Energy-Based Models

Muchang Bahng

Spring 2023

Contents

1 Energy Models

Now this is the first time we talk about unsupervised learning. Note from our graphical models notes that for unsupervised tasks, we have estimated the density of complex distributions by factoring them in *graphical models*, e.g. Bayesian networks or Markov random fields. We will extend this into deep learning architectures.[1]

Note that many of the theories behind energy models are very old (from the 1980s) and established, especially in the prevalence of the *Boltzmann distribution* in statistical mechanics. All of them essentially rely on the fact that we can model a probability density as

$$p(x) = \frac{e^{-E(x)}}{Z} \tag{1}$$

for some function $E : \mathbb{R}^n \to \mathbb{R}$, which we call the *negative energy*, and Z is what we call the **partition function**. We have seen from the Hammersley-Clifford theorem that we can model a joint probability distribution on an undirected graph with the product of potential functions on the max cliques.[2] Therefore, we will take advantage of this in the specific instance of MRFs that are *bipartite graphs*, i.e. a non-feedforward (since its cyclic) 2-layer neural network. We will talk about RBMs, deep belief networks, and hopfield networks. Diffusion models, which can also be considered an energy model, will be talked separately.

The most straightforward application is that we can just have a neural network approximate this function E_θ, which gives us a parameterized family of distributions $p_\theta(x) = \frac{1}{Z} e^{-E_\theta(x)}$ that would hopefully approximate the true distribution $p^*(x)$. This is called an **energy based model (EBM)**.

1.1 Training with MCMC

Therefore, we could like to maximize the log-likelihood (which gets rid of the partition function term), giving us

$$\begin{aligned} \underset{\theta}{\operatorname{argmax}} \, \mathbb{E}_{x\sim p^*}\big[\log p_\theta(x)\big] &= \underset{\theta}{\operatorname{argmax}} \, \mathbb{E}_{x\sim p^*}\big[-E_\theta(x)\big] - \mathbb{E}_{x\sim p^*}\big[\log Z_\theta\big] && (2) \\ &\approx \underset{\theta}{\operatorname{argmin}} \sum_{i=1}^{N} E_\theta(x^{(i)}) + \sum_{i=1}^{N} \log Z_\theta && (3) \end{aligned}$$

Note that even though Z_θ is constantly 1 with respect to x, the actual value of the integral will change with respect to θ. Therefore this term also contributes to the argmax. Focusing on a single sample, we attempt to compute the gradient of this. The first gradient $\nabla_\theta E_\theta(x)$ is easy by automatic differentiation since E is a neural net. However, the second term is quite tricky. But using some mathematical identities mentioned in [SK21], we have

[1] This is *not* to be confused with graph neural networks (GNNs), which are designed for tasks whose inputs are graphs.
[2] But finding max cliques is NP-hard?

$$\nabla_\theta \log Z_\theta = \nabla_\theta \log \int \exp(-E_\theta(x))dx \tag{4}$$
$$= \left(\int \exp(-E_\theta(x))dx\right)^{-1} \nabla_\theta \int \exp(-E_\theta(x))dx \tag{5}$$
$$= \left(\int \exp(-E_\theta(x))dx\right)^{-1} \int \nabla_\theta \exp(-E_\theta(x))dx \tag{6}$$
$$= \left(\int \exp(-E_\theta(x))dx\right)^{-1} \int \exp(-E_\theta(x))(-\nabla_\theta E_\theta(x))dx \tag{7}$$
$$= \int \left(\int \exp(-E_\theta(x))dx\right)^{-1} \exp(-E_\theta(x))(-\nabla_\theta E_\theta(x))dx \tag{8}$$
$$= \int \frac{\exp(-E_\theta(x))}{Z_\theta}(-\nabla_\theta E_\theta(x))dx \tag{9}$$
$$= \int p_\theta(x)(-\nabla_\theta E_\theta(x))dx \tag{10}$$
$$= \mathbb{E}_{x\sim p_\theta(x)}[-\nabla_\theta E_\theta(x)] \tag{11}$$

This gives us hope. Therefore, we can estimate the intractable gradient as an expectation of the gradient of the neural network with respect to the current estimate $p_\theta(x)$ (not the true p^*!).

$$\nabla_\theta \mathbb{E}_{x\sim p^*}[\log p_\theta(x)] = \mathbb{E}_{x\sim p^*}[\nabla_\theta \log p_\theta(x)] \tag{12}$$
$$\approx \sum_{i=1}^{N} -\nabla_\theta E_\theta(x) - \nabla_\theta \log Z_\theta \tag{13}$$
$$= \sum_{i=1}^{N} \Big\{ -\nabla_\theta E_\theta(x) + \mathbb{E}_{x\sim p_\theta(x)}[\nabla_\theta E_\theta(x)] \Big\} \tag{14}$$
$$\approx \sum_{i=1}^{N} \Big\{ -\nabla_\theta E_\theta(x) + \sum_{j=1}^{M} \nabla_\theta E_\theta(\tilde{x}) \Big\} \tag{15}$$

where the final step is from using a Monte Carlo sample with a size M batch of $\tilde{x}$ from $p_\theta(x)$. We can draw samples using MCMC, with Langevin dynamics MCMC being the most popular.

1.2 Boltzmann Machines

Okay, so we've learned to model an arbitrary distribution by approximating it with an energy model. While they are not explicitly build into energy models, it is possible to include them. One such method is through *Boltzmann machines*. Consider the graph $x_1, \dots, x_D$ which represents a random vector x for which we would like to model the probability distribution of.

What we can do is model the dependencies between these random elements with linear parameters W and b, which essentially gives us a Markov Random Field. Let's consider when x_i's are all Bernoulli, so $x \in \{0,1\}^D$, which are known as *Ising models* in statistical mechanics. By Hammersley-Clifford, we don't even need to specify the individual functions over the maximal cliques, and rather we can just specify the energy function $E(x)$ of the Boltzmann distribution that the MRF encodes. We parameterize $\theta = \{W, b\}$.

Example 1.1 (Bernoulli Pairwise Markov Random Fields)

We define it to capture the interactions between Bernoulli random variables x_i up to order 2.

$$p_\theta(x) = \frac{1}{Z}\exp(E(x)) = \frac{1}{Z}\exp\left(\sum_{ij\in E} x_i x_j W_{ij} + \sum_{i\in V} x_i b_i\right) = \frac{1}{Z}\exp\left(x^T W x + b^T x\right) \tag{16}$$

Now let's check its conditional distribution. Let x_{-k} denote the joint distribution of all random variables minus x_k.

$$\begin{aligned} p(x_k = 1 \mid x_{-k}) &= \frac{p(x_k = 1, x_{-k})}{p(x_{-k})} && (17)\\ &= \frac{p(x_k = 1, x_{-k})}{p(x_k = 0, x_{-k}) + p(x_k = 1, x_{-k})} && (18)\\ &= \frac{\exp\left(\sum_{kj\in E} x_j W_{kj} + x_k b_k\right)}{\exp(0) + \exp\left(\sum_{kj\in E} x_j W_{kj} + x_k b_k\right)} && (19)\\ &= \sigma\left\{ - b_k x_k - \sum_{kj\in E} x_j W_{kj}\right\} && (20) \end{aligned}$$

where the penultimate step comes from evaluating

$$\begin{aligned} p(x_k = 1, x_{-k}) &= \frac{1}{Z(\theta)}\exp\left(\sum_{ij\in E, k\neq i,j} x_i x_j W_{ij} + \sum_{ij\in E, k=i,j} x_i x_j W_{ij} + \sum_{i\in V, i\neq k} x_i b_i + x_k b_k\right) && (21)\\ &= \frac{1}{Z(\theta)}\exp\left(\sum_{ij\in E, k\neq i,j} x_i x_j W_{ij} + \sum_{kj\in E} x_j W_{kj} + \sum_{i\in V, i\neq k} x_i b_i + b_k\right) && (22)\\ p(x_k = 0, x_{-k}) &= \frac{1}{Z(\theta)}\exp\left(\sum_{ij\in E, k\neq i,j} x_i x_j W_{ij} + \sum_{i\in V, i\neq k} x_i b_i\right) && (23) \end{aligned}$$

and canceling out like terms in the numerator and denominator. This tells us that MRFs are related to logistic function.

Example 1.2 (Gaussian Markov Random Fields)

If we assume that $p_\theta(x)$ follows a multivariate Gaussian distribution, we have

$$p(x \mid \mu, \Sigma) = \frac{1}{Z}\exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right) \tag{24}$$

Since the Gaussian distribution represents at most second-order relationships, it automatically encodes a pairwise MRF. Therefore, we can rewrite

$$p(x) = \frac{1}{Z}\exp\left(-\frac{1}{2}x^T J x + g^T x\right) \tag{25}$$

where $J = \Sigma^{-1}$ and $\mu = J^{-1}g$.

However, this is still quite a limited model. For one, due to the linearity of the weight matrix, it always turns out that the probability of $x_k = 1$ is always given by a linear model (logistic regression) from the values of the other units. This family of distributions parameterized by $\theta = \{W, b\}$ may not be broad enough to capture the true $p(x)$. Therefore, we can add latent variables that can act similarly to hidden units in a MLP

and model higher-order interactions among the visible units. Just as the addition of hidden units to convert logistic regression into MLP results in the MLP being a universal approximator of functions, a Boltzmann machine with hidden units is not longer limited to modeling linear relationships between variables. Instead, the Boltzmann machine becomes a universal approximator of probability mass functions over discrete random variables.

Definition 1.1 (Boltzmann Machine)

The original **Boltzmann machine** has the energy function

$$E(v,h) = -v^T Rv - v^T Wh - h^T Sh - b^T v - c^T h \tag{26}$$

It can represent the undirected graph that has connections within the x, within the h, and between the x and h.

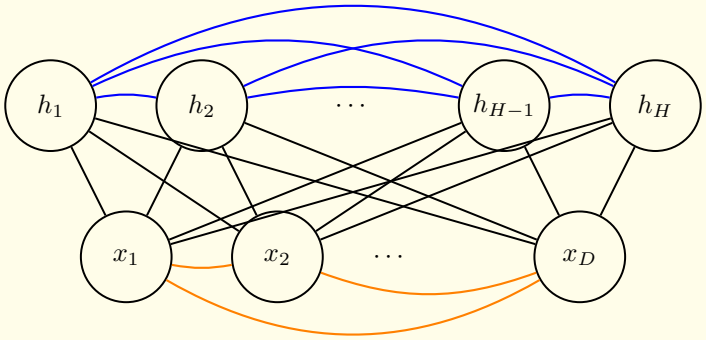

Figure 1: 2-layer undirected graph representing a Boltzmann machine.

Therefore, by adding latent variables and connecting everything together, this gives us a very flexible model that can capture a lot of distributions.

1.3 Restricted Boltzmann Machines

Unfortunately, there are problems with training this, and so the restricted Boltzmann machine allowed for efficient training. Therefore, we will limit ourselves to **pairwise MRFs**, which only capture dependencies between cliques of maximum size 2. We usually write x as the observed and z as the latent, but in the literature v and h are used, respectively. Now, if we put a restriction saying that there cannot be any intra-connections in the x and h, then we get the *restricted Boltzmann machine*, which has a slightly more resticted form of the energy function than the general BM.

Definition 1.2 (Restricted Boltzmann Machine)

The **restricted Boltzmann machine** has the energy function

$$E(v,h) = -v^T Wh - b^T v - c^T h \tag{27}$$

with connections only allowed between x_i's and h_j's, known as a **bipartite graph**, implying that the maximum clique length is 2. This model allows the elements of x to be dependent, but this architecture allows for *conditional independence*, and not just for x given h, but also h given x.

Therefore, we already have the extremely nice property that

$$p(x \mid h) = \prod_{k=1}^{D} p(x_k \mid h) \tag{28}$$

$$p(h \mid x) = \prod_{j=1}^{F} p(h_j \mid x) \tag{29}$$

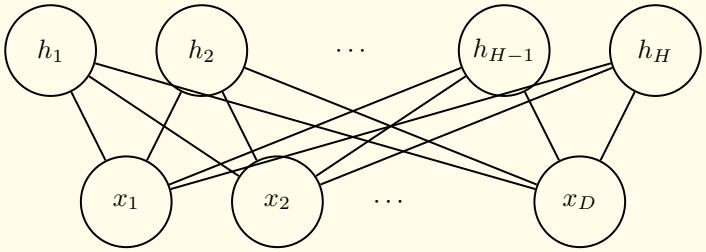

Figure 2: 2-layer undirected graph representing a restricted Boltzmann machine. Note that the intra-connections (blue and orange) are gone.

Note that this form of the probability $E(x)$ is equivalent to the product of max-cliques, which are of size 2 in this case.

$$\begin{aligned} p(v,h) &= \exp(-E(v,h))/Z && (30)\\ &= \exp(v^T W h + b^T v + c^T h)/Z && (31)\\ &= \exp(v^T W h)\exp(b^T v)\exp(c^T h)/Z && (32)\\ &= \frac{1}{Z}\prod_j \prod_k \exp(W_{jk} h_j v_k) \prod_k \exp(c_k v_k) \prod_j \exp(b_j h_j) && (33) \end{aligned}$$

Therefore, we can think of the $\exp(h^T W x)$ as encoding the cliques of length 2 and the others as cliques of length 1. The fact that we can calculate $p(h \mid x)$ means that inferring the distribution over the hidden variables is easy.

1.3.1 Contrastive Divergence

Now that we've done this, we can finally get to training the model. Now, essentially this is density estimation problem given dataset $\mathcal{D} = \{x^{(t)}\}$ of iid random variables, we want to maximize the likelihood of p_θ, which is really just equivalent to optimizing E_θ. So, let's take the average negative log-likelihood and take the derivative of it

$$\frac{\partial}{\partial \theta} \frac{1}{T} \sum_t -\log p_\theta(x^{(t)}) \tag{34}$$

There's a lot of computation to do here, so let's focus on one sample $x^{(t)}$ and claim that the gradient ultimately ends up as the following.

Theorem 1.1 (Decomposition of Derivative)

The derivative of the log-likelihood decomposes into the following terms.

$$\frac{\partial}{\partial \theta} - \log p(x^{(t)}) = \sum_h p(h \mid x^{(t)}) \frac{\partial E(x^{(t)}, h)}{\partial \theta} - \sum_{x,h} p(x,h) \frac{\partial E(x,h)}{\partial \theta} \tag{35}$$

$$= \underbrace{\mathbb{E}_h \left[\frac{\partial E(x^{(t)}, h)}{\partial \theta} \;\middle|\; x^{(t)} \right]}_{\text{positive phase}} - \underbrace{\mathbb{E}_{x,h} \left[\frac{\partial E(x,h)}{\partial \theta} \right]}_{\text{negative phase}} \tag{36}$$

which reduces to

$$-\nabla_\theta \ln p(x) = \begin{cases} -\nabla_W \ln p(x) & = \sum_h p(h \mid x) h x^T - \sum_{x,h} p(x,h) h x^T \\ -\nabla_b \ln p(x) & = \sum_h p(h \mid x) h - \sum_{x,h} p(x,h) h \\ -\nabla_c \ln p(x) & = \sum_h p(h \mid x) x - \sum_{x,h} p(x,h) x \end{cases} \tag{37}$$

Proof.

From the energy model form, we can see that $Z = \sum_{x,h} \exp(-E(x,h))$. Therefore,

$$\ln(Z) = \ln \left(\sum_{x,h} \exp(-E(x,h)) \right) \tag{38}$$

$$\frac{\partial}{\partial \theta} \ln(Z) = \frac{1}{Z} \sum_{x,h} \exp(-E(x,h)) \cdot -1 \cdot \frac{\partial}{\partial \theta} E(x,h) \tag{39}$$

$$= -\frac{1}{Z} \sum_{x,h} \exp(-E(x,h)) \cdot \frac{\partial}{\partial \theta} E(x,h) \tag{40}$$

$$= -\frac{1}{Z} \sum_{x,h} Z \cdot p(x,h) \cdot \frac{\partial}{\partial \theta} E(x,h) \tag{41}$$

$$= -\sum_{x,h} p(x,h) \frac{\partial E(x,h)}{\partial \theta} \tag{42}$$

We have by definition

$$-\ln p(x) = -\ln \left\{ \sum_h \exp\big(-E(x,h)\big) \right\} + \ln(Z) \tag{43}$$

and so when taking the derivative, the second term is solved from above and the first term, we apply the chain rule to get

$$-\frac{\partial}{\partial \theta} \ln p(x) = \frac{\sum_h \exp\big(-E(x,h)\big) \frac{\partial E(x,h)}{\partial \theta} / Z}{\sum_h \exp\big(-E(x,h)\big)/Z} + \frac{\partial \ln(Z)}{\partial \theta} \tag{44}$$

$$= \frac{\sum_h p(x,h) \frac{\partial E(x,h)}{\partial \theta}}{p(x)} + \frac{\partial \ln(Z)}{\partial \theta} \tag{45}$$

$$= \sum_h p(h \mid x) \frac{\partial E(x,h)}{\partial \theta} - \sum_{x,h} p(x,h) \frac{\partial E(x,h)}{\partial \theta} \tag{46}$$

We can use the closed form of E to then compute its partials.

$$\left(\frac{\partial E(x,h)}{\partial W}\right)_{ij} = \frac{\partial E(x,h)}{\partial W_{ij}} = \frac{\partial}{\partial W_{ij}}\left\{ -\sum_{i,j} W_{ij}h_i x_j - \sum_j c_j x_j - \sum_i b_i h_i \right\} = h_i x_j \tag{47}$$

$$\left(\frac{\partial E(x,h)}{\partial b}\right)_{i} = \frac{\partial E(x,h)}{\partial b_i} = h_i \tag{48}$$

$$\left(\frac{\partial E(x,h)}{\partial c}\right)_{j} = \frac{\partial E(x,h)}{\partial c_j} = x_j \tag{49}$$

which in matrix form is

$$\nabla_\theta E(x,h) = \{\nabla_W E(x,h), \nabla_b E(x,h), \nabla_c E(x,h)\} = \{hx^T, h, x\} \tag{50}$$

The conditional probability can be factorized out given x and computed easily by conditional independence.

$$p(h \mid x) = \prod_j p(h_j \mid x) \tag{51}$$

and so given that we have computed this for all h, we can write our gradient as

$$-\nabla_\theta \ln p(x) = \begin{cases} -\nabla_W \ln p(x) & = \sum_h p(h \mid x) hx^T - \sum_{x,h} p(x,h) hx^T \\ -\nabla_b \ln p(x) & = \sum_h p(h \mid x) h - \sum_{x,h} p(x,h) h \\ -\nabla_c \ln p(x) & = \sum_h p(h \mid x) x - \sum_{x,h} p(x,h) x \end{cases} \tag{52}$$

Therefore, we can easily compute the left summation in the gradient form, but the right summation requires us to compute $p(x,h)$ as a general joint distribution, which is intractable. So we just approximate this with a Monte Carlo estimator, specifically Gibbs sampling. This method is known as *contrastive divergence* which was introduced in 2002 by Geoffrey Hinton in [Hin02].

Algorithm 1.1 (Contrastive Divergence)

The general idea is to replace by the expectation by a point estimate at $\tilde{x}$, which we can obtain by sampling the conditions over and over through Gibbs. Since we know $p(x \mid h)$ and $p(h \mid x)$ easily, we can start sampling the chain for some predetermined K steps (actually $2K$ since we are sampling the x and h back and forth), and whatever $\tilde{x}, \tilde{h}$ you sample at the end is your estimate. You then use this to approximate the negative phase

$$\mathbb{E}_{x,h}\left[\frac{\partial E(x,h)}{\partial \theta}\right] \approx \frac{\partial}{\partial \theta} E(\tilde{x}, \tilde{h}) \tag{53}$$

Here are the steps.

1. Initialize $x^0 = x$ and sample h^0 from $p(h \mid x^0)$.
2. For $k = 1, \dots, K$,
   (a) Sample x^k from $p(x \mid h^{k-1})$.
   (b) Sample h^k from $p(h \mid x^k)$.
3. Once you get $\tilde{x} = x^K$,[a] use this to approximate

$$-\nabla_W \ln p(x) = \sum_h p(h \mid x) hx^T - \sum_{x,h} p(x,h) hx^T \approx \sum_h p(h \mid x) hx^T - \sum_h p(h \mid \tilde{x}) h\tilde{x}^T \tag{54}$$

$$-\nabla_b \ln p(x) = \sum_h p(h \mid x) h - \sum_{x,h} p(x,h) h \approx \sum_h p(h \mid x) h - \sum_h p(h \mid \tilde{x}) h \tag{55}$$

$$-\nabla_c \ln p(x) = \sum_h p(h \mid x) x - \sum_{x,h} p(x,h) x \approx \sum_h p(h \mid x) x - \sum_h p(h \mid \tilde{x}) \tilde{x} \tag{56}$$

We can tweak this procedure, such as **persistent CD**, where instead of initializing the chain to $x^{(t)}$, we can initialize the chain to the negative sample of the last iteration.

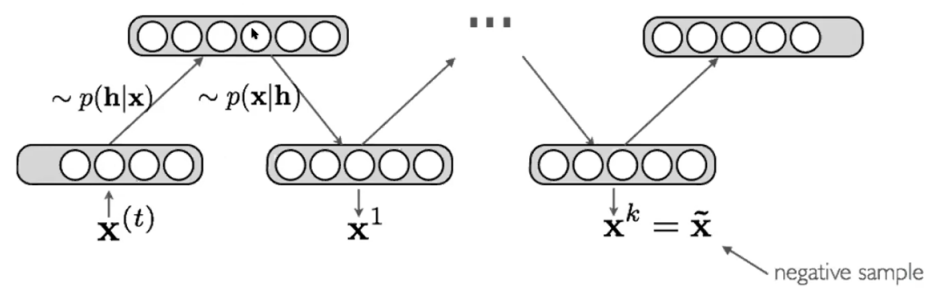

Figure 3: In general, the bigger k is, the less biased the estimate of the gradient will be, and in practice $k = 1$ works well for learning good features. The reason this is called contrastive divergence is that in the gradient update step, we have a positive sample and a negative sample that both approximates the expected gradient, which contrasts to each other.

Therefore, contrastive divergence with k iterations gives us the **CD-k algorithm**.

[a]Note that we do not use $\tilde{h} = h^K$.

Algorithm 1.2 (Fitting)

Therefore, for updating θ, we get the following

$$\begin{aligned} W &= W - \alpha\big(\nabla_W(-\log p(x^{(t)}))\big) && (57)\\ &= W - \alpha\big(\mathbb{E}_h[\nabla_W E(x^{(t)}, h) \mid x^{(t)}] - \mathbb{E}_{x,h}[\nabla_W E(x, h)]\big) && (58)\\ &= W - \alpha\big(\mathbb{E}_h[\nabla_W E(x^{(t)}, h) \mid x^{(t)}] - \mathbb{E}_h[\nabla_W E(\bar{x}, h) \mid \bar{x}]\big) && (59)\\ &= W + \alpha\big(h(x^{(t)})(x^{(t)})^T - h(\bar{x})\bar{x}^T\big) && (60) \end{aligned}$$

and doing this over all three parameters leads to

$$\begin{aligned} W &\leftarrow W + \alpha\big(h(x^{(t)})(x^{(t)})^T - h(\bar{x})\bar{x}^T\big) && (61)\\ b &\leftarrow b + \alpha\big(h(x^{(t)}) - h(\bar{x})\big) && (62)\\ c &=\leftarrow c + \alpha\big(x^{(t)} - \hat{x}\big) && (63) \end{aligned}$$

Example 1.3 (Collaborative Filtering)

Netflix dataset.

1.3.2 Inference with Bernoulli-Bernoulli RBMs

We have talked about RBMs of a general form, but the standard is that the hidden units are almost always Bernoulli, while the visible ones are either Bernoulli or Gaussian. Let's talk about when x, h are both Bernoulli, which allows us to simplify the general form of training.

Definition 1.3 (Bernoulli-Bernoulli RBM)

For now, let us assume that we are trying to estimate the distribution of a Bernoulli random vector $x \in \{0,1\}^D$ with Bernoulli latent variables $h \in \{0,1\}^F$. Then, the energy of the joint configuration is

$$E(v,h;\theta) = -\sum_{ij} W_{ij} v_i h_j - \sum_i b_i v_i - \sum_j a_j h_j = -v^T W h - b^T v - a^T h \tag{64}$$

where $\theta = \{W, a, b\}$ are the model parameters.

Let's get some calculations out of the way.

Lemma 1.1 (Conditional Distributions)

For the Bernoulli RBM, we have

$$p(h_j = 1 \mid x) = \sigma(b_j + W_{j,:}x) \tag{65}$$
$$p(x_k = 1 \mid h) = \sigma(c_k + h^T W_{:,k}) \tag{66}$$

Proof.

Just use the definition of conditional probability and substitute the result below in the denominator. The terms will cancel out.

Lemma 1.2 (Free Energy)

For the Bernoulli RBM, we want to compute the marginal $p(x)$ as

$$\begin{aligned} p(x) &= \frac{\exp(-F(x))}{Z} \\ &= \frac{1}{Z}\exp\left(c^T x + \sum_{j=1}^{H} \log\left(1 + \exp(b_j + W_{j,:}x)\right)\right) \\ &= \frac{1}{Z}\exp\left(c^T x + \sum_{j=1}^{H} \text{softplus}(b_j + W_{j,:}x)\right) \end{aligned}$$

where F is called the **free energy** and the softplus is defined.

$$\text{softplus}(x) = \ln(1 + e^x) \tag{67}$$

Therefore, $p(x)$ is calculated by taking the product of these terms, which is why it's known as a **product of experts model**.

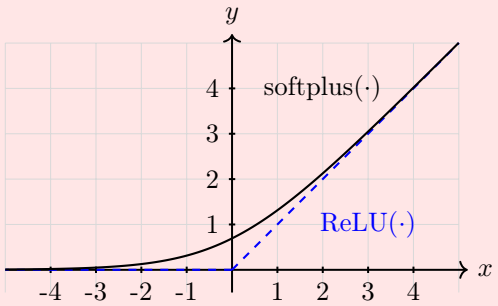

Figure 4: A graph of the softplus activation function, with the dotted ReLU.

Proof.

We have

$$p(x) = \sum_{h\in\{0,1\}^H} \exp\big(hWx + c^Tx + b^Th\big)/Z \tag{68}$$

$$= \exp(c^Tx) \sum_{h_1=0,1} \dots \sum_{h_H=0,1} \exp\left(\sum_j h_j W_{j,:}x + b_j h_j\right)/Z \tag{69}$$

$$= \exp(c^Tx)\left(\sum_{h_1=0,1} \exp(h_1 W_{1,:}x + b_1 h_1)\right) \dots \left(\sum_{h_H=0,1} \exp(h_H W_{H,:}x + b_H h_H)\right)/Z \tag{70}$$

$$= \exp(c^Tx)\big(1+\exp(b_1 + W_{1,:}x)\big)\dots\big(1+\exp(b_H + W_{H,:}x)\big)/Z \tag{71}$$

$$= \exp(c^Tx)\exp\big\{\log\big(1+\exp(b_1 + W_{1,:}x)\big)\big\}\dots\exp\big\{\log\big(1+\exp(b_H + W_{H,:}x)\big)\big\}/Z \tag{72}$$

$$= \frac{1}{Z}\exp\left(c^Tx + \sum_{j=1}^{H}\log\big(1+\exp(b_j + W_{j,:}x)\big)\right) \tag{73}$$

When training, we can use the closed form to simplify our calculations.

$$\frac{\partial E(x,h)}{\partial W_{jk}} = \frac{\partial}{\partial W_{jk}} \tag{74}$$

and so

$$\mathbb{E}_h\left[\frac{\partial E(x,h)}{\partial W_{jk}}\middle| x\right] = \mathbb{E}_h[-h_j x_k \mid x] = \sum_{h_j=0,1} -h_j x_k\, p(h_j \mid x) = -x_k p(h_j = 1 \mid x) \tag{75}$$

where the final term is a sigmoid. Hence, we have

$$\mathbb{E}_h[\nabla_W E(w,h) \mid x] = -h(x)x^T, \text{ where } h(x) \coloneqq \begin{pmatrix} p(h_1 = 1 \mid x) \\ \vdots \\ p(h_H = 1 \mid x)\end{pmatrix} = \sigma(b + Wx) \tag{76}$$

Now we can substitute what we solved into the second expectation, but again this is infeasible to calculate

$$\mathbb{E}_{x,h}\left[\frac{\partial E(x,h)}{\partial \theta}\right] = \sum_{x,h} h(x)x^T p(x,h) \tag{77}$$

1.3.3 Inference with Gaussian-Bernoulli RBMs

Now we can talk about Gaussian Bernoulli RBMs.

Definition 1.4 (Gaussian-Bernoulli RBM)

If we assume that v is a real-valued (unbounded) input that follows a Gaussian distribution (with h still Bernoulli), then we can add a quadratic term to the energy function

$$E(x,h) = -h^TWx - c^Tx - b^Th - \frac{1}{2}x^Tx \tag{78}$$

In this case, $p(x \mid h)$ becomes a Gaussian distribution $N(c+W^Th, I)$. The training process is slightly harder for this, so what we usually do is normalize the training set by subtracting the mean off each input and dividing the input by the training set standard deviation to get

$$E(v,h;\theta) = \sum_i \frac{(v_i - b_i)^2}{2\sigma_i^2} - \sum_{ij} W_{ij}h_j\frac{v_i}{\sigma_i} - \sum_j a_j h_j \tag{79}$$

You should also use a smaller learning rate α compared to Bernoulli RBM.

Algorithm 1.3 (Implementation)

For an implementation with PyTorch, see here.

1.4 Score Matching

Due to the computational cost of MCMC algorithms in graphical models in general, Hyvarinen in 2005 introduced an indirect method to estimate the log likelihood by calculating the gradient of the PDF with respect to the sample, called *score matching*. It is motivated by the following theorem.

Theorem 1.2 ()

If two continuously differentiable real-valued functions $f(x), g(x)$ have equal first derivatives everywhere, then $f(x) = g(x) + c$ for some constant c.

If we apply this to our energy model, due to the normalization requirement it is sufficient that $f(x) = g(x)$ if they match in their first derivatives. This gradient has a special name.

Definition 1.5 (Score)

Let X be a continuous random variable defined on $\mathbb{R}^n$, and let p be its pdf. The **score function** of p is the gradient of the log-pdf with respect to the sample.[a]

$$\psi(x) = \nabla_x \log p(x) = \begin{bmatrix} \frac{\partial \log p(x)}{\partial x_1} \\ \vdots \\ \frac{\partial \log p(x)}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \psi_1(x) \\ \vdots \\ \psi_n(x) \end{bmatrix} \tag{80}$$

Note that the score is a function $\psi : \mathbb{R}^n \to \mathbb{R}^n$.

[a] Note that this is w.r.t. the sample, not the parameter, unlike what we do usually in machine learning.

The reason Hyvarinen introduced this score function in 2005 is because we want to have such a score is that it does not depend on the normalizing constant Z. Our original problem was having the normalizing constant

$$Z = \int p_\theta(x)\, dx \tag{81}$$

indeed changes w.r.t. θ, but is stays constantly 1 with respect to z, and so by taking the log-derivative w.r.t. x, we can actually get rid of this term completely. [Hyv05] Therefore, rather than maximizing the likelihood, we want to minimize the expected L2 distance between the score functions, called the *Fisher divergence*.[3]

Definition 1.6 (Fisher divergence)

The **Fisher divergence** of two probability distributions p, p_θ is

$$D_F(p(x)||p_\theta(x)) := \mathbb{E}_{p(x)}\left[\frac{1}{2}||\psi(x) - \psi_\theta(x)||^2\right] \tag{82}$$

$$= \mathbb{E}_{p(x)}\left[\frac{1}{2}||\nabla_x \log p(x) - \nabla_x \log p_\theta(x)||^2\right] \tag{83}$$

[3] Not to be confused with the Fisher score, which is a scalar and uses the same notation s in some of the papers.

Therefore, rather than minimizing the KL-divergence, i.e. maximizing the log-likelihood, we can choose to minimize the Fisher divergence. So we have come up with another way to conduct inference by an alternative objective to minimize. But unsurprisingly, the KL and Fisher divergences are related.

Theorem 1.3 (Relationship Between Fisher and KL Divergnces)

Consider two random variables X, Y with pdfs $p(x), q(y)$, and let us take the continuous time stochastic process perturbed by *independent* normals

$$X_t = X + \mathcal{N}(0,t), \qquad Y_t = Y + \mathcal{N}(0,t) \tag{84}$$

and let $p_t(x), q_t(y)$ be the pdfs of X_t, Y_t. Then we have

$$D_F(p,q) = -2\frac{d}{dt} D_{KL}(p_t, q_t)\Big|_{t=0} \tag{85}$$

This leads to the dual prime inequality, aka 2nd law of thermodynamics.[a]

$$\lim_{t\to 0} \frac{D_{KL}(p_t||q_t) - D_{KL}(p||q)}{t} \tag{86}$$

[a] What Tarokh said but need to confirm on this.

Proof.

Follows from de Bruijn equality.

$$J_F(X) = 2\frac{d}{dt}[H(X_t)] \tag{87}$$

The reason the Fisher divergence is so attractive is that under some mild regularity conditions, the parameter estimated by minimizing the Fisher divergence converges to the true parameters in the limit of infinite data. Furthermore, we can decompose it into the following.

Theorem 1.4 (No Need for True Score to Optimize Fisher Divergence Objective)

The Fisher divergence can be decomposed to

$$D_F(p(x)||p_\theta(x)) = \mathbb{E}_{p(x)}\left[\underbrace{\frac{1}{2}||\nabla_x \log p_\theta(x)||^2}_{s_P(x,p)} + \Delta_x \log p_\theta(x)\right] + c^* \tag{88}$$

where Δ represents the trace of the Hessian, s_P is called the Fisher score, and c^* is term that depends only on p the true data generating distribution.

Proof.

The full proof is on the appendix of [Hyv05], but we show for one variable gone through class.

$$D_F(p(x)||p_\theta(x)) = \frac{1}{2}\int p(x)\big(|\nabla_x \log p_\theta(x)|^2 + |\nabla_x \log p(x)|^2 - 2\nabla_x \log p(x)\,\nabla_x \log p_\theta(x)\}\big)\,dx \tag{89}$$

The first term is $\frac{1}{2}\mathbb{E}_p(x)[|\nabla_x \log p_\theta(x)|^2]$, the second term is only dependent on p and we set that as

c^*, and the third term can be decomposed by integration by parts into

$$\int p(x)\frac{[p(x)]'}{p(x)} \cdot \nabla_x \log p_\theta(x)\, dx = \int [p(x)]' \cdot \nabla_x \log p_\theta(x)\, dx \tag{90}$$

$$= p(x)\nabla_x \log p_\theta(x)\Big|_{-\infty}^{+\infty} - \int p(x)\Delta_x \log p_\theta(x)\, dx \tag{91}$$

As a hand-wavy way, we can see that since $p(x)$ is a pdf, it must be the case that

$$0 = \lim_{x\to+\infty} p(x) = \lim_{x\to-\infty} p(x) \tag{92}$$

and since the log probability is bounded the first term is 0, leaving us with the expectation of the Hessian.

This is the key theorem in Hyvarinen's paper. In the original objective, we must minimize the L2 distance between the original score and the score that we can compute by the neural network (at this point, we are assuming still that the neural net $E_\theta(x) \approx \log p_\theta(x)$, and the score is obtained by backpropagating through x). But we have no clue what the true score it, so how are we supposed to minimize the L2 distance of our model outputs with something that we don't even know? The theorem states that by minimizing the reformulated objective above (which does *not* depend on the true score at all), we can actually have our output converge onto the true score without ever having to know the true score!

Therefore, by taking the gradient w.r.t. θ, we can put this inside the expectation and get an unbiased estimator to optimize this objective.

$$\nabla_\theta D_F(p(x)||p_\theta(x)) = \nabla_\theta \mathbb{E}_{p(x)}\left[\frac{1}{2}||\nabla_x \log p_\theta(x)||^2 + \Delta_x \log p_\theta(x)\right] \tag{93}$$

$$= \frac{1}{2}\mathbb{E}_{p(x)}\left[\nabla_\theta||\nabla_x \log p_\theta(x)||^2 + \nabla_\theta\Delta_x \log p_\theta(x)\right] \tag{94}$$

$$\approx \frac{1}{2N}\sum_{i=1}^{N} \nabla_\theta||\nabla_x \log p_\theta(x^{(i)})||^2 + \nabla_\theta\Delta_x \log p_\theta(x^{(i)}) \tag{95}$$

where $x^{(i)}$ are the true samples generated from $p(x)$.

Now when we try to actually optimize this with SGD, note what would happen if we modeled our neural net $E_\theta(x) \approx \log p(x)$ to output to log probability. There are two huge problems.

1. We would have to backprop on x not just once, but twice due to the Hessian calculation. This is already $O(d^2)$ in dimension d. Even though it only requires the trace of the Hessian, this does not scale well to large dimensions.
2. After backpropagating with x, in PyTorch we can't just backprop on θ after since it still computes first derivatives. The feature to compute the second derivatives w.r.t. x and θ is not possible in PyTorch, and it also scales terribly.

To solve the first problem, note that one can immediately see that the Fisher divergence objective requires us to know the Fisher score, which is itself a gradient, and so the model that we should be training $E_\theta(x)$ should really be outputting the Fisher score, not the log-probability!

$$E_\theta(x) \approx \nabla_x \log p_\theta(x) \tag{96}$$

If we can rather have the neural network output the score itself rather than the log-probability, then we won't have to take the second derivative. This also does not affect inference since we can use the score to sample from the distribution using MCMC, such as Langevin dynamics which directly requires the value of $\nabla_x \log p_\theta(x)$ in each iteration. Therefore, modifying the neural net to output scores rather than the log

probability helps us with training, and at the same time, does not hinder our ability to sample from the model.[4] Essentially, assuming that $s_\theta(x^{(i)}) = E_\theta(x^{(i)}$, our gradient update step would look like this

$$\nabla_\theta D_F(p(x)||p_\theta(x)) \approx \frac{1}{2N}\sum_{i=1}^{N} \nabla_\theta ||s_\theta(x^{(i)})||^2 + \nabla_\theta \nabla_x s_\theta(x^{(i)}) \tag{97}$$

$$\approx \frac{1}{2N}\sum_{i=1}^{N} \nabla_\theta ||E_\theta(x^{(i)})||^2 + \nabla_\theta \nabla_x E_\theta(x^{(i)}) \tag{98}$$

This is better, but the second problem remains. That is, once we backprop on x we get $\nabla_x E_\theta(x)$, but we must treat this as a function itself and do a second derivative w.r.t. θ. This still isn't possible in PyTorch, and to solve this, the next models we introduce—denoising score matching and sliced score matching—estimates noisy versions of the Hessian with first-order approximations. In summary, both models avoid having to calculate the derivative w.r.t. x at all, and rather approximate it in some way $\nabla_\theta \nabla_x E_\theta(x) \approx \nabla_\theta g(E_\theta(x))$ for some simple function g.

1.4.1 Denoising Score Matching

The score matching objective requires several regularity conditions for the true $\log p(x)$, notable that it should be continuously differentiable and finite everywhere. However, these conditions may not always hold in practice. For example, a distribution of digital images is typically discrete and bounded with values restricted to $\{0, \dots, 255\}$. Therefore $\log p(x)$ is discontinuous and may reach $-\infty$.

To alleviate this difficult, we can add a bit of noise to each data point and generate new samples $\tilde{x} = x + \epsilon$. As long as the noise distribution $p(\epsilon)$ is smooth, the resulting noisy data distribution $q(\tilde{x}) = \int q(\tilde{x} \mid x)\, p(x)\, dx$ is also smooth, and thus the Fisher divergence is a proper objective. In 2010, Kigma and LeCun in [KC10] showed that the objective with noisy data can be approximated by the noiseless score matching objective plus a regularization term. However, this still required computing expensive second-order derivatives.

In 2011, Vincent showed in [Vin11] that rather than trying to estimate $p_\theta(x) \approx p(x)$, we should focus on estimating the noisy $q(\tilde{x})$, an idea called *denoising score matching* or *DSM*. The biggest advantage of this is the following.

Theorem 1.5 (Decomposition of Noisy Fisher Divergence)

The noisy Fisher divergence can be decomposed to

$$D_F(q(\tilde{x})||p_\theta(\tilde{x})) = \mathbb{E}_{q(\tilde{x})}\left[\frac{1}{2}||\nabla_x \log q(\tilde{x}) - \nabla_x \log p_\theta(\tilde{x})||_2^2\right] \tag{99}$$

$$= \mathbb{E}_{q(x,\tilde{x})}\left[\frac{1}{2}||\nabla_x \log q(\tilde{x} \mid x) - \nabla_x \log p_\theta(\tilde{x})||_2^2\right] + \text{constant} \tag{100}$$

Proof.

Letting $s_\theta(\tilde{x}) = \nabla_x \log p_\theta(\tilde{x})$, we can expand this term.

$$\frac{1}{2}\mathbb{E}_{x\sim q}\left[\|\nabla_x \log q(\tilde{x}) - s_\theta(\tilde{x})\|_2^2\right] = \frac{1}{2}\int q(\tilde{x})\|\nabla_x \log q(\tilde{x})\|_2^2 d\tilde{x} + \frac{1}{2}\int q(\tilde{x})\|s_\theta(\tilde{x})\|_2^2 d\tilde{x} \tag{101}$$

$$- \int q(\tilde{x})\nabla_x \log q(\tilde{x})^T s_\theta(\tilde{x}) d\tilde{x} \tag{102}$$

Note that the first integral is a constant w.r.t. θ, and the second term is $\frac{1}{2}\mathbb{E}_{q(x)}[||s_\theta(\tilde{x})||_2^2]$. If we

[4] Note that before we can model our estimated pdf as $p_\theta(x) = \frac{1}{Z}e^{-E_\theta(x)}$, but now there is no simple form. But we haven't really lost anything. Since even in the old model, we must use MCMC to sample since Z is intractable, and in our score-output model, our sampling is still identically done with MCMC.

manipulate the final term, we get

$$
\begin{aligned}
-\int q(\tilde{x})\nabla_{\tilde{x}}\log q(\tilde{x})^T s_\theta(\tilde{x})d\tilde{x} &= -\int q(\tilde{x})\frac{1}{q(\tilde{x})}\nabla_{\tilde{x}}q(\tilde{x})^T s_\theta(\tilde{x})d\tilde{x} && (103)\\
&= -\int \nabla_{\tilde{x}}q(\tilde{x})^T s_\theta(\tilde{x})d\tilde{x} && (104)\\
&= -\int \nabla_{\tilde{x}}\left(\int p(x)q(\tilde{x}|x)dx\right)^T s_\theta(\tilde{x})d\tilde{x} && (105)\\
&= -\int \left(\int p(x)\nabla_{\tilde{x}}q(\tilde{x}|x)dx\right)^T s_\theta(\tilde{x})d\tilde{x} && (106)\\
&= -\int \left(\int p(x)q(\tilde{x}|x)\nabla_{\tilde{x}}\log q(\tilde{x}|x)dx\right)^T s_\theta(\tilde{x})d\tilde{x} && (107)\\
&= -\iint p(x)q(\tilde{x}|x)\nabla_{\tilde{x}}\log q(\tilde{x}|x)^T s_\theta(\tilde{x})dxd\tilde{x} && (108)\\
&= -\mathbb{E}_{x\sim p(x),\tilde{x}\sim q(\tilde{x}|x)}\left[\nabla_{\tilde{x}}\log q(\tilde{x}|x)^T s_\theta(\tilde{x})\right] && (109)\\
&= -\mathbb{E}_{q(x,\tilde{x})}\left[\nabla_{\tilde{x}}\log q(\tilde{x}|x)^T s_\theta(\tilde{x})\right] && (110)
\end{aligned}
$$

By the tower rule, we have $\mathbb{E}_{q(x)}[||s_\theta(\tilde{x})||] = \mathbb{E}_{q(x,\tilde{x})}[||s_\theta(\tilde{x})||]$ and summing them back up we have

$$
\begin{aligned}
D_F(q(\tilde{x})||p_\theta(\tilde{x})) &= c + \frac{1}{2}\mathbb{E}_{q(x,\tilde{x})}[||s_\theta(\tilde{x})||_2^2] - \mathbb{E}_{q(x,\tilde{x})}\left[\nabla_{\tilde{x}}\log q(\tilde{x}|x)^T s_\theta(\tilde{x})\right] && (111)\\
&= c + \frac{1}{2}\mathbb{E}_{q(x,\tilde{x})}[||\nabla_{\tilde{x}}\log q(\tilde{x}\mid x) - s_\theta(\tilde{x})||_2^2] - \frac{1}{2}\mathbb{E}_{q(x,\tilde{x})}[||\nabla_{\tilde{x}}\log q(\tilde{x}\mid x)||_2^2] && (112)
\end{aligned}
$$

but again, note that the third term is not dependent on θ and can therefore be treated as a constant.

Therefore we have not only just completely removed the hessian term, but also removed the expectation over the true $p(x)$. Doing this has drawbacks however. When $p(x)$ is already a well-behaved distribution that satisfies the regularity conditions, then $D_F(q(x\tilde{x})||p_\theta(\tilde{x})) \neq D_F(p(x)||p_\theta(x))$, and we are optimizing the noisy q. Therefore, we are not really minimizing the true objective, and so our job is to add a little bit a noise so that $q(x) \approx p(x)$.

Since we can construct a pretty simple form of $q(\tilde{x} \mid x)$, the gradient is easy to calculate and can be done in closed form in fact. The nice property is that we are matching $\nabla_{\tilde{x}}\log q(\tilde{x} \mid x)$ and not $\nabla_x \log p(x)$. One way we can construct q is to perturb it by a small Gaussian, so $\tilde{x} = x + \sigma z$ for $z \sim \mathcal{N}(0, I)$. Then $q(\tilde{x} \mid x) = \mathcal{N}(\tilde{x} \mid x, \sigma^2 I)$, and since the log-derivative of the conditional Gaussian is $-\frac{(\tilde{x}-x)}{\sigma^2} = -\frac{z}{\sigma}$, the corresponding objective—which we use the reparameterization trick to modify the expectation—is

$$
\begin{aligned}
D_F(q(\tilde{x})||p_\theta(\tilde{x})) &= \mathbb{E}_{p(x)}\mathbb{E}_{z\sim\mathcal{N}(0,1)}\left[\frac{1}{2}\left|\left|\frac{z}{\sigma} + \nabla_x \log p_\theta(x+\sigma z)\right|\right|_2^2\right] && (113)\\
&\approx \frac{1}{2N}\sum_{i=1}^{N}\left|\left|\frac{z^{(i)}}{\sigma} + \nabla_x \log p_\theta(x^{(i)} + \sigma z^{(i)})\right|\right|_2^2 && (114)
\end{aligned}
$$

where $x^{(i)}$ are the data samples and $z^{(i)}$ are some Gaussian samples. However, note that while we cannot set σ too high due to $q \not\approx p$, we also can't set it too small since the variance of the expression explodes.

$$
\mathbb{E}_{p(x)}\mathbb{E}_{z\sim\mathcal{N}(0,1)}\left[\frac{1}{2}\frac{\|z\|_2^2}{\sigma} + s_\theta(x+\sigma z)^T\frac{z}{\sigma} + \frac{1}{2}\left\|s_\theta(x+\sigma z)\right\|_2^2\right] \tag{115}
$$

Since both the variances of $\frac{\|z\|_2^2}{\sigma}$ and $s_\theta(x+\sigma z)^T\frac{z}{\sigma}$ will both grow unbounded as $\sigma \to 0$, we must fine-tune σ, and [SSK+20] in 2019 suggested to choose a decreasing sequence of positive numbers $\sigma_1 > \sigma_2 > \ldots > \sigma_L$,

and use the denoising score matching objective with Gaussian noise of variance σ_i to train the score function at the ith epoch.

Algorithm 1.4 (Denoised Score Matching)

Now we are finally ready to present the algorithm. Note that in here, we never know the *true* scores!

Algorithm 1 Denoised Score Matching

Require: Input data $x^{(i)}$
Require: Batch size B
Require: Neural network that outputs the score $s_\theta(x) = E_\theta(x)$.
1: **while** not converged **do**
2: Sample minibatch of datapoints $x^{(1)}, \dots, x^{(B)} \sim p(x)$
3: Sample minibatch of Gaussian noise $z^{(1)}, \dots, z^{(B)} \sim \mathcal{N}(0, I)$
4: Sample some small, but not too small, standard deviations $\sigma^{(1)}, \dots, \sigma^{(B)}$.[a]
5: Forward prop through the neural net to generate noisy scores

$$s_\theta(x^{(i)}) = E_\theta(x^{(i)} + \sigma^{(i)} z^{(i)}) \tag{116}$$

6: Compute the noisy Fisher divergence with empirical means

$$\frac{1}{B} \sum_{i=1}^{B} \left\| \frac{z^{(i)}}{\sigma^{(i)}} + s_\theta(x^{(i)}) \right\|_2^2 \tag{117}$$

7: Backprop the equation through E.
8: **end while**

Note that we can use more than one projections per data point as well.

[a] There should really be a scheduler to fine-tune these values.

1.4.2 Sliced Score Matching

By adding noise to the data, DSM avoids the expensive computation of the second-order derivatives. However, the optimal EBM that minimizes the DSM objective corresponds to the distribution of the noise-perturbed data $q(\tilde{x})$, not the original $p(x)$. Therefore, in 2019, Song introduced in [SGSE19] as an alternative that is *both* consistent and computationally efficient. The general idea is that instead of minimizing the Fisher divergence between two vector-valued scores, *sliced score matching* randomly samples a projection vector v, takes the inner product between v and the two scores, and then compare the resulting two scalars. More specifically, it minimizes the following.

Definition 1.7 (Sliced Fisher Divergence)

The **sliced Fisher divergence** is

$$D_{SF}(p(x)||p_\theta(x)) = \mathbb{E}_{p(x)} \mathbb{E}_{p(v)} \left[\frac{1}{2} \left(v^T \nabla_x \log p(x) - v^T \nabla_x \log p_\theta(x) \right)^2 \right] \tag{118}$$

where $p(v)$ denotes a projection distribution such that $\mathbb{E}_{p(v)}[vv^T]$ is positive definite. A common distribution to sample from is $v \sim \mathcal{N}(0, I)$ or the multivariate normal with unit length.

Similar to Fisher divergence, we can again decompose this into a form that does not involve the unknown

$\nabla_x \log p(x)$, given by

$$D_{SF}(p(x)||p_\theta(x)) = \mathbb{E}_{p(x)}\mathbb{E}_{p(v)}\left[\frac{1}{2}\{v^T\nabla_x \log p_\theta(x)\}^2 + v^T[\Delta_x \log p_\theta(x)]v\right] + c \tag{119}$$

At first glance, it may seem like we have the problematic Hessian term, but by contracting v first with the first derivatives before computing the second derivatives, we can get a linear cost in d.

$$v^T[\Delta_x \log p_\theta(x)]v = \sum_{i=1}^d \sum_{j=1}^d \frac{\partial^2 \log p_\theta(x)}{\partial x_i \partial x_j} v_i v_j = \sum_{i=1}^d \frac{\partial}{\partial x_i} \underbrace{\left(\sum_{j=1}^d \frac{\partial \log p_\theta(x)}{\partial x_i} v_i\right)}_{\text{contract first}} v_i \tag{120}$$

Since the contracted part is the same for different values for i, we only need to compute the value once, backprop on this scalar, and then take the linear combination with weights v_i, which requires two linear passes.

Algorithm 1.5 (Sliced Score Matching)

Now we are finally ready to present the algorithm. Note that in here, we never know the *true* scores!

Algorithm 2 Sliced Score Matching

Require: Input data $x^{(i)}$
Require: Batch size B
Require: Neural network that outputs the score $s_\theta(x) = E_\theta(x)$.

1: **while** not converged **do**
2: Sample minibatch of datapoints $x^{(1)}, \dots, x^{(B)} \sim p(x)$
3: Sample minibatch of projection directions $v^{(1)}, \dots, v^{(B)} \sim p(v)$
4: Forward prop through the neural net to generate scores

$$s_\theta(x^{(i)}) = E_\theta(x^{(i)}) \tag{121}$$

5: Compute the sliced score matching loss with empirical means

$$\frac{1}{B}\sum_{i=1}^B \left[(v^{(i)})^T[\nabla_x s_\theta(x^{(i)})]v^{(i)} + \frac{1}{2}\left\{(v^{(i)})^T s_\theta(x^{(i)})\right\}^2\right] \tag{122}$$

Note that this can be done in linear time by contracting it first. That is, compute $s_\theta(x) \cdot v^{(i)}$, and then backprop on x, and then contract it with $v^{(i)}$ again.
6: Backprop the equation through E, treating $v^{(i)}$ as fixed.[a]
7: **end while**

Note that we can use more than one projections per data point as well.

[a] Remember this is the reparamaterization trick!

1.5 Deep Belief Network

Proposed in 2006

So far, BMs and RBMs aren't really *deep* since they are shallow 2-layer networks, and I could have placed them in my machine learning notes in graphical models. However, RBMs provided the foundations for deep belief networks, a pivotal moment in deep learning, and thus they are considered part of the deep learning curriculum.

1.6 Hopfield Networks

They are EBMs and RNNs!

References

[Hin02] Geoffrey E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14(8):1771–1800, 2002.

[Hyv05] Aapo Hyvärinen. Estimation of non-normalized statistical models by score matching. *Journal of Machine Learning Research*, 6(24):695–709, 2005.

[KC10] Durk P Kingma and Yann Cun. Regularized estimation of image statistics by score matching. In J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 23. Curran Associates, Inc., 2010.

[SGSE19] Yang Song, Sahaj Garg, Jiaxin Shi, and Stefano Ermon. Sliced score matching: A scalable approach to density and score estimation. *CoRR*, abs/1905.07088, 2019.

[SK21] Yang Song and Diederik P. Kingma. How to train your energy-based models. *CoRR*, abs/2101.03288, 2021.

[SSK+20] Yang Song, Jascha Sohl-Dickstein, Diederik P. Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. Score-based generative modeling through stochastic differential equations. *CoRR*, abs/2011.13456, 2020.

[Vin11] Pascal Vincent. A connection between score matching and denoising autoencoders. *Neural Computation*, 23(7):1661–1674, 2011.