Day 5: Learning Rules
• Error-Correction Learning • Memory-Based Learning • Hebbian Learning

---

Error-Correction Learning

A model:

- **Makes a prediction**
- **Checks if it is correct**
- **If wrong → corrects itself**
- **If right → does nothing**

The model learns by **reducing its mistakes** that's why it's called **error-correction**.

What is "Error"?

Error means the **difference between what we wanted and what we got**.

- $$\text{Error} = \text{Target} - \text{Output}$$
- If error = 0 → prediction is perfect
- If error ≠ 0 → model must learn

Key Formula

$$\Delta w = \eta(\text{Target} - \text{Output})x$$

Meaning of Each Term (Very Simple)

Symbol	Meaning
Δw	How much to change the weight
η	Learning speed (small value like 0.1)
Target	Correct answer
Output	Model's answer
x	Input value

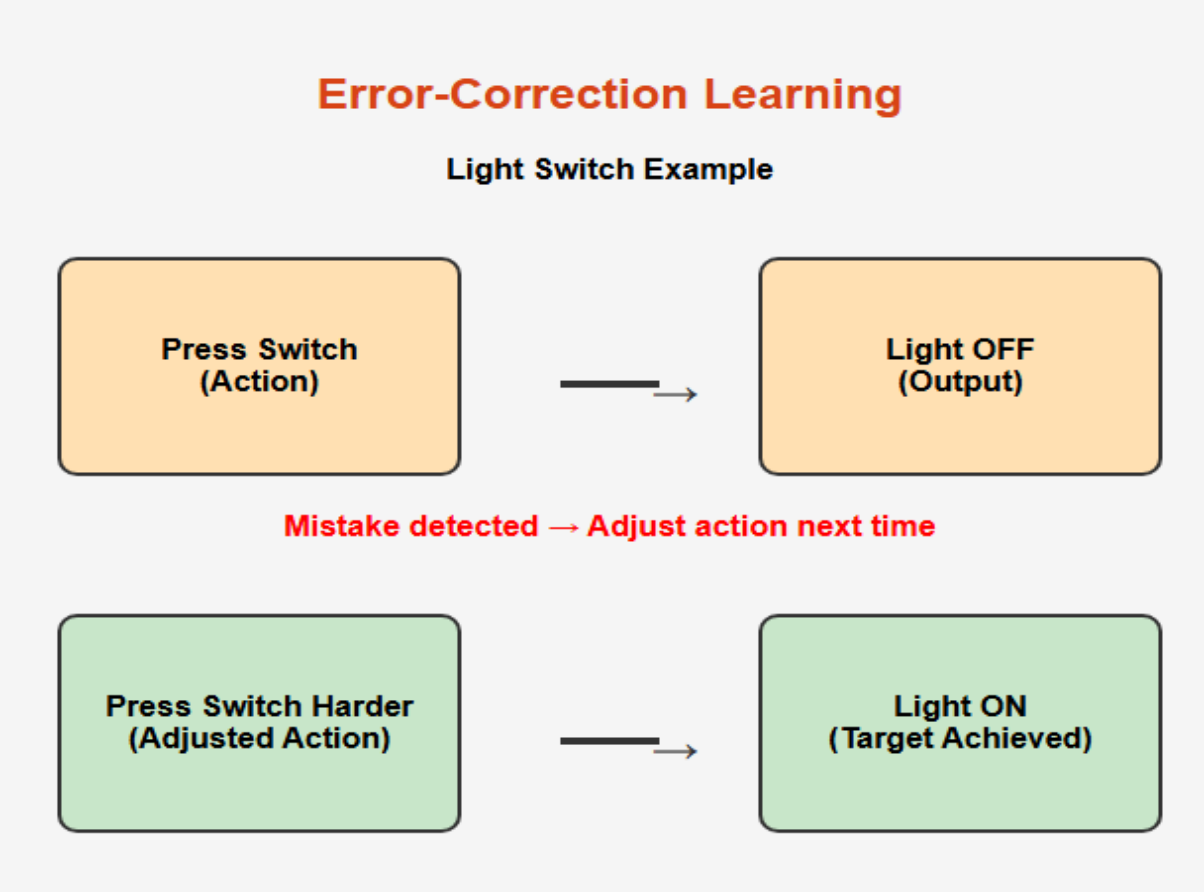
👉 **Weight change depends on how wrong the model is**

Example: Learning to Switch On a Light

Situation

You want to turn on a light using a switch.

- **Target (desired result):** Light ON (1)
- **Output (what happens):** Light OFF (0)



What the System Does

1. You press the switch
2. Light does **not** turn on → **mistake**
3. Next time, you press harder or adjust the switch

You **change your action because the result was wrong**

This is **error-correction learning**.

Mapping to Learning Terms

Real World	Learning Term
Want light ON	Target
Light OFF	Output
Adjust switch	Change weight
Press harder	Increase weight
Press softer	Decrease weight

Learning Room Temperature Control

Imagine you have a **smart heater** that tries to maintain a room temperature of **22°C**.

Step 1: Define the problem

- **Target temperature:** 22°C
- **Current room temperature (Output):** 20°C
- **Input:** How much heating the heater applies (we'll call it x=1)
- **Learning rate:** η=0.1

Step 2: Calculate the error

$$\text{Error} = \text{Target} - \text{Output} = 22 - 20 = 2$$

Step 3: Update the "weight" (heater power)

$$\Delta w = \eta \times \text{Error} \times x$$

$$\Delta w = 0.1 \times 2 \times 1 = 0.2$$

Step 4: Apply the change

- Old heater setting (weight) = 5 units
- New heater setting = 5 + 0.2 = **5.2 units**

The heater will now give **slightly more heat**, bringing the room closer to the target temperature.

Step 5: Next iteration

- Suppose after applying 5.2 units, the room temperature rises to 21°C.
- Error = 22 - 21 = 1
- $\Delta w = 0.1 \times 1 \times 1 = 0.1$
- New heater setting = 5.2 + 0.1 = **5.3 units**

The heater **gradually learns** the correct power to reach exactly 22°C.

Summary Table (Iterations)

Iteration	Room Temp (Output)	Error	Δw	New Heater Setting
1	20	2	0.2	5.2
2	21	1	0.1	5.3
3	21.7	0.3	0.03	5.33

- The heater **learns from mistakes** (error between target and actual temperature)
- **Small adjustments** are made each time
- Over time, it **converges** to the desired temperature

```
# Error-Correction Learning: Room Temperature Control

# Parameters
target_temp = 22.0      # Target temperature in Celsius
room_temp = 20.0        # Initial room temperature
heater_setting = 5.0    # Initial heater power (weight)
learning_rate = 0.1     # η
x = 1                   # Input factor (how strongly heater affects temp)

# Simulation parameters
max_iterations = 20     # Maximum steps to run

print("Step | Room Temp | Error | ΔHeater | New Heater Setting")
```

```
print("-"*50)

for step in range(1, max_iterations+1):
    # Step 1: Calculate error
    error = target_temp - room_temp

    # Step 2: Update heater setting (Δw = η * error * x)
    delta_w = learning_rate * error * x
    heater_setting += delta_w

    # Step 3: Approximate new room temperature
    # Using scaling factor to show effect of heater (for simulation)
    room_temp += delta_w * 5

    # Stop if room temperature is close enough to target
    if abs(target_temp - room_temp) < 0.01:
        room_temp = target_temp

    # Display step results
    print(f"{step:^4} | {room_temp:^9.2f} | {error:^5.2f} | {delta_w:^7.2f} |
{heater_setting:^16.2f}")

    # Stop simulation if target is reached
    if room_temp == target_temp:
        break
```

Program 1: Throwing a Paper into a Trash Can

- **# Parameters**
- **target_distance = 3.0 # meters**
- **throw_strength = 2.0 # initial throw strength**
- **learning_rate = 0.2 # eta**
- **input_strength = 1.0 # x**
- **num_throws = 30 # number of throws to simulate**

Step 1: Error = 3 − 2 = 1

Step 2: Δw = 0.2 × 1 × 1 = 0.2

Step 3: New throw strength = 5 + 0.2 = 5.2

Step 4: Next throw → distance approaches 3 m

Program 2: Pouring Water into a Cup

- **# Parameters**
- **target_volume = 250.0 # ml**
- **hand_tilt = 10.0 # initial hand tilt units**
- **learning_rate = 0.3 # eta**
- **input_tilt = 1.0 # x**
- **num_pours = 30 # number of pours to simulate**
- **tolerance = 0.1 # consider volume correct if within 0.1 ml**

Step 1: Error = 250 − 200 = 50

Step 2: Δw = 0.1 × 50 × 1 = 5

Step 3: New hand tilt = 10 + 5 = 15 units

Step 4: Next pour → volume approaches 250 ml

Program 3: Setting Alarm Volume

- **# Parameters**
- **target_volume = 80.0 # target alarm volume**
- **current_volume = 60.0 # initial volume**
- **learning_rate = 0.3 # eta**
- **input_knob = 1.0 # x**
- **num_adjustments = 30 # number of adjustments**
- **tolerance = 0.01 # stopping condition**

Step 1: Error = 80 − 60 = 20

Step 2: Δw = 0.1 × 20 × 1 = 2

Step 3: New volume = 60 + 2 = 62

Step 4: Next adjustment → volume gradually reaches 80

Program 4: Balancing a Bicycle

- **# Parameters**
- **target_angle = 0.0 # perfectly upright**
- **current_angle = 10.0 # initial tilt**
- **learning_rate = 0.3 # eta**
- **input_adjustment = 1.0 # x**
- **num_steps = 30**
- **tolerance = 0.01**

Step 1: Error = 0 − 10 = −10

Step 2: Δw = 0.3 × (−10) × 1 = −3

Step 3: New handle adjustment = 5 − 3 = 2

Step 4: Next adjustment → bike becomes more upright

Program 5: Shooting Basketball Free Throw

- **# Parameters**
- **target_score = 1.0 # want to make the basket**
- **current_score = 0.0 # missed shot**
- **shooting_force = 5.0 # initial shooting force**
- **learning_rate = 0.1 # eta**
- **input_force = 1.0 # x**
- **num_shots = 5 # number of attempts**
- **tolerance = 0.01 # stopping condition**

Step 1: Error = 1 − 0 = 1
Step 2: Δw = 0.5 × 1 × 1 = 0.5
Step 3: New shooting force = 5 + 0.5 = 5.5
Step 4: Next throw → improves chance of scoring

Memory-Based Learning

Basic Idea (Very Simple)

In **memory-based learning**, the system:

- **Stores past examples**
- **Does not learn weights or rules**
- **Uses memory to make decisions**

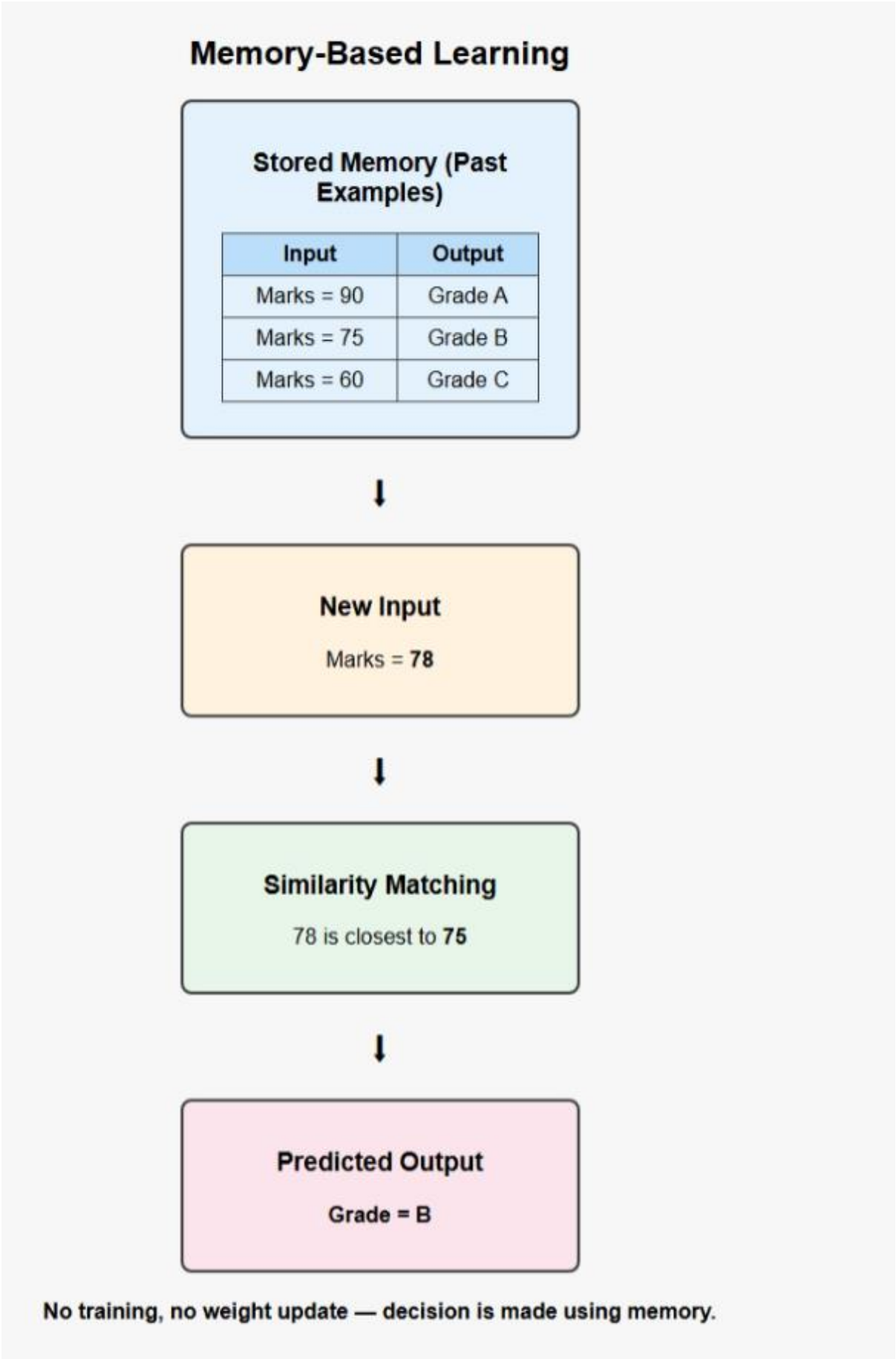
Key Characteristics

- No training phase
- Stores all examples in memory
- Decision is made at **query time**
- Uses **similarity** between new input and stored data

Memory-Based Learning is a learning method where the system stores previous examples and predicts the output of a new input by comparing it with stored data.

How Memory-Based Learning Works

1. Store training data
2. Receive a new input
3. Find the **most similar stored example**
4. Use its output as the prediction


Memory-Based Learning
Stored Memory (Past Examples)
Input
Output
Marks = 90
Grade A
Marks = 75
Grade B
Marks = 60
Grade C
New Input
Marks = 78
Similarity Matching
78 is closest to 75
Predicted Output
Grade = B
No training, no weight update — decision is made using memory.

Student Marks → Grade Prediction

Step 1: Stored Memory (Past Examples)

The system already has **past data stored in memory**:

Marks	Grade
90	A
75	B
60	C

This is the **memory**.

No learning or training happens here.

The system just **stores the data as it is**.

Step 2: New Input

A new student's marks are given:

- **Marks = 78**

The system has **never seen 78 before**, so it must use memory to decide.

Step 3: Similarity Matching

The system now **compares 78 with all stored marks**:

- Distance from 90 → $|78 - 90| = 12$
- Distance from 75 → $|78 - 75| = 3$
- Distance from 60 → $|78 - 60| = 18$

✔ The **closest value is 75**

Step 4: Output Prediction

- Stored grade for **75** is **B**
- So, the system predicts:

Grade = B

The system assumes:

“Since 78 is closest to 75, the grade should be the same.”

Important Points to Understand

- □ No formula is learned
- □ No weights areupdated
- □ No error calculation

✔ Decision is made **only by memory and similarity**

Why This Is Memory-Based Learning

- The system **remembers past examples**
- It **compares new input with stored data**
- It **uses the closest match to predict output**

Memory-Based Learning	Description
Advantages	- Simple to understand - No training time - Works well with small data
Disadvantages	- Requires large memory - Slow for large datasets - Sensitive to noise

Comparison with Error-Correction Learning

Feature	Error-Correction	Memory-Based
Learns weights	Yes	No
Uses error	Yes	No
Stores data	No	Yes
Example	Heater control	Grade Prediction

Memory-Based Learning learns by remembering past examples and making decisions based on similarity.

```
# Memory-Based Learning Example
# Student Marks -> Grade Prediction

# Step 1: Stored memory (past examples)
memory = {
    90: "A",
    75: "B",
    60: "C"
}

# Step 2: New input
new_marks = 78

# Step 3: Find the closest marks in memory
closest_marks = None
minimum_distance = float('inf')

for marks in memory:
    distance = abs(new_marks - marks)
    if distance < minimum_distance:
        minimum_distance = distance
        closest_marks = marks

# Step 4: Predict output
predicted_grade = memory[closest_marks]

# Display result
```

```
print("Memory-Based Learning Example")
print("-----------------------------")
print("Stored Data:", memory)
print("New Marks:", new_marks)
print("Closest Stored Marks:", closest_marks)
print("Predicted Grade:", predicted_grade)
```

Practice Question 1: Nearest Number Finder

Problem:

You are given a list of stored numbers:

[10, 20, 30, 40, 50]

Write a Python program to:

- Take a new number as input
- Find the **closest number** from the list

Practice Question 2: Marks to Grade Prediction

Problem:

Stored data:

Marks	Grade
90	A
75	B
60	C

Write a Python program that:

- Stores this data
- Takes a student's marks as input
- Predicts the grade using **nearest match**

Practice Question 3: Simple k-NN (k = 1)

Problem:

Stored points:

(2,3) → A, (5,4) → B, (2,4) → A

Write a Python program to:

- Take a new point (x, y)
- Find the **nearest point**
- Output its class

Practice Question 4: Phone Contact Search

Problem:

You have stored phone contacts:

Number	Name
9876543210	Ravi
9123456789	Neha

Write a Python program that:

- Takes a phone number as input
- Displays the contact name if it exists
- Displays **"Contact Not Found"** otherwise

Practice Question 5: Movie Recommendation

Problem:

Stored user preferences:

User	Genre
A	Action
B	Comedy
C	Action

Write a Python program that:

- Takes a user's favorite genre as input
- Recommends a movie watched by users with the **same preference**

Hebbian Learning

Hebbian Learning is based on the principle:

“Neurons that fire together, wire together.”

This means:

- If an input neuron and an output neuron are active at the same time,
- The connection (weight) between them becomes stronger.

Learning happens due to simultaneous activation, not due to error.

Hebbian Learning is an unsupervised learning rule in which the connection strength between two neurons increases when both neurons are active simultaneously.

Important Characteristics

- **Unsupervised learning**
- No target output required
- No error calculation
- Weight increases when neurons fire together
- Based on biological learning in the brain

Hebbian Learning Rule

Mathematical Formula

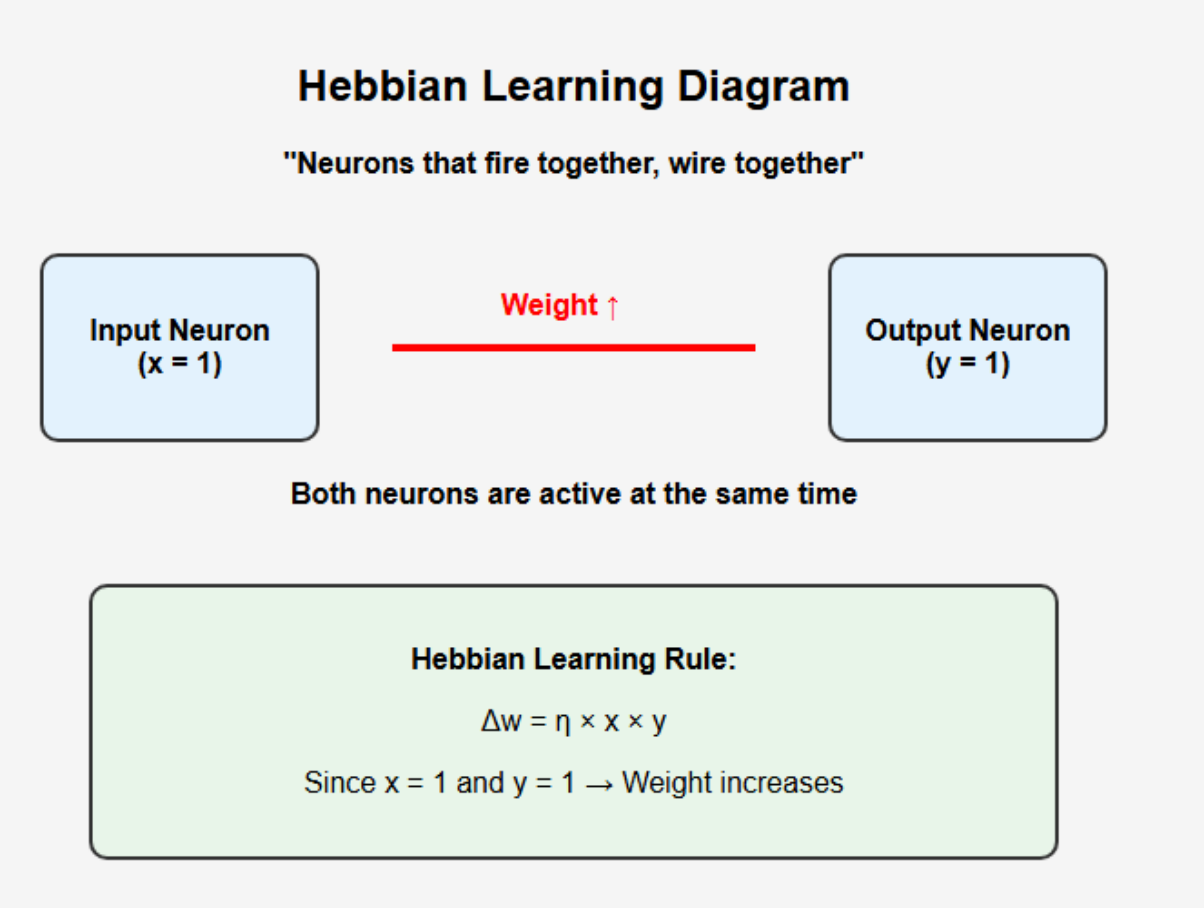
$$\Delta w = \eta\, x\, y$$

Where:

- Δw = change in weight
- η = learning rate
- x = input neuron activation
- y = output neuron activation

✔ If both x and y are active (1), weight increases

✔ If either is 0, no learning happens



Example1

Child Learning Fire is Hot

- Touch fire → pain
- Fire (input) + pain (output) occur together
- Brain links fire with pain
- Next time, child avoids fire

Association formed due to simultaneous activity.

Example 2

Given:

- Learning rate $\eta = 0.1$
- Input $x = 1$
- Output $y = 1$
- Initial weight $w = 0.5$

Calculation:

$$\Delta w = 0.1 \times 1 \times 1 = 0.1$$

Updated weight:

$$w_{new} = 0.5 + 0.1 = 0.6$$

✔ Weight increased because both neurons were active.

When One Neuron is Inactive

Given:

- $\eta = 0.1$
- $x = 1$
- $y = 0$

Calculation:

$$\Delta w = 0.1 \times 1 \times 0 = 0$$

✔ Weight does **not change**

✔ No learning occurs

Step-by-Step Numeric Table

Input (x)	Output (y)	Δw	Weight Update
1	1	+0.1	Weight increases
1	0	0	No change
0	1	0	No change
0	0	0	No change

Advantages of Hebbian Learning

✔ Simple and biologically inspired

✔ No labeled data needed

✔ Good for pattern association

Disadvantages of Hebbian Learning

 Weights can grow indefinitely

 No mechanism to reduce error

 Not suitable for precise prediction

Comparison with Other Learning Rules

Feature	Hebbian Learning
Learning type	Unsupervised
Uses error	No
Needs target	No
Weight update	Based on co-activation

Hebbian Learning strengthens the connection between neurons when they are activated simultaneously, based on the principle "neurons that fire together, wire together."

```
# Hebbian Learning Example

# Initial parameters
learning_rate = 0.1     # η
weight = 0.5            # Initial weight

# Training data (input x, output y)
# 1 = active, 0 = inactive
training_data = [
    (1, 1),
    (1, 1),
    (1, 0),
    (0, 1),
    (1, 1)
]

print("Hebbian Learning")
print("----------------------------")
print("Initial Weight:", weight)
print()

# Hebbian learning process
for i, (x, y) in enumerate(training_data, start=1):
    delta_w = learning_rate * x * y
    weight += delta_w

    print(f"Step {i}")
    print(f"Input (x) = {x}, Output (y) = {y}")
    print(f"Δw = {learning_rate} × {x} × {y} = {delta_w}")
    print(f"Updated Weight = {weight}")
    print("----------------------------")

print("Final Weight after Hebbian Learning:", weight)
print("----------------------------")

# Test new input (simple version)
test_x = int(input("Enter test input x (0 or 1): "))
test_y = weight * test_x
print(f"Predicted output y = weight × x = {weight} × {test_x} = {test_y}")
```

1. Study → Good Marks Association

Problem:

A student studies daily, and sometimes gets good marks.

- Input: Studied today (1 = yes, 0 = no)
- Output: Got good marks (1 = yes, 0 = no)
- Learning rate: 0.1

```
# Parameters
learning_rate = 0.1
weight = 0.0  # initial association weight

# Training data: (studied, good_marks)
# 1 = yes, 0 = no
data = [
    (1, 1),
    (1, 0),
    (0, 1),
    (1, 1),
    (0, 0)
]
```

Task:

- Simulate 5 days of data
- Update the "association weight" using **Hebbian learning**
- Print the weight after each day
- Display the predicted result and interpret whether the chance of getting good marks is high or low.

2. Exercise → Energy Level

Problem:

A person exercises and feels energetic.

- Input: Exercise done today (1 = yes, 0 = no)
- Output: Energy felt (1 = yes, 0 = no)
- Learning rate: 0.2

```
# Parameters
learning_rate = 0.2
weight = 0.0  # initial association weight

# Simulate 7 days of data (input, output)
# 1 = yes, 0 = no
daily_data = [
    (1, 1),  # exercised, felt energetic
    (0, 1),  # no exercise, felt energetic
    (1, 0),  # exercised, no energy
    (1, 1),  # exercised, felt energetic
    (0, 0),  # no exercise, no energy
    (1, 1),  # exercised, felt energetic
    (0, 1)   # no exercise, felt energetic
]
```

Task:

- Simulate daily experiences for 7 days
- Use Hebbian learning to update the connection
- Print daily updates and final association strength
- Allow the user to enter a new input (0 or 1) and predict the energy level

3. Song → Happiness

Problem:

A person hears a song, and sometimes feels happy.

- Input: Heard the song (1 = yes, 0 = no)
- Output: Felt happy (1 = yes, 0 = no)

```
# Parameters
learning_rate = 0.15   # learning rate
weight = 0.0           # initial association weight

# 6 experiences (input, output)
# 1 = yes, 0 = no
experiences = [
    (1, 1),  # heard song, felt happy
    (1, 0),  # heard song, not happy
    (0, 1),  # didn't hear song, felt happy
    (1, 1),  # heard song, felt happy
    (0, 0),  # didn't hear song, not happy
    (1, 1)   # heard song, felt happy
]
```

Task:

- Store 6 experiences
- Use Hebbian learning to strengthen the connection
- Show how the association weight increases only when **both happen**
- Allows the user to test a new input and predicts happiness.

4. Coffee → Alertness

Problem:

Drinking coffee can make a person alert.

- Input: Coffee consumed (1 = yes, 0 = no)
- Output: Felt alert (1 = yes, 0 = no)
- Learning rate: 0.15

```
# Parameters
learning_rate = 0.15
weight = 0.0  # initial association weight

# 5 days of experiences (coffee, alertness)
# 1 = yes, 0 = no
daily_experiences = [
    (1, 1),  # drank coffee, felt alert
    (1, 0),  # drank coffee, not alert
    (0, 1),  # no coffee, felt alert
    (1, 1),  # drank coffee, felt alert
    (0, 0)   # no coffee, not alert
]
```

Task:

- Take 5 days of experience
- Update the weight according to Hebbian learning
- Print the updated weight each day
- User can test a new situation after learning

5. Traffic Light → Go Action

Problem:

A person sees a green light and presses the accelerator.

- Input: Green light (1 = yes, 0 = no)
- Output: Pressed accelerator (1 = yes, 0 = no)
- Learning rate: 0.1

```
# Parameters
learning_rate = 0.1
weight = 0.0  # initial association weight

# 5 trials of experiences (green light, pressed accelerator)
# 1 = yes, 0 = no
trials = [
    (1, 1),  # green light, pressed accelerator
    (1, 0),  # green light, did not press
    (0, 1),  # not green light, pressed accelerator
    (1, 1),  # green light, pressed accelerator
    (0, 0)   # not green light, did not press
]
```

Task:

- Simulate 5 trials of traffic observations
- Update the association weight
- Print each trial and final connection strength
- User can test a new traffic light situation

```python
# Paper throwing simulation with Error-Correction Learning

# Parameters
target_distance = 3.0       # meters
throw_strength = 2.0        # initial throw strength
learning_rate = 0.2         # eta
input_strength = 1.0        # x
num_throws = 30             # number of throws to simulate

print("Throw # | Throw Strength | Distance | Error | Updated?")
print("-----------------------------------------------------------")

for throw in range(1, num_throws + 1):
    # Make a prediction (throw)
    distance = throw_strength * input_strength

    # Calculate error and round to 2 decimals
    error = round(target_distance - distance, 2)

    # Check if the prediction is correct
    if error != 0:
        # Correct itself (update throw strength)
        throw_strength += learning_rate * error * input_strength
        updated = "Yes"
    else:
        updated = "No"

    # Print the results using the distance before update
    print(f"{throw:^7}| {throw_strength:^14.2f}| {distance:^8.2f}|
{error:^5.2f}| {updated:^8}")
```

```python
# Pouring Water simulation with Error-Correction Learning

# Parameters
target_volume = 250.0       # ml
hand_tilt = 10.0            # initial hand tilt units
learning_rate = 0.3         # eta
input_tilt = 1.0            # x
num_pours = 30            # number of pours to simulate
tolerance = 0.1             # consider volume correct if within 0.1 ml

print("Pour # | Hand Tilt | Poured Volume | Error | Updated?")
print("-----------------------------------------------------------")

for pour in range(1, num_pours + 1):
    # Prediction: amount poured proportional to hand tilt
    poured_volume = hand_tilt * input_tilt  # simplified linear model
    error = round(target_volume - poured_volume, 2)

    # Check if the prediction is correct
    if abs(error) > tolerance:
        hand_tilt += learning_rate * error * input_tilt  # adjust hand tilt
        updated = "Yes"
    else:
        updated = "No"

    # Print results
    print(f"{pour:^7}| {hand_tilt:^10.2f}| {poured_volume:^13.2f}|
{error:^5.2f}| {updated:^8}")
```

```python
# Setting Alarm Volume using Error-Correction Learning

# Parameters
target_volume = 80.0        # target alarm volume
current_volume = 60.0       # initial volume
learning_rate = 0.3         # eta
input_knob = 1.0            # x
num_adjustments = 30        # number of adjustments
tolerance = 0.01            # stopping condition

print("Step | Current Volume | Error | Updated?")
print("-----------------------------------------")

for step in range(1, num_adjustments + 1):
    # Calculate error
    error = round(target_volume - current_volume, 2)

    # Error-correction rule
    if abs(error) > tolerance:
        current_volume += learning_rate * error * input_knob
        updated = "Yes"
    else:
        updated = "No"

    # Print results
    print(f"{step:^5}| {current_volume:^15.2f}| {error:^6.2f}| {updated:^8}")
```

```python
# Balancing a Bicycle using Error-Correction Learning

# Parameters
target_angle = 0.0        # perfectly upright
current_angle = 10.0      # initial tilt
learning_rate = 0.3      # eta
input_adjustment = 1.0   # x
num_steps = 30
tolerance = 0.01

print("Step | Angle | Error | Updated?")
print("----------------------------------")

for step in range(1, num_steps + 1):
    # Calculate error
    error = round(target_angle - current_angle, 2)

    # Error-correction rule
    if abs(error) > tolerance:
        current_angle += learning_rate * error * input_adjustment
        updated = "Yes"
    else:
        updated = "No"

    print(f"{step:^5}| {current_angle:^7.2f}| {error:^6.2f}| {updated:^8}")
```

```python
# Shooting Basketball Free Throw using Error-Correction Learning

# Parameters
target_score = 1.0          # want to make the basket
current_score = 0.0         # missed shot
shooting_force = 5.0        # initial shooting force
learning_rate = 0.1         # eta
input_force = 1.0           # x
num_shots = 5               # number of attempts
tolerance = 0.01            # stopping condition

print("Shot | Shooting Force | Score | Error | Updated?")
print("-------------------------------------------------")

for shot in range(1, num_shots + 1):
    # Prediction: score depends on shooting force
    score = current_score

    # Calculate error
    error = round(target_score - score, 2)

    # Error-correction learning rule
    if abs(error) > tolerance:
        shooting_force += learning_rate * error * input_force
        # Improved force increases chance of scoring
        current_score += 0.2 * shooting_force
        if current_score > 1:
            current_score = 1  # cap at success
        updated = "Yes"
    else:
        updated = "No"

    print(f"{shot:^5}| {shooting_force:^15.2f}| {score:^6.2f}| {error:^6.2f}|
{updated:^8}")
```

```python
# Nearest Number Finder using Memory-Based Learning

# Memory (stored numbers)
memory = [10, 20, 30, 40, 50]

# Take input from user
new_number = int(input("Enter a number: "))

# Initialize nearest number
nearest = memory[0]
min_distance = abs(new_number - nearest)

# Compare with all stored numbers (memory-based learning)
for number in memory:
    distance = abs(new_number - number)

    if distance < min_distance:
        min_distance = distance
        nearest = number

# Output result
print("Stored numbers:", memory)
print("Input number:", new_number)
print("Nearest number:", nearest)
```

```python
# Marks to Grade Prediction using Memory-Based Learning

# Stored data (memory)
memory = {
    90: 'A',
    75: 'B',
    60: 'C'
}

# Take input from user
student_marks = int(input("Enter student marks: "))

# Initialize nearest match
nearest_marks = None
min_distance = float('inf')

# Find nearest marks
for marks in memory:
    distance = abs(student_marks - marks)

    if distance < min_distance:
        min_distance = distance
        nearest_marks = marks

# Predict grade
predicted_grade = memory[nearest_marks]

# Output result
print("Stored data:", memory)
print("Student marks:", student_marks)
print("Predicted grade:", predicted_grade)
```

```python
# Simple k-NN (k = 1) using Memory-Based Learning

import math

# Memory: stored points with class labels
memory = [
    ((2, 3), 'A'),
    ((5, 4), 'B'),
    ((2, 4), 'A')
]

# Take input for new point
x = float(input("Enter x value: "))
y = float(input("Enter y value: "))

# Initialize nearest neighbor
nearest_class = None
min_distance = float('inf')

# Compare new point with stored points
for (px, py), label in memory:
    distance = math.sqrt((x - px) ** 2 + (y - py) ** 2)

    if distance < min_distance:
        min_distance = distance
        nearest_class = label

# Output result
print("New point:", (x, y))
print("Predicted class:", nearest_class)
```

```python
# Phone Contact Search using Memory-Based Learning

# Memory: stored contacts
contacts = {
    "9876543210": "Ravi",
    "9123456789": "Neha"
}

# Take phone number as input
phone_number = input("Enter phone number: ")

# Search in memory
if phone_number in contacts:
    print("Contact Name:", contacts[phone_number])
else:
    print("Contact Not Found")
```

```python
# Movie Recommendation using Memory-Based Learning

# Memory: stored user preferences
user_preferences = {
    "A": "Action",
    "B": "Comedy",
    "C": "Action"
}

# Sample movies watched by users
movies_watched = {
    "A": "Mad Max",
    "B": "The Mask",
    "C": "Avengers"
}

# Take input from user
favorite_genre = input("Enter your favorite genre: ")

# Find users with the same preference
recommendations = []
```

```
for user, genre in user_preferences.items():
    if genre.lower() == favorite_genre.lower():
        recommendations.append(movies_watched[user])

# Output recommendation
if recommendations:
    print("Recommended movies based on your preference:")
    for movie in recommendations:
        print("-", movie)
else:
    print("No recommendation found for this genre")
```

```
# Hebbian Learning: Study -> Good Marks

# Parameters
learning_rate = 0.1
weight = 0.0  # initial association weight

# Training data: (studied, good_marks)
# 1 = yes, 0 = no
data = [
    (1, 1),
    (1, 0),
    (0, 1),
    (1, 1),
    (0, 0)
]

print("Hebbian Learning : Study -> Good Marks")
print("-------------------------------------------------")
print("Day | Studied | Good Marks | Weight")
print("-----------------------------------")

# Training phase
for day, (studied, good_marks) in enumerate(data, start=1):
    delta_w = learning_rate * studied * good_marks
    weight += delta_w
    print(f"{day:^3} | {studied:^7} | {good_marks:^10} | {weight:.2f}")

print("-----------------------------------")
print(f"Final Learned Weight: {weight:.2f}")
print("-----------------------------------")

# Testing phase
```

```
print("\nTesting New Input")
test_input = int(input("Did the student study? (1 = Yes, 0 = No): "))

# Prediction using learned weight
predicted_output = weight * test_input

print(f"\nPredicted Good Marks Value = {weight:.2f} × {test_input} =
{predicted_output:.2f}")

# Optional interpretation
if predicted_output > 0:
    print("Prediction: Higher chance of getting good marks.")
else:
    print("Prediction: Low chance of getting good marks.")
```

```
# Hebbian Learning : Exercise -> Energy Level

# Parameters
learning_rate = 0.2
weight = 0.0  # initial association weight

# Simulate 7 days of data (input, output)
# 1 = yes, 0 = no
daily_data = [
    (1, 1),  # exercised, felt energetic
    (0, 1),  # no exercise, felt energetic
    (1, 0),  # exercised, no energy
    (1, 1),  # exercised, felt energetic
    (0, 0),  # no exercise, no energy
    (1, 1),  # exercised, felt energetic
    (0, 1)   # no exercise, felt energetic
]

print("Hebbian Learning: Exercise -> Energy Level")
print("-------------------------------------------------")
print("Day | Exercise | Energy | Weight")
print("----------------------------------")

# Training phase
for day, (exercise, energy) in enumerate(daily_data, start=1):
    delta_w = learning_rate * exercise * energy
    weight += delta_w

    print(f"{day:^3} | {exercise:^8} | {energy:^6} | {weight:.2f}")
```

```
print("---------------------------------")
print("Final association weight:", round(weight, 2))
print("---------------------------------")

# Testing phase
print("\nTesting New Input")
test_input = int(input("Did the person exercise? (1 = Yes, 0 = No): "))

predicted_energy = weight * test_input

print(f"\nPredicted Energy Value = {weight:.2f} × {test_input} =
{predicted_energy:.2f}")

# Simple interpretation
if predicted_energy > 0:
    print("Prediction: Higher chance of feeling energetic.")
else:
    print("Prediction: Low chance of feeling energetic.")
```

```
# Hebbian Learning Simulation: Song -> Happiness

# Parameters
learning_rate = 0.15   # learning rate
weight = 0.0           # initial association weight

# 6 experiences (input, output)
# 1 = yes, 0 = no
experiences = [
    (1, 1),  # heard song, felt happy
    (1, 0),  # heard song, not happy
    (0, 1),  # didn't hear song, felt happy
    (1, 1),  # heard song, felt happy
    (0, 0),  # didn't hear song, not happy
    (1, 1)   # heard song, felt happy
]

print("Hebbian Learning: Song -> Happiness")
print("-----------------------------------------------")
print("Exp | Heard Song | Felt Happy | Weight")
print("---------------------------------------")

# Training phase
for i, (heard, happy) in enumerate(experiences, start=1):
```

```
    delta_w = learning_rate * heard * happy
    weight += delta_w

    print(f"{i:^3} | {heard:^10} | {happy:^10} | {weight:.2f}")

print("----------------------------------------")
print("Final Association Weight:", round(weight, 2))
print("----------------------------------------")

# Testing phase
print("\nTesting New Experience")
test_input = int(input("Did the person hear the song? (1 = Yes, 0 = No): "))

predicted_happiness = weight * test_input

print(f"\nPredicted Happiness Value = {weight:.2f} × {test_input} =
{predicted_happiness:.2f}")

# Simple interpretation
if predicted_happiness > 0:
    print("Prediction: Higher chance of feeling happy.")
else:
    print("Prediction: Low chance of feeling happy.")
```

```
# Hebbian Learning: Coffee -> Alertness

# Parameters
learning_rate = 0.15
weight = 0.0  # initial association weight

# 5 days of experiences (coffee, alertness)
# 1 = yes, 0 = no
daily_experiences = [
    (1, 1),  # drank coffee, felt alert
    (1, 0),  # drank coffee, not alert
    (0, 1),  # no coffee, felt alert
    (1, 1),  # drank coffee, felt alert
    (0, 0)   # no coffee, not alert
]

print("Hebbian Learning: Coffee -> Alertness")
print("--------------------------------------")
print("Day | Coffee | Alert | Weight")
print("------------------------------")
```

```
# Training phase
for day, (coffee, alert) in enumerate(daily_experiences, start=1):
    delta_w = learning_rate * coffee * alert
    weight += delta_w

    print(f"{day:^3} | {coffee:^6} | {alert:^5} | {weight:.2f}")

print("--------------------------------")
print("Final Association Weight:", round(weight, 2))
print("--------------------------------")

# Testing phase
print("\nTesting New Situation")
test_input = int(input("Did the person drink coffee? (1 = Yes, 0 = No): "))

predicted_alertness = weight * test_input

print(f"\nPredicted Alertness Value = {weight:.2f} × {test_input} =
{predicted_alertness:.2f}")

# Interpretation
if predicted_alertness > 0:
    print("Prediction: Higher chance of feeling alert.")
else:
    print("Prediction: Low chance of feeling alert.")
```

```
# Hebbian Learning : Traffic Light -> Go Action

# Parameters
learning_rate = 0.1
weight = 0.0  # initial association weight

# 5 trials of experiences (green light, pressed accelerator)
# 1 = yes, 0 = no
trials = [
    (1, 1),  # green light, pressed accelerator
    (1, 0),  # green light, did not press
    (0, 1),  # not green light, pressed accelerator
    (1, 1),  # green light, pressed accelerator
    (0, 0)   # not green light, did not press
]

print("Hebbian Learning: Traffic Light -> Go Action")
print("-----------------------------------------------")
print("Trial | Green Light | Accelerator | Weight")
```

```
print("-----------------------------------------")

# Training phase
for i, (green, accel) in enumerate(trials, start=1):
    delta_w = learning_rate * green * accel
    weight += delta_w

    print(f"{i:^5} | {green:^11} | {accel:^11} | {weight:.2f}")

print("-----------------------------------------")
print("Final Association Weight:", round(weight, 2))
print("-----------------------------------------")

# Testing phase
print("\nTesting New Situation")
test_input = int(input("Is the traffic light green? (1 = Yes, 0 = No): "))

predicted_action = weight * test_input

print(f"\nPredicted Go Action Value = {weight:.2f} × {test_input} =
{predicted_action:.2f}")

# Interpretation
if predicted_action > 0:
    print("Prediction: Likely to press the accelerator (Go).")
else:
    print("Prediction: Unlikely to press the accelerator (Stop).")
```