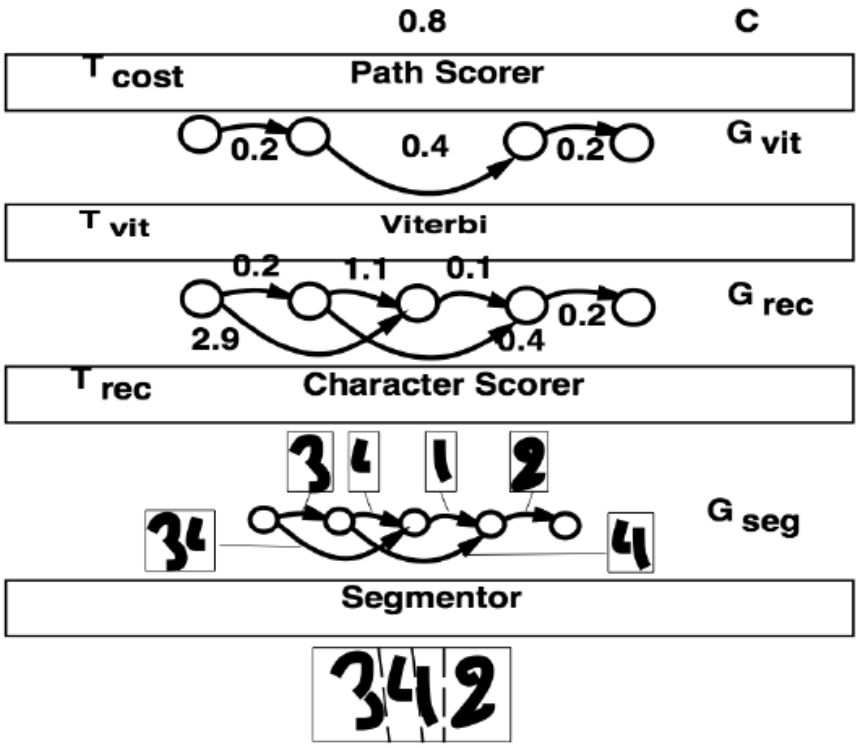

Figure 19.6: Illustration of the graph transformer network that has been used for finding the best segmentation of a handwritten word, for handwriting recognition. Reproduced with permission from Bottou *et al.* (1997).

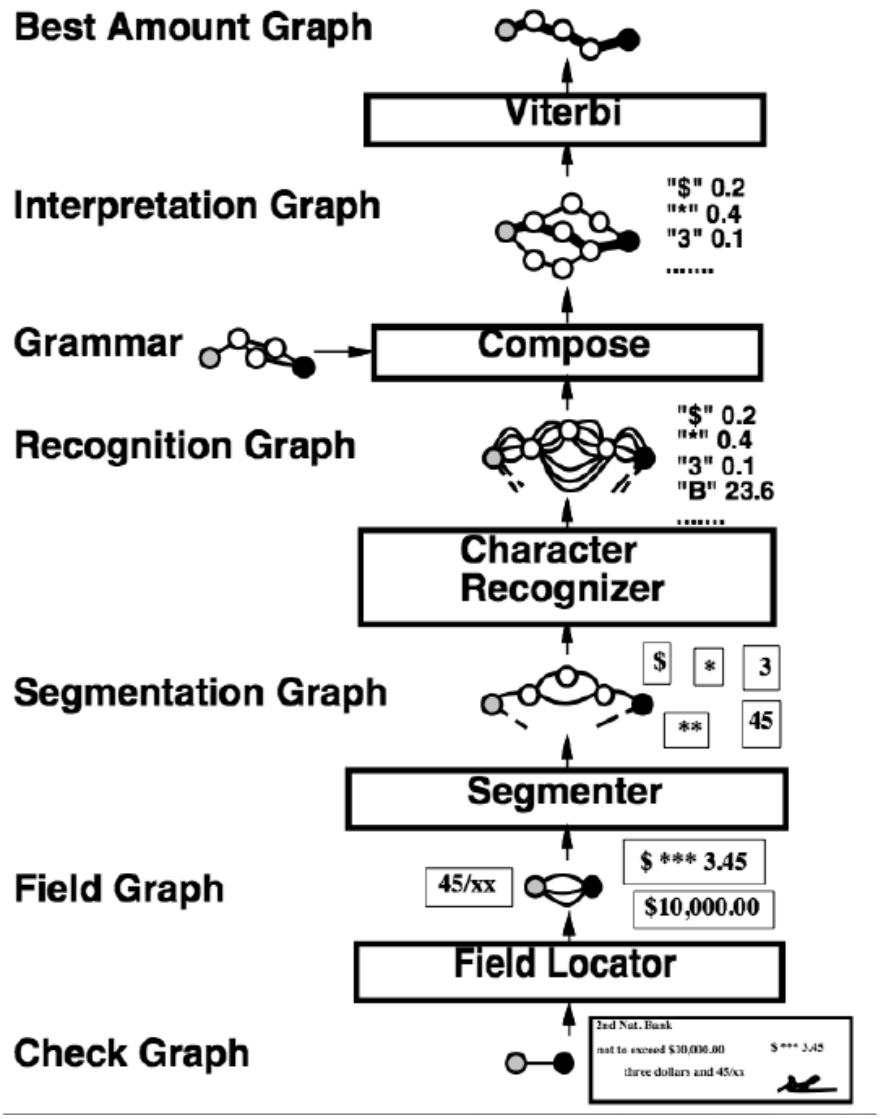

Figure 19.7: Illustration of the graph transformer network that has been used for reading amounts on checks, starting from the single graph containing the image of the graph to the recognized sequences of characters corresponding to the amount on the graph, with currency and other recognized marks. Note how the grammar graph transformer composes the grammar graph (allowed sequences of characters) and the recognition graph (with character hypotheses associated with specific input segments, on the arcs) into an interpretation graph that only contains the recognition graph paths that are compatible with the grammar. Reproduced with permission from Bottou *et al.* (1997).

on the arcs of the input graph into real-valued scores on the arcs of the output graph. These transformations can be parametrized and learned by gradient-based optimization over the whole series of graph transformers.

19.5.1 Approximate Search

Unfortunately, as in the above example, when the number of nodes of the graph becomes very large (e.g., considering all previous n words to condition the log-probability of the next one, for n large), even dynamic programming (whose computation scales with the number of arcs) is too slow for practical applications such as speech recognition or machine translation. A common example is when a recurrent neural network is used to compute the arcs log-score, e.g., as in neural language models (Section 12.4). Since the prediction at step t depends on all $t-1$ previous choices, the number of states (nodes of the search graph $\mathcal{G}$) grows exponentially with the length of the sequence. In that case, one has to resort to *approximate search*.

Beam Search

In the case of sequential structures as discussed in this chapter, a common family of approximate search algorithms is the *beam search* (Lowerre, 1976). To perform beam search with *beam width* k, we do the following:

- Break the nodes of the graph into g groups containing only "comparable nodes", e.g., the group of nodes n for which the maximum length of the paths ending at n is exactly t.
- Process these groups of nodes sequentially, keeping only at each step t a selected subset $\mathbb{S}_t$ of the nodes (the "beam"), chosen based on the subset $\mathbb{S}_{t-1}$. Each node in $\mathbb{S}_t$ is associated with a score $\hat{v}(\mathcal{G}^n)$ that represents an approximation (a lower bound) on the maximum total log-score of the path ending at the node, $v(\mathcal{G}^n)$ (defined in Eq. 19.7, Viterbi decoding).
- $\mathbb{S}_t$ is obtained by following all the arcs from the nodes in $\mathbb{S}_{t-1}$, and sorting all the resulting group t nodes n according to their estimated (lower bound) score
$$\hat{v}(\mathcal{G}^n) = \max_{n' \in \mathbb{S}_{t-1} and n' \in \text{pred}(n)} \hat{v}(\mathcal{G}^{n'}) + a_{n',n},$$
while keeping track of the argmax in order to trace back the estimated best path. Only the k nodes with the highest log-score are kept and stored in $\mathbb{S}_t$.
- The estimated best final node can be read off from $\max_{n \in \mathbb{S}_T} \hat{v}(\mathcal{G}^n)$ and the estimated best path from the associated argmax choices made along the way, just like in the Viterbi algorithm.

One problem with beam search is that the beam often ends up lacking in diversity, making the approximation poor. For example, imagine that we have two "types" of solutions, but that each type has exponentially many variants (as a function of t), due, e.g., to small independent variations in ways in which the type can be expressed at each time step t. Then, even though the two types may have close best log-score up to time t, the beam could be dominated by the one that wins slightly, eliminating the other type from the search, although later time steps might reveal that the second type was actually the best one.

19.6 Variational Inference and Learning

One common difficulty in probabilistic modeling is that the posterior distribution $p(\boldsymbol{h} \mid \boldsymbol{v})$ is infeasible to compute for many models with hidden variables $\boldsymbol{h}$ and visible variables $\boldsymbol{v}$. Expectations with respect to this distribution may also be intractable.

Consider as an example the *binary sparse coding* model. In this model, the input $\boldsymbol{v} \in \mathbb{R}^n$ is formed by adding Gaussian noise to the sum of m different components which can each be present or absent. Each component is switched on or off by the corresponding hidden unit in $\boldsymbol{h} \in \{0, 1\}^m$:

$$p(h_i = 1) = \sigma(b_i)$$
$$p(\boldsymbol{v} \mid \boldsymbol{h}) = \mathcal{N}(\boldsymbol{v} \mid \boldsymbol{W}\boldsymbol{h}, \beta^{-1})$$

where $\boldsymbol{b}$ is a learn-able set of biases, $\boldsymbol{W}$ is a learn-able weight matrix, and $\boldsymbol{\beta}$ is a learn-able, diagonal precision matrix.

Training this model with maximum likelihood requires taking the derivative with respect to the parameters. Consider the derivative with respect to one of the biases:

$$\frac{\partial}{\partial b_i} \log p(\boldsymbol{v})$$
$$= \frac{\frac{\partial}{\partial b_i} p(\boldsymbol{v})}{p(\boldsymbol{v})}$$
$$= \frac{\frac{\partial}{\partial b_i} \sum_{\boldsymbol{h}} p(\boldsymbol{h}, \boldsymbol{v})}{p(\boldsymbol{v})}$$
$$= \frac{\frac{\partial}{\partial b_i} \sum_{\boldsymbol{h}} p(\boldsymbol{h}) p(\boldsymbol{v} \mid \boldsymbol{h})}{p(\boldsymbol{v})}$$
$$= \frac{\sum_{\boldsymbol{h}} p(\boldsymbol{v} \mid \boldsymbol{h}) \frac{\partial}{\partial b_i} p(\boldsymbol{h})}{p(\boldsymbol{v})}$$

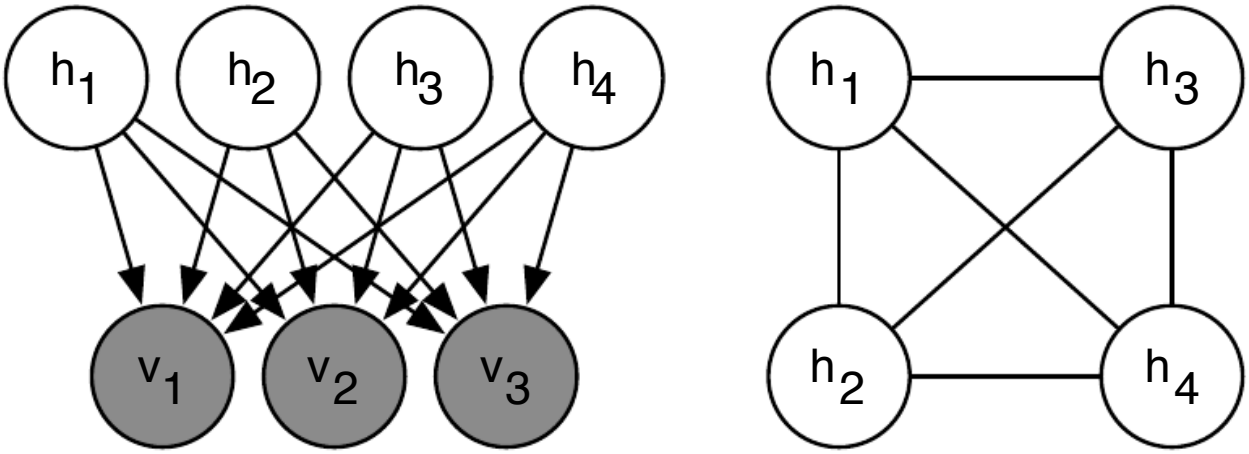

Figure 19.8: The graph structure of a binary sparse coding model with four hidden units. Left) The graph structure of $p(\boldsymbol{h}, \boldsymbol{v})$. Note that the edges are directed, and that every two hidden units co-parents of every visible unit. Right) The graph structure of $p(\boldsymbol{h} \mid \boldsymbol{v})$. In order to account for the active paths between co-parents, the posterior distribution needs an edge between all of the hidden units.

$$= \sum_{\boldsymbol{h}} p(\boldsymbol{h} \mid \boldsymbol{v}) \frac{\frac{\partial}{\partial b_i} p(\boldsymbol{h})}{p(\boldsymbol{h})}$$

$$= \sum_{\boldsymbol{h}} p(\boldsymbol{h} \mid \boldsymbol{v}) \frac{\frac{\partial}{\partial b_i} p(\boldsymbol{h})}{p(\boldsymbol{h})}$$

$$= \mathbb{E}_{\boldsymbol{h}\ p(\boldsymbol{h}|\boldsymbol{v})} \frac{\partial}{\partial b_i} \log p(\boldsymbol{h}).$$

This requires computing expectations with respect to $p(\boldsymbol{h} \mid \boldsymbol{v})$. Unfortunately, $p(\boldsymbol{h} \mid \boldsymbol{v})$ is a complicated distribution. See Fig. 19.8 for the graph structure of $p(\boldsymbol{h}, \boldsymbol{v})$ and $p(\boldsymbol{h} \mid \boldsymbol{v})$. The posterior distribution corresponds to the complete graph over the hidden units, so variable elimination algorithms do not help us to compute the required expectations any faster than brute force.

One solution to this problem is to use *variational methods*. Variational methods involve using a simple distribution $q(\boldsymbol{h})$ to approximate the true, complicated posterior $p(\boldsymbol{h} \mid \boldsymbol{v})$. The name "variational" derives from their frequent use of a branch of mathematics called *calculus of variations*. However, not all variational methods use calculus of variations.

TODO variational inference involves maximization of a BOUND TODO variational inference also usually involves a restriction on the function family

TODO

19.6.1 Discrete Latent Variables

TODO– BSC example

19.6.2 Calculus of Variations

Many machine learning techniques are based on minimizing a function $J(\boldsymbol{\theta})$ by finding the input vector $\boldsymbol{\theta} \in \mathbb{R}^n$ for which it takes on its minimal value. This can be accomplished with multivariate calculus and linear algebra, by solving for the critical points where $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = 0$. In some cases, we actually want to solve for a function $f(\boldsymbol{x})$, such as when we want to find the probability density function over some random variable. This is what *calculus of variations* enables us to do.

A function of a function f is known as a *functional* $J[f]$. Much as we can take partial derivatives of a function with respect to elements of its vector-valued argument, we can take *functional derivatives*, also known as *variational derivatives* of a functional $J[f]$ with respect to respect to individual values of the function $f(\boldsymbol{x})$. The functional derivative of the functional J with respect to the value of the function f at point $\boldsymbol{x}$ is denoted $\frac{\delta}{\delta f(x)} J$.

A complete formal development of functional derivatives is beyond the scope of this book. For our purposes, it is sufficient to state that for differentiable functions $f(\boldsymbol{x})$ and differentiable functions $g(y, \boldsymbol{x})$ with continuous derivatives, that

$$\frac{\delta}{\delta f(\boldsymbol{x})} \int g\left(f(\boldsymbol{x}), \boldsymbol{x}\right) d\boldsymbol{x} = \frac{\partial}{\partial y} g(f(\boldsymbol{x}), \boldsymbol{x}). \tag{19.12}$$

To gain some intuition for this identity, one can think of $f(\boldsymbol{x})$ as being a vector with uncountably many elements, indexed by a real vector $\boldsymbol{x}$. In this (somewhat incomplete view), the identity providing the functional derivatives is the same as we would obtain for a vector $\boldsymbol{\theta} \in \mathbb{R}^n$ indexed by positive integers:

$$\frac{\partial}{\partial \theta_i} \sum_j g(\theta_j, j) = \frac{\partial}{\partial \theta_i} g(\theta_i, i).$$

Many results in other machine learning publications are presented using the more general *Euler-Lagrange equation* which allows g to depend on the derivatives of f as well as the value of f, but we do not need this fully general form for the results presented in this book.

To optimize a function with respect to a vector, we take the gradient of the function with respect to the vector and solve for the point where every element of the gradient is equal to zero. Likewise, we can optimize a functional by solving for the function where the functional derivative at every point is equal to zero.

As an example of how this process works, consider the problem of finding the

probability distribution function over $x \in \mathbb{R}$ that has maximal Shannon entropy.

Recall that the entropy of a probability distribution $p(x)$ is defined as

$$H[p] = -\mathbb{E}_x \log p(x).$$

For continuous values, the expectation is an integral:

$$H[p] = -\int p(x) \log p(x) dx.$$

We cannot simply maximize $H(x)$ with respect to the function $p(x)$, because the result might not be a probability distribution. Instead, we need to use Lagrange multipliers, to add a constraint that $p(x)$ integrate to 1. Also, the entropy increases without bound as the variance increases, so we can only search for the distribution with maximal entropy for fixed variance σ^2. Finally, the problem is underdetermined because the distribution can be shifted arbitrarily without changing the entropy. To impose a unique solution, we add a constraint that the mean of the distribution be μ. The Lagrangian functional for this optimization problem is

$$\mathcal{L}[p] = \lambda_1 \left(\int p(x) dx - 1 \right) + \lambda_2 \left(\mathbb{E}[x] - \mu \right) + \lambda_3 \left(\mathbb{E}[(x-\mu)^2] - \sigma^2 \right) + H[p]$$

$$= \int \left(\lambda_1 p(x) + \lambda_2 p(x) x + \lambda_3 p(x)(x-\mu)^2 - p(x) \log p(x) \right) dx - \lambda_1 - \mu\lambda_2 - \sigma^2 \lambda_3 .$$

To minimize the Lagrangian with respect to p, we set the functional derivatives equal to 0:

$$\forall x, \frac{\delta}{\delta p(x)} \mathcal{L} = \lambda_1 + \lambda_2 x + \lambda_3 (x-\mu)^2 - 1 - \log p(x) = 0.$$

This condition now tells us the functional form of $p(x)$. By algebraically rearranging the equation, we obtain

$$p(x) = \exp\left(-\lambda_1 - \lambda_2 x + \lambda_3 (x-\mu)^2 + 1 \right).$$

We never assumed directly that $p(x)$ would take this functional form; we obtained the expression itself by analytically minimizing a functional. To finish the minimization problem, we must choose the λ values to ensure that all of our constraints are satisfied. We are free to choose any λ values, because the gradient of the Lagrangian with respect to the λ variables is zero so long as the constraints are satisfied. To satisfy all of the constraints, we may set $\lambda_1 = \log \sigma \sqrt{2\pi}$, $\lambda_2 = 0$, and $\lambda_3 = -\frac{1}{2\sigma^2}$ to obtain

$$p(x) = \mathcal{N}(x \mid \mu, \sigma^2).$$

This is one reason for using the normal distribution when we do not know the true distribution. Because the normal distribution has the maximum entropy, we impose the least possible amount of structure by making this assumption.

What about the probability distribution function that *minimizes* the entropy? It turns out that there is no specific function that achieves minimal entropy. As functions place more mass on $x = \mu \pm \sigma$ and less on all other values of x, they lose entropy. However, any function placing exactly zero mass on all but two points does not integrate to one, and is not a valid probability distribution. There thus is no single minimal entropy probability distribution function, much as there is no single minimal positive real number.

19.6.3 Continuous Latent Variables

TODO: Gaussian example from IG's thesis? TODO: S3C example

19.7 Stochastic Inference

TODO: Charlie Tang's SFNNs? Is there anything else where sampling-based inference actually gets used?

19.8 Learned Approximate Inference

TODO: wake-sleep algorithm

In chapter 18.2 we saw that one possible explanation for the role of dream sleep in human beings and animals is that dreams could provide the negative phase samples that Monte Carlo training algorithms use to approximate the negative gradient of the log partition function of undirected models. Another possible explanation for biological dreaming is that it is providing samples from $p(\boldsymbol{h}, \boldsymbol{v})$ which can be used to train an inference network to predict $\boldsymbol{h}$ given $\boldsymbol{v}$. In some senses, this explanation is more satisfying than the partition function explanation. Monte Carlo algorithms generally do not perform well if they are run using only the positive phase of the gradient for several steps then with only the negative phase of the gradient for several steps. Human beings and animals are usually awake for several consecutive hours then asleep for several consecutive hours, and it is not readily apparent how this schedule could support Monte Carlo training of an undirected model. Learning algorithms based on maximizing $\mathcal{L}$ can be run with prolonged periods of improving q and prolonged periods of improving $\boldsymbol{\theta}$, however. If the role of biological dreaming is to train networks for predicting q, then this explains how animals are able to remain awake for several hours (the

longer they are awake, the greater the gap between $\mathcal{L}$ and $\log p(v)$, but $\mathcal{L}$ will remain a lower bound) and to remain asleep for several hours (the generative model itself is not modified during sleep) without damaging their internal models. Of course, these ideas are purely speculative, and there is no hard evidence to suggest that dreaming accomplishes either of these goals. Dreaming may also serve reinforcement learning rather than probabilistic modeling, by sampling synthetic experiences from the animal's transition model, on which to train the animal's policy. Or sleep may serve some other purpose not yet anticipated by the machine learning community.

TODO: DARN and NVIL? TODO: fast DBM inference