
Hedgehog Rock

# University of Science and Technology Bannu

**Deep Learning**

**Lesson 11**

**October,7 2024**

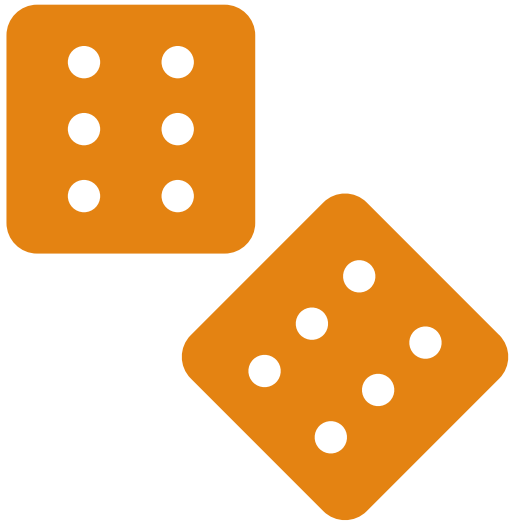
# Learning Objectives

- Need for a RBMS
- **Why RBMS**
- **RBMS Architecture**

# *Boltzmann* Machine

*Boltzmann* Machine was first invented in 1985 by Geoffrey Hinton, a professor at the University of Toronto. He is a leading figure in the deep learning community and is referred to by some as the "Godfather of Deep Learning".

# What is *Boltzmann* Machine*?*

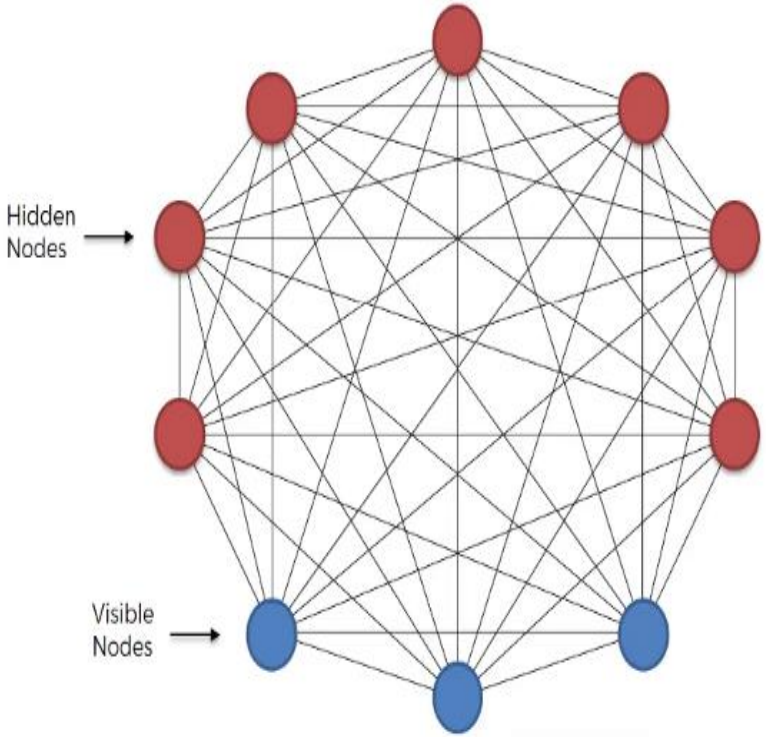


*Boltzmann* Machine *is a* generative unsupervised model, which involves learning a probability distribution from an original dataset and using it to make inferences about never-before-seen data.

In this machine, there are two layers named visible layer or input layer and hidden layer.

The visible layer is denoted as **v** and the hidden layer is denoted as the **h.** In Boltzmann machine, there is no output layer. Boltzmann machines are random and generative neural networks capable of learning internal representations and are able to represent and (given enough time) solve tough combinatoric problems.

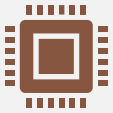
# Structure




- *Boltzmann* Machine has an input layer (also referred to as the *visible layer*) and one or several hidden layers (also referred to as the *hidden layer*).
- Boltzmann Machine uses neural networks with neurons that are connected not only to other neurons in other layers but also to neurons within the same layer.
- Everything is connected to everything. Connections are bidirectional, *visible* neurons connected to each other and *hidden* neurons also connected to each other
- *Boltzmann Machine doesn't expect input data, it generates data.* Neurons *generate information regardless they are hidden or visible.*
- *For Boltzmann Machine all n*eurons are the same, it doesn't discriminate between *hidden* and *visible n*eurons. *For Boltzmann Machine whole things are system and its generating* state *of the system.*

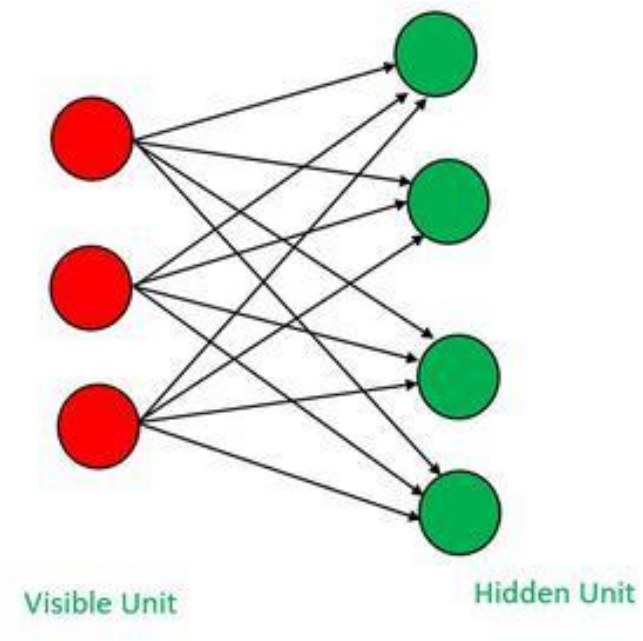
# What are Restricted Boltzmann Machines (RBM)?

The RBM algorithm was proposed by Geoffrey Hinton (2007), which learns probability distribution over its sample training data inputs.



It has seen wide applications in different areas of supervised/unsupervised machine learning such as feature learning, dimensionality reduction, classification, collaborative filtering, and topic modeling.

# What are Restricted Boltzmann Machines (RBM)?




A restricted term refers to that we are not allowed to connect the same type layer to each other. In other words, the two neurons of the input layer or hidden layer can't connect to each other.

Although the hidden layer and visible layer can be connected to each other.

The learning process is typically done using a variant of gradient descent called **contrastive divergence**, which updates the weights by minimizing the difference between the input data and the model distribution.

# Summary

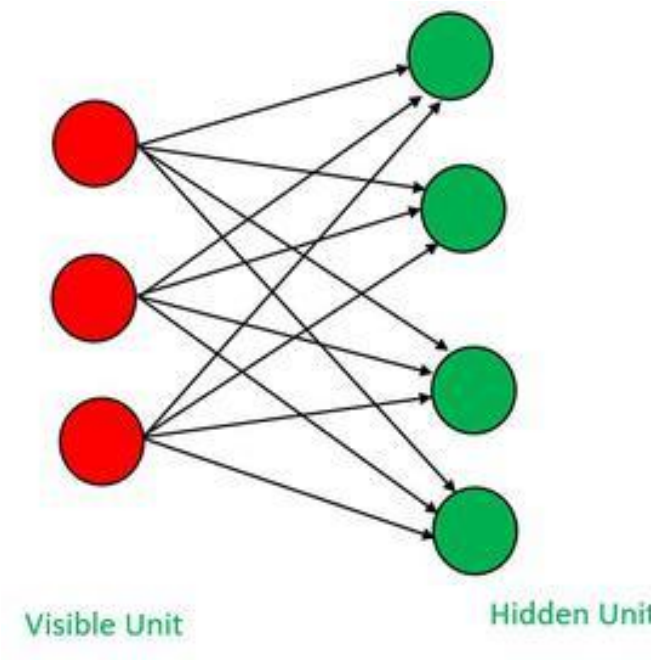
It belongs to a class of models known as **energy-based models**, and its purpose is to learn the distribution of input data. RBMs are composed of two layers with each node being a binary unit that can take on the value of **0 or 1**:

**Visible layer (v)**: Represents the observed data or input.

**Hidden layer (h)**: Captures latent or hidden features from the input data.

The key feature of an RBM is that there are **no connections within a layer**, only between the visible and hidden layers. This is called **restricted** because of this lack of intra-layer connections.

# Architecture of RBM

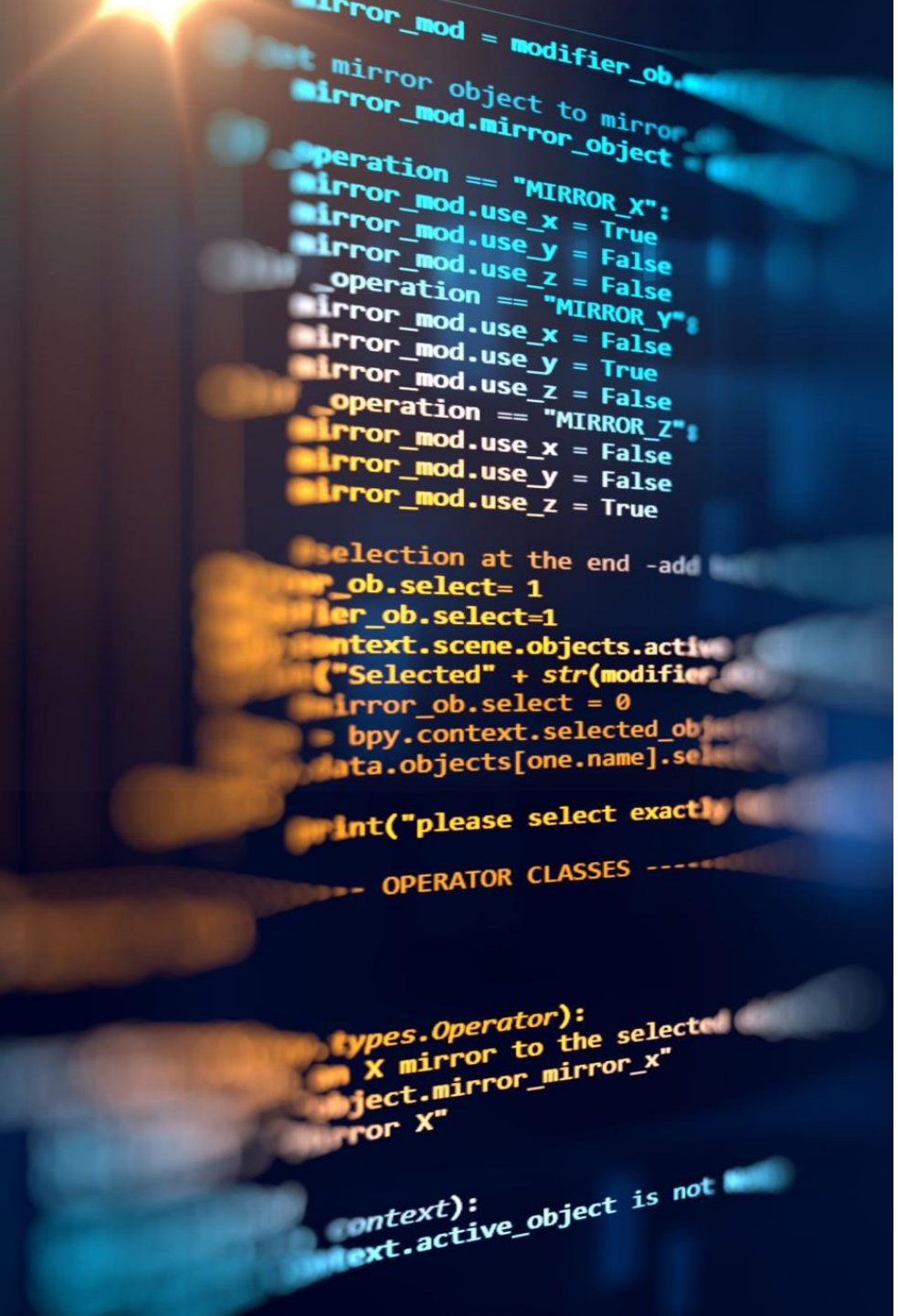



The architecture consists of two main parts:

- **Visible units (neurons)**: These correspond to the input features. For instance, if you are working with images, each visible unit could correspond to a pixel value.
- **Hidden units (neurons)**: These detect patterns or features from the input data. The hidden units try to capture higher-level representations.

There are no output neurons in the traditional sense (like in a classifier), but RBMs learn a probabilistic model to generate data.

**Key Point**: While there are no separate input/output neurons, the visible units act as "input" when providing data to the model and also as "output" when reconstructing or generating data.




# How RBMs Take Input

- The visible layer is where the input data goes. Each visible unit is assigned the value from the corresponding input feature. For example, if you are using binary inputs, the visible units take values of either 0 or 1.
- For non-binary data, such as real-valued data (like images with pixel values), RBMs can be adjusted (with some modifications, like Gaussian RBMs).

# The Role of Weights

**Weights (W)** connect the visible units to the hidden units. These weights are learned during training and help capture the relationships between the input data (visible layer) and the latent features (hidden layer).

Each visible unit is connected to every hidden unit through a weight. The strength of these weights determines how much influence a visible unit has on a hidden unit.

# How do Restricted Boltzmann Machines work?

In RBM there are two phases through which the entire RBM works:

**1: Forward Pass (Visible to Hidden)**

When an input is provided to the visible units, the following steps occur:

1. Each hidden unit computes the weighted sum of the inputs it is connected to.
2. The hidden unit applies a **sigmoid activation function** to this sum to compute a probability that it should be "activated" (turned on).
3. Based on this probability, the hidden unit is either turned on (1) or off (0).

This step is where the RBM extracts features from the input data, effectively compressing or encoding it into a latent representation in the hidden units.

# How do Restricted Boltzmann Machines work?

**2nd Phase: Backward Pass (Hidden to Visible)**

As we don't have any output layer. Instead of calculating the output layer, we are reconstructing the input layer through the activated hidden state. This process is said to be Feed Backward Pass. We are just backtracking the input layer through the activated hidden neurons. Once the hidden units have been activated, RBMs can reconstruct the input through the following process:

1. Each visible unit now computes the weighted sum of the hidden units it is connected to.
2. Similar to the hidden units, a sigmoid function is applied to calculate the probability of activation for each visible unit.
3. Based on these probabilities, the RBM either turns each visible unit on or off, generating a reconstruction of the original input.

# How RBMs Generate Outputs

- The output of the RBM is typically the reconstructed visible layer. After the forward pass (encoding), the backward pass (decoding) generates a reconstruction of the input data.
- The quality of this reconstruction is used to guide the training process. If the reconstruction is close to the original input, the RBM has successfully captured the structure of the data.

# Energy Function and Probability

$$E(v,h) = -\sum_i \sum_j v_i W_{ij} h_j - \sum_i v_i b_i - \sum_j h_j c_j$$

An RBM is governed by an **energy function** that defines the probability of a particular configuration (pair of visible and hidden states). The lower the energy, the more likely that configuration is to occur. The energy function for an RBM is defined as:

Where:

- $v_i$ is the state of visible unit $i$,
- $h_j$ is the state of hidden unit $j$,
- $W_{ij}$ 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 $j$.

The goal of training is to minimize this energy by adjusting the weights $W_{ij}$, and biases $b_i$ and $c_j$.

# Training RBMs

Training an RBM involves adjusting the weights and biases to maximize the likelihood that the model will reconstruct input data accurately. The common approach used is **Contrastive Divergence (CD)**:

1. **Positive phase**: Use the input data to activate the hidden units.
2. **Negative phase**: Reconstruct the visible layer using the hidden units, then re-compute the hidden activations from this reconstruction.
3. The weights are updated based on the difference between the input-visible/hidden activations and the reconstructed-visible/hidden activations.

# Applications of Restricted Boltzmann Machines (RBM)?

RBMs are often used as building blocks for more complex deep learning models.

hey have also been used as building blocks for more complex deep learning models, such as deep belief networks and deep autoencoders.

# Applications of Restricted Boltzmann Machine

- Restricted Boltzmann Machines (RBMs) have found numerous applications in various fields, some of which are:
- **Collaborative filtering:** RBMs are widely used in collaborative filtering for recommender systems. They learn to predict user preferences based on their past behavior and recommend items that are likely to be of interest to the user.
- **Image and video processing:** RBMs can be used for image and video processing tasks such as object recognition, image denoising, and image reconstruction. They can also be used for tasks such as video segmentation and tracking.
- **Natural language processing:** RBMs can be used for natural language processing tasks such as language modeling, text classification, and sentiment analysis. They can also be used for tasks such as speech recognition and speech synthesis.

- **Bioinformatics:** RBMs have found applications in bioinformatics for tasks such as protein structure prediction, gene expression analysis, and drug discovery.
- **Financial modeling:** RBMs can be used for financial modeling tasks such as predicting stock prices, risk analysis, and portfolio optimization.
- **Anomaly detection:** RBMs can be used for anomaly detection tasks such as fraud detection in financial transactions, network intrusion detection, and medical diagnosis.

# To Summarize the Input/Output Process:

- **Input**: The visible layer directly takes input data.
- **Output**: The visible layer is also the output, but after going through hidden representations and reconstructing the original input in the form of an output (which is the reconstruction).

This input/output relationship is different from traditional neural networks because RBMs do not predict labels but instead learn the underlying structure of the data.

# Code: Loading and Visualizing MNIST Data

In [1]:

```
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
import numpy as np
import matplotlib.pyplot as plt

def plot_example(X, y):
    """Plot the first 5 images and their labels in a row."""
    for i, (img, y) in enumerate(zip(X[:5].reshape(5, 28, 28), y[:5])):
        plt.subplot(151 + i)
        plt.imshow(img, cmap='gray')
        plt.xticks([])
        plt.yticks([])
        plt.title(y)

X, Y = fetch_openml("mnist_784", version=1, return_X_y=True, as_frame=False)
print(X.shape)
plot_example(X, Y)
```

```
(70000, 784)
```

# Preparing the Data

To prepare the MNIST data for our RBM, we need to normalize and binarize the pixel values and then split the dataset. First, we normalize the pixel values from the range [0, 255] to [0, 1], and then binarize them.

Next, we split the data into training and test sets. We reserve 10,000 samples for testing and use the rest for training:

In [2]:
```
# normalize data
X /= 255

# binarize data
X = np.where(X > 0.5, 1, 0)

# train - test split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=10000, random_state=42, stratify=Y)
```

```python
from sklearn.utils import gen_batches

class RBM():

  def __init__(self, visible_dim, hidden_dim):
    self.visible_dim = visible_dim
    self.hidden_dim = hidden_dim
    # weights
    self.W = np.random.randn(visible_dim, hidden_dim)
    # biases
    self.bh = np.random.randn(hidden_dim)
    self.bv = np.random.randn(visible_dim)


  def fit(self, X, epochs=10, batch_dim=16, lr=0.01):
    '''
    Trains the restricted boltzmann machine
    '''
    for epoch in range(epochs):
      # saving the image of the weights
      self.plot_weights(step=epoch, save=True)

      error_epoch = 0
      batches = list(gen_batches(X.shape[0], batch_dim))

      for batch in batches:
        batch = X[batch.start:batch.stop]
        batch_size = batch.shape[0]
```

# Implementing the RBM

We'll now implement the RBM from scratch using NumPy. Our RBM class includes methods for training the model using Contrastive Divergence, reconstructing data, and visualizing the learned weights.

# Training the RBM

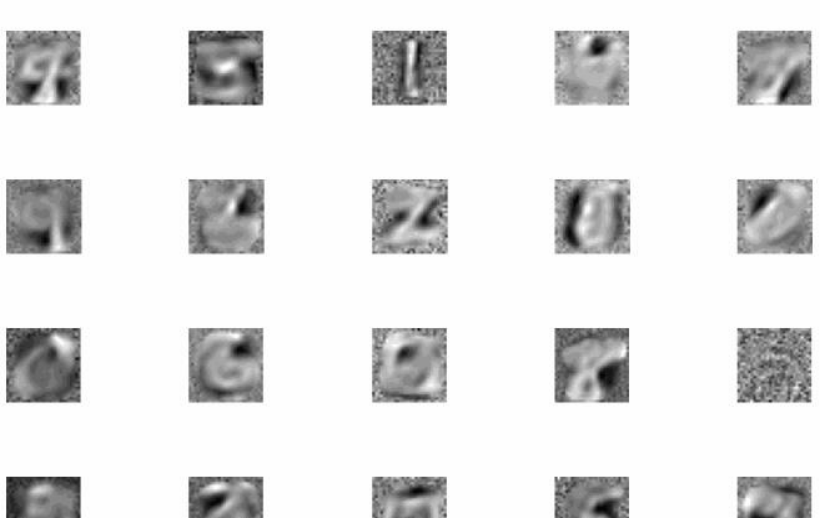
```
# Turn off interactive mode of matplotlib to avoid showing multiple plots
plt.ioff() # Initialize RBM with 30 hidden units
rbm = RBM(visible_dim=X_train.shape[1], hidden_dim=30)
# Train the RBM
rbm.fit(X_train, epochs=20, batch_dim=128, lr=0.7)
# Turn interactive mode back on
plt.ion()
```

With our RBM implemented, it’s time to train it using the prepared MNIST dataset. We’ll configure the model and then run the training process.

We initialize the RBM with a specified number of hidden units. In this case, we use 30 hidden units, which means the RBM will learn to capture 30 different features from the input data.

```python
# show the weights over epochs
import glob
from PIL import Image
import re
import math
from pathlib import Path

file_pattern = re.compile(r'.*?(\d+).*?')
def get_order(file):
    match = file_pattern.match(Path(file).name)
    if not match:
        return math.inf
    return int(match.groups()[0])
```



# Visualizing the Weights During Training

To understand how the RBM learns and what features it captures, we can visualize the weights at different epochs during training. This helps us see the evolution of learned features over time.

**Generating Weight Visualizations**

During training, we save images of the weights at each epoch. These images show how the weights corresponding to each hidden unit evolve. We then compile these images into a GIF for an animated view of the training progress.

Here's the code to create a GIF from the weight images:

# Testing and Visualizing Reconstruction

To evaluate the performance of our trained RBM, we will test it on a subset of the MNIST dataset and visualize how well it can reconstruct the input images. This allows us to assess how effectively the RBM has learned to capture and reproduce the features of handwritten digits.

```
# get one image of the test set for each digit
# Create a list of indices for each digit
digit_indices = [np.where(Y_test == str(i))[0][0] for i in range(10)]
resticted_set = X_test[digit_indices]

reconstructed = rbm.reconstruct(resticted_set)

# show 10 sample images
rows = 2
columns = 10

fig, axes = plt.subplots(rows, columns,sharey = True,figsize=(30, 6))
for i in range(rows):
    for j in range(columns):
```