Energy-based models

Energy-based models assign an energy value to each possible configuration, where lower energy means higher compatibility with the data. Learning then means shaping the energy function so that real data have low energy, and unlikely configurations have high energy.

1. Hopfield network (associative memory)

The activation of a neuron i is denoted as $s_i \in \{-1, +1\}$ (binary units). The vector of all network activations $\boldsymbol{s}$, also known as the *state* or *configuration* of the network.

$$\begin{aligned}
E(\boldsymbol{s}) &= -\sum_{\substack{i,j\\ i<j}} w_{ij}\, s_i\, s_j - \sum_i b_i\, s_i \\
&= -\frac{1}{2}\sum_{i,j} w_{ij}\, s_i\, s_j - \sum_i b_i\, s_i \\
&= -\frac{1}{2}\boldsymbol{s}^\top \boldsymbol{W} \boldsymbol{s} - \boldsymbol{b}^\top \boldsymbol{s}
\end{aligned}$$

The weights of the network are symmetrical: $w_{ij} = w_{ji}$, and self-weights (the diagonal of the weight matrix) are set to zero: $w_{ii} = 0$.

1.1. Inference (recalling data)

The energy gap of neuron i is the difference in energy between the network state with neuron i being on (1) and neuron i being off (-1):

$$\Delta E_i = E(s_i = -1) - E(s_i = 1)$$

At inference time, if ΔE_i is positive, activate neuron i (*i.e.* set $s_i \leftarrow 1$), otherwise don't activate it (set $s_i \leftarrow -1$).

The goal of inference is to minimize the energy E with respect to $\boldsymbol{s}$ for fixed weights $\boldsymbol{W}$. This is achieved by iterating over all neurons i, computing the energy gap and activating the neuron if the energy gap is positive, and deactivating it if it's non-positive. What is means is that we recall the memory located at that minimum in the energy landscape. You can initialize (and clamp) part of the network configuration close to a memory, and then recall the full memory via inference.

1.2. Training (memorizing data)

Training a Hopfield network means digging wells in the energy landscape such that at inference time, descending the energy landscape corresponds to recalling particular memories. This is achieved using **Hebb's rule**. Recall that Hebbian learning is a form of associative learning ("neurons that fire together, wire together"). If two activations, s_i and s_j agree (they are both 1, or they are both -1), the weight will be 1 (assuming a single training example), and otherwise -1.

$$w_{ij} \leftarrow s_i\, s_j$$

and for m training examples (sum agreement across all examples):

$$w_{ij} \leftarrow s_i^{(1)}\, s_j^{(1)} + \ldots + s_i^{(m)}\, s_j^{(m)}$$

and in vector form (for number of neurons n, number of examples m):

$$\boldsymbol{W} \leftarrow \frac{1}{n} \sum_{k=1}^{m} \boldsymbol{s}^{(k)} \boldsymbol{s}^{(k)\top}$$

$$w_{ii} \leftarrow 0$$

1.3. Hopfield networks as feature learners

Besides associative memory, another use case of Hopfield networks is to clamp some neurons to a given sensory input, and then infer the remaining neurons, treating them as hidden units. These hidden units correspond to a latent representation of the input. But these models are trained somewhat differently, which we will not get into here.

1.4. Stochasticity

Let's make the decision whether to activate a neuron stochastic, rather than deterministic in the energy gap. For this, we simply put the energy gap inside a sigmoid using some temperature constant T. This idea is called **stochastic binary units**.

$$p(s_i = 1) = \sigma(\Delta E_i / T) = \frac{1}{1 + e^{-\Delta E_i / T}}$$

Now, inference is a sampling operation.

Thermal equilibrium means the system's states are sampled according to the Boltzmann distribution. The probabilities of being in different energy states have stabilized.

Once we have stochastic units, we can use **simulated annealing**, which randomly allows for steps that increase energy to escape local minima, while reducing this randomness over time, thereby eventually settling in a minimum.

2. Boltzmann machine

We distinguish between two types of neurons: **visible units** v_i and **hidden units** h_j, and the state is defined as a tuple $(\boldsymbol{v}, \boldsymbol{h})$. The network is still fully connected and the energy is defined identically, except we decompose it explicitly into these terms for visible and hidden units:

$$
\begin{aligned}
E(\boldsymbol{v}, \boldsymbol{h}) = &- \sum_{i,k} w_{ik}^{(vh)}\, v_i\, h_k - \sum_{\substack{i,j \\ i<j}} w_{ij}^{(vv)}\, v_i\, v_j - \sum_{\substack{k,l \\ k<l}} w_{kl}^{(hh)}\, h_k\, h_l \\
&- \sum_i b_i^{(v)}\, v_i - \sum_k b_k^{(h)}\, h_k \\
&= -\frac{1}{2}\, \boldsymbol{v}^\top \boldsymbol{W}_{vh} \boldsymbol{h} - \boldsymbol{v}^\top \boldsymbol{W}_{vv} \boldsymbol{v} - \boldsymbol{h}^\top \boldsymbol{W}_{hh} \boldsymbol{h} - \boldsymbol{b}_v^\top \boldsymbol{v} - \boldsymbol{b}_h^\top \boldsymbol{h}
\end{aligned}
$$

So, from the weights we have defined a notion of energy, and now from energy, we can also define a notion of the probability of a state $(\boldsymbol{v}, \boldsymbol{h})$, by simply plugging the energy in the softmax function. This is called the **Boltzmann distribution**:

$$
p(\boldsymbol{v}, \boldsymbol{h}) = \operatorname{softmax}(-E(\boldsymbol{v}, \boldsymbol{h})) = \frac{e^{-E(\boldsymbol{v}, \boldsymbol{h})}}{\sum_{\boldsymbol{v}', \boldsymbol{h}'} e^{-E(\boldsymbol{v}', \boldsymbol{h}')}} = \frac{1}{Z}\, e^{-E(\boldsymbol{v}, \boldsymbol{h})}
$$

We can marginalize this across all possible $\boldsymbol{h}$ to get the probability of just the visible units (*i.e.* of the actual data):

$$
p(\boldsymbol{v}) = \sum_{\boldsymbol{h}} p(\boldsymbol{v}, \boldsymbol{h})
$$

Now that we have related the energy to the probability, we can train a Boltzmann machine by maximizing the (log) likelihood of the data with respect to the model weights. We won't go through the full derivation here, but importantly for the Boltzmann machine, we can decompose the gradient of the log-likelihood into two terms: the **data term** and the **model term**.

$$\begin{aligned}
\log p(\boldsymbol{v}) &= \log \sum_{\boldsymbol{h}} p(\boldsymbol{v}, \boldsymbol{h}) \\
&= \log \left[\frac{1}{Z} \sum_{\boldsymbol{h}} e^{-E(\boldsymbol{v},\boldsymbol{h})} \right] \\
&= \log \frac{1}{Z} + \log \sum_{\boldsymbol{h}} e^{-E(\boldsymbol{v},\boldsymbol{h})} \\
&= \log \sum_{\boldsymbol{h}} e^{-E(\boldsymbol{v},\boldsymbol{h})} - \log Z \\
&= \log \sum_{\boldsymbol{h}} e^{-E(\boldsymbol{v},\boldsymbol{h})} - \log \sum_{\boldsymbol{v},\boldsymbol{h}} e^{-E(\boldsymbol{v},\boldsymbol{h})}
\end{aligned}$$

$$\begin{aligned}
\frac{\partial \log p(\boldsymbol{v})}{\partial \boldsymbol{W}} &= \frac{\partial}{\partial \boldsymbol{W}} \left[\log \sum_{\boldsymbol{h}} e^{-E(\boldsymbol{v},\boldsymbol{h})} \right] - \frac{\partial}{\partial \boldsymbol{W}} \left[\log \sum_{\boldsymbol{v},\boldsymbol{h}} e^{-E(\boldsymbol{v},\boldsymbol{h})} \right] \\
&= -\sum_{\boldsymbol{h}} p(\boldsymbol{h} \mid \boldsymbol{v}) \frac{\partial E(\boldsymbol{v},\boldsymbol{h})}{\partial \boldsymbol{W}} + \sum_{\boldsymbol{v},\boldsymbol{h}} p(\boldsymbol{v},\boldsymbol{h}) \frac{\partial E(\boldsymbol{v},\boldsymbol{h})}{\partial \boldsymbol{W}} \\
&= \underbrace{-\,\mathbb{E}_{p(\boldsymbol{h}|\boldsymbol{v})} \left[\frac{\partial E(\boldsymbol{v},\boldsymbol{h})}{\partial \boldsymbol{W}} \right]}_{\text{data term}} + \underbrace{\mathbb{E}_{p(\boldsymbol{v},\boldsymbol{h})} \left[\frac{\partial E(\boldsymbol{v},\boldsymbol{h})}{\partial \boldsymbol{W}} \right]}_{\text{model term}}
\end{aligned}$$

$$\begin{aligned}
\frac{\partial \log p(\boldsymbol{v})}{\partial w_{ij}} &= -\mathbb{E}_{p(\boldsymbol{h}|\boldsymbol{v})} \left[-v_i \, h_j \right] + \mathbb{E}_{p(\boldsymbol{v},\boldsymbol{h})} \left[-v_i \, h_j \right] \\
&= \mathbb{E}_{p(\boldsymbol{h}|\boldsymbol{v})} \left[v_i \, h_j \right] - \mathbb{E}_{p(\boldsymbol{v},\boldsymbol{h})} \left[v_i \, h_j \right] \\
&= \langle v_i \, h_j \rangle_{\text{data}} - \langle v_i \, h_j \rangle_{\text{model}}
\end{aligned}$$

$$\boldsymbol{W} \leftarrow \boldsymbol{W} + \alpha \frac{\partial \log p(\boldsymbol{v})}{\partial \boldsymbol{W}}$$

The data term can be thought of as a **Hebbian term** (strengthening patterns found in the data), while the model term acts as an **Anti-Hebbian term** (weakening model-generated samples, or model-induced deviations from patterns found in the data). Maximizing the data log-likelihood then amounts to a form of **contrastive Hebbian learning**.

The problem with this definition of $p(\boldsymbol{v}, \boldsymbol{h})$ is that in the model term of the gradient we have to sum over all possible states $(\boldsymbol{v}, \boldsymbol{h})$, which is computationally intractable. We will look at two solutions to this (that build on top of each other) that allow for the training of Boltzmann machines.

2.1. Sampling instead of summing (Gibbs sampling)

We can use Markov Chain Monte Carlo (MCMC), and a particular variant of MCMC called **Gibbs sampling**, to approximate the model term (for R samples from the model):

$$\mathbb{E}_{p(\boldsymbol{v},\boldsymbol{h})}\left[v_i\, h_j\right] \approx \frac{1}{R}\sum_{r=1}^{R} v_i^{(r)}\, h_j^{(r)}$$

We initialize the network activations randomly, and repeatedly update all units s_i according to the conditional probability of being active given all other units:

$$p(v_i = 1 \mid \boldsymbol{v}, \boldsymbol{h}) = \sigma(b_i^{(v)} + \textstyle\sum_{j \neq i} w_{ij}^{(vv)}\, v_j + \sum_k w_{ik}^{(vh)}\, h_k)$$

$$p(h_k = 1 \mid \boldsymbol{v}, \boldsymbol{h}) = \sigma(b_k^{(h)} + \textstyle\sum_{l \neq k} w_{kl}^{(hh)}\, h_l + \sum_i w_{ik}^{(vh)}\, v_i)$$

Or written more simply by defining the vector of all neurons $\boldsymbol{s}$, regardless of whether they're visible or hidden (using w for the weights between all neurons, and b for biases of all neurons):

$$\begin{aligned} p(s_i = 1 \mid \boldsymbol{s}) &= \sigma(b_i + \textstyle\sum_{j \neq i} w_{ij}\, s_j) \\ &= \sigma(-\Delta E_i) \end{aligned}$$

2.2. Bipartite structure

Instead of having a fully connected network, if we prohibit connections from hidden units to hidden units and from visible units to visible units, we get a bipartite graph, and we can update all hidden units (given the state of visible units) at once, and all visible units (given the state of hidden units) at once. Before this, we had to iterate over all neurons, and update one neuron at a time. By placing this constraint on the structure, we make training and inference more efficient. This model is called a **Restricted Boltzmann machine** and it's so important that the next chapter is dedicated to it.

3. Restricted Boltzmann machine

The energy function of the RBM simplifies to this:

$$\begin{aligned} E(\boldsymbol{v}, \boldsymbol{h}) &= -\sum_{i,k} w_{ik}^{(vh)} \, v_i \, h_k - \sum_i b_i^{(v)} \, v_i - \sum_k b_k^{(h)} \, h_k \\ &= -\frac{1}{2} \boldsymbol{v}^\top \boldsymbol{W}_{vh} \boldsymbol{h} - \boldsymbol{b}_v^\top \boldsymbol{v} - \boldsymbol{b}_h^\top \boldsymbol{h} \end{aligned}$$

The conditional probabilities simplify a lot:

$$p(v_i = 1 \mid \boldsymbol{h}) = \sigma(a_i + \textstyle\sum_j w_{ij} \, h_j)$$

$$p(h_j = 1 \mid \boldsymbol{v}) = \sigma(b_j + \textstyle\sum_i w_{ij} \, v_i)$$

and if $\boldsymbol{v}$ is clamped to some data, inferring $\boldsymbol{h}$ corresponds to encoding the data to a latent representation. And inferring an example $\boldsymbol{v}$ from some latent $\boldsymbol{h}$ corresponds to generating (decoding) a new example. The RBM is therefore a generative model.

$$\boldsymbol{h} \sim p(\boldsymbol{h} \mid \boldsymbol{v})$$

$$\boldsymbol{v} \sim p(\boldsymbol{v} \mid \boldsymbol{h})$$

3.1. Contrastive divergence (CD$_k$)

Compared to naive MCMC, contrastive divergence has a shortcut: when estimating the model term, instead of starting from a random initialization, we start with some actual data, and then encode it and decode it, giving us a model generation, and repeating this k times (not necessarily until convergence).

We're maximizing the likelihood of $\boldsymbol{v}^{(0)}$ (data term) while minimizing the likelihood of the reconstruction $\boldsymbol{v}^{(k)}$ (model term).

1. **Positive phase** (data-driven): clamp $\boldsymbol{v}$ to data, compute $p(h_j \mid \boldsymbol{v})$ for all j, and compute data term $\langle v_i h_j \rangle_{\text{data}}$
2. **Negative phase** (model-driven): initialize Gibbs chain at $\boldsymbol{v}^{(0)}$ (data), and do k steps to sample $\boldsymbol{h}^{(1)}, \boldsymbol{v}^{(1)}, \ldots, \boldsymbol{h}^{(k)}, \boldsymbol{v}^{(k)}$, and then use this to approximate the model term $\langle v_i h_j \rangle_{\text{model}}$
3. **Parameter update**

4. Deep belief network

We can train a RBM to sample $\boldsymbol{h}_1$ from $\boldsymbol{v}_1$ (using weights $\boldsymbol{W}_1$), and after finishing training, we can train another RBM to sample $\boldsymbol{h}_2$ from $\boldsymbol{h}_1$ (using weights $\boldsymbol{W}_2$). The overall model is called a deep belief network.

What is interesting about this approach is that each layer is trained separately, which means it's *greedy training*. Why does this work?

$$p(\boldsymbol{v}) = \sum_{\boldsymbol{h}} p(\boldsymbol{h})\, p(\boldsymbol{v} \mid \boldsymbol{h})$$

Doing greedy training amounts to only optimizing $p(\boldsymbol{h})$, which still improves $p(\boldsymbol{v})$, even if $p(\boldsymbol{v} \mid \boldsymbol{h})$ stays constant.

4.1. Generative model

In order to generate a new instance, we start at the top layer and pass between $\boldsymbol{h}_2$ and $\boldsymbol{h}_3$ (*equilibrium sample*) and then pass it down the layers to get $\tilde{\boldsymbol{v}}$.

4.2. Discriminative model

The above procedure can be done as a form of *greedy pre-training*. Then you can add a classifier or regressor head on top of the final RBM and do *supervised fine-tuning* using proper backpropagation (which is non-greedy, of course).