Learning Probalities, K-means, and Boltzmann

Alan Yuille and Dan Kersten

May 6, 2015

Learning Probability Distributions

- How do we learn these probability distributions? Can we learn them in ways that are *neuronally plausible*? I.e. can be learnt by local operations and ideally related to Hebbs rule for which there is some empirical evidence (Kandel and Apylsia).
- Learning can be formulated within Bayes Decision Theory with priors and loss functions. But the importance of priors is small unless there are small amounts of training data because the likelihood terms are usually dominant unless there is limited data (see T. Griffiths et al. MIT Press. 2024).
- To apply BDT to learn probabilities requires parameterizing the distribution and treatung the parameters as variables to be estimated (i.e., the decision rule is to estimate the parameters).
- Learning can be done in two modes. The first mode uses all the training data at the same time. The second mode trains incrementally using data points sequentially, which is called *online learning*. The first mode used to be used *batch learning* by neural network researchers, but now batch learning means online learning by selecting subbatches of the data. Online methods are more neuronally plausible.

Learning Probability Distributions without Hidden Variables

- To learn a probability distribution $P(x|\alpha)$ from data $\mathcal{X} = (x_1, ..., x_n)$, we use BDT to estimate the parameters α from the data. The simplest BDT criterion is maximum likelihood (ML):

$$\alpha^* = \arg\max P(\mathcal{X}|\alpha) = \prod_{i=1}^{n} P(x_i|\alpha) = \arg\min\{-\sum_{i=1}^{n} \log P(x_i|\alpha)\}$$

.

- If the distribution $P(x|\alpha)$ is a Gaussian, with mean μ and variance σ^2, then MLE yields the standard estimates $\mu^* = \frac{1}{n}\sum_{i=1}^{n} x_i$ and $.\sigma^{2,*} = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu^*)^2$. The is batch learning with all the data used at the same time.
- Online learning – AKA stochastic gradient descent – starts by initializing α_0. Then proceed as follows. At time t with parameter α_t, select a data point x_t at random, and update $\alpha_{t+1} = \alpha_t + \zeta_t \frac{\partial \log P(x_t|\alpha_t)}{\partial \alpha}$ where ζ_t is the *learning rate* (a small term which can decrease slowly with t).
- Theoretical properties of online learning – Robbins-Monroe theory –show that online learning is more likely to estimate the correct parameters α (and not get stcuk in a local minimum).

Learning Probability Distributions without Hidden Variables

- If we use a prior, batch learning tries to maximize the posterior $P(\alpha|\mathcal{X}) \propto P(\mathcal{X}|\alpha)P(\alpha)$, which is equivalent to minimizing $-\sum_{i=1}^{n} \log P(x_i|\alpha) - \log P(\alpha)$, which is the sum of n likelihood terms and one prior term.
- Hence the prior is usually unimportant unless there is little data (Griffith et al. MIT Press. 2024).
- Loss function are sometimes necessary, particularly for learning conditional distributions (see next slide).
- Selecting the functional form of the probability model $P(x|\alpha)$ is beyond the scope of the course. Model selection is one strategy for doing this – search over the space of models to find the one which best explains the data.

Learning Probability Distributions without Hidden Variables

- MLE can an alternative interpretation which relates to cross entropy loss. This capture the intuition that we want to learn a probability for the data.
- From training data (X) we specify a non-parametric distribution for the data $P_{\mathcal{X}}(x) = \frac{1}{N}\sum_{n=1}^{N}\delta(x - x_n)$.
- We train the algorithm to minimize Kullback-Leiber divergence between $P(x|\alpha)$ and $P_{\mathcal{X}}(x)$ specified by

$$\sum_{x} P_{\mathcal{X}}(x) \log \frac{P_{\mathcal{X}}(x)}{P(x:\alpha)}$$

.

- This corresponds to MLE.

Learning Conditional Probability Distributions without Hidden Variables

- The same approach can be used for learning a conditional distribution $P(y|x, \alpha)$ (this includes many types of deep networks) from training data $\mathcal{X} = \{(x_1, y_1)..., (x_n, y_n)\}$.
- We use the maximum likelihood estimator $\alpha^* = \prod_{i=1}^{n} P(y_i|x_i, \alpha) = \arg\min\{-\sum_{i=1}^{n} \log P(y_i|x_i, \alpha)\}$.
- Online learning – stochastic gradient descent – is typically used (it is impractical to use all the training data in one batch).
- For neural networks the parameters are the weights of the networks. Different neural architectures – AlexNet, ResNet, Transformers, etc. – specify different models.
- The comments about priors and loss functions still apply. Neural networks for tasks like edge detection require loss functions to penalize failure to detect an edge.

Learning Probability Distributions with Hidden Variables

- We now consider learning models with hidden, latent, or missing variables, which are variables that are not specified in the training data and need to be inferred.
- Hidden, latent, or missing variables arise in many disciplines and hence there are different names for the same underlying concept. For example, Statisticians found that statistical tables often have gaps – missing data – which need to be inferred. Engineers used Hidden Markov Models (HMMs) for speech recognition where "hidden" referred to variables that were not specified by training data.
- Statisticians formulated a general theory for dealing with hidden variables and an algorithm called *Expectation Maximization* (EM) for learning in these situations. This approach was re-formulated by AI researchers (Neal and Hinton) using a free energy formulations.

Learning Probability Distributions with Hidden Variables

- K-means is an important example of the EM algorithm for learning with hidden variables. It is a method for clustering data into different classes and is used for unsupervised learning (and neural networks, particularly transformers, often contain k-means as subcomponents).
- We first describe the *hard* version of k-means, and then the *soft* version.
- Hard and soft k-means can be implemented in a neural network where some nodes represent the inputs x and there are k output nodes which compete to explain the data (Hertz et al. Introduction to the Theory of Neural Computation. 1991).
- Soft K-means is a special case of the EM algorithm where the data is generated by a mixture of Gaussians.

K-means (1)

- The input to K-means is a set f unlabeled data: $D = \{x_1, ..., x_n\}$. The goal is to decompose it into disjoint classes $w_1, ..., w_k$ where k is known. The basic assumption is that the data D is clustered round (unknown) mean values $m_1, ..., m_k$.
- We defines an association variable V_{ia}. $V_{ia} = 1$ if datapoint x_i is associated to mean m_a and $V_{ia} = 0$ otherwise. we have the constraint $\sum_a V_{ia} = 1$ for all i (i.e. each datapoint is assigned to a single mean). This gives a decomposition of the data. $D_a = \{i : V_{ia} = 1\}$ is the set of datapoints associated to mean m_a. The set $D = \bigcup_a D_a$ is the set of all datapoints. $D_a \bigcap D_b = \phi$ for all $a \neq b$, where ϕ is the empty set.
- We defines a goodness of fit:

$$E(\{V\}, \{m\}) = \sum_{i=1}^{n} \sum_{a=1}^{k} V_{ia}(x_i - m_a)^2 = \sum_{a=1}^{k} \sum_{x \in D_a} (x - m_a)^2 \qquad (1)$$

- The goal of the k-means algorithm is to minimize $E(\{V\}, \{m\})$ with respect to $\{V\}$ and $\{m\}$. $E(.,.)$ is a non-convex function and no known algorithm can find its global miminum. But k-means converges to a local minimum.

K-means (2)

- **The k-means algorithm**
- 1. Initialize a partition $\{D_a^0 : a = 1 \text{ to k}\}$ of the data. (I.e. randomly partition the datapoints – or use K++).
- 2. Compute the mean of each cluster D_a, $m_a = \frac{1}{|D_a|} \sum_{x \in D_a} x$.
- 3. For i=1 to n, compute $d_a(x_i) = |x_i - m_a|^2$. Assign x_i to cluster D_{a^*} s.t. $a^* = \arg\min\{d_a(x_i), ..., d_k(x_i)\}$
- 4. Repeat steps 2 & 3 until convergence.
- This will converge to a minimum of the energy function because steps 2 and 3 each decrease the energy function (or stop if the algorithm is at a local minimum). This will divide the space into disjoint regions.
- k-means can be formulated in terms of the assignment variable. At step 2, $m_a = \frac{1}{\sum_i V_{ia}} \sum_i V_{ia} x_i$. At step 3. $V_{ia} = 1$ if $|x_i - m_a|^2 = \min_b |x_i - m_b|^2$ and $V_{ia} = 0$ otherwise.

The online version of k-means minimizes the energy function $E(\{V\},\{m\}) = \sum_{i=1}^{n}\sum_{a=1}^{k} V_{ia}(x_i - m_a)^2 = \sum_{a=1}^{k}\sum_{x\in D_a}(x - m_a)^2$ by steepest descent.

This has a neural network formulation. We have k output neurons which have receptive fields $m_{k,a}$ which represent the means. We have an input vector x_a (a specifies the component). The output units receive input $\sum_a m_{k,a}x_a$ and we impose inhibition between them so that only the unit with biggest input fires – i.e. unit k^* fires if $\sum_a m_{k^*,a}x_a \geq \sum_a m_{k,a}x_a$ for all k. This means that the input is matched to the k^* unit.

We update the weight $m_{k^*,a}$ by differentiating $\sum_a (x_a - m_{k^*,a})^2$ with respect to $m_{k^*,a}$. I.e. $m_{k^*,a} \mapsto m_{k^*,a} - \zeta 2(m_{k^*,a} - x_a)$. The weights of the other neurons remain fixed,

This is stochastic gradient descent on $E(\{V\},\{m\})$. At time t we select a new input x^t, we find which output unit has mean (i.e. weight) closest to x^t (we normalize the weights and the inputs, so minimizing the square distance is equivalent to maximizing the dot product) and update its weight, keeping the other weights fixed. (This relates to Fisher von Mises distributions, see later).

Soft K-means. Mixture of Gaussians. (1)

- A "softer" version of k-means – which is an example of the Expectation-Maximization (EM) algorithm. Assign datapoints $\underline{x}_i$ to each cluster with probability $(P_1, \ldots, P_k)$
- 1. Initialize a partition of the datapoints.
- 2. For j=1 to n. Compute the probability that x_j belongs to ω_a. $P(\omega_a|x_j) = \frac{\exp -\frac{1}{2\sigma^2}(x_j - m_a)^2}{\sum_b \exp -\frac{1}{2\sigma^2}(x_j - m_b)^2}$.
- 3. Compute the mean for each cluster: $m_a = \sum_j x_j P(\omega_a|x_j)$
- 4 Repeat steps 2 & 3 until convergence.
- In this version the *hard-assign* variable V_{ia} is replaced by a *soft-assign* variable $P(\omega_a|x_j)$. Observe that $\sum_a P(\omega_a|x_j) = 1$. Also observe that the softness is controlled by σ^2. In the limit, as $\sigma^2 \mapsto 0$, the distribution $P(\omega_a|x_j)$ will become binary valued, and soft k-means will be the same as k-means.

Soft K-means. Mixture of Gaussians. (2)

- Soft k-means can be reformulated in terms of mixtures of Gaussians and the Expectation-Maximization (EM) algorithm.
- This assumes that the data is generated by a mixture of Gaussian distributions with means $\{m\}$ and variance $\sigma^2\mathbf{I}$.
  $P(x|\{V\},\{m\}) = \frac{1}{Z}\exp\{-\sum_{ia} V_{ia}\frac{||x_i - m_a||^2}{\sigma^2}\}$.
- This is equivalent to a mixture of Gaussians:
  $P(x|V,m) = \mathcal{N}(x : \sum_a V_{ia} m_a, \sigma^2)$, where the variable V identifies the mixture component (i.e. $V_{ia} = 1$ if datapoint x_i was generated by mixture a, and $\sum_a V_{ia} = 1 \ \ \forall i$).
- This requires imposing a prior $P(\{V\})$ on the assignment variable V. It is natural to choose a uniform distribution $P(V) = 1/Z$, where Z is the number of possible assignments of the datapoints to the means.

Soft K-means. Mixture of Gaussians. (3)

- This gives distributions $P(x, \{V\}|\{m\}) = P(x|\{V\}, \{m\})P(\{V\})$. This form enables us to use the EM algorithm (see later lecture). The basic idea is to do maximum likelihood estimation for the parameters $\{m\}$ from the *marginal distribution* $P(x|\{m\}) = \sum_{\{V\}} P(x, \{V\}|\{m\})$.
- EM is an iterative algorithm for doing this. It replaces the missing variables $\}$ by probabilities $(V)\}$ for their values (equivalent to $P(\omega_a|x_j)$ in the earlier slides). Then it estimates $\{m\}$ and $(V)\}$ alternatively (earlier slide).
- More generally, suppose we have model $P(x|h, \theta)$, where x is the observed data, h in the hidden variable, and θ are the parameters we want to estimate.
- Then we specify a prior $P(h)$ and can formulate the problem as MLE on $P(x|\theta) = \sum_h (P(x|h, \theta)P(h)$. EM specifies update rules for estimating θ and $Q(h)$ in alteration ($Q(.)$ is a probability). EM can be derived by showing that minimizing $P(x|\theta)$ with respect to x is equivalent to minimizing a free energy function of x and $Q(h)$ with respect to each alternatively.

Mixture of Von Mises Fisher

- A related example arises if we require that the data has unit norm $|x_i| = 1, \forall i$ and hence lies on the unit sphere. This can be used to deal with the scaling of images. Recall $I(x) \mapsto aI(x) + b$, where a is the scale (contrast) and b is the background. We set $b = 0$ and normalize the images by $I(x) \mapsto \frac{I(x)}{|I(x)|}$ (so that $I(x)$ has unit norm).
- The Von Mises Fisher distribution is $P(x)|k, \lambda_k) = \frac{\exp\{\lambda_k m_k \cdot x\}}{Z(\lambda_k)}$. Here $x| = |m_k| = 1$, and σ_k is a positive constant.
- Note that this distribution is related to the Gaussian distribution (with spherical covariance). The exponent of this Gaussian is $-\frac{(x-m_k)^2}{2\sigma^2}$. If we require $|x| = |m_k| = 1$, then the exponent becomes $\frac{(x \cdot m_k - 1)}{\sigma^2}$. So if we identify λ_k with $1/\sigma_k^2$ we recover Von Mises Fisher. In other words, Von Mises Fisher is the natural way to re-formulate mixtures of Gaussians for data that lies on the unit sphere.

Boltzmann Machine: The Gibbs Distribution

- The probability distribution for N neurons $\vec{S} = (s_1, ..., s_N)$, where each s_i takes value 0 or 1, is defined by a Gibbs distribution with energy $E(\vec{S}) = \frac{-1}{2} \sum_{ij} \omega_{ij} s_i s_j$ and distribution:

$$P(\vec{S}) = \frac{1}{Z} \exp\{-E(\vec{S})/T\}. \tag{2}$$

- State configurations $\vec{S}$ with low energy $E(\vec{S})$ will correspond to high probabilities $P(\vec{S})$. Z is specified by the normalization condition $\sum_{\vec{S}} P(\vec{S}) = 1$, by $Z = \sum_{\vec{S}} \exp\{-E(\vec{S})/T\}$. The ω_{ij} are the weights of the distribution (like weights in a neural network) and are symmetric $\omega_{ij} = \omega_{ji}\ \forall i, j$ with $\omega_{ii} = 0,\ \forall i$.

- The "temperature" T controls the "sharpness" of the distribution. For very small T, the distribution is strongly peaked about $\vec{S}^* = \arg\min_{\vec{S}} E(\vec{S})$. As T increases, the distribution becomes less peaked as T becomes large ($T \mapsto \infty$) all states become equally likely. Intuitively, T is similar to the variance.

Boltzmann Machine: Inference

- The inference task is to compute, or estimate, the most probably state(s) $\vec{S}^* = \arg\max_{\vec{S}} P(\vec{S}) = \arg\min_{\vec{S}} E(\vec{S})$. But this is impossible because $\vec{S}$ takes 2^N possible states and so we cannot simply evaluate the probability of every state and find the maximum, and similarly we cannot compute Z. (But there are a few special cases where computing $\vec{S}^*$ is possible).
- We have discussed two types of algorithm that can get approximate estimates of $\vec{S}^*$: (I) Gibbs Sampling. (II) Mean Field Theory.
- In this lecture we will be using Gibbs sampling. Recall that this: (i) initializes the states $\vec{S}$ randomly, (ii) selects a node i at random, (iii) samples s_i from the conditional distribution $P(s_i|\vec{S}/i) = \frac{\exp s_i\{\sum_j w_{ij} s_j\}}{1+\exp\{\sum_j w_{ij} s_j\}}$, and (iv) repeat (ii) and (iii).
- It can be shown that Gibbs sampling converges to samples $\vec{S}$ from $P(\vec{S})$. This implies that the final states will have high probabilities. So if we have a set $\{\vec{S}^n : n = 1, ..., N\}$ from $P(\vec{S})$ then they are likely to have high probabilities $\{P(\vec{S}^n) : n = 1, ..., N\}$ and be close to $\vec{S}^*$. Importantly, for this lecture, we can approximate the expected statistics of $P(\vec{S})$ by $< s_i s_j >= \sum_{\vec{S}} s_i s_j P(\vec{S}) \approx \sum_{n=1}^{N} s_i^n s_j^n$.

Boltzmann Machine: Learning

- Divide the nodes into two classes $\mathcal{V}_o$ and $\mathcal{V}_h$, which are the observed (input) and hidden nodes respectively. $\vec{S}_o$ and $\vec{S}_h$ denote the states of the observed and the hidden nodes respectively. The components of $\vec{S}_o$ and $\vec{S}_h$ are $\{S_i : i \in \mathcal{V}_o\}$ and $\{S_i : i \in \mathcal{V}_h\}$ respectively. $\vec{S} = (\vec{S}_o, \vec{S}_h)$.
- We re-express the distribution over the states as:

$$P(\vec{S}_o, \vec{S}_h) = \frac{1}{Z} \exp\{-E(\vec{S})/T\}. \tag{3}$$

  The marginal distribution over the observed nodes is

$$P(\vec{S}_o) = \sum_{\vec{S}_h} \frac{1}{Z} \exp\{-E(\vec{S})/T\}. \tag{4}$$

- We estimate a distribution $R(\vec{S}_0)$ of the observed nodes (from the observed data $\{\vec{S}_o^n : n = 1, ..., N\}$ where N are the number of training examples). *The goal of learning is to adjust the weights* $\vec{\omega}$ of the model (i.e. the $\{\omega_{ij}\}$) so that the marginal distribution $P(\vec{S}_o)$ of the model is as similar as possible to the observed model $R(\vec{S}_0)$.
- This requires specifying a similarity criterion which is chosen to be the Kullback-Leibler divergence:

$$KL(\vec{w}) = \sum R(\vec{S}_o) \log \frac{R(\vec{S}_o)}{P(\vec{S}_o)} \tag{5}$$

Boltzmann Machine: The Learning Rule

- The Boltzmann Machine adjusts the weights by the iterative update rule:

$$w_{ij} \mapsto w_{ij} + \Delta w_{ij} \tag{6}$$

$$\Delta w_{ij} = -\delta \frac{\partial KL(\vec{w})}{\omega_{ij}} \tag{7}$$

$$\Delta w_{ij} = -\frac{\delta}{T}\{< S_i S_j >_{clamped} - < S_i S_j >\} \tag{8}$$

- Here δ is a small positive constant. The derivation of the update rule is given in later slides (so is how to compute the update rule).
- $< S_i S_j >_{clamped}$ and $< S_i S_j >$ are the expectation (e.g., correlation) between the state variables S_i, S_j when the data is generated by the *clamped* distribution $R(\vec{S}_o)P(\vec{S}_h|\vec{S}_o)$ and by the distribution $P(\vec{S}_o, \vec{S}_h)$ respectively.
- I.e. $< S_i S_j >= \sum_{\vec{S}} S_i S_j P(\vec{S})$. The conditional distribution $P(\vec{S}_h|\vec{S}_o)$ is the distribution over the hidden states conditioned on the observed states. So it is given by $P(\vec{S}_h|\vec{S}_o) = P(\vec{S}_h, \vec{S}_o)/P(\vec{S}_o)$.

Boltzmann Machine: Understanding the Learning Rule

- The learning rule, equation (8), has two components. The first term $< S_i S_j >_{clamped}$ is Hebbian and the second term $< S_i S_j >$ is anti-Hebbian (because of the sign). This is a balance between the activity of the model when it is driven by input data (i.e. clamped) and when it is driven by itself. A wild speculation is that the Hebbian learning is done when you are awake, hence exposed to external stimuli, while the anti-Hebbian learning is done when you are asleep with your eyes shut but, by sampling from $P(\vec{S}_o|\vec{S}_h)$ you are creating images, or dreaming.

- The algorithm will convergence when the model accurately fits the data, i.e.. when $< S_i S_j >_{clamped} = < S_i S_j >$ and the right hand side of the update rule, equation (8), is zero.

- What is the observed distribution $R(\vec{S}_o)$? We do not know $R(\vec{S}_o)$ exactly and so we approximate it by the *training data* $\{\vec{S}_o^{\mu}; \mu = 1, ..., N\}$. This is equivalent to assuming that

$$R(\vec{S}) = \frac{1}{N} \sum_{\mu=1}^{N} \delta(\vec{S}_o - \vec{S}_o^{\mu}) \tag{9}$$

.

Estimating the $< S_i S_j >$

- The Boltzmann Machine requires computing $< S_i S_j >_{clampaed}$ and $< S_i S_j >$. This is done by Gibbs sampling (earlier lectures). .
- By performing Gibbs sampling multiple times on the distribution $P(\vec{S}_o, \vec{S}_h)$ we obtain M samples $\underline{\vec{S}}^1, ..., \underline{\vec{S}}^M$. Then we can approximate $< S_i S_j >$ by:

$$< S_i S_j > \approx \frac{1}{M} \sum_{a=1}^{M} \underline{S}_i^a \underline{S}_j^a \tag{10}$$

- Similarly we can obtain samples from $R(\vec{S}_o)P(\vec{S}_h|\vec{S}_o)$ (the clamped case) by first generating samples $\underline{\vec{S}_o}^1, ..., \underline{\vec{S}_o}^M$ from $R(\vec{S}_0)$ and then converting them to samples

$$\underline{\vec{S}}^1, ..., \underline{\vec{S}}^M \tag{11}$$

where $\underline{\vec{S}} = (\underline{\vec{S}_o}^i, \underline{\vec{S}_h}^i)$, and $\underline{\vec{S}_h}^i$ is a random sample from $P(\vec{S}_h|\vec{S}_o)$, again performed by Gibbs sampling.

- How do we sample from $R(\vec{S}_o)$? Recall that we only know samples $\{\vec{S}_o^\mu; \mu = 1, ..., N\}$ (the training data). Hence sampling from $R(\vec{S}_o)$ reduces to selecting one of the training examples at random.
- Gibbs sampling is not a very effective algorithm. So Boltzmann machines are hard to use in practice (with extra ingredients).

Derivation of the BM update rule (I)

- To justify the learning rule, equation (8), we need to take the derivative of the cost function $\partial KL(\vec{\omega})/\partial \omega_{ij}$.

$$\frac{\partial KL(\vec{w})}{\partial \omega_i j} = -\sum_{\vec{S}_o} \frac{R(\vec{S}_o)}{P(\vec{S}_o)} \frac{\partial P(\vec{S}_o)}{\partial \omega_{ij}} \tag{12}$$

- Expressing $P(\vec{S}_o) = \frac{1}{Z} \sum_{\vec{S}_h} \exp\{-E(\vec{S})/T\}$, we can express $\frac{\partial P(\vec{S}_o)}{\partial \omega_{ij}}$ in two terms:

$$\frac{1}{Z} \frac{\partial}{\partial \omega_{ij}} \sum_{\vec{S}_h} \exp\{-E(\vec{S})/T\} - \frac{1}{Z} \sum_{\vec{S}_h} \exp\{-E(\vec{S})/T)\} \frac{\partial \log Z}{\partial \omega_{ij}} \tag{13}$$

- This can be re-expressed as:

$$\frac{-1}{T} \sum_{\vec{S}_h} S_i S_j P(\vec{S}) + \{\sum_{\vec{S}_h} P(\vec{S}) \frac{1}{T} \sum_{\vec{S}} S_i S_j P(\vec{S})\} \tag{14}$$

Derivation of the BM update rule (II)

- Hence we can compute:

$$\frac{\partial P(\vec{S}_o)}{\partial \omega_{ij}} = \frac{-1}{T} \sum_{\vec{S}_h} S_i S_j P(\vec{S}) + P(\vec{S}_o) \frac{1}{T} \sum_{\vec{S}} S_i S_j P(\vec{S}) \tag{15}$$

- Substituting equation (15) into equation (12) yields

$$\frac{\partial KL(\vec{w})}{\partial \omega_i j} = \frac{1}{T} \sum_{\vec{S}_h, \vec{S}_o} S_i S_j \frac{P(\vec{S})}{P(S_o)} R(\vec{S}_o) - \frac{1}{T} \{ \sum_{\vec{S}_o} R(\vec{S}_o) \} \sum_{\vec{S}} S_i S_j P(\vec{S}) \tag{16}$$

- Which can be simplified to give:

$$\frac{\partial KL(\vec{w})}{\partial \omega_i j} = \frac{1}{T} \sum_{\vec{S}} S_i S_j P(\vec{S}_h | \vec{S}_o) R(\vec{S}_o) - \frac{1}{T} \sum_{\vec{S}} S_i S_j P(\vec{S}) \tag{17}$$

- Note this derivation requires $\partial \log Z / \partial w_{ij} = \sum_{\vec{S}} S_i S_j P(\vec{S})$.

Boltzmann Machine is Maximum Likelihood Learning

- The Kullback-Leibler criterion, equation (5), can be expressed as:

$$KL(\vec{\omega}) = \sum_{\vec{s}} R(\vec{S}_o) \log R(\vec{S}_o) - \sum_{\vec{s}} R(\vec{S}_o) \log P(\vec{S}_o) \tag{18}$$

- Only the second term depends on $\vec{\omega}$ so we can ignore the first (since we want to minimize $KL(\vec{\omega})$ with respect to $\vec{\omega}$).

- Using the expression for $R(\vec{S}_o)$ in terms of the training data, equation (9), we can express the second term as:

$$-\frac{1}{N} \sum_{\vec{S}_o} \frac{1}{N} \sum_{a=1}^{N} \delta(\vec{S}_o - \vec{S}_o^a) \log P(\vec{S}_o) \tag{19}$$

$$-\frac{1}{N} \frac{1}{N} \sum_{a=1}^{N} \log P(\vec{S}_o^a) \tag{20}$$

- This is precisely, the Maximum Likelihood criterion for estimating the parameters of the distribution $P(\vec{S}_o)$. This shows that Maximum Likelihood is a good strategy to learn a distribution *even if we do not know the correct form of the distribution*. We are simply finding the best fit model.

Boltzmann Machine learns by Expectation-Maximization

- The Boltzmann Machine (BM) learning is a special case of the Expectation-Maximization (EM) algorithm. This algorithm can be applied to any learning problem where some variables are unobservable.
- For the BM, the distribution is $P(\vec{S}_o, \vec{S}_h; \omega)$ with observed data $\{\vec{S}_o^n : n = 1, ..., N\}$. We do not know the $\{\vec{S}_h^n : n = 1, ..., N\}$, so the $\vec{S}_h$ are *hidden, missing, or latent* variables.
- In theory we can compute the marginal distribution $P(\vec{S}_o; \omega) = \sum_{\vec{S}_h} P(\vec{S}_o, \vec{S}_h; \omega)$. Then we can learn the weights $\{\omega_{ij}\}$ by Maximum Likelihood: minimizing

$$-\sum_{n=1}^{N} \log P(\vec{S}_o; \omega), \quad \text{w.r.t. } \omega.$$

- The problem is that we cannot compute $P(\vec{S}_o; \omega)$ explicitly. This is where we need EM.

BM and EM: part 1

- We define a new (unknown) distributions $Q^n(\vec{S}_h) = \prod_{i=1}^m q_i^n(S_h^i)$ $n = 1, ..N$, where the $\{S_h^i : i = 1, .., m\}$ are the components of the hidden variables $\vec{S}_h$.
- We define a free energy:

$$\mathcal{F}(Q, \omega) = -\sum_{n=1}^{N} \log P(\vec{S}_o^n; \omega) + \sum_{n=1}^{N} \sum_{\vec{S}_h^n} Q^n(\vec{S}_h^n) \log \frac{Q^n(\vec{S}_h^n)}{P(\vec{S}_h^n | \vec{S}_o^n; \omega)}.$$

- This has two important properties. Firstly, we can minimize $\mathcal{F}(Q, \omega)$ with respect to each $Q^n(.)$ to obtain $Q^n(\vec{S}_h^n) = P(\vec{S}_h^n | \vec{S}_o^n; \omega)$. Substituting this value of $Q^n(.)$ back into $\mathcal{F}(Q, \omega)$ yields $-\sum_{n=1}^{N} \log P(\vec{S}_o^n; \omega)$.
- Therefore minimizing $\mathcal{F}(Q, \omega)$ with respect to Q and ω is equivalent to performing ML on $P(\vec{S}_o; \omega)$.
- This follows from the facts that $\sum_{\vec{S}} Q(\vec{S}) \log \frac{Q(\vec{S})}{P(\vec{S})} \geq 0$ and $= 0$ only when $Q(\vec{S}) = P(\vec{S})$.

BM and EM: part 2

- The second property is that we can minimize $\mathcal{F}(Q, \omega)$ by alternatively minimizing with respect to Q and to ω. This is the EM algorithm.
- Minimizing w.r.t. $Q(.)$ gives $Q^n(\vec{S}_h^n) = P(\vec{S}_h^n | \vec{S}_o^n; \omega)$.
- Minimizing w.r.t. ω gives:

$$\omega_{ij} = \arg\min \sum_{n=1}^{N} Q^n(\vec{S}_h^n) \log P(\vec{S}; \omega) = \arg\min -\{\sum_{n=1}^{N} Q^n(\vec{S}_h^n) E(\vec{S}) - \log Z(\omega)\}.$$

- This exploits $P(\vec{S}_h | \vec{S}_o; \omega) P(\vec{S}_o; \omega) = P(\vec{S}_h, \vec{S}_o; \omega)$.
- For the BM, these minimizations reduce to the BM learning rule (after some algebra). Gibbs sampling is needed to perform each step. Note: there is no guarantee that the EM algorithm will converge to the global optimum (i.e. to the real ML estimate).

The Restricted Boltzmann Machine

- RBMs are a special case of Boltmann Machines where there are no weights connecting the hidden nodes to each other with energy:

$$E(\vec{S}) = \sum_{i \in \mathcal{V}_o,\ j \in \mathcal{V}_h} \omega_{ij} S_i S_j. \tag{21}$$

- The conditional distributions $P(\vec{S}_h|\vec{S}_o)$ and $P(\vec{S}_o|\vec{S}_h)$ can both be factorized:

$$P(\vec{S}_o|\vec{S}_h) = \prod_{i \in \mathcal{V}_o} P(S_i|\vec{S}_h), \quad P(\vec{S}_h|\vec{S}_o) = \prod_{j \in \mathcal{V}_h} P(S_j|\vec{S}_o) \tag{22}$$

- For $i \in \mathcal{V}_o$, $P(S_i|\vec{S}_h) = \frac{1}{Z_i} \exp\{-(1/T) S_i (\sum_{j \in \mathcal{V}_h} \omega_{ij} S_j)\}$. Z_i is the normalization constant $Z_i = \sum_{S_i \in \{0,1\}} \exp\{-(1/T) S_i (\sum_{j \in \mathcal{V}_h} \omega_{ij} S_j)\}$ – and similarly for $P(S_j|\vec{S}_o)$ for $j \in \mathcal{V}_h$.

- These factorization means that we can sample from $P(\vec{S}_o|\vec{S}_h)$ and $P(\vec{S}_h|\vec{S}_o)$ very rapidly (e.g., by sampling from $P(S_i|\vec{S}_h)$). This makes learning fast and practical. Estimating $< S_i S_j >_{clamped}$ requires sampling from $P(\vec{S}_h|\vec{S}_o)$, which is very fast. Estimating $< S_i S_j >$, requires sampling from $P(\vec{S}_o, \vec{S}_h)$ by alternatively sampling from $P(\vec{S}_o|\vec{S}_h)$ and $P(\vec{S}_h|\vec{S}_o)$. This must be done multiple times until convergence (but it is much faster than Gibbs sampling).

- RBMs are too restricted to anything useful. But Hinton (2006) suggested stacking them on top of each other to create a Deep Network.