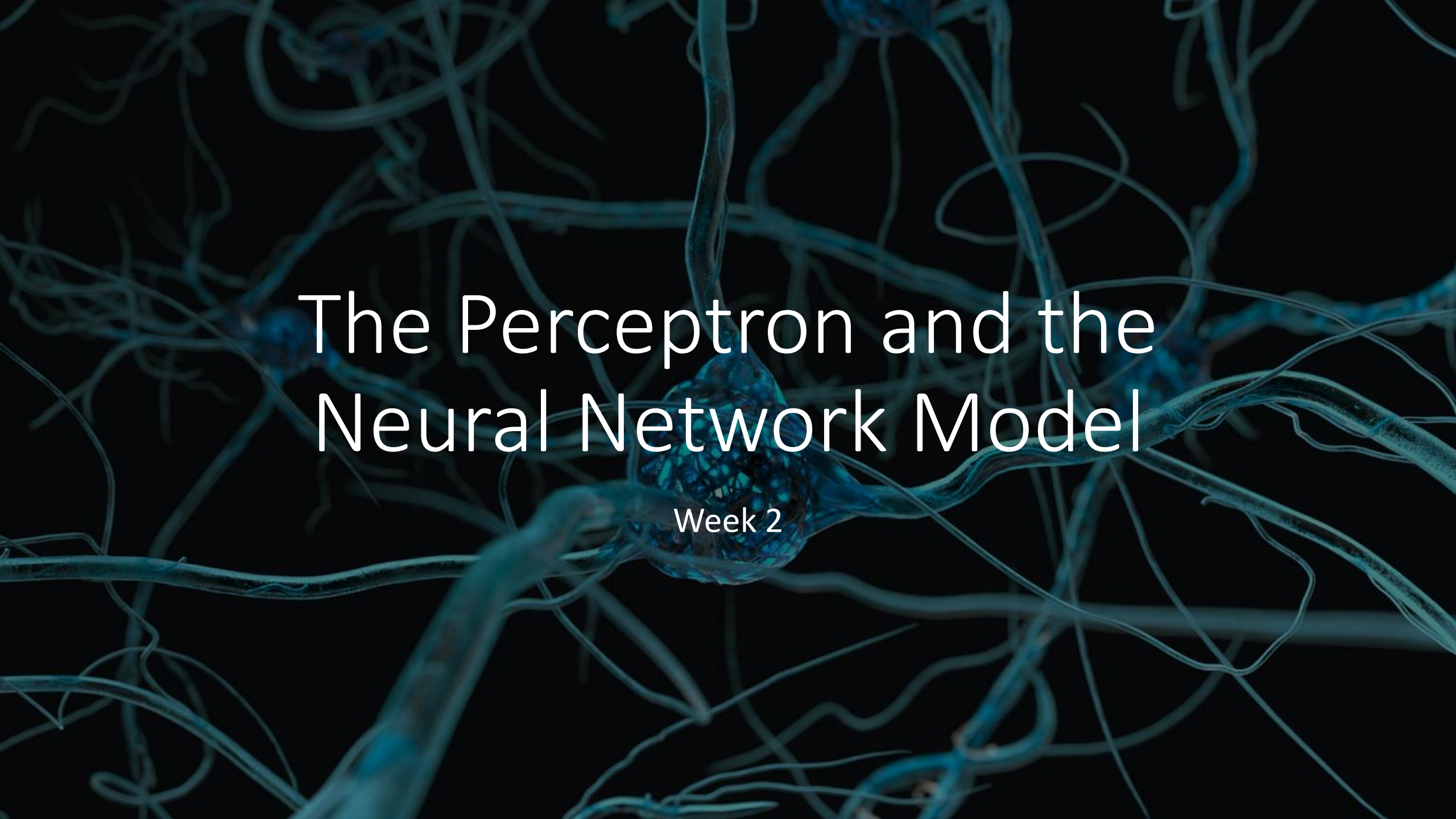


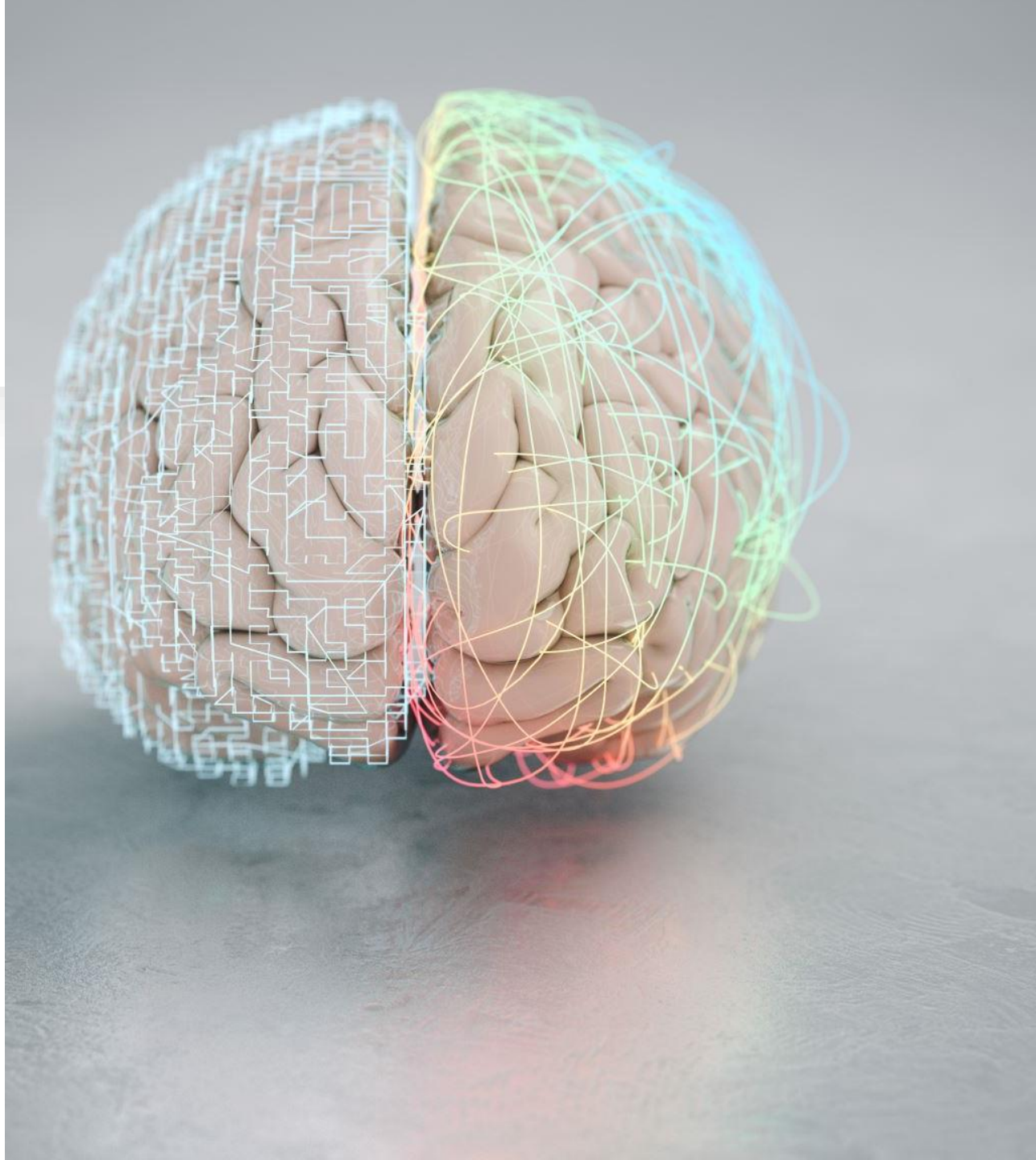
The Perceptron and the Neural Network Model

Week 2

Understanding the Brain

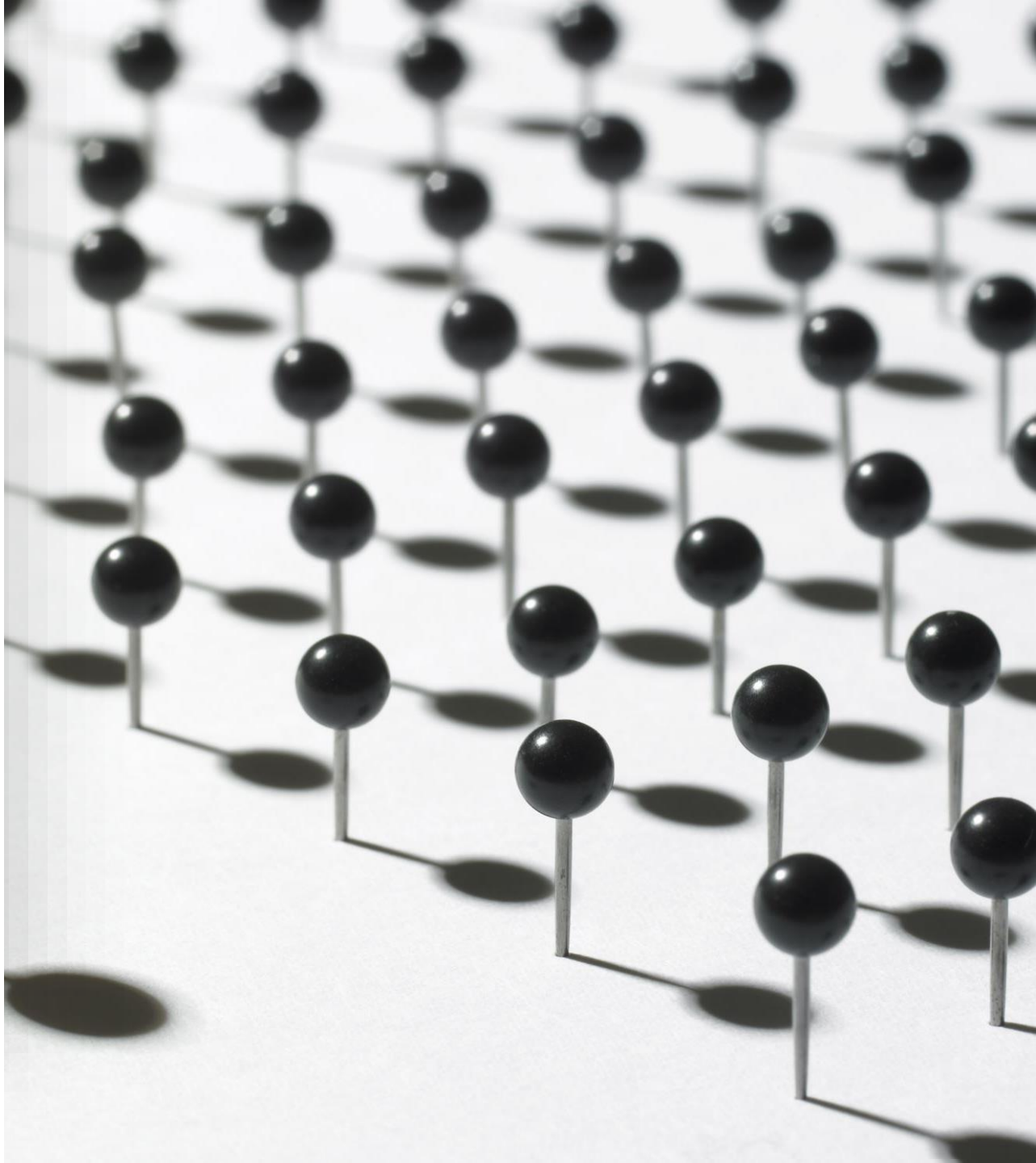
- In 1982, David Marr proposed three levels of analysis for understanding information processing systems, similar to the human brain.

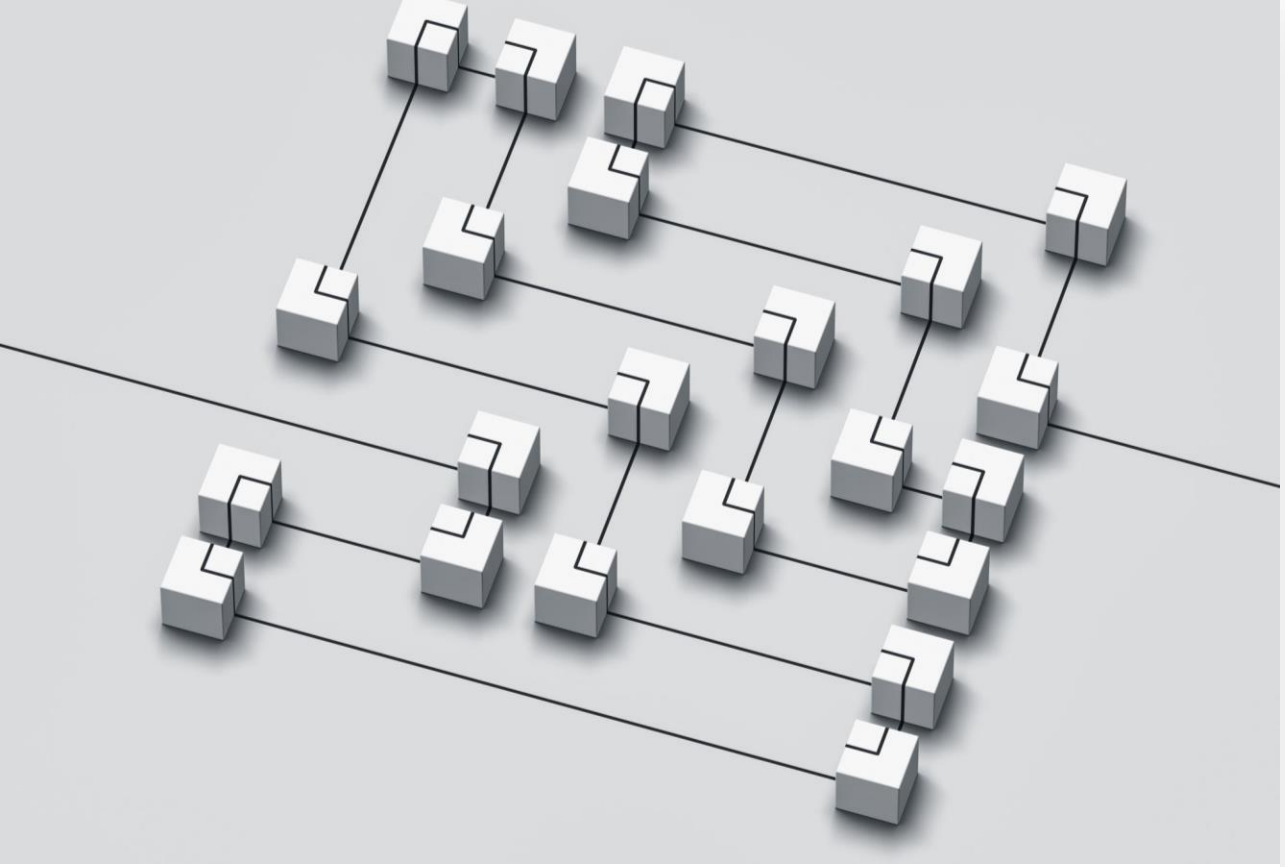
- Computation Theory
- Representation Algorithm
- Hardware Implementation



Computation Theory

- Concerned with the goal of computation and an abstract definition of the task.
- This could be the task T
- Example: For sorting, the computational theory is to order a given set of elements.



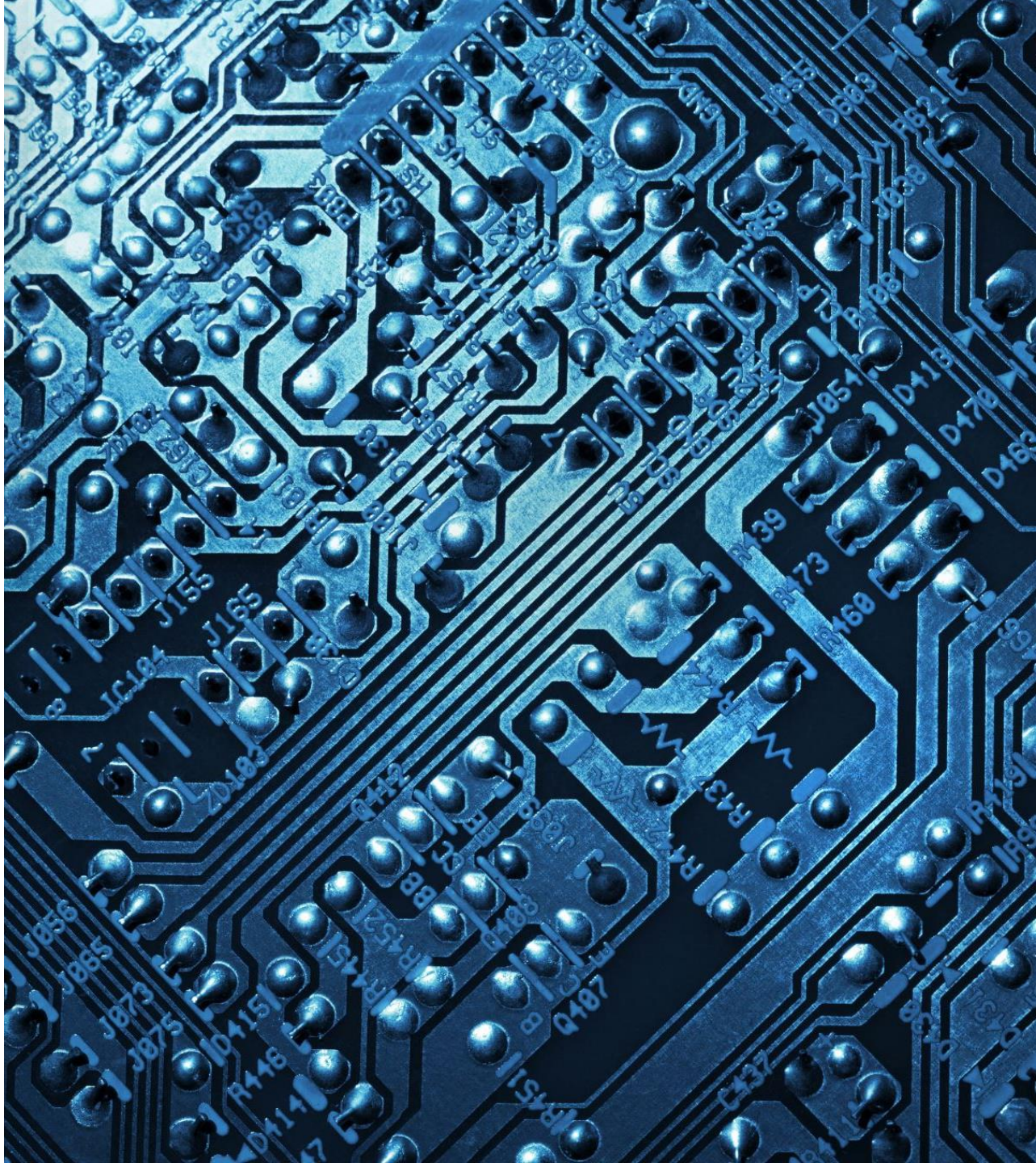


Representation Algorithm

- Involves how input and output are represented, and the specification of the algorithm for transforming input to output.
- This could be the information being fed or produced or representation or E experience.
- Example: In sorting, the representation may use integers, and the algorithm may be Quicksort.

Hardware Implementation

- Refers to the actual physical realization of the system.
- This could be the GPUs, CPUs, memory, and other hardware involved.
- Example: After compilation, the executable code for sorting integers on a particular processor represented in binary is one hardware implementation.



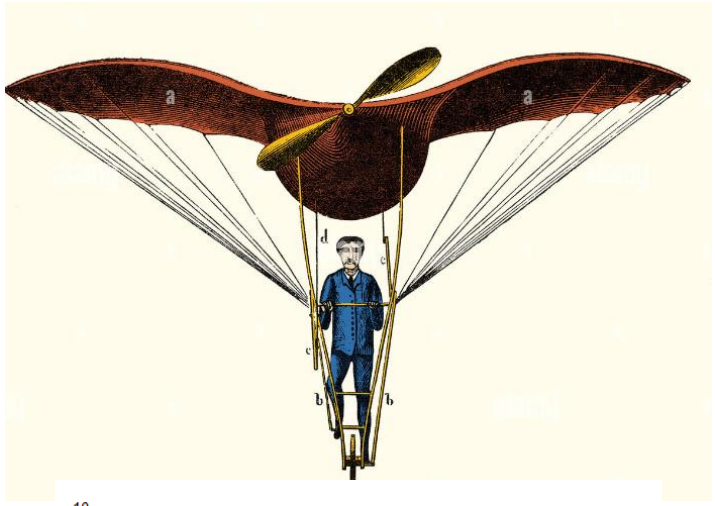
What does it tell us?

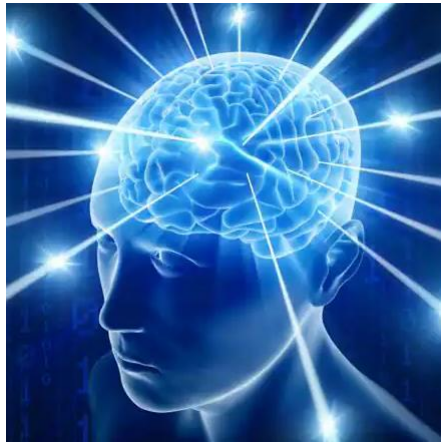
- The same computational theory can have multiple representations and algorithms, allowing for adaptability and varied approaches to problem-solving.

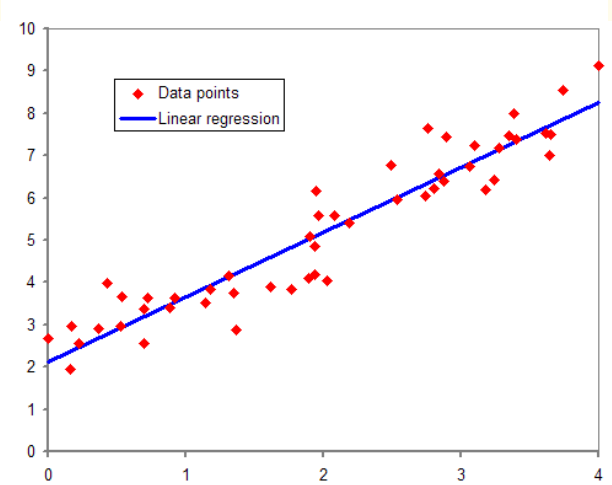
- Different hardware implementations can result from the same representation and algorithm, highlighting the compatibility with various technological constraints.

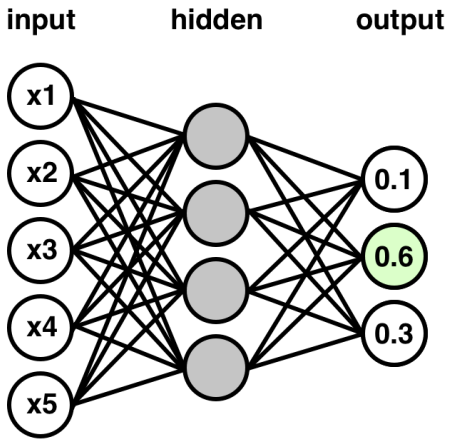
- Drawing parallels between natural and artificial flying machines to emphasize that different hardware implementations can achieve the same computational theory, akin to how a sparrow and an airplane both implement aerodynamics.

Reverse Engineering the Brain





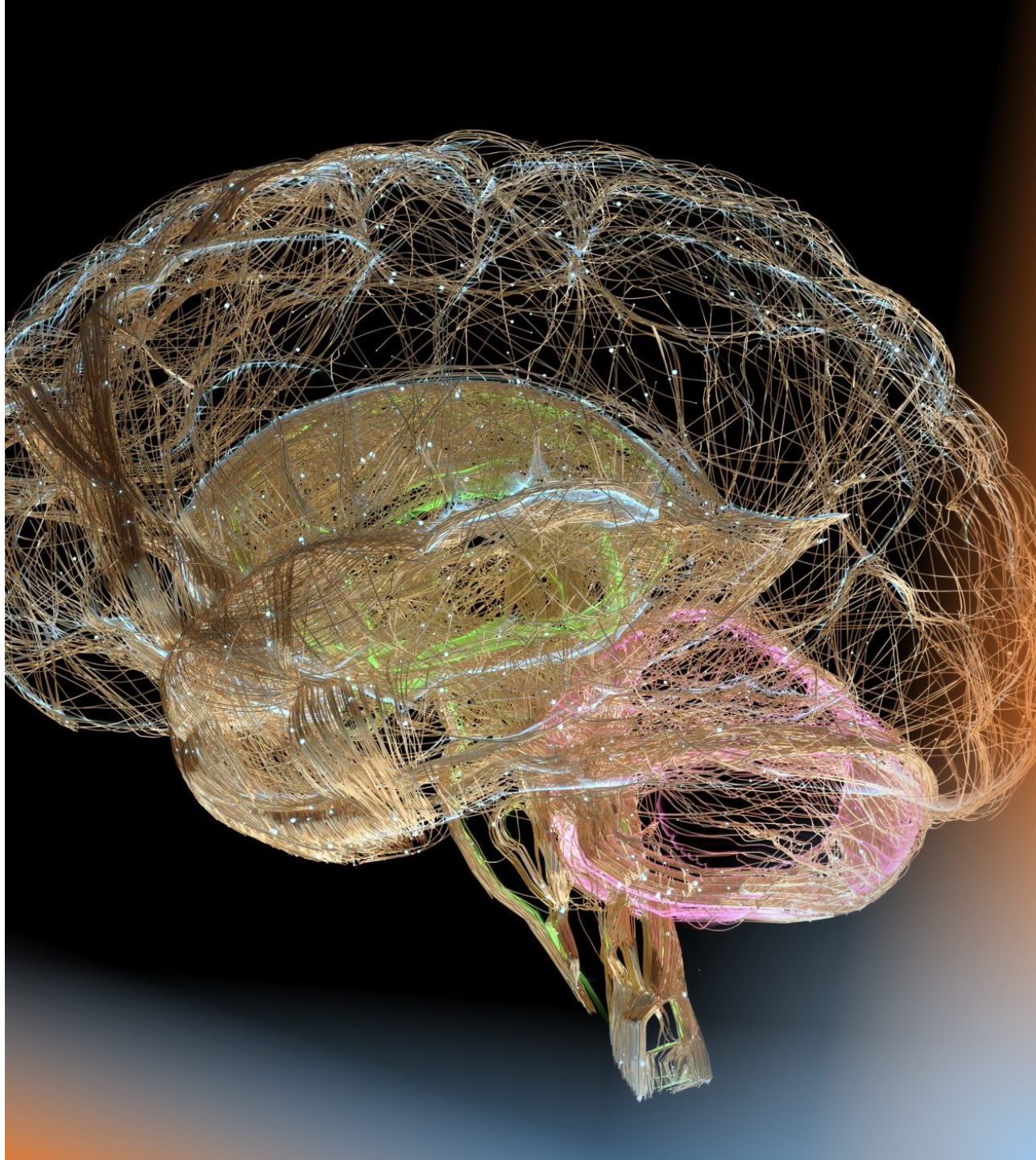




The brain as a hardware for implementation

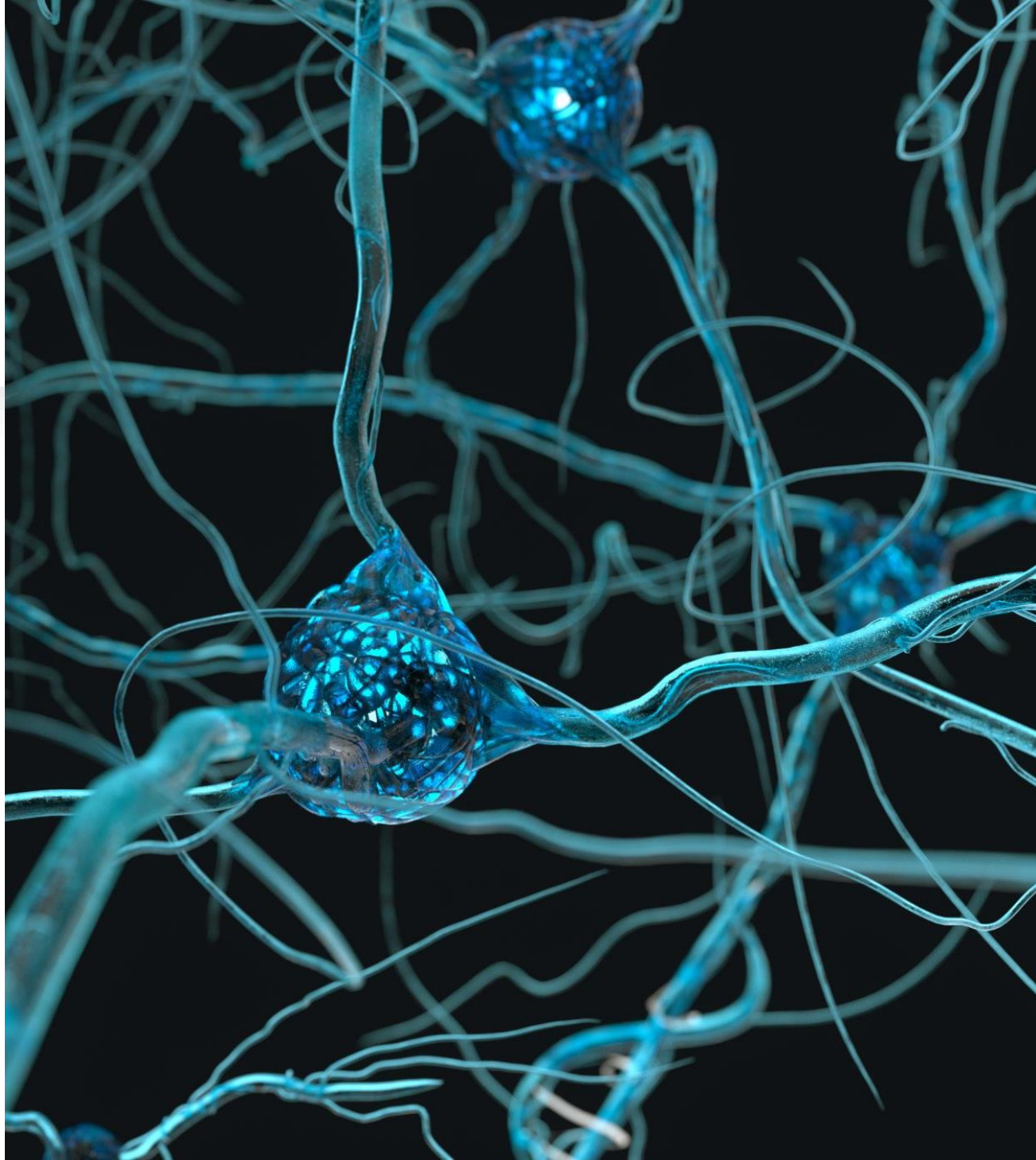
- Viewing the brain as a hardware implementation for learning or pattern recognition, with the possibility of reverse engineering to extract representation and algorithm, ultimately leading to a more adaptable hardware implementation.

- The analogy suggests that initial attempts to replicate brain-like structures may resemble the brain until the computational theory of intelligence is fully understood.

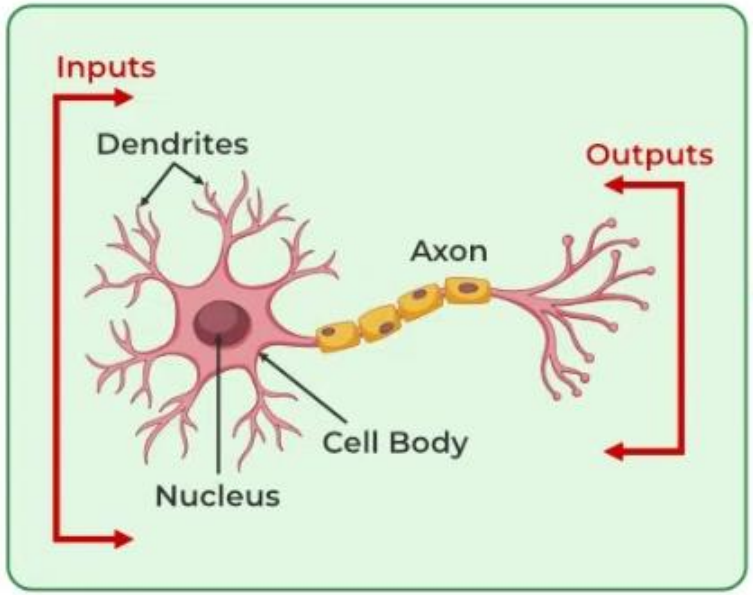


Neural Networks as a Paradigm for Parallel Processing

- “Artificial” neural networks are inspired by the organic brain, translated to the computer.
- It’s not a perfect comparison, but there are neurons, activations, and lots of interconnectivity, even if the underlying processes are quite different.



Biological Neuron



A human brain has billions of neurons. **Neurons** are interconnected nerve cells in the human brain that are involved in processing and transmitting chemical and electrical signals.

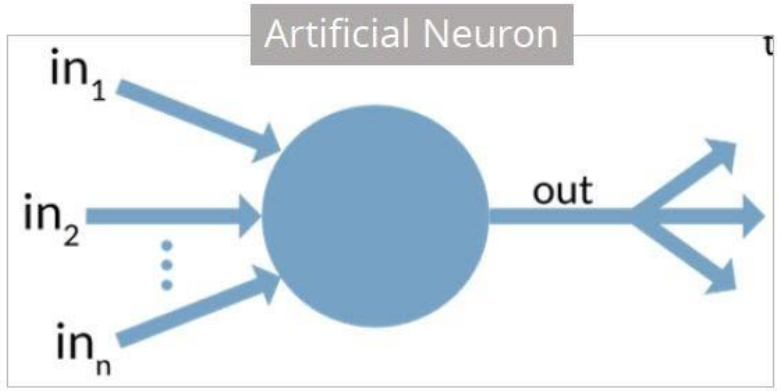
Dendrites are branches that receive information from other neurons.

Cell **nucleus or Soma** processes the information received from dendrites.

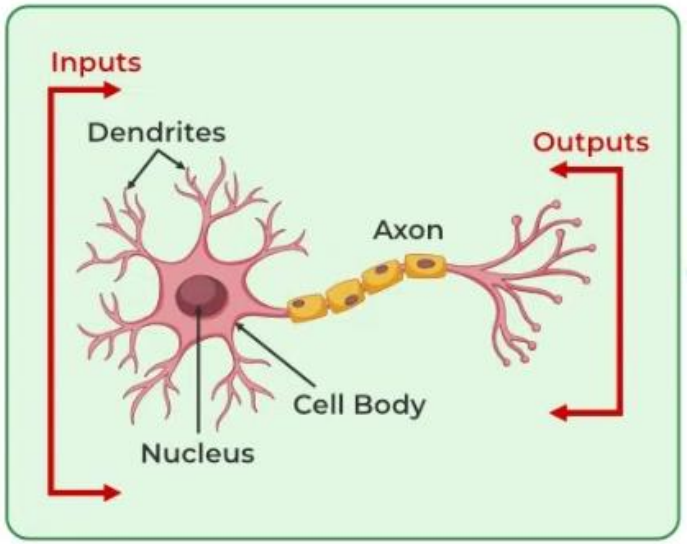
Axon is a cable that is used by neurons to send information. Synapse is the connection between an axon and other neuron dendrites.

Artificial Neurons

- Researchers **Warren McCulloch and Walter Pitts** published their first concept of simplified brain cell in **1943**. This was called **McCulloch-Pitts (MCP) neuron**.
- They described such a nerve cell as a simple logic gate with binary outputs.
- Multiple signals arrive at the dendrites and are then integrated into the cell body, and, if the accumulated signal exceeds a certain threshold, an output signal is generated that will be passed on by the axon. In the next section, let us talk about the artificial neuron.

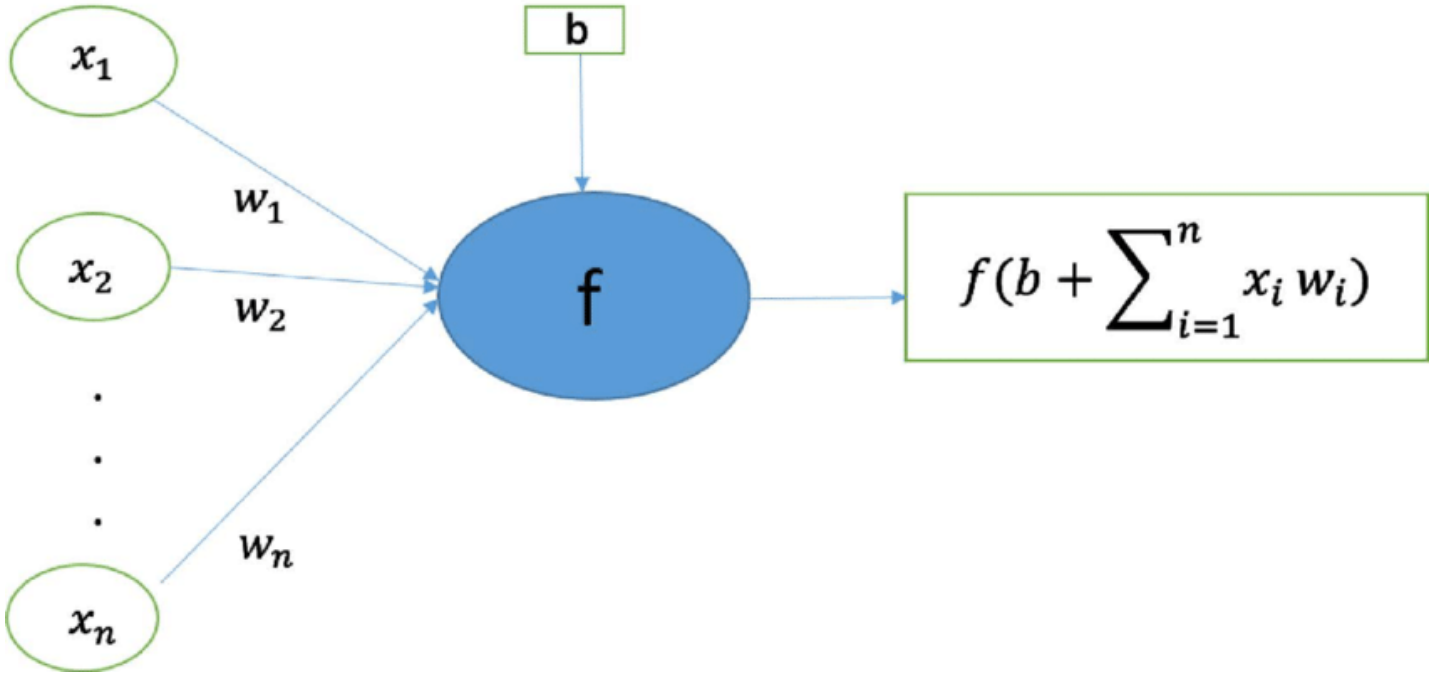


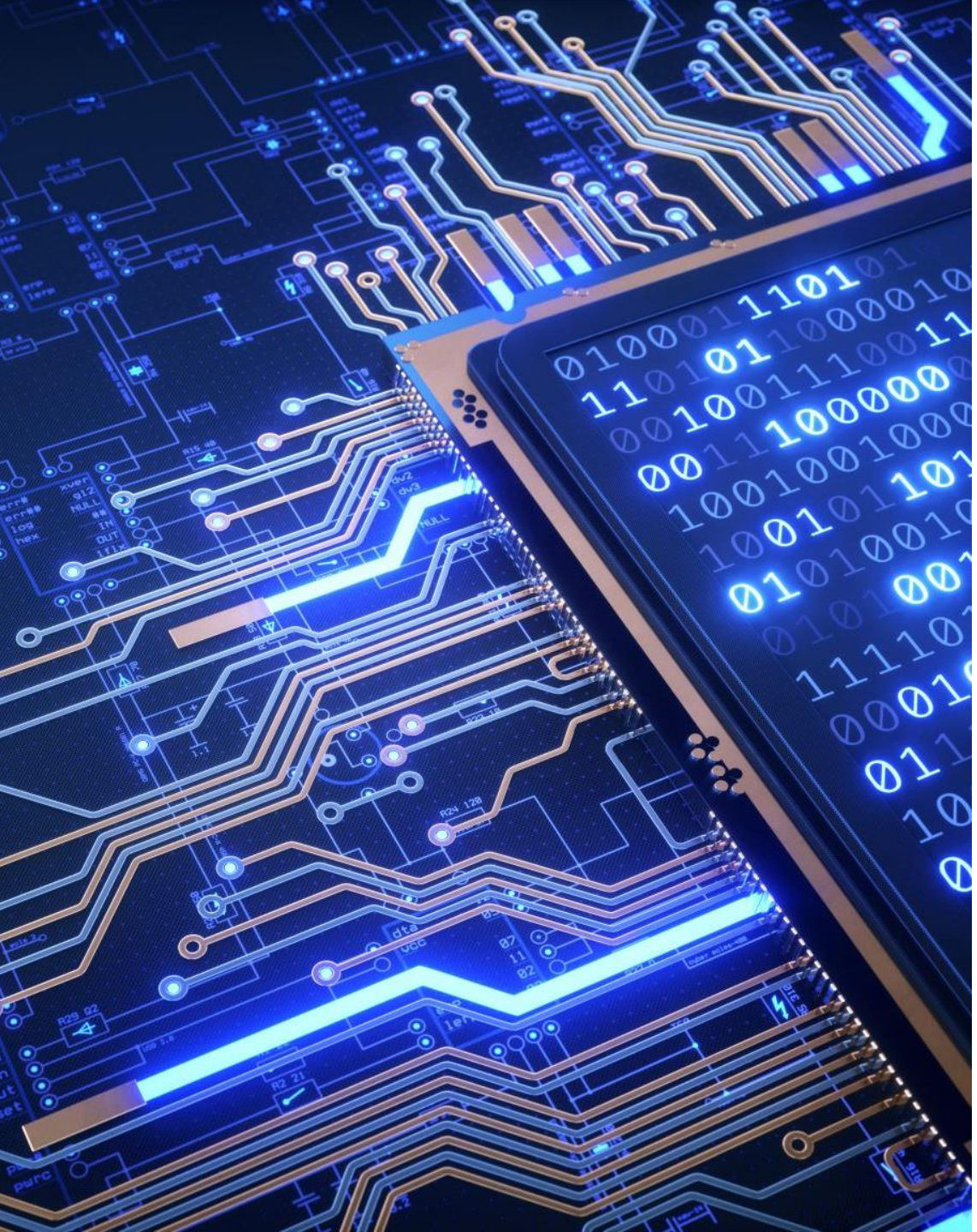
Biological and Artificial Neuron



Biological Neuron	Artificial Neuron
Dendrite	Inputs
Cell nucleus or Soma	Nodes
Synapses	Weights
Axon	Output

Structure of an Artificial Neuron





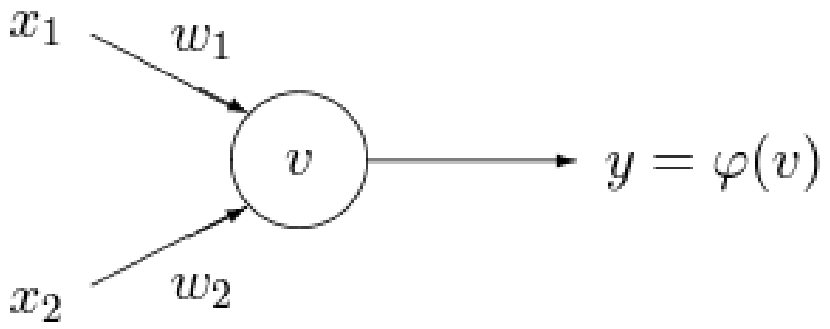
Boolean Functions

- All types of functional dependence including Boolean functions can be represented using neural networks.
- Logical operators (i.e. NOT, AND, OR, XOR, etc) are the building blocks of any computational device.

Boolean Functions

For example, operator AND returns true only whenall its arguments are true, otherwise(if any of the arguments is false) it returns false.

If we denote truth by 1 and false by 0, then logical function AND can be represented by the given table.



x_1 :	0	1	0	1
x_2 :	0	0	1	1
x_1 AND x_2 :	0	0	0	1

if the weights are $w_1 = 1$ and $w_2 = 1$ and the activation function is:

$$\varphi(v) = \begin{cases} 1 & \text{if } v \geq 2 \\ 0 & \text{otherwise} \end{cases}$$

Note that the threshold level is 2 ($v \geq 2$).

AND Truth Table

Inputs		Output
A	B	Y = A.B
0	0	0
0	1	0
1	0	0
1	1	1

OR Truth Table

Inputs		Output
A	B	Y = A+B
0	0	0
0	1	1
1	0	1
1	1	1

XOR Truth Table

Inputs		Output
A	B	Y = $A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

NAND Truth Table

Inputs		Output
A	B	Y = $\overline{A.B}$
0	0	1
0	1	1
1	0	1
1	1	0

Preliminaries

Let $\{x_1, x_2, x_3, \ldots, x_n\}$, be Boolean variables, $x_i \in \{0,1\} = B$.

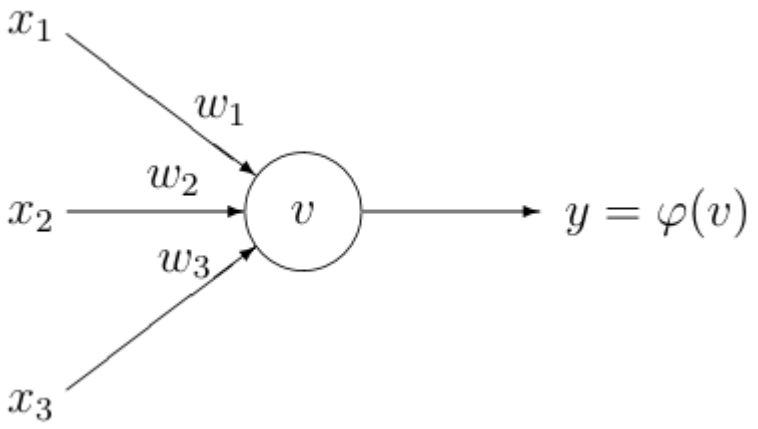
A Boolean function f is defined B= {(0 ... 00), (0 ... 01), (0 ... 10),..., (1 ... 11)}. The number of vectors in the Boolean space B_n is $N = 2^n$.

The simplest representation of a Boolean function is a table that defines the function value for each of the 2^n input vectors

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Example

- The node has three inputs $x = (x_1, x_2, x_3)$that receive only binary signals (either 0 or 1).
- **How many different input patterns this node can receive?**
- What if the node had four inputs? Five? Can you give a formula that computes the number of binary input patterns for a given number of inputs?



Example

- You may check that for five inputs the number of combinations will be 32. Note that 8 = 23, 16 = 24 and 32 = 25 (for three, four and five inputs). Thus, the formula for the number of binary input patterns is 2^n, where n in the number of inputs.

For three inputs the number of combinations of 0 and 1 is 8:

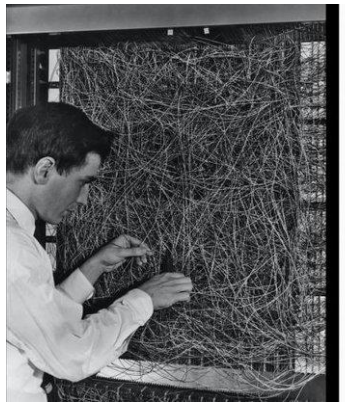
x_1	0	1	0	1	0	1	0	1
x_2	0	0	1	1	0	0	1	1
x_3	0	0	0	0	1	1	1	1

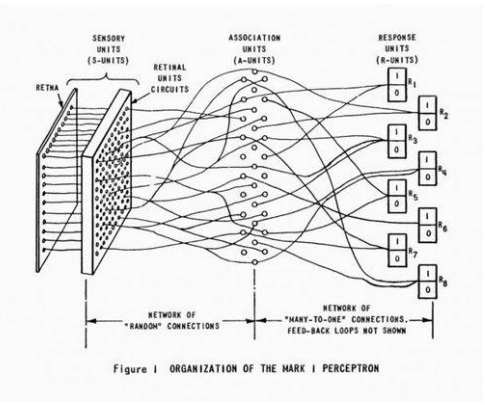
and for four inputs the number of combinations is 16:

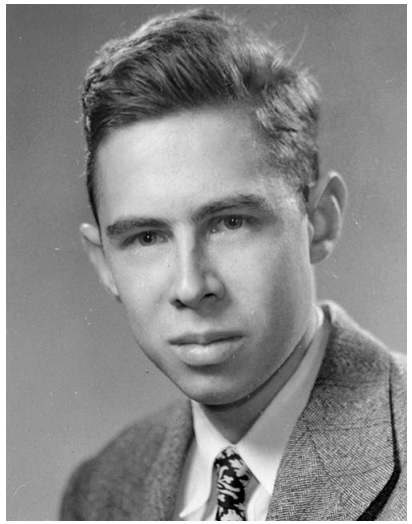
x_1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
x_2	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
x_3	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
x_4	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

The Mark I Perceptron

- Physically made up of an array of 400 photocells connected to perceptron whose weights were recorded in potentiometers, as adjusted by electric motors.
- In 1957, this machine was one of the first artificial neural networks ever created.
- Based on the MP perceptron of 1943.
- Trained and improves by feeding from data.
- Made at the Cornell Aeronautical Laboratory.







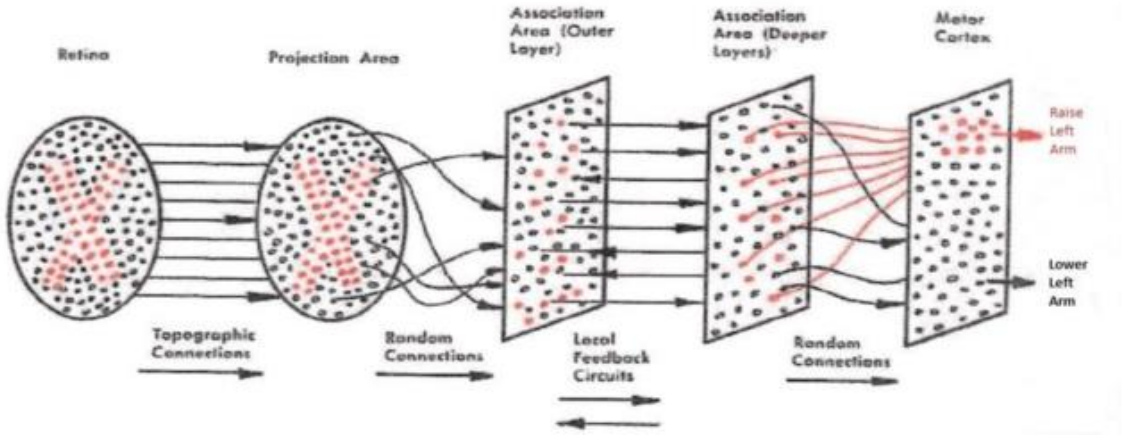


FIG. 1 — Organization of a biological brain. (Red areas indicate active cells, responding to the letter X.)

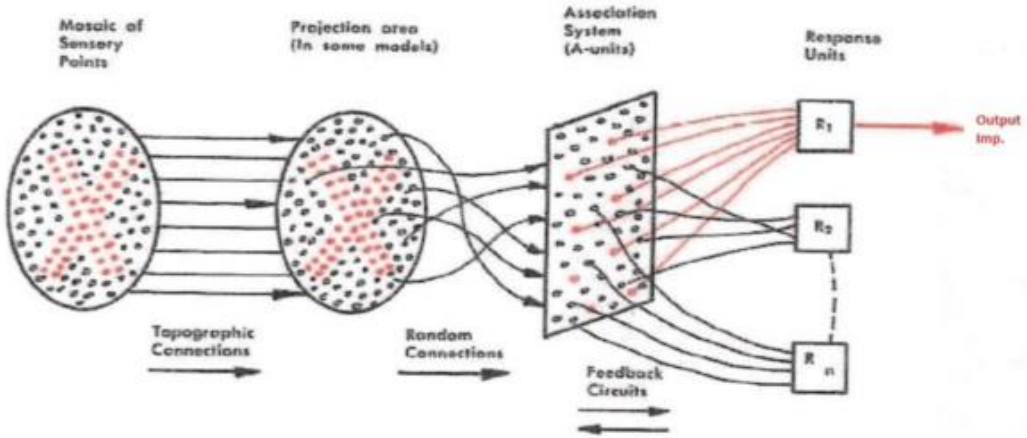


FIG. 2 — Organization of a perceptron.

SAR, A three layered perceptron

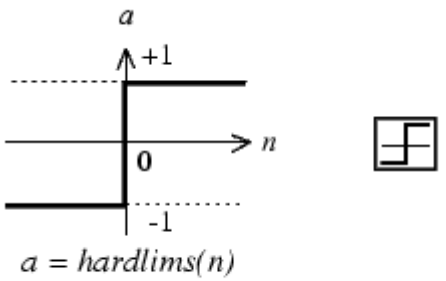
- Sensory (S) Units: 20 x 20 grid that connects to 40 A-Units
- Association (A) Units: A 512-unit hidden layer that eliminates intentional bias
- Response (R) Units: 8 perceptrons
- Weights are fixed and not learned.
- Rosenblatt was adamant about setting randomness.
- A-Units are connected to R-Units with adjustable weights
- Changing of weights were performed by electric motors.

Why did Mark I fail?

- Mark 1 Perceptron failed to address non-linearly separable problems and was limited in its ability to handle more complex tasks.
- It’s a pure binary classifier that relies on linearity.
- It had a limited number of layers, as it utilizes large physical components to add more.
- However, its development marked an important step in the history of artificial neural networks, paving the way for future advancements in the field.

Limitations of the perceptron

- The output of a perceptron can only be a binary number (0 or 1) due to the hard limit transfer function.
- Perceptron can only be used to classify the linearly separable sets of input vectors.
- If input vectors are non-linear, it is not easy to classify them properly.

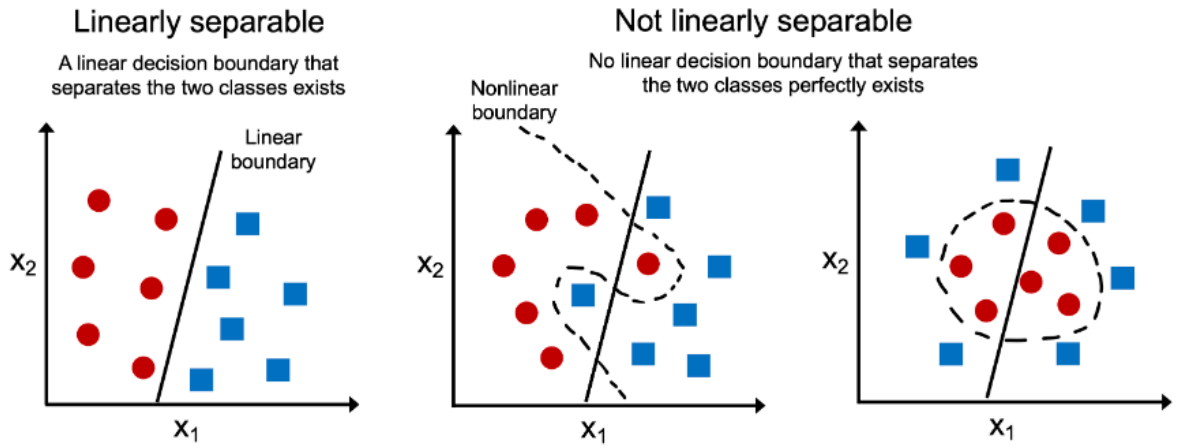


Symmetric Hard-Limit Trans. Funct.

$$hardlim(n) = 1, if\ \geq 0; -1\ otherwise = \begin{cases} 1, & if\ n \geq 0 \\ -1, & if\ n < 0 \end{cases}$$

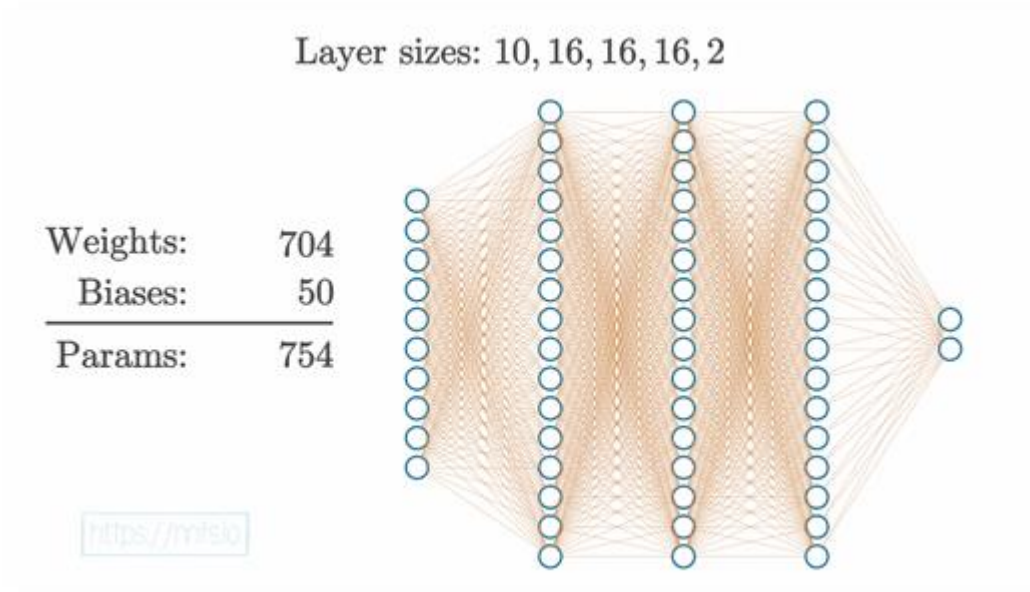
Linear and Non-Linear

- In linear data, the relationship between the input features and the output can be represented by a straight line (or hyperplane in higher dimensions).
- Image classification tasks often involve non-linear problems. The relationships between pixel values in an image and the corresponding class labels are typically complex and non-linear.



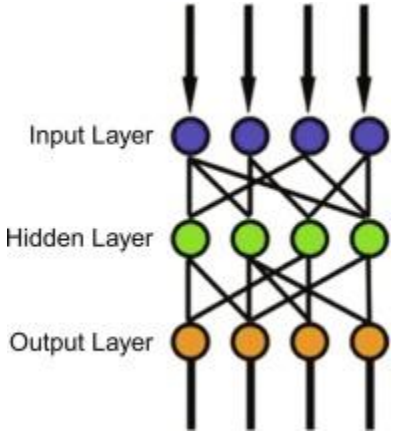
Multiple Neurons are needed

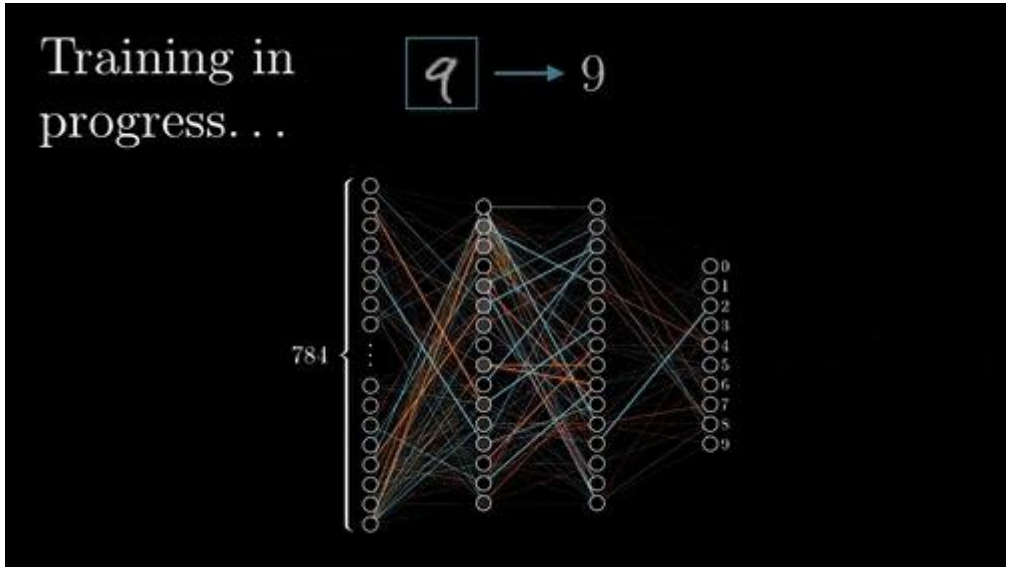
- A single neuron by itself is relatively useless, but, when combined with hundreds or thousands (or many more) of other neurons, the interconnectivity produces relationships and results that frequently outperform any other machine learning methods.



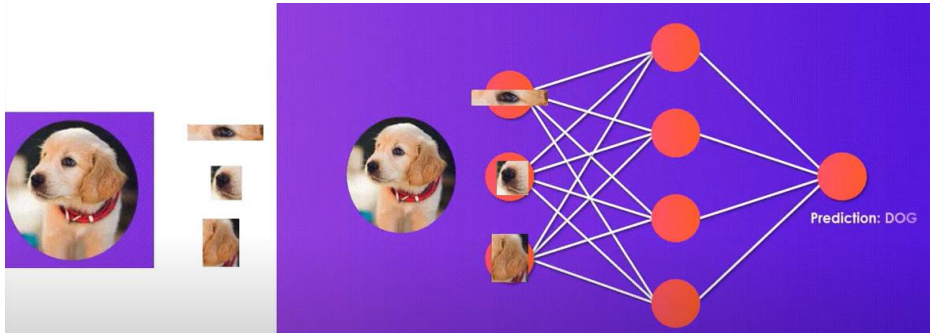
Multilayer Perceptron

- Reflects the organization of the human brain.
- MLP is also equal to the feed-forward ANN
- The learning phase is continuously repeated until the value of the error will be less than the threshold level

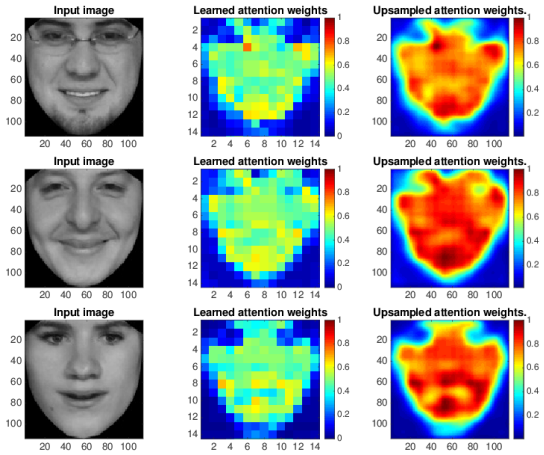


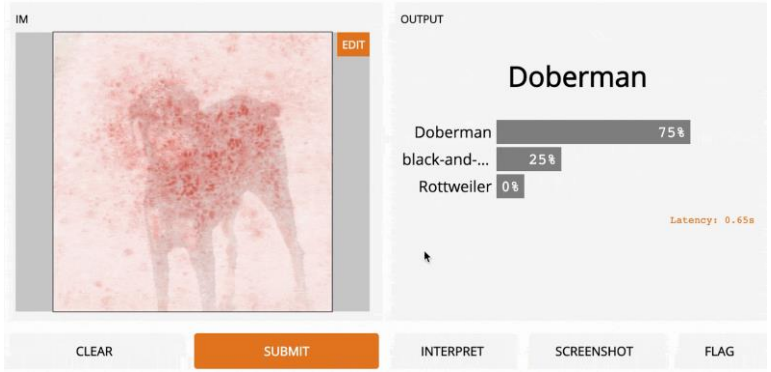


Weights



- Control the strength of the connection between two neurons.
- Inputs are typically multiplied by weights, defining how much influence the input will have on the output.
- Their values are first randomly initialized and then learned, updated, and optimized by the network during the training process.
- We can think of them as coefficients that shifts the impact of incoming data.
- NOT ALL WEIGHTS ARE EQUAL.





Biases

- Bias terms are additional constants attached to neurons and added to the weighted input before the activation function is applied.
- Bias terms help models represent patterns that do not necessarily pass through the origin.
- They add flexibility and adaptability to the neural network to provide better prediction from unseen samples.
- They are not connected to specific inputs, but added to the neuron’s output.
- They are learnable or updates with the weights.
- They are offsets or threshold, allowing neurons to activate when the weighted sum of their inputs is not sufficient on its own.

Why Biases are flexible?

- In the real-world, data coming to our brain does not always align in specific thresholds. Biases imitates this approach.
- Without bias, a neuron might only activate when it detects the exact pixel brightness.
- With bias, a neuron can react better even when the brightness of that pixel is not perfect or aligned with the given threshold.

Weights and Biases

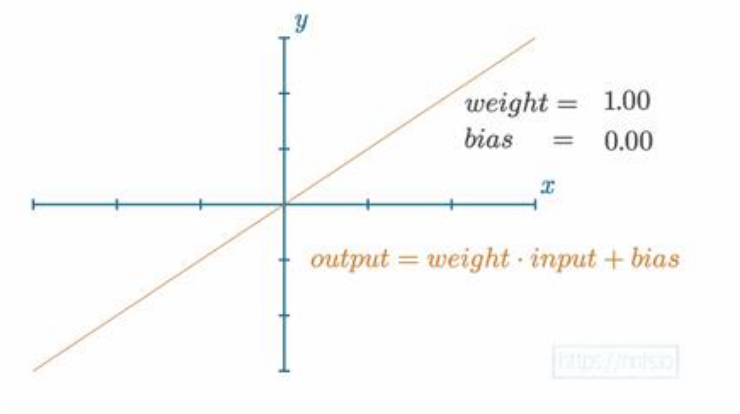
Weights and biases can be thought of as "knobs" that we can tune to fit our model to data.

Neural networks, often have thousands or even millions of these parameters tuned by the optimizer during training. Some may ask, "why not just have biases or just weights?"

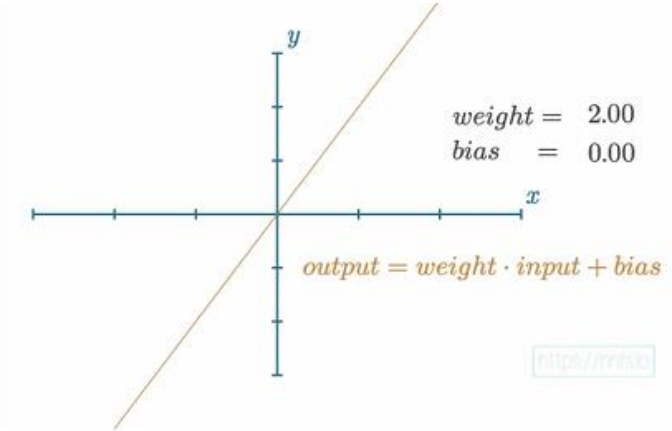
Biases and weights are both tunable parameters, and both will impact the neurons' outputs, but they do so in different ways. Since weights are multiplied, they will only change the magnitude or even completely flip the sign from positive to negative, or vice versa.

$$output = (weight * input) + bias$$

is not unlike the equation for a line y = mx+b. We can visualize this with:



Graph of a single-input neuron's output with a weight of 1, bias of 0 and input x



Adjusting the weight will impact the slope of the function.

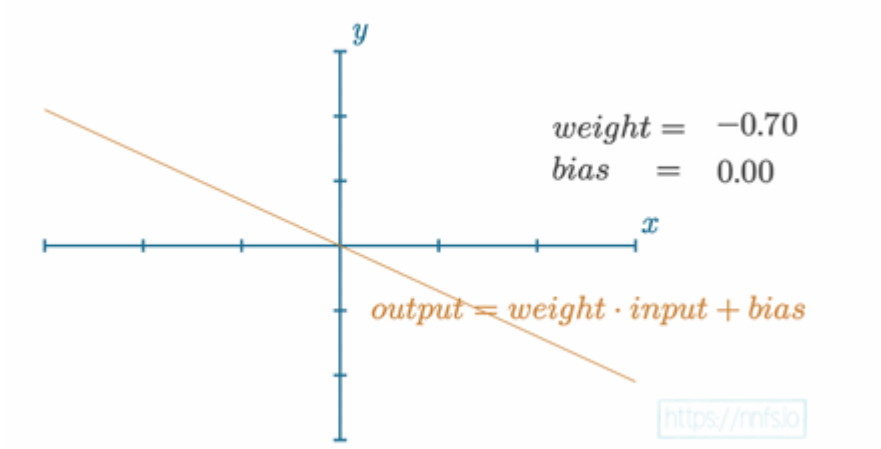
Weighted Input

- A neuron’s input equals the sum of weighted outputs from all neurons in the previous layer.
- Each input is multiplied by the weight associated with the synapse connecting the input to the current neuron.
- Post that, an activation function is applied on the above result.

$$Y = \text{Activation}(\Sigma(weight * input) + bias)$$

Adjusting the weight

- As we increase the value of the weight, the slope will get steeper. If we decrease the weight, the slope will decrease.
- If we negate the weight, the slope turns to a negative:

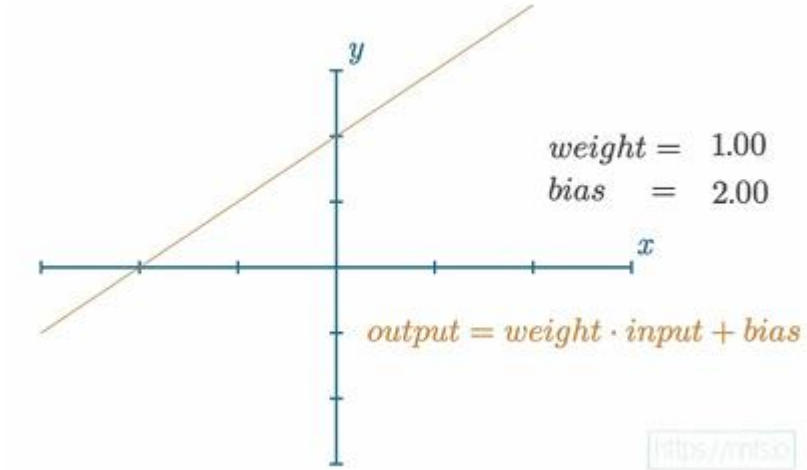


Weights adjust during training in a process called gradient descent.

Graph of a single-input neuron's output with a weight of -0.70, bias of 0 and input x.

Adjusting the Bias

- The bias offsets the overall function. For example, with a weight of 1.0 and a bias of 2.0:

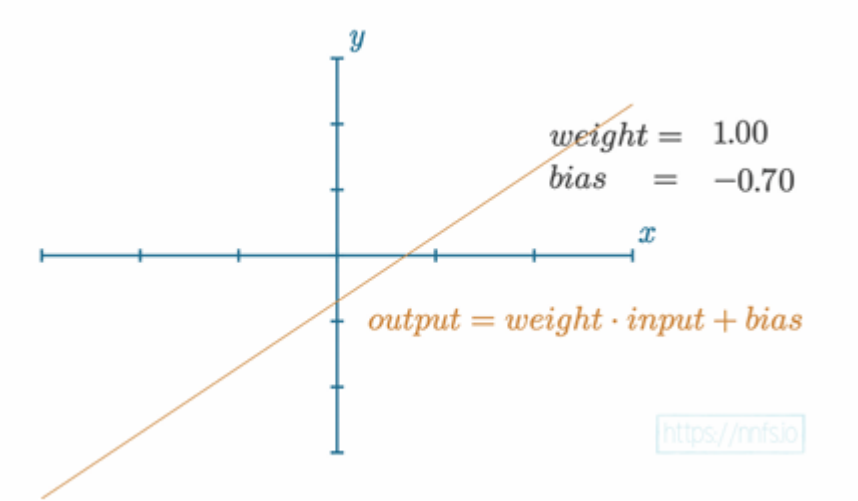


a single-input neuron’s output with a weight of 1, bias of 2 and input x.

Unlike weights, bias can be defined as a constant

Adjusting the Bias

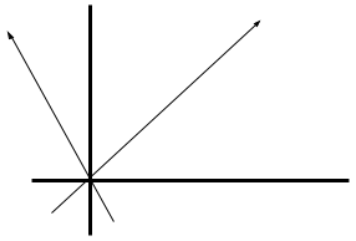
- As we increase the bias, the function output overall shifts upward. If we decrease the bias, then the overall function output will move downward. For example, with a negative bias:



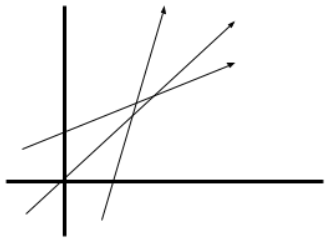
A single-input neuron's output with a weight of 1.0, bias of -0.70 and input x.

What if there’s no bias?

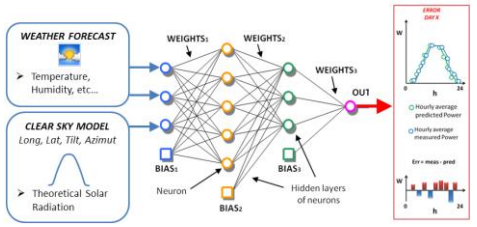
- Due to absence of bias, model will train over point passing through origin only, which is not in accordance with real-world scenario.
- Also, with the introduction of bias, the model will become more flexible.
- Bias helps in controlling the value at which activation function will trigger.



Y = mx



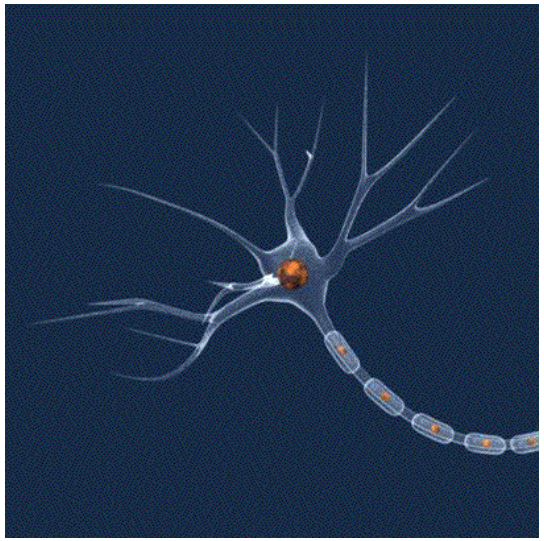
Y = mx + c

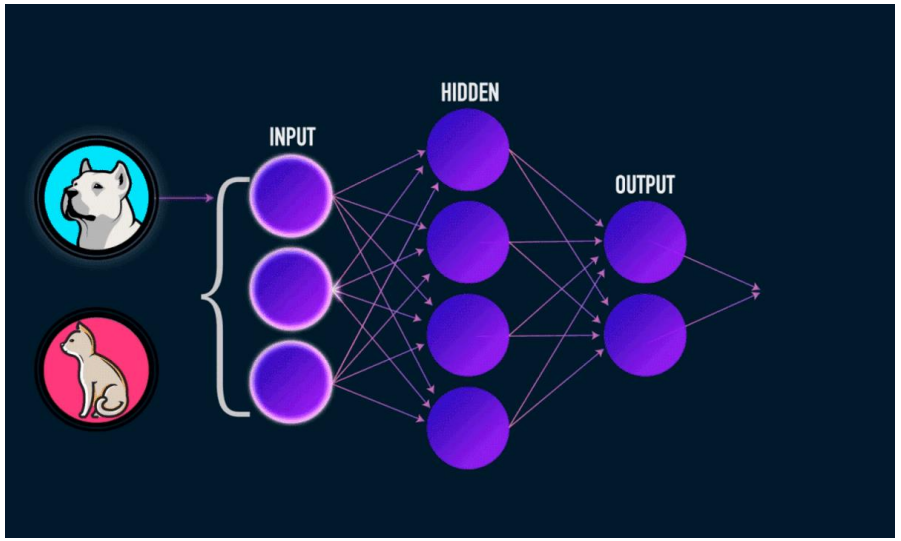


Imagine you have a model that tries to make predictions, like predicting if it's sunny based on temperature and humidity. If there's no bias in your model, it's like saying it will only make predictions if both temperature and humidity are exactly zero. This is not practical in the real world.

Imitating the Brain's firing mechanism

- As a very general overview, the step function meant to mimic a neuron in the brain, either "firing" or not — like an on-off switch. In programming, an on-off switch as a function would be called a step function because it looks like a step if we graph it.





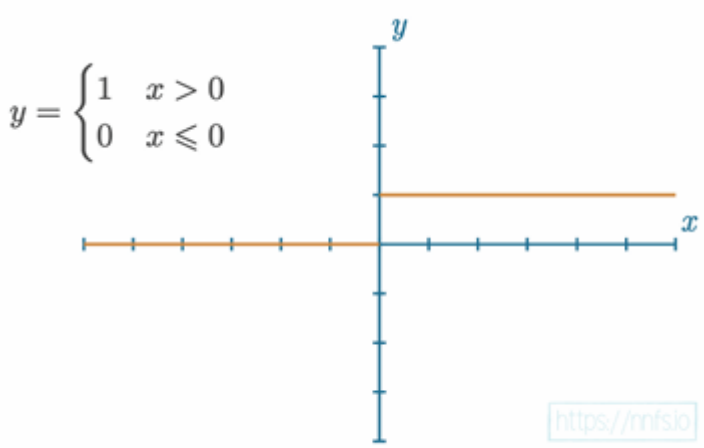
Step Function

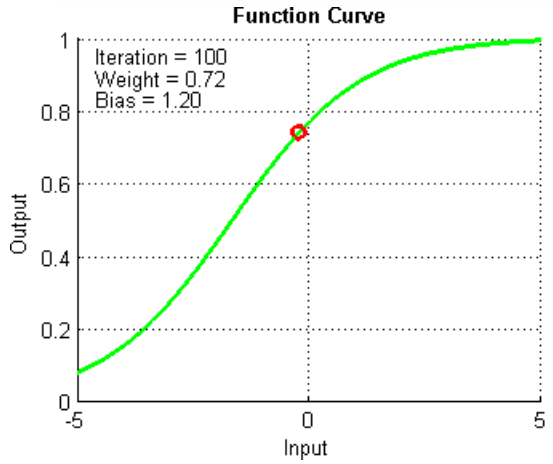
For a step function, if the neuron's output value, which is calculated by sum (inputs · weights) + bias, is greater than 0, the neuron fires (so it would output a 1). Otherwise, it does not fire and would pass along a 0. The formula for a single neuron might look something like:

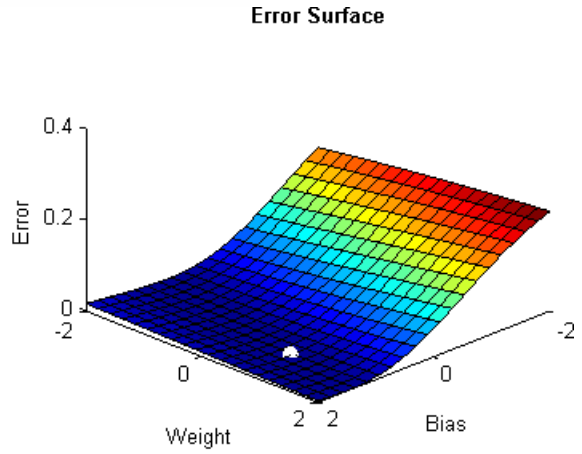
output = sum(inputs * weights) + bias

We then usually apply an activation function to this output, noted by activation():

output = activation(output)

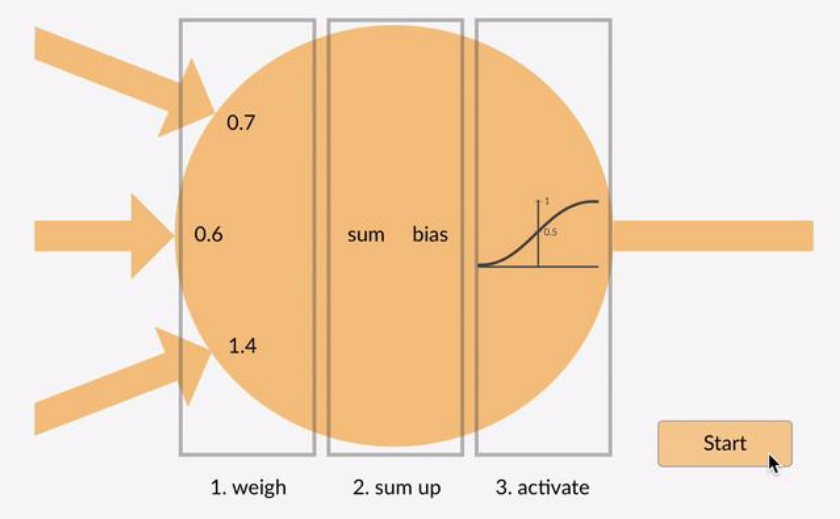






Activation Functions

- While you can use a step function for your activation function, we tend to use something slightly more advanced.
- Neural networks of today tend to use more informative activation functions
- (rather than a step function), such as the Rectified Linear
- (ReLU) activation function



Simulation Example

- Input values of 1 and 2 are given.
- Each input will have random weights -0.55 for input 1 and 0.1 for input 2.
- Using the equation weight * input we get our output.
- Using a steep function σ, it will be determining whether the neuron should fire or not. For 0, it will not fire, if 1 it will fire.
- Without bias:
- $output = \sigma\big(1(-.055) + 2(0.1)\big)$
- $output = \sigma(-0.35)$ or 0
- With bias:
- $output = \sigma\big(1(-.055) + 2(0.1)\big) + b$
- Where b can be 1.
- $output = \sigma(0.65)$ or 1
- Being 1, the steep function will fire the neuron.