

AGA U3

Deep Belief Networks (DBNs)

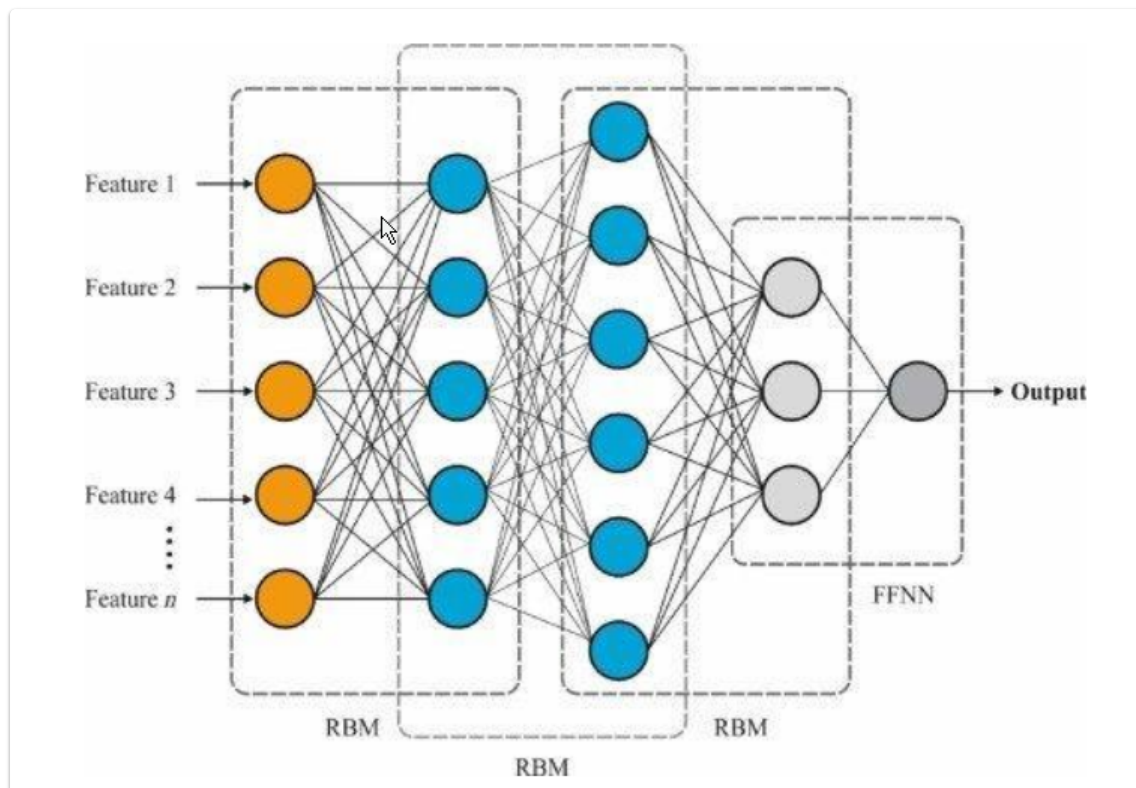
A **Deep Belief Network (DBN)** is a generative graphical model composed of multiple layers of stochastic, hidden variables. It is essentially built by stacking **Restricted Boltzmann Machines (RBMs)** on top of each other.

Each layer in a DBN learns to capture higher-level representations of the data. The first layer learns directly from the input, the second layer learns from the activations of the first hidden layer, and so on.

Architecture Overview

- **Visible/Input layer ($\mathbf{v}$)**: This layer contains observed data.
- **Hidden layers ($\mathbf{h}^{(1)}, \mathbf{h}^{(2)}, \dots$)**: These learn latent features.
- **Connections**:
 - **Undirected** connections exist **between** the top two layers (as in an RBM).
 - **Directed** connections go **downward** from top to bottom layers.

*The top two layers form an undirected **RBM**, and the lower layers form a directed generative model.*



Mathematical Formulation

Joint Distribution in a DBN

A DBN defines a joint distribution over visible vector $\mathbf{v}$ and hidden layers $\mathbf{h}^{(1)}, \dots, \mathbf{h}^{(L)}$ as:

$$P(\mathbf{v}, \mathbf{h}^{(1)}, \dots, \mathbf{h}^{(L)}) = P(\mathbf{h}^{(L-1)}, \mathbf{h}^{(L)}) \prod_{l=1}^{L-2} P(\mathbf{h}^{(l)} | \mathbf{h}^{(l+1)})$$

Where:

- $P(\mathbf{h}^{(L-1)}, \mathbf{h}^{(L)})$ is modeled as an RBM (undirected).
- $P(\mathbf{h}^{(l)} | \mathbf{h}^{(l+1)})$ is a conditional probability (directed downward).

RBM Energy Function

Each RBM layer models the joint probability between two adjacent layers via an energy function:

$$E(\mathbf{v}, \mathbf{h}) = -\mathbf{b}^\top \mathbf{v} - \mathbf{c}^\top \mathbf{h} - \mathbf{v}^\top \mathbf{W} \mathbf{h}$$

Then, the joint distribution is:

$$P(\mathbf{v}, \mathbf{h}) = \frac{1}{Z} \exp(-E(\mathbf{v}, \mathbf{h}))$$

Where:

- $\mathbf{v}$: Visible layer
 - $\mathbf{h}$: Hidden layer
 - $\mathbf{W}$: Weight matrix between visible and hidden units
 - $\mathbf{b}, \mathbf{c}$: Bias vectors for visible and hidden units respectively
 - Z : Partition function (normalization constant)
-

Conditional Probabilities

Each layer's activation is computed using the **sigmoid function**:

- **Hidden unit activation** given visible:

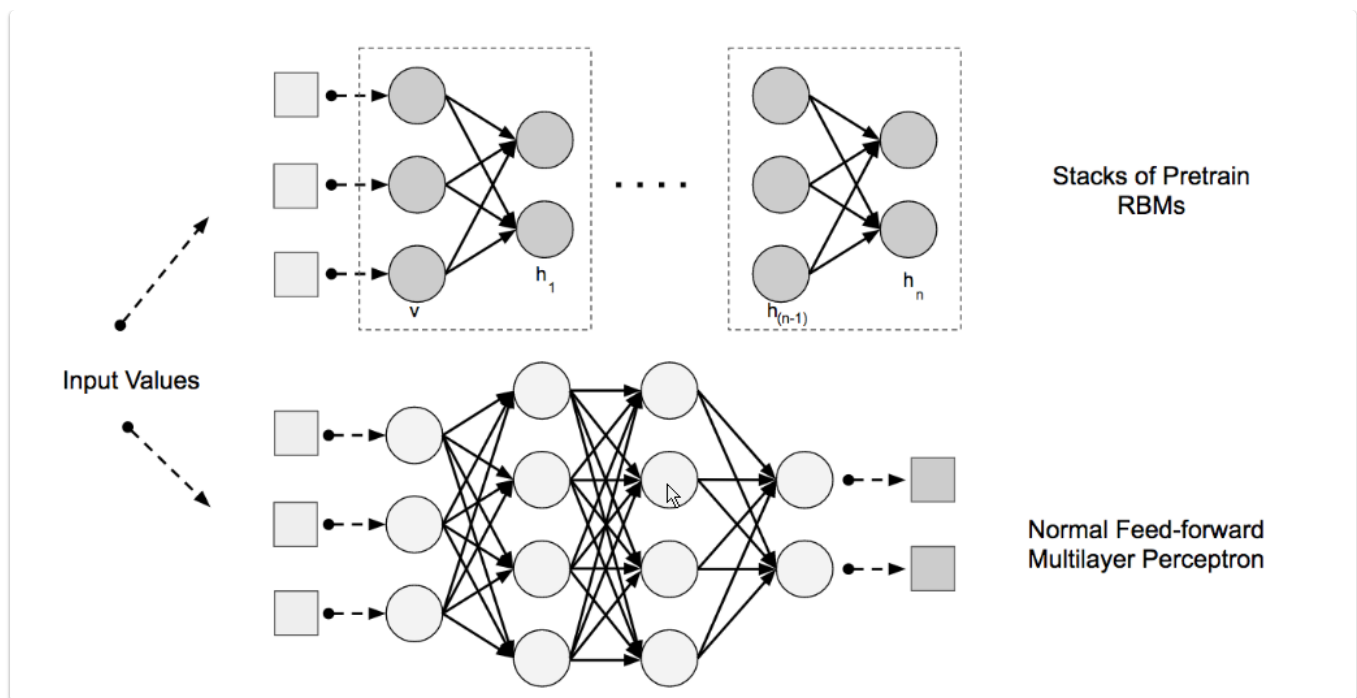
$$P(h_i = 1 | \mathbf{v}) = \sigma \left(\sum_j W_{ji} v_j + c_i \right)$$

- **Visible unit reconstruction** given hidden:

$$P(v_j = 1 | \mathbf{h}) = \sigma \left(\sum_i W_{ji} h_i + b_j \right)$$

Where $\sigma(x) = \frac{1}{1+\exp(-x)}$ is the sigmoid function.

How DBNs Work



DBNs are trained using a two-phase process:

Phase 1: Pre-training (Greedy Layer-wise Training)

- Each layer is trained as a **Restricted Boltzmann Machine (RBM)** using the **Contrastive Divergence (CD)** algorithm.
- Once the first RBM is trained, its hidden activations become the input to the next RBM.
- This phase is **unsupervised** and helps the network to initialize weights intelligently.
- The previous stages are frozen during the training of the subsequent stages.

Contrastive Divergence:

- **Positive Phase:** Sample hidden units from the actual input data.
- **Negative Phase:** Reconstruct visible units from hidden, and again sample hidden.
- **Update the weights** (Rinse and Repeat)
- The weight update encourages the network to better reconstruct the original input.

Phase 2: Fine-tuning (Discriminative Training)

- After pre-training, stack the learned layers into a **Multi-Layer Perceptron (MLP)**.
- Fine-tune using **backpropagation** to minimize classification or regression loss.
- This step is **supervised** and tunes the network for the specific task.

Training Summary

1. **Train RBM 1** on the raw input data **v** using contrastive divergence.
2. **Train RBM 2** on the hidden activations **$h^{(1)}$** from RBM 1.
3. **Repeat** the process to stack multiple RBMs.
4. This creates a deep network that models the data hierarchically.

Challenges in DBNs

- **Intractable Inference**: Due to explaining-away effects and large network size.
 - **Partition Function Complexity**: Computing Z in RBMs is expensive.
 - **No Joint Training**: Most DBNs are trained greedily layer by layer; joint optimization is difficult.
 - **Variational Bounds**: Training improves a variational lower bound on the log-likelihood.
-

Practical Use

After pre-training and fine-tuning:

- The DBN can serve as a **generative model**.
 - Or, use the weights to initialize an **MLP** for better classification performance.
 - Optionally, apply **wake-sleep algorithm** for generative fine-tuning.
-

Conclusion

Deep Belief Networks were an important step in the history of deep learning. They showed how unsupervised learning and generative models could improve supervised tasks by initializing deep architectures more effectively than random initialization.

However, due to their training complexity and inference issues, DBNs are less commonly used today compared to other deep models like autoencoders, CNNs, and transformers.

Numerical:

1. Compute $P(h1=1|v)$, $P(h2=1|v)$, $P(h3=1|h1=1, h2=1)$
 - Input layer: 3 visible units ($v1, v2, v3$)
 - Hidden Layer 1: 2 units ($h1, h2$) — RBM 1
 - Hidden Layer 2: 1 unit ($h3$) — RBM 2

RBM 1 ($v \rightarrow h1, h2$):

- Weights:
 - $w_{v1,h1} = 0.6, w_{v2,h1} = -0.3, w_{v3,h1} = 0.8$
 - $w_{v1,h2} = 0.4, w_{v2,h2} = 0.2, w_{v3,h2} = -0.5$
- Hidden biases: $c_1 = 0.2, c_2 = 0.3$

RBM 2 ($h1, h2 \rightarrow h3$):

- Weights: $w_{h1,h3} = 0.5, w_{h2,h3} = -0.4$
- Hidden bias: $c_3 = 0.1$

Input vector: $v = (1, 0, 1)$

For $h1$:

$$\text{input}_1 = 0.2 + (1)(0.6) + (0)(-0.3) + (1)(0.8) = 1.6 \Rightarrow P(h1 = 1) = \sigma(1.6) \approx 0.832$$

For $h2$:

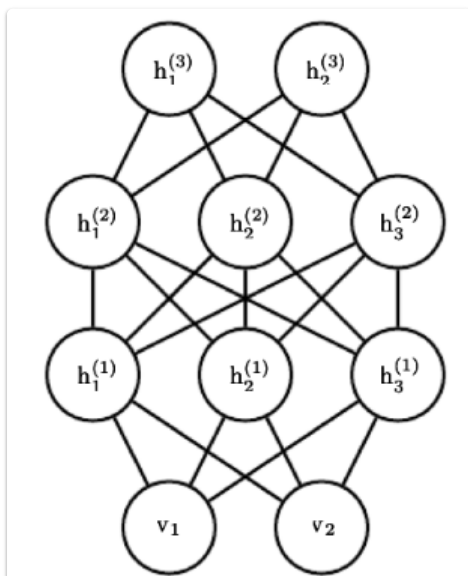
$$\text{input}_2 = 0.3 + (1)(0.4) + (0)(0.2) + (1)(-0.5) = 0.3 + 0.4 - 0.5 = 0.2 \Rightarrow P(h2 = 1) = \sigma(0.2) \approx 0.5498$$

$$P(h3 = 1 | h1, h2)$$

$$\text{input}_3 = 0.1 + (1)(0.5) + (1)(-0.4) = 0.1 + 0.5 - 0.4 = 0.2 \Rightarrow P(h3 = 1) = \sigma(0.2) \approx 0.5498$$

Deep Boltzmann Machine

A **Deep Boltzmann Machine (DBM)** is a **probabilistic generative model** consisting of multiple layers of hidden units. Each layer connects only to adjacent layers and all connections are **undirected**, forming a type of **Markov Random Field**. DBMs can learn hierarchical representations from data. They are trained using **greedy layer-wise pre-training** followed by **fine-tuning** using backpropagation.



Energy Function of DBM

The energy function defines a scalar energy for a configuration of visible and hidden units. For a DBM with visible units v and three hidden layers $h^{(1)}, h^{(2)}, h^{(3)}$, the energy function is:

simplified without bias

$$E(v, h^{(1)}, h^{(2)}, h^{(3)}) = -v^\top W^{(1)} h^{(1)} - h^{(1)\top} W^{(2)} h^{(2)} - h^{(2)\top} W^{(3)} h^{(3)}$$

Including Bias:

$$E(v, h^{(1)}, h^{(2)}, h^{(3)}) = -v^\top W^{(1)} h^{(1)} - h^{(1)\top} W^{(2)} h^{(2)} - h^{(2)\top} W^{(3)} h^{(3)} - b^\top v - c^{(1)\top} h^{(1)} - c^{(2)\top} h^{(2)}$$

where:

- v is the visible layer (input data),
- $h^{(i)}$ is the i -th hidden layer,
- $W^{(i)}$ are the weight matrices connecting adjacent layers.
- $b^\top$ is the bias of the input layer
- $c^{(i)\top}$ are the bias of the hidden layers

output of $V \rightarrow h^{(1)}$ and $h^{(1)} \rightarrow h^{(2)}$ and so on

Probability Distribution

The joint probability of a visible and hidden configuration is given by:

$$P(v, h^{(1)}, h^{(2)}, h^{(3)}) = \frac{1}{Z} \exp(-E(v, h^{(1)}, h^{(2)}, h^{(3)}))$$

where Z is the partition function:

$$Z = \sum_{v, h^{(1)}, h^{(2)}, h^{(3)}} \exp(-E(v, h^{(1)}, h^{(2)}, h^{(3)}))$$

This sum over all configurations makes exact inference intractable for deep models.

Conditional Distributions

Due to the structure of the DBM (only adjacent layers are connected), the conditional distributions factorize nicely. For binary stochastic units, the activation probabilities for each layer are:

- For the first hidden layer:

$$P(h_i^{(1)} = 1 \mid v, h^{(2)}) = \sigma \left(\sum_j W_{ji}^{(1)} v_j + \sum_k W_{ik}^{(2)} h_k^{(2)} \right)$$

- For the second hidden layer:

$$P(h_k^{(2)} = 1 \mid h^{(1)}, h^{(3)}) = \sigma \left(\sum_i W_{ik}^{(2)} h_i^{(1)} + \sum_l W_{kl}^{(3)} h_l^{(3)} \right)$$

- For the third hidden layer:

$$P(h_l^{(3)} = 1 \mid h^{(2)}) = \sigma \left(\sum_k W_{kl}^{(3)} h_k^{(2)} \right)$$

Here, $\sigma(x) = \frac{1}{1+e^{-x}}$ is the sigmoid function.

Training

Training a DBM involves maximizing the likelihood of the training data. Due to the intractability of computing the partition function Z , approximate methods like **Contrastive Divergence**, **Persistent Contrastive Divergence**, or **Mean-Field Variational Inference** are used.

Typically, DBMs are **pre-trained** greedily, layer-by-layer, using **Restricted Boltzmann Machines (RBMs)** and then fine-tuned jointly.

Numerical:

Structure:

- **Visible layer (v):** 2 units $\rightarrow v_1, v_2$
- **Hidden layer 1 (h¹):** 2 units $\rightarrow h_1^1, h_2^1$
- **Hidden layer 2 (h²):** 1 unit $\rightarrow h_1^2$

Weights:

- Between visible and first hidden layer:

$$W^{(1)} = \begin{bmatrix} 0.5 & -0.3 \\ 0.8 & 0.2 \end{bmatrix} \quad (\text{size: } 2 \times 2)$$

- Between hidden layer 1 and hidden layer 2:

$$W^{(2)} = \begin{bmatrix} 0.7 \\ -0.6 \end{bmatrix} \quad (\text{size: } 2 \times 1)$$

Assume:

- $\mathbf{v} = [1, 0]$
- $\mathbf{h}^1 = [1, 1]$
- $\mathbf{h}^2 = [0]$

The energy function for a DBM (simplified without bias) is:

$$E(v, h^1, h^2) = -v^\top W^{(1)} h^1 - h^{1\top} W^{(2)} h^2$$

Break it down:

Part 1: $-v^\top W^{(1)} h^1$

$$v^\top = [1, 0], \quad h^1 = [1, 1] \Rightarrow [1, 0] \cdot W^{(1)} = [0.5, -0.3] \Rightarrow [0.5, -0.3] \cdot [1, 1] = 0.5 - 0.3 = 0.2$$

Part 2: $-h^{1\top} W^{(2)} h^2$

$$h^1 = [1, 1], \quad h^2 = [0] \Rightarrow h^{1\top} W^{(2)} = [0.7, -0.6] \Rightarrow [0.7, -0.6] \cdot [0] = 0$$

Total Energy:

$$E = -(0.2 + 0) = -0.2$$

Compute Unnormalized Probability

$$P(v, h^1, h^2) \propto e^{-E} = e^{0.2} \approx 1.221$$

Numerical 2:

Structure:

- Visible layer (v): 2 units $\rightarrow v_1, v_2$
- Hidden layer 1 (h^1): 2 units $\rightarrow h_1^1, h_2^1$
- Hidden layer 2 (h^2): 1 unit $\rightarrow h_1^2$

Parameters:

Weights:

- $W^{(1)} = \begin{bmatrix} 0.4 & -0.2 \\ 0.7 & 0.1 \end{bmatrix}$
- $W^{(2)} = \begin{bmatrix} 0.6 \\ -0.5 \end{bmatrix}$

Biases:

- Visible bias: $b_v = [0.3, -0.1]$
- Hidden layer 1 bias: $b_{h^1} = [0.2, 0.4]$
- Hidden layer 2 bias: $b_{h^2} = [0.1]$

Let:

- $\mathbf{v} = [1, 1]$
- $\mathbf{h}^1 = [0, 1]$
- $\mathbf{h}^2 = [1]$

The energy of a DBM is:

$$E(v, h^1, h^2) = - \left(\sum_i b_i^v v_i + \sum_j b_j^{h^1} h_j^1 + \sum_k b_k^{h^2} h_k^2 + v^\top W^{(1)} h^1 + h^{1\top} W^{(2)} h^2 \right)$$

Let's compute each term:

A. Bias terms:

- $\sum b_i^v v_i = 0.3 \cdot 1 + (-0.1) \cdot 1 = 0.2$
- $\sum b_j^{h^1} h_j^1 = 0.2 \cdot 0 + 0.4 \cdot 1 = 0.4$
- $\sum b_k^{h^2} h_k^2 = 0.1 \cdot 1 = 0.1$

→ Total bias sum = $0.2 + 0.4 + 0.1 = 0.7$

B. Interaction terms:

1. $v^\top W^{(1)} h^1$

$$v = [1, 1], h^1 = [0, 1]$$

→ Compute $v^\top W^{(1)} = [1, 1] \cdot \begin{bmatrix} 0.4 & -0.2 \\ 0.7 & 0.1 \end{bmatrix} = [1.1, -0.1]$

→ Dot with $h^1 = [0, 1]$:

$$[1.1, -0.1] \cdot [0, 1] = -0.1$$

2. $h^{1\top} W^{(2)} h^2$

$$h^1 = [0, 1], h^2 = [1]$$

→ $h^{1\top} W^{(2)} = [0.6, -0.5]$,

→ Dot with $h^2 = [1]$:

$$[0.6, -0.5] \cdot [1] = -0.5 \text{ (since only second hidden node is active)}$$

Total Energy:

$$E = -(0.7 + (-0.1) + (-0.5)) = -(0.7 - 0.1 - 0.5) = -0.1$$

Compute Unnormalized Probability

$$P(v, h^1, h^2) \propto e^{-E} = e^{0.1} \approx 1.105$$

Boltzmann Machines for Real-Valued Data

While Boltzmann Machines were originally developed for **binary data**, many real-world applications such as **image** and **audio modeling** require modeling **real-valued data**.

Representing Real Values in Binary Models

- For inputs like grayscale images (with pixel intensities in $[0, 1]$), we can treat them as the **expected value** of a binary variable.
- In this approach:
 - Each pixel represents the **probability of being 1**.
 - Binary images are **sampled independently** from this distribution.

This shortcut is common for evaluating binary models on real-valued datasets like MNIST. However, it's not theoretically satisfying and often results in **noisy-looking binary images**.

Gaussian-Bernoulli RBM

To natively support **real-valued visible units**, the Gaussian-Bernoulli RBM is used:

- **Visible units**: real-valued (Gaussian distributed)
- **Hidden units**: binary

- The conditional distribution of visible units is **Gaussian**, with the **mean depending on hidden unit activations**.

Energy Function of Gaussian-Bernoulli RBM

To allow real-valued inputs, we modify the standard RBM energy function by adding a **quadratic term**. One common formulation assumes **unit variance (identity covariance matrix)**:

$$E(v, h) = \sum_{i=1}^D \frac{(v_i - b_i)^2}{2} - \sum_{i=1}^D \sum_{j=1}^F v_i W_{ij} h_j - \sum_{j=1}^F c_j h_j$$

- v_i is the i -th visible unit (real-valued)
- h_j is the j -th hidden unit (binary)
- W_{ij} is the weight between visible unit i and hidden unit j
- b_i is the bias of visible unit i
- c_j is the bias of hidden unit j
- D : number of visible units
- F : number of hidden units

Generalized Form (Non-Unit Variance)

If we **do not assume unit variance**, we introduce the **standard deviation σ_i** for each visible unit:

$$E(v, h) = \sum_{i=1}^D \frac{(v_i - b_i)^2}{2\sigma_i^2} - \sum_{i=1}^D \sum_{j=1}^F \frac{v_i}{\sigma_i} W_{ij} h_j - \sum_{j=1}^F c_j h_j$$

- This version adjusts the interaction term to account for variance in visible units.
- Biases b_i and c_j remain the same.

Key Notes

- This model belongs to the family of **exponential family** RBMs, tailored to specific types of data distributions.
- The **partition function** normalizes the probability distribution, so any constant term f that only depends on parameters (not variables) can be omitted from the energy function.

Undirected Models of Conditional Covariance

The Challenge

Much of the information in natural images lies in the **covariance between pixels** rather than their raw pixel values.

Since the **Gaussian RBM** only models the **conditional mean** of the input given the hidden units, it **cannot capture conditional covariance** information.

This means important structure in data like images cannot be represented adequately.

Solutions

To address this, several models have been proposed:

- Mean and Covariance RBM (mcRBM)
 - Mean-product of Student's t-distribution (mPoT)
 - Spike and Slab RBM (ssRBM)
-

a) Mean and Covariance RBM (mcRBM)

The **mcRBM** independently models the **conditional mean and covariance** of the visible units.

- The hidden units are split into:
 - Binary **mean units** ($h^{(m)}$)
 - Binary **covariance units** ($h^{(c)}$)

The **total energy function** is a sum of two components:

$$E_{\text{mc}}(v, h^{(m)}, h^{(c)}) = E_m(v, h^{(m)}) + E_c(v, h^{(c)})$$

- (E_m) is the standard Gaussian-Bernoulli RBM energy function.
- (E_c) is the energy function of the covariance RBM.

The covariance energy term is typically written as:

$$E_c(v, h^{(c)}) = \sum_j h_j^{(c)} \left(\frac{1}{2} v^\top r_j r_j^\top v - b_j^{(c)} \right)$$

Where:

- (r_j) is the covariance weight vector for unit ($h_j^{(c)}$)
- ($b^{(c)}$) is the bias for covariance units

Training is challenging due to non-diagonal covariance: Gibbs sampling requires inversion of (C_{mc}^{-1}) per iteration.

b) Mean-Product of Student's t-distributions (mPoT)

This model uses Student's t-distributions (which have heavier tails than Gaussians) to better capture outlier structure.

The **energy function** is given by:

$$E_{\text{mPoT}}(v, h) = \frac{1}{2} \|v\|^2 - \sum_j h_j \left(\frac{1}{2} v^\top r_j r_j^\top v - b_j \right)$$

Where:

- (r_j) is a covariance vector
- (h_j) are binary hidden units

As with mCRBM, this defines a multivariate Gaussian with **non-diagonal covariance**, but based on Student's t-prior.

c) Spike and Slab RBM (ssRBM)

The **ssRBM** introduces for each hidden unit:

- A **binary spike** variable ($h_i \in \{0, 1\}$)
- A **real-valued slab** variable ($s_i \in \mathbb{R}$)

Together, these form a flexible unit that can scale and gate contributions to the visible layer.

The **mean** of visible units conditioned on hidden units is:

$$\mathbb{E}[v \mid h, s] = (h \odot s) W^\top$$

Where:

- $(\odot)$ is the element-wise product
- (W) is the weight matrix
- (h) controls whether a component is active
- (s) controls its intensity

The **spike** variable determines whether the component is included at all, and the **slab** controls its strength.

By marginalizing out the slab variables (s), the **conditional over observations** becomes:

$$p(v \mid h) = \mathcal{N}(v \mid \mu(h), \Sigma(h))$$

This model:

- Supports sparse overcomplete representations
- Allows Gibbs sampling
- Avoids matrix inversion (unlike mCRBM)

However, if parameters are poorly chosen, the covariance matrix may become **non-positive definite**, which invalidates the model.

Comparison Summary

- **mCRBM and mPoT**: Use **hidden activations** ($h_j > 0$) to constrain conditional covariance along directions (r_j). Less ideal in **overcomplete** settings due to difficulty removing overlapping constraints.

- **ssRBM**: Uses **inactive spikes** ($h_i = 0$) to “pinch” the precision matrix along directions ($W_{:,j}$). Sparse activations help maintain a tractable and expressive model in **overcomplete** settings.

Convolutional Boltzmann Machines

The Problem with High-Dimensional Inputs

Natural images are **high-dimensional** and exhibit **spatially structured patterns**. Standard Restricted Boltzmann Machines (RBMs) connect **every visible unit to every hidden unit** using a unique weight, making them inefficient and computationally costly for large inputs like images.

This global connectivity:

- Requires too many parameters
- Ignores spatial locality
- Doesn't scale well to large images

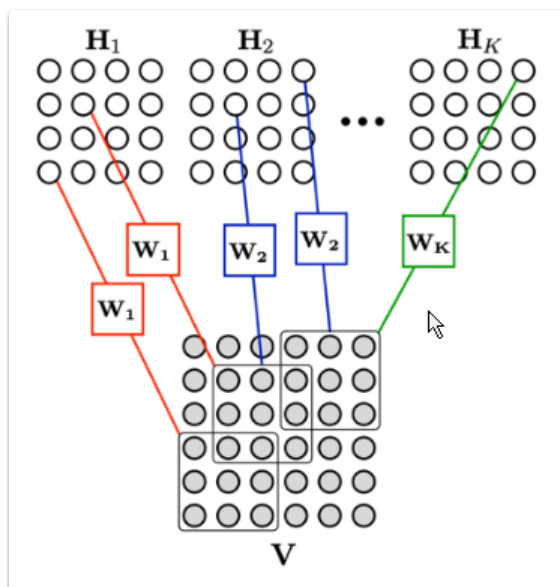
Local Features with Patch-based RBMs

To address this, **patch-based RBMs** train on small local image patches instead of the full image. This:

- Reduces parameter count
- Encourages reuse of local features
- Improves generalization

Limitation: Each patch is treated independently, so **spatial relationships between patches are lost**, resulting in **redundant or uncoordinated features** across neighboring patches.

Enter the Convolutional RBM (CRBM)



The **Convolutional RBM (CRBM)** introduces **structured connectivity** and **weight sharing** to respect the spatial structure of images.

Key characteristics:

- Visible and hidden units are arranged as **matrices** (like images)

- Hidden units are grouped into **feature maps**
- Each feature map detects a specific local pattern at **multiple locations**
- Connections between hidden and visible units are **local** and use **shared filters**

Each hidden feature map (H_k):

- Is a binary matrix indicating where feature (k) is present
- Connects to the visible matrix (V) via a **shared kernel** (W_k)
- Uses **convolution** to apply the same filter across different spatial locations

This allows the CRBM to:

- **Preserve spatial relationships**
- Capture **cooperative features** across overlapping receptive fields
- Extract **hierarchical** representations of images

Pooling in Convolutional Models

Deep convolutional models typically reduce the spatial size of feature maps using **pooling**. In feedforward nets, this is usually:

- **Max pooling**: Selects the most active response
- **Average pooling**: Takes the mean

However, applying these in **energy-based models** like RBMs is not straightforward.

The Challenge with Max Pooling in Boltzmann Machines

Suppose we want a pooling unit (p) to be **on** only if **one of its associated detector units** ($\{d_i\}$) is on.

Naïve implementation:

- Enforces ($p = \max_i d_i$)
- Adds **infinite energy** to any state violating this constraint
- But results in an exponential number of configurations to evaluate (e.g., ($2^9 = 512$) for a 3×3 region)

This quickly becomes **intractable**.

Solution: Probabilistic Max Pooling

Instead, we use **Probabilistic Max Pooling**, a more efficient relaxation.

How it works:

- Each set of (n) detector units has **only ($n + 1$)** valid states:
 - All detectors off
 - Exactly one detector on
- Pooling unit is **on if any detector is on**
- State with all detectors off has **energy zero**
- All other invalid combinations (e.g., multiple detectors on) have **infinite energy**

This reduces complexity from (2^n) to ($n + 1$) states.

Benefits:

- Efficient computation
- Encourages **mutual exclusivity** among detector units (only one active at a time)

Limitations:

- Does not allow **overlapping pooling regions**
 - Acts as a **hard constraint**—can reduce flexibility in modeling
 - Performs worse than standard convolutional classifiers in supervised settings
 - Scaling to variable input sizes or large pooling regions is awkward
 - Edge pixels are difficult to handle cleanly
-

Summary

- CRBMs extend RBMs by using **shared, local filters** to capture **translation-invariant features**
- They structure both visible and hidden units spatially and extract features over **overlapping receptive fields**
- **Probabilistic max pooling** is used to reduce spatial dimensions while maintaining tractability in energy-based models
- Despite their elegance, CRBMs face practical challenges in terms of pooling scalability and classification performance

Boltzmann Machines for Structured/Sequential Outputs: Simplified Notes

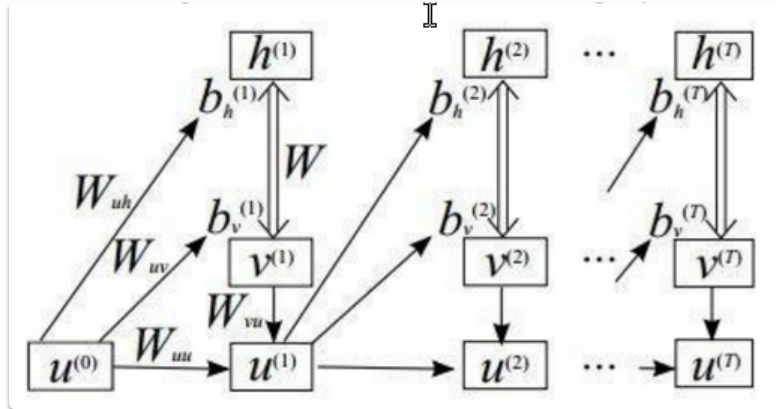
Structured Output Problems

In structured output scenarios, the goal is to predict interdependent elements of a vector or sequence rather than isolated values. For example, in **speech synthesis**, the output y is a waveform where each audio value depends on others to form a coherent sentence. Here, we model the **conditional probability distribution** $p(y|x)$, where dependencies between output elements are critical. Boltzmann Machines (BMs) can be extended to model this by capturing interactions across the entire structured output.

Sequence Modeling

In tasks like **text generation** or **music composition**, we model sequences $p(x^{(1)}, \dots, x^{(\tau)})$ where each element $x^{(t)}$ depends on previous ones. This is often factorized as $p(x^{(t)} | x^{(1)}, \dots, x^{(t-1)})$. For instance, in **motion capture**, sequences of joint angles must evolve smoothly over time. Conditional Boltzmann Machines help model

dependencies between frames, ensuring temporal coherence.



RNN-RBM Architecture for Music Composition

The **RNN-RBM** combines two components:

1. **RNN (Recurrent Neural Network)**: Captures temporal structure by processing sequences step-by-step.
2. **RBM (Restricted Boltzmann Machine)**: Models correlations within each time step (e.g., which musical notes harmonize at time t).

How It Works

- At each time step t , the RNN receives past sequence information and generates **time-varying biases** for the RBM:
 - Visible layer bias: $b_v^{(t)}$ (for notes played at t).
 - Hidden layer bias: $b_h^{(t)}$ (for latent features at t).
- **RBM weights (W)** are **shared across all time steps** and remain fixed.
- The RBM's energy function at time t becomes:

$$E(v^{(t)}, h^{(t)}) = -v^{(t)}W h^{(t)} - b_v^{(t)}v^{(t)} - b_h^{(t)}h^{(t)}.$$

Training the RNN-RBM

1. **Loss Computation**: Contrastive Divergence (CD) approximates the RBM's gradient, measuring divergence between data and model distributions.
2. **Backpropagation Through Time (BPTT)**: The RNN's gradients are computed by unrolling the sequence backward, allowing updates to both RNN and RBM parameters.

Resolving the Parameter Contradiction

A key clarification arises in the RNN-RBM architecture:

- **Misstatement**: Early descriptions claim the RNN emits "all RBM parameters, including weights."
- **Reality**: Only **biases** ($b_v^{(t)}, b_h^{(t)}$) are time-dependent and emitted by the RNN. The **weights (W)** are **shared globally** and not time-varying.

This design balances flexibility (via adaptive biases) and stability (via fixed weights), as defined in the original RNN-RBM model (Boulanger-Lewandowski et al., 2012).

Key Takeaways

- **Structured Output**: Requires modeling dependencies in outputs (e.g., waveforms).
- **Conditional BMs**: Extend BMs to model $p(y|x)$ for structured tasks.
- **RNN-RBM Synergy**:
 - RNN handles **temporal dynamics** (horizontal). (refer to above image)
 - RBM enforces **intra-step structure** (vertical). (refer to above image)
- **Parameter Sharing**: Critical for scalability; only biases adapt per time step.

Other Boltzmann Machines: Training Criteria & Higher-Order Interactions

Generative vs Discriminative Training Criteria

Boltzmann Machines (BM) can be optimized for different objectives:

1. **Generative Criterion**: Maximize $\log p(v)$
 - **Goal**: Model the **data distribution** to generate realistic samples.
 - **Mechanism**: Trained to assign high probability to observed data v (e.g., images, audio).
 - **Example**: Training a BM on MNIST digits to generate new handwritten digits by maximizing $\log p(v)$, where v represents pixel values.
2. **Discriminative Criterion**: Maximize $\log p(y|v)$
 - **Goal**: Predict labels y given inputs v (supervised learning).
 - **Mechanism**: Focuses on learning the conditional distribution $p(y|v)$, ignoring the input's generative properties.
 - **Example**: Classifying MNIST digits by training a BM to maximize $\log p(y|v)$, where y is the digit label and v is the image.

Higher-Order Boltzmann Machines

Most BMs use **second-order interactions** in their energy functions (e.g., pairwise terms like $v_i h_j$). **Higher-order BMs** include terms with **three or more variables**, enabling richer modeling of complex dependencies.

Applications of Higher-Order Interactions

1. **3-Way Interactions**
 - **Example**: A term like $h_j v_i^{(t)} v_k^{(t+1)}$ models spatial transformations between consecutive video frames.
 - **Use Case**: Learning motion patterns in video data, where hidden units (h_j) capture transformations (e.g., rotation, scaling) between frames $v_i^{(t)}$ and $v_k^{(t+1)}$.
2. **Class-Conditional Interactions**
 - Multiply interactions by **one-hot class variables** to gate connections.

- **Example:** For digit classification, a term like $y_c v_i h_j$ allows the relationship between visible (v_i) and hidden (h_j) units to depend on the class y_c .

3. Two Hidden Groups

- **Group A:** Interacts with both input v and label y (class-specific features).
- **Group B:** Interacts only with v (general input features).
- **Use Case:** Jointly learning discriminative and generative features (e.g., classifying digits while generating them).

4. Gated Features with Binary Masks

- Attach a binary mask to visible units to enable/disable features dynamically.
- **Example:** Masking noisy pixels in an image by setting their mask variables to 0, forcing the BM to ignore them during training.

VII. Back-Propagation Through Random Operations

Deterministic vs. Stochastic Computation

Traditional Neural Networks

- **Deterministic Behavior:** For input x , the output is fixed (e.g., $f(x) = \text{ReLU}(Wx + b)$).
Example: Classifying MNIST digits — same image input always produces the same predicted label.

Stochastic Computation in Generative Models

Generative tasks (e.g., image synthesis, text generation) require **controlled randomness** to produce diverse outputs.

- **Solution:** Introduce a **random variable** z (e.g., sampled from $z \sim \mathcal{N}(0, I)$) as an additional input.
- **Stochastic Function:** Define the model as $f(x, z)$, where:
 - x : Task input (e.g., a class label).
 - z : Randomness "dial" — different z values produce variations of the output.

Example:

- Task: Generate diverse images of cats conditioned on $x = \text{"cat"}$.
- Sampling $z_1 \sim \mathcal{N}(0, I) \rightarrow$ Cat sitting.
- Sampling $z_2 \sim \mathcal{N}(0, I) \rightarrow$ Cat jumping.

Why Is Backpropagation Through Randomness Challenging?

- **Problem:** Standard backprop assumes **differentiable, deterministic operations**. Random sampling (e.g., drawing z) is non-differentiable.
- **Consequence:** Direct gradients $\frac{\partial f(x, z)}{\partial \theta}$ cannot be computed if z is sampled naively.

Solutions for Backpropagating Through Randomness

1. Reparameterization Trick (Used in VAEs)

- **Idea:** Decouple randomness from the computational graph by expressing z as a **differentiable function** of parameters.

Example: For $z \sim \mathcal{N}(\mu, \sigma^2)$, rewrite sampling as:

$$z = \mu + \sigma \cdot \epsilon, \quad \epsilon \sim \mathcal{N}(0, 1)$$

Now, gradients can flow through μ and σ (learnable parameters).

- **Result:** The model learns to adjust μ and σ to generate better samples.

2. Score Function Estimator (REINFORCE)

- **Use Case:** For discrete variables (e.g., categorical sampling).
- **Mechanism:** Approximate gradients using Monte Carlo sampling:

$$\nabla_{\theta} \mathbb{E}_{z \sim p_{\theta}}[f(z)] \approx \frac{1}{N} \sum_{i=1}^N f(z_i) \nabla_{\theta} \log p_{\theta}(z_i)$$

- **Drawback:** High variance — requires many samples for stable training.

3. Gumbel-Softmax Trick

- **Use Case:** Differentiable sampling from categorical distributions.
- **Steps:**
 1. Add Gumbel noise to logits: $g = -\log(-\log(\epsilon))$, $\epsilon \sim \text{Uniform}(0, 1)$.
 2. Use softmax with temperature τ to approximate one-hot samples:

$$y_i = \frac{\exp((\log p_i + g_i)/\tau)}{\sum_j \exp((\log p_j + g_j)/\tau)}$$

As $\tau \rightarrow 0$, y becomes discrete; during training, τ is annealed.

Abstraction in Practice: Hiding z for User-Friendly Sampling

In deployed models, the randomness is often **automatically managed**:

- **User Input:** Only x (e.g., "generate a cat").
- **Internal Process:**

```
def generate(x):  
    z = sample_z() # z ~ N(0, I), hidden from the user  
    return model(x, z)
```

Backpropagation in Stochastic Settings

Basic Problem with Randomness in Neural Networks

When dealing with neural networks that include random variables, backpropagation faces a significant challenge. While backpropagation works seamlessly in deterministic networks, the introduction of a random variable z

creates a complication: you cannot directly backpropagate through a random sampling operation (like `z = torch.randn(...)`) because sampling is not differentiable.

The Reparameterization Trick

To overcome this challenge, we can employ the reparameterization trick, which allows backpropagation through a stochastic node. Instead of sampling $z \sim N(\mu, \sigma^2)$ directly, we reformulate the sampling process as:

$$z = \mu + \sigma \cdot \epsilon, \text{ where } \epsilon \sim N(0, 1)$$

This approach separates the randomness from the parameters of the distribution. Here, μ and σ are learnable parameters through which gradients can flow, while ϵ provides the source of randomness.

But this raises a question: don't we still have a term (ϵ) that's being sampled randomly? Wouldn't that require further reparameterization?

The key insight is that we don't need to differentiate through the sampling process of ϵ itself. When we sample $\epsilon \sim N(0, 1)$, it becomes a constant input for that particular forward pass. We only need to backpropagate through the learnable parameters μ and σ . The randomness in z depends on ϵ , but the relationship between z and the parameters μ and σ remains differentiable because they're involved linearly in the expression for z .

Why Not Just Sample from a Standard Normal?

This naturally leads to another question: why bother with learning μ and σ at all? Why not just sample z directly from $N(0, I)$?

The answer lies in the purpose of these models. If we simply sampled $z \sim N(0, I)$, we'd just be generating random noise with no relation to our input data. In models like Variational Autoencoders (VAEs), the goal is to learn a data-dependent distribution where:

1. The encoder learns to map input data x to distribution parameters $\mu(x)$ and $\sigma(x)$
2. We sample from this learned distribution $z \sim N(\mu(x), \sigma(x)^2)$
3. The decoder reconstructs the data from z

By learning $\mu(x)$ and $\sigma(x)$, we capture meaningful features of the input data, allowing the model to generate similar but varied outputs.

The Necessity of Randomness

But why include randomness (ϵ) at all? Why not just use $z = \mu(x)$ directly?

Using only μ would make z entirely deterministic, eliminating the distributional aspect of the model. This would severely limit the model's ability to generate diverse outputs, which is crucial for generative tasks. The randomness provided by ϵ allows the model to sample different points from the learned distribution, giving it the power to generate varied but statistically similar outputs.

If we attempted alternatives like $z = \mu + \sigma$ (without multiplying by ϵ), we'd still be creating a deterministic mapping with no true sampling from the distribution.

The beauty of the reparameterization trick is that it gives us the best of both worlds:

- The ability to sample randomly from our learned distribution

- The ability to backpropagate gradients through the parameters of that distribution

Summary of Approaches

Approach	Backprop possible?	Method
Deterministic $f(x)$	✓ Yes	Normal backprop
Stochastic $f(x, z)$ with Gaussian z	✓ Yes	Reparameterization trick
Stochastic z from discrete/categorical	✓ Yes (but harder)	Score function estimator (e.g., REINFORCE)

The reparameterization trick thus allows us to train models with stochastic components through standard backpropagation, enabling powerful generative models like VAEs to learn meaningful distributions over latent variables.

Backpropagation through Discrete Stochastic Operations

The Challenge with Discrete Outputs

Machine learning models with discrete outputs present a significant challenge for gradient-based optimization because their cost functions are not differentiable at certain points. To overcome this limitation, we use the expectation of the cost function for the gradient descent algorithm. This approach is formalized in the REINFORCE algorithm (REward Increment = nonnegative Factor $\times$ Offset Reinforcement $\times$ Characteristic Eligibility).

The REINFORCE Algorithm

The core insight of the REINFORCE algorithm (Williams, 1992) is that even though $J(f(z; \omega))$ might be a step function with useless derivatives, the expected cost $E_{z \sim p(z)}[J(f(z; \omega))]$ is often a smooth function amenable to gradient descent. While this expectation is typically not tractable for high-dimensional outputs, it can be estimated without bias using a Monte Carlo average.

We can derive the simplest version of REINFORCE by differentiating the expected cost:

$$\nabla_{\omega} E_{z \sim p(z; \omega)}[J(f(z))] = E_{z \sim p(z; \omega)}[J(f(z)) \nabla_{\omega} \log p(z; \omega)]$$

This equation assumes that J does not reference ω directly.

The Variance Problem and Its Solution

A major issue with the simple REINFORCE estimator is its high variance, which can make training unstable. To address this, we can use variance reduction techniques that modify the estimator while keeping its expected value unchanged.

In the context of REINFORCE, this typically involves computing a baseline $b(\omega)$ that is used to offset $J(y)$. Any offset that doesn't depend on y won't change the expectation of the estimated gradient because:

$$E_{z \sim p(z; \omega)}[b(\omega) \nabla_{\omega} \log p(z; \omega)] = b(\omega) E_{z \sim p(z; \omega)}[\nabla_{\omega} \log p(z; \omega)] = b(\omega) \cdot 0 = 0$$

This gives us the variance-reduced gradient estimate:

$$(J(f(z)) - b(\omega)) \nabla_{\omega} \log p(z; \omega)$$

Understanding Stochastic vs. Discrete

An important distinction needs to be made between stochastic processes and discrete outcomes. When a model makes choices like:

- Sampling a word from a vocabulary
- Deciding between yes or no
- Picking an action in reinforcement learning

These are indeed stochastic choices because they involve sampling from a probability distribution. However, they also produce discrete outcomes that aren't differentiable.

The stochastic nature comes from the random sampling process, while the discrete nature refers to the categorical outcome. This combination creates the challenge for backpropagation, as traditional gradient methods require differentiable functions.

Clarifying the Distinction

- **Stochastic**: Involves randomness (sampling from a distribution)
- **Discrete**: Outcomes are categorical or distinct values
- **Differentiable**: Functions that are smooth and continuous, allowing gradient computation

A process can be stochastic regardless of whether its output is discrete or continuous:

- Sampling a word from a softmax is both stochastic (random) and discrete (specific word)
- Sampling a number from a Gaussian is stochastic but continuous

Practical Implementation

REINFORCE-based estimators work by correlating choices of y with corresponding values of $J(y)$. If a good value of y is unlikely under the current parametrization, it might take a long time to obtain it by chance and get the signal that this configuration should be reinforced.

The algorithm essentially says: "If a decision gave a good outcome, make it more likely in the future." This allows us to train models with discrete stochastic operations despite the non-differentiability of the sampling process itself.

In practice, we sample $z \sim p(z; \omega)$, evaluate $J(f(z))$, and use the gradient estimate $J(f(z)) \nabla_{\omega} \log p(z; \omega)$ (or its variance-reduced version) to update the parameters ω .

Intuition Behind Expected Loss and Monte Carlo Estimation in REINFORCE

The Problem: Non-Differentiable Discrete Sampling

When dealing with stochastic networks that involve discrete sampling, traditional backpropagation fails because we cannot compute gradients through a sampling operation. For example, if we have a neural network that predicts actions from a distribution:

$$a \sim \pi(a|x; \theta)$$

where action a is sampled from policy π with parameters θ , we cannot directly apply gradient-based methods because sampling makes the process non-differentiable.

Step 1: Optimize Expected Loss Instead of Sampled Loss

Rather than trying to minimize the loss for a single sampled action, we shift our focus to minimizing the expected loss over all possible actions:

$$E_{a \sim \pi(a|x;\theta)}[J(a)]$$

This means we want to minimize the average loss we'd get if we sampled many actions from our policy π . This expected loss is a smooth function of θ , making it amenable to optimization. However, computing this expectation directly is typically intractable because there could be too many possible actions.

Step 2: Monte Carlo Estimation of the Expectation

Since we can't compute the exact expectation, we approximate it using Monte Carlo sampling:

$$E_{a \sim \pi}[J(a)] \approx \frac{1}{K} \sum_{k=1}^K J(a^{(k)}) \quad \text{where } a^{(k)} \sim \pi(a|x;\theta)$$

We sample a few actions, compute the loss for each, and average them. This gives us a reasonable approximation of the expected loss. The key insight here is that Monte Carlo estimation gives us an unbiased estimate of the expectation, even with just a few samples.

Step 3: The REINFORCE Gradient Estimator

Now we need to optimize θ , but we still can't backpropagate through the sampled actions. This is where the REINFORCE gradient trick comes in:

$$\nabla_{\theta} E_{a \sim \pi(a|\theta)}[J(a)] = E_{a \sim \pi(a|\theta)}[J(a) \nabla_{\theta} \log \pi(a|\theta)]$$

This elegant result moves the gradient inside the expectation using the log-derivative trick. What makes this powerful is that:

- We don't need to backpropagate through a (which is discrete)
- We just score how good a was and reinforce the likelihood of good actions

The Monte Carlo version of this gradient estimator is:

$$\nabla_{\theta} E_a[J(a)] \approx \frac{1}{K} \sum_{k=1}^K J(a^{(k)}) \nabla_{\theta} \log \pi(a^{(k)}|\theta)$$

The Problem with REINFORCE: High Variance

While this method works, it often suffers from high variance in the gradient estimates, making training unstable and inefficient. The outcomes can jump around a lot from sample to sample.

Step 4: Variance Reduction with a Baseline

To address the high variance problem, we modify the gradient estimator by introducing a baseline:

$$\nabla_{\theta} E[J(a)] \approx E[(J(a) - b) \nabla_{\theta} \log \pi(a|\theta)]$$

Where b is a baseline, often implemented as a moving average of rewards or a learned value function. Subtracting this baseline keeps the gradient estimate unbiased but significantly reduces variance.

The intuition is:

- If your sampled action is better than usual (above baseline) → reinforce it more
- If it's worse than usual (below baseline) → reduce its probability

This approach allows the policy to learn more stably, making smaller adjustments when the outcome is close to what's expected.

Summary of How These Components Build on Each Other

1. **Challenge**: Backpropagation doesn't work through discrete/stochastic choices
2. **Solution 1**: Optimize expected loss instead of individual samples
3. **Solution 2**: Approximate expectations with Monte Carlo sampling
4. **Solution 3**: Use the REINFORCE estimator to get gradients without backpropagating through sampling
5. **Solution 4**: Reduce variance using a baseline for more stable learning

This elegant series of techniques allows us to train models with discrete stochastic components using gradient-based methods, opening up possibilities for reinforcement learning, discrete VAEs, and many other applications.

Sigmoid Belief Networks (SBNs)

Core Concept and Structure

A **Sigmoid Belief Network (SBN)** is a directed probabilistic generative model structured as a **Directed Acyclic Graph (DAG)**, typically organized in layers. Unlike undirected models such as Restricted Boltzmann Machines (RBMs), SBNs have directed connections between layers, creating a clear hierarchical structure.

The fundamental building blocks of SBNs are **binary stochastic units** (neurons that output either 0 or 1). Each unit represents a **Bernoulli random variable**, where the probability of activation is computed using a sigmoid function applied to a weighted sum of inputs from the preceding layer:

$$P(h_i^l = 1 | h^{l+1}) = \sigma \left(\sum_j w_{ij} h_j^{l+1} + b_i \right)$$

Where:

- h_i^l is unit i in layer l
- h^{l+1} represents the layer above
- w_{ij} are the weights connecting the layers
- b_i is the bias term
- σ is the sigmoid function

Generative Process

SBNs generate data through a **top-down** process:

1. Sample the top-most hidden layer from a prior distribution (typically uniform or simple Bernoulli)
2. Sample each subsequent layer conditioned on the layer above it
3. Finally generate the visible (data) layer

This generative process can be represented mathematically as:

$$P(v, h^1, h^2, \dots, h^L) = P(h^L) \prod_{l=1}^{L-1} P(h^l | h^{l+1}) \cdot P(v | h^1)$$

Where:

- v is the visible layer
- h^l are the hidden layers
- $P(h^L)$ is the prior distribution for the top layer

Distinction from Deep Belief Networks

Unlike Deep Belief Networks (DBNs), which use an RBM at the top layer, SBNs have independent units in their top layer. This creates a fully directed model throughout the entire network, while DBNs are hybrid models with both directed and undirected connections.

The Inference Challenge

One of the key challenges with SBNs is that **inference** (computing the posterior distribution $P(\text{hidden}|\text{observed})$) is **intractable** in general. This is because calculating the exact posterior would require summing over all possible configurations of hidden variables, which grows exponentially with the network size.

The inference problem can be expressed as:

$$P(h|v) = \frac{P(v, h)}{\sum_{h'} P(v, h')}$$

Where the denominator becomes computationally infeasible for any reasonably sized network.

Variational Methods for Training

Due to the intractability of exact inference, SBNs are typically trained using **variational methods**. These methods:

1. Introduce an approximate posterior distribution $Q(h|v)$ that is easier to compute
2. Minimize the divergence between the approximate and true posterior
3. Optimize a variational lower bound on the log-likelihood instead of the true likelihood

A common approach is to use the variational lower bound:

$$\log P(v) \geq \mathbb{E}_{Q(h|v)}[\log P(v, h)] - \mathbb{E}_{Q(h|v)}[\log Q(h|v)]$$

This bound can be optimized using stochastic gradient descent, potentially with techniques like REINFORCE or reparameterization tricks to handle the stochastic units.

Applications and Significance

SBNs are important in the development of deep generative models as they:

- Provide a probabilistic framework for generating complex data
- Allow for hierarchical representation learning
- Serve as building blocks for more complex models
- Help bridge the gap between directed graphical models and deep neural networks

Despite the inference challenges, SBNs remain valuable for understanding the historical development of generative models that eventually led to more modern approaches like Variational Autoencoders (VAEs) and deep generative models.

Numerical:

Network Setup:

1 hidden layer, 2 hidden units: h_1, h_2

1 visible unit: v

All units are **binary stochastic units** (0 or 1)

Weight matrix from hidden to visible:

$$W = [0.8, -1.0]$$

(i.e., $W_1 = 0.8$ from h_1 , and $W_2 = -1.0$ from h_2)

Bias for visible unit:

$$b = 0.2$$

Hidden layer values:

$$h = [1, 0] \text{ (i.e., } h_1 = 1, h_2 = 0 \text{)}$$

Compute the **probability of the visible unit being 1** given hidden values (i.e., $P(v=1 | h)$), and **sample v**.

1: Compute the Input to Visible Unit

$$z = W_1 \cdot h_1 + W_2 \cdot h_2 + b = (0.8 \cdot 1) + (-1.0 \cdot 0) + 0.2 = 0.8 + 0 + 0.2 = 1.0$$

Apply Sigmoid Activation

unction:

$$\sigma(z) = \frac{1}{1 + e^{-z}} = \frac{1}{1 + e^{-1.0}} \approx 0.731$$

$$P(v = 1 | h) \approx 0.731$$

Sample from the Probability

To simulate binary sampling:

- Generate a random number, say $r=0.65$
- Since $r < 0.731$, we sample $v=1$

Given hidden values: $h=[1,0]$

Probability that $v=1$

Sampled visible unit: **1**

Generator Networks (includes Differentiable Generator Nets)

Basic Concept

Generator networks are parameterized computational procedures designed to generate samples from distributions. These networks define a family of possible distributions to sample from, where the specific distribution is selected by the network parameters.

The core idea is to transform simple input distributions (like random noise) into more complex target distributions through learned transformations.

Architectural Framework

The architecture of a generator network provides the family of distributions it can represent, while its weights determine which specific distribution from that family it will generate samples from. This means:

- The network structure defines what *types* of distributions are possible
- The specific weights select one particular distribution from that family
- Once trained, sampling becomes straightforward: draw input noise and pass it through the network

Example: Generating Samples from Normal Distribution

A simple yet illustrative example is generating samples from a normal distribution $N(\mu, \Sigma)$ with a specified mean and covariance matrix:

1. Begin with samples z from the standard normal distribution $N(0, I)$
2. Feed these samples into a generator network with a single affine layer: $x = g(z) = \mu + Lz$

Where L is the Cholesky decomposition of Σ such that $\Sigma = LL^T$ (with L being lower triangular)

This affine transformation (a linear mapping method that preserves straight lines and planes) effectively warps the standard normal distribution into the desired normal distribution with mean μ and covariance Σ .

Generating Samples from Arbitrary Distributions

For distributions beyond normal, we can apply inverse transform sampling:

1. Define the cumulative distribution function (CDF) $h(c)$ of the target distribution $p(c)$
 - The CDF will have values between 0 and 1
 - Computing $h(c)$ requires an indefinite integral
2. Take input z from uniform distribution $z \sim U(0, 1)$
3. Produce output $h^{-1}(z)$ as the value of $g(z)$

This process requires the ability to compute the inverse of the CDF.

Handling Complex Distributions

For distributions that are:

- Difficult to specify directly
- Challenging to integrate over
- Have integrals that are difficult to invert

We use feedforward networks to represent a parametric family of nonlinear functions g and use training data to infer the parameters that select the desired function.

Understanding Weight Parameterization

When we say "weights determine which specific distribution they represent from a family of possible ones," we mean that:

- The network architecture defines what range of distributions can be modeled (the family)
- The specific weight values select one particular distribution from that family

For example, in a digit generator:

- With one set of weights, the network might generate primarily 3s
- With another set of weights, it might generate mostly 7s
- A third configuration might generate a mixture of digits with particular stylistic properties

Each weight configuration essentially defines a different probability distribution over the output space, while maintaining the same input sampling process (e.g., from standard normal distribution).

Analogy

The generator network can be understood as similar to a music synthesizer:

- The architecture defines the range of possible sounds it can produce
- The weights (like knobs and sliders) select a specific instrument or sound
- The random input values (like keys pressed) determine the actual notes played

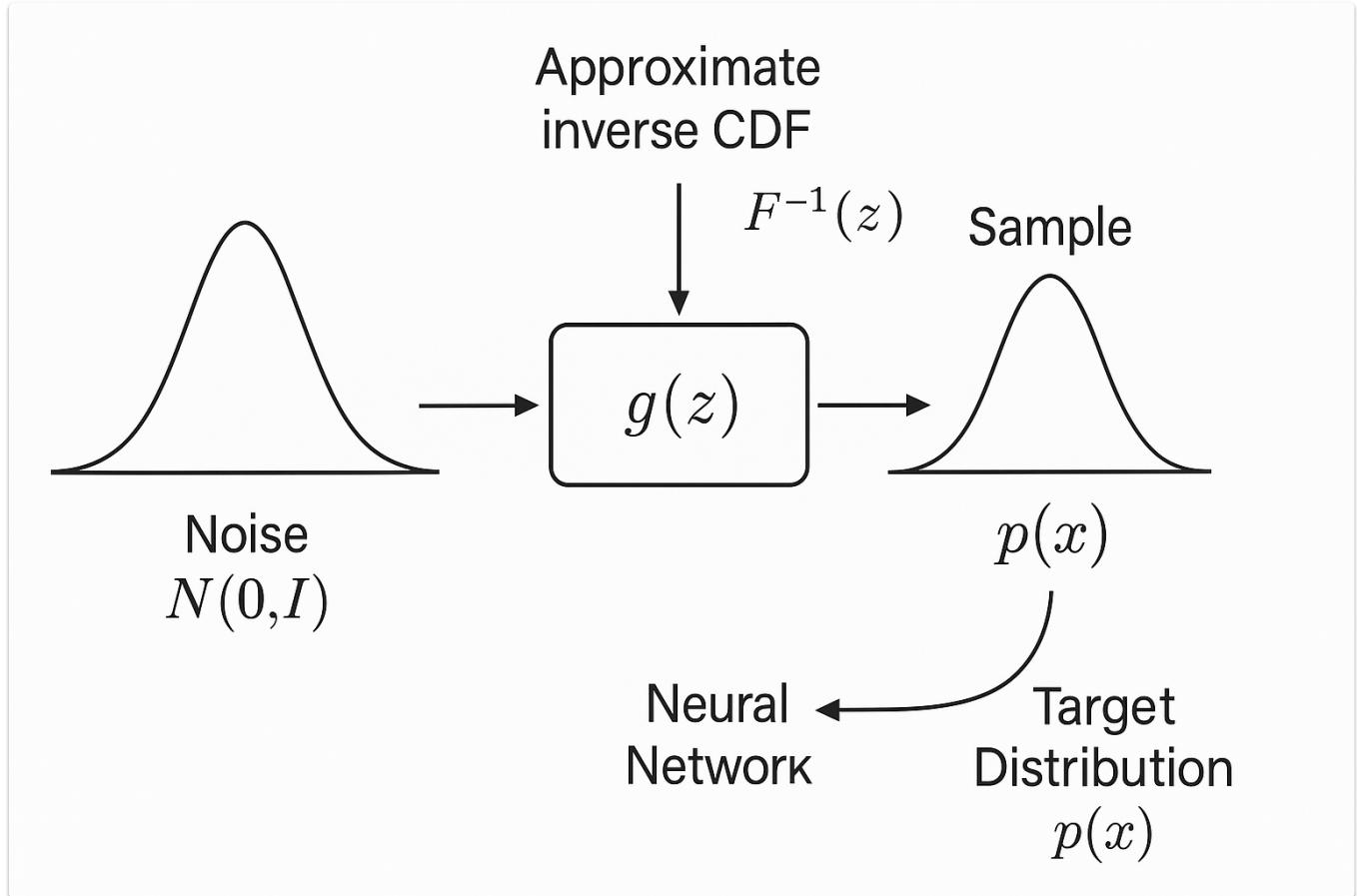
This relationship—where the architecture defines possibilities and weights select specifics—is fundamental to how generator networks transform simple random inputs into complex, structured outputs.

Applications

Generator networks form the foundation of many generative models including:

- Generative Adversarial Networks (GANs)
- Variational Autoencoders (VAEs)
- Flow-based models

These enable applications ranging from image synthesis and style transfer to data augmentation and simulation.



Nonlinear Transformations in Generative Modeling

Mapping Z to X: Change of Variables Principle

Generator networks implement a nonlinear change of variables to transform the distribution over the latent space z into the desired distribution over the output space x . This transformation is governed by the fundamental relationship:

$$p(x) = p(z) \cdot \det \left(\frac{\partial z}{\partial x} \right)$$

This formula implicitly imposes a distribution over x based on:

- The original distribution of z (typically simple, like normal)
- The Jacobian determinant of the transformation

However, this formula is often difficult to evaluate directly for complex generator networks, leading to indirect learning approaches rather than direct maximization of $\log p(x)$.

The Challenge of Generative Modeling

Generative modeling presents greater complexity than classification because:

1. Learning requires optimizing intractable criteria
2. The training data specifies only the output (x) without corresponding inputs (z)

This contrasts with classification, where:

- Both input and output are provided in the training data

- Optimization only needs to learn the mapping between known quantities

In generative modeling, the learning procedure must simultaneously:

- Determine how to arrange the latent z space in a useful way
- Learn how to map z to x effectively

Approaches to Generation

Conditional Generation Strategies

Instead of having g provide samples directly, we can use g to define a conditional distribution over x :

1. Emitting Parameters of a Conditional Distribution:

- Example: Using a generator network whose final layer has sigmoid outputs to provide mean parameters of Bernoulli distributions: $p(x|z) = \text{Bernoulli}(x; \mu = g(z))$
- The overall distribution is defined by marginalizing over z : $p(x) = E_z[p(x|z)]$
- **Advantage:** Can generate both discrete and continuous data

2. Directly Generating a Sample:

- The network directly outputs $x = g(z)$
- **Limitation:** Can only generate continuous data effectively
- While discretization could be introduced in forward propagation, this would break backpropagation

Both approaches define a distribution $p_g(x)$ and allow training using various criteria through the reparameterization trick.

Indirect Learning of $g(z)$

As I'm already familiar with GANs, this indirect learning approach makes more sense now. Since the function $g(z)$ is meant to approximate the original data distribution, but modeling $g()$ directly is challenging because:

1. Computing the true probability $\log p(x)$ requires evaluating complex Jacobian determinants
2. Dealing with intractable integrals becomes computationally infeasible

Instead of direct computation, we learn g indirectly through techniques like:

- Adversarial training (GANs)
- Variational inference (VAEs)
- The reparameterization trick

The generator function $g(z)$ essentially:

- Takes inputs from a simple distribution (e.g., normal)
- Maps them to a complex distribution that mimics real data
- Does this without requiring explicit calculation of probability densities

Motivation for Differentiable Generators

Approaches based on differentiable generator networks are motivated by the success of gradient descent in classification tasks. The central question becomes: can this success transfer to generative modeling?

By using differentiable architectures, we can leverage the power of backpropagation to train these complex transformations, despite the challenges of not having paired (z, x) training data.

This strategy has proven remarkably successful, enabling the development of increasingly powerful generative models that can create realistic data across multiple domains.

Generative Models: Purpose and Comparison

Why Generative Models?

Generative models address fundamental limitations of discriminative models by offering capabilities beyond classification. The key differences include:

Discriminative Models

- **Purpose:** Given an input X , predict a label Y
- **Focus:** Estimate the conditional probability $P(Y|X)$
- **Application:** Classification, regression tasks
- **Example:** CNN classifying images into categories like "cat" or "dog"
- **Limitation:** Cannot model the probability of seeing certain data $P(X)$, thus cannot generate new samples

Generative Models

- **Purpose:** Model the data distribution itself
- **Focus:** Estimate the joint probability $P(X, Y)$ or marginal probability $P(X)$
- **Applications:** Data generation, density estimation, unsupervised learning
- **Capabilities:** Can generate new samples resembling training data
- **Example:** GANs creating new realistic images that weren't in the training set

Generative Adversarial Networks (GANs)

GANs, introduced by Goodfellow et al. (2014), represent a powerful approach to generative modeling based on differentiable generator networks. Their operation involves:

Adversarial Framework

- **Generator Network:** Directly produces samples $x = g(z; \theta^{(g)})$ where z is random noise
- **Discriminator Network:** Attempts to distinguish between real training data and fake samples from the generator
- **Adversarial Dynamic:** The discriminator outputs a probability $d(x; \theta^{(d)})$ indicating the likelihood that x is real rather than generated

Game Theoretic Interpretation

GANs establish a two-player minimax game where:

- The generator tries to minimize the discriminator's ability to distinguish real from fake
- The discriminator tries to maximize its accuracy in detecting fake samples
- This setup leads to a Nash equilibrium where the generator produces samples indistinguishable from real data

Practical Considerations

- GAN training stabilization remains challenging
- Architecture and hyperparameter selection significantly impact performance
- Deep Convolutional GANs (DCGANs) by Radford et al. (2015) demonstrated impressive results for image synthesis
- The latent space of well-trained GANs captures meaningful factors of variation in the data

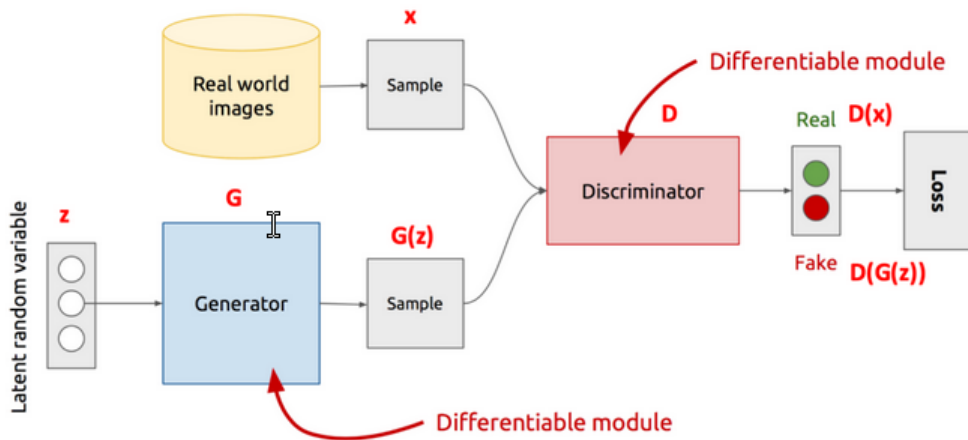
Connecting the Concepts

The indirect learning of the generator function $g(z)$ discussed previously finds its practical application in GANs:

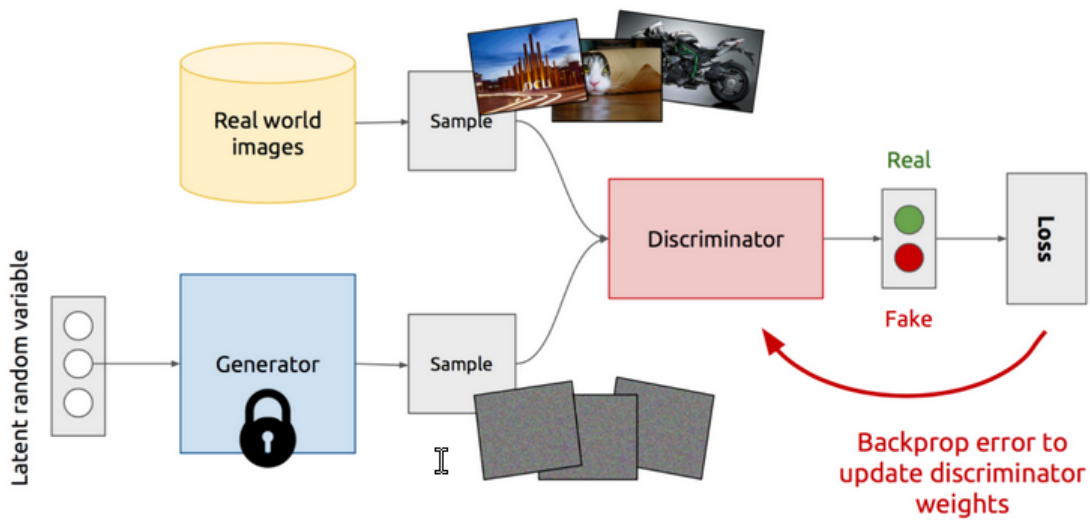
1. Instead of directly computing complex probability distributions:
 - GANs use the discriminator as a learned loss function
 - The generator improves by fooling an increasingly sophisticated discriminator
2. The nonlinear change of variables principle manifests through:
 - The generator network transforming the latent space distribution
 - The gradual reshaping of this transformation to match the real data distribution
3. The two approaches to generation find specific implementations:
 - Direct sample generation is the standard in most GANs
 - Conditional generation can be implemented as conditional GANs where both g and d receive conditioning information

This framework leverages the power of differentiable generator networks while circumventing the need to explicitly compute intractable probability densities, demonstrating how the theoretical foundations of generative modeling translate into practical implementations.

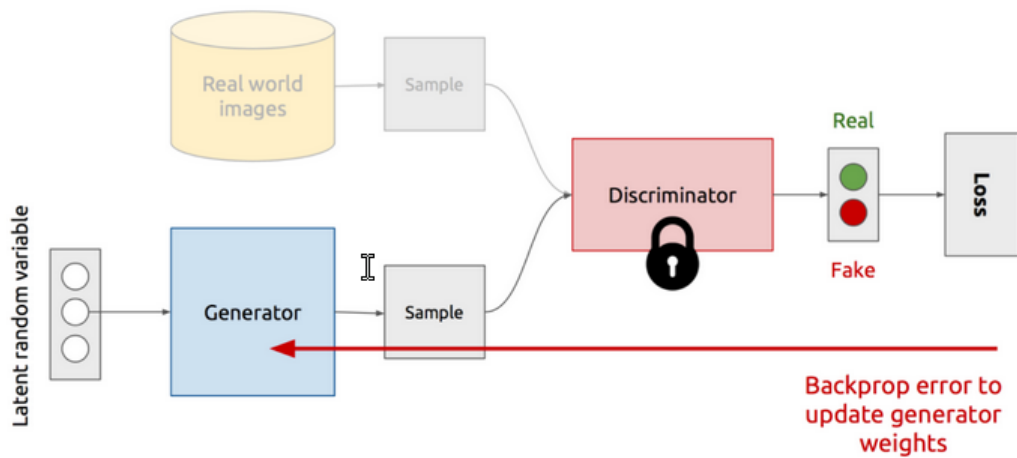
GAN's Architecture



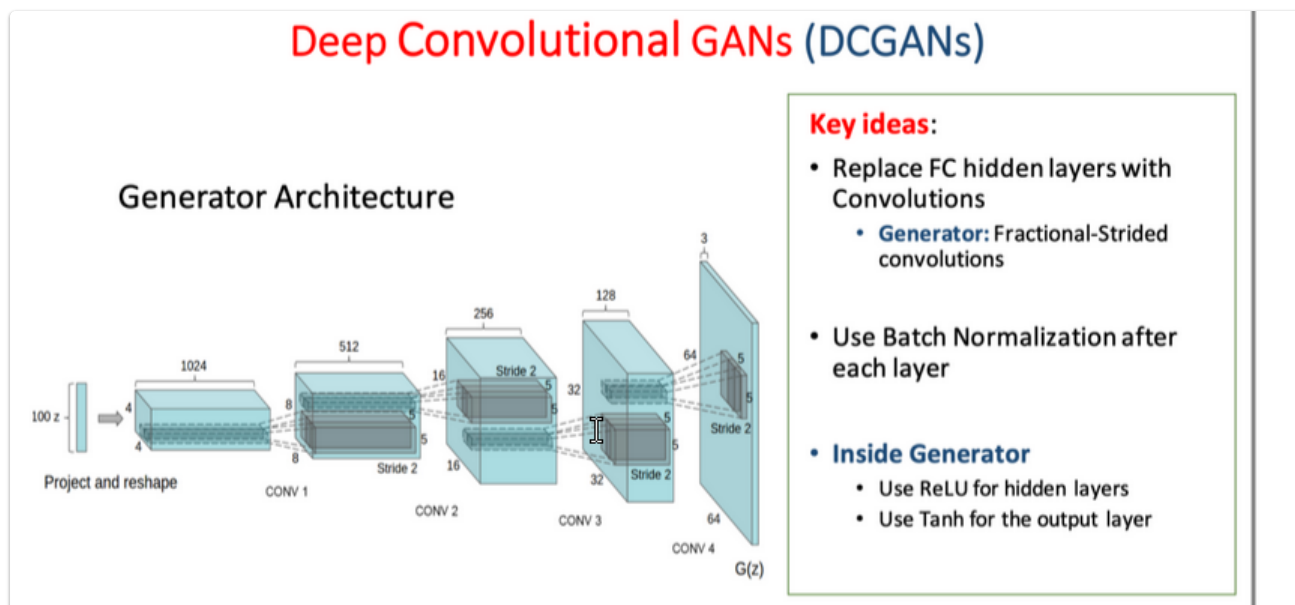
Training Discriminator



Training Generator



Conditional GANs (cGANs)



In a standard GAN, the generator samples from a marginal distribution $p(x)$. However, **Conditional GANs**, introduced by **Mirza and Osindero (2014)**, modify this setup to sample from a **conditional distribution** $p(x | y)$ instead.

Here, y can be any label or additional data (e.g., class labels or attributes), allowing the generator to produce outputs **conditioned on** y . This is incredibly useful for controlled generation — such as generating handwritten digits of a specific class (e.g., '3', '7', etc.).

By feeding y to both the generator and the discriminator, the GAN learns to align the generated data with the conditioning input. This approach opens the door to a lot of fine-grained generation tasks, including conditional image generation and style transfer.

LAPGAN (Laplacian Pyramid of GANs)

To push the envelope further, **Denton et al. (2015)** introduced a stacked approach known as the **LAPGAN** (Laplacian GAN). It consists of a **series of conditional GANs**, each responsible for generating images at **increasing levels of detail**.

This draws from the idea of a **Laplacian pyramid**, a classic image processing technique where an image is represented at multiple resolutions:

- A **Gaussian pyramid** progressively blurs and downsamples the image.
- The **Laplacian pyramid** stores the difference between each level of the Gaussian pyramid and an upsampled version of the next lower level.

In LAPGANs, the generator at each level generates a residual image that is added to an upsampled lower-resolution image, effectively **refining the output** stage by stage. The top level generates a **low-res base image**, and subsequent levels add high-frequency details.

What makes this so compelling is that LAPGAN-generated images can **fool not just discriminators but humans as well**. In fact, experimental subjects mistook up to **40%** of LAPGAN outputs as real data — a testament to its visual realism.

A Strange Trait of GANs: Zero-Probability Support

One fascinating property of GANs is that they can model **distributions that assign zero probability to the training points** themselves. Unlike models that directly maximize the **log-likelihood** $\log p(x)$ (which would lead to overfitting on training points), GANs train the generator to produce samples **on a manifold** that resembles the training data — without necessarily matching exact samples.

This leads to a **paradox**:

*A GAN may assign a log-likelihood of $-\infty$ to a test point (i.e., zero probability), yet still **generate samples that look convincingly real**.*

To avoid the hard-zero support problem, a workaround is to modify the **final layer of the generator** to **add Gaussian noise** to the outputs. This change allows the generator to act as the mean of a **conditional Gaussian distribution**, ensuring that **all outputs have non-zero probability**, and makes the sampling distribution smooth and continuous.

Dropout and Regularization in Discriminators

Another interesting training insight concerns **Dropout**, especially within the **discriminator**. It was found that **using dropout stochastically while computing gradients for the generator** improves learning. This means that during the training step where the generator is updated, the discriminator's outputs should include stochastic drops (randomly deactivating units).

I was curious: "Why not just halve the weights deterministically instead of using dropout?"

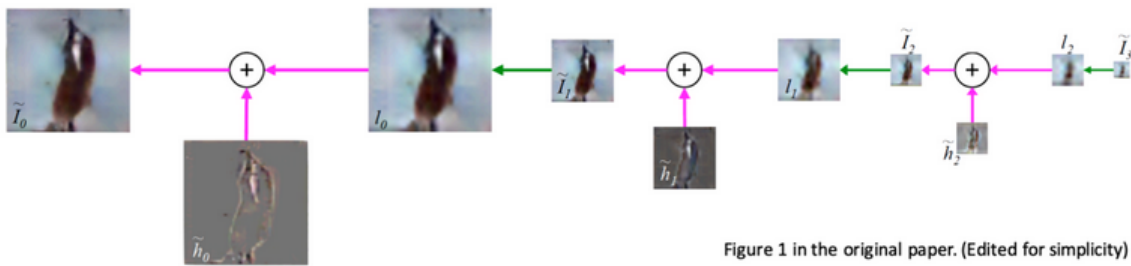
Turns out, using the **deterministic approximation** (e.g., using the weights divided by 2, as in test-time dropout) is **less effective**. Moreover, completely **omitting dropout** yields poor results — reinforcing that the **stochastic behavior adds useful noise** that regularizes the adversarial game and stabilizes gradients.

Summary and Intuition

- **Conditional GANs** allow the generator to produce outputs based on a given condition y , enabling more targeted generation.
- **LAPGAN** stacks multiple conditional GANs in a pyramid structure to generate high-quality images progressively, mimicking the Laplacian pyramid from signal processing.
- GANs **don't need to assign high likelihoods** to data points to work well — they just need to generate samples that look real.
- Adding **Gaussian noise** at the output helps avoid degenerate distributions with zero support.
- **Dropout** in the discriminator is key to keeping the adversarial learning balanced and preventing overfitting.

These ideas push the GAN framework from "just another generator" to **a sophisticated pipeline capable of realism, detail, and control**.

Laplacian Pyramid of Adversarial Networks



- Based on the Laplacian Pyramid representation of images. (1983)
- Generate high resolution (dimension) images by using a hierarchical system of GANs
- Iteratively increase image resolution and quality.

Laplacian Pyramid of Adversarial Networks

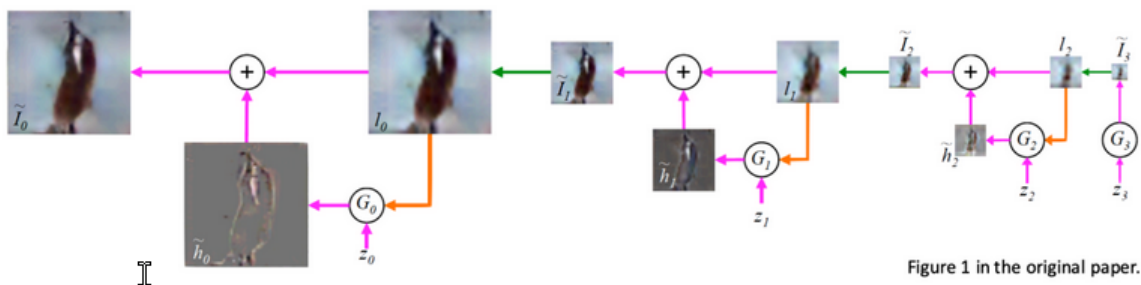


Image Generation using a LAPGAN

- Generator G_3 generates the base image $\tilde{I}_3$ from random noise input z_3 .
- Generators (G_2, G_1, G_0) iteratively generate the *difference image* ($\tilde{h}$) **conditioned on previous small image (I)**.
- This *difference image* is added to an **up-scaled version of previous smaller image**.

Generative Moment Matching Networks (GMMNs) Explained

1. What is Moment Matching?

The goal is to train a generator network such that the **statistical moments** (like mean, variance, skewness, etc.) of its outputs match those of the real data.

- **First moment:** Mean $\rightarrow E[x]$
- **Second moment:** Variance $\rightarrow E[x^2] - (E[x])^2$
- **Higher moments:** Describe shape (e.g., skewness, kurtosis)

In high-dimensional data, moments like $x_i x_j$ multiply in number and complexity, making direct computation expensive or infeasible.



$$\mathbb{E}_{\mathbf{x}} \prod_i x_i^{n_i}$$

where $\mathbf{n} = [n_1, n_2, \dots, n_d]^T$ is a vector of non-negative integers.

2. The Problem with Traditional Moment Matching

- **Computational Burden**: Matching up to second-order moments already requires $\mathcal{O}(d^2)$ operations for d -dimensional data.
 - **Limited Expressiveness**: Low-order moments approximate only **simple distributions** (e.g., Gaussians), missing out on complex real-world patterns in data.
-

3. GMMNs and the Maximum Mean Discrepancy (MMD)

GMMNs avoid explicit moment calculations by minimizing **Maximum Mean Discrepancy (MMD)** — a cost that compares two distributions using kernel-based feature spaces.

Kernel Trick:

Maps data to a high/infinite-dimensional space implicitly:

- Gaussian kernel:

$$k(x, y) = \exp\left(-\frac{\|x - y\|^2}{2\sigma^2}\right)$$

- We never compute the actual feature mapping $\phi(x)$ — the kernel function computes the **inner product** in that space:

$$k(x, y) = \langle \phi(x), \phi(y) \rangle$$

4. Why MMD Works

- **Captures all moments**: By comparing **means** in infinite-dimensional space, MMD implicitly compares all orders of moments.
- **Efficiency**: Thanks to kernels, the computation stays practical without ever explicitly entering infinite dimensions.

Why Infinite Dimensions?

The goal is to **capture all possible moments** (mean, variance, skewness, etc.) of the data. In finite dimensions, we can only represent a limited number of moments. By mapping to an infinite-dimensional space, we can implicitly represent **all moments simultaneously**, allowing us to compare entire distributions, not just specific statistics.

5. Training GMMNs

- Use a **generator network** to map noise to synthetic samples.
 - Use **MMD as the loss function**, comparing real and generated data.
 - No need for adversarial training → Stable and direct.
-

6. Practical Considerations

- **Batch Size**: Larger batches help estimate MMD more accurately.
 - **No explicit probability**: Like GANs, GMMNs don't assign exact probabilities to data points → Encourages **generalization** over memorization.
-

7. GMMNs vs GANs

Feature	GANs	GMMNs
Training	Adversarial (with a discriminator)	Direct (via MMD loss)
Stability	Often unstable	More stable
Flexibility	Learns adaptive focus on discrepancies	Relies on fixed (or learned) kernel
Discrete Data	Needs tricks	Not well-suited either

8. What Does "Mapping to Infinite-Dimensional Space" Mean?

When we say the data is "mapped to an infinite-dimensional space," it means:

- Using a **kernel function** like the Gaussian kernel to **implicitly** compute comparisons in a feature space rich enough to capture all moments.
- We don't compute $\phi(x)$ directly, just the inner products via:

$$k(x, y) = \langle \phi(x), \phi(y) \rangle$$

9. What Are "First Moments in Infinite-Dimensional Space"?

Think of it as taking the **mean of feature vectors** in this rich kernel space.

- Real data mean:

$$\mu_p = \mathbb{E}_{x \sim p}[\phi(x)]$$

- Generated data mean:

$$\mu_q = \mathbb{E}_{x \sim q}[\phi(x)]$$

- **MMD Loss**:

$$\text{MMD}(p, q) = \|\mu_p - \mu_q\|$$

If **MMD** = 0, then the distributions p and q are considered identical in terms of all moments.

Summary

GMMNs offer a **stable, interpretable** alternative to GANs by directly matching the generator output distribution to real data using **kernel-based moment matching** (MMD). Instead of playing a game like GANs, GMMNs focus on making the synthetic data **statistically indistinguishable** from the real data — across **all moments** — using the kernel trick and some clever math.

Convolutional Generative Networks Explained

1. Overview

A **Convolutional Generative Network** is a type of neural network designed to generate high-quality images. It leverages convolutional operations to efficiently model spatial and textural details in images, starting from a low-dimensional input (e.g., random noise or encoded features).

2. Key Components

- **Input:** A low-dimensional latent vector (e.g., random noise). This acts as the "seed" for image generation.
 - **Layers:**
 - **Transposed Convolutions (Deconvolutions):** Upsample low-resolution feature maps to higher resolutions. For example, turning a 4x4 feature map into an 8x8 image.
 - **Un-pooling Layers (optional):** Reverse the effect of pooling layers (explained below).
 - **Activation Functions:**
 - **ReLU** in intermediate layers (for nonlinearity and sparsity).
 - **Tanh/Sigmoid** at the output layer (to scale pixel values to $[-1, 1]$ or $[0, 1]$).
 - **Output:** A realistic image (e.g., 64x64 pixels) with fine details.
-

3. The Challenge with Pooling Layers

In standard **convolutional networks (CNNs)**, pooling layers (e.g., max-pooling) discard spatial information to reduce dimensionality. For example:

- A 2x2 max-pooling layer with stride 2 reduces a 4x4 grid to 2x2 by keeping only the maximum value in each 2x2 window.
 - **Problem:** Pooling is **lossy and non-invertible**. You can't perfectly "undo" pooling because discarded values (e.g., non-maximal pixels) are lost.
-

4. The Workaround: Un-pooling

To reverse pooling in generative networks, **un-pooling** is used. It approximates the inverse of max-pooling under specific assumptions:

1. **Stride = Pooling Width:** If pooling uses a 2x2 window with stride 2, un-pooling places each value back into the top-left corner of a 2x2 region.
2. **Max Value Placement:** The max value from pooling is assumed to originate from the upper-left corner of the original pooling window.
3. **Fill Zeros:** All other positions in the un-pooled region are filled with zeros.

Example:

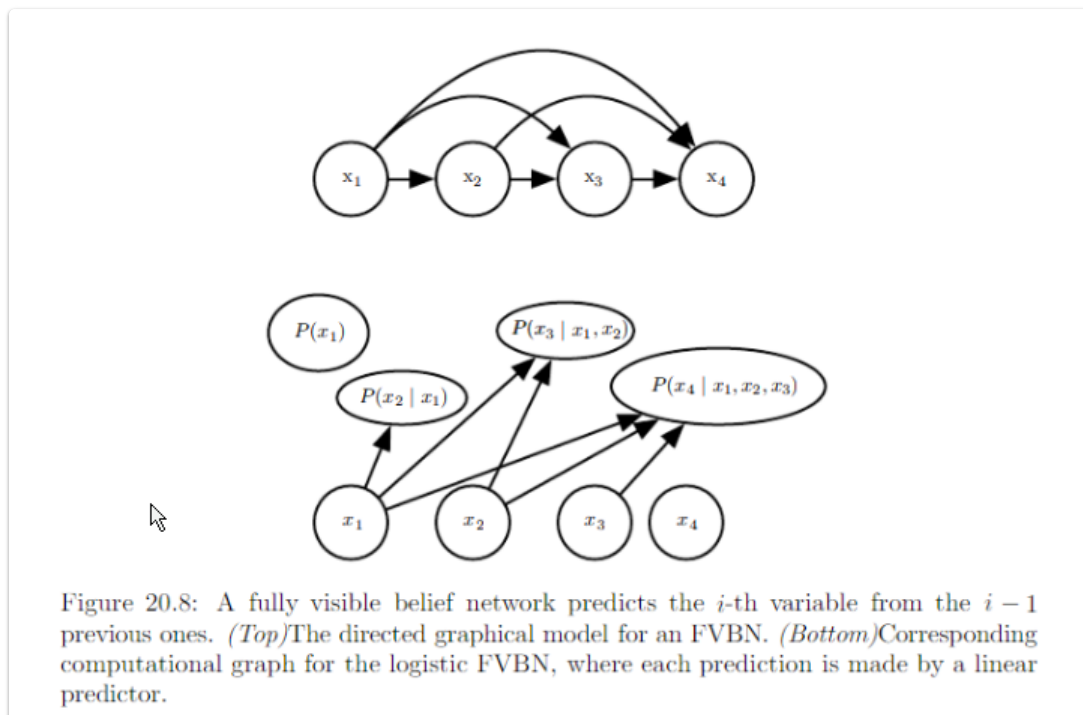
- After max-pooling, a 2x2 region `[[5, 3], [2, 4]]` becomes `5`.
- Un-pooling reverses this to `[[5, 0], [0, 0]]`.

5. Why Does Un-pooling Work Despite Being Unrealistic?

- **Subsequent Layers Compensate:** Layers after un-pooling (e.g., transposed convolutions) learn to "fill in" the zeros during training. For instance, convolutions can infer patterns from neighboring regions to generate plausible textures.
- **End-to-End Training:** The network is trained to minimize the difference between generated and real images. Over time, it learns to correct artifacts introduced by un-pooling.

Linear Auto-Regressive Networks Explained

1. Definition and Structure



Auto-Regressive Model:

Predicts each variable x_i in a sequence based on its predecessors:

$$P(x_i | x_{i-1}, \dots, x_1)$$

Linearity:

Each conditional is modeled **linearly**, depending on data type:

- **Continuous data:** Linear regression
- **Binary data:** Logistic regression
- **Discrete data:** Softmax regression

Characteristics:

- No hidden layers
- No parameter sharing
- Direct mappings from input to output

Example (for $d = 3$ variables):

- x_1 : predicted unconditionally
 - x_2 : predicted from x_1
 - x_3 : predicted from x_1, x_2
-

2. Parameter Complexity

- For d variables, number of parameters grows **quadratically**: $\mathcal{O}(d^2)$
- Why? Each x_i depends on $i - 1$ previous variables:

$$\text{Total parameters} = \sum_{i=1}^d (i - 1) = \frac{d(d - 1)}{2}$$

Implication:

- Doesn't scale well to high-dimensional data like images or long text sequences.
-

3. Connection to Multivariate Gaussian

For continuous variables, the full joint distribution is a **Multivariate Gaussian**:

- Linear conditional dependencies imply specific **covariance structures**
- Example:

$$x_2 = w_1 x_1 + \epsilon \quad \Rightarrow \quad \text{Cov}(x_1, x_2) \neq 0$$

4. Comparison to Linear Classifiers

Aspect	Linear Classifiers	Linear Auto-Regressive Models
Type	Discriminative	Generative
Task	Predict labels	Generate sequences
Example Models	Logistic Regression	Linear regression (sequential)

Pros (Shared):

- **Convex loss** functions (e.g., MSE, cross-entropy)
- **Closed-form solutions** available for linear regression

Cons (Shared):

- Poor at capturing **nonlinear** patterns without manual engineering
-

5. Overcoming Limitations

To capture **nonlinear relationships**:

Basis Expansion:

- Add nonlinear features manually: $x, x^2, x^3, \dots$

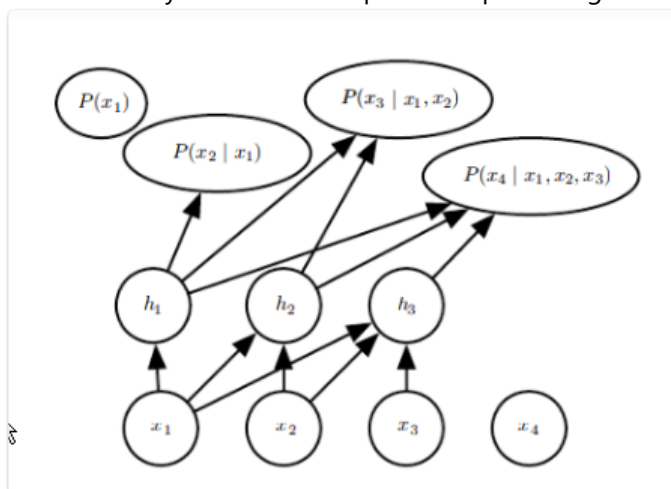
Kernel Trick:

- Implicitly map to high-dimensional space
- Examples: **Radial Basis Function (RBF)**, **polynomial kernels**
- Allows capturing complex dependencies without explicit expansion

Neural auto-regressive network

Neural auto-regressive networks have the same left-to-right graphical model as logistic auto-regressive networks (figure 20.8 above). The new parametrization is more powerful in the sense that its capacity can be increased as much as needed, allowing approximation of any joint distribution. The new parametrization can also improve generalization by introducing a parameter sharing and feature sharing principle common to deep learning in general. By using neural network, 2 advantages are obtained:

1. The parametrization of each $P(X_i | X_1, \dots, X_{i-1})$ by neural network with $(i-1) * k$ input and k output (if the variables are discrete and take k values, encoded one-hot) enables one to estimate the conditional probability without requiring an exponential number of parameters (and examples), yet still is able to capture high order dependencies between the random variables.
2. Instead of having a different neural network for the prediction of each X_i , a left to right connectivity allows one to merge all the neural networks into one. Equivalently, it means that the hidden layer features computed for predicting X_i can be reused for predicting X_{i+k}



this enabled the model to capture and generate non linear relations unlike linear auto regressive networks

How is it Parameterized?

Let's say each variable X_i is discrete and takes k values (e.g., digits 0-9, so $k = 10$).

- X_i is **one-hot encoded** as a vector of length k
- The input to the network at step i is the concatenation of $(i-1)$ one-hot vectors

- So the input size is $(i - 1) \times k$
- The output is a k-dimensional softmax vector predicting probabilities for X_i

Example (Let's say $d = 4$ and $k = 2$)

We want to model 4 binary variables: X_1, X_2, X_3, X_4

We use a **single neural network** to predict all conditionals:

- For X_2 , input: X_1
- For X_3 , input: X_1, X_2
- For X_4 , input: X_1, X_2, X_3

The **network architecture** is designed to take the **full-length input vector**, e.g., $X_1 X_2 \dots X_{d-1}$ to X_d

But we:

- **Zero or mask** irrelevant inputs (e.g., X_4 can't see X_5)
- Keep parameters **fixed**
- Only the **input context** changes

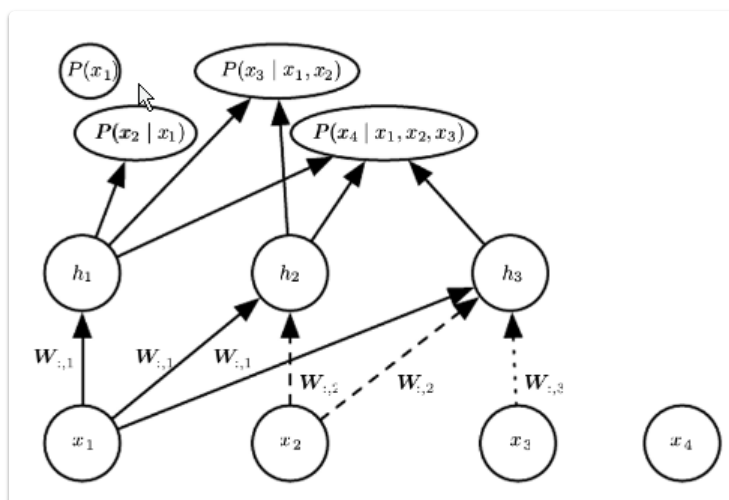
Example Input Vector (for $d = 4$):

```
At time i = 2: [X1, 0, 0, 0]
At time i = 3: [X1, X2, 0, 0]
At time i = 4: [X1, X2, X3, 0]
```

We **don't change** the network weights each time — instead:

- The input layer is padded or zeroed appropriately
- A **mask** in the network ensures each prediction only "sees" allowed inputs

Neural Autoregressive Density Estimator (NADE) :



What is NADE?

NADE is a neural network-based probabilistic model that estimates the joint probability distribution of a multivariate input (e.g., a vector of binary values or pixels in an image).

Instead of modeling the full distribution directly (which is difficult for high-dimensional data), NADE factorizes it as follows (one by one, using the prev variables as input):

$$P(\mathbf{x}) = \prod_{i=1}^d P(x_i \mid x_1, \dots, x_{i-1})$$

This is known as autoregressive factorization — each variable is predicted based on all previous ones.

Key Concepts

Autoregressive

The model predicts each variable using previous variables. This is similar to time series forecasting, where the current value depends on past values.

Conditional Distributions

Rather than modeling the full joint distribution directly, NADE models each conditional distribution ($P(x_i \mid x_{<i})$), which is easier and more tractable.

Neural Networks

Each conditional probability is modeled using a neural network that takes previous variables ($x_1, \dots, x_{i-1}$) as input and outputs a prediction (e.g., a probability for a binary variable).

How NADE Works

1. Factorization

The joint probability is expressed as a product of conditionals:

$$P(x_1, \dots, x_d) = P(x_1) \cdot P(x_2 \mid x_1) \cdot \dots \cdot P(x_d \mid x_1, \dots, x_{d-1})$$

2. Neural Network Training

The neural networks are trained using a dataset of observations, learning the conditional dependencies between variables.

Each conditional is modeled using a feedforward neural network. Instead of having (d) separate networks, NADE uses parameter sharing and masking to allow a single network to handle all conditionals.

3. Density Estimation

Once trained, the model can be used to estimate the probability of any given data point by calculating the product of the conditional probabilities.

To compute the probability of a given input $\mathbf{x}$, the *neural model* outputs the probabilities of each variable conditioned on its predecessors and takes their product which is used for the prediction of the next data point.

4. Sampling

NADE can also be used to generate new data points by sampling from the estimated conditional distributions, starting from an initial value and iteratively generating the next variables

New data can be generated by sampling variables sequentially:

Estimating the **odds (probability)** of choosing each possible value for that variable — **given all the previous ones**.

- Sample $x_1 \sim P(x_1)$
- Sample $x_2 \sim P(x_2 | x_1)$
- ...
- Sample $x_d \sim P(x_d | x_1, \dots, x_{d-1})$
That " $\sim$ " symbol means you're sampling — not taking the most likely (argmax), but **drawing a value according to the probability distribution**.

So for example, if:

$$P(x_3 = 1 | x_1, x_2) = 0.9, P(x_3 = 0 | x_1, x_2) = 0.1 \quad P(x_3 = 1 | x_1, x_2) = 0.9, \quad P(x_3 = 0 | x_1, x_2) = 0.1$$

Then there's a 90% chance you'll get 1, but **10% of the time** you'll get 0.

Deterministic vs. Stochastic Sampling

You've identified a critical difference:

- **If you always pick the most likely value (argmax):**
→ You'll get the same output every time. It's **deterministic generation**.
- **If you truly sample (using random numbers):**
→ You'll get **different outputs each time**, even from the same starting point.
This is **stochastic generation**, which is what NADE is meant to do.

Advantages of NADE

- **Tractability:** The factorized structure allows for exact likelihood computation.
- **Flexibility:** Can handle discrete and continuous variables (with extensions).
- **Generalization:** Learns structured dependencies and generalizes well to new data.

Limitations

- **Sequential Nature:** Generation is slow because it requires sequential sampling.
- **Model Capacity:** Limited by the expressiveness of the neural networks used.
- **Training Complexity:** Training on large datasets can be computationally demanding.

Connection to RBMs and Mean Field Inference (NOT IMPORTANT)

NADE is conceptually similar to Restricted Boltzmann Machines (RBMs) in terms of capturing dependencies among variables, but they differ in directionality and structure.

- RBM is undirected (bidirectional connections), while NADE is directed (left-to-right, sequential).
- Forward propagation in NADE resembles mean field inference in RBMs for filling in missing inputs.

"Forward propagation in a NADE model would loosely resemble the computations performed in mean field inference to fill in missing inputs in an RBM."

This means:

- Mean field inference in RBMs involves iterative passes to infer missing values.
- NADE also uses activations in a neural network, but does so in a single feedforward pass.

"The first step of that inference is the same as in NADE."

"The only difference is that with NADE, the output weights connecting the hidden units to the output are parameterized independently from the weights connecting the input units to the hidden units."

This implies:

- In NADE, input-to-hidden and hidden-to-output weights are **independent**.
- In RBMs, hidden-to-output weights are simply the transpose of the input-to-hidden weights (i.e., tied weights).

This gives NADE more flexibility but at the cost of more parameters.

Summary

- NADE is a neural autoregressive model that decomposes joint probability into a product of conditionals.
- It is efficient, expressive, and tractable due to parameter sharing and clever architecture.
- NADE inspired future models like MADE, PixelCNN, and autoregressive transformers.

Autoencoders, Probability Distributions, and Sampling

1. Autoencoders as Generative Models

A standard autoencoder learns to reconstruct input data by compressing it into a latent space and then decoding it:

$$\hat{x} = g(f(x))$$

Where:

- $f(x)$ is the encoder that maps the input x to a latent representation h
- $g(h)$ is the decoder that reconstructs x from h
- The training objective minimizes the reconstruction loss between x and $\hat{x}$

However, most autoencoders are **deterministic** and do **not** define a probabilistic model of the data distribution $p(x)$.

Probabilistic Autoencoders (e.g., Variational Autoencoders)

Probabilistic autoencoders like **VAEs** model a distribution over the latent space and the data:

- $q(z | x)$ is a learned approximate posterior (encoder)
- $p(x | z)$ is the decoder, modeling the likelihood of x given z
- $p(z)$ is a prior over latent variables (usually standard normal)

To generate new samples:

1. Sample $z \sim p(z)$
2. Sample $x \sim p(x | z)$

This is **ancestral sampling**, since it follows the chain:

$$p(x, z) = p(z) \cdot p(x | z)$$

2. Ancestral Sampling (e.g., NADE)

Autoregressive models like NADE model the **joint distribution** of a multivariate input $x = (x_1, x_2, \dots, x_n)$ using the **chain rule of probability**:

$$p(x_1, x_2, \dots, x_n) = p(x_1) \cdot p(x_2 | x_1) \cdot p(x_3 | x_1, x_2) \cdots p(x_n | x_1, \dots, x_{n-1})$$

Each conditional $p(x_i | x_1, \dots, x_{i-1})$ is modeled by a neural network.

Sampling with NADE

To sample a new x :

1. Sample $x_1 \sim p(x_1)$
2. Sample $x_2 \sim p(x_2 | x_1)$
3. Sample $x_3 \sim p(x_3 | x_1, x_2)$
4. ...
5. Sample $x_n \sim p(x_n | x_1, \dots, x_{n-1})$

This is straightforward **ancestral sampling** since we generate each variable one at a time in order.

3. MCMC Sampling (When Direct Sampling is Hard)

When models do not allow factorization of the joint distribution, we use **Markov Chain Monte Carlo (MCMC)** methods.

Gibbs Sampling (a type of MCMC)

If we have $x = (x_1, x_2, \dots, x_n)$, Gibbs sampling involves:

1. Fix all variables except one, say x_i
2. Sample x_i from its conditional distribution:

$$x_i \sim p(x_i | x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n)$$

3. Repeat for all x_i in a loop
4. After enough steps, the samples approximate the true distribution

This process defines a **Markov Chain** where each state depends only on the previous one.

4. Markov Chain Sampling with Denoising Autoencoders

Denoising autoencoders are trained to **reconstruct clean data** from a **corrupted version**:

$$\tilde{x} \sim q(\tilde{x} | x), \quad \text{then } \hat{x} = g(f(\tilde{x}))$$

Over time, this process can define a **Markov Chain** from which we can sample.

One Step of the Denoising Sampling Process

Let $x^{(t)}$ be the current sample.

1. **Corrupt** the current input:

$$\tilde{x}^{(t)} \sim q(\tilde{x} | x^{(t)})$$

Add noise to simulate a degraded version of the input.

2. **Encode** the corrupted input:

$$h^{(t)} = f(\tilde{x}^{(t)})$$

This gets the latent representation.

3. **Decode** to get reconstruction parameters:

$$w^{(t)} = g(h^{(t)})$$

The decoder outputs the parameters of the distribution $p(x | w)$ (e.g., logits or means).

4. **Sample the next state**:

$$x^{(t+1)} \sim p(x | w^{(t)})$$

This gives a new (cleaner) input sample to continue the chain.

Repeat steps 1–4 to generate diverse samples from the model distribution.

5. Clamping and Conditional Sampling

Sometimes we want to sample **only part of** the input while keeping other parts fixed.

This is known as **clamping**.

Conditional Sampling

To sample from a conditional distribution like:

$$p(x_{\text{free}} | x_{\text{obs}})$$

We:

1. Clamp (fix) x_{obs} — the observed or known part of the input
2. Only resample the **free variables** x_{free}
3. Use the same Markov chain steps, but skip corruption or resampling for clamped variables

This is analogous to **Gibbs sampling** in models like the Restricted Boltzmann Machine.

Summary

Concept	Description
Variational Autoencoder (VAE)	Defines a probability distribution, supports ancestral sampling
NADE	Autoregressive model that samples using chain rule
MCMC	Needed when explicit sampling is not possible

Generative Stochastic Networks (GSNs)

Generative Stochastic Networks (GSNs), introduced by Bengio et al. (2014), are generalizations of denoising autoencoders that include latent variables h in the generative Markov chain, in addition to the visible variables (usually denoted x).

Unlike classical probabilistic models (directed or undirected), which explicitly define the joint distribution $P(x, h)$, GSNs instead **parameterize the generative process itself**. The joint distribution is defined only **implicitly** — if it exists — as the **stationary distribution of the generative Markov chain**.

Architecture

A GSN is defined by two **learned conditional probability distributions** that specify one step of the Markov chain:

1. **Reconstruction distribution**:

$$p(x^{(k)} | h^{(k-1)})$$

This generates the next visible variable $x^{(k)}$ given the current latent state $h^{(k-1)}$. This idea is also used in denoising autoencoders, RBMs, DBNs, and DBMs.

2. **Latent state update**:

$$p(h^{(k)} | x^{(k)}, h^{(k-1)})$$

This updates the latent state h using both the new visible variable and the previous latent state.

Thus, a GSN learns to stochastically transform $(x^{(k-1)}, h^{(k-1)}) \rightarrow (x^{(k)}, h^{(k)})$ through these two steps.

How GSN Sampling Works

GSNs define a **learnable stochastic Markov chain**, where each transition is built on the previous step:

1. **Initialize** with a random latent state $h^{(0)}$.
2. For each step $k = 1, 2, \dots$:
 - Sample a visible variable:
 $x^{(k)} \sim p(x | h^{(k-1)})$
 - Sample the next latent state:
 $h^{(k)} \sim p(h | x^{(k)}, h^{(k-1)})$

Each step depends only on the **previous state** (not the full history), making it a Markov chain.

Is it Random?

Yes — GSN sampling is **stochastic**, but not fully random.

- The distributions $p(x | h)$ and $p(h | x, h)$ are **learned** from data via neural networks.
- The **networks deterministically output** parameters like μ, σ , but **sampling from the distribution is random**.

For example:

- Let $\mu = f(h^{(k-1)})$
- Then:
 $x^{(k)} \sim \mathcal{N}(\mu, \sigma^2 I)$

So the randomness is **guided by learned conditionals** — the process is stochastic but **built on top of the previous sample**.

Comparison to Traditional MCMC

GSNs are related to MCMC, but with key differences:

Feature	Traditional MCMC	GSN
Proposal $q(x^* x)$	Hand-designed	Not used
Transition distributions	Not learned	Learned via neural nets
Stochastic?	Yes	Yes
Acceptance step	Often required	Not needed

Unlike MCMC, which uses a proposal + acceptance scheme, GSN directly learns transitions that gradually move the samples toward the data distribution.

Walk Back Training Procedure

What's the Problem It Solves?

In traditional **denoising autoencoder (DAE)** training, we:

1. Take a data point x
2. Corrupt it to get $\tilde{x}$
3. Train the model to reconstruct x from $\tilde{x}$

But there's a catch:

That process only trains the model to “walk back” **from slightly noisy data** — i.e., from points **very close to the real data**.

So it **might not learn how to recover from points far away** — like samples the model might encounter when it's actually generating new data.

Enter: Walk-Back Training

Instead of doing **just one step** of encode-decode, this method simulates **a mini-Markov chain** of steps, starting at a real data point.

Imagine this:

1. Start at a real data point x
2. Add noise → get $\tilde{x}^{(1)}$
3. Reconstruct it → get $x^{(1)}$
4. Add noise again → get $\tilde{x}^{(2)}$
5. Reconstruct again → get $x^{(2)}$
6. Repeat for k steps...

Then, penalize the model **not just for the first reconstruction**, but **for how far off $x^{(k)}$ is from the original x** , or for all the intermediate ones.

Why Is That Smart?

It **forces the model to learn how to “walk back”** from more corrupted or misaligned places — not just from the local neighborhood of real data.

- Think of it as **teaching the model to navigate back home**, not just from next door, but from **blocks away**.
 - The further it walks, the more it learns about the “terrain” around real data — and how to avoid **spurious (fake) modes** that don’t look like real data.
-

Key Intuition:

- **Regular DAE**: Learns to clean up slightly messed-up inputs
 - **Walk-Back DAE**: Learns to clean up a sequence of increasingly messed-up inputs
 - **Why it helps**: The generative process (like in a GSN) might land in messy places — this teaches it how to find its way back reliably
-

Bottom Line:

Walk-back training simulates how the model would **actually behave during sampling** — walking through space, not jumping once.

It makes the model **robust over longer chains**, and helps **eliminate bad generations** that are far from any real data.