6CS012 – Artificial Intelligence and Machine Learning.
Tutorial – 03
Getting Started with Artificial Neural Network.
An Introduction to Artificial Neuron.
MCP to The Perceptron.

Siman Giri {Module Leader – 6CS012}

1. Towards Generalization.

{ Why we do Train – Test Split?}

1.1 Idea of Generalizations.

- In layman terms **Generalizations** is
  - the study of how well your model works on unseen data.
- Obviously, it is not that simple, but we will try to build an intuition about it.
- Once you **trained your model** on some **observed sample data**, two possibilities arise:
  - **Scenario – 1 - Overfitting:**
    - You got very good result i.e. very **less error value**
      - But when you tried to predict on some unobserved or unseen data you got very bad result i.e. **very high error value**
      - {Intuition → Something is wrong with our training regime.}
  - **Scenario – 2 – Underfitting:**
    - You got very bad result i.e. high error on observed sample data,
    - {Intuition → we did not select the right model to fit our data}
- Generalizations theory provides **the framework** to study and understand it further …

1.2 Intuition behind Error in Machine Learning.

- **Total Error**: The total error for a model, {usually measured by a metric like Mean Squared Error (MSE) for regression}, can be decomposed into three key components:
- $\mathbf{Total\ Error} = \mathbf{Bias}^2 + \mathbf{Varaince} + \mathbf{Irreducible\ Error}.$
- **Bias:**
  - **What it is**: Bias refers to the error introduced by approximating a complex real-world problem with a simplified model. It reflects how far the model's predictions are from the true values on average.
  - **Cause**: Bias is often high in Underfitting models, such as linear models used to approximate nonlinear relationships.
- **Variance**:
  - **What it is**: Variance reflects the model's sensitivity to fluctuations in the training data. High variance implies that the model captures noise as if it were a signal.
  - **Cause**: Variance is often high in overfitting models that are too complex, like deep neural networks trained on small datasets.
- **Irreducible Error:**
  - **What it is**: This represents the inherent noise in the data that no model can explain. It's due to factors like random measurement errors or unmolded factors.
  - **Cause**: Comes from the data itself and cannot be reduced by improving the model.

1.2.1 Bias – Variance Tradeoff.

- Models **with high bias** will **have low variance** – Characterized by **Underfitting**.
  - Characteristics of High Biased model:
    - Underfitts.
    - Does not capture true trend in dataset.
- Models with **high variance** will **have a low bias** – Characterized by **Overfitting**.
  - Characteristics of High Variance model:
    - Noise in the dataset
    - Overfitts
    - Complex models
    - Trying to fit all the datapoints including irrelevant information

Fig: Bias Variance Tradeoff.

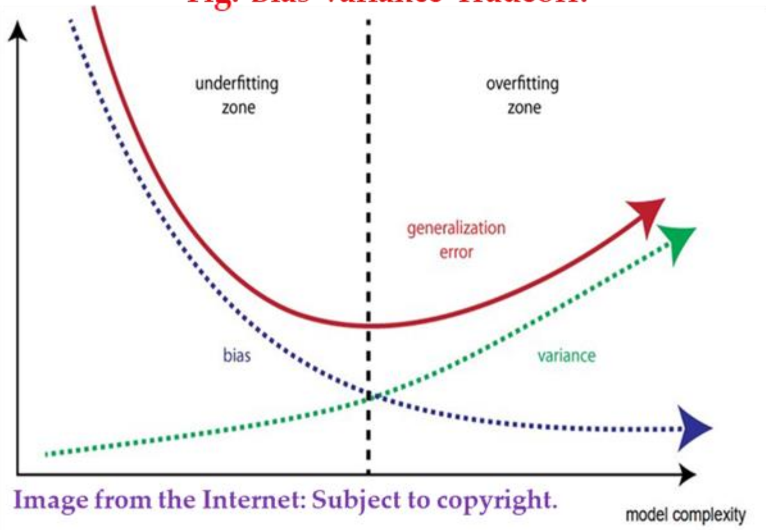


Image from the Internet: Subject to copyright.

1.3 Achieving Generalizations.

- For a data that is:
    - Observed Sample data: data $\mathcal{D}_{in}$ used to train model.
    - Out of sample data: data $\mathcal{D}_o$ used to measure the predictive quality of model.
- We want to build a model:
    - $\hat{\mathbf{f}}(\mathbf{x}) \approx \mathbf{f}(\mathbf{x});$
- such that it can make a best prediction on sampled data
    - $\hat{y}_i = \hat{f}(x_i) | E_{in} \ell(\hat{y}, y) \approx 0$ **{Expected loss is close to 0}**
- and we want similar performance also on observed data:
    - $\widehat{y_o} = \hat{f}(x_o) | E_o \ell(\widehat{y_o}, y_o) \approx 0$. **{Expected loss is close to 0}**
- {We want our model to perform well on both trained and unseen data}

- $\mathbf{E_{in}}$ - **Error on observed sampled data** which is typically defined as the average of *Pointwise errors* from data points in the sample data i.e.
    - $\mathbf{E_{in}}(\hat{\mathbf{f}}, \mathbf{f}) = \frac{1}{n} \sum_i \mathbf{err_i}$.
    - [re-substitution error → how well the model fits the learning data?]
- $\mathbf{E_o}$ **- Error on unseen data** is the *theoretical mean(expected value)* of the *Pointwise errors* over the entire **input space**:
    - $\mathbf{E_o}(\hat{\mathbf{f}}, \mathbf{f}) = \mathbf{E_X}\left[\mathbf{err}\left(\hat{\mathbf{f}}(\mathbf{x}), \mathbf{f}(\mathbf{x})\right)\right]$
    - [generalization error → how well the model fits the data?]
- The point $\mathbf{x}$ denotes a general data point in the **input space** $\mathcal{X}$. And as we said , the expectation is taken over the input space $\mathcal{X}$.
- As we only **have sample** of the input space $\mathcal{X}_{\mathbb{P}_{x,y}}$:
    - This means that the nature of $\mathbf{E_o}$ is highly theoretical.
    - In practice, we will **never** be able to **compute this quantity**.
    - **What do we do in practice?**

1.3.1 Achieving Generalizations in Practice.

- In Application we split our **Observed sample data $\mathcal{D}_{\text{in}}$** into:

• training data $\mathcal{D}_{train}$:
  • used to fit model (behaves as in-sample Data)
Train Error
For a Dataset:
$D_{train} = \{(x_1, y_1), \dots, (x_{n-a}, y_{n-a})\}.$
$E_{train}(\hat{f}, f) = \frac{1}{n-a} \sum_{i=1}^{n-a} err[\hat{f}(x_i), y_i];$
Training error typically underestimates test error.
	• and test data $\mathcal{D}_{test}$
	  • check generalization error (behaves as out-sample-data).
	Test Error
	For a Dataset:
	$D_{test} = \{(x_1, y_1), \dots, (x_a, y_a)\}: a > 1.$
	$E_{test}(\hat{f}, f) = \frac{1}{a} \sum_{l=1}^{a} err[\hat{f}(x_l), y_l];$
	E_{test} is unbiased estimate of $E_{out}(h)$.

- **How do we split?**

1.3.1 Achieving Generalizations Train – Test Split.

- Holdout Method:
  - For available data
    - $\mathcal{D}_{in} = \{(x_1, y_1), \dots, (x_n, y_n)\}$.
    - We could split a data into two random sets as:
      - $\mathcal{D}_{in} \rightarrow \begin{cases} D_{train} & \rightarrow \text{size } (n - a) \\ D_{test} & \rightarrow \text{size } a \end{cases}$
      - D_{train}: **training set, used to fit/train the model.**
      - D_{test}: **test set, used to test or access the model.**
  - Reserving some points D_{test} from a **learning set D** to use them for testing purposes is a very wise idea.
    - It allows us to have a way for estimating the performance of model "$f_{\theta^*} \in \mathcal{F}$", when applied to out of sample points.
- **Holdout method is a conceptual starting point**,
  - There are many conventions as to how to pick a *"a"*:
  - once common rule-of-thumb is to assign 80% of your data to the training set, and remaining 20% to the test set.
  - {We will discuss some another method like cross-validation in upcoming weeks.}

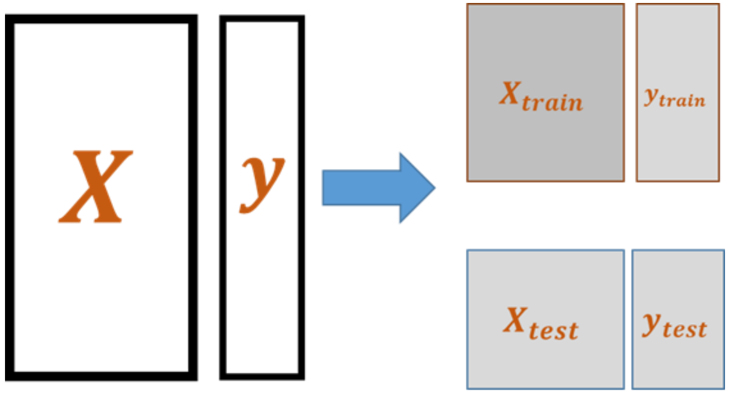


Fig: General Idea of Train-Test Split.

1.3.2 Achieving Generalizations Train – Test Split.

- Extension to Holdout Method:
  - Three-way Holdout Method:
    - We can extend the idea behind the holdout method, to go from one holdout set to two holdout sets.
    - Idea!!! Instead of splitting D into two sets (training and test), we split it into three sets (randomly):

$$\mathcal{D}_{in} \rightarrow \begin{cases} D_{train} \rightarrow size(n-(b+a)) \\ D_{val} \rightarrow size\,(b) \\ D_{test} \rightarrow size\,(a) \end{cases}$$

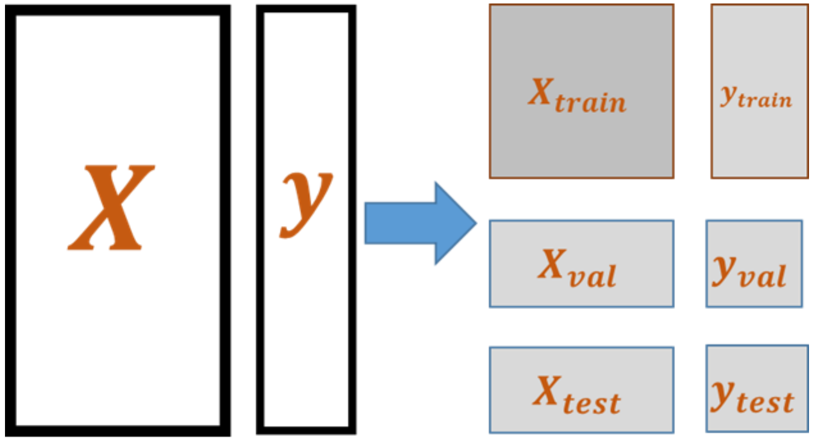


Fig: Three Way Holdout Methods.

1.4 Redefining Over and Under Fitting.

- Overfitting can be thought as:
  - We have low error in train set (**Low training error**) but high error in test set (**High Test error**).
  - **High Variance.**
- Underfitting can be thought as:
  - High error in Train set (**High training error**),
  - **High Bias.**
    - {If we have high error in train set we do not expect to get high in test set}

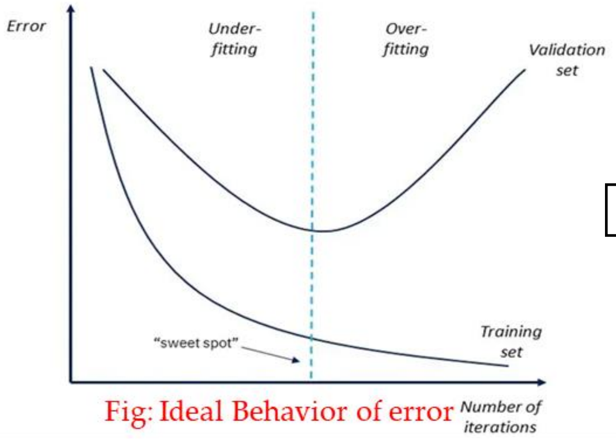
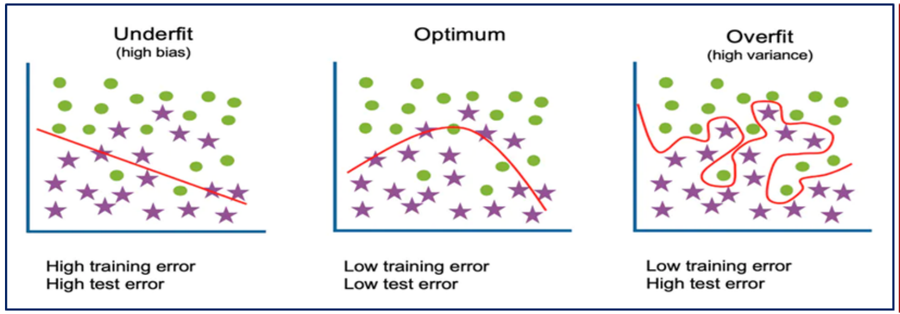


Fig: Ideal Behavior of error

In Practice

For Classification Task.



For Regression Task.

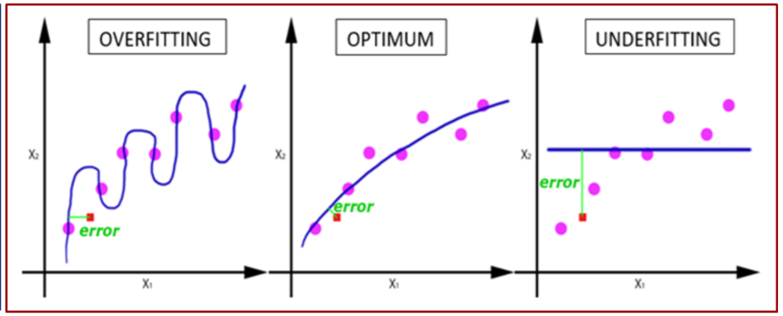


Fig: Graphical Analogy of Over and Underfitting

1.5 How do we handle Overfitting in Practice?

- Some Techniques to avoid overfitting:
  - Early stopping
  - Train with more data
  - Feature Selection or Data augmentation
  - Cross-Validation
  - Ensemble Methods
  - Regularization
- {We will cover all this in upcoming weeks in the perspective of Deep Learning.}

1.6 Machine Vs. Deep Learning.

Fig: A Machine Learning.

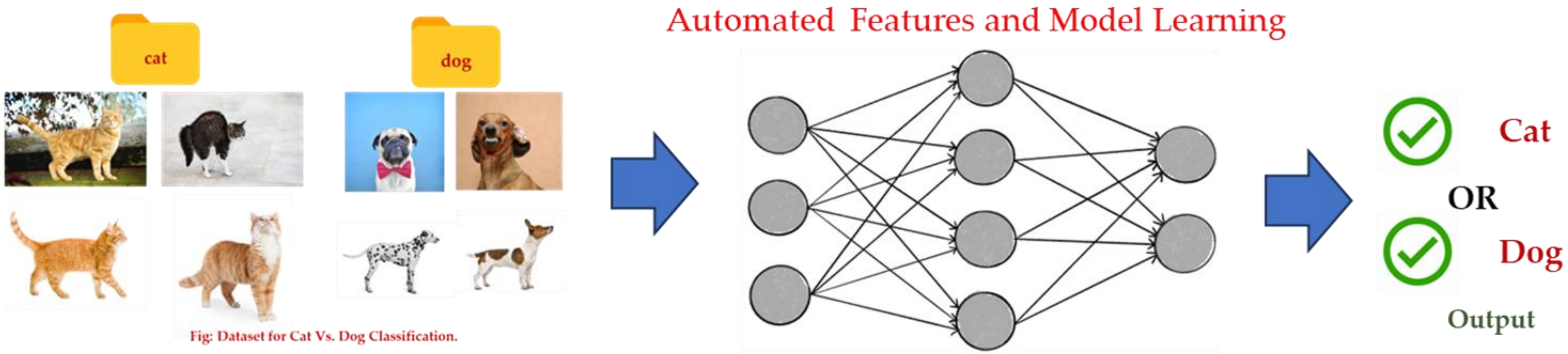
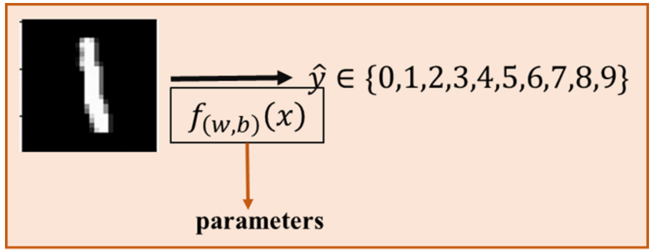


Fig: A Deep Learning.

1.7 Supervised Classification: Soup to Nuts.

- **General Overview of {Supervised} Machine Learning(Deep Learning Perspective):**
  - **Training Set:**
    - A sample of objects with known class labels is called training set and is written as $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ **i.e.**
      - X: **Feature Space** $\in [x_1^1 \quad \dots \quad x_n^d]$ **where** $(1, \dots, d)$: $\rightarrow$ **columns and** $(1, \dots, n)$: $\rightarrow$ **rows.**
      - **where** $y_i \in \{-1, 1\}$ **is the class label of vector** x_i **and** n **is the size of the training set.**
  - **Target Function:**
    - $f: X \rightarrow Y$, **unknown.**
    - Relationship{Assumption Linear $f: \rightarrow wx + b$}:
      - $y_n = f_{w,b}(x_n)$
  - **Minimizing loss function:**
    - Ideally, we want: $\widehat{y_n} = f_{w,b}(x_n) \approx y_n$ we try to achieve this by: $argmin\mathbb{L}(y, \hat{y})$
  - **Decision Function: assigns class.**



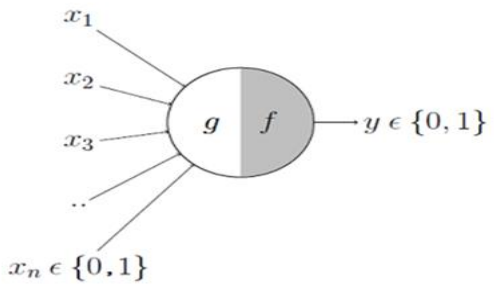
2. Understanding the Neurons.

{ Foundational Building Block of Artificial Neural Network and Inspired by Human Brain.}

2.1 MuCulloch – Pits Neurons.

- aka MCP neuron was a first computational model of a human neuron proposed by Warren MuCulloch and Walter Pitts in 1943.
- Following is the Mathematical Formalization of MCP:
  - MCP describes the activity of single neuron with **two states:** firing(1) or not firing(0).

For input $\mathbf{x_i} \in \{0, \text{or } 1\}$ and output $\mathbf{y_i} = \{0 \text{ or } 1\}$
{
    $\mathbf{y = 0}$ if any $\mathbf{x_i}$ is inhibitory,
else:
    {
    $\mathbf{g(x_1, x_2, x_3 \ldots, x_n) = g(X) = \sum_{i=1}^{n} x_i}$
    {
    if: $\mathbf{g(X) \geq T}$
        $\mathbf{y = f(g(X)) = 1}$
    else: $\mathbf{g(X) < T}$
        $\mathbf{y = f(g(X)) = 0}$
    }



We will use this notation for easy representations.

2.2 Limitations of MCP Neurons.

- A single **McCulloch Pitts Neuron** can be used to represent boolean functions which are linearly separable.
  - **Linear separability** (**for boolean functions**) :
    - There exists a line (plane) such that
      - all inputs which produce a 1 lie on one side of the line (plane)
      - and all inputs which produce a 0 lie on other side of the line (plane)
- The **MCP Neuron Architecture** lacks **several characteristics** of **biological networks**:
  - **Complex connectivity patterns i.e. represents single neuron only**
  - **Processing of continuous values**
  - **A measure of importance**
  - **A learning procedure**
- Regardless of limitation, MCP is considered a significant first step towards development of ANN models

- **Towards Better Model…**

2.3 The Perceptron.

- The **Perceptron**, proposed by **Rosenblatt(1958),**
  - was an improvement over **McCulloch-Pitts (MCP) neurons**,
    - introducing a computational model with
      - **weighted inputs** and a **decision function** along with **learning theory** for **classification task**.

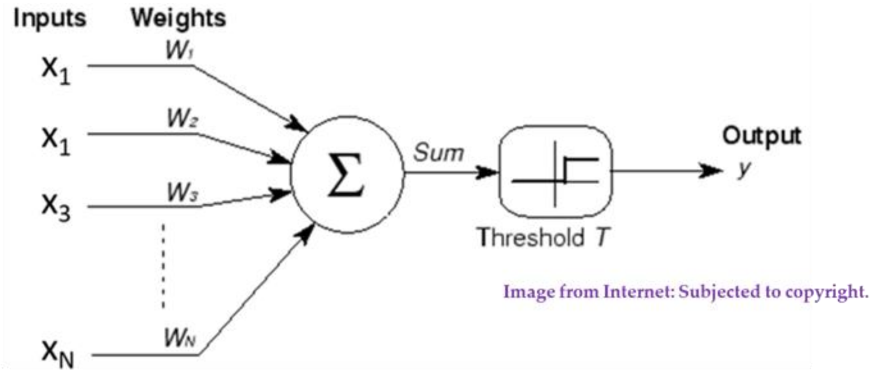
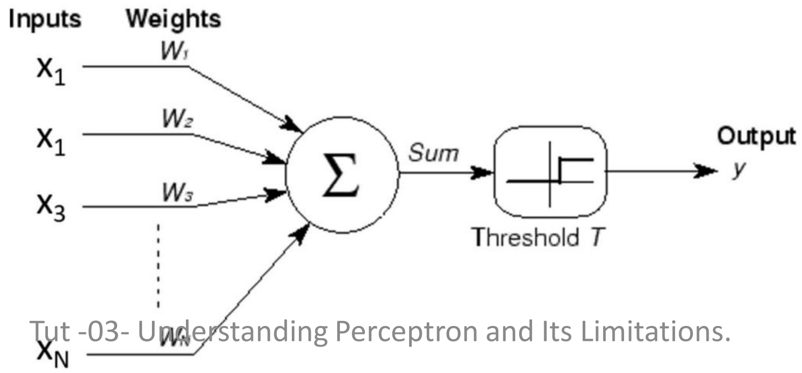


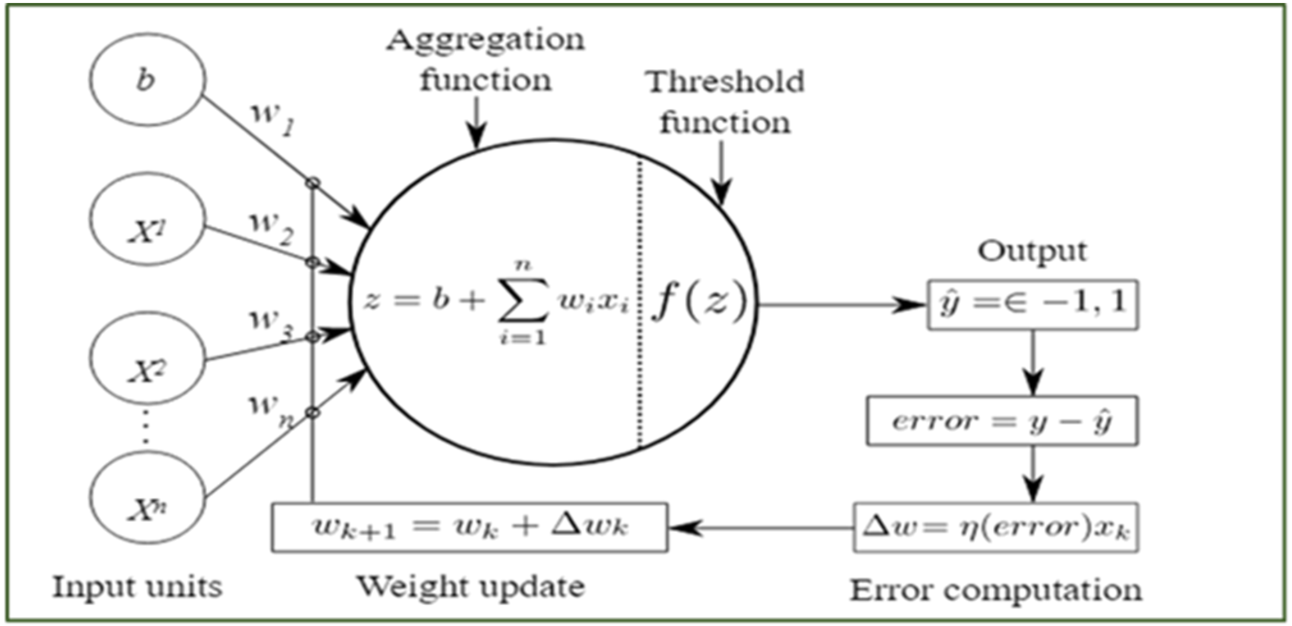
Fig: A Computational Representation of The Perceptron.

2.4 Simplified Computational Representation of Perceptron.

- **Mathematical Formulation:**
  - A perceptron takes a set of inputs $\mathbf{x_1, x_2, \ldots, x_n}$, assigns them weights $\mathbf{w_1, w_2, \ldots, w_n}$ and computes a weighted sum:
    - $\mathbf{z = \sum_{i=1}^{n} w_i x_i + w_o}$
    - here:
      - w_i → weights learned during training.
      - w_o → bias also learned during training. (adjusts the decision boundary)
      - z → net weighted input.
- The perceptron model then applies a threshold activation function (aka step function) on **z (net weighted input)** given by:
  - $\mathbf{f(z) = \begin{cases} 1 \text{ if } z \geq 0 \\ 0 \text{ if } z < 0 \end{cases}}$
  - This **threshold function** determines whether the perceptron activates (outputs 1) or remains inactive (outputs 0).



2.4.1 The Perceptron Learning Algorithm.



The Perceptron with Learning Algorithm

Fig: A Pictorial Representation of Perceptron Learning Algorithm.

2.4.2 Implementation of Perceptron Learning Algorithm.

Perceptron Learning Algorithm with Random Initialization

Algorithm 1 Perceptron Learning Algorithm

Require: Training dataset $D = \{(x_i, y_i)\}$, where:

x_i is the input feature vector (including bias term)

$y_i \in \{0, 1\}$ or $y_i \in \{-1, 1\}$

Require: Learning rate η (small positive value, e.g., 0.01)

Require: Number of epochs (max iterations)

Initialize: Randomly assign small values to $w = (w_1, w_2, \ldots, w_n)$

Randomly initialize bias b (or include it in w)

for each epoch **do**

  ConvergenceFlag = **True**

  for each training sample (x_i, y_i) **do**

    Compute weighted sum:

$$z = \sum w_i x_i + b$$

    Apply activation function (step function):

$$\hat{y} = \begin{cases} 1, & \text{if } z \geq 0 \\ 0, & \text{if } z < 0 \end{cases}$$

    if $\hat{y} \neq y$ **then** ▷ Update weights if misclassified

      Update weights:

$$w_i = w_i + \eta(y - \hat{y})x_i$$

      Update bias:

$$b = b + \eta(y - \hat{y})$$

      ConvergenceFlag = **False**

    end if

  end for

  if ConvergenceFlag is **True then**

    Break ▷ Stop if no updates occur (convergence)

  end if

end for

Output: Learned weights w and bias b

3. The Perceptron and Boolean Function Approximations.

{ Can we use Perceptron Learning Theory to Learn Boolean function?}

3.1 Perceptron for “OR” Function.

OR Function

Step 1: Define the OR Function

The OR function follows this truth table:

x_1	x_2	OR Output y
0	0	0
0	1	1
1	0	1
1	1	1

The perceptron will learn the weights (w_1, w_2) and bias w_0 to correctly classify these examples.

3.1.1 Step – 2: Initialization of Weights.

Step 2: Initialize Weights Randomly

We initialize weights with small random values:

$$w_1 = 0.2, \quad w_2 = -0.1, \quad w_0 = 0.1 \text{ (bias)}$$

Learning rate:

$$\eta = 0.1$$

3.1.2 Step – 3: Start of the Training Process.

Step 3: Training Process (Epoch 1)

For each training example, compute the weighted sum:

$$z = w_1 x_1 + w_2 x_2 + w_0$$

Apply the step function:

$$\hat{y} = \begin{cases} 1, & \text{if } z \geq 0 \\ 0, & \text{if } z < 0 \end{cases}$$

Update rule (if $y \neq \hat{y}$):

$$w_i = w_i + \eta(y - \hat{y})x_i$$

$$w_0 = w_0 + \eta(y - \hat{y})$$

3.1.3 Step – 4: Epoch – 1 – Step by Step Weight update.

1st Training Example: (0,0) $\rightarrow$ Expected $y = 0$

Compute weighted sum:

$$z = (0.2 \times 0) + (-0.1 \times 0) + 0.1 = 0.1$$

Apply step function:

$$\hat{y} = 1 \quad (\text{since } 0.1 \geq 0)$$

Misclassified $(y \neq \hat{y})$, so update weights:

$$w_1 = 0.2 + 0.1(0 - 1) \times 0 = 0.2$$

$$w_2 = -0.1 + 0.1(0 - 1) \times 0 = -0.1$$

$$w_0 = 0.1 + 0.1(0 - 1) = 0.0$$

3.1.3 Step – 4: Epoch – 1 – Step by Step Weight update.

2nd Training Example: (0,1) $\rightarrow$ Expected $y = 1$

Compute weighted sum:

$$z = (0.2 \times 0) + (-0.1 \times 1) + 0.0 = -0.1$$

Apply step function:

$$\hat{y} = 0 \quad (\text{since } -0.1 < 0)$$

Misclassified, so update weights:

$$w_1 = 0.2 + 0.1(1 - 0) \times 0 = 0.2$$

$$w_2 = -0.1 + 0.1(1 - 0) \times 1 = 0.0$$

$$w_0 = 0.0 + 0.1(1 - 0) = 0.1$$

3.1.3 Step – 4: Epoch – 1 – Step by Step Weight update.

3rd Training Example: (1,0) $\rightarrow$ Expected $y = 1$

Compute weighted sum:

$$z = (0.2 \times 1) + (0.0 \times 0) + 0.1 = 0.3$$

Apply step function:

$$\hat{y} = 1 \quad (\text{since } 0.3 \geq 0)$$

Correctly classified ($y = \hat{y}$), so no update.

4th Training Example: (1,1) $\rightarrow$ Expected $y = 1$

Compute weighted sum:

$$z = (0.2 \times 1) + (0.0 \times 1) + 0.1 = 0.3$$

Apply step function:

$$\hat{y} = 1 \quad (\text{since } 0.3 \geq 0)$$

Correctly classified, so no update.

3.1.4 Epoch – 1 – Final Weight Update.

Weight Updates During Epoch 1

Training Sample (x_1, x_2, y)	**Computed** z	**Predicted** $\hat{y}$	**Error** $y - \hat{y}$	**Weight Updates**
$(0, 0, 0)$	0.1	1	$0 - 1 = -1$	$w_0 = 0.1 - 0.1 = 0.0$
$(0, 1, 1)$	-0.1	0	$1 - 0 = 1$	$w_2 = -0.1 + 0.1 = 0.0$, $w_0 = 0.0 + 0.1 = 0.1$
$(1, 0, 1)$	0.3	1	$1 - 1 = 0$	No update
$(1, 1, 1)$	0.3	1	$1 - 1 = 0$	No update

- **Final Weights at the end of Epoch 1:**
  1. $w_1 = 0.2 \rightarrow$ unchanged.
  2. $w_2 = 0.0$
  3. $w_0 = 0.1$

3.1.5 Start of Epoch – 2.

Epoch 2 - Re-evaluating All Points

x_1	x_2	y	z	$\hat{y}$	**Update?**
0	0	0	0.1	1	Yes, $w_0 = 0.1 - 0.1 = 0.0$
0	1	1	0.0	1	?
1	0	1	0.2	1	?
1	1	1	0.2	1	?

3.1.5 Start of Epoch – 2.

Epoch 2 - Re-evaluating All Points

x_1	x_2	y	z	$\hat{y}$	**Update?**
0	0	0	0.1	1	Yes, $w_0 = 0.1 - 0.1 = 0.0$
0	1	1	0.0	1	No
1	0	1	0.2	1	No
1	1	1	0.2	1	No

- Final Learned Weights:
  1. $w_1 = 0.2$
  2. $w_2 = 0.0$
  3. $w_0 = 0.0$

At this point, all classifications are correct, and no weight updates occur. The perceptron has converged.

3.1.6 Final Decision Boundary.

Step 4: Final Decision Boundary

Final Equation:

$$z = 0.2x_1 + 0.0x_2 + 0.0$$

Classification Rules:

- If $x_1 = 1$ or $x_2 = 1$, then $z \geq 0 \Rightarrow y = 1$.
- If $x_1 = 0$ and $x_2 = 0$, then $z < 0 \Rightarrow y = 0$.

Conclusion: The perceptron successfully learns the OR function.

3.2 Python Implementations.

Class Definition:

```
import numpy as np
class Perceptron:
    """
    A simple Perceptron classifier for binary classification.
    Attributes:
    -----------
    weights : np.ndarray
        Array of weights including the bias term, initialized randomly.
    learning_rate : float
        The step size for weight updates.
    epochs : int
        The number of training iterations over the dataset.

    Methods:
    --------
    step_function(z)
        Activation function that returns 1 if z >= 0, else 0.

    predict(x)
        Predicts the output for a given input sample x.

    train(X, Y)
        Trains the perceptron on the given dataset (X: inputs, Y: target labels).
    Example:
    --------
    X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
    Y = np.array([0, 1, 1, 1]) # OR function
    perceptron = Perceptron(input_size=2, learning_rate=0.1, epochs=10)
    perceptron.train(X, Y)
    print(perceptron.predict([1, 1])) # Expected output: 1
    """
```

3.2 Python Implementations.

Defining " init" for initializations :

```
def __init__(self, input_size, learning_rate=0.1, epochs=10):
        """
        Initializes the perceptron with small random weights.
        Parameters:
        -----------
        input_size : int
            Number of input features.
        learning_rate : float, optional (default=0.1)
            Step size for weight updates.
        epochs : int, optional (default=10)
            Number of times to iterate over the dataset.
        """
        self.weights = np.random.rand(input_size + 1) * 0.2 - 0.1 # Small random weights
        self.learning_rate = learning_rate
        self.epochs = epochs
```

3.2 Python Implementations.

Making Decision using Step Function.

```
def step_function(self, z):
        """
        Step activation function.

        Parameters:
        -----------
        z : float
            Weighted sum of inputs and weights.

        Returns:
        --------
        int
            1 if z >= 0, else 0.
        """
        return 1 if z >= 0 else 0
```

3.2 Python Implementations.

Making a Prediction Function:

```
def predict(self, x):
    """
    Predicts the output for a given input sample.

    Parameters:
    -----------
    x : array-like
        Input feature vector.

    Returns:
    --------
    int
        Predicted class label (0 or 1).
    """
    x = np.insert(x, 0, 1) # Bias term
    z = np.dot(self.weights, x)
    return self.step_function(z)
```

3.2 Python Implementations.

Making a Prediction Function:

```
def predict(self, x):
    """
    Predicts the output for a given input sample.

    Parameters:
    -----------
    x : array-like
        Input feature vector.

    Returns:
    --------
    int
        Predicted class label (0 or 1).
    """
    x = np.insert(x, 0, 1) # Bias term
    z = np.dot(self.weights, x)
    return self.step_function(z)
```

3.2 Python Implementations.

Implementing Perceptron Learning Algorithm for Training:

```
def train(self, X, Y):
        """
        Trains the perceptron using the perceptron learning rule.

        Parameters:
        -----------
        X : np.ndarray
            Training data (each row is an input sample).
        Y : np.ndarray
            Corresponding target labels (0 or 1).

        Prints:
        -------
        Updates the weights and prints them after each epoch.
        """
        X = np.c_[np.ones(X.shape[0]), X] # Add bias term to input
        for epoch in range(self.epochs):
            for i in range(X.shape[0]):
                z = np.dot(self.weights, X[i])
                y_pred = self.step_function(z)
                error = Y[i] - y_pred
                self.weights += self.learning_rate * error * X[i]
            print(f"Epoch {epoch+1}, Weights: {self.weights}")
```

3.2 Python Implementations.

Putting it all Together:

```
import numpy as np
# I Removed Docstring to Fit in the Slide # Not a Good Practise.
class Perceptron:
    def __init__(self, input_size, learning_rate=0.1, epochs=10):
        self.weights = np.random.rand(input_size + 1) * 0.2 - 0.1 # Small random weights
        self.learning_rate = learning_rate
        self.epochs = epochs

    def step_function(self, z):
        return 1 if z >= 0 else 0

    def predict(self, x):
        x = np.insert(x, 0, 1) # Bias term
        z = np.dot(self.weights, x)
        return self.step_function(z)

    def train(self, X, Y):
        X = np.c_[np.ones(X.shape[0]), X] # Add bias term to input
        for epoch in range(self.epochs):
            for i in range(X.shape[0]):
                z = np.dot(self.weights, X[i])
                y_pred = self.step_function(z)
                error = Y[i] - y_pred
                self.weights += self.learning_rate * error * X[i]
            print(f"Epoch {epoch+1}, Weights: {self.weights}")
```

3.2 Python Implementations.

Testing:

```
# Example usage with adjustable input size
learning_rate = 0.1
epochs = 10
input_size = 3
# Create training data (modify as needed for different functions and input sizes)
X = np.array([
    [0, 0, 0],
    [0, 0, 1],
    [0, 1, 0],
    [0, 1, 1],
    [1, 0, 0],
    [1, 0, 1],
    [1, 1, 0],
    [1, 1, 1],
])
# Example: OR function with 3 inputs
Y = np.array([0, 1, 1, 1, 1, 1, 1, 1])
# Initialize and train the perceptron
perceptron = Perceptron(input_size, learning_rate=learning_rate, epochs=epochs)
perceptron.train(X, Y)
# Test the perceptron
test_samples = X # Use the same training samples for testing
print("\nPredictions:")
for sample in test_samples:
    print(f"Input: {sample}, Prediction: {perceptron.predict(sample)}")
```

3.3 Let's Try for XOR.

XOR Problem Recap

XOR Problem Recap:
The XOR (exclusive OR) gate produces the following outputs:

x_1	x_2	Output (Y)
0	0	0
0	1	1
1	0	1
1	1	0

We need to train a perceptron to solve this. The perceptron computes the weighted sum of inputs, applies a step function, and updates the weights based on the error.

3.3.2 Step – 1 – Initialization.

Step 1: Initialize Weights and Parameters

Start by initializing the weights w_0, w_1, w_2 (bias and input weights) to small random values (e.g., 0.2, -0.1). Set the learning rate η to 0.1.

$$w_0 = 0.1, \quad w_1 = 0.2, \quad w_2 = -0.1, \quad \eta = 0.1$$

3.3.3 Step – 2 – Weighted Sum and Step Function.

Step 2: Compute Weighted Sum and Apply Step Function

For each input pair (x_1, x_2), compute the weighted sum $z = w_1 x_1 + w_2 x_2 + w_0$, then apply the step function:

$$\hat{y} = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

If the prediction $\hat{y}$ is incorrect, update the weights:

$$w_i = w_i + \eta(y - \hat{y})x_i \quad \text{for } i = 1, 2$$

$$w_0 = w_0 + \eta(y - \hat{y})$$

3.3.4 Step – 3 – Training for XOR.

Step 3: Training for XOR

Let's train using the XOR truth table.

Epoch 1 - Update for Each Training Example:

Initial weights: $w_0 = 0.1, w_1 = 0.2, w_2 = -0.1$

Training Example 1: (0, 0) $\rightarrow$ **Expected** $y = 0$

- Compute weighted sum:

$$z = (0.2 \times 0) + (-0.1 \times 0) + 0.1 = 0.1$$

- Apply step function:

$$\hat{y} = 1 \quad (\text{since } 0.1 \geq 0)$$

- Misclassified, update weights:

$$w_1 = 0.2 + 0.1 \times (0 - 1) \times 0 = 0.2 \quad (\text{no change})$$

$$w_2 = -0.1 + 0.1 \times (0 - 1) \times 0 = -0.1 \quad (\text{no change})$$

$$w_0 = 0.1 + 0.1 \times (0 - 1) = 0.0$$

3.3.4 Step – 3 – Training for XOR.

Step 4: Training Example 2

Training Example 2: (0, 1) to Expected $y = 1$

- Compute weighted sum:

$$z = (0.2 \times 0) + (-0.1 \times 1) + 0.0 = -0.1$$

- Apply step function:

$$\hat{y} = 0 \quad \text{(since -0.1 ¡ 0)}$$

- Misclassified, update weights:

$$w_1 = 0.2 + 0.1 \times (1 - 0) \times 0 = 0.2 \quad \text{(no change)}$$

$$w_2 = -0.1 + 0.1 \times (1 - 0) \times 1 = 0.0$$

$$w_0 = 0.0 + 0.1 \times (1 - 0) = 0.1$$

3.3.4 Step – 3 – Training for XOR.

Step 5: Training Example 3

Training Example 3: (1, 0) $\rightarrow$ **Expected** $y = 1$

- Compute weighted sum:

$$z = (0.2 \times 1) + (0.0 \times 0) + 0.1 = 0.3$$

- Apply step function:

$$\hat{y} = 1 \quad (\text{since } 0.3 \geq 0)$$

- Correctly classified, no update.

3.3.4 Step – 3 – Training for XOR.

Step 6: Training Example 4:

Training Example 4: (1, 1) → Expected $y = 0$

- Compute weighted sum:

$$z = (0.2 \times 1) + (0.0 \times 1) + 0.1 = 0.3$$

- Apply step function:

$$\hat{y} = 1 \quad (\text{since } 0.3 \geq 0)$$

- Misclassified, update weights:

$$w_1 = 0.2 + 0.1 \times (0 - 1) \times 1 = 0.1$$
$$w_2 = 0.0 + 0.1 \times (0 - 1) \times 1 = -0.1$$
$$w_0 = 0.1 + 0.1 \times (0 - 1) = 0.0$$

3.3.5 Can you try for Epoch – 2.

What happened?

3.3.6 Realization of “XOR” Problem.

Step 7: Epoch 2 and Final Conclusion

Epoch 2

Repeat the process for another epoch (i.e., another pass through the training examples). Students will realize that the perceptron does not converge and continues making mistakes on the XOR problem.

Final Conclusion: After Epoch 2, if the perceptron still doesn't learn the XOR function, it demonstrates that XOR is not linearly separable. A single-layer perceptron cannot solve the XOR problem. The perceptron will keep making errors because XOR needs a more complex decision boundary.

3.4 Python Implementation.

- Check the Starter Code:

4. The Perceptron for “XOR”.

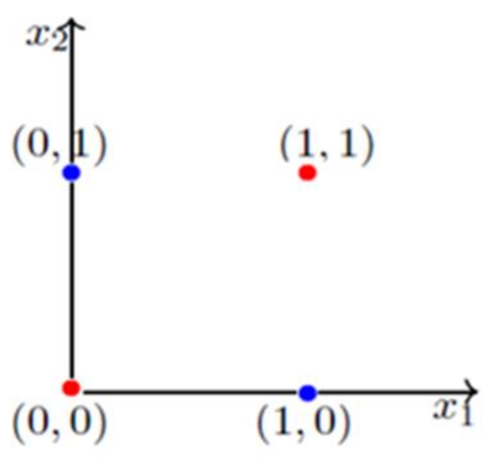
{Understanding Minsky and Papert Correction}

4.1 Why the perceptron will not converge for XOR?

- XOR is not linearly separable:

x_1	x_2	XOR	
0	0	0	$w_0 + \sum_{i=1}^{2} w_i x_i < 0$
1	0	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$
0	1	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$
1	1	0	$w_0 + \sum_{i=1}^{2} w_i x_i < 0$

$$w_0 + w_1 \cdot 0 + w_2 \cdot 0 < 0 \implies w_0 < 0$$
$$w_0 + w_1 \cdot 0 + w_2 \cdot 1 \geq 0 \implies w_2 > -w_0$$
$$w_0 + w_1 \cdot 1 + w_2 \cdot 0 \geq 0 \implies w_1 > -w_0$$
$$w_0 + w_1 \cdot 1 + w_2 \cdot 1 \geq 0 \implies w_1 + w_2 < -w_0$$

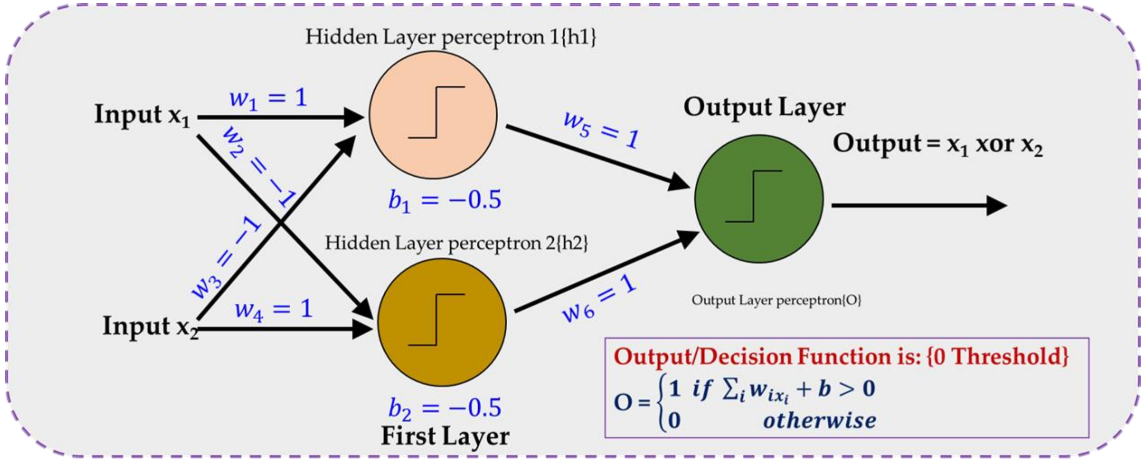


4.2 Why the perceptron will not converge for XOR?

- **The perceptron update rule keeps adjusting the weights:**
  - After each epoch, if the perceptron makes a mistake
    - (i.e., when predicted output $\hat{y}$ does not match the expected output y), it updates the weights using the rule:
      - $\mathbf{w_i} = \mathbf{w_i} + \boldsymbol{\eta}(\mathbf{y} - \hat{\mathbf{y}})\mathbf{x_i}$
  - Since XOR is not linearly separable, the perceptron will continue to misclassify certain points.
  - As long as the perceptron misclassifies an example,
    - it will keep adjusting the weights based on the learning rule.
  - When will it stop?
- **Probably Never – The weights are constantly updated:**
  - In the case of XOR, the perceptron will continuously misclassify some examples because it cannot find a line (or hyperplane) that divides the data correctly.
  - For each mistake, the weights will be adjusted, but there will always be some examples where the perceptron gets it wrong, causing further weight updates.
  - Thus, No Convergence.

4.3 Solving the "XOR" with Multilayer Perceptron.

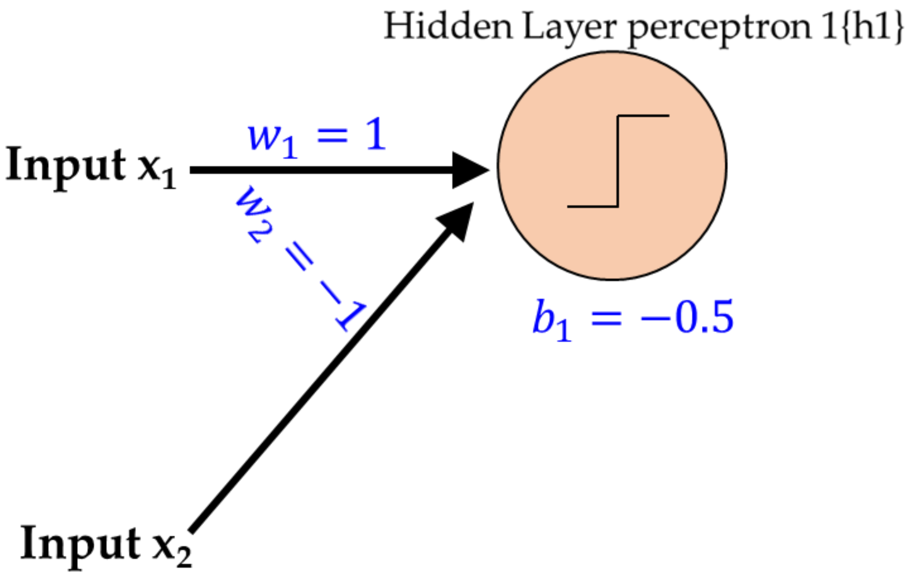
- Below Perceptron can solve for "XOR":



x_1	x_2	x_1 xor x_2
0	0	0
0	1	1
1	0	1
1	1	0

Solving for First Input Row @h1

x_1	x_2	x_1 xor x_2
0	0	0



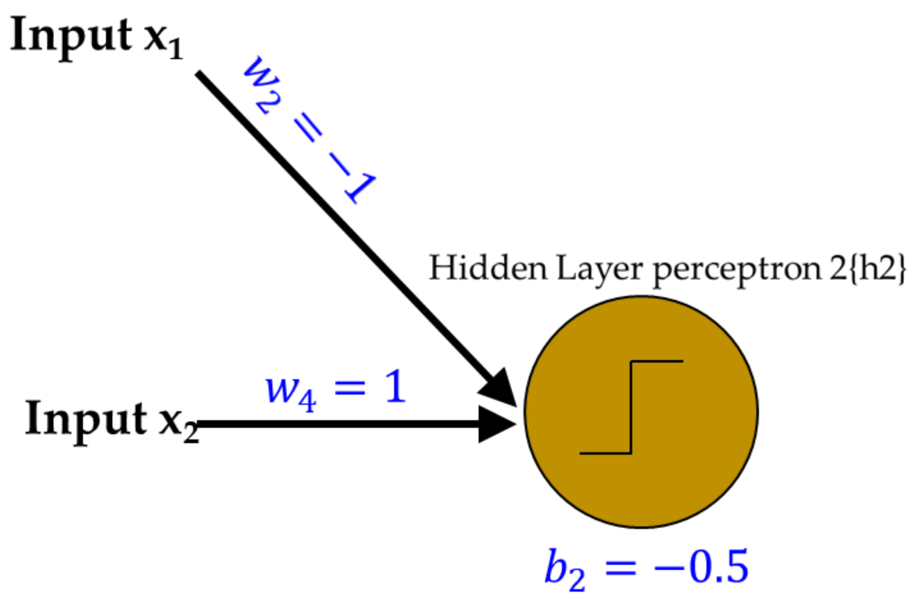
Output of h1O1

$$h1O1 = \sum_i w_i x_i + b > 0$$
$$= 1 \times 0 + (-1) \times 0 + (-0.5) > 0$$
$$= 0 + 0 - 0.5 > 0 = -0.5 > 0$$

False Thus assigned class: h1o1 = 0

Solving for First Input Row @h2

x_1	x_2	x_1 xor x_2
0	0	0

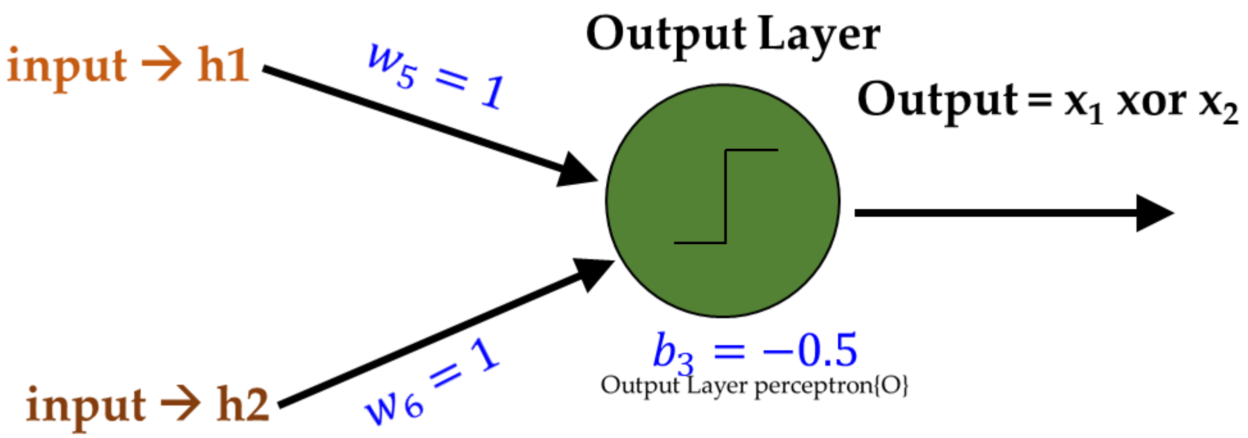


Your Turn to solve.

?

Solving for First Input Row @O

x_1	x_2	x_1 xor x_2
0	0	0



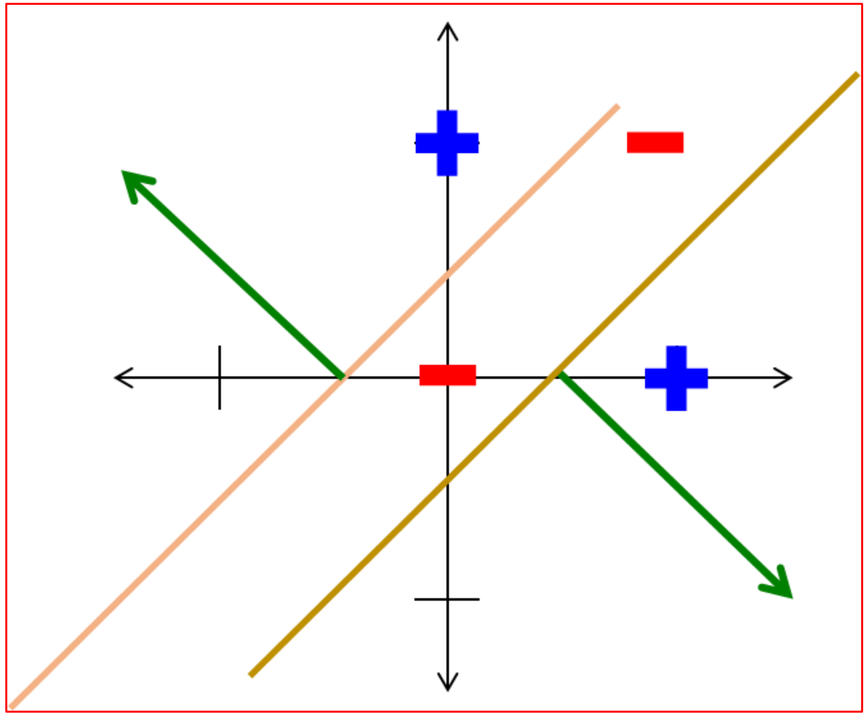
Your Turn to solve.

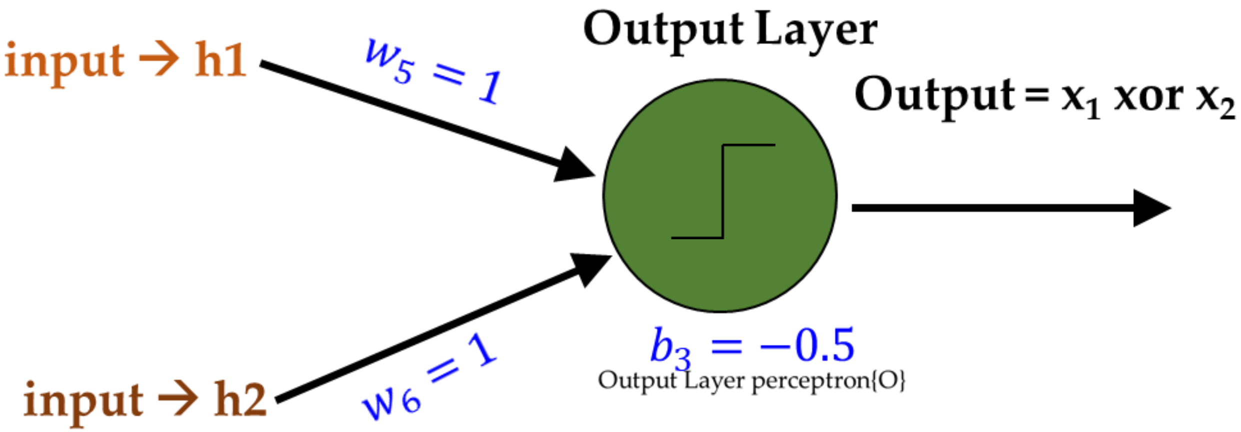
?

Did it Work? If Yes what type of Decision Boundary it Created?

Solving for First Input Row @O

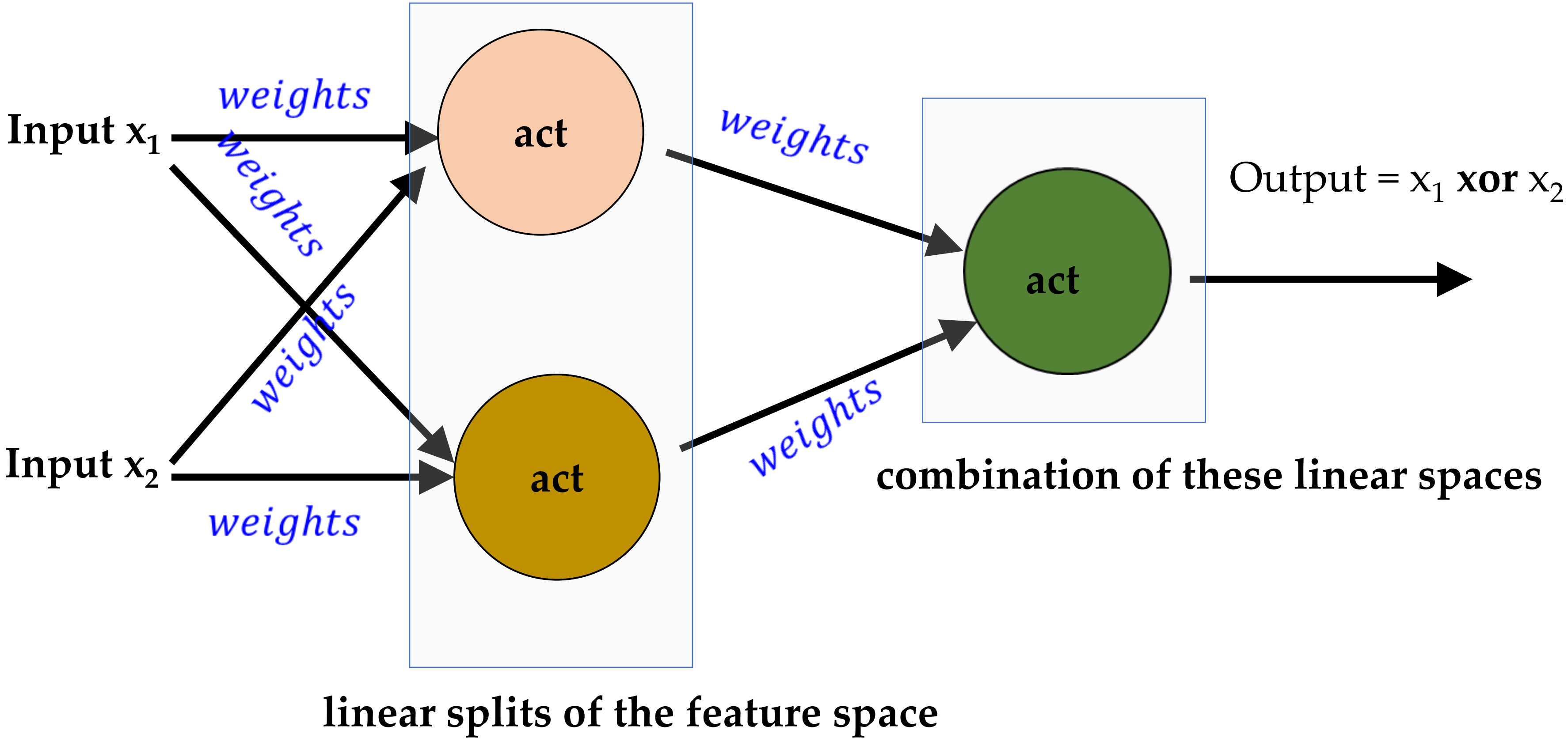
x_1	x_2	x_1 xor x_2
0	0	0

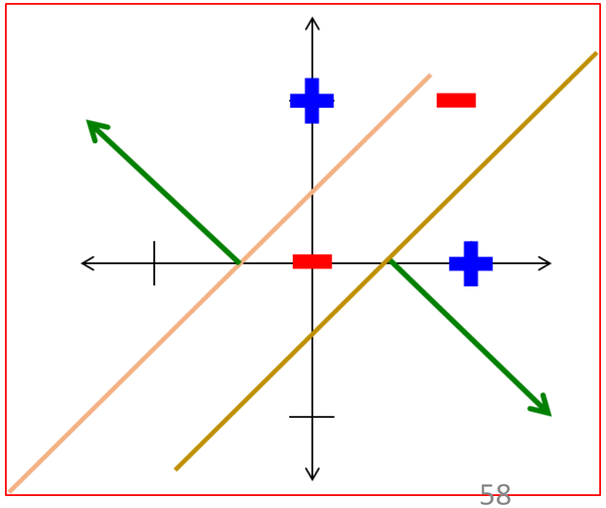




Can you solve for all the inputs. Also try for three inputs.

Decision Boundary for MLP:





Home Work – Now Solve with Sigmoid Neuron.

x_1	x_2	x_1 xor x_2
0	0	0
0	1	1
1	0	1
1	1	0