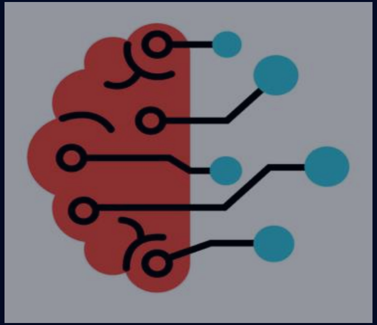
DATA, MSML, BIOI 602

Principles of Data Science

Artificial Neural Networks



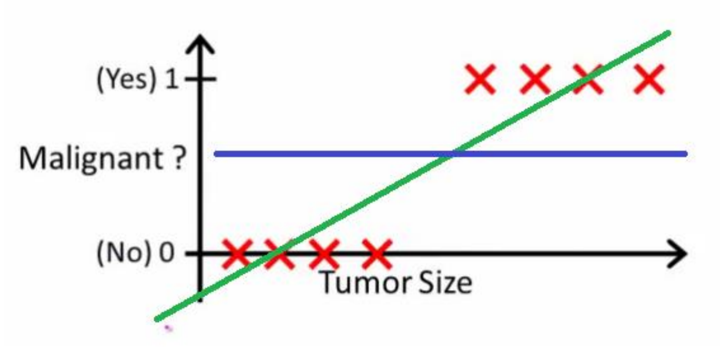
[Slides from Fardina Fathmiul Alam]

Before that, revisit “Logistic Regression” quickly

Logistic Regression

Logistic Regression is a modification of linear regression to deal with binary categories or binary outcomes.

- Similar to Linear Regression but is specifically used when the dependent variable is categorical.
  - Predicts a categorical dependent variable using numeric or categorical independent variables.
  - It is based on the concept of probability.
  - Mainly used for binary classification
    - Ex. Tumor Malignant or Benign? Given a person's height, what is their gender (M or F)? Is this email spam or not? etc.

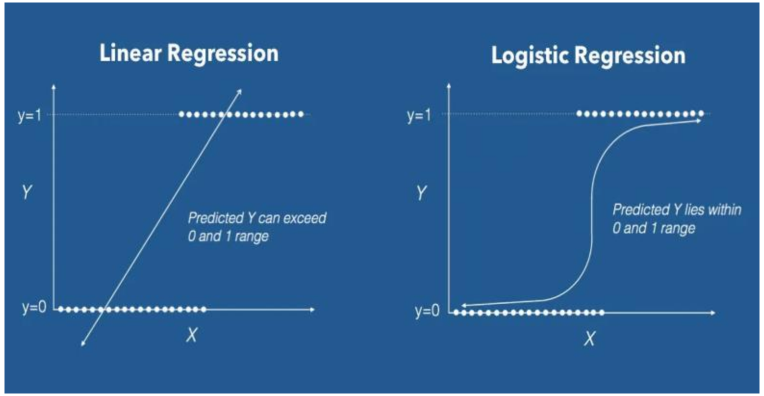


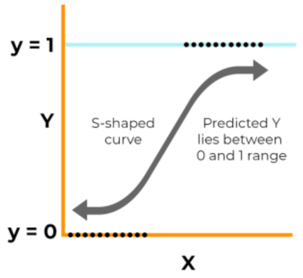
Logistic Regression : Output

For Input Feature X,

Output: Probability Y between 0 and 1 (instead of output 0 or 1).

- 0 ≤ Y ≤ 1, where Y is the probability that the output label is 1 given the input X.
- **How?** Logistic regression uses the logistic sigmoid function to transform its output into a probability value.
    - Create an S-Curve: Converts linear outputs into probabilities.


Linear Regression
y=1
Y
y=0
X
Predicted Y can exceed 0 and 1 range
Logistic Regression
y=1
Y
y=0
X
Predicted Y lies within 0 and 1 range



Logistic Function - Sigmoid

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

- **Given, the linear combination** **z** **of the input features:**

$$\mathbf{z} = \beta_0 + \beta_1\ x_1 + \beta_2 x_2 + \ldots\ldots\ldots\ldots\ldots + \beta_n x_n$$

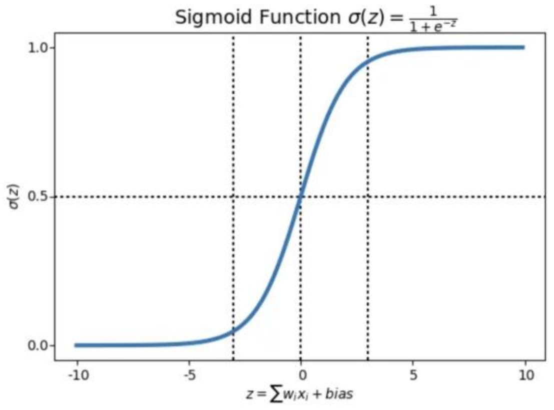
Where, βo= intercept term (bias), β1,β2,…,βn: Coefficients for each input feature, and x1,x2,…,xn: Input features

- **Transformation Using Sigmoid Function:** Apply the logistic sigmoid function to the linear combination z to get the probability Y

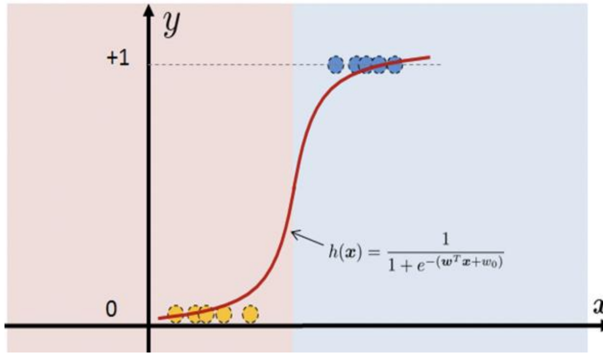
$$Y = \text{sigmoid}(z) = \frac{1}{1+e^{-z}}$$

$$\text{sigmoid}(z) = \frac{1}{1+e^{-(\beta_1 x_1+\beta_2 x_2+\ldots+\beta_n x_n+b)}}$$

Where, e: Euler's number (approximately 2.718)



Sigmoid Function Graph

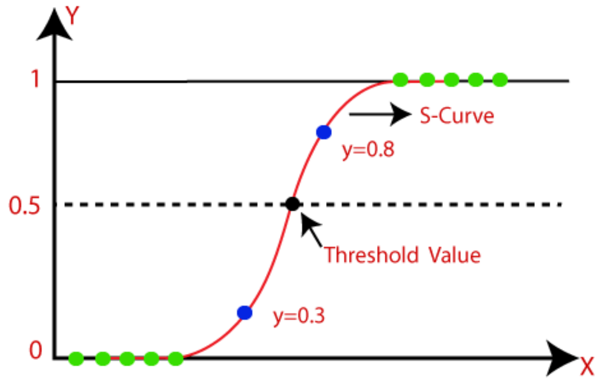


Logistic Regression: Training and Threshold for Binary Classification

Logistic regression is trained using gradient descent to minimize the logistic loss function (error between the predicted probabilities and the actual class labels in the training data.) by to find the optimal values of the coefficients.

A common threshold is 0.5, but it can be adjusted based on the problem. Rule:

- If $Y \geq 0.5$, predict class 1.
- If $Y < 0.5$, predict class 0.



Remember:

❑ Higher Threshold: Increases precision (reduces false positives).
❑ Lower Threshold: Increases recall (reduces false negatives).

Logistic Function - Sigmoid

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

SIGMOID FUNCTION

If , **Z** is very large σ(z) = 1/1+0 = **1**

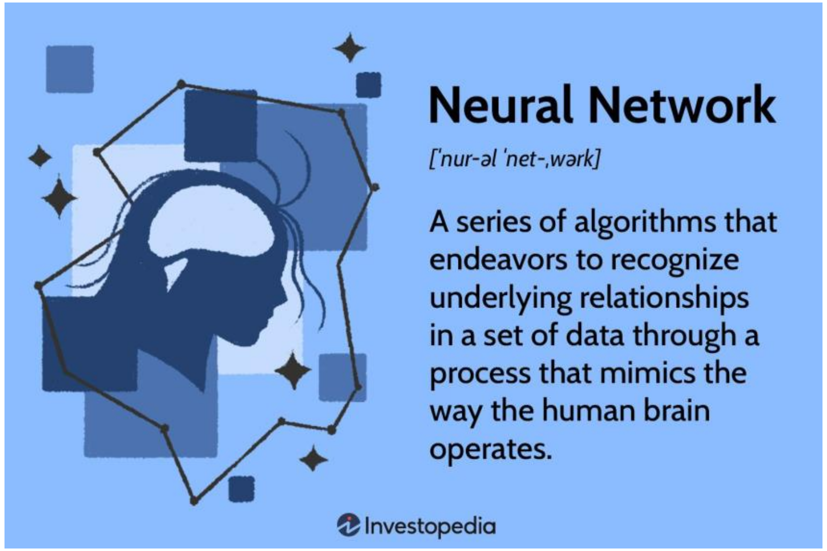
If, **Z** is very small (large negative number) σ(z) = 1/1+big number = **0**

$$Y = \sigma\overbrace{(w^T X + B)}^{Z}$$

$$Y = \sigma(Z)$$

It turns out that logistic regression can be viewed as a very very small neural network.

Artificial Neural Network



Neural Network

['nur-əl 'net-,wərk]

A series of algorithms that endeavors to recognize underlying relationships in a set of data through a process that mimics the way the human brain operates.

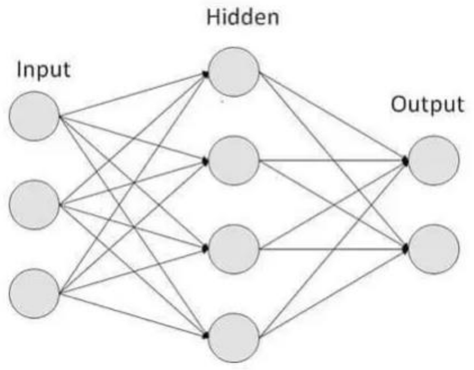
Investopedia

Deep Learning and Neural Network

Deep learning is the field of artificial intelligence (AI) that teaches computers to process data in a way **inspired by the human brain**.

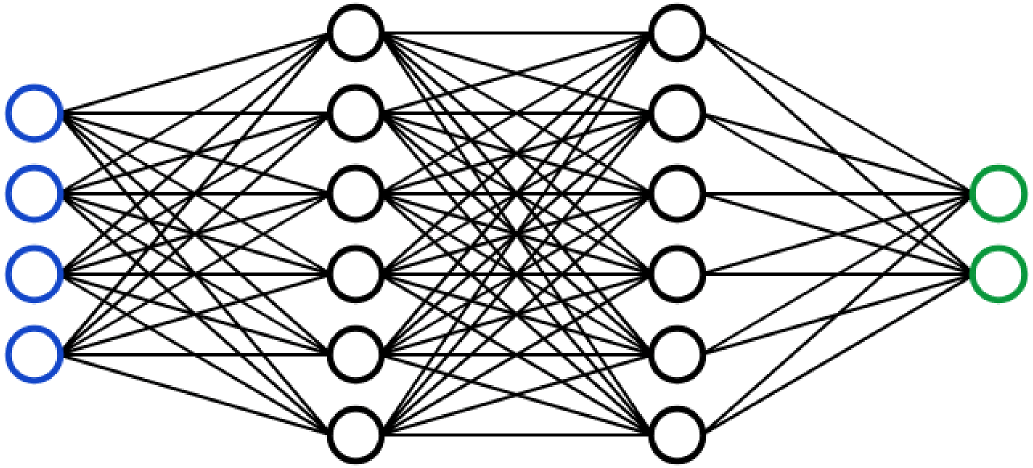
- Deep learning models can recognize data patterns like complex pictures, text, and sounds to produce accurate insights and predictions.

A neural network is the underlying technology in deep learning.



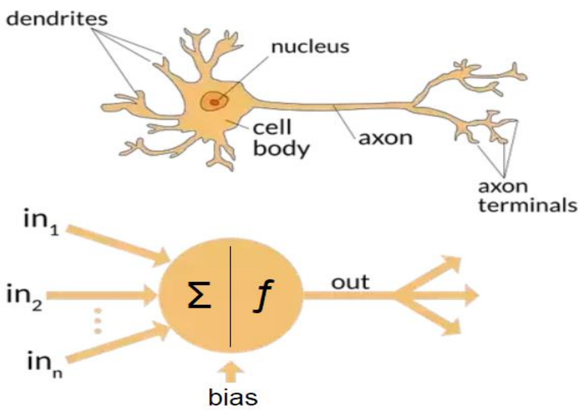
What is Neural Network (NN)

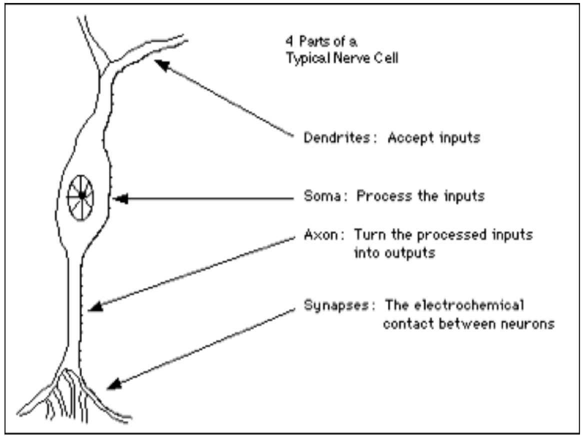
A neural network is a computational model in artificial intelligence inspired by the structure and functioning of the human brain.



What is Neural Network (NN)

A neural network is a deep learning model that uses layer(s) of interconnected nodes or neurons that resembles the human brain.




4 Parts of a Typical Nerve Cell
Dendrites: Accept inputs
Soma: Process the inputs
Axon: Turn the processed inputs into outputs
Synapses: The electrochemical contact between neurons

What is Neural Network (NN)

It processes input data to make predictions or decisions by learning from the data, solving complex problems like document summarization, pattern recognition, and facial recognition with greater accuracy.

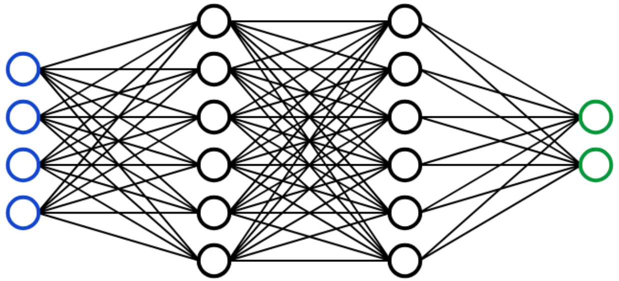
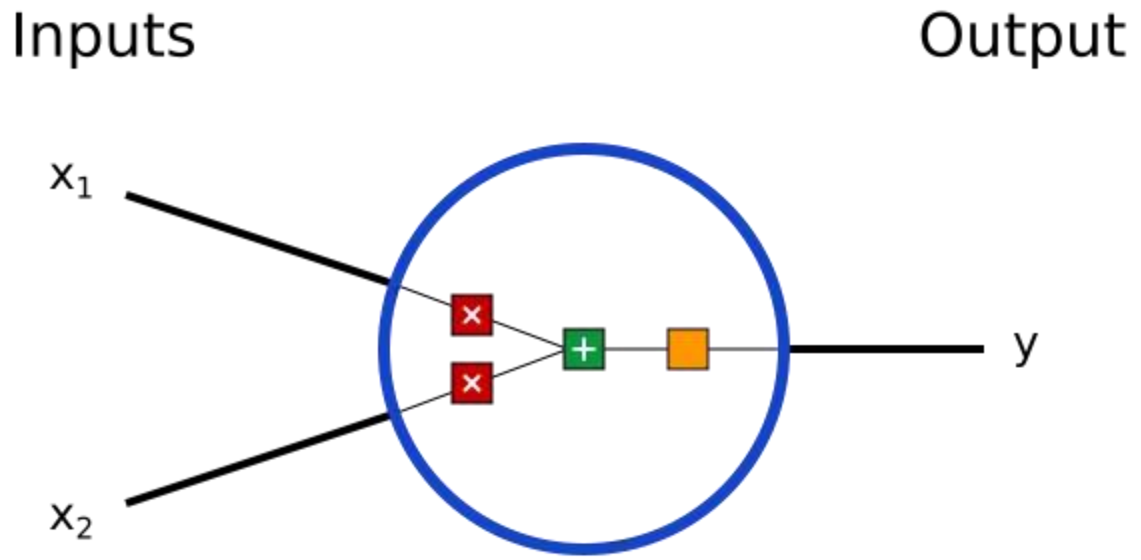


Fig: Neural networks consist of interconnected nodes, known as neurons, arranged in layers.

Building Blocks: Neurons

The basic unit of a neural network:



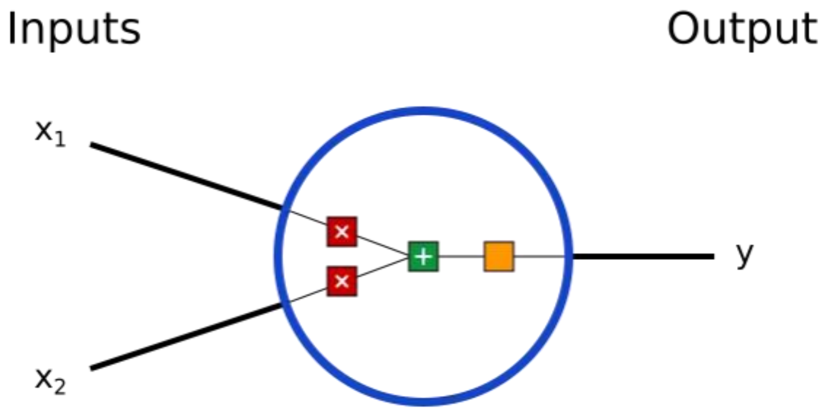
Neuron takes <u>multiple input signals</u>, <u>processes</u> them, and <u>produces an output</u>.

Building Blocks: Neurons

First, each input is multiplied by a weight

$$x_1 \rightarrow x_1 * w_1$$

$$x_2 \rightarrow x_2 * w_2$$



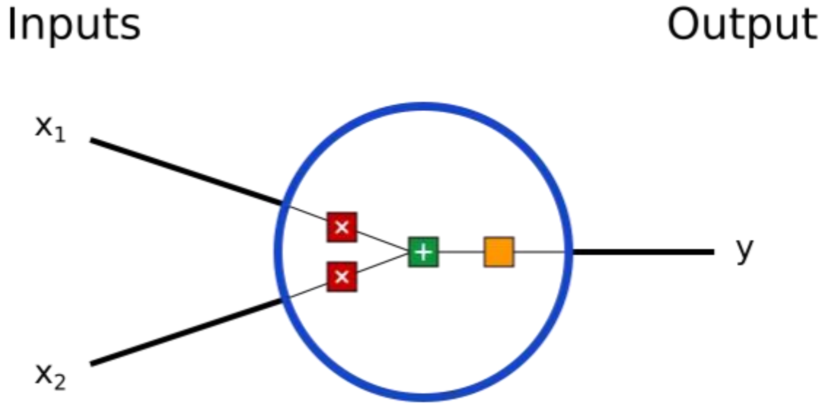
Next, all the weighted inputs are added together with a bias b:

Finally, the sum is passed through an activation function:

Building Blocks: Neurons

Next, all the weighted inputs are added together with a bias b:

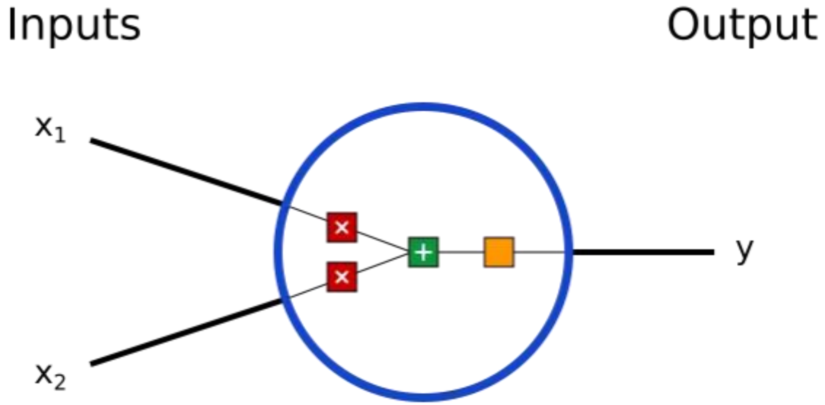
$$(x_1 * w_1) + (x_2 * w_2) + b$$



Building Blocks: Neurons

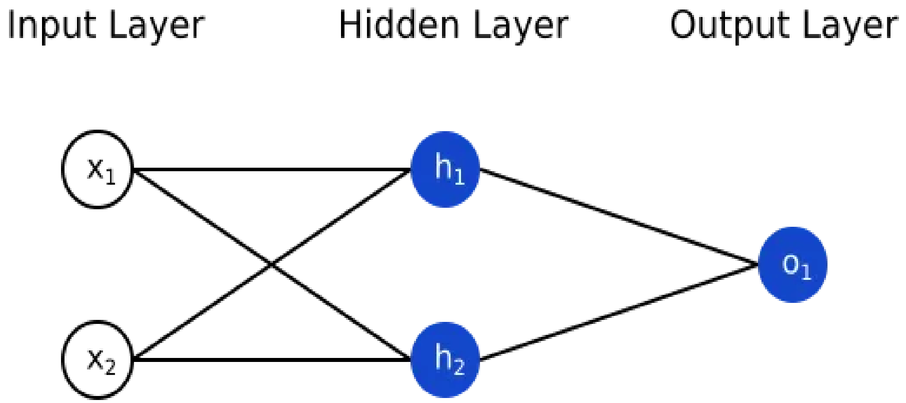
Finally, the sum is passed through **an activation function**:

$$y = f(x_1 * w_1 + x_2 * w_2 + b)$$



Combining Neurons into a Neural Network

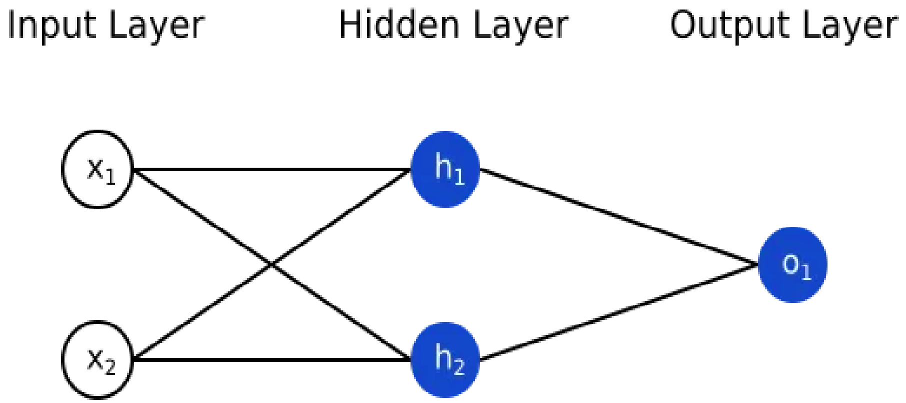
- Neural networks are made up of nodes or units, connected by links
    - A hidden layer is any layer between the input (first) layer and output (last) layer. There can be multiple hidden layers!



Combining Neurons into a Neural Network

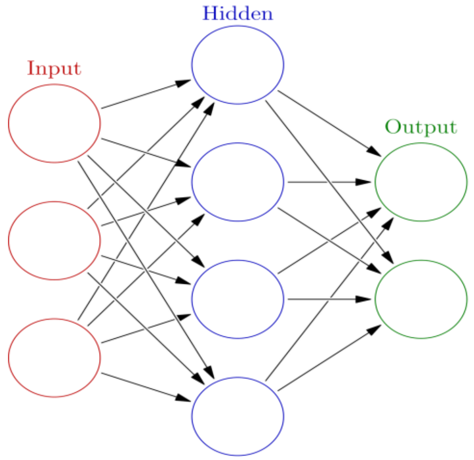
Example: This network has

- 2 inputs,
- A hidden layer with 2 neurons (h1 and h2), and
- an output layer with 1 neuron (o1).



Mostly deep neural networks are feed-forward

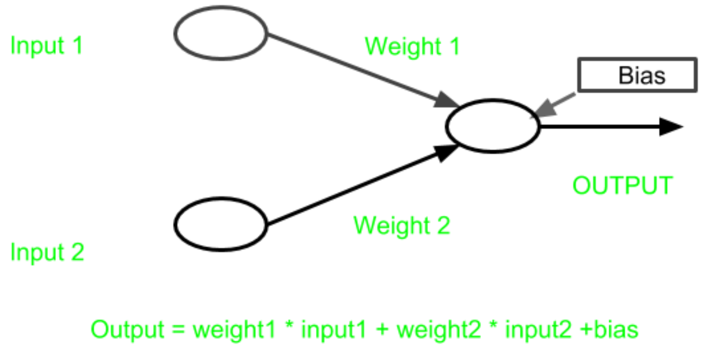
They only flow in one direction from input to output.



Parameters in NN

Each node has an input function (typically summing over weighted inputs), an activation function, and an output

- **Weights:** learns during training; represent the strength of connections between neurons in different layers.
- **Biases:** Additional parameters in each neuron,is a constant added to the input of an activation function.
  - It helps adjust the output of each neuron alongside the weighted sum of inputs, even if all inputs are zero.
  - This term allows the activation function to shift horizontally, affecting when it activates. Essentially, bias controls how



Both weights and biases are **learned during the training process**, adjusted iteratively to **minimize the differenc between predicted and actual outputs.**

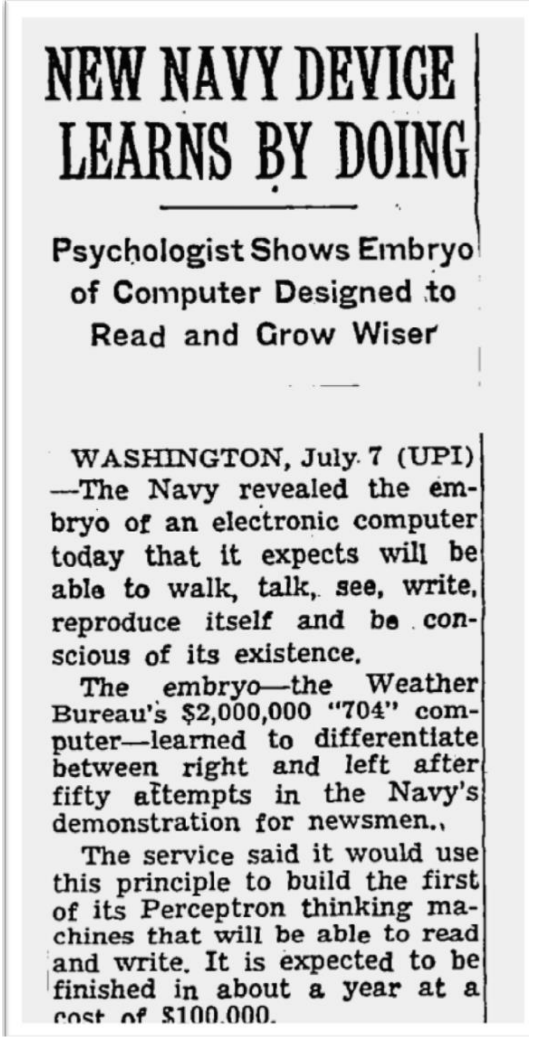
A Brief History of Neural Network:

Early Beginnings (1940s-1950s):

- Warren McCulloch and Walter Pitts proposed artificial neural networks (ANNs) in 1943, a simplified mathematical model of neurons.
- Frank Rosenblatt introduced the perceptron in 1957, a simple neural network model.

First Winter (1970s-1980s):

- Limited computing power and theoretical challenges slowed progress.
- Marvin Minsky and Seymour Papert's book "Perceptrons" (1969) contributed to a decline in interest.



NEW NAVY DEVICE LEARNS BY DOING

Psychologist Shows Embryo of Computer Designed to Read and Grow Wiser

WASHINGTON, July 7 (UPI) —The Navy revealed the embryo of an electronic computer today that it expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence.

The embryo—the Weather Bureau's $2,000,000 "704" computer—learned to differentiate between right and left after fifty attempts in the Navy's demonstration for newsmen.

The service said it would use this principle to build the first of its Perceptron thinking machines that will be able to read and write. It is expected to be finished in about a year at a cost of $100,000.

A Brief History of Neural Network:

Rebirth (Late 1980s-Early 1990s):

Advances in computing power and algorithms reignited interest. Backpropagation and new neural network architectures led to breakthroughs.

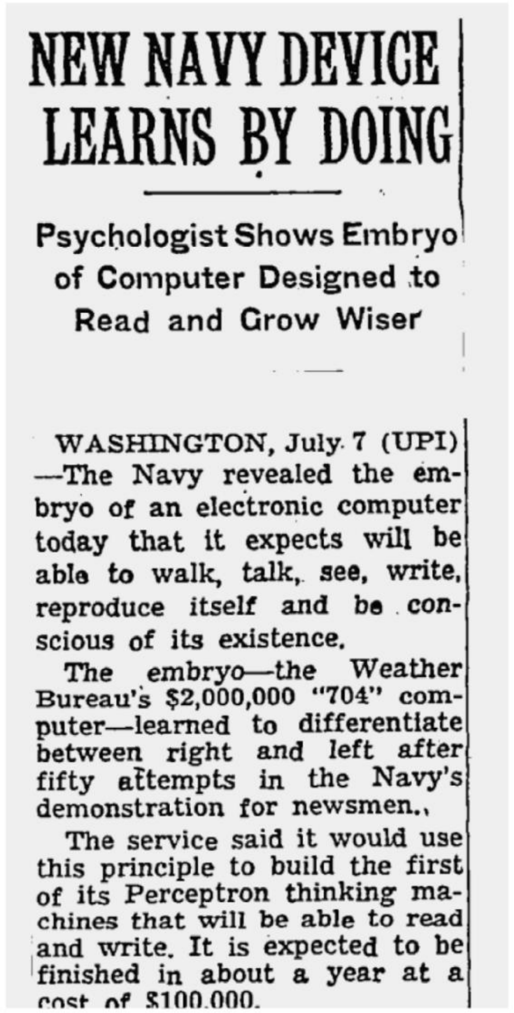
Boom and Bust (1990s-2000s): Surge of interest, but also skepticism and setbacks. Neural networks lost favor in some circles.

Graphics for Halo (2000s):

- Demand for advanced graphics in video games like Halo drove innovation.
- Graphics processing units (GPUs) improved, enabling more sophisticated neural network applications.

Modern Renaissance (2010s-Present):

- Big data, improved algorithms, and hardware advancements revived interest.
- Deep learning emerged as a dominant approach, achieving remarkable success across various domains.



NEW NAVY DEVICE LEARNS BY DOING

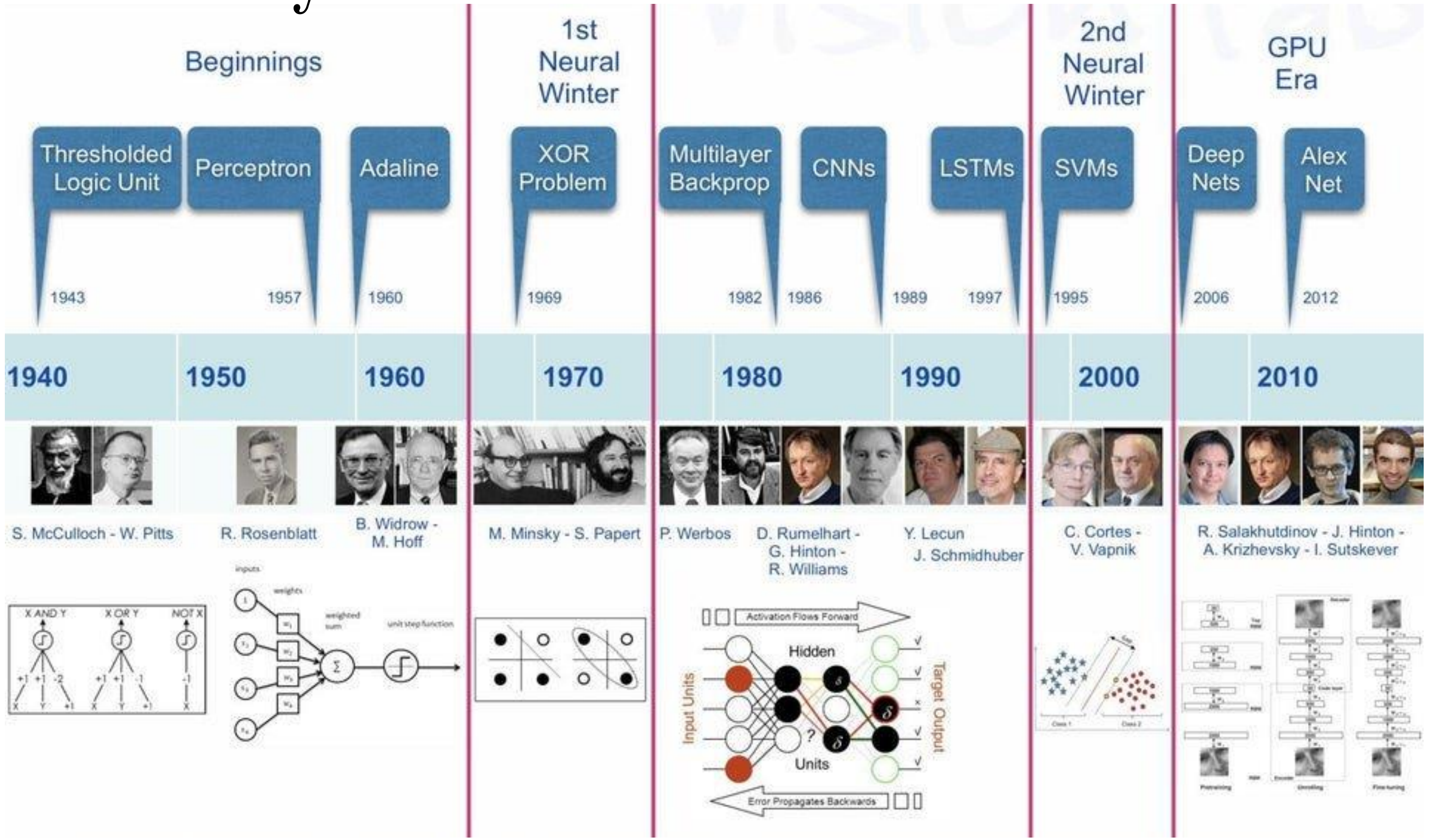
Psychologist Shows Embryo of Computer Designed to Read and Grow Wiser

WASHINGTON, July 7 (UPI) —The Navy revealed the embryo of an electronic computer today that it expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence.

The embryo—the Weather Bureau's $2,000,000 "704" computer—learned to differentiate between right and left after fifty attempts in the Navy's demonstration for newsmen.

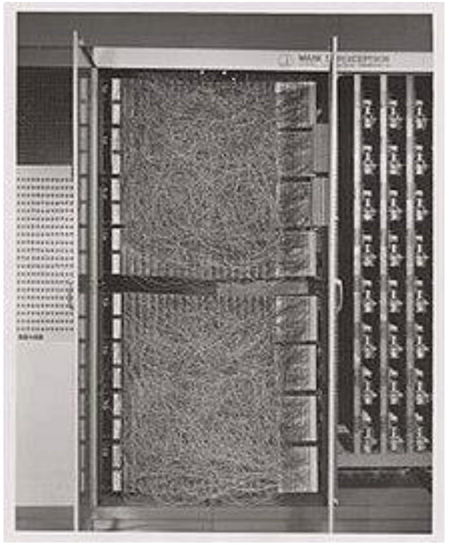
The service said it would use this principle to build the first of its Perceptron thinking machines that will be able to read and write. It is expected to be finished in about a year at a cost of $100,000.

A Brief History of Neural Network:



Early Neural Networks: The Perceptron

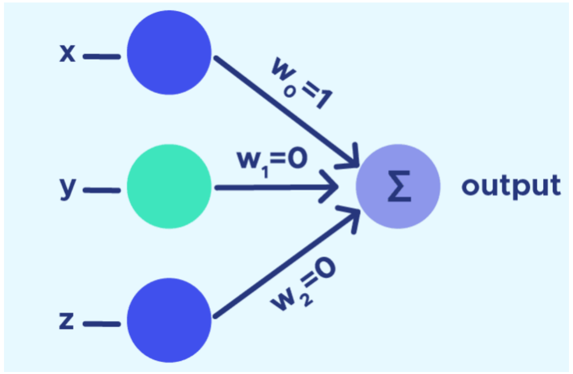
Originally, the perceptron algorithm was a hardware setup, not an algorithm. A series of photovoltaic cells were hooked up to a series of simple linear classifiers, hooked up to a series of outputs. The end result was intended to be a machine that could do image processing.



Early Neural Networks: The Perceptron

A perceptron is a type of artificial neural network (ANN) and the simplest form of a neural network used for binary classification tasks.

It was introduced by Frank Rosenblatt in 1957 and is the foundational building block of more complex neural networks.

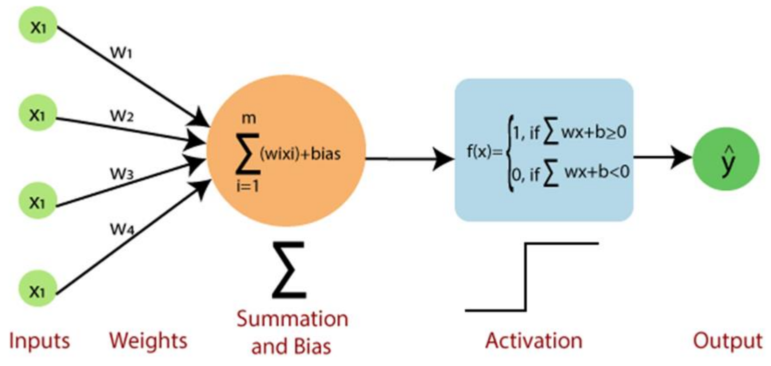


Early Neural Networks: The Perceptron

The perceptron algorithm uses a linear classifier that takes a multi-dimensional, real valued input X, and learns a series of weights, w.

The input is classified as positive if X*w is greater than zero, and is otherwise classified as negative:

$$w_1 x_1 + w_2 x_2 + \ldots + w_n \overset{+b}{x_n} > 0.$$


x1
x1
x1
x1
w1
w2
w3
w4
$\sum_{i=1}^{m}$(wixi)+bias
$f(x)=\begin{cases}1, \text{ if } \sum wx+b\geq 0\\0, \text{ if } \sum wx+b<0\end{cases}$
$\hat{y}$
Inputs
Weights
Summation and Bias
Activation
Output

The Perceptron

Simplest feedforward single layer neural network (No hidden layer).

Output= (vector of weights * vector of inputs)+bias

$$w_1x_1 + w_2x_2 + \ldots + w_n\overset{+b}{x_n} > 0.$$

We can define a perceptron by step function-

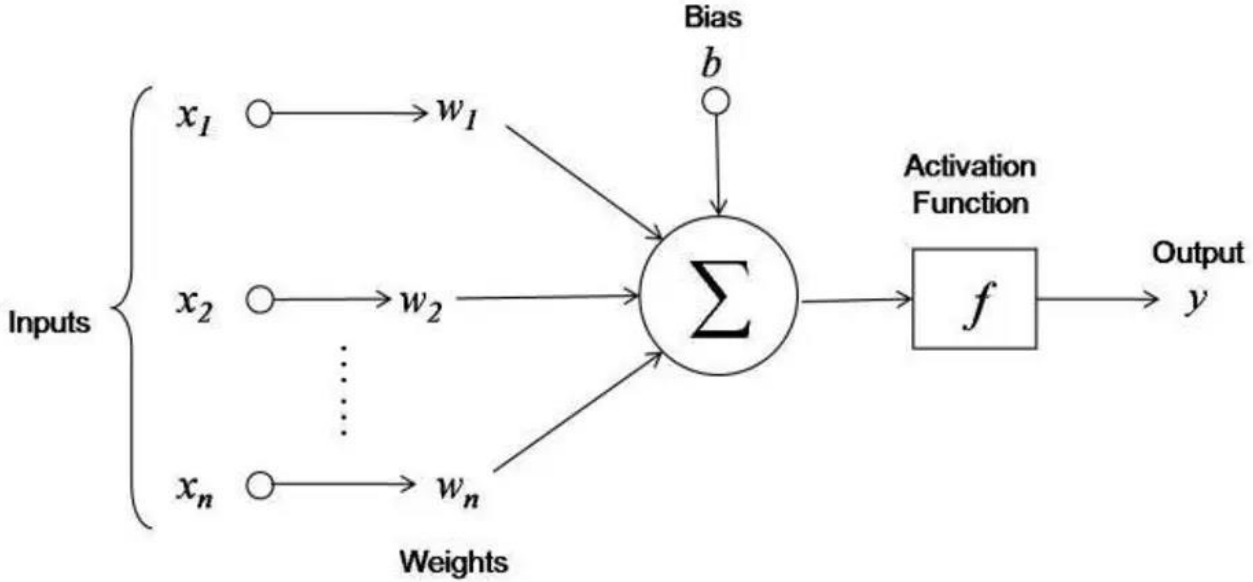
DEFINITION

$$o = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} + b \geq 0 \\ 0 & \text{if } \vec{w} \cdot \vec{x} + b < 0. \end{cases}$$

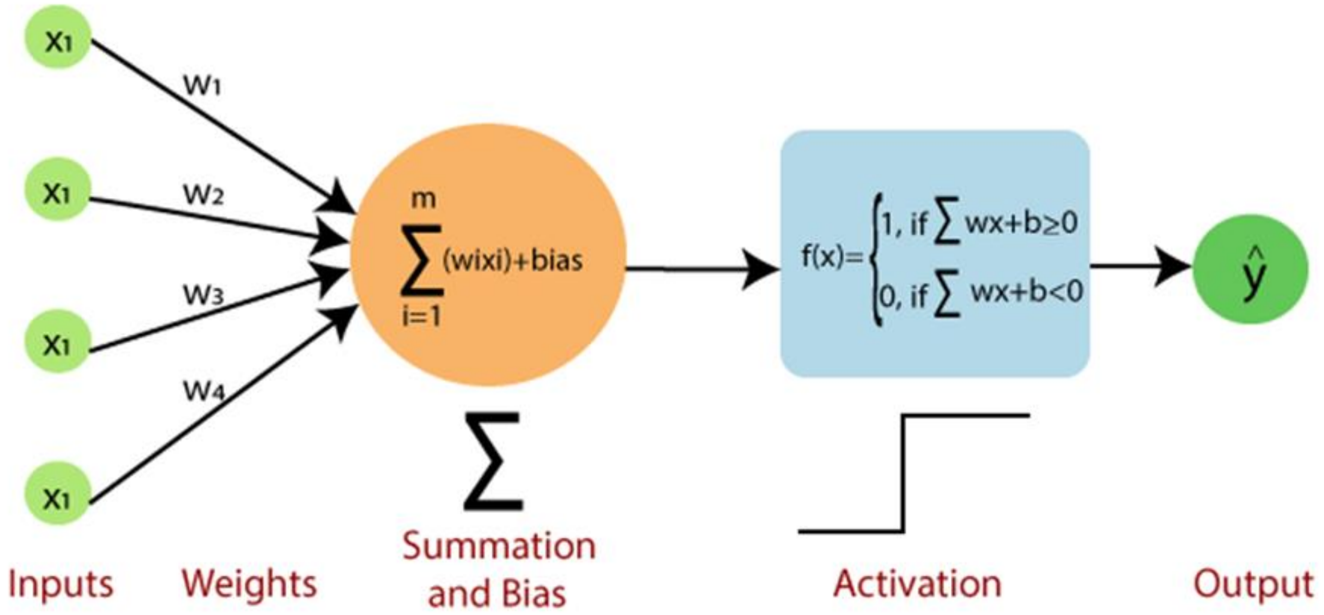
Generally speaking, a perceptron with activation function g(x),has

$$o = g(\vec{w} \cdot \vec{x} + b).$$

A Visual with Activation Function:



A Visual with Activation Function:



X1
X1
X1
X1
W1
W2
W3
W4
$\sum_{i=1}^{m}$(wixi)+bias
$f(x)=\begin{cases}1, \text{ if } \sum wx+b\geq 0\\ 0, \text{ if } \sum wx+b<0\end{cases}$
$\hat{y}$
$\sum$
Inputs
Weights
Summation and Bias
Activation
Output

Here Activation function **applies step rule which converts the numerical value to 0 or 1** so that it will be easy for data set to classify.

The Algorithm:

1. **Initialization:** Initialize the weights wi and bias b to small random values.
2. **Training:** For each training example (x,y), compute predicted output using current weight

Training: Perceptron Learning Process

The algorithm iterates through the training examples, adjusting the weight vector.

- Adjust the weights and bias based on the error between the predicted output and the actual target. This is usually done using the Perceptron Learning Rule, a simple form of gradient descent.
- Whenever a mistake in classification occurs, with the aim to update the weights to correctly classify examples and converge towards the correct decision boundary.

The Algorithm: Perceptron Learning Rule

And if the prediction is wrong or in other words the model has misclassified that example, we make the update for the parameters Weight. We don't update when the prediction is correct (or the same as the true/target value y).

Adjust the weights and bias based on the error between the predicted output and the actual target. This is usually done using the Perceptron Learning Rule, a simple form of gradient descent

A mistake is an incorrect prediction,

The Algorithm: Perceptron Learning Rule

Prediction can be wrong in two ways: False Positive or False Negative

- **Update Rule**
  - If $\hat{y} \neq y$:
    - **Mistake on Positive (False Negative)**: If y' =−1 (predicting negative), but y=+1 (true label is positive)
      $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \alpha \cdot \mathbf{x}$ [update the weight vector for the next iteration]
    - **Mistake on Negative (False Positive)**:
      $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \alpha \cdot \mathbf{x}$

Where α is a value (hyperparameter) between zero and one called the learning rate

- **Explanation**
  - **Mistake on Positive**: Update weights to correct for $\mathbf{w} \cdot \mathbf{x} < 0$.
    - $\mathbf{w}$ moves towards correctly classifying positive examples.
  - **Mistake on Negative**: Update weights to correct for $\mathbf{w} \cdot \mathbf{x} > 0$.
    - $\mathbf{w}$ moves towards correctly classifying negative examples.

Check:https://www.cs.tau.ac.il/~mansour/ml-course-10/scribe4.pdf

The Algorithm:

3. Prediction: Given an new input example x, compute predicted output using learned weight

i. predict positive **iff wt · x > 0,** else negative.

Limitation of Perceptrons

The perceptron algorithm had a number of weaknesses, primarily, it could only classify **linearly separable data.**

- Lacks hidden layers, limiting its capability to represent complex relationships in the data. It cannot learn more intricate patterns or hierarchical features.
- only learn and model linearly separable patterns. For non-linear data, it fails to converge to a solution as can never separate data that are not linearly separable.
- Used only for Binary Classification problems.

Perceptrons

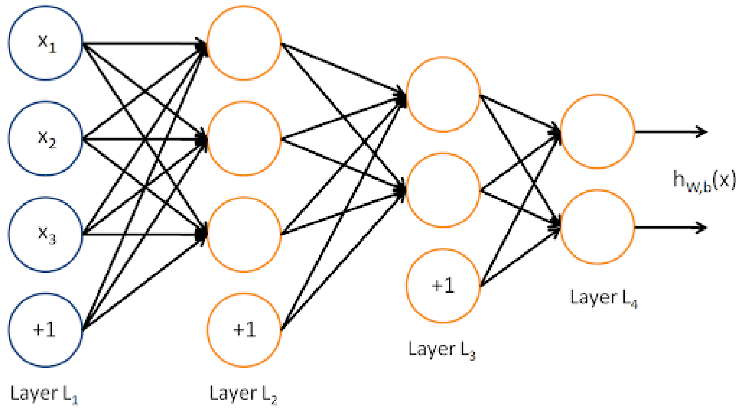
The perceptron algorithm had a number of weaknesses, primarily, it could only classify **linearly separable data.**

- Lacks hidden layers, limiting its capability to represent complex relationships in the data. It cannot learn more intricate patterns or hierarchical features.. → **ADD LAYERS**
- only learn and model linearly separable patterns. It's incapable of handling datasets that are not linearly separable. For non-linear data, it fails to converge to a solution. → **USE SOME ACTIVATION FUNCTIONS THAT SUPPORT NONLINEARITY**

Multi-Layer Neural Networks

Multi-layer neural networks were invented next. They had the following properties:

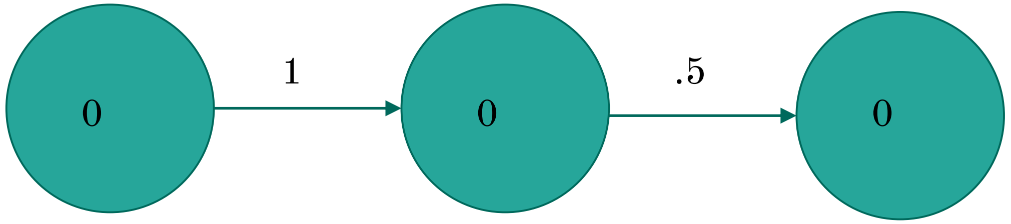
- Each computational unit, or neuron, is a node in a directed acyclic graph
- Neurons can have **activation values**, representing how activated one particular neuron is.
- Neurons can pass their activation values on to their neighbors, based on the weight of the connection between them.
- A multi-layer neural network usually has an **input layer** and an **output layer.**



Activation Rule

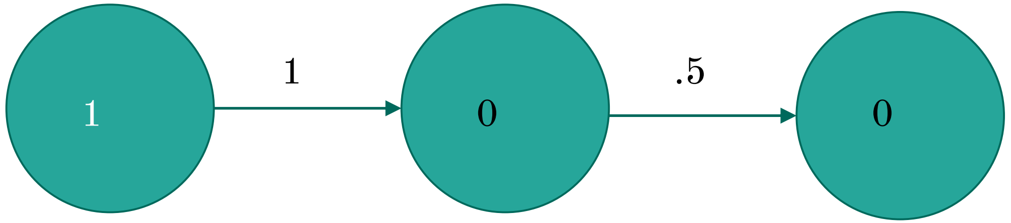
A node's activation is as follows: $\sum_{i}^{N} w_i a_i$, where N is the set of nodes in the previous layer.

An Example:



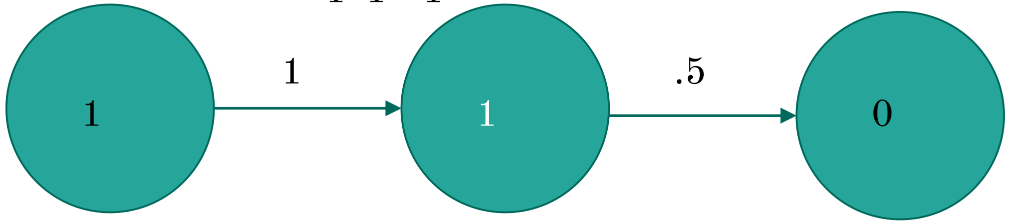
An Example:

We start by activating the node on the input layer:



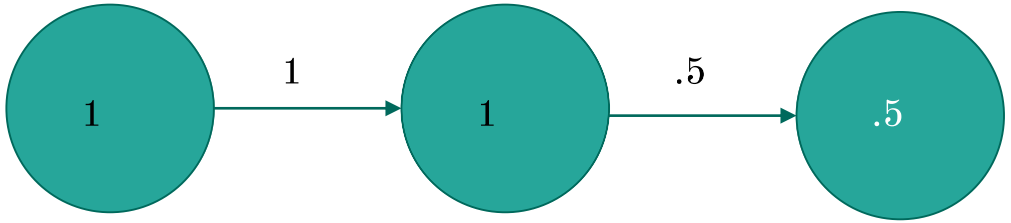
An Example:

At t=1, we activate this node by multiplying the input activation by the input weight.
1*1=1

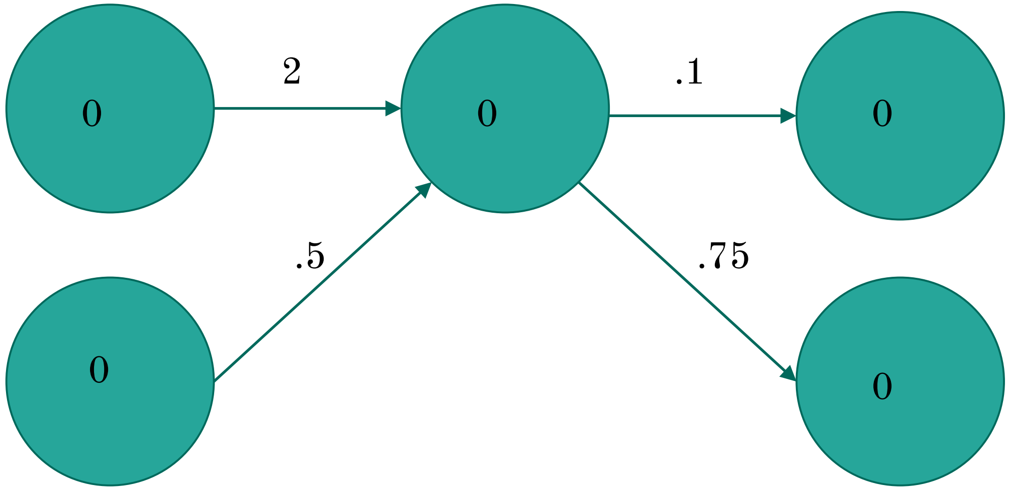


An Example:

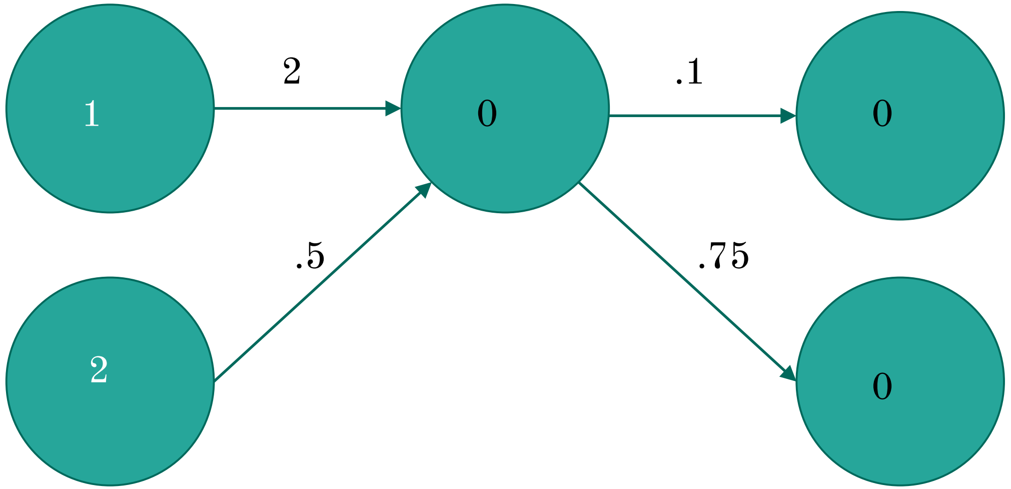
At t=2, we activate the output node. Our incoming activation is 1, and our incoming weight is .5, so...



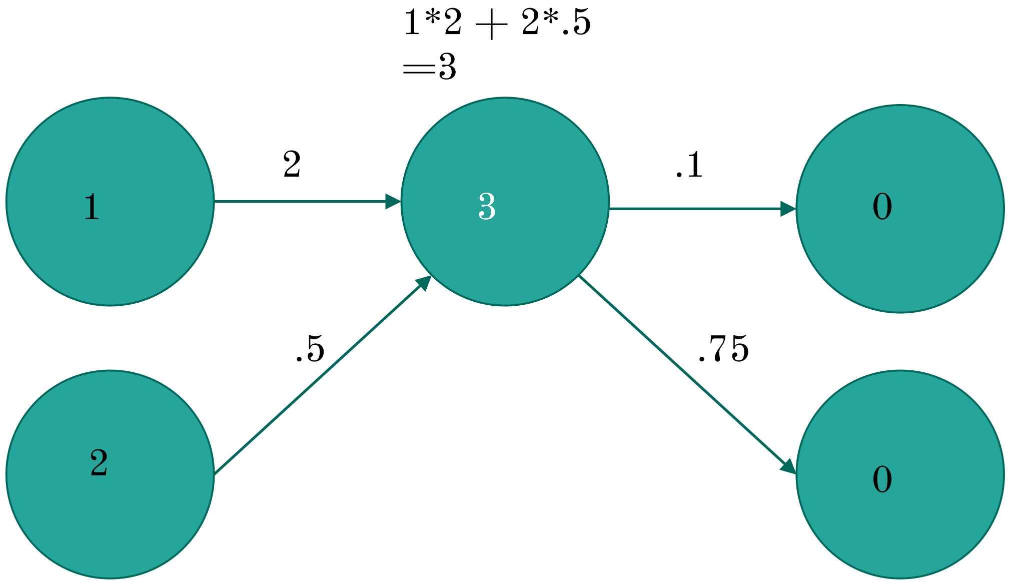
A More Complicated Example:



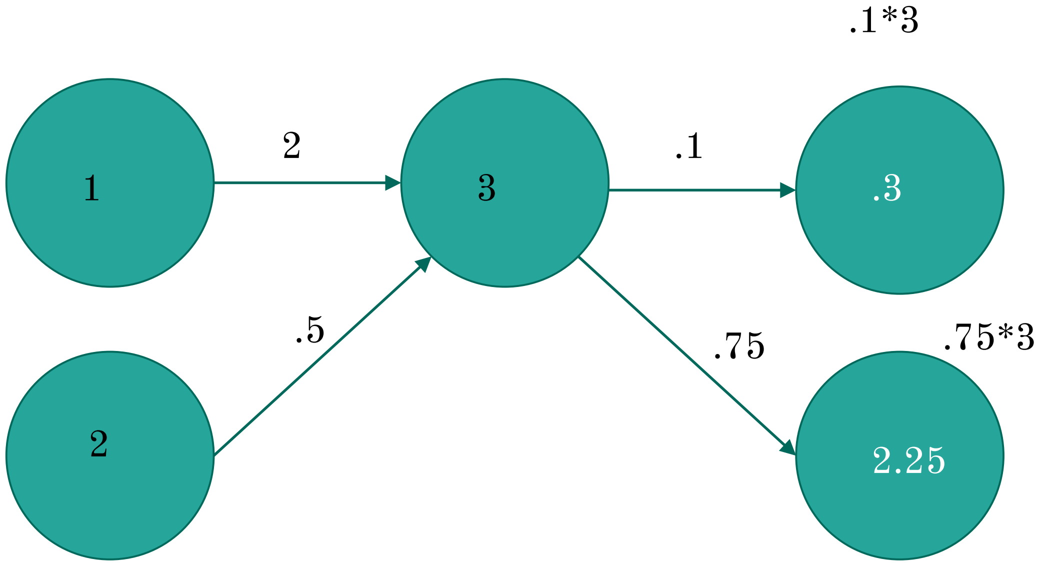
A More Complicated Example:



A More Complicated Example:



A More Complicated Example:

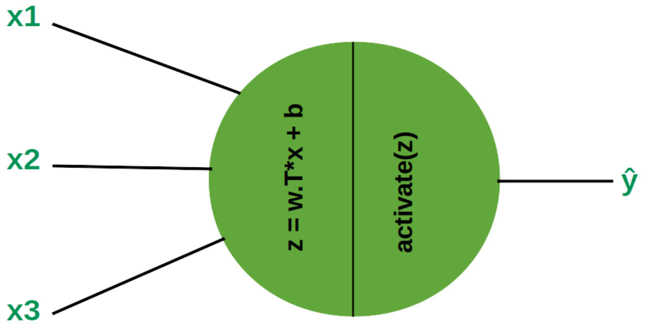


Activation Function

We can introduce “Non-Linearity” using different activation functions.

An activation function is a function that is added into an artificial neural network in order to help the network learn complex patterns in the data

- Mathematical “gate” between the input to current neuron and its output going to the next layer.



Y = Activation (∑(Weight*Input) +Bias)

Aim: introduce non-linearity into the output of a neuron

Types of Activation Function

The Activation Functions can be basically divided into 2 types-

- Linear or Identity Activation Function
    - Directly proportional to input x, f(x)=x

- Non-linear Activation Functions
    - Introduce nonlinearity (curves, bends, or transformations that are not straight lines) into neural networks, crucial for learning complex patterns and relationships in data. Types:
        - Sigmoid
        - TanH
        - Relu (Rectified Linear Unit)
        - Many more…

Why use a Non-linear Activation Function: Example: Sigmoid

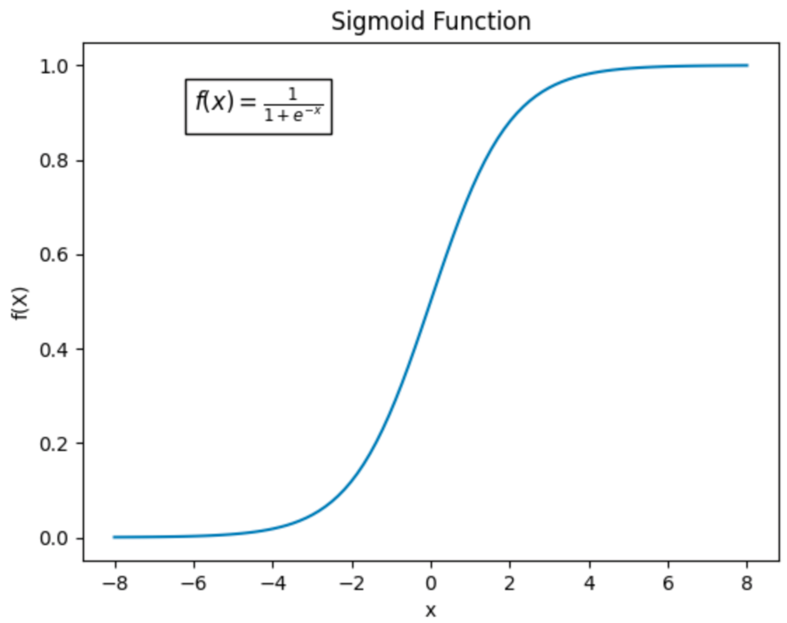
Unfortunately, in multi-layer neural networks, large input numbers can lead to nodes receiving excessively high activation values.

So, instead we compute a node's activation as:

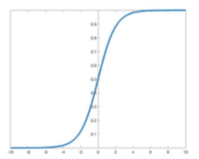
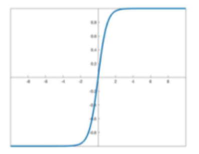
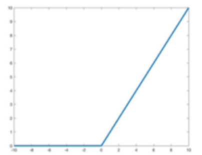
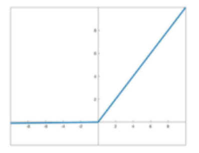
$$\sigma(\sum_{i}^{N} w_i a_i)$$

Where sigma σ is any function bounded between zero and one, most often the sigmoid function:

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



More Activation Functions

Activation function	Equation	Graph
Sigmoid	$S(x) = \frac{1}{1+e^{-x}}$	
Tanh	$\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	
ReLU	$RELU(x) = \begin{cases} 0 \; if \; x < 0 \\ x \; if \; x >= 0 \end{cases}$	
Leaky ReLU	$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0.01x & \text{otherwise} \end{cases}$	

Outputs range from 0 to 1, suitable for binary classification tasks.

Outputs range from -1 to 1, providing stronger gradients for learning compared to sigmoid.

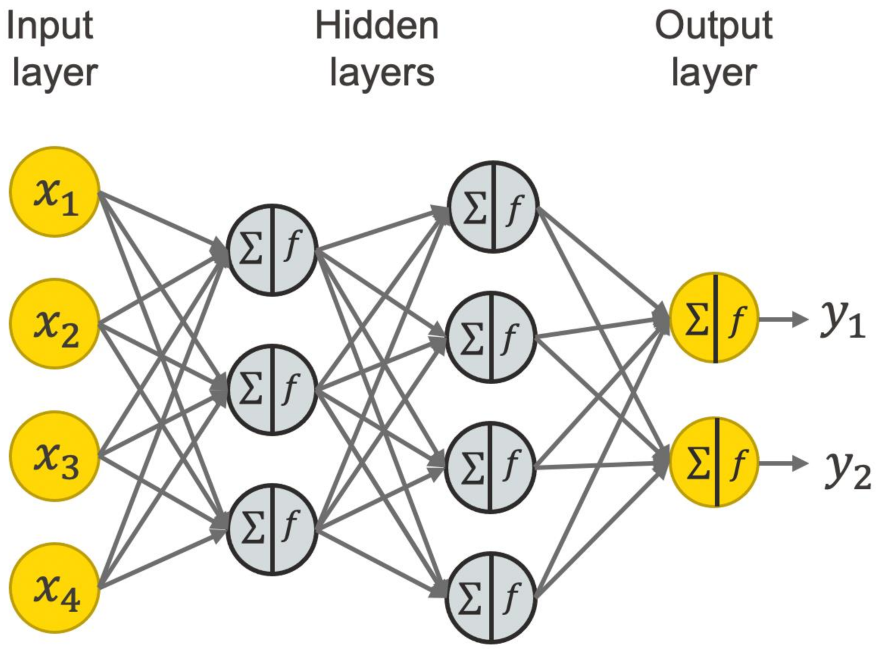
Simple and effective, sets negative values to zero (Outputs x if x is positive; otherwise outputs 0.), commonly used in hidden layers.

Addresses the dying ReLU problem by allowing a small, non-zero gradient (ex. 0.01) for negative inputs

By Definition

Multi-Layer Feed-Forward Neural Networks

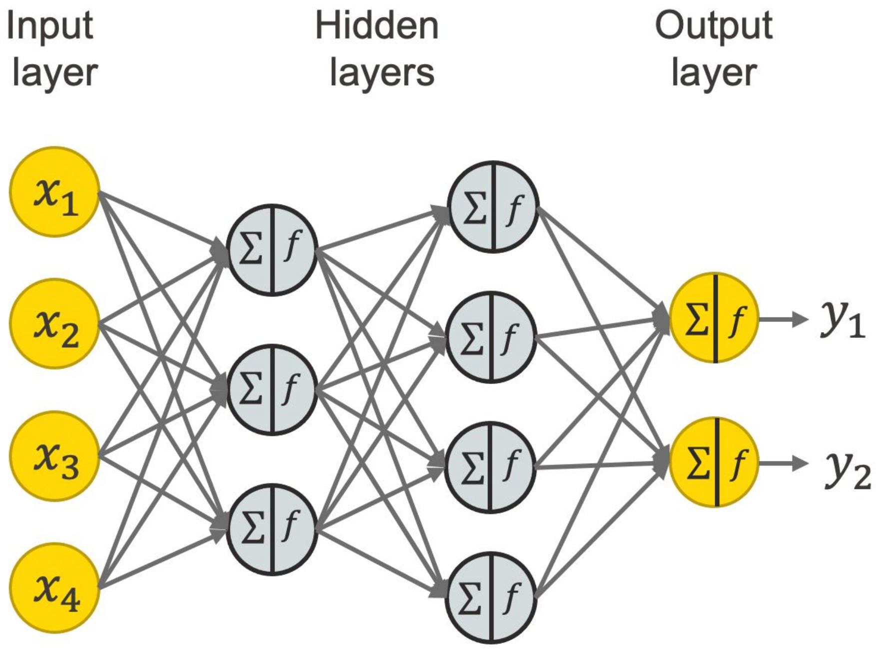
A type of artificial neural network architecture that consists of multiple layers of interconnected nodes, each layer passing information in a forward direction without any cycles



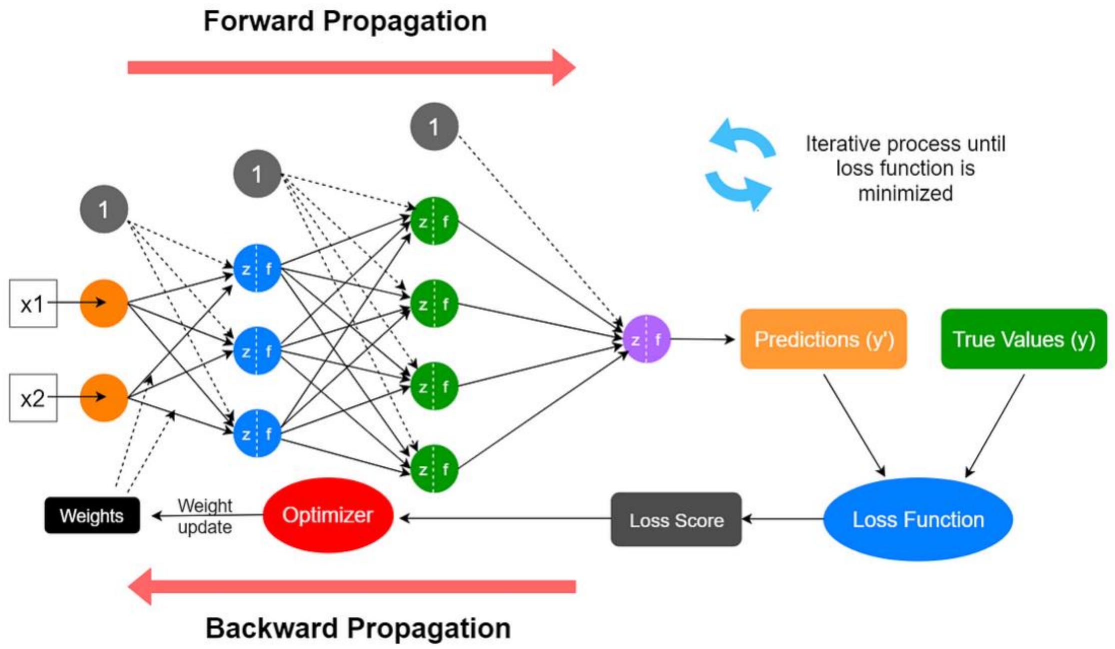
Multi-Layer Feed-Forward Neural Networks Mechanism

To classify something:

- Activate the input layer with the values of the input vector we want to classify.
- Propagate the activation to each subsequent hidden layer
- Propagate the activation to the output layer



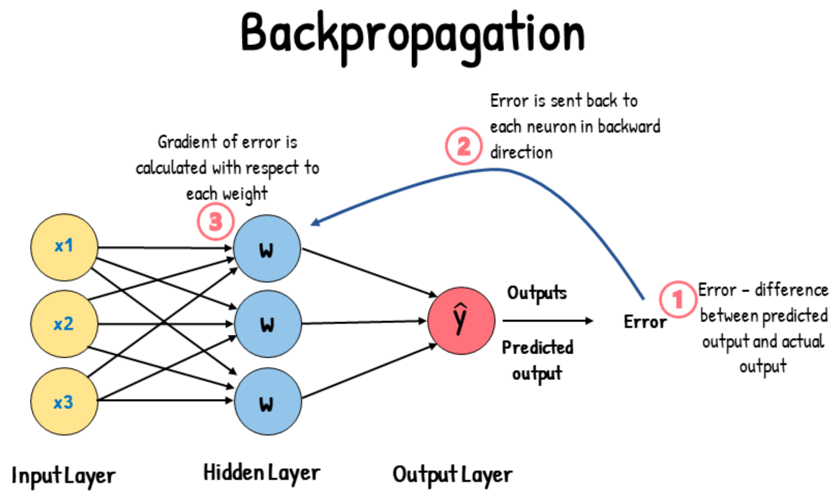
Next: BackPropagation Algorithm



Backpropagation

Backpropagation is the algorithm by which neural networks are trained.

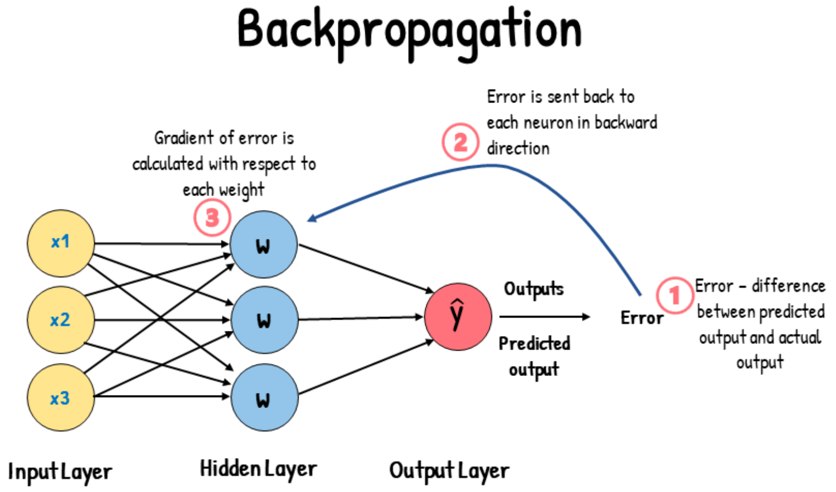
- **Goal with backpropagation:** Update each of the weights in the network
  - Minimize the error for each output neuron and the network as a whole.
- **Why?** so that they cause the actual output to be closer the target output


Backpropagation
Error is sent back to each neuron in backward direction
2
Gradient of error is calculated with respect to each weight
3
x1
x2
x3
w
w
w
ŷ
Outputs
Predicted output
Error
1
Error - difference between predicted output and actual output
Input Layer
Hidden Layer
Output Layer

Backpropagation

Backpropagation is the algorithm by which neural networks are trained.

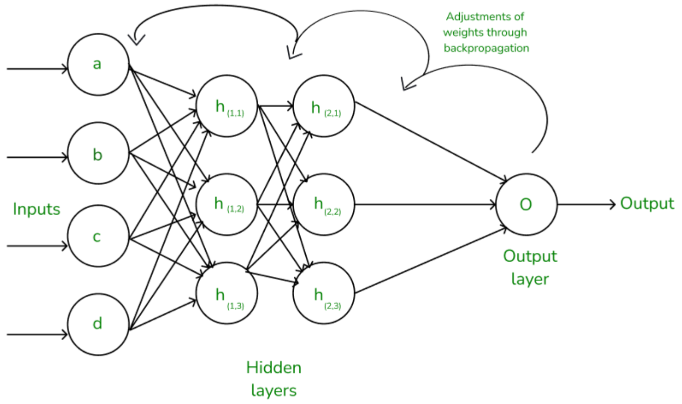
- **How?** For every training example, we compute the loss function, and then iteratively **adjust each weight (and biases)** based on the gradient of the loss function, which indicates how much each parameter contributed to the prediction error.


Backpropagation
Gradient of error is calculated with respect to each weight
3
2
Error is sent back to each neuron in backward direction
x1
x2
x3
w
w
w
ŷ
Outputs
Predicted output
Error
1
Error - difference between predicted output and actual output
Input Layer
Hidden Layer
Output Layer

Backpropagation

How? Steps:

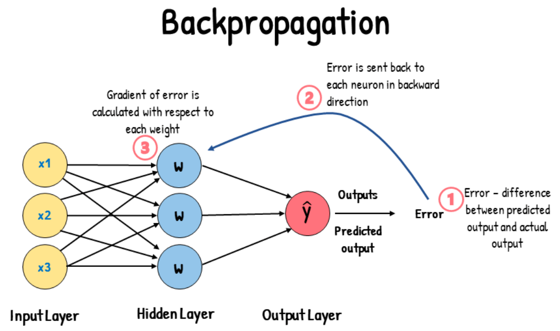
- We calculate the loss (after completing forward pass)
- The error is then propagated backward through the network.
- Using the chain rule, it calculates the **contribution of each weight** to the overall error.
- The derivative of the error with respect to each weight is computed, adjusting the weights to minimize the error.



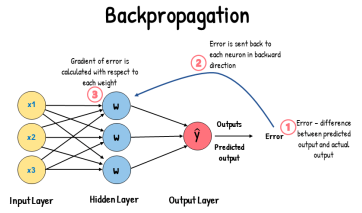
The Process of Backpropagation

Follows following steps:

1. Forward Pass / Propagation & Loss Calculation
2. Backward Pass (Gradient Calculation or Backpropagation)

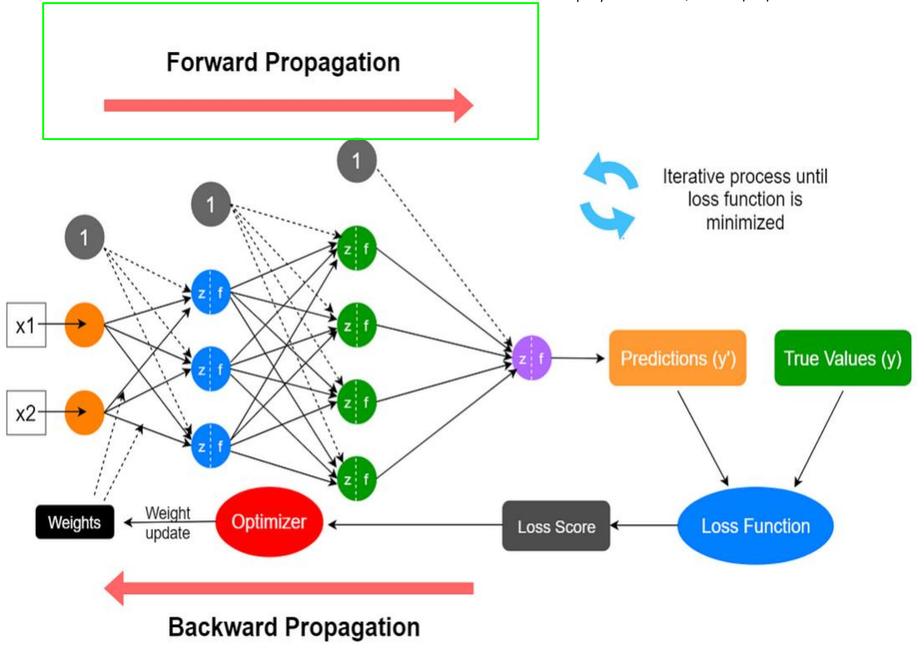

Backpropagation
Gradient of error is calculated with respect to each weight
3
Error is sent back to each neuron in backward direction
2
x1
x2
x3
w
w
w
ŷ
Outputs
Predicted output
Error
1
Error - difference between predicted output and actual output
Input Layer
Hidden Layer
Output Layer

Backpropagation: **Forward Pass**


Backpropagation
Error is sent back to each neuron in backward direction
2
Gradient of error is calculated with respect to each weight
3
x1
x2
x3
W
W
W
ŷ
Outputs
Predicted output
Error
1
Error - difference between predicted output and actual output
Input Layer
Hidden Layer
Output Layer

Step 1.1 : Forward Pass: The input data is passed **forward** through the network, producing an output prediction.

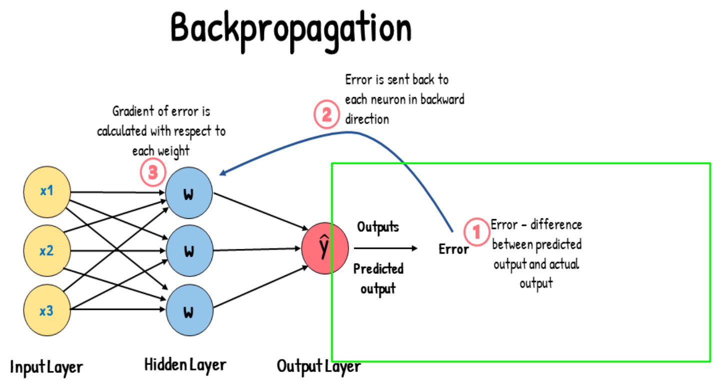
- Each layer computes a weighted sum of inputs and applies an activation function.
- Outputs are generated layer by layer until the final prediction is obtained.

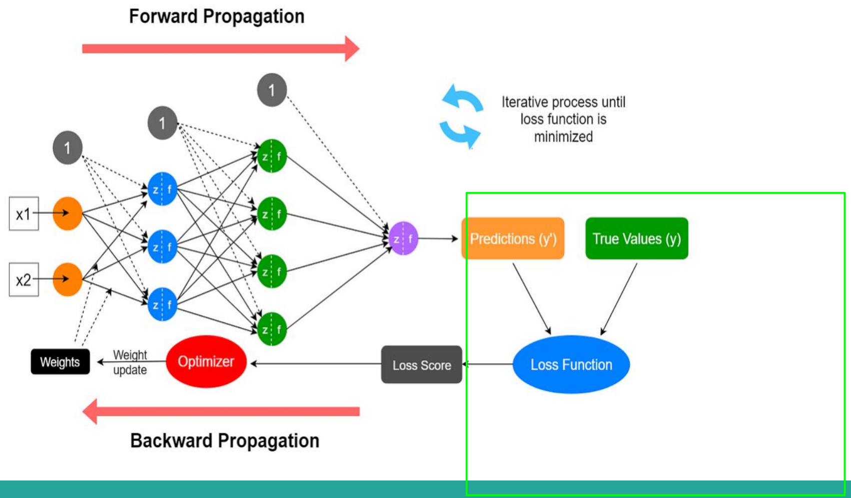


Backpropagation: **Forward Pass**

Step 1.2 : Calculating Loss / Error: Quantifies the difference between predicted and actual values.

- Compare the predicted output to the expected target values to compute the error or loss, typically using a loss function such as Mean Square Error (MSE).

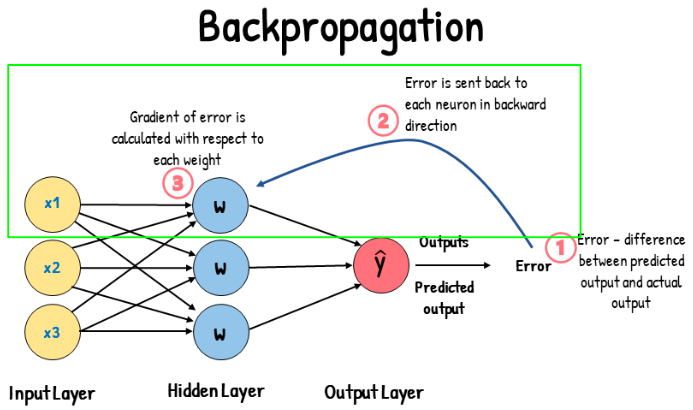

Backpropagation
Error is sent back to each neuron in backward direction
2
Gradient of error is calculated with respect to each weight
3
x1
x2
x3
W
W
W
ŷ
Outputs
Predicted output
Error
1
Error - difference between predicted output and actual output
Input Layer
Hidden Layer
Output Layer

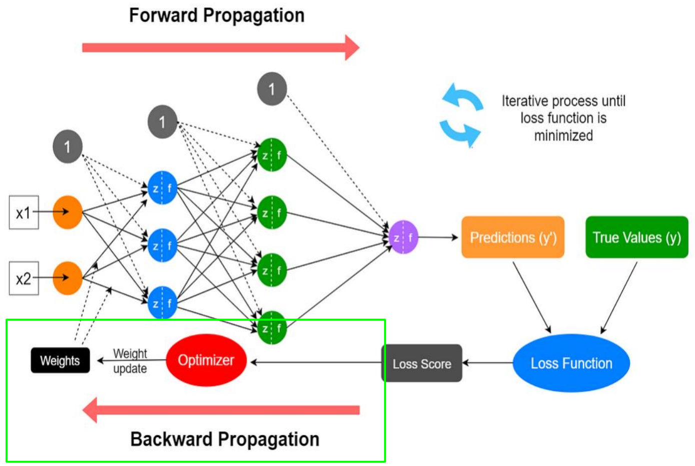


Backpropagation: **Backward Pass**

Step 2: Backward Pass (Backpropagation):

- The **error is then propagated backward through the network.**
- Adjustments to the network's internal parameters (weights and biases) are made based on the calculated error, aiming to minimize the difference between the predicted and actual output.


Backpropagation
Error is sent back to each neuron in backward direction
2
Gradient of error is calculated with respect to each weight
3
x1
x2
x3
w
w
w
ŷ
Outputs
Predicted output
Error
1
Error – difference between predicted output and actual output
Input Layer
Hidden Layer
Output Layer



Backpropagation

Say we have the following loss function:

$$L = (predicted - actual)^2$$

What we would like to know is

$$\frac{\delta L}{\delta w_i}$$

represents the partial derivative of the loss function (L) w.r.t the weights (w) of the neural network. This derivative indicates the rate of change of the loss with respect to a particular weight.

For all the different weights in the network, weights with a high $\frac{\delta L}{\delta w_i}$ strongly contributed to the incorrect classification (indicating they had a significant impact on the error); conversely, weights with a low $\frac{\delta L}{\delta w_i}$ had less influence on the incorrect classification.

Backpropagation

$L = (predicted - actual)^2$ is actually $L = (\sigma(\sum_{i}^{previouslayer} w_i * a_i) - actual)^2$

Using the **chain rule**, we can compute $\frac{\delta L}{\delta w_i}$ for every node.

By applying the chain rule, we can calculate the influence or impact of each node on the final outcome. This helps us understand the role of each node in the network.

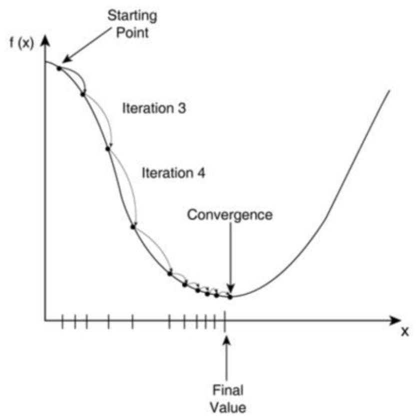
Once we do, we can say: $w_{new} = w_{old}$ - $\alpha \frac{\delta L}{\delta w_i}$

*** The chain rule tells us how to find the **derivative of a composite function (one function is nested over the other)**..

Why Backpropagation?

Backpropagation along with Gradient Descent forms the backbone and powerhouse of neural networks.

- While Gradient Descent constantly updates and moves the weights and bias towards the minimum of the cost function,
  - backpropagation evaluates the gradient of the cost w.r.t. weights and biases, the magnitude and direction of which is used by gradient descent to evaluate the size and direction of the corrections to weights and bias parameters.



A simple visual description of the movement towards the minima of a 2d function. The step-size of jump is determined by the value of the gradient at each point.

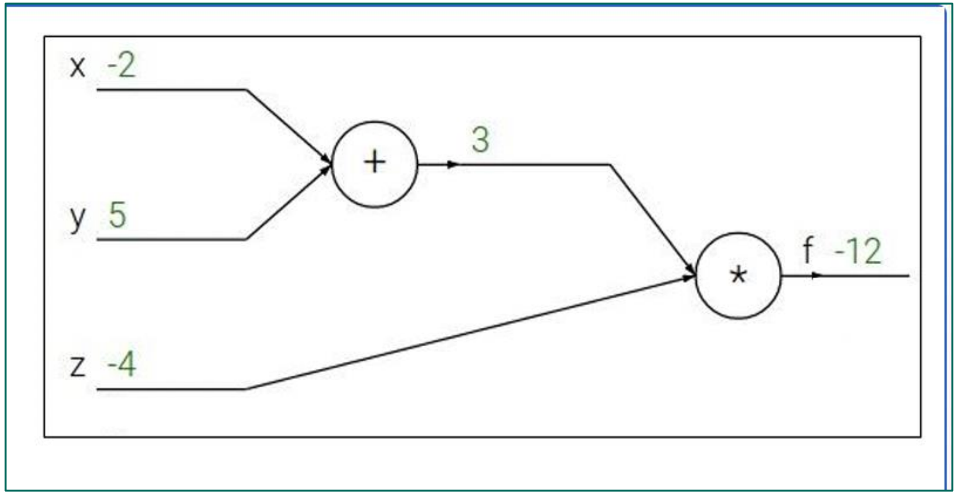
Next: Backpropagation: a simple example

Backpropagation: a simple example

Given, f(x,y,z) = (x+y).z

e.g. x = -2, y = 5, z = -4

(consider all weights are 1 and bias is 0 for simplicity)



Backpropagation: a simple example

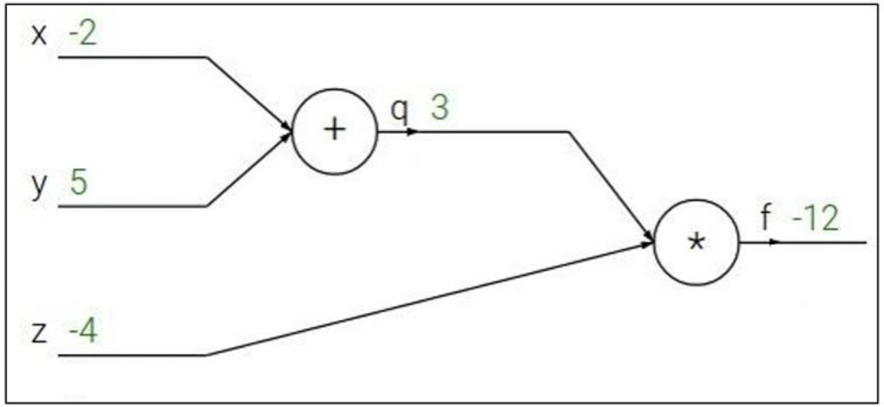
Given, f(x,y,z) = (x+y).z

e.g. x = -2, y = 5, z = -4

1. **Forward Pass (Left to Right): Compute Output**

q = x + y = -2 + 5 = 3

f = q.z = 3 * -4 = 12



Backpropagation: a simple example

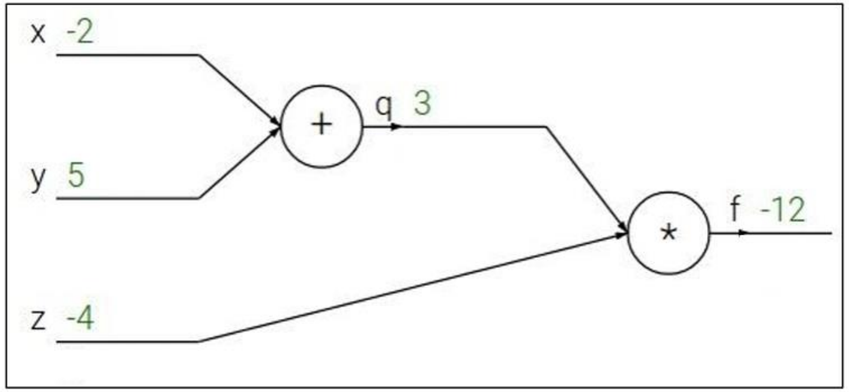
Given, f(x,y,z) = (x+y).z

e.g. x = -2, y = 5, z = -4

1. **Forward Pass (Left to Right): Compute Output**

q = x + y = -2 + 5 = 3

f = q.z = 3 * -4 = 12



1. **Backward Pass(Right to Left): Compute Deriv** (of the output w.r.t each of the input x,y,z)

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

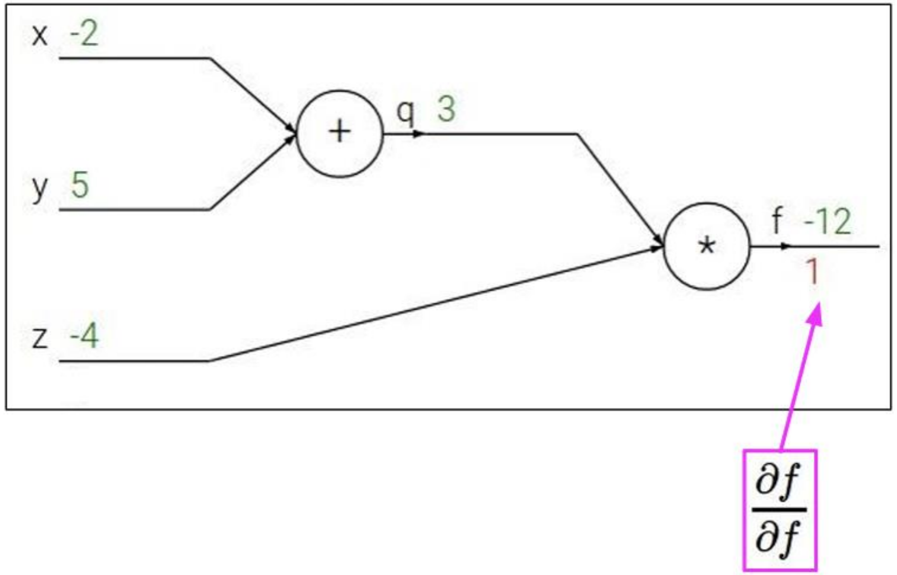
Backpropagation: Compute: $\frac{\partial f}{\partial f}$

Given, f(x,y,z) = (x+y).z

e.g. x = -2, y = 5, z = -4

Backward Pass: Compute Derivatives

Start with the base case



Backpropagation: Compute: $\frac{\partial f}{\partial z}$

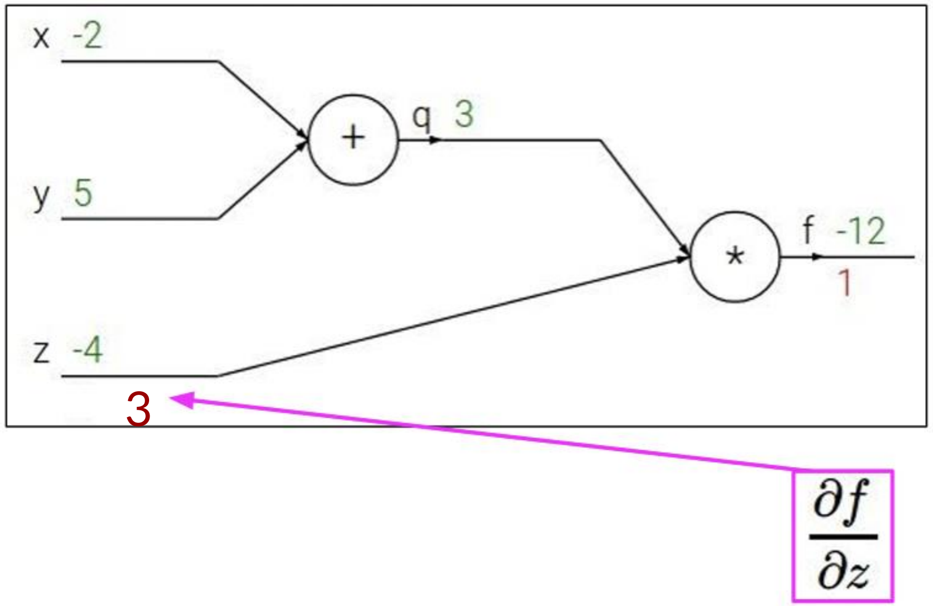
Given, f(x,y,z) = (x+y).z

e.g. x = -2, y = 5, z = -4

Backward Pass: Compute Derivatives

$\frac{\partial f}{\partial z}$: we know **f = qz**

$$\boxed{\frac{\partial f}{\partial z} = q}$$ = 3



Backpropagation: Compute: $\frac{\partial f}{\partial q}$

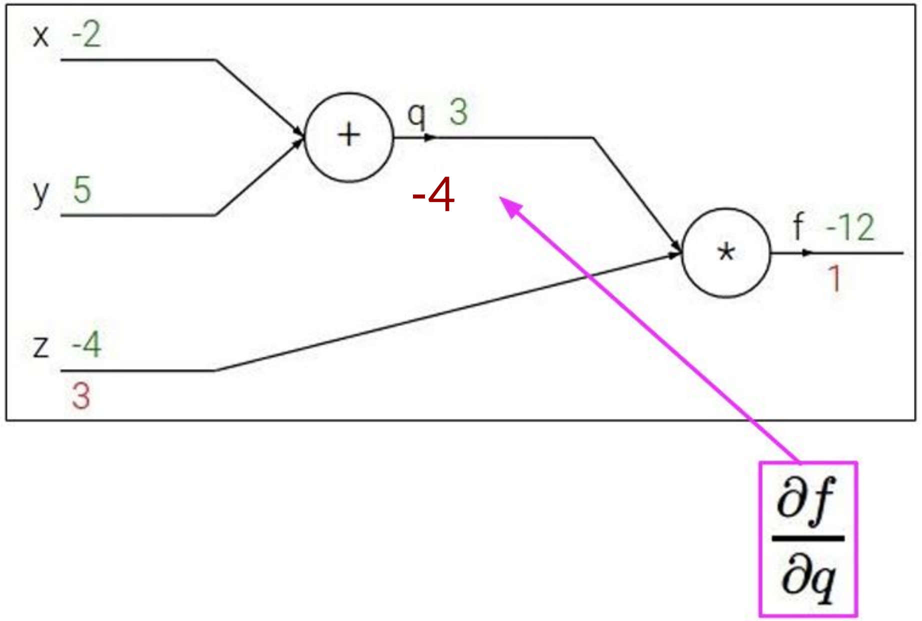
Given, f(x,y,z) = (x+y).z

e.g. x = -2, y = 5, z = -4

Backward Pass: Compute Derivatives

$\frac{\partial f}{\partial q}$: we know **f = qz**

$\frac{\partial f}{\partial q} = z$, = -4



Backpropagation: Compute: $\frac{\partial f}{\partial y}$

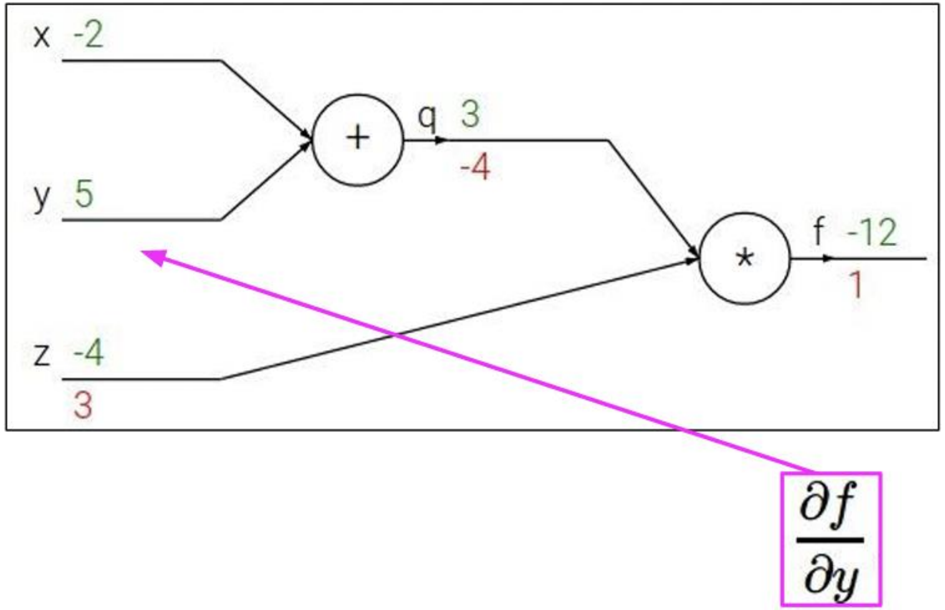
Given, f(x,y,z) = (x+y).z

e.g. x = -2, y = 5, z = -4

Backward Pass: Compute Derivatives

Continue process to left

$$\frac{\partial f}{\partial y}$$



Here, the value of Y is not directly connected to the output value F → so need chain rule to compute derivative

Backpropagation: Compute: $\frac{\partial f}{\partial y}$

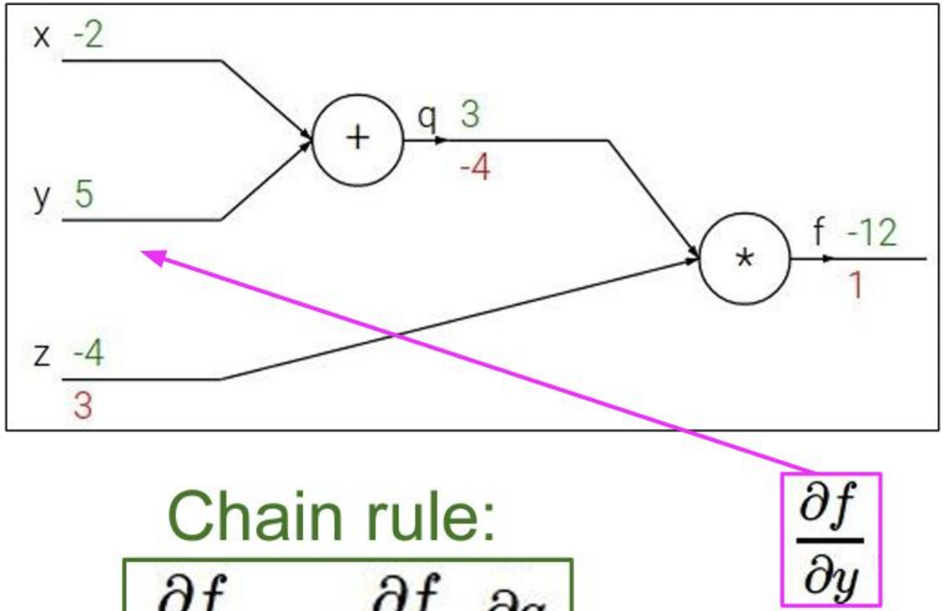
Given, f(x,y,z) = (x+y).z

e.g. x = -2, y = 5, z = -4

Backward Pass: Compute Derivatives

$$\frac{\partial f}{\partial y}$$

Here, the chain rule take into account the influence of Y on the intermediate variable Q.



Chain rule:

$$\boxed{\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y}}$$

HELPING SLIDES

Chain Rule

If y = f(g(x)), then y' = f'(g(x)). g'(x).

The chain rule states that the instantaneous rate of change of f relative to g relative to x helps us calculate the instantaneous rate of change of f relative to x.

If $y = f(u)$, where $u = g(x)$

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}$$

Read: https://www.khanacademy.org/math/ap-calculus-ab/ab-differentiation-2-new/ab-3-1a/a/chain-rule-review

Backpropagation: Compute: $\frac{\partial f}{\partial y}$

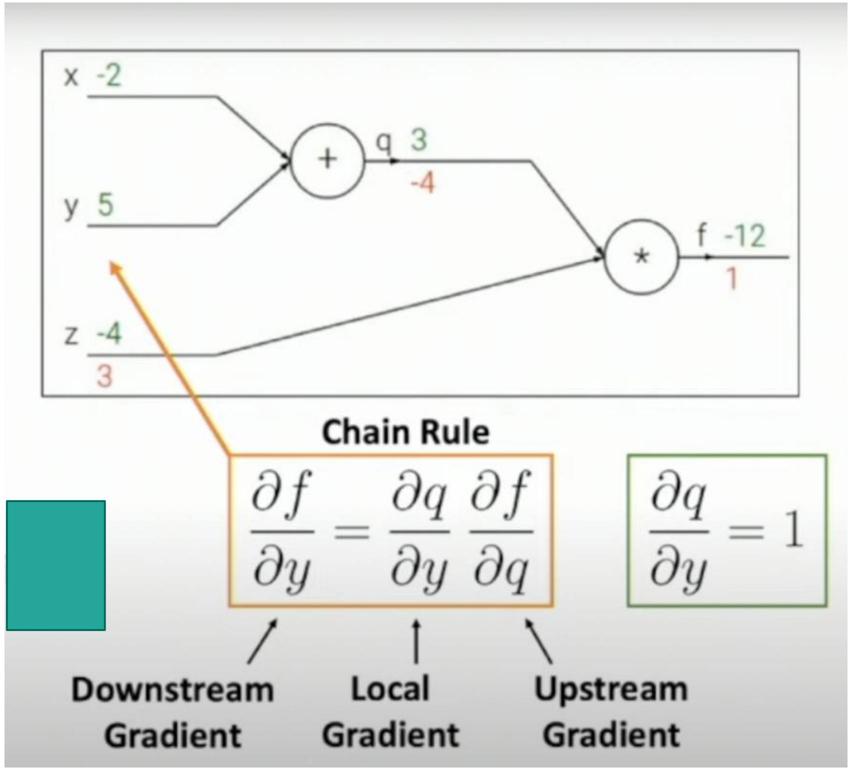
Given, f(x,y,z) = (x+y).z

e.g. x = -2, y = 5, z = -4

Backward Pass: Compute Derivatives

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$\frac{\partial f}{\partial y}$ = 1 * -4 =-4



$$\frac{\partial f}{\partial y} = \frac{\partial q}{\partial y}\frac{\partial f}{\partial q}$$

$$\frac{\partial q}{\partial y} = 1$$

Backpropagation: Compute: $\frac{\partial f}{\partial x}$

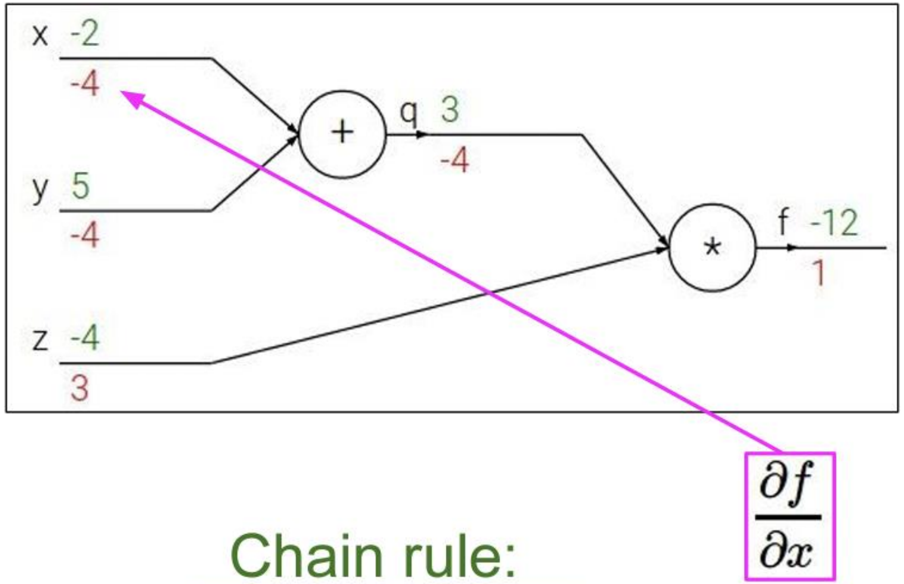
Given, f(x,y,z) = (x+y).z

e.g. x = -2, y = 5, z = -4

Backward Pass: Compute Derivatives

$$\boxed{q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1}$$

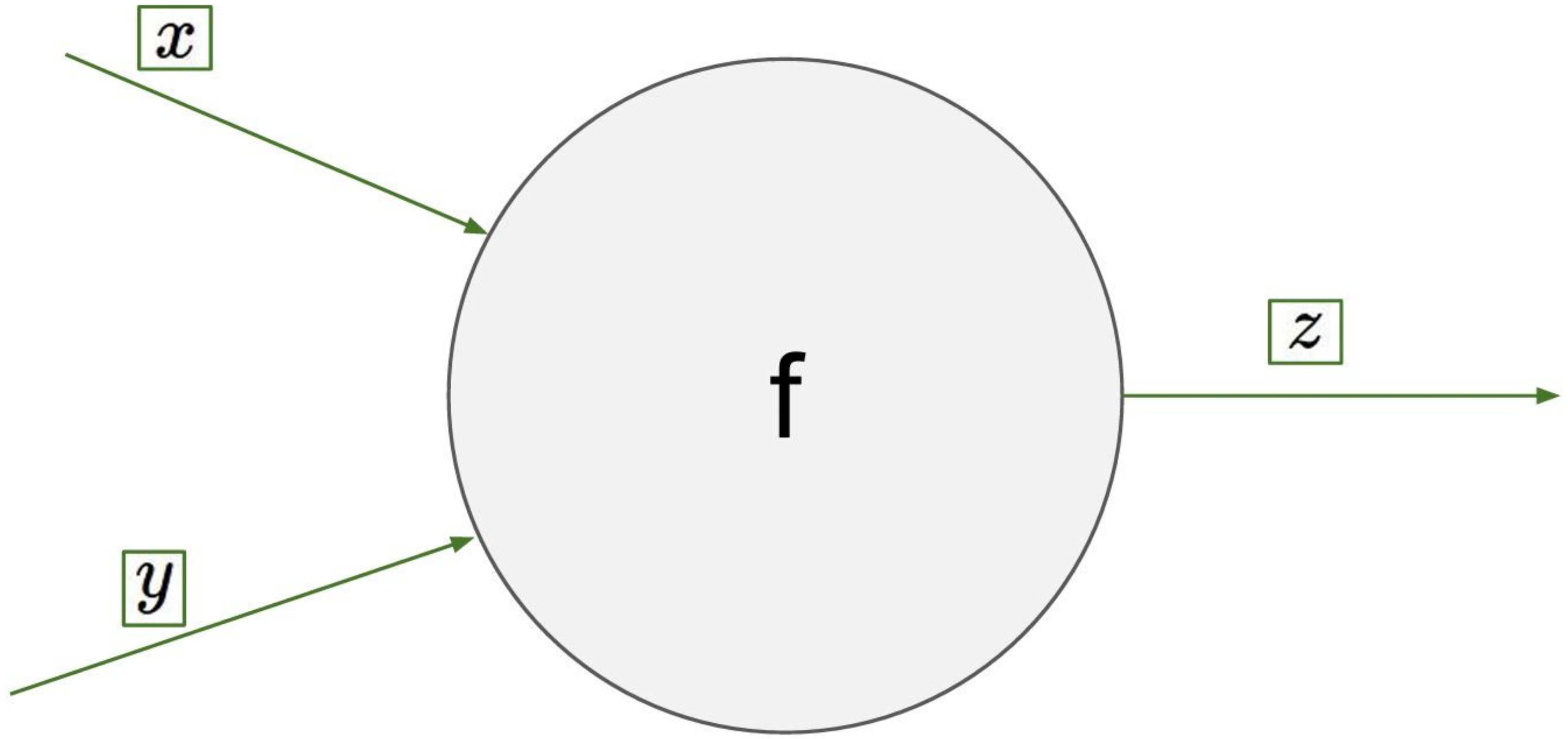
$$\frac{\partial f}{\partial x} = 1 * -4 = -4$$

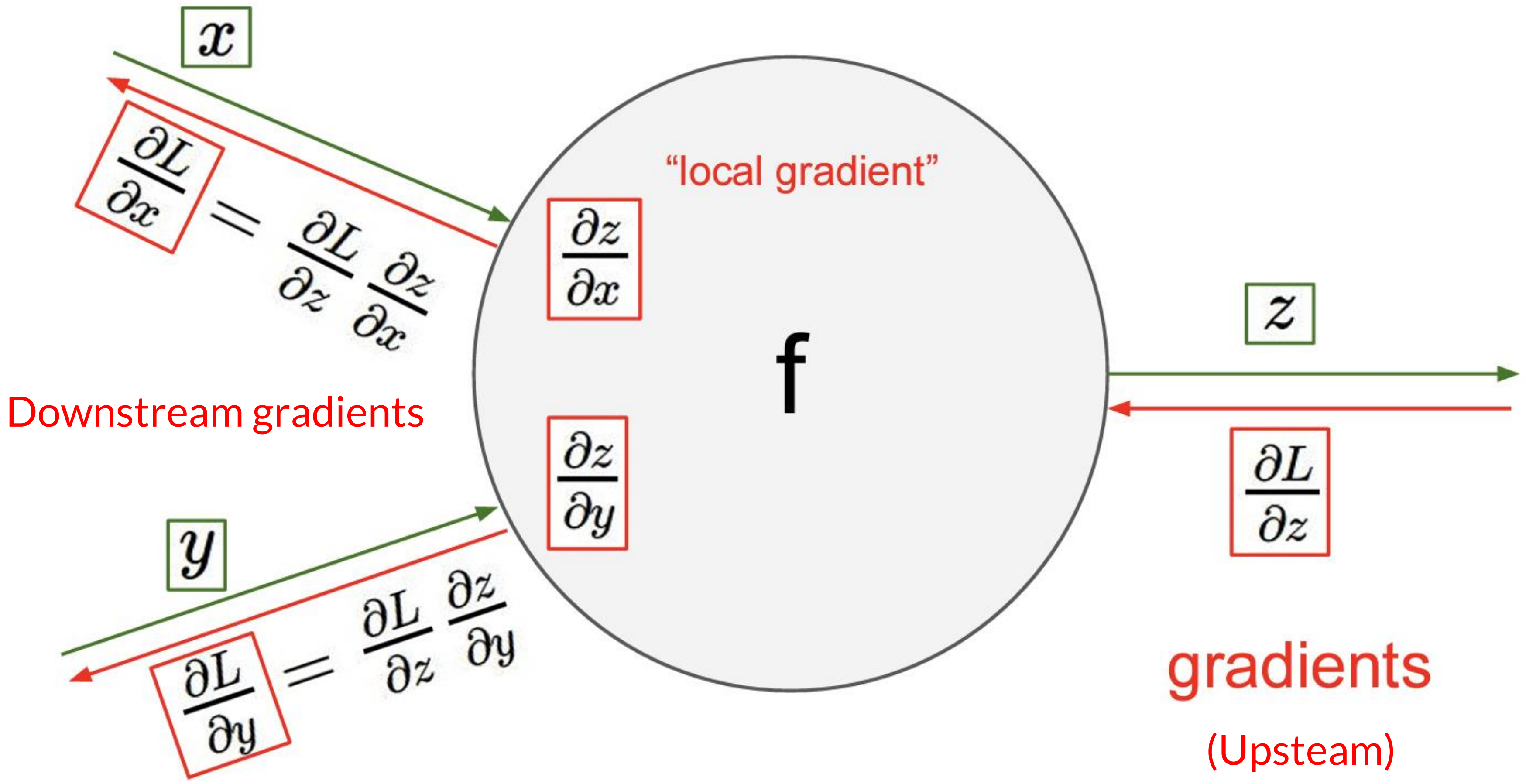


Chain rule:

$$\boxed{\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}}$$

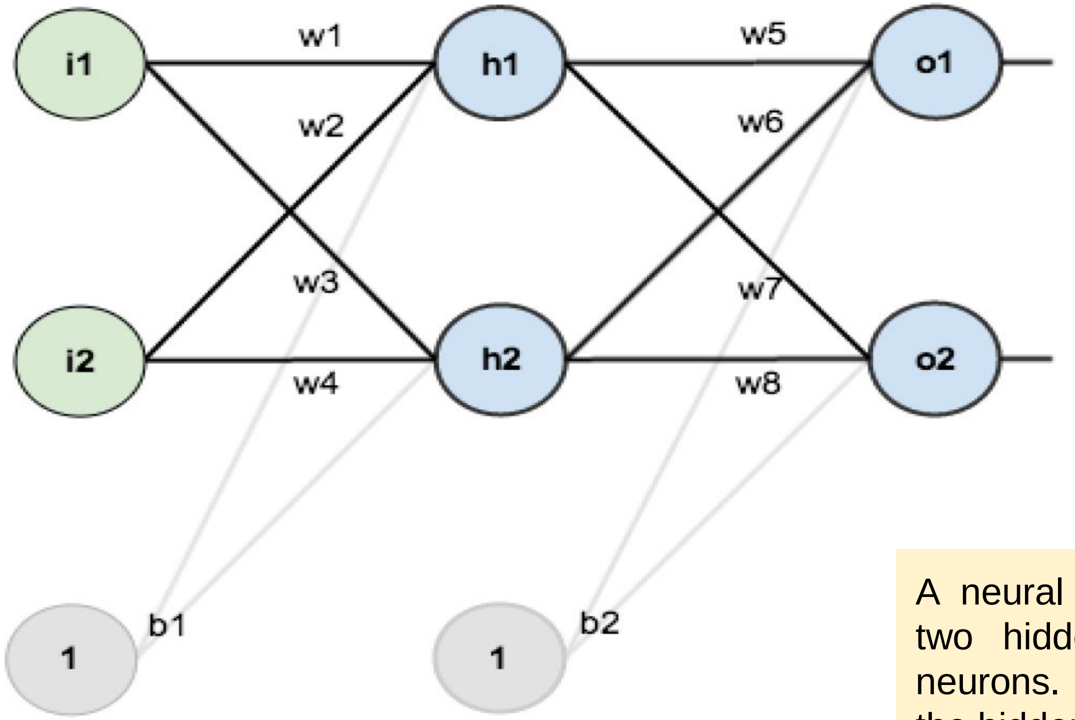
GIVEN




x
y
z
f
"local gradient"
$\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial y}$
$\frac{\partial L}{\partial z}$
$\boxed{\frac{\partial L}{\partial x}} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$
$\boxed{\frac{\partial L}{\partial y}} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}$
Downstream gradients
gradients
(Upsteam)

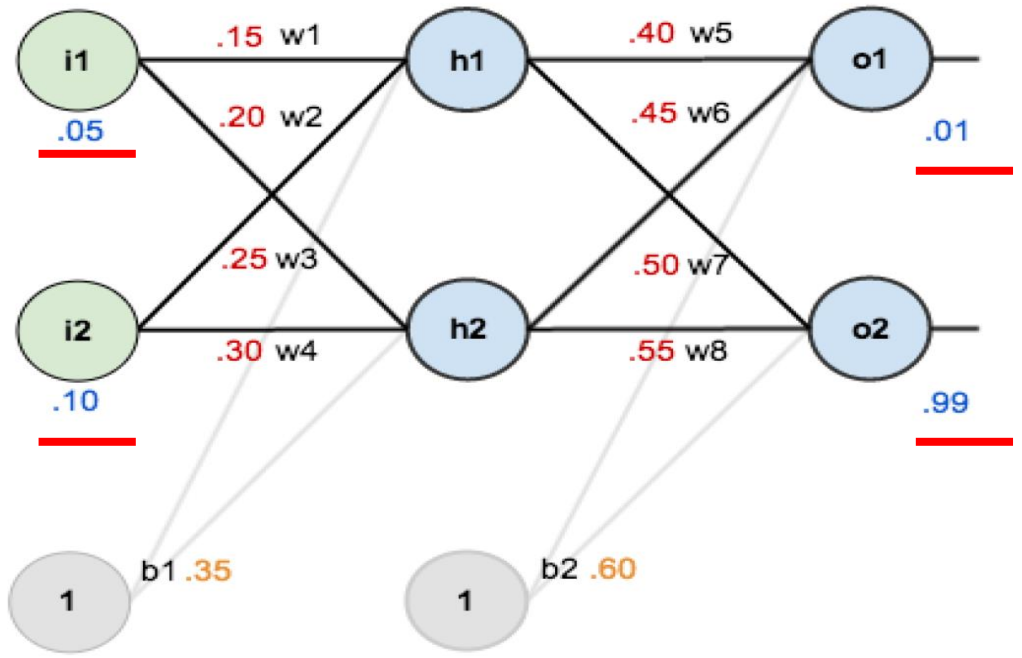
Check next calculations by yourselves!

Example: given neural network



A neural network with two inputs, two hidden neurons, two output neurons. A bias is included in the the hidden and output neurons.

Example: given neural network



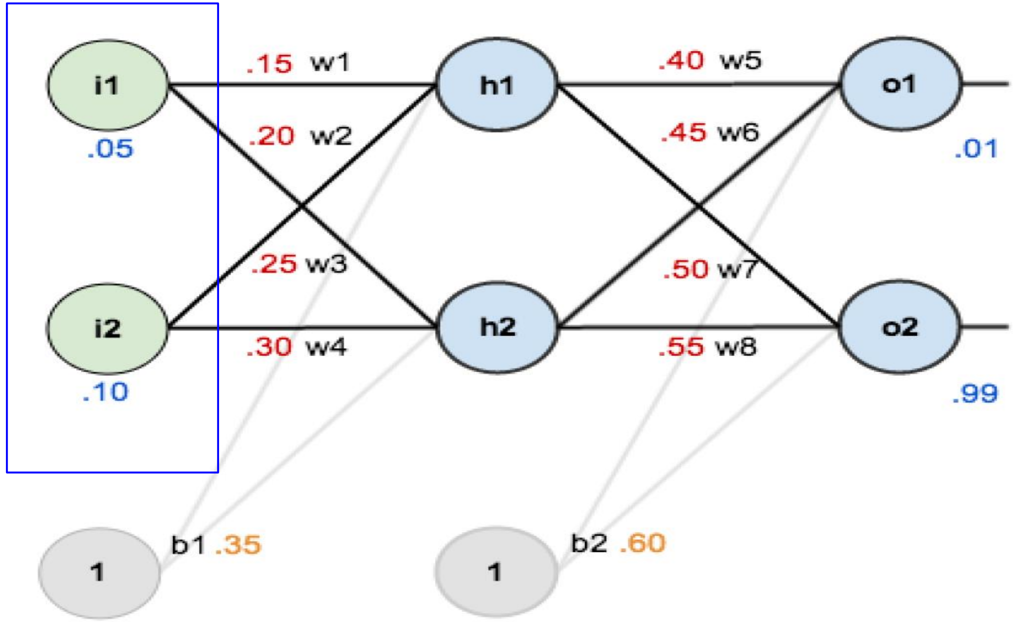
Here are the initial weights, the biases, and training inputs/outputs

For the rest of this example, we're going to work with a single training set. Given,

- Inputs **0.05** and **0.10**
- we want the neural network to output **0.01** and **0.99**.

Next: Feed Forward Pass:

Focus: what the neural network currently predicts given the weights and biases and inputs of 0.05 and 0.10?



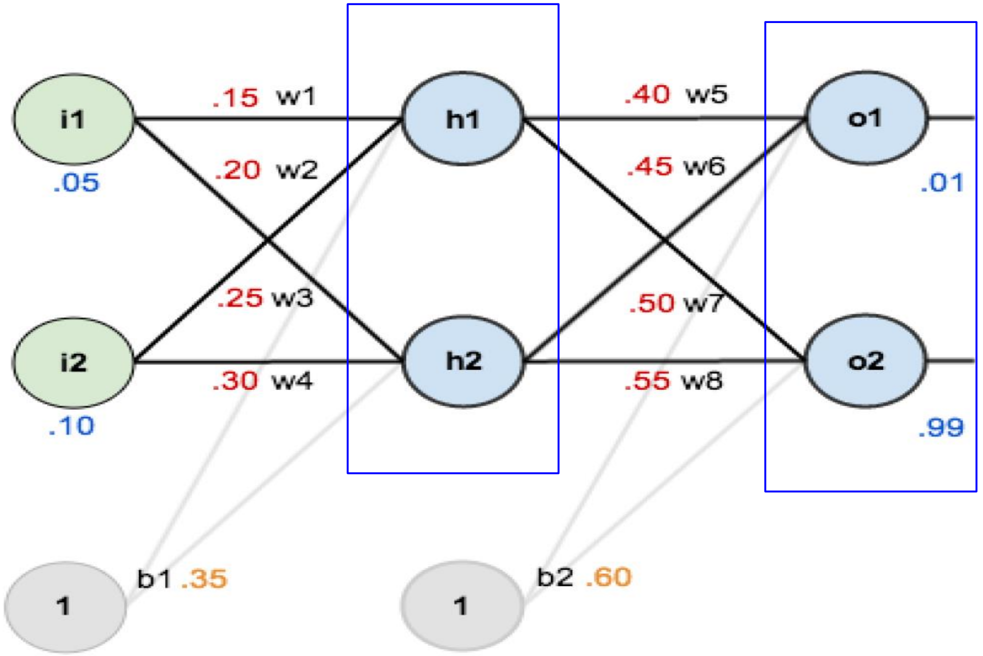
To do this we'll feed those inputs i1,i2 forward though the network.

Feed Forward Pass:

Next:

Step 1: Calculate the total net input to each hidden layer neuron.

$$\text{Total net, } x = \sum_{j=1}^{n} i_j w_j$$



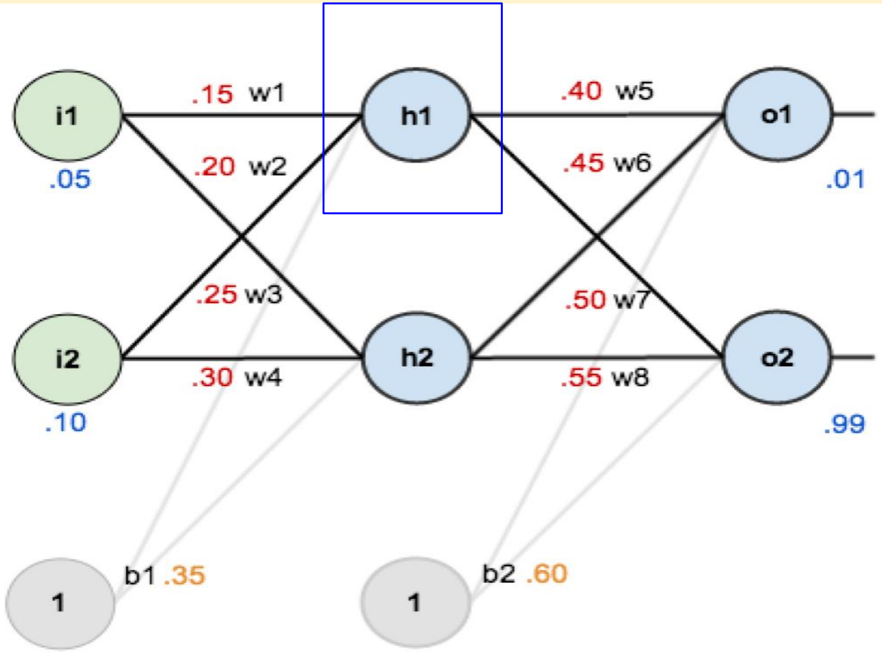
Step 2: Squash the total net input using an activation function (e.g.: use Sigmoid function)

$$\sigma\,(\text{Total net, } x)\text{, where: } \quad \sigma(x) = \frac{1}{1+e^{-x}}$$

Repeat the process with the output layer neurons.

Feed Forward Pass:

Step 1: Calculate the total net input to each hidden layer neuron.



First hidden node h_1:

Step 1: Calculate the total net input

$$net_{h1}=(w1*i1)+(w2*i2)+b1$$
$$= (.15*0.05) +(.20*.10)+(0.35*1)$$
$$= 0.3775$$

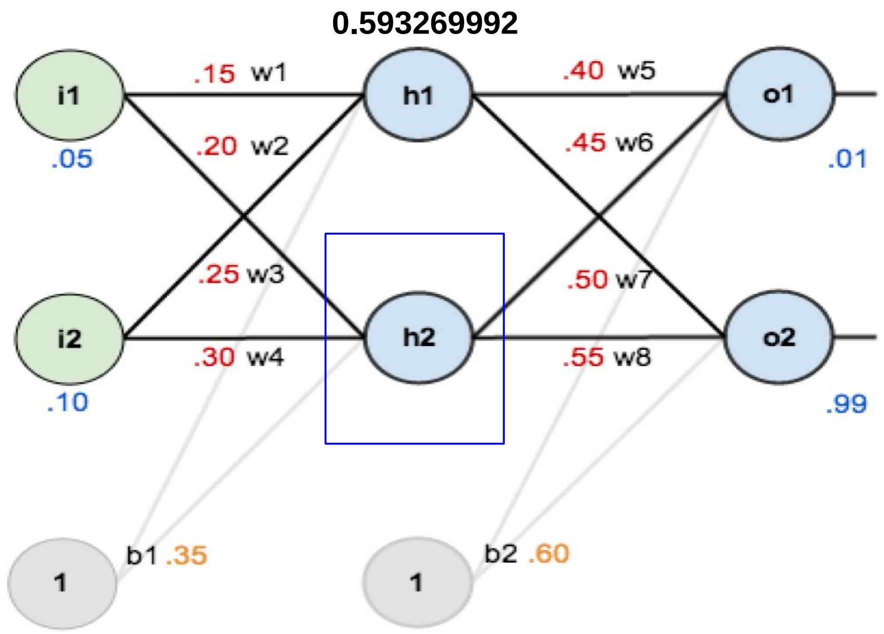
Step 2: Apply Activation Function:

squash it using the Sigmoid function to get the output of h1:

$$out_{h1} = \frac{1}{1+e^{-net_{h1}}} = \frac{1}{1+e^{-0.3775}} = 0.593269992$$

Feed Forward Pass:

Step 1: Calculate the total net input to each hidden layer neuron.



Second hidden node h_2:

Step 1: Calculate the total net input

net_{h2}=(w3*i1)+(w4*i2)+b1

=

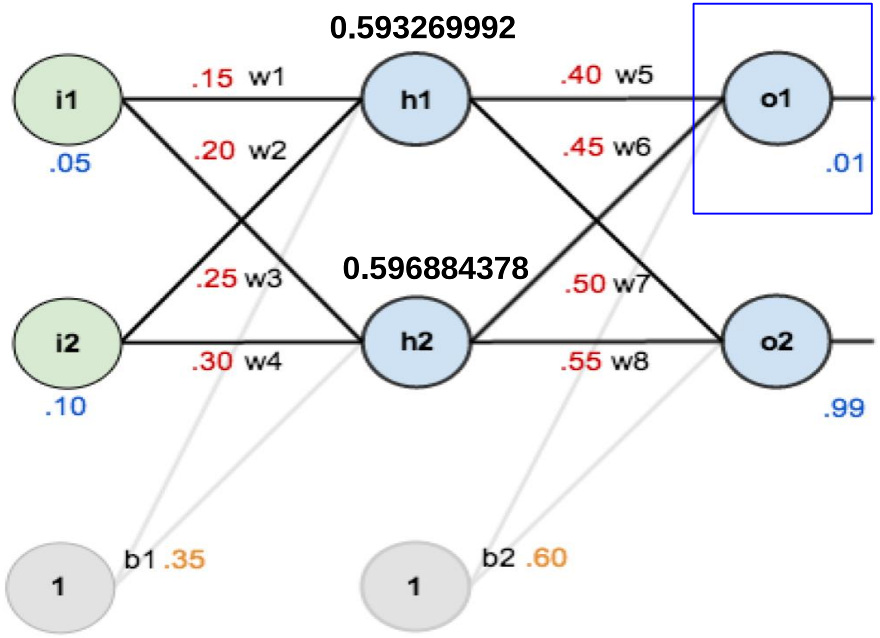
Step 2: Apply Activation Function:

squash it using the Sigmoid function to get the output of h1:

out_{h2}=0.596884378

Feed Forward Pass:

Next: We repeat this process for the output layer neurons, using the output from the hidden layer neurons as inputs.



a. The output for o_1:

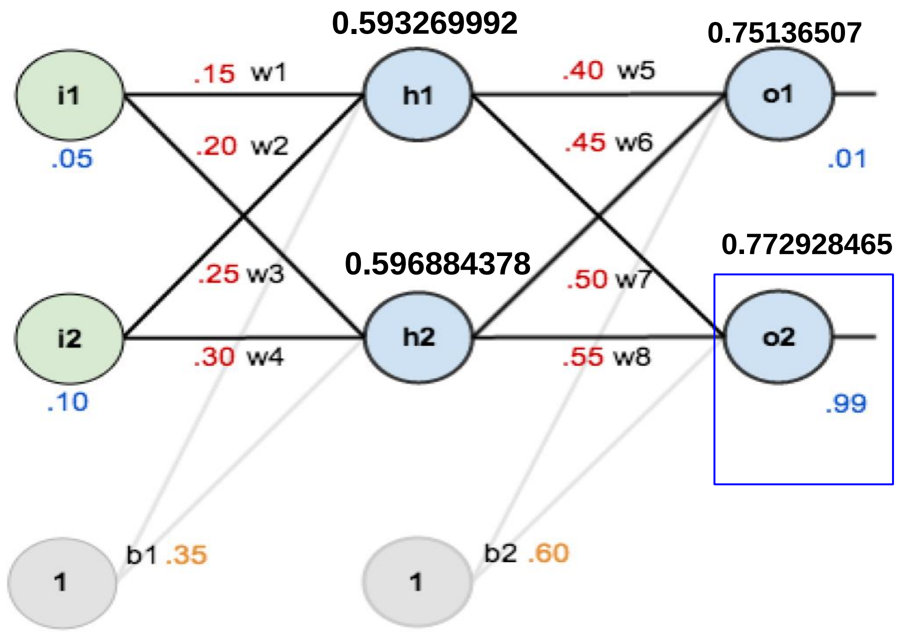
$$net_{o1}=(w5*out_{h1})+(w6*out_{h2})+(b2*1)$$
$$= (.40*0.593269992$$
$$+(.45*0.596884378 +(0.60*1)$$
$$= 1.105905967$$

a. Apply Activation Function:

$$out_{o1} = \frac{1}{1+e^{-net_{o1}}} = \frac{1}{1+e^{-1.105905967}} = 0.75136507$$

Feed Forward Pass:

Next: We repeat this process for the output layer neurons, using the output from the hidden layer neurons as inputs.



a. **The output for o_2:**

$net_{o1}=(w7*out_{h1})+(w8*out_{h2})+(b2*1)$
$=$

a. **Apply Activation Function:**

$out_{o2}=0.772928465$

Next: Calculating the Total Error

We can now calculate the error for each output neuron using the squared error function and sum them to get the total error:

Error = 1/2 (output/predicted -actual/target)^2

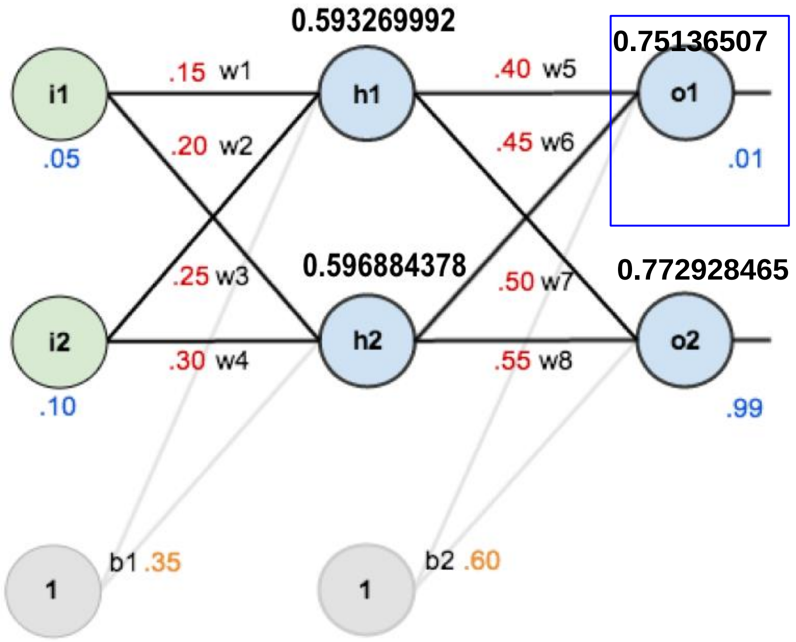
The 1/2 is included so that exponent is cancelled when we differentiate later on. The result is eventually multiplied by a learning rate anyway (Recap: $w_{new} = w_{old} - \alpha \frac{\delta L}{\delta w_i}$) so it doesn’t matter that we introduce a constant here

Next: Calculating the Total Error

- The target output $\mathbf{o_1}$ for is 0.01
- The neural network output out_{o1} is 0.75136507

So, error for $\mathbf{o_1}$ is:

$$
\begin{aligned}
E_{01} &= 1/2\,(\text{output/predicted} - \text{actual/target})^2 \\
&= 1/2\,(0.75136507 - 0.01)^2 \\
&= ½ * 0.549622167 \\
&= 0.274811083
\end{aligned}
$$

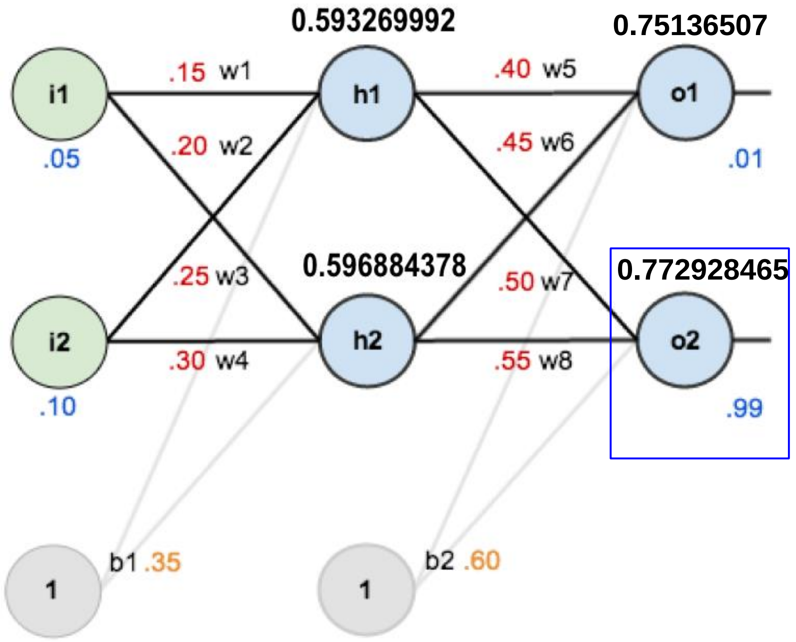


Next: Calculating the Total Error

- Repeating this process for $\mathbf{o_2}$ (remembering that the target is 0.99) we get:

Error for $\mathbf{o_2}$ is, E_{02} : 0.023560026

$= 0.274811083 + 0.023560026$
$= 0.298371109$



Next: Calculating the Total Error

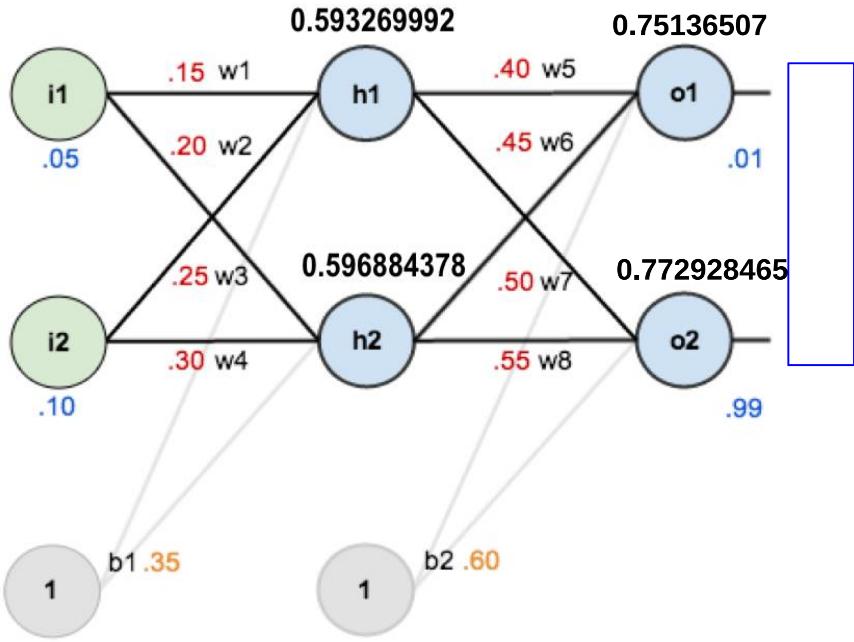
- Repeating this process for **o**$_2$ (remembering that the target is 0.99) we get:

Error for **o**$_2$ is, E_{02} : 0.023560026

Total Error, $E_{Total} = E_{01} + E_{02}$

$= 0.274811083 + 0.023560026$

$= 0.298371109$



Recap: Backward Pass (Back Propagation)

Goal with backpropagation: Update each of the weights in the network

Why? so that they cause the actual output to be closer the target output,

How? Minimize the error for each output neuron and the network as a whole.

- How? The error is then propagated backward through the network.
- Using the chain rule, it calculates the **contribution of each weight** to the overall error.
- The derivative of the error with respect to each weight is computed, starting from w5 to w8, and then w1 to w4, adjusting the weights to minimize the error.

Backpropagation

Next, we will describe how can we calculates the contribution of weight w5 to the overall error (E_{total})

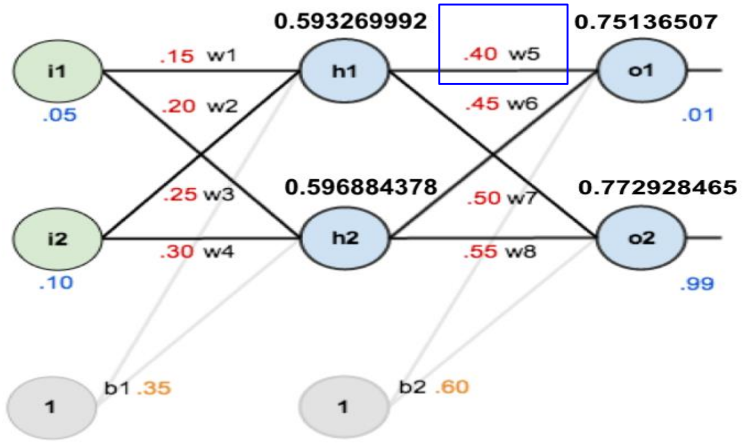
Backward Pass

Output Layer: Consider w5.

We want to know how much a change in **w5** affects the total error E_{Total} : $\frac{\partial E_{total}}{\partial w_5}$

$\frac{\partial E_{total}}{\partial w_5}$ = the partial derivative of E_{Total} with respect to w5.

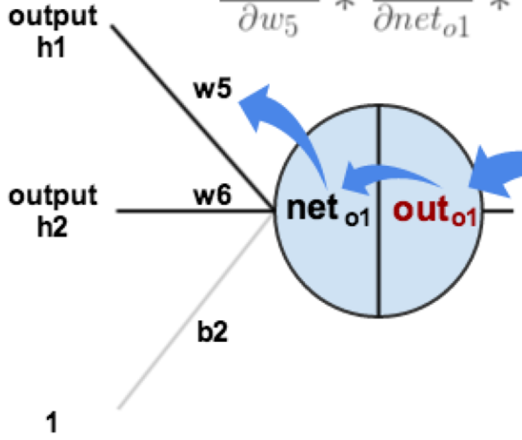
(we can also say the gradient with respect to w5)



By Applying Chain Rule we know

adjust w5 to minimize the total error

$$\frac{\partial net_{o1}}{\partial w_5} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial E_{total}}{\partial out_{o1}} = \frac{\partial E_{total}}{\partial w_5}$$



$$E_{o1} = \tfrac{1}{2}(target_{o1} - out_{o1})^2$$

$$E_{total} = E_{o1} + E_{o2}$$

In the backpropagation process, working backward involves understanding how to start from the total error (E_{total}) and trace back to the weight w5.

- For instance, to determine E_{total}, , we need to know the output o1.
- This output o1 is influenced by the weighted sum net o1 ,
- and one of the contributing weights to this neto1 calculation is w5.

Therefore, the chain of dependencies follows: $E_{total} \rightarrow$ o1 $\rightarrow$ net01 $\rightarrow$ w5

We need to figure out each piece in this equation.

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

After we computed its respective gradient ($\frac{\partial E_{total}}{\partial w_5}$ we update the weight w5 using the gradient descent formula:

$$w_{new} = w_{old} - \alpha \frac{\delta L}{\delta w_i} \quad = W_{5new} = w_5 - \alpha \frac{\partial E_{total}}{\partial w_5}$$

Same process is applicable for other subsequent weights.

Additional (If you want to see more detailed calculation)

https://hmkcode.com/ai/backpropagation-step-by-step/

Summary: Neural Networks

- Multi-layer, feed-forward networks using backpropagation make decent classifiers.
- They do especially well for unstructured data, like images
- Somewhat finicky--learning rate can be hard to tune, and the number of hidden nodes can affect the output.
- Next class, we'll learn about how they've been adapted to modern day!