Deep Generative Models

We are interested in the joint distribution of the random cause if we know the joint distribution then we can infer the information we need. For example if we need to classify 'y' given some random variable $x_1, x_2, \ldots, x_n$. The we can find it using the joint distribution as:

$$P(y|x_1,x_2,\ldots,x_n)=\frac{P(y,x_1,x_2,\ldots,x_n)}{P(x_1,x_2,\ldots,x_n)}$$
$$=\frac{P(y,x_1,x_2,\ldots,x_n)}{\sum_y P(y,x_1,x_2,\ldots,x_n)}$$

We can also find the marginal distribution of 'y' i.e. $P(y)=\sum_{x_1,x_2,\ldots x_n} P(x_1,x_2,\ldots,x_n)$

How do we represent Joint distribution ?

Suppose there are n random variables $x_1,x_2,\ldots,x_n$. For simplicity we will assume they take binary values 0 and 1. To specify the joint distribution, we need to specify 2^n-1 values.

Challenges with explicit representation :-

1)Computational : Expensive to manipulate and too large to to store

2)Cognitive : Impossible to acquire so many numbers from a human

3)Statistical : Need huge amounts of data to learn the parameters

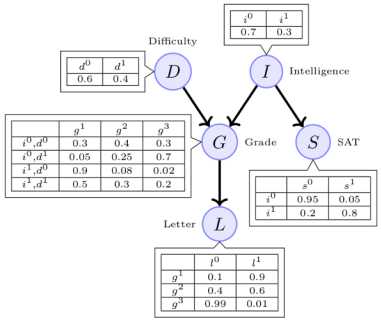
To represent Joint distribution, we need conditional independence between random variable.

Using Graph to learn joint Distribution

Bayesian Network In Bayesian Network , the nodes are the random variables and the edges tells us the dependencies.

The Bayesian network can be viewed as a data structure. It provides a skeleton for representing a joint distribution compactly by factorization.

Each node is associated with a local probability model (because it represents the dependencies of each variable on its parents).There are 5 such local probability models associated with the graph on the right. Each variable is associated with a conditional probability distribution. The graph gives us a natural factorization for the joint distribution



P(I,D,G,S,L) = P(I) P(D) P(G|I,D) P(S|I) P(L|G)

Example, $P(I=1,D=0,G=g^2,S=1,L=0)=0.3\times0.6\times0.08\times0.8\times0.4$

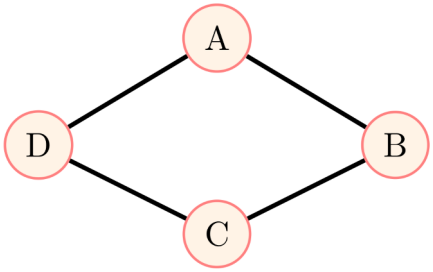
Independencies encoded by Bayesian Network :

A Bayesian Network structure G is a directed acyclic graph where nodes represent random variables $X_1,X_2,\ldots,X_n$. Let $Pa_G(X_i)$ denote the parents of X_i in G and NonDescendants(X_i) denote the variables in the graph that are not descendants of X_i . Then G encodes the following set of conditional independence assumptions called the local independencies and denoted by $I_i(G)$ for each variable X_i.

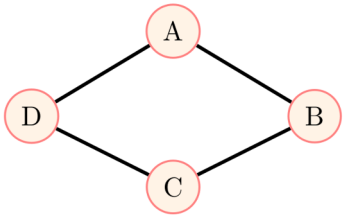
$(X_i \perp \text{NonDescendants}(X_i) \mid \text{Pa}_G(Xi))$

Notation $A \perp B \mid C$: A is independent of B given C

Markov Network In Markov Network, we have undirected graph and instead of capturing dependencies, we capture strength interactions between the random variables.



We want factors to capture interactions (affinity) between connected nodes. We have factors $\phi_1(A,B), \phi_2(B,C), \phi_3(C,D), \phi_4(D,A)$ which capture the affinity between the corresponding nodes.



$\phi_1(A,B)$			$\phi_2(B,C)$			$\phi_3(C,D)$			$\phi_4(D,A)$		
a^0	b^0	30	a^0	b^0	100	a^0	b^0	1	a^0	b^0	100
a^0	b^1	5	a^0	b^1	1	a^0	b^1	100	a^0	b^1	1
a^1	b^0	1	a^1	b^0	1	a^1	b^1	100	a^1	b^0	1
a^1	a^1	10	a^1	b^1	100	a^1	b^1	1	a^1	b^1	100

$\phi_1(A,B)$ asserts that it is more likely for A and B to agree

[$\because$ weights for a^0b^0,a^1b^1 > a^0 b^1,a^1b^0]

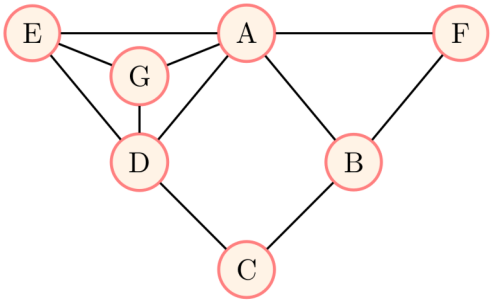
These tables do not represent probability distributions. They are just weights which can be interpreted as the relative likelihood of an event.

But we can write this as probability distribution as

$$P(a,b,c,d)=\frac{1}{Z}\phi_1(a,b)\phi_1(b,c)\phi_1(c,d)\phi_1(d,a)$$

where,

$$Z=\sum_{a,b,c,d}\phi_1(a,b)\phi_1(b,c)\phi_1(c,d)\phi_1(d,a)$$



One more Example with more number of nodes

Instead of having factors $\phi(E,G)\phi(G,A)\ldots$,we can have factors which corresponds to the maximal clique. That is we can have our joint distribution as :

$$P(A,B,\ldots,G)=\frac{1}{Z}\phi(A,E,G,D)\phi(A,B,F)\phi(C,D)\phi(B,C)$$

Formally, we can say A distribution factorizes over a Markov Network H if P can be expressed as

$$P(X_1,\ldots,X_n)=\frac{1}{Z}\prod_{i=1}^{m}\phi(D_i)$$

where each D i is a complete sub-graph (or maximal clique) in H.

Local Independencies in Markov Model :

Let U be the set of all random variables in our joint distribution. Let X, Y, Z be some distinct subsets of U. A distribution P over these Random Variables would imply X⊥Y |Z if and only if, we can write

$$P(X)=\phi_1(X,Z)\phi_2(Y,Z)$$

For a given Markov network H, we define Markov Blanket of a Random Variable X to be the neighbors of X in H . Analogous to the case of Bayesian Networks we can define the local independences associated with H to be:

$$X \perp (U-\{X\}-MB_H) \mid MB_H(X)$$

Concept of Latent Variable

Consider Natural Images of size 32*32, the neighboring pixels in the image are dependent on each other(because we expect them to have the same color, texture, etc.)

We can say that there are certain underlying hidden (latent) characteristics which are determining the pixels and their interactions. We could think of these as additional (latent) random variables in our distribution These are latent because we do not observe them unlike the pixels which are observable random variables. The pixels depend on the choice of these latent variables.

More formally we now have visible (observed) variables or pixels $V=\{V_1,V_2,V_3,\ldots,V_{1024}\}$ and hidden variables $H=\{H_1,H_2,\ldots,H_n\}$. We can think of a different Markov network involving these latent variables where the pixels (observed variables) are dependent on the latent variables or the interactions between the pixels are captured through the latent variables

Two main concepts related to these latent variables are :

- Abstraction : Suppose, we are able to learn the joint distribution P (V, H). Using this distribution we can find

$$P(H|V)=\frac{P(V,H)}{\sum_H P(V,H)}$$

In other words, given an image, we can find the most likely latent configuration (H = h) that generated this image, h captures a latent representation or abstraction of the image! In other words, it captures the most important properties of the image

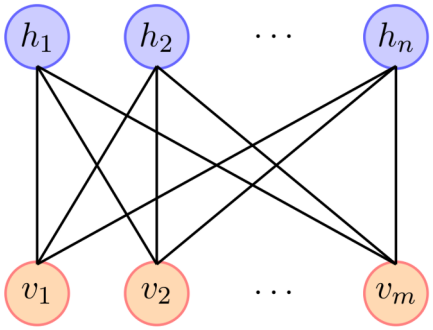
- Generation : Assume that we are able to learn the joint distribution P (V, H). Using this distribution we can find

$$P(V|H)=\frac{P(V,H)}{\sum_V P(V,H)}$$

In other words, we can generate images given certain latent variables,or given h = [….] find the corresponding V, which maximizes P (V |H)

From now ,we will assume that all our variables take only boolean values. Thus, the vector V will be a boolean vector $\in \{0, 1\}^m$ (there are a total of 2^m values that V can take) and the vector H will be a boolean vector $\in \{0, 1\}^n$ (there are a total of 2^n values that H can take).

Restricted Boltzmann Machine



We look at Markov Network containing hidden variables and visible variables (We will get rid of the image and just keep the hidden and latent variables) We have edges between each pair of (hidden, visible) variables. We do not have edges between (hidden, hidden) and (visible, visible) variables.
In Markov network the joint probability distribution can be written as a product of factors corresponding to maximal cliques (here every pair of visible and hidden node forms a clique Total = m × n)
We can write the joint pdf as a product of the following factors

$$P(V,H)=\frac{1}{Z}\prod_i\prod_j \phi_{ij}(v_i,h_j)$$

We can also add additional corresponding to the nodes and write

$$P(V,H)=\frac{1}{Z}\prod_i\prod_j \phi_{ij}(v_i,h_j)\prod_i \psi_i(v_i)\prod_j \xi_j(h_j)$$

Z is the partition function and is given by

$$Z=\sum_V\sum_H\prod_i\prod_j \phi_{ij}(v_i,h_j)\prod_i \psi_i(v_i)\prod_j \xi_j(h_j)$$

$\phi_{ij}(v_i,h_j)$ is a factor which takes the values of $v_i \in \{0,1\}$ and $h_j \in \{0,1\}$ and returns a value indicating the affinity between these two variables. Similarly, $\psi_i(v_i)$ takes the value of $v_i \in \{0,1\}$ and gives us a number which roughly indicates the possibility of v_i taking on the value 1 or 0. A similar interpretation can be made for $\xi_j(h_j)$
We will introduce a parametric form for these clique potentials and then learn these parameters
The specific parametric form chosen by RBMs is

$$\phi_{ij}(v_i,h_j)=e^{w_{ij}v_ih_j}$$
$$\psi_i(v_i)=e^{b_iv_i}$$
$$\xi_j(h_j)=e^{c_jh_j}$$

The Joint Distribution looks like,

$$P(V,H)=\frac{1}{Z}\prod_i\prod_j \phi_{ij}(v_i,h_j)\prod_i \psi_i(v_i)\prod_j \xi_j(h_j)$$
$$=\frac{1}{Z}\prod_i\prod_j e^{w_{ij}v_ih_j}\prod_i e^{b_iv_i}\prod_j e^{c_jh_j}$$

$$P(V,H)=\frac{1}{Z}\exp\left(\sum_i\sum_j w_{ij}v_ih_j\right)\exp\left(\sum_i b_iv_i\right)\exp\left(\sum_j c_jh_j\right)$$
$$=\frac{1}{Z}\exp\left(\sum_i\sum_j w_{ij}v_ih_j + \sum_i b_iv_i + \sum_j c_jh_j\right)$$
$$=\frac{1}{Z}e^{-E(V,H)}$$

where, $E(V,H)=-\sum_i\sum_j w_{ij}v_ih_j - \sum_i b_iv_i - \sum_j c_jh_j$

RBM as Stochastic Neural Network

We derive the formula for P(V|H) and P(H|V)

Define $\alpha_l(H)=-\sum_{i=1}^{n} w_{il}h_i-b_l \;\;;\;\; \beta(V_{-l},H)=-\sum_{i=1}^{n}\sum_{j=1,\,j\neq l}^{m} w_{ij}h_iv_j-\sum_{j=1,\,j\neq l}^{m} b_iv_i-\sum_{i=1}^{n} c_ih_i$

Notice $E(V,H)=v_l\,\alpha(H)+\beta(V_{-l},H)$

We can write $p(v_l=1|H)$ as

$$p(v_l=1|H)=P(v_l=1|V_{-l},H)$$
$$=\frac{p(v_l=1,V_{-l},H)}{p(V_{-l},H)}$$
$$=\frac{e^{-E(v_l=1,V_{-l},H)}}{e^{-E(v_l=1,V_{-l},H)}+e^{-E(v_l=0,V_{-l},H)}}$$
$$=\frac{e^{-\beta(V_{-l},H)-1.\,\alpha_l(H)}}{e^{-\beta(V_{-l},H)-1.\,\alpha_l(H)}+e^{-\beta(V_{-l},H)-0.\,\alpha_l(H)}}$$
$$=\frac{e^{-\beta(V_{-l},H)}e^{-\alpha_l(H)}}{e^{-\beta(V_{-l},H)}e^{-\alpha_l(H)}+e^{-\beta(V_{-l},H)}}$$
$$=\frac{e^{-\alpha_l(H)}}{e^{-\alpha_l(H)}+1}$$
$$=\frac{1}{1+e^{\alpha_l(H)}}$$
$$=\sigma(-\alpha_l(H))$$
$$=\sigma\left(\sum_{i=1}^{n} w_{il}h_i+b_l\right)$$

Similarly, we can show that $p(h_l|V)=\sigma(\sum_{i=1}^{m} w_{il}v_i+c_l)$

The RBM can thus be interpreted as a stochastic neural network, where the nodes and edges correspond to neurons and synaptic connections, respectively. The conditional probability of a single (hidden or visible) variable being 1 can be interpreted as the firing rate of a (stochastic) neuron with sigmoid activation function

Unsupervised Learning with RBM

We are interested in learning P (v, h) which we have parameterized as

$$P(V,H)=\frac{1}{Z}e^{-\left(-\sum_i\sum_j w_{ij}v_ih_j-\sum_i b_iv_i-\sum_j c_jh_j\right)}$$

We would want P(X=x) to be maximum for any x belonging to our training data to be high

So now can you think of an objective function,

$$maximize \prod_{i=1}^{N} P(X = x_i)$$

or log likelihood

$$\ln L(\theta) = \ln \prod_{i=1}^{N} p(x_i|\theta) = \sum_{i=1}^{N} \ln p(x_i|\theta)$$

For learning we will consider the loss for a single training example

$$\begin{aligned}\ln L(\theta) &= \ln p(V|\theta) = \ln \frac{1}{Z} \sum_{H} e^{-E(V,H)} \\ &= \ln \sum_{H} e^{-E(V,H)} - \ln \sum_{V,H} e^{-E(V,H)}\end{aligned}$$

$$\begin{aligned}\frac{\partial \ln L(\theta)}{\partial \theta} &= \frac{\partial}{\partial \theta}\left(\ln \sum_{H} e^{-E(V,H)} - \ln \sum_{V,H} e^{-E(V,H)}\right) \\ &= -\frac{1}{\sum_{H} e^{-E(V,H)}} \sum_{H} e^{-E(V,H)} \frac{\partial E(V,H)}{\partial \theta} + \frac{1}{\sum_{V,H} e^{-E(V,H)}} \sum_{V,H} e^{-E(V,H)} \frac{\partial E(V,H)}{\partial \theta} \\ &= -\sum_{H} \frac{e^{-E(V,H)}}{\sum_{H} e^{-E(V,H)}} \frac{\partial E(V,H)}{\partial \theta} + \sum_{V,H} \frac{e^{-E(V,H)}}{\sum_{V,H} e^{-E(V,H)}} \frac{\partial E(V,H)}{\partial \theta}\end{aligned}$$

Now,

$$\frac{e^{-E(V,H)}}{\sum_{V,H} e^{-E(V,H)}} = p(V,H)$$

$$\begin{aligned}\frac{e^{-E(V,H)}}{\sum_{H} e^{-E(V,H)}} &= \frac{\frac{1}{Z} e^{-E(V,H)}}{\frac{1}{Z} \sum_{H} e^{-E(V,H)}} \\ &= \frac{p(V,H)}{P(V)} = P(H|V)\end{aligned}$$

$$\begin{aligned}\frac{\partial \ln L(\theta)}{\partial \theta} &= -\sum_{H} \frac{e^{-E(V,H)}}{\sum_{H} e^{-E(V,H)}} \frac{\partial E(V,H)}{\partial \theta} + \sum_{V,H} \frac{e^{-E(V,H)}}{\sum_{V,H} e^{-E(V,H)}} \frac{\partial E(V,H)}{\partial \theta} \\ &= -\sum_{H} p(H|V) \frac{\partial E(V,H)}{\partial \theta} + \sum_{V,H} p(V,H) \frac{\partial E(V,H)}{\partial \theta}\end{aligned}$$

θ is a collection of all the parameters in our model, i.e., $w_{ij}, b_i, c_j \, \forall \, i \in \{1, \ldots, m\}; \forall \, j \in \{1, \ldots, n\}$

Parial derivative wrt w_{ij}

$$\begin{aligned}\frac{\partial \ln L(\theta)}{\partial w_{ij}} &= -\sum_{H} p(H|V) \frac{\partial E(V,H)}{\partial w_{ij}} + \sum_{V,H} p(V,H) \frac{\partial E(V,H)}{\partial w_{ij}} \\ &= \sum_{H} p(H|V) h_i v_j - \sum_{V,H} p(V,H) h_i v_j \\ &= E_{p(h|V)}[v_i h_j] - E_{p(V,H)}[v_i h_j]\end{aligned}$$

The first summation can actually be simplified. However, the second summation contains an exponential number of terms and hence intractable in practice. To deal with this we have do sampling. Consider the case that visible variables correspond to pixels from natural images) Clearly some images are more likely than the others!

Hence, we cannot assume that all samples are equally likely. We need to draw more samples from the high probability region and fewer samples from the low probability region. To draw a sample (V, H), we need to know its probability P(V, H) And of course, we also need this P(V, H) to compute the expectation But, unfortunately computing P (V, H) is intractable because of the partition function Z Also, approximating the summation by using a few samples is not straightforward!
Solution : Gibbs Sampling [Draw samples from an easier distribution (say, Q) as long as I am sure that if I keep drawing samples from Q eventually my samples will start looking as if they were drawn from P]

Gibbs Sampling

Suppose instead of a single random variable $X \in \Re^n$, we have a chain of random variables $X_1, X_2, \ldots, X_K$ each $X_i \in \Re^n$ The 'i' here corresponds to a time step For example, X_i could be a n-dimensional vector containing the number of customers in a given set of n restaurants on day 'i' or X_i could be a 1024 dimensional image. For ease of illustration we will stick to the restaurant example and assume that instead of actual counts we are interested only in binary counts (high=1, low=0) , Thus $X_i \in \{0,1\}^n$

Formally, we may be interested in the following distribution

$$P(X_i = x_i | X_1 = x_1, X_2 = x_2, \ldots, X_{i-1} = x_{i-1})$$

Suppose the chain exhibits the Markov property

$$P(X_i = x_i \mid X_1 = x_1, X_2 = x_2, \ldots, X_{i-1} = x_{i-1}) = P(X_i = x_i \mid X_{i-1} = x_{i-1})$$

Let us assume $2^n = l$ (i.e., X_i can take l values). We need to l^2 values to specify the distribution
We can represent this as a matrix $T \in l \times l$ where the entry $T_{a,b}$ of the matrix denotes the probability of transitioning from state a to state b i.e., $P(X_i = b | X_{i-1} = a)$ (T is called transition matrix).
We need to define this transition matrix T_{ab}, i.e.,

$$P(X_i = b | X_{i-1} = a) \forall a, b \quad \forall i$$

The transition probabilities may be different for different time steps, But we will assume that the Markov chain is time homogeneous $T_1 = T_2 = \ldots = T_k = T$
In other words

$$P(X_i = b | X_{i-1} = a) = T_{ab} \quad \forall a, b \quad \forall i$$

Suppose the starting distribution at time step 0 is given by μ_0. Denote $\mu_a^0 = P(X_0 = a)$
Let us consider $P(X_1 = b)$

$$\begin{aligned} P(X_1 = b) &= \sum_a P(X_0 = a, X_1 = b) \\ &= \sum_a P(X_0 = a) P(X_1 = b | X_0 = a) \\ &= \sum_a \mu_a^a T_{ab} \end{aligned}$$

Similarly, $P(X_2 = b) = \sum_a \mu_a^1 T_{ab}$

We can write $P(X_2)$ compactly as $P(X_2) = \mu^1 T = (\mu^0 T) T = \mu^0 T^2$. In general $P(X_k) = \mu^0 T^k$
Note that this is still computationally expensive because it involves a product of $\mu^0(2^n)$ and $T^k(2^n \times 2^n)$
If at a certain time step t, μ^t reaches a distribution π such that $\pi T = \pi$ Then for all subsequent time steps $\mu^j = \pi (j \geq t)$ π is then called the stationary distribution of the Markov chain
Important: If we run a Markov Chain for a large number of time steps then after a point we start getting samples x_t , x_{t+1} , x_{t+2} ,… which are essentially being drawn from the stationary distribution, i.e.we start drawing a sample $X_0 \sim \mu^0$ and then continue drawing samples

$$X_1 \sim \mu^0 T, X_2 \sim \mu^0 T^2, X_3 \sim \mu^0 T^3, \ldots, X_t \sim \pi, X_{t+1} \sim \pi, X_{t+2} \sim \pi \ldots$$

But it is not easy to draw these samples.
Theorem: If $X_0, X_1, \ldots, X_t$ is an irreducible time homogeneous discrete Markov Chain with stationary distribution π, then

$$\frac{1}{t}\sum_{i=1}^{t} f(X_i) \xrightarrow[t \to \infty]{\text{Converges almost surely}} E_\pi[f(X)]$$

where $X \in \mathcal{X}$ and $X \sim \pi$, for any function $f : \mathcal{X} \to \Re$
If, further the Markov Chain is aperiodic then $P(X_t = x_t \mid X_0 = x_0) \to \pi(X)$ as $t \to \infty$ $\forall x, x_0 \in \mathcal{X}$
Part A of the theorem essentially tells us that if we can set up the chain $X_0, X_1, \ldots, X_t$ such that it is tractable then using samples from this chain we can compute $E_\pi[f(X)]$
Similarly Part B of the theorem says that if we can set up the chain $X_0, X_1, \ldots, X_t$ such that it is tractable then we can essentially get samples as if they were drawn from π(X)

We begin by defining our Markov Chain. $X = \{V, H\} \in \{0, 1\}^{n+m}$, so at time step 0 we create a random vector $X \in \{0,1\}^{n+m}$ For transition from a state $X = x \in \{0,1\}^{n+m}$ to $y \in \{0,1\}^{n+m}$
Sample a value $i \in \{1, 2, \ldots, m+n\}$ using a distribution q(i) (say, uniform distribution). Fix the value of all variables except X_i. Sample a new value for X_i (could be V or H) using the conditional distribution $P(X_i = y_i \mid X_{-i} = x_{-i})$ Repeat the above process for many many time steps.
Formally our transition matrix is defined as :

$$p_{xy} = \begin{cases} q(i)P(y_i \mid x_{-i}), & \text{if } \exists i \in X \text{ so that } \forall v \in X \text{ with } v \neq i \;\; x_v = y_v \\ 0, & \text{otherwise} \end{cases}$$

At each step we are changing only one of the n + m random variables using the probability

$$P(X_i = y_i \mid X_{-i} = x_{-i}) = \frac{P(X)}{P(X_{-i})}$$

Consider the case when $i \leq m$ (i.e., we have decided to transition the value of one of the visible variables V_1 to V_m)

$$P(V_i = y_i \mid V_{-i}, H) = P(V_i = y_i \mid H) = \begin{cases} z, & \text{if } y_i = 1 \\ 1 - z, & \text{if } y_i = 0 \end{cases}$$

where $z = \sigma\left(\sum_{j=1}^{m} w_{ij} v_j + c_i\right)$
Once you compute the above probability, with probability 'z' you will set the value of V_i to 1 and with probability '1−z' you will set it to 0.
We have our Markov chain and transition probability, now we need to show that stationary distribution of this chain is P(X)

Detailed Balance Condition

To show that a distribution π is a stationary distribution for a Markov Chain described by the transition probabilities p_{xy}, x, y ∈ Ω, it is sufficient to show that ∀x, y ∈ Ω, the following condition holds:

$$\pi(x)p_{xy} = \pi(y)p_{yx}$$

To prove: $\pi(x)p_{xy} = \pi(y)p_{yx}$
There are 3 cases that we need to consider :
Case 1: x and y differ in the state of more than one random variable

$$\pi(x)p_{xy} = \pi(x) * 0 = 0$$
$$\pi(y)p_{yx} = \pi(y) * 0 = 0$$

Hence the detailed balance condition holds trivially
Case 2: x and y are equal (i.e., they do not differ in the state of any random variable)

$$\pi(x)p_{xy} = \pi(x)p_{xx}$$
$$\pi(y)p_{yx} = \pi(x)p_{xx}$$

Hence the detailed balance condition holds trivially

Case 3: x and y differ in the state of only one random variable

$$\begin{aligned}\pi(x)\,p_{xy} &= \pi(x)\,q(i)\,\pi(y_i|x_{-i}) \\ &= q(i)\,\pi(x_i, x_{-i})\,\frac{\pi(y_i, x_{-i})}{\pi(x_{-i})} \\ &= \pi(y_i, x_{-i})\,q(i)\,\frac{\pi(x_i, x_{-i})}{\pi(x_{-i})} \\ &= \pi(y_i)\,q(i)\,\pi(x_i|x_{-i}) \\ &= \pi(y_i)\,p_{yx}\end{aligned}$$

Hence the stationary distribution of this markov chain is the distribution P(X), ak=lso it can be seen intuitively that the irreducible and aperiodic (irreducible means we can get from any state in to any other in finite number of transaction)

Training RBM with Gibbs Sampling

We were interested in computing the partial derivative of the log likehood w.r.t. one of the parameters (w_{ij})

$$\begin{aligned}\frac{\partial \ln L(\theta)}{\partial w_{ij}} &= E_{p(h|V)}[v_i h_j] - E_{p(V,H)}[v_i h_j] \\ &= \sum_h p(h|v)\,h_i v_j - \sum_{v,h} p(v,h)\,h_i v_j \\ &= \sum_h p(h|v)\,h_i v_j - \sum_v p(v) \sum_h p(h|v)\,h_i v_j\end{aligned}$$

We will first focus on $\sum_h p(h|v)\,h_i v_j$

$$\begin{aligned}\sum_h p(h|v)\,h_i v_j &= \sum_{h_i} \sum_{h_{-i}} p(h_i|v)\,p(h_{-i}|v)\,h_i v_j \\ &= \sum_{h_i} p(h_i|v)\,h_i v_j \sum_{h_{-i}} p(h_{-i}|v) \\ &= p(H_i = 1|v)\,v_j \\ &= \sigma\left(\sum_{j=1}^{m} w_{ij} v_j + c_i\right) v_j\end{aligned}$$

$$\frac{\partial L(\theta)}{\partial w_{ij}} = \sigma\left(\sum_{j=1}^{m} w_{ij} v_j + c_i\right) v_j - \sum_v p(v)\,\sigma\left(\sum_{j=1}^{m} w_{ij} v_j + c_i\right) v_j$$

In vector and matrix form we can write,

$$\begin{aligned}\frac{\partial L(\theta)}{\partial w_{ij}} &= \sigma\left(\sum_{j=1}^{m} w_{ij} v_j + c_i\right) v_j - \sum_v p(v)\,\sigma\left(\sum_{j=1}^{m} w_{ij} v_j + c_i\right) v_j \\ &= \sigma(\boldsymbol{w_i v} + c_i)\,v_j - \sum_v p(\boldsymbol{v})\,\sigma(\boldsymbol{w_i v} + c_i)\,v_j \\ \nabla_W L(\theta) &= \sigma(\boldsymbol{W v} + \boldsymbol{c})\,\boldsymbol{v^T} - E_{\boldsymbol{v}}[\,\sigma(\boldsymbol{W v} + \boldsymbol{c})\,\boldsymbol{v^T}]\end{aligned}$$

Similarly,

$$\frac{\partial L(\theta)}{\partial b_j} = E_{p(H|V)}[v_j] - E_{p(V,H)}[v_j]$$

$$= \sum_h p(h|v) v_j - \sum_{v,h} p(v,h) v_j$$

$$\frac{\partial L(\theta)}{\partial b_j} = \sum_h p(h|v) v_j - \sum_v p(v) \sum_h p(h|v) v_j$$

$$= v_j \sum_h p(h|v) - \sum_v p(v) v_j \sum_h p(h|v)$$

$$= v_j - \sum_v p(v) v_j$$

$$\nabla_b L(\theta) = \boldsymbol{v} - \sum_v p(\boldsymbol{v}) v$$

$$= \boldsymbol{v} - E_v[\boldsymbol{v}]$$

$$\frac{\partial L(\theta)}{\partial c_i} = E_{p(H|V)}[h_i] - E_{p(V,H)}[h_i]$$

$$= \sum_h p(h|v) h_i - \sum_{v,h} p(v,h) h_i$$

$$= \sum_h p(h|v) h_i - \sum_v p(v) \sum_h p(h|v) h_i$$

$$= p(H_i = 1|v) - \sum_v p(v) p(H_i = 1|v)$$

$$= \sigma\left(\sum_{j=1}^{m} w_{ij} v_j + c_i\right) - \sum_v p(v) \sigma\left(\sum_{j=1}^{m} w_{ij} v_j + c_i\right)$$

$$\nabla_c L(\theta) = \sigma(\boldsymbol{W}\boldsymbol{v} + \boldsymbol{c}) - \sum_v p(\boldsymbol{v}) \sigma(\boldsymbol{W}\boldsymbol{v} + \boldsymbol{c})$$

$$= \sigma(\boldsymbol{W}\boldsymbol{v} + \boldsymbol{c}) - E_v[\sigma(\boldsymbol{W}\boldsymbol{v} + \boldsymbol{c})]$$

All 3 gradients expressions have an intractable expectation term, which we estimate by Gibbs Sampling

$$E_v[\sigma(Wv + c) v^T] \approx \frac{1}{k} \sum_{i=1}^{k} \sigma(W v^{(k)} + c) v^{(k)T}$$

$$E_v[v] \approx \frac{1}{k} \sum_{i=1}^{k} v^{(k)}$$

$$E_v[\sigma(Wv + c)] \approx \frac{1}{k} \sum_{i=1}^{k} \sigma(W v^{(k)} + c)$$

In practice, Gibbs Sampling can be very inefficient because for every step of stochastic gradient descent we need to run the Markov chain for many many steps and then compute the expectation using the samples drawn from this chain.

We have a more efficient algorithm called k-contrastive divergence which is used in practice for training RBMs.

Contrastive divergence uses the following idea :

Instead of starting the Markov Chain at a random point ($V = v^0$), start from $v^{(t)}$ where $v^{(t)}$ is the current training instance. Run Gibbs Sampling for k steps and denote the sample at the k^{th} step by $\tilde{v}$.

Replace the expectation by a point estimate

$$E_{p(V,H)}[v_j h_i] = \sum_v p(\boldsymbol{v}) \sigma(\boldsymbol{w_i}\boldsymbol{v} + c_i) v_j \approx \sigma(\boldsymbol{w_i}\tilde{\boldsymbol{v}} + c_i) \tilde{v}_j$$

Over time as our model becomes better and better $\tilde{v}$ should start looking more and more like our training (empirical) samples.

$$\frac{\partial L(\theta)}{\partial w_{ij}} = \sigma(\boldsymbol{w_i v} + c_i) v_j - \sum_{v} p(\boldsymbol{v}) \sigma(\boldsymbol{w_i v} + c_i) v_j$$

We have two summations here:
The first term can be thought of as summation over a single point 'v' from training example. The second term, the summation over $\tilde{v}$ is being replaced by a point estimate computed from the model sample.
As training progresses and $\tilde{v}$ starts looking more and more like our training (empirical) samples, the difference between the two terms will be small and the parameters of the model will stabilize.

Algorithm 0 : RBM training with Block Gibbs Sampling

Input : RBM ($V_1, \dots, V_m, H_1, \dots, H_n$) , training batch D
Output : Learned parameters **W**,**b**,**c**
init **W**,**b**,**c**
for all $v_d \in D$, do
    Randomly Initialize $v^{(0)}$
    for t=0, … , k , k+1 , … ,k+r ;do
        for i = 1, … ,n ; do
            sample $h_i^{(t)} \sim p(h_i | v^{(t)})$
        end
        for j = 1 , … , m ; do
            sample $v_j^{(t+1)} \sim p(v_j | h^{(t)})$
        end
    end

$$W \leftarrow W + \eta \nabla_W L(\theta) \left[\sigma(W v_d + c) v_d^T - \frac{1}{r} \sum_{t=k+1}^{t=k+r} \sigma(W v^{(t)} + c) v^{(t)T} \right]$$

$$b \leftarrow b + \eta \nabla_b L(\theta) \left[v_d - \frac{1}{r} \sum_{t=k+1}^{t=k+r} v^{(t)} \right]$$

$$c \leftarrow c + \eta \nabla_c L(\theta) \left[\sigma(W v_d + c) - \frac{1}{r} \sum_{t=k+1}^{t=k+r} \sigma(W v^{(t)} + c) \right]$$

end

Algorithm 1 : k-step Constrastive Divergence

Input : RBM ($V_1, \dots, V_m, H_1, \dots, H_n$) , training batch D
Output : Learned parameters **W**,**b**,**c**
init **W**,**b**,**c**
forall $v_d \in D$, do
    Randomly Initialize $v^{(0)} \leftarrow v$
    for t=0, … , k ;do
        for i = 1, … ,n ; do
            sample $h_i^{(t)} \sim p(h_i | v^{(t)})$
        end
        for j = 1 , … , m ; do
            sample $v_j^{(t+1)} \sim p(v_j | h^{(t)})$
        end

end

$W \leftarrow W + \eta \nabla_W L(\theta)\left[\sigma(W v_d + c) v_d^T - \sigma(W \tilde{v} + c)\tilde{v}^T\right]$

$b \leftarrow b + \eta \nabla_b L(\theta)\left[v_d - \tilde{v}\right]$

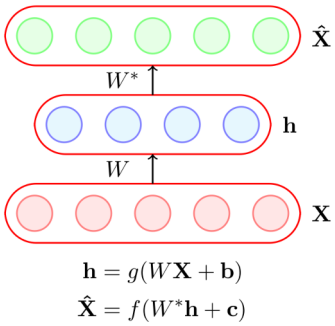
$c \leftarrow c + \eta \nabla_c L(\theta)\left[\sigma(W v_d + c) - \sigma(W \tilde{v} + c)\right]$

end

Variational Autoencoders

With autoencoders, we take an input and simply reconstructing it and we represent it by a good abstraction of the input. But we want to do something more besides abstraction (able to do generation)

Can we do generation with autoencoders ?



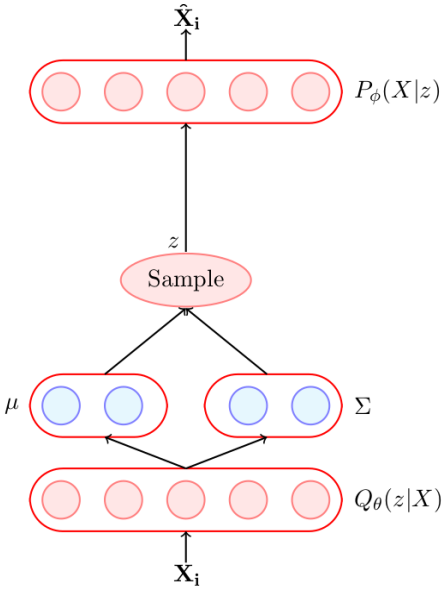
In other words, once the autoencoder is trained can I remove the encoder, feed a hidden representation 'h' to the decoder and decode a $\hat{X}$ from it ?

In principle, yes! But in practice there is a problem with this approach. 'h' is a very high dimensional vector and only a few vectors in this space would actually correspond to meaningful latent representations of our input. So, we are interested in sampling from P (h|X) so that we pick only those 'h' which have a high probability

But autoencoders learn a hidden representation h but not a distribution P(h|X). Similarly the decoder is also deterministic and does not learn a distribution over X, P(X|h)

Variational Autoencoders : The Neural Network Perspective

Let $\{X = x_i\}_{i=1}^N$ be the training data.We are interested in learning an abstraction and generation. In probabilistic terms we are interested in P(z|X) and P(X|z). VAEs use a neural network based encoder to learn a distribution over the latent variables (Q(z|X)) and a neural network based decoder to learn a distribution over the visible variables (P (X|z))



Encoder : We want to learn the parameters of the distribution. We assume that the latent variables come from a standard normal distribution $\mathcal{N}(0, I)$ and the job of the encoder is to then predict the parameters of this distribution

Decoder : The job of the decoder is to predict a probability distribution over X : P (X|z). We could assume that P (X|z) is a Gaussian distribution with unit variance. The job of the decoder f would then be to predict the mean of this distribution as $f_\phi(z)$

For any given training sample x_i, it should maximize $P(x_i)$ given by

$$P(x_i) = \int P(z) P(x_i|z)\, dz$$
$$= E_{z \sim Q_\theta(z|x_i)}[\log P_\phi(x_i|z)]$$

(We take log for numerical stability)

This is the loss function for one data point $\{l_i(\theta) = -E_{z \sim Q_\theta(z|x_i)}[\log P_\phi(x_i|z)]\}$ and we will just sum over all the data points to get the total loss L (θ)

$$L(\theta) = \sum_{i=1}^{m} l_i(\theta)$$

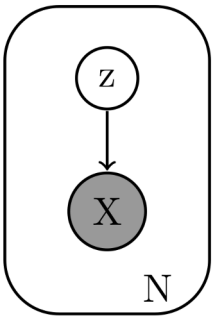
In addition, we also want a constraint on the distribution over the latent variables. Specifically, we had assumed P (z) to be $\mathcal{N}(0, I)$ and we want Q(z|X) to be as close to P (z) as possible. Thus, we will modify the loss function such that

$$l_i(\theta, \phi) = -E_{z \sim Q_\theta(z|x_i)}[\log P_\phi(x_i|z)] + KL(Q_\theta(z|x_i) \| P(z))$$

The second term in the loss function can actually be thought of as a regularizer. It ensures that the encoder does not cheat by mapping each x_i to a different point (a normal distribution with very low variance) in the Euclidean space. In other words, in the absence of the regularizer the encoder can learn a unique mapping for each x_i and the decoder can then decode from this unique mapping.
Even with high variance in samples from the distribution, we want the decoder to be able to reconstruct the original data very well (motivation similar to the adding noise). To summarize, for each data point we predict a distribution such that, with high probability a sample from this distribution should be able to reconstruct the original data point.

Variational Autoencoders: The graphical model perspective

Here we can think of z and X as random variables. We are then interested in the joint probability distribution P (X, z) which factorizes as P(X,z) = P(z) P(X|z). This factorization is natural because we can imagine that the latent variables are fixed first and then the visible variables are drawn based on the latent variables.



Now at inference time, we are given an X (observed variable) and we are interested in finding the most likely assignments of latent variables z which would have resulted in this observation
Mathematically, we want to find

$$P(z|X) = \frac{P(X|z)P(z)}{P(X)}$$

This is hard to compute because the LHS contains P(X) which is intractable

$$\begin{aligned} P(X) &= \int P(X|z)P(z)\,dz \\ &= \int \int \dots \int P(X|z_1, z_2, \dots, z_n)P(z_1, z_2, \dots, z_n)\,dz_1, \dots, dz_n \end{aligned}$$

In RBMs, we had a similar integral which we approximated using Gibbs Sampling. VAEs, on the other hand, cast this into an optimization problem and learn the parameters of the optimization problem. Specifically, in VAEs, we assume that instead of P (z|X) which is intractable, the posterior distribution is given by $Q_\theta(z|X)$. and $Q_\theta(z|X)$ is a Gaussian whose parameters are determined by a neural network.
We want the proposed distribution $Q_\theta(z|X)$ to be as close to the true distribution. We can capture this using the following objective function

$$minimize \quad KL(Q_\theta(z|X) \| P(z|X))$$

Expanding KL divergence term

$$\begin{aligned} D[Q_\theta(z|X) \| P(z|X)] &= \int Q_\theta(z|X) \log Q_\theta(z|X)\,dz - \int Q_\theta(z|X) \log P(z|X)\,dz \\ &= E_{z \sim Q_\theta(z|X)}[\log Q_\theta(z|X) - \log P(z|X)] \end{aligned}$$

Denote $E_Q = E_{z \sim Q_\theta(z|X)}$ and substitute $P(z|X) = \frac{P(X|z)P(z)}{P(X)}$

$$\begin{aligned} D[Q_\theta(z|X) \| P(z|X)] &= E_Q[\log Q_\theta(z|X) - \log P(X|z) - \log P(z) + \log P(X)] \\ &= E_Q[\log Q_\theta(z|X) - \log P(z)] - E_Q[\log P(X|z)] + \log P(X) \\ &= D[Q_\theta(z|X) \| p(z)] - E_Q[\log P(X|z)] + \log P(X) \end{aligned}$$

$$\log P(X) = E_Q[\log P(X|z)] - D[Q_\theta(z|X) \| P(z|X)] + D[Q_\theta(z|X) \| p(z)]$$

Since KL divergence (the red term) is always ≥ 0 we can say that

$$E_Q[\log P(X|z)] - D[Q_\theta(z|X) \| P(z|X)] \leq \log P(X)$$

The quantity on the LHS is thus a lower bound for the quantity that we want to maximize and is knows as the Evidence lower bound (ELBO). Maximizing this lower bound is the same as maximizing log P(X) and hence our equivalent objective now becomes

$$maximize \;\; E_Q[\log P(X|z)] - D[Q_\theta(z|X) \| P(z)]$$

First we will do a forward prop through the encoder using X_i and compute μ(X) and Σ(X). The second term in the above objective function is the difference between two normal distribution $\mathcal{N}$(μ(X), Σ(X)) and $\mathcal{N}$(0, I).

$$D[\mathcal{N}(\mu(X), \Sigma(X)) \| \mathcal{N}(0, I)] = \frac{1}{2}\left(tr(\Sigma(X)) + (\mu(X))^T(\mu(X)) - k - \log det(\Sigma(X))\right)$$

where k is the dimensionality of the latent variables.

This term can be computed easily because we have already computed μ(X) and Σ(X) in the forward pass.

Now the other term in the objective function

$$\sum_{i=1}^{n} E_Q[\log P_\phi(X|z)]$$

This is again an expectation and hence intractable (integral over z). In VAEs, we approximate this with a single 'z' sampled from $\mathcal{N}$(μ(X), Σ(X)). Hence this term is also easy to compute.

We assume that P (X|z) is a Gaussian with mean μ(z) and variance I then

$$\log P(X = X_i|z) = C - \frac{1}{2}\|X_i - \mu(z)\|^2$$

μ(z) in turn is a function of the parameters of the decoder and can be written as $f_\varphi(z)$

$$\log P(X = X_i|z) = C - \frac{1}{2}\|X_i - f_\phi(z)\|^2$$

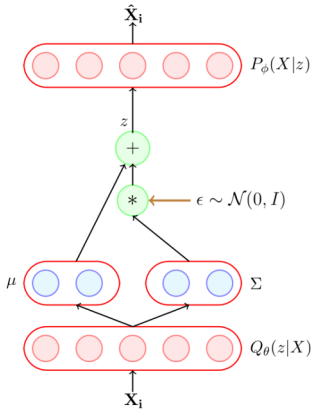
Our effective objective function thus becomes

$$minimize_{\theta,\phi} \;\; \sum_{i=1}^{n}\left[\frac{1}{2}\left(tr(\Sigma(X_i)) + (\mu(X_i))^T(\mu(X_i)) - k - \log det(\Sigma(X_i))\right) + \|X_i - f_\phi(z)\|^2\right]$$

The above loss can be easily computed and we can update the parameters θ of the encoder and ϕ of decoder using backpropagation. However, the network is not end to end differentiable because the output $f_\phi(z)$ is not an end to end differentiable function of the input X. Due to the sampling step, entire process non-deterministic.

To get around this problem, VAE use reparameterization trick. For 1 dimensional case, given μ and σ we can sample from N (μ, σ) by first sampling $\epsilon \sim N(0,1)$, and then computing $z = \mu + \sigma * \epsilon$

The adjacent figure shows the difference between the original network and the reparamterized network. The randomness in $f_\phi(z)$ is now associated with ϵ and not X or the parameters of the model.



Abstraction : After the model parameters are learned we feed a X to the encoder. By doing a forward pass using the learned parameters of the model we compute μ(X) and Σ(X). We then sample a z from the distribution μ(X) and Σ(X) or using the same reparameterization trick.

Generation : After the model parameters are learned we remove the encoder and feed a $z \sim \mathcal{N}(0, I)$ to the decoder. The decoder will then predict $f_\phi(z)$ and we can draw an $X \sim \mathcal{N}(f_\phi(z), I)$

Neural Autoregressive Density Estimator(NADE)

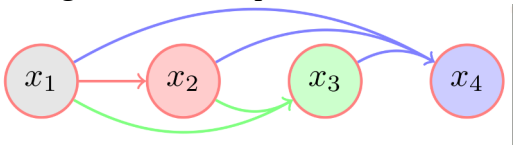
Autoregressive (AR) Models do not contain any latent variables.AR models do not make any independence assumption but use the default factorization of p(x) given by the chain rule

$$p(x)=\prod_{i=1}^{n} p(x_i|x_{<k})$$

The above factorization contains n factors and some of these factors contain many parameters ($O(2^n)$ in total). It is infeasible to learn such an exponential number of parameters. AR models work around this by using a neural network to parameterize these factors and then learn the parameters of this neural network.

At the output layer we want to predict 'n' conditional probability distributions (each corresponding to one of the factors in our joint distribution). At the input layer we are given the 'n' input variables

The n^{th} output should only be connected to the previous n-1 inputs.



The Neural Autoregressive Density Estimator (NADE) proposes a simple solution for this First, for every output unit, we compute a hidden representation using only the relevant input units. For example, for the k^{th} output unit, the hidden representation will be computed using:

$$h_k=\sigma(W_{.,<k}\,x_{<k}+b)$$

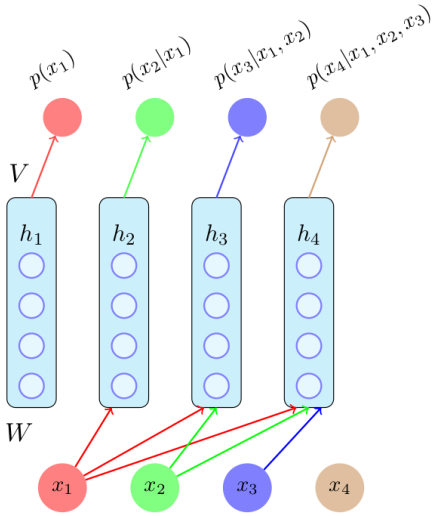
where $h_k\in\Re^d, W\in\Re^{d\times n}, W_{.,<k}$ are the first k columns of W

We now compute the output $p(x_k|\boldsymbol{x}_1^{k-1})$ as:

$$y_k=p(x_k|\boldsymbol{x}_1^{k-1})=\sigma(V_k h_k+c_k)$$

Notation : $\boldsymbol{x}_1^{k-1}$ denote ($x_1, x_2, \ldots, x_{k-1}$)

$W\in\Re^{d\times n}$ and $b\in\Re^d$ are shared parameters and are used for computing h_k, and $V_k\in\Re^d$ and $c_k\in\Re$ for each of the n factors. There is also an additional parameter $h_1\in\Re^d$ (initial state). The total number of parameters in the model is $2nd+n+2d$ which is linear in n (the model does not have an exponential number of parameters)



For every output node we know the true probability distribution For example, for a given training instance, if $X_3 = 1$ then the true probability distribution is given by $p(x_3=1|x_2,x_1)=1, p(x_3=0|x_2,x_1)=0$ or $p=[0,1]$. If the predicted distribution is q = [0.7, 0.3], then we can just take the cross entropy between p and q as the loss function. The total loss will be the sum of this cross entropy loss for all the n output nodes.

By design NADE are not made for abstraction. For generation, we first compute $p(x_1=1)$ as $y_1=\sigma(V_1 h_1+c_1)$ We will then sample a value for x_1 from the distribution Bernoulli(y_1). We will now use the sampled value of x_1 and compute h_2 as $h_2=\sigma(W_{.,<2}x_{<2}+b)$. Using h_2 we will compute $P(x_2=1|x_1=x_1)$ as $y_2=\sigma(V_2 h_2+c_2)$. We will then sample a value for x_2 from the distribution Bernoulli(y_2). We will then continue this process till x_n generating the value of one random variable at a time. If x is an image then this is equivalent to generating the image one pixel at a time, the model requires a lot of computations because for generating each pixel we need to compute

$$h_k=\sigma(W_{.,<k}x_{<k}+b)$$
$$y_k=p(x_k|x_1^{k-1})=\sigma(V_k h_k+c_k)$$

However notice that

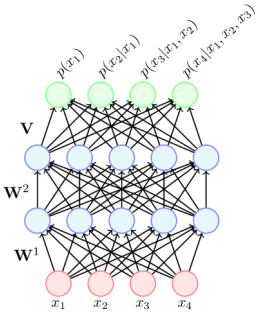
$$W_{.,<k+1}\,x_{<k+1} + b = W_{.,<k}\,x_{<k} + b + W_{.,k}\,x_k$$

Thus we can reuse some of the computations done for pixel k while predicting the pixel k+1 (this can be done even at training time)

Masked Autoencoder Density Estimator (MADE)

Motivation : Tweaking an autoencoder so that its output units predict the n conditional distributions instead of reconstructing the inputs.

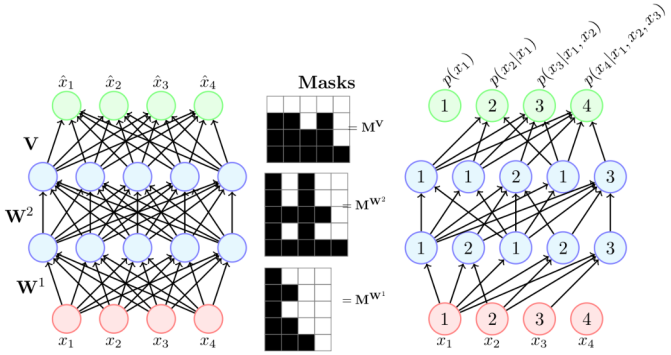
But this is not straightforward because we need to make sure that the k^{th} output unit only depends on the previous k-1 inputs In a standard autoencoder with fully connected layers the k-th unit obviously depends on all the input units. We cannot allow this if we want to predict the conditional distributions $p(x_k|x_{<k})$



We could ensure this by masking some of the connections in the network to ensure that y_k only depends on $x_{<k}$. We will start by assuming some ordering on the inputs and just number them from 1 to n.

Now we will randomly assign each hidden unit a number between 1 to n-1 which indicates the number of inputs it will be connected to. We will do a similar assignment for all the hidden layers. We can implement this by taking the weight matrices W_1,W_2 and V and applying an appropriate mask to them so that the disallowed connections are dropped.

For Example, we can apply the mask at layer 2:



$$\begin{bmatrix} W^2_{11} & W^2_{12} & W^2_{13} & W^2_{14} & W^2_{15} \\ W^2_{21} & W^2_{22} & W^2_{23} & W^2_{24} & W^3_{25} \\ W^2_{31} & W^2_{32} & W^2_{33} & W^2_{34} & W^2_{35} \\ W^2_{41} & W^2_{42} & W^2_{43} & W^2_{44} & W^2_{45} \\ W^2_{51} & W^2_{52} & W^2_{53} & W^2_{54} & W^2_{55} \end{bmatrix} \odot \begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

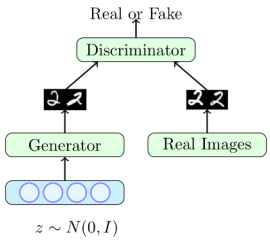
The objective function for this network would be a sum of cross entropies. The network can be trained using backpropagation such that the errors will only be propagated along the active (unmasked) connections (similar to what happens in dropout)

Similar to NADE, this model is not designed for abstraction but for generation and using the same iterative process that we used with NADE, we will do genration.

First sample a value of x_1. Now feed this value of x_1 to the network and compute y_2. Now sample x_2 from Bernoulli (y_2) and repeat the process till you generate all variables upto x_n

Generative Adversarial Networks

The idea of GANs is to sample from a simple tractable distribution (say, $z \sim N(0, I)$) and then learn a complex transformation from this to the training distribution.



We use Neural Network for such transformation and we train it using a two player game, where are two players are : a generator and a discriminator. The job of the generator is to produce images which look so natural that the discriminator thinks that the images came from the real data distribution. The job of the discriminator is to get better and better at distinguishing between true images and generated (fake) images.

Given an image generated by the generator as $G_\phi(z)$ the discriminator assigns a score $D_\theta(G_\phi(z))$ to it. This score will be between 0 and 1 and will tell us the probability of the image being real or fake.

For a given z, the generator would want to maximize $\log D_\theta(G_\phi(z))$ (log likelihood) or minimize $\log\left(1-D_\theta(G_\phi(z))\right)$

This is just for a single z and the generator would like to do this for all possible values of z. For example, if z was discrete and drawn from a uniform distribution (i.e., p(z) = 1/N ∀z) then the generator's objective function would be

$$min_\phi \sum_{i=1}^{N} \frac{1}{N} \log\left(1-D_\theta(G_\phi(z))\right)$$

However, in our case, z is continuous and not uniform $(z \sim \mathcal{N}(0, I))$ so the equivalent objective function would be

$$min_\phi \int p(z) \log(1 - D_\theta(G_\phi(z)))$$
$$min_\phi E_{z\sim p(z)}[\log(1 - D_\theta(G_\phi(z)))]$$

Also the task of the discriminator is to assign a high score to real images and a low score to fake images and it should do this for all possible real images and all possible fake images. In other words, it should try to maximize the following objective function

$$max_\theta \; E_{x\sim p_{data}}[\log D_\theta(x)] + E_{z\sim p(z)}[\log(1 - D_\theta(G_\phi(z)))]$$

If we put the objectives of the generator and discriminator together we get a minimax game

$$min_\phi \; max_\theta \left[E_{x\sim p_{data}} \log D_\theta(x) + E_{z\sim p(z)} \log(1 - D_\theta(G_\phi(z)))\right]$$

The first term in the objective is only w.r.t. the parameters of the discriminator(θ). The second term in the objective is w.r.t. the parameters of the generator(ϕ) as well as the discriminator(θ). The discriminator wants to maximize the second term whereas the generator wants to minimize it (hence it is a two-player game). So the overall training proceeds by alternating between these two step

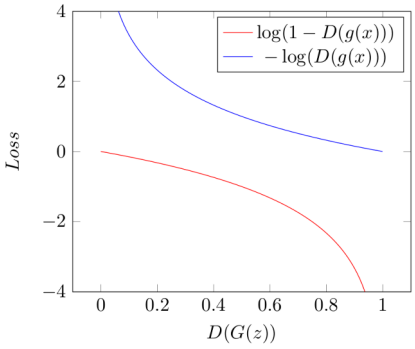
Step 1 : Gradient Ascent on Discriminator

$$max_\theta \left[E_{x\sim p_{data}} \log D_\theta(x) + E_{z\sim p(z)} \log(1 - D_\theta(G_\phi(z)))\right]$$

Step 2 : Gradient Descent on Generator

$$min_\phi \; E_{z\sim p(z)}\left[\log(1 - D_\theta(G_\phi(z)))\right]$$

In practice, the above generator objective does not work well and we use a slightly modified objective



When the sample is likely fake, we want to give a feedback to the generator (using gradients). However, in this region where D(G(z)) is close to 0, the curve of the loss function is very flat and the gradient would be close to 0.

Trick: Instead of minimizing the likelihood of the discriminator being correct, maximize the likelihood of the discriminator being wrong. In effect, the objective remains the same but the gradient signal becomes better

Algorithm 2 : GANs Training

for number of training iterations do

for k steps do

Sample minibatch of m noise samples $\{z^{(1)}, \ldots, z^{(m)}\}$ from noise prior $p_g(z)$

Sample minibatch of m examples $\{x^{(1)},\dots,x^{(m)}\}$ from data generating distribution $p_{data}(x)$

Update the discriminator by ascending its stochastic gradient:

$$\nabla_\theta \frac{1}{m}\sum_{i=1}^{m}\left[\log D_\theta(x^{(i)})+\log(1-D_\theta(G_\phi(Z^{(i)})))\right]$$

end for

Sample minibatch of m noise samples $\{z^{(1)},\dots,z^{(m)}\}$ from noise prior $p_g(z)$

Update the generator by ascending its stochastic gradient

$$\nabla_\phi \frac{1}{m}\sum_{i=1}^{m}\left[\log(D_\theta(G_\phi(z^{(i)})))\right]$$

end for

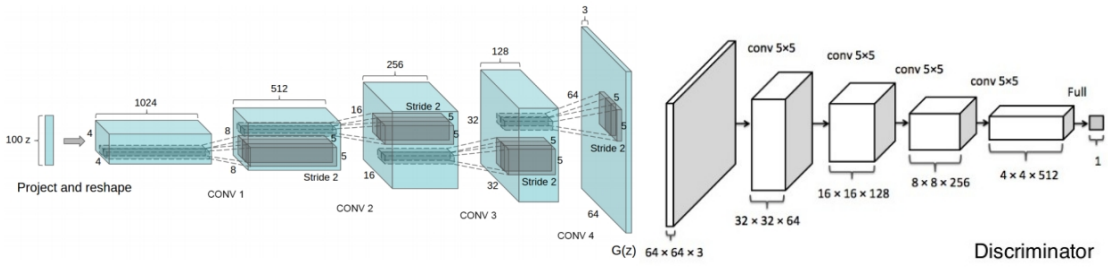
Deep Convolutional GANs

For discriminator, any CNN based classifier with 1 class (real) at the output can be used (e.g. VGG, ResNet, etc.)

Architecture guidelines for stable Deep Convolutional GANs :

- Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).

- Use batchnorm in both the generator and the discriminator.
- Remove fully connected hidden layers for deeper architectures.
- Use ReLU activation in generator for all layers except for the output, which uses tanh.
- Use LeakyReLU activation in the discriminator for all layers



Suppose we denote the true data distribution by $p_{data}(x)$ and the distribution of the data generated by the model as $p_G(x)$. We want $p_G(x) = p_{data}(x)$ at the end of training.

Theorem

The global minimum of the virtual training criterion $C(G)=max_D V(G,D)$ achieved if and only if $p_G = p_{data}$, is equivalent to :

- If $p_G = p_{data}$ then the global minimum of the virtual training criterion $C(G)=max_D V(G,D)$ is achieved and
- The global minimum of the virtual training criterion $C(G)=max_D V(G,D)$ is achieved only if $p_G = p_{data}$