Lecture 8

A.L. Yuille

February 12, 2012

1 Introduction

This lecture describes learning with hidden variables. The standard approach is the Expectation Maximization (EM) algorithm. This involves computations which can be very difficult to do. We describe dynamic programming which can be used to perform these computations in polynomial time. The next lecture describes hidden markov models which can be seen as a special case of this approach (although they were developed earlier). Another example, is learning probabilistic context free grammars.

1.1 Free Energy and the Expectation Maximization (EM) algorithm

Suppose we have a parameterized distribution $P(\vec{x}, \vec{h}|\vec{\lambda})$ where $\vec{x}$ is observed, $\vec{h}$ are the hidden states, and $\vec{\lambda}$ are the model parameters (to be learnt). Given training data $\{\vec{x}_\mu\}$ we should sum out the hidden states $\vec{h}$ to obtain $P(\vec{x}_\mu|\vec{\lambda}) = \sum_{\vec{h}} P(\vec{x}, \vec{h}|\vec{\lambda})$ and estimate $\vec{\lambda}$ by maximum likelihood (ML) by computing:

$$\vec{\lambda}^* = \arg\max_{\vec{\lambda}} \prod_{\mu} P(\vec{x}_\mu|\vec{\lambda}). \tag{1}$$

But usually it is impractical to compute $P(\vec{x}_\mu|\vec{\lambda})$ from $P(\vec{x}, \vec{h}|\vec{\lambda})$. (If we can compute $P(\vec{x}_\mu|\vec{\lambda})$ and express it in exponential form then we can learn $\vec{\lambda}$ by the methods described in earlier lectures).

The standard way to learn with missing/hidden data is the EM algorithm (Dempster et al.). We will describe this from the free energy formulation (Hinton and Neal, Hathaway) because it is more general than the standard EM formulation.

The free energy formulation starts by introduction a new variable $q(\vec{h})$ which is a probability distribution over the hidden variables – hence $q(\vec{h}) \geq 0 \; \forall \; \vec{h}$ and $\sum_{\vec{h}} q(\vec{h}) = 1$. See figure (1).

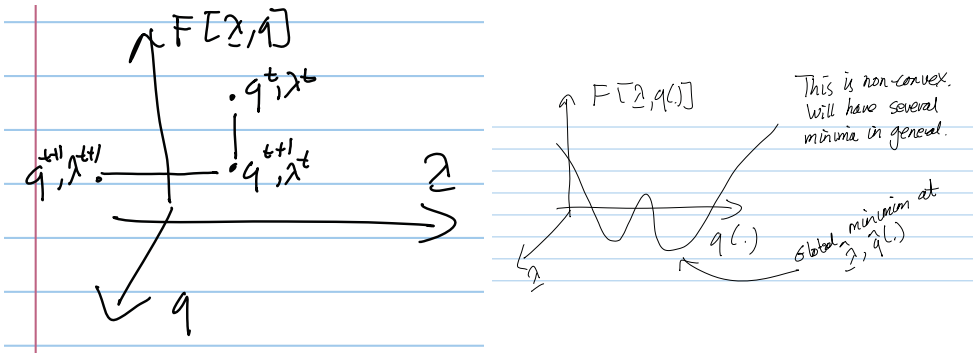

Figure 1: The EM algorithm minimizes the free energy with respect to each variable in turn (left panel). But it is only guaranteed to converge to a local minimum (right panel).

Instead we can apply the EM algorithm to minimize (local minima) a free energy:

$$F(\{q_\mu\}, \vec{\lambda}) = -\log P(\vec{x}|\vec{\lambda}) + \sum_{\vec{h}} q(\vec{h}) \log \frac{q(\vec{h})}{P(\vec{h}|\vec{x}, \vec{\lambda})},$$
$$F(\{q_\mu\}, \vec{\lambda}) = \sum_\mu \sum_{\vec{h}^\mu} q_\mu(\vec{h}^\mu) \log q_\mu(\vec{h}^\mu) - \sum_\mu \sum_{\vec{h}^\mu} q_\mu(\vec{h}^\mu) \log P(\vec{x}_\mu, \vec{h}^\mu|\vec{\lambda}). \quad (2)$$

The top equation (for one image only) add a Kullback-Leibler (KL) divergence term between $q(\vec{h})$ and $P(\vec{h}|\vec{x}, \vec{\lambda})$. The KL divergence takes its minimal value of 0 where $q(\vec{h}) = P(\vec{h}|\vec{x}, \vec{\lambda})$. So the global minimum of $F(\{q_\mu\}, \vec{\lambda})$ corresponds to setting $\vec{\lambda}^* = \arg\min_{\vec{\lambda}} -\log P(\vec{x}|\vec{\lambda})$ and $q^*(\vec{h}) = P(\vec{h}|\vec{x}, \vec{\lambda}^*)$. Hence the minimum of $F(\{q_\mu\}, \vec{\lambda})$ yields the maximum likelihood estimation of $\vec{\lambda}$ – see equation (1). The bottom equation differs in two respects. Firstly it includes all the data, which requires specifying a distribution $q_\mu(\vec{h}^\mu)$ for each image. Secondly, we rearrange the terms using the following manipulations: (i) $\log \frac{q(\vec{h})}{P(\vec{h}|\vec{x}, \vec{\lambda})} = \log q(\vec{h}) - \log P(\vec{h}|\vec{x}, \vec{\lambda})$, (ii) $-\log P(\vec{x}|\vec{\lambda}) = -\sum_{\vec{h}} q(\vec{h}) \log P(\vec{x}|\vec{\lambda})$, and (iii) $\log P(\vec{h}|\vec{x}, \vec{\lambda}) + \log P(\vec{x}|\vec{\lambda}) = \log P(\vec{x}, \vec{h}|\vec{\lambda})$.

NOTE that we have distributions $q_\mu(.)$ for the states $\vec{h}_\mu$ of all training examples, but the parameters $\vec{\lambda}$ are common for all examples.

The EM algorithm can be derived by doing *coordinate descent* on $F(\{q_\mu\}, \vec{\lambda})$ – i.e. minimizing with respect to each variable in alternation keeping the other variable fixed. This is guaranteed to decrease the energy and converge to a local minima of F. It can be shown that $F(\{q_\mu\}, \vec{\lambda})$ is convex in each variable separately.

This gives the following two steps – which correspond to the E- and M- steps of the EM algorithm:

$$\text{E} - \text{Step} \quad q_\mu^{t+1}(\vec{h}_\mu) = P(\vec{h}_\mu|\vec{x}_\mu, \vec{\lambda}^t),$$
$$\text{M} - \text{step} \quad \vec{\lambda}^{t+1} = \arg\min \sum_\mu \sum_{\vec{h}_\mu} q_\mu^{t+1}(\vec{h}_\mu) \log P(\vec{x}_\mu, \vec{y}_\mu)|\vec{\lambda}). \quad (3)$$

These update rules take particularly simple, and insightful, forms if the distributions are exponential – $P(\vec{x}, \vec{h}|\vec{\lambda}) = \frac{1}{Z[\vec{\lambda}]} \exp\{\vec{\lambda} \cdot \vec{\phi}(\vec{x}, \vec{h})\}$. In this case, the M-step reduces to the update rule:

$$(1/N) \sum_\mu \sum_{\vec{h}_\mu} \vec{\phi}(\vec{x}_\mu, \vec{h}_\mu) = \sum_{\vec{x}, \vec{h}} \vec{\phi}(\vec{x}, \vec{h}) P(\vec{x}, \vec{h}|\vec{\lambda}). \quad (4)$$

This matches the statistics of the data (the left hand side) with the expected statistics of the model (right hand side), *except* that since we cannot observe the states of the hidden variables $\{\vec{h}_\mu\}$ instead we take their expectations with respect to the current estimates $\{q_\mu(\vec{h}_\mu)\}$ of the hidden variables. This reduces to equation for ML if we have no hidden variables (previous lectures) if additional information is provided so that the states of the hidden variables are known (i.e. the limit where $q_\mu(\vec{h}_\mu)$ is peaked at the correct values of the hidden states).

Equation (4) is (almost always) to hard to be solved directly. But, like the case without hidden variables, it can be expressed in terms of minimizing a convex function:

$$\vec{\lambda}^{t+1} = \arg\min\{\sum_\mu \sum_{\vec{h}_\mu} q_\mu^{t+1}(\vec{h}_\mu) \vec{\lambda} \cdot \vec{\phi}(\vec{x}_\mu, \vec{h}_\mu) - N \log Z[\vec{\lambda}], \quad (5)$$

which can be performed by steepest descent or CCCP. But it does require us to be able to compute the expectations $\sum_{\vec{x}, \vec{h}} P(\vec{x}, \vec{h}|\vec{\lambda}) \vec{\phi}(\vec{x}, \vec{h})$ and $\sum_{\vec{h}} q(\vec{h}) \vec{\phi}(\vec{x}, \vec{h})$.

The free energy formulation allows us to generalize the basic EM algorithm in two ways: (I) we can attempt to minimize F by other algorithms which are guaranteed to decrease it (e.g., steepest descent) –

in other words, we can replace the E- and M- steps by any updates for $q_\mu(.)$ and $\vec{\lambda}$ which decrease F (note that the EM algorithm can be derived as discrete iterative algorithm and re-expressed as CCCP). (II) We can restrict the class of probability distributions q_μ allowed during the minimization of F (the purpose is to simply the computations). For example, we could restrict $q(.)$ to be a factorizable model. This relates to mean field theory/variational methods (described earlier in the course) and has similar motivations.

Note that there are Machine Learning methods – e.g., Latent Support vector machines and multiple instance learning – which also deal with hidden variables. These methods involve algorithms, like EM, which estimate the hidden variables and the parameters alternatively. These methods, and relations to the probabilistic methods described here, are discussed in a handout (Yuille and He) which will be put up on the web page. (It is possible to re-derive the machine learning methods as computationally simpler approximations to the probabilistic methods – but this involves introducing loss functions).

2 Computation and EM

The previous section shows that EM requires the following computations.

Firstly, the E-step requires us to compute the conditional distribution $P(\vec{h}|\vec{x},\vec{\lambda})$ from $P(\vec{h},\vec{x}|\lambda)$. Formally this is given by $P(\vec{h}|\vec{x},\vec{\lambda}) = \frac{P(\vec{h},\vec{x}|\lambda)}{\sum_{\vec{h}} P(\vec{h},\vec{x}|\lambda)}$. If we use exponential distributions, $P(\vec{h},\vec{x}|\lambda) = \frac{1}{Z[\vec{\lambda}]} \exp\{\vec{\lambda}\cdot\vec{\phi}(\vec{x},\vec{h})\}$, then this requires the ability to compute $\sum_{\vec{h}} \exp\{\vec{\phi}(\vec{x},\vec{h})\}$.

Secondly, the M-step requires computing: (i) the expected value of the statistics with respect to $q(.)$: $\sum_{\vec{h}} q(\vec{h})\vec{\phi}(\vec{x},\vec{h})$, (ii) the expectation $\sum_{\vec{x},\vec{h}} \vec{\phi}(\vec{x},\vec{h})P(\vec{x},\vec{h}|\vec{\lambda})$, and (iii) the minimum $\vec{\lambda}^{t+1} = \arg\min\{\sum_\mu \sum_{\vec{h}_\mu} q_\mu^{t+1}(\vec{h}_\mu)\vec{\lambda}\cdot\vec{\phi}(\vec{x}_\mu,\vec{h}_\mu) - N\log Z[\vec{\lambda}]$.

These computations are very demanding. Usually no algorithms exist which are guaranteed to perform them in polynomial time. We have to rely on approximate algorithms such as variational methods, belief propagation, or sampling techniques like MCMC (e.g., Gibbs sampling or Metropolis-Hastings).

The one exception is if the probability distributions can be expressed in terms of graphical models defined over graphs without closed loops. In this case dynamic programming (DP) can be used to perform these computations efficiently. DP includes a range of different methods – sometimes called Viterbi, foreward/backward, Baum-Welch. Indeed DP can be thought of as a special case of the general A* class of algorithms (e.g., see review by Coughlan and Yuille) which also includes Dijkstra's algorithm. Note that dynamic programming can be extended to graphs with a limited amount of closed loops by the junction trees method (Lauritzen Spiegelhalter).

Dynamic programming makes it possible to perform these computations efficiently. All of them require the ability to either compute a sum or an expectation efficiently. We will describe how DP does this in the next section. Also we can simply things even further by using dynamic programming to convert a probability distribution defined over an undirected graph (but without closed loops) to a directed probability distribution – e.g. from $P(\vec{x}) = (1/Z)\exp\{\sum_{i=1}^N \phi(x_{i+1},x_i)\}$ to a directed (i.e. conditional distribution $P(\vec{x}) = P(x_1)\prod_{i=1}^N P(x_{i+1}|x_i)$. In this case, we can sometimes compute the expectations — $\sum_{\vec{h}} q(\vec{h})\vec{\phi}(\vec{x},\vec{h}),\sum_{\vec{x},\vec{h}} \vec{\phi}(\vec{x},\vec{h})P(\vec{x},\vec{h}|\vec{\lambda})$ – analytically (i.e. as a function of $\vec{\lambda}$) which makes it possible to perform the M-step directly by solving equation (4) (i.e. without needing to solve equation (5) by steepest descent or GIS), which is illustrated in the next lecture by HMMs.

2.1 Dynamic Programming for Inference

If the underlying graph is a tree (i.e. it has no closed loops), dynamic programming (DP) algorithms can be used to exploit this structure, see figure (2). The intuition behind DP can be illustrated by planning the shortest route for a trip from Los Angeles to Boston. To determine the cost of going via Chicago, you only need to calculate the shortest route from LA to Chicago and then, independently, from Chicago to Boston. Decomposing the route in this way, and taking into account the linear nature of the trip, gives an efficient algorithm with convergence rates which are polynomial in the number of nodes and hence are often

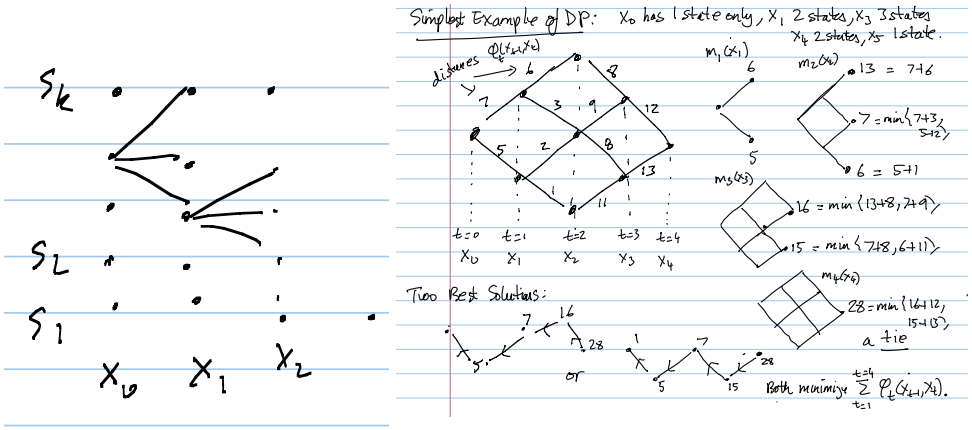

Figure 2: Examples of Dynamic Programming. Left panel: travelling from the west coast of the USA to the east coast, where each state $x_0, ...$ can take a limited set of values $s_1, ...$ Right panel: picking the shortest path to travel from the start to the finish.

feasible for computation. Equation (**??**) is one illustration for how the dynamic programming can exploit the structure of the problem to simplify the computation.

Dynamic Programming can be used when we can express the graph as a tree structure and hence order the nodes. But we will only describe a special class of these probability models to give the main ideas. This distribution $P(\vec{x})$ is defined over variables $\vec{x} = (x_0, ..., x_N)$ where each x_i can take k possible states $x_i \in S = \{s_1, ..., s_k\}$. The distribution is of form:

$$P(\vec{x}) = \frac{1}{Z} \exp\{-E(\vec{x})\}, \text{ where } E(\vec{x}) = \sum_{t=0}^{N-1} \phi_t(x_t, x_{t+1}). \tag{6}$$

The max rule version of dynamic programming computes $\hat{\vec{x}} = \arg\max P(\vec{x}) = \arg\min E(\vec{x})$. This is performed by a forward pass followed by a backward pass. The forward pass defines a function $m_i(x_i)$ which is initialized by $m_1(x_1) = \min_{\{s_i : i=1,...,k\}} \phi(x_0 = s_i, x_1)$ and then computed recursively by:

$$m_t(x_t) = \min_{\{s_i : i=1,...,k\}} \{m_{t-1}(x_{t+1} = s_i) + \phi_t(x_{t-1} = s_i, x_t)\}, \tag{7}$$

We claim that the minimum value of $E(\vec{x})$ is obtained by:

$$\min_{\vec{x}} E(\vec{x}) = \min_{\{s_i : i=1,...,k\}} m_N(x_n = s_i). \tag{8}$$

This is computed in $O(k^2 N)$ time. This claim can be proved by induction – $\min_{x_t \in S} m_t(x_t) = \min_{x_0,...,x_{t-1} \in S} \{\phi_1(x_0, x_1) + ... + \phi_t(x_{t-1}, x_t)\}$.

The backwards pass finds the optimal path (knowing the optimal cost). To find the optimal path we trace backwards. Let $\hat{}_N = \arg\min_{s_i \in S} m_N(x_N = s_i)$. Then proceed recursively by $\hat{x}_t = \arg\min_{s_i \in S} \{m_t(sx_t = s_i) + \phi_{t+1}(x_t = s_i, \hat{x}_{t+1})\}$ where ties are broken arbitrarily. This gives the solution $\hat{\vec{x}} = (\hat{x}_0, ..., \hat{x}_N)$. The backward pass is $O(Nk)$ so it is faster than the forward pass.

Dynamic programming can also be used to compute other properties of the distribution $P(\vec{x})$. These include the normalization constant $Z = \sum_{\vec{x}} \exp\{-E(\vec{x})\}$, the marginals $P_i(x_i) = \sum_{\vec{x}/x_i} P(\vec{x})$, the conditionals $P(x_{t_1}|x_t)$ and $P(x_t|x_{t+1})$, the expectation $\sum_{\vec{x}} P(\vec{x}) \sum_{t=0}^{N} h_t(x_t, x_{t+1})$. Note that the ability to compute the conditionals means that we can translate the distribution $P(\vec{x})$, which is specified in exponential form, into a directed distribution $P_0(x_0) \prod_{t=1}^{N} P(x_t|x_{t-1})$ or $P_N(x_N) \prod_{t=1}^{N} P(x_{t-1}|x_t)$.

These computations are done by the sum rule of dynamic programming. For example, define $V_1(x_1) = \sum_{s_i \in S} \exp\{-\phi_1(x_0 = s_i, x_1)\}$. Then recursively compute:

$$V_t(x_t) = \sum_{s_i \in S} V_{t-1}(x_{t-1} = s_i) \exp\{-\phi_t(x_{t-1} = s_i, x_t)\}. \tag{9}$$

This enables us to compute the following quantities efficiently in $O(k^2 N)$ time: (I) the normalization constant $Z = \sum_{x_n \in S} V_N(x_N)$. (II) The marginal $P_N(x_N) = \frac{V_N(x_N)}{Z}$. (III) The conditionals $P_t(x_t|x_{t+1}) = \frac{V_t(x_t)\exp\{-\phi_{t+1}(x_t,x_{t+1})\}}{\sum_{x_t \in S} V_t(x_t)\exp\{-\phi_{t+1}(x_t,x_{t+1})\}}$. (IV) The marginals $P_t(x_t) = \sum_{\vec{x}/x_t} P(\vec{x}) = \sum_{x_N} ... \sum_{x_{t+1}} P(x_t|x_{t+1})...P(x_{N-1}|x_N)P(x_N)$.

This algorithm can be modified – by starting at x_{N+1} and working backwards – to compute $\pi_0(x_0)$ and the conditionals $P_t(x_{t+1}|x_t)$.

In addition, we can compute the expectation $\sum_{\vec{x}} P(\vec{x})h(\vec{x})$ efficiently provided $h(\vec{x})$ has specific forms. For example, if $h(\vec{x}) = \sum_{t=0}^{N} h_t(x_t)$, then we simply modify the DP sum rule to be $\overline{V}_t(x_t) = \sum_{x_{t+1} \in S} \overline{V}_{t+1}(x_{t+1}) \exp\{-\phi_{t+1}(x_{t+1},$ Alternatively, if $h(\vec{x}) = \sum_{t=0}^{N-1} h_h(x_t, x_{t+1})$ then we set $\overline{V}_t(x_t) = \sum_{x_{t+1} \in S} \overline{V}_{t+1}(x_{t+1}) \exp\{-\phi_{t+1}(x_{t+1}, x_t)\} h_t(x_t, x_{t+1})$.

Note that computations for learning can be done more easily of the distributions are expressed in terms of directed graphs $P(\vec{x}) = P_0(x_0) \prod_{t=0}^{N-1} P_t(x_{t+1}|x_t)$. This is because we can learn the conditional distributions $P(x_{t+1}|x_t)$ directly – the normalization constant of these is easy to compute.

This can be used to convert an undirected graphical model (provided it is defined on a graph without closed loops) into a conditional distribution.

What if you do not have a tree structure (i.e., you have closed loops)? There is an approach called *junction trees* which shows that you can transform any probability distribution on a graph into a probability distribution on a tree by enhancing the variables. The basic idea is triangulation (Lauritzen and Spiegelhalter). But this, while useful, is limited because the resulting trees can be enormous if the original graphs contain many closed loops.

3 Examples of ML learning

Consider a simple model with $P(x_1, x_2|\vec{\lambda}) = \frac{1}{Z[\lambda]} \exp\{\vec{\lambda} \cdot \vec{\phi}(\vec{x})\}$. Suppose these take states $x_i \in \{s_1, ..., s_M\}$. Then the potentials are $\delta_{x_1,s_a}\delta_{x_2,s_b}$ with parameters $\lambda_{a,b}$ for $a, b \in \{s_1, ..., s_N\}$. Hence:

$$P(x_1, x_2) = \frac{1}{Z[\lambda]} \exp\{\sum_{a,b=1}^{M} \lambda_{ab}\delta_{x_1,s_a}\delta_{x_2,s_b}\},$$

$$\text{where } Z[\vec{\lambda}] = \sum_{x_1,x_2} \exp\{\sum_{a,b=1}^{M} \lambda_{ab}\delta_{x_1,s_a}\delta_{x_2,s_b}\} = \sum_{a,b=1}^{M} \exp\{\lambda_{ab}\}. \tag{10}$$

The ML equation $(1/N)\sum_{i=1}^{N} \vec{\phi}(x_1^i, x_2^i) = \sum_{(x_1,x_2)} \vec{\phi}(x_1, x_2)P(x_1, x_2 : \vec{\lambda})$. We can compute the right hand side by taking $\frac{\partial}{\partial \vec{\lambda}} \log Z[\vec{\lambda}]$. This gives $\frac{\partial \log Z[\vec{\lambda}]}{\partial \lambda_{ab}} = \frac{\exp\{\lambda_{ab}\}}{\sum_{cd}\exp\{\lambda_{cd}\}}$. The statistics are $n_{ab} = \sum_{i=1}^{N} \delta_{x_1,s_a}\delta_{x_2,s_b}$. This can be solved by setting $\lambda_{cd} = \log n_{cd}\ \forall c, d$ (note there is a scaling ambiguity – $\vec{\lambda} \mapsto \vec{\lambda} + K\vec{e}$, where K is any constant and $\vec{e} = (1, ..., 1)$.