**NNFS CA 1**

**Class: D16AD**

**Aim:**

Implement the assigned presentation topic in the lab. Make a proper document with output screenshots and related theory and conclusion. Upload the same in the classroom.

**Problem Statement:**Implement Boltzmann Machine

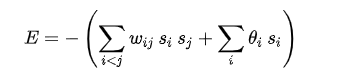
**Theory:**

A Boltzmann machine (also called Sherrington–Kirkpatrick model with external field or stochastic Ising model), named after Ludwig Boltzmann is a stochastic spin-glass model with an external field, i.e., a Sherrington–Kirkpatrick model, that is a stochastic Ising model. It is a statistical physics technique applied in the context of cognitive science. It is also classified as a Markov random field.

They are named after the Boltzmann distribution in statistical mechanics, which is used in their sampling function. They were heavily popularized and promoted by Geoffrey Hinton, Terry Sejnowski and Yann LeCun in cognitive sciences communities, particularly in machine learning, as part of "energy-based models" (EBM), because Hamiltonians of spin glasses as energy are used as a starting point to define the learning task.

**Architecture:**

A Boltzmann machine, like a Sherrington–Kirkpatrick model, is a network of units with a total "energy" (Hamiltonian) defined for the overall network. Its units produce binary results. Boltzmann machine weights are stochastic. The global energy E in a Boltzmann machine is identical in form to that of Hopfield networks and Ising models:



**Training:**

As we know that Boltzmann machines have fixed weights, hence there will be no training algorithm as we do not need to update the weights in the network. However, to test the network we have to set the weights as well as to find the consensus function CF.

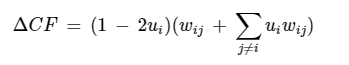
The Boltzmann machine has a set of units Ui and Uj and has bi-directional connections on them.

* We are considering the fixed weight, say wij.
* wij ≠ 0 if Ui and Uj are connected.
* There also exists a symmetry in weighted interconnection, i.e. wij = wji.
* wii also exists, i.e. there would be the self-connection between units.
* For any unit Ui, its state ui would be either 1 or 0.

The main objective of Boltzmann Machine is to maximize the Consensus Function CF which can be given by the following relation

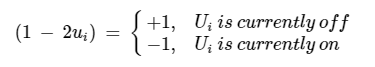


Now, when the state changes from either 1 to 0 or from 0 to 1, then the change in consensus can be given by the following relation −



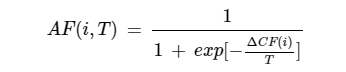
Here ui is the current state of Ui.

The variation in coefficient (1 - 2ui) is given by the following relation −



Generally, unit Ui does not change its state, but if it does then the information would be residing local to the unit. With that change, there would also be an increase in the consensus of the network.

Probability of the network to accept the change in the state of the unit is given by the following relation −

**

Here, T is the controlling parameter. It will decrease as CF reaches the maximum value.

**Code:**

| import torch  import torch.nn.functional as F  class BoltzmannMachine:  def \_\_init\_\_(self, num\_visible, num\_hidden, learning\_rate=0.01):  self.num\_visible = num\_visible  self.num\_hidden = num\_hidden  self.learning\_rate = learning\_rate    # Initialize weights and biases  self.weights = torch.randn(num\_visible, num\_hidden) \* 0.01  self.visible\_bias = torch.zeros(num\_visible)  self.hidden\_bias = torch.zeros(num\_hidden)  def sample\_hidden(self, v):  """ Sample hidden units given visible units. """  hidden\_probs = torch.sigmoid(F.linear(v, self.weights.t(), self.hidden\_bias))  return hidden\_probs, torch.bernoulli(hidden\_probs)  def sample\_visible(self, h):  """ Sample visible units given hidden units. """  visible\_probs = torch.sigmoid(F.linear(h, self.weights, self.visible\_bias))  return visible\_probs, torch.bernoulli(visible\_probs)  def contrastive\_divergence(self, v0, k=1):  """ Perform k-step contrastive divergence. """  # Step 1: Sample hidden states from visible states  h0\_probs, h0 = self.sample\_hidden(v0)  # Step 2: Sample visible states from hidden states  v1\_probs, v1 = self.sample\_visible(h0)  # Step 3: Sample hidden states from new visible states  hk\_probs, hk = h0\_probs, h0 # Initial hidden probabilities and states  for \_ in range(k):  hk\_probs, hk = self.sample\_hidden(v1)  # Update weights and biases  self.weights += self.learning\_rate \* (torch.outer(v0, h0\_probs) - torch.outer(v1, hk\_probs))  self.visible\_bias += self.learning\_rate \* (v0 - v1)  self.hidden\_bias += self.learning\_rate \* (h0\_probs - hk\_probs)  def fit(self, data, epochs=1000, k=1):  """ Train the Boltzmann Machine on the dataset. """  for epoch in range(epochs):  for v0 in data:  self.contrastive\_divergence(v0)  def generate\_samples(self, num\_samples=10):  """ Generate samples from the model. """  samples = []  v = torch.bernoulli(torch.zeros(self.num\_visible)) # Initial visible state  for \_ in range(num\_samples):  h\_probs, h = self.sample\_hidden(v)  v\_probs, v = self.sample\_visible(h)  samples.append(v)  return torch.stack(samples)  # Sample data: 5 binary vectors of length 4  data = torch.FloatTensor([[1, 0, 1, 0],  [0, 1, 0, 1],  [1, 1, 0, 0],  [0, 0, 1, 1],  [1, 0, 1, 1]])  # Instantiate the Boltzmann Machine  bm = BoltzmannMachine(num\_visible=4, num\_hidden=2, learning\_rate=0.1)  # Train the model  bm.fit(data, epochs=1000, k=1)  # Generate samples  samples = bm.generate\_samples(num\_samples=5)  print("Generated Samples:")  print(samples) |
| --- |

**Output:**

| Generated Samples:  tensor([[0., 1., 0., 1.],  [0., 1., 0., 1.],  [0., 1., 0., 0.],  [0., 1., 0., 1.],  [1., 1., 0., 1.]]) |
| --- |