Introduction to neural networks II

MUSTAFA HAJIJ

Objectives

- Recall the feedforward of a neural network
- The binary classification problem
- How to build a neural network that can classify a binary labeled data?
- How can we build a neural network that can classify a multi-labeled data?
- Introduction of the sigmoid function
- Introduction of the softmax function
- Computational Graphs
- Backprop and automatic differentiation
- The approximation power of neural networks (universal approximation theorem)

How do we use a neural network as a classifier?

Last time we learned the following:

- The building block of a neural network

How do we use a neural network as a classifier?

Last time we learned the following:

- The building block of a neural network
- How to build a neural network.

How do we use a neural network as a classifier?

Last time we learned the following:

- The building block of a neural network
- How to build a neural network.
- Neural network is essentially a mathematical function $f: R^n \rightarrow R^m$.

How do we use a neural network as a classifier?

Last time we learned the following:

- The building block of a neural network
- How to build a neural network.
- Neural network is essentially a mathematical function $f: R^n \to R^m$.
- Given an input x, how to feedforward x through a neural network and obtain an output f(x)

How do we use a neural network as a classifier?

Last time we learned the following:

- The building block of a neural network
- How to build a neural network.
- Neural network is essentially a mathematical function $f: R^n \rightarrow R^m$.
- Given an input x, how to feedforward x through a neural network and obtain an output f(x)
- How to train a neural network :
    - Define a cost function
    - For each example in the training set feedforward that example and compute the error
    - Use backpropagation to adjust the weights of the network so that it behaves better with respect to the input example

How do we use a neural network as a classifier?

Last time we learned the following:

- The building block of a neural network
- How to build a neural network.
- Neural network is essentially a mathematical function $f: R^n \to R^m$.
- Given an input x, how to feedforward x through a neural network and obtain an output f(x)
- How to train a neural network :
    - Define a cost function
    - For each example in the training set feedforward that example and compute the error
    - Use backpropagation to adjust the weights of the network so that it behaves better with respect to the input example

Lets recall the feedforward algorithm before first.

Feedforward Neural Network

How do we compute a feedforward neural network on an input x ?

Feedforward Neural Network

Start with an input $x = a^{(0)}$. In the picture, this is represented by the first layer of nodes. We will call this layer 0.

$$x = a^{(0)}$$

Feedforward Neural Network

We apply the weight $W^{(1)}$ coming from the edges between layer 0 and layer 1 and add the biases and then apply the Activation function on the resulting vector coordinate-wise.

$$x = a^{(0)} \longrightarrow \boxed{\sigma(W^{(1)}a^{(0)}+b^{(1)})}$$

$W^{(1)}$: Edges between layer 0 and layer 1
$a^{(0)}$: input
$b^{(1)}$: biases applied to layer 1
σ : activation function

Feedforward Neural Network

We will call the output of this computation $a^{(1)}$. This is now represented by the nodes in layer 1.

$$x = a^{(0)} \longrightarrow \boxed{\sigma(W^{(1)}a^{(0)}+b^{(1)})} \xrightarrow{a^{(1)}}$$

$W^{(1)}$: Edges between layer 0 and layer 1
$a^{(0)}$: input
$b^{(1)}$: biases applied to layer 1
σ : activation function

Feedforward Neural Network

Repeat.

$$x = a^{(0)} \longrightarrow \boxed{\sigma(W^{(1)}a^{(0)}+b^{(1)})} \xrightarrow{a^{(1)}} \boxed{\sigma(W^{(2)}a^{(1)}+b^{(2)})} \xrightarrow{a^{(2)}}$$

$W^{(2)}$: Edges between layer 1 and layer 2

$a^{(1)}$: input from layer 1

$b^{(2)}$: biases applied to layer 2

σ : activation function

Feedforward Neural Network

Until you finish the neural network and get the final output.

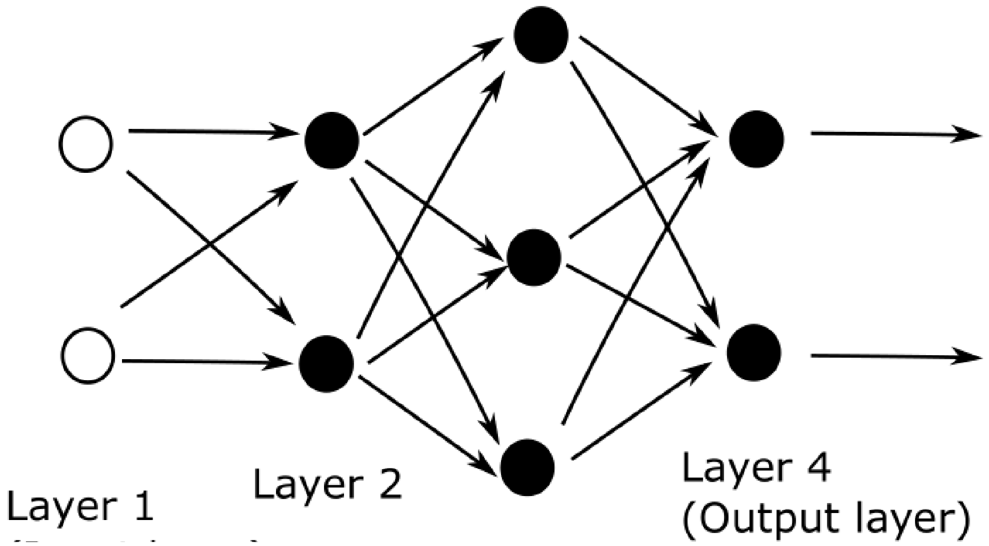
$$x = a^{(0)} \rightarrow \boxed{\sigma(W^{(1)}a^{(0)}+b^{(1)})} \xrightarrow{a^{(1)}} \boxed{\sigma(W^{(2)}a^{(1)}+b^{(2)})} \xrightarrow{a^{(2)}} \boxed{\sigma(W^{(3)}a^{(2)}+b^{(3)})} \dashrightarrow$$

$$\dashrightarrow \boxed{\sigma(W^{(n)}a^{(n-1)}+b^{(n)})} \rightarrow a^{(n)}=y$$

Example

We will use an example from this paper.

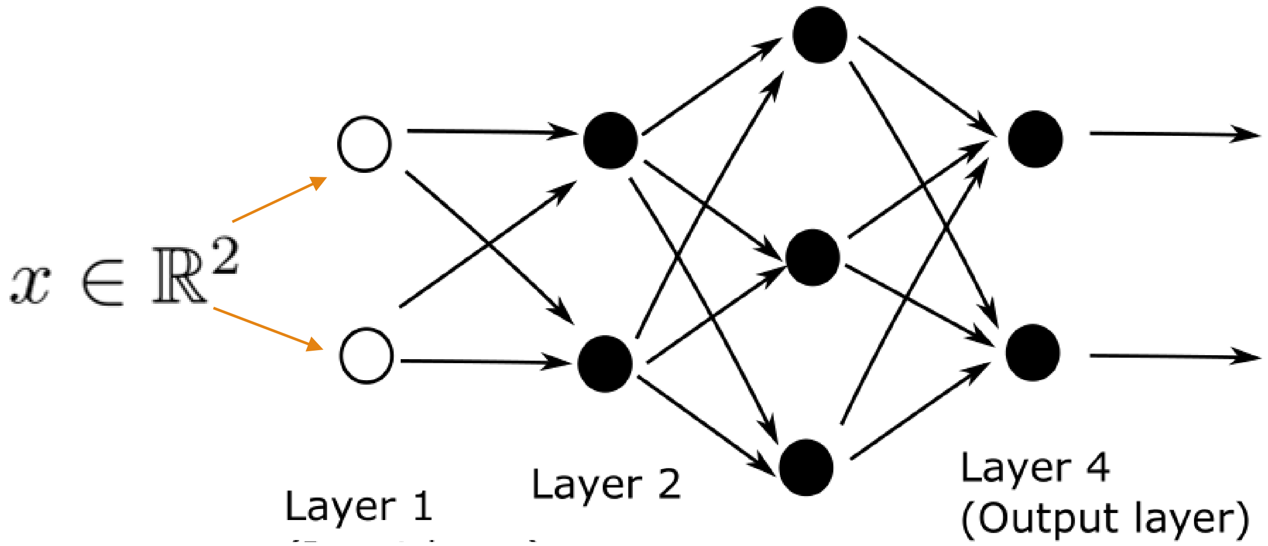
(note that the convention of the index is a little different here)



Example

We will use an example from this paper.

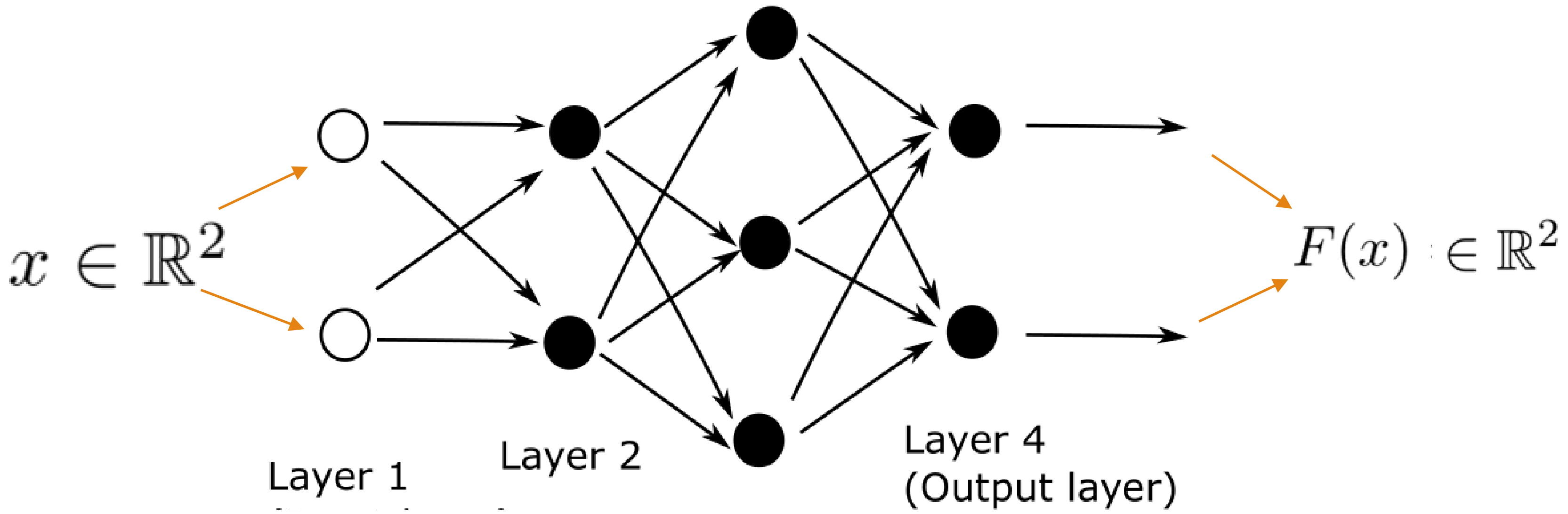
(note that the convention of the index is a little different here)



Example

We will use an example from this paper.

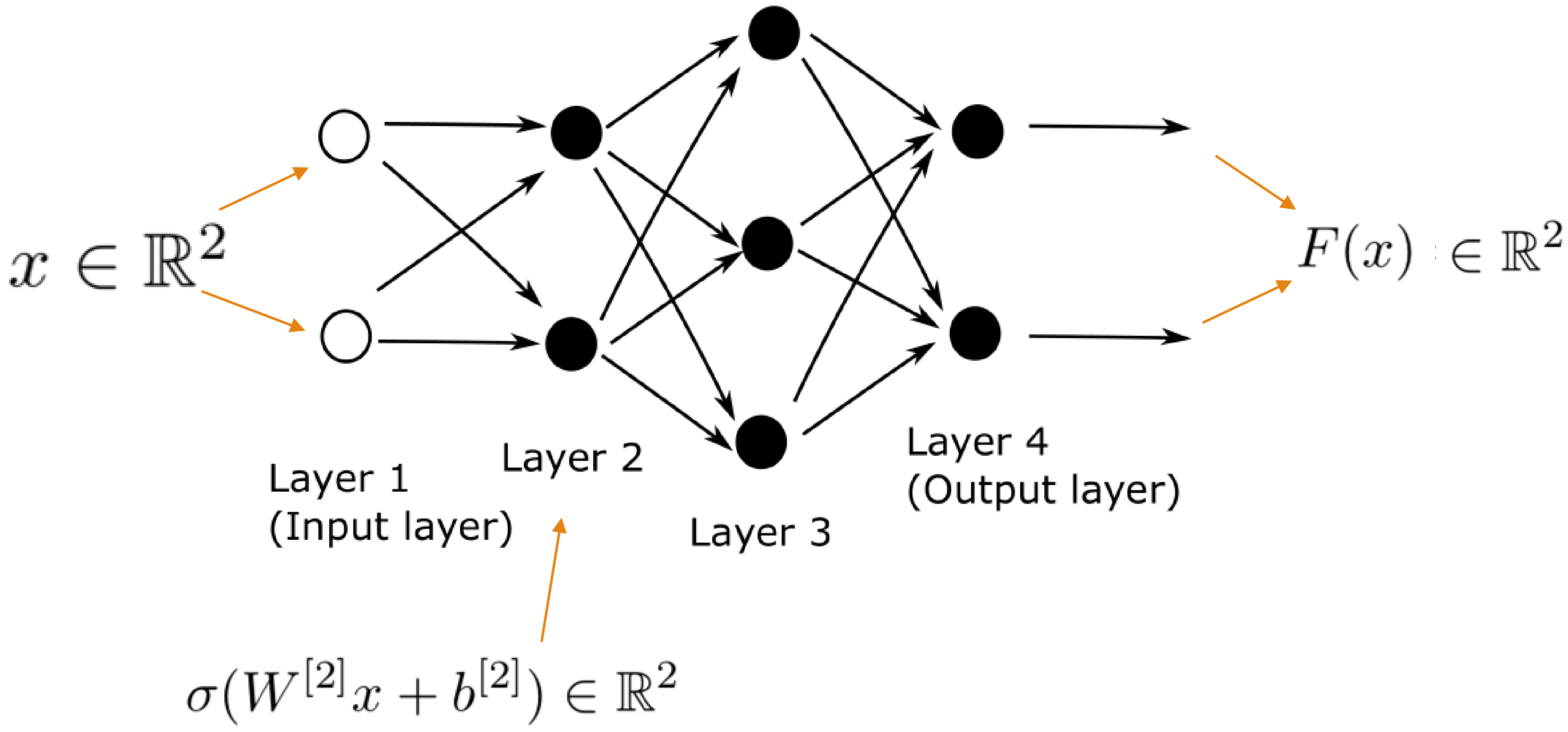
(note that the convention of the index is a little different here)



Example

We will use an example from this paper.

(note that the convention of the index is a little different here)

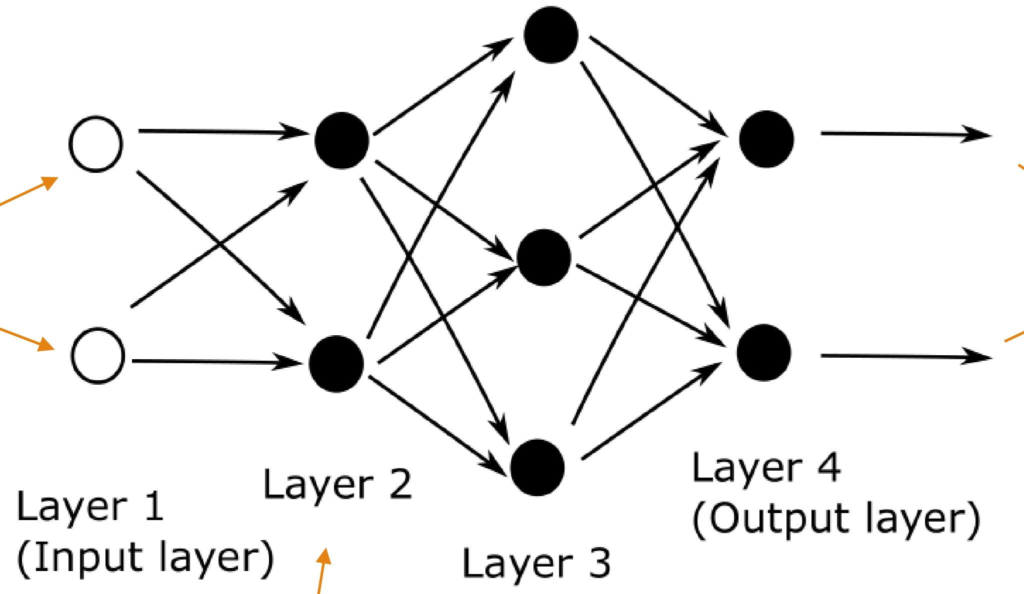


Example

We will use an example from this paper.

(note that the convention of the index is a little different here)

$x \in \mathbb{R}^2$



$F(x) \in \mathbb{R}^2$

$$\sigma(W^{[2]}x + b^{[2]}) \in \mathbb{R}^2$$

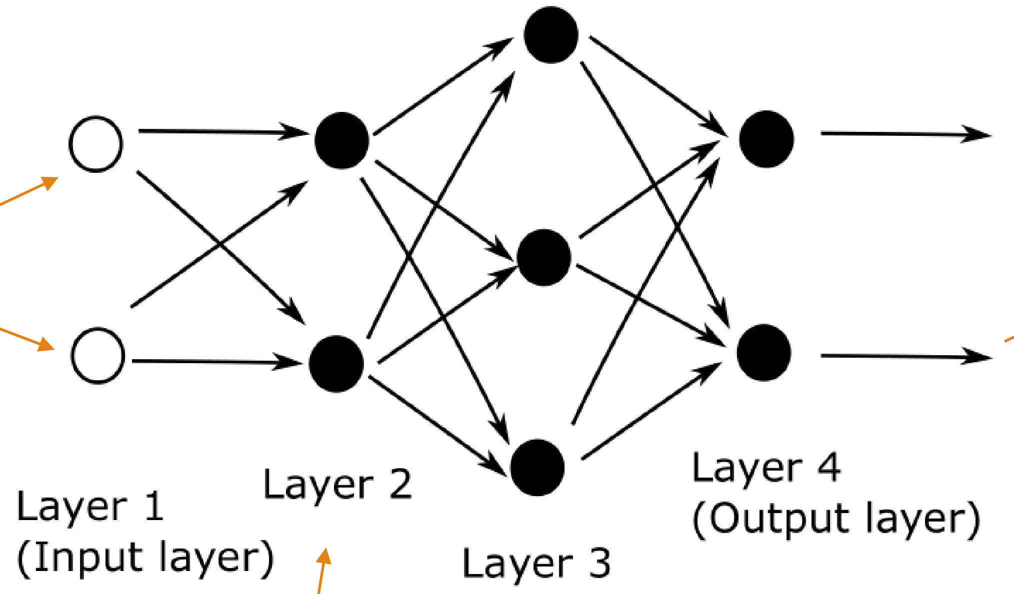
$$\sigma\left(W^{[3]}\sigma(W^{[2]}x + b^{[2]}) + b^{[3]}\right) \in \mathbb{R}^3$$

Example

We will use an example from this paper.

(note that the convention of the index is a little different here)

$$x \in \mathbb{R}^2$$



$$F(x) \in \mathbb{R}^2$$

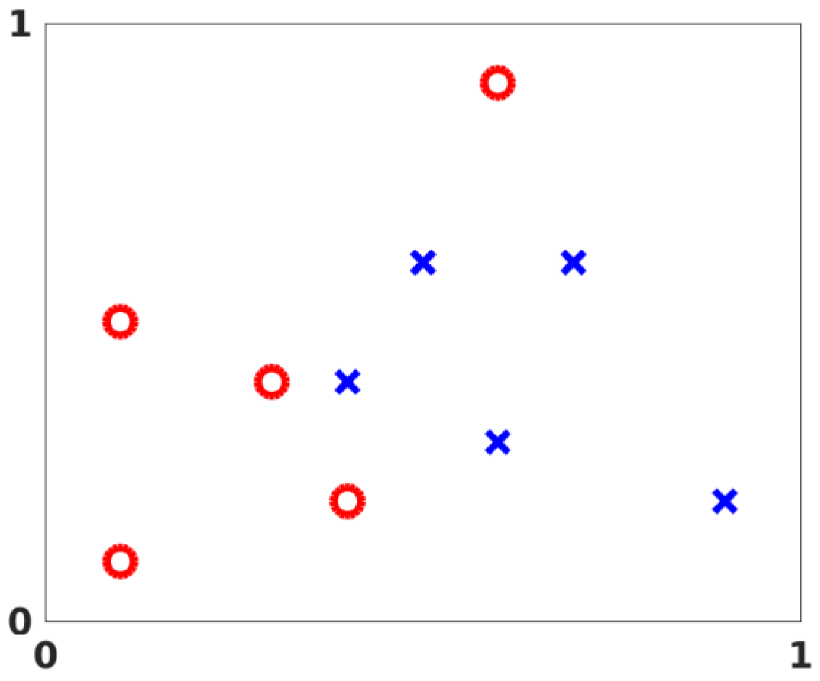
$$\sigma(W^{[2]}x + b^{[2]}) \in \mathbb{R}^2$$

$$\sigma\left(W^{[3]}\sigma(W^{[2]}x + b^{[2]}) + b^{[3]}\right) \in \mathbb{R}^3$$

Final function representing the neural network

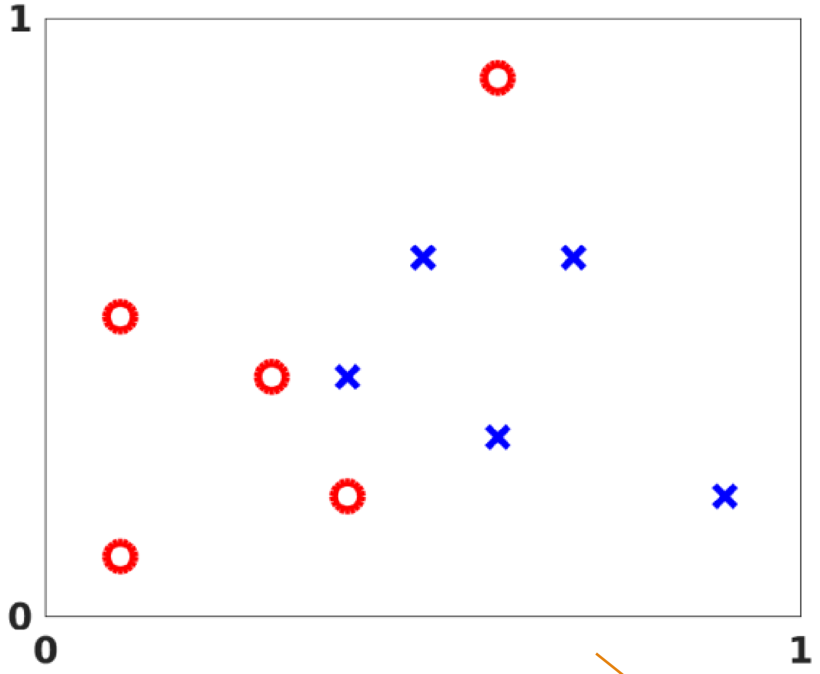
$$F(x) = \sigma\left(W^{[4]}\sigma\left(W^{[3]}\sigma(W^{[2]}x + b^{[2]}) + b^{[3]}\right) + b^{[4]}\right) \in \mathbb{R}^2.$$

Example



Input : labeled data X

Example

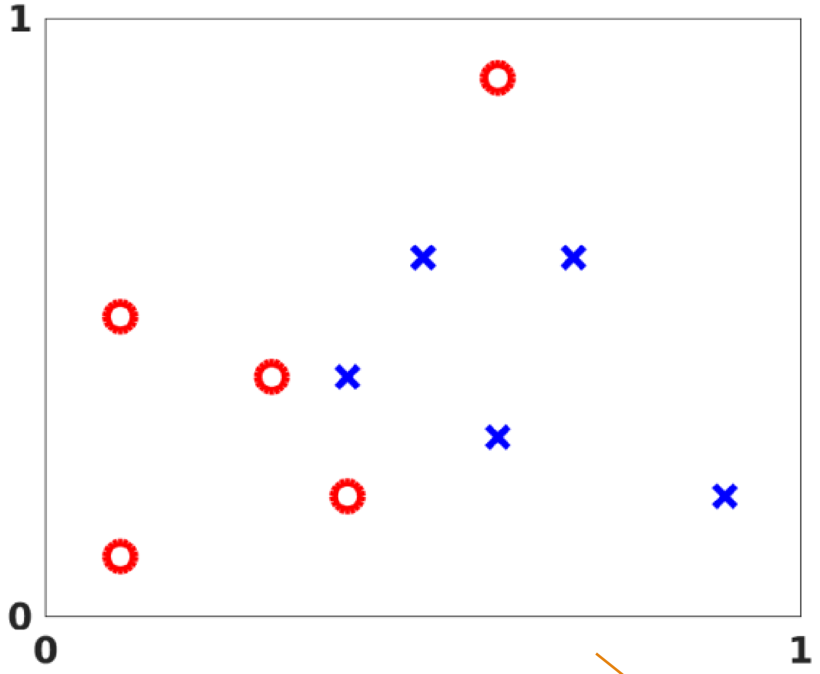


Input : labeled data X

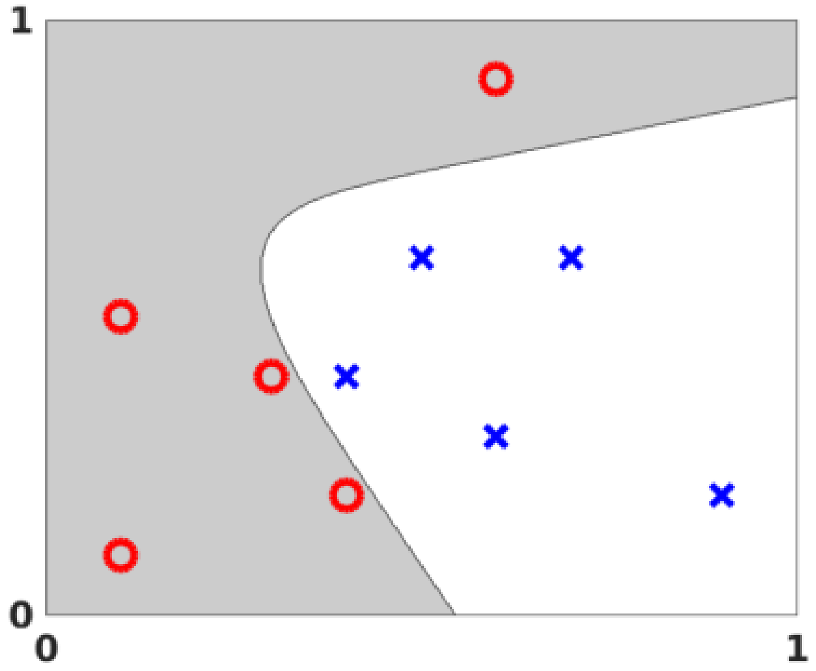
$$\text{Cost}\left(W^{[2]}, W^{[3]}, W^{[4]}, b^{[2]}, b^{[3]}, b^{[4]}\right) = \frac{1}{10}\sum_{i=1}^{10} \tfrac{1}{2}\|y(x^{\{i\}}) - F(x^{\{i\}})\|_2^2.$$

the difference between the output given by the network and the actual label

Example



Input : labeled data X



Minimize the cost function

$$\mathrm{Cost}\left(W^{[2]},W^{[3]},W^{[4]},b^{[2]},b^{[3]},b^{[4]}\right)=\frac{1}{10}\sum_{i=1}^{10}\tfrac{1}{2}\|y(x^{\{i\}})-F(x^{\{i\}})\|_2^2.$$

the difference between the output given by the network and the actual label

Binary classification

Now suppose that we have data set that consists of images of cats and dogs and we built a neural network that takes as input an image from this data set and gives out a vector in R^1 (a real number).
How exactly do we use this vector for our classification task ? In general the output f(x) coming from the neural network Does not match the class $\{\pm 1\}$ of the input point x (it could be any real number).

Binary classification

This function takes a tensor of size input_size and returns a real number.

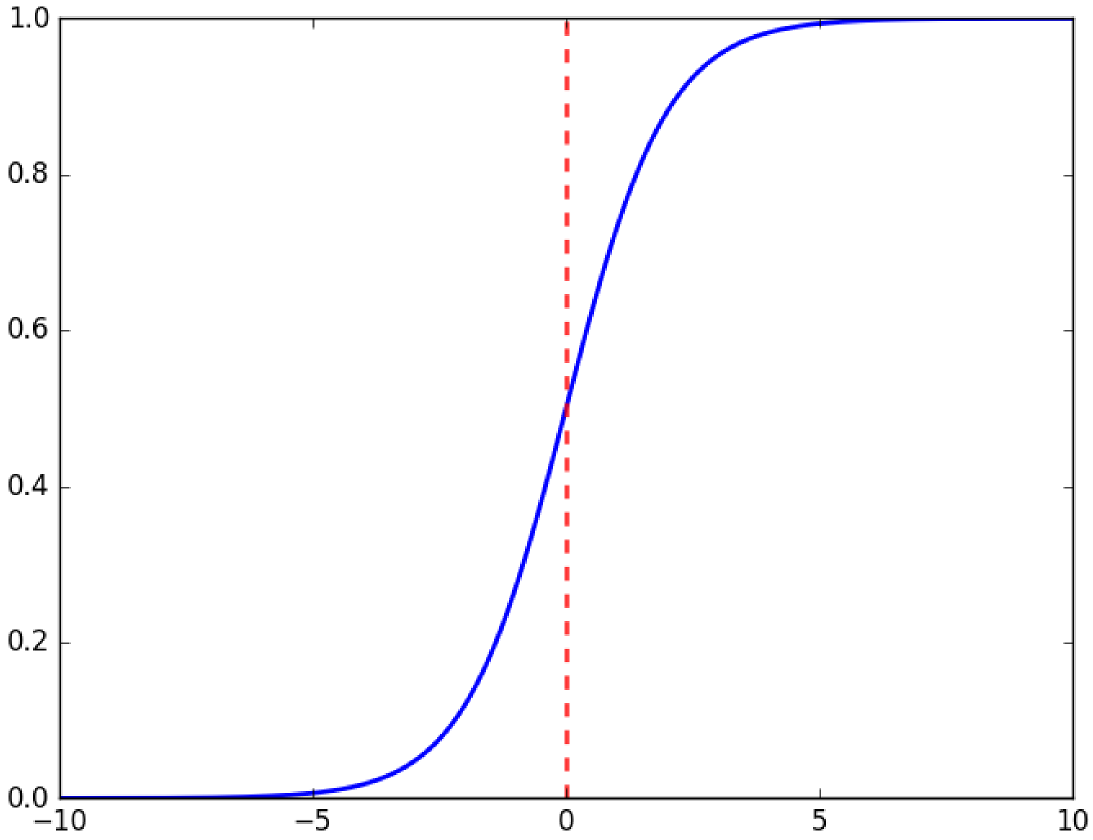
How can we constrain the output to be between -1 and +1?

```
import torch
import torch.nn as nn

class Net(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, 1)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Binary classification

To obtain the required binary classification, we pass the output f(x) through another function :

$$g(z) = 1/(1 + e^{-z})$$



The graph of the sigmoid function

Binary classification

To obtain the required binary classification, we pass the output f(x) through another function :

$$g(z) = 1/(1 + e^{-z})$$

This function returns an output between 0 and 1. The binary classification is set as follows :

If ($g(z) \geq 0.5$) assign the input the positive class
Else assign the input to the negative class

Binary classification

To obtain the required binary classification, we pass the output f(x) through another function :

$$g(z) = 1/(1 + e^{-z})$$

This function returns an output between 0 and 1. The binary classification is set as follows :

If ($g(z) \geq 0.5$) assign the input the positive class
Else assign the input to the negative class

But what do we do in the multi-class classification ?

Multi-class classification : the softmax functic

In the case of multi-class classification, we use the softmax activation function.
Suppose that we have k classes then the softmax activation function is define by :

$$\operatorname{softmax}(z)_i = \frac{\exp(z_i)}{\sum_{l=1}^{k} \exp(z_l)}$$

Here z_i represents the ith element of the input to softmax, which corresponds to class i.

Multi-class classification : the softmax functic

In the case of multi-class classification, we use the softmax activation function.
Suppose that we have k classes then the softmax activation function is define by :

$$\text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_{l=1}^{k} \exp(z_l)}$$

Here z_i represents the ith element of the input to softmax, which corresponds to class i.
The result is a vector containing the probabilities that sample x belong to each class.

Multi-class classification : the softmax functic

In the case of multi-class classification, we use the softmax activation function.
Suppose that we have k classes then the softmax activation function is define by :

$$\text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_{l=1}^{k} \exp(z_l)}$$

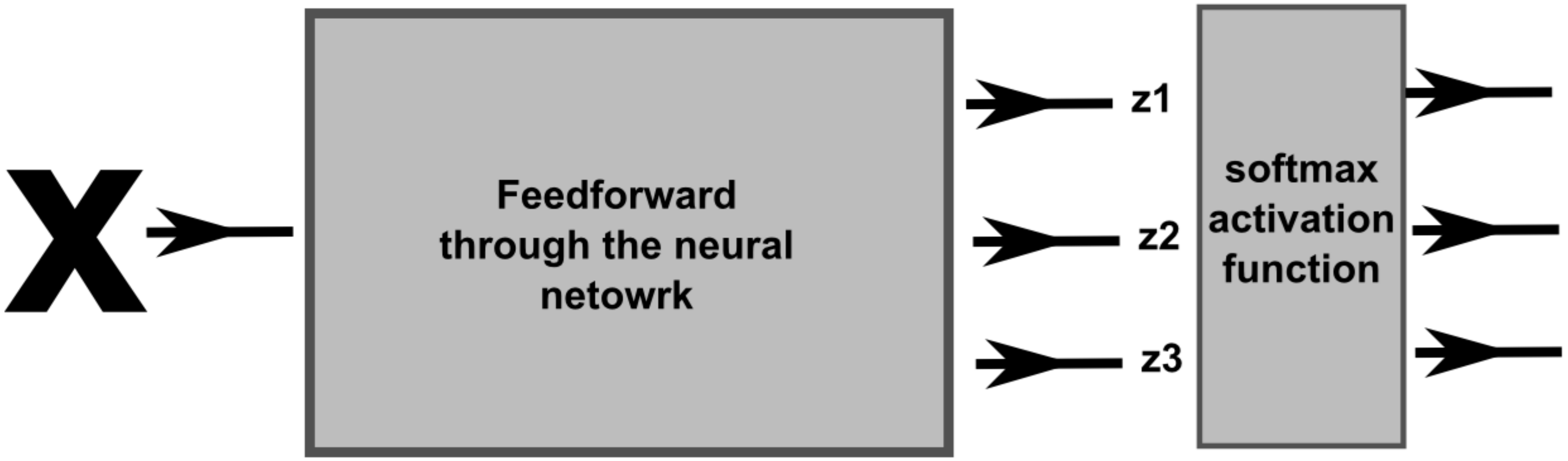
Here z_i represents the ith element of the input to softmax, which corresponds to class i.
The result is a vector containing the probabilities that sample x belong to each class.
The output is the class with the highest probability.

Multi-class classification : the softmax functic

In the case of multi-class classification, we use the softmax activation function.
Suppose that we have k classes then the softmax activation function is define by :

$$\mathrm{softmax}(z)_i = \frac{\exp(z_i)}{\sum_{l=1}^{k} \exp(z_l)}$$

Here z_i represents the ith element of the input to softmax, which corresponds to class i.
The result is a vector containing the probabilities that sample x belong to each class.
The output is the class with the highest probability.



The softmax function in Python

The softmax function is a mathematical function used to convert a vector of real numbers into a probability distribution.

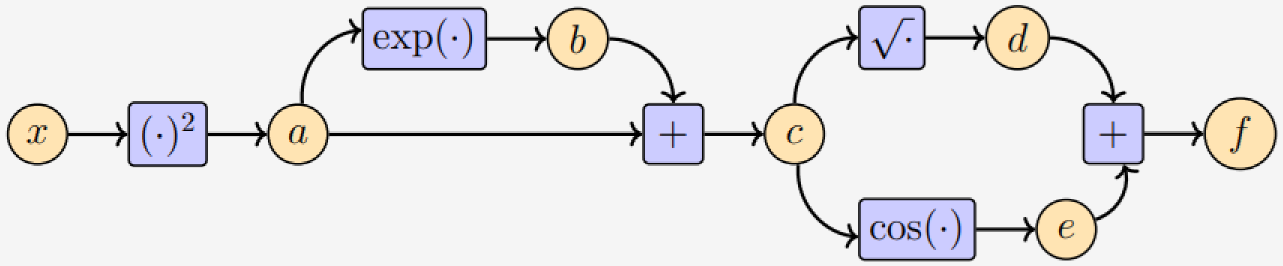
It takes an input vector and returns another vector of the same length, where each element is transformed to a value between 0 and 1, representing the probability of that element being selected. In simple terms, the softmax function normalizes the input vector and makes it easier to interpret as probabilities. Here's a Python example:

```
import numpy as np

def softmax(x):
    exp_values = np.exp(x)
    probabilities = exp_values / np.sum(exp_values)
    return probabilities

input_vector = np.array([2.0, 1.0, 0.5])
output_vector = softmax(input_vector)
print(output_vector)
[0.62842832 0.2312239  0.14034778]
```

What is a computational graph ?

$$f(x) = \sqrt{x^2 + \exp(x^2)} + \cos\left(x^2 + \exp(x^2)\right)$$

$$
\begin{aligned}
a &= x^2\,, \\
b &= \exp(a)\,, \\
c &= a + b\,, \\
d &= \sqrt{c}\,, \\
e &= \cos(c)\,, \\
f &= d + e\,.
\end{aligned}
$$



Computation graph of f

Image source

https://mml-book.github.io/book/mml-book.pdf

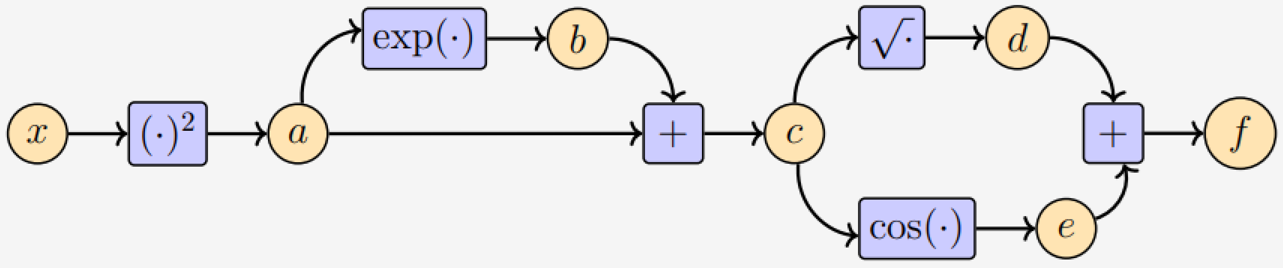
What is a computational graph ?

- A computational graph, also known as a computational or directed acyclic graph (DAG), is a directed graph that represents a computational process or a sequence of computations.

- It is a graph structure where nodes represent operations or computations, and directed edges represent dependencies between these operations.

$$f(x) = \sqrt{x^2 + \exp(x^2)} + \cos\left(x^2 + \exp(x^2)\right)$$

$$\begin{aligned} a &= x^2\,, \\ b &= \exp(a)\,, \\ c &= a + b\,, \\ d &= \sqrt{c}\,, \\ e &= \cos(c)\,, \\ f &= d + e\,. \end{aligned}$$



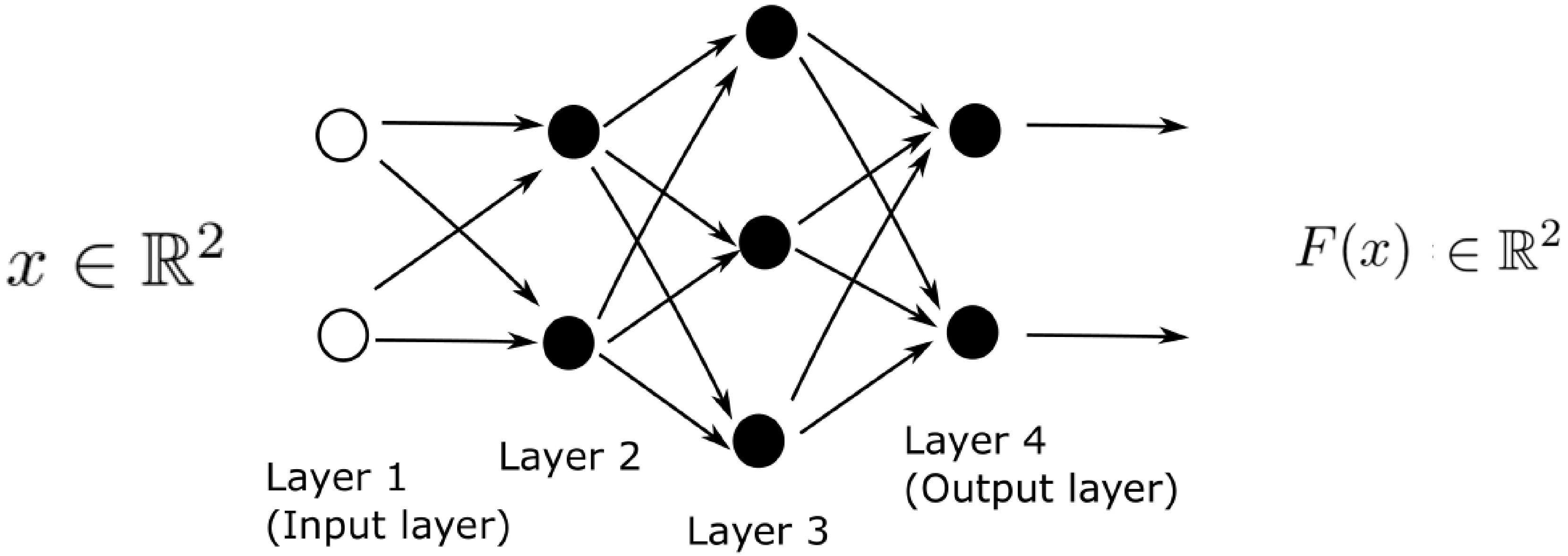
Computation graph of f

- Note: yellow nodes in the graph here are placeholders and not really part of the computational graph. They get executed when we insert a certain input to the graph

Image source

https://mml-book.github.io/book/mml-book.pdf

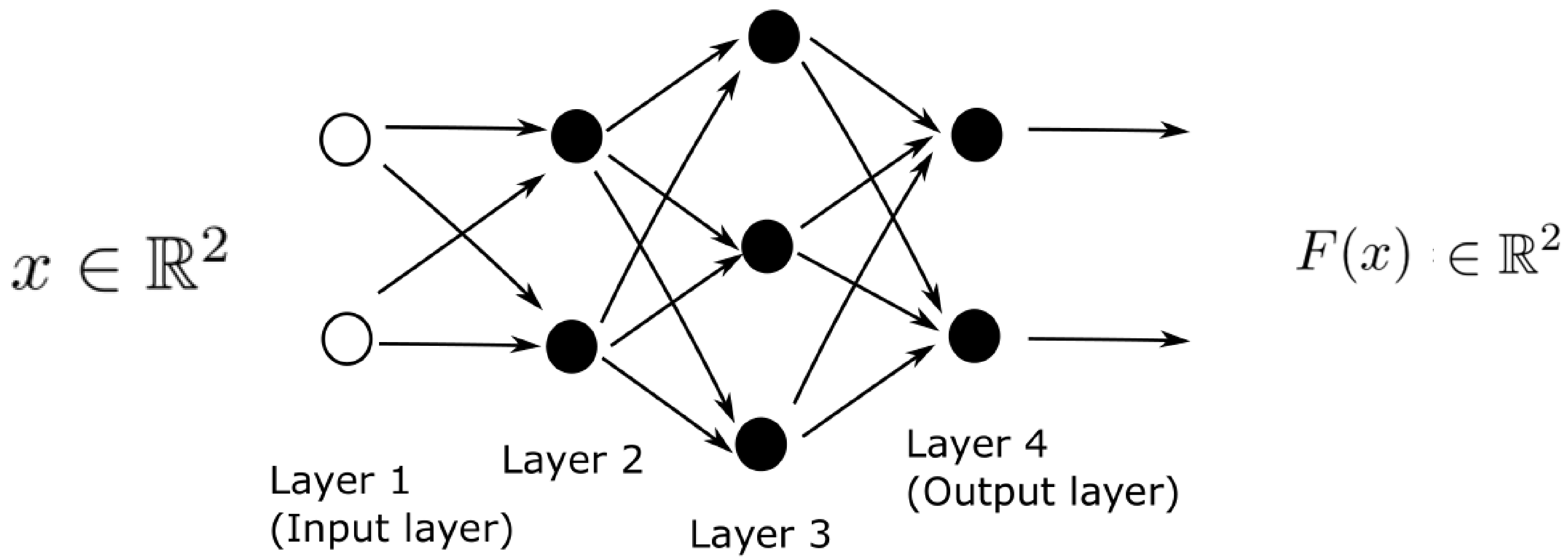
Neural Networks are computational graphs



Neural networks can be considered as computational graphs.

Why this is a useful fact? Modern DL packages such as tensorflow and pytorch use this fact for automatic differentiation.

Neural Networks are computational graphs

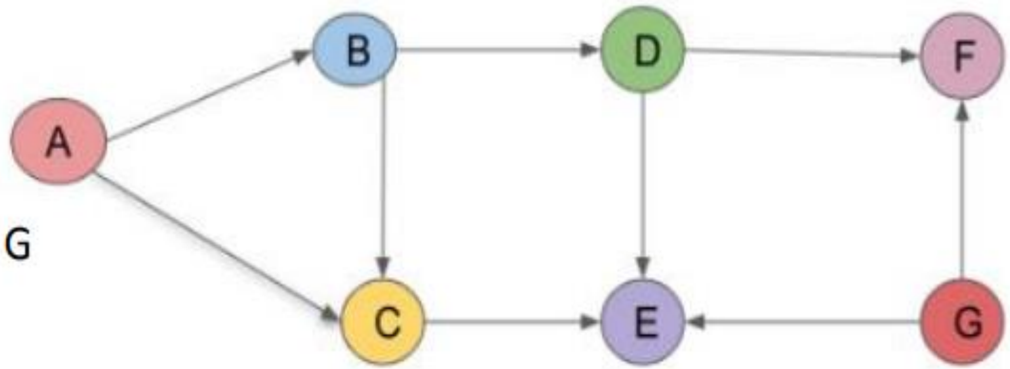


Key fact : feedforward computation of a NN is defined to be the computations that one executes on a computational graph that defines that network given an input and a topological order of the nodes of the computational graph of a NN

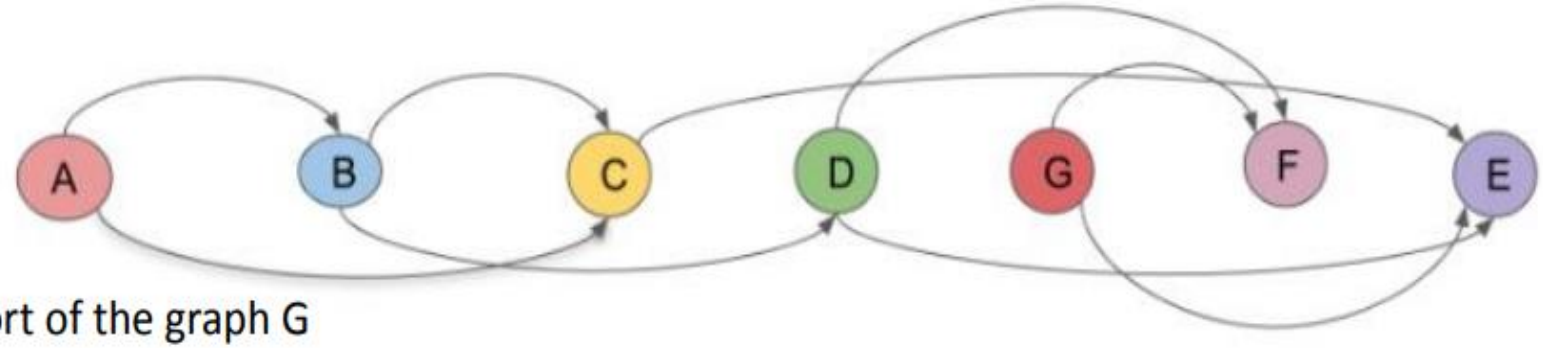
Recall topological sort

Recall that a topological soft of a DAG is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering.

Example



G



A topological sort of the graph G

Backprop in nutshell : Example

$$f(x, y, z) = (x + y)\max(y, z)$$
$$x = 2, y = 1, z = 0$$

Forward prop step

$$a = x + y$$
$$b = \max(y, z)$$
$$f = ab$$

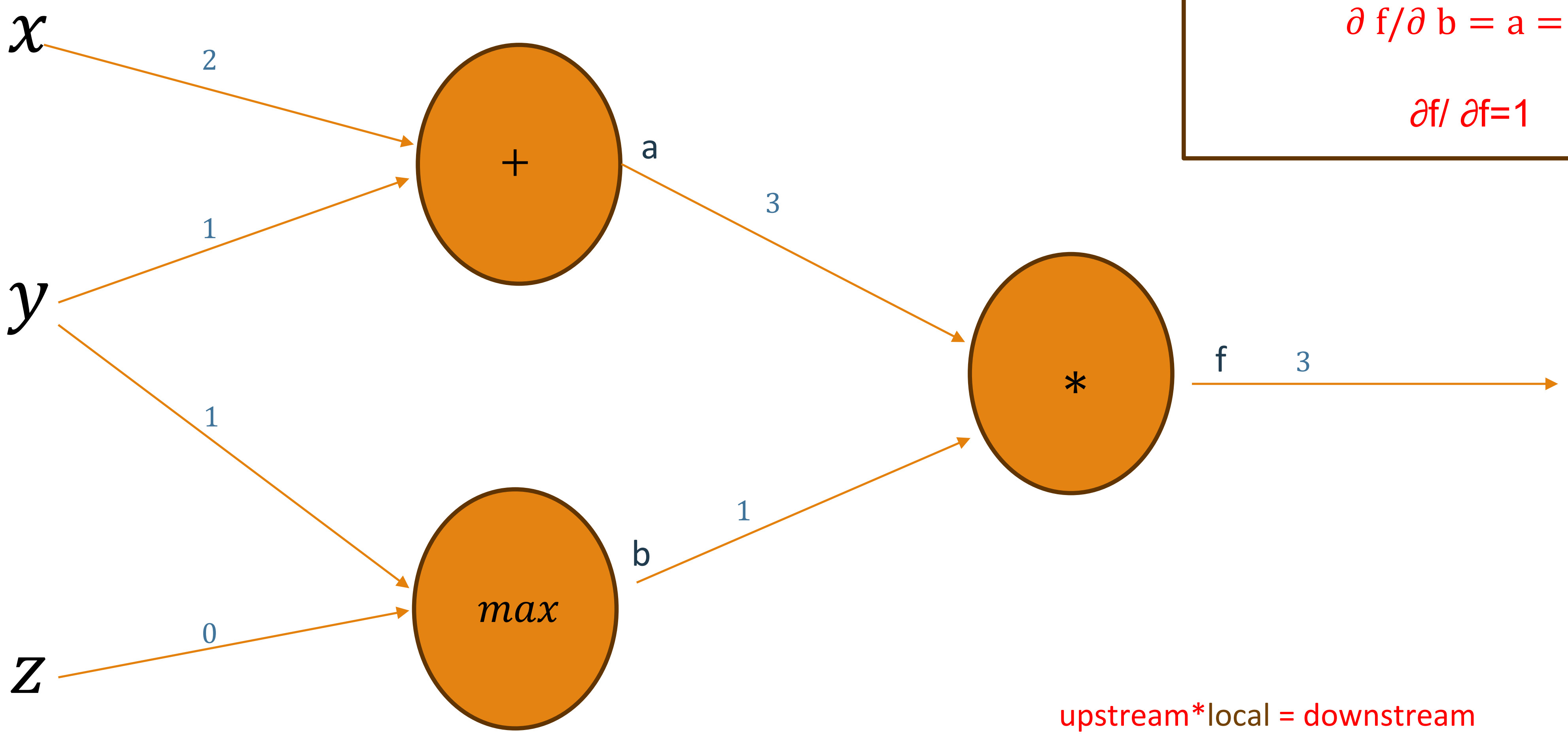
Back prop step (local gradients)

$\partial a/\partial x = 1, \partial a/\partial y = 1$

$\partial b/\partial y = \mathbf{1}(y>z)$, $\partial b/\partial z = \mathbf{1}(z>y) = 0$

$\partial f/\partial a = b = 3,$
$\partial f/\partial b = a = 1$

$\partial f/\partial f = 1$



upstream*local = downstream

Backprop in nutshell : Example

$$f(x,y,z) = (x+y)\max(y,z)$$
$$x = 2, y = 1, z = 0$$

Forward prop step

$$a = x + y$$
$$b = \max(y,z)$$
$$f = ab$$

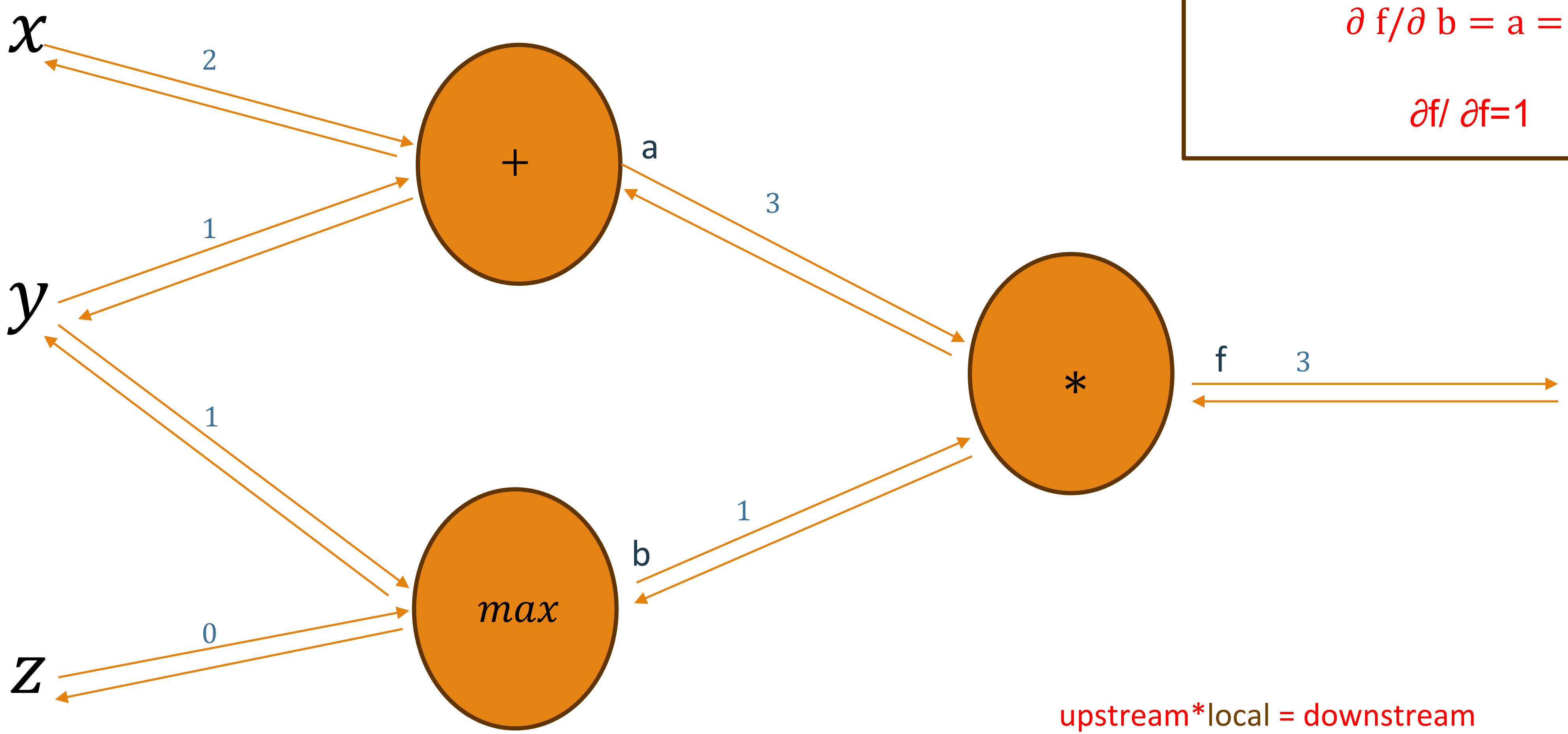
Back prop step (local gradients)

$\partial a/\partial x = 1, \partial a/\partial y = 1$

$\partial b/\partial y = \mathbf{1}(y>z)$, $\partial b/\partial z = \mathbf{1}(z>y) = 0$

$\partial f/\partial a = b = 3,$
$\partial f/\partial b = a = 1$

$\partial f/\partial f = 1$



upstream*local = downstream

Backprop in nutshell : Example

$$f(x,y,z) = (x+y)\max(y,z)$$
$$x = 2, y = 1, z = 0$$

Forward prop step

$$a = x + y$$
$$b = \max(y,z)$$
$$f = ab$$

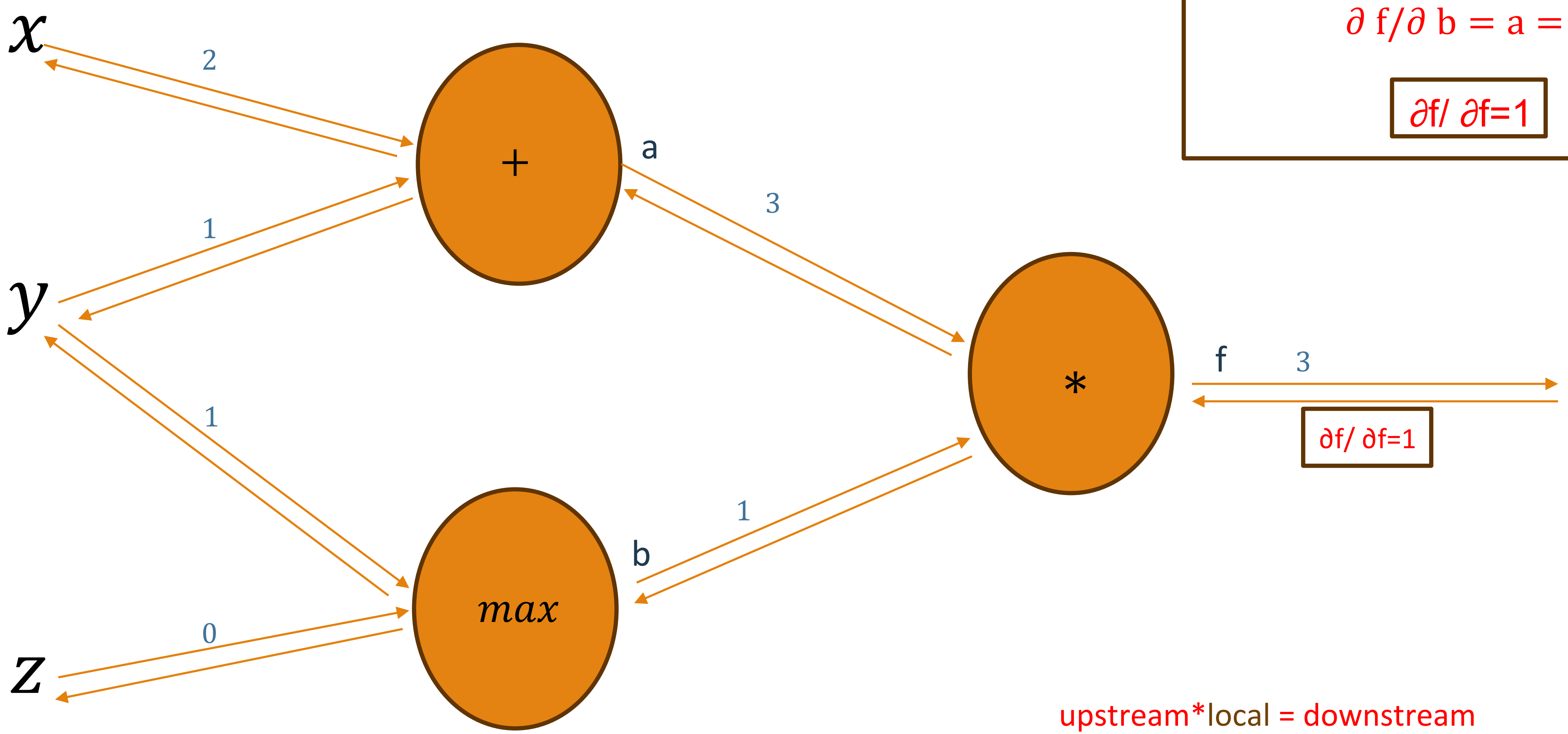
Back prop step (local gradients)

$\partial a/\partial x = 1, \partial a/\partial y = 1$

∂b/ ∂y=**1**(y>z), ∂b/∂z=**1**(z>y)=0

$\partial f/\partial a = b = 3,$
$\partial f/\partial b = a = 1$

$\boxed{\partial f/ \partial f=1}$



upstream*local = downstream

Backprop in nutshell : Example

$$f(x,y,z) = (x+y)\max(y,z)$$
$$x = 2, y = 1, z = 0$$

Forward prop step

$$\begin{aligned} a &= x + y \\ b &= \max(y,z) \\ f &= ab \end{aligned}$$

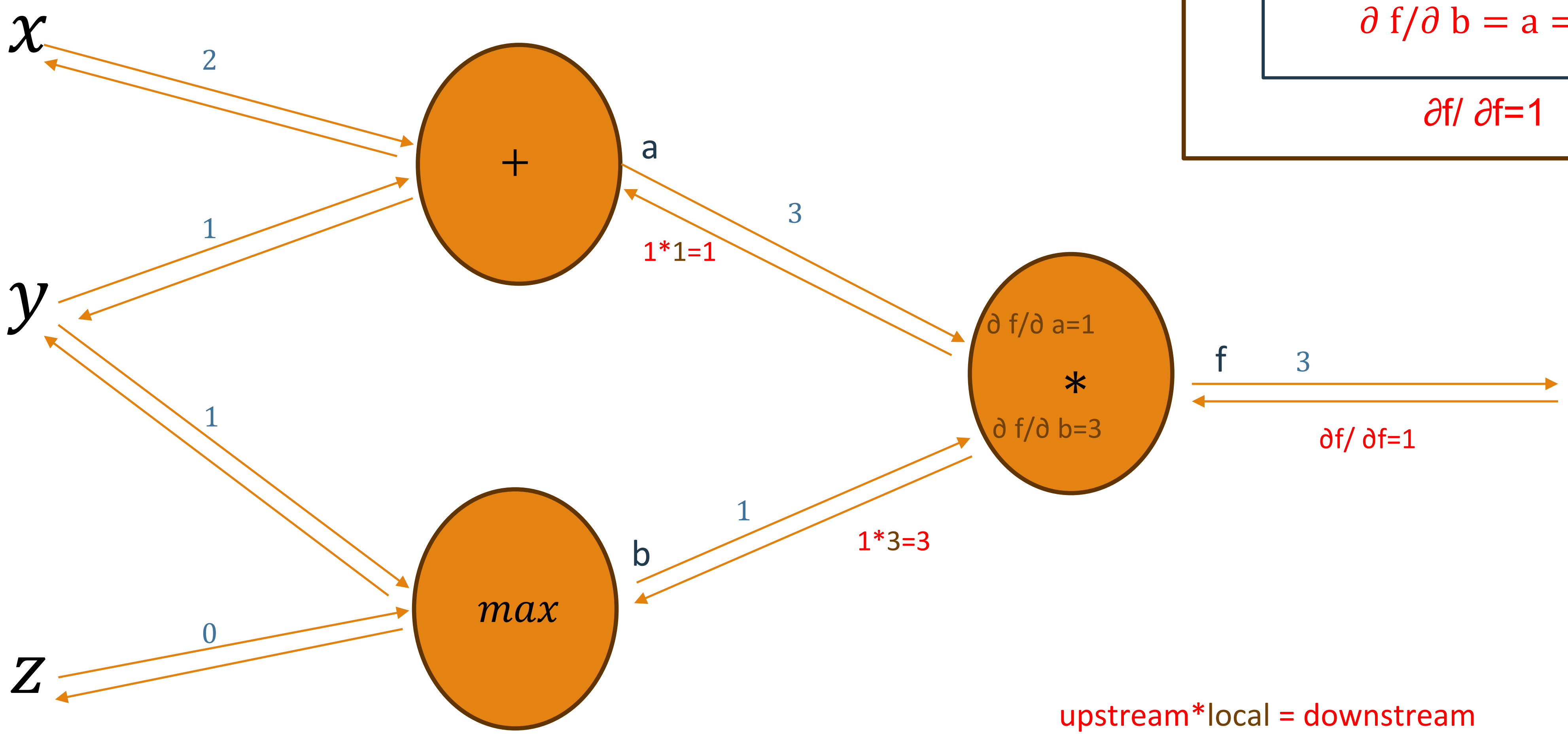
Back prop step (local gradients)

$\partial a/\partial x = 1, \partial a/\partial y = 1$

$\partial b/\partial y = \mathbf{1}(y>z)$, $\partial b/\partial z = \mathbf{1}(z>y) = 0$

$\partial f/\partial a = b = 3,$
$\partial f/\partial b = a = 1$

$\partial f/\partial f = 1$



upstream*local = downstream

Backprop in nutshell : Example

$$f(x,y,z) = (x+y)\max(y,z)$$
$$x = 2, y = 1, z = 0$$

Forward prop step

$$a = x + y$$
$$b = \max(y,z)$$
$$f = ab$$

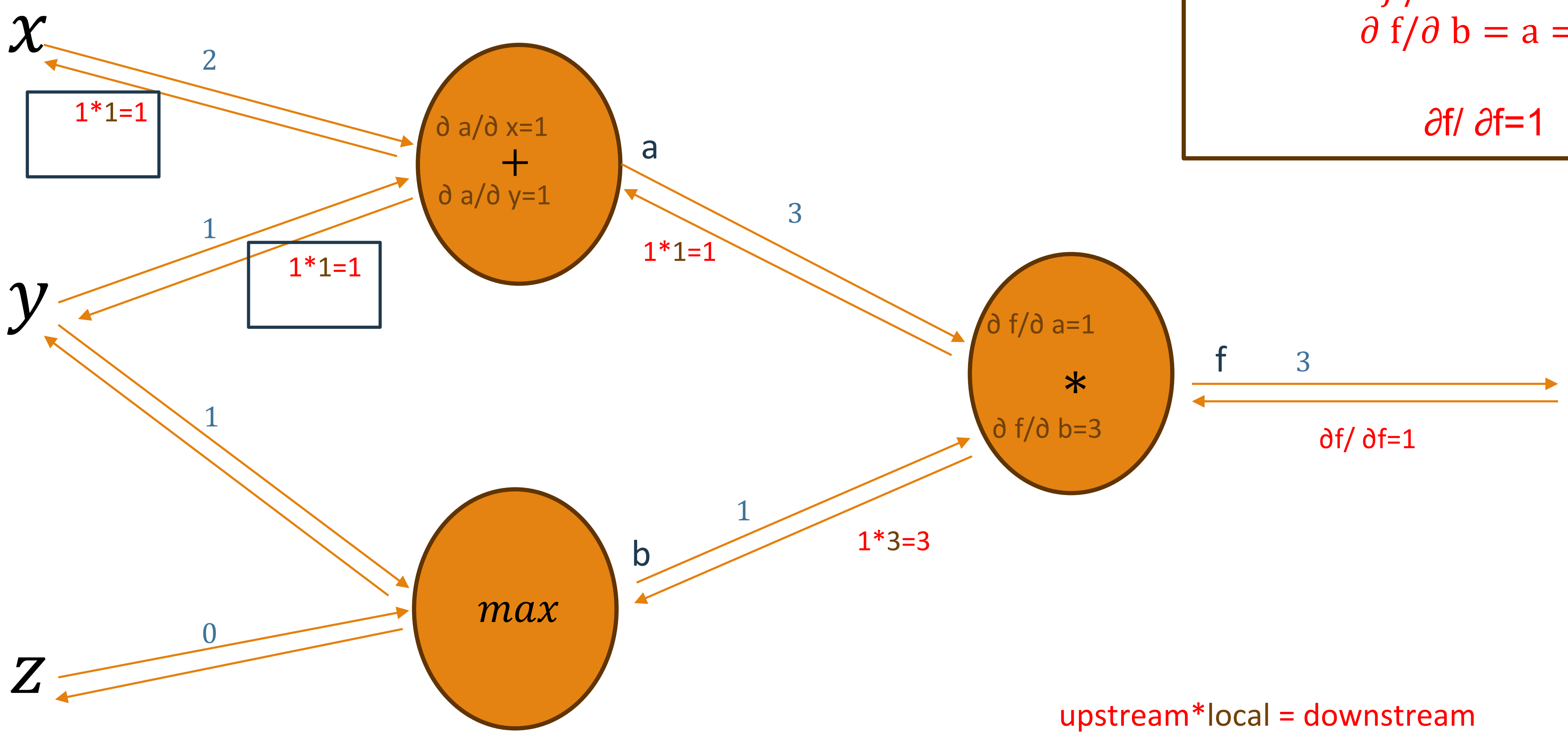
Back prop step (local gradients)

$$\boxed{\partial a/\partial x = 1, \partial a/\partial y = 1}$$

∂b/ ∂y=**1**(y>z), ∂b/∂z=**1**(z>y)=0

$$\partial f/\partial a = b = 3,$$
$$\partial f/\partial b = a = 1$$

∂f/ ∂f=1



upstream*local = downstream

Backprop in nutshell : Example

$$f(x,y,z) = (x+y)\max(y,z)$$
$$x = 2, y = 1, z = 0$$

Forward prop step

$$a = x + y$$
$$b = \max(y,z)$$
$$f = ab$$

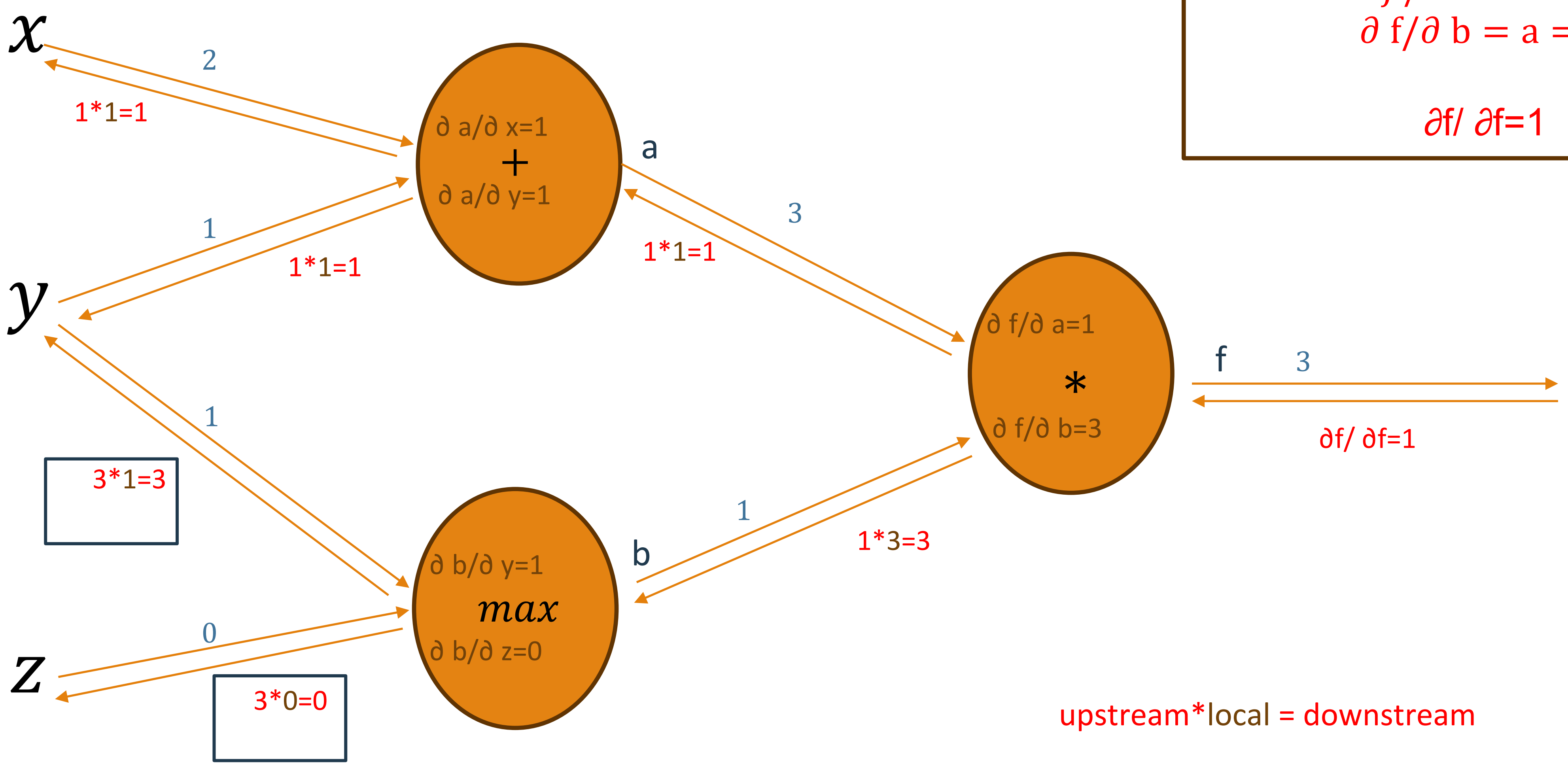
Back prop step (local gradients)

$\partial a/\partial x = 1, \partial a/\partial y = 1$

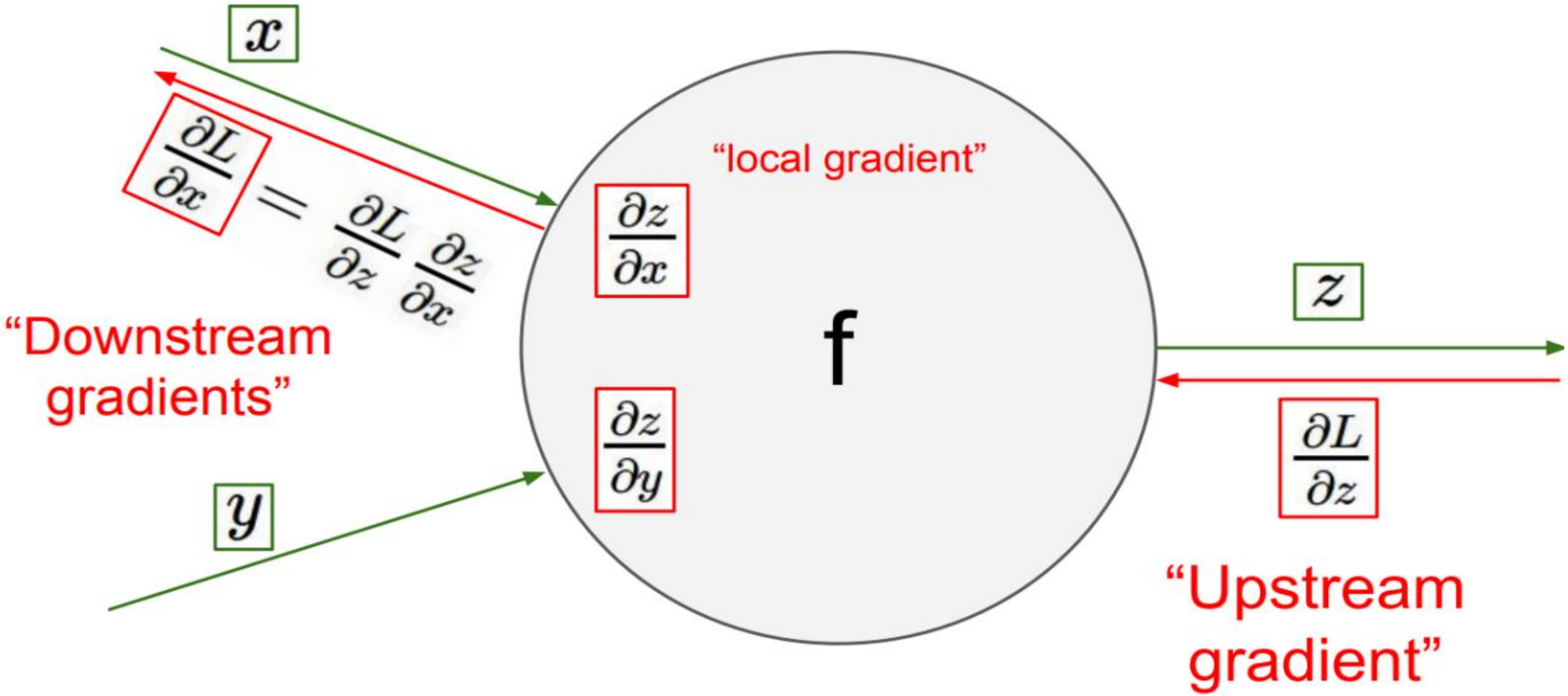
$\boxed{\partial b/ \partial y=\mathbf{1}(y>z),\ \partial b/\partial z=\mathbf{1}(z>y)=0}$

$\partial f/\partial a = b = 3,$
$\partial f/\partial b = a = 1$

$\partial f/ \partial f=1$



Backprop in nutshell



Backprop in nutshell

More general :

Automatic Differentiation – Reverse Mode (aka. Backpropagation)

Forward Computation

1. Write an **algorithm** for evaluating the function $y = f(\mathbf{x})$. The algorithm defines a **directed acyclic graph**, where each variable is a node (i.e. the "**computation graph**")
2. Visit each node in **topological order**.
   For variable u_i with inputs $v_1, \ldots, v_N$
   a. Compute $u_i = g_i(v_1, \ldots, v_N)$
   b. Store the result at the node

Backward Computation

1. **Initialize** all partial derivatives dy/du_j to 0 and $dy/dy = 1$.
2. Visit each node in **reverse topological order**.
   For variable $u_i = g_i(v_1, \ldots, v_N)$
   a. We already know dy/du_i
   b. Increment dy/dv_j by $(dy/du_i)(du_i/dv_j)$
   (Choice of algorithm ensures computing (du_i/dv_j) is easy)

Return partial derivatives dy/du_i for all variables

The approximation power of neural networks

Approximation Theorems using shallow NN

Let $\varphi : \mathbb{R} \rightarrow \mathbb{R}$ be a constant, bounded, and continuous function.

Consider summation of the form :

$$\sum_{i=1}^{N} v_i \varphi \left(w_i \; x + b_i \right)$$

Real numbers (the weights)

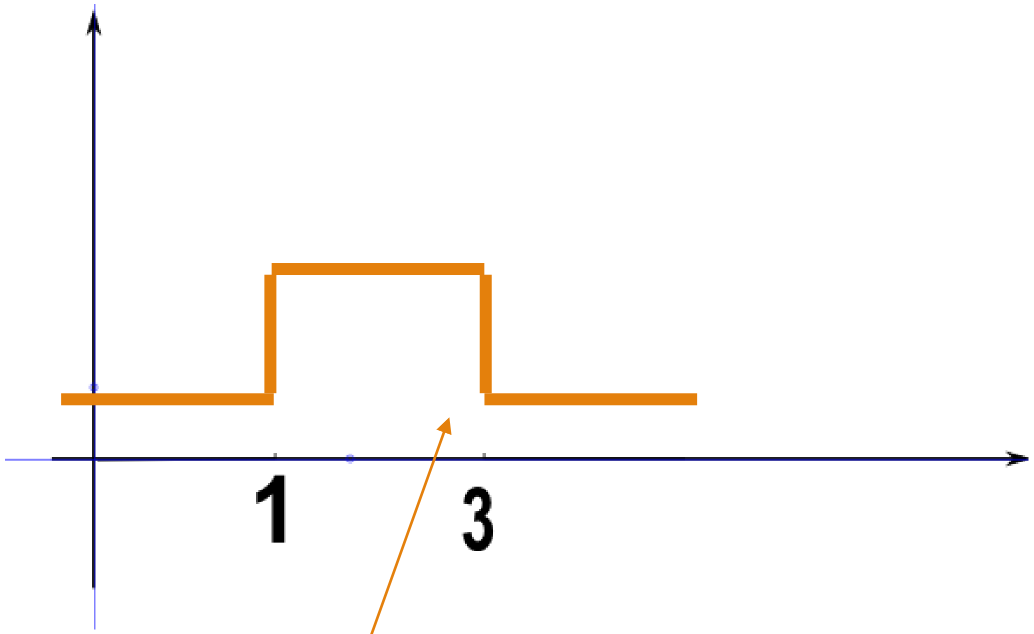
Approximation Theorems using shallow NN

Let $\varphi : \mathbb{R} \to \mathbb{R}$ be a constant, bounded, and continuous function.

Consider summation of the form :

$$\sum_{i=1}^{N} v_i \varphi \left(w_i \, x + b_i \right)$$

Real numbers (the weights)



Let φ(x)=1 when x>=0 and zero otherwise and consider: 0.5(φ (x-1) + φ (-x+3))

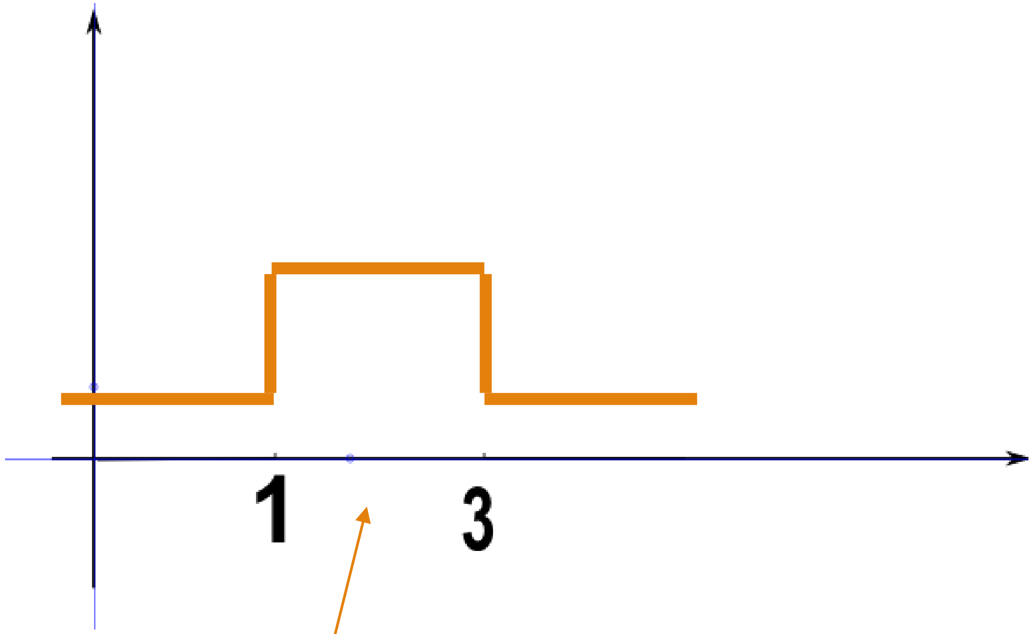
Approximation Theorems using shallow NN

Let $\varphi : \mathbb{R} \to \mathbb{R}$ be a constant, bounded, and continuous function.

Consider summation of the form :

$$\sum_{i=1}^{N} v_i \varphi \left(w_i \, x + b_i \right)$$

Real numbers (the weights)

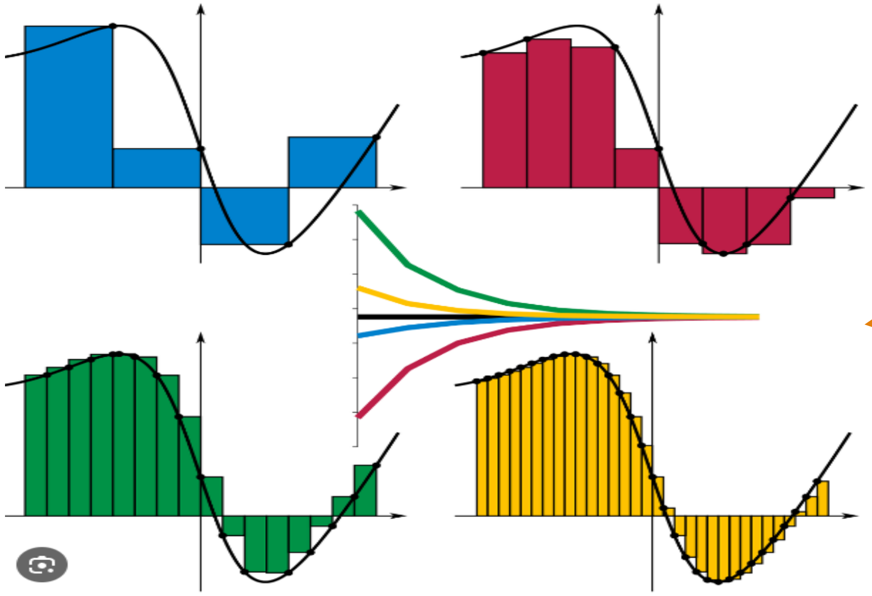


$0.5(\varphi (x-1) + \varphi (-x+3))$

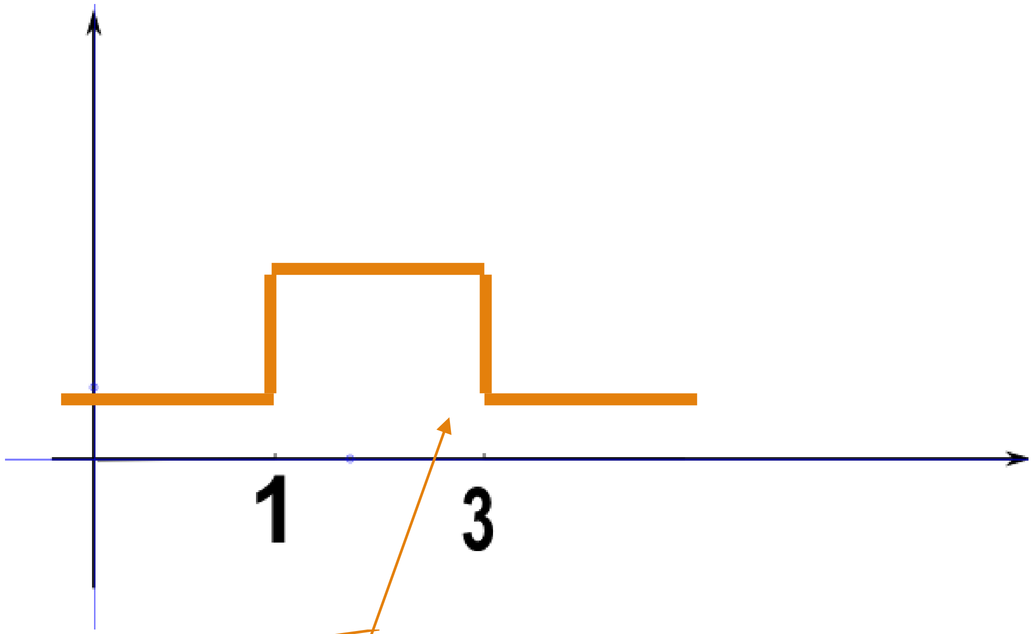
Let $\varphi(x)=1$ when $x>=0$ and zero otherwise and consider:

Can you imagine building more complex functions if we have more
Summations and maybe vary the weights ? –what are the functions you can build?

Approximation Theorems using shallow NN



https://en.wikipedia.org/wiki/Riemann_sum

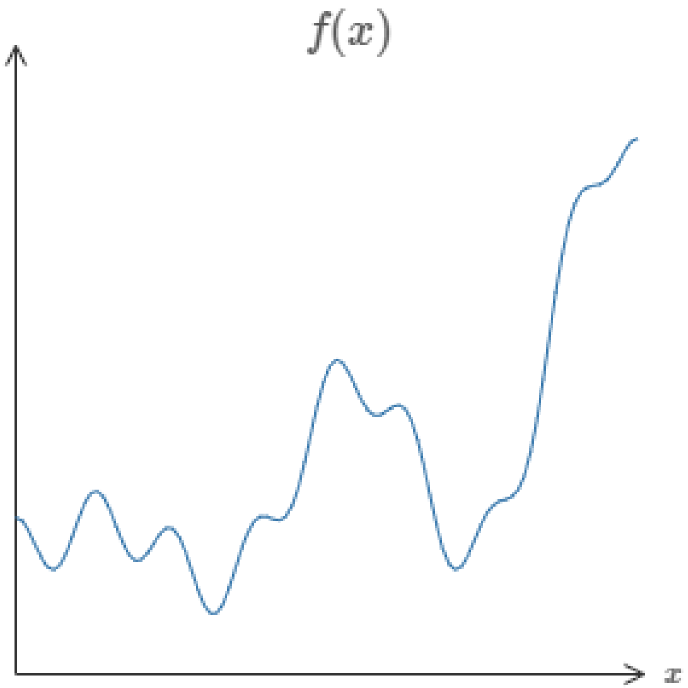


Familiar ? Riemann sum from Calc?

Approximation Theorems using shallow NN

Question : can you imagine building more complex functions if we have more summations and maybe vary the weights vi’s, wi’s, and bi’s ?
what are the functions you can build?

$$\sum_{i=1}^{N} v_i \varphi \left(w_i \, x + b_i \right)$$



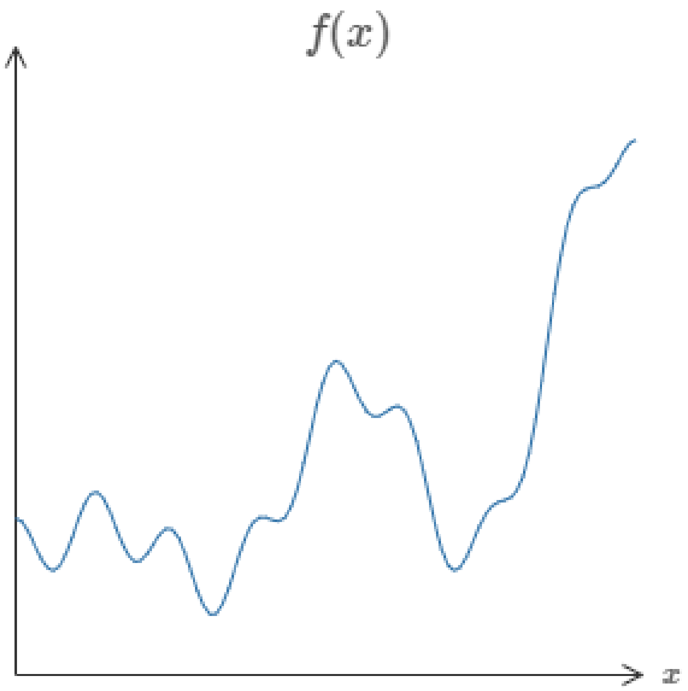
Given a function f as above, can
we find vi’s, wi’s, and bi’s such that the summation above is as close we like to f ?

This is the essence of the universal approximation theorem : it can always be done.

Approximation Theorems using shallow NN

Question : can you imagine building more complex functions if we have more summations and maybe vary the weights vi’s, wi’s, and bi’s ?
what are the functions you can build?

$$\sum_{i=1}^{N} v_i \varphi\left(w_i\, x + b_i\right)$$



Given a function f as above, can
we find vi’s, wi’s, and bi’s such that the summation above is as close we like to f ?

It turns out that the answer is yes as long as we are willing to increase N (increase number of kernels).
Lets see a few examples.

Approximation Theorems using shallow NN

It turns out that this theorem generalizes to higher dimension the same way. More precisely, the following summations:

$$\sum_{i=1}^{N} v_i \varphi\left(w_i^T x + b_i\right) \qquad \begin{array}{l} w_i \in \mathbb{R}^m \\ v_i, b_i \in \mathbb{R} \end{array}$$

can approximate any continuous real valued function on $[0,1]^m$.

Approximation Theorems using shallow NN

It turns out that this theorem generalizes to higher dimension the same way. More precisely, the following summations:

$$\sum_{i=1}^{N} v_i \varphi\left(w_i^T x + b_i\right) \qquad \begin{aligned} w_i &\in \mathbb{R}^m \\ v_i, b_i &\in \mathbb{R} \end{aligned}$$

can approximate any continuous real valued function on $[0,1]^m$.

Universal approximation theorem. Let $\varphi : \mathbb{R} \to \mathbb{R}$ be a nonconstant, bounded, and continuous function (called the *activation function*). Let I_m denote the *m*-dimensional unit hypercube $[0,1]^m$. The space of real-valued continuous functions on I_m is denoted by $C(I_m)$. Then, given any $\varepsilon > 0$ and any function $f \in C(I_m)$, there exist an integer N, real constants $v_i, b_i \in \mathbb{R}$ and real vectors $w_i \in \mathbb{R}^m$ for $i = 1, \ldots, N$, such that we may define:

$$F(x) = \sum_{i=1}^{N} v_i \varphi\left(w_i^T x + b_i\right)$$

as an approximate realization of the function f; that is,

$$|F(x) - f(x)| < \varepsilon$$

https://en.wikipedia.org/wiki/Universal_approximation_theorem