Perceptron

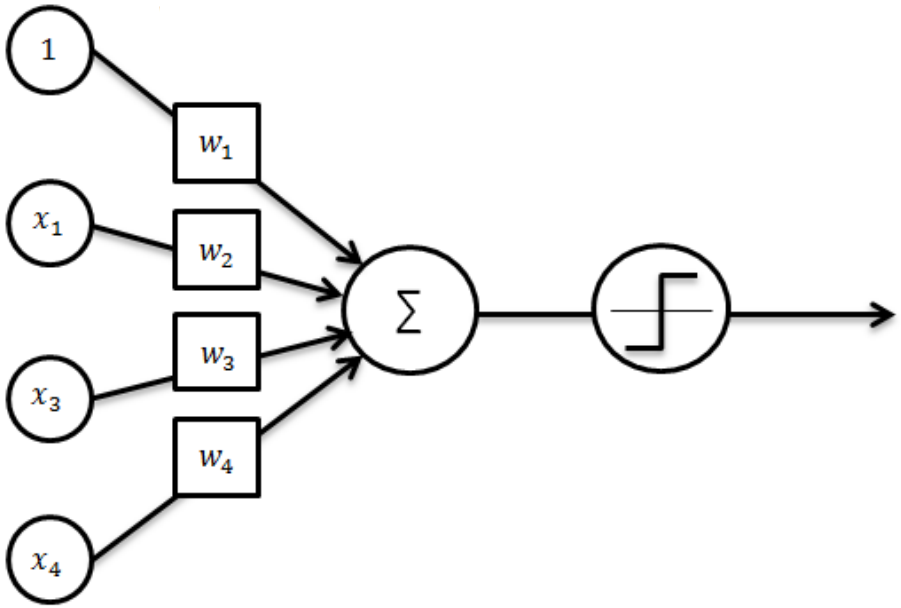


Image source

Perceptron

Perceptron *is a single layer neural network.*

Perceptron is a linear classifier (binary). Also, it is used in supervised learning.

It helps to classify the given input data.

The perceptron consists of 4 parts.

- Input values or One input layer
- Weights and Bias
- Net sum
- Activation Function

The Neural Networks work the same way as the perceptron.

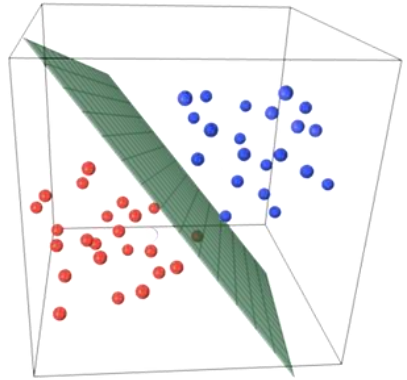
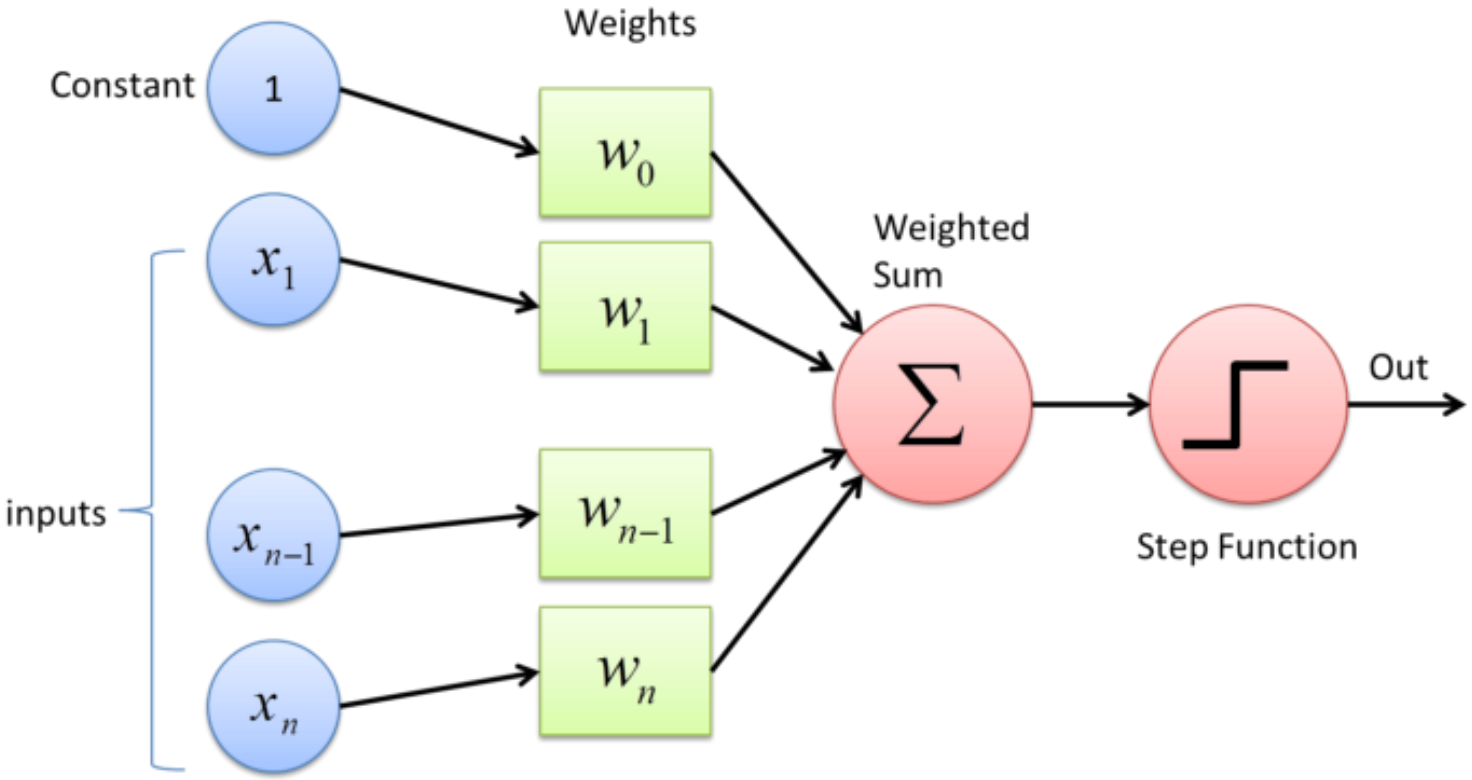
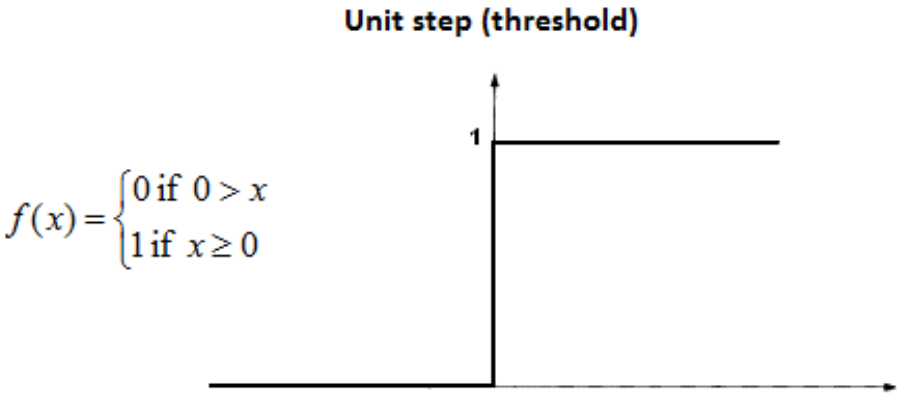


Image source

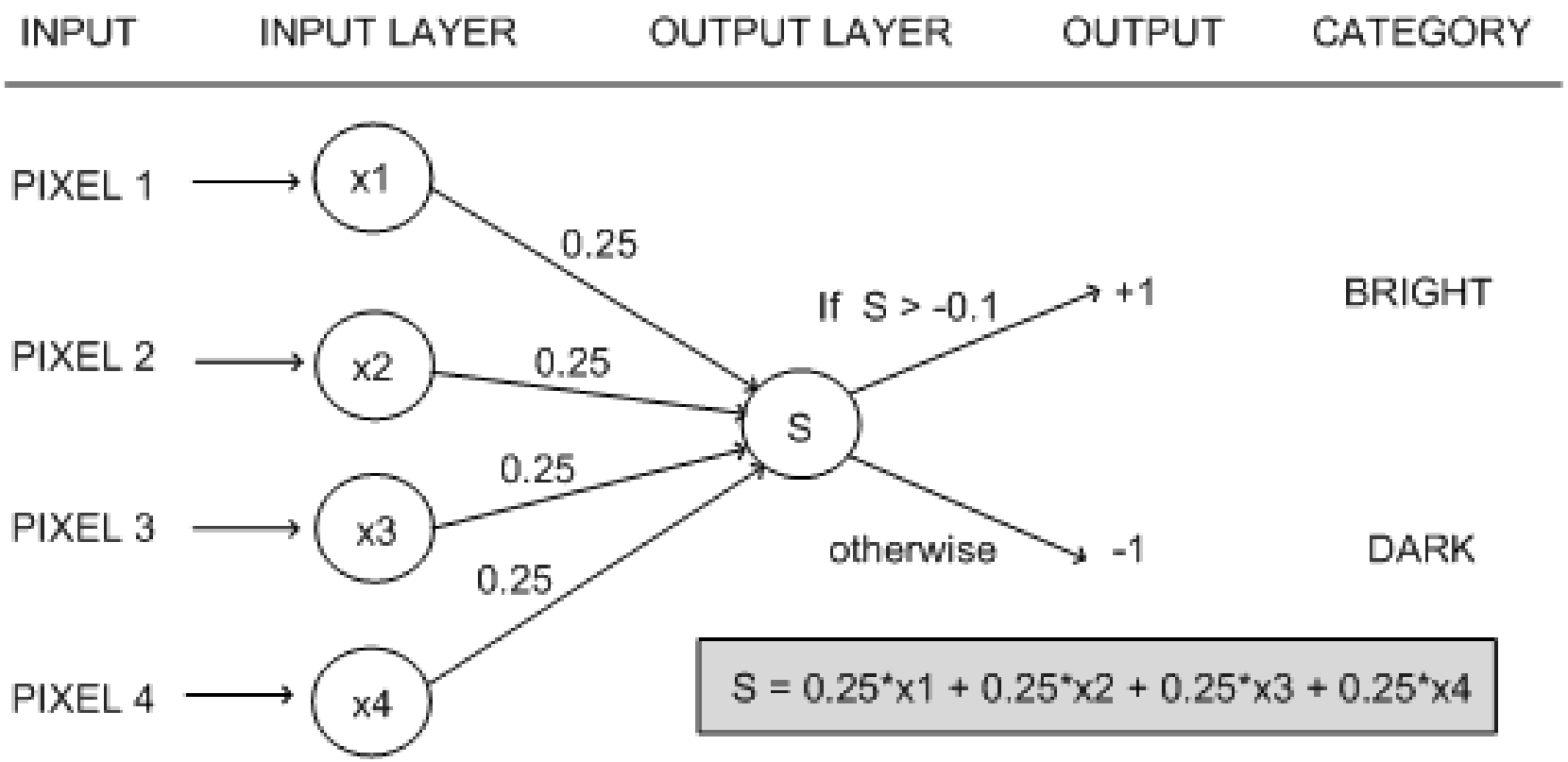
Formalization





https://towardsdatascience.com/what-the-hell-is-perceptron-

Formalization



https://towardsdatascience.com/what-the-hell-is-perceptron-626217814f53

Activation Function

In short, the activation functions are used to map the input between the required values like (0, 1) or (-1, 1).

It’s just a thing function that you use to get the output of node. It is also known as ***Transfer Function***.

The Activation Functions can be basically divided into 2 types-

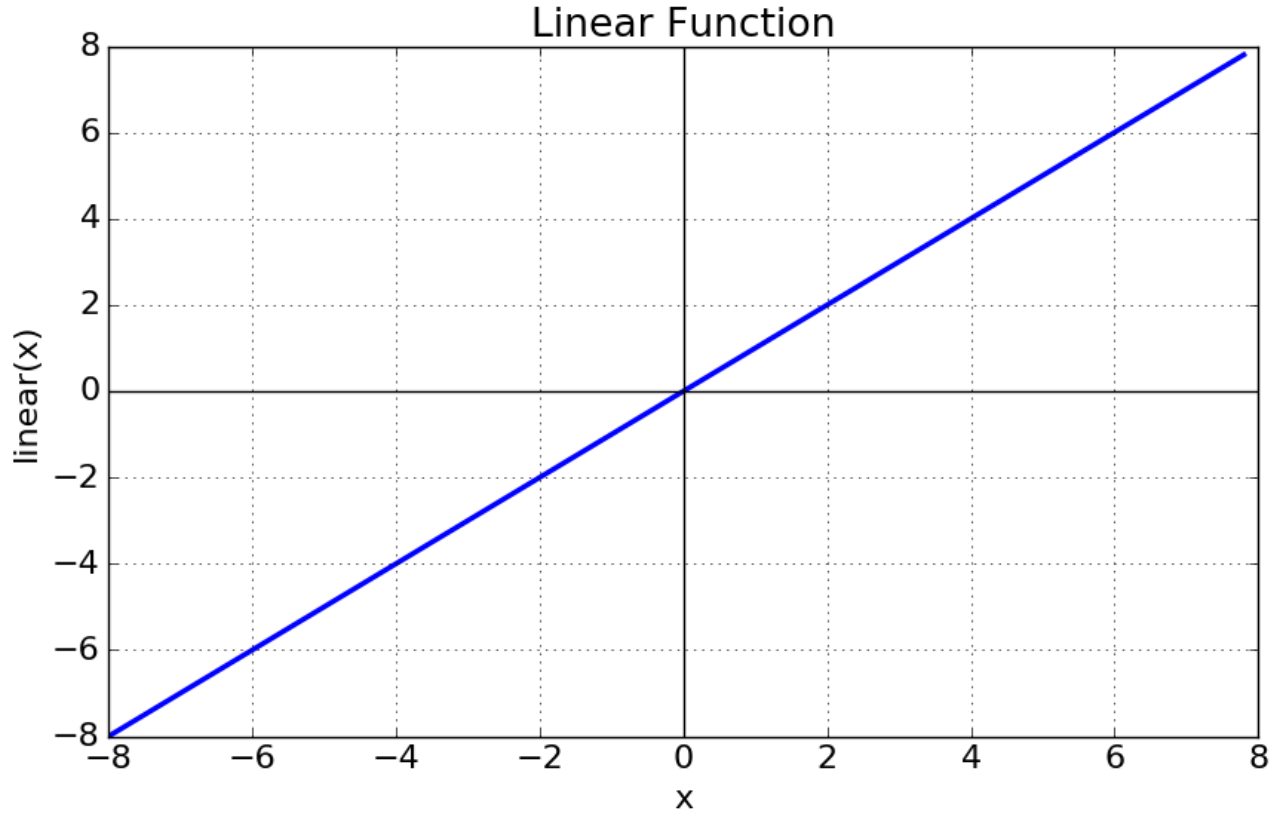
- Linear Activation Function
- Non-linear Activation Functions

Linear or Identity Activation Function

Equation : f(x) = x

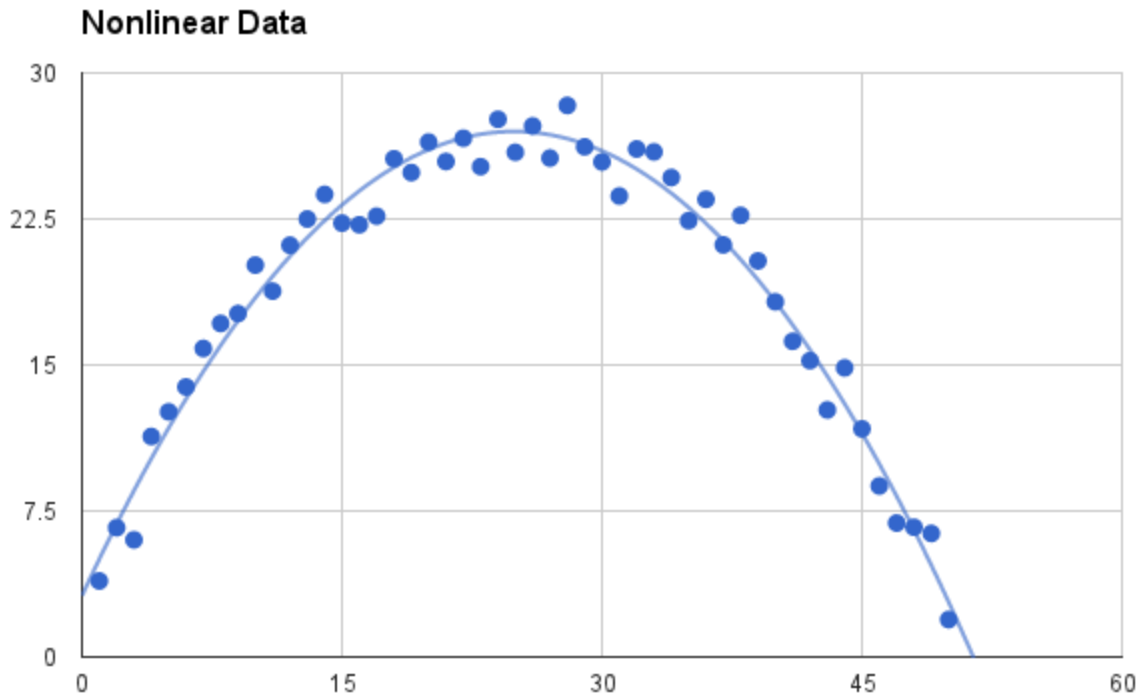
Range : (-infinity to infinity)

It doesn’t help with the complexity or various parameters of usual data that is fed to the neural networks.



Non-linear Activation Function

The Nonlinear Activation Functions are the most used activation functions.



It makes it easy for the model to generalize or adapt with variety of data and to differentiate between the output.

Non-linear Activation Function

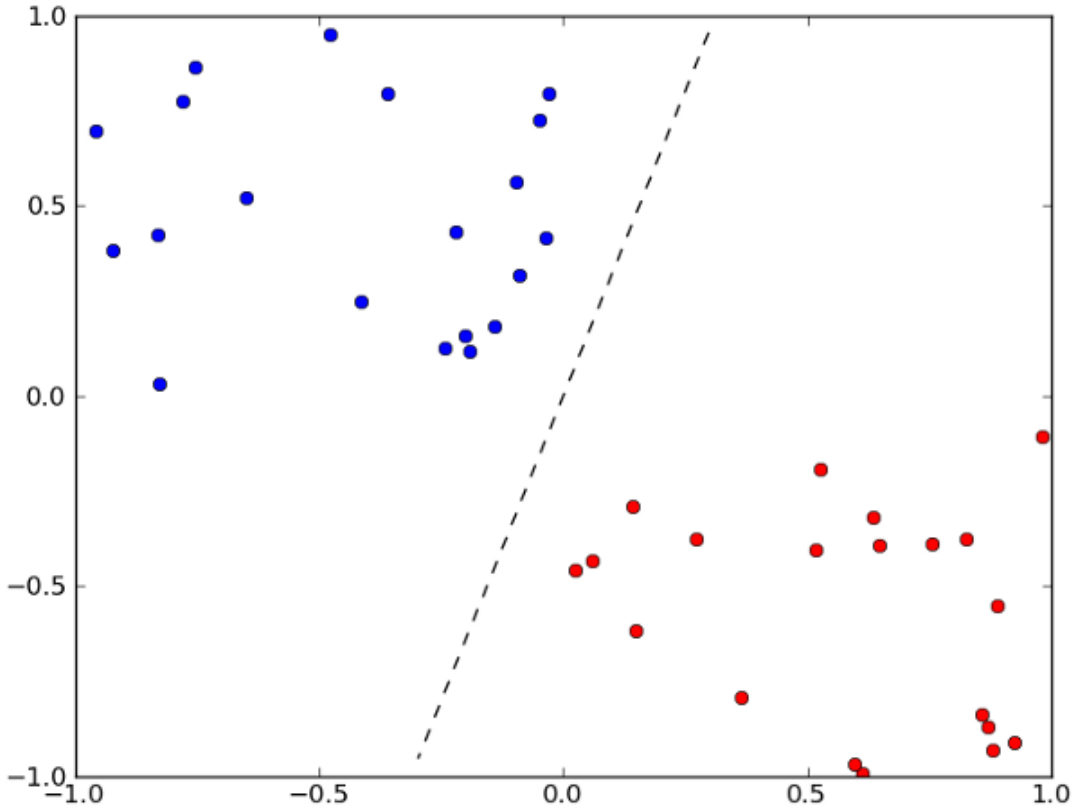
The main terminologies needed to understand for nonlinear functions are:

- Derivative or Differential: Change in y-axis w.r.t. change in x-axis. It is also known as slope.
- Monotonic function: A function which is either entirely non-increasing or non-decreasing.

The Nonlinear Activation Functions are mainly divided on the basis of their **range or curves**-

- **Sigmoid or Logistic Activation Function**
- **Tanh or hyperbolic tangent Activation Function**
- **ReLU (Rectified Linear Unit) Activation Function**

Perceptron is usually used to classify the data into two parts. Therefore, it is also known as a Linear Binary Classifier.

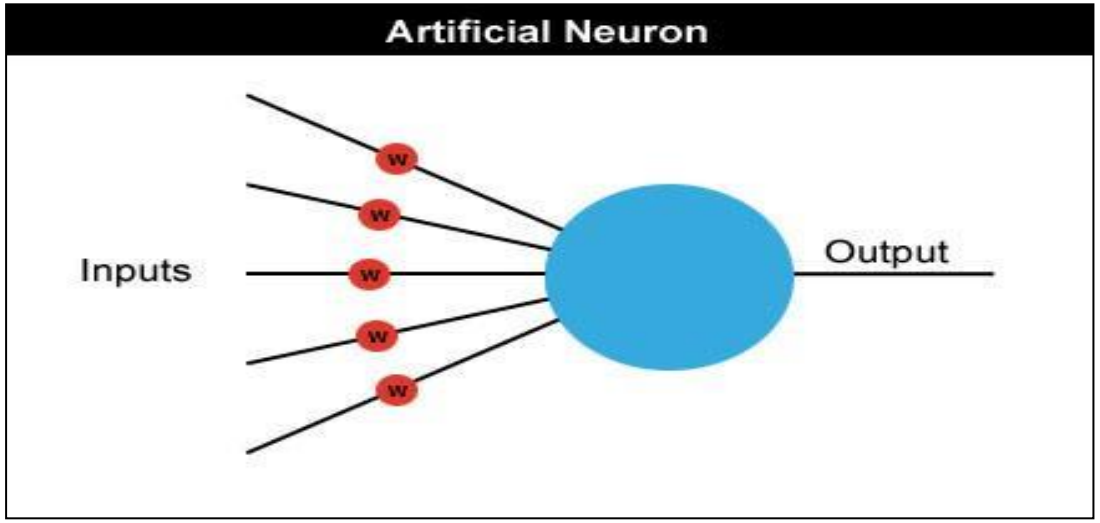


Neural networks

Neural networks are made up of many artificial neurons.

Each input into the neuron has its own weight associated with it illustrated by the red circle.

A weight is simply a floating point number and it's these we adjust when we eventually come to train the network.



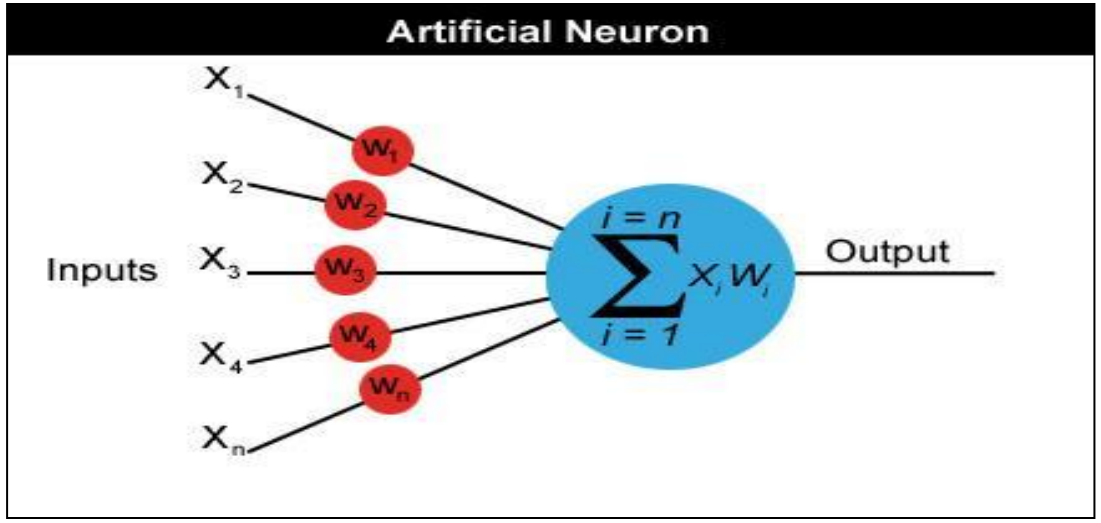
Neural networks

A neuron can have any number of inputs from one to n, where n is the total number of inputs.

The inputs may be represented therefore as x_1, x_2, x_3… x_n.

And the corresponding weights for the inputs as w_1, w_2, w_3… w_n.

Output $a = x_1w_1+x_2w_2+x_3w_3… +x_nw_n$

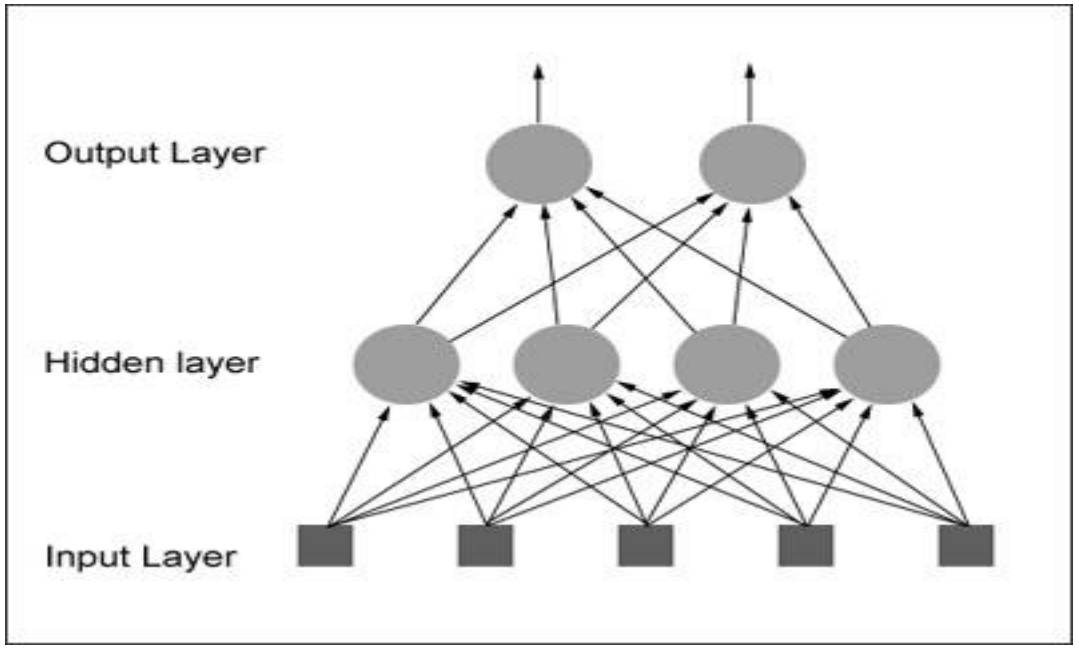


How do we actually *use* an artificial neuron?

feedforward network: The neurons in each layer feed their output forward to the next layer until we get the final output from the neural network.

There can be any number of hidden layers within a feedforward network.

The number of neurons can be completely arbitrary.



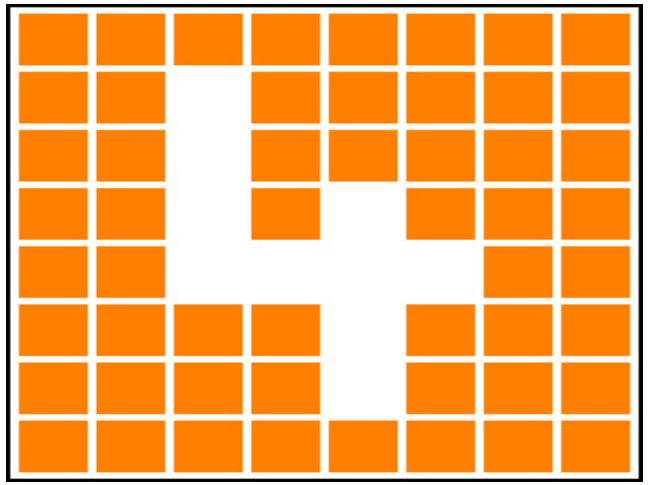
Neural Networks by an Example

let's design a neural network that will detect the number '4'.

Given a panel made up of a grid of lights which can be either on or off, we want our neural net to let us know whenever it thinks it sees the character '4'.

The panel is eight cells square and looks like this:

the neural net will have 64 inputs, each one representing a particular cell in the panel and a hidden layer consisting of a number of neurons (more on this later) all feeding their output into just one neuron in the output layer

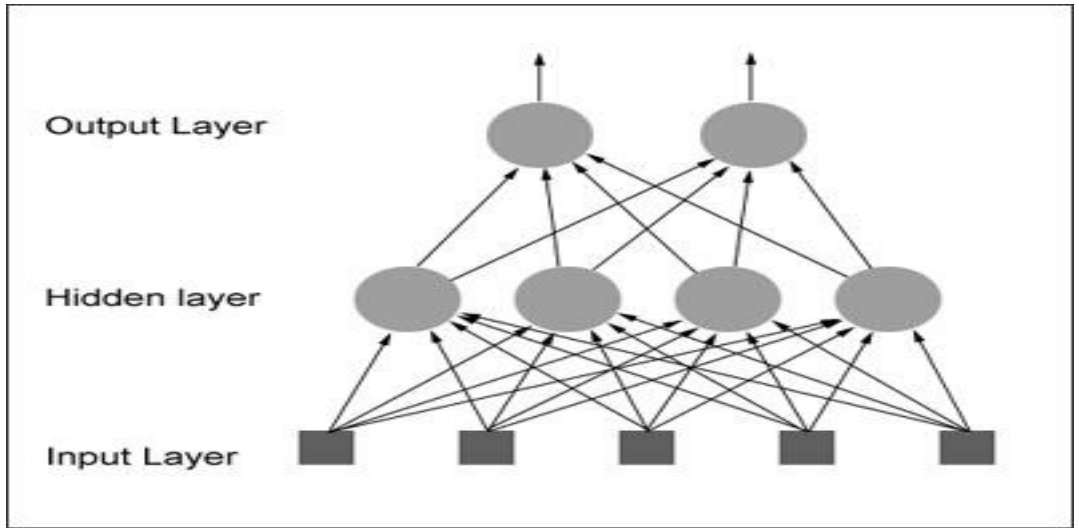


Neural Networks by an Example

initialize the neural net with random weights

feed it a series of inputs which represent, in this example, the different panel configurations

For each configuration we check to see what its output is and adjust the weights accordingly so that whenever it sees something looking like a number 4 it outputs a 1 and for everything else it outputs a zero.

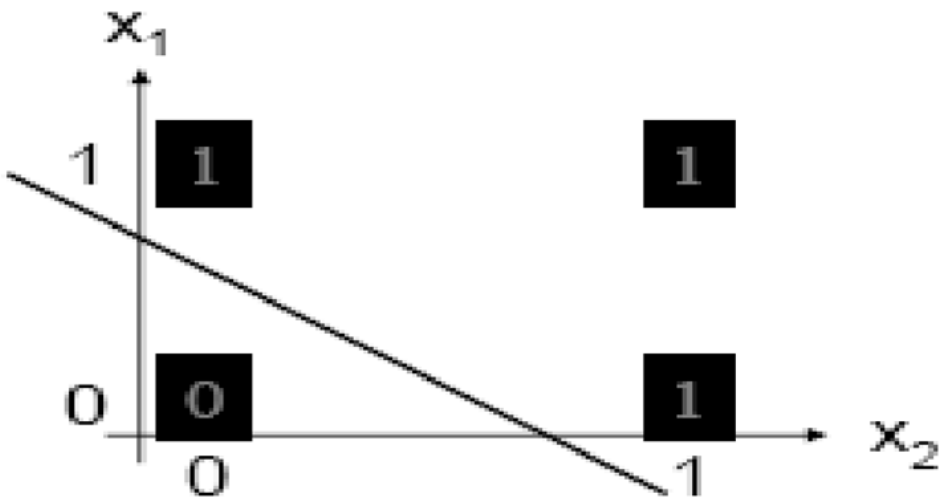


Perceptron Learning Theorem

Recap: A perceptron (threshold unit) can *learn* anything that it can *represent* (i.e. anything separable with a hyperplane)

OR function

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

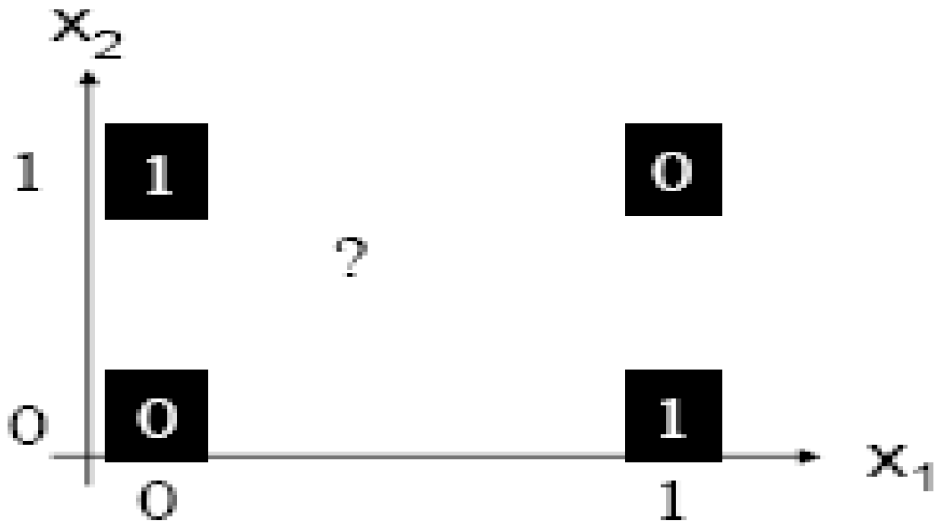


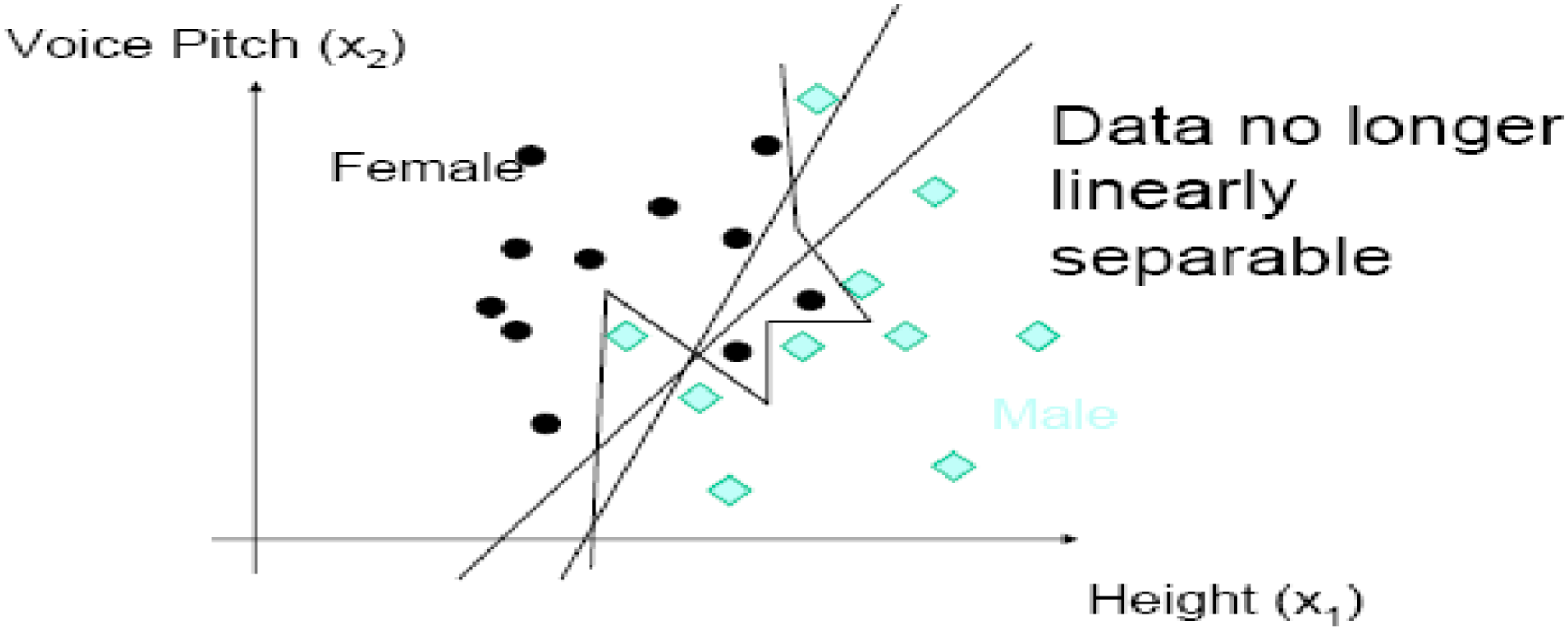
The Exclusive OR problem

A Perceptron cannot represent Exclusive OR since it is not linearly separable.

XOR function

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

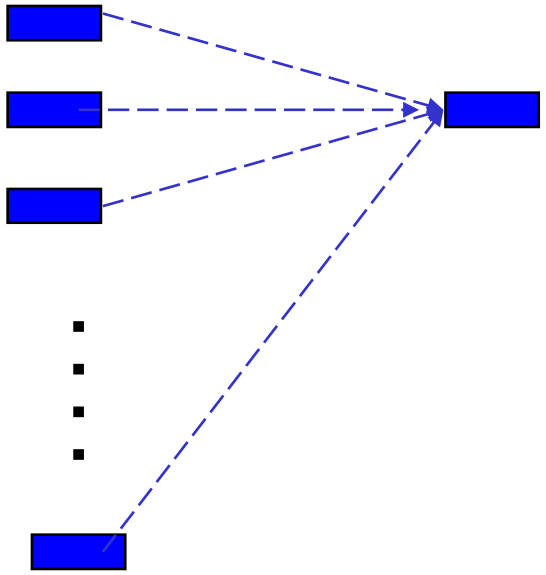




What is a good decision boundary ?

Properties of NN architecture

- No connections within a layer
- No direct connections between input and output layers
- Fully connected between layers
- Often more than 3 layers
- Number of output units need not equal number of input units
- Number of hidden units per layer can be more or less than input or output units



Each unit is a perceptron

Often include bias as an extra weight

Perceptron training algorithm

- Initialize weights randomly
- Cycle through training examples in multiple passes (*epochs*)
- For each training example (x_i, y_i):
- If current prediction $\text{sgn}(w^T x_i)$ does not match y_i then update weights:

$$w \leftarrow w + \eta\, y_i x_i$$

where η is a *learning rate* that should decay slowly* over time

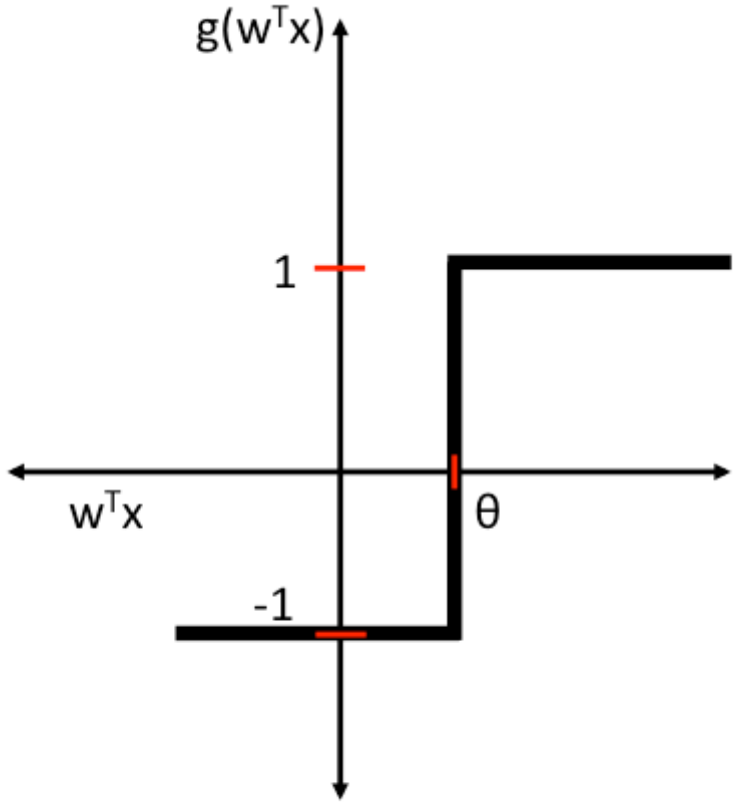
$$g(\mathbf{z}) = \begin{cases} 1 & \text{if } \mathbf{z} \geq \theta \\ -1 & \text{otherwise.} \end{cases}$$

where

$$\mathbf{z} = w_1 x_1 + \cdots + w_m x_m = \sum_{j=1}^{m} x_j w_j$$
$$= \mathbf{w}^T \mathbf{x}$$

$\mathbf{w}$ is the feature (weight) vector, and $\mathbf{x}$ is an m-dimensional sample from the training dataset:

$$\mathbf{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$$



Unit step function.

Perceptron Learning Rule

Rosenblatt's initial perceptron rule is fairly simple and can be summarized by the following steps:

Initialize the weights to 0 or small random numbers.

For each training sample x(i):

- Calculate the *output* value.
- Update the weights.

- The output value is the class label predicted by the unit step function that we defined earlier (output =g(z))
- the weight update can be written more formally as $w_j := w_j + \Delta w_j$

$$\Delta w_j = \eta \, (\text{target}^{(i)} - \text{output}^{(i)}) \, x_j^{(i)}$$

where η is the learning rate (a constant between 0.0 and 1.0), "target" is the true class label, and the "output" is the predicted class label.

It is important to note that all weights in the weight vector are being updated $\Delta w_0 = \eta(\text{target}^{(i)} - \text{output}^{(i)})$ **onal dataset, we w**

$$\Delta w_1 = \eta(\text{target}^{(i)} - \text{output}^{(i)}) \, x_1^{(i)}$$

$$\Delta w_2 = \eta(\text{target}^{(i)} - \text{output}^{(i)}) \, x_2^{(i)}$$

In the two scenarios where the perceptron predicts the class label correctly, the weights remain unchanged:

$$\Delta w_j = \eta(-1^{(i)} - -1^{(i)})\ x_j^{(i)} = 0$$
$$\Delta w_j = \eta(1^{(i)} - 1^{(i)})\ x_j^{(i)} = 0$$

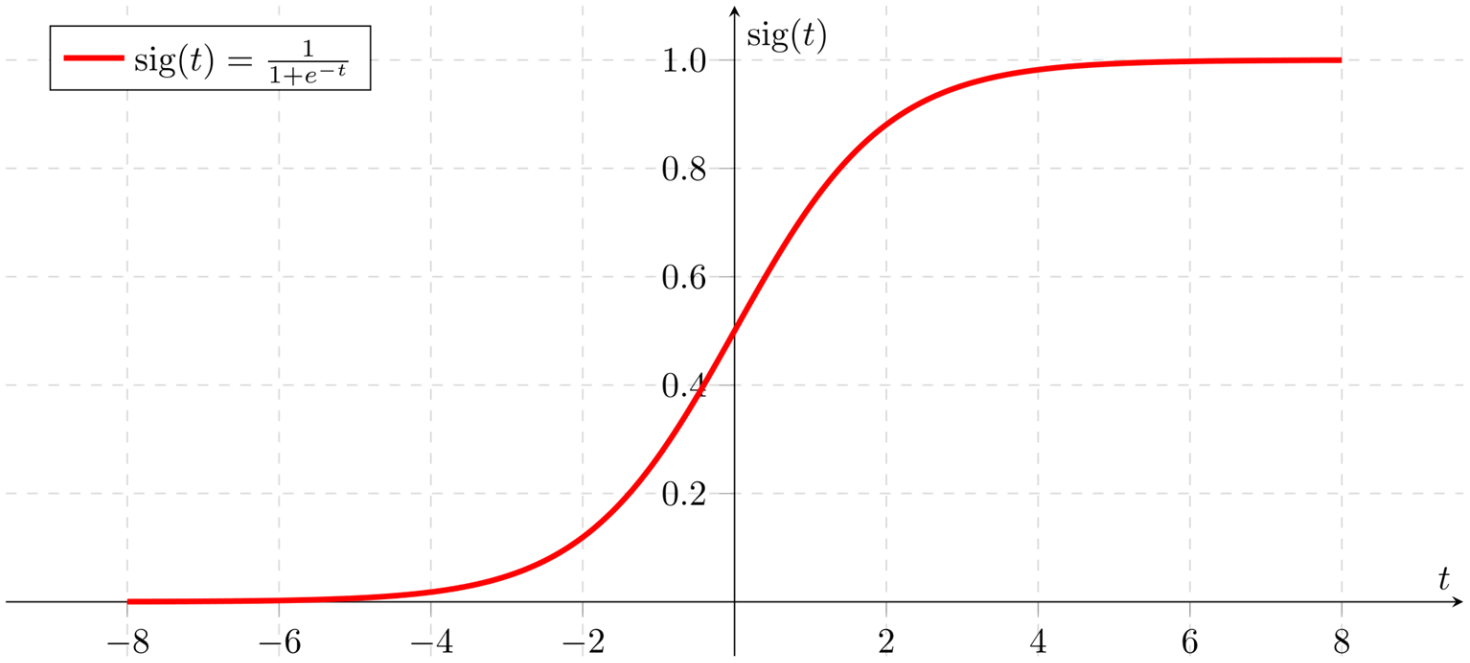
However, in case of a wrong prediction, the weights are being "pushed" towards the direction of the positive or negative target class, respectively:

$$\Delta w_j = \eta(1^{(i)} - -1^{(i)})\ x_j^{(i)} = \eta(2)\ x_j^{(i)}$$
$$\Delta w_j = \eta(-1^{(i)} - 1^{(i)})\ x_j^{(i)} = \eta(-2)\ x_j^{(i)}$$

Sigmoid function

- Squash the linear response of the classifier to the interval [0,1]:

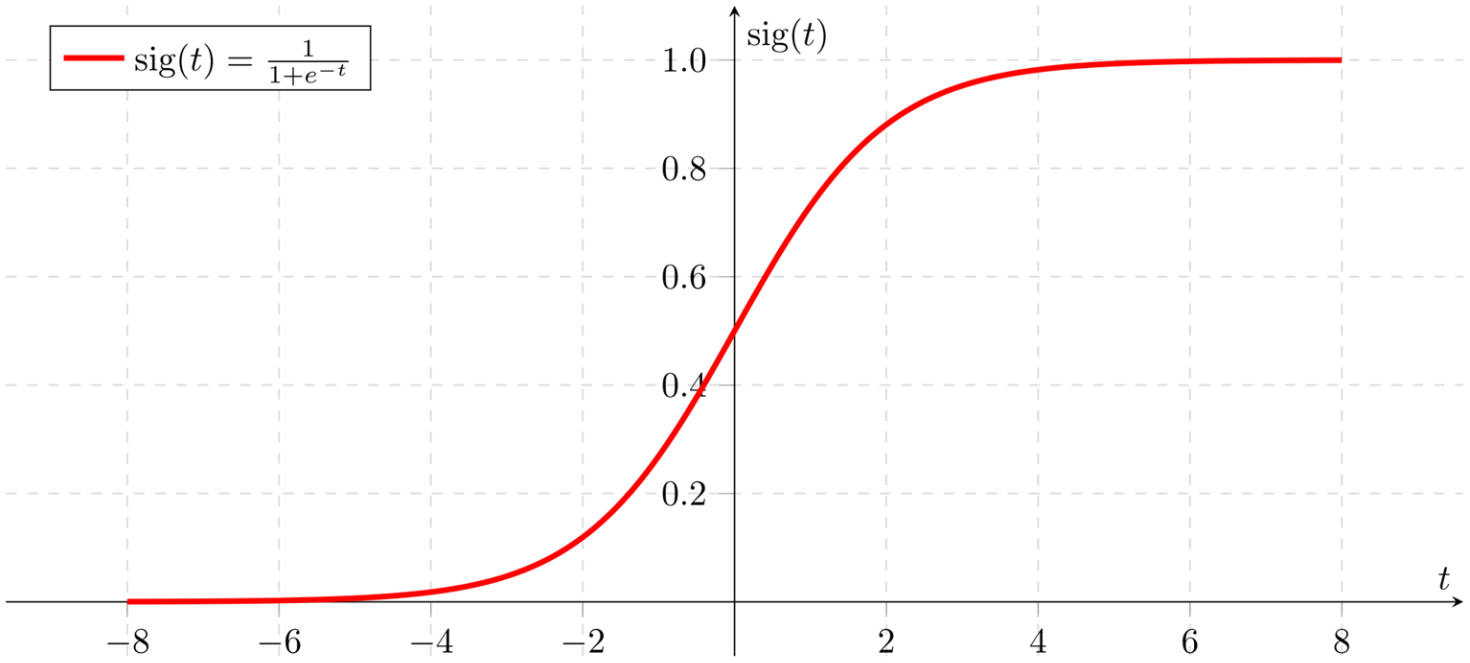
$$\sigma(w^T x) = \frac{1}{1 + \exp(-w^T x)}$$



Sigmoid function

- Output of sigmoid can be interpreted as posterior label probability or confidence returned by classifier:

$$P_w(y = 1|x) = \sigma(w^T x) = \frac{1}{1+\exp(-w^T x)}$$

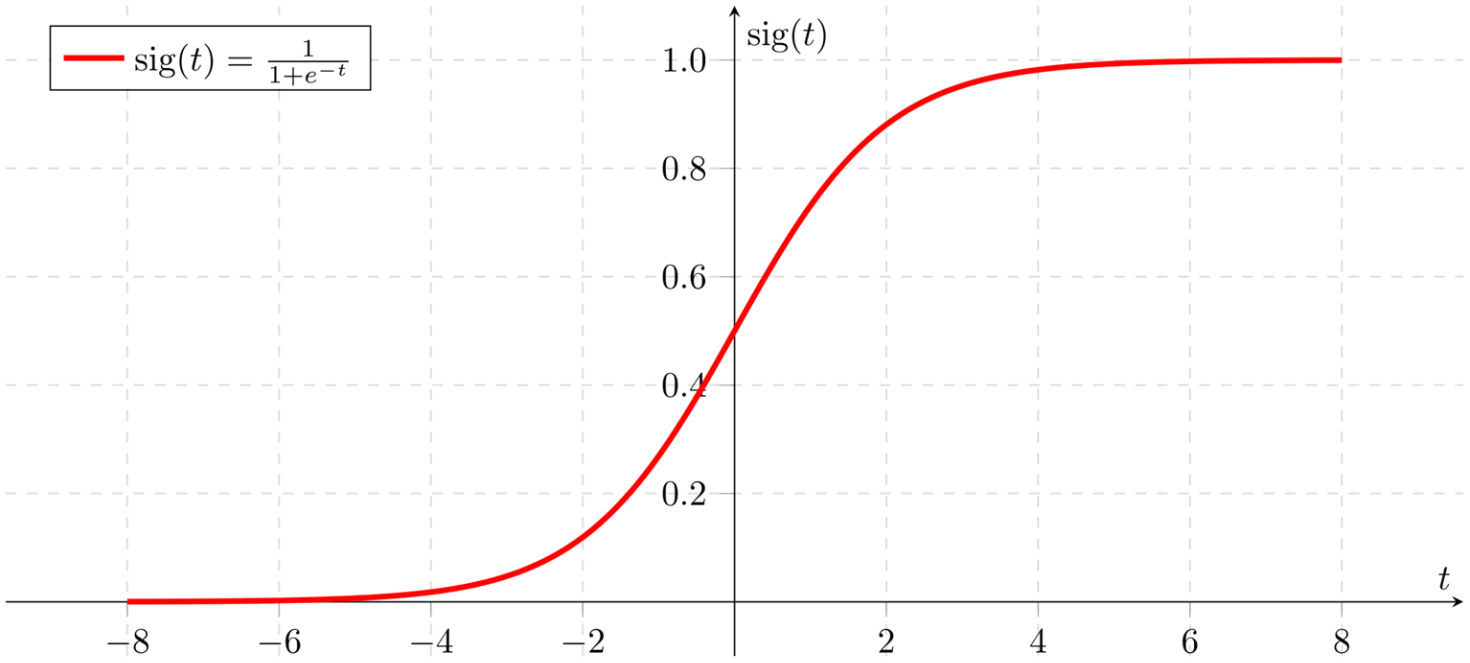


Sigmoid function

- Output of sigmoid can be interpreted as posterior label probability or confidence returned by classifier:

$$P_w(y = 1|x) = \sigma(w^T x) = \frac{1}{1+\exp(-w^T x)}$$

- Sigmoid is *symmetric*: $1 - \sigma(t) = \sigma(-t)$

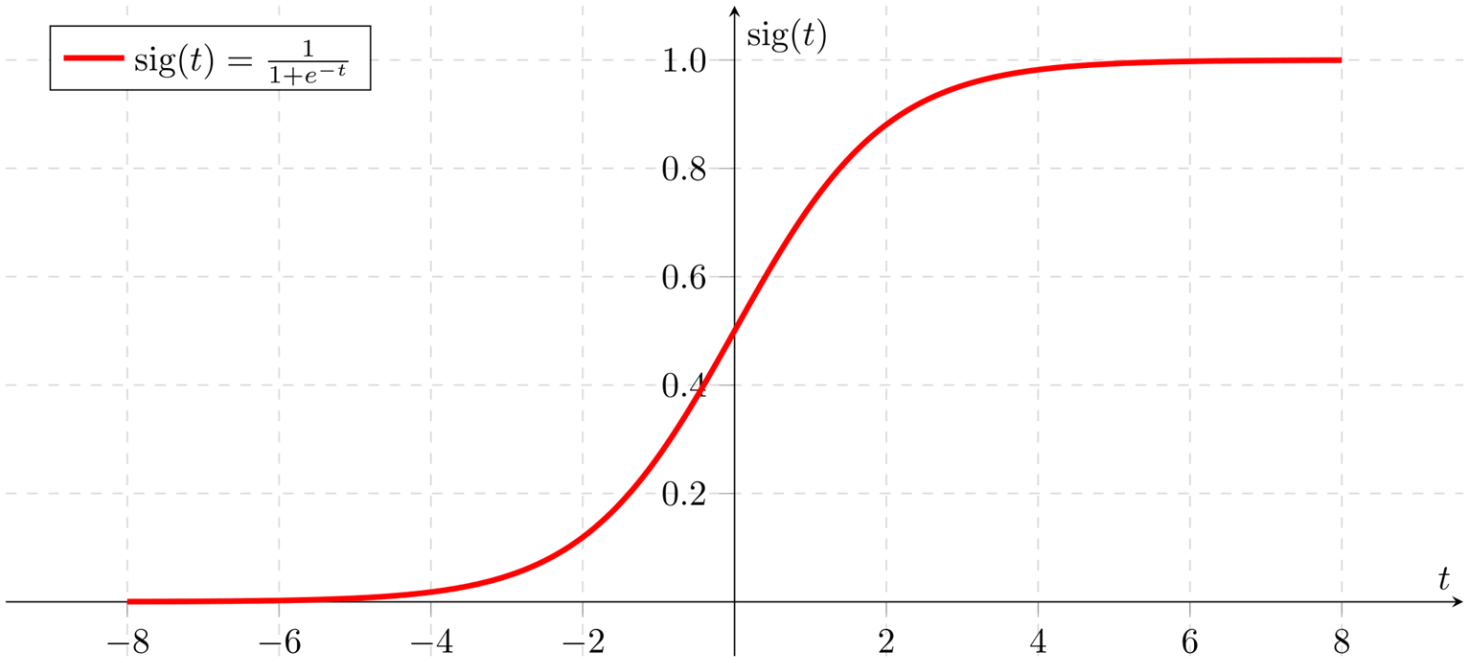


Sigmoid function

- Output of sigmoid can be interpreted as posterior label probability or confidence returned by classifier:

$$P_w(y = 1|x) = \sigma(w^T x) = \frac{1}{1+\exp(-w^T x)}$$

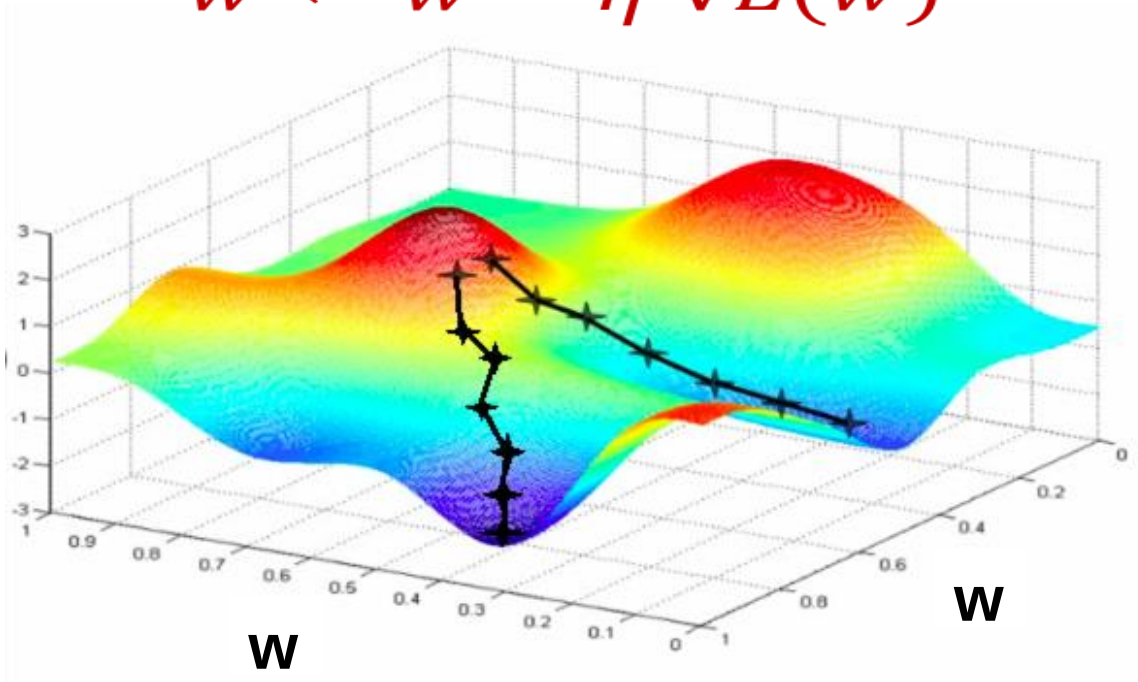
- What happens if we scale w by a constant?



Gradient descent

- Goal: find w to minimize loss $\hat{L}(w)$
- Start with some initial estimate of w
- At each step, find $\nabla\hat{L}(w)$, the *gradient* of the loss w.r.t. w, and take a small step in the *opposite* direction

$$w \leftarrow w - \eta\, \nabla\hat{L}(w)$$



Stochastic gradient descent (SGD)

- At each iteration, take a single data point (x_i, y_i) and perform a parameter update using $\nabla l(w, x_i, y_i)$, the gradient of the loss for that point:

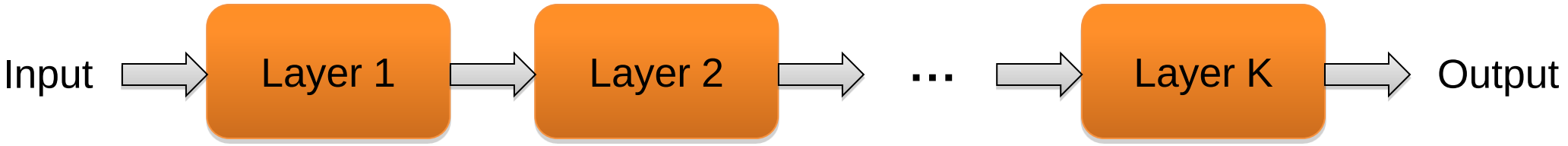
$$w \leftarrow w - \eta \nabla l(w, x_i, y_i)$$

- This is called an *online* or *stochastic* update

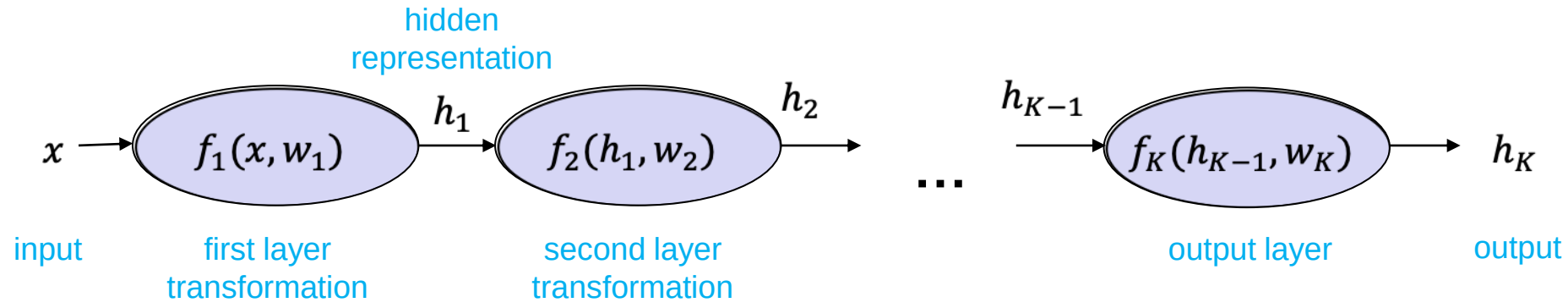
How to train a multi-layer network?

Multi-layer neural networks

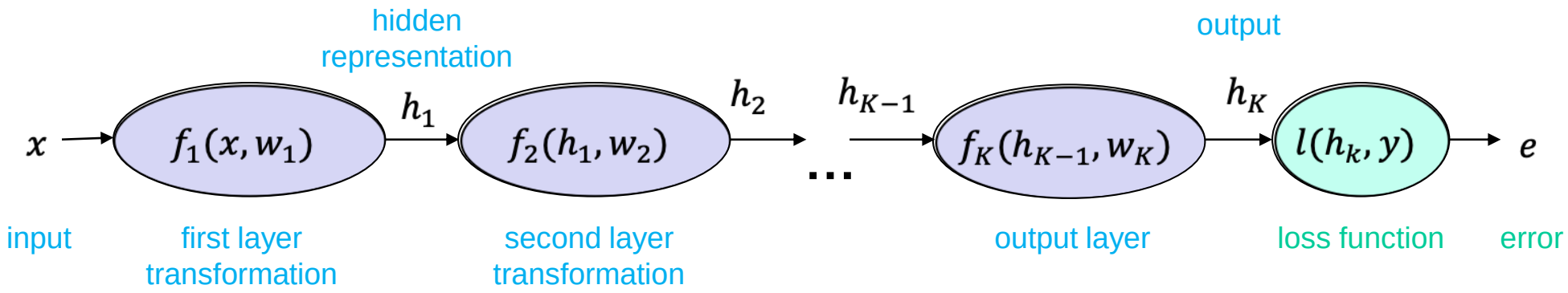
- The function computed by the network is a composition of the functions computed by individual layers (e.g., linear layers and nonlinearities):



- More precisely:



Training a multi-layer network

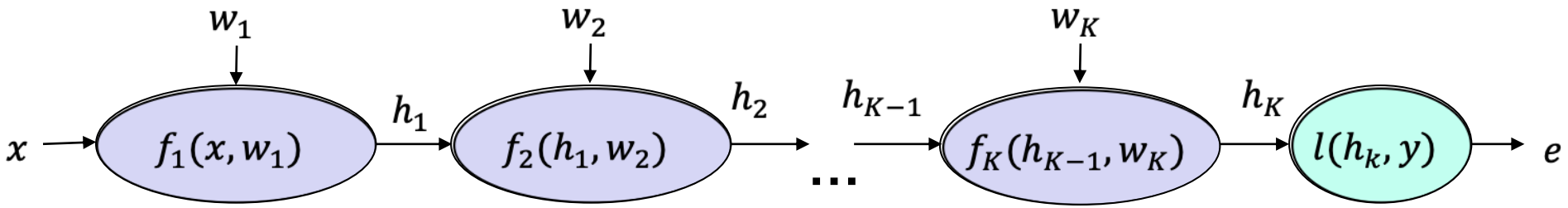


- What is the SGD update for the parameters w_k of the kth layer?

$$w_k \leftarrow w_k - \eta \frac{\partial e}{\partial w_k}$$

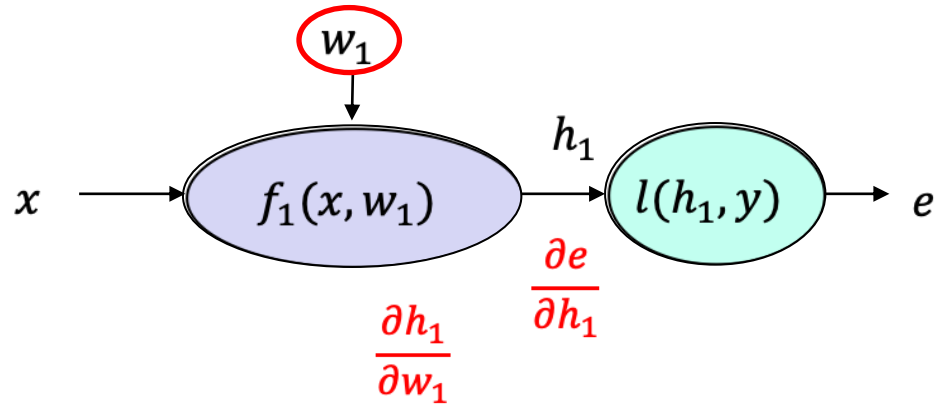
- To train the network, we need to find the gradient of the error w.r.t. the parameters of each layer, $\frac{\partial e}{\partial w_k}$

Computation graph



Chain rule

Let's start with $k = 1$



$$e = l(f_1(x, w_1), y)$$

$$\frac{\partial}{\partial w_1} l(f_1(x, w_1), y) =$$

Example: $e = (y - w_1^T x)^2$

$$h_1 = f_1(x, w_1) = w_1^T x$$

$$e = l(h_1, y) = (y - h_1)^2$$

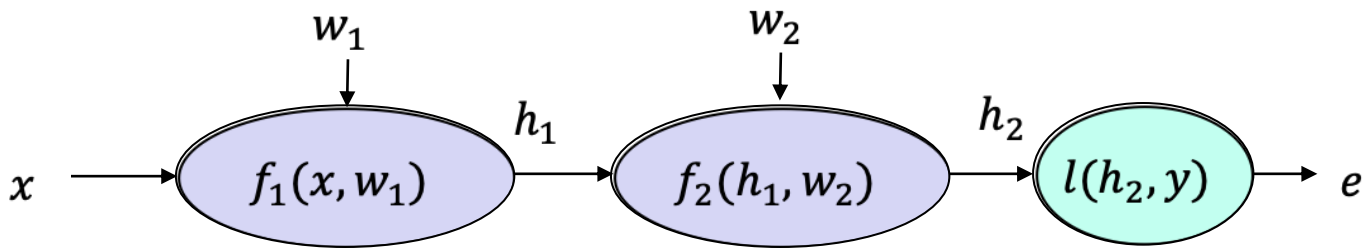
$$\frac{\partial h_1}{\partial w_1} =$$

$$\frac{\partial e}{\partial h_1} =$$

$$\frac{\partial e}{\partial w_1} = \frac{\partial e}{\partial h_1} \frac{\partial h_1}{\partial w_1} = -2(y - w_1^T x)$$

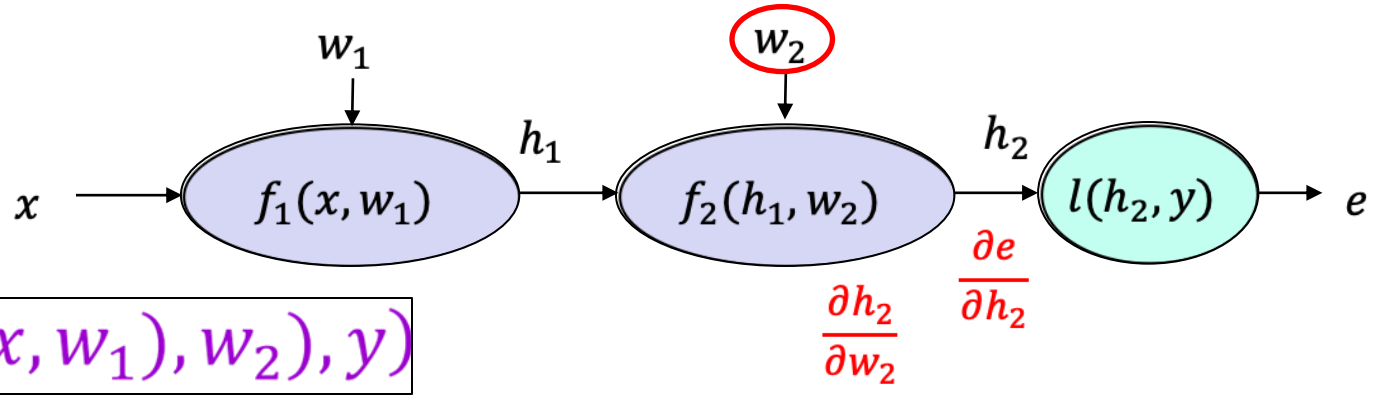
Chain rule

$k = 2$



Chain rule

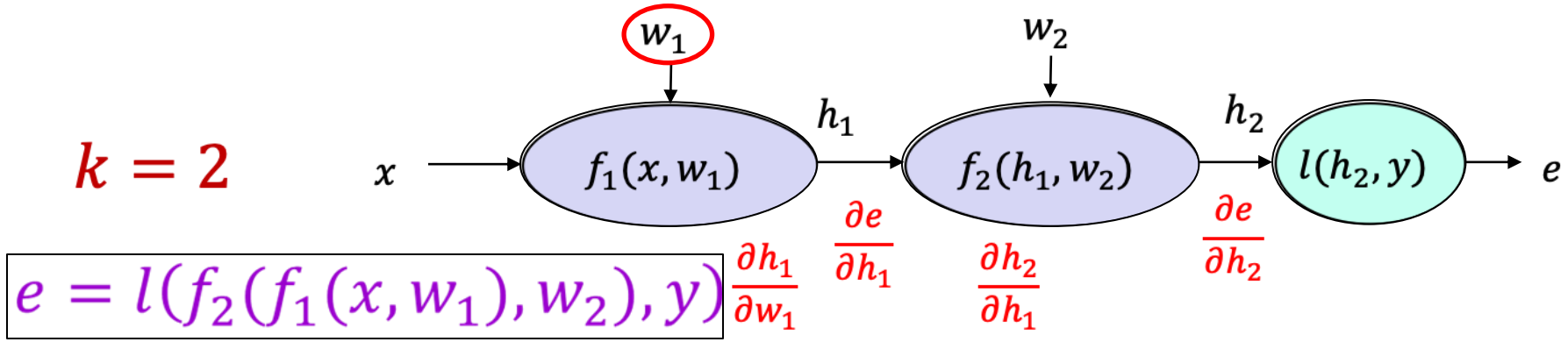
$k = 2$



$$e = l(f_2(f_1(x, w_1), w_2), y)$$

$$\frac{\partial e}{\partial w_2} =$$

Chain rule



$$\frac{\partial e}{\partial w_2} = \frac{\partial e}{\partial h_2}\frac{\partial h_2}{\partial w_2}$$

Example: $e = -\log\left(\sigma(w_1^T x)\right)$ (assume $y = 1$)

$h_1 = f_1(x, w_1) = w_1^T x$

$h_2 = f_2(h_1) = \sigma(h_1)$

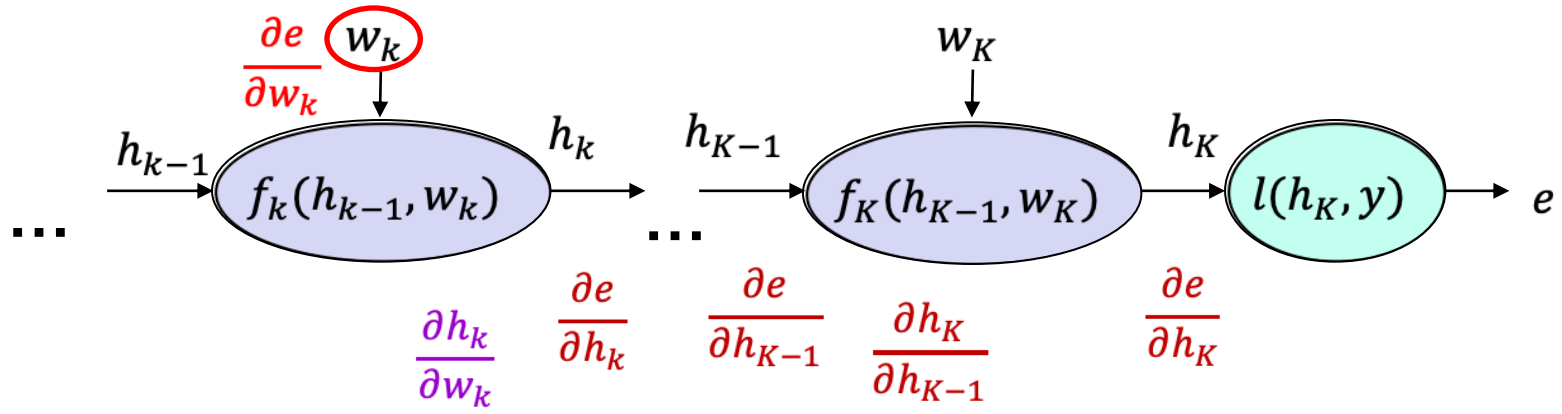
$e = l(h_2, 1) = -\log(h_2)$

$\frac{\partial h_1}{\partial w_1} =$

$\frac{\partial h_2}{\partial h_1} =$

$\frac{\partial e}{\partial h_2} =$

Chain rule



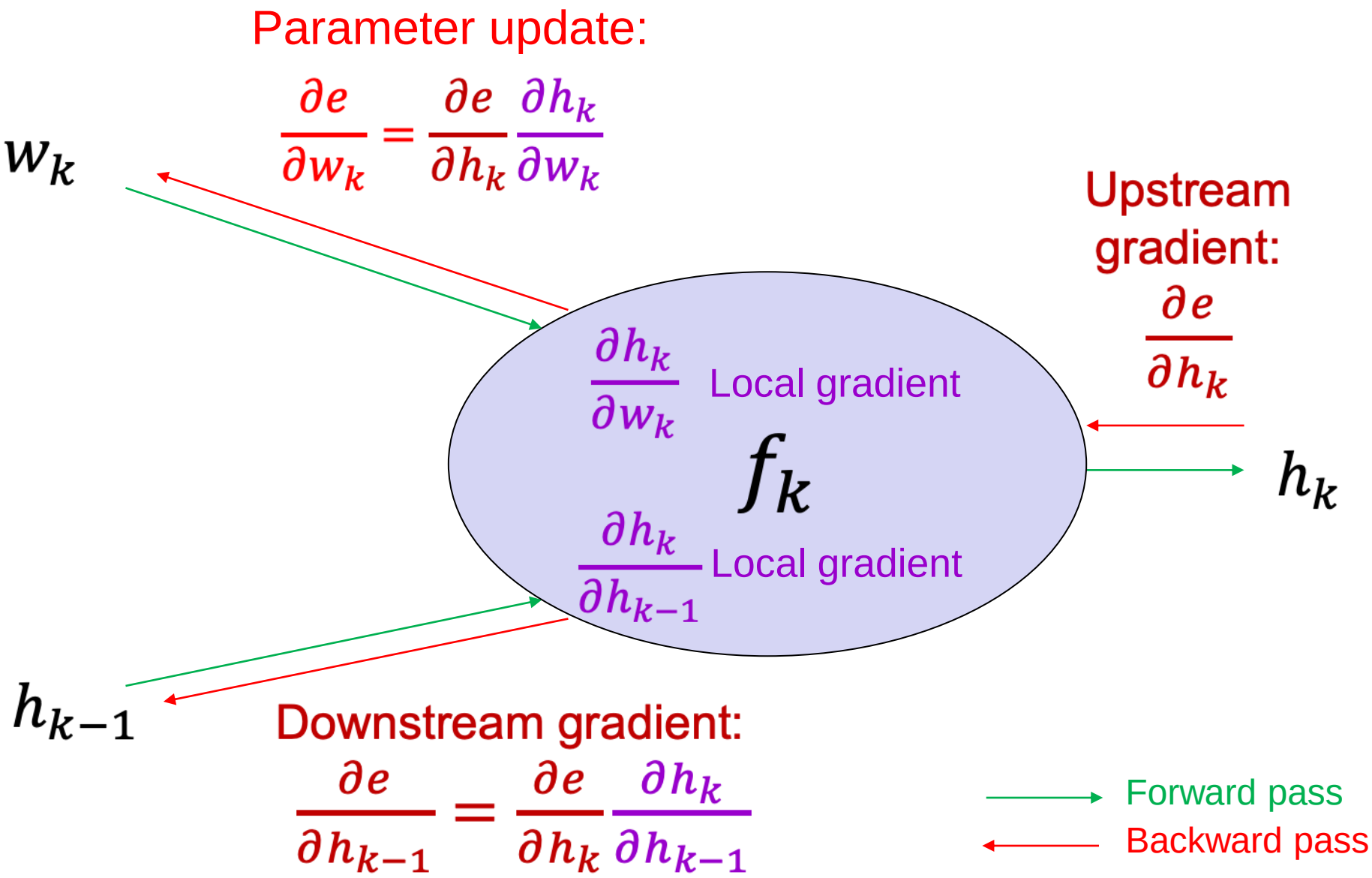
General case:

$$\frac{\partial e}{\partial w_k} = \boxed{\frac{\partial e}{\partial h_K} \frac{\partial h_K}{\partial h_{K-1}} \cdots \frac{\partial h_{k+1}}{\partial h_k}} \frac{\partial h_k}{\partial w_k}$$

Upstream gradient $\frac{\partial e}{\partial h_k}$

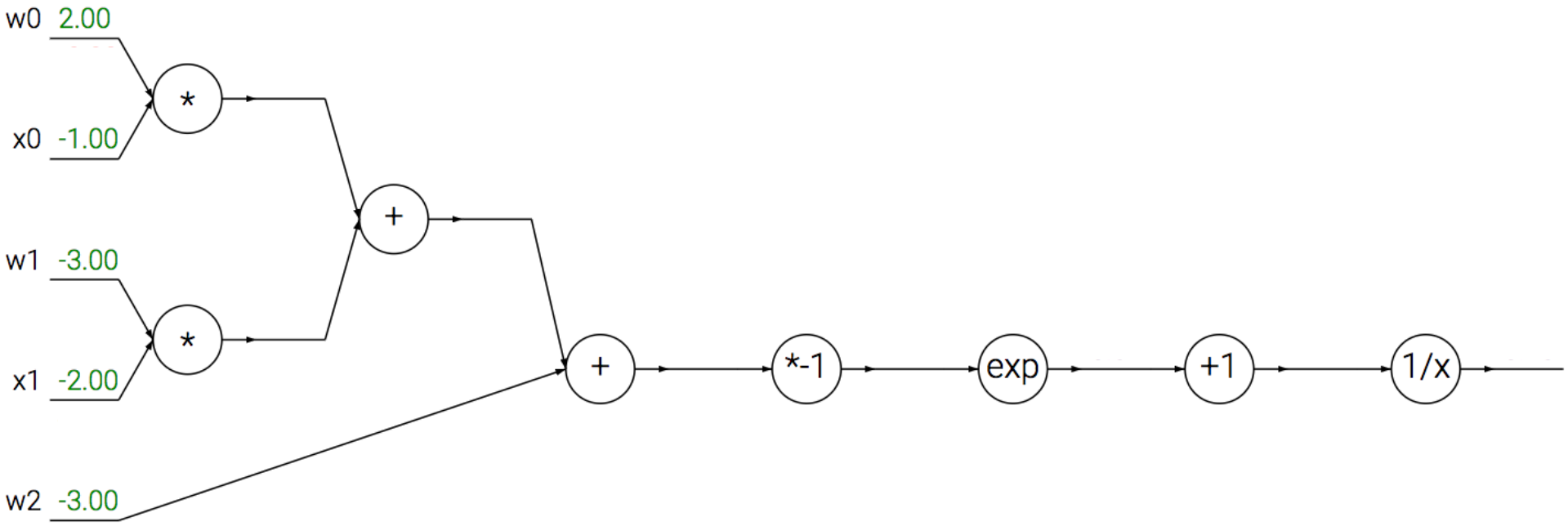
Local gradient

Backpropagation summary



A detailed example

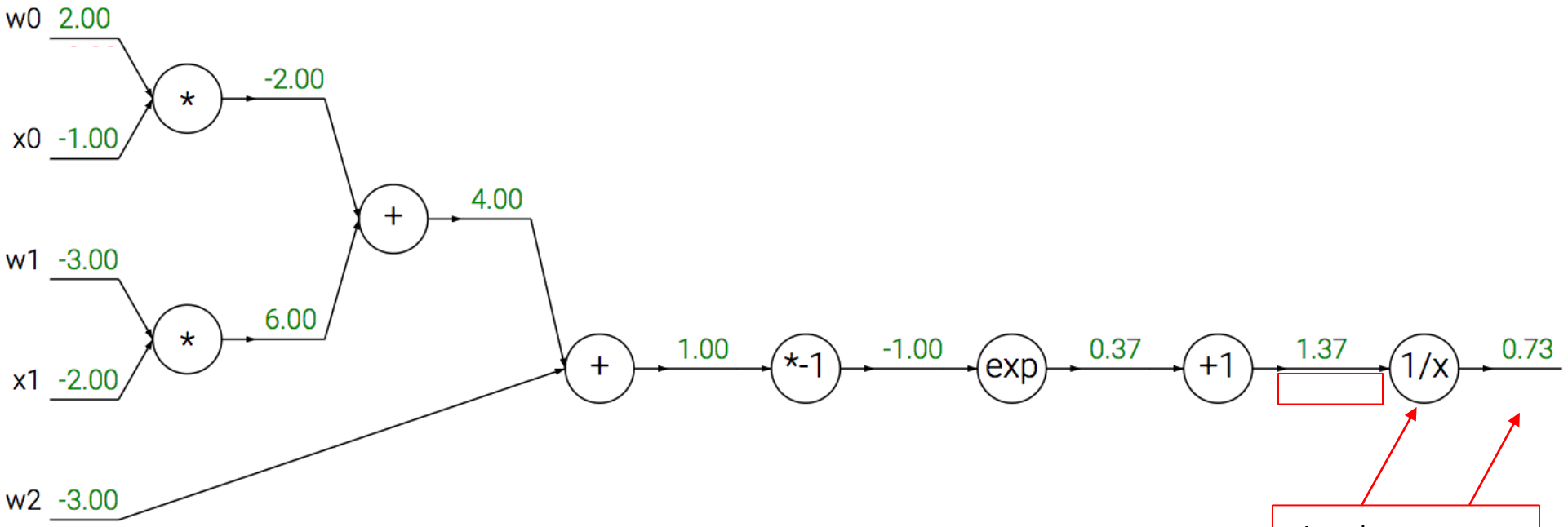
$$f(x,w) = \frac{1}{1+\exp[-(w_0x_0 + w_1x_1 + w_2)]}$$



Source: Stanford 231n

A detailed example

$$f(x,w) = \frac{1}{1+\exp[-(w_0x_0 + w_1x_1 + w_2)]}$$



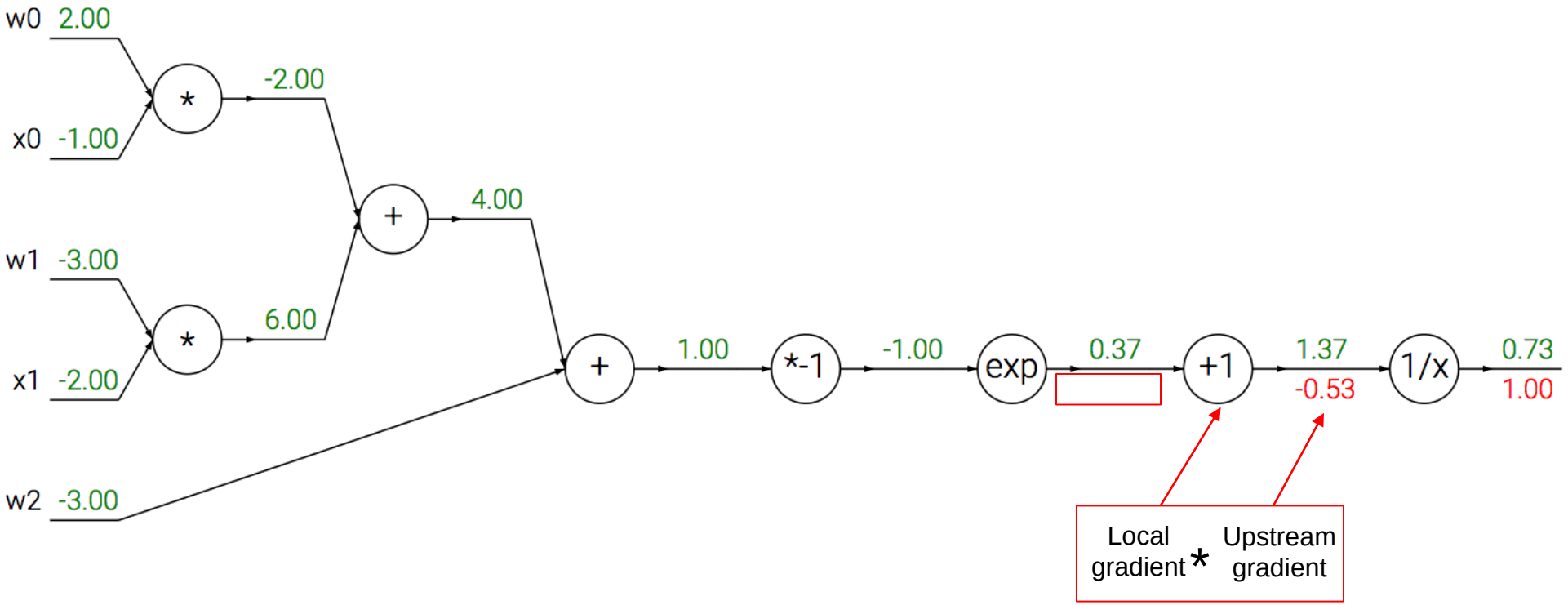
$$(1/x)' = -1/x^2$$

$$-\frac{1}{1.37^2} * 1 = -0.53$$

Source: Stanford 231n

A detailed example

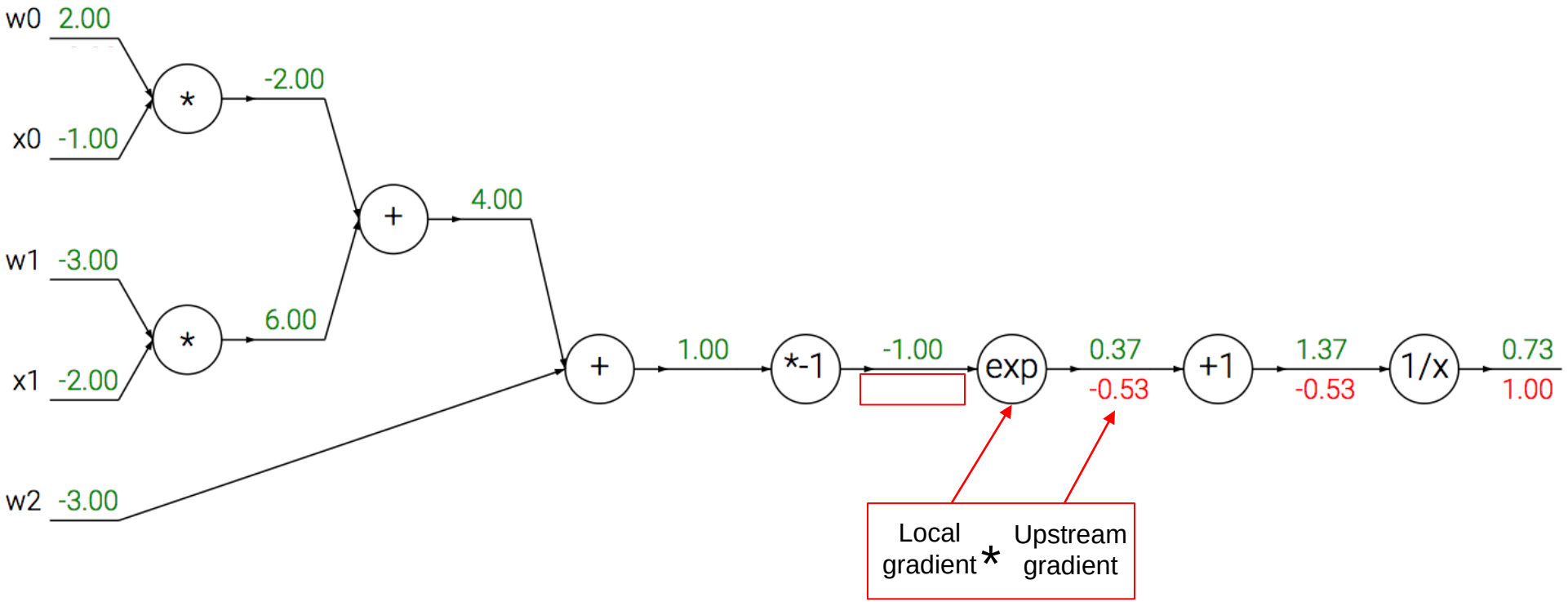
$$f(x,w) = \frac{1}{1+\exp[-(w_0x_0 + w_1x_1 + w_2)]}$$



Source: Stanford 231n

A detailed example

$$f(x,w) = \frac{1}{1+\exp[-(w_0x_0 + w_1x_1 + w_2)]}$$

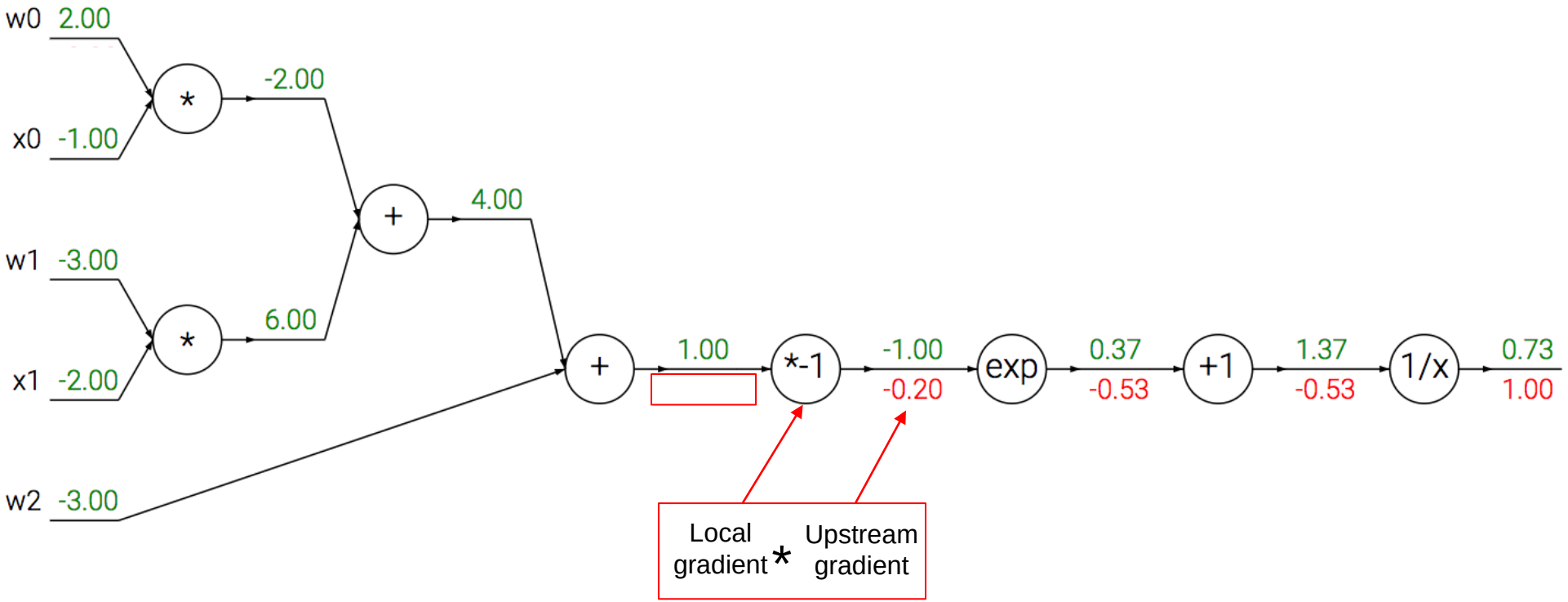


$$\exp(-1) * (-0.53) = -0.20$$

Source: Stanford 231n

A detailed example

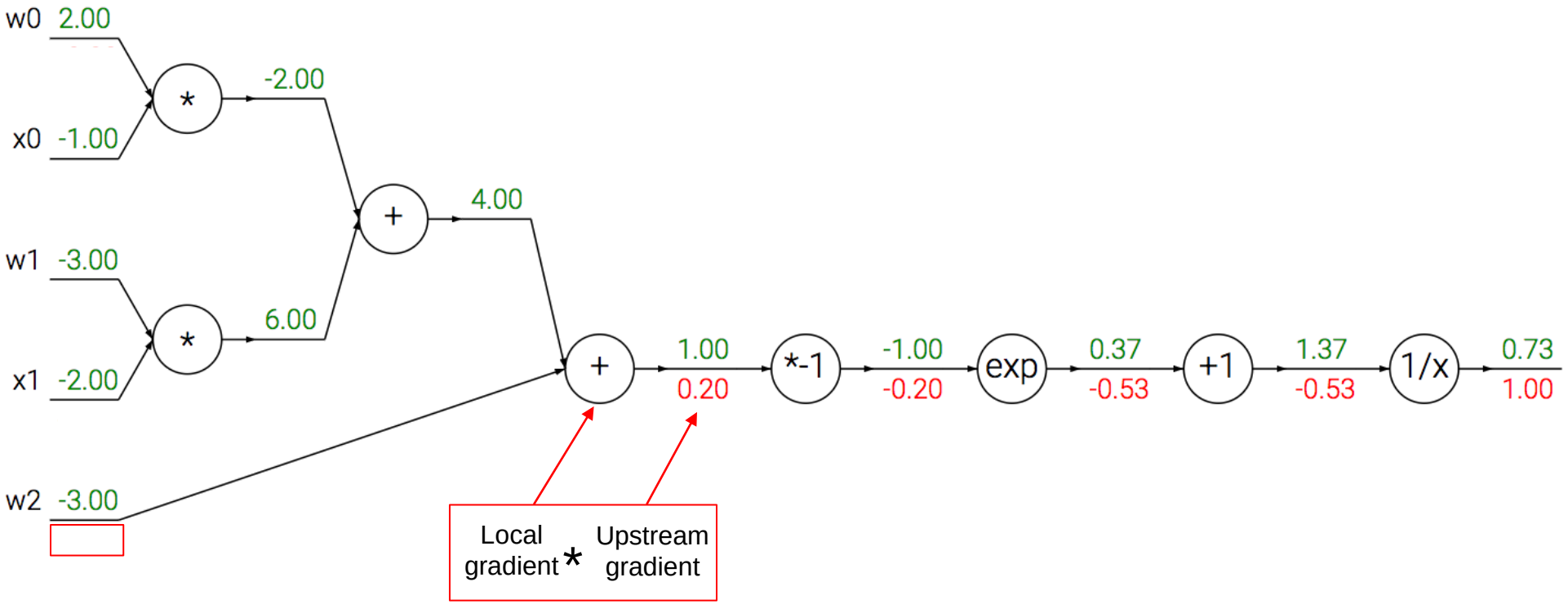
$$f(x,w) = \frac{1}{1+\exp[-(w_0x_0 + w_1x_1 + w_2)]}$$



Source: Stanford 231n

A detailed example

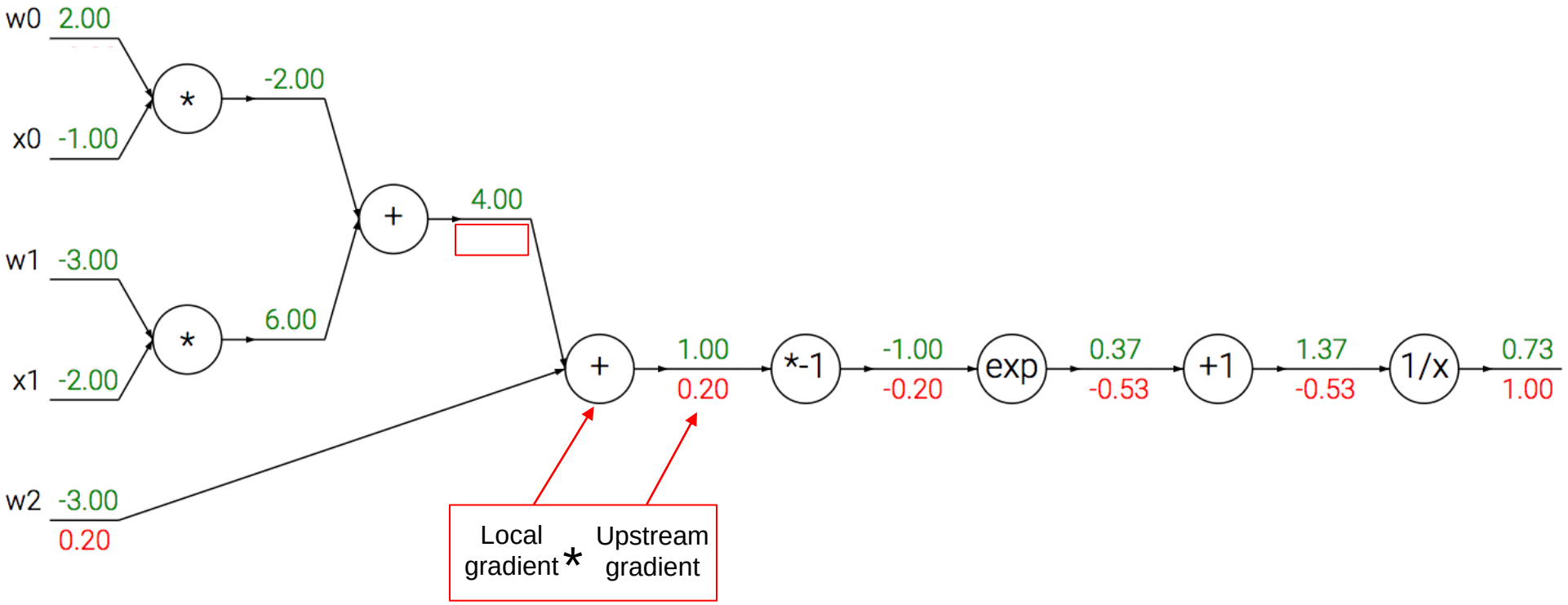
$$f(x,w) = \frac{1}{1+\exp[-(w_0x_0 + w_1x_1 + w_2)]}$$



Source: Stanford 231n

A detailed example

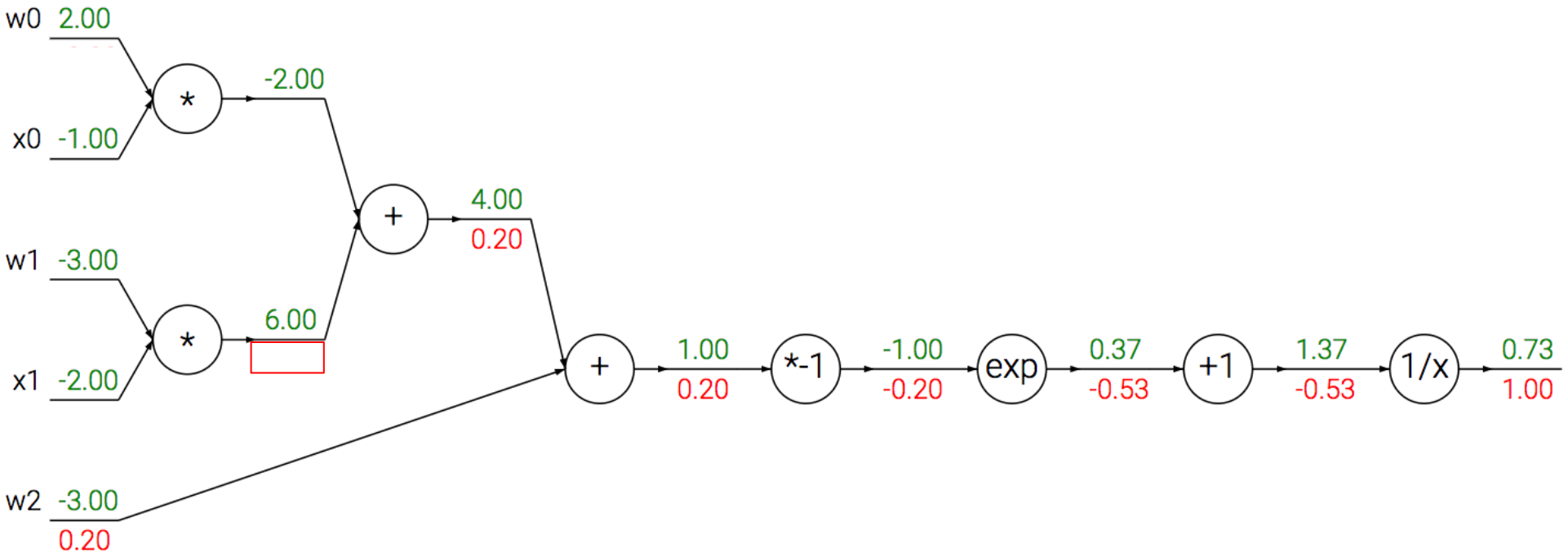
$$f(x,w) = \frac{1}{1+\exp[-(w_0x_0 + w_1x_1 + w_2)]}$$



Source: [Stanford 231n]

A detailed example

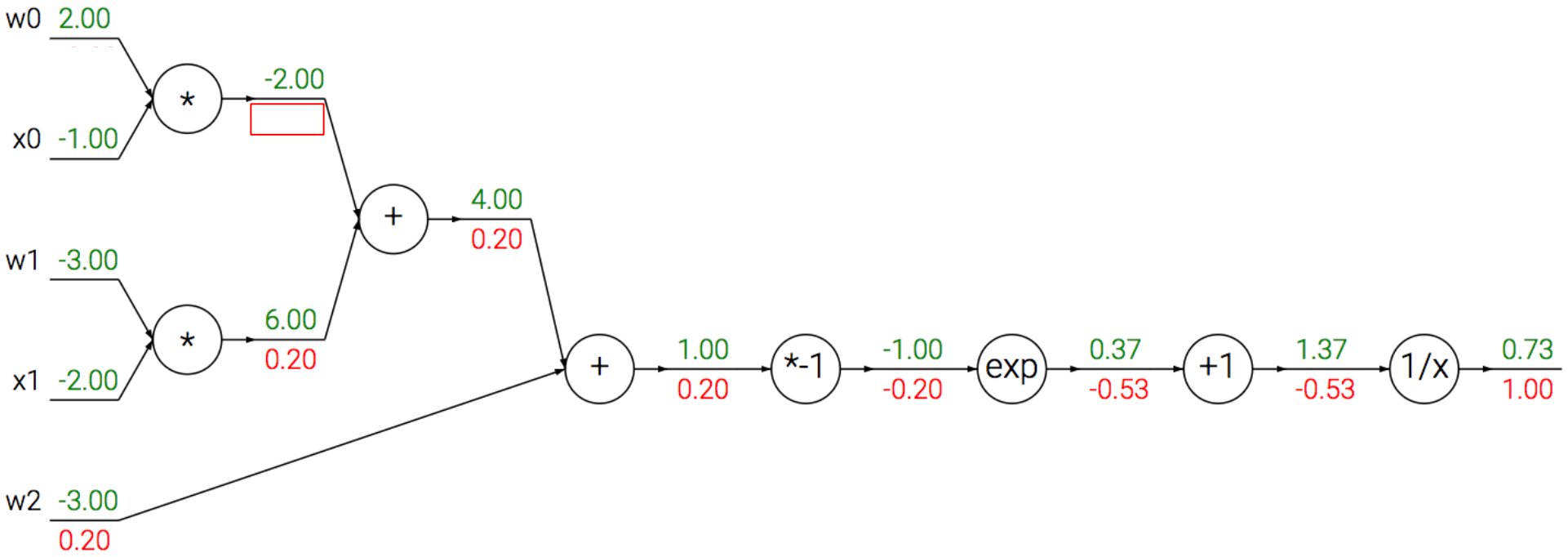
$$f(x,w) = \frac{1}{1+\exp[-(w_0x_0 + w_1x_1 + w_2)]}$$



Source: [Stanford 231n]

A detailed example

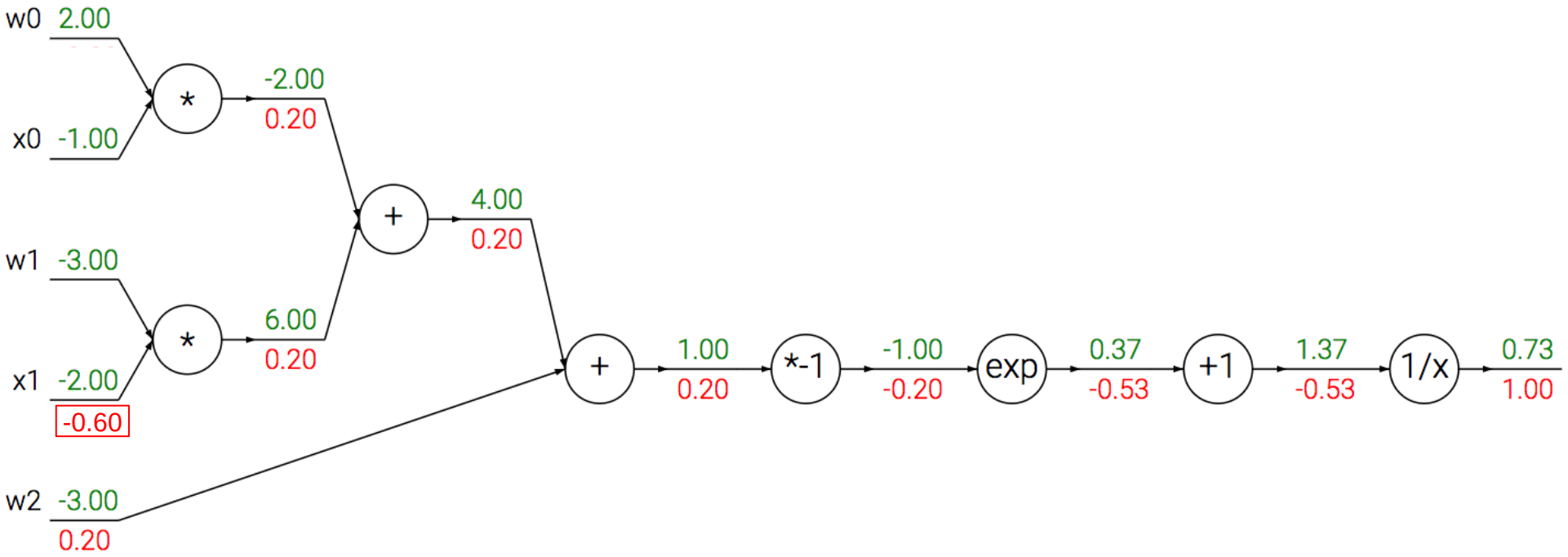
$$f(x,w) = \frac{1}{1+\exp[-(w_0x_0 + w_1x_1 + w_2)]}$$



Source: [Stanford 231n]

A detailed example

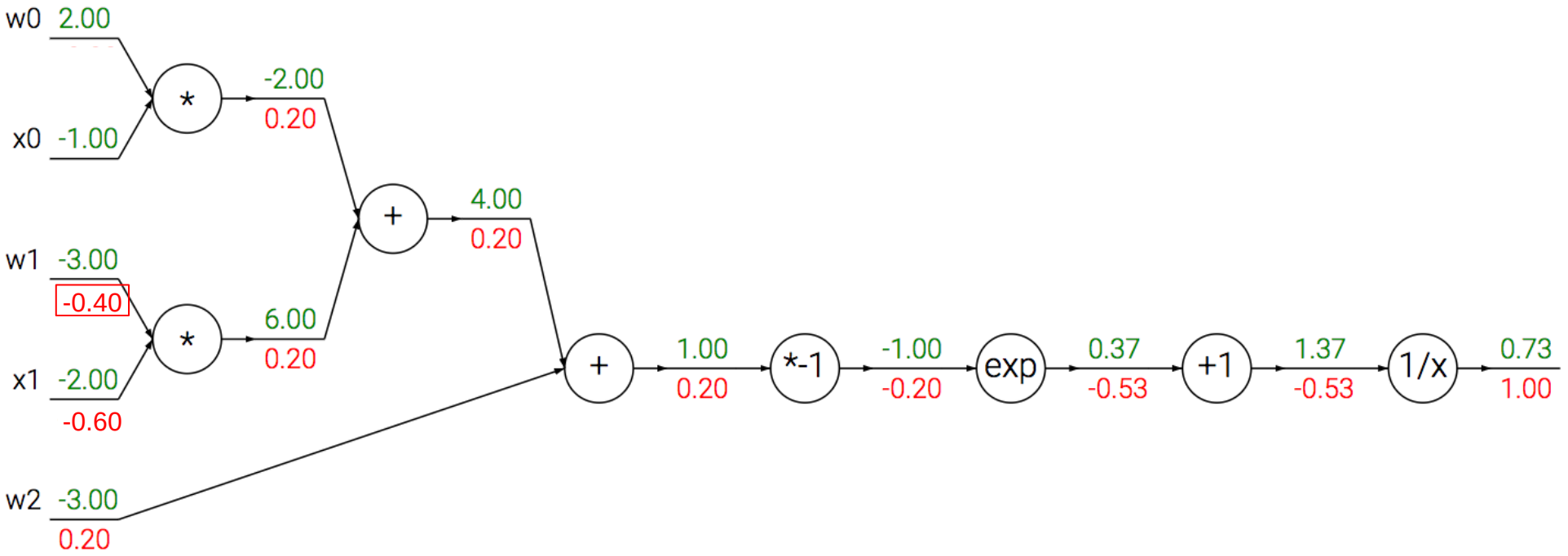
$$f(x,w) = \frac{1}{1+\exp[-(w_0x_0 + w_1x_1 + w_2)]}$$



Source: [Stanford 231n]

A detailed example

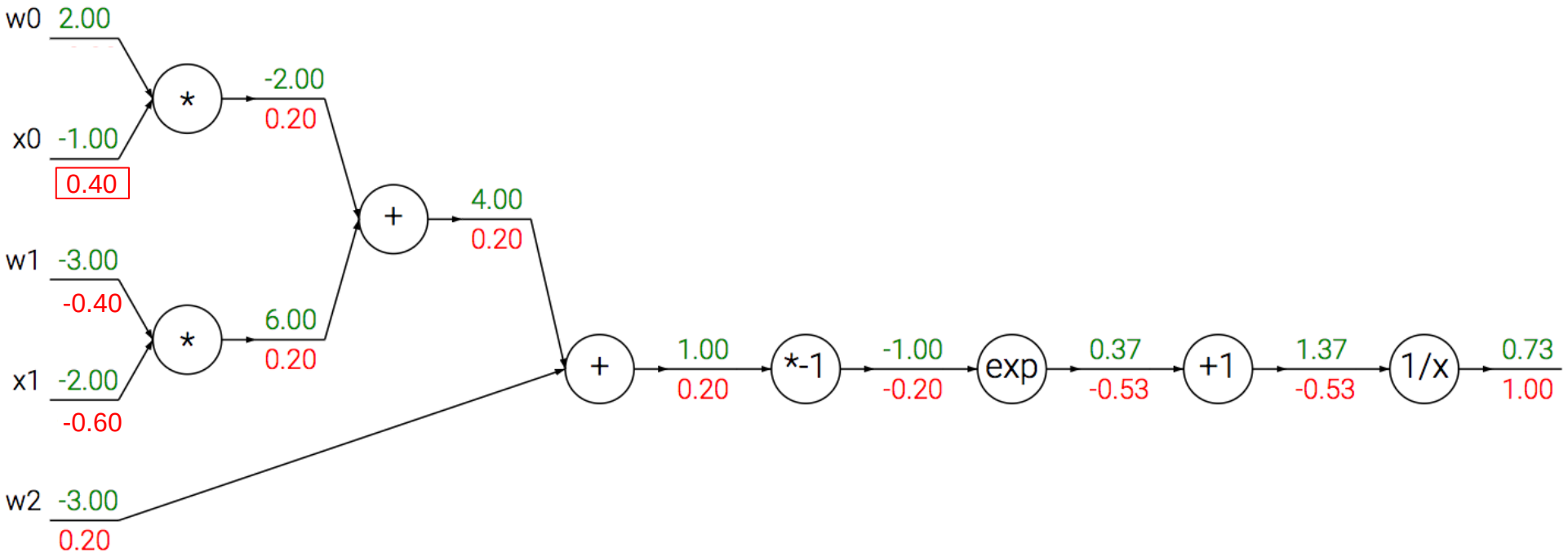
$$f(x,w) = \frac{1}{1+\exp[-(w_0x_0 + w_1x_1 + w_2)]}$$



Source: [Stanford 231n]

A detailed example

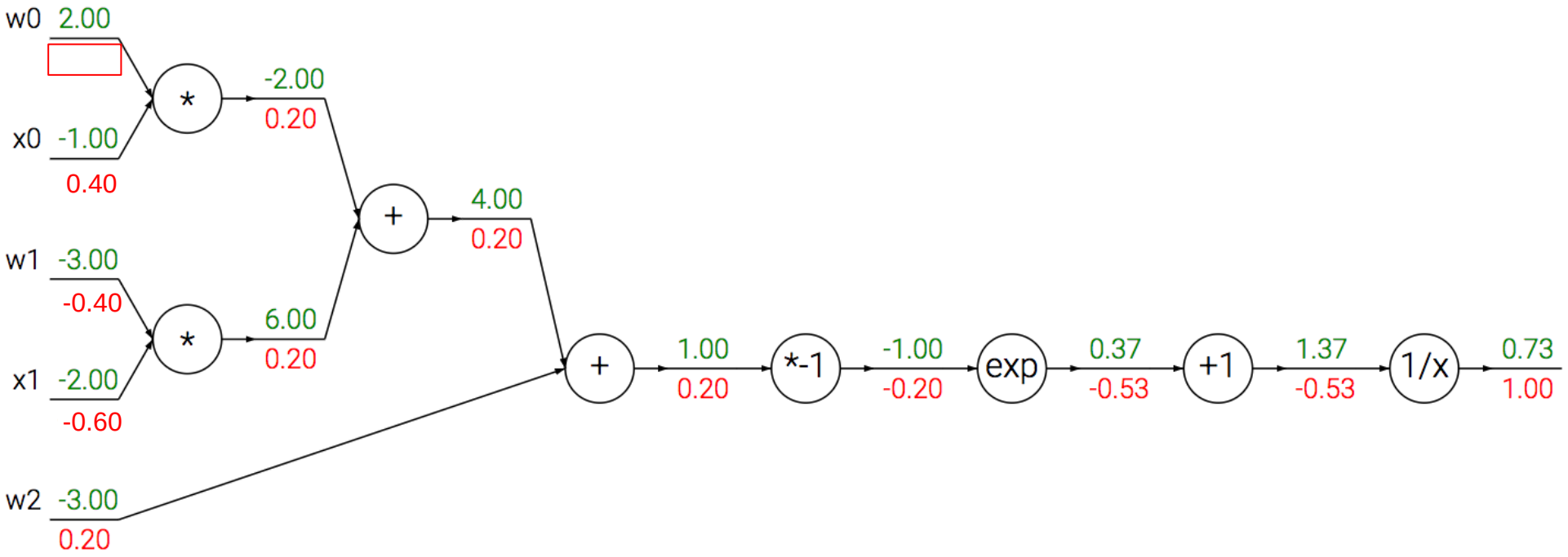
$$f(x,w) = \frac{1}{1+\exp[-(w_0x_0 + w_1x_1 + w_2)]}$$



Source: [Stanford 231n]

A detailed example

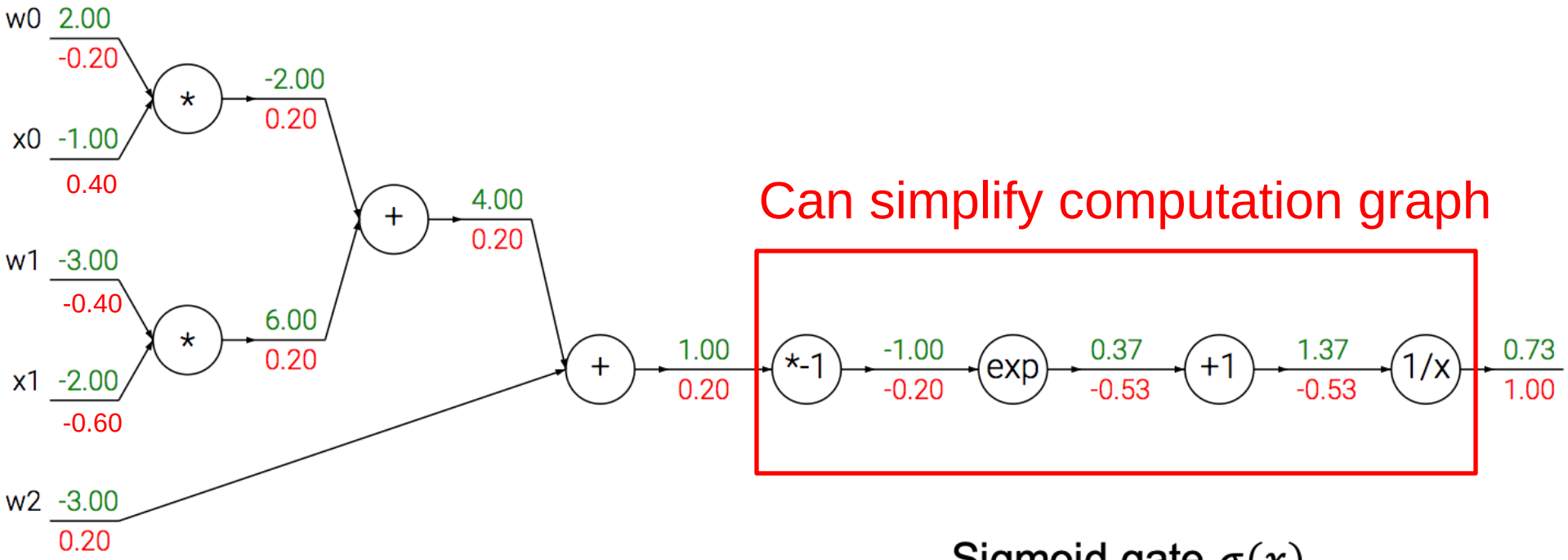
$$f(x,w) = \frac{1}{1+\exp[-(w_0x_0 + w_1x_1 + w_2)]}$$



Source: [Stanford 231n]

A detailed example

$$f(x,w) = \frac{1}{1+\exp[-(w_0x_0 + w_1x_1 + w_2)]}$$



Can simplify computation graph

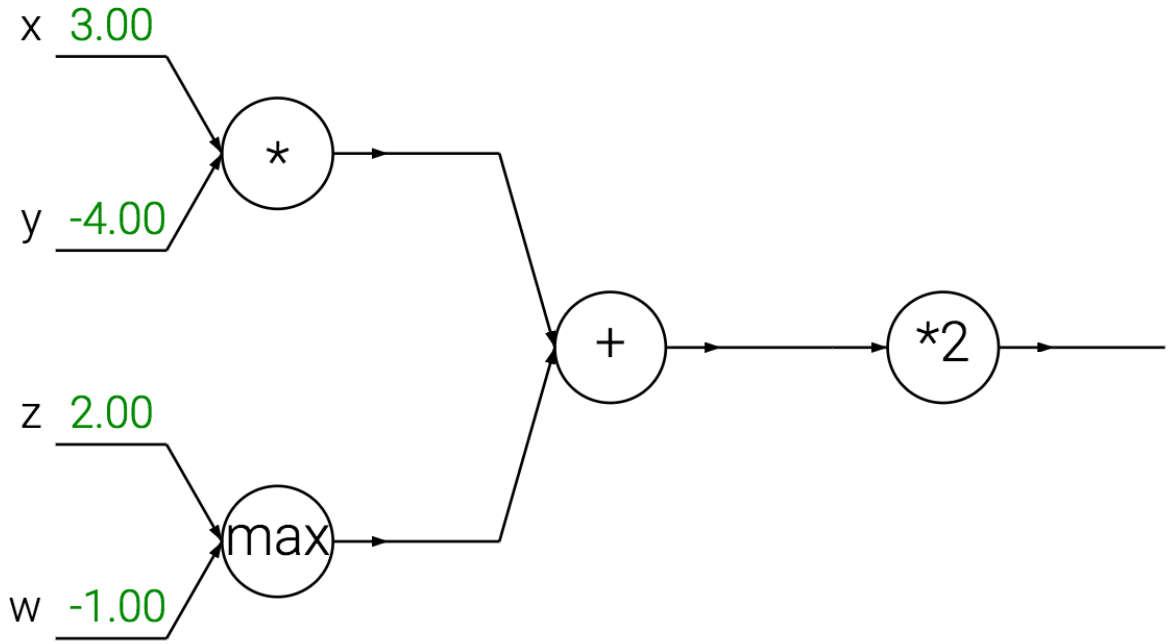
Sigmoid gate $\sigma(x)$

$$\sigma'(x) = \sigma(x)\big(1-\sigma(x)\big)$$

$$\sigma(1)\big(1-\sigma(1)\big) = 0.73 * (1-0.73) = 0.20$$

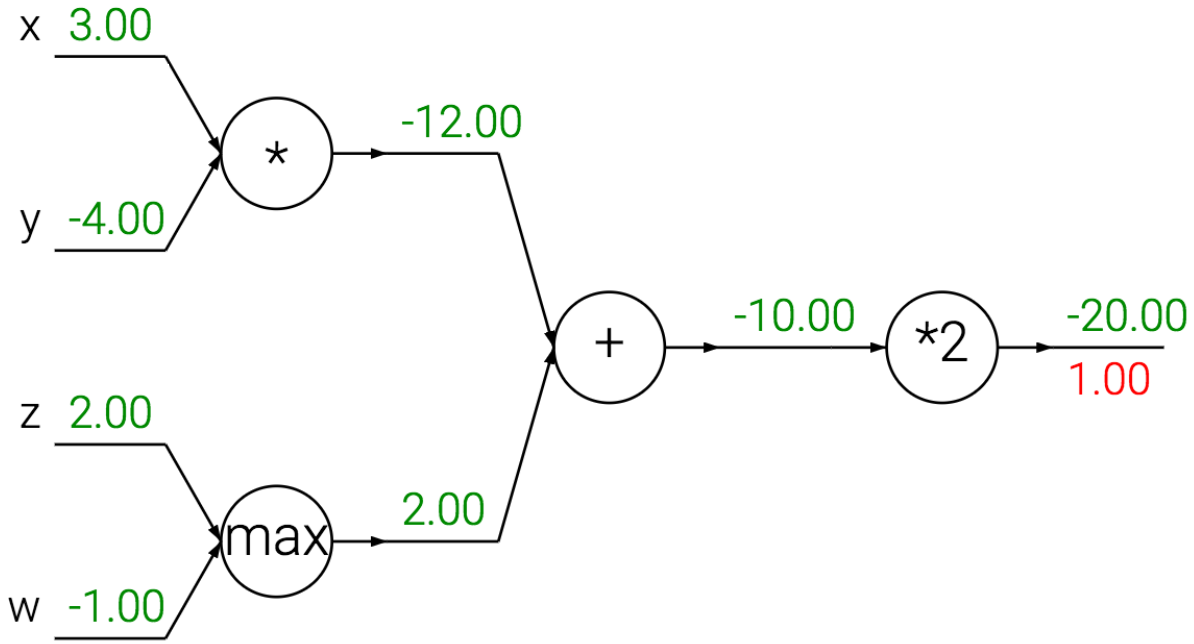
Source: Stanford 231n

Patterns in gradient flow



Source: Stanford 231n

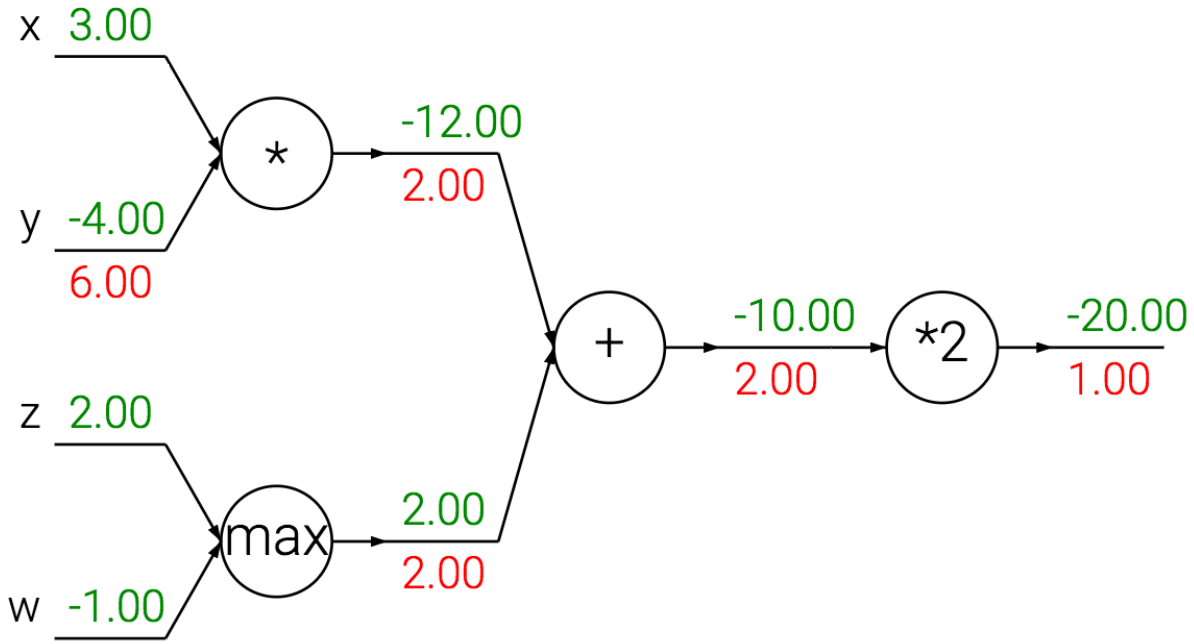
Patterns in gradient flow



Add gate: “gradient distributor”

Source: Stanford 231n

Patterns in gradient flow

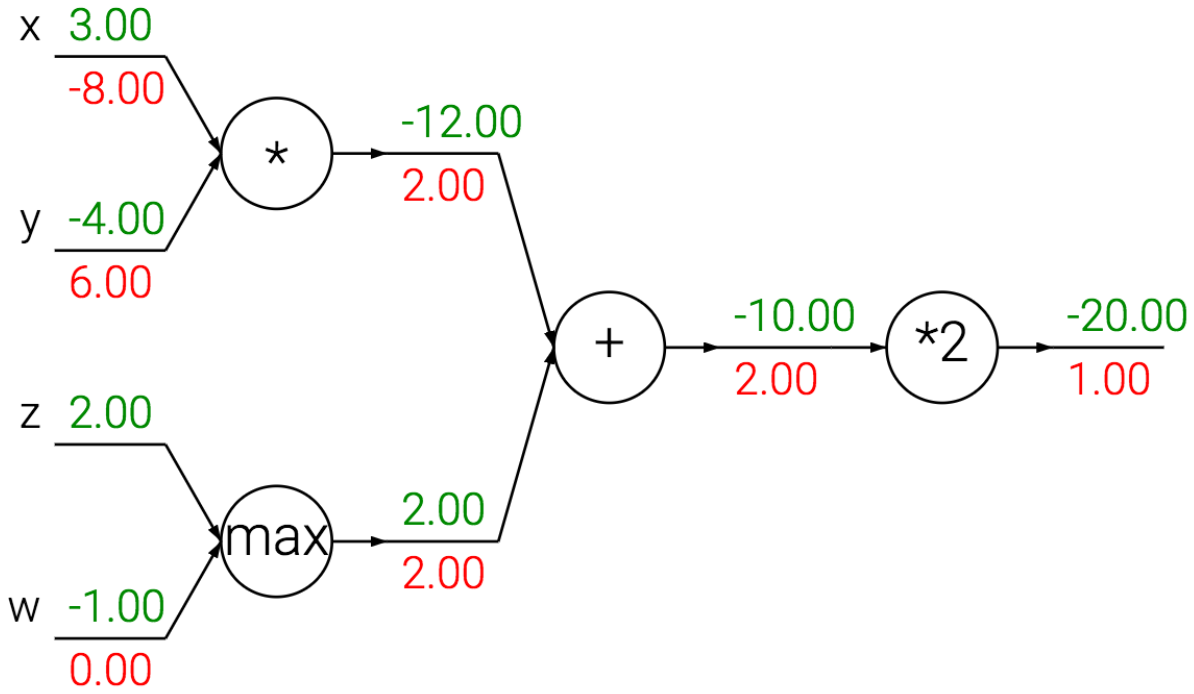


Add gate: “gradient distributor”

Multiply gate: “gradient switcher”

Source: Stanford 231n

Patterns in gradient flow



Add gate: “gradient distributor”

Multiply gate: “gradient switcher”

Max gate: “gradient router”

Source: Stanford 231n

General tips

- Derive error signal (upstream gradient) directly, avoid explicit computation of huge local derivatives
- Write out expression for a single element of the Jacobian, then deduce the overall formula
- Keep consistent indexing conventions, order of operations
- Use dimension analysis

- **For further reading:**
  - Lecture 4 of Stanford 231n
  - Yes you should understand backprop by Andrej Karpathy

Acknowledgement

Thanks to the following courses and corresponding researchers for making their teaching/research material online

- Deep Learning, Stanford University
- Introduction to Deep Learning, University of Illinois at Urbana-Champaign
- Introduction to Deep Learning, Carnegie Mellon University
- Convolutional Neural Networks for Visual Recognition, Stanford University
- Natural Language Processing with Deep Learning, Stanford University
- And Many More .....