Welcome to the *dark side*

Roadmap

- Putting things in perspective.
- On how a single neuron works.
- Creating your first artificial neuron, from scratch.
- Creating your first *Artificial Neural Network*, step by step.
- Helping machines to make a prediction.
- Helping machines to learn from *experience*.
- Helping machines to get smarter.

What is machine learning?

It is a subfield of artificial intelligence.
It gives computers the ability to learn without being explicitly programmed.

Artificial Intelligence

Agents that perceive the world around them. They plan and make decisions in order to achieve their goals.

Machine Learning

Gives "computers the ability to learn without being explicitly programmed"

Supervised Learning

Learn in a supervised manner. The machine is told what is right or wrong.

Classification, Regression

Unsupervised Learning

Explore the data and find out the emerging patterns.

Clustering, dimensionality reduction

Reinforcement Learning

Rewards-based learning.

Rewards-maximization

Computer Vision

NLP

Planning

...

How ANNs differ from conventional programming?

Algorithmic Programming

Data → Computer

Program → Computer

Computer → Output

Artificial Neural Networks

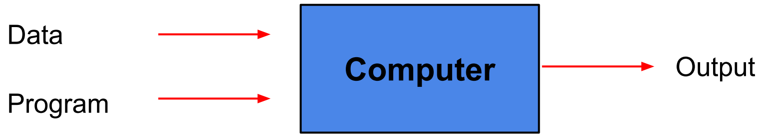
Data → Computer

Output → Computer

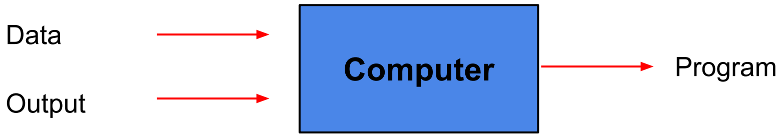
Computer → Program

How ANNs differ from conventional programming?

Algorithmic Programming



Artificial Neural Networks

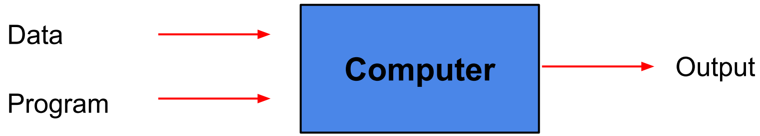


We tell the computer what to do and how to do it.

- It solves a problem by following a set of instructions.

How ANNs differ from conventional programming?

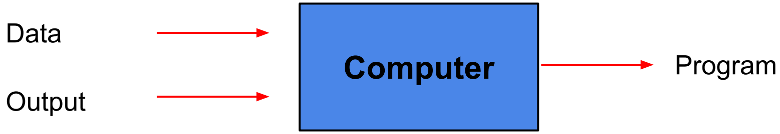
Algorithmic Programming



We tell the computer what to do and how to do it.

- It solves a problem by following a set of instructions.

Artificial Neural Networks

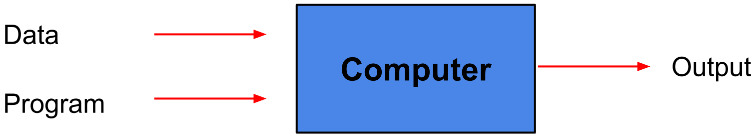


It produces a pattern based on sample data and expectations.

- It finds out how to solve the problem by itself.

How ANNs differ from conventional programming?

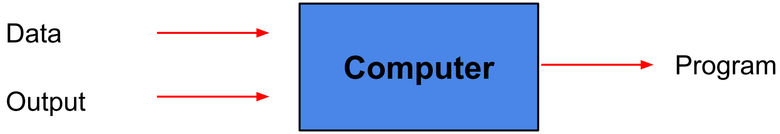
Algorithmic Programming



We tell the computer what to do and how to do it.

- It solves a problem by following a set of instructions.
- It never learns.

Artificial Neural Networks

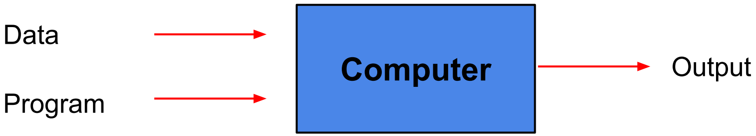


It produces a pattern based on sample data and expectations.

- It finds out how to solve the problem by itself.
- It learns by example.

How ANNs differ from conventional programming?

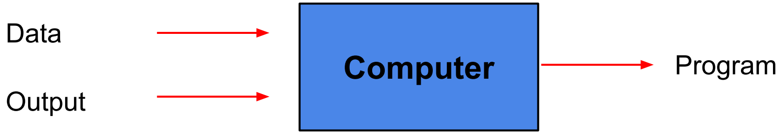
Algorithmic Programming



We tell the computer what to do and how to do it.

- It solves a problem by following a set of instructions.
- It never learns.
- Focus on the algorithm.

Artificial Neural Networks

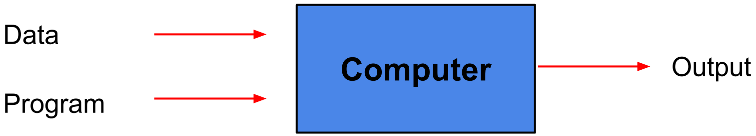


It produces a pattern based on sample data and expectations.

- It finds out how to solve the problem by itself.
- It learns by example.
- Focus on the data.

How ANNs differ from conventional programming?

Algorithmic Programming



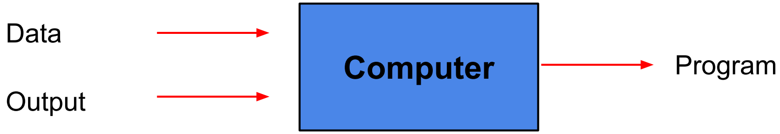
We tell the computer what to do and how to do it.

- It solves a problem by following a set of instructions.
- It never learns.
- Focus on the algorithm.

The output is always the same.

- No mistakes.
- 2 + 2 = 4 *(always)*

Artificial Neural Networks

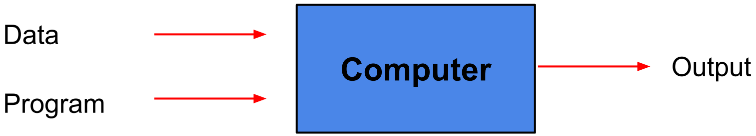


It produces a pattern based on sample data and expectations.

- It finds out how to solve the problem by itself.
- It learns by example.
- Focus on the data.

How ANNs differ from conventional programming?

Algorithmic Programming



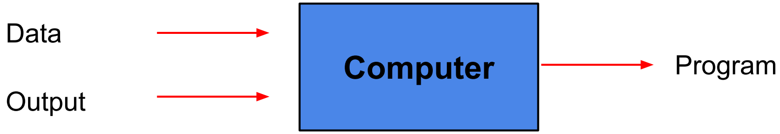
We tell the computer what to do and how to do it.

- It solves a problem by following a set of instructions.
- It never learns.
- Focus on the algorithm.

The output is always the same.

- No mistakes.
- 2 + 2 = 4 *(always)*

Artificial Neural Networks



It produces a pattern based on sample data and expectations.

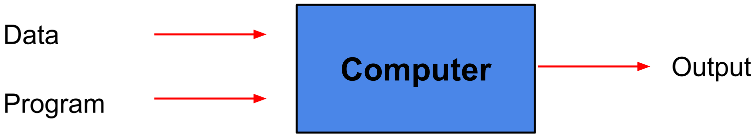
- It finds out how to solve the problem by itself.
- It learns by example.
- Focus on the data.

The output depends on network performance *(training)*.

- They make mistakes.
- 2 + 2 = 3 *(if network training is insufficient)*

How ANNs differ from conventional programming?

Algorithmic Programming



We tell the computer what to do and how to do it.

- It solves a problem by following a set of instructions.
- It never learns.
- Focus on the algorithm.

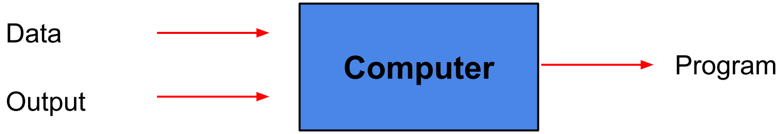
The output is always the same.

- No mistakes.
- 2 + 2 = 4 *(always)*

Representational power

- *How much is 3 plus 5?*

Artificial Neural Networks



It produces a pattern based on sample data and expectations.

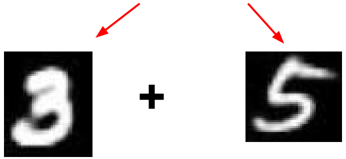
- It finds out how to solve the problem by itself.
- It learns by example.
- Focus on the data.

The output depends on network performance *(training)*.

- They make mistakes.
- 2 + 2 = 3 *(if network training is insufficient)*

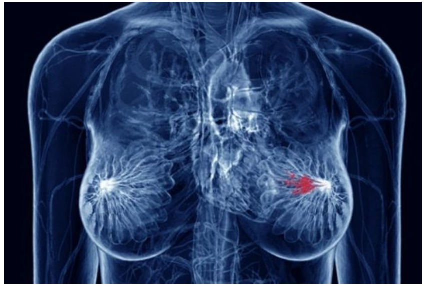
Representational power

- *How much is this plus this?*



The kind of problems we can solve with AI

- Data Security
- Personal Security
- Financial Trading
- Healthcare
- Marketing Personalization
- Fraud Detection
- Recommendations
- Online Search
- Natural Language Processing (NLP)
- Smart Cars
- etc







The kind of problems we can solve with AI

Sample problem *(quick & easy)*

- I want to buy a car. I don't want to spend too much money.

Scenario.

- Buying/Selling goods *(could be enhanced to take bidding into account)*

What I did.

- I visited a couple of local web sites and created the spreadsheet below.
- This is the first time I do this so I collected 20-30 offers.

Brand	Model	State	Year	Mileage (Km)	Market Price
Fiat	Adventure	Used	2008	60,000	$180,000.00
Fiat	Adventure	Used	2009	70,000	$190,000.00
Fiat	Adventure	Used	2013	110,000	$216,000.00
Fiat	Adventure	New	2017	0	$380,000.00

(more samples...)

What I want.

I want to get the best car available in the market at the minimum price possible.

My questions:

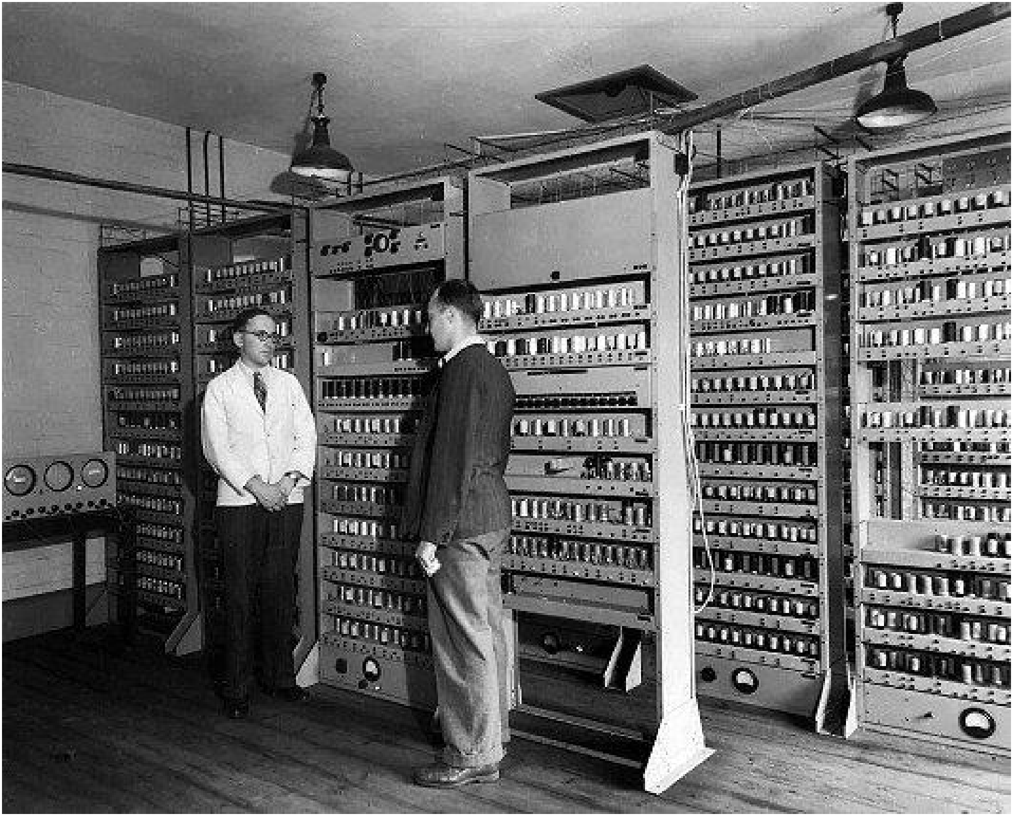
Question #1.
What is the most convenient offer (*best pick*) based on my needs?

Question #2.
Should I bid for the following car? →
Year: 2015, Mileage: 90k, Price: $120k

Question #3.
What are the top 3 options that best fit my needs?

Putting things in perspective

May 6, 1949 - Mathematical Laboratory at Cambridge University



"Today in 1949,
the Electronic Delayed Storage Automatic Computer (EDSAC),
the first practical stored-program computer,
ran its first program and performed its first calculation."

Putting things in perspective

August 25, 1950 - Toronto, Canada

"Bertie the Brain" is exhibited at the Canadian National Exhibition. It was just a computer game *(one of the first games)*

What makes it so special?
It used *Artificial Intelligence* to play tic-tac-toe with humans.

Humans win, mostly.

Putting things in perspective

May 12, 1997.

IBM supercomputer ***Deep Blue*** annihilates world champion Garry Kasparov at chess, **in 19 moves**

Putting things in perspective

2011

IBM **Watson** beats human opponents at *Jeopardy.*

Watson developed reasoning to overcome human opponents.

(The improved version for competing against humans became fully operational on Feb 2010)

Putting things in perspective

April, 2016
DeepMind AlphaGo (Google) defeats the world's best human at Go (*Lee Sedol*)
Go has more possible moves than there are atoms in the Universe.

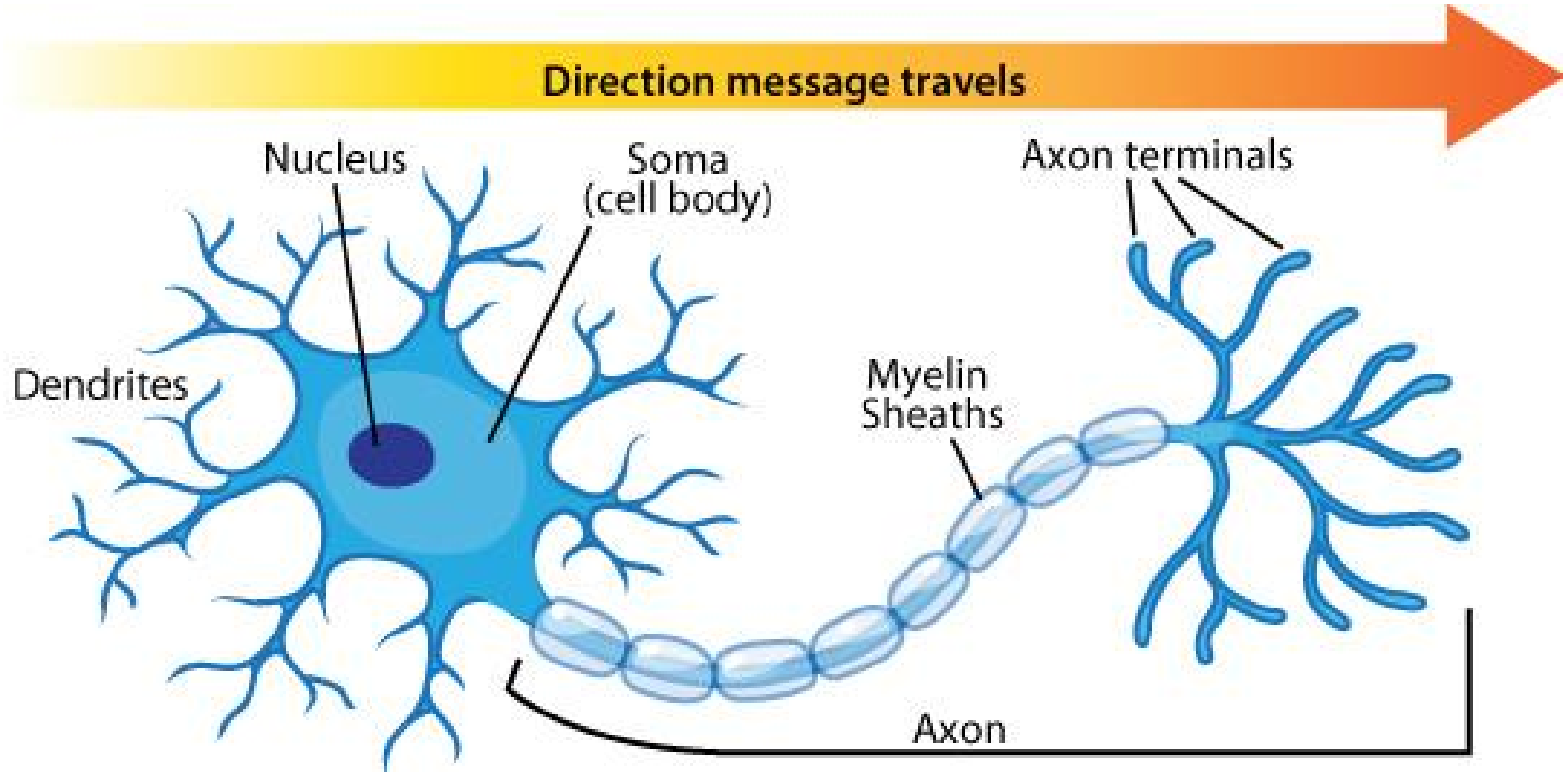
The network had to develop intuition.

- Their authors *(human programmers)* didn't understand why AlphaGo was doing what it was doing.
- Possible moves in Go: 10^{170}
- Number of atoms in the universe: 10^{80}

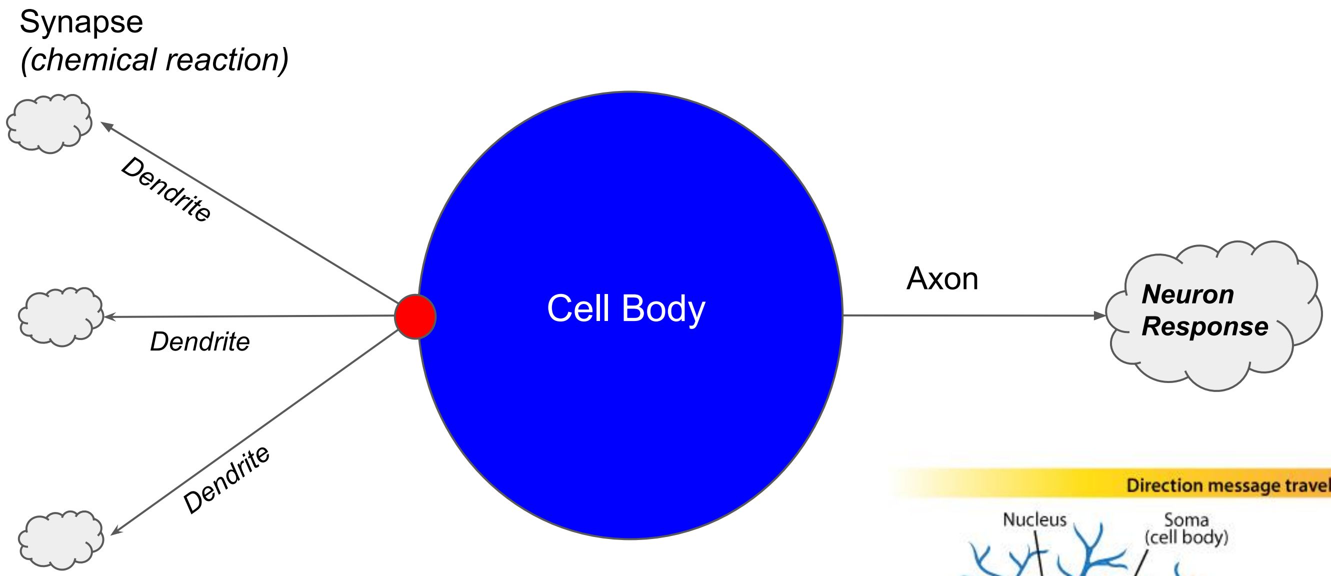
Heads up!

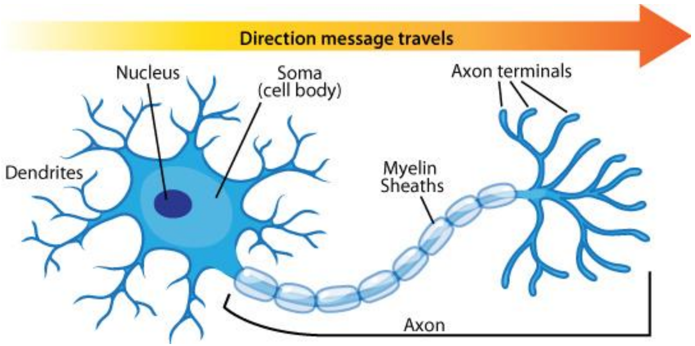
Hands ON,
From now ON

Biological model

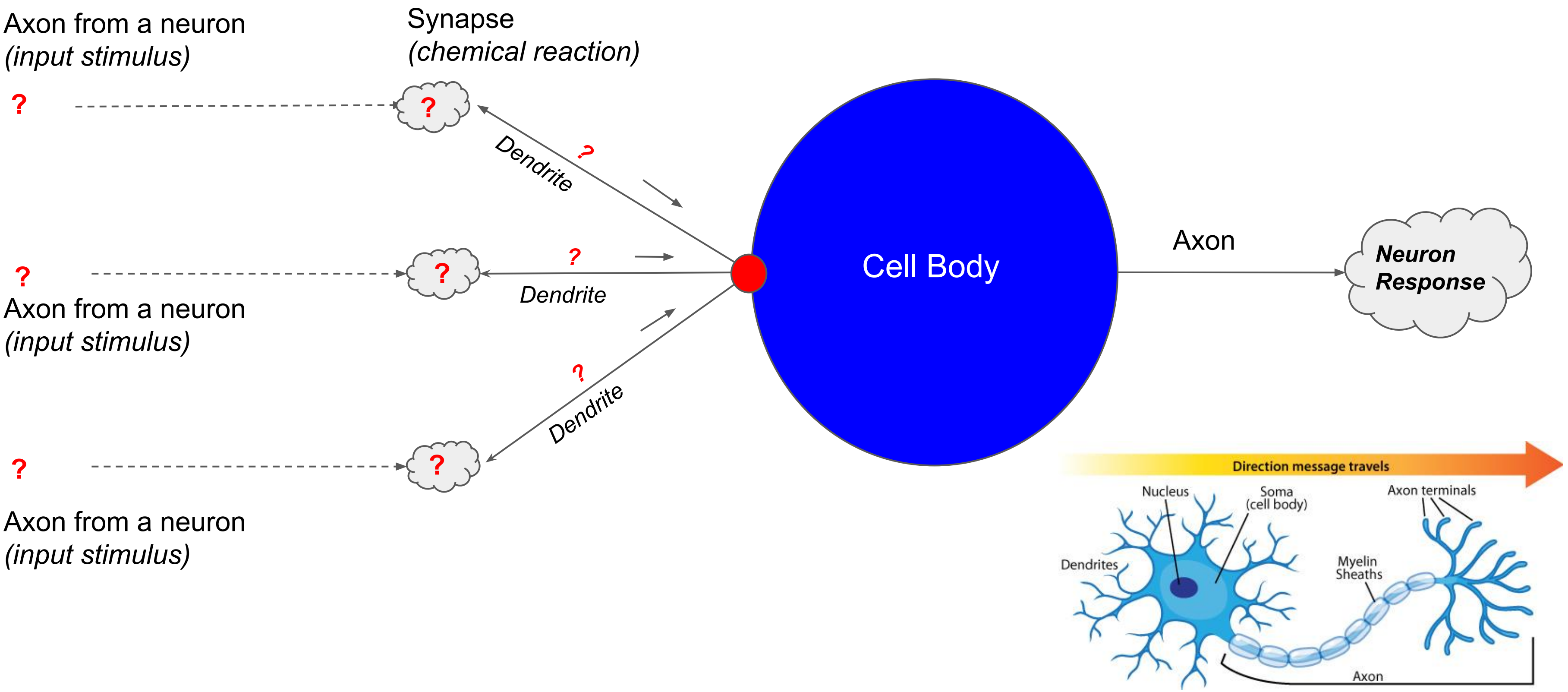


Modeling one Neuron (*Perceptron Model*)

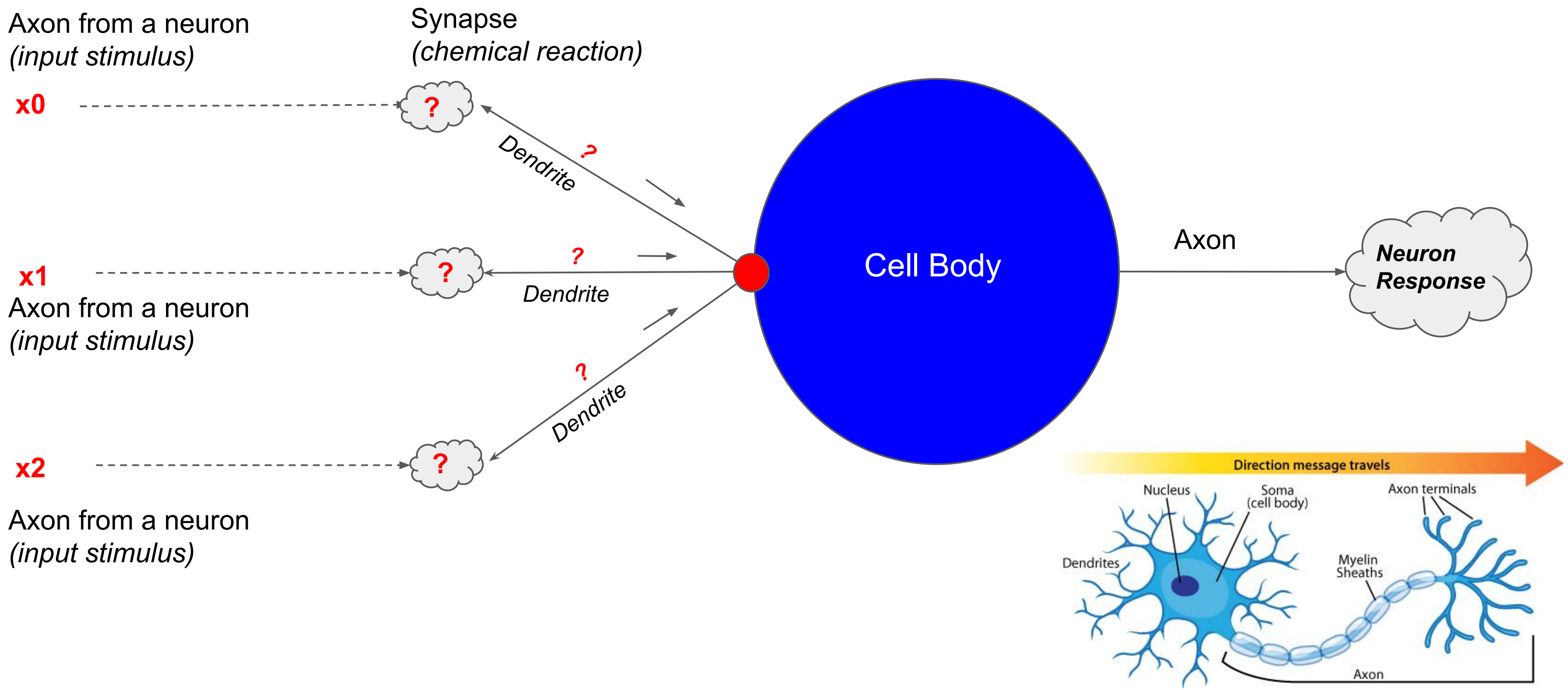




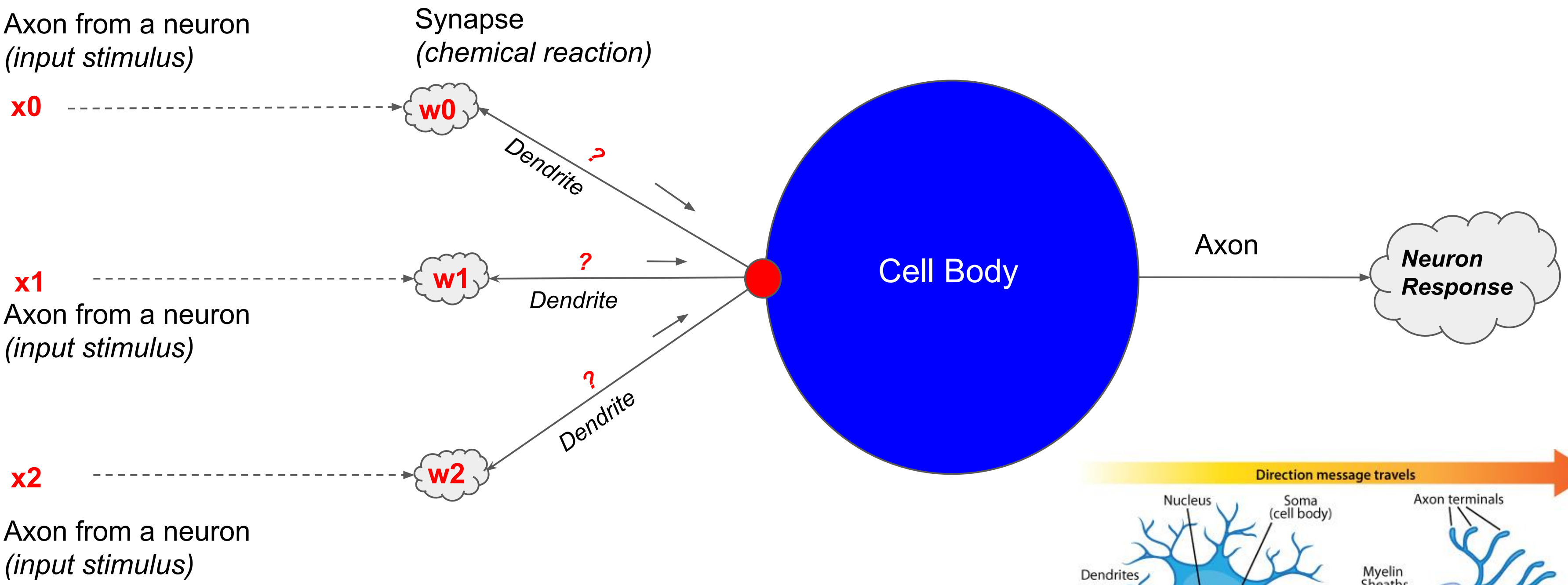
Modeling one Neuron (*Perceptron Model*)

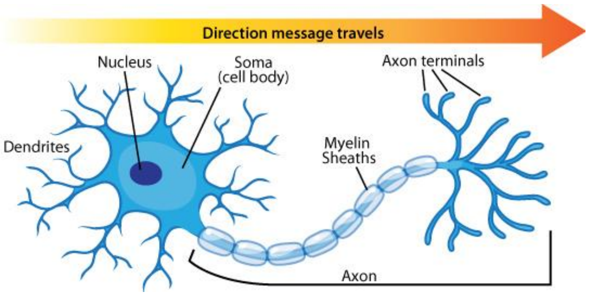


Modeling one Neuron (*Perceptron Model*)

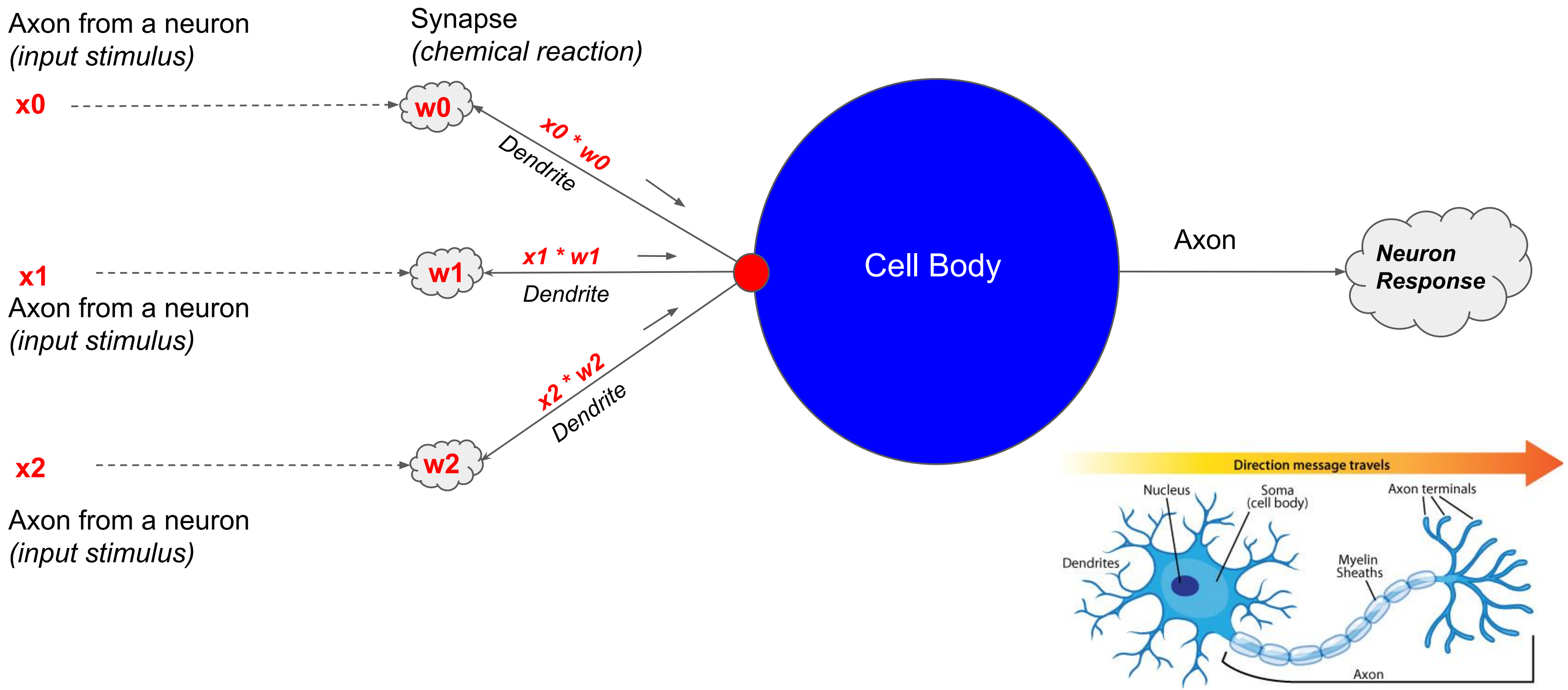


Modeling one Neuron (*Perceptron Model*)

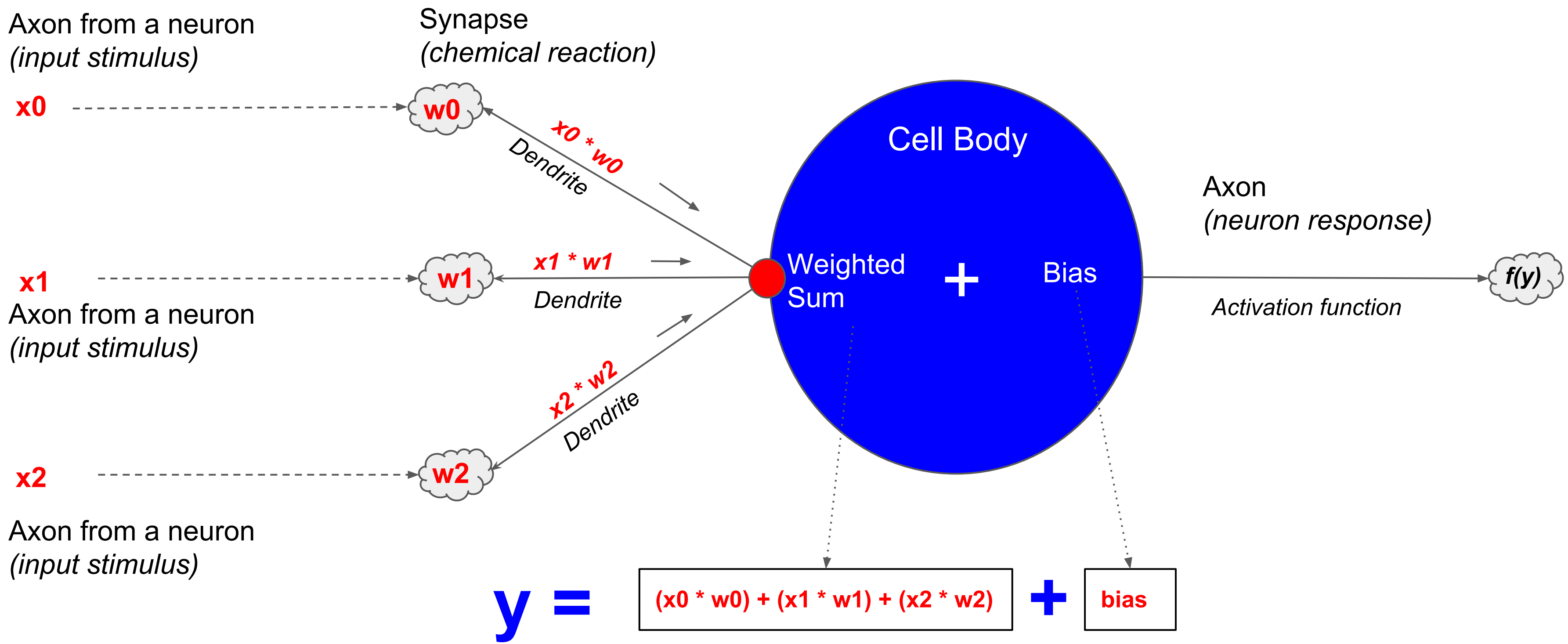




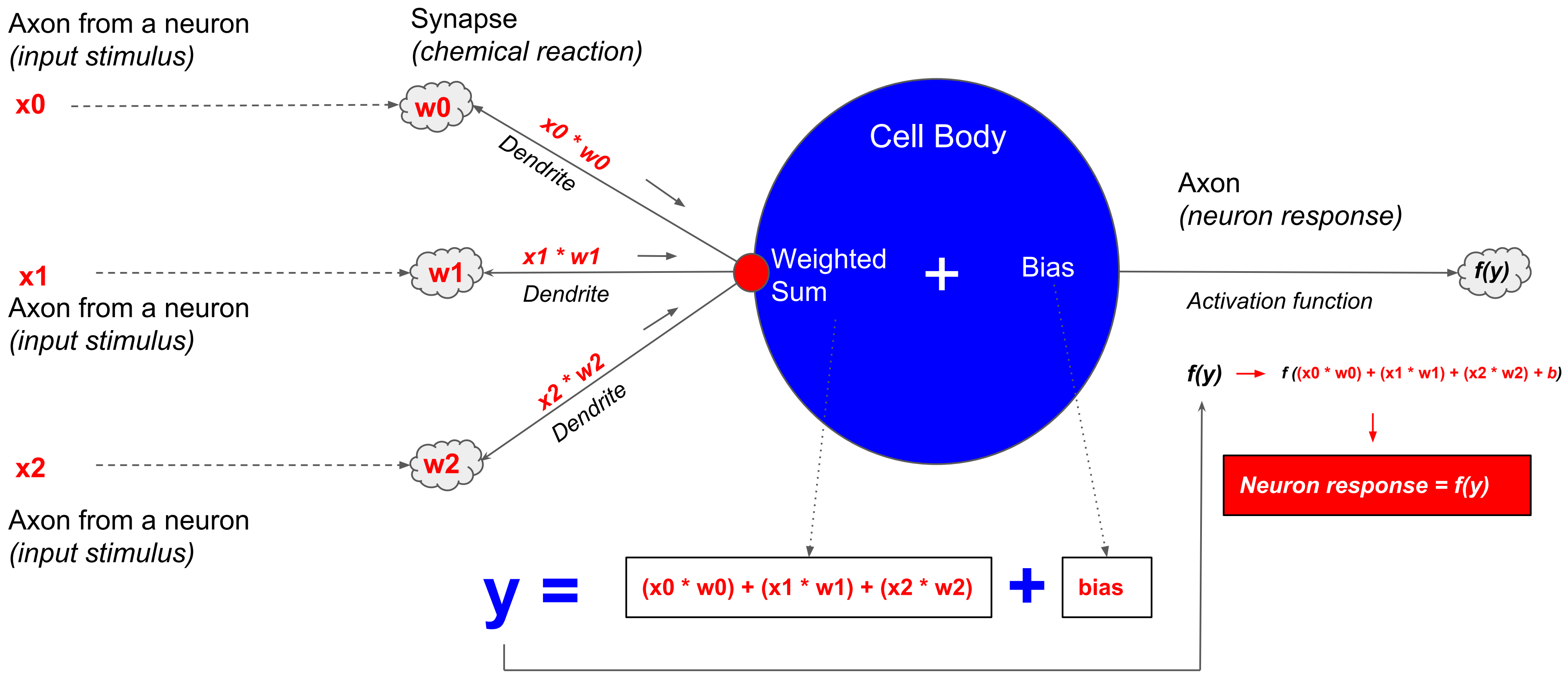
Modeling one Neuron (*Perceptron Model*)



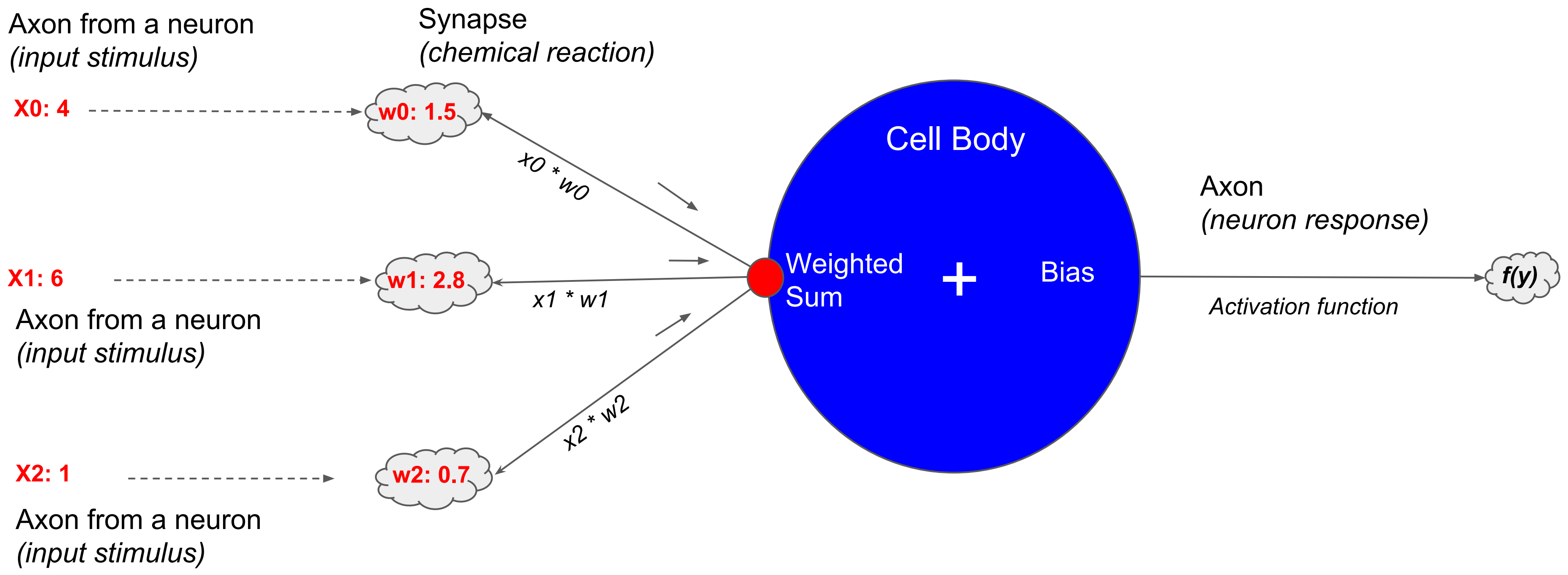
Modeling one Neuron (*Perceptron Model*)



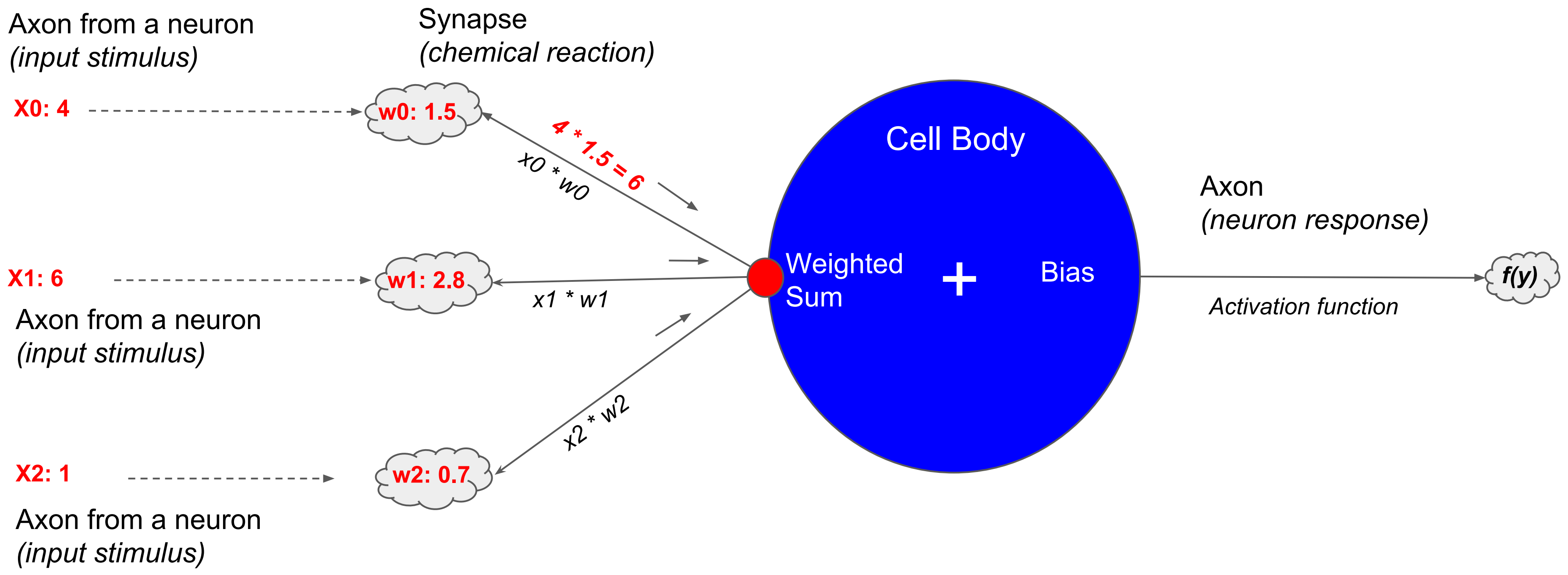
Modeling one Neuron (*Perceptron Model*)



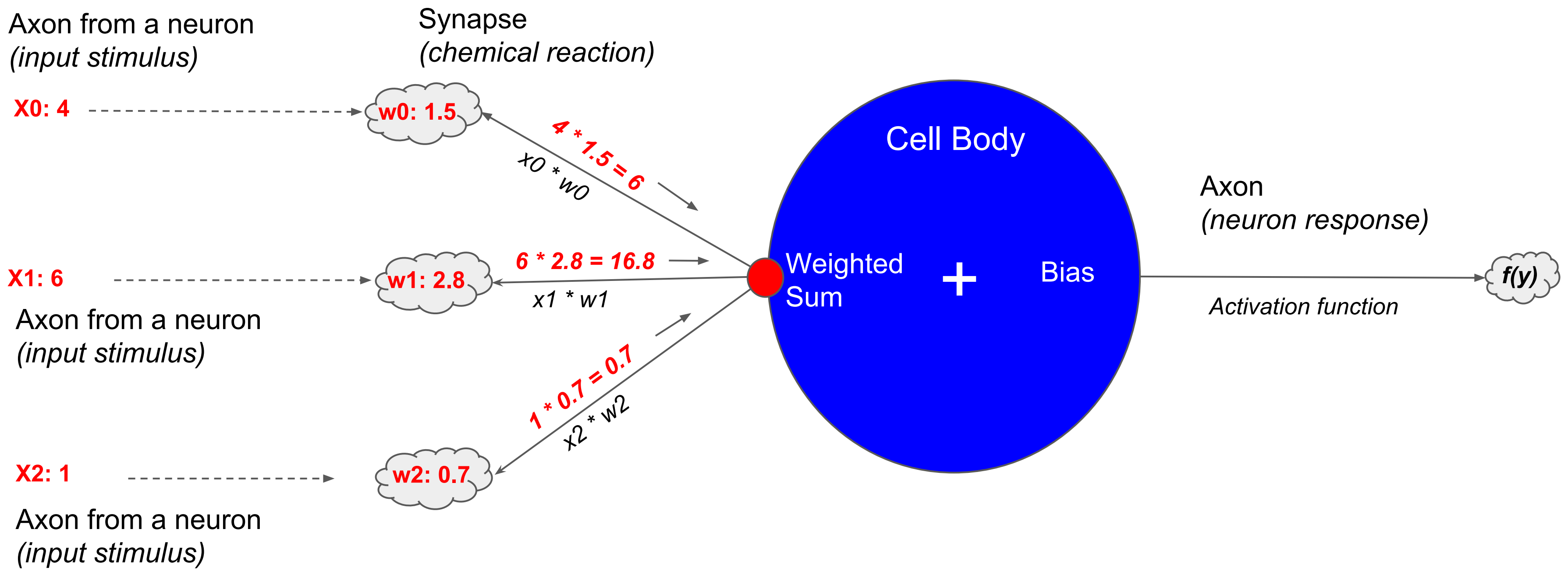
Forward Propagation - *A worked example*



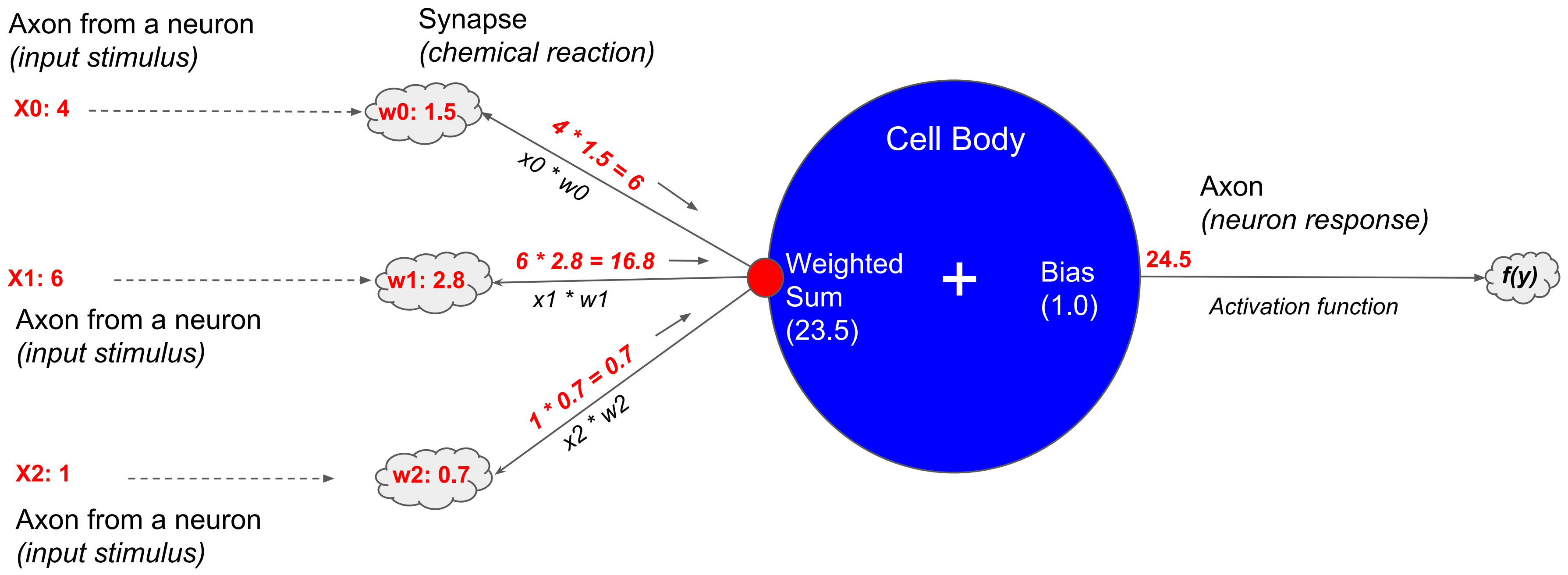
Forward Propagation - *A worked example*



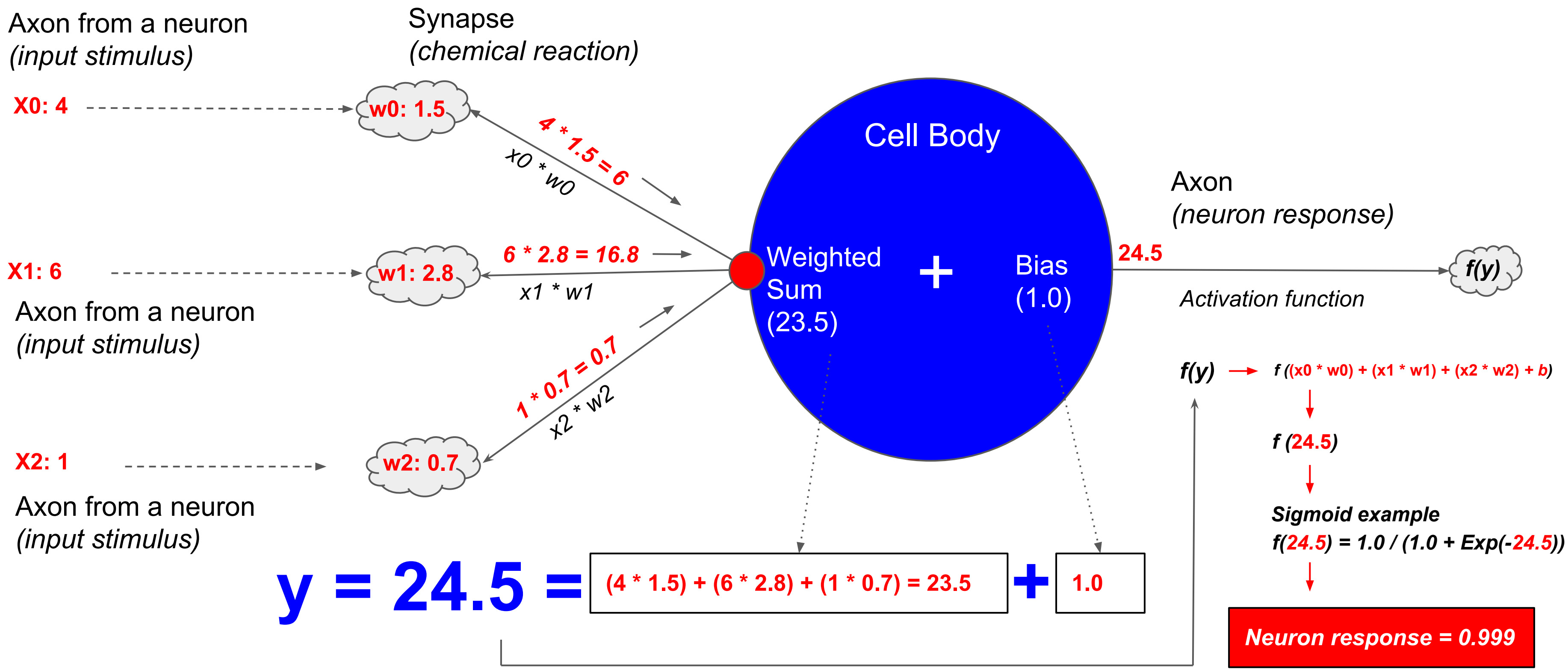
Forward Propagation - *A worked example*



Forward Propagation - *A worked example*



Forward Propagation - *A worked example*



Forward Propagation - *A worked example*

So let's start modeling this in Excel...

Dendrites (*Neuron Inputs*)

Axon from another neuron	
Axon	Input Value
x0	4
x1	6
x2	1

Dendrite	
Name	Connection Weight
w0	1.5
w1	2.8
w2	0.7

Synapse (chemical reaction)	
Name	Value
x0 * w0	6
x1 * w1	16.8
x2 * w2	0.7
Weigthed Sum	**23.5**

Cell Body (*Process*)

Cell Body	
Body Input	23.5
Bias	1
Body Response	**24.5**

Response (Output)

Neuron Response	
Activation Type	**Output**
Identity	24.5
Sigmoid	0.999999999937759
Tanh	1
ReLU	24.5

Hands on →

Forward Propagation - *A worked example*

```
/// <summary>
/// Computes the weighted summation of the input values specified.
/// </summary>
/// <param name="inputValues">Specifies the input values</param>
/// <returns>Returns the weighted sum</returns>
protected double[] ComputeWeightedResponse(double[] inputValues)
{
    // The response of each neuron available at the current layer
    double[] neuronResponse = new double[NumberOfNeurons];

    for (int targetNeuronIndex = 0; targetNeuronIndex < NumberOfNeurons; targetNeuronIndex++)
    {
        // Compute the weighted summation of the inputs
        double weightedSum = 0.0D;
        for (int sourceNeuronIndex = 0; sourceNeuronIndex < PreviousLayer.NumberOfNeurons; sourceNeuronIndex++)
        {
            weightedSum += inputValues[sourceNeuronIndex] * Weights[targetNeuronIndex][sourceNeuronIndex];
        }

        // Compute the weighted sum plus a bias contribution
        neuronResponse[targetNeuronIndex] = weightedSum + Biases[targetNeuronIndex];
    }

    // Apply the activation function
    IActivationFunction activationFunction = ActivationFunctionsFactory.CreateActivationFunction(ActivationFunctionType);
    neuronResponse = activationFunction.Execute(neuronResponse);

    return neuronResponse;
}
```

Source code is provided for educational purposes only

Hands on →

Forward Propagation - *A worked example*

```
#region Interface Implementation

/// <summary>
/// Computes the response of the network based on the inputs specified.
/// </summary>
/// <param name="inputValues">Specifies the input values</param>
/// <returns>Returns the response of the network</returns>
public double[] Predict(double[] inputValues)
{
    // Check the network configuration
    VerifyNetworkDefinition(inputValues);

    // Compute the response of each layer
    int depth = Layers.Length;
    for (int layerIndex = 0; layerIndex < depth; layerIndex++)
    {
        double[] previousLayerResponse = (layerIndex == 0) ? inputValues : Layers[layerIndex - 1].LayerResponse;
        Layers[layerIndex].LayerResponse = Layers[layerIndex].Predict(previousLayerResponse);
    }

    // Return the response of the network
    return Layers[depth - 1].LayerResponse;
}
```

Source code is provided for educational purposes only

Hands on →

Why do we need a bias? What it does?

A bias neuron **allows the classifier to shift the decision boundary** to the left or to the right.
Without a bias, your layer (or network) is prone to produce a probability of 50% - 50%

Scenario.
Imagine that all your inputs are zero, or all weights are zero, or both at the same time.

f(y) → *f (0)*

f(y) → *sigmoid (0)* → *0.5*

Example 1 → ***(hidden layer)***

Hidden Layer Response			
Neuron	Weighted Sum	Bias	Neuron Output
H1	0.000000	0	**0.50000000**
H2	0.000000	0	**0.50000000**

Example 2 → ***(output layer followed by SoftMax)***

Output Layer Response			
Neuron	**Weighted Sum**	**Bias**	**Neuron Output**
O1	0	0	**0.5**
O2	0	0	**0.5**

Neuron	Output value	Exp(o)	Probability
O1	0.5	1.648721271	**50.00%**
O2	0.5	1.648721271	**50.00%**
Scale		3.297442541	

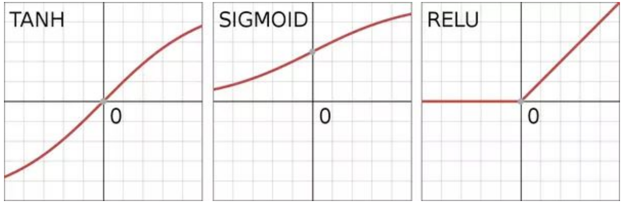
That’s why.

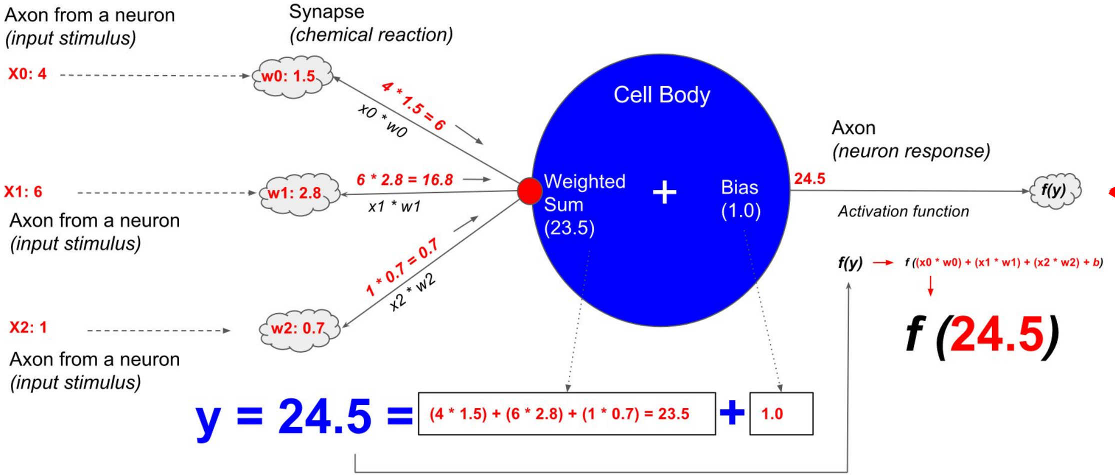
Hands on →

Forward Propagation - *A worked example*

What if we try with different activation functions?

Activation Type	Formula	Neuron Response
Sigmoid	f(24.5) = 1.0 / (1.0 + Exp(-24.5))	f(24.5) = 0.999999999937759
Hyperbolic Tangent	f(24.5) = Tanh(24.5)	f(24.5) = 1.0
Relu	f(24.5) = 24.5 < 0 ? 0 : 24.5	f(24.5) = 24.5





Sigmoid = 0.999999999937759

Tanh = 1.0

Relu = 24.5

Takeaway

A neuron can be modeled as a function → *f(y)*

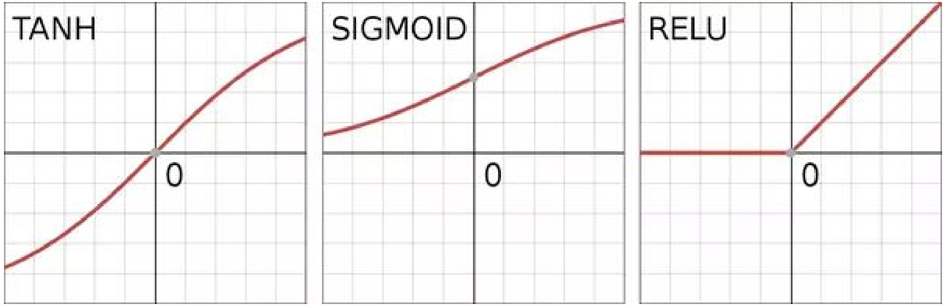
Neuron Activation Functions

Purpose.

- An activation function mimics the **<u>response of a neuron</u>**.
- Any activation function has their <u>derivative counterpart</u>, by definition.

Commonly used functions:

- Rectified Linear Unit (*aka ReLU*)
- Logistic Function (*aka Sigmoid*)
- Hyperbolic Tangent (*aka TanH*)
- SoftMax
- ...and others



Hyperbolic Tangent:
f(y) = Tanh(y)

Sigmoid:
f(y) = 1.0 / (1.0 + Exp(-y))

RELU:
f(y) = y < 0.0 ? 0 : y

Neuron Activation Functions *(example - ReLU)*

Cell Body

Weighted Sum + Bias

Axon
(neuron response)

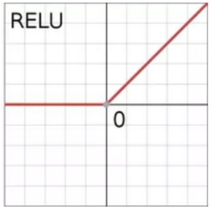
f(y)

Body response — *Activation function* — *Neuron response*

	Body response	Activation function		Neuron response
Example #1	**y = 3.5**	**Is 3.5 higher than 0?**	***(yes)***	**f(y) = 3.5**
Example #2	**y = -0.1**	**Is -0.1 higher than 0?**	***(no)***	**f(y) = 0**
Example #3	**y = -0.0**	**Is -0.0 higher than 0?**	***(no)***	**f(y) = 0**

ReLU - How it works.
If the value is higher than zero then get excited.
Otherwise get inhibited (produce a “0”)

```
return y < 0 ? 0 : y;
```



Creating a neural network - *step by step*

Input Layer

?

?

?

The values submitted to the network

Hidden Layer

Neurons in this layer help the network to identify emerging patterns.

They act as a **descriptor**.

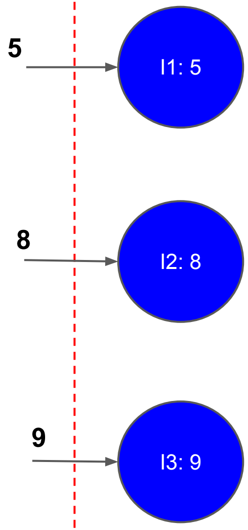
Output Layer

Neurons in this layer are responsible for classifying the pattern identified.

They act as a **classifier**.

Creating a neural network - *step by step*

Input Layer



Hidden Layer

Neurons in this layer help the network to identify emerging patterns.

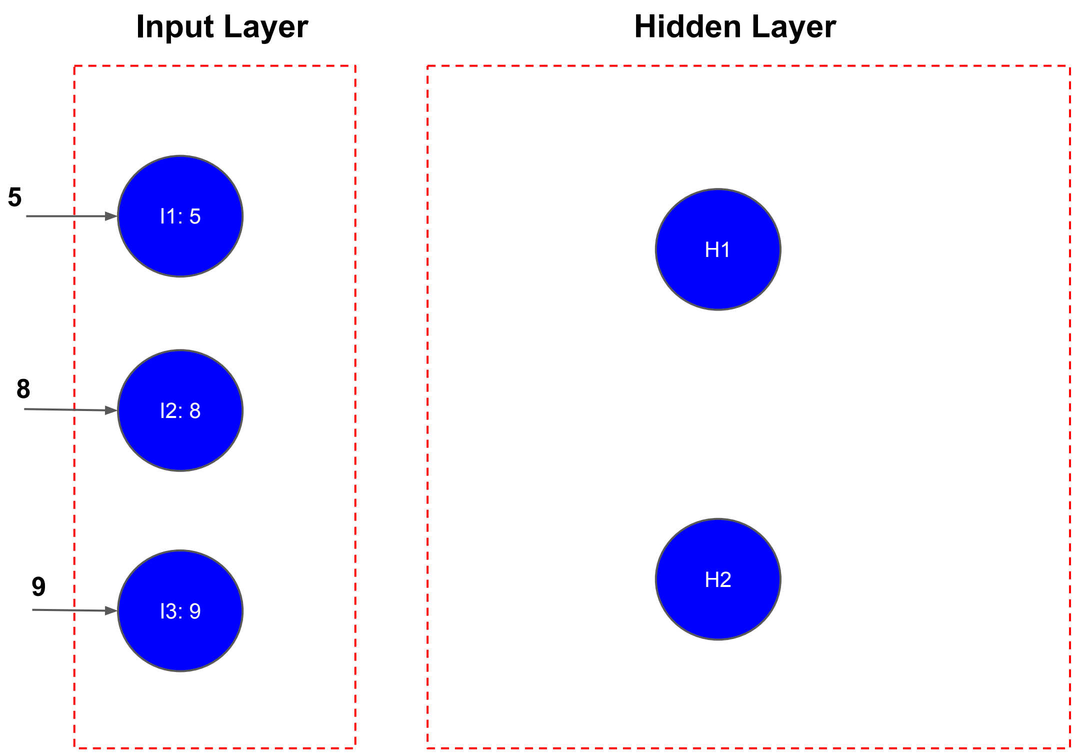
They act as a **descriptor**.

Output Layer

Neurons in this layer are responsible for classifying the pattern identified.

They act as a **classifier**.

Creating a neural network - *step by step*

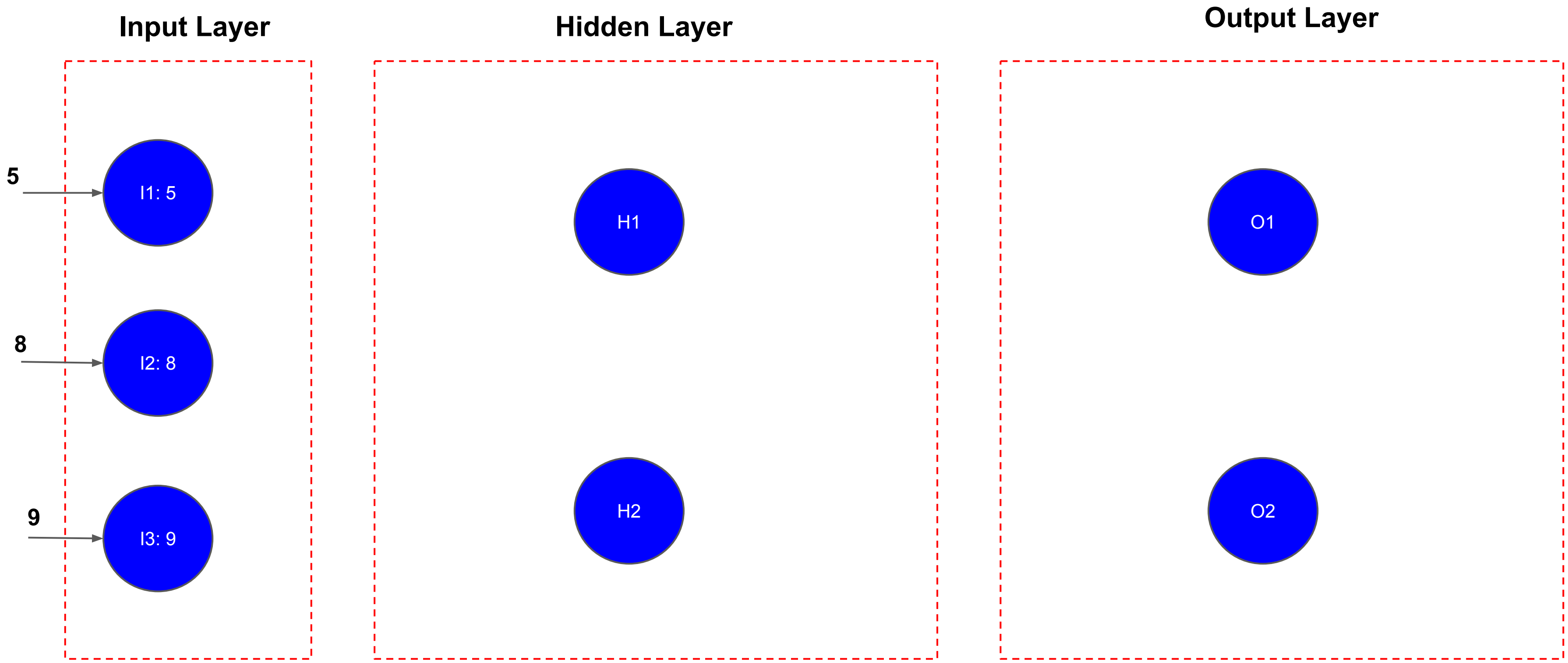


Output Layer

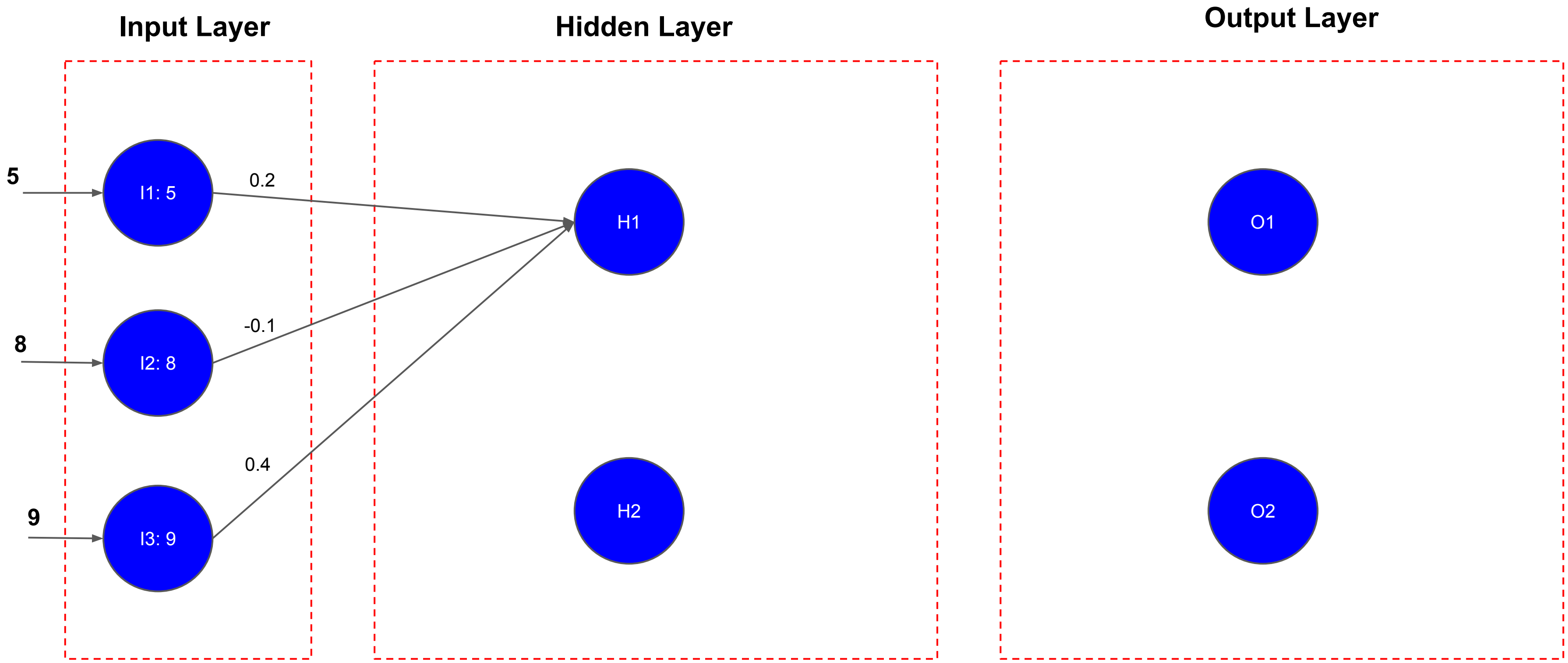
Neurons in this layer are responsible for classifying the pattern identified.

They act as a **classifier**.

Creating a neural network - *step by step*

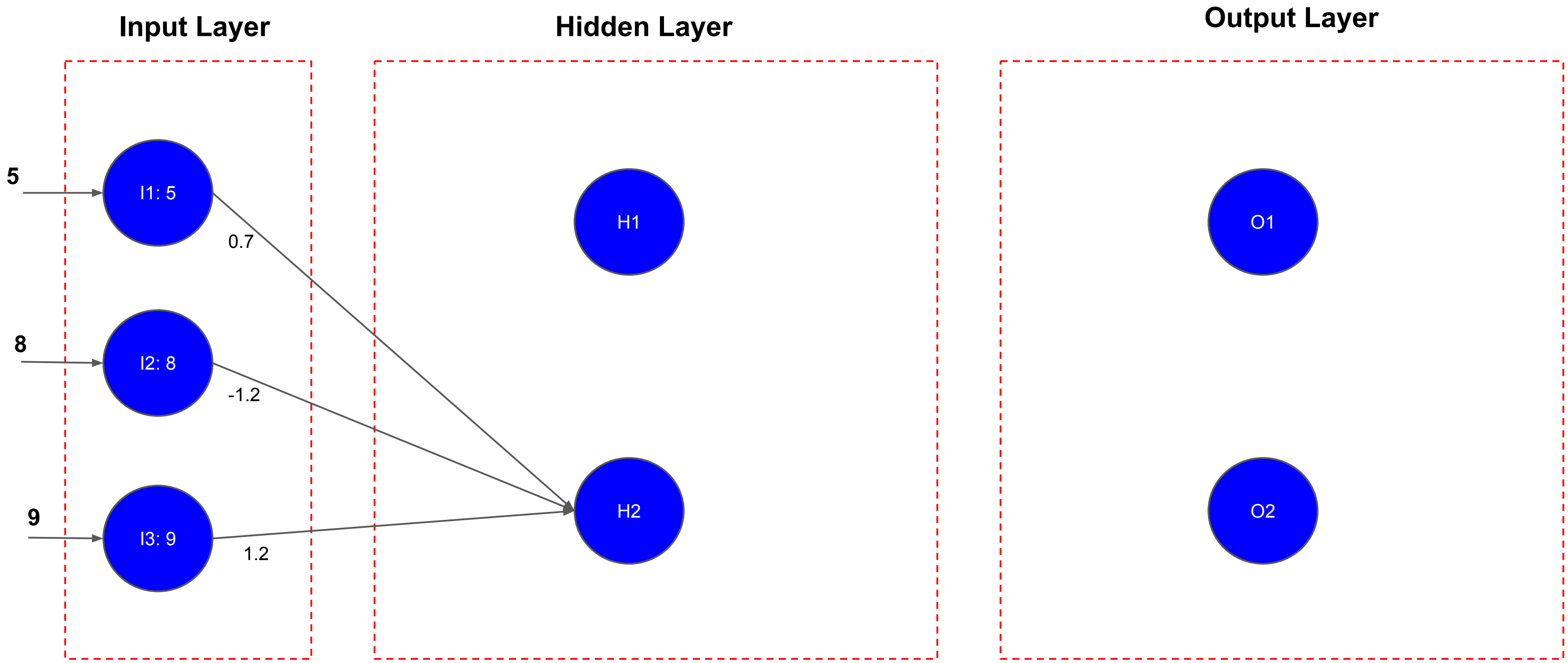


Creating a neural network - *step by step*



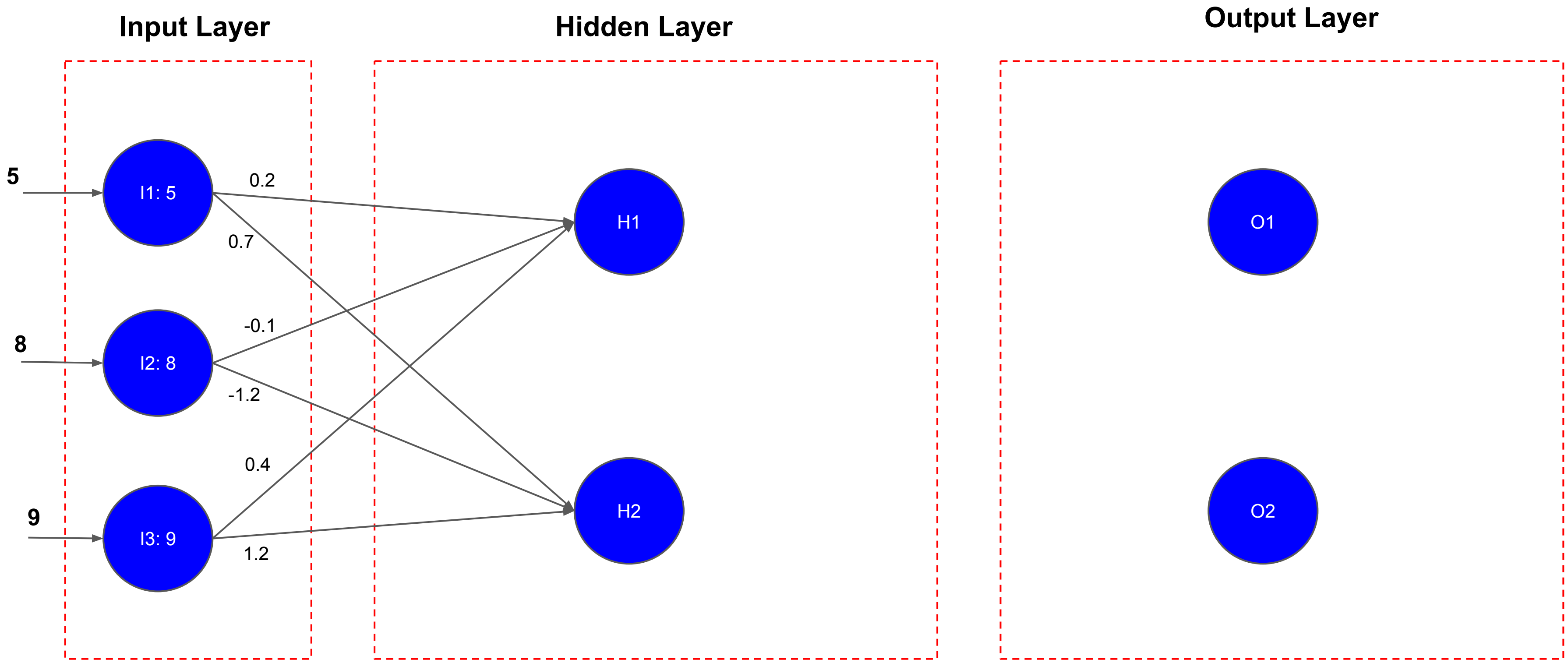
Establish a connection between each neuron. Assign a "*weight*" to each connection.

Creating a neural network - *step by step*

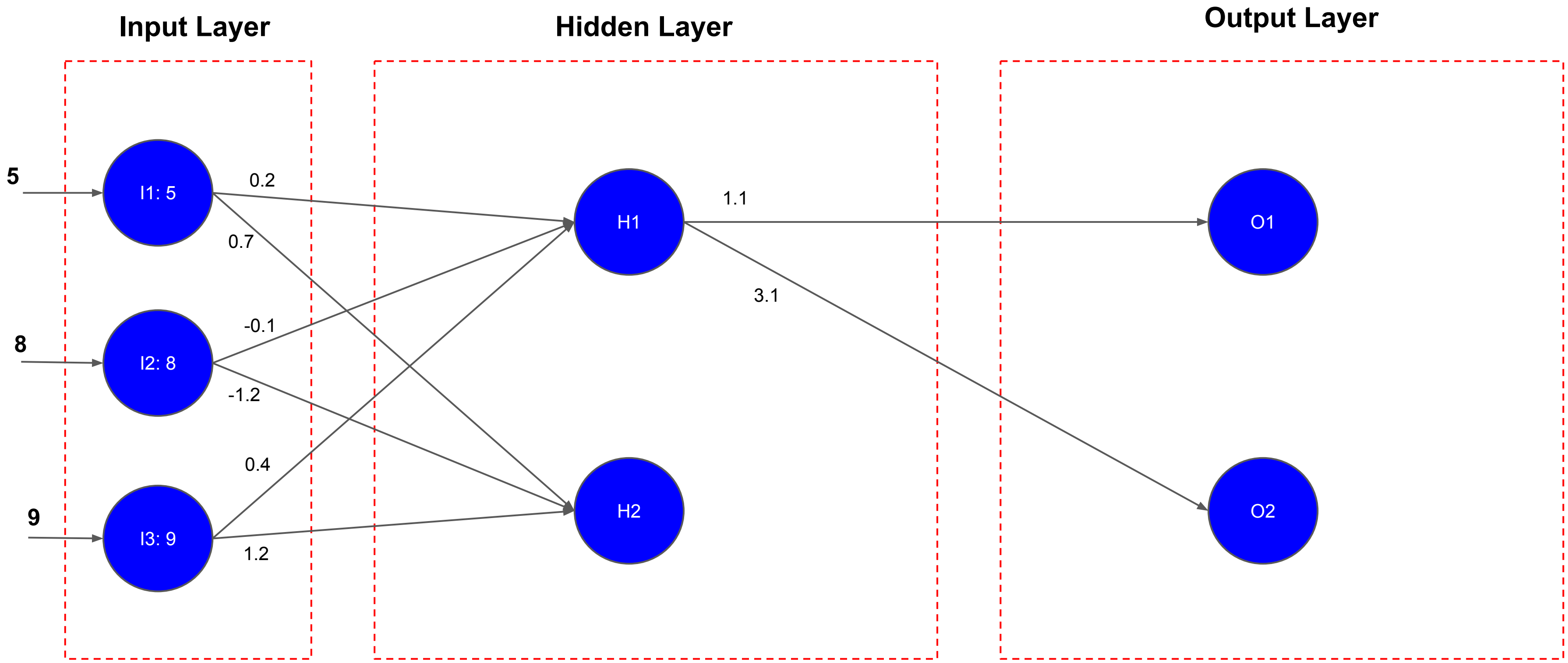


Establish a connection between each neuron. Assign a "*weight*" to each connection.

Creating a neural network - *step by step*

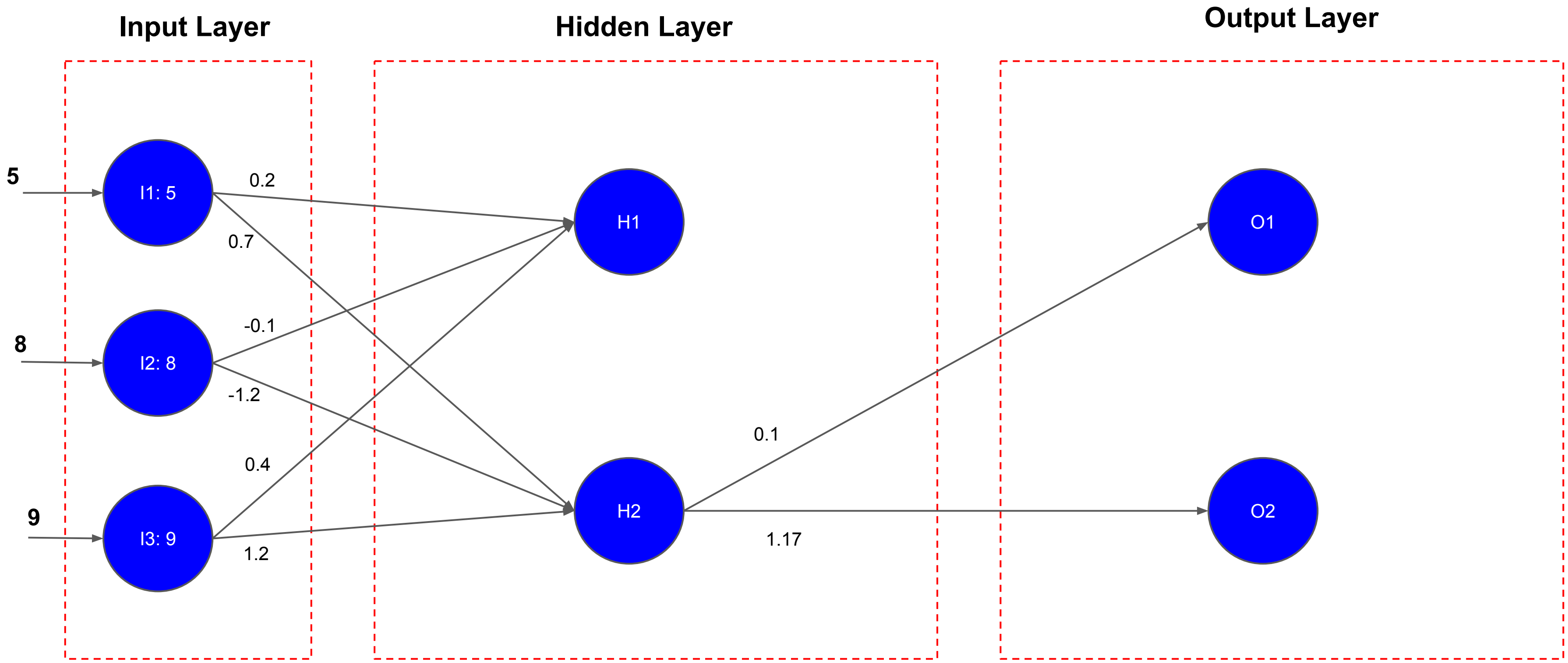


Establish a connection between each neuron. Assign a "*weight*" to each connection.



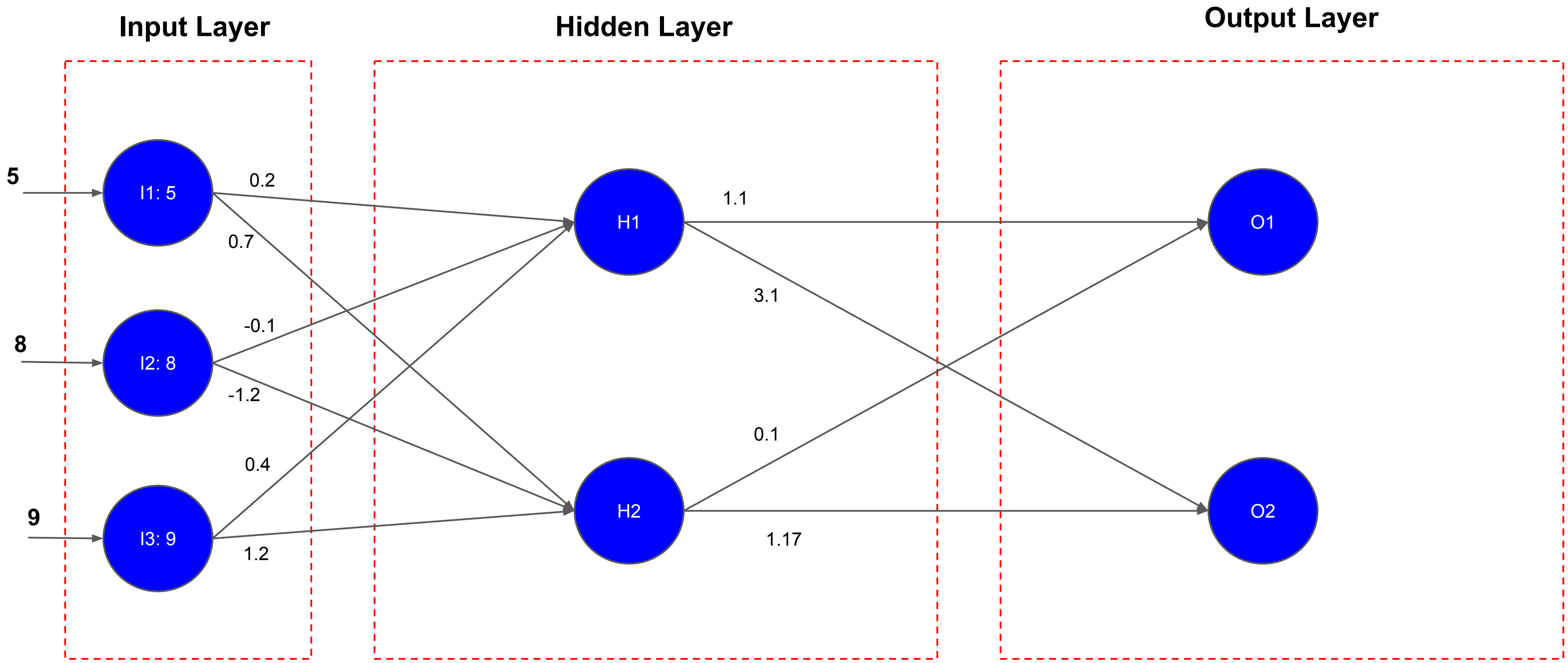
Establish a connection between each neuron. Assign a "*weight*" to each connection.

Creating a neural network - *step by step*



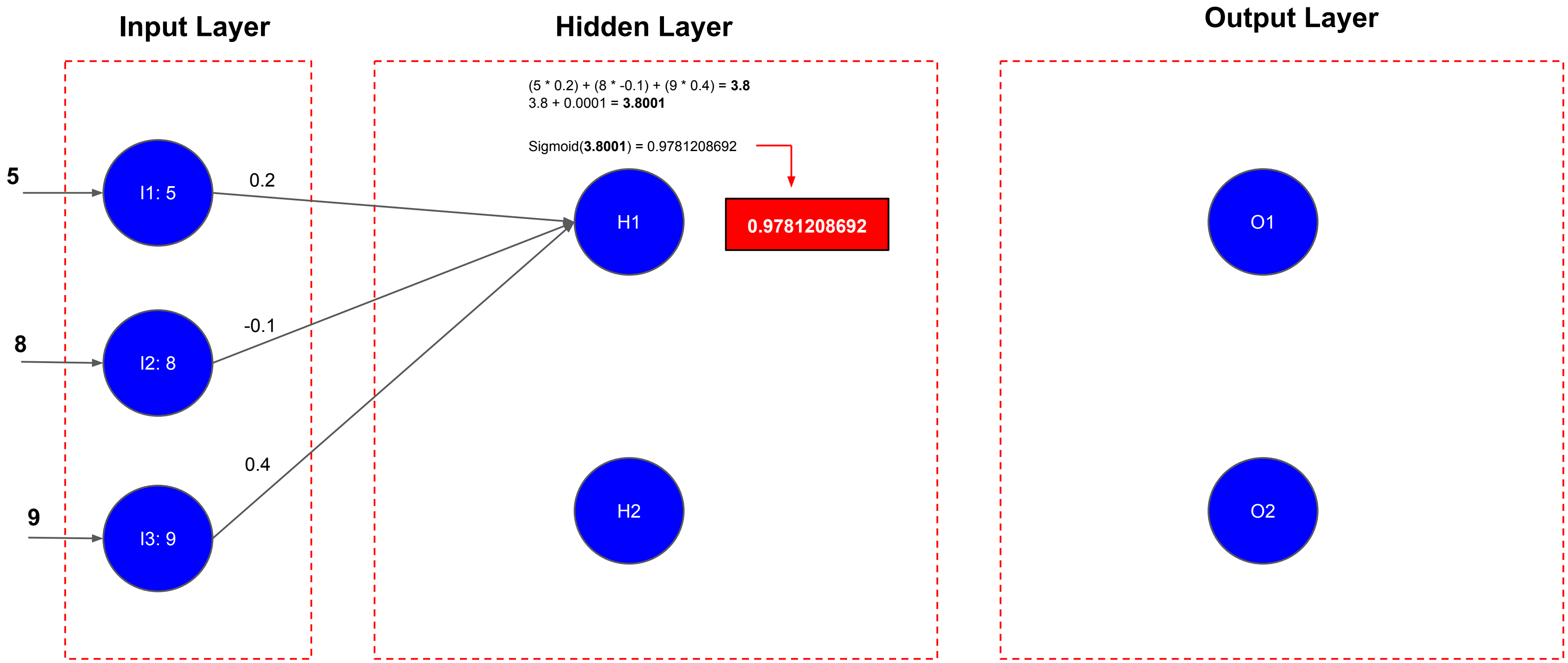
Establish a connection between each neuron. Assign a "*weight*" to each connection.

Creating a neural network - *step by step*



Establish a connection between each neuron. Assign a "*weight*" to each connection.

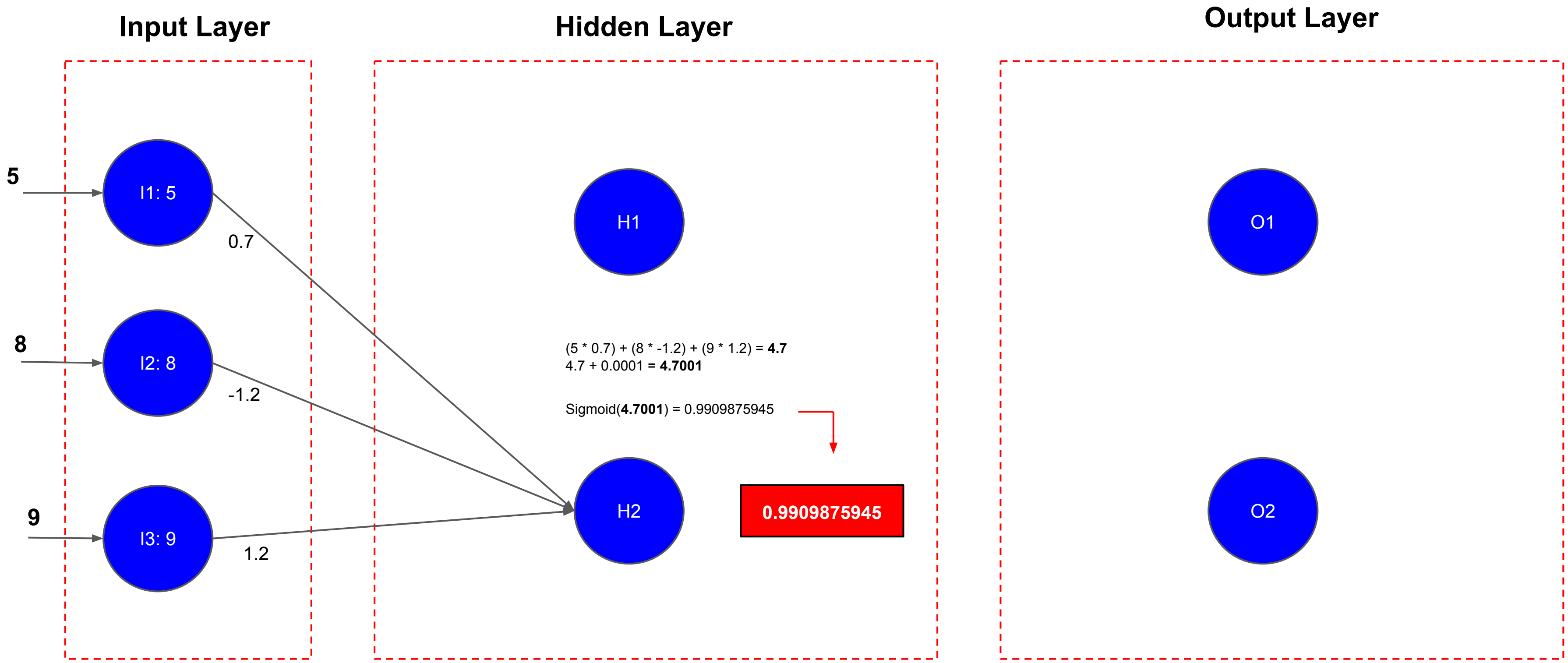
Creating a neural network - *step by step*



Compute the response of each neuron.

<u>As an example *(just to make it easier)*</u>:
Assume the bias is 0.0001 for all neurons
Assume the activation function is a Sigmoid for all neurons

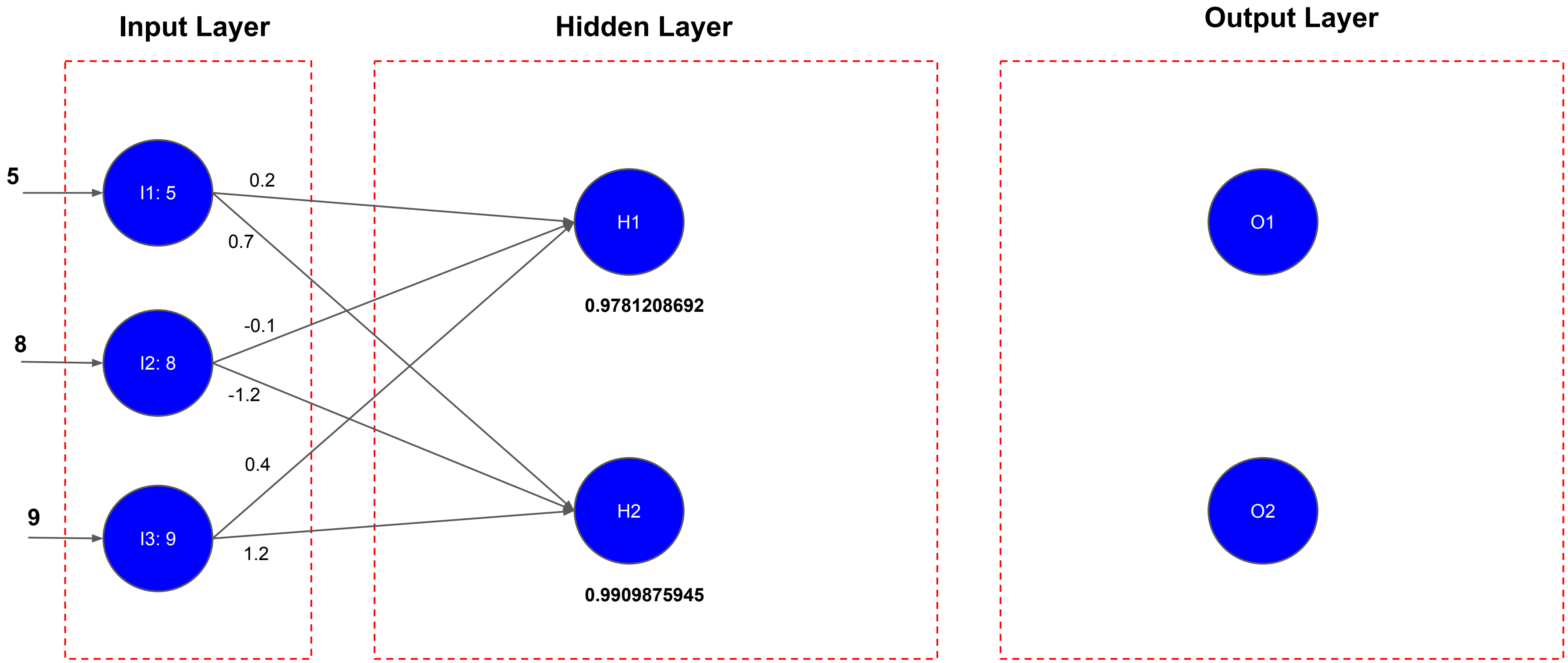
Creating a neural network - *step by step*



Compute the response of each neuron.

As an example *(just to make it easier)*:
Assume the bias is 0.0001 for all neurons
Assume the activation function is a Sigmoid for all neurons

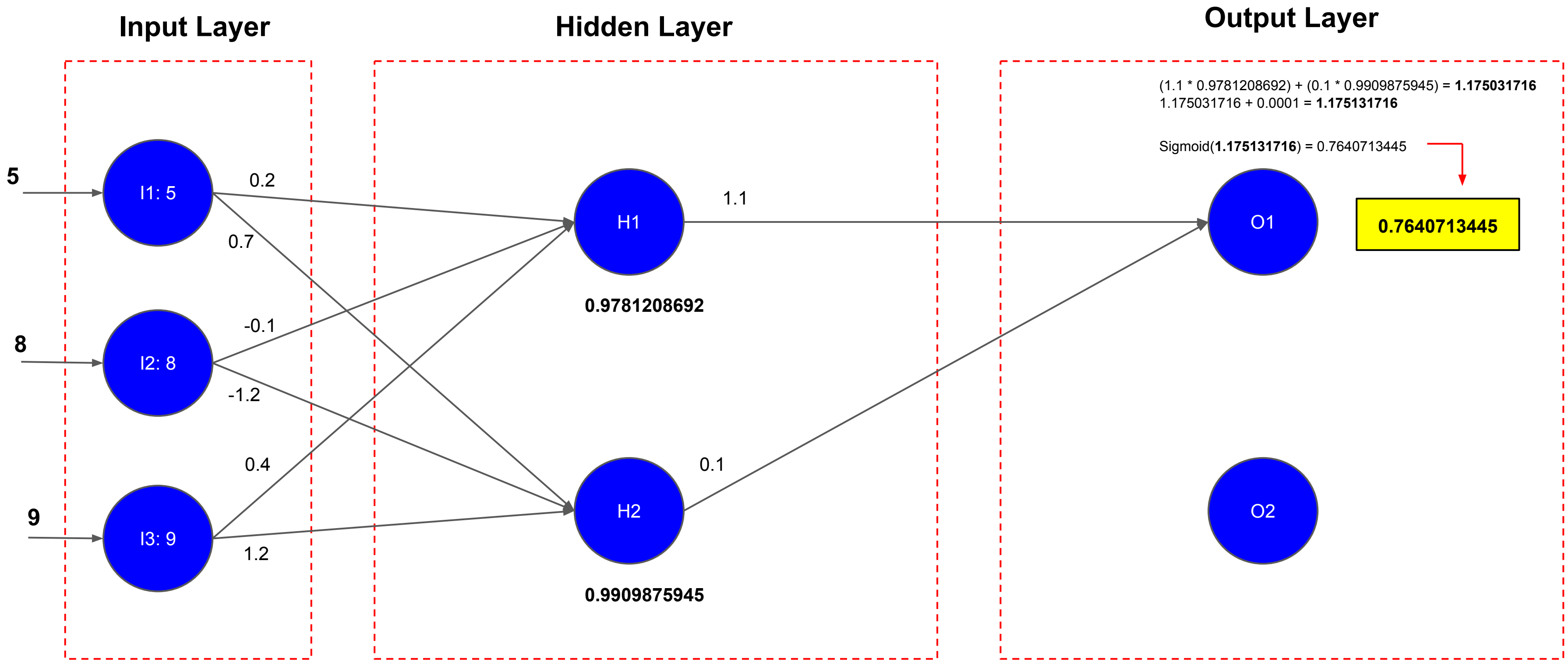
Creating a neural network - *step by step*



Compute the response of each neuron.

<u>As an example *(just to make it easier)*</u>:
Assume the bias is 0.0001 for all neurons
Assume the activation function is a Sigmoid for all neurons

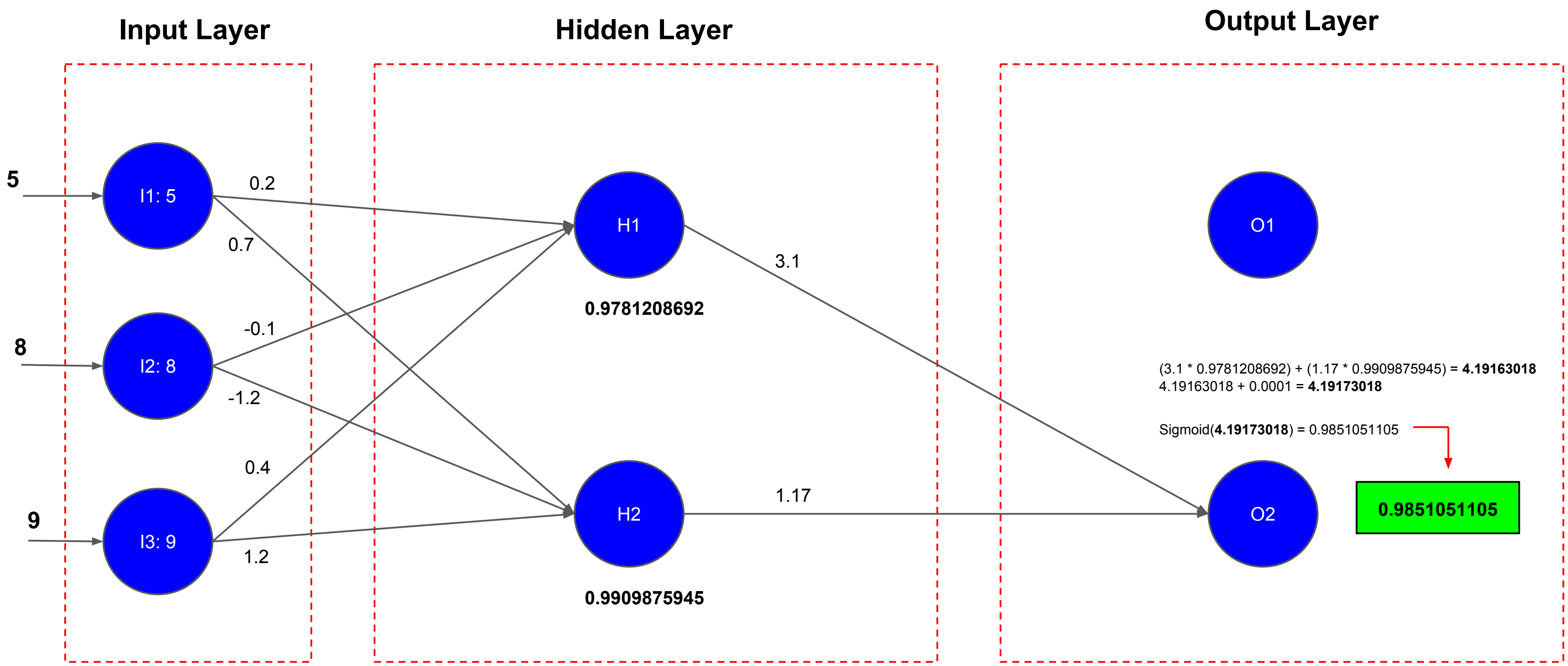
Creating a neural network - *step by step*



Compute the response of each neuron.

As an example *(just to make it easier)*:
Assume the bias is 0.0001 for all neurons
Assume the activation function is a Sigmoid for all neurons

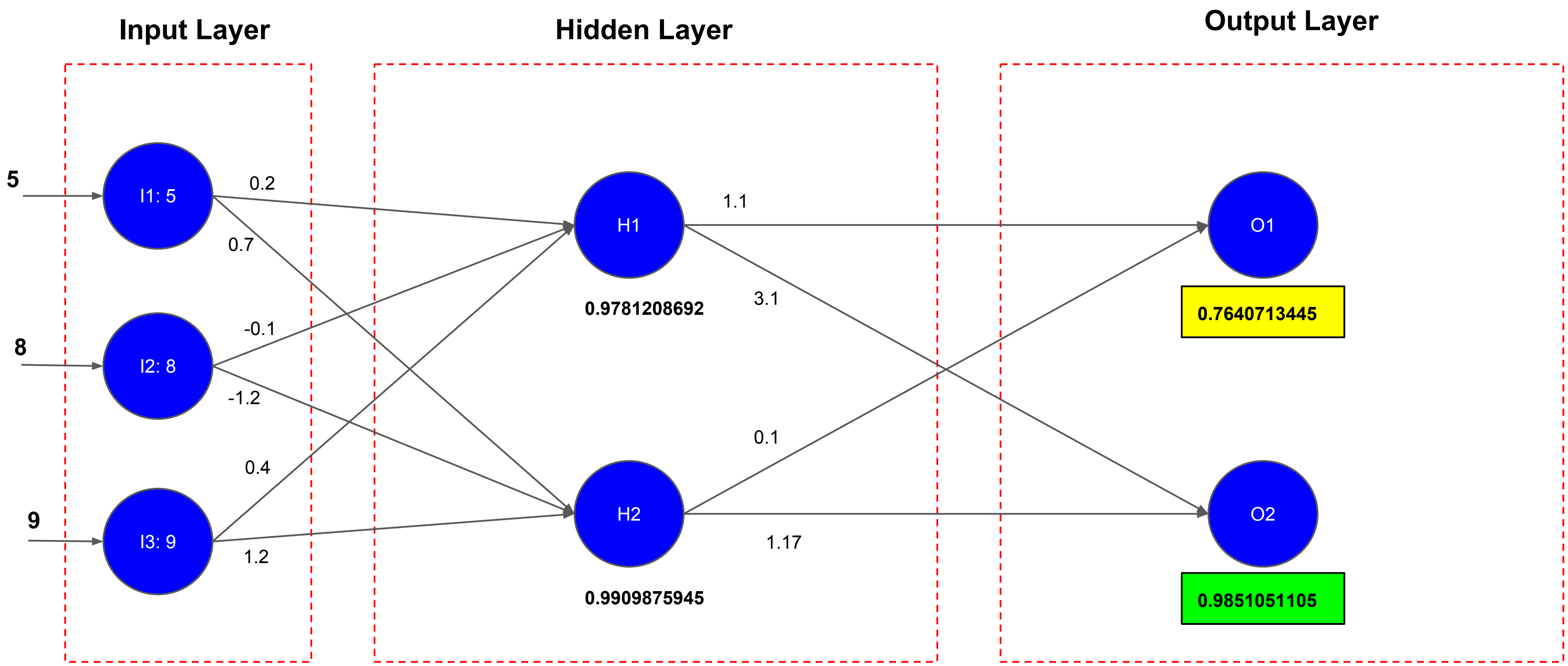
Creating a neural network - *step by step*



Compute the response of each neuron.

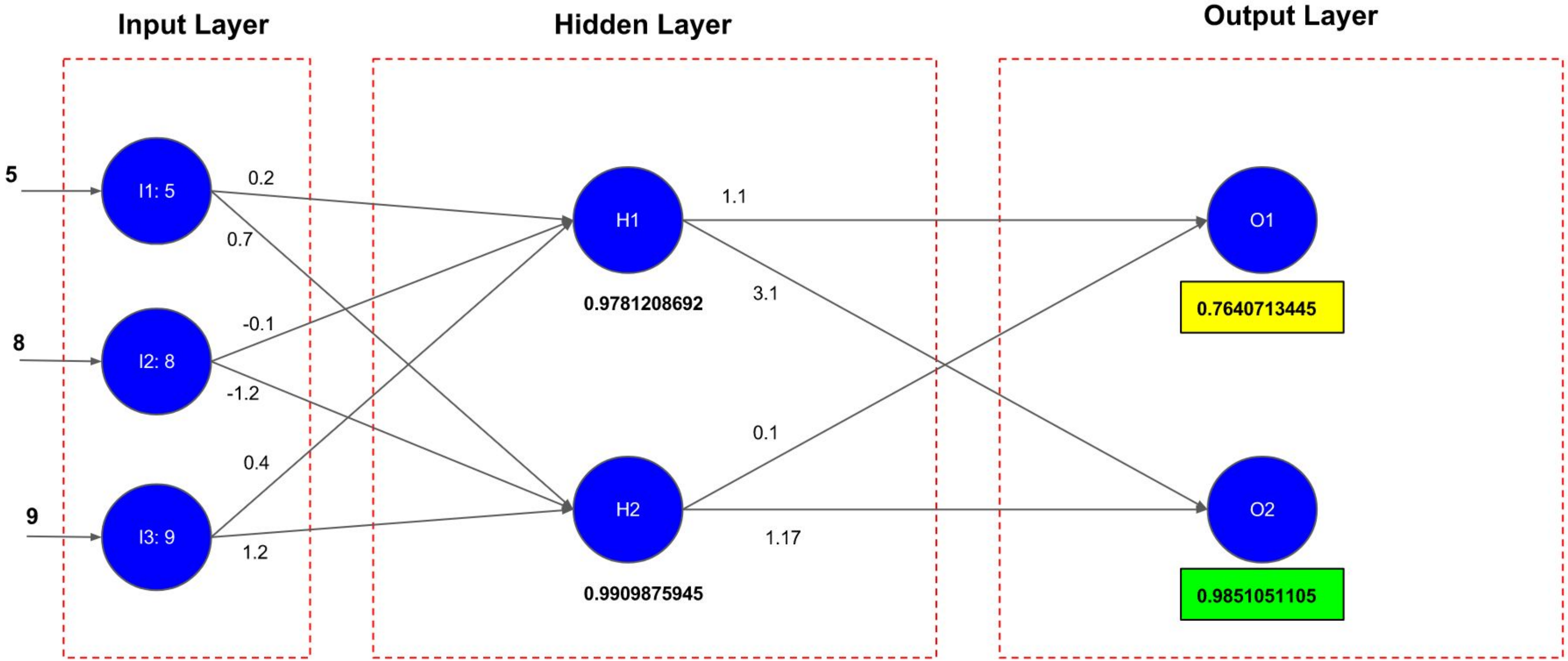
<u>As an example *(just to make it easier)*</u>:
Assume the bias is 0.0001 for all neurons
Assume the activation function is a Sigmoid for all neurons

Creating a neural network - *step by step*



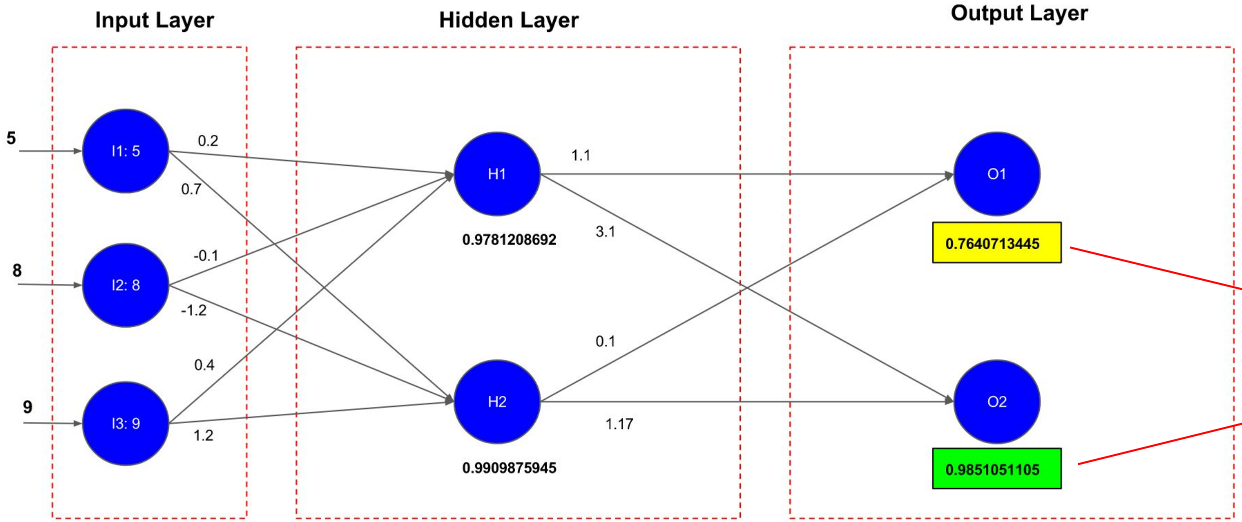
Done!

Checkpoint: Are we good so far?



Next: How to turn this data into useful information →

Creating a neural network - *step by step*



Neuron	Output value	Exp(o)
O1	0.7640713445	2.146999626
O2	0.9851051105	2.678093365
Scale	*Sum of values*	**4.825092991**

Neuron	Output value	Exp(o)	Scale	Scaled value	Probability
O1	0.7640713445	2.146999626	4.825092991	0.44496544	**44.4965%**
O2	0.9851051105	2.678093365	4.825092991	0.55503456	**55.5035%**
			Sum check!	1	100%

My prediction, as a neural network:

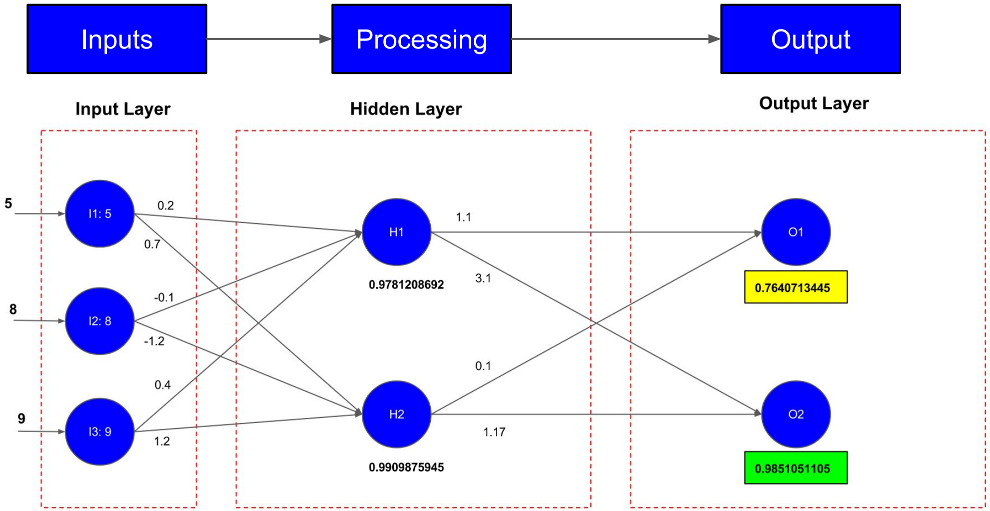
"I am 55.5% sure that O2 is the right answer."

Hands on →

Summary

Takeaways:

- A ***neuron*** can be modeled as function → *f(y)*
- Neurons connect with other neurons. The strength *(weight)* of such connections represent **knowledge**.
- One or more neurons grouped together represent a ***layer*** → *(e.g. input layer, hidden layer, output layer)*
- One or more layers grouped together make a Neural Network.
- A neural network is a function too.



Checkpoint

Achievements.

1. **You created your first Artificial Neural Network from scratch - *step by step.***
   a. You defined the topology of your first network.

2. **You now have a comprehensive understanding on how a single neuron works.**
   a. You modeled a single neuron.
   b. You know how neurons work.

3. **Now you know how a neural network makes a prediction.**
   a. You managed to provide a network response based on a set of input values.
   b. You managed to interpret the response of the Neural Network.
   c. You were able to turn mere data into useful information.

Level up →

How can we make the network learn from this *first* experience?

Learning from mistakes

Supervised Learning

Actual Response

Machine _(network response)_

O1 **0.7640713445**

O2 **0.9851051105**

Network prediction

(This is what the machine predicted)

Feedback

(This is what the human wants)

Desired Response

Human Supervisor _(me, Claudio)_

1

This neuron should be a "1"

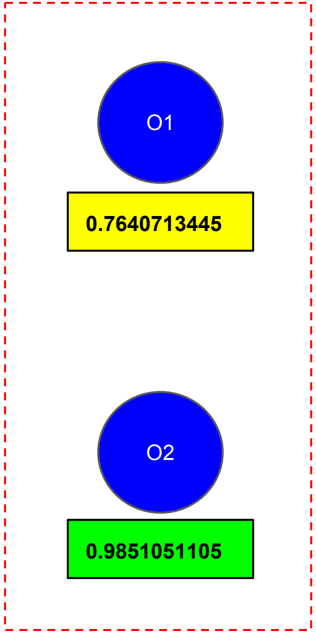
0

This neuron should be a "0"

Backpropagation

What is the distance between the desired response and the actual response? ***(cost function)***

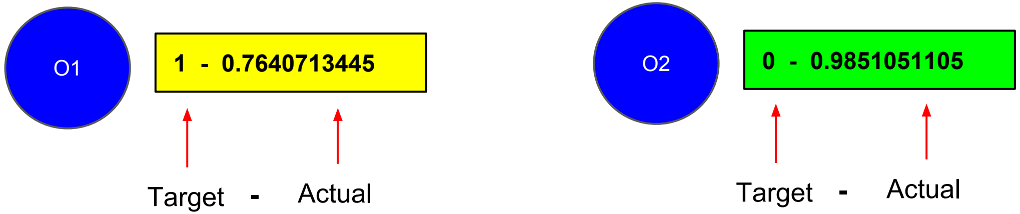
Output Layer



<u>For example</u>:

- Suppose we want neuron O1 to produce a “1”
- Suppose we want neuron O2 to produce a “0”

So the error at each neuron is:
The **expected value** - The **actual value** produced by the neuron



Backpropagation

What is the distance between the desired response and the actual response? ***(cost function)***

Output Layer

O1

0.7640713445

O2

0.9851051105

For example:

- Suppose we want neuron O1 to produce a “1”
- Suppose we want neuron O2 to produce a “0”

So the error at each neuron is:
The **expected value** - The **actual value** produced by the neuron

O1 **1 - 0.7640713445**

Target - Actual

Neuron O1 failed by: **0.2359286555**

(should be zero)

O2 **0 - 0.9851051105**

Target - Actual

Neuron O2 failed by: **-0.9851051105**

(should be zero)

Our goal is to minimize these values

Step #1 - Compute the error at the output Layer

O1

(this is the response of neuron O1)

0.7640713445

(this is what we want for neuron O1)

1

1 - 0.7640713445

*(the **distance** between "**what we want**" and "**what we've got**")*
Expected Value - Actual Value

0.2359286555

(this is the cost at neuron O1)

Step #1 - Compute the error at the output Layer

O1

(this is the response of neuron O1)

0.7640713445

(this is what we want for neuron O1)

1

1 - 0.7640713445

*(the __**distance**__ between "**what we want**" and "**what we've got**")*
Expected Value - Actual Value

0.2359286555

(this is the cost at neuron O1)

Derivative of the value produced by neuron O1

0.180266325

0.180266325

y * (1 - y) → *(Sigmoid derivative)*

0.7640713445 * (1 - 0.7640713445)

Step #1 - Compute the error at the output Layer

O1

(this is the response of neuron O1)

0.7640713445

(this is what we want for neuron O1)

1

1 - 0.7640713445

(the ***distance*** *between “****what we want****” and “****what we’ve got****”)*

Expected Value - Actual Value

0.2359286555

(this is the cost at neuron O1)

Derivative of the value produced by neuron O1

0.180266325

y * (1 - y) → *(Sigmoid derivative)*

0.7640713445 * (1 - 0.7640713445)

(this is the error signal at neuron O1)

0.180266325 * **0.2359286555** = **0.04252999169**

Backpropagation

Step #1 - Compute the error at the output Layer

O2

(this is the response of neuron O2)

0.9851051105

(this is what we want for neuron O2)

0

0 - 0.9851051105

*(the **__distance__** between "**what we want**" and "**what we've got**")*
Expected Value - Actual Value

-0.9851051105

(this is the cost at neuron O2)

Backpropagation

Step #1 - Compute the error at the output Layer

O2

(this is the response of neuron O2)

0.9851051105

(this is what we want for neuron O2)

0

0 - 0.9851051105

*(the **<u>distance</u>** between "**what we want**" and "**what we've got**")*
Expected Value - Actual Value

-0.9851051105

(this is the cost at neuron O2)

Derivative of the value produced by neuron O2

0.01467303175 → **0.01467303175**

y * (1 - y) → *(Sigmoid derivative)*

0.01467303175 * (1 - 0.01467303175)

Backpropagation

Step #1 - Compute the error at the output Layer

O2

(this is the response of neuron O2)

0.9851051105

(this is what we want for neuron O2)

0

0 - 0.9851051105

*(the __**distance**__ between “**what we want**” and “**what we’ve got**”)*
Expected Value - Actual Value

-0.9851051105

(this is the cost at neuron O2)

Derivative of the value produced by neuron O2

0.01467303175

y * (1 - y) → *(Sigmoid derivative)*

0.01467303175 * (1 - 0.01467303175)

(this is the error signal at neuron O2)

0.01467303175 * **-0.9851051105** = **-0.01445447857**

Backpropagation

Step #1 - Compute the error at the output Layer

Error Signal at the Output Layer				
Neuron	Neuron Output	Feedback	Output Derivative	Error Signal at (O)
O1	0.7640713445	0.2359286555	0.180266325	**0.04252999169**
O2	0.9851051105	-0.9851051105	0.01467303175	**-0.01445447857**

Derivative * *Feedback (distance)* = *Error signal*

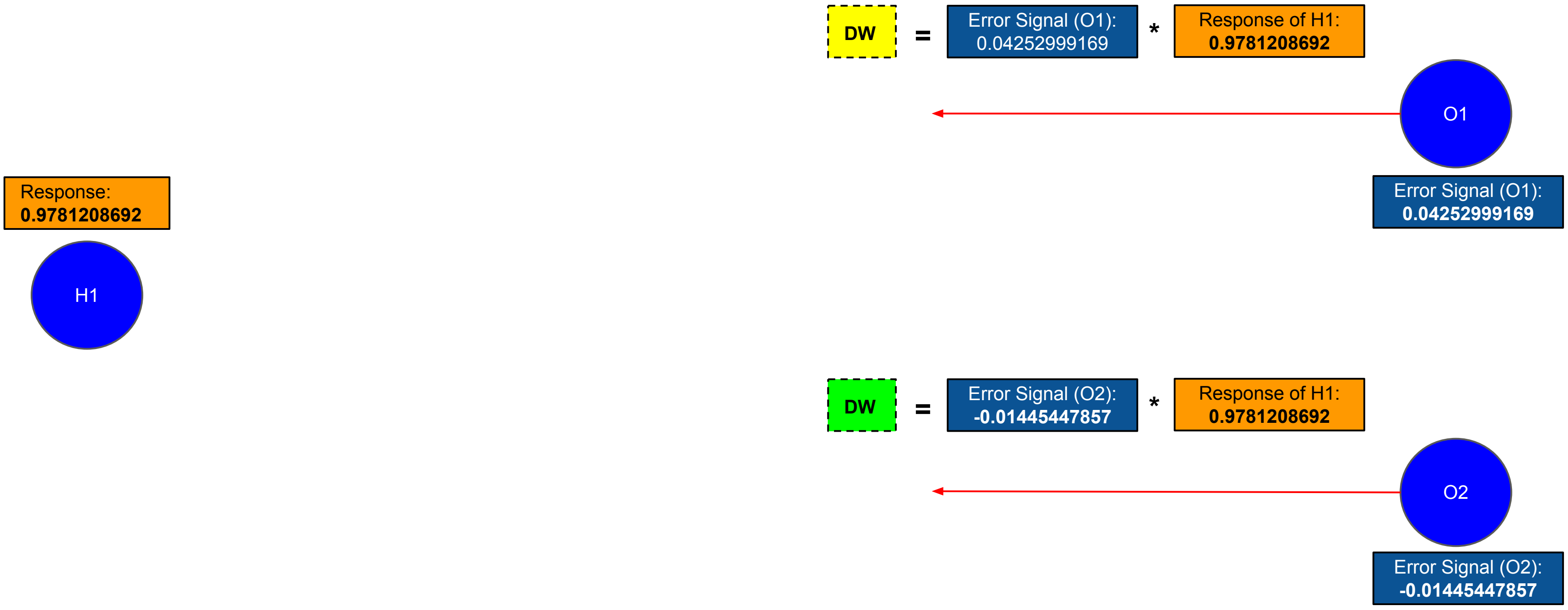
O1: **0.180266325** * **0.2359286555** = **0.04252999169**
(this is the error signal at neuron O1)

O2: **0.01467303175** * **-0.9851051105** = **-0.01445447857**
(this is the error signal at neuron O2)

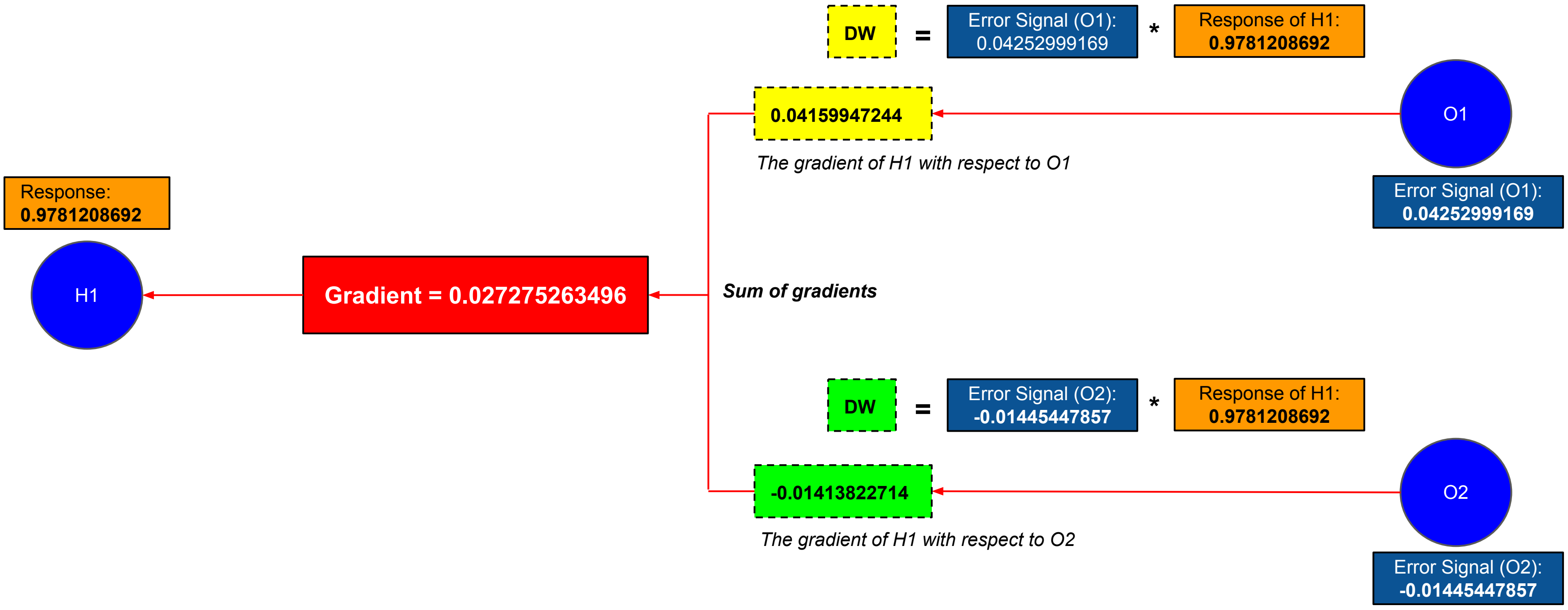
This is the error at the output layer of the network

Backpropagation

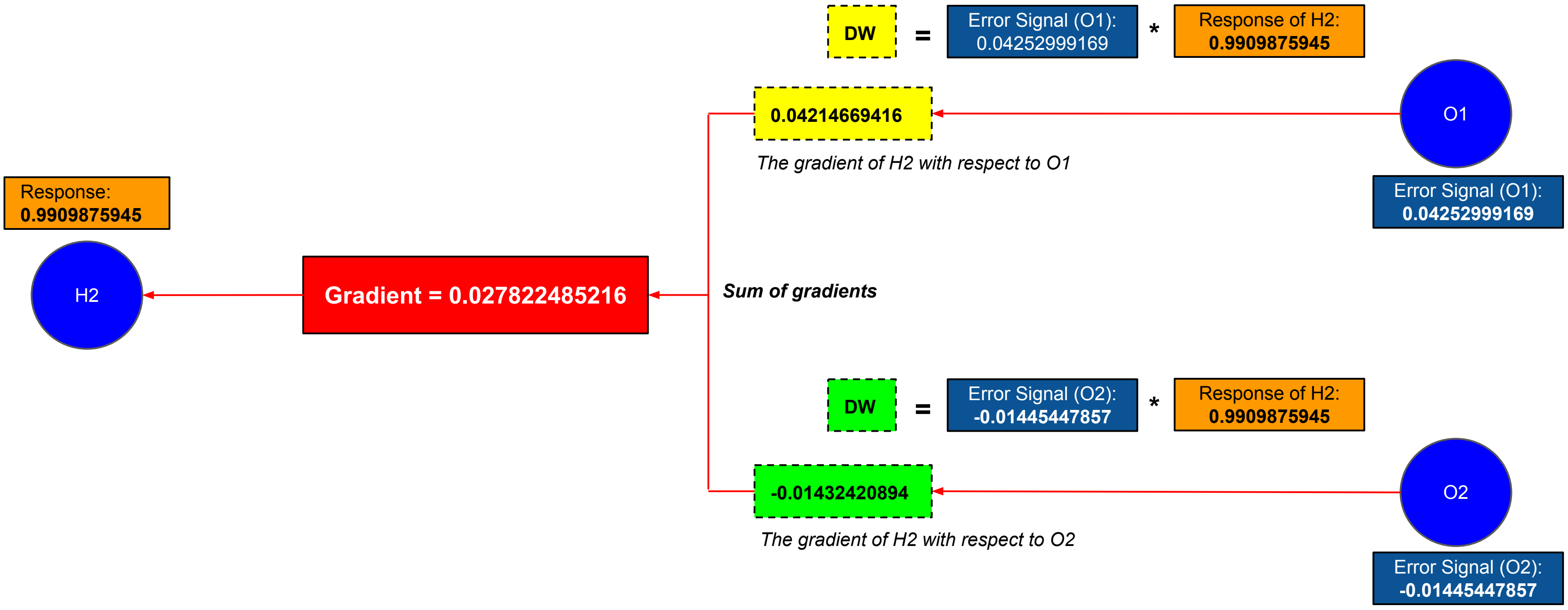
Step #2 - Compute the gradients of the Hidden Layer (H1 and H2)



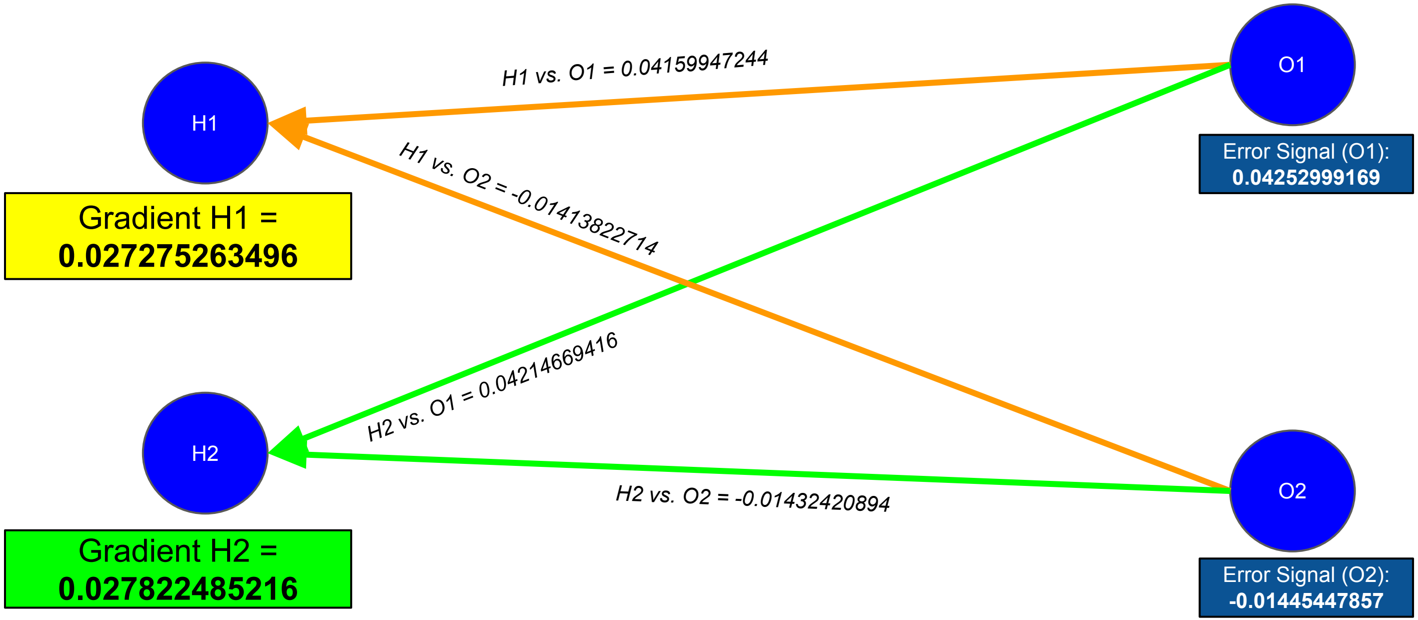
Step #2 - Compute the gradients of the Hidden Layer (H1 and H2)



Step #2 - Compute the gradients of the Hidden Layer (H1 and H2)



Summary - Compute the gradients of the Hidden Layer (H1 and H2)



Backpropagation

Summary - Computing the hidden gradients in a spreadsheet

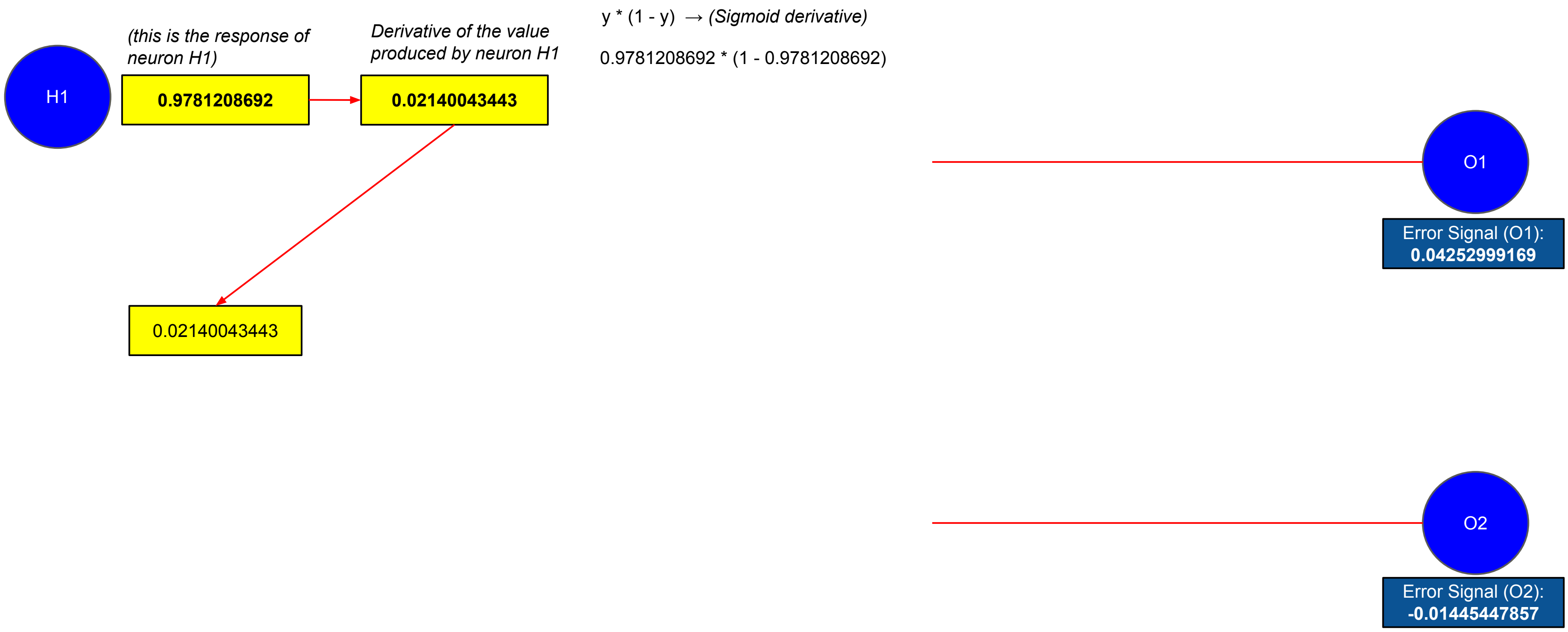
Error Signal at the Output Layer				
Neuron	Neuron Output	Feedback	Output Derivative	Error Signal at (O)
O1	0.7640713445	0.2359286555	0.180266325	**0.04252999169**
O2	0.9851051105	-0.9851051105	0.01467303175	**-0.01445447857**

Weight Gradients between O-H				
Upper Neuron	Lower Neuron	Error at (O)	Input from (H)	Weight Gradient (O-H)
O1	H1	0.04252999169	0.9781208692	0.04159947244
O1	H2	0.04252999169	0.9909875945	0.04214669416
O2	H1	-0.01445447857	0.9781208692	-0.01413822714
O2	H2	-0.01445447857	0.9909875945	-0.01432420894

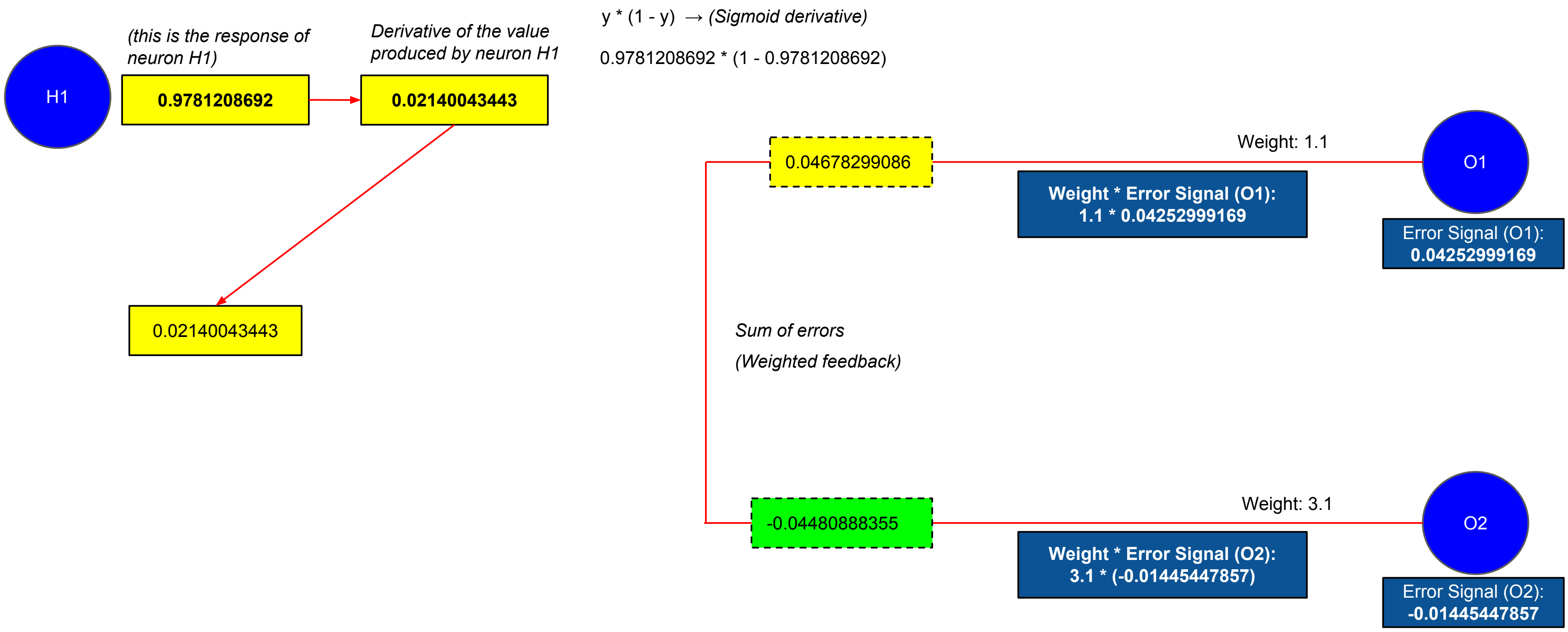
Hands on →

Backpropagation

Step #3 - Compute the error at the Hidden Layer

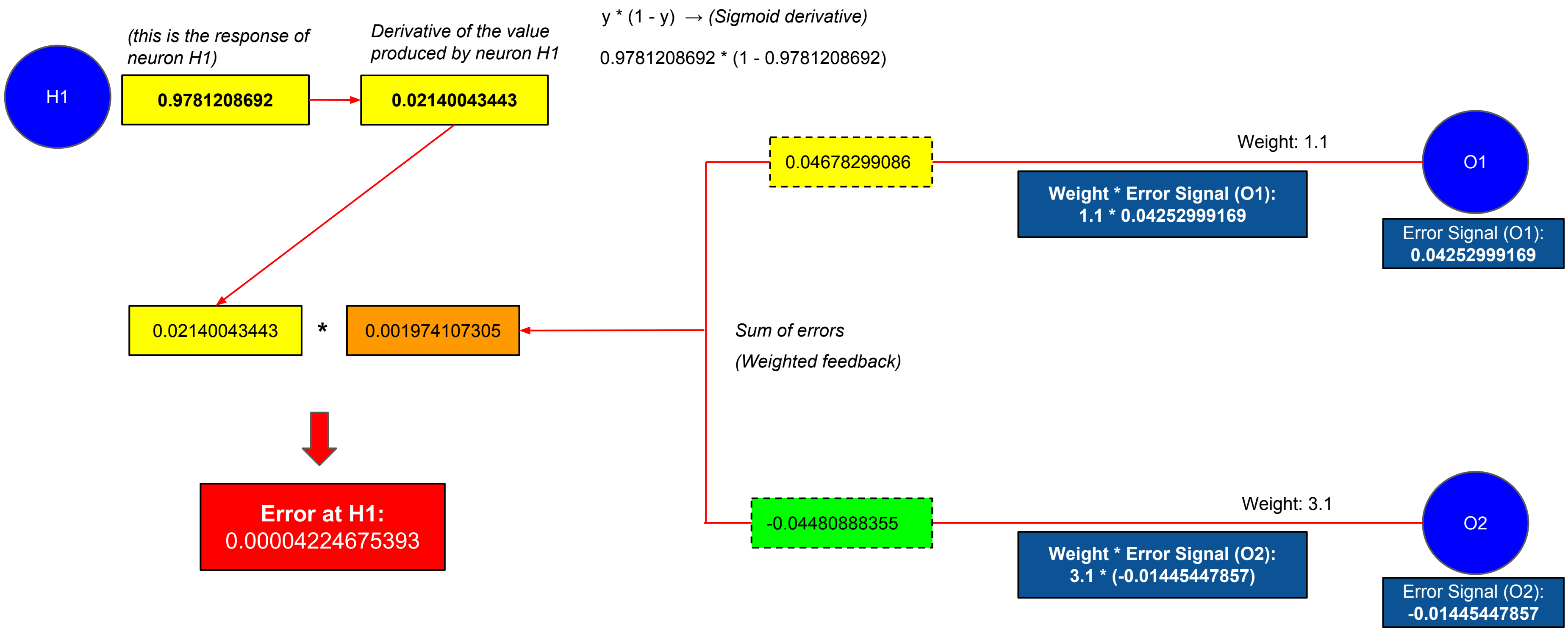


Step #3 - Compute the error at the Hidden Layer

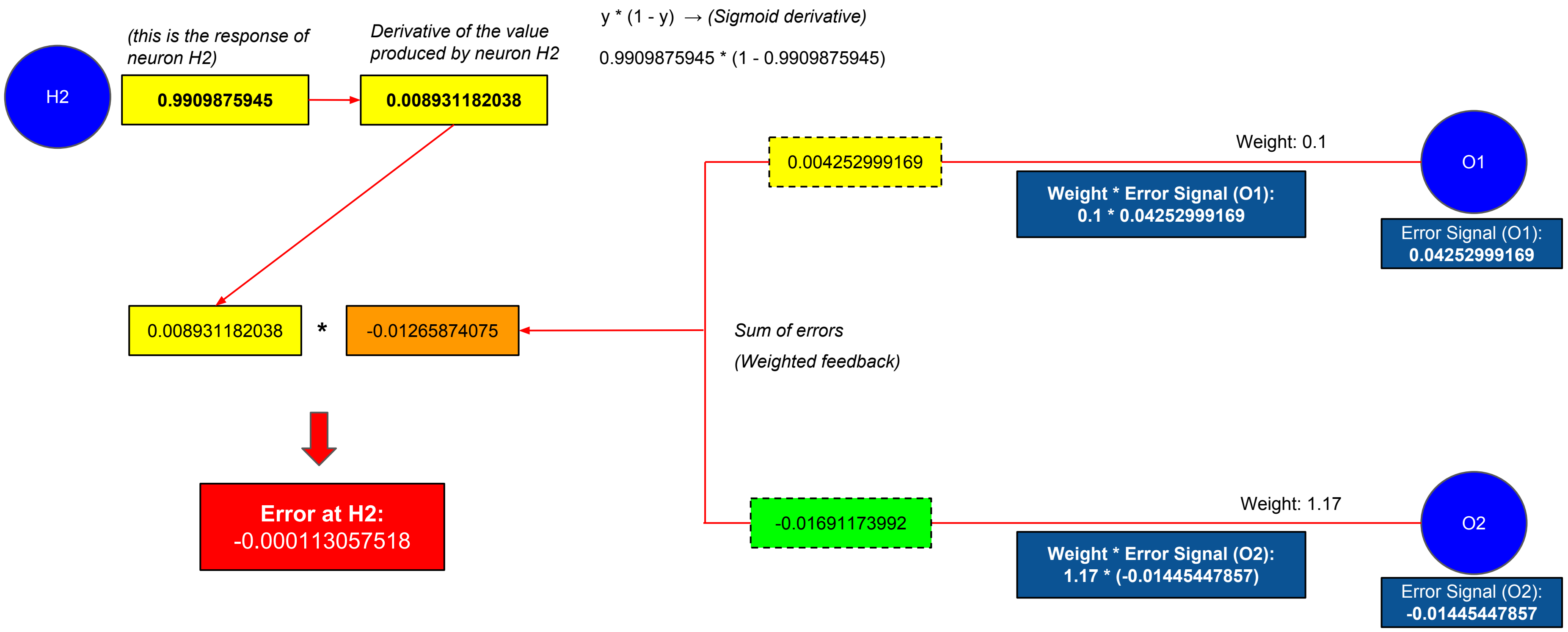


Backpropagation

Step #3 - Compute the error at the Hidden Layer

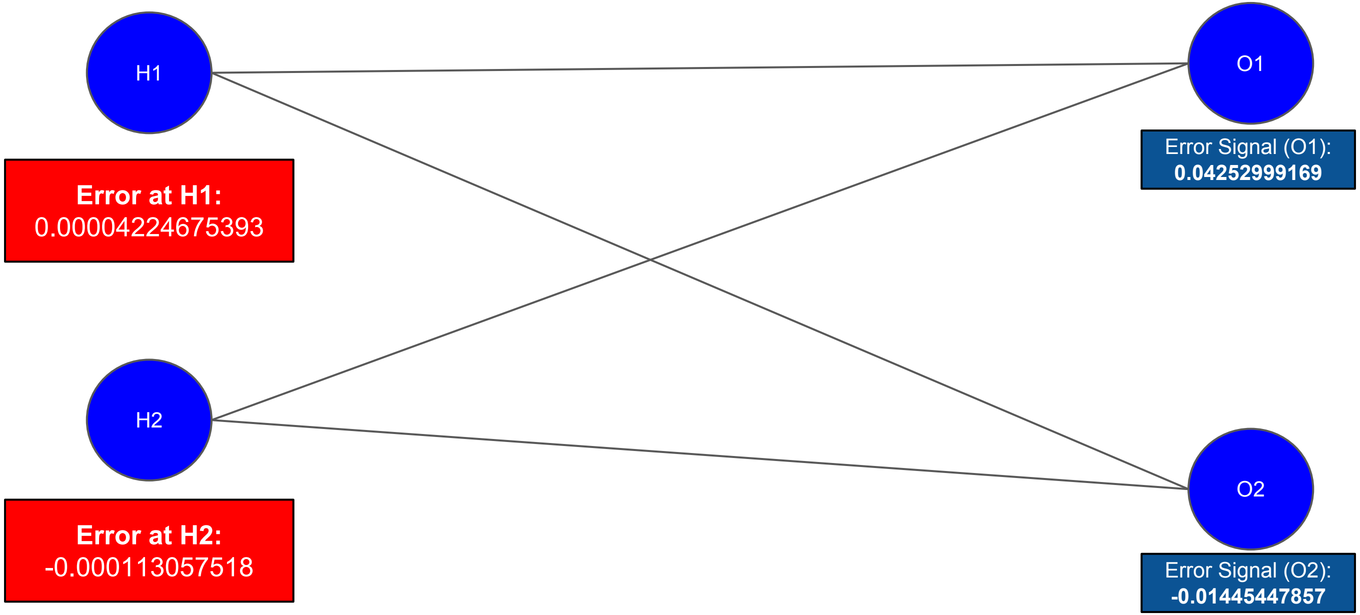


Step #3 - Compute the error at the Hidden Layer



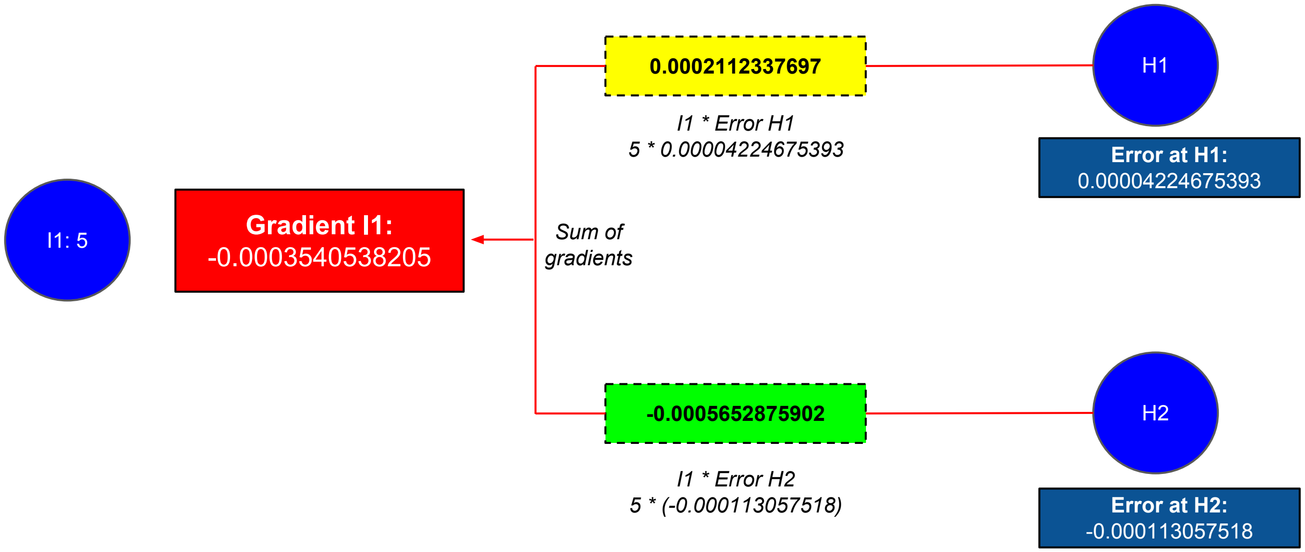
Backpropagation

Step #3 - Compute the error at the Hidden Layer



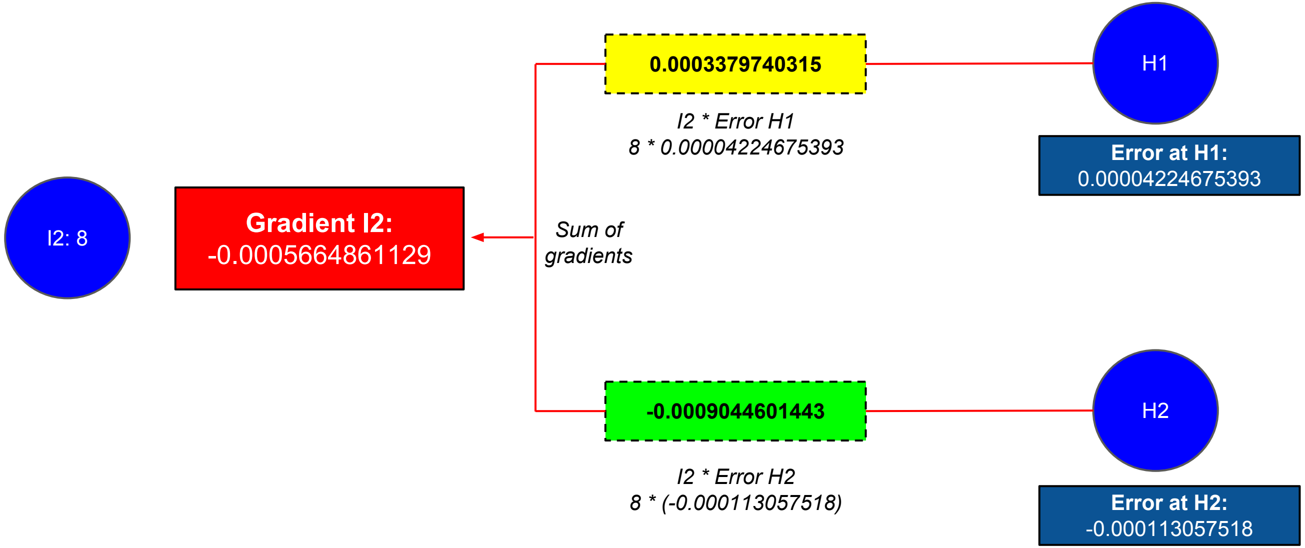
Backpropagation

Step #4 - Compute the gradients of the input layer (I1, I2, I3)



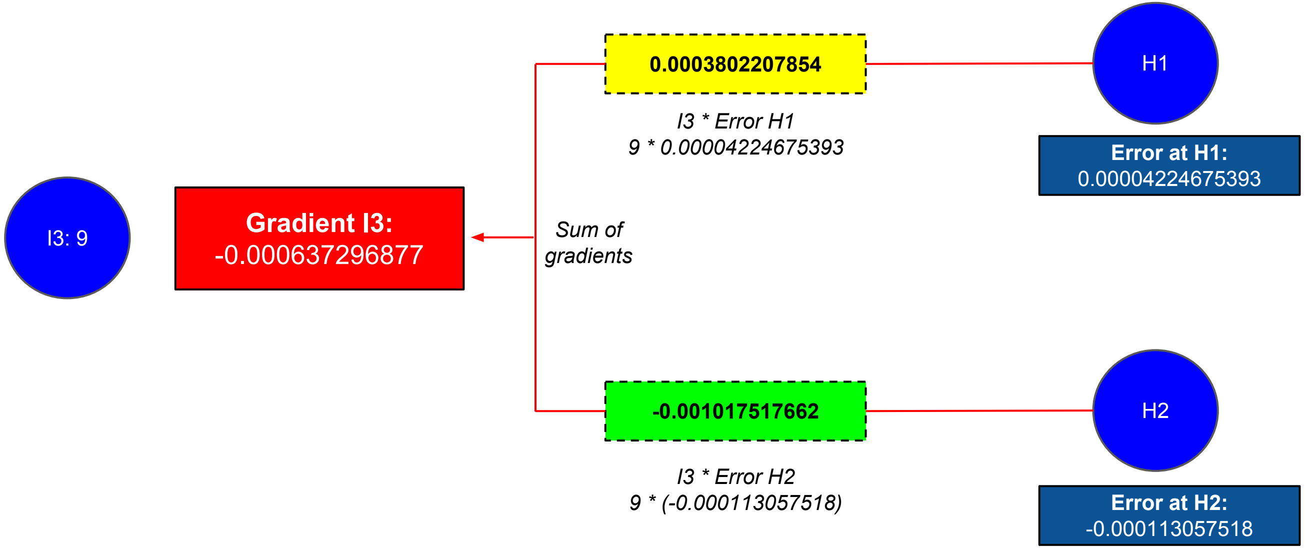
Backpropagation

Step #4 - Compute the gradients of the input layer (I1, I2, I3)



Backpropagation

Step #4 - Compute the gradients of the input layer (I1, I2, I3)



Step #4 - Compute the gradients of the input layer (I1, I2, I3)

Error Signal at the Hidden Layer				
Neuron	Neuron Output	Feedback	Output Derivative	Error Signal at (H)
H1	0.9781208692	0.001974107305	0.02140043443	**0.00004224675393**
H2	0.9909875945	-0.01265874075	0.008931182038	**-0.000113057518**

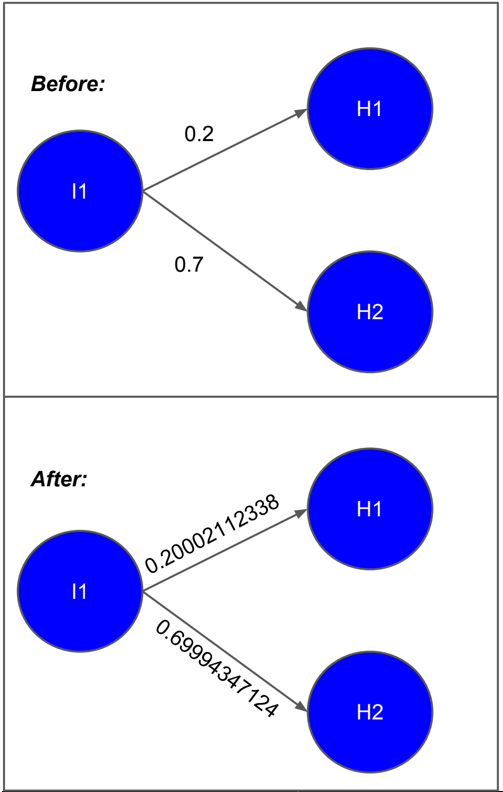
Weight Gradients between H-I				
Upper Neuron	Lower Neuron	Error at (H)	Input from (I)	Weight Gradient (H-I)
H1	I1	0.00004224675393	5	0.0002112337697
H1	I2	0.00004224675393	8	0.0003379740315
H1	I3	0.00004224675393	9	0.0003802207854
H2	I1	-0.000113057518	5	-0.0005652875902
H2	I2	-0.000113057518	8	-0.0009044601443
H2	I3	-0.000113057518	9	-0.001017517662

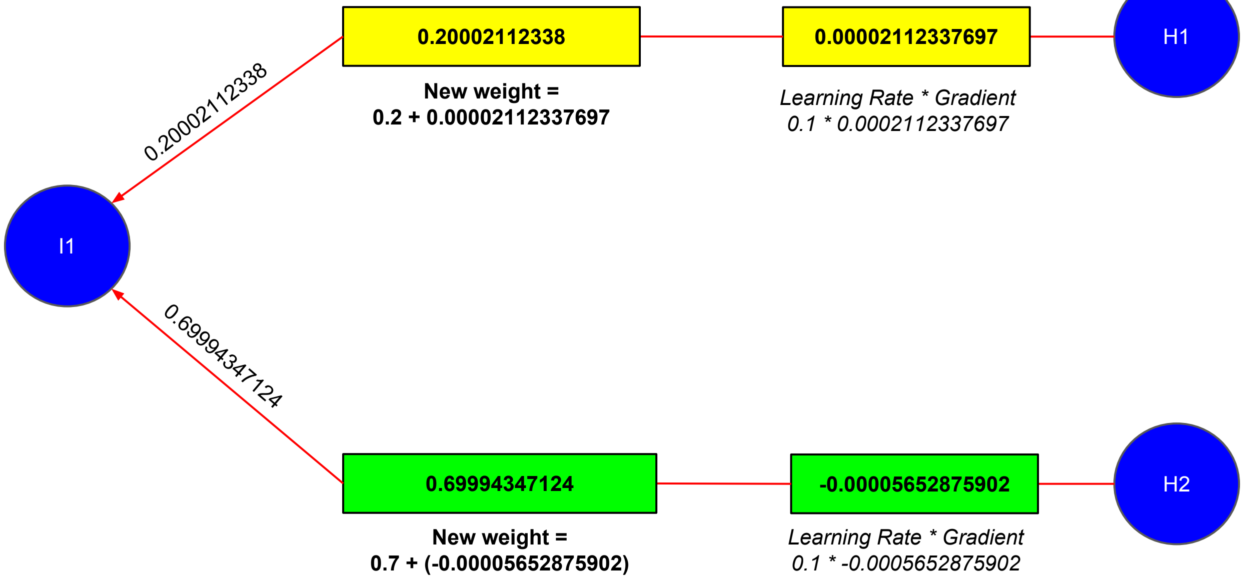
Backpropagation

Weights update

New weight = Current weight + (Gradient * Learning Rate)

Learning Rate: 0.1

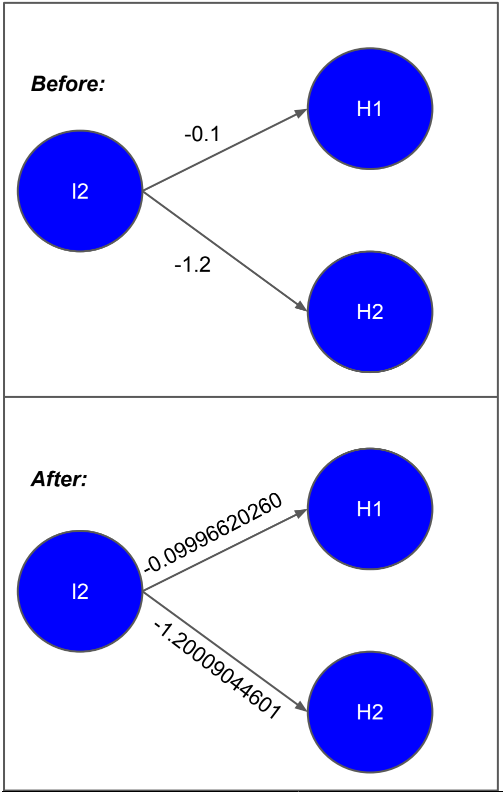


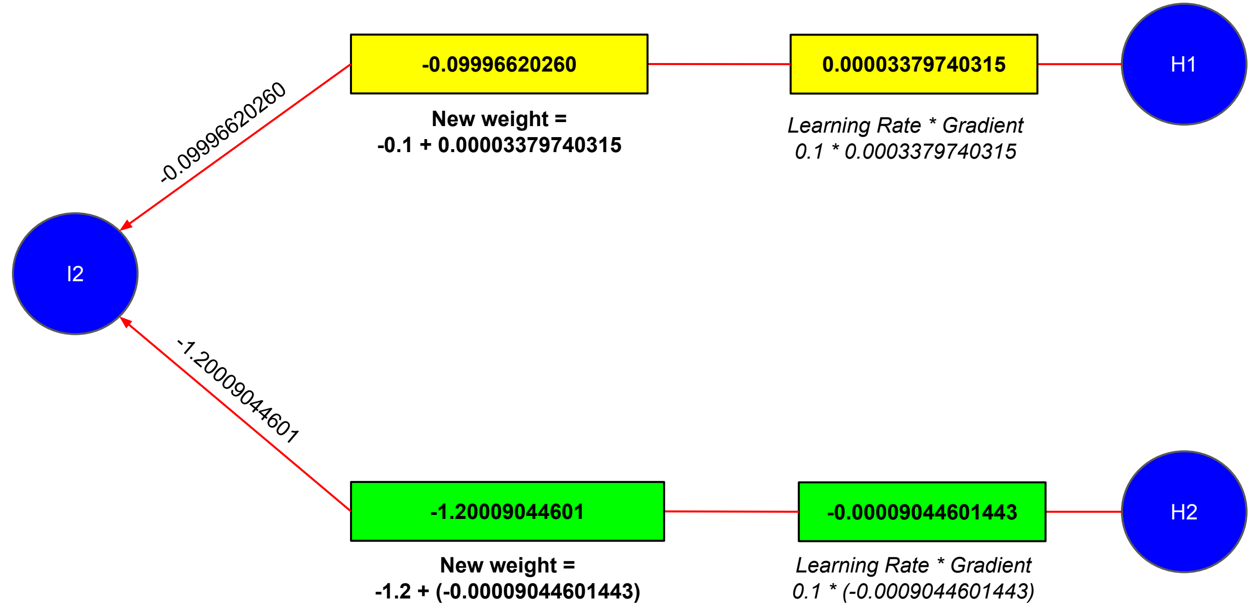


Weights update

New weight = Current weight + (Gradient * Learning Rate)

Learning Rate: 0.1





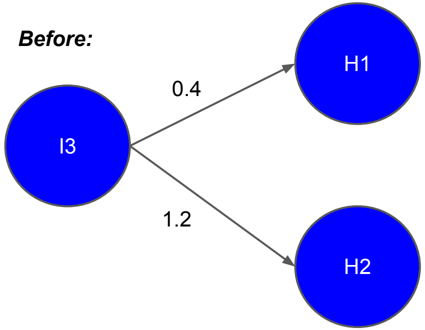
Backpropagation

Weights update

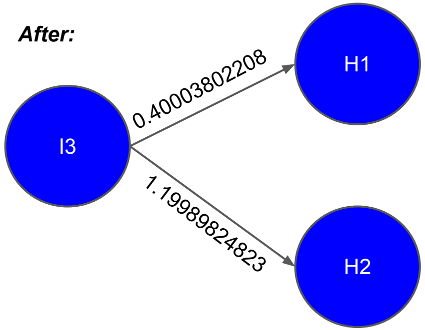
New weight = Current weight + (Gradient * Learning Rate)

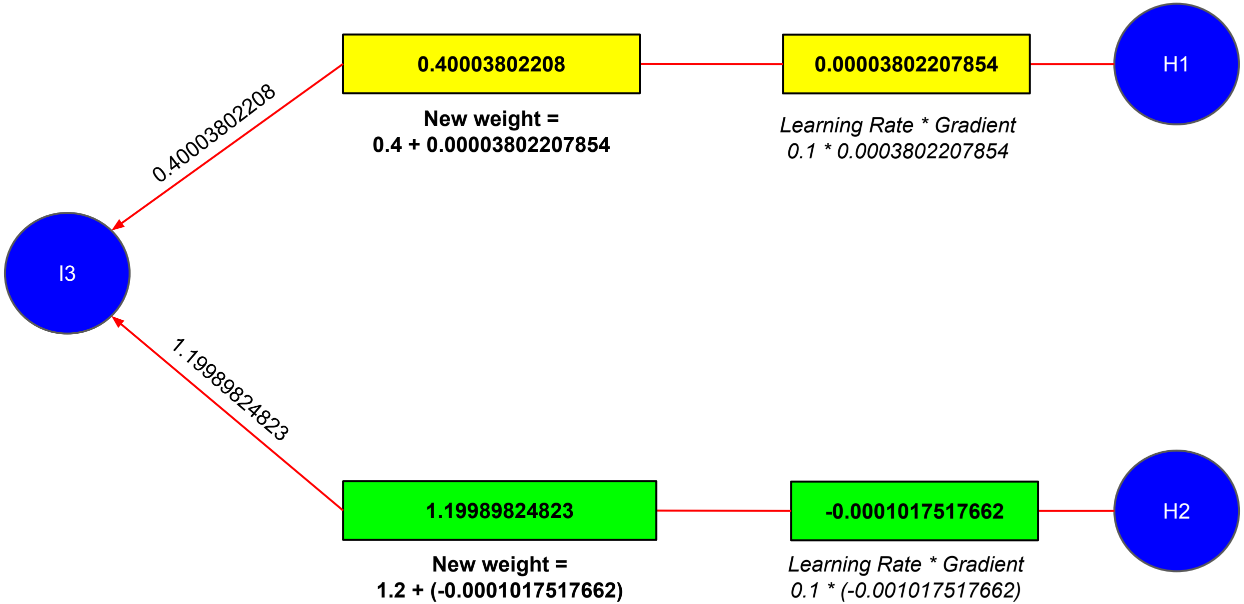
Learning Rate: 0.1

Before:



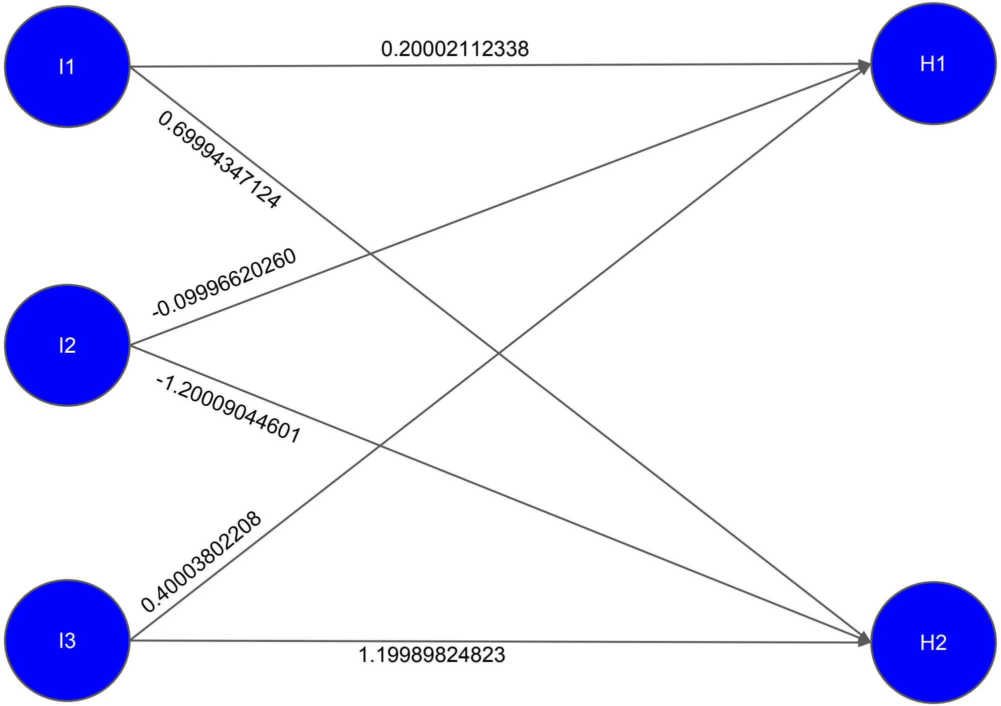
After:





Backpropagation

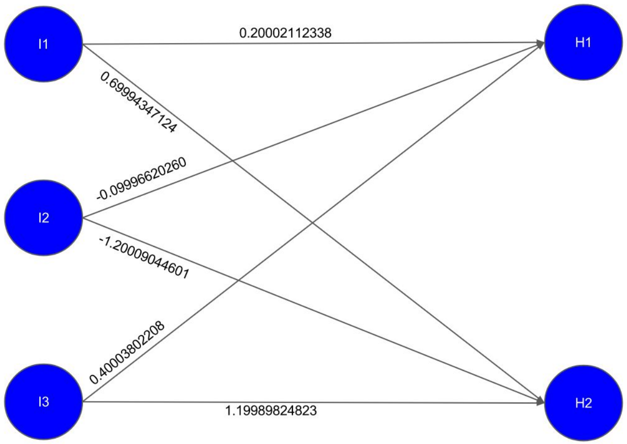
Weights update



Weights update

Weight updates (H-I)				
Upper Neuron	Lower Neuron	Old Weight	Weight Gradient (O-H)	Gradient x Learning Rate
H1	I1	0.2	0.0002112337697	0.00002112337697
H1	I2	-0.1	0.0003379740315	0.00003379740315
H1	I3	0.4	0.0003802207854	0.00003802207854
H2	I1	0.7	-0.0005652875902	-0.00005652875902
H2	I2	-1.2	-0.0009044601443	-0.00009044601443
H2	I3	1.2	-0.001017517662	-0.0001017517662

New Weights, with Momentum (H-I)		
Previous Delta	Delta Momentum	New Weight
0	0	**0.20002112338**
0	0	**-0.09996620260**
0	0	**0.40003802208**
0	0	**0.69994347124**
0	0	**-1.20009044601**
0	0	**1.19989824823**

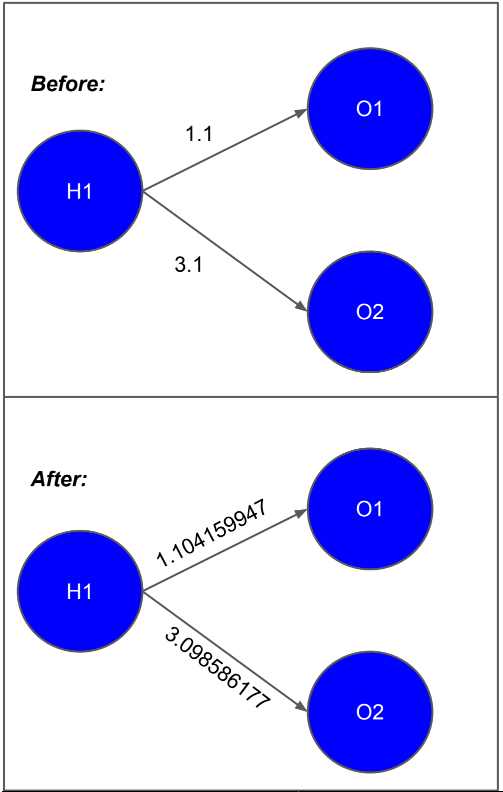


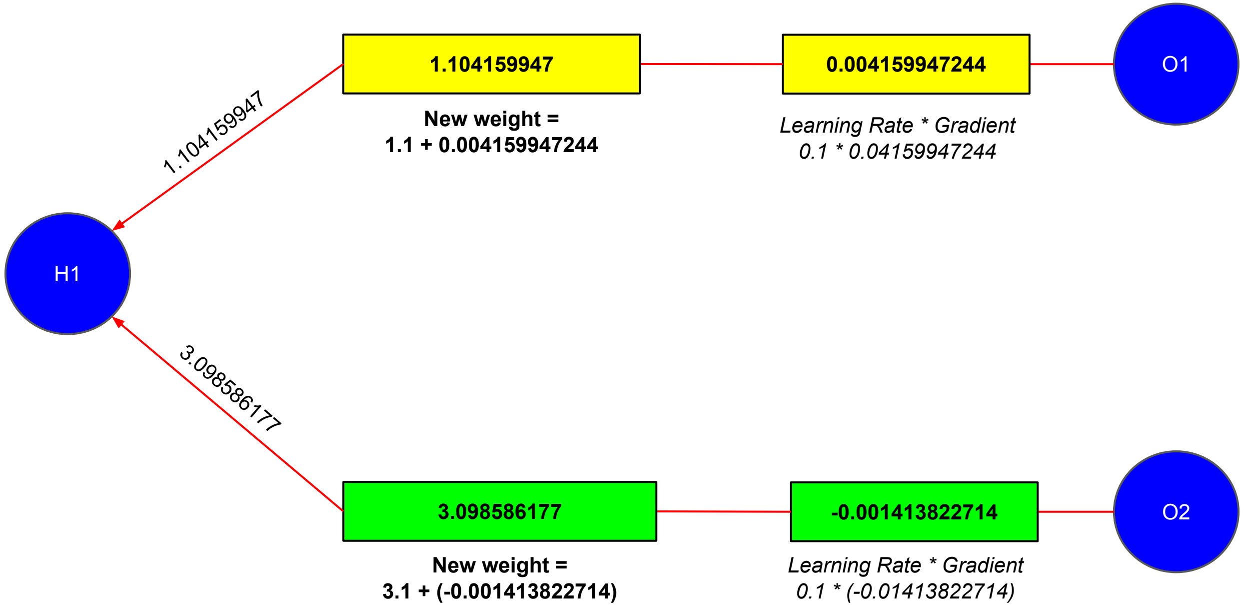
Backpropagation

Weights update

New weight = Current weight + (Gradient * Learning Rate)

Learning Rate: 0.1

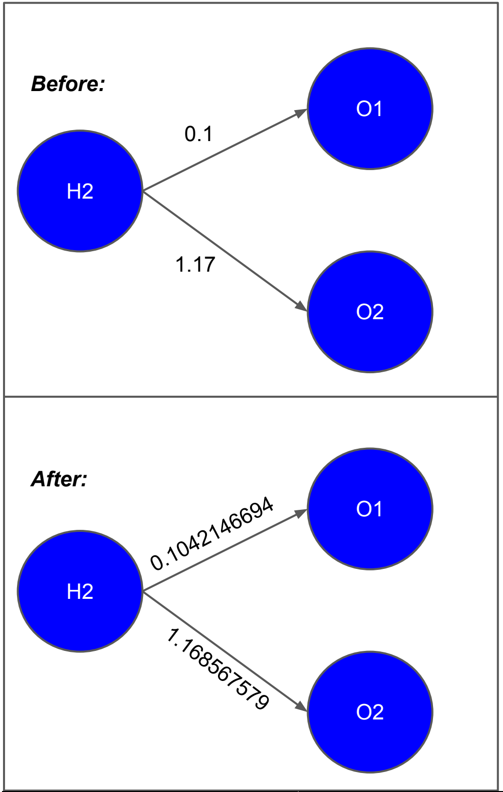


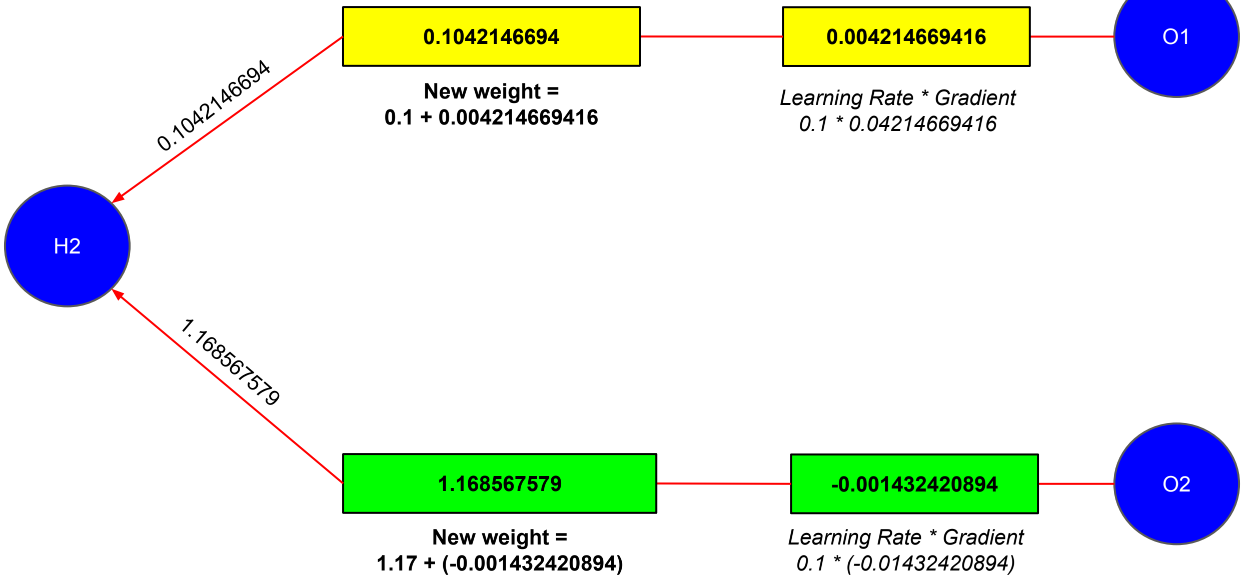


Weights update

New weight = Current weight + (Gradient * Learning Rate)

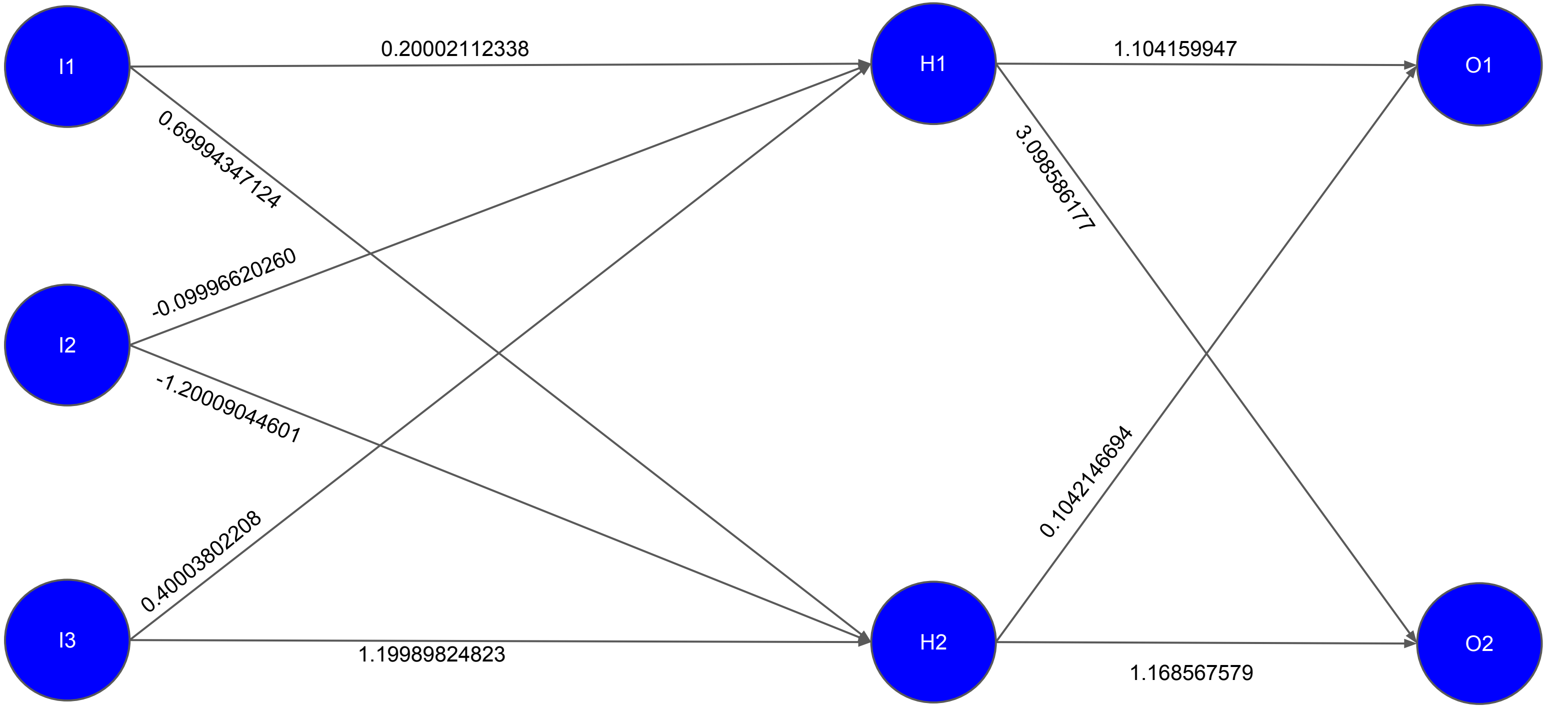
Learning Rate: 0.1





Backpropagation

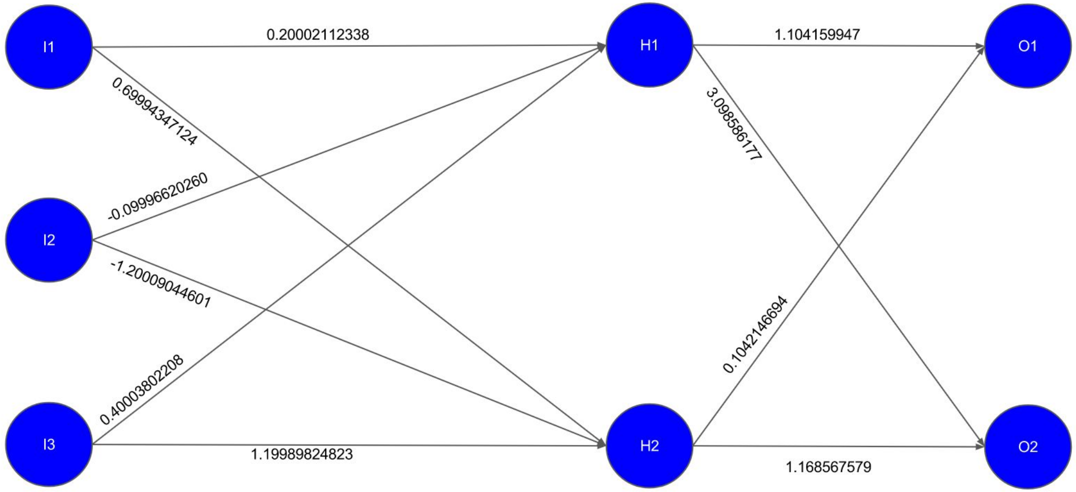
Weights update



Weights update

Weight updates (O-H)				
Upper Neuron	Lower Neuron	Old Weight	Weight Gradient (O-H)	Gradient x Learning Rate
O1	H1	1.1	0.04159947244	0.004159947244
O1	H2	0.1	0.04214669416	0.004214669416
O2	H1	3.1	-0.01413822714	-0.001413822714
O2	H2	1.17	-0.01432420894	-0.001432420894

New Weights, with Momentum (O-H)		
Previous Delta	Delta Momentum	New Weight
0	0	**1.104159947**
0	0	**0.1042146694**
0	0	**3.098586177**
0	0	**1.168567579**



Checkpoint

Achievements.

Now you know how a Neural Network learns from mistakes.

- **Trial** → It makes a prediction, to the best of their ability. This is called *“forward pass”*.
- **Error** → It measures the error.
- **Learn** → It adjusts their weights accordingly, in order to perform better next time.
  This is called backpropagation.

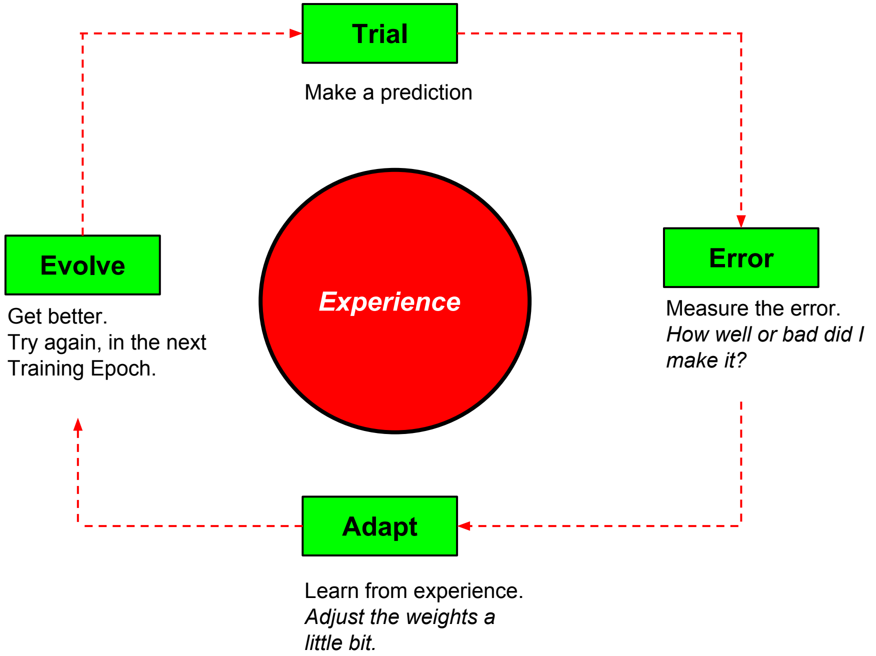
Experience *is what you get when things don't go as expected.*

Training a Neural Network

The mechanics.
Networks learn through back propagation
(trial → error → ***adapt****)*

The algorithms.
Networks can be trained through a wide variety of algorithms:

- Stochastic gradient descent (*aka: SGD*)
- Genetic Algorithms
- *etc*



Training a Neural Network

Training Algorithm
- SGD
- Genetics
- Others

Stochastic gradient descent (SGD)
- Online
- Batch

Data
It could be anything:
- Pictures
- Songs
- Videos
- Criminal records
- Global Temperatures
- Stock Prices
- Industry ratios
- Custom database records
- etc

Training a Neural Network

Stochastic gradient descent (SGD)

- **Online**
  - Better accuracy rate
  - Not suitable for large datasets
  - Training takes more time

- **Batch**
  - Faster training, because you update the weights only once.
  - The network accuracy is worst compared with the Online method.
  - Suitable for large datasets

Training a Neural Network

Dataset

Training set	Testing set	Validation set

Training set.
The samples we will use to to adjust the weights of the network.

This is what the network will learn.

Testing set.
The samples we will use to assess the performance of the network.

The network has never seen these examples before.

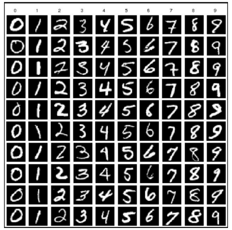
Validation set.
The samples we will use to fine tune the hyperparameters of the network.

For example:

- Number of hidden units
- Learning Rate
- Momentum

Also used to prevent *over-fitting*.

Example.
MNIST dataset
(digits recognition)



Stochastic gradient descent *(online)*

Procedure *(pseudo-code)*

1.3) For each training sample:

- **Trial** → Present the training sample to the network and get the network response.
- **Error** → Learn from experience. Measure the error and backpropagate the gradients.
- **Learn** → Update the network parameters.

Stochastic gradient descent *(online)*

Procedure ***(pseudo-code)***

1) For each Training Epoch *(also called "iteration")*

1.3) For each training sample:

- **Trial** → Present the training sample to the network and get the network response.
- **Error** → Learn from experience. Measure the error and backpropagate the gradients.
- **Learn** → Update the network parameters.

Stochastic gradient descent *(online)*

Procedure ***(pseudo-code)***

1) For each Training Epoch *(also called "iteration")*

1.2) Reorder your training set, in a random manner.

- In online learning the training samples need to be presented to the network in random order. This is a key factor.

1.3) For each training sample:

Trial → Present the training sample to the network and get the network response.

Error → Learn from experience. Measure the error and backpropagate the gradients.

Learn → Update the network parameters.

Stochastic gradient descent *(online)*

Procedure ***(pseudo-code)***

1) For each Training Epoch *(also called "iteration")*

→ 1.1) Measure the *Network Performance*, to see how the network is doing.
- If the network achieved the desired accuracy then stop training.
- Otherwise continue with the training process.

→ 1.2) Reorder your training set, in a random manner.
- In online learning the training samples need to be presented to the network in random order. This is a key factor.

→ 1.3) For each training sample:

→ **Trial** → Present the training sample to the network and get the network response.

→ **Error** → Learn from experience. Measure the error and backpropagate the gradients.

→ **Learn** → Update the network parameters.

Stochastic gradient descent *(online)*

```
// Run all epochs...
for (int trainingEpoch = 0; trainingEpoch < maxTrainingEpochs; trainingEpoch++)
{
    Measure the Network Performance

    // ---------------------------------------------------------------------------
    // Learning
    // ---------------------------------------------------------------------------
    // Generate a random training sequence in order to mitigate over-fitting,
    // local-minima and other well-known side effects.
    int[] trainingSequence = GenerateRandomSequence(trainingSet);

    for (int i = 0; i < totalSamples; i++)
    {
        // Pick a training sample, randomly
        int randomSampleIndex = trainingSequence[i];
        TrainingSample trainingSample = trainingSet.Samples[randomSampleIndex];

        // Learn about this training sample in particular.
        // Adjust the network weights accordingly.
        LearnTrainingSample(trainingSample.Sample, trainingSample.ExpectedOutput, networkTrainingParameters);
    }
}
```

Source code is provided for educational purposes only.

In plain English:

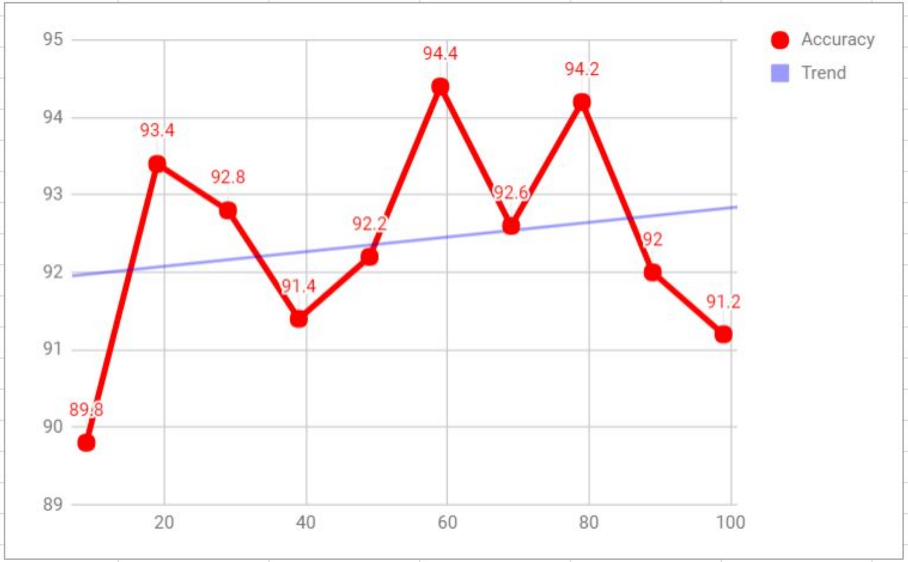
- We learn about every training sample over a set of training epochs.
- At the beginning of each training epoch, we reorder the training samples randomly. This is crucial for online-SGD to succeed.

You could speed up the training process by learning multiple training samples at the same time.
Tips:

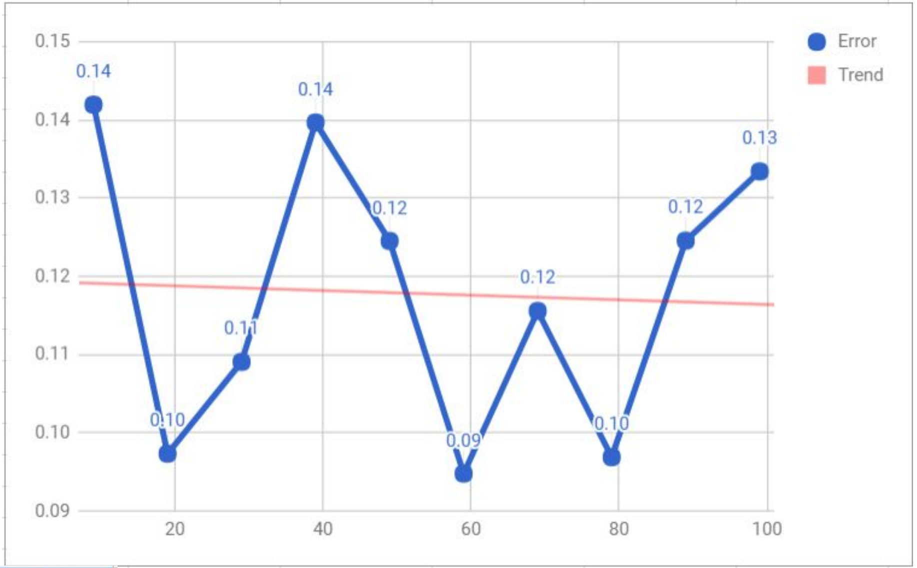
- Parallelization
- Multi-Threading

Training a Neural Network

Network Accuracy



Network Error

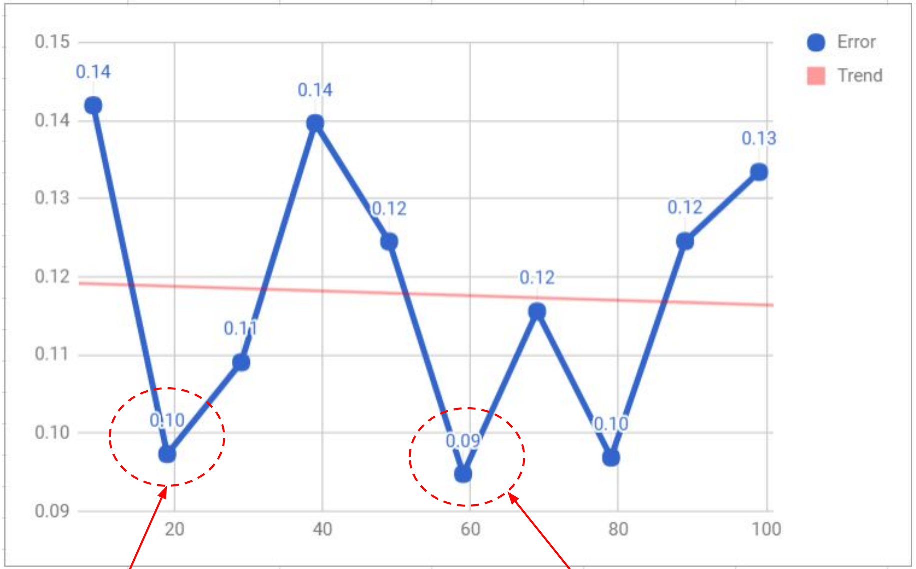


Epoch	Error	Accuracy
9	0.14	89.8
19	0.10	93.4
29	0.11	92.8
39	0.14	91.4
49	0.12	92.2
59	0.09	94.4
69	0.12	92.6
79	0.10	94.2
89	0.12	92
99	0.13	91.2

Epochs: 100
Number of Training Samples: 500
Number of Testing Samples: 2500
Learning rate: 0.1
Momentum: 0.5

Best Training Epoch: 59
Epoch Accuracy: 94.4%

Local Minima



Local minima

Not the best solution

Global minima

Best solution

Happens when you get stuck in a valley
Descending too fast or too slow is a problem.

Fine tune your hyperparameters.
Try with different learning rates and momentum.
Also check your network topology.

Momentum helps the network to overcome obstacles (local minima) in the error surface and settle down at or near the global minimum.

Repeated training.
A widely used approach is to train the network more than once, starting with a random set of weights. Use different partitions for training/test/validation.
This is known as cross-validation.
Downside: it takes more time to train networks.

You nailed it!

Thank you

Ready to level up?

Machine Learning datasets.

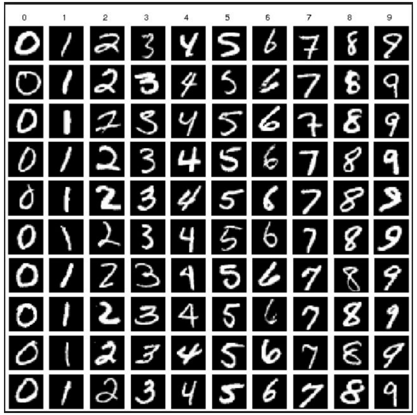
The links below contain Machine Learning datasets:

- https://en.wikipedia.org/wiki/List_of_datasets_for_machine_learning_research
- https://archive.ics.uci.edu/ml/datasets.html

Homework:

- Pick any of the datasets of your preference.
- Create a neural network.
- Train it.

Example.
MNIST dataset
(digits recognition)



Link:
http://yann.lecun.com/exdb/mnist/