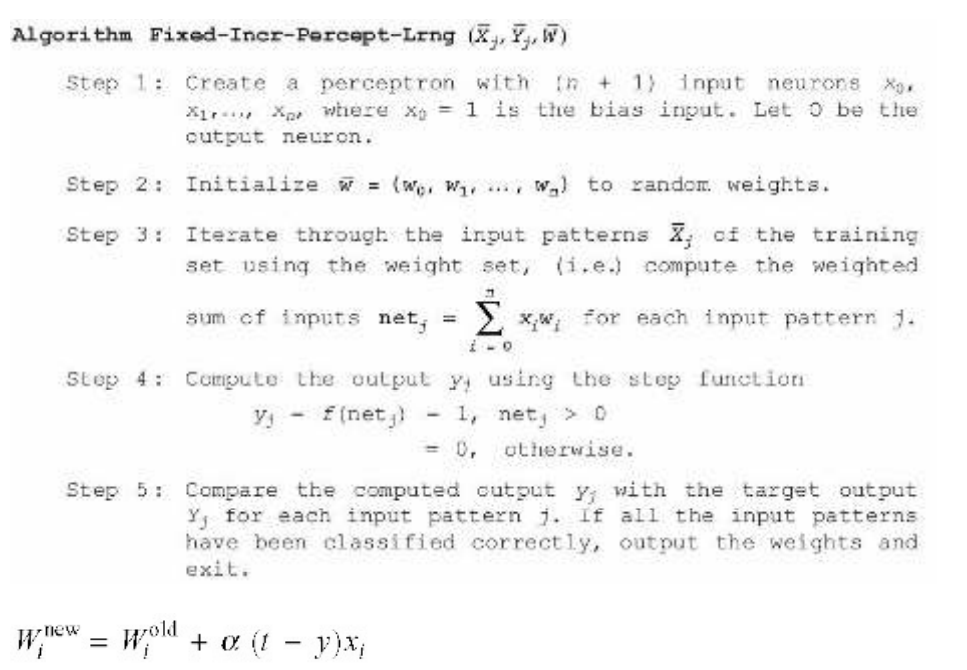
**Date: 10th January 2025**

**Experiment 4**

**AIM:** Implement the Fixed Increment Perceptron Learning algorithm for 2 class classification problem as presented in the attached images.



A white paper with black numbers and a black text

AI-generated content may be incorrect.

**Introduction:**

A **perceptron** is the simplest type of artificial neural network model, primarily used for binary classification tasks.

The **Fixed Increment Perceptron Learning Algorithm** is a supervised learning algorithm used for binary classification in neural networks. It is an early and fundamental algorithm for training **single-layer perceptrons**.

A perceptron consists of:

1. Input Layer – Takes numerical inputs (features).
2. Weights – Each input has an associated weight that determines its importance.
3. Summation Function – Computes the weighted sum of inputs:
4. Σ (weight × input) + bias
5. Activation Function – Applies a step function (or another function) to decide the output (0 or 1).
6. Output – A binary classification (e.g., yes/no, spam/not spam).

**How Perceptron Algorithm works:**

1. **Initialize weights** (randomly or as zeros).
2. **For each training example**:
   * Compute weighted sum.
   * Apply activation function.
   * Update weights using the perceptron learning rule:   
     **New Weight = Old Weight + Learning Rate × (Expected Output - Predicted Output) × Input**
3. Repeat until the model learns or a stopping criterion is met.

**Code:**

|  |
| --- |
| ***Perceptron*** |
| import numpy as np  class Neuron:      def \_\_init\_\_(self, num\_weights):          np.random.seed(4)  **# Small random values**          self.weights = np.random.randn(num\_weights) \* 0.01      def getOutput(self, inputs):          newInputs = np.array([1] + inputs)  **# Add bias term**          return 1 if np.dot(self.weights, newInputs) > 0 else 0      def train(self, inputs, outputs, alpha=0.1, max\_iterations=1000):          for \_ in range(max\_iterations):              solved = True              for ipt, opt in zip(inputs, outputs):                  result = self.getOutput(ipt)                  if result != opt:                      res = opt - result                      self.weights += alpha \* res \* np.array([1] + ipt)                      solved = False              if solved:                  break  **# Stop if converged**  **# Example: AND gate (works)**  and\_inputs = [[1, 0], [0, 1], [1, 1], [0, 0]]  and\_outputs = [0, 0, 1, 0]  neuron = Neuron(3)  neuron.train(and\_inputs, and\_outputs)  print("AND Gate Results:")  for ipt in and\_inputs:      print(f"Input: {ipt}, Output: {neuron.getOutput(ipt)}")  **# Example: XOR gate (fails with perceptron)**  xor\_inputs = [[1, 0], [0, 1], [1, 1], [0, 0]]  xor\_outputs = [1, 1, 0, 0]  neuron = Neuron(3)  neuron.train(xor\_inputs, xor\_outputs)  print("\nXOR Gate Results (Expected to Fail):")  for ipt in xor\_inputs:  **# Incorrect results expected**      print(f"Input: {ipt}, Output: {neuron.getOutput(ipt)}") |
| **Output:** |

**Limitations**

1. **Only works for linearly separable data** (fails for XOR-type problems).
2. **No convergence for non-separable data**, leading to infinite updates.
3. **Fixed learning rate** may slow down or destabilize training.
4. **Single-layer limitation**—cannot handle complex patterns.
5. **Sensitive to initial weights**, affecting convergence speed.
6. **No probability output**, unlike logistic regression.

**Applications**

1. **Binary classification** (spam detection, sentiment analysis).
2. **Pattern recognition** (basic speech, image recognition).
3. **Edge detection in images**.
4. **Medical diagnosis** (linear symptom-based disease classification).
5. **Signal processing** (simple filtering and detection).
6. **Stock trend analysis** (basic models with limited features).

**Conclusion:**

The perceptron successfully learns linearly separable functions like **AND** but fails for non-linearly separable ones like **XOR**, as it can only model linear decision boundaries. Its performance depends on a fixed learning rate, which may cause slow convergence or oscillations. While useful for simple binary classification, single-layer perceptrons are limited in handling complex patterns. To solve problems like **XOR**, a **Multi-Layer Perceptron (MLP)** with activation functions such as **ReLU or sigmoid** is required.