Perceptron Learning Algorithm

CT-466 | Week 2 - Lecture 5

Instructor: Mehar Fatima Shaikh

Learning Rules

- **Error-Correction Learning**
- **Competitive Learning**
- **Memory-Based Learning**
- **Hebbian Learning**
- **Boltzmann Learning**

Error-Correction Learning Rule

- The Error-Correction Learning Rule, also known as the Perceptron Learning Rule, is a fundamental supervised learning technique in neural networks. It is based on adjusting the weights of the model whenever there is a mismatch between the predicted output and the actual target output.
- The weight update rule is defined as:
- $W(new) = W(old) + \eta (y - \hat{y}) x$

  Where:
  - W(new): Updated weight vector
  - W(old): Previous weight vector
  - η (eta): Learning rate (a small positive constant)
  - y: Actual target output
  - ŷ: Predicted output
  - x: Input vector

Example

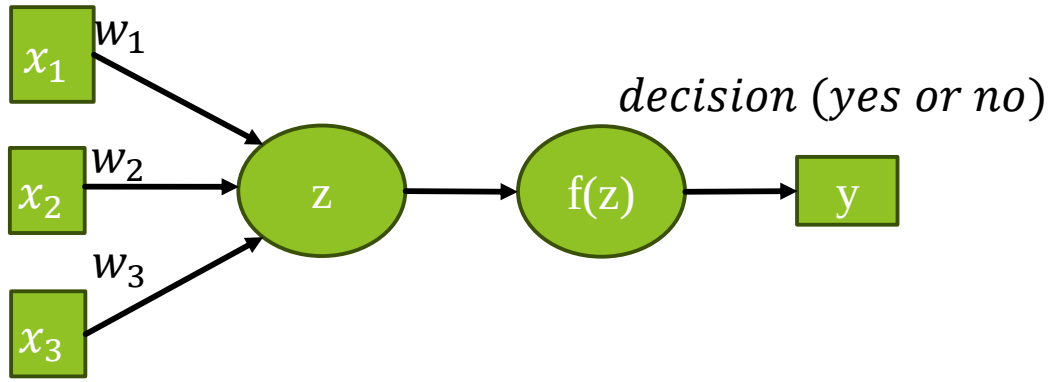
- Consider the following training examples:
- **Input #3**
- x = [2.5, 2.1, 1], y = 1
  f(net) = f(-3.01*2.5 - 2.06*2.1 - 1.0*1) = f(-12.851) = -1
  Mismatch ⇒ apply learning rule

  W^3 = [-3.01, -2.06, -1.0] + 0.4*[2.5, 2.1, 1]
  = [-2.01, -1.22, -0.6]
- **Input #4**
- x = [8.0, 7.7, 1], y = -1
  f(net) = f(-2.01*8.0 - 1.22*7.7 - 0.6)
  = f(-26.074) = -1

  Predicted = actual ⇒ No update
  $W^4 = W^3$ = [-2.01, -1.22, -0.6]
- **Conclusion**
- The Error-Correction Learning Rule ensures that weights are adjusted only when the predicted output differs from the actual target. This rule guarantees convergence for linearly separable data.

Practice Problem

- **Initial Conditions**
- Initial Weights:
- w1 = 0.75
- w2 = 0.5
- w3 = -0.6
- Bias (b) = 0
- Learning Rate (α) = 0.2



Input Dataset

x1	x2	x3	y
1.0	1.0	1	1
9.4	6.4	1	-1
2.5	2.1	1	1
8.0	7.7	1	-1
0.5	2.2	1	1
7.9	8.4	1	-1
7.0	7.0	1	-1
2.8	0.8	1	1
1.2	8.0	1	1
7.8	6.1	1	-1

Solution

- **Input #1**
- x = [1.0, 1.0, 1], y = 1
- f(net) = f(0.75*1.0 + 0.5*1.0 - 0.6*1) = f(0.65) = 1
- Since predicted = actual, no weight update
- $W^2 = W^1$
- **Input #2**
- x = [9.4, 6.4, 1], y = -1
- f(net) = f(0.75*9.4 + 0.5*6.4 - 0.6*1) = f(9.65) = 1
- Mismatch: predicted ≠ actual ⇒ apply learning rule
- $W^2 = [0.75, 0.5, -0.6] - 0.4*[9.4, 6.4, 1] = [-3.01, -2.06, -1.0]$

Solution

- **Input #3**
- x = [2.5, 2.1, 1], y = 1
- f(net) = f(-3.01*2.5 -2.06*2.1 -1.0*1) = f(-12.851) = -1
- Mismatch ⇒ apply learning rule
- W^3 = [-3.01, -2.06, -1.0] + 0.4*[2.5, 2.1, 1] = [-2.01, -1.22, -0.6]
- **Input #4**
- x = [8.0, 7.7, 1], y = -1
- f(net) = f(-2.01*8.0 -1.22*7.7 -0.6) = f(-26.074) = -1
- Predicted = actual ⇒ No update
- $W^4 = W^3$

Solution

- **Input #5**
- x = [0.5, 2.2, 1], y = 1
- f(net) = f(-2.01*0.5 -1.22*2.2 -0.6) = f(-4.289) = -1
- Mismatch ⇒ apply learning rule
- W^5 = [-2.01, -1.22, -0.6] + 0.4*[0.5, 2.2, 1] = [-1.81, -0.34, -0.2]
- **Input #6**
- x = [7.9, 8.4, 1], y = -1
- f(net) = f(-1.81*7.9 -0.34*8.4 -0.2) = f(-17.355) = -1
- Predicted = actual ⇒ No update
- $W^6 = W^5$

Solution

- **Input #7**
- x = [7.0, 7.0, 1], y = -1
- f(net) = f(-1.81*7.0 -0.34*7.0 -0.2) = f(-15.25) = -1
- Predicted = actual ⇒ No update
- $W^7 = W^6$
- **Input #8**
- x = [2.8, 0.8, 1], y = 1
- f(net) = f(-1.81*2.8 -0.34*0.8 -0.2) = f(-5.54) = -1
- Mismatch ⇒ apply learning rule
- W^8 = [-1.81, -0.34, -0.2] + 0.4*[2.8, 0.8, 1] = [-0.69, -0.02, 0.2]

Solution

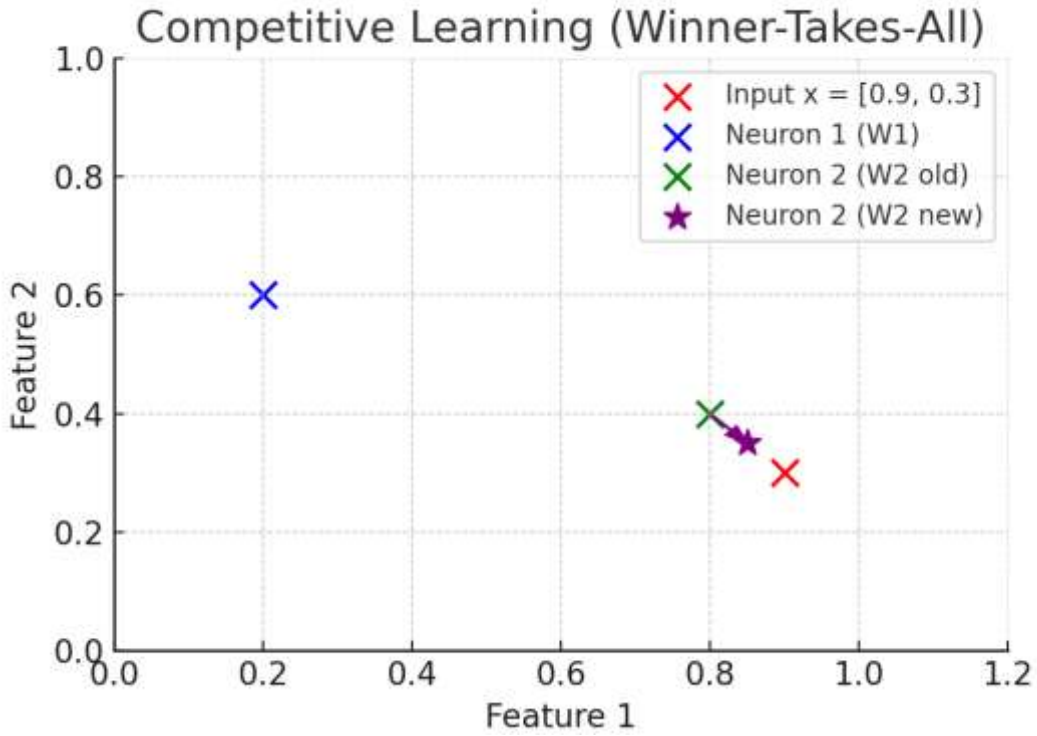
- **Input #9**
- x = [1.2, 8.0, 1], y = 1
- f(net) = f(-0.69*1.2 -0.02*8.0 + 0.2) = f(-0.788) = -1
- Mismatch ⇒ apply learning rule
- W^9 = [-0.69, -0.02, 0.2] + 0.4*[1.2, 8.0, 1] = [-0.21, 3.18, 0.6]
- **Input #10**
- x = [7.8, 6.1, 1], y = -1
- f(net) = f(-0.21*7.8 +3.18*6.1 +0.6) = f(18.36) = 1
- Mismatch ⇒ apply learning rule
- W^{10} = [-0.21, 3.18, 0.6] - 0.4*[7.8, 6.1, 1] = [-3.33, 0.74, 0.2]

Competitive Learning Rule

- Competitive Learning is an unsupervised learning rule used in neural networks, particularly in clustering and self-organizing maps. In this rule, neurons compete with each other to become active (the 'winner'). Only the winning neuron updates its weights, while all other neurons remain unchanged. This process helps in creating specialized neurons that respond to different input patterns.
- The weight update rule is defined as:
- $W(new) = W(old) + \eta (x - W(old))$ if neuron wins
  $W(new) = W(old)$ otherwise

  Where:
  - W(new): Updated weight vector of the winning neuron
  - W(old): Previous weight vector
  - η (eta): Learning rate ($0 < \eta < 1$)
  - x: Input vector
  - Winning neuron: The neuron whose weight vector is closest to the input vector (using a distance measure such as Euclidean distance)

Competitive Learning Rule



Example

- Suppose we have two neurons with initial weights:
  W1 = [0.2, 0.6], W2 = [0.8, 0.4]

  Input vector: x = [0.9, 0.3]

- **Step 1: Distance Calculation**

- Compute Euclidean distance between input vector and weight vectors:
  d1 = ||x - W1|| = sqrt((0.9 - 0.2)^2 + (0.3 - 0.6)^2)
  = sqrt(0.49 + 0.09) = sqrt(0.58) ≈ 0.76

  d2 = ||x - W2|| = sqrt((0.9 - 0.8)^2 + (0.3 - 0.4)^2)
  = sqrt(0.01 + 0.01) = sqrt(0.02) ≈ 0.14

  Since d2 < d1, neuron 2 wins.

- **Step 2: Update Winning Neuron**

- Update W2 using learning rate η = 0.5:

  W2(new) = W2(old) + η(x - W2(old))
  = [0.8, 0.4] + 0.5([0.9, 0.3] - [0.8, 0.4])
  = [0.8, 0.4] + 0.5([0.1, -0.1])
  = [0.8, 0.4] + [0.05, -0.05]
  = [0.85, 0.35]

  Thus, the updated weights are:
  W1 = [0.2, 0.6] (unchanged)
  W2 = [0.85, 0.35] (updated)

- **Conclusion**

- In Competitive Learning, only the winning neuron updates its weights. Over time, different neurons specialize in recognizing different input patterns, which makes this rule particularly useful in clustering and unsupervised feature learning

Comparison of *Error-Correction vs Competitive Learning*

- In **Competitive Learning**:
- It is an **unsupervised algorithm** → there is **no target output (y)** given.
- Neurons **compete** based on their **net input** (distance or similarity to the input vector).
- The **winner neuron** (closest to the input) updates its weights **towards the input vector**.
- Other neurons do nothing.
- That's why we **don't compare predicted output vs. actual output** (like in Error-Correction Learning).
  Because there is **no label** — the network is just **self-organizing** by clustering inputs.

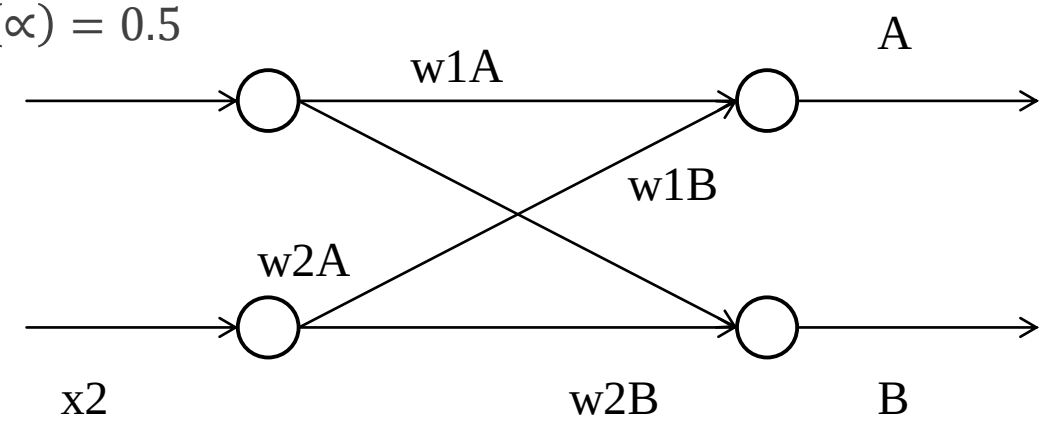
Sample problem

- Sure, here's the provided data in table form:

x1	x2	y
1.0	1.0	1
9.4	6.4	-1
2.5	2.1	1
8.0	7.7	-1
0.5	2.2	1
7.9	8.4	-1
7.0	7.0	-1
2.8	0.8	1
1.2	8.0	1
7.8	6.1	-1

Competitive Learning

- Initial weight
  - W1=(7,2)
  - W2=(2,9)
- $learning\ rate(\propto) = 0.5$



Solution:

- Iteration 1: Data Point [1.0, 1.0, 1]
- **Calculate Distances :**

  Distance to W1=(7,2):

  Distance=$(1-7)^2+(1-2)^2=36+1=37$

  Distance to W2=(2,9):

  Distance=$(1-2)^2+(1-9)^2=1+64=65$
- **Determine Closest Weight**:
  - W1 is closer to (1.0,1.0).
- **Update Weights**:
  - W1(t+1)=(7,2)+0.5·((1,1)−(7,2)) =(7,2)+0.5·(−6,−1)=(7,2)+(−3,−0.5)=(4,1.5)

Solution : continue

- Iteration 2: Data Point [9.4, 6.4, -1]
- Calculate Distances:
- Distance to W1=(4,1.5):
- Distance = (9.4-4)^2 + (6.4-1.5)^2 = 27.04 + 22.09 = 49.13
- Distance to W2=(2,9):
- Distance = (9.4-2)^2 + (6.4-9)^2 = 56.16 + 8.64 = 64.8
- Determine Closest Weight:
- W1 is closer to (9.4, 6.4, -1).
- Update Weights:
- W1(t+1) = (4,1.5) + 0.5 * ((9.4,6.4) - (4,1.5))
- = (4,1.5) + 0.5 * (5.4, 4.9)
- = (4,1.5) + (2.7, 2.45)
- = (6.7, 3.95)

Solution : continue

- Iteration 3: Data Point [2.5, 2.1, 1]
- Calculate Distances:
- Distance to W1=(6.7, 3.95):
- Distance = (2.5-6.7)^2 + (2.1-3.95)^2 = 17.64 + 3.1025 = 20.7425
- - Distance to W2=(2, 9):
- Distance = (2.5-2)^2 + (2.1-9)^2 = 0.25 + 49= 49.25
- Determine Closest Weight:
- W1 is closer to (2.5, 2.1, 1).
- Update Weights:
- W1(t+1) = (6.7, 3.95) + 0.5 * ((2.5, 2.1) - (6.7, 3.95))
- = (6.7, 3.95) + 0.5 * (-4.2, -1.85,)
- = (6.7, 3.95) + (-2.1, -0.925)
- = (4.6, 3.025)

Solution : continue

- Iteration 4: Data Point [8.0, 7.7, -1]
- Calculate Distances:
- Distance to W1=(4.6, 3.025):
- Distance = (8.0-4.6)^2 + (7.7-3.025)^2 = 11.56 + 21.025 = 32.585
- Distance to W2=(2, 9):
- Distance = (8.0-2)^2 + (7.7-9)^2 = 36 + 1.69 = 37.69
- Determine Closest Weight:
- W1 is closer to (8.0, 7.7, -1).
- Update Weights:
- W1(t+1) = (4.6, 3.025) + 0.5 * ((8.0, 7.7) - (4.6, 3.025))
- = (4.6, 3.025) + 0.5 * (3.4, 4.675)
- = (4.6, 3.025) + (1.7, 2.3375)
- = (6.3, 5.3625)

Solution : continue

- Iteration 5: Data Point [0.5, 2.2]
- Calculate Distances:
- Distance to W1=(6.3, 5.3625):
- Distance = (0.5-6.3)^2 + (2.2-5.3625)^2 = 36.64 + 10.91 = 47.55
- Distance to W2=(2, 9):
- Distance = (0.5-2)^2 + (2.2-9)^2 = 2.25 + 47.04= 49.29
- Determine Closest Weight:
- W1 is closer to (0.5, 2.2).
- Update Weights:
- W1(t+1) = (6.3, 5.3625) + 0.5 * ((0.5, 2.2) - (6.3, 5.3625))
- = (6.3, 5.3625) + 0.5 * (-5.8, -3.1625)
- = (6.3, 5.3625) + (-2.9, -1.58125)
- = (3.4, 3.78125)

Solution : continue

- Iteration 6: Data Point [7.9, 8.4]
- Calculate Distances:
- Distance to W1=(3.4, 3.78125):
- Distance = (7.9-3.4)^2 + (8.4-3.78125)^2= 19.36 + 20.446 = 39.806
- Distance to W2=(2, 9):
- Distance = (7.9-2)^2 + (8.4-9)^2 = 31.36 + 0.16 = 31.52
- Determine Closest Weight:
- W2 is closer to (7.9, 8.4).
- Update Weights:
- W2(t+1) = (2, 9) + 0.5 * ((7.9, 8.4) - (2, 9))
- = (2, 9) + 0.5 * (5.9, -0.6)
- = (2, 9) + (2.95, -0.3)
- = (4.95, 8.7)

Solution : continue

- Iteration 7: Data Point [7.0, 7.0]
- Calculate Distances:
- Distance to W1=(3.4, 3.78125):
- Distance = (7.0-3.4)^2 + (7.0-3.78125)^2 = 12.96 + 10.24 = 23.2
- Distance to W2=(4.95, 8.7):
- Distance = (7.0-4.95)^2 + (7.0-8.7)^2 = 4.2025 + 6.76 = 10.9625
- Determine Closest Weight:
- W2 is closer to (7.0, 7.0).
- Update Weights:
- W2(t+1) = (4.95, 8.7) + 0.5 * ((7.0, 7.0) - (4.95, 8.7))
- = (4.95, 8.7) + 0.5 * (2.05, -1.7)
- = (4.95, 8.7) + (1.025, -0.85)
- = (5.975, 7.85)

Solution : continue

- Iteration 8: Data Point [2.8, 0.8]
- Calculate Distances:
- Distance to W1=(3.4, 3.78125):
- Distance = (2.8-3.4)^2 + (0.8-3.78125)^2 = 0.36 + 9.136 = 9.496
- Distance to W2=(5.975, 7.85):
- Distance = (2.8-5.975)^2 + (0.8-7.85)^2 = 10.0225 + 53.1025 = 63.125
- Determine Closest Weight:
- W1 is closer to (2.8, 0.8).
- Update Weights:
- W1(t+1) = (3.4, 3.78125) + 0.5 * ((2.8, 0.8) - (3.4, 3.78125))
- = (3.4, 3.78125) + 0.5 * (-0.6, -2.98125)
- = (3.4, 3.78125) + (-0.3, -1.490625)
- = (3.1, 2.290625)

Solution : continue

- Iteration 8: Data Point [2.8, 0.8]
- Calculate Distances:
- Distance to W1=(3.4, 3.78125):
- Distance = (2.8-3.4)^2 + (0.8-3.78125)^2 = 0.36 + 9.136 = 9.496
- Distance to W2=(5.975, 7.85):
- Distance = (2.8-5.975)^2 + (0.8-7.85)^2 = 10.0225 + 53.1025 = 63.125
- Determine Closest Weight:
- W1 is closer to (2.8, 0.8).
- Update Weights:
- W1(t+1) = (3.4, 3.78125) + 0.5 * ((2.8, 0.8) - (3.4, 3.78125))
- = (3.4, 3.78125) + 0.5 * (-0.6, -2.98125)
- = (3.4, 3.78125) + (-0.3, -1.490625)
- = (3.1, 2.290625)

Solution : continue

- Iteration 9: Data Point [1.2, 3.0]
- Calculate Distances:
- Distance to W1=(3.1, 2.290625):
- Distance = (1.2-3.1)^2 + (3.0-2.290625)^2 = 3.61 + 0.504414 = 4.114414
- Distance to W2=(5.975, 7.85):
- Distance = (1.2-5.975)^2 + (3.0-7.85)^2 = 22.657025 + 18.9225 = 41.579525
- Determine Closest Weight:
- W1 is closer to (1.2, 3.0).
- Update Weights:
- W1(t+1) = (3.1, 2.290625) + 0.5 * ((1.2, 3.0) - (3.1, 2.290625))
- = (3.1, 2.290625) + 0.5 * (-1.9, 0.709375)
- = (3.1, 2.290625) + (-0.95, 0.3546875)
- = (2.15, 2.6453125)

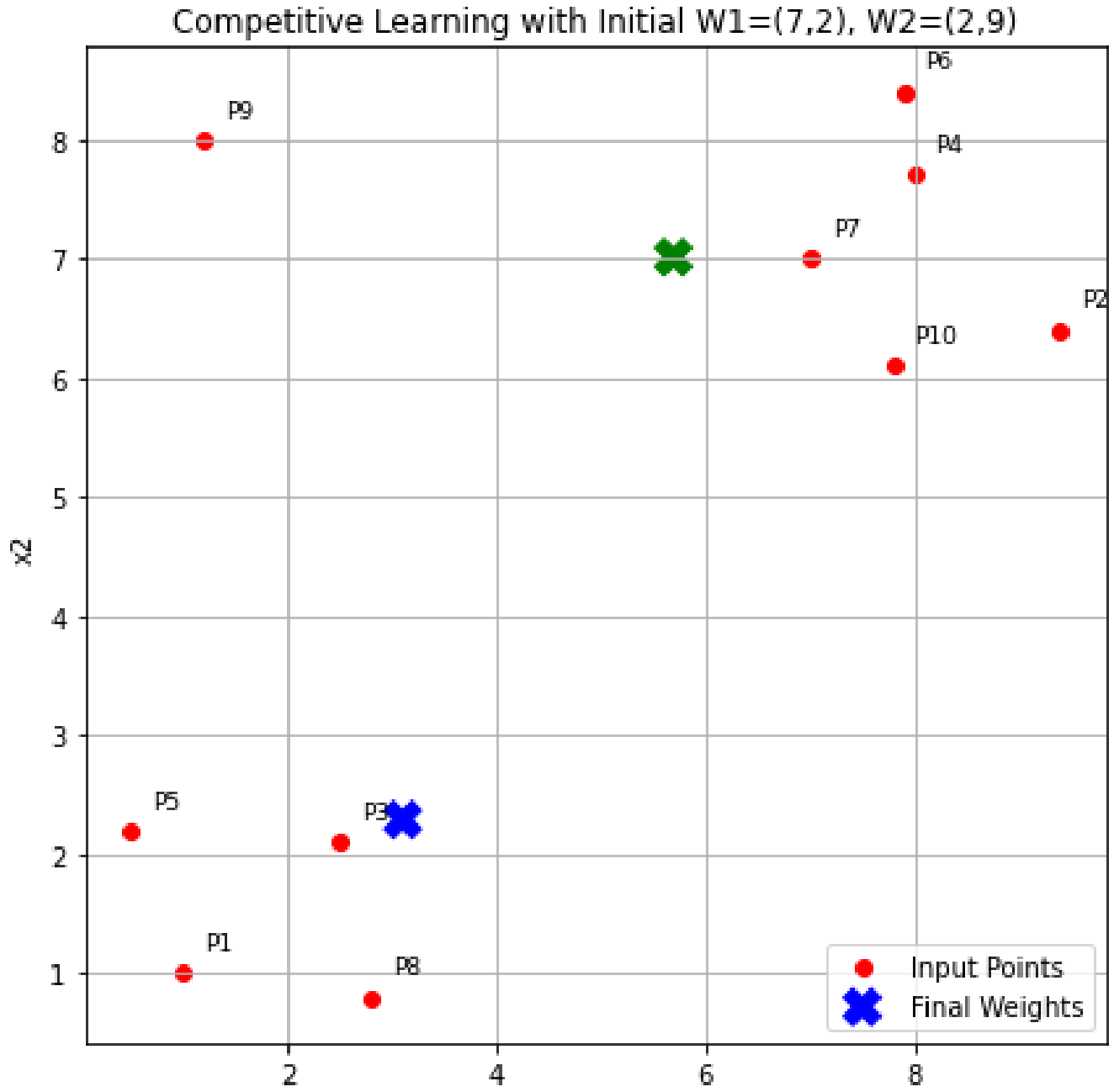
Solution : continue

- Iteration 10: Data Point [7.8, 6.1]
- Calculate Distances:
- Distance to W1=(2.15, 2.6453125):
- Distance = (7.8-2.15)^2 + (6.1-2.6453125)^2 = 30.7225 + 11.517601465 = 42.240101465
- Distance to W2=(5.975, 7.85):
- Distance = (7.8-5.975)^2 + (6.1-7.85)^2 = 3.31225 + 3.0225 = 6.33475
- Determine Closest Weight:
- W2 is closer to (7.8, 6.1).
- Update Weights:
- W2(t+1) = (5.975, 7.85) + 0.5 * ((7.8, 6.1) - (5.975, 7.85))
- = (5.975, 7.85) + 0.5 * (1.825, -1.75)
- = (5.975, 7.85) + (0.9125, -0.875)
- = (6.8875, 6.975)

Python implementation

```
import matplotlib.pyplot as plt
import numpy as np


# Dataset (x1, x2)
inputs = np.array([
    [1.0, 1.0],
    [9.4, 6.4],
    [2.5, 2.1],
    [8.0, 7.7],
    [0.5, 2.2],
    [7.9, 8.4],
    [7.0, 7.0],
    [2.8, 0.8],
    [1.2, 8.0],
    [7.8, 6.1]
])


# Plot input points
plt.figure(figsize=(7, 7))
plt.scatter(inputs[:, 0], inputs[:, 1], c="red", s=80, marker="o")

# Add labels P1, P2, ..., P10
for i, (x, y) in enumerate(inputs):
    plt.text(x + 0.1, y + 0.1, f"P{i+1}", fontsize=9)


plt.title("Input Points with Labels")
plt.xlabel("x1")
plt.ylabel("x2")
plt.grid(True)
plt.show()
```

Results


Competitive Learning with Initial W1=(7,2), W2=(2,9)
P6
P9
P4
P7
P2
P10
P5
P3
P1
P8
x2
1
2
3
4
5
6
7
8
2
4
6
8
Input Points
Final Weights

Outcome

- The prototype move forward toward cluster center.
- Concept: It operates by selecting the neuron with the highest activation as the winner, which then updates its weights to better fit the input pattern, potentially inhibiting other neurons in the process.
- The Winner Takes All algorithm works by choosing the neuron that responds the strongest to input, adjusting its weights to better match the input pattern. This process can sometimes suppress or limit the response of other neurons.

Applications of Learning Rules

- **1. Error-Correction Learning**
- **Where used:** Supervised learning (Perceptron, ADALINE, Backpropagation in Neural Networks).
- **Applications:**
  - Pattern recognition (digits, characters, speech signals).
  - Spam email classification.
  - Medical diagnosis (classifying diseases from patient data).
  - Image classification tasks.

Applications of Learning Rules

- **2. Competitive Learning**
- **Where used:** Unsupervised clustering (Winner-Takes-All strategy).
- **Applications:**
  - Clustering customer data in marketing.
  - Vector quantization in speech and image compression.
  - Self-Organizing Maps (SOMs) for data visualization.
  - Robotics (environment mapping and sensor clustering).

Applications of Learning Rules

- **3. Memory-Based Learning**
- **Where used:** Instance-based learning methods (e.g., k-Nearest Neighbors).
- **Applications:**
  - Recommender systems (suggesting products based on similarity).
  - Handwritten digit recognition (MNIST dataset using kNN).
  - Case-based reasoning in decision-making (law, healthcare).
  - Information retrieval and search engines.

Applications of Learning Rules

- **4. Hebbian Learning**
- **Where used:** Unsupervised learning, models biological learning ("neurons that fire together, wire together").
- **Applications:**
  - Feature extraction (PCA-like learning).
  - Associative memory networks (e.g., Hopfield Networks).
  - Data compression and dimensionality reduction.
  - Modeling brain functions in neuroscience.

Applications of Learning Rules

- **5. Boltzmann Learning**
- **Where used:** Stochastic learning using energy-based models (Boltzmann Machines, Restricted Boltzmann Machines).
- **Applications:**
  - Deep learning (as a foundation for Deep Belief Networks).
  - Feature learning and dimensionality reduction.
  - Collaborative filtering in recommender systems (e.g., Netflix movie recommendations).
  - Optimization problems where randomness helps escape local minima.