Boltzmann Machines (BM)

Introduction

Boltzmann Machines are **stochastic neural networks** that learn to represent and reproduce complex data distributions.
They consist of **visible** and **hidden** nodes connected in an **undirected, symmetric** fashion.
The model is based on the **principle of energy minimization** — lower energy states correspond to more probable configurations.
This mechanism enables BMs to discover underlying patterns and dependencies in data without explicit supervision.

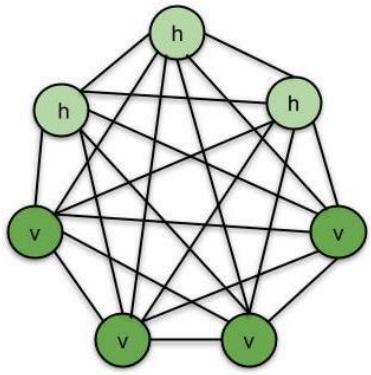
Architecture of Boltzmann Machine

A Boltzmann Machine is composed of:

- **Visible units (v):** Represent observed data (inputs).
- **Hidden units (h):** Capture latent or abstract features not directly seen in the input.

All visible and hidden units are interconnected. Each connection is associated with a **weight (w_{ij})** indicating the strength of interaction.
There are no self-connections (a node cannot connect to itself).

The goal of training is to adjust weights so that the network assigns **high probability to observed data** and **low probability to random or noisy data**.



v - visible nodes, h - hidden nodes

Restricted Boltzmann Machine (RBM)

Training a full BM is computationally heavy, so a simplified version called **Restricted Boltzmann Machine (RBM)** is used.

Restrictions in RBM:

- No visible-to-visible connections.
- No hidden-to-hidden connections.
  Only **visible–hidden connections** exist, making learning easier and faster

Training RBMs (Contrastive Divergence)

RBMs are trained using **Contrastive Divergence (CD)**, an efficient approximation of maximum likelihood estimation.

Steps:

1. **Initialize** weights randomly.
2. **Forward pass:** Compute hidden activations from visible data.
3. **Reconstruction:** Recreate visible data from hidden activations.
4. **Compare:** Measure difference between original and reconstructed data.
5. **Update:** Adjust weights to reduce reconstruction error.
6. **Repeat** until convergence.

This iterative process involves **Gibbs Sampling**, where the model samples repeatedly between visible and hidden layers to refine its understanding of data distribution.

Working of RBM - Illustrative Example -

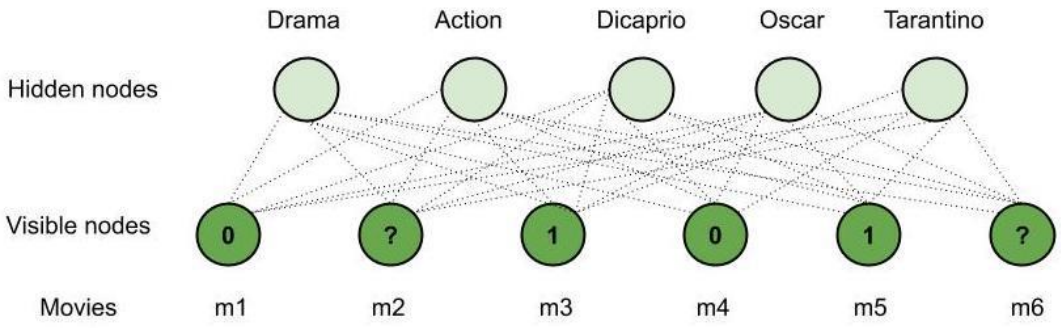
Consider - Mary watches four movies out of the six available movies and rates four of them. Say, she watched m_1, m_3, m_4 and m_5 and likes m_3, m_5 (rated 1) and dislikes the other two, that is m_1, m_4 (rated 0) whereas the other two movies - m2, m6 are unrated. Now, using our RBM, we will recommend one of these movies for her to watch next. Say -

- m_3, m_5 are of 'Drama' genre.
- m_1, m_4 are of 'Action' genre.
- 'Dicaprio' played a role in m_5.
- m_3, m_5 have won 'Oscar.'
- 'Tarantino' directed m_4.
- m_2 is of the 'Action' genre.
- m_6 is of both the genres 'Action' and 'Drama', 'Dicaprio' acted in it and it has won an 'Oscar'.

We have the following observations -

- Mary likes m_3, m_5 and they are of genre 'Drama,' she probably **likes 'Drama'** movies.
- Mary dislikes m_1, m_4 and they are of action genre, she probably **dislikes 'Action'** movies.
- Mary likes m_3, m_5 and they have won an 'Oscar', she probably **likes** an **'Oscar'** movie.
- Since 'Dicaprio' acted in m_5 and Mary likes it, she will probably **like** a movie in which **'Dicaprio'** acted.
- Mary does not like m_4 which is directed by Tarantino, she probably **dislikes** any movie directed by **'Tarantino'**.

Therefore, based on the observations and the details of m_2, m_6; our RBM **recommends m6** to Mary ('Drama', 'Dicaprio' and 'Oscar' matches both Mary's interests and m_6). This is how an RBM works and hence is used in recommender systems.



Deep Belief Networks (DBN)

A **Deep Belief Network** is a **stack of multiple RBMs** trained layer by layer.
Each RBM's hidden layer serves as the visible layer for the next one.
This hierarchical stacking enables the network to learn **progressively abstract representations** of data.

Key Characteristics:

- **Lower layers:** Capture simple patterns (edges, textures, words).
- **Higher layers:** Capture abstract features (shapes, genres, object types).
- **Connections:**
    - Between layers → Directed (top-down during generation).
    - Within layers → Undirected.

Training Process:

1. Train the first RBM on raw input.
2. Use its hidden layer activations as input to train the next RBM.
3. Continue stacking until the desired depth is reached.
4. Optionally fine-tune the entire network using backpropagation.

Advantages:

- Efficient unsupervised pre-training.
- Reduces vanishing-gradient issues.
- Builds a rich, hierarchical feature space.

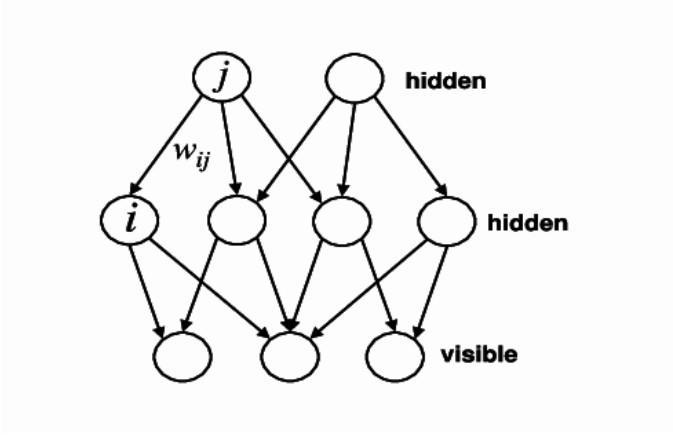
Deep Boltzmann Machines (DBMs)

A **Deep Boltzmann Machine (DBM)** extends RBMs into multiple layers of hidden units.
Each layer captures increasingly **abstract representations** of data.
Connections are **undirected** across all layers, allowing for deep, bidirectional information flow.

Structure:

- **Layer 1 (Lower Layer):** Captures simple features (e.g., edges, colors, words).
- **Layer 2 (Middle Layer):** Represents combinations of features or concepts.
- **Layer 3 (Top Layer):** Captures high-level abstract categories (e.g., genres, emotions, object classes).

The **hierarchical representation** helps the DBM model complex dependencies and perform unsupervised feature learning at multiple levels of abstraction.



Comparison: DBN vs DBM

Feature	Deep Belief Network (DBN)	Deep Boltzmann Machine (DBM)
Connection Type	Directed between layers	Undirected across all layers
Training	Layer-wise pretraining, faster	Joint training, slower
Representation Power	Moderate	Higher (more expressive)
Inference	Easier	Computationally expensive

Applications

- **Recommender Systems** – Learning user preferences for movies, music, etc.
- **Feature Learning and Extraction** – Capturing complex hidden structures.

- **Image and Speech Recognition**
- **Anomaly Detection**
- **Dimensionality Reduction**

Sigmoid Belief Network (SBN)

A Sigmoid Belief Network (SBN) is a type of directed graphical model (also called a Bayesian network) where each node represents a stochastic binary variable (0 or 1) whose probability is determined by a sigmoid (logistic) function of its parent nodes.

It is a directed generative model — meaning it models how data could be generated probabilistically.

Structure:

- The network has visible units (input layer) and hidden units (latent layer).
- Connections are directed (top-down) — from hidden to visible units.
- Each node is binary (0 or 1).
- Each node's state is determined probabilistically using a sigmoid activation of weighted inputs.

Example (Real-Life Scenario)

Let's model the behavior of buying an ice cream 🍦:

- Hidden variable (h): Weather (1 = hot, 0 = cold)
- Visible variable (v): Person buys ice cream (1 = yes, 0 = no)

$$P(v = 1 \mid h) = \sigma(w \times h + b)$$

If weather = hot (h=1), sigmoid output gives a **high probability** of buying ice cream.
If weather = cold (h=0), probability decreases.

This shows how **SBN captures cause-effect** with uncertainty.

Directed Generative Network (DGN)

A Directed Generative Network (DGN) is a probabilistic model that represents how observed data (x) are generated from latent (hidden) variables (z) through a top-down directed structure — like a Bayesian network but implemented using neural networks.

It defines a joint probability distribution over all variables using directed edges, meaning each variable depends only on its parent nodes.

In simple words:
It is a neural network that learns to generate data samples (like images, sounds, or text) by modeling the cause-and-effect relationship from latent variables to visible data.

Basic Idea

A DGN tries to model the true data distribution $P_{data}(x)$.
It assumes data is generated through a process involving some latent causes (z):

$$z \sim P(z) \text{(prior distribution, e.g., Gaussian)}$$
$$x \sim P_\theta(x \mid z)$$

Here:

- z: latent (hidden) variable, usually low-dimensional
- x: observed variable (e.g., image, sound, text)
- $P_\theta(x \mid z)$: a conditional probability distribution modeled by a neural network (parameterized by θ)

Thus, the model defines the joint probability:

$$P_\theta(x, z) = P(z)P_\theta(x \mid z)$$

Structure

A DGN is a directed acyclic graph (DAG) with:

- Latent variables at the top
- Observed variables at the bottom
- Directed edges pointing from causes → effects

Working / Generative Process

The generative process is defined in two steps:

Step 1: Sample from the latent prior

$$z \sim P(z)$$

Usually a simple Gaussian: $z \sim \mathcal{N}(0, I)$

Step 2: Generate data from latent code

$$x = f_\theta(z) + \epsilon$$

where

- f_θ: neural network decoder (maps latent to data)
- ϵ: small random noise

After training, by sampling random z, the network can generate new synthetic samples $x$similar to the training data.

Training Techniques

1. **Maximum Likelihood Estimation (MLE):**
    - Optimize parameters θ to maximize $\log\ P_\theta(x)$.
    - Needs approximation for latent variables.
2. **Variational Inference:**
    - Introduces an inference network (encoder) $q_\phi(z \mid x)$to approximate the posterior $P_\theta(z \mid x)$.
    - Used in **Variational Autoencoders (VAEs)**.
3. **Wake-Sleep Algorithm:**
    - Used in early directed networks like Deep Belief Networks (DBNs).
4. **Reparameterization Trick (in VAEs):**
    - Allows differentiable sampling by expressing $z = \mu + \sigma \odot \epsilon$, with $\epsilon \sim \mathcal{N}(0, I)$.

Real-Life Example

Example 1: Generating Human Faces

- **Latent Variables (z):**
  age, gender, smile intensity, hairstyle, lighting, etc.
- **Decoder (Neural Network):**
  learns the mapping $f_\theta(z)$that turns these attributes into a realistic human face.

After training, you can:

- Input random latent vectors z→ generate new, unique faces.
- Modify $z$slightly → get the same person smiling, older, etc.

Drawing Samples from Autoencoders

What is an Autoencoder?

An Autoencoder is a type of *neural network* used to learn a compressed (latent) representation of data.

It has two main parts:

1. **Encoder** → compresses the input data into a smaller code (latent vector).
2. **Decoder** → reconstructs the original input from that code.

$$x \xrightarrow{\text{Encoder}} z \xrightarrow{\text{Decoder}} \hat{x}$$

Here,

- x= input data (e.g., image, text, sound)
- z= latent representation (encoded features)
- $\hat{x}$= reconstructed data

Can Autoencoders Generate New Data?

Normally, a *basic autoencoder* is not generative — it just learns to reconstruct input data, not create new examples.

But we can draw samples (generate new data) if we understand the distribution of the latent space (the space of $z$values).

Drawing Samples

To **draw samples**, we:

1. Learn the **latent space distribution** (what encoded data looks like).
2. **Sample random points** from this distribution (new $z$values).
3. Pass them through the **decoder** to generate new outputs.

$$z \sim p(z) \text{then} \hat{x} = f_{decoder}(z)$$

Example:

Imagine an autoencoder trained on handwritten digits (like MNIST dataset).

- The encoder learns compressed codes $z$that represent each digit (0–9) in a continuous space.
- The decoder knows how to turn $z$back into images.

To draw a new digit, we:

- Pick a random point z(similar to codes for “5”s).
- Feed it to the decoder → it generates a *new handwritten 5* that didn’t exist before!

Challenge in Basic Autoencoders

In basic (deterministic) autoencoders:

- The latent space is not continuous or smooth.
- Random $z$values might not map to valid data points → outputs look like *noise*.

Hence, we cannot directly draw realistic samples from a plain autoencoder.

Solution: Variational Autoencoder (VAE)

To fix this, we use Variational Autoencoders (VAEs) — a special kind of autoencoder designed for sampling and generation.

VAEs assume:

- The latent space follows a known probability distribution (like Gaussian $\mathcal{N}(0,1)$).
- During training, it forces the encoded features $z$to follow this distribution.

Then, we can sample $z$directly from $\mathcal{N}(0,1)$and decode it to generate realistic new data.

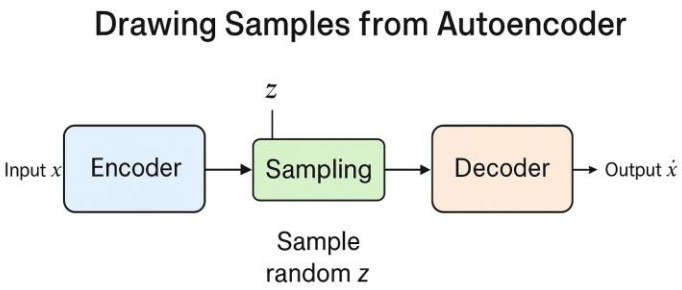
Process:

$$z \sim \mathcal{N}(0,1) \Rightarrow \hat{x} = \text{Decoder}(z)$$

Steps to Draw Samples from Autoencoders (Generalized)

Step	Description
1	Train the Autoencoder on real data (images, sounds, etc.)
2	Encode all training data to get latent vectors z.
3	Model the latent space distribution (often Gaussian).
4	Sample random z from this learned distribution.
5	Feed $z$into the decoder to generate new data.

Step Description



Real-Life Analogy

Think of an autoencoder like an artist:

- During training, the artist learns how faces look by studying many examples.
- The encoder learns how to describe a face with just a few features (e.g., round face, curly hair, big eyes).
- The decoder learns how to paint a face from that description. When you sample a random combination of features (z) and give it to the artist (decoder), it creates a new, never-seen-before face!