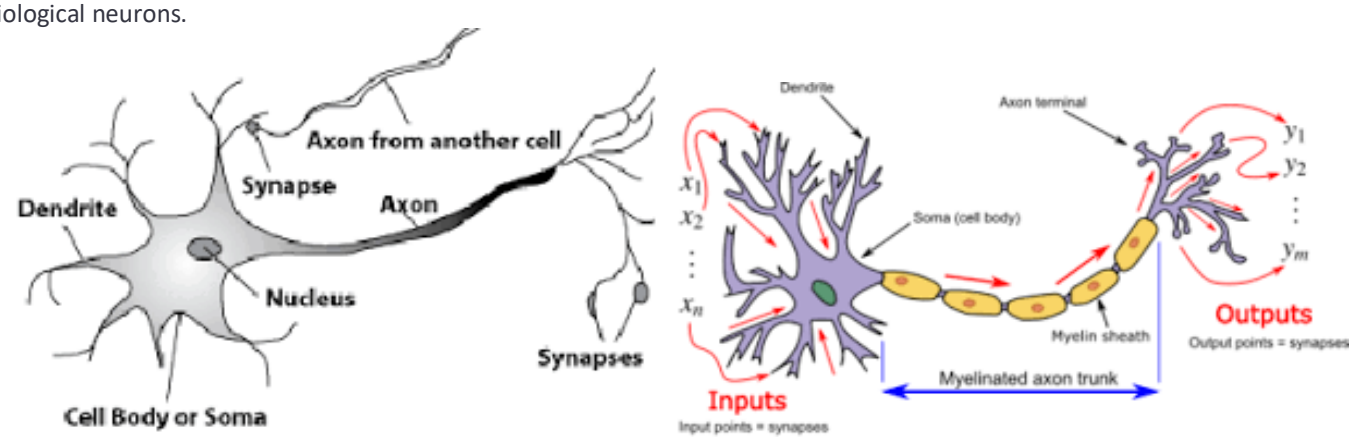
Unit-2: Introduction to Artificial Neural Network

Biological Model of Neuron

The biological model of a neuron is a representation of how actual neurons function in the human nervous system. These neurons are the basic building blocks of the nervous system and play a crucial role in transmitting electrical and chemical signals throughout the body. The artificial neural networks used in machine learning are inspired by the structure and behavior of biological neurons.



Here's a simplified overview of the key components of a biological neuron:

1. **Cell Body (Soma):** The cell body contains the nucleus and most of the organelles responsible for the neuron's metabolic functions. It integrates incoming signals from dendrites and decides whether to transmit the signal further.
2. **Dendrites:** Dendrites are branched extensions that receive signals from other neurons or sensory receptors. They serve as the input channels for the neuron. Dendrites receive neurotransmitters released by other neurons across synapses.
3. **Axon:** The axon is a long, slender projection that conducts electrical impulses away from the cell body and towards other neurons, muscles, or glands. The end of the axon forms axon terminals, where neurotransmitters are released to communicate with other neurons.
4. **Myelin Sheath:** In many neurons, the axon is covered by a fatty substance called the myelin sheath, which acts as an insulating layer. Myelination increases the speed at which electrical signals travel along the axon.
5. **Synapses:** Synapses are specialized junctions between neurons. They allow communication between neurons through the transmission of neurotransmitters. Neurotransmitters are released from axon terminals and received by receptors on the dendrites of the next neuron.
6. **Action Potential:** An action potential is a rapid change in the neuron's membrane potential. It is initiated when a stimulus causes the membrane potential to reach a threshold level. This triggers an electrical impulse that travels along the axon to the axon terminals.
7. **Resting Membrane Potential:** Neurons have a resting membrane potential, which is the difference in electrical charge across the cell membrane when the neuron is at rest. This potential is maintained by the active transport of ions across the cell membrane.
8. **Neurotransmitters:** Neurotransmitters are chemical messengers that transmit signals between neurons across synapses. They bind to receptors on the postsynaptic neuron's dendrites, influencing whether an action potential will be generated.
9. **Receptors:** Receptors are proteins located on the postsynaptic neuron's membrane. They bind to specific neurotransmitters, triggering changes in the membrane potential and influencing whether an action potential will be initiated.

The biological model of a neuron forms the basis for understanding how information is processed and transmitted in the nervous system. Artificial neural networks are inspired by these biological neurons and attempt to simulate their behavior through mathematical models, activation functions, and interconnected layers.

ANN model

An artificial neural network (ANN) model is a mathematical model inspired by the biological neural networks that constitute animal brains. ANNs are used to solve a wide range of problems, including classification, regression, and forecasting.

An ANN model is typically composed of three layers:

- Input layer: This layer receives the input data.
- Hidden layer(s): The hidden layer(s) perform the actual computation of the ANN model. The number of hidden layers can vary depending on the complexity of the problem being solved.
- Output layer: This layer produces the output of the ANN model.

Each neuron in an ANN model has a number of inputs and outputs. The inputs are multiplied by a set of weights and then summed together. This sum is then passed through an activation function to produce the output of the neuron. The activation function is a non-linear function that helps to introduce non-linearity into the ANN model. This non-linearity is important for allowing the ANN model to learn complex relationships between the input and output data.

ANN models are trained using a process called backpropagation. Backpropagation is an iterative process that adjusts the weights of the ANN model to minimize the error between the predicted output of the ANN model and the actual output data.

ANN models have been shown to be very effective at solving a wide range of problems. They are particularly well-suited for problems that involve complex relationships between the input and output data. ANN models are also able to learn from data that is noisy or incomplete.

Some of the advantages of ANN models include:

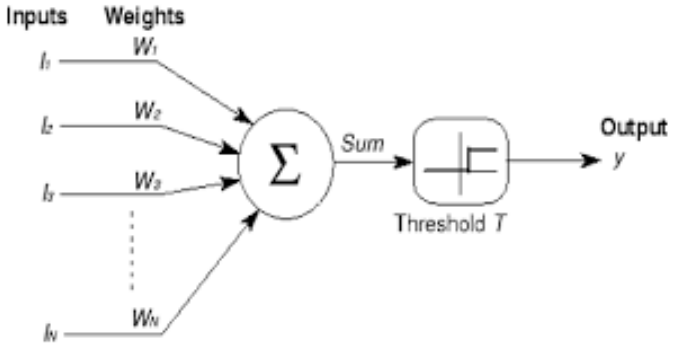
- They can learn complex relationships between input and output data.
- They are able to learn from data that is noisy or incomplete.
- They can be used to solve a wide range of problems.

Some of the disadvantages of ANN models include:

- They can be computationally expensive to train.
- They can be difficult to interpret.
- They can be sensitive to the choice of hyperparameters.

McCulloch and Pitts model

The McCulloch and Pitts model is a mathematical model of a neuron that was proposed by Warren McCulloch and Walter Pitts in 1943. It is a very simple model, but it is still considered to be the foundation of artificial neural networks.



The McCulloch and Pitts model has the following components:

- Inputs: The neuron has a set of inputs, which can be either 0 or 1.
- Weights: Each input has a weight associated with it, which determines how much the input contributes to the output of the neuron.
- Threshold: The neuron has a threshold value, which is the minimum value that the weighted sum of the inputs must be in order for the neuron to fire.
- Output: If the weighted sum of the inputs is greater than or equal to the threshold, the neuron fires and outputs a 1. Otherwise, the neuron does not fire and outputs a 0.

The McCulloch and Pitts model is a very simple model, but it can be used to simulate the behavior of real neurons in a number of ways. For example, the inputs to the neuron can represent the levels of neurotransmitters at the synapses of the neuron, the weights can represent the strengths of the synapses, and the threshold can represent the firing threshold of the neuron.

The McCulloch and Pitts model is not a perfect model of a real neuron, but it is a good starting point for understanding how artificial neural networks work. It has been used to develop a number of important concepts in artificial neural networks, such as the perceptron and the backpropagation algorithm.

Here are some of the limitations of the McCulloch and Pitts model:

- It is a very simple model, and it does not take into account many of the complexities of real neurons.
- It is not possible to train the McCulloch and Pitts model using real data.
- The McCulloch and Pitts model is not very efficient, and it can be slow to compute.

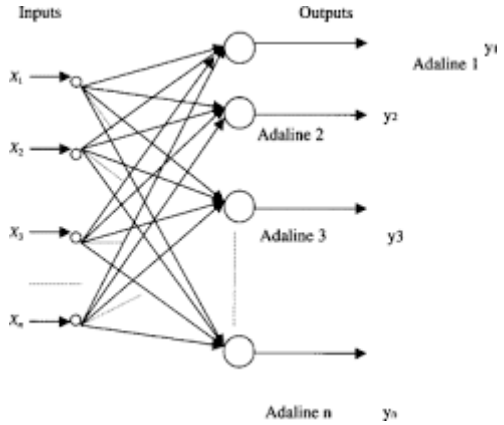
Despite its limitations, the McCulloch and Pitts model is an important milestone in the development of artificial neural networks. It is a simple and elegant model that has helped to pave the way for more complex and sophisticated artificial neural networks.

Adaline

Adaline (Adaptive Linear Neuron) is a single-layer artificial neural network that was developed by Bernard Widrow and Ted Hoff in 1960. It is a simple model, but it is still considered to be an important milestone in the development of artificial neural networks.

Adaline is a linear model, which means that it can only learn linear relationships between the input and output data. However, it can learn these relationships very effectively using a simple learning algorithm called the delta rule.

The delta rule is an iterative algorithm that adjusts the weights of the Adaline model to minimize the error between the predicted output of the model and the actual output data. The delta rule is very efficient and it can be used to train Adaline on large datasets.



Linear Activation Function: Adaline uses a linear activation function to produce continuous-valued outputs. The linear activation function computes the weighted sum of inputs without applying a threshold.
Continuous Output: The output of Adaline is not binary (0 or 1) like in the McCulloch-Pitts model. Instead, it produces a continuous output that reflects the weighted sum of inputs.
Least Mean Squares (LMS) Algorithm: Adaline employs the LMS algorithm for training. The goal of training is to minimize the difference between the predicted output and the actual target values by adjusting the weights.

Adaline is not as powerful as more complex neural networks, such as multilayer perceptrons, but it is still a useful tool for solving simple classification and regression problems. It is also a good starting point for understanding how artificial neural networks work.

Here are some of the advantages of Adaline:

- It is a simple model, which makes it easy to understand and implement.
- It is very efficient, and it can be used to train on large datasets.
- It can learn linear relationships between the input and output data very effectively.

Here are some of the disadvantages of Adaline:

- It is a linear model, which means that it can only learn linear relationships.
- It is not as powerful as more complex neural networks.
- It can be sensitive to noise in the data.

Perceptron

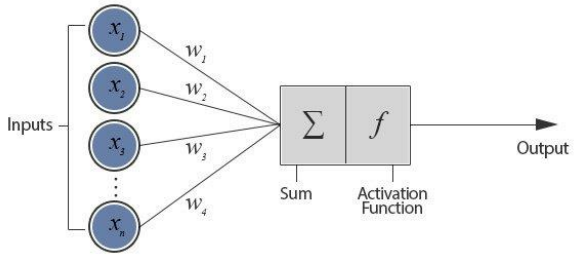
A perceptron is a single-layer artificial neural network that can be used for supervised learning. It is a linear classifier, which means that it can only learn linear relationships between the input and output data.

The perceptron was first introduced by Frank Rosenblatt in 1957. It is a simple model, but it is still considered to be an important milestone in the development of artificial neural networks.

The perceptron has the following components:

- Inputs: The perceptron has a set of inputs, which can be either 0 or 1.
- Weights: Each input has a weight associated with it, which determines how much the input contributes to the output of the perceptron.
- Threshold: The perceptron has a threshold value, which is the minimum value that the weighted sum of the inputs must be in order for the perceptron to fire.
- Output: If the weighted sum of the inputs is greater than or equal to the threshold, the perceptron fires and outputs a 1. Otherwise, the perceptron does not fire and outputs a 0.

The perceptron can be trained to classify data by adjusting the weights of the perceptron. The weights are adjusted using a simple learning algorithm called the perceptron learning rule.



The perceptron learning rule works as follows:

1. The perceptron is presented with a training example.
2. The perceptron calculates the weighted sum of the inputs.
3. If the weighted sum is greater than or equal to the threshold, the perceptron outputs a 1. Otherwise, the perceptron outputs a 0.
4. If the perceptron's output is incorrect, the weights of the perceptron are adjusted.
5. Steps 2-4 are repeated until the perceptron is able to classify all of the training examples correctly.

The perceptron is a simple model, but it can be used to solve a variety of problems. It is particularly well-suited for problems that involve linear relationships between the input and output data.

Here are some of the advantages of perceptrons:

- They are simple to understand and implement.
- They can be trained quickly and efficiently.
- They can be used to solve a variety of problems.

Here are some of the disadvantages of perceptrons:

- They can only learn linear relationships.
- They can be sensitive to noise in the data.
- They can be computationally expensive to train for large datasets.

Activation functions

Activation functions are a crucial component of artificial neural networks (ANNs) and other machine learning models. They introduce non-linearity to the network, enabling it to model complex relationships in the data. Activation functions determine the output of a neuron or a node in a neural network based on its input.

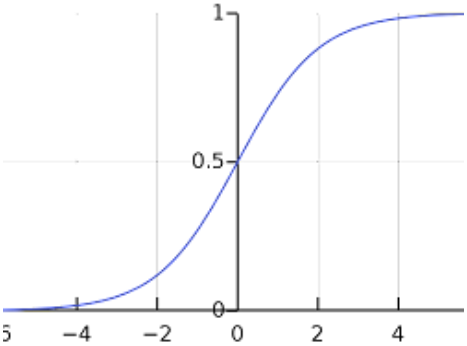
Here are some common activation functions used in neural networks:

Step Function:

- Outputs 0 for inputs below a certain threshold and 1 otherwise.
- Used in the original perceptron model for binary classification.
- Not differentiable, which makes it unsuitable for gradient-based optimization.

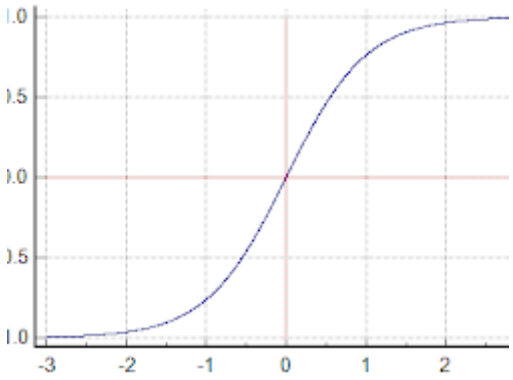
Sigmoid Function (Logistic Function):

The sigmoid function is a S-shaped function that outputs a value between 0 and 1. It is often used in ANNs for classification problems.



- Outputs values between 0 and 1.
- Smooth and differentiable, which allows gradient-based optimization.
- Commonly used in the past but less popular now due to vanishing gradient problem.

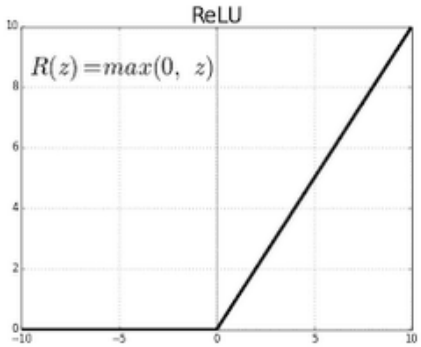
Hyperbolic Tangent (tanh) Function:



The tanh function is similar to the sigmoid function, but it outputs a value between -1 and 1. It is often used in ANNs for regression problems.

- Outputs values between -1 and 1.
- Similar to the sigmoid function but centered at 0.
- Smooth and differentiable, but still susceptible to vanishing gradients.

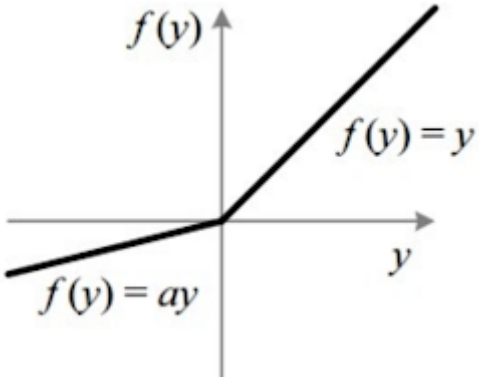
Rectified Linear Unit (ReLU):



The ReLU function is a rectified linear unit function that outputs the input value if it is positive, and 0 if it is negative. It is a popular activation function in deep learning ANNs.

- Outputs the input if it's positive, and 0 otherwise.
- Widely used due to its simplicity and effectiveness in deep networks.
- Solves the vanishing gradient problem for many cases.
- Not always differentiable at 0 (subderivatives are used in practice).

Leaky ReLU:



The leaky ReLU function is a modification of the ReLU function that allows for a small non-zero output value for negative input values. This makes it less sensitive to dead neurons than the ReLU function.

- Similar to ReLU, but allows a small gradient when the input is negative.
- Addresses the "dying ReLU" problem where neurons can become inactive during training.

Parametric ReLU (PReLU):

- Generalization of Leaky ReLU where the slope of the negative part is learned.
- Can further alleviate the dying ReLU problem.

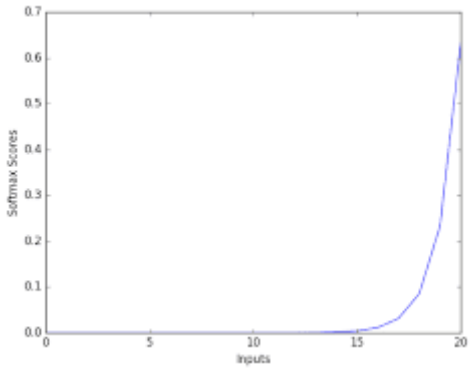
Exponential Linear Unit (ELU):

- Similar to ReLU, but smoothly curves for negative inputs.
- Introduces a non-zero gradient for negative values, addressing the dying ReLU problem.
- Can lead to improved learning in certain cases.

Scaled Exponential Linear Unit (SELU):

- A self-normalizing activation function that helps maintain mean and variance of activations.
- Designed for deeper networks and can automatically adjust to maintain activations.

Softmax Function:
The softmax function is a normalization function that outputs a vector of values that sum to 1. It is often used in ANNs for classification problems with multiple output classes.



- Used in the output layer for multi-class classification.
- Converts raw scores into probability distribution over multiple classes.
- Ensures that the sum of output probabilities is 1.

Activation Function	Output Range	Non-linear?	Efficient?	Used for
Linear	Same as input	No	Yes	Simple tasks
Sigmoid	0 to 1	Yes	No	Classification
Tanh	-1 to 1	Yes	No	Classification
ReLU	Positive values	Yes	Yes	Regression, classification
Leaky ReLU	Positive values and a small positive value for negative values	Yes	Yes	Regression, classification

realizing logic gates using perceptron

A perceptron is the simplest form of an artificial neural network, consisting of a single neuron that takes inputs, applies weights to them, and produces an output based on a threshold. Let's go through how you can implement some basic logic gates using perceptrons:

AND Gate:

The AND gate outputs 1 only when both inputs are 1, otherwise, it outputs 0.

Input 1	Input 2	Output
0	0	0
0	1	0
1	0	0
1	1	1

To realize the AND gate using a perceptron, you need to set the weights and threshold accordingly. For example, if you set weights as (0.5, 0.5) and the threshold as 0.7, the perceptron will act as an AND gate:

Input 1 weight: 0.5
Input 2 weight: 0.5
Threshold: 0.7

The perceptron calculates the weighted sum of inputs: weighted_sum = input1 * 0.5 + input2 * 0.5. If weighted_sum >= 0.7, it outputs 1; otherwise, it outputs 0.

OR Gate:
The OR gate outputs 1 when at least one of the inputs is 1.

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	1

To realize the OR gate using a perceptron, set the weights and threshold accordingly. For instance, if you set weights as (0.5, 0.5) and the threshold as 0.3:

Input 1 weight: 0.5
Input 2 weight: 0.5
Threshold: 0.3
The perceptron calculates the weighted sum of inputs: weighted_sum = input1 * 0.5 + input2 * 0.5. If weighted_sum >= 0.3, it outputs 1; otherwise, it outputs 0.

NOT Gate:
The NOT gate simply negates the input.

Input	Output
0	1
1	0

To realize the NOT gate using a perceptron, set the weight as -1 and the threshold as -0.5:

Input weight: -1
Threshold: -0.5
The perceptron calculates the weighted sum of the input: weighted_sum = input * -1. If weighted_sum >= -0.5, it outputs 0; otherwise, it outputs 1.

Remember that perceptrons are quite limited compared to more complex neural networks, and they can only represent linear decision boundaries. To represent more complex logic gates or operations, you might need to use multi-layer perceptrons (also known as feedforward neural networks) or other more advanced neural network architectures.

---

implementing perceptron using Python

```
import random

class Perceptron:
    def __init__(self, inputs, weights, threshold):
        self.inputs = inputs
        self.weights = weights
        self.threshold = threshold

    def predict(self, inputs):
        sum_of_weighted_inputs = 0
        for i in range(len(self.inputs)):
            sum_of_weighted_inputs += self.weights[i] * inputs[i]

        if sum_of_weighted_inputs >= self.threshold:
            return 1
        else:
            return 0

    def train(self, inputs, labels):
        for i in range(len(inputs)):
            output = self.predict(inputs[i])
            error = labels[i] - output
```

```
            for j in range(len(self.weights)):
                self.weights[j] += error * inputs[i][j]

            self.threshold += error

def main():
    inputs = [[0, 0], [0, 1], [1, 0], [1, 1]]
    labels = [0, 0, 0, 1]

    perceptron = Perceptron(inputs[0], [1, 1], 2)
    perceptron.train(inputs, labels)

    for i in range(len(inputs)):
        print(perceptron.predict(inputs[i]), labels[i])

if __name__ == "__main__":
    main()
```

This code first defines a class called Perceptron. This class has three attributes: inputs, weights, and threshold. The inputs attribute is a list of the input features. The weights attribute is a list of the weights for each input feature. The threshold attribute is the threshold value.

The predict() method of the Perceptron class takes a list of input features as input and returns the predicted output of the perceptron. The train() method of the Perceptron class takes a list of input features and labels as input and trains the perceptron on the data.

The main() function of the code first defines the input features and labels. Then, it creates a perceptron object and trains it on the data. Finally, it prints the predicted output of the perceptron for each input feature.

---

implementing functionality of logic gates using perceptron in python.

```
import random

class Perceptron:
    def __init__(self, inputs, weights, threshold):
        self.inputs = inputs
        self.weights = weights
        self.threshold = threshold

    def predict(self, inputs):
        sum_of_weighted_inputs = 0
        for i in range(len(self.inputs)):
            sum_of_weighted_inputs += self.weights[i] * inputs[i]

        if sum_of_weighted_inputs >= self.threshold:
            return 1
        else:
            return 0

    def train(self, inputs, labels):
        for i in range(len(inputs)):
            output = self.predict(inputs[i])
            error = labels[i] - output

            for j in range(len(self.weights)):
                self.weights[j] += error * inputs[i][j]

            self.threshold += error

def AND(a, b):
    perceptron = Perceptron([a, b], [1, 1], 2)
    perceptron.train([[0, 0], [0, 1], [1, 0], [1, 1]], [0, 0, 0, 1])
    return perceptron.predict([a, b])
```

```
def OR(a, b):
    perceptron = Perceptron([a, b], [1, 1], 1)
    perceptron.train([[0, 0], [0, 1], [1, 0], [1, 1]], [0, 1, 1, 1])
    return perceptron.predict([a, b])

def NOT(a):
    perceptron = Perceptron([a], [-1], 0)
    perceptron.train([[0], [1]], [1, 0])
    return perceptron.predict([a])

def XOR(a, b):
    perceptron = Perceptron([a, b], [1, -1], 1)
    perceptron.train([[0, 0], [0, 1], [1, 0], [1, 1]], [0, 1, 1, 0])
    return perceptron.predict([a, b])

if __name__ == "__main__":
    print(AND(0, 0))
    print(AND(0, 1))
    print(AND(1, 0))
    print(AND(1, 1))

    print(OR(0, 0))
    print(OR(0, 1))
    print(OR(1, 0))
    print(OR(1, 1))

    print(NOT(0))
    print(NOT(1))

    print(XOR(0, 0))
    print(XOR(0, 1))
    print(XOR(1, 0))
    print(XOR(1, 1))
```

This code first defines the classes Perceptron, AND, OR, NOT, and XOR. The Perceptron class is the same as the one we defined earlier. The AND, OR, NOT, and XOR classes simply create perceptrons with the appropriate weights and threshold values.

The main() function of the code first prints the output of the AND, OR, NOT, and XOR functions for different input values. The output shows that the perceptrons are able to correctly implement the functionality of the logic gates.

(Or)

```
class Perceptron:
    def __init__(self, num_inputs, threshold):
        self.weights = [0] * num_inputs
        self.threshold = threshold

    def predict(self, inputs):
        weighted_sum = sum(w * x for w, x in zip(self.weights, inputs))
        if weighted_sum >= self.threshold:
            return 1
        else:
            return 0

    def train(self, training_data, epochs=10, learning_rate=0.1):
        for _ in range(epochs):
            for inputs, target in training_data:
                prediction = self.predict(inputs)
                error = target - prediction
                for i in range(len(self.weights)):
                    self.weights[i] += learning_rate * error * inputs[i]
```

```
# Define the training data for logic gates
and_training_data = [([0, 0], 0),
                     ([0, 1], 0),
                     ([1, 0], 0),
                     ([1, 1], 1)]

or_training_data = [([0, 0], 0),
                    ([0, 1], 1),
                    ([1, 0], 1),
                    ([1, 1], 1)]

not_training_data = [([0], 1),
                     ([1], 0)]

# Create perceptrons for logic gates
and_perceptron = Perceptron(num_inputs=2, threshold=0.7)
or_perceptron = Perceptron(num_inputs=2, threshold=0.3)
not_perceptron = Perceptron(num_inputs=1, threshold=0.5)

# Train perceptrons on the training data
and_perceptron.train(and_training_data)
or_perceptron.train(or_training_data)
not_perceptron.train(not_training_data)

# Test the perceptrons
test_data = [([0, 0], "AND"),
             ([0, 1], "AND"),
             ([1, 0], "AND"),
             ([1, 1], "AND"),
             ([0, 0], "OR"),
             ([0, 1], "OR"),
             ([1, 0], "OR"),
             ([1, 1], "OR"),
             ([0], "NOT"),
             ([1], "NOT")]

for inputs, gate_type in test_data:
    if gate_type == "AND":
        result = and_perceptron.predict(inputs)
    elif gate_type == "OR":
        result = or_perceptron.predict(inputs)
    elif gate_type == "NOT":
        result = not_perceptron.predict(inputs)
    print(f"Inputs: {inputs}, Gate: {gate_type}, Output: {result}")
```

In this code, we define training data for the AND, OR, and NOT gates, create separate perceptrons for each gate, train the perceptrons using their respective training data, and then test their predictions on various inputs. The perceptrons learn to approximate the logic behavior of the gates by adjusting their weights during training.