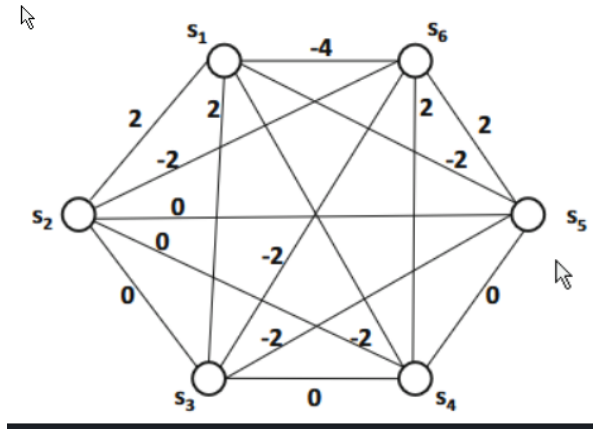


HOPFIELD NETWORKS



Hopfield Networks and Their Learning Process

A **Hopfield network** is a fully connected **recurrent neural network** used for **associative memory**. Each **binary state** in the network corresponds to a dimension (bit) in the training dataset. To store a **d-dimensional dataset**, the Hopfield network requires **d units**, where the **i-th state** represents the **i-th bit** in a training example.

Weights and Their Interpretation

- **Large positive weights** ($w_{ij} > 0$) indicate a strong **positive correlation**, encouraging similar states ($s_i = s_j$).
- **Large negative weights** ($w_{ij} < 0$) indicate a strong **negative correlation**, encouraging opposite states ($s_i \neq s_j$).
- The weights act as **parameters** that encode the relationships between attributes in the training data.

Energy Function in Hopfield Networks and Associative Memory

A **Hopfield network** recalls memories by **iteratively updating states** to minimize the **energy function** until it reaches a **local minimum**. Given an initial **input pattern**, the network flips bits to **reduce energy**, converging to the nearest **stored pattern**.

The **energy** of a Hopfield network for a given state configuration $\mathbf{s} = (s_1, s_2, \dots, s_d)$ is:

$$E = - \sum_i b_i s_i - \sum_{i,j; i < j} w_{ij} s_i s_j$$

(REFER TO how +ve and -ve cause attraction and repulsion in the TRAINING OF HOPFIELD NETWORK PART)

- The **first term** ($-b_i s_i$) encourages **highly biased states** to be **active**.
- The **second term** ($-w_{ij} s_i s_j$) ensures:
 - If $w_{ij} > 0$ (positive correlation), s_i and s_j **tend to align** (attraction).
 - If $w_{ij} < 0$ (negative correlation), s_i and s_j **tend to differ** (repulsion).

Associative Memory Mechanism

1. **Initial Input**: The network starts with a noisy or partial pattern.
2. **Energy Minimization**: It updates states **iteratively**, flipping bits to reduce energy.
3. **Local Minimum Convergence**: The network settles in a state where **no further reduction** in energy is possible.
4. **Memory Retrieval**: The final state **closely resembles** one of the training patterns, effectively recalling a stored memory.

Since Hopfield networks store patterns as **energy minima**, they function as a **content-addressable memory**, retrieving stored patterns from **incomplete or distorted inputs**. 🚀

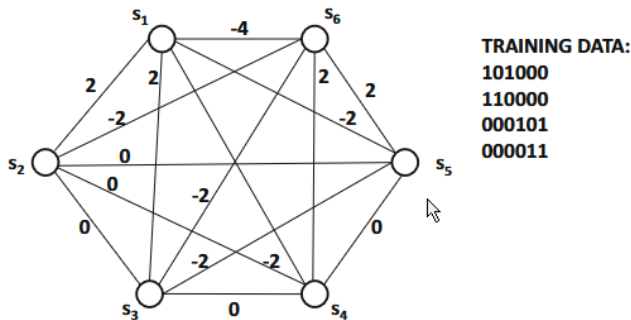
Memory and Retrieval

Hopfield networks are **unsupervised models** that **memorize training data** by adjusting weights to minimize energy when states match binary training attributes. This enables the retrieval of stored data points from **incomplete or corrupted queries** by converging to the nearest **local minimum** in the energy function.

Capacity Limitation

A Hopfield network has a **storage limit** known as its **capacity**. For a network with d units, it can **memorize roughly $0.15d$ independent patterns** before errors occur due to overlapping memory states.

Thus, Hopfield networks function as **content-addressable memory**, efficiently retrieving stored patterns based on noisy or partial inputs.



Finding Local Minima in Hopfield Networks

Once the **weights** of the Hopfield network are **fixed**, the closest **local minimum** to a given state configuration can be found using a **threshold update rule**. This rule iteratively updates each state in the network, moving it towards a **global energy minimum**.

Energy Gap Between State Transitions

To determine whether flipping a state s_i from 0 to 1 is beneficial, we compare the **energy function** values for both cases. Substituting two different values of s_i into the **energy equation**, we obtain the **energy gap**:

$$\Delta E_i = E_{s_i=0} - E_{s_i=1} = b_i + \sum_{j:j \neq i} w_{ij} s_j$$

For the flip of s_i from 0 to 1 to be **attractive**, this energy difference must be **positive**. This leads to the **update rule**:

$$s_i = \begin{cases} 1, & \text{if } \sum_{j:j \neq i} w_{ij} s_j + b_i \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

This rule is applied iteratively to each state s_i until the network reaches a **stable state** (a local energy minimum).

Associative Memory Recall

The **local minima** of a Hopfield network depend on its **trained weights**. To **recall a memory**, one provides a **d-dimensional vector** similar to a stored pattern. The network **converges** to the closest local minimum, which often corresponds to a stored memory. If given a **partial input**, the network can also **recover missing attributes**.

Example: Emergence of Spurious Minima

Consider a Hopfield network where training data contains four binary vectors. The learned weights ensure that these vectors have **low energy**, but **spurious minima** (e.g., 111000) may also appear. Although 111000 is not explicitly present in the training data, it still reflects a key **correlation** in the data:

- The **first three bits** (s_1, s_2, s_3) are **positively correlated**.
- The **last three bits** (s_4, s_5, s_6) are also **positively correlated**.
- **Negative correlations** exist **between these two groups**.

Thus, the **training phase** sets the weights to reflect these data-specific patterns.

Generalization and Capacity

Each **local minimum** found by the **update rule** corresponds to a **memorized pattern** or a **spurious minimum**. However, if the **number of training patterns** exceeds the **network capacity**, closely related patterns may merge into a **generalized cluster center** rather than being memorized separately.

For example, if the training set contains:

- 1110111101
- 1110111110

The Hopfield network may learn 11011111 as a local minimum. This behavior represents **generalization**, where the network implicitly learns a **cluster center** rather than memorizing every training point.

Applications of Hopfield Networks

Hopfield networks can be used for:

1. **Associative memory recall** – retrieving stored patterns from partial inputs.
2. **Error correction** – cleaning corrupted data by converging to a learned pattern.
3. **Attribute completion** – filling in missing attributes by iteratively updating unknown values until convergence.

In attribute completion, the observed states are **fixed**, while the unknown states **update iteratively** until a stable representation is found.

Training a Hopfield Network

To ensure that the **local minima** of a Hopfield network correspond to **training data patterns**, the network must **learn weights** that reflect the structure of the data. This is done using the **Hebbian learning rule**, which strengthens connections between **highly correlated** neurons.

Hebbian Learning Rule

Inspired by biological learning, **Hebbian learning** states that "**neurons that fire together, wire together.**" This means that if **two units frequently have the same value**, their **connection weight increases**, whereas if they often disagree, their **connection weight decreases**.

For a **training set** with n instances, let $x_{ij} \in \{0, 1\}$ be the j th bit of the i th training point. The **weights** in the Hopfield network are computed as:

$$w_{ij} = 4 \sum_{k=1}^n (x_{ki} - 0.5)(x_{kj} - 0.5) \div n$$

Understanding Attraction and Repulsion in Hopfield Networks

The **Hopfield network** stores patterns as **energy minima** in a system, updating states to reach lower energy. The given energy function is:

$$E = - \sum_i b_i s_i - \sum_{i,j;i < j} w_{ij} s_i s_j$$

where:

- (s_i) represents the **state** of the (i^{th}) neuron (typically (+1) or (-1) in binary Hopfield networks).
- (b_i) is a **bias** term influencing whether a neuron tends to be active or inactive.
- (w_{ij}) is the **weight** between neurons (i) and (j).
- The network minimizes (E), leading it to stable states.

1. How Positive Weights Cause State "Attraction"

Consider the term:

$$- \sum_{i,j;i < j} w_{ij} s_i s_j$$

- If ($w_{ij} > 0$) (positive weight), then **minimizing energy** means we want (s_i) and (s_j) to have the **same sign** (both (+1) or both (-1)).

- This means that neurons (i) and (j) tend to "attract" each other, reinforcing the same state.

✓ Example:

- Suppose we have $w_{12} = +2$ between two neurons.
- If $s_1 = +1$, the energy is lower when $s_2 = +1$, so the network prefers this state.
- If $s_1 = -1$, the network prefers $s_2 = -1$.
- This means **neurons prefer to be in the same state** when w_{ij} is positive.

2. How Negative Weights Cause State "Repulsion"

- If $w_{ij} < 0$ (negative weight), then minimizing energy means we want (s_i) and (s_j) to have opposite signs** (one ($+1$) and the other (-1)).
- This means that neurons (i) and (j) tend to "repel" each other, discouraging the same state.

✓ Example:

- Suppose $w_{12} = -3$.
- If $s_1 = +1$, then the energy is minimized when $s_2 = -1$.
- If $s_1 = -1$, then the energy is minimized when $s_2 = +1$.
- So **neurons prefer to be in opposite states** when w_{ij} is negative.

3. Summary of Effects

Weight (w_{ij})	Effect	Example
($w_{ij} > 0$)	Attraction (same state preferred)	($s_1 = +1 \Rightarrow s_2 = +1$)
($w_{ij} < 0$)	Repulsion (opposite states preferred)	($s_1 = +1 \Rightarrow s_2 = -1$)

The network updates itself based on these interactions until it reaches a **stable pattern** (local minimum in energy). These stable states represent **stored memories** in the Hopfield network.

Implications of Hebbian Learning

- **Positively correlated attributes** get **strong positive weights**, reinforcing their similarity.
- **Negatively correlated attributes** receive **negative weights**, ensuring they remain distinct.
- The weights learned reflect **underlying structures and dense regions** in the training data.

By applying this rule, the **Hopfield network memorizes training patterns**, allowing it to **recall stored memories** when given partial or noisy inputs. 🚀

EXAMPLE FOR HOPFIELD NW (Scenario based question [can skip])

Building a Toy Recommender and Its Limitations

Hopfield networks are **memory-based** and are not ideal for **generalization tasks** like standard machine learning. This section explores their **limitations** in **binary collaborative filtering** (recommendation systems).

1 Hopfield Network for Movie Recommendations

- Each user is represented as a **binary state vector**, where **1 = watched movie**, **0 = not watched**.
- Example:
 - Bob watched **Shrek** and **Aladdin**.
 - Alice watched **Gandhi**, **Nero**, and **Terminator**.
- A **fully connected Hopfield network** can be built using all movies as states.
- However, for **large datasets** (e.g., 10^6 movies), there would be 10^{12} **edges**, making it computationally expensive.

2 Addressing the Sparsity Problem

- **Negative sampling** is used to reduce complexity:
 - Each user's network includes **watched movies** + a **small random sample** of unwatched movies.
 - Example:
 - **Alice's network**: 3 watched + 20 sampled unwatched → **23 states**.
 - **Bob's network**: 2 watched + 20 sampled unwatched → **22 states**.
- The **weight-sharing strategy** allows learning across different user networks.
- The **training process** iterates over multiple **smaller networks**, rather than a **single large network**, making it more efficient.

3 Making Movie Recommendations

- A new user (e.g., **Mary**, who watched **E.T.** and **Shrek**) gets **recommendations** based on energy minimization:
 1. Initialize **watched movies** = **1**, others = **0**.
 2. Run the Hopfield network to find the **minimum energy state**.
 3. **Movies with final state = 1** are recommended.

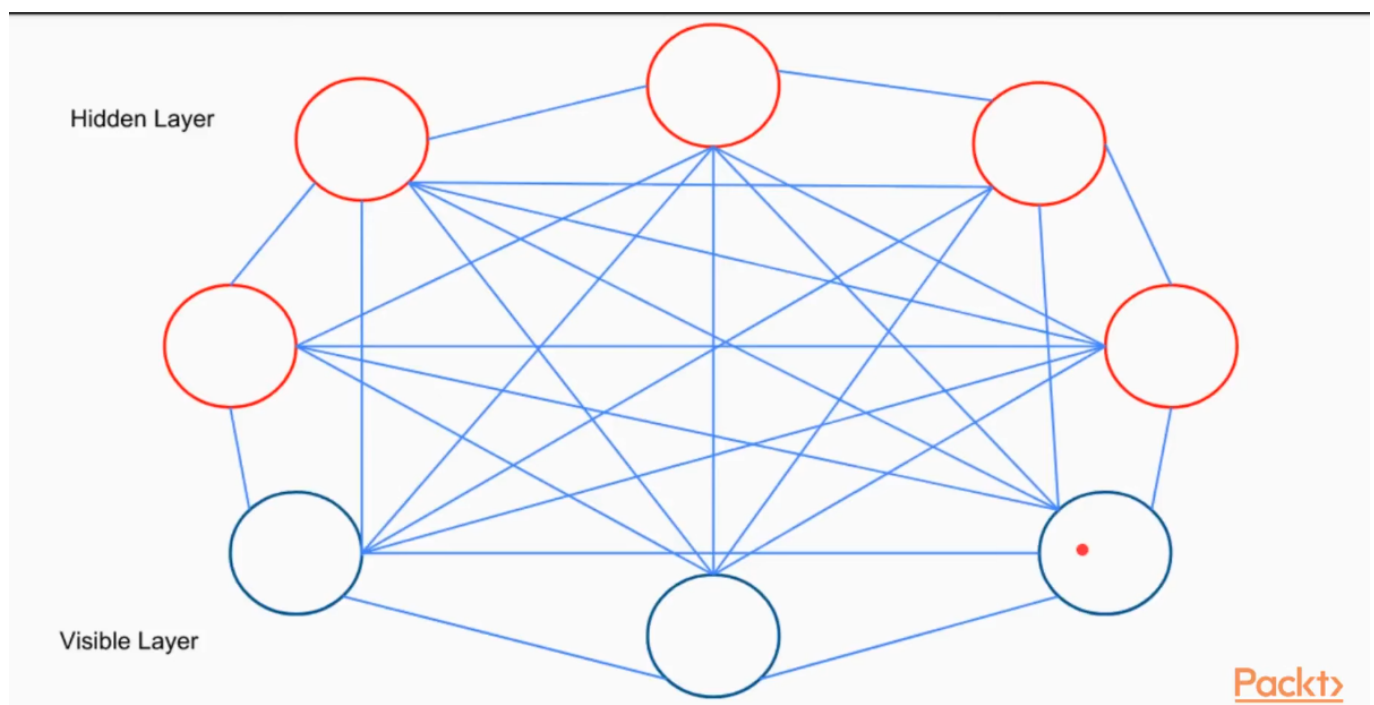
4 Ordering the Recommendations

- Ideally, we **rank** recommendations, but the Hopfield network only produces **binary outputs**.
- **Energy gap** between the two states of a movie can be used for ranking.
- This approach is **naïve** because the Hopfield network is **deterministic**, while recommendation systems often require **probabilistic estimations**.

5 Improving Expressiveness

- A better approach would be to apply **probabilistic transformations** (e.g., **sigmoid function**) to estimate recommendation probabilities.
- Additionally, **latent states** could help capture **hidden patterns** among movies.
- This highlights the need for **more expressive techniques** to enhance Hopfield networks in recommendation tasks.

BOLTZMAN MACHINES: A Probabilistic Generalization of Hopfield Networks



A **Boltzmann machine** extends the **Hopfield network** by introducing **probabilistic state updates** instead of **deterministic ones**.

Energy Function in Boltzmann Machines

A **Boltzmann machine** defines a **probability distribution** over different state configurations **based on energy**. The energy function is similar to that of a **Hopfield network**, but it is used to determine the **likelihood of a state** rather than setting deterministic values.

Energy Function

For a given **state configuration** $s = (v, h)$ (including visible and hidden states), the **energy** is defined as:

$$E(s) = E([v, h]) = - \sum_i b_i s_i - \sum_{i,j:i < j} w_{ij} s_i s_j$$

where:

- $b_i \rightarrow$ Bias term for state s_i .
- $w_{ij} \rightarrow$ Weight between states s_i and s_j .
- $s_i, s_j \rightarrow$ Binary states (0 or 1).
- **Summation constraints** ($i < j$) ensure that each pair is counted only once.

Understanding the Terms

- The **first term** ($-\sum_i b_i s_i$) represents the **bias energy contribution** of each state.
- The **second term** ($-\sum_{i,j:i < j} w_{ij} s_i s_j$) represents the **pairwise interactions** between states.
- Lower energy states are **more probable**, while higher energy states are **less probable**.

Probability of a State Configuration

The probability of a particular **state configuration** s is given by the **Boltzmann distribution**:

$$P(s) = \frac{e^{-E(s)}}{Z}$$

where Z (the **partition function**) ensures that probabilities sum to 1:

$$Z = \sum_s e^{-E(s)}$$

Energy Gap in Boltzmann Machines (and Hopfield Networks)

In a Hopfield network, each state s_i is deterministically set based on the **energy gap**:

$$\Delta E_i = E_{s_i=0} - E_{s_i=1} = b_i + \sum_{j \neq i} w_{ij} s_j$$

- If $\Delta E_i > 0$, then $s_i = \text{closeto} : 1$.
- If $\Delta E_i < 0$, then $s_i = \text{closeto} : 0$.
- This **hard decision rule** makes Hopfield networks deterministic.

Probabilistic Behavior in Boltzmann Machines

Unlike Hopfield networks, a **Boltzmann machine** assigns a **probability** to s_i depending on the **energy gap** instead of a **fixed value**.

- If ΔE_i is **positive**, s_i is **more likely** to be 1 (or close to it).
- Positive energy gaps are assigned probabilities that are larger than 0.5
- The probability of s_i being 1 is given by the **sigmoid function**:

$$P(s_i = 1) = \frac{1}{1 + e^{-\Delta E_i}} \quad \text{eq6.7}$$

This probabilistic nature allows **Boltzmann machines to escape local minima**, unlike Hopfield networks.

This equation gives the **probability that a single unit s_i is activated (1)** based on the energy difference ΔE_i when flipping its state.

- **Use case:** It is used in **Gibbs Sampling** to iteratively update the states in the Boltzmann Machine.
- **Key insight:** A neuron is more likely to be **on (1)** if flipping it lowers energy.

Probability of a State in a Boltzmann Machine (eq 6.9)

In a **Boltzmann Machine**, the probability of a particular state s is determined by its energy $E(s)$. Lower energy states are more likely to occur than higher energy ones. This follows from the fundamental probability distribution:

$$P(s) \propto \exp(-E(s))$$

Since probabilities must sum to 1, we introduce a normalization factor Z , also known as the **partition function**, to ensure a valid probability distribution:

$$P(s) = \frac{\exp(-E(s))}{Z}$$

where Z is computed as:

$$Z = \sum_s \exp(-E(s))$$

This gives the **probability of an entire system configuration** s based on its total energy $E(s)$.

- **Use case:** It is used to define the **overall probability distribution** over all possible states of the system.
- **Key insight:** Lower-energy states are exponentially more likely.

Explanation:

- **Exponentially Weighted Probability:** The probability of a state decreases exponentially with increasing energy. Low-energy states are naturally more probable.
- **Normalization with (Z):** The partition function (Z) ensures that all state probabilities sum to 1, making it a valid probability distribution.
- **Significance:** This probability formulation is crucial in training Boltzmann Machines since it dictates how likely each state is, impacting sampling and learning processes.

By using this formulation, Boltzmann Machines can model complex probability distributions, capturing intricate dependencies in data.

Visible and Hidden States in Boltzmann Machines

A Boltzmann machine consists of **two types of states**:

- **Visible states (v)** → Represent **observed data**.
- **Hidden states (h)** → Capture **latent features** to improve learning.

If a Boltzmann machine has q **total states**, consisting of d **visible states** and m **hidden states**, then:

- The state configuration can be written as:

$$s = (s_1, s_2, \dots, s_q)$$

- Alternatively, explicitly separating visible and hidden states:

$$s = (v, h)$$

where:

- v = set of **visible** states
- h = set of **hidden** states

Data Generation in a Boltzmann Machine

Iterative Sampling & Thermal Equilibrium

In a Boltzmann Machine, generating data is challenging due to **circular dependencies** among states. Since each state influences others, we must iteratively sample new values using a conditional distribution generated from the state values from the previous iteration to satisfy Equation 6.7 across all states.

Ex:

(Imagine you have a group of friends deciding what to wear based on each other's choices. If one person changes their outfit, it might influence others to change theirs too. This creates a **loop of dependencies**—no one can finalize their choice until everyone is satisfied. (refer to the image at the start of Boltzmann machine to identify the loop)

A **Boltzmann Machine** works the same way. Each "neuron" (or state) affects others, so we **can't set all states at once**. Instead, we:

1. Pick one state at a time.
2. Look at the current choices of the others.
3. Decide the best value for it using probabilities.
4. Repeat this over and over until everyone's choices stabilize.)

Processes used:

1. Gibbs Sampling (MCMC Sampling) Process:

- Start with a **random state initialization**.
- Use **Equation 6.7** to compute the conditional probabilities for each state.
- Iteratively sample new values using these probabilities.
- Continue updating until the system reaches **thermal equilibrium** (where probabilities stabilize).

2. Burn-in Time:

- The time required to reach **thermal equilibrium** is called **burn-in time**.
- Once equilibrium is reached, the generated visible state values represent **random samples from the model**.

Correlation in Generated Data

- States with **high weights** between them tend to be **highly correlated**.
- In applications like **text mining**, word states that belong to the same topic will frequently appear together in generated samples.
- This learned correlation structure allows the Boltzmann Machine to **generate meaningful data**.

Comparison with Other Models

- Unlike **Gaussian Mixture Models**, where sampling is direct, a Boltzmann Machine requires **iterative updates to reach equilibrium** before generating valid samples.
- This **undirected nature** makes both **data generation** and **weight learning** significantly more complex.

Learning the Weights of a Boltzmann Machine

In a **Boltzmann Machine**, the goal is to learn weights w_{ij} that maximize (maximize the log likelihood of eq 6.9) the **log-likelihood** of the training data. The log-likelihood is obtained by taking the logarithm of the probability equation:

$$\log P(s) = -E(s) - \log Z$$

where $E(s)$ is the energy function and Z is the partition function.

Computing the Weight Updates

To optimize the weights, we compute the partial derivative of the log-likelihood with respect to w_{ij} :

$$\frac{\partial \log P(s)}{\partial w_{ij}} = \langle s_i s_j \rangle_{\text{data}} - \langle s_i s_j \rangle_{\text{model}}$$

where:

- $\langle s_i s_j \rangle_{\text{data}}$: The average value of $s_i s_j$ when the visible states are fixed to training data.
- $\langle s_i s_j \rangle_{\text{model}}$: The average value of $s_i s_j$ at **thermal equilibrium**, without fixing visible states.

Intuition Behind the Update Rule

- The update strengthens connections between states that are **correlated** in the training data.
- It compares how often s_i and s_j co-occur in the **training data** vs. in the **unrestricted model**.
- By adjusting weights to minimize this difference, the model better captures data dependencies.

This approach ensures that the **learned weights** enhance meaningful correlations present in the training data while reducing the impact of spurious correlations.

From the above discussion, it is clear that two types of samples need to be generated in order to perform the updates:

Data-Centric and Model-Centric Samples

In **Boltzmann Machines**, learning requires generating two types of samples:

data-centric samples and **model-centric samples**. These help compute weight updates by comparing correlations in training data vs. an unconstrained model.

1. Data-Centric Samples

These samples are generated by **clamping** the **visible states** to a randomly chosen training vector and allowing **hidden states** to evolve.

Process:

1. Fix visible states to a training vector.
2. Initialize hidden states randomly from a **Bernoulli distribution** ($p = 0.5$).
3. Recompute hidden states' probabilities using:

$$P(s_i = 1 | s_1, \dots, s_{i-1}, s_{i+1}, \dots, s_q) = \frac{1}{1 + e^{-\Delta E_i}}$$

4. Sample new hidden state values from these probabilities.
5. Repeat until **thermal equilibrium** is reached.
6. The final hidden state values provide the **data-centric samples**.

Key Point:

- Visible states remain fixed, while hidden states adjust based on training data.

2. Model-Centric Samples

These samples are drawn **without constraints**, meaning **both visible and hidden states** evolve freely.

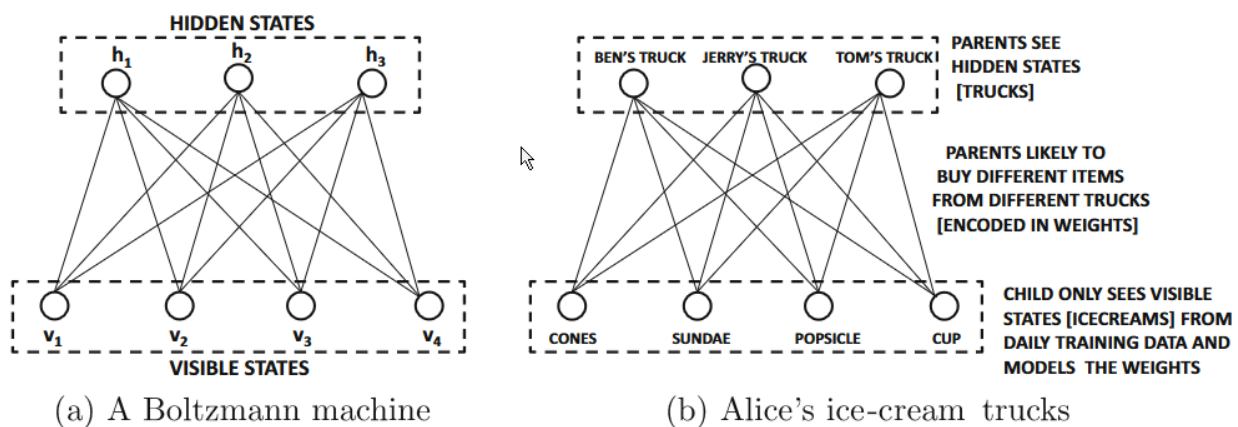


Figure 6.3: A Restricted Boltzmann machine. Note the *restriction* of there being no interactions among either visible or hidden units.

Process:

1. Initialize **both visible and hidden states** randomly.
2. Compute their probabilities using the same equation as in data-centric sampling.
3. Sample new values for **both visible and hidden states**.
4. Repeat until **thermal equilibrium** is reached.

Key Point:

- No fixed visible states; the model generates its own sample distributions.

Why Are These Samples Important?

- **Data-centric samples** help capture correlations between states **given real training data**.
- **Model-centric samples** capture the model's **inherent correlations**, even without training data.

Using **Gibbs Sampling**, we compute:

$$\frac{\partial \log P(s)}{\partial w_{ij}} = \langle s_i s_j \rangle_{\text{data}} - \langle s_i s_j \rangle_{\text{model}}$$

where $\langle s_i s_j \rangle_{\text{data}}$ and $\langle s_i s_j \rangle_{\text{model}}$ are averages over data- and model-centric samples, respectively.

Weight and Bias Updates (Not important)

Using the computed differences:

$$w_{ij} \leftarrow w_{ij} + \alpha (\langle s_i s_j \rangle_{\text{data}} - \langle s_i s_j \rangle_{\text{model}})$$

Bias update follows similarly:

$$b_i \leftarrow b_i + \alpha (\langle s_i, 1 \rangle_{\text{data}} - \langle s_i, 1 \rangle_{\text{model}})$$

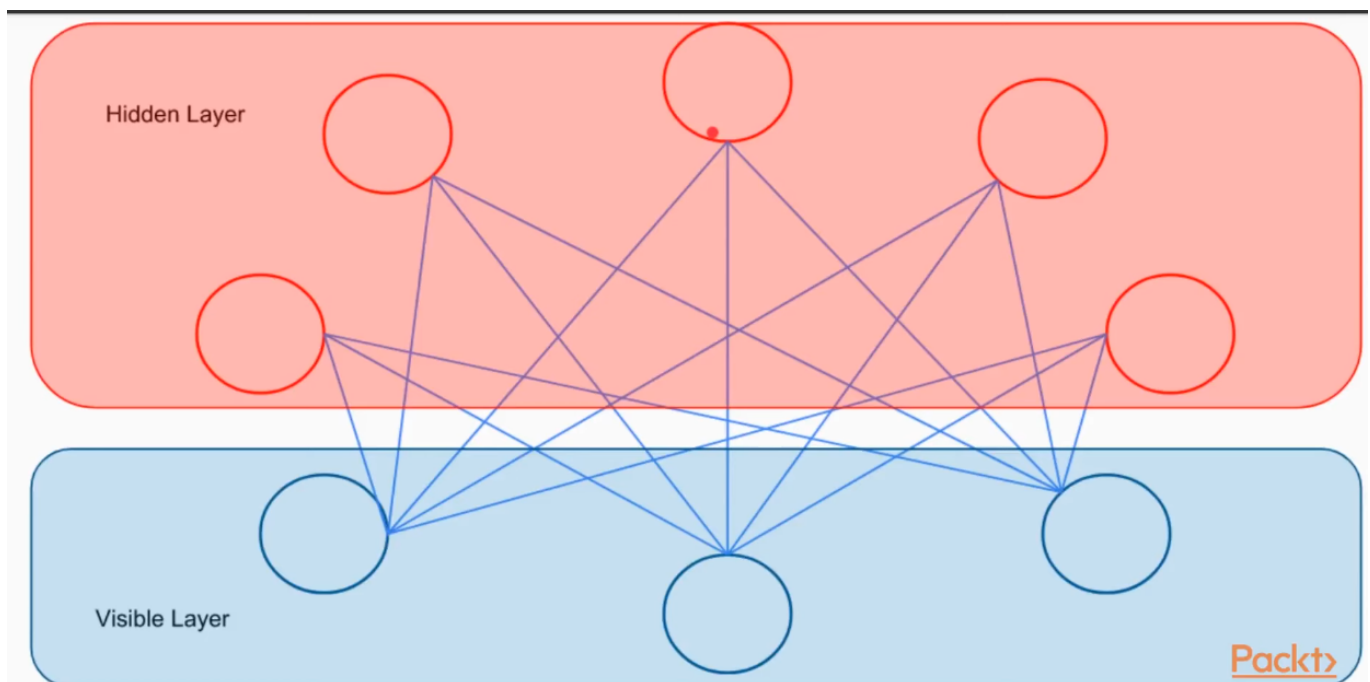
This ensures that weights strengthen meaningful **data-driven correlations** while removing unwanted **model-based correlations**.

Summary:

- **Data-centric samples** → Fix visible states, update hidden states.
- **Model-centric samples** → Update both visible & hidden states freely.
- **Weight updates** use the difference between the two to fine-tune model learning.

This process, while effective, is computationally expensive due to **Monte Carlo sampling**, leading to faster approximations in **Restricted Boltzmann Machines (RBMs)**.

RESTRICTED BOLTZMANN MACHINE



A **Restricted Boltzmann Machine (RBM)** is a simplified version of a Boltzmann Machine that is **bipartite**—connections are only allowed **between hidden and visible units**. Unlike general Boltzmann Machines, **hidden-to-hidden** and **visible-to-visible** connections are not allowed.

RBM Structure

- **Visible units:** $v_1, v_2, \dots, v_d$ (observed features)
- **Hidden units:** $h_1, h_2, \dots, h_m$ (latent representations)
- **Weights:** w_{ij} (between visible unit v_i and hidden unit h_j)
- **Biases:**
 - $b(v)_i$ for visible nodes
 - $b(h)_j$ for hidden nodes

Each visible unit **only connects to hidden units**, and vice versa. This structure simplifies inference, making learning faster and more efficient.

Hidden State Activation in RBMs

In an RBM, given the visible states, the probability of a hidden unit being **activated** ($h_j = 1$) follows: (eq 6.15)

$$P(h_j = 1 \mid v) = \frac{1}{1 + \exp \left(-b_j^{(h)} - \sum_{i=1}^d v_i w_{ij} \right)}$$

Key Observations:

- The **bias term** $b(h)_j$ shifts the activation probability.
- The **weights** w_{ij} determine how much each visible unit influences h_j .
- **No unknown hidden variables** appear in this equation, making it computationally simpler than unrestricted Boltzmann Machines.

Training the RBM

The weights of a Restricted Boltzmann Machine (RBM) are computed using a learning rule similar to that of Boltzmann Machines. A mini-batch-based algorithm is used for efficient training. The weights w_{ij} are initialized to small values and updated as follows:

Positive Phase

The algorithm uses a mini-batch of training instances, and computes the probability of the state of each hidden unit in exactly one step using Equation 6.15.

Then a single sample of the state of each hidden unit is generated from this probability. This process is repeated for each element in a mini-batch of training instances. The correlation between these different training instances of v_i and generated instances of h_j is computed; it is denoted by $\langle v_i, h_j \rangle^{pos}$. This correlation is essentially the average product between each such pair of visible and hidden units.

A mini-batch of training instances is used to compute the probability of the state of each hidden unit in one step:

$$P(h_j = 1|v) = \sigma \left(\sum_i w_{ij} v_i + b_j^{(h)} \right)$$

where $\sigma(x)$ is the sigmoid function. A single sample of each hidden unit's state is then generated from this probability. For each training instance, the correlation between v_i and h_j is computed:

$$\langle v_i, h_j \rangle^{pos} = \frac{1}{m} \sum_{k=1}^m v_i^{(k)} h_j^{(k)}$$

where m is the mini-batch size.

Negative Phase

In the negative phase, the algorithm starts with a mini-batch of training instances. Then, for each training instance, it goes through a phase of Gibbs sampling after starting with randomly initialized states. This is achieved by repeatedly using Equations 6.15 and 6.17 to compute the probabilities of the visible and hidden units, and using these probabilities to draw samples. The values of v_i and h_j at thermal equilibrium are used to compute $\langle v_i, h_j \rangle^{neg}$ in the same way as the positive phase.

Starting with a mini-batch of training instances, Gibbs sampling is performed by iteratively using:

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

and the hidden probability equation above. After reaching thermal equilibrium, the correlation is computed:

$$\langle v_i, h_j \rangle^{neg} = \frac{1}{m} \sum_{k=1}^m v_i^{(k)} h_j^{(k)}$$

Weight and Bias Updates

Using these correlations, the weights and biases are updated as:

$$w_{ij} \leftarrow w_{ij} + \alpha (\langle v_i, h_j \rangle^{pos} - \langle v_i, h_j \rangle^{neg})$$

$$b_i^{(v)} \leftarrow b_i^{(v)} + \alpha (\langle v_i, 1 \rangle^{pos} - \langle v_i, 1 \rangle^{neg})$$

$$b_j^{(h)} \leftarrow b_j^{(h)} + \alpha (\langle 1, h_j \rangle^{pos} - \langle 1, h_j \rangle^{neg})$$

where α is the learning rate, $\langle v_i, 1 \rangle$ represents the average value of v_i over the mini-batch, and $\langle 1, h_j \rangle$ represents the average value of h_j .

Using RBMs Beyond Binary Data Types

Restricted Boltzmann Machines (RBMs) are commonly used for binary data, but many real-world applications involve more complex data types such as categorical, ordinal, and continuous numerical values. By adapting RBMs, we can extend their usability to a wider range of tasks, including recommendation systems, natural language processing, and financial modeling.

Categorical and Ordinal Data

Categorical and ordinal data require a probability distribution that reflects their nature. This is often handled using softmax units. Softmax units modify the probability distribution of visible units to accommodate discrete classes or ordered categories. For real-valued ordinal data, discretization can be used, but it comes with a loss of precision.

Real-Valued Data and Gaussian RBMs

For real-valued data, Gaussian visible units are a more appropriate choice. Instead of binary activations, the visible and hidden units use continuous values, with hidden units typically employing ReLU activation functions. The energy function for a given configuration of visible and hidden units is defined as:

$$E(v, h) = \sum_i \frac{(v_i - b_i)^2}{2\sigma_i^2} - \sum_j b_j h_j - \sum_{i,j} \frac{v_i}{\sigma_i} h_j w_{ij}$$

The first term serves as a **containment function**, encouraging v_i to stay near b_i . This adaptation allows RBMs to model continuous data distributions effectively while maintaining the generative power of the original framework.

Challenges and Stability Considerations

While Gaussian RBMs improve flexibility, they introduce new challenges, particularly instability due to the variance parameter σ . The primary issues include:

- **Small updates to visible layers:** This can slow down learning and reduce model effectiveness.
- **Large updates to hidden layers:** These may lead to unstable training dynamics.

To mitigate these issues, the following strategies are commonly used:

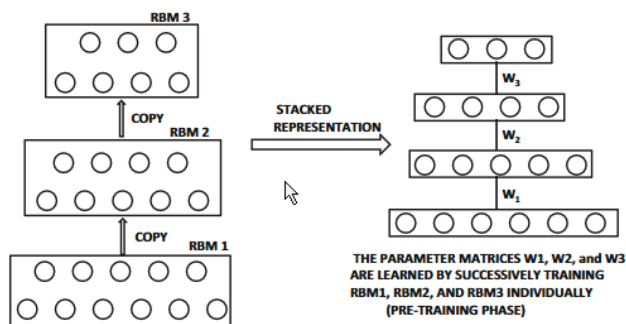
- Increasing the number of hidden units relative to visible units.
- Normalizing input data to have unit variance, allowing a fixed choice of $\sigma = 1$.
- Modifying ReLU activation by adding controlled Gaussian noise:

$$\text{noise} \sim \mathcal{N}(0, \log(1 + \exp(v)))$$

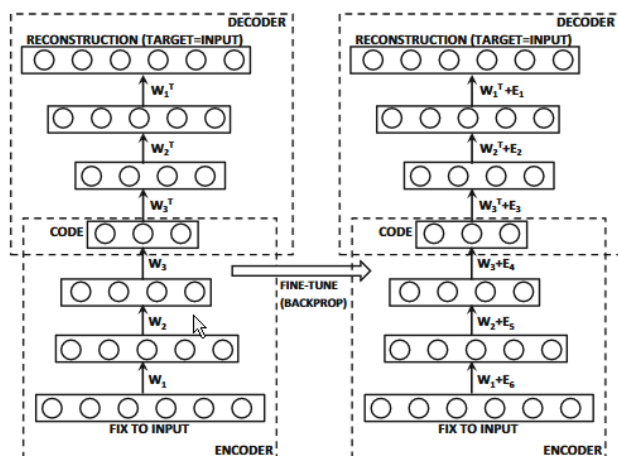
This modification allows ReLU units to approximate binomial units, enabling them to capture more information while preserving RBM functionality. Gibbs sampling and weight updates proceed similarly to binary RBMs, but careful tuning of learning rates is necessary to maintain stability and prevent divergence.

By incorporating these techniques, RBMs can be extended beyond binary data, making them applicable to a broader range of machine learning problems while maintaining their core generative and feature-learning capabilities.

Stacking Restricted Boltzmann Machines (RBMs)



(a) The stacked RBMs are trained sequentially in pretraining



(b) Pretraining is followed by fine-tuning with backpropagation

Figure 6.10: Training a multi-layer RBM

Why Stack RBMs?

The power of neural networks comes from their depth—multiple layers allow for richer representations and more complex function approximations while maintaining fewer parameters. The question arises: can we leverage RBMs to construct deep networks? The answer is yes. RBMs can be stacked to form deep architectures, a technique that predates deep neural networks and was instrumental in early deep learning models.

How Does It Work?

RBMs are trained using **Gibbs sampling**, which helps in learning a probability distribution over inputs. The trained weights from an RBM are then **grandfathered** into a neural network with continuous **sigmoid activations**, replacing the discrete sampling process. This transition is crucial because training RBMs differs fundamentally from training traditional feedforward neural networks. Specifically, RBMs use **contrastive divergence**, which **jointly trains all layers**, mitigating the **vanishing and exploding gradient problems** seen in conventional deep networks.

The Challenge of Creating Deep Networks with RBMs

RBMs are **not** standard feedforward layers; they operate as **undirected graphical models**, meaning visible and hidden units are symmetrically connected. To stack multiple RBMs, we need a method for linking these models while preserving learned representations. The key insight is that even though RBMs are **symmetric and discrete**, their learned **weights can be adapted to a directed neural network** that operates in a continuous activation space. Since these weights have already been trained via **discrete sampling**, they are close to an optimal solution and require only **modest fine-tuning** via backpropagation.

The Process of Stacking RBMs

1. **Train the first RBM:** Suppose our input data has d dimensions. We create an RBM with d visible units and m_1 hidden units.
2. **Generate hidden representations:** Once trained, the m_1 hidden unit activations now form a reduced representation of the data.
3. **Train the second RBM:** We use the m_1 outputs as inputs to a second RBM, which has m_1 visible units and m_2 hidden units.
4. **Repeat the process:** We continue stacking k RBMs, each using the hidden representations of the previous RBM as input, forming an architecture of size $m_{k-1} \times m_k$ at the final layer.

Structure of a Stacked RBM

- The left diagram in Figure 6.10(a) illustrates a multi-layered RBM architecture.
- The right diagram provides a more compact representation.
- The output layer of the r th RBM **exactly matches** the input layer of the $(r+1)$ th RBM, ensuring seamless data flow.

Transition to a Directed Model

One critical change occurs when we stack RBMs: the model ceases to be **undirected**. Unlike a standalone RBM, which maintains symmetric connections, stacking creates a **directed hierarchy** where information flows **one way—from lower layers to upper layers**. This shift transforms each RBM into a **computational unit**, with outputs of one unit serving as the inputs to the next.

Fine-Tuning with Backpropagation

Once we have stacked the RBMs, we can refine the entire model using **backpropagation**:

- If we replace binary sampling with **real-valued sigmoid activations**, we create a more conventional deep neural network.
- The use of real-valued activations is an **approximation**, but an effective one, since the RBMs were already trained on meaningful feature representations.
- **Backpropagation fine-tunes the model**, making it particularly useful in **supervised learning** tasks where labeled data is available.
- Since backpropagation can be performed on **any computational graph**, it is well-suited for refining the stacked RBM.

Summary

Stacking RBMs allows us to build **deep unsupervised models** that can later be fine-tuned with supervised learning. The advantage of RBMs over traditional deep networks lies in their training mechanism—contrastive divergence trains layers **jointly**, avoiding **gradient-related issues** that plague deep backpropagation-based networks. This method forms the foundation of **deep belief networks (DBNs)** and early deep learning architectures.

Greedy Layer-Wise Unsupervised Learning

What is it?

A training method where each layer in a deep neural network is trained independently, one at a time, in an **unsupervised** fashion before fine-tuning the whole network.

The term "greedy" refers to the fact that each layer is optimized on its own, without considering future layers yet.

Why is it needed?

Early deep learning models faced major challenges:

- **Vanishing gradients**: lower layers didn't learn effectively
- **Poor convergence**: due to random initialization
- **Difficulty training deep networks** from scratch

The idea:

Instead of training the whole deep network end-to-end initially, **pre-train each layer one by one in an unsupervised way**, then fine-tune the network.

How it works (step-by-step)

Given a deep neural network (e.g., with 3 layers):

Step 1: Train the first layer

- Input: raw data x
- Train the first layer as an **autoencoder** or **RBM**
- Only keep the **encoder** part

Step 2: Train the second layer

- Pass the encoded output from layer 1 as input
- Train the second layer on this representation

- Again, keep only the encoder

Step 3: Repeat for all layers

- Each new layer models the output of the previous one
- Stack encoders to build a deep network

Step 4: Fine-tune (optional but beneficial)

- Add a **supervised layer** (e.g., classifier) on top
- Fine-tune the whole network using **backpropagation**

Key Concepts

Term	Description
Unsupervised Pre-training	Each layer learns from inputs without labels
Greedy	Layers are trained one at a time
Layer-wise	Each layer is trained in isolation
Fine-tuning	Adjust all layers for a specific supervised task

Example with Autoencoders

Imagine your input is an image:

- **1st AE** learns to reconstruct the image → captures **edges**
- **2nd AE** reconstructs the output of 1st → captures **shapes**
- **3rd AE** reconstructs output of 2nd → captures **object parts**

Stack all encoders and add a final classifier. Then fine-tune.

Why it helps

- **Better initialization**: each layer starts from meaningful features
- **Faster convergence**: network learns faster during fine-tuning
- **Acts as regularization**: avoids overfitting on small datasets
- **Improves generalization**: builds a hierarchy of features:
 - low-level → mid-level → high-level

Transfer Learning & Domain Adaptation

Transfer Learning

Definition:

Transfer learning is a technique where a model trained on one task or domain is reused or adapted for a different, but related task or domain.

Rather than training a model from scratch, we **transfer knowledge** (features, weights, etc.) from a source task to improve learning in a target task.

Motivation

- Training deep models from scratch requires **lots of data** and **compute**.
- In many real-world tasks, **labeled data is scarce**.
- Features learned from one domain (e.g., edges in images) often generalize to other tasks.

Mathematical Setting

Let:

- $D_S = \{X_S, P_S(X)\}$ be the source domain
- T_S be the source task

- $D_T = \{X_T, P_T(X)\}$ be the target domain
- T_T be the target task

Transfer learning tries to improve $P_T(Y|X)$ using knowledge from D_S, T_S .

Types of Transfer Learning

1. Inductive Transfer Learning

- $T_S \neq T_T$
- Target domain **has labeled data**
- Example: Pretraining a model on ImageNet, then using it for medical image classification

Use Case: Few-shot learning, multitask learning

2. Transductive Transfer Learning (Domain Adaptation)

- $D_S \neq D_T$, but $T_S = T_T$
- The **task is the same**, but **domains differ**
- Useful when no or few labeled target samples exist

Examples:

- Sentiment classification in different topics
 - Object recognition across different lighting conditions
-

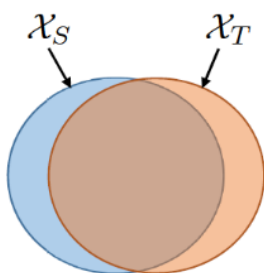
3. Unsupervised Transfer Learning

- No labels in either domain
 - Both tasks are unsupervised (e.g., clustering, representation learning)
-

Subcategories Based on Knowledge Transfer Strategy

Instance-based Transfer Learning

- Source and target domains have a lot of overlapping features or even share the same feature spaces
- Select or **reweight source instances** to match the target distribution
- Example: Using importance sampling where instance weights depend on similarity between $P_S(X)$ and $P_T(X)$

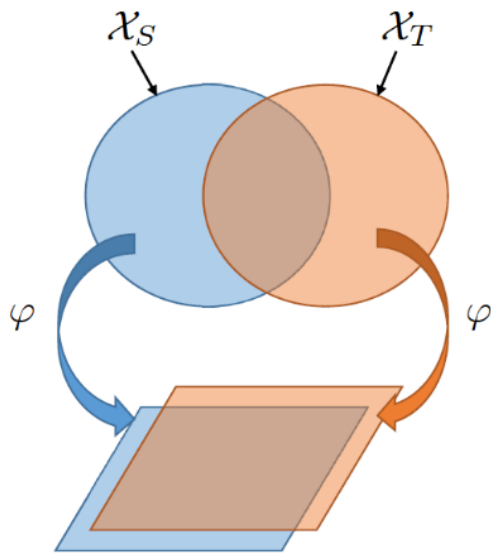


Parameter-based Transfer Learning

- Reuse **model parameters** (weights) trained on the source task from a well trained model and transfer to another if the tasks are related
- Often used in deep learning by **freezing or fine-tuning layers**

Feature-based Transfer Learning

- When source and target domains only have some overlapping features
- Learn a **common feature representation** for both domains
- Aim: Map source and target data to a common space where $P_S(Z) \approx P_T(Z)$



Techniques:

- Domain adversarial training (DANN)
- Maximum Mean Discrepancy (MMD)
- Correlation Alignment (CORAL)

Special Learning Paradigms Related to Transfer

One-Shot Learning

- Learn a new class from **only one** labeled example
- Often implemented with **metric learning** (e.g., Siamese Networks, Prototypical Networks)

Zero-Shot Learning

- Model predicts unseen classes using **semantic information** like class attributes or descriptions
- Involves **learning a mapping between visual and semantic spaces**

$$P(y \mid x) = P(y \mid a(y), x)$$

$P(y \mid x)$: This is the probability of class y given an **input x** (e.g., an image, a sentence).

$a(y)$: These are the **attributes** or **semantic description** of the class y .

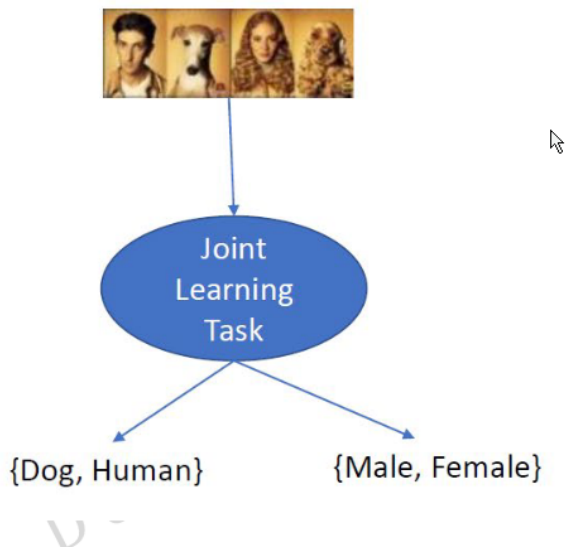
- If y is "zebra", $a(y)$ could be ["has stripes", "is a mammal", "herbivore"].

$P(y \mid a(y), x)$:

- This is the probability of class y , given both the input x and its **semantic description $a(y)$** .

Multitask Learning

- Jointly learn **multiple tasks** using a **shared representation**
- Helps regularization and learning **task-invariant features**



Benefits of Transfer Learning

Benefit	Explanation
Reduced Training Time	Leverages pretrained models, requiring less data and time for new tasks.
Better Performance with Limited Data	Especially useful when labeled data in the target domain is scarce.
Improved Generalization	Models can learn more robust and general features from source domains.
Resource Efficiency	Saves computational resources by reusing models trained on large datasets.
Faster Convergence	Pretrained weights offer a good starting point, leading to quicker optimization.
Knowledge Reusability	Useful knowledge from related tasks can be reused, avoiding starting from scratch.
Useful for Complex Tasks	Helps with complex tasks where data collection is expensive (e.g., medical imaging, NLP).

Applications of Transfer Learning

Domain	Application
Computer Vision	Pretrained models like VGG, ResNet for image classification, object detection, medical imaging.
NLP	BERT, GPT for sentiment analysis, chatbots, machine translation.
Speech Recognition	Adapting models to different accents/languages using limited voice samples.
Robotics	Transferring knowledge from simulation to real-world tasks (Sim2Real).
Healthcare	Diagnosis from radiographs, MRIs using general datasets (e.g., COVID-19 detection).
Autonomous Vehicles	Object recognition and scene understanding adapted to specific environments.
Finance	Fraud detection or stock prediction using similar domain-trained models.

Multimodal Learning

- Learn from multiple **input modalities** (e.g., image + text)
- Encourages richer representations and cross-modal alignment

Domain Adaptation

Definition:

A special case of transfer learning where **the task remains the same** ($T_S = T_T$), but **the domain changes** ($D_S \neq D_T$).

Challenges

- Different data distributions:
 - $P_S(X) \neq P_T(X)$
 - Sometimes even $P_S(Y|X) \neq P_T(Y|X)$
- Features learned on the source domain might not transfer well

Approaches

1. **Feature Alignment:**
 - Learn a shared feature space where both domains look similar.
 - Use methods like Maximum Mean Discrepancy (MMD) or adversarial training.
2. **Instance Re-weighting:**
 - Give more importance to source examples that are similar to the target domain.
3. **Domain-Adversarial Training:**
 - Learn features that confuse a domain classifier, forcing domain-invariant representations.

Example

- Train on photos of objects (source)
- Test on sketches or edge maps (target)
- Both are object classification, but domains differ

Summary Table

Concept	Description
Transfer Learning	Reusing knowledge from a source task/domain
Domain Adaptation	Same task, different domains
Inductive	Source and target tasks differ
Transductive	Same task, different input distributions (domains)
Feature Transfer	Use pretrained features (e.g., CNN layers from ImageNet)
Fine-tuning	Adapt source model weights to fit target data

Benefits of Domain Adaptation

Benefit	Explanation
Adapts to New Data Distributions	Enables learning when the data distribution changes between domains.
Reduces Domain Shift	Techniques like adversarial training or MMD align source and target distributions.
Requires No or Few Target Labels	Works even when target domain has few or no labels.
Boosts Cross-Domain Accuracy	Enhances performance when applying models trained on one domain to another.
Enables Real-World Applications	Essential where retraining on a new domain is impractical (e.g., robotics).
Supports Continuous Learning	Facilitates learning across evolving environments and data streams.

Applications of Domain Adaptation

Domain	Application
Spam Detection	Adapting spam filters from one platform/language to another (e.g., Gmail to Outlook).
Cross-Lingual NLP	Transferring models from high-resource to low-resource languages for translation/analysis.
E-commerce	Adapting recommender systems across regions or demographics.
Agriculture	Plant disease detection models adapted across regions or crop types.
Manufacturing	Fault detection systems adapted across different machines or factories.
Surveillance	Person recognition across various camera angles or lighting conditions.
Smart Assistants (IoT)	Adapting speech/text understanding to individual users over time.