Jingzhuo Hui
Damoun Ayman
Github repository
Master M2 IA

TCP4
Deep Probabilistice Generative Models
Caio Filippo Corro
November 7, 2020

Lab 2: Restricted Boltzmann Machine

0.1 Preliminaries

The RBM is a two-layered neural network—the first layer is called the visible layer and the second layer is called the hidden layer. They are called shallow neural networks because they are only two layers deep. They were first proposed in 1986 by Paul Smolensky (he called them Harmony Networks [2]) and later by Geoffrey Hinton who in 2006 proposed Contrastive Divergence (CD) as a method to train them. All neurons in the visible layer are connected to all the neurons in the hidden layer, but there is a restriction—no neuron in the same layer can be connected. These methods are applied to imagere cognition, speech recognition, etc., and show state-of-the-art performance in benchmark tests [1].
RBMs can be used for dimensionality reduction, feature extraction, and collaborative filtering. The training of RBMs can be divided into three parts: forward pass, backward pass, and then compare.

- **Forward pass:** The information at visible units (V) is passed via weights (W) and biases (c) to the hidden units (h).
- **Backward pass:**The hidden unit representation (h) is then passed back to the visible units through the same weights, W, but different bias, c, where they reconstruct the input.

Bernoulli-Bernoulli RBM (BBRBM) refers to the RBM assuming that the distribution of both visible and hidden unitsis binary. A Gaussian-Bernoulli RBM (GBRBM) refers to the RBM assuming that the distribution of the visible unit is the Gaussian distribution and that the distribution of the hidden unit is binary.

0.2 Bernoulli-Bernoulli RBM

The restricted Boltzmann machine (RBM) consists of m visible units v=(v_1, ..., v_m) and n hidden units h=(h_1, ..., h_n), with fully connecting between them. But there are no visible to visible and hidden to hidden connections.
The RBM has been mainly developed to model binary variables $(\boldsymbol{v}, \boldsymbol{h})$ which take the binary values of $(\boldsymbol{v}, \boldsymbol{h}) \in \{0,1\}^{m+n}$. A joint probability of $(\boldsymbol{v}, \boldsymbol{h})$ can be expressed with the BBRBM as:

$$p(\boldsymbol{v}, \boldsymbol{h}) = \frac{1}{Z} e^{-E(v,h)} \tag{1}$$

where Z is the normalizing constant and the energy function E

$$E(\boldsymbol{v}, \boldsymbol{h}) = -\sum_{i=1}^{m}\sum_{j=1}^{n} w_{ij} v_i h_j - \sum_{i=1}^{m} b_i v_i - \sum_{j=1}^{n} c_j h_j \tag{2}$$

for all $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$, w_{ij} is a real valued weight associated with the edge between units v_i and h_j and b_i and c_j are real valued bias terms associated with the i -th visible and the j -th hidden variable, respectively. From Eqs. (1) and (2), $E(\boldsymbol{v}, \boldsymbol{h})$ with a low energy are given a high probability.
In terms of probability, the hidden units are independent from the visible units and vice versa, as shown

in Eqs. (3) and (4). When binary data are given in visible units, the conditional probability is estimated from the neural network propagation rule by Eqs. below.

$$
\begin{aligned}
p(\boldsymbol{v} \mid \boldsymbol{h}) &= \textstyle\prod_{i=1} p\left(v_i \mid \boldsymbol{h}\right) \\
p(\boldsymbol{h} \mid \boldsymbol{v}) &= \textstyle\prod_{i=1} p\left(h_j \mid \boldsymbol{v}\right) \\
p\left(v_i = 1 \mid \boldsymbol{h}\right) &= f\left(b_i + \textstyle\sum_{j=1} h_j w_{ij}\right) \\
p\left(h_j = 1 \mid \boldsymbol{v}\right) &= f\left(c_j + \textstyle\sum_{i=1} v_i w_{ij}\right)
\end{aligned}
$$

where $f(\cdot)$ is the sigmoid activation function.
The model with the energy function has been developed tomodel the random binary variables. Therefore, this model isnot suitable to model a continuous value data. To address thisissue, the GBRBM has proposed.

0.3 Gaussian-Bernoulli RBM

The Gaussian-Bernoulli RBM (GBRBM) has visible units with real-value v_m and binary hidden units h_n. Based on the same idea as the BBRBM, the energy function of the GBRBM is defined as

$$
E(\boldsymbol{v}, \boldsymbol{h}) = -\sum_{i=1}^{m} \sum_{j=1}^{n} w_{ij} h_j \frac{v_i}{\sigma_i} - \sum_{i=1}^{m} \frac{\left(v_i - b_i\right)^2}{2\sigma_i^2} - \sum_{j=1}^{n} c_j h_j
$$

where b_i and c_j are biases corresponding respectively to visible and hidden units, w_{ij} are the connecting weights between the visible and hidden units and σ_i is the standard deviation associated with Gaussian visible units v_i. Conditional probabilities for visible and hidden units are

$$
\begin{aligned}
p\left(v_i = v \mid \boldsymbol{h}\right) &= N\left(\boldsymbol{v} \mid b_i + \sum_j h_j w_{ij}, \sigma_i^2\right) \\
p\left(h_j = 1 \mid \boldsymbol{v}\right) &= f\left(c_j + \sum_i w_{ij} \frac{v_i}{\sigma_i^2}\right)
\end{aligned}
$$

where $N\left(\cdot \mid \mu, \sigma^2\right)$ denotes the Gaussian pro density function with mean μ and standard deviation σ.

0.4 Implementation

In this lab exercise, we coded a Restricted Boltzmann Machine with Gaussian observed random variables and Bernoulli latent variables. The data we used in the lab exercise looks like a ring. The architecture of the RMB is is shown below:

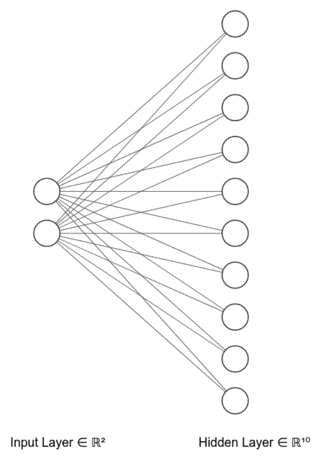

Figure 1: the architecture of RMB implemented

We define a class RBM . The class __init_() function initializes the number of neurons in the visible layer (input_size) and the number of neurons in the hidden layer (output_size). The function initializes the weights and biases for both hidden and visible layers. In the following code we have initialized them to zero.

The free energy is computed as folow:

$$F(x) = -\log \sum_h \exp(-E(x,h)) \quad (3)$$

$$= -a^\top x - \sum_j \log(1 + \exp(W_j^\top x + b_j)) \,. \quad (4)$$

```python
# returns the free energy of x
def forward(self, x):
    vbias_term = x.mv(self.b)
    wx_b = F.linear(x/torch.exp(self.log_sigma_squared),self.W.t(),self.c)
    hidden_term = wx_b.exp().add(1).log().sum(1)
    return - vbias_term - hidden_term
```

We define methods to provide the forward and backward passes:

```python
# compute the parameters of the conditional distribution p(x | z)
def p_x_given_z(self, z):
    sigma_squared = self.log_sigma_squared.exp()
    wz = torch.mm(z, self.W.t())
    mu = F.linear(z,self.W,self.b)
    return mu, sigma_squared.unsqueeze(0).repeat(z.shape[0], 1)
```

```python
# compute the parameters of the Bernoulli conditional distribution p(z | x)
def p_z_given_x(self, x):
    mean = torch.sigmoid(self.c + (x/torch.exp(self.log_sigma_squared)) @ self.W)
    return mean
```

The training loop:

```python
for _ in range(n_epoch):
    for i in range(0, target_samples.shape[0], batch_size):
        optimizer.zero_grad()
        data_samples = target_samples[i:i+batch_size]

        with torch.no_grad():
            # generate sample from your model!
            # and store the in a variable called model_samples
            # BEGIN TODO
            p_h = machine.p_z_given_x(data_samples)
            z = torch.bernoulli(p_h)
            mu, sigma_squared = machine.p_x_given_z(z)
            model_samples = torch.normal(mu,sigma_squared)
            # END TODO

        # the loss function is quite simple
        loss = (machine(data_samples) - machine(model_samples)).mean()
        losses.append(loss.item())

        loss.backward()
        # we use gradient clipping to stabilize training
        torch.nn.utils.clip_grad_norm_(machine.parameters(), 1)
        optimizer.step()
```

Sampling in an RBM:

samples can be obtained by running a Markov chain to convergence, using Gibbs sampling as the transition operator. ibbs sampling of the joint of N random variables $S = (S_1, ..., S_N)$ is done through a sequence of N sampling sub-steps of the form $S_i \sim p(S_i|S_{-i})$ where S_{-i} contains the N-1 other random variables in S excluding S_i.

For RBMs, S consists of the set of visible and hidden units. However, since they are conditionally independent, one can perform block Gibbs sampling. In this setting, visible units are sampled simultaneously given fixed values of the hidden units. Similarly, hidden units are sampled simultaneously given the visibles. A step in the Markov chain is thus taken as follows:

$$h^{(n+1)} \sim \text{sigm}\left(W'v^{(n)} + c\right)$$
$$v^{(n+1)} \sim \text{sigm}\left(Wh^{(n+1)} + b\right)$$

where $h^{(n)}$ refers to the set of all hidden units at the n-th step of the Markov chain. What it means is that, for example, $h_i^{(n+1)}$ is randomly chosen to be 1 (versus 0) with probability $sigm(W_i'v^{(n)} + c_i)$, and similarly, $v_j^{(n+1)}$ is randomly chosen to be 1 (versus 0) with probability $sigm(W_{.j}h^{(n+1)} + b_j)$.

This can be illustrated graphically:

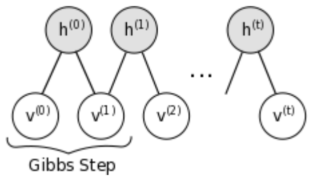

Figure 2: markov chain

Results

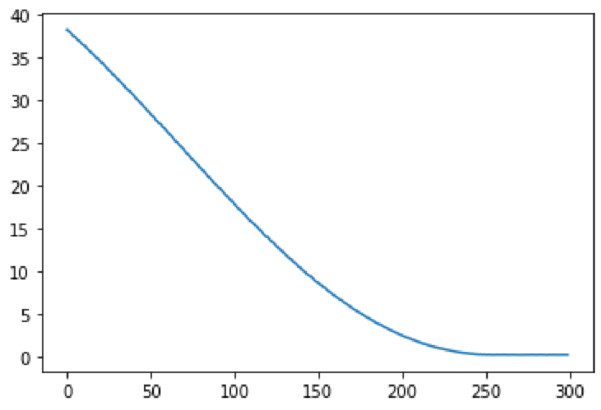

Figure 3: plot the training loss

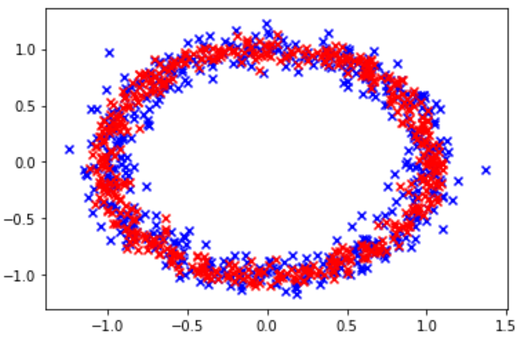

Figure 4: plot the samples at the last timestep of each markov chaine

Bibliography

[1] Alex Krizhevsky. “Learning Multiple Layers of Features from Tiny Images”. In: *University of Toronto* (May 2012).

[2] Jonathon Shlens. “A Tutorial on Principal Component Analysis”. In: *CoRR* abs/1404.1100 (2014). arXiv: 1404.1100. URL: http://arxiv.org/abs/1404.1100.