# CSC 591-145/ ECE 492-55 / ECE 592-106: Foundations of Generative AI for Systems

Instructor: Dr. Samira Mirbagher Ajorpaz
Department of Electrical
Computer Engineering
Email: smirbag@ncsu.edu

Spring 2025

# Objective or Description

This course provides a comprehensive exploration of AI techniques and their application in computer architecture and systems.

**Key Goals:**

- Hands-on projects engaging with foundational and advanced machine learning concepts.
- Application of generative AI to enhance system performance in areas like:
  - Final Project: A surrogate model for Computer System Processor (CPU and memory system)
  - Homeworks: A surrogate for CPU modules such as Cache replacement policies, A chat bot to act as an LLM as a performance judge and analyser.
- Focus on practical understanding and impactful solutions.
  - Trustworthiness and Interpretability of AI for systems.

# Prerequisites or willingness to self learn .

- Ability to read papers and understand novel topics.
- Proficiency in C++, Python.
- Familiar with Data Structures/Algorithms.

- **Not essential but best if you know**
  - Foundational Computer Architecture Topics.
    - (**Required background knowledge will be covered in the class.**)
  - Foundational AI, ML and Neural Network topics.
    - (**Required background knowledge will be covered in the class.**)

- Ability to work in teams.
- Interested in research base course
- Interested in project base course

# Resources and Recommended Textbook

**A collection of papers (IEEE/ACM) in ML & Computer Architecture**

**Machine Learning:**

- *Machine Learning* by Tom Mitchell.
- *Deep Learning* by Ian Goodfellow, Yoshua Bengio, and Aaron Courville.
- *Generative Deep Learning: Teaching Machines to Paint, Write, Compose, and Play*.
- *Generative AI with LangChain: Build Large Language Model (LLM) Apps*.
- *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*.

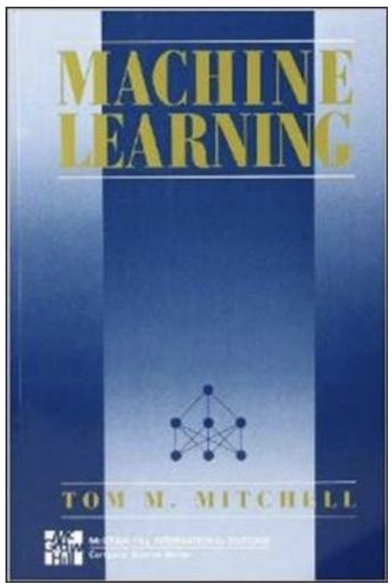
**Computer Architecture:**

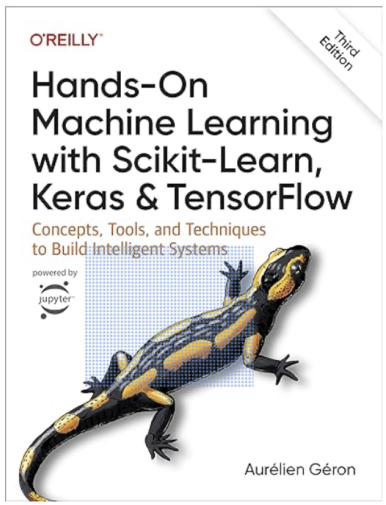
- *Modern Processor Design: Fundamentals of Superscalar Processors* by John P. Shen and Mikko Lipasti.
- *Computer Architecture: A Quantitative Approach* (Morgan Kaufmann Series).

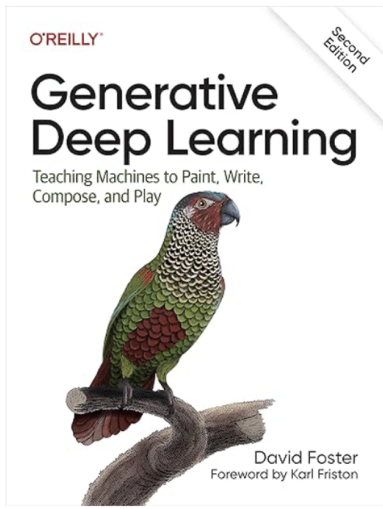
# Resources

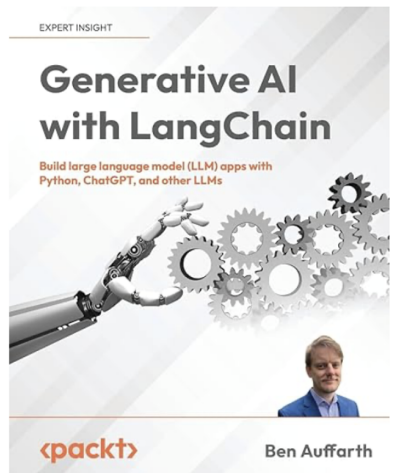
## References for ML:

- A Collection of Papers from conferences and journals.
- Machine Learning, Tom Mitchell, McGraw Hill, 1997.
- Generative Deep Learning: Teaching Machines To Paint, Write, Compose, and Play
- Generative AI with LangChain: Build large language model (LLM) apps with Python, ChatGPT and other LLMs
- Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems



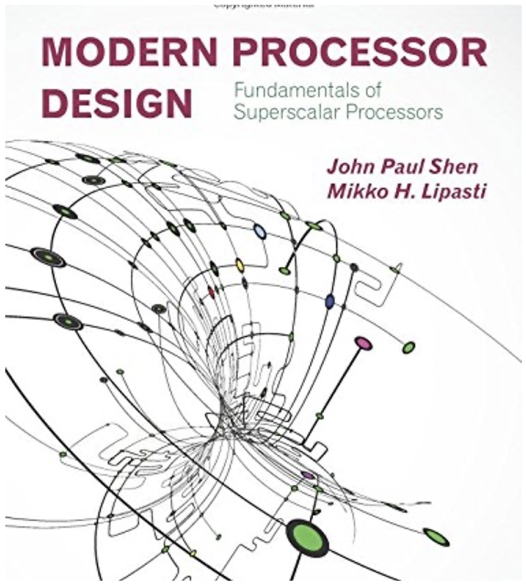



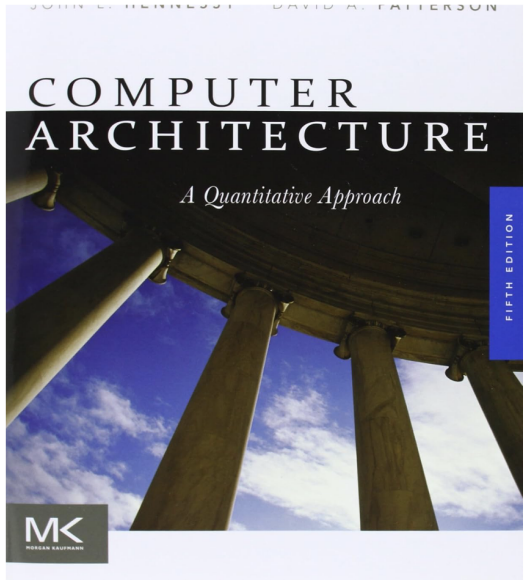



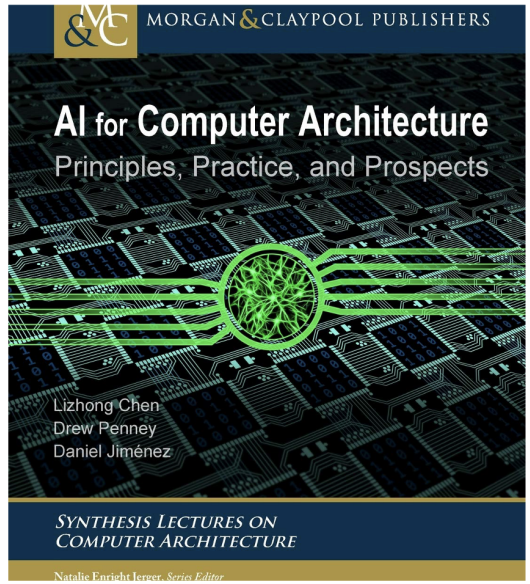
# Resources

References for Computer architecture:

- A Collection of Papers from conferences and journals.
- Modern Processor Design: Fundamentals of Superscalar Processors, John P. Shen and Mikko Lipasti, Waveland Press, Inc.
- Computer Architecture: A Quantitative Approach (The Morgan Kaufmann Series in Computer Architecture and Design) (any edition is fine)



# Topics and Content

**Machine Learning Models and Techniques:**

- Foundational Techniques:
  - Perceptron & Single-Layer Neural Networks.
  - Multi-Layer Perceptrons (MLP).
  - Recurrent Neural Networks (RNN).
  - Long Short-Term Memory Networks (LSTM).
- Advanced Generative Models:
  - Large Language Models (LLMs).
  - Generative Adversarial Networks (GANs).
  - Variational Autoencoders (VAEs).
  - Genetic Algorithm.
- Trustworthiness and Interpretability:
  - Attention mechanisms.
  - Emergent abilities in LLMs.

# Grading

**Grading Components:**

- **ECE 592 and CSC 591**
  - Homework 1: 8% (Individual)
  - Homework 2: 12% (Individual)
  - Class Presentation: 10% (Team, 1-2 paper)
  - Quizzes: 15% (Individual)
  - Midterm: 15% (Individual)
  - Final Project: 20% (Team)
  - Final Exam: 20% (Individual)
- **ECE 492**
  - Class Presentation: 15% (Teams, 1-2 papers)
  - Quizzes: 20%
  - Midterm: 25%
  - Final Exam: 40%

*Note: 492 students may choose to do Final project instead of the final exam, or a combination which ever helps their overall grade.*

# Attendance Policy

**Attendance Requirement:** Attendance is mandatory for success in this course.
Unexcused absences will result in a failing grade.
*Engage actively in all sessions for optimal learning.*
**Class Recording Link:**
`https://ncsu.hosted.panopto.com/Panopto/Pages/Sessions/List.aspx?folderID=464e93d8-356b-4b42-b3ed-b24409034d1`
**Class recordings are provided only for specific purposes such as:**

- Medical emergencies that prevent attendance.
- Material reviews to reinforce understanding of topics already covered in class.

**Important:**

- Do not rely on recordings as a primary method for learning.
- Active participation in live classes is essential for success in this course.
- Recordings are meant to support, not replace, the interactive and dynamic learning experience of in-class engagement.

# Teaching Assistant Information

**TA:** Azam Ghanbari (PhD Student)
**Email:** aghanba2@ncsu.edu
**Office:** DESK 147 in 2036 Engineering Building II (EB2)
**Office Hours:** Thursdays, 2:00 PM - 3:00 PM
**Zoom Link:** Click Here to be directed to TA Zoom Room

*Note: If you need to meet in person, please email the TA the day before to confirm availability. If no in-person meeting is needed, the TA may not be available at the desk for the entire office hour. Reach out to your TA via Zoom during this time or email if need to meet in person.*

# Course Schedule Overview (Dates are approximate)

| Date Range | Topic Category | Highlights |
|---|---|---|
| Jan 8 – Feb 8 | **Introduction & Essential ML/SYS Background** | Comp Architecture, Foundations in ML: Perceptron, MLPs, RNNs, LSTMs, gradient descent, Transformers, RL, GANs. |
| **Homework 1 Due: Feb 8** | | |
| Feb 8 – March 8 | **Advanced ML Models & LLMs** | Emergent LLM abilities, Chain of Thought (CoT) reasoning, RAG integration. Residual networks, genetic algorithms |
| **Homework 2 Due: Mar 8** | | |
| Mar 8 – Apr 8 | **Advanced Techniques & Applications** | ML applications for System (e.g., Ithemal, DiffTune), RL for hardware. |
| **Project Presentation Due: Apr 22** | | |
| **Final Exam: Apr 25, 3:30 PM - 6:00 PM** | | |
| **Final Project Report Due: Apr 28** | | |

*Note: Detailed schedule, topics, and assignments will be provided in Moodle.*

**Link:** *Click here for the detailed tentative course schedule.*

# Administrative Announcements

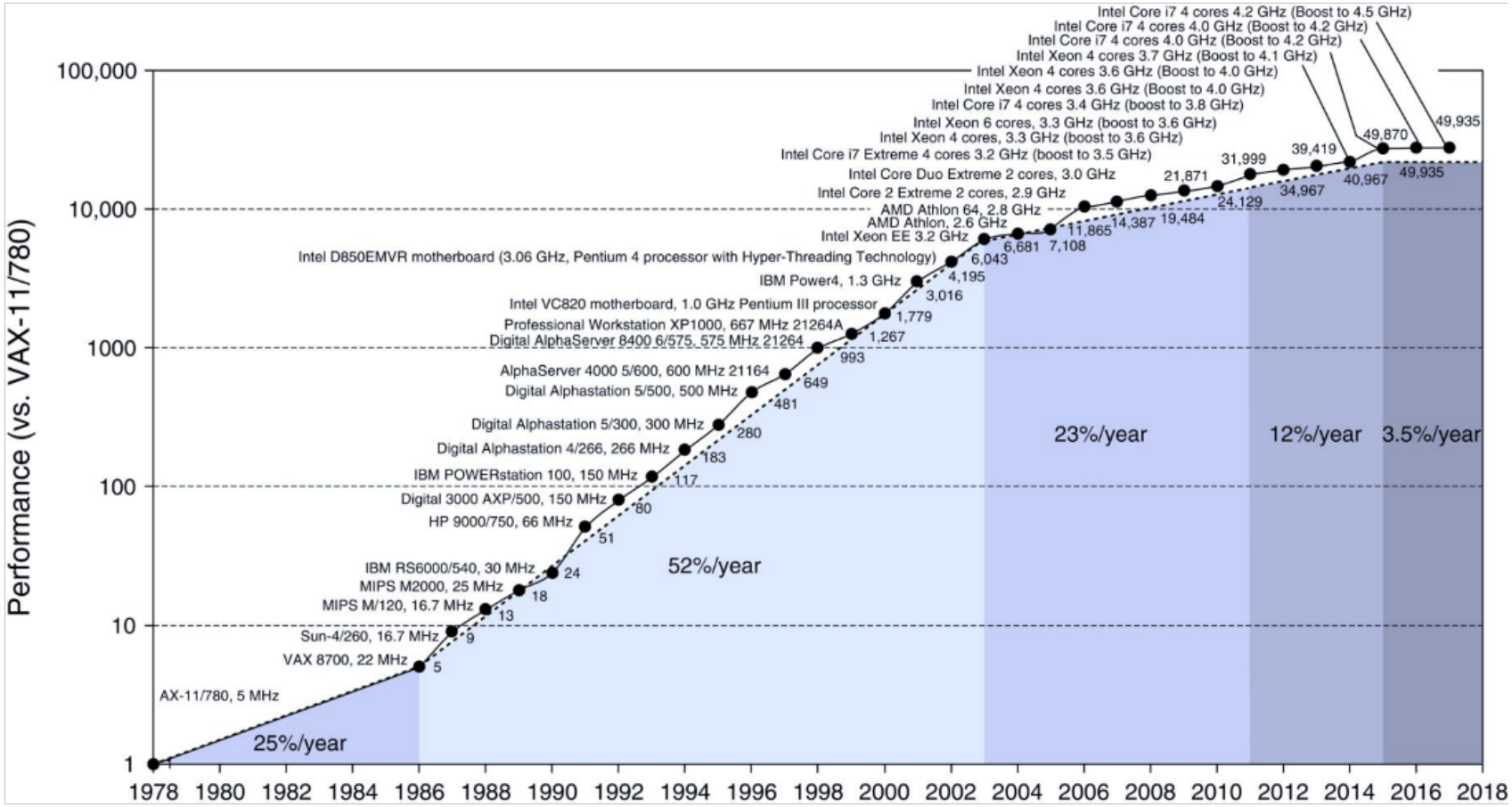
**Class Updates:**

- The class enrollment is finalized with **46 students**.
- Please **form teams of three students** and add your names to the following sheet **ASAP:** `https://docs.google.com/spreadsheets/d/1byLTrW468mQ06be_-5G52UJWF89s0obc0pUdi99xTZg/edit?usp=sharing`

**Presentations and Schedule:**

- The paper for presentations and tentative dates will be announced at least one week before to each team. (Dates may shift by one or two sessions depending on class progress).
- Students auditing the class must **present at least one paper**.
- Each presentation should be **35-40 minutes long**.
- **Quizzes:** May be conducted spontaneously on any day.
- **Midterm:** Planned around Spring Break, either before or after, based on the progress of the material.
- **Office Hours:** By request: After class on both **Monday and Wednesday from 4:15 to 5:30 PM.**

# Processor Performance Trend



# Processor Performance Trends Over Time

- **1978–1986 (25% per year):** The performance increased steadily, driven by architectural and process improvements.
- **1986–2002 (52% per year):** A rapid growth period, largely attributed to advancements in microarchitecture, semiconductor technology scaling (Dennard scaling), and increases in clock frequency.
- **2002–2012 (23% per year):** The growth rate slowed due to physical limits on clock frequency scaling (e.g., power and thermal constraints). Performance improvements shifted to multi-core processors and more efficient designs.
- **2012 onward (3.5–12% per year):** A further slowdown is observed as the end of Moore's Law scaling approaches. Improvements are driven by increased parallelism (more cores), better utilization of power, and architectural optimizations.

# Dennard Scaling

**Definition:** Dennard Scaling describes how transistor performance improves as they shrink, while power density remains constant if voltage and current scale proportionally.

**Key Principles:**

- Transistor dimensions, voltage, and current scale down proportionally.
- Power density remains constant, enabling higher transistor densities without excessive heat.
- Smaller transistors switch faster, increasing performance.

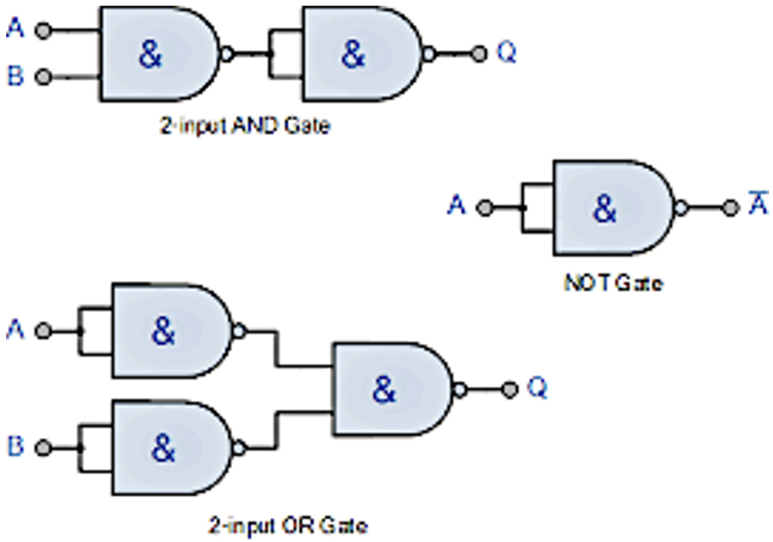
**Breakdown:**

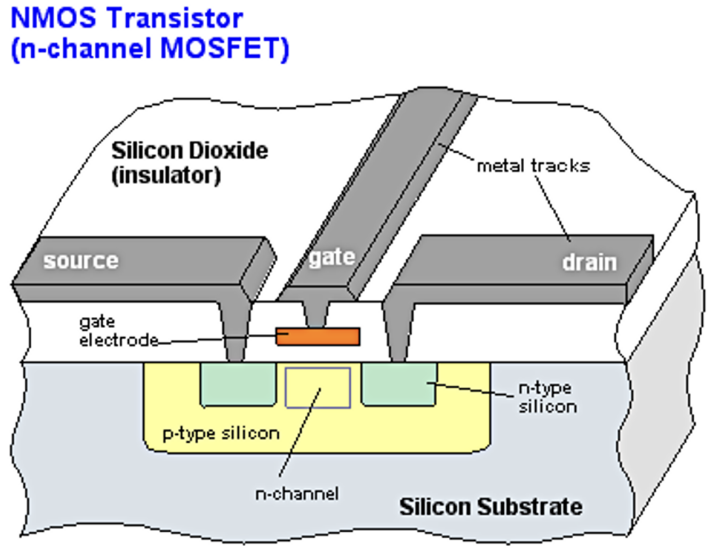
- **Physical Limits:** Increased leakage currents and voltage scaling constraints.
- **Thermal Issues:** Rising heat with higher transistor densities.

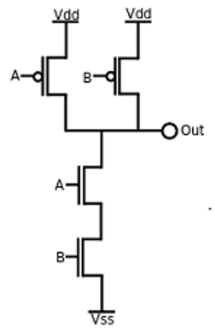
**Post-Dennard Era:**

- Shift to multi-core architectures and specialized hardware.
- Focus on energy efficiency and alternative designs.

# Dennard Scaling and NMOS Transistor




Universal Law of NAND gates:
In digital circuit design, this universality means that a hardware device could potentially be built using just NAND gate

# Foundational Paper: Moore's Law

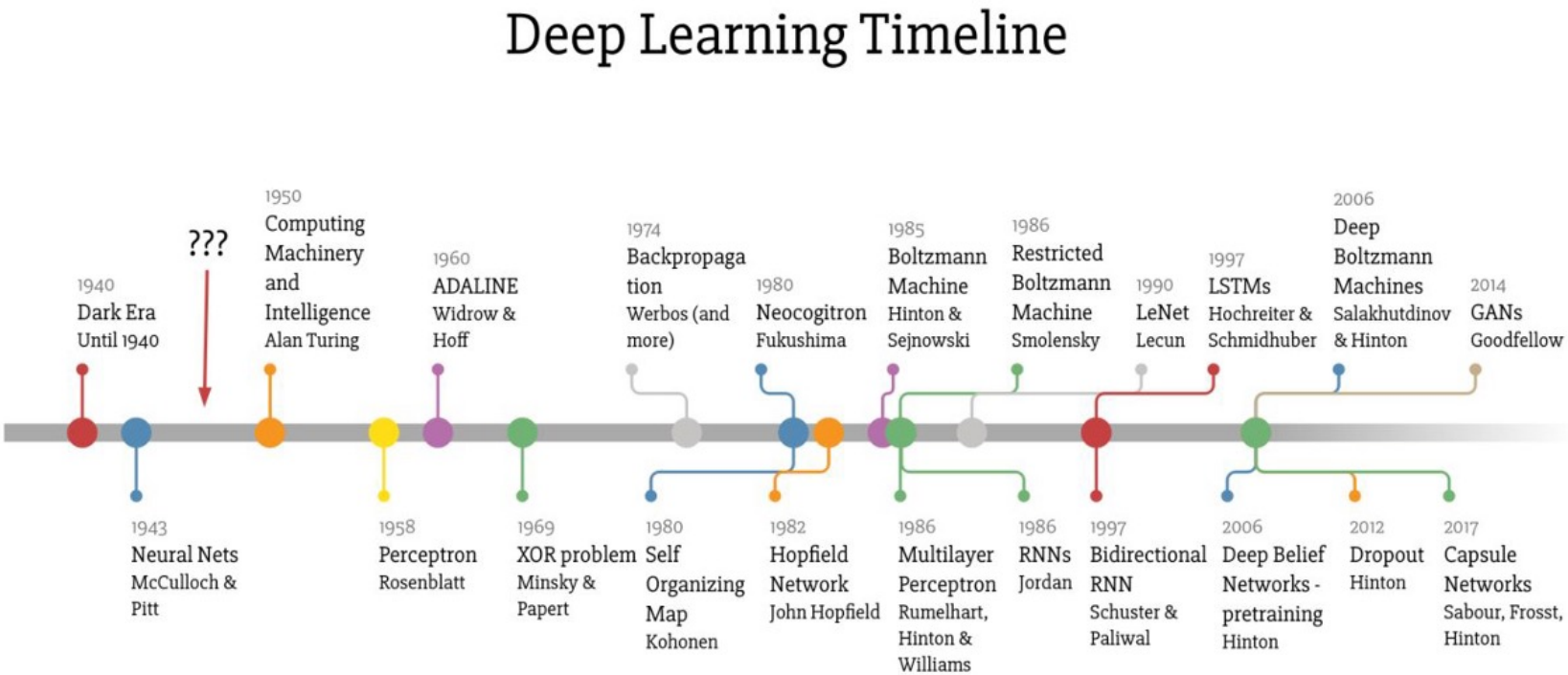
**Recommended Reading:**
Explore the seminal work that predicted the exponential growth of integrated circuits:

- **Title:** Cramming More Components onto Integrated Circuits
- **Author:** Gordon E. Moore
- **Published:** April 19, 1965, in *Electronics* magazine

**Access the Paper:**
`https://www.cs.utexas.edu/ fussell/courses/cs352h/papers/moore.p`

**Abstract:**
In this pioneering article, Moore observed that the number of components per integrated circuit was doubling annually, forecasting the trend that has driven the rapid advancement of electronics and computing technologies.

# Notice 1943


Deep Learning Timeline
1940
Dark Era
Until 1940
???
1943
Neural Nets
McCulloch & Pitt
1950
Computing Machinery and Intelligence
Alan Turing
1958
Perceptron
Rosenblatt
1960
ADALINE
Widrow & Hoff
1969
XOR problem
Minsky & Papert
1974
Backpropagation
Werbos (and more)
1980
Self Organizing Map
Kohonen
1980
Neocogitron
Fukushima
1982
Hopfield Network
John Hopfield
1985
Boltzmann Machine
Hinton & Sejnowski
1986
Multilayer Perceptron
Rumelhart, Hinton & Williams
1986
Restricted Boltzmann Machine
Smolensky
1986
RNNs
Jordan
1990
LeNet
Lecun
1997
Bidirectional RNN
Schuster & Paliwal
1997
LSTMs
Hochreiter & Schmidhuber
2006
Deep Belief Networks - pretraining
Hinton
2006
Deep Boltzmann Machines
Salakhutdinov & Hinton
2012
Dropout
Hinton
2014
GANs
Goodfellow
2017
Capsule Networks
Sabour, Frosst, Hinton
Made by Favio Vázquez

# Foundational Paper: McCulloch-Pitts Neuron Model

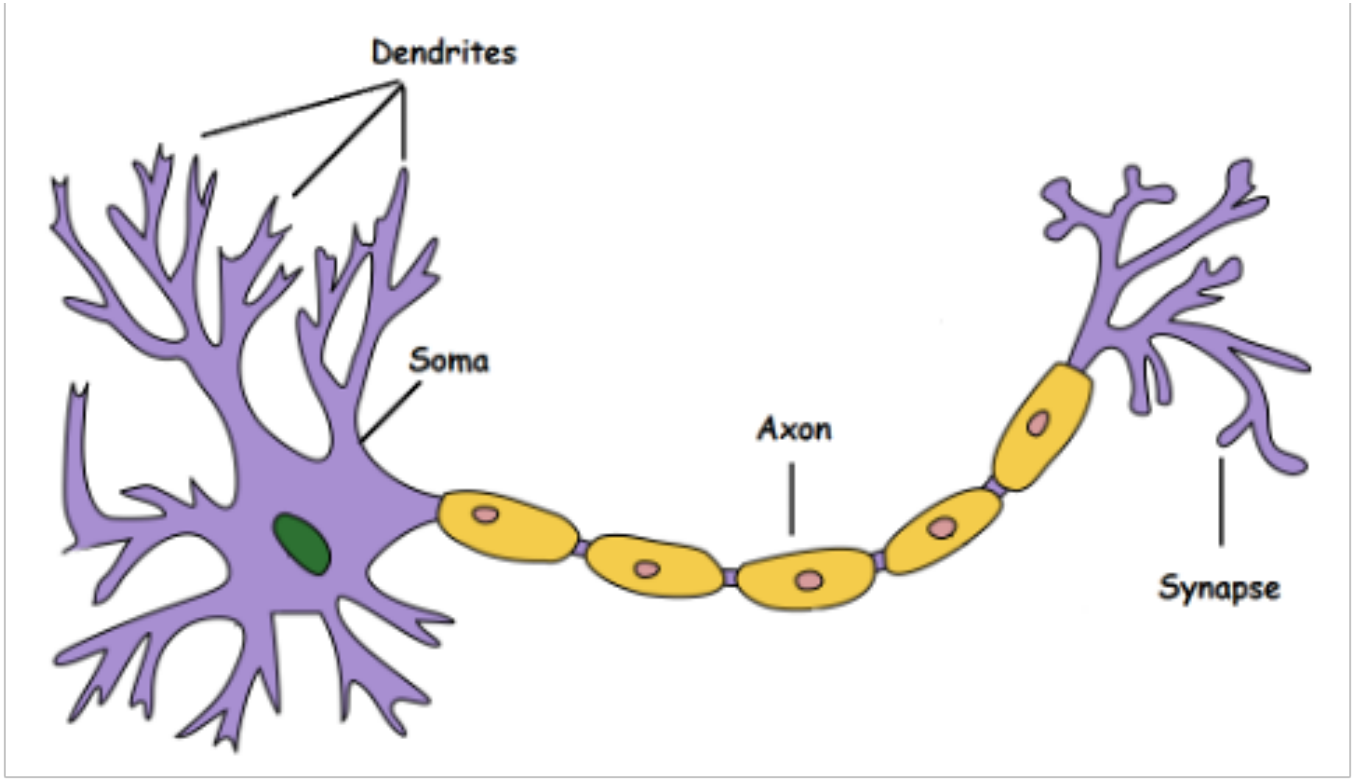
**Recommended Reading:**
Explore the seminal work that laid the foundation for artificial neural networks:

- **Title:** A Logical Calculus of the Ideas Immanent in Nervous Activity
- **Authors:** Warren S. McCulloch and Walter Pitts
- **Published:** 1943 in the Bulletin of Mathematical Biophysics

**Access the Paper:**
https://home.csulb.edu/ cwallis/382/readings/482/mccolloch.logical.calc

**Abstract:**
This pioneering paper introduces a mathematical model of neural activity, proposing that neural networks can compute any arithmetic or logical function. It serves as a cornerstone in the fields of neuroscience and artificial intelligence.

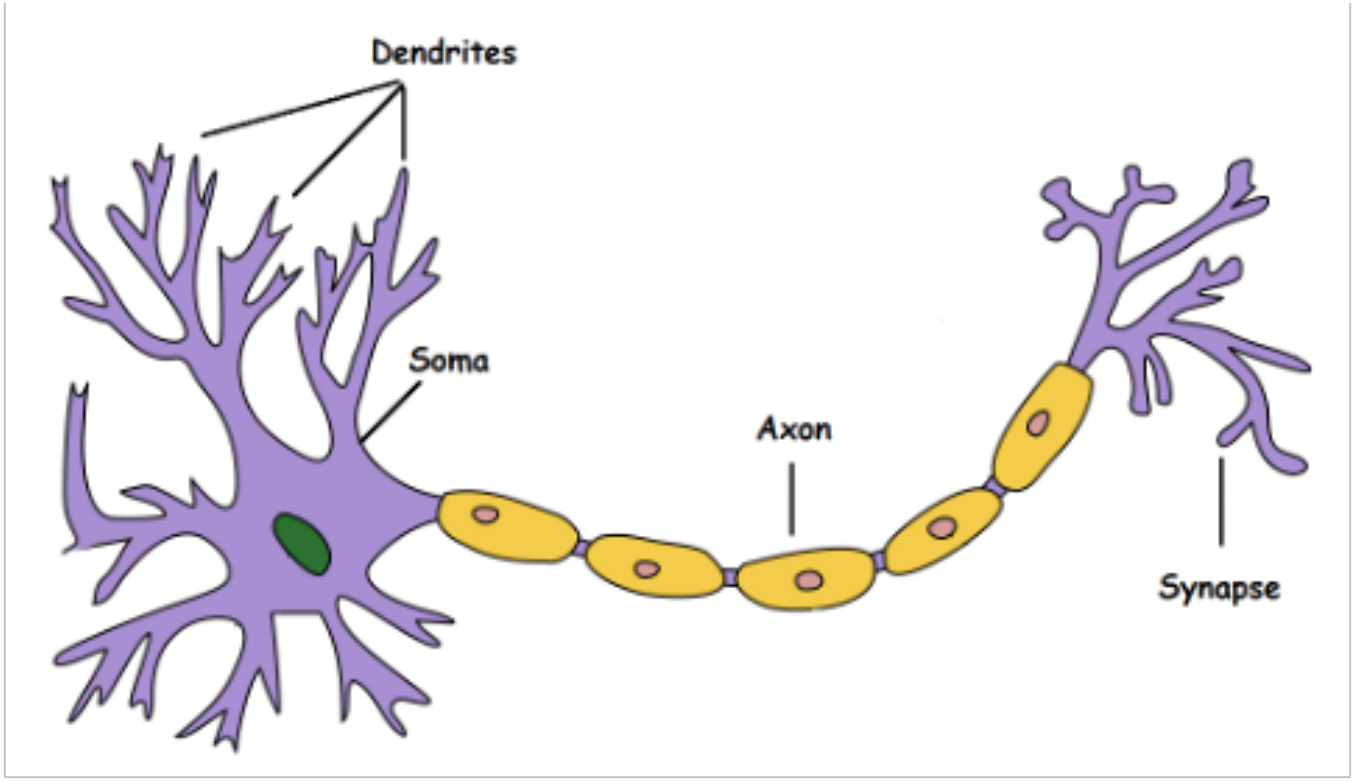
# Neuron Function: Dendrite




**Dendrite:**

- The "input" part of the neuron.
- Receives signals (chemical or electrical) from other neurons or sensory receptors.
- Signals can be excitatory (encouraging the neuron to fire) or inhibitory (discouraging it from firing).

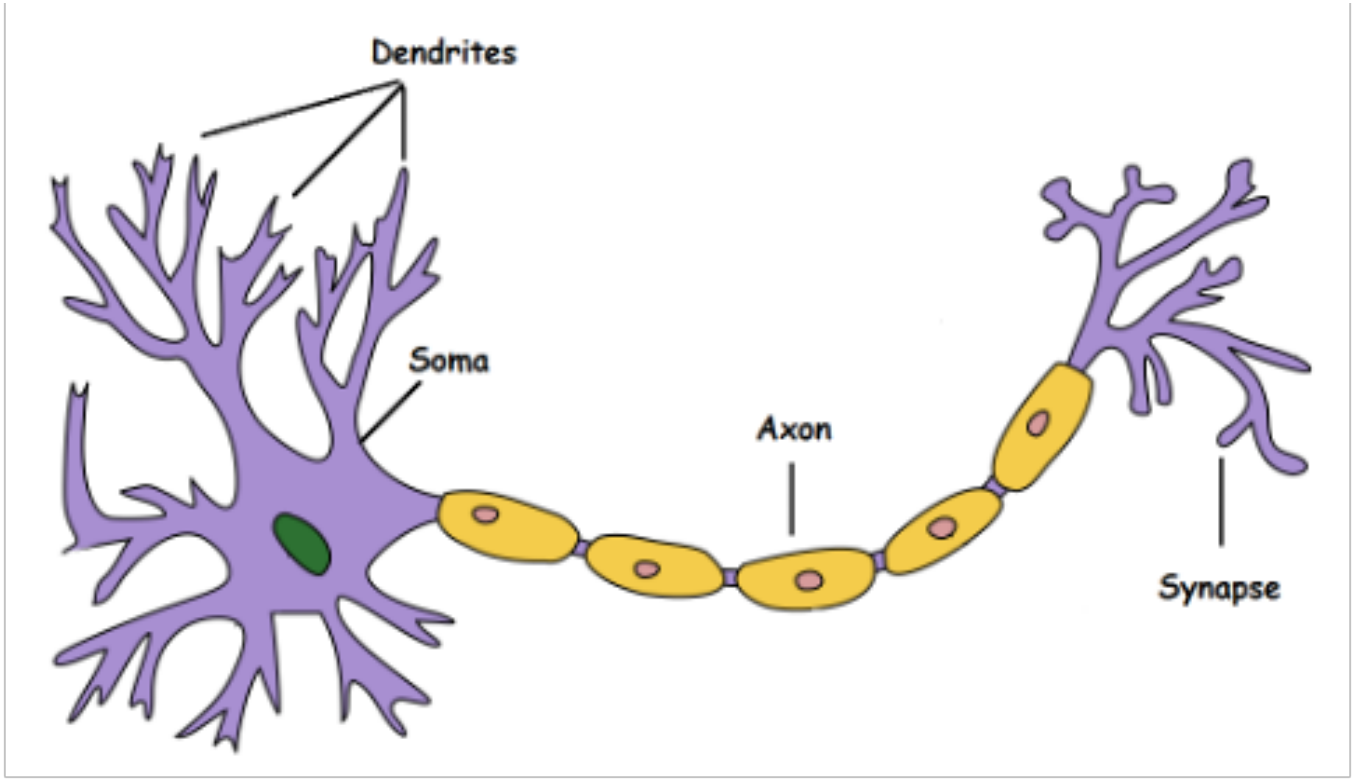
# Neuron Function: Soma (Cell Body)




**Soma (Cell Body):**

- The "processing center" of the neuron.
- Integrates the signals received from the dendrites.
- If the sum of the signals reaches a certain threshold, the soma initiates an electrical impulse (action potential).

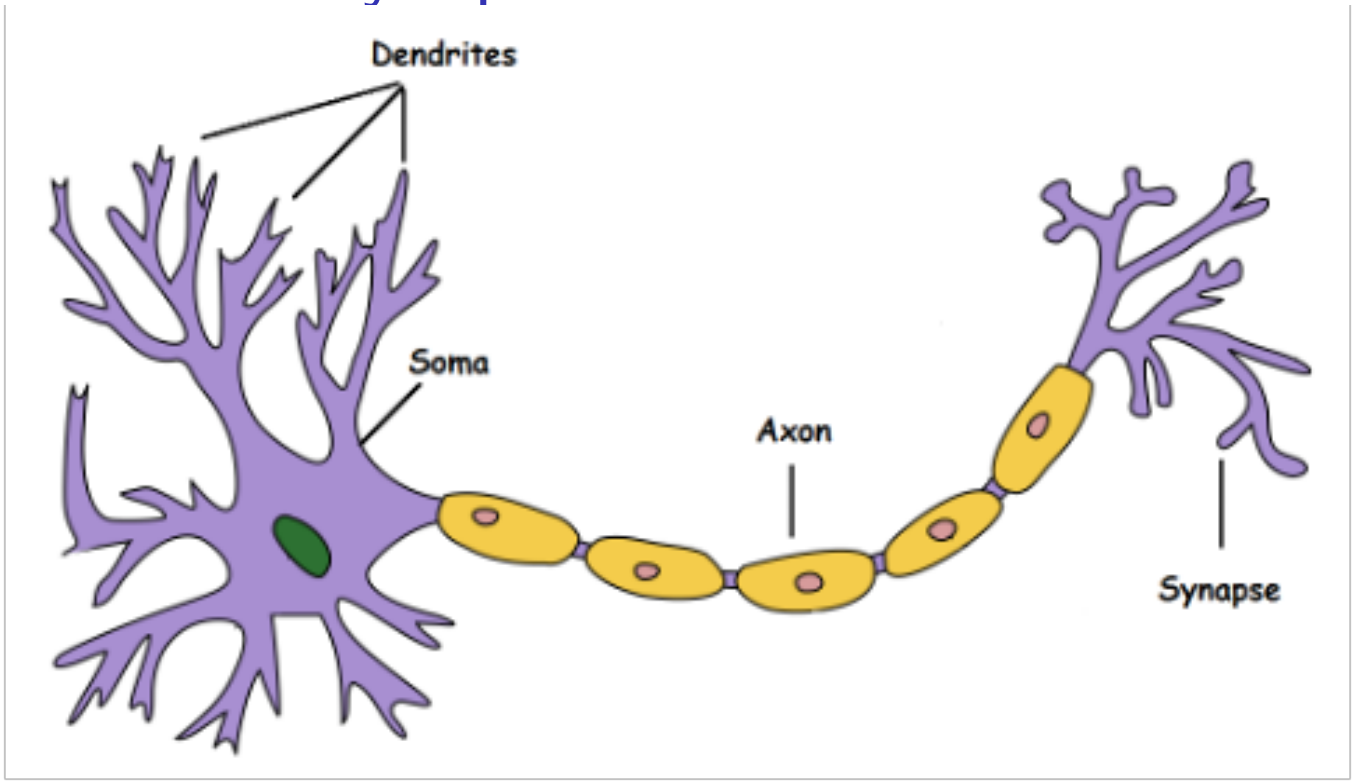
# Neuron Function: Axon




**Axon:**

- The "output" part of the neuron.
- A long, thin fiber that carries the action potential away from the soma.
- The action potential travels down the axon to the axon terminals.

# Neuron Function: Synapse




**Synapse:**

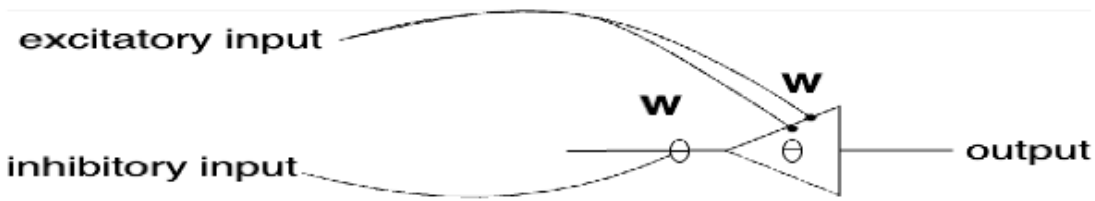
- The "connection point" between neurons.
- The axon terminal of one neuron meets the dendrite of another neuron.
- When the action potential reaches the axon terminal, it triggers the release of neurotransmitters into the synapse.
- Neurotransmitters bind to receptors on the next neuron, passing the signal along.

# McCulloch-Pitts (M-P) Model: Pioneering Model

**Pioneering Model:**

- The first computational model of a neuron, proposed by Warren McCulloch and Walter Pitts in 1943.
- Laid the groundwork for the development of artificial neural networks.

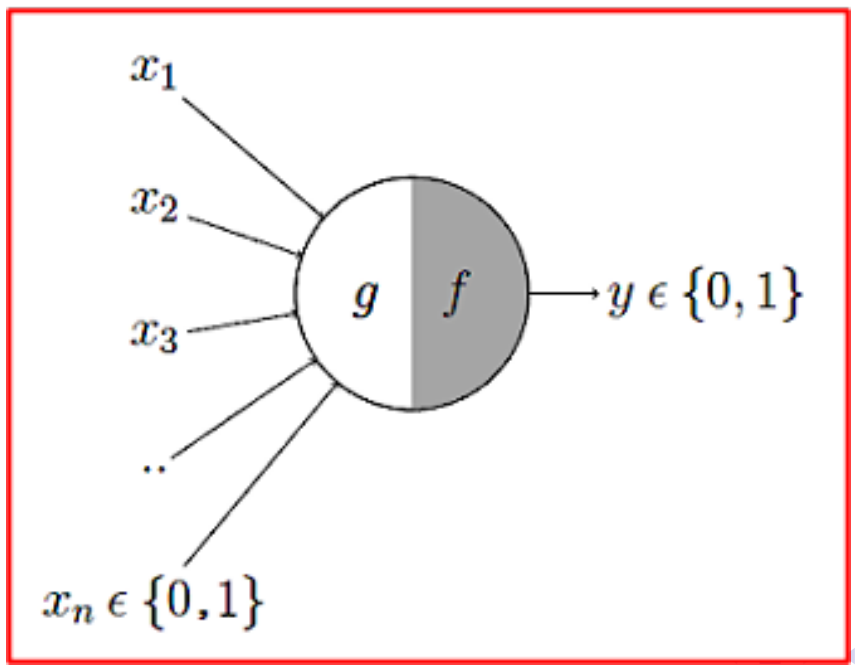
$$V_i = \begin{cases} 1 & \text{if } \sum_j W V_j \geq \theta \text{ AND no inhibition} \\ 0 & \text{otherwise} \end{cases}$$



# McCulloch-Pitts (M-P) Model: Two-Part Structure

**Two-Part Structure:**

- **Part 1 (g):**
  - Takes inputs (analogous to dendrites receiving signals).
  - Performs aggregation (sums the input values).
- **Part 2 (f):**
  - Makes a decision based on the aggregated value.
  - Outputs a 1 (neuron fires) if the aggregated value is greater than or equal to a threshold ($\theta$).
  - Outputs a 0 (neuron doesn't fire) otherwise.

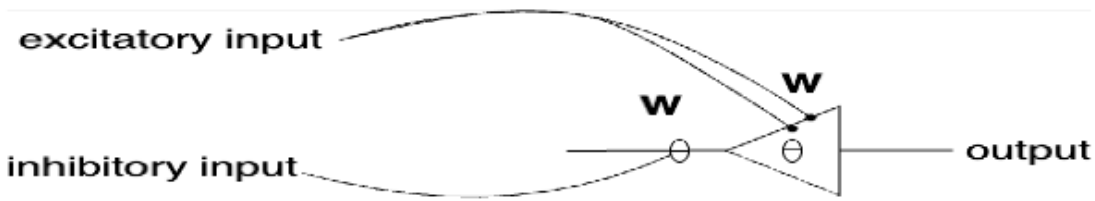


# McCulloch-Pitts (M-P) Model: Types of Inputs - Excitatory

**Excitatory Inputs:**

- Contribute positively to the aggregated value.
- May cause the neuron to fire when combined with other excitatory inputs.

$$V_i = \begin{cases} 1 & \text{if } \sum_j W V_j \geq \theta \text{ AND no inhibition} \\ 0 & \text{otherwise} \end{cases}$$



# McCulloch-Pitts (M-P) Model: Types of Inputs - Inhibitory

**Inhibitory Inputs:**

- Have a strong negative impact on the aggregated value.
- Can prevent the neuron from firing, even if other excitatory inputs are present.

$$g(x_1, x_2, x_3, \ldots, x_n) = \sum_{i=1}^{n} x_i$$

$$y = f(g(x)) = \begin{cases} 1 & \text{if } g(x) \geq \theta \\ 0 & \text{if } g(x) < \theta \end{cases}$$

# M-P "Neurons" Assumptions (Part 1)

**Assumptions:**

1. They are binary devices ($V_i = [0, 1]$).
2. Each neuron has a fixed threshold, $\theta$.
3. The neuron receives inputs from excitatory synapses, all having identical weights.
   - It may receive multiple inputs from the same source, so the excitatory weights are effectively positive integers.
4. Inhibitory inputs have an absolute veto power over any excitatory inputs.

# M-P "Neurons" Assumptions (Part 2)

**Additional Assumptions:**

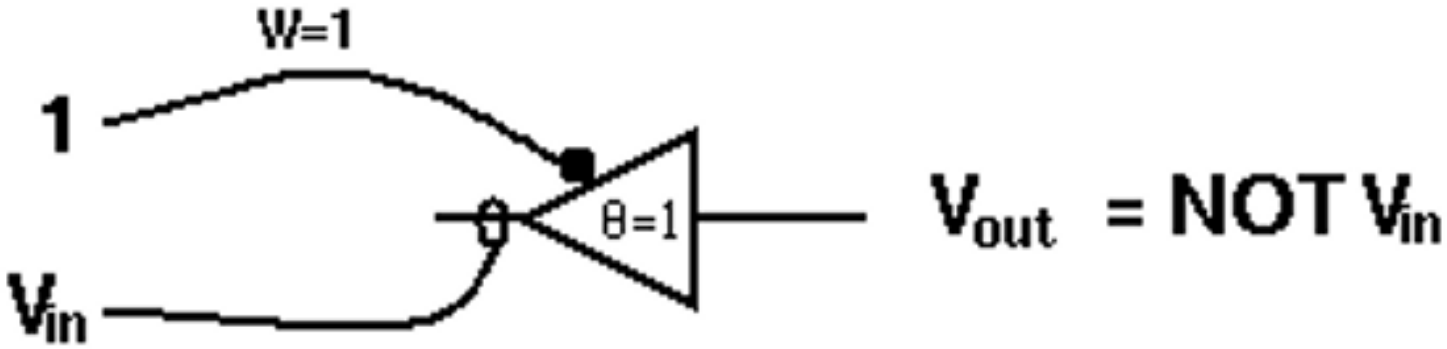
5. At each time step, the neurons are simultaneously (synchronously) updated by summing the weighted excitatory inputs and setting the output ($V_i$) to 1:
   - If the sum is greater than or equal to the threshold ($\theta$) **AND** the neuron receives no inhibitory input.
6. These rules are summarized with the McCulloch-Pitts output rule:

$$V_i = \begin{cases} 1 & \text{if } \sum_j W_j V_j \geq \theta \text{ AND no inhibition} \\ 0 & \text{otherwise.} \end{cases}$$

**Boolean Logic:**

- Using this scheme, we can implement any Boolean logic function.
- With a NOT function and either an OR or AND, you can build XORs, adders, shift registers, and more to perform computation.

# Logical Operations Illustrated




**Key Elements:**

- **Large Triangle with Labels:**
  - Represent the neurons or "cell bodies" in the network.
  - These are the points where information processing occurs.
- **Small, Filled Circles:**
  - Represent excitatory connections between neurons.
  - These connections increase the likelihood that the connected neuron will fire when the input neuron is active.
- **Small, Open Circles:**
  - Represent inhibitory connections.
  - These connections decrease the likelihood that the connected neuron will fire when the input neuron is active.

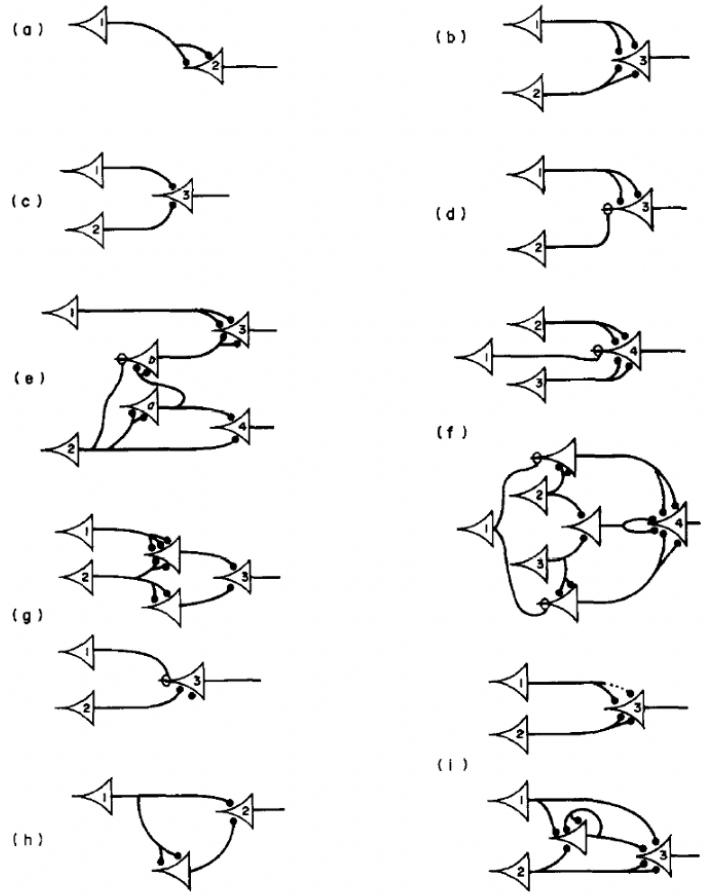



Figure 1. The neuron $c_i$ is always marked with the numeral $i$ upon the body of the cell, and the corresponding action is denoted by "$N$" with $i$ s subscript, as in the text:

(a) $N_2(t) .\equiv. N_1(t-1)$;

(b) $N_3(t) .\equiv. N_1(t-1) \vee N_2(t-1)$;

(c) $N_3(t) .\equiv. N_1(t-1) . N_2(t-1)$;

(d) $N_3(t) .\equiv N_1(t-1) . \sim N_2(t-1)$;

(e) $N_3(t) :\equiv: N_1(t-1) . \vee . N_2(t-3) . \sim N_2(t-2)$;
$N_4(t) .\equiv. N_2(t-2) . N_2(t-1)$;

(f) $N_4(t) :\equiv: \sim N_1(t-1) . N_2(t-1) \vee N_3(t-1) . \vee . N_1(t-1) .$
$N_2(t-1) . N_3(t-1)$
$N_4(t) :\equiv: \sim N_1(t-2) . N_2(t-2) \vee N_3(t-2) . \vee . N_1(t-2) .$
$N_2(t-2) . N_3(t-2)$;

(g) $N_3(t) .\equiv. N_2(t-2) . \sim N_1(t-3)$;

(h) $N_2(t) .\equiv. N_1(t-1) . N_1(t-2)$;

(i) $N_3(t) :\equiv: N_2(t-1) . \vee . N_1(t-1) . (Ex) t-1 . N_1(x) . N_2(x)$.

# NOT Gate with McCulloch-Pitts Neurons

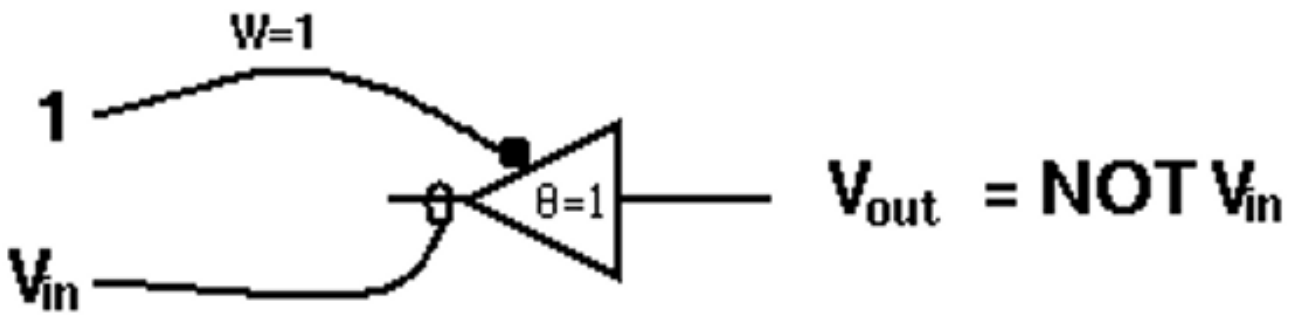
$$V_i = \begin{cases} 1 & \text{if } \sum_j W_j V_j \geq \theta \text{ AND no inhibition} \\ 0 & \text{otherwise.} \end{cases}$$

**Key Points:**

- This setup can implement any Boolean logic function.
- With a NOT gate and either an OR or AND gate, you can construct XORs, adders, shift registers, and more complex operations.
- The output for various inputs is represented as a **truth table**, where:

| Vin | Vout |
|---|---|
| 1 | 0 |
| 0 | 1 |

- When $W = 1$ and $\theta = 1$, the truth table represents the logical NOT gate.



# AND and OR Gates: Formula

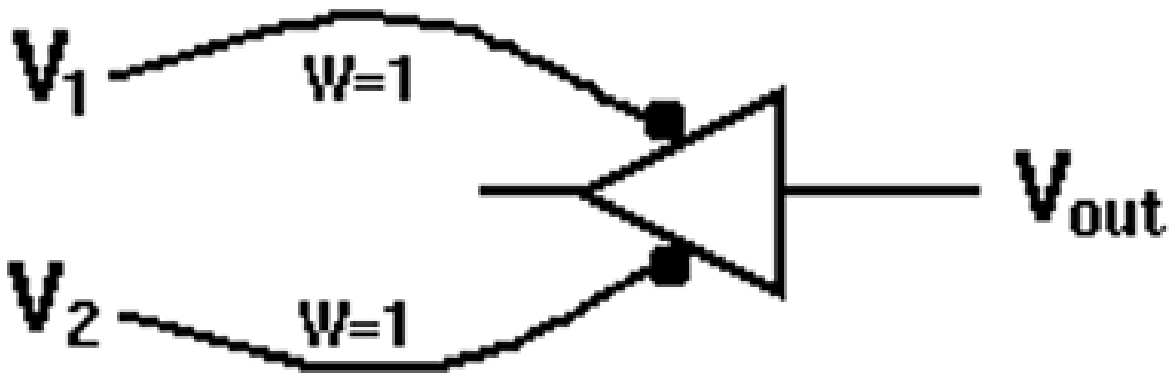
| V1 | V2 | OR | AND |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 |

# AND and OR Gates: Formula

$$V_i = \begin{cases} 1 & \text{if } \sum_j W_j V_j \geq \theta \text{ AND no inhibition} \\ 0 & \text{otherwise.} \end{cases}$$

**Setup:**

- Two excitatory inputs $V_1$ and $V_2$ with weights $W = 1$.
- Output ($V_{out}$) depends on the value of $\theta$.
- Varying $\theta$ allows the implementation of either an OR gate or an AND gate.

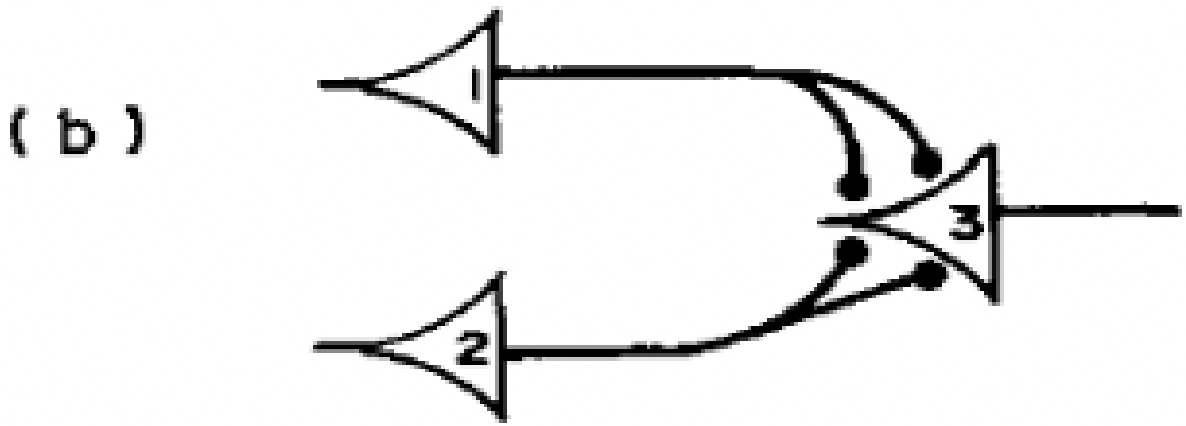


# AND and OR Gates: OR Gate

**Condition:**

$$\text{If } \theta = 1 \implies V_{out} = V_1 \text{ OR } V_2$$

- Output fires if at least one input is active ($V_1 = 1$ or $V_2 = 1$).
- Represents a logical OR operation.

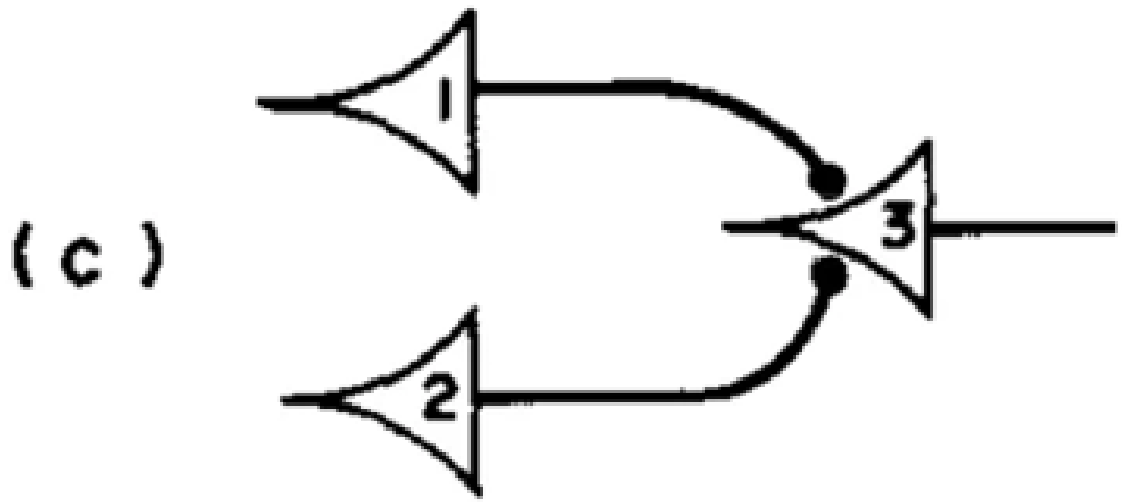


# AND and OR Gates: AND Gate

**Condition:**

$$\text{If } \theta = 2 \implies V_{out} = V_1 \text{ AND } V_2$$

- Output fires only if both inputs are active ($V_1 = 1$ and $V_2 = 1$).
- Represents a logical AND operation.




McCulloch-Pitt's Neurons linked to form an 'AND' gate. Illustration from "A Logical Calculus of the Ideas Immanent in Nervous Activity"

# AND and OR Gates: Truth Table

**Truth Table:**

| $V_1$ | $V_2$ | OR | AND |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 |

**Verification:**

- Test the table using weights ($W = 1$) and thresholds ($\theta = 1$ or $\theta = 2$).
- Confirm that the output matches logical OR and AND gate behavior.

# AND and OR Gates: Diagram Summary

**McCulloch-Pitts Neuron: Threshold Logic**

- $\theta = 1$: Implements an OR gate.
- $\theta = 2$: Implements an AND gate.

# XOR Gate with McCulloch-Pitts Neurons

**Exclusive OR (XOR):**

- The XOR gate outputs 1 only when exactly one of the inputs is 1.
- Truth table:

| $V_1$ | $V_2$ | XOR |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**Note:** The XOR gate is also referred to as a "1-bit adder."

# XOR Gate with McCulloch-Pitts Neurons

**Key Point:**

- XOR cannot be represented with a single McCulloch-Pitts neuron.
- XOR requires a combination of basic logic gates such as AND, OR, and NOT.

# XOR as a Combination of Logic Gates

**Constructing XOR:**

- XOR can be represented as:

$$\text{XOR} = (V_1 \text{ OR } V_2) \text{ AND NOT } (V_1 \text{ AND } V_2)$$

**Extended Truth Table:**

| $V_1$ | $V_2$ | OR | AND | XOR |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 |

**Observation:**

- XOR combines multiple gates to achieve its output.
- This demonstrates the compositional nature of logic gates in neural circuits.

# XOR Gate with McCulloch-Pitts Neurons

**The Relationship:**

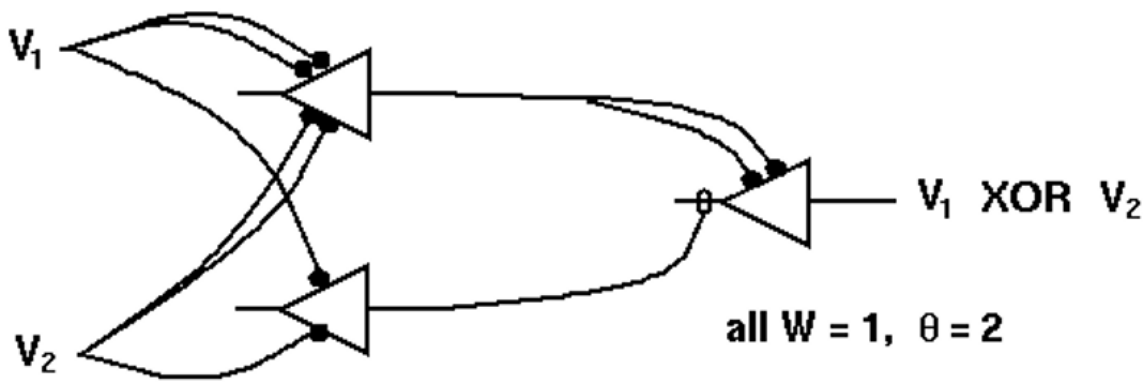
$$\text{XOR} = (V_1 \text{ OR } V_2) \text{ AND NOT } (V_1 \text{ AND } V_2)$$

- This suggests that XOR can be represented using a network of neurons.
- Each neuron implements a specific logic function (AND, OR, or NOT).

# XOR Gate Implementation Details

**McCulloch-Pitts Network Key Points:**

- The function of the NOT operation is implemented using **inhibitory connections**.
- The inhibitory connection:
  - Directly connects the AND neuron to the final neuron.
  - Negates the AND output when active.
- The final neuron:
  - Combines the positive input from the OR gate.
  - Includes the inhibitory input from the AND gate (negated output).

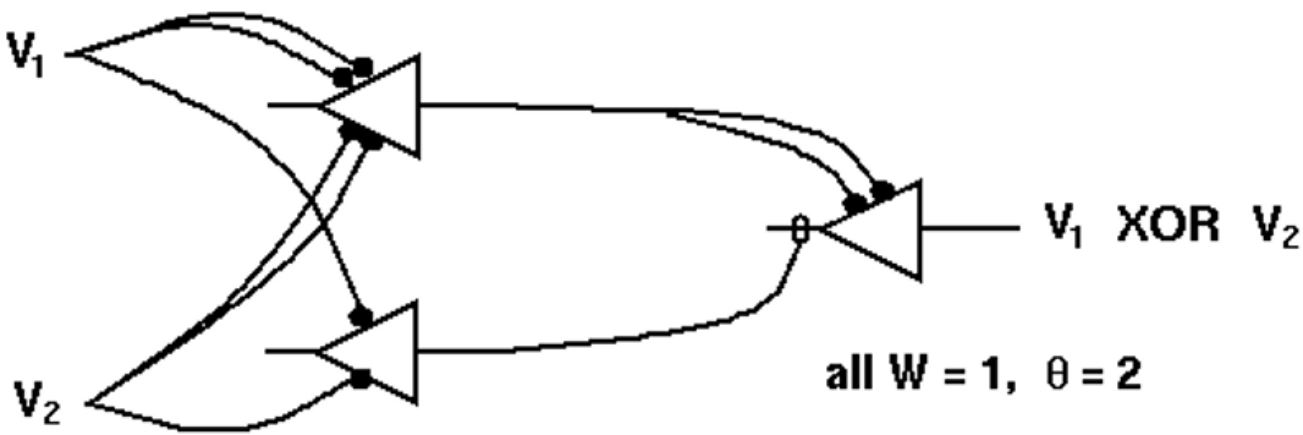


# XOR Gate Summary

**Summary of the XOR Gate Implementation:**

- All weights ($W$) are set to 1.
- Threshold ($\theta$) is set to 2 for the final neuron.
- The XOR gate combines the following operations:
  - OR gate: Positive input.
  - AND gate: Input negated through an inhibitory connection.
- This structure ensures the XOR behavior:

$$V_1 \text{ XOR } V_2$$

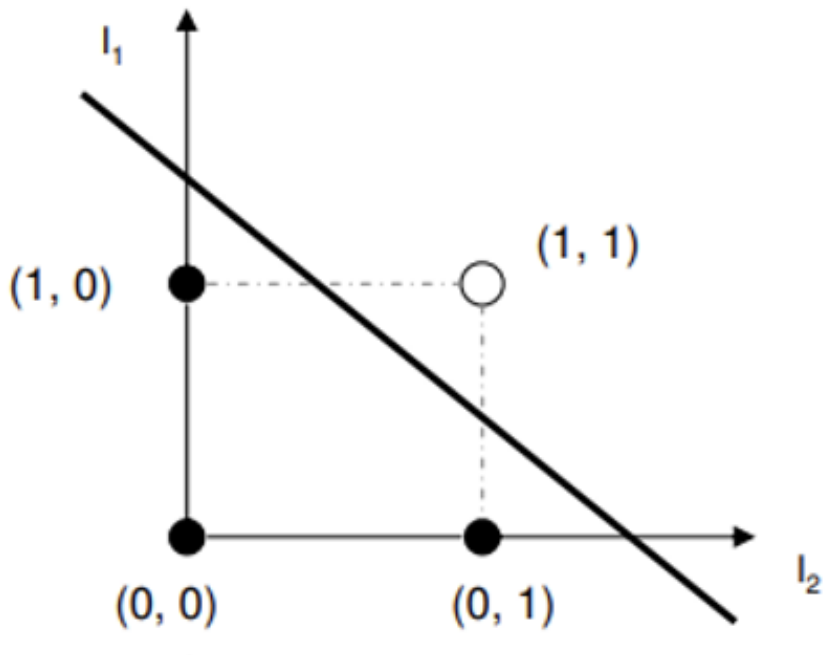


# XOR Representation with a Network

**Key Insights:**

- The XOR function can be represented using a combination of basic logic gates (AND, OR, NOT).
- Truth tables and geometric representations are shown for:
  - AND operation: Linear separability using a single line.

| AND | | |
|---|---|---|
| $I_1$ | $I_2$ | out |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

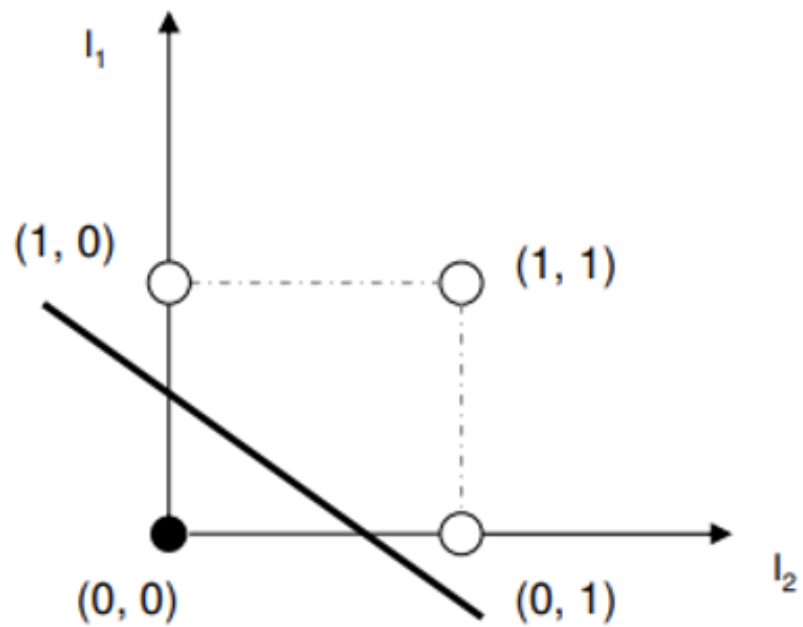


# XOR Representation with a Network

**Key Insights:**

- The XOR function can be represented using a combination of basic logic gates (AND, OR, NOT).
- Truth tables and geometric representations are shown for:
  - AND operation: Linear separability using a single line.
  - OR operation: Linear separability using a single line.

| OR | | |
|---|---|---|
| $I_1$ | $I_2$ | out |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

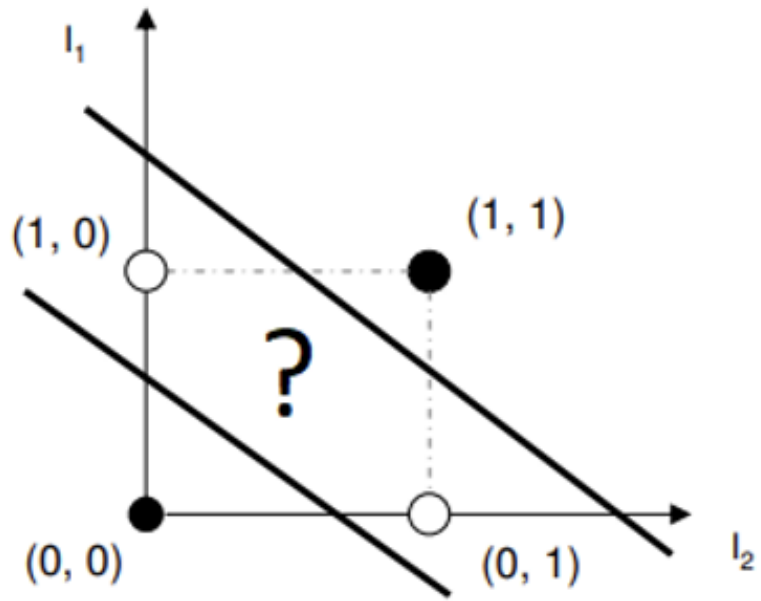


# XOR Representation with a Network

**Key Insights:**

- The XOR function can be represented using a combination of basic logic gates (AND, OR, NOT).
- Truth tables and geometric representations are shown for:
  - AND operation: Linear separability using a single line.
  - OR operation: Linear separability using a single line.
  - XOR operation: Requires multiple lines for separation, illustrating that it is **not linearly separable**.

| XOR | | |
|---|---|---|
| $I_1$ | $I_2$ | out |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |



# Limitation of McCulloch-Pitts (M-P) Model

**Key Importance:**

- The M-P model demonstrated that networks of simple neuron-like elements could compute logical functions.

**Limitation:**

???

# Limitation of McCulloch-Pitts (M-P) Model

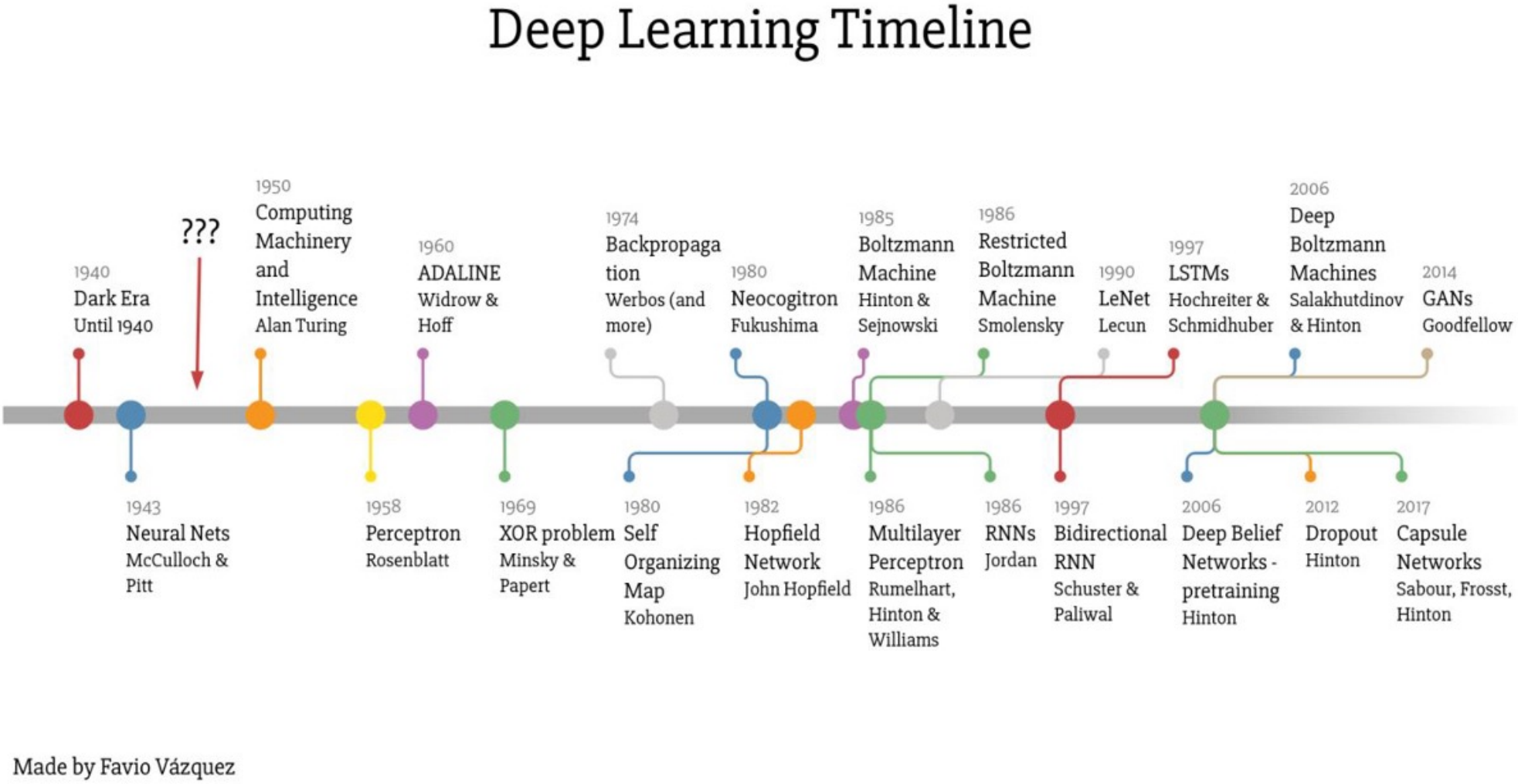
**Key Importance:**

- The M-P model demonstrated that networks of simple neuron-like elements could compute logical functions.

**Limitation:**

These results were very encouraging, but these networks displayed no learning. They were essentially "hard-wired" logic devices. One had to figure out the weights and connect up the neurons in the appropriate manner to perform the desired computation. **Thus, there is no real advantage over any conventional digital logic circuit.**

# Notice 1958


Deep Learning Timeline
1940
Dark Era
Until 1940
???
1943
Neural Nets
McCulloch & Pitt
1950
Computing Machinery and Intelligence
Alan Turing
1958
Perceptron
Rosenblatt
1960
ADALINE
Widrow & Hoff
1969
XOR problem
Minsky & Papert
1974
Backpropagation
Werbos (and more)
1980
Neocogitron
Fukushima
1980
Self Organizing Map
Kohonen
1982
Hopfield Network
John Hopfield
1985
Boltzmann Machine
Hinton & Sejnowski
1986
Multilayer Perceptron
Rumelhart, Hinton & Williams
1986
Restricted Boltzmann Machine
Smolensky
1986
RNNs
Jordan
1990
LeNet
Lecun
1997
Bidirectional RNN
Schuster & Paliwal
1997
LSTMs
Hochreiter & Schmidhuber
2006
Deep Belief Networks - pretraining
Hinton
2006
Deep Boltzmann Machines
Salakhutdinov & Hinton
2012
Dropout
Hinton
2014
GANs
Goodfellow
2017
Capsule Networks
Sabour, Frosst, Hinton
Made by Favio Vázquez

# The Original Perceptron Paper

**Title:** *The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain*
**Authors:** Frank Rosenblatt
**Published:** 1958, *Psychological Review*

**Key Contributions:**

- Introduced the concept of the perceptron, a simple model of a biological neuron.
- Demonstrated the perceptron's ability to learn and classify linearly separable patterns.
- Provided insights into the potential of neural networks for solving computational problems.

**Link to the Paper:**
`https://www.ling.upenn.edu/courses/cogs501/Rosenblatt1958.pdf`

**Abstract:**
The perceptron was introduced as a probabilistic model of information storage and organization in the brain, laying the groundwork for modern neural networks.

# The Perceptron

The next major advance was the perceptron, introduced by Frank Rosenblatt in his 1958 paper.
The perceptron had the following differences from the McCullough-Pitts neuron:

1. The weights and thresholds were not all identical.
2. Weights can be positive or negative.
3. There is no absolute inhibitory synapse.
4. Although the neurons were still two-state, the output function $f(u)$ goes from $[-1, 1]$, not $[0, 1]$.
   - (This is no big deal, as a suitable change in the threshold lets you transform from one convention to the other.)
5. Most importantly, there was a learning rule.

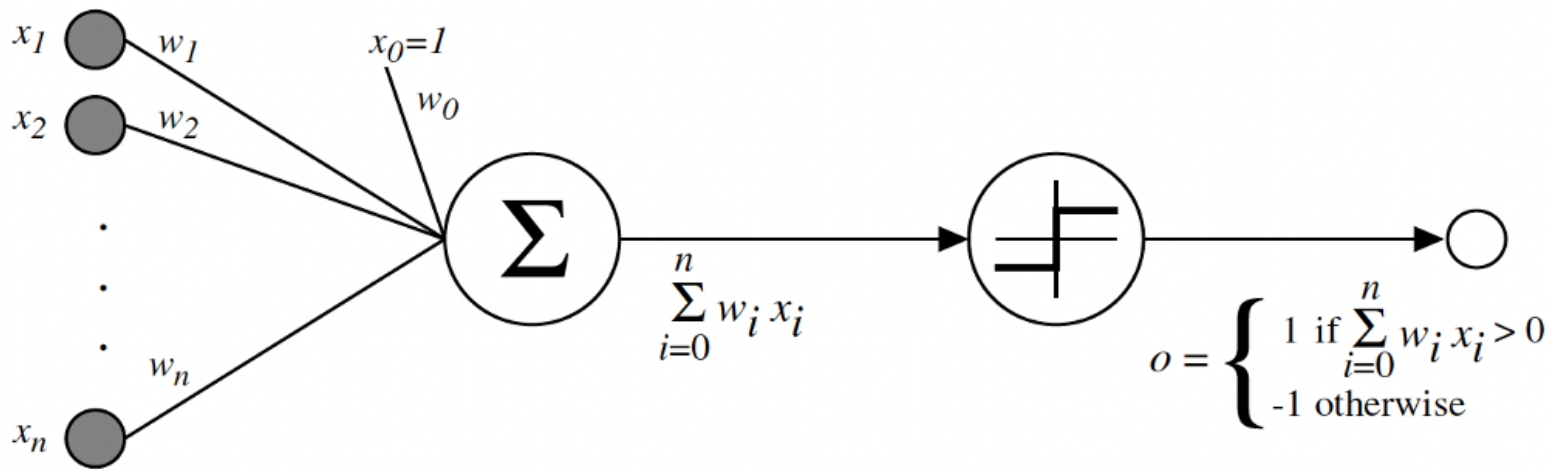
# The Perceptron: Mathematical Representation

**Output Function:**

$$o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

**Simpler Vector Notation:**

$$o(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} > 0 \\ -1 & \text{otherwise.} \end{cases}$$



# Perceptron Training Rule

- The Perceptron outputs are defined as:

$$o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0, \\ -1 & \text{otherwise.} \end{cases}$$

- In vector notation:

$$o(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} > 0, \\ -1 & \text{otherwise.} \end{cases}$$

**Perceptron Weight Update Rule:**

$$w_i \leftarrow w_i + \Delta w_i$$

**Where:**

$$\Delta w_i = \eta (t - o) x_i$$

- $t$: Target value.
- $o$: Perceptron output.
- $\eta$: Small constant (e.g., 0.1) called the **learning rate**.

# Perceptron Training Rule

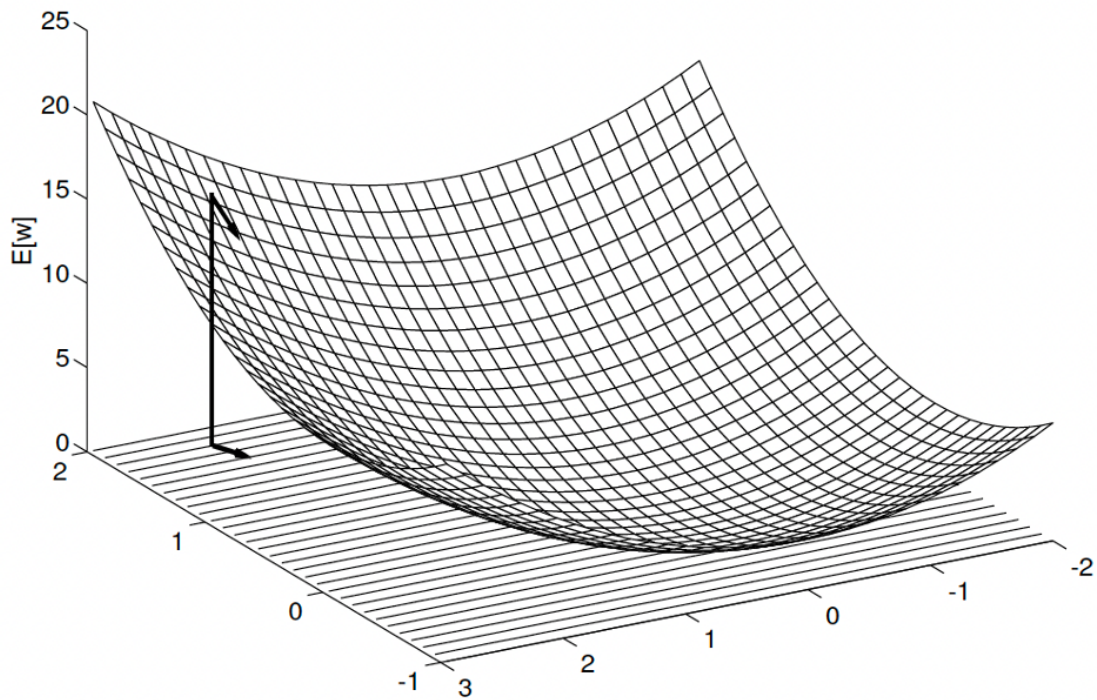
**Perceptron Weight Update Rule:**

$$w_i \leftarrow w_i + \Delta w_i$$

**Where:**

$$\Delta w_i = \eta(t - o)x_i$$

- The role of $\eta$ is to moderate the degree to which weights are adjusted at each step.
- Weight updates continue until the perceptron classifies all training examples correctly.



# Perceptron Learning Rule

$$w_i \leftarrow w_i + \Delta w_i, \quad \text{where } \Delta w_i = \eta(t - o)x_i$$

**Intuitive Explanation (specific cases):**

- If $t - o = 0$: No weight update is required as the example is already correctly classified.
- If $t = +1$ and $o = -1$: Increase $w_i$ if $x_i > 0$ to bring the perceptron closer to the correct classification.
- If $t = -1$ and $o = +1$: Decrease $w_i$ for positive $x_i$.

# Perceptron Learning Rule

$$w_i \leftarrow w_i + \Delta w_i, \quad \text{where } \Delta w_i = \eta(t - o)x_i$$

**Intuitive Explanation:**

- If $t - o = 0$: No weight update is required as the example is already correctly classified.
- If $t = +1$ and $o = -1$: Increase $w_i$ if $x_i > 0$ to bring the perceptron closer to the correct classification.
- If $t = -1$ and $o = +1$: Decrease $w_i$ for positive $x_i$.

**Example Calculation:**

- Given $x_i = 0.8, \eta = 0.1, t = 1, o = -1$:

$$\Delta w_i = \eta(t - o)x_i = 0.1(1 - (-1))0.8 = 0.16$$

- Adjust $w_i$ by adding $\Delta w_i$.

# Perceptron Learning Rule

$$w_i \leftarrow w_i + \Delta w_i, \quad \text{where } \Delta w_i = \eta(t - o)x_i$$

**Intuitive Explanation:**

- If $t - o = 0$: No weight update is required as the example is already correctly classified.
- If $t = +1$ and $o = -1$: Increase $w_i$ if $x_i > 0$ to bring the perceptron closer to the correct classification.
- If $t = -1$ and $o = +1$: Decrease $w_i$ for positive $x_i$.

**Example Calculation:**

- Given $x_i = 0.8, \eta = 0.1, t = 1, o = -1$:

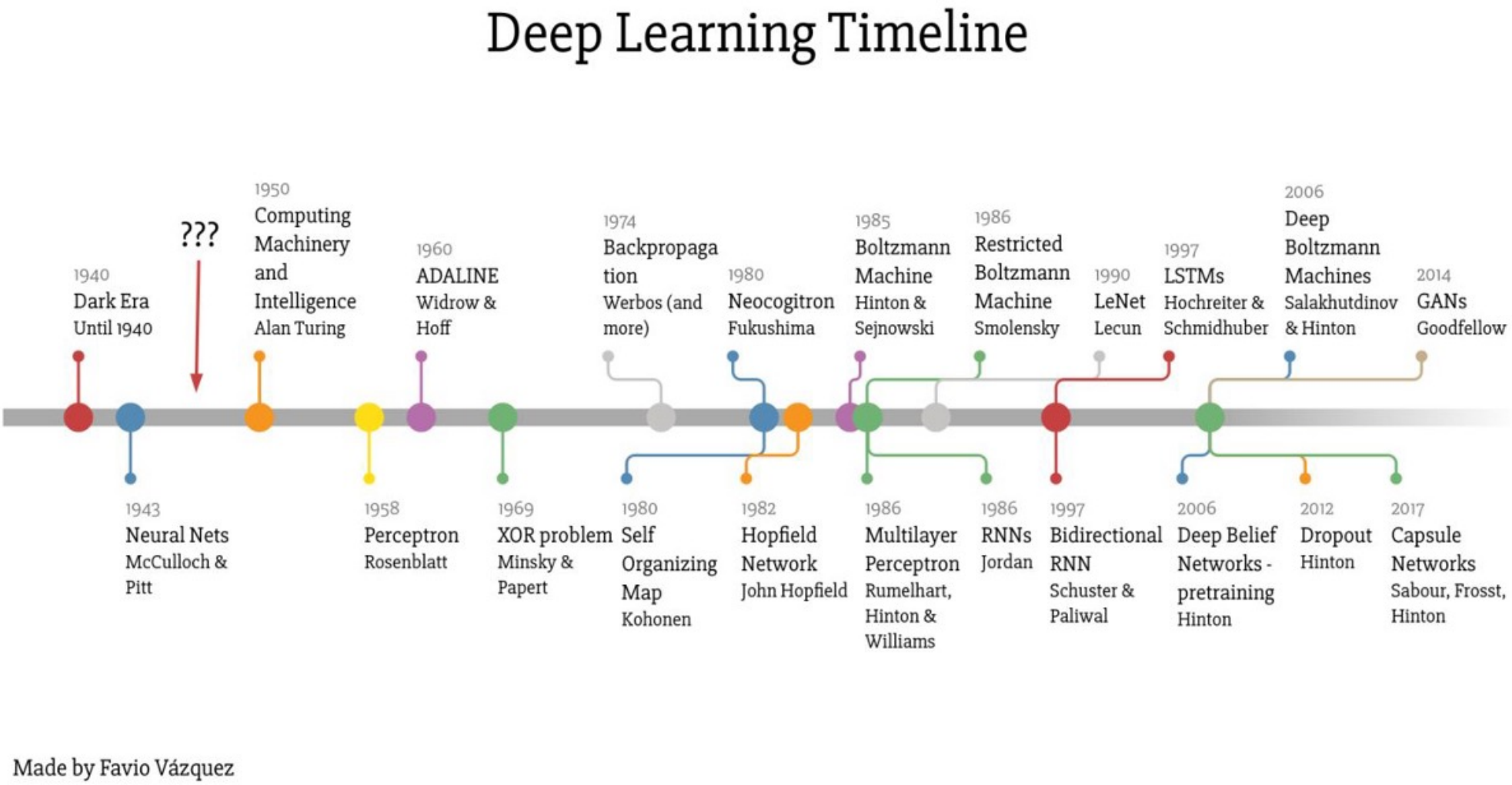
$$\Delta w_i = \eta(t - o)x_i = 0.1(1 - (-1))0.8 = 0.16$$

- Adjust $w_i$ by adding $\Delta w_i$.

**Convergence:**

- The learning procedure is guaranteed to converge to a weight vector that correctly classifies all training examples if:
  - The data are linearly separable.
  - A sufficiently small $\eta$ is used.
- For non-linearly separable data, convergence is not guaranteed (Minsky and Papert, 1969).

# Notice 1969


Deep Learning Timeline
1940
Dark Era
Until 1940
???
1943
Neural Nets
McCulloch & Pitt
1950
Computing Machinery and Intelligence
Alan Turing
1958
Perceptron
Rosenblatt
1960
ADALINE
Widrow & Hoff
1969
XOR problem
Minsky & Papert
1974
Backpropagation
Werbos (and more)
1980
Neocogitron
Fukushima
1980
Self Organizing Map
Kohonen
1982
Hopfield Network
John Hopfield
1985
Boltzmann Machine
Hinton & Sejnowski
1986
Multilayer Perceptron
Rumelhart, Hinton & Williams
1986
Restricted Boltzmann Machine
Smolensky
1986
RNNs
Jordan
1990
LeNet
Lecun
1997
Bidirectional RNN
Schuster & Paliwal
1997
LSTMs
Hochreiter & Schmidhuber
2006
Deep Belief Networks - pretraining
Hinton
2006
Deep Boltzmann Machines
Salakhutdinov & Hinton
2012
Dropout
Hinton
2014
GANs
Goodfellow
2017
Capsule Networks
Sabour, Frosst, Hinton
Made by Favio Vázquez

# Introduction to Minsky and Papert (1969)

**Title:** *Perceptrons: An Introduction to Computational Geometry*
**Authors:** Marvin Minsky and Seymour Papert
**Published:** 1969
**Key Contributions:**

- Explored the capabilities and limitations of the perceptron model.
- Demonstrated that perceptrons cannot solve problems requiring non-linear decision boundaries (e.g., XOR problem).
- Discussed the limitations of single-layer neural networks and the necessity of multi-layer models.
- Highlighted the importance of network structure in computational learning.

**Impact:**

- Marked a turning point in AI research, leading to the "AI Winter."
- Motivated the development of multi-layer perceptrons and backpropagation in the 1980s.

**Link to the Paper:**
`https://archive.org/details/perceptronsintro00mins/page/n3/mode/`

# The AI Winter

A period of reduced funding, interest, and optimism in AI research due to unmet expectations and technological limitations.

- **Unrealistic Expectations:** Early AI pioneers over-promised capabilities that couldn't be delivered.
- **Limitations of Early AI Models:** - Minsky and Papert's 1969 critique of perceptrons highlighted their inability to solve non-linear problems.
- **Lack of Computational Power:** Hardware of the time was inadequate for scaling AI systems.
- **Over-Promise and Under-Delivery:** Failures in natural language processing, vision, and robotics.
- Reduced funding from governments and industries.
- Loss of interest and stagnation in AI research.
- Shift of focus to other areas in computer science and engineering.

**Revival After AI Winter:**

- Development of multi-layer perceptrons and backpropagation (1986).
- Advances in computational power (e.g., GPUs).
- Availability of large datasets and improved algorithms.

# Gradient Descent and the Delta Rule

**Motivation:**

- The perceptron rule works for linearly separable data, but it fails to converge for non-linearly separable data.
- The **Delta Rule** is introduced to address this issue:
  - Uses gradient descent to find weights that minimize the training error.
  - Converges toward the best-fit approximation to the target concept, even for non-linearly separable data.

**Importance of the Delta Rule:**

- Forms the basis for the **Backpropagation algorithm**, which is used in multi-layer networks.
- Provides a foundation for algorithms that search hypothesis spaces of continuously parameterized hypotheses.

**Linear Unit:**

- The Delta Rule is applied to **linear units**, which are perceptrons without a threshold.
- Output is given by:

$$o(\vec{x}) = \vec{w} \cdot \vec{x}$$

# Perceptron Learning Rule vs. Delta Rule

**Key Differences:**

- **Objective:**
  - **Perceptron Learning Rule:** Updates weights to correct misclassifications. Works for linearly separable data.
  - **Delta Rule:** Uses gradient descent to minimize the squared error, targeting both classification and regression tasks.
- **Output Function:**
  - **Perceptron Rule:** Step function:

$$o = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} > 0 \\ -1 & \text{otherwise.} \end{cases}$$

  - **Delta Rule:** Linear output:

$$o = \vec{w} \cdot \vec{x}$$

- **Convergence:**
  - **Perceptron Rule:** Converges only for linearly separable data.
  - **Delta Rule:** Converges to the best-fit approximation, even for non-linearly separable data.

# Gradient Descent and the Delta Rule

**Training Error for Linear Units:**

- A common measure for training error is:

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

- Where:
  - $D$: Set of training examples.
  - $t_d$: Target output for training example $d$.
  - $o_d$: Output of the linear unit for training example $d$.

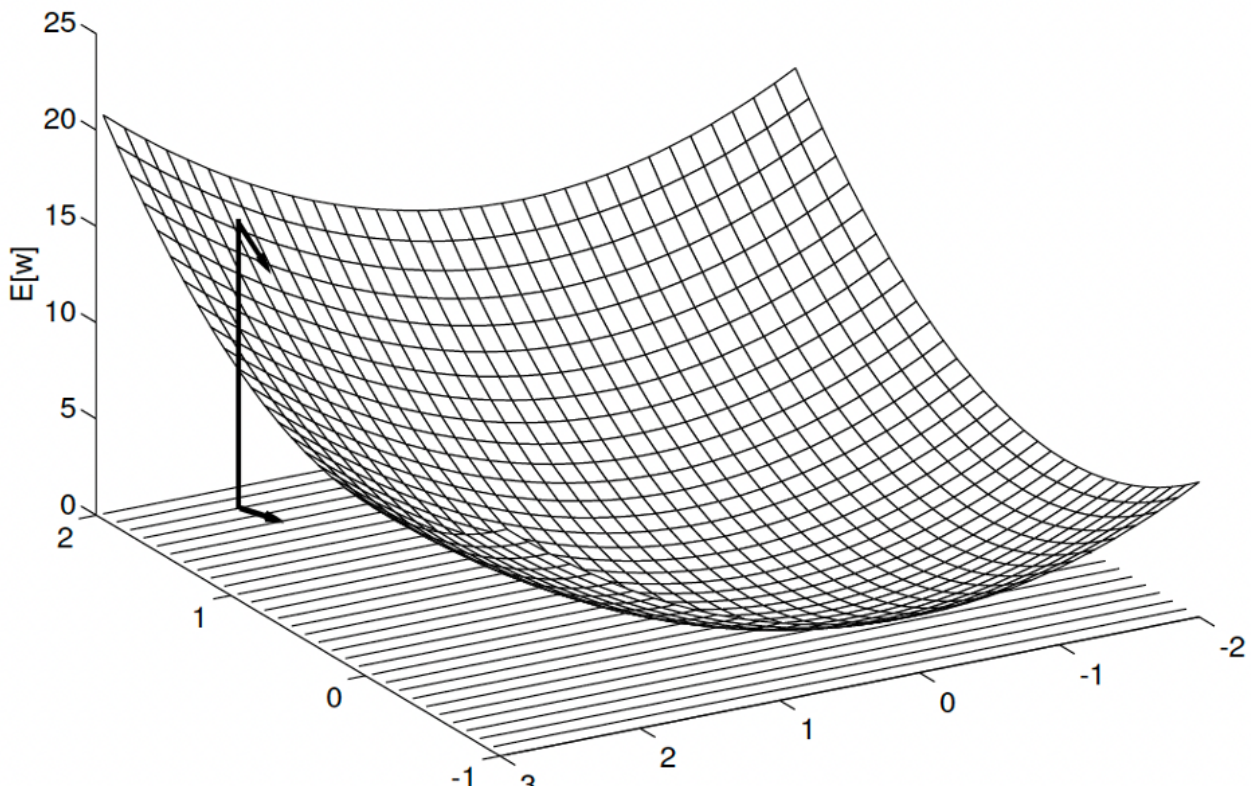
**Interpretation of $E(\vec{w})$:**

- Represents the squared difference between target output ($t_d$) and actual output ($o_d$).
- Summed over all training examples.
- Characterized as a function of the weight vector $\vec{w}$.
- Depends on:
  - The weight vector $\vec{w}$.
  - The specific set of training examples.

# Understanding the Hypothesis Space

**Key Concept:**

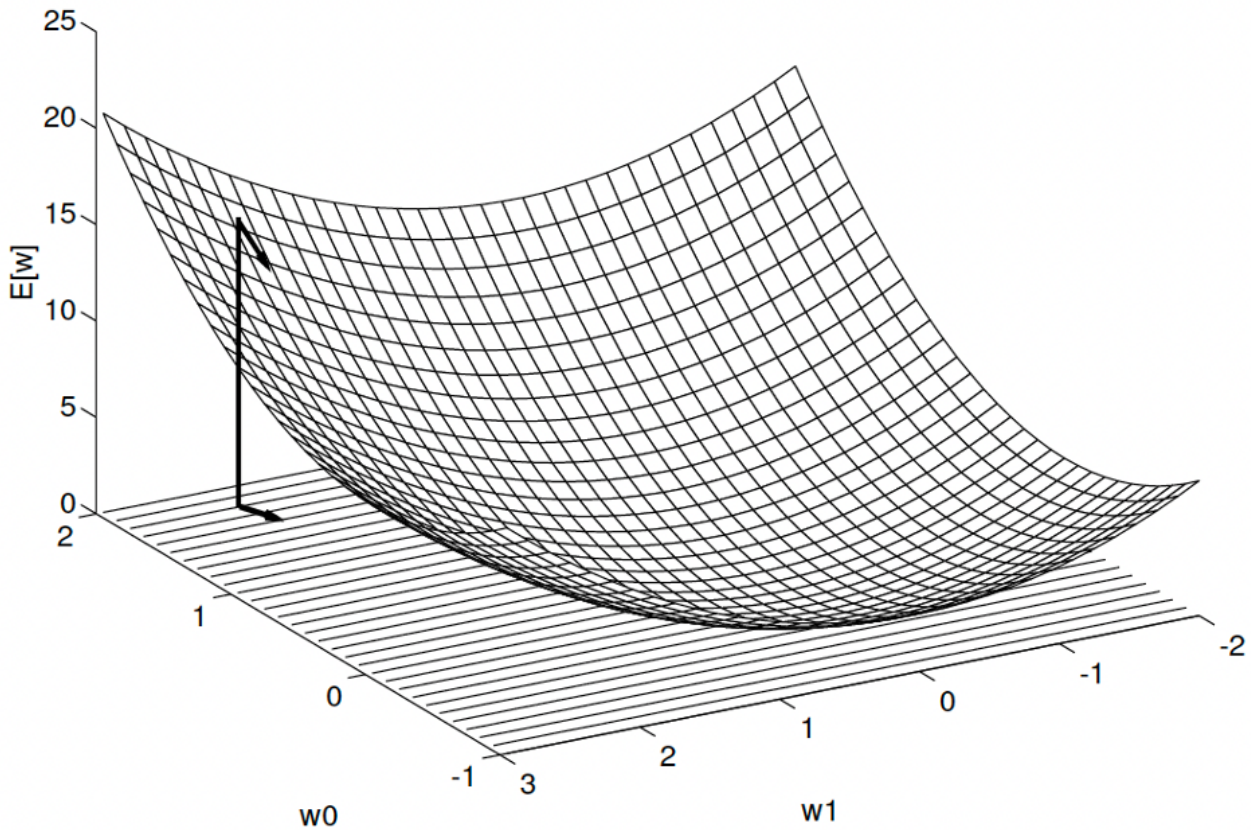
- The hypothesis space consists of all possible weight vectors $(w_0, w_1, \ldots)$ and their associated error values $E[w]$.
- The $w_0, w_1$ plane represents possible values for two weights of a simple linear unit.
- The vertical axis represents the error $E[w]$ relative to a fixed set of training examples.



# Understanding the Error Surface

**Error Surface:**

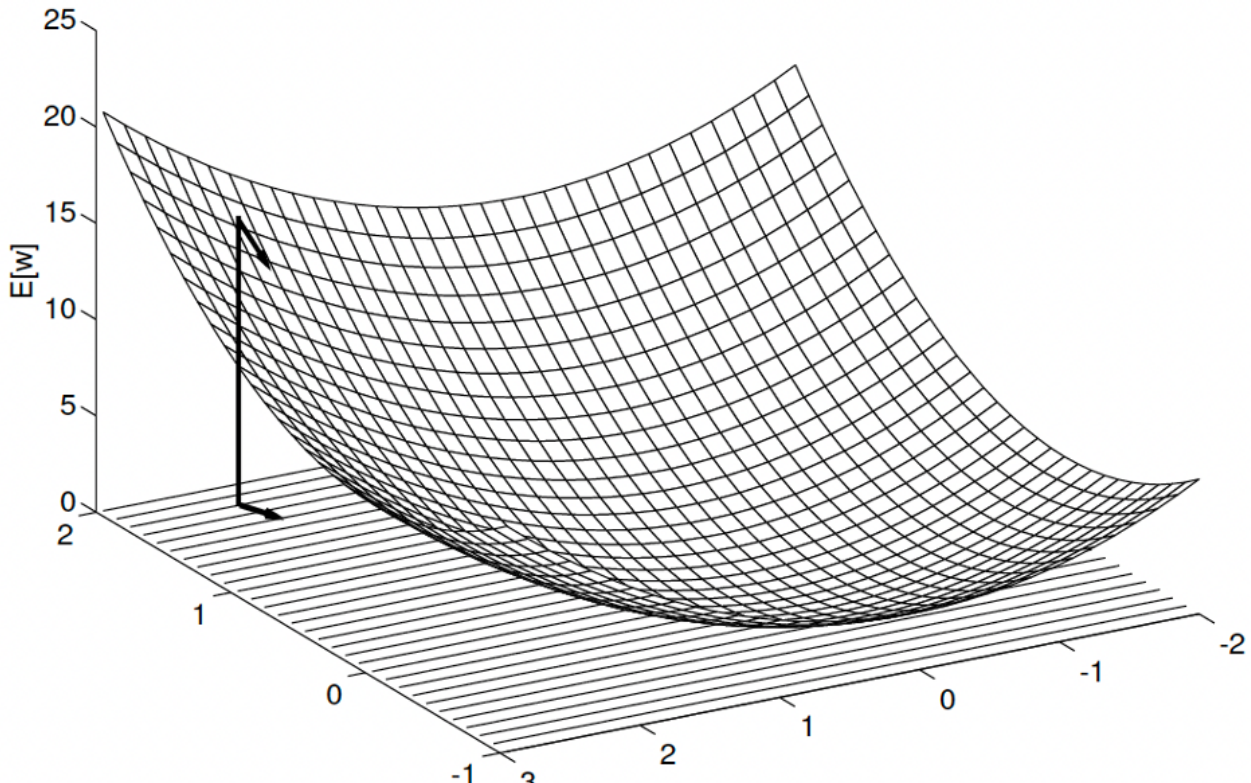
- Visualized as a 3D surface.
- Each point represents the error $E$ for a specific pair of weights $(w_0, w_1)$.
- Goal: Identify the global minimum, where error $E[w]$ is minimized.



# Gradient Descent Algorithm: Overview

**Key Idea:**

- Gradient descent determines the weight vector that minimizes error $E[w]$.
- Starts with an arbitrary initial weight vector and modifies it in small steps.
- At each step, the weight vector moves in the direction of steepest descent along the error surface.



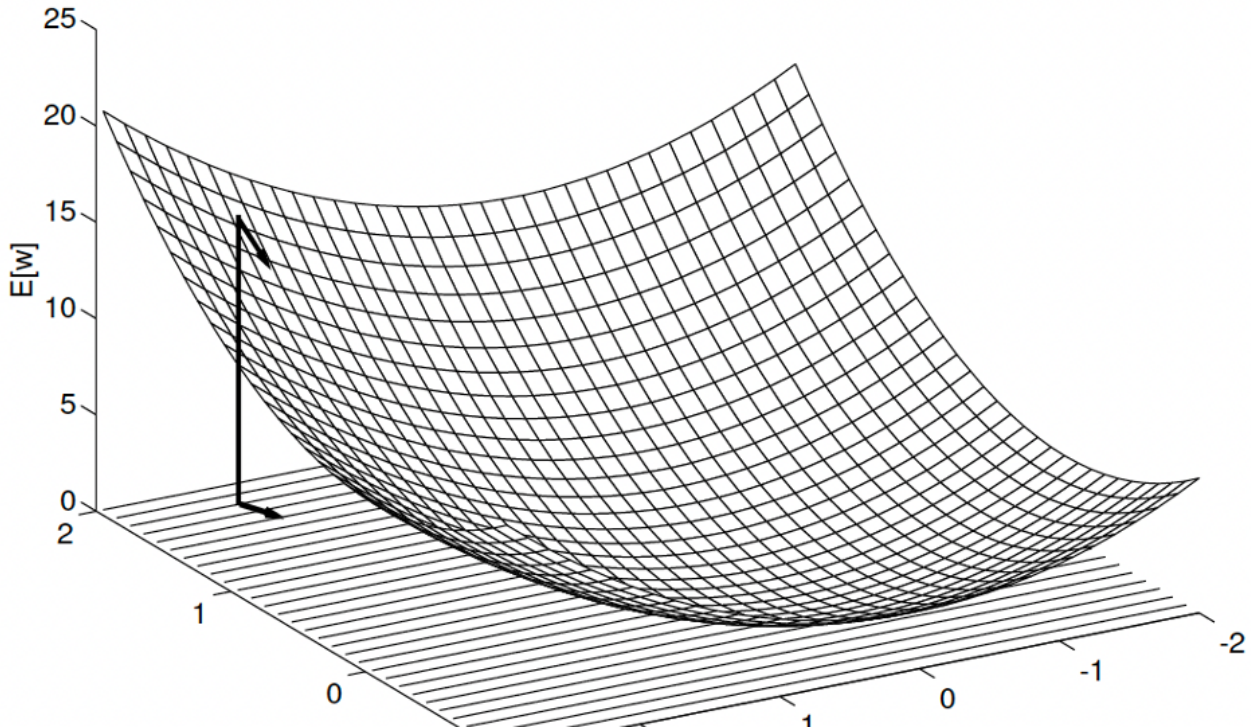
# Gradient Descent Algorithm: Process

**Process:**

1. Compute the gradient of the error surface at the current weight vector.
2. Update weights using:

$$w \leftarrow w - \eta \nabla E[w]$$

   where $\eta$ is the learning rate.
3. Repeat until the global minimum error is reached.



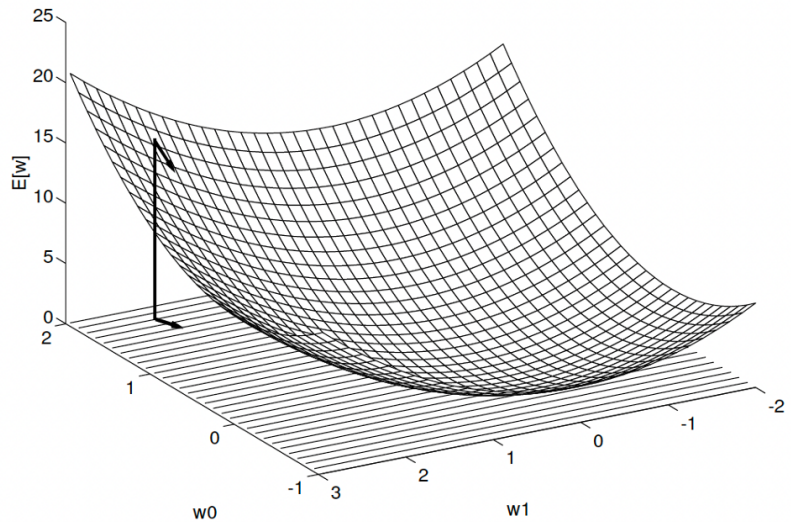
# Derivation of the Gradient Descent Rule

**Gradient:**

$$\nabla E[\vec{w}] \equiv \left[ \frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n} \right]$$

**Key Points:**

- $\nabla E[\vec{w}]$ is a vector of partial derivatives of $E$ with respect to each weight $w_i$.
- In weight space, the gradient points in the direction of the steepest increase in $E$.
- The negative of this gradient points in the direction of the steepest decrease in $E$.




*Note: The gradient specifies the steepest ascent direction. Moving in the negative direction of the gradient decreases $E$.*

# Derivation of the Gradient Descent Rule

**Gradient:**

$$\nabla E[\vec{w}] \equiv \left[ \frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_n} \right]$$

**Training Rule for Gradient Descent:**

$$\vec{w} \leftarrow \vec{w} + \Delta \vec{w}$$

$$\Delta \vec{w} = -\eta \nabla E(\vec{w})$$

**Explanation:**

- $\eta$ is a positive constant called the **learning rate**.
- The negative sign ensures the weight vector moves in the direction that decreases $E$.
- The rule can also be expressed in component form:

$$w_i \leftarrow w_i + \Delta w_i, \quad \Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

- This shows that steepest descent is achieved by altering each component $w_i$ of $\vec{w}$ in proportion to $\frac{\partial E}{\partial w_i}$.

# Derivation of the Gradient Descent Rule

**Error Function:**

$$E[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

**Gradient of the Error Function:**

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \left( \frac{1}{2} \sum_{d} (t_d - o_d)^2 \right)$$

**Where:**

- $E[\vec{w}]$: Error function.
- $t_d$: Target value for data point $d$.
- $o_d$: Output value for data point $d$.
- $w_i$: Weight component for which the gradient is being computed.
- $D$: Dataset.

# Derivation of the Gradient Descent Rule

**Error Function:**

$$E[\vec{w}] \equiv \frac{1}{2}\sum_{d\in D}(t_d - o_d)^2, \quad \text{where } o = w_0 + w_1 x_1 + \cdots + w_n x_n$$

**Computing the Gradient:**

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i}\left(\frac{1}{2}\sum_d (t_d - o_d)^2\right)$$

$$= \frac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2$$

$$= \frac{1}{2}\sum_d 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d)$$

$$= \sum_d (t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x}_d)$$

# Derivation of the Gradient Descent Rule

**Error Function:**

$$E[\vec{w}] \equiv \frac{1}{2}\sum_{d\in D}(t_d - o_d)^2, \quad \text{where } o = w_0 + w_1 x_1 + \cdots + w_n x_n$$

**Computing the Gradient:**

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\left(\frac{1}{2}\sum_d (t_d - o_d)^2\right) \\
&= \frac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_d (t_d - o_d)\frac{\partial}{\partial w_i}(-\vec{w}\cdot\vec{x}_d) \\
&= \sum_d (t_d - o_d)(-x_{i,d})
\end{aligned}$$

**Weight Update Rule:**

$$\Delta w_i = -\eta\frac{\partial E}{\partial w_i}, \quad \Delta w_i = \eta\sum_{d\in D}(t_d - o_d)x_{i,d}$$

# Derivation of the Gradient Descent Rule

**Error Function:**

$$E[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2, \quad \text{where } o = w_0 + w_1 x_1 + \cdots + w_n x_n$$

**Weight Update Rule:**

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}, \quad \Delta w_i = \eta \sum_{d \in D} (t_d - o_d) x_{i,d}$$

**Explanation:**

- $x_{i,d}$: The $i$-th input feature for training example $d$.
- $t_d$: Target output for example $d$.
- $o_d$: Predicted output for example $d$.
- $\eta$: Learning rate (controls the step size).

# Error Function (Step By Step Explanation)

The goal of gradient descent is to minimize the error function:

$$E[\mathbf{w}] = \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

- The $\frac{1}{2}$ is a convenient scaling factor, which simplifies the derivative later.
- $t_d$: Target value for the data point $d$.
- $o_d$: Output (or prediction) from the model for data point $d$.
- $D$: The set of all training data.

This represents the Mean Squared Error (MSE), a common metric in optimization.

# Computing the Gradient (Step By Step Explanation)

To minimize the error, we compute the gradient:

$$\frac{\partial E}{\partial w_i}$$

- This measures how the error changes with respect to the weight $w_i$.
- We aim to adjust $w_i$ to reduce $E$.
- Substituting $E[\mathbf{w}]$ into the derivative:

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i}\left(\frac{1}{2}\sum_d (t_d - o_d)^2\right)$$

# Expanding the Derivative (Step By Step Explanation)

Using the derivative of the error function:

$$\frac{\partial E}{\partial w_i} = \frac{1}{2} \sum_d \frac{\partial}{\partial w_i} (t_d - o_d)^2$$

- The summation sign indicates that we compute the error over all data points $d$.
- The derivative operates on the quadratic error term $(t_d - o_d)^2$.

# Applying the Chain Rule (Step By Step Explanation)

Using the chain rule to expand the quadratic term:

$$\frac{\partial E}{\partial w_i} = \sum_d (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d)$$

- Differentiating the square brings down the factor of 2.
- The $\frac{1}{2}$ from the error function cancels with the 2, simplifying the expression.
- We're left with $(t_d - o_d)$, which measures the difference between the target and the output.

# Differentiating the Output (Step By Step Explanation)

The term $o_d$ (the model's output) depends on the weights $w_i$. Recall:

$$o_d = \mathbf{w} \cdot \mathbf{x}_d$$

The derivative of $o_d$ with respect to $w_i$ is:

$$\frac{\partial o_d}{\partial w_i} = x_{i,d}$$

- $x_{i,d}$: The $i$-th feature of the $d$-th data point.
- This shows that the gradient depends on the input features.

# Substituting Back (Step By Step Explanation)

Substituting $\frac{\partial o_d}{\partial w_i}$ into the gradient equation:

$$\frac{\partial E}{\partial w_i} = \sum_d (t_d - o_d)(-x_{i,d})$$

Simplifying:

$$\frac{\partial E}{\partial w_i} = -\sum_d (t_d - o_d)x_{i,d}$$

- The negative sign indicates that we move weights in the opposite direction of the gradient to minimize the error.

# Weight Update Rule (Step By Step Explanation)

Using the gradient, the weight update rule is:

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

Substitute the gradient:

$$\Delta w_i = \eta \sum_d (t_d - o_d) x_{i,d}$$

- $\eta$: Learning rate, which controls the step size for updates.
- This update moves the weights to reduce the error $E$ in each iteration.

# Derivation of the Gradient Descent Rule

**Weight Update Rule:**

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}, \quad \Delta w_i = \eta \sum_{d \in D} (t_d - o_d) x_{i,d}$$

**Explanation:**

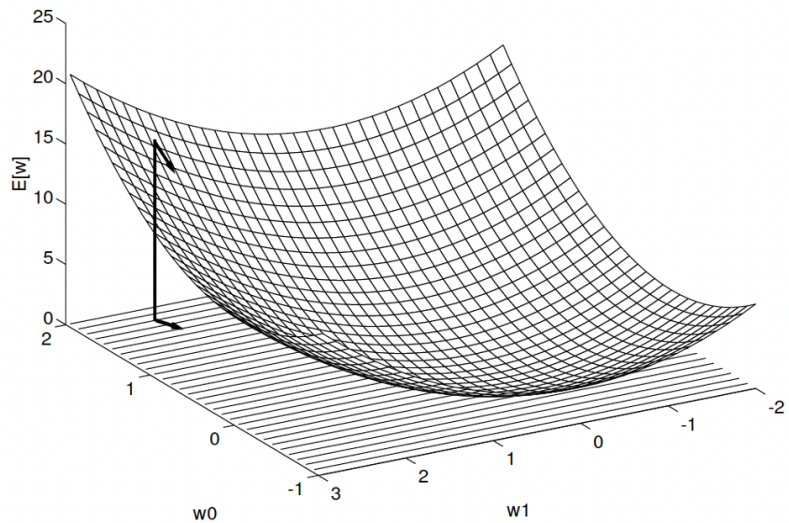
- $x_{i,d}$: The $i$-th input feature for training example $d$.
- $t_d$: Target output for example $d$.
- $o_d$: Predicted output for example $d$.
- $\eta$: Learning rate (controls the step size).

**The weight update formula links the gradient with the linear unit inputs $x_{i,d}$, outputs $o_d$, and target values $t_d$ for all training examples, enabling gradient descent optimization.**

# Perceptron Learning Algorithm

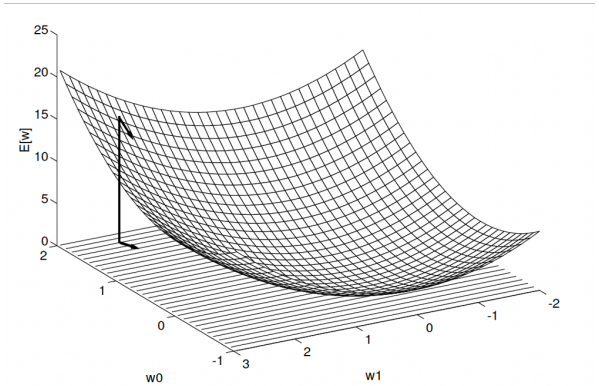
**Gradient-Descent(training_examples, $\eta$)**

- *Each training example is a pair of the form $(\vec{x}, t)$, where $\vec{x}$ is the vector of input values, and $t$ is the target output value. $\eta$ is the learning rate (e.g., 0.05).*
- Initialize each $w_i$ to some small random value.
- Until the termination condition is met, Do:
  - Initialize each $\Delta w_i$ to zero.
  - For each $(\vec{x}, t)$ in *training_examples*, Do:
    - Input the instance $\vec{x}$ to the unit and compute the output $o$.
    - For each linear unit weight $w_i$, Do:
    - $\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$
  - For each linear unit weight $w_i$, Do:
    - $w_i \leftarrow w_i + \Delta w_i$



# Perceptron Learning Algorithm

**Gradient-Descent(training_examples, $\eta$)**




**Key Points:**

- Since the error surface contains only a single global minimum, the algorithm will converge to a weight vector with minimum error.
- This convergence is guaranteed regardless of whether the training examples are linearly separable, provided a sufficiently small learning rate $\eta$ is used.
- If the learning rate $\eta$ is too large, the gradient descent search may overshoot the minimum in the error surface rather than settling into it.
- To address this, a common modification to the algorithm is to gradually reduce the value of $\eta$ as the number of gradient descent steps increases.

# Batch Mode vs. Incremental Mode Gradient Descent

**Batch Mode Gradient Descent**

- Processes the **entire dataset** $D$ at once.
- Computes the gradient:

$$\nabla E_D[\vec{w}] = \frac{\partial}{\partial \vec{w}} \left( \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \right)$$

- Weight update:

$$\vec{w} \leftarrow \vec{w} - \eta \nabla E_D[\vec{w}]$$

- **Advantages:**
  - Stable and precise updates.
  - Converges smoothly to the minimum.
- **Disadvantages:**
  - Computationally expensive for large datasets.
  - Requires processing all data before updating.

**Incremental Mode (Stochastic) Gradient Descent**

- Processes **one example at a time**.
- Computes the gradient:

$$\nabla E_d[\vec{w}] = \frac{\partial}{\partial \vec{w}} \left( \frac{1}{2} (t_d - o_d)^2 \right)$$

- Weight update:

$$\vec{w} \leftarrow \vec{w} - \eta \nabla E_d[\vec{w}]$$

- **Advantages:**
  - Faster updates for large datasets.
  - Adapts quickly to changes in data.
- **Disadvantages:**
  - Noisy updates—can "bounce" around the minimum.
  - May take longer to converge overall.

# Batch Mode vs. Incremental Mode Gradient Descent

**Batch Mode Gradient Descent**

- Processes the **entire dataset** $D$ at once.
- Computes the gradient:

$$\nabla E_D[\vec{w}] = \frac{\partial}{\partial \vec{w}} \left( \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \right)$$

- Weight update:

$$\vec{w} \leftarrow \vec{w} - \eta \nabla E_D[\vec{w}]$$

**Incremental Mode (Stochastic) Gradient Descent**

- Processes **one example at a time**.
- Computes the gradient:

$$\nabla E_d[\vec{w}] = \frac{\partial}{\partial \vec{w}} \left( \frac{1}{2} (t_d - o_d)^2 \right)$$

- Weight update:

$$\vec{w} \leftarrow \vec{w} - \eta \nabla E_d[\vec{w}]$$

**Key Insight:**

- Incremental Gradient Descent can approximate Batch Gradient Descent if the learning rate $\eta$ is small enough and the dataset $D$ is large.

# Comparison of Perceptron and Delta Rule

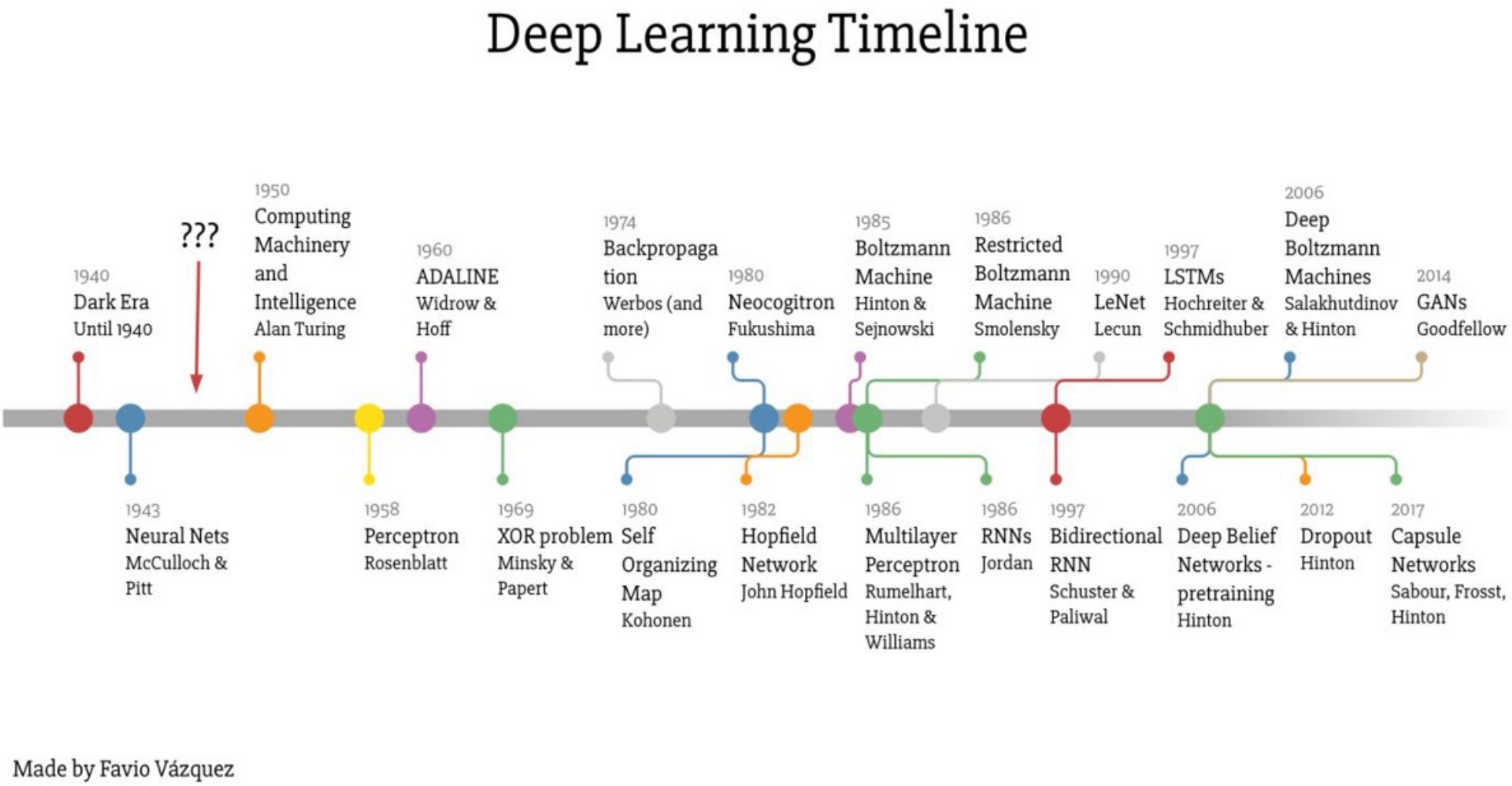
**Key Differences:**

- **Perceptron Training Rule:**
  - Updates weights based on the error in the **thresholded perceptron output**. Converges to a hypothesis that perfectly classifies the data if and only if the training examples are **linearly separable**. Finite number of iterations required for convergence.
- **Delta Rule:**
  - Updates weights based on the error in the **unthresholded linear combination of inputs**. Converges asymptotically to the **minimum error hypothesis**, even if the data is not linearly separable. May require unbounded time for convergence.

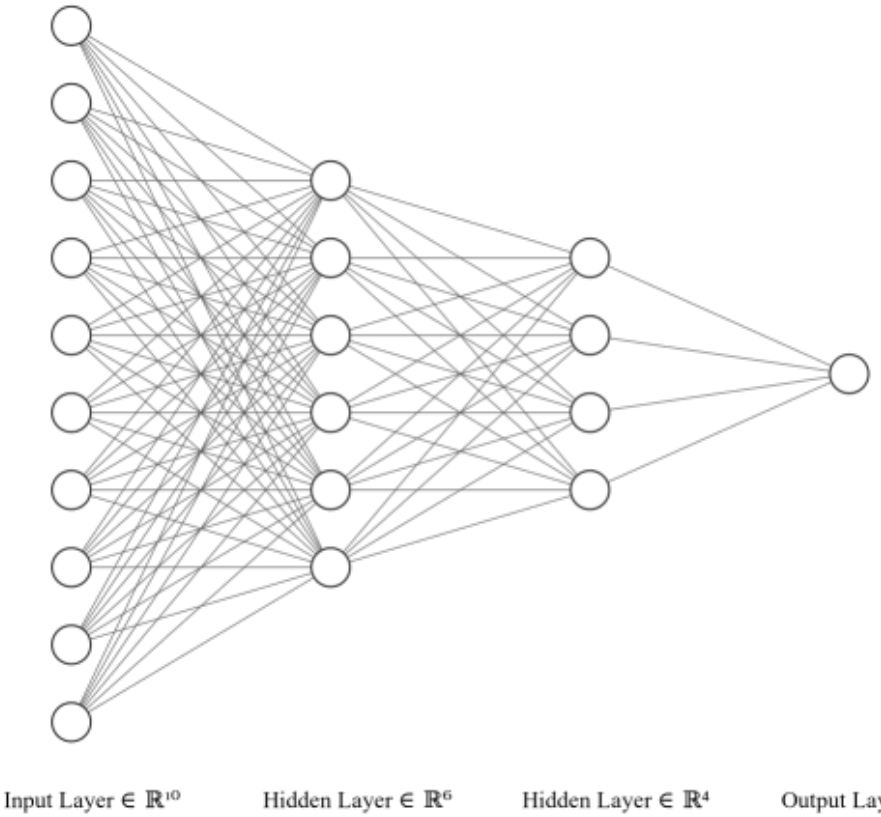
**Alternative Approach: Linear Programming**

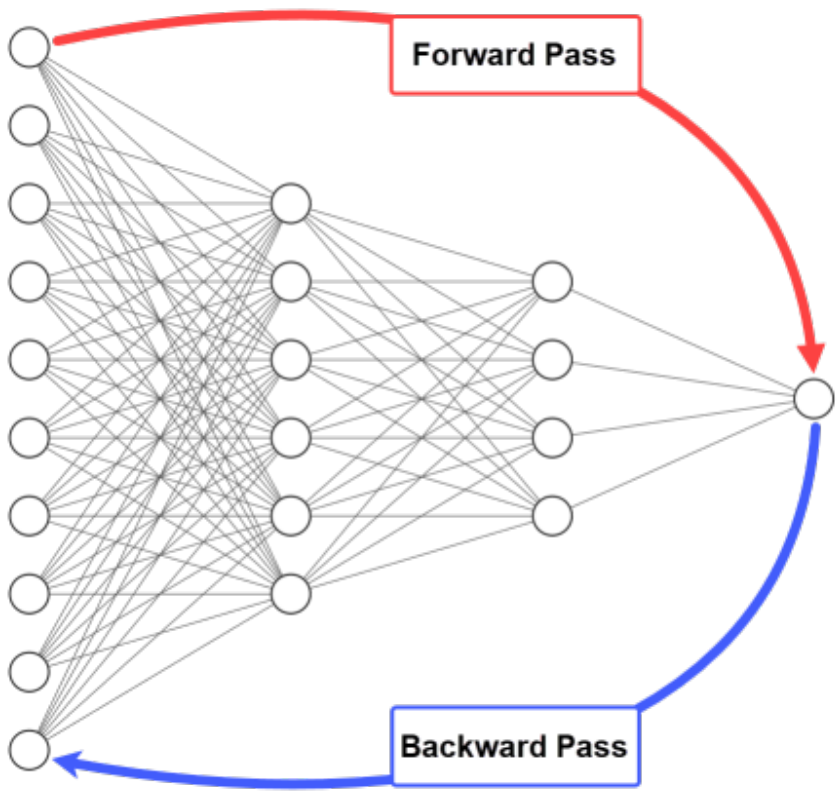
- (such as Simplex Algorithm and Interior-Point Methods– taught in linear algebra or numerical analysis course)
- Solves sets of linear inequalities ($\vec{w} \cdot \vec{x} > 0$ or $\vec{w} \cdot \vec{x} \leq 0$). Effective for linearly separable cases but does not scale well to multilayer networks. Gradient descent, used in the delta rule, extends naturally to multilayer networks. **Reference:** Hertz et al. (1991), Duda and Hart (1973)

# Notice 1974: Backpropogation



# Forward and backward passes in Neural Networks




**Note:** The notation $\mathbb{R}^n$ represents an $n$-dimensional vector space of real numbers. For example, $\mathbb{R}^{10}$ indicates the input layer with 10 real-valued features, $\mathbb{R}^6$ and $\mathbb{R}^4$ represent hidden layers with 6 and 4 neurons respectively, and $\mathbb{R}$ indicates the output layer with a single real-valued output. This compact notation describes the structure of the network and the dimensions of its layers.

# Multi-Layer Networks of Sigmoid Units

**Boolean Functions:**

- Every Boolean function can be represented by a network with a single hidden layer.
- However, this might require an exponential number (in terms of the number of inputs) of hidden units.
- Every bounded continuous function can be approximated with arbitrarily small error by a network with one hidden layer, provided the hidden layer has enough neurons and an appropriate activation function (e.g., sigmoid). Key References: Cybenko (1989) Hornik et al. (1989)
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers. Key Reference: Cybenko (1988)

**Key Insights:**

- Networks with sigmoid activation units are versatile and capable of approximating both Boolean and continuous functions.
- The number of required hidden units and layers depends on the type of function and the desired accuracy.

# Universal Approximation with One Hidden Layer

**Key Result:**

- A neural network with **one hidden layer** can approximate any **bounded continuous function** on a compact domain with arbitrary accuracy.
- Requires:
  - A sufficient number of hidden neurons.
  - A suitable non-linear activation function (e.g., sigmoid).

**References:**

- Cybenko (1989): Proved the **Universal Approximation Theorem** for single hidden layer networks.
- Hornik et al. (1989): Refined Cybenko's work, establishing theoretical robustness for broader classes of activation functions.

**Limitations:**

- For complex functions, the number of neurons required may grow **exponentially** with the number of inputs, making this approach inefficient.

# Two Hidden Layers for Broader Function Classes

**Key Result:**

- A neural network with **two hidden layers** can approximate **any function** (including non-continuous functions) with arbitrary accuracy.

**Why Two Hidden Layers?**

- Extends beyond continuous functions to handle:
  - Piecewise functions.
  - Discontinuous functions.
- Enables more **efficient representations**, especially for complex functions where a single hidden layer might require an exponential number of neurons.

**Reference:**

- Cybenko (1988): Demonstrated the efficiency and versatility of networks with two hidden layers for approximating complex function classes.

**Implication:**

- While a single hidden layer is sufficient for continuous functions, **two hidden layers improve efficiency and applicability to non-continuous cases**.

# Practical Implications of 1988 and 1989 Results

**Key Takeaways:**

- **One Hidden Layer:**
  - Sufficient for approximating bounded continuous functions (1989).
  - May require an **exponential number of neurons** for complex cases.
- **Two Hidden Layers:**
  - Extends to all functions, including non-continuous ones (1988).
  - More efficient for representing complex functions, reducing the need for a large number of neurons.

**Universal Approximation Theorem:**

- Single hidden layers establish sufficiency for approximation.
- Depth (two or more layers) offers efficiency and versatility in practice.

# Practical Implications of 1988 and 1989 Results

**Key Takeaways:**

- **One Hidden Layer:**
  - Sufficient for approximating bounded continuous functions (1989).
  - May require an **exponential number of neurons** for complex cases.
- **Two Hidden Layers:**
  - Extends to all functions, including non-continuous ones (1988).
  - More efficient for representing complex functions, reducing the need for a large number of neurons.

**Universal Approximation Theorem:**

- Single hidden layers establish sufficiency for approximation.
- Depth (two or more layers) offers efficiency and versatility in practice but inference may require more latency.

# Boolean Functions with a Single Hidden Layer

**Representation of Boolean Functions:**

- Every Boolean function can be represented by a network with a single hidden layer.
- The number of neurons required depends on:
  - The number of inputs ($n$).
  - The complexity of the Boolean function.

**Neuron Requirement:**

- For $n$ inputs:
  - Up to $2^n$ neurons in the hidden layer may be needed for the most complex Boolean functions.
  - Simple functions like AND/OR require fewer neurons.
  - Complex functions (e.g., XOR) require more neurons:
    - XOR for 2 inputs: 2 neurons.
    - XOR for $n$ inputs: $n$ neurons.
- The exponential growth ($2^n$) makes it impractical for large $n$.

# Efficiency of Deep Networks vs. Shallow Networks

**Why Deep Networks Require Fewer Neurons:**

- **Hierarchical Representation:**

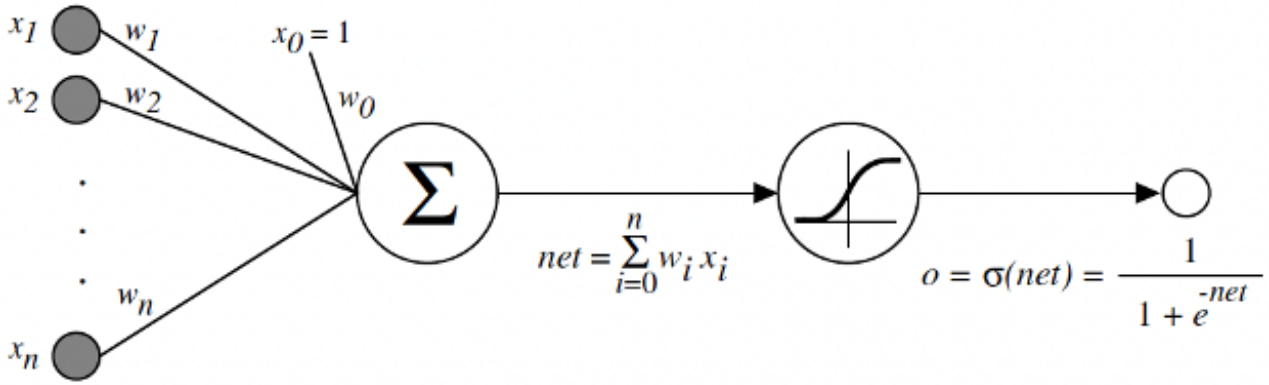
  Deep networks represent complex functions as compositions of simpler functions. Each layer learns intermediate abstractions, reducing redundancy.

- **Reusability of Features:**
  - Neurons in lower layers detect reusable features (e.g., edges, shapes).
  - Higher layers build complex structures from these shared features.
- **Empirical Observations:** Deep architectures approximate functions with exponentially fewer neurons.

**Theoretical Justification:**

- **Delalleau & Bengio (2011):** Deep networks with sigmoid activations represent polynomials more efficiently. Functions needing $O(2^n)$ neurons in shallow networks can use $O(n)$ neurons in deep networks.
- **Montufar et al. (2014):** Expressivity of deep networks grows exponentially with depth.

# Sigmoid Unit




- $\sigma(x)$ is the sigmoid function:

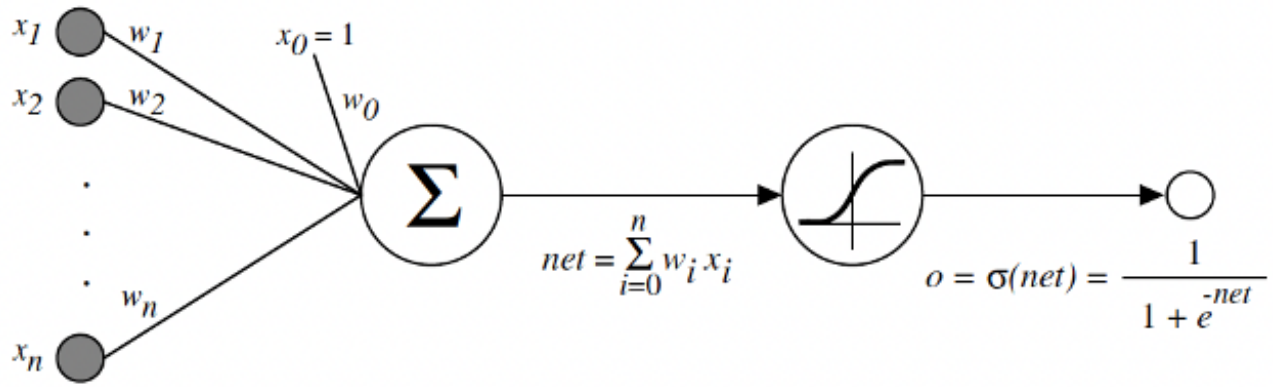
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- **Nice Property:**

$$\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$$

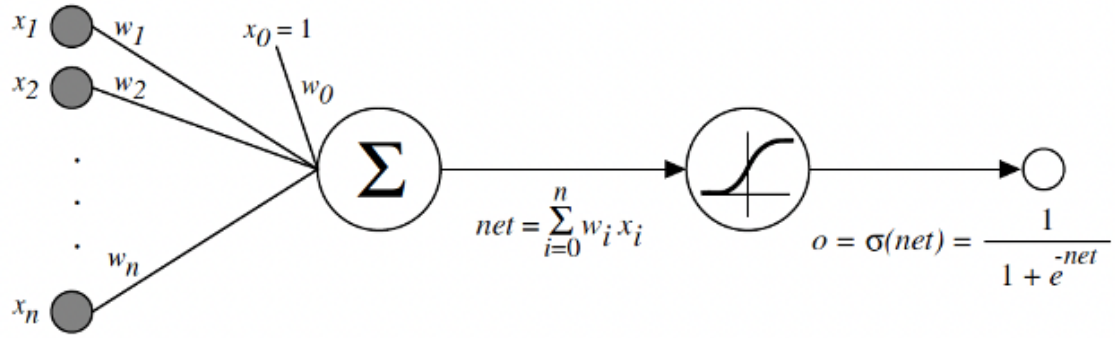
- We can derive gradient descent rules to train:
  - One sigmoid unit
  - *Multilayer networks* of sigmoid units $\rightarrow$ Backpropagation

# Error Gradient for a Sigmoid Unit




$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

# Error Gradient for a Sigmoid Unit




$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

$$= \frac{1}{2} \sum_{d} \frac{\partial}{\partial w_i} (t_d - o_d)^2$$

$$= \frac{1}{2} \sum_{d} 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d)$$

# Error Gradient for a Sigmoid Unit

**Sigmoid Unit:**

$$o = \sigma(\text{net}) = \frac{1}{1 + e^{-\text{net}}}, \quad \text{where net} = \sum_{i=0}^{n} w_i x_i$$

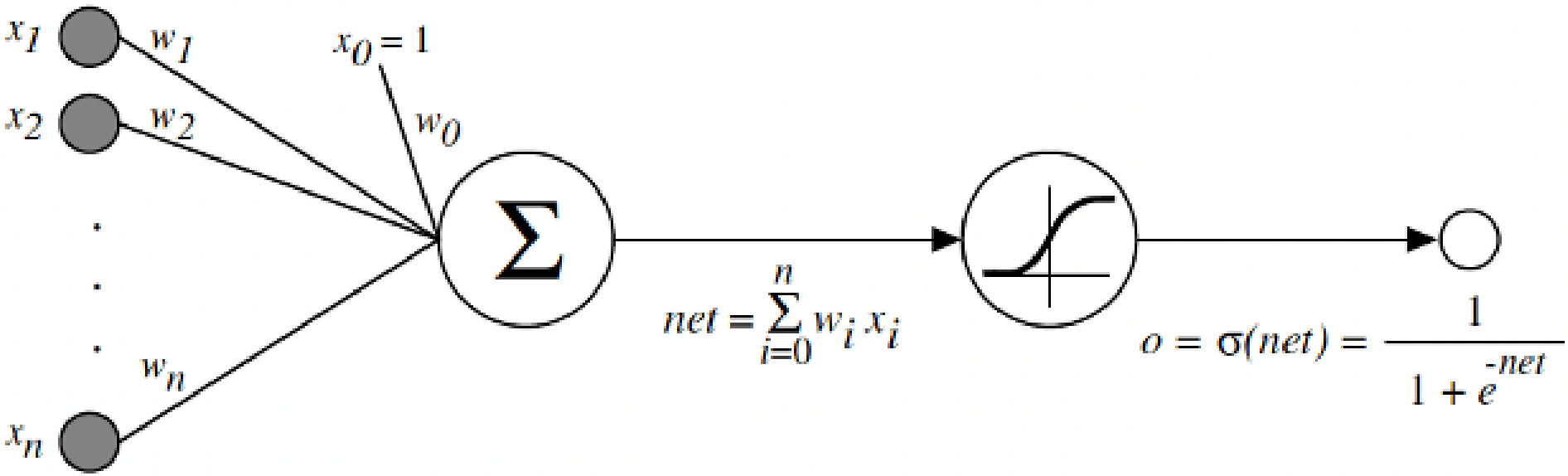
**Error Gradient:**

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 = \frac{1}{2} \sum_{d} \frac{\partial}{\partial w_i} (t_d - o_d)^2 = \sum_{d} (t_d - o_d) \left( -\frac{\partial o_d}{\partial w_i} \right)$$

**Where:**

- $t_d$: Target value for data point $d$.
- $o_d$: Output value for data point $d$, computed by the sigmoid activation function.
- $\frac{\partial o_d}{\partial w_i}$: The derivative of the sigmoid output with respect to weight $w_i$.

# Notice the Net and Output Value



# Reminder: Chain Rule

**What is the Chain Rule?**

The chain rule is a fundamental principle in calculus used to compute the derivative of a composite function.

$$\text{If } y = f(g(x)), \quad \text{then } \frac{dy}{dx} = \frac{df}{dg} \cdot \frac{dg}{dx}$$

**Example:**

$$h(x) = (3x + 1)^2$$

1. Let $u = g(x) = 3x + 1$ (inner function).
2. Let $h(x) = f(u) = u^2$ (outer function).

Using the chain rule:

$$\frac{dh}{dx} = \frac{df}{du} \cdot \frac{dg}{dx}$$

$$\frac{df}{du} = 2u, \quad \frac{dg}{dx} = 3$$

$$\therefore \frac{dh}{dx} = 2(3x + 1) \cdot 3 = 6(3x + 1)$$

# Error Gradient for a Sigmoid Unit

**Sigmoid Unit:**

$$o = \sigma(\text{net}) = \frac{1}{1 + e^{-\text{net}}}, \quad \text{where net} = \sum_{i=0}^{n} w_i x_i$$

**Error Gradient Derivation:**

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 = \frac{1}{2} \sum_d \frac{\partial}{\partial w_i} (t_d - o_d)^2 = \sum_d (t_d - o_d) \left( -\frac{\partial o_d}{\partial w_i} \right) =$$

$$\sum_d (t_d - o_d) \left( -\frac{\partial o_d}{\partial \text{net}_d} \frac{\partial \text{net}_d}{\partial w_i} \right) = -\sum_d (t_d - o_d) \frac{\partial o_d}{\partial \text{net}_d} \frac{\partial \text{net}_d}{\partial w_i}$$

**Where:**

- $t_d$: Target value for data point $d$.
- $o_d$: Output value for data point $d$, computed by the sigmoid activation function.
- $\text{net}_d = \sum_{i=0}^{n} w_i x_i$: Weighted sum of inputs for data point $d$.
- $\frac{\partial o_d}{\partial \text{net}_d}$: Derivative of sigmoid function with respect to $\text{net}_d$.
- $\frac{\partial \text{net}_d}{\partial w_i} = x_{di}$: Partial derivative of $\text{net}_d$ with respect to weight $w_i$.

# Error Gradient for a Sigmoid Unit

**Sigmoid Unit:**

$$o = \sigma(\text{net}) = \frac{1}{1 + e^{-\text{net}}}, \quad \text{where net} = \sum_{i=0}^{n} w_i x_i$$

**Error Gradient Derivation:**

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 = \frac{1}{2} \sum_d \frac{\partial}{\partial w_i} (t_d - o_d)^2 = \sum_d (t_d - o_d) \left( -\frac{\partial o_d}{\partial w_i} \right)$$

$$= \sum_d (t_d - o_d) \left( -\frac{\partial o_d}{\partial \text{net}_d} \frac{\partial \text{net}_d}{\partial w_i} \right) = -\sum_d (t_d - o_d) \frac{\partial o_d}{\partial \text{net}_d} \frac{\partial \text{net}_d}{\partial w_i}$$

**But we know:**

$$\frac{\partial o_d}{\partial \text{net}_d} = \frac{\partial \sigma(\text{net}_d)}{\partial \text{net}_d} = o_d(1 - o_d)$$

# Error Gradient for a Sigmoid Unit

**Sigmoid Unit:**

$$o = \sigma(\text{net}) = \frac{1}{1 + e^{-\text{net}}}, \quad \text{where net} = \sum_{i=0}^{n} w_i x_i$$

**Nice Property:**

$$\frac{d\sigma(x)}{dx} = \sigma(x) \cdot (1 - \sigma(x))$$

**Error Gradient Derivation:**

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 = \sum_{d} (t_d - o_d) \left( -\frac{\partial o_d}{\partial \text{net}_d} \cdot \frac{\partial \text{net}_d}{\partial w_i} \right)$$

**But we know:**

$$\frac{\partial o_d}{\partial \text{net}_d} = o_d(1 - o_d), \quad \frac{\partial \text{net}_d}{\partial w_i} = x_{i,d}$$

# Error Gradient for a Sigmoid Unit

**Error Gradient Derivation:**

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{d\in D}(t_d - o_d)^2 = \frac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2 = \sum_d (t_d - o_d)\left(-\frac{\partial o_d}{\partial w_i}\right)$$

$$= \sum_d (t_d - o_d)\left(-\frac{\partial o_d}{\partial \text{net}_d}\frac{\partial \text{net}_d}{\partial w_i}\right) = -\sum_d (t_d - o_d)\frac{\partial o_d}{\partial \text{net}_d}\frac{\partial \text{net}_d}{\partial w_i}$$

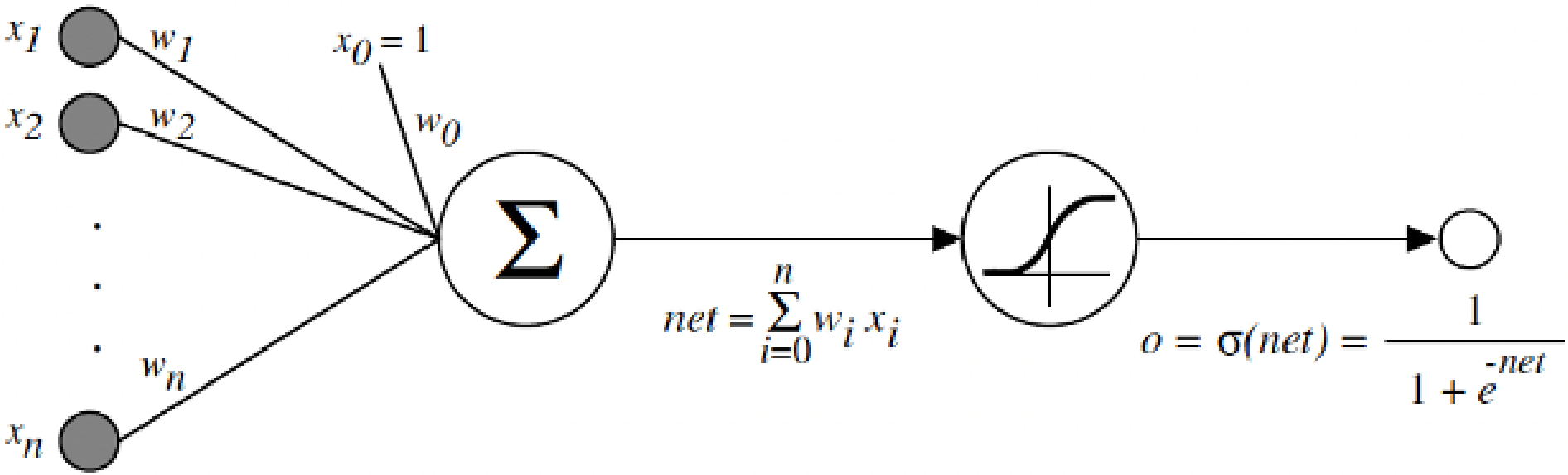
**But we know: (remember the nice property)**

$$\frac{\partial o_d}{\partial \text{net}_d} = \frac{\partial \sigma(\text{net}_d)}{\partial \text{net}_d} = o_d(1 - o_d), \quad \frac{\partial \text{net}_d}{\partial w_i} = \frac{\partial(\vec{w}\cdot\vec{x}_d)}{\partial w_i} = x_{i,d}$$

**So:**

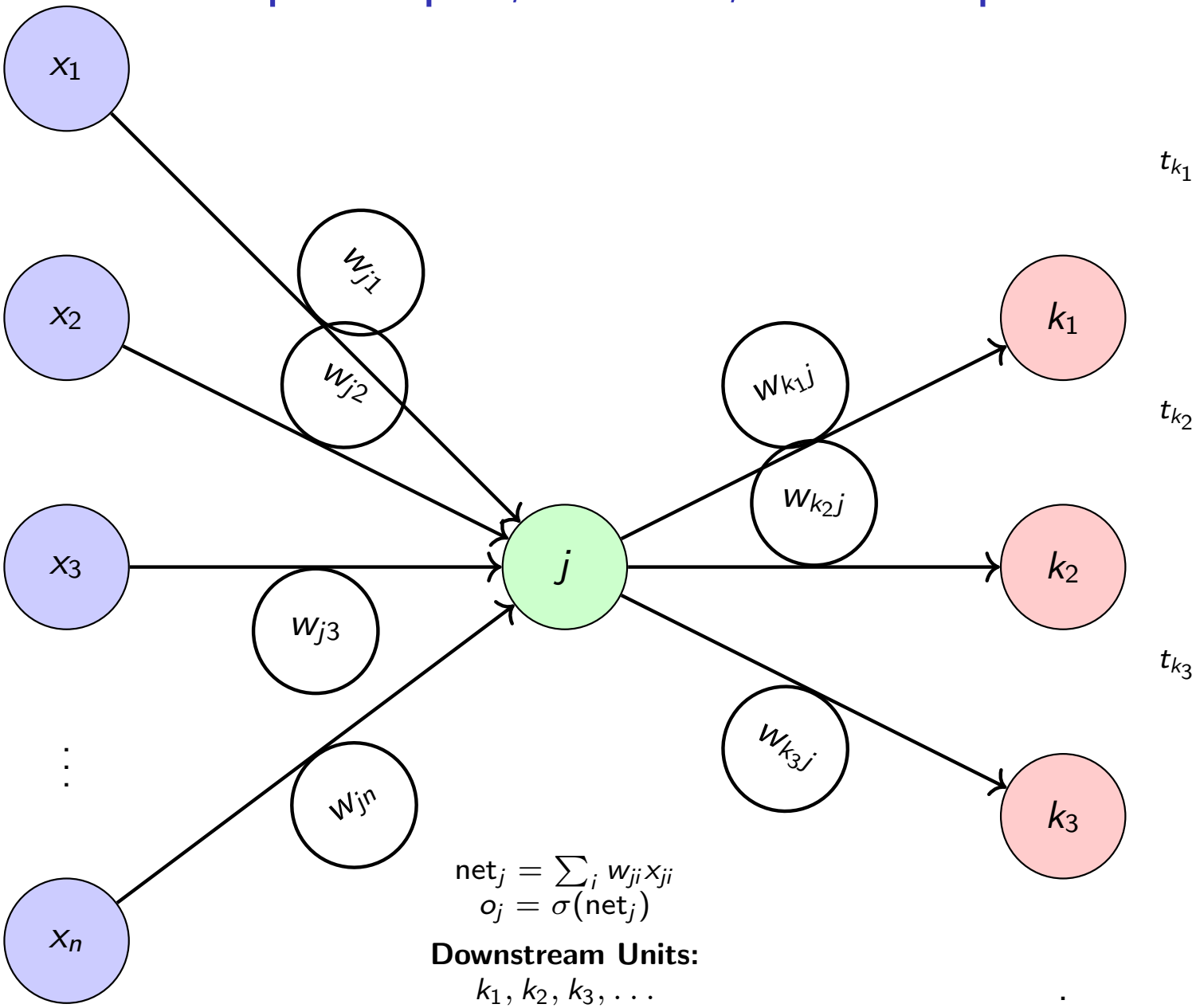
$$\frac{\partial E}{\partial w_i} = -\sum_{d\in D}(t_d - o_d)o_d(1 - o_d)x_{i,d}$$

# We derived the training algorithm for perceptron and sigmoid unit so far



# Derivation of the Backpropagation Rule for NN

# Extended Graph: Input, Hidden, and Output Neurons




$\text{net}_j = \sum_i w_{ji} x_{ji}$

$o_j = \sigma(\text{net}_j)$

**Downstream Units:**

$k_1, k_2, k_3, \ldots$

# Derivation of the Backpropagation Rule

**Objective:** Derive the stochastic gradient descent weight-tuning rule for a neural network.

**Approach:**

- For each training example $d$, compute the error gradient $E_d$.
- Update weights $w_{ji}$ using:

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$$

**Error Function:**

$$E_d(\vec{w}) = \frac{1}{2} \sum_{k \in \text{outputs}} (t_k - o_k)^2$$

where:

- $t_k$: Target value of unit $k$ for training example $d$.
- $o_k$: Output of unit $k$ for training example $d$.
- outputs: Set of output units in the network.

# Definitions and Notation

**Notation:**

- $x_{ji}$: $i$-th input to unit $j$.
- $w_{ji}$: Weight associated with $x_{ji}$.
- $\text{net}_j = \sum_i w_{ji} x_{ji}$: Weighted sum of inputs for unit $j$.
- $o_j = \sigma(\text{net}_j)$: Output computed by unit $j$.
- $t_j$: Target output for unit $j$.
- $\sigma$: Sigmoid activation function.
- outputs: Set of units in the final layer of the network.
- Downstream($j$): Set of units whose inputs include the output of unit $j$.

**Objective:** Find an expression for $\frac{\partial E_d}{\partial w_{ji}}$.

# Using the Chain Rule

**Gradient Decomposition:**

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial \text{net}_j} \cdot \frac{\partial \text{net}_j}{\partial w_{ji}}$$

$$\frac{\partial \text{net}_j}{\partial w_{ji}} = x_{ji}$$

$\text{net}_j = \sum_i w_{ji} x_{ji}$: Weighted sum of inputs for unit $j$.

**Simplification:** To derive $\frac{\partial E_d}{\partial \text{net}_j}$, we consider two cases:

1. When unit $j$ is an **output** unit.
2. When unit $j$ is a **hidden** unit.

# Case 1: Training Rule for Output Units

**Key Idea:** The error $E_d$ influences the network only through $\text{net}_j$, and $\text{net}_j$ influences the network only through $o_j$.

**Chain Rule:**

$$\frac{\partial E_d}{\partial \text{net}_j} = \frac{\partial E_d}{\partial o_j} \cdot \frac{\partial o_j}{\partial \text{net}_j}$$

**Components:**

$$\frac{\partial E_d}{\partial o_j} = -(t_j - o_j)$$

$$\frac{\partial o_j}{\partial \text{net}_j} = o_j(1 - o_j)$$

**Gradient:**

$$\frac{\partial E_d}{\partial \text{net}_j} = -(t_j - o_j)o_j(1 - o_j)$$

# Weight Update for Output Units

**Weight Update Rule:**

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$$

$$\Delta w_{ji} = \eta (t_j - o_j) o_j (1 - o_j) x_{ji}$$

**Summary:**

- Output units directly compare their outputs to the target values.
- Weight updates depend on the gradient $-(t_j - o_j) o_j (1 - o_j)$.

# Backpropagation Algorithm

**Algorithm:**

- Initialize all weights to small random numbers.
- Until satisfied, Do:
  - For each training example, Do:
    1. Input the training example to the network and compute the network outputs.
    2. For each output unit $k$:

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

    3. For each hidden unit $h$:

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in \text{outputs}} w_{h,k}\delta_k$$

    4. Update each network weight $w_{i,j}$:

$$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$$

where

$$\Delta w_{i,j} = \eta \delta_j x_{i,j}$$

# Summary of Hidden and Output Units

**Key Differences Between Hidden and Output Units:**

- **Hidden Units ($j$):**

$$\delta_j = o_j(1 - o_j) \sum_{k \in \text{Downstream}(j)} \delta_k w_{kj}$$

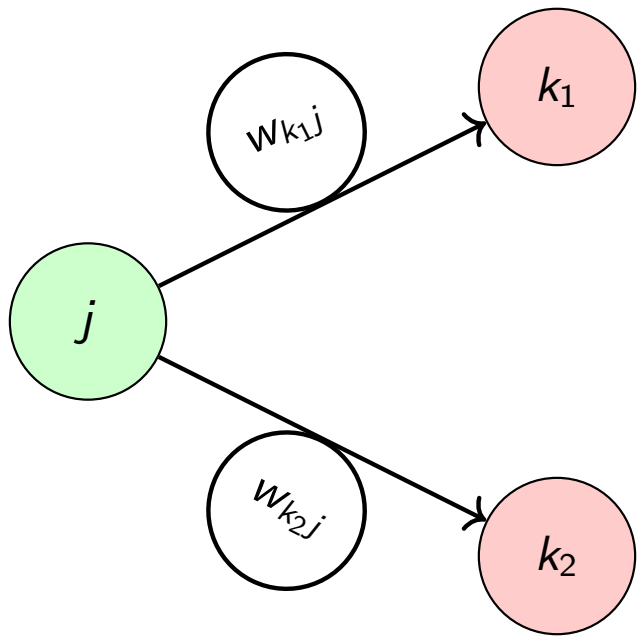
- **Output Units ($k$):**

$$\delta_k = (t_k - o_k)o_k(1 - o_k)$$

- Hidden units rely on backpropagated errors from downstream units.
- Output units compute gradients directly by comparing their output to the target.

# Key Idea: Hidden Units Influence Error Indirectly

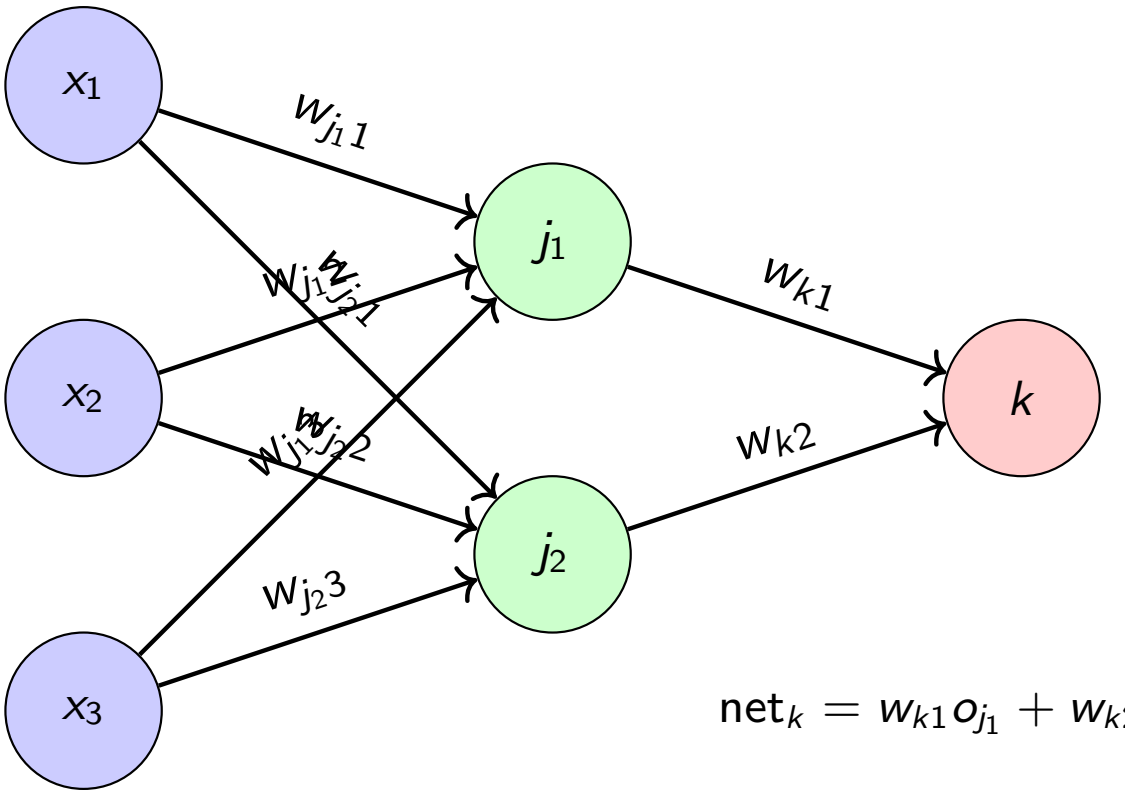
**Concept:** Hidden units ($j$) influence the network error indirectly via downstream units ($k$).
Hidden unit $j$ connects to downstream units $k_1$ and $k_2$ through weights $w_{k_1 j}$ and $w_{k_2 j}$.




Hidden units do not directly compare their output to target values but influence error propagation through connections to downstream units.

# Simple Neural Network with $\text{net}_j$ and $\text{net}_k$

$$\text{net}_{j_1} = w_{j_1 1} x_1 + w_{j_1 2} x_2 + w_{j_1 3} x_3 + b_{j_1}$$
$$\text{net}_{j_2} = w_{j_2 1} x_1 + w_{j_2 2} x_2 + w_{j_2 3} x_3 + b_{j_2}$$

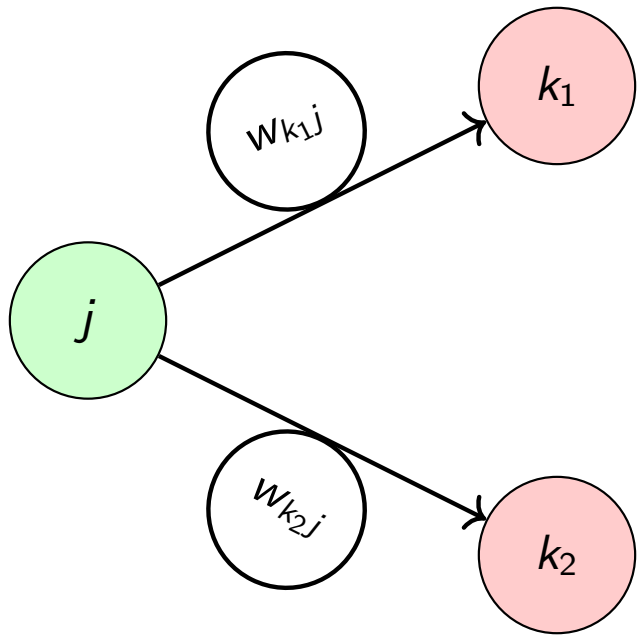



$$\text{net}_k = w_{k1} o_{j_1} + w_{k2} o_{j_2} + b_k$$

# Chain Rule for Gradient Computation

**Formula:**

$$\frac{\partial E_d}{\partial \text{net}_j} = \sum_{k \in \text{Downstream}(j)} \frac{\partial E_d}{\partial \text{net}_k} \cdot \frac{\partial \text{net}_k}{\partial \text{net}_j}$$



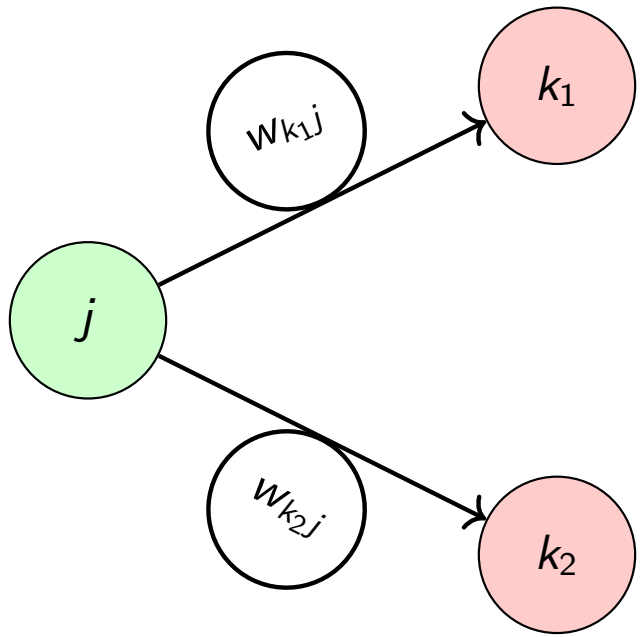

The gradient of the error with respect to $\text{net}_j$ depends on:

- Errors propagated from downstream units ($\frac{\partial E_d}{\partial \text{net}_k}$).
- Weighted connections ($w_{kj}$).

# Components of the Gradient

**Key Component:**

$$\frac{\partial \text{net}_k}{\partial \text{net}_j} = w_{kj} \cdot o_j(1 - o_j)$$




Each weight $w_{kj}$ contributes to the propagation of error gradients from downstream units to the hidden unit $j$.

# Training Rule for Hidden Unit Weights: Focus on $\frac{\partial \text{net}_k}{\partial \text{net}_j}$

**Key Component:**

$$\frac{\partial \text{net}_k}{\partial \text{net}_j}$$

**Derivation Steps:**

$$\frac{\partial \text{net}_k}{\partial \text{net}_j} = \frac{\partial \text{net}_k}{\partial o_j} \cdot \frac{\partial o_j}{\partial \text{net}_j}$$

$$\frac{\partial \text{net}_k}{\partial o_j} = w_{kj}$$

$$\frac{\partial o_j}{\partial \text{net}_j} = o_j(1 - o_j)$$

**Final Expression:**

$$\frac{\partial \text{net}_k}{\partial \text{net}_j} = w_{kj} \cdot o_j(1 - o_j)$$

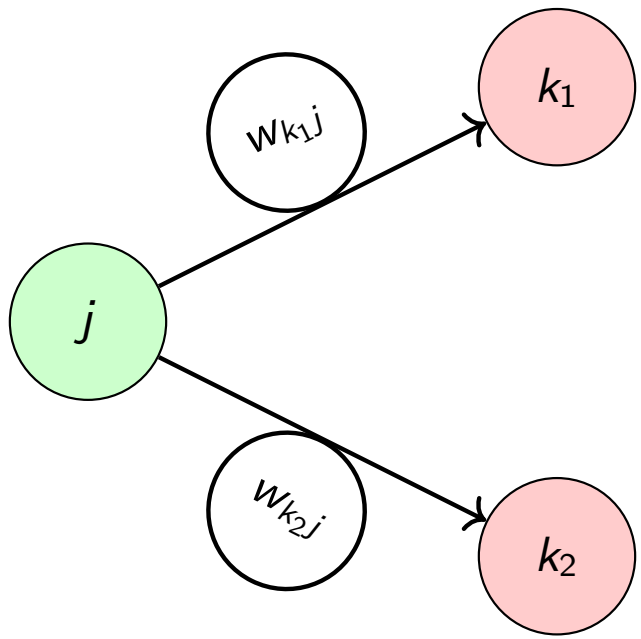
# Gradient Formula for Hidden Units

**Gradient:**

$$\frac{\partial E_d}{\partial \text{net}_j} = \sum_{k \in \text{Downstream}(j)} \delta_k w_{kj} o_j (1 - o_j)$$

where:

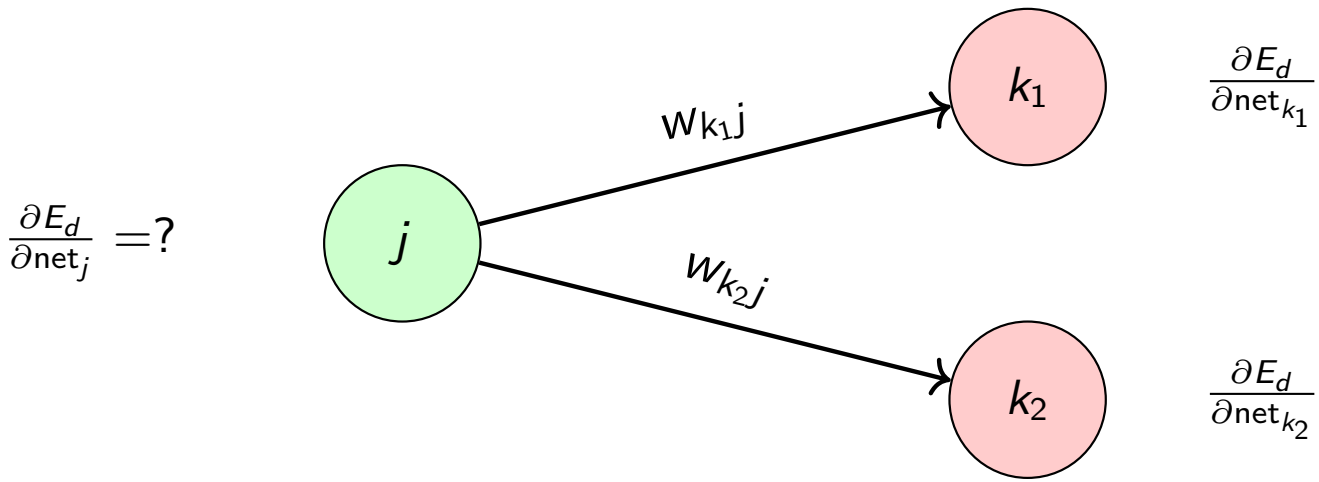
$$\delta_k = \frac{\partial E_d}{\partial \text{net}_k}$$




The gradient at the hidden unit $j$ is a combination of:

- Weighted contributions from downstream units ($w_{kj}$).
- Errors from downstream units ($\delta_k$).

# Visualization of Backpropagation




**Key Point:** The error gradient at $j$ is the sum of its contributions to all downstream neurons $k_1, k_2, \ldots$.

# Training Rule for Hidden Unit Weights (Case 2)

**Key Formula:**

$$\frac{\partial E_d}{\partial \text{net}_j} = \sum_{k \in \text{Downstream}(j)} \frac{\partial E_d}{\partial \text{net}_k} \cdot \frac{\partial \text{net}_k}{\partial \text{net}_j}$$

**Step-by-Step Derivation:**

$$\frac{\partial E_d}{\partial \text{net}_j} = \sum_{k \in \text{Downstream}(j)} -\delta_k \cdot \frac{\partial \text{net}_k}{\partial \text{net}_j}$$

$$= \sum_{k \in \text{Downstream}(j)} -\delta_k \cdot \frac{\partial \text{net}_k}{\partial o_j} \cdot \frac{\partial o_j}{\partial \text{net}_j}$$

$$= \sum_{k \in \text{Downstream}(j)} -\delta_k \cdot w_{kj} \cdot \frac{\partial o_j}{\partial \text{net}_j}$$

$$= \sum_{k \in \text{Downstream}(j)} -\delta_k \cdot w_{kj} \cdot o_j(1 - o_j)$$

**Final Form:**

$$\delta_j = o_j(1 - o_j) \sum_{k \in \text{Downstream}(j)} \delta_k \cdot w_{kj}$$

# Case 2: Training Rule for Hidden Units

**Key Idea:** Hidden units influence the network error indirectly via downstream units.

**Chain Rule:**

$$\frac{\partial E_d}{\partial \text{net}_j} = \sum_{k \in \text{Downstream}(j)} \frac{\partial E_d}{\partial \text{net}_k} \cdot \frac{\partial \text{net}_k}{\partial \text{net}_j}$$

**Components:**

$$\frac{\partial \text{net}_k}{\partial \text{net}_j} = w_{kj} \cdot o_j(1 - o_j)$$

**Gradient:**

$$\frac{\partial E_d}{\partial \text{net}_j} = \sum_{k \in \text{Downstream}(j)} \delta_k w_{kj} o_j(1 - o_j)$$

where $\delta_k = \frac{\partial E_d}{\partial \text{net}_k}$.

# Error Term and Weight Update for Hidden Units

**Error Term:**

$$\delta_j = o_j(1 - o_j) \sum_{k \in \text{Downstream}(j)} \delta_k w_{kj}$$

**Weight Update Rule:**

$$\Delta w_{ji} = \eta \delta_j x_{ji}$$

**Summary:**

- The error term for hidden units is a weighted sum of downstream error terms.
- Weight updates use the same rule as for output units but rely on backpropagated error.

# Summary of Backpropagation

**Key Equations:**

- For output units:

$$\delta_j = (t_j - o_j) o_j (1 - o_j)$$

- For hidden units:

$$\delta_j = o_j(1 - o_j) \sum_{k \in \text{Downstream}(j)} \delta_k w_{kj}$$

**Weight Update Rule:**

$$\Delta w_{ji} = \eta \delta_j x_{ji}$$

**Why the Difference?**

- Output units directly compare their output to the target.
- Hidden units rely on backpropagated error from downstream units.

# Backpropagation Algorithm

**Algorithm:**

- Initialize all weights to small random numbers.
- Until satisfied, Do:
  - For each training example, Do:
    1. Input the training example to the network and compute the network outputs.
    2. For each output unit $k$:

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

    3. For each hidden unit $h$:

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in \text{outputs}} w_{h,k}\delta_k$$

    4. Update each network weight $w_{i,j}$:

$$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$$

where

$$\Delta w_{i,j} = \eta \delta_j x_{i,j}$$