Chapter 6
Energy-Based Models

6.1 Introduction

So far, we have discussed various deep generative models for modeling the marginal distribution over observable variables (e.g., images), $p(\mathbf{x})$, such as, autoregressive models (ARMs), flow-based models (flows, for short), Variational Auto-Encoders (VAEs), and hierarchical models like hierarchical VAEs and diffusion-based deep generative models (DDGMs). However, from the very beginning, we advocate for using deep generative modeling in the context of finding the joint distribution over observables and decision variables that is factorized as $p(\mathbf{x}, y) = p(y|\mathbf{x})p(\mathbf{x})$. After taking the logarithm of the joint we obtain two additive components: $\ln p(\mathbf{x}, y) = \ln p(y|\mathbf{x}) + \ln p(\mathbf{x})$. We outlined how such a joint model could be formulated and trained in the hybrid modeling setting (see Chap. 5). The drawback of hybrid modeling though is the necessity of weighting both distributions, i.e., $\ell(\mathbf{x}, y\lambda) = \ln p(y|\mathbf{x}) + \lambda \ln p(\mathbf{x})$, and for $\lambda \neq 1$ this objective does not correspond to the log-likelihood of the joint distribution. The question is whether it is possible to formulate a model to learn with $\lambda = 1$. Here, we are going to discuss a potential solution to this problem using probabilistic **energy-based models** (EBMs) [1].

The history of EBMs is long and dates back to 80 of the previous century when models dubbed **Boltzmann Machines** were proposed [2, 3]. Interestingly, the idea behind Boltzmann machines is taken from statistical physics and was formulated by cognitive scientists. A nice mix-up, isn't it? In a nutshell, instead of proposing a specific distribution like Gaussian or Bernoulli, we can define an **energy function**, $E(\mathbf{x})$, that assigns a value (*energy*) to a given state. There are no restrictions on the energy function so you can already think of parameterizing it with neural networks. Then, the probability distribution could be obtained by transforming the energy to the unnormalized probability $e^{-E(\mathbf{x})}$ and normalizing it by $Z = \sum_{\mathbf{x}} e^{-E(\mathbf{x})}$ (a.k.a. a *partition function*) that yields the Boltzmann (also called Gibbs) distribution:

J. M. Tomczak, *Deep Generative Modeling*,
https://doi.org/10.1007/978-3-030-93158-2_6

$$p(\mathbf{x}) = \frac{e^{-E(\mathbf{x})}}{Z}. \tag{6.1}$$

If we consider continuous random variables, then the sum sign should be replaced by the integral. In physics, the energy is scaled by an inverse of temperature [4], however, we skip it to keep the notation uncluttered. Understanding how the Boltzmann distribution works is relatively simple. Imagine a grid 5-by-5. Then, assign some value (energy) to each of the 25 points where a larger value means that a point has higher energy. Exponentiating the energy ensures that we do not obtain negative values. To calculate the probability for each point, we need to divide all exponentiated energies by their sum, in the same way how we do it for calculating softmax. In the case of continuous random variables, we must normalize by calculating an integral (i.e., a sum over all infinitesimal regions). For instance, the Gaussian distribution could be also expressed as the Boltzmann distribution with an analytically tractable partition function and the energy function of the following form:

$$E(x; \mu, \sigma^2) = \frac{1}{2\sigma^2}(x - \mu)^2, \tag{6.2}$$

that yields

$$p(x) = \frac{e^{-E(x)}}{\int e^{-E(x)} \mathrm{d}x} \tag{6.3}$$

$$= \frac{e^{\frac{1}{2\sigma^2}(x-\mu)^2}}{\sqrt{2\pi\sigma^2}}. \tag{6.4}$$

In practice, most energy functions do not result in a nicely computable partition function. And, typically, the partition function is the key element that is problematic in learning energy-based models. The second problem is that, in general, it is hard to sample from such models. Why? Well, we know the probability for each point but there is no generative process like in ARMs, flows, or VAEs. It is unclear how to start and what is the graphical model for an EBM. We can think of the EBM as a box that for a given $\mathbf{x}$ can tell us the (unnormalized) probability of that point. Notice that the energy function does not distinguish variables in any way, it does not care about any structure in $\mathbf{x}$. It says: Give me $\mathbf{x}$ and I will return the value. That's it! In other words, the energy function defines mountains and valleys over the space of random variables.

A curious reader (yes, I am referring to you!) may ask why we want to bother with EBMs. Previously discussed models are at least tractable and comprehensible in the sense that some stochastic dependencies are defined. Now we suddenly invert the logic and say that we do not care about modeling the structure and instead we want to model an energy function that returns unnormalized probabilities. Is it beneficial? Yes, for at least three reasons. First, in principle, the energy function is unconstrained, it could be any function! Yes, you have probably guessed already, it could be a neural network! Second, notice that the energy function could be multimodal without being defined as such (i.e., opposing to a mixture distribution). Third, there is no difference if we define it over discrete or continuous variables. I hope you see now that EBMs have a lot of advantages! They possess a lot of deficiencies too but hey, let us stick to the positive aspects before we start, ok?

6.2 Model Formulation

As mentioned earlier, we formulate an energy function with some parameters θ over observable and decision random variables, $E(\mathbf{x}, y; \theta)$, that assigns a value (an energy) to a pair $(\mathbf{x}, y)$ where $\mathbf{x} \in \mathbb{R}^D$ and $y \in \{0, 1, \ldots, K-1\}$. Let $E(\mathbf{x}, y; \theta)$ be parameterized by a neural network $NN_\theta(\mathbf{x})$ that returns K values: $NN_\theta : \mathbb{R}^D \rightarrow \mathbb{R}^K$. In other words, we can define the energy as follows:

$$E(\mathbf{x}, y; \theta) = -NN_\theta(\mathbf{x})[y], \tag{6.5}$$

where we indicate by $[y]$ the specific output of the neural net $NN_\theta(\mathbf{x})$. Then, the joint probability distribution is defined as the Boltzmann distribution:

$$p_\theta(\mathbf{x}, y) = \frac{\exp\{NN_\theta(\mathbf{x})[y]\}}{\sum_{\mathbf{x},y} \exp\{NN_\theta(\mathbf{x})[y]\}} \tag{6.6}$$

$$= \frac{\exp\{NN_\theta(\mathbf{x})[y]\}}{Z_\theta}, \tag{6.7}$$

where we define the partition function as $Z_\theta = \sum_{\mathbf{x},y} \exp\{NN_\theta(\mathbf{x})[y]\}$.

Since we have the joint distribution, we can calculate the marginal distribution and the conditional distribution. First, let us take a look at the marginal $p(\mathbf{x})$. Applying the sum rule to the joint distribution yields:

$$p_\theta(\mathbf{x}) = \sum_y p_\theta(\mathbf{x}, y) \tag{6.8}$$

$$= \frac{\sum_y \exp\{NN_\theta(\mathbf{x})[y]\}}{\sum_{\mathbf{x},y} \exp\{NN_\theta(\mathbf{x})\}[y]} \tag{6.9}$$

$$= \frac{\sum_y \exp\{NN_\theta(\mathbf{x})[y]\}}{Z_\theta}. \tag{6.10}$$

Let us notice that we can express this distribution differently. First, we can re-write the numerator in the following manner:

$$\sum_y \exp\{NN_\theta(\mathbf{x})[y]\} = \exp\left\{\log\left(\sum_y \exp\{NN_\theta(\mathbf{x})[y]\}\right)\right\} \tag{6.11}$$

$$= \exp\left\{\text{LogSumExp}_y\{NN_\theta(\mathbf{x})[y]\}\right\} \tag{6.12}$$

where we define $\text{LogSumExp}_y\{f(y)\} = \ln\sum_y \exp\{f(y)\}$. In other words, we can say that the energy function of the marginal distribution is expressed as $-\text{LogSumExp}_y\{NN_\theta(\mathbf{x})[y]\}$. Then, the marginal distribution could be defined as follows:

$$p_\theta(\mathbf{x}) = \frac{\exp\left\{\text{LogSumExp}_y\{NN_\theta(\mathbf{x})[y]\}\right\}}{Z_\theta}. \tag{6.13}$$

Now, we can calculate the conditional distribution $p_\theta(y|\mathbf{x})$. We know that $p_\theta(\mathbf{x}, y) = p_\theta(y|\mathbf{x})\ p_\theta(\mathbf{x})$ thus:

$$p_\theta(y|\mathbf{x}) = \frac{p_\theta(\mathbf{x}, y)}{p_\theta(\mathbf{x})} \tag{6.14}$$

$$= \frac{\frac{\exp\{NN_\theta(\mathbf{x})[y]\}}{Z_\theta}}{\frac{\sum_y \exp\{NN_\theta(\mathbf{x})[y]\}}{Z_\theta}} \tag{6.15}$$

$$= \frac{\exp\{NN_\theta(\mathbf{x})[y]\}}{\sum_y \exp\{NN_\theta(\mathbf{x})[y]\}}. \tag{6.16}$$

The last line should resemble something, do you see it? Yes, you are right, it is the **softmax** function! We have shown that the energy-based model could be used either as a classifier or as a marginal distribution. And it is enough to define a single neural network for that! Isn't it beautiful? The same observations were made in [5] that any classifier could be seen as an energy-based model.

Interestingly, the logarithm of the joint distribution is the following:

$$\ln p_\theta(\mathbf{x}, y) = \ln p_\theta(y|\mathbf{x}) + \ln p_\theta(\mathbf{x}) \tag{6.17}$$

$$= \ln \frac{\exp\{NN_\theta(\mathbf{x})[y]\}}{\sum_y \exp\{NN_\theta(\mathbf{x})[y]\}} + \ln \frac{\sum_y \exp\{NN_\theta(\mathbf{x})[y]\}}{Z_\theta} \tag{6.18}$$

$$= \ln \text{softmax}\{NN_\theta(\mathbf{x})[y]\} + \left(\text{LogSumExp}_y\{NN_\theta(\mathbf{x})[y]\} - \ln Z_\theta\right), \tag{6.19}$$

where we define $\text{LogSumExp}_y\{f(y)\} = \ln\sum_y \exp\{f(y)\}$. We clearly see that the model requires a shared neural network that is used for calculating both

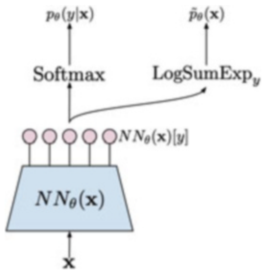

Fig. 6.1 A schematic representation of an EBM. We denote the output of LogSumExp$_y$ by $\tilde{p}_\theta(\mathbf{x})$ to highlight that it is the unnormalized distribution since calculating the partition function is troublesome

distributions. To obtain a specific distribution, we pick the final activation function. The model is schematically presented in Fig. 6.1.

6.3 Training

We have a single neural network to train and the training objective is the logarithm of the joint distribution. Since the training objective is a sum of the logarithm of the conditional $p_\theta(y|\mathbf{x})$ and the logarithm the marginal $p_\theta(\mathbf{x})$, calculating the gradient with respect to the parameters θ requires taking the gradient of each of the component separately. We know that there is no problem with learning a classifier so let us take a closer look at the second component, namely:

$$\nabla_\theta \ln p_\theta(\mathbf{x}) = \nabla_\theta \mathrm{LogSumExp}_y\{NN_\theta(\mathbf{x})[y]\} - \nabla_\theta \ln Z_\theta \tag{6.20}$$

$$= \nabla_\theta \mathrm{LogSumExp}_y\{NN_\theta(\mathbf{x})[y]\}+ - \nabla_\theta \ln \sum_{\mathbf{x}} \exp\left\{\mathrm{LogSumExp}_y\{NN_\theta(\mathbf{x})[y]\}\right\} \tag{6.21}$$

$$= \nabla_\theta \mathrm{LogSumExp}_y\{NN_\theta(\mathbf{x})[y]\}+ - \sum_{\mathbf{x}'} \frac{\exp\left\{\mathrm{LogSumExp}_y\{NN_\theta(\mathbf{x}')[y]\}\right\}}{\sum_{\mathbf{x}'',y''} \exp\{NN_\theta(\mathbf{x}'')[y'']\}} \nabla_\theta \mathrm{LogSumExp}_y\{NN_\theta(\mathbf{x}')[y]\} \tag{6.22}$$

$$= \nabla_\theta \mathrm{LogSumExp}_y\{NN_\theta(\mathbf{x})[y]\}+ - \mathbb{E}_{\mathbf{x}'\sim p_\theta(\mathbf{x})}\left[\nabla_\theta \mathrm{LogSumExp}_y\{NN_\theta(\mathbf{x}')[y]\}\right]. \tag{6.23}$$

We can decipher what has just happened here. The gradient of the first part, $\nabla_\theta \mathrm{LogSumExp}_y\{NN_\theta(\mathbf{x})[y]\}$, is calculated for a given datapoint $\mathbf{x}$. The log-sum-exp function is differentiable, so we can apply autograd tools. However, the second part, $\mathbb{E}_{\mathbf{x}'\sim p_\theta(\mathbf{x})}\left[\nabla_\theta \mathrm{LogSumExp}_y\{NN_\theta(\mathbf{x}')[y]\}\right]$, is a totally different story for two reasons:

- First, the gradient of the logarithm of the partition function turns into the expected value over $\mathbf{x}$ distributed according to the model! That is really a problem because the expected value cannot be analytically calculated and sampling from the marginal distribution $p_\theta(\mathbf{x})$ is non-trivial.
- Second, we need to calculate the expected value of the log-sum-exp of $NN_\theta(\mathbf{x})$. That is good news because we can do it using automatic differentiation tools.

Thus, the only problem lies in the expected value. Typically, it is approximated by Monte Carlo samples, however, it is not clear how to sample effectively and efficiently from an EBM. Grathwohl et al. [5] proposes to use the Langevin dynamics [6] that is an MCMC method. The Langevin dynamics in our case starts with a randomly initialized $\mathbf{x}_0$ and then uses the information about the landscape of the energy function (i.e., the gradient) to seek for new $\mathbf{x}$, that is

$$\mathbf{x}_{t+1} = \mathbf{x}_t + \alpha \nabla_{\mathbf{x}_t} \mathrm{LogSumExp}_y \{NN_\theta(\mathbf{x})[y]\} + \sigma \cdot \epsilon, \tag{6.24}$$

where $\alpha > 0$, $\sigma > 0$, and $\epsilon \sim \mathcal{N}(0, I)$. The Langevin dynamics could be seen as the stochastic gradient descent in the observable space with a small Gaussian noise added at each step. Once we apply this procedure for η steps, we can approximate the gradient as follows:

$$\nabla_\theta \ln p_\theta(\mathbf{x}) \approx \nabla_\theta \mathrm{LogSumExp}_y\{NN_\theta(\mathbf{x})[y]\} - \nabla_\theta \mathrm{LogSumExp}_y\{NN_\theta(\mathbf{x}_\eta)[y]\}, \tag{6.25}$$

where $\mathbf{x}_\eta$ denotes the last step of the Langevin dynamics procedure.

We are ready to put it all together! Please remember that our training objective is the following:

$$\ln p_\theta(\mathbf{x}, y) = \ln \mathrm{softmax}\{NN_\theta(\mathbf{x})[y]\} + \Big(\mathrm{LogSumExp}_y\{NN_\theta(\mathbf{x})[y]\} - \ln Z_\theta\Big), \tag{6.26}$$

where the first part is for learning a classifier, and the second part is for learning a generator (so to speak). As a result, we can say we have a sum of two objectives for a fully shared model. The gradient with respect to the parameters is the following:

$$\begin{aligned}\nabla_\theta \ln p_\theta(\mathbf{x}, y) =& \nabla_\theta \ln \mathrm{softmax}\{NN_\theta(\mathbf{x})[y]\} + \\ &+ \nabla_\theta \mathrm{LogSumExp}_y\{NN_\theta(\mathbf{x})[y]\} + \\ &- \mathbb{E}_{\mathbf{x}' \sim p_\theta(\mathbf{x})}\left[\nabla_\theta \mathrm{LogSumExp}_y\{NN_\theta(\mathbf{x}')[y]\}\right].\end{aligned} \tag{6.27}$$

The last two components come from calculating the gradient of the marginal distribution. Remember that the problematic part is only the last component! We will approximate this part using the Langevine dynamics (i.e., a sampling procedure) and a single sample. The final training procedure is the following:

1. Sample $\mathbf{x}_n$ and y_n from a dataset.

2. Calculate $NN_{\theta}(\mathbf{x}_n)[y]$.
3. Initialize $\mathbf{x}_0$ using, e.g., a uniform distribution.
4. Run the Langevin dynamics for η steps:

$$\mathbf{x}_{t+1} = \mathbf{x}_t + \alpha \nabla_{\mathbf{x}_t} \mathrm{LogSumExp}_y \left\{ NN_{\theta}(\mathbf{x})[y] \right\} + \sigma \cdot \epsilon. \tag{6.28}$$

5. Calculate the objective:

$$L_{clf}(\theta) = \sum_{y} \mathbf{1}[y = y_n]\, \theta_y \ln \left\{ NN_{\theta}(\mathbf{x}_n)[y] \right\} \tag{6.29}$$

$$L_{gen}(\theta) = \mathrm{LogSumExp}_y\{NN_{\theta}(\mathbf{x})[y]\} - \mathrm{LogSumExp}_y\{NN_{\theta}(\mathbf{x}_{\eta})[y]\} \tag{6.30}$$

$$L(\theta) = L_{clf}(\theta) + L_{gen}(\theta). \tag{6.31}$$

6. Apply the autograd tool to calculate gradients $\nabla_{\theta} L(\theta)$ and update the neural network.

Notice that $L_{clf}(\theta)$ is nothing else than the cross-entropy loss, and $L_{gen}(\theta)$ is a (crude) approximation to the log-marginal distribution over $\mathbf{x}$'s.

6.4 Code

What do we need to code then? First, we must specify the neural network that defines the energy function. (let us call it the *energy net*.) Classifying using the energy net is rather straightforward. The main problematic part is sampling from the model using the Langevin dynamics. Fortunately, the autograd tools allow us to easily access the gradient with respect $\mathbf{x}$! In fact, it is a single line in the code below. Then we require writing a loop to run the Langevin dynamics for η iterations with the steps size α and the noise level equal σ. In the code, we assume the data are normalized and scaled to $[-1, 1]$ similarly to [5].

```python
class EBM(nn.Module):
    def __init__(self, energy_net, alpha, sigma, ld_steps, D):
        super(EBM, self).__init__()

        print('EBM by JT.')

        # the neural net used by the EBM
        self.energy_net = energy_net

        # the loss for classification
        self.nll = nn.NLLLoss(reduction='none')  # it requires
    log-softmax as input!!

        # hyperparams
        self.D = D

        self.sigma = sigma

        self.alpha = torch.FloatTensor([alpha])

        self.ld_steps = ld_steps

    def classify(self, x):
        f_xy = self.energy_net(x)
        y_pred = torch.softmax(f_xy, 1)
        return torch.argmax(y_pred, dim=1)

    def class_loss(self, f_xy, y):
        # - calculate logits (for classification)
        y_pred = torch.softmax(f_xy, 1)

        return self.nll(torch.log(y_pred), y)

    def gen_loss(self, x, f_xy):
        # - sample using Langevin dynamics
        x_sample = self.sample(x=None, batch_size=x.shape[0])

        # - calculate f(x_sample)[y]
        f_x_sample_y = self.energy_net(x_sample)

        return -(torch.logsumexp(f_xy, 1) - torch.logsumexp(
    f_x_sample_y, 1))

    def forward(self, x, y, reduction='avg'):
        # =====
        # forward pass through the network
        # - calculate f(x)[y]
        f_xy = self.energy_net(x)

        # =====
        # discriminative part
        # - calculate the discriminative loss: the cross-entropy
        L_clf = self.class_loss(f_xy, y)

        # =====
        # generative part
        # - calculate the generative loss: E(x) - E(x_sample)
        L_gen = self.gen_loss(x, f_xy)

        # =====
        # Final objective
        if reduction == 'sum':
            loss = (L_clf + L_gen).sum()
        else:
            loss = (L_clf + L_gen).mean()

        return loss

    def energy_gradient(self, x):
        self.energy_net.eval()

        # copy original data that doesn’t require grads!
        x_i = torch.FloatTensor(x.data)
        x_i.requires_grad = True  # WE MUST ADD IT, otherwise
    autograd won’t work

        # calculate the gradient
        x_i_grad = torch.autograd.grad(torch.logsumexp(self.
    energy_net(x_i), 1).sum(), [x_i], retain_graph=True)[0]

        self.energy_net.train()

        return x_i_grad

    def langevine_dynamics_step(self, x_old, alpha):
        # Calculate gradient wrt x_old
        grad_energy = self.energy_gradient(x_old)
        # Sample eta ~ Normal(0, alpha)
        epsilon = torch.randn_like(grad_energy) * self.sigma

        # New sample
        x_new = x_old + alpha * grad_energy + epsilon

        return x_new

    def sample(self, batch_size=64, x=None):
        # - 1) Sample from uniform
        x_sample = 2. * torch.rand([batch_size, self.D]) - 1.

        # - 2) run Langevin Dynamics
        for i in range(self.ld_steps):
            x_sample = self.langevine_dynamics_step(x_sample,
    alpha=self.alpha)

        return x_sample
```

Listing 6.1 A EBM class

And we are done, this is all we need to have! After running the code (take a look at: https://github.com/jmtomczak/intro_dgm) and training the EBM, we should obtain results similar to those in Fig. 6.2.

6.5 Restricted Boltzmann Machines

The idea of defining a model through the energy function is the foundation of a broad family of Boltzmann machines (BMs) [2, 7]. The Boltzmann machines define an energy function as follows:

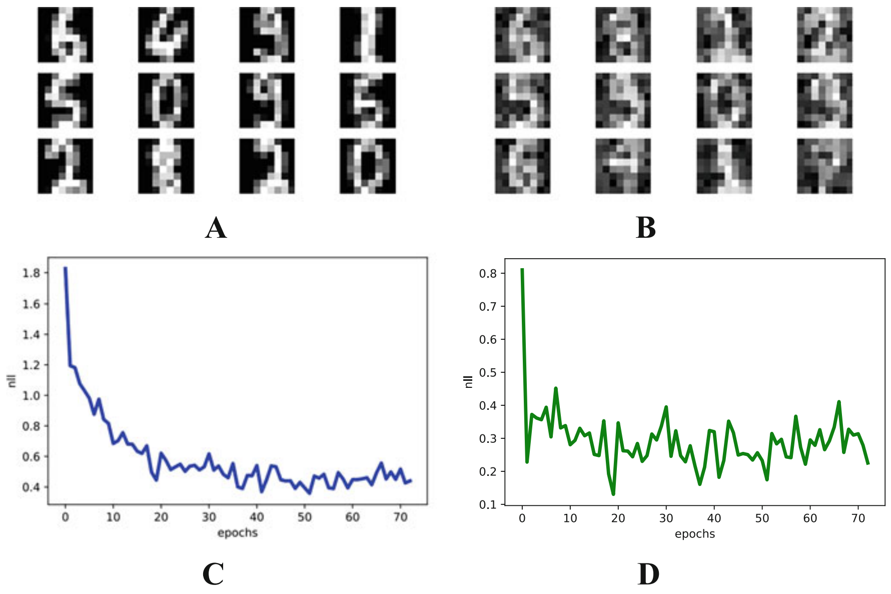

Fig. 6.2 An example of outcomes after the training: (**a**) Randomly selected real images. (**b**) Unconditional generations from the EBM after applying $\eta = 20$ steps of the Langevin dynamics. (**c**) An example of a validation curve of the objective $(L_{clf} + L_{gen})$. (**d**) An example of a validation curve of the generative objective L_{gen}

$$E(\mathbf{x}; \theta) = -\left(\mathbf{x}^{\top}\mathbf{W}\mathbf{x} + \mathbf{b}^{\top}\mathbf{x}\right), \tag{6.32}$$

where $\theta = \{\mathbf{W}, \mathbf{b}\}$ and $\mathbf{W}$ is the weight matrix and $\mathbf{b}$ is the bias vector (bias weights), which is the same as that in Hopfield networks and Ising models. The problem with BM is that they are hard to train (due to the partition function). However, we can alleviate the problem by introducing latent variables and restricting connections among observables.

Restricting BMs

Let us consider a BM that consists of binary observable variables $\mathbf{x} \in \{0, 1\}^D$ and binary latent (hidden) variables $\mathbf{z} \in \{0, 1\}^M$. The relationships among variables are specified through the following *energy function*:

$$E(\mathbf{x}, \mathbf{z}; \theta) = -\mathbf{x}^{\top}\mathbf{W}\mathbf{z} - \mathbf{b}^{\top}\mathbf{x} - \mathbf{c}^{\top}\mathbf{z}, \tag{6.33}$$

where $\theta = \{\mathbf{W}, \mathbf{b}, \mathbf{c}\}$ is a set of parameters, $\mathbf{W} \in \mathbb{R}^{D\times M}$, $\mathbf{b} \in \mathbb{R}^D$, and $\mathbf{c} \in \mathbb{R}^M$ are, respectively, weights, observable biases, and hidden biases. For the energy function in Eq. (6.33), RBM is defined by the *Gibbs distribution*:

$$p(\mathbf{x}, \mathbf{z}|\theta) = \frac{1}{Z_\theta}\exp\big(-E(\mathbf{x}, \mathbf{z}; \theta)\big), \tag{6.34}$$

where

$$Z_\theta = \sum_{\mathbf{x}}\sum_{\mathbf{z}}\exp\big(-E(\mathbf{x}, \mathbf{z}; \theta)\big) \tag{6.35}$$

is the *partition function*. The marginal probability over observables (the likelihood of an observation) is

$$p(\mathbf{x}|\theta) = \frac{1}{Z_\theta}\exp\big(-F(\mathbf{x}; \theta)\big), \tag{6.36}$$

where $F(\cdot)$ is the *free energy*:[1]

$$F(\mathbf{x}; \theta) = -\mathbf{b}^\top\mathbf{x} - \sum_j \log\Big(1 + \exp\big(\mathbf{c}_j + (\mathbf{W}_{\cdot j})^\top\mathbf{x}\big)\Big). \tag{6.37}$$

The presented model is called a Restricted Boltzmann Machine (RBM). It possesses the very useful property that the conditional distribution over the hidden variables factorizes given the observable variables and *vice versa*, which yields the following:

$$p(\mathbf{z}_m = 1|\mathbf{x}, \theta) = \mathrm{sigm}\big(\mathbf{c}_m + (\mathbf{W}_{\cdot m})^\top\mathbf{x}\big), \tag{6.38}$$

$$p(\mathbf{x}_d = 1|\mathbf{z}, \theta) = \mathrm{sigm}(\mathbf{b}_d + \mathbf{W}_{d\cdot}\mathbf{z}). \tag{6.39}$$

Learning RBMs
For given data $\mathcal{D} = \{\mathbf{x}_n\}_{n=1}^N$, we can train an RBM using the maximum likelihood approach that seeks the maximum of the log-likelihood function:

$$\ell(\theta) = \frac{1}{N}\sum_{\mathbf{x}_n \in \mathcal{D}} \log p(\mathbf{x}_n|\theta). \tag{6.40}$$

The gradient of the learning objective $\ell(\theta)$ with respect to θ takes the following form:

$$\nabla_\theta \ell(\theta) = -\frac{1}{N}\sum_{n=1}^N \Big(\nabla_\theta F(\mathbf{x}_n; \theta) - \sum_{\hat{\mathbf{x}}} p(\hat{\mathbf{x}}|\theta)\nabla_\theta F(\hat{\mathbf{x}}; \theta)\Big). \tag{6.41}$$

[1] We use the following notation: for given matrix $\mathbf{A}$, $\mathbf{A}_{ij}$ is its element at location (i, j), $\mathbf{A}_{\cdot j}$ denotes its jth column , $\mathbf{A}_{i\cdot}$ denotes its ith row, and for given vector $\mathbf{a}$, $\mathbf{a}_i$ is its ith element.

In general, the gradient in Eq. (6.41) cannot be computed analytically because the second term requires summing over all configurations of observables. One way to sidestep this issue is the standard stochastic approximation of replacing the expectation under $p(\mathbf{x}|\theta)$ by a sum over S samples $\{\hat{\mathbf{x}}_1, \ldots, \hat{\mathbf{x}}_S\}$ drawn according to $p(\mathbf{x}|\theta)$ [8]:

$$\nabla_\theta \ell(\theta) \approx -\Big(\frac{1}{N}\sum_{n=1}^{N} \nabla_\theta F(\mathbf{x}_n; \theta) - \frac{1}{S}\sum_{s=1}^{S} \nabla_\theta F(\hat{\mathbf{x}}_s; \theta)\Big). \tag{6.42}$$

A different approach, *contrastive divergence*, approximates the expectation under $p(\mathbf{x}|\theta)$ in Eq. (6.41) by a sum over samples $\bar{\mathbf{x}}_n$ drawn from a distribution obtained by applying K steps of the block-Gibbs sampling procedure:

$$\nabla_\theta \ell(\theta) \approx -\frac{1}{N}\sum_{n=1}^{N} \Big(\nabla_\theta F(\mathbf{x}_n; \theta) - \nabla_\theta F(\bar{\mathbf{x}}_n; \theta)\Big). \tag{6.43}$$

The original CD [9] used K steps of the Gibbs chain, starting from each datapoint $\mathbf{x}_n$ to obtain a sample $\bar{\mathbf{x}}_n$ and is restarted after every parameter update. An alternative approach, *Persistent Contrastive Divergence* (PCD), does not restart the chain after each update; this typically results in a slower convergence rate but eventually better performance [10].

Defining Higher-Order Relationships Through the Energy Function

The energy function is an interesting concept because it allows modeling higher-order dependencies among variables. For instance, the binary RBM could be extended to third-order multiplicative interactions by introducing two kinds of hidden variables, i.e., subspace units and gate units. The subspace units are hidden variables that reflect variations of a feature, and, thus, they are more robust to invariances. The gate units are responsible for activating the subspace units and they can be seen as pooling features composed of the subspace features.

Let us consider the following random variables: $\mathbf{x} \in \{0, 1\}^D$, $\mathbf{h} \in \{0, 1\}^M$, $\mathbf{S} \in \{0, 1\}^{M \times K}$. We are interested in the situation where there are three variables connected, namely one observable x_i and two types of hidden binary units, a gate unit h_j and a subspace unit s_{jk}. Each gate unit is associated with a group of subspace hidden units. The energy function of a joint configuration is then defined as follows:[2]

$$E(\mathbf{x}, \mathbf{h}, \mathbf{S}; \boldsymbol{\theta}) = -\sum_{i=1}^{D}\sum_{j=1}^{M}\sum_{k=1}^{K} W_{ijk} x_i h_j s_{jk} - \sum_{i=1}^{D} b_i x_i - \sum_{j=1}^{M} c_j h_j - \sum_{j=1}^{M} h_j \sum_{k=1}^{K} D_{jk} s_{jk}, \tag{6.44}$$

[2] Unlike in other cases, we use sums instead of matrix products because now we have third-order multiplications that would complicate the notation.

where the parameters are $\boldsymbol{\theta} = \{\mathbf{W}, \mathbf{b}, \mathbf{c}, \mathbf{D}\}$, where $\mathbf{W} \in \mathbb{R}^{D\times M\times K}$, $\mathbf{b} \in \mathbb{R}^{D}$, $\mathbf{c} \in \mathbb{R}^{M}$, and $\mathbf{D} \in \mathbb{R}^{M\times K}$.

The Gibbs distribution with the energy function in (6.44) is called *subspace Restricted Boltzmann Machine* (subspaceRBM) [11]. For the subspaceRBM the following conditional dependencies hold true:[3]

$$p(x_i = 1|\mathbf{h}, \mathbf{S}) = \mathrm{sigm}\Big(\sum_j \sum_k W_{ijk} h_j s_{jk} + b_i\Big), \tag{6.45}$$

$$p(s_{jk} = 1|\mathbf{x}, h_j) = \mathrm{sigm}\Big(\sum_i W_{ijk} x_i h_j + h_j D_{jk}\Big), \tag{6.46}$$

$$p(h_j = 1|\mathbf{x}) = \mathrm{sigm}\Big(- K \log 2 + c_j + \sum_{k=1}^{K} \mathrm{softplus}\Big(\sum_i W_{ijk} x_i + D_{jk}\Big)\Big), \tag{6.47}$$

which can be straightforwardly used in formulating a contrastive divergence-like learning algorithm. Notice that in (6.47) a term $-K \log 2$ imposes a natural penalty of the hidden unit activation which is linear to the number of subspace hidden variables. Therefore, the gate unit is inactive unless the sum of softplus of total input exceeds the penalty term and the bias term.

The example of the subspaceRBM shows that the energy function is handy and allows the modeling of various stochastic relations. The subspaceRBM was used to model invariance features but other modifications of the energy function in RBMs could be formulated to allow training spatial transformations [12] or spike-and-slab features [13].

6.6 Final Remarks

The paper of [5] is definitely a milestone in the EBM literature because it shows that we can use *any* energy function parameterized by a neural network. However, to get to that point there was a lot of work on the energy-based models.

Restricted Boltzmann Machines RBMs possess a couple of useful traits. First, the bipartite structure helps training that could be further used in formulating an efficient training procedure for RBMs called contrastive divergence [9] that takes advantage of block-Gibbs sampling. As mentioned earlier, a chain is initialized either at a random point or a sample of latents and then, conditionally, the other set of variables are trained. Similar to the ping-pong game, we sample some variables given the others until convergence or until we decide to stop. Second, the distribution

[3] $\mathrm{softplus}(a) = \log\big(1 + \exp(a)\big)$.

over latent variables could be calculated analytically. Moreover, it could be seen as being parameterized by logistic regression. That is an interesting fact that the sigmoid function arises naturally from the definition of the energy function! Third, the restrictions among connections show that we can still build powerful models that are (partially) analytically tractable. This opened a new research direction that aimed for formulating models with more sophisticated structures like spike-and-slab RBMs [13] and higher-order RBMs [11, 12], or RBMs for categorical observables [14] or real-valued observables [15]. Moreover, RBMs could be also modified to handle temporal data [16] that could be applied to, e.g., human motion tracking [17]. The precursor of the idea presented in [5] was the work on classification RBMs [18, 19]. The training of RBMs is based on the MCMC techniques, e.g., the contrastive divergence algorithm. However, RBMs could be trained to achieve specific features by regularization [20] or other learning algorithms like the Perturb-and-MAP method [21, 22], minimum probability flow [23], or other algorithms [8, 24].

Deep Boltzmann Machines A natural extension of BMs are models with a deep architecture or hierarchical BMs. As indicated by many, the idea of hierarchical models plays a crucial role in AI [25], therefore, there are many extensions of BMs with hierarchical (deep) architectures [15, 26, 27].

Training of deep BMs is even more challenging due to the complexity of the partition function [28]. One of the main approaches to the training of deep BMs relies on treating each pair of consecutive layers as an RBM and training them layer by layer where the layer at the lower layer is treated as observed [27]. This procedure was successfully applied in the seminal paper on unsupervised pre-training of neural nets [29].

Approximating the Partition Function The crucial quantity in the EBMs is the partition function because it allows calculating the Boltzmann distribution. Unfortunately, summing over all values of random variables is computationally infeasible. However, we can use one of the available approximation techniques:

- *Variational methods*: There are a few variational methods that lower bound the log-partition function using the Bethe approximation [30, 31] or upper-bound the log-partition function using a tree-reweighted sum-product algorithm [32]. *Perturb-and-MAP methods*: an alternative approach is to relate the partition function to the max-statistics of random variables and apply the Perturb-and-MAP method [33].
- *Stochastic approximations*: a different approach, probably the most straightforward, is to utilize a sampling procedure. One widely used technique is the Annealed Importance Sampling [28].

Some of the approximations are useful for specific BMs, e.g., BMs with binary variables, BMs with a specific structure. In general, however, approximating the partition function remains an open question and is the main road-blocker for using EMBs in practice and on a large scale.

EBMs Are the Future?

There is definitely a lot of potential in EBMs for at least two reasons:

1. They do not require using any fudge factor to balance the classification loss and the generative loss like in the hybrid modeling approach.
2. The results obtained by Grathwohl et al.[5] clearly indicate that EBMs can achieve the SOTA classification error, synthesize images of high fidelity and be of great use for the out-of-distribution selection.

However, there is one main problem that has not been yet solved: The calculation of $p(\mathbf{x})$. As I claim all the time, the deep generative modeling paradigm is useful not only because we can synthesize nice-looking images but rather because an AI system can assess the uncertainty of the surrounding environment and share this information with us or other AI systems. Since calculating the marginal distribution in EBMs is troublesome, it is doubtful we can use these models in many applications. However, it is an extremely interesting research direction, and figuring out how to efficiently calculate the partition function and how to efficiently sample from the model is crucial for training powerful EBMs.

References

1. Yann LeCun, Sumit Chopra, Raia Hadsell, M Ranzato, and F Huang. A tutorial on energy-based learning. *Predicting structured data*, 1(0), 2006.
2. David H Ackley, Geoffrey E Hinton, and Terrence J Sejnowski. A learning algorithm for Boltzmann machines. *Cognitive science*, 9(1):147–169, 1985.
3. Paul Smolensky. Information processing in dynamical systems: Foundations of harmony theory. Technical report, Colorado Univ at Boulder Dept of Computer Science, 1986.
4. Edwin T Jaynes. *Probability theory: The logic of science*. Cambridge university press, 2003.
5. Will Grathwohl, Kuan-Chieh Wang, Joern-Henrik Jacobsen, David Duvenaud, Mohammad Norouzi, and Kevin Swersky. Your classifier is secretly an energy based model and you should treat it like one. In *International Conference on Learning Representations*, 2019.
6. Max Welling and Yee W Teh. Bayesian learning via stochastic gradient Langevin dynamics. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 681–688. Citeseer, 2011.
7. Geoffrey E Hinton, Terrence J Sejnowski, et al. Learning and relearning in Boltzmann machines. *Parallel distributed processing: Explorations in the microstructure of cognition*, 1(282-317):2, 1986.
8. Benjamin Marlin, Kevin Swersky, Bo Chen, and Nando Freitas. Inductive principles for restricted Boltzmann machine learning. In *Proceedings of the thirteenth International Conference on Artificial Intelligence and Statistics*, pages 509–516, 2010.
9. Geoffrey E Hinton. Training products of experts by minimizing contrastive divergence. *Neural computation*, 14(8):1771–1800, 2002.
10. Tijmen Tieleman. Training restricted Boltzmann machines using approximations to the likelihood gradient. In *ICML*, pages 1064–1071, 2008.
11. Jakub M Tomczak and Adam Gonczarek. Learning invariant features using subspace restricted Boltzmann machine. *Neural Processing Letters*, 45(1):173–182, 2017.
12. Roland Memisevic and Geoffrey E Hinton. Learning to represent spatial transformations with factored higher-order Boltzmann machines. *Neural computation*, 22(6):1473–1492, 2010.