ANFÄNGERPRAKTIKUM
NEURAL NETWORKS FROM SCRATCH
XOR AND MLP

Hendrik Borras, Franz Kevin Stehle
hendrik.borras@ziti.uni-heidelberg.de, kevin.stehle@ziti.uni-heidelberg.de
HAWAII Group, Institute of Computer Engineering
Heidelberg University

RECAP

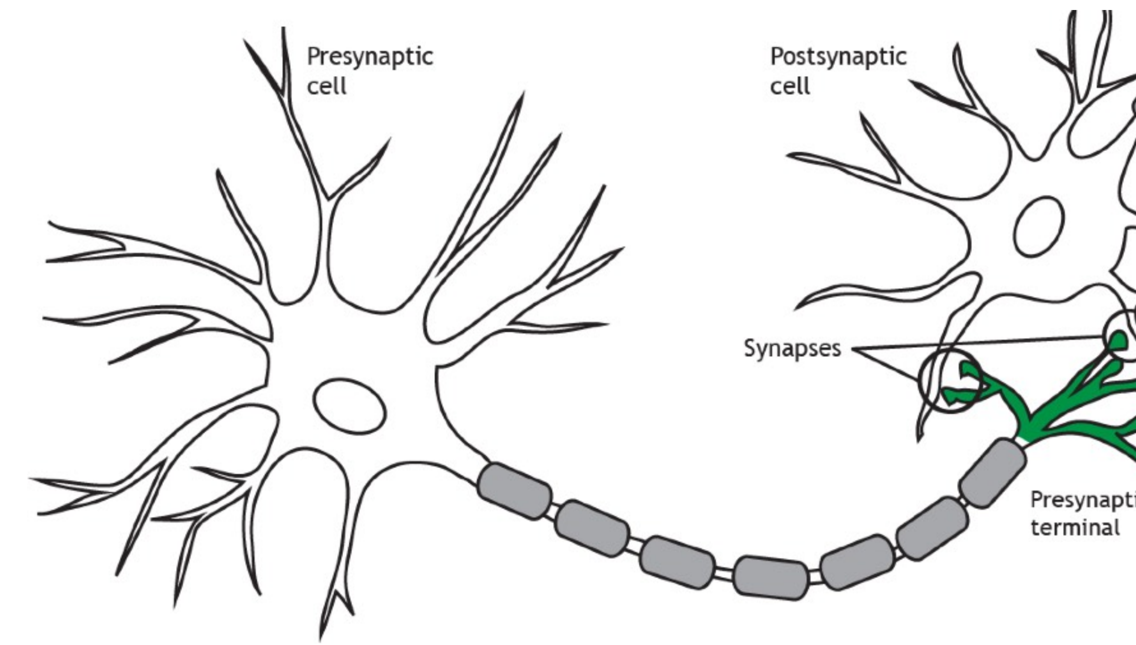
- Linear regression example
  - Parameters and their training
  - Model selection
  - Overfitting and regularization
- LRs are not universal approximators
  - But Artificial Neural Networks (ANNs) are
  - Subject of today

ARTIFICIAL NEURAL NETWORKS

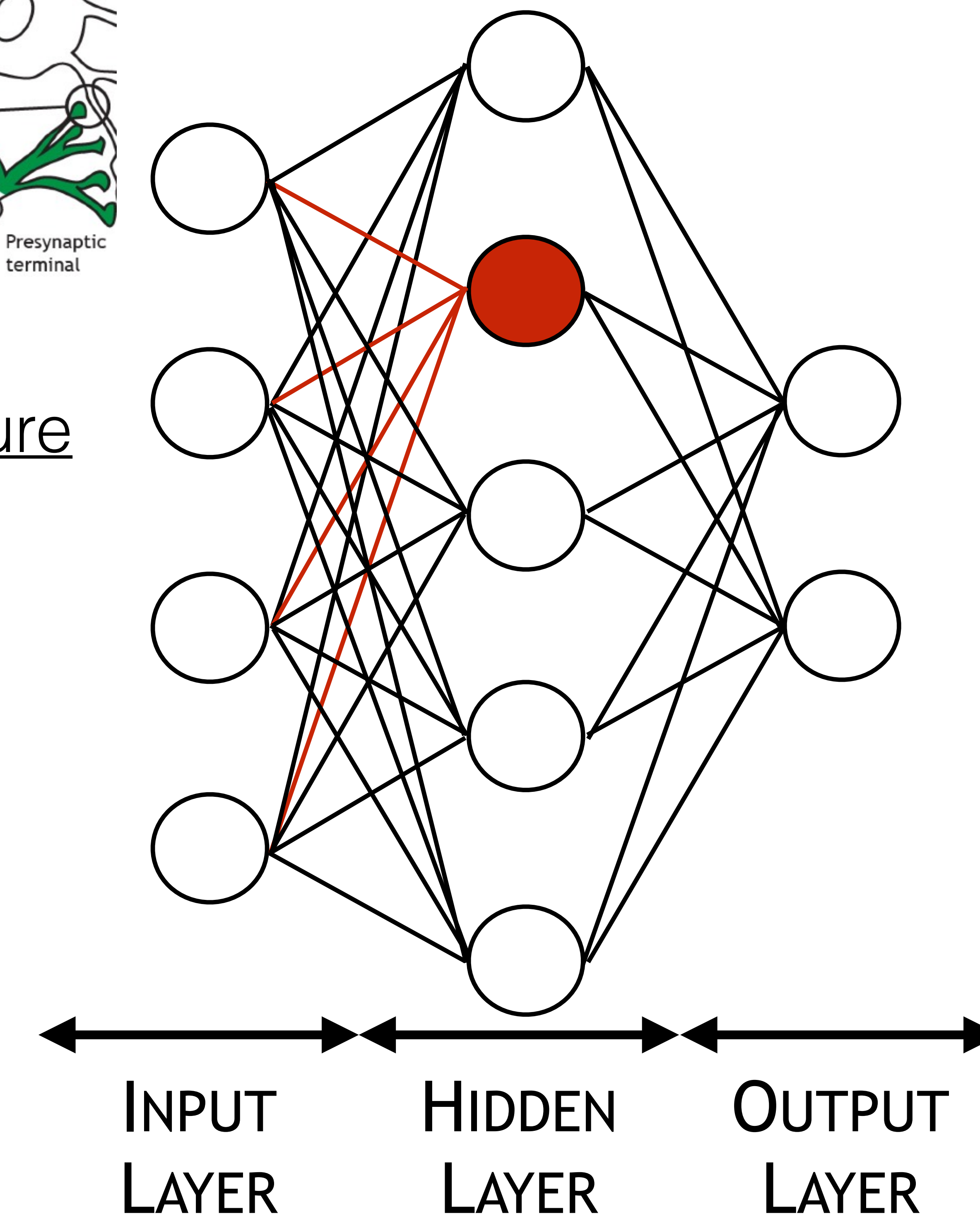
ARTIFICIAL NEURAL NETWORKS (ANNS)

Kind of inspired by biology

- Term “biologically inspired” is often a complaint
- != spiking neural networks
- c.f. “non-differentiable”



Biological Synapse Structure



More complex problems require more complex models

- Informal term of “model capacity”
- Curse of dimensionality: one pixel = one dimension

“Universal approximation theorems imply that neural networks can represent a wide variety of interesting functions when given appropriate weights”

Deep neural networks (DNNs) = increasing number of hidden layers

MULTI-LAYER PERCEPTRON (MLP)

E.g.: MNIST: 28x28 images in 10 classes => MLP with 28x28 inputs ($\mathbf{x}$) & 10 outputs ($\mathbf{y}$)

For neuron k of a given layer

$$y_k = f\left(\sum_j (w_{k,j} \cdot x_j) + b_k \right)$$

f: non-linear function (sigmoid, reLU, …)

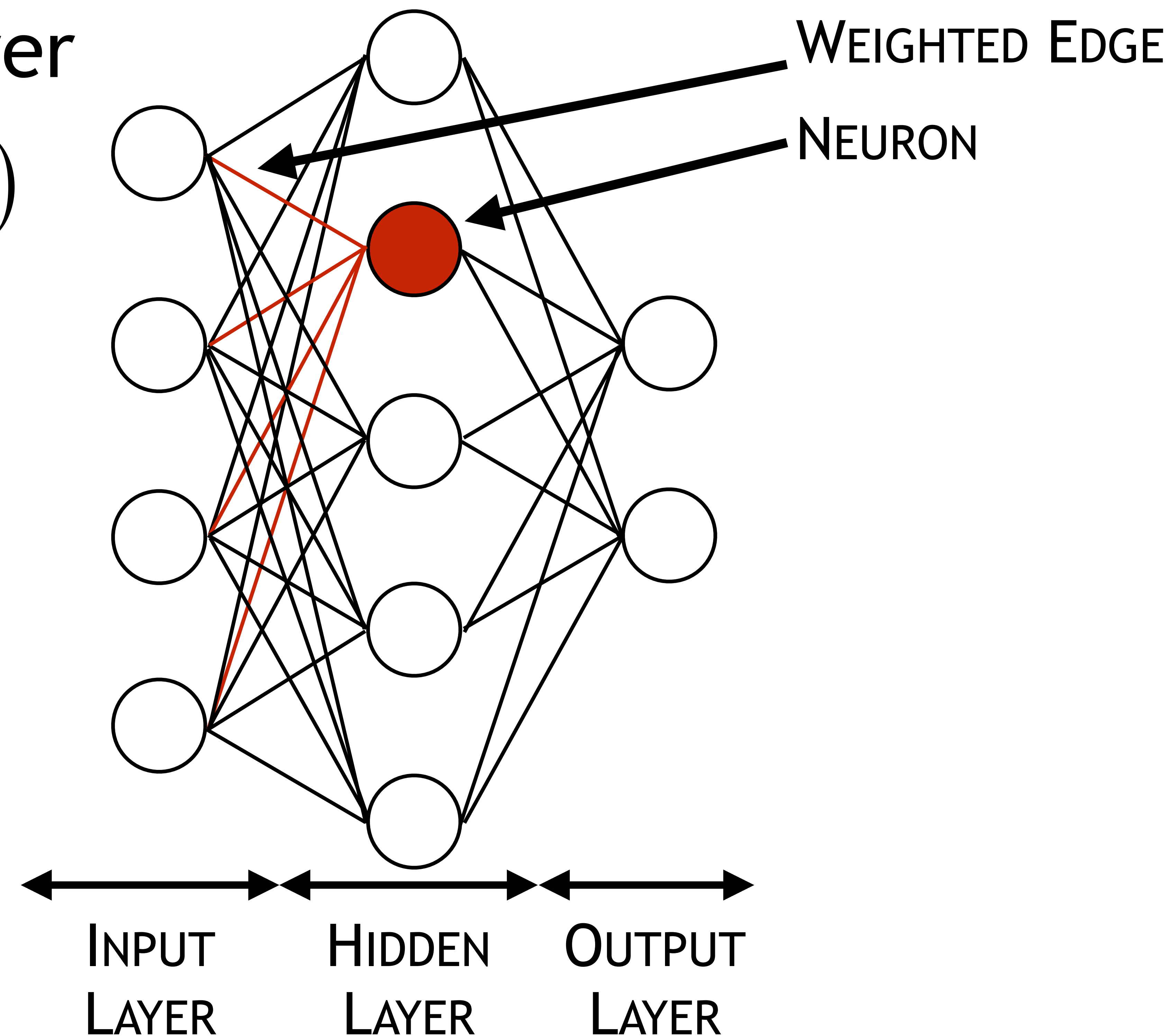
$\mathbf{W}$: weight matrix

$\mathbf{x}$: activation vector

$\mathbf{b}$: bias vector (hidden)

Vector notation for layer l

$$\mathbf{x}_l = f(\mathbf{W}_l \cdot \mathbf{x}_{l-1} + \mathbf{b}_l)$$



VECTOR AND MATRIX NOTATION

Matrix $\mathbf{W}$, composed of elements $w_{k,j}$

Matrix = bold uppercase

Matrix element $w_{k,j}$ has row k, column j

Vector $\mathbf{x}$, composed of elements x_i

Vector = bold lowercase

Vectors are vertical, use $\mathbf{x}^T$ for horizontal vectors

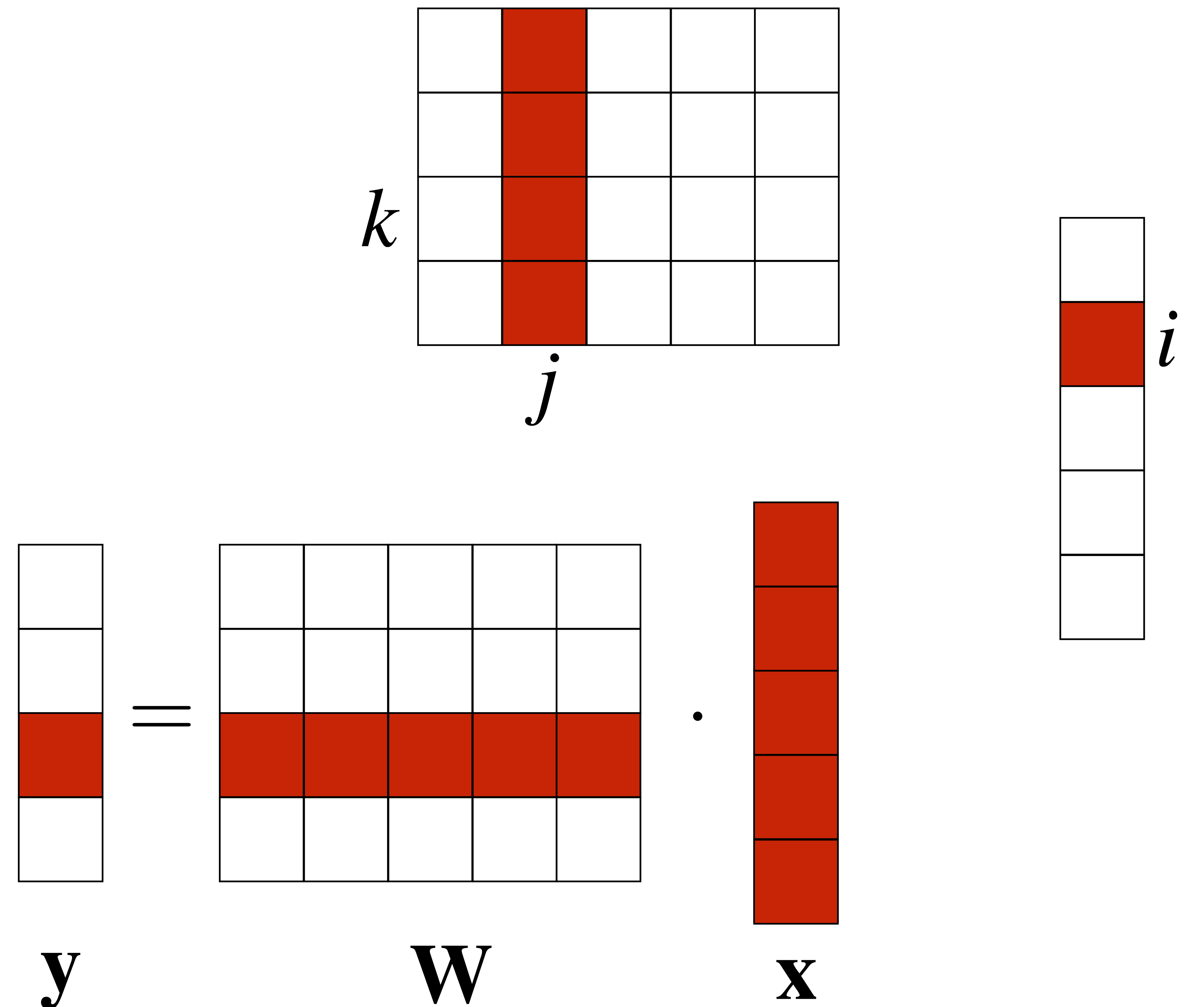
Matrix-vector multiplication

Length of the vector equals the number of columns of the matrix

$y_k = \sum_j (w_{k,j} \cdot x_j)$, resp. $\mathbf{y} = \mathbf{W} \cdot \mathbf{x}$

Vector-vector multiplication (dot product)

$a = \sum_j (b_j \cdot c_j)$, resp. $a = \mathbf{b} \cdot \mathbf{c}^T = \mathbf{c} \cdot \mathbf{b}^T$



MULTI-LAYER PERCEPTRON (MLP)

E.g.: MNIST: 28x28 images in 10 classes => MLP with 28x28 inputs ($\mathbf{x}$) & 10 outputs ($\mathbf{y}$)

For neuron k of a given layer

$$y_k = f\Big(\sum_j (w_{k,j} \cdot x_j) + b_k\Big)$$

f: non-linear function (sigmoid, reLU, …)

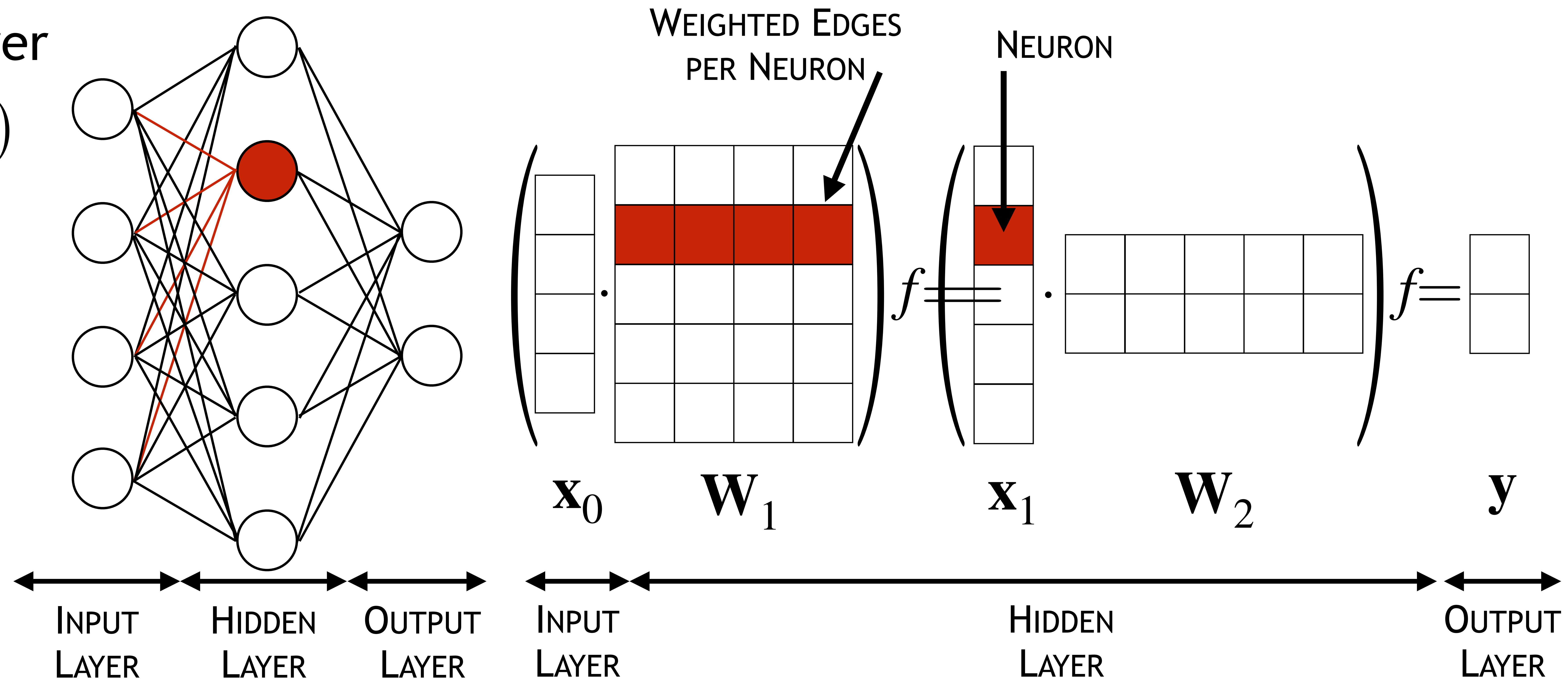
$\mathbf{W}$: weight matrix

$\mathbf{x}$: activation vector

$\mathbf{b}$: bias vector (hidden)

Vector notation for layer l

$$\mathbf{x}_l = f(\mathbf{W}_l \cdot \mathbf{x}_{l-1} + \mathbf{b}_l)$$



FORWARD PROP ON ONE SLIDE

(Deep) Neural Networks: L stacked processing units, where each unit computes an activation function

$x_l = f(\mathbf{W}_l \oplus \mathbf{x}_{l-1} + \mathbf{b}_l)$, for nonlinear activation function $f(\cdot)$, linear operation $\oplus$, and weight matrix $\mathbf{W}$, input activations $\mathbf{x}$, and bias $\mathbf{b}$ of layer l

Bias vector $\mathbf{b}$ is usually encoded in the weight matrix $\mathbf{W}$ by introducing another activation element which is fixed to (e.g., $x_0 = 1$)

Then a complete MLP with L layers is

$$\mathbf{y}(\mathbf{W}, \mathbf{x}_0) = \mathbf{x}_L = \mathbf{W}_L \oplus f(\mathbf{W}_{L-1} \oplus f(\ldots \oplus f(\mathbf{W}_1 \oplus \mathbf{x}_0))\ldots)$$

Reminder: *"Universal approximation theorems imply that neural networks can represent a wide variety of interesting functions when given appropriate weights"*

EXAMPLE NONLINEARITIES

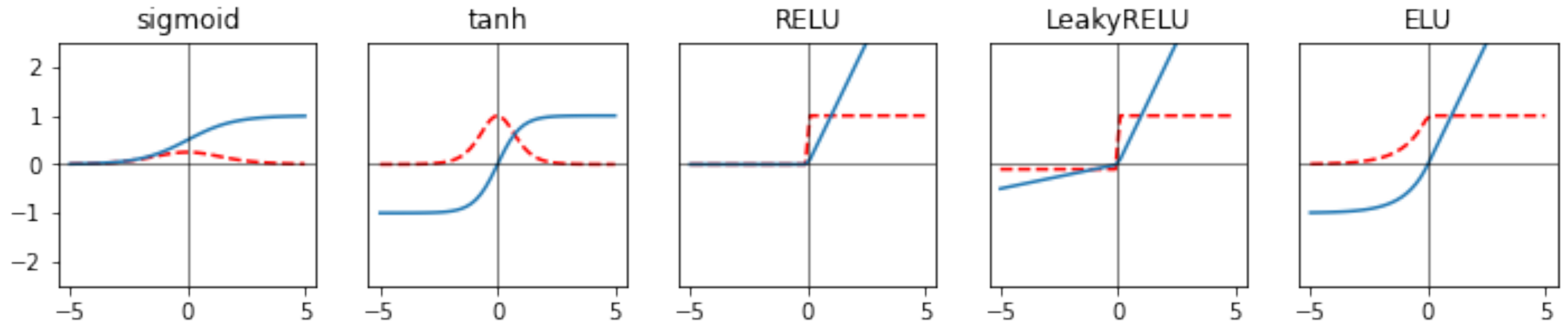
sigmoid: $f(x) = \frac{1}{1 + e^{-x}}$ => output in range $[0,1]$

tanh: $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ => output in range $[-1,1]$

ReLU: $f(x) = \max(x,0)$ => no negative output

LeakyReLU: $f(x) = \begin{cases} x; x \geq 0 \\ \alpha x; x < 0 \end{cases}$ => no clamping to zero for negative inputs

ELU: $f(x) = \begin{cases} x; x \geq 0 \\ e^x - 1; x < 0 \end{cases}$ => smoother gradient



EXAMPLE NONLINEARITIES

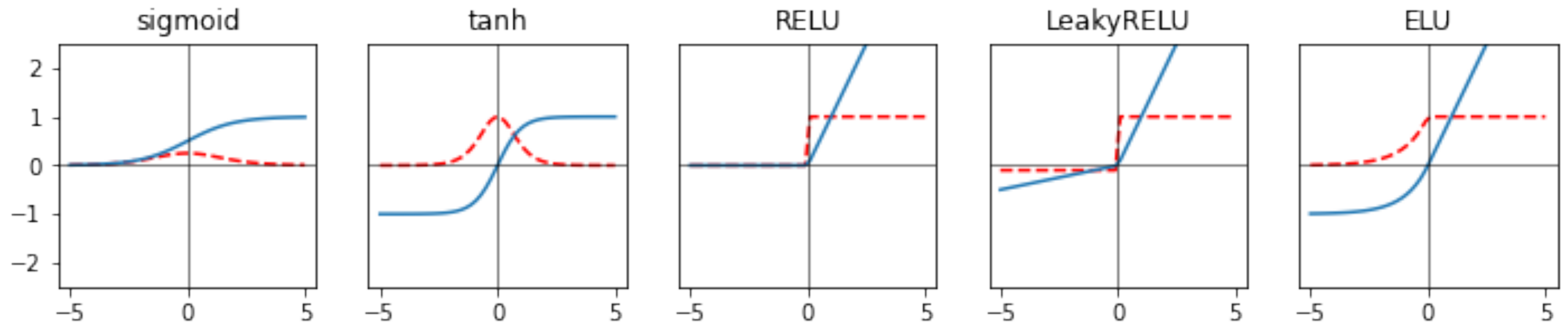
sigmoid: $f(x) = \frac{1}{1+e^{-x}}$ => output in range $[0,1]$

tanh: $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ => output in range $[-1,1]$

ReLU: $f(x) = \max(x,0)$ => no negative output

LeakyReLU: $f(x) = \begin{cases} x; x \geq 0 \\ \alpha x; x < 0 \end{cases}$ => no clamping to zero for negative inputs

ELU: $f(x) = \begin{cases} x; x \geq 0 \\ e^x - 1; x < 0 \end{cases}$ => smoother gradient



Basically any non-linear function can be used

LAST LAYER FOR CLASSIFICATION TASKS

For classification tasks: output of last layer uncalibrated and difficult to interpret

Softmax provides us with a multinomial categorical output

- See logistic function for binomial categories

Softmax: $\phi(x) = e^{x_i} / \Sigma_{j=1}^{K} e^{x_j}$, for K classes

Usually used in classification tasks to normalize the set of output activations to 1

- Do not confuse softmax with a mathematically sound probability

Sometimes numerically unstable (divisions by large numbers)

$$\frac{e^{x_i}}{\Sigma_{j=1}^{K} e^{x_j}} = \frac{Ce^{x_i}}{C\Sigma_{j=1}^{K} e^{x_j}} = \frac{e^{x+logC}}{\Sigma_{j=1}^{K} e^{x_j} + logC}, \text{ for } logC = -\max_j x_j$$

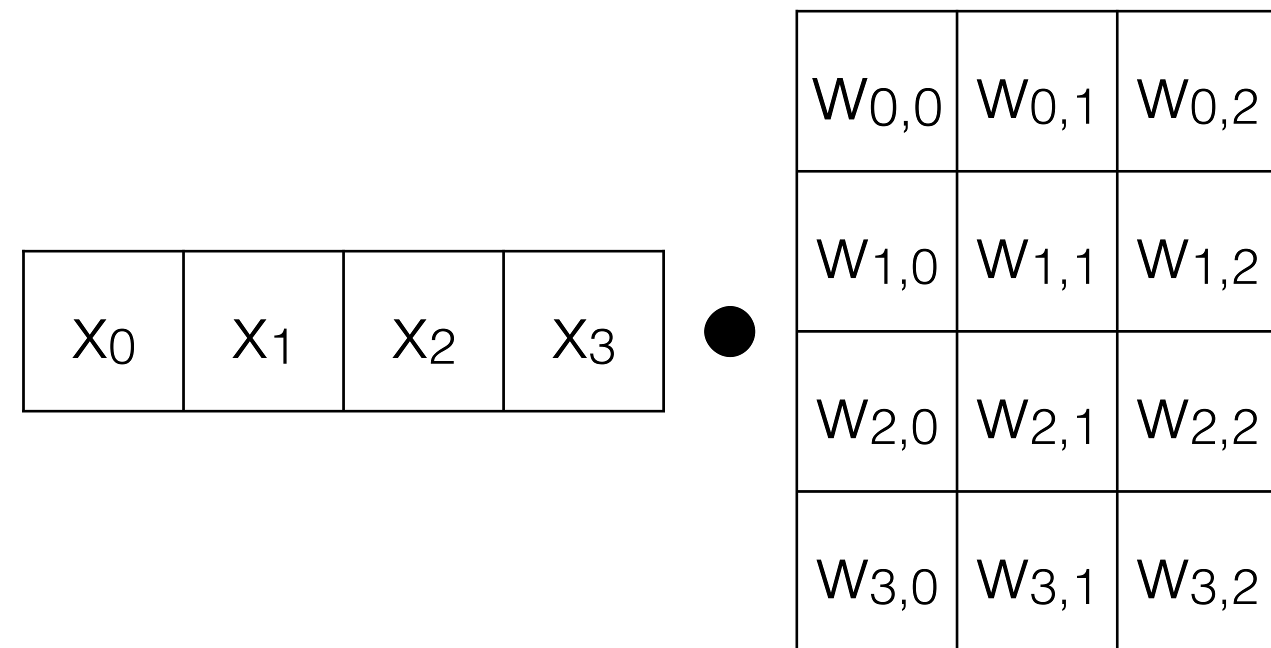
x	argmax(x)	softmax(x)	norm(x)
1	0.0 %	7.0 %	13.1 %
0.1	0.0 %	2.9 %	1.3 %
3	100.0 %	51.9 %	39.2 %
0.1	0.0 %	2.9 %	1.3 %
0.4	0.0 %	3.9 %	5.2 %
2	0.0 %	19.1 %	26.1 %
1	0.0 %	7.0 %	13.1 %
0.05	0.0 %	2.7 %	0.7 %
0	0.0 %	2.6 %	0.0 %

CALCULATING A NEURAL NETWORK

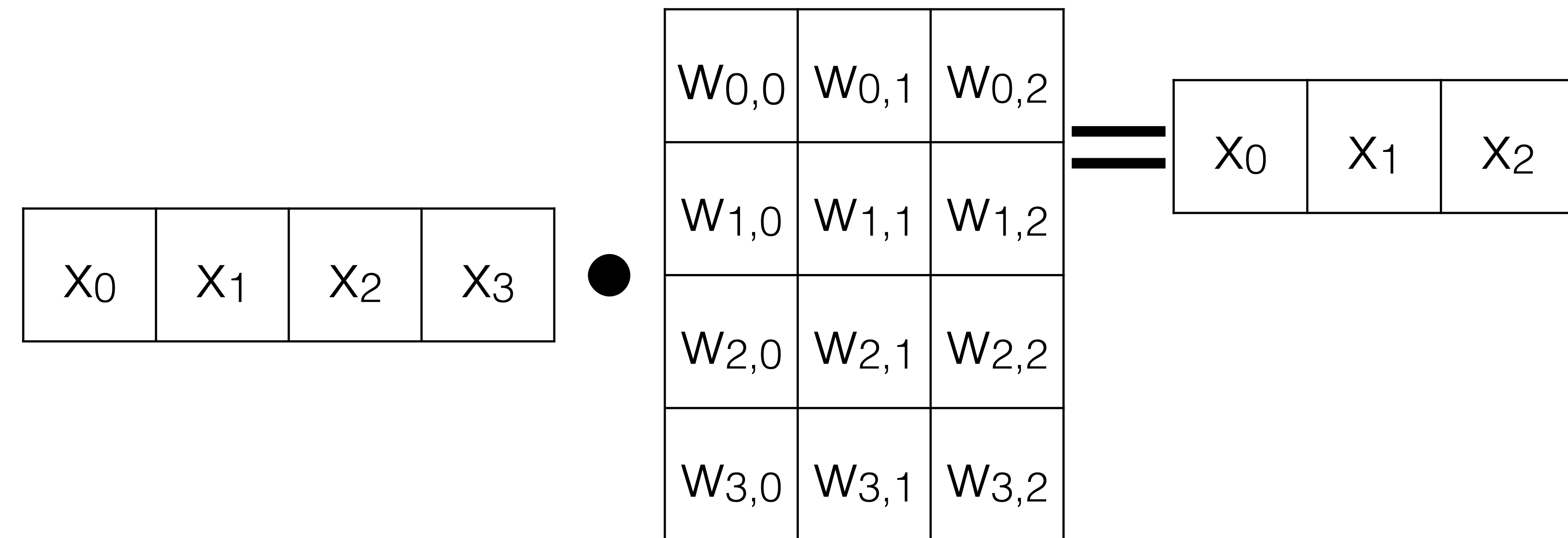
CALCULATING A NEURAL NETWORK

x_0	x_1	x_2	x_3

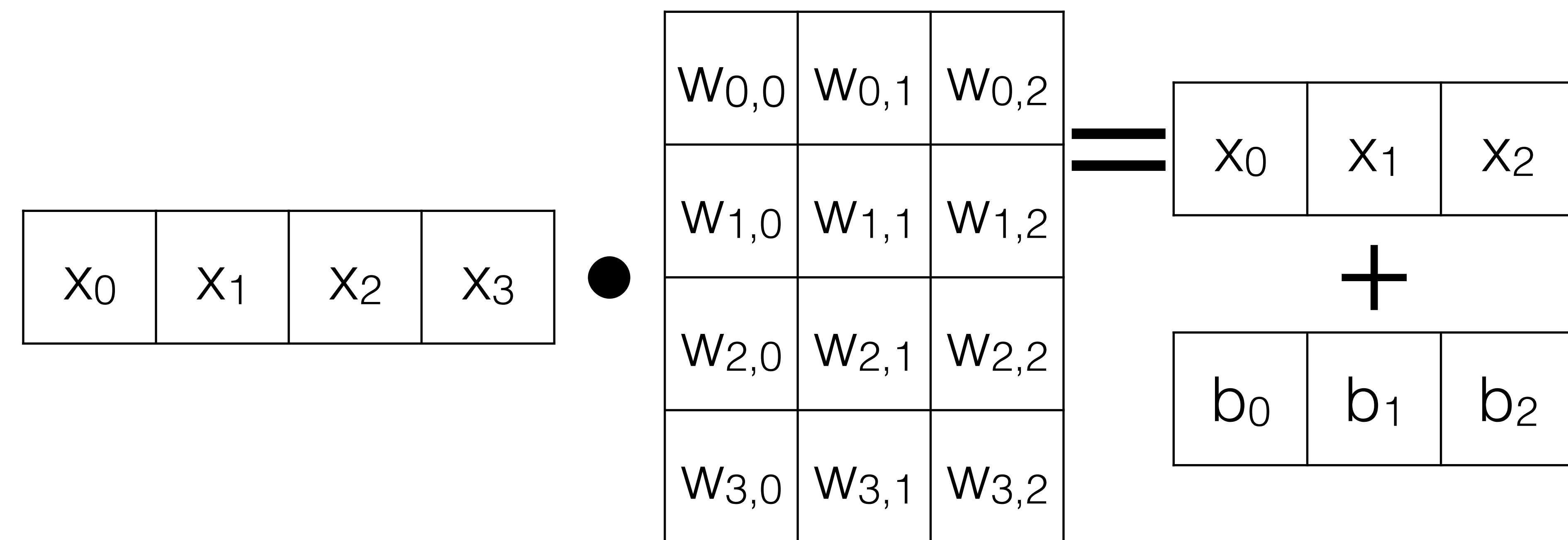
CALCULATING A NEURAL NETWORK



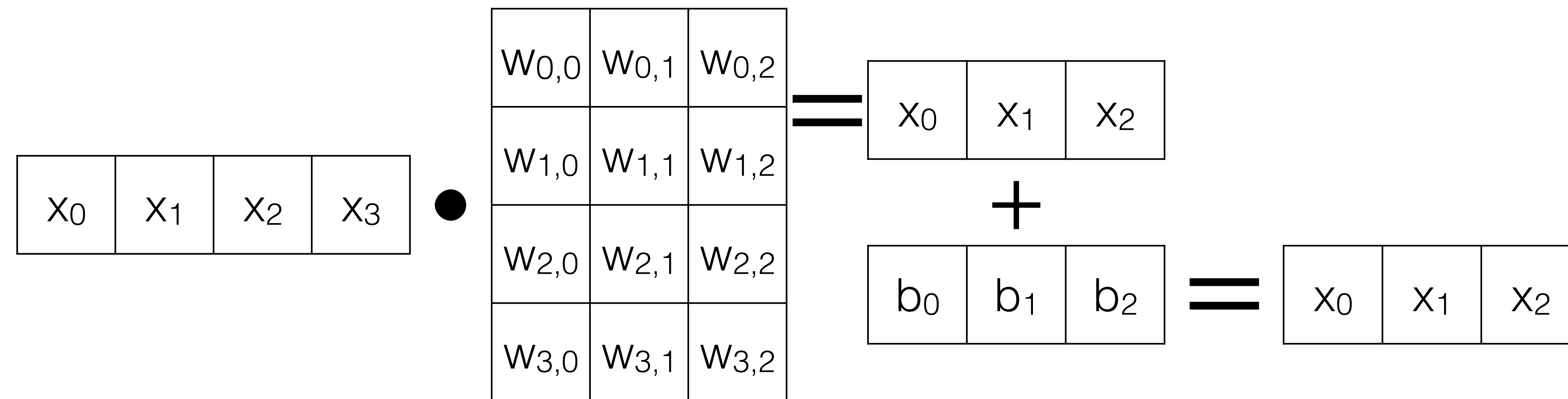
CALCULATING A NEURAL NETWORK



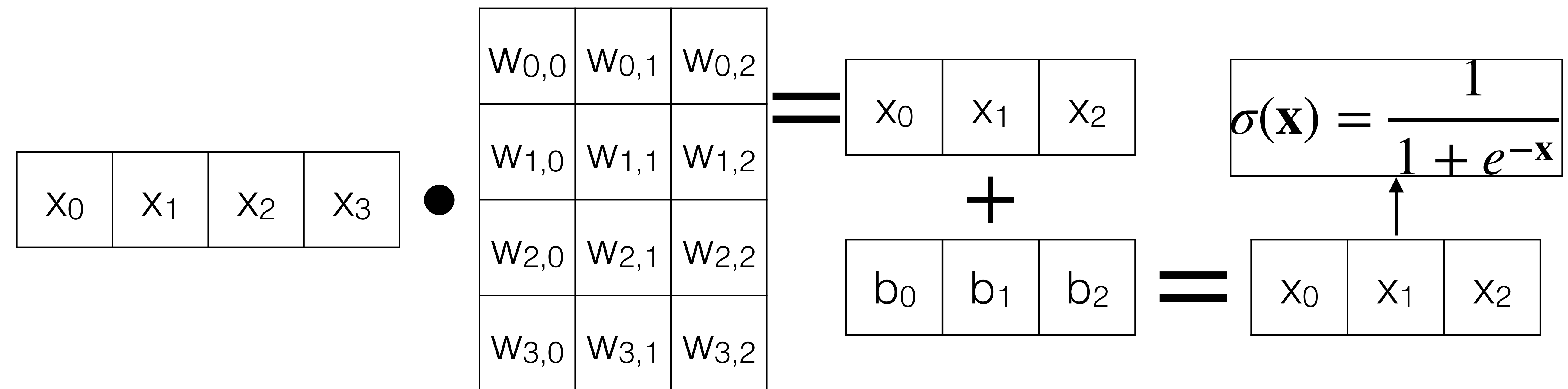
CALCULATING A NEURAL NETWORK



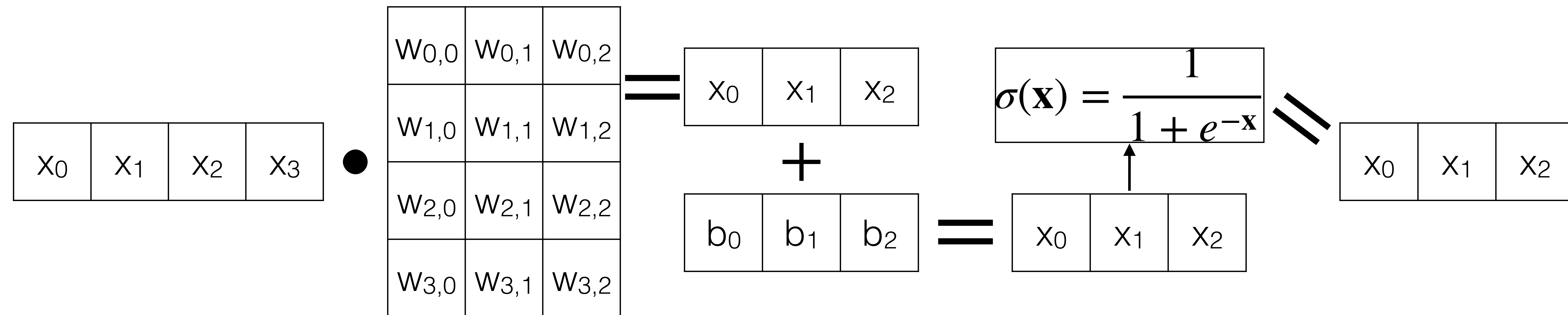
CALCULATING A NEURAL NETWORK



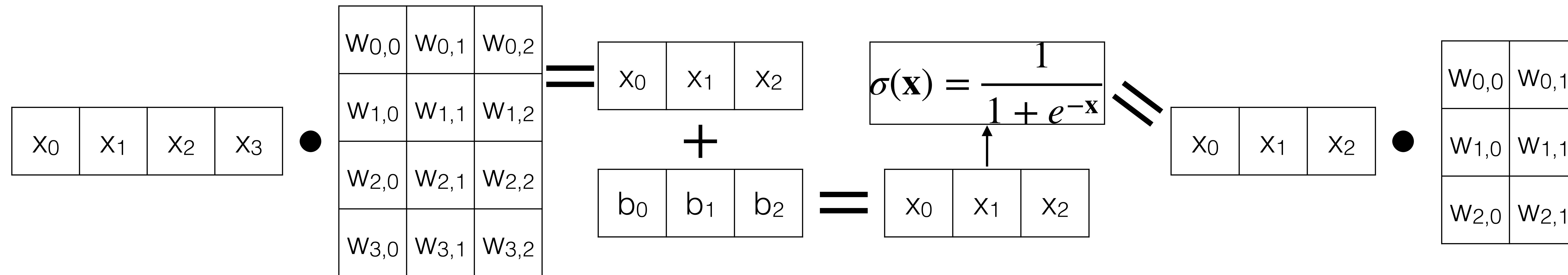
CALCULATING A NEURAL NETWORK



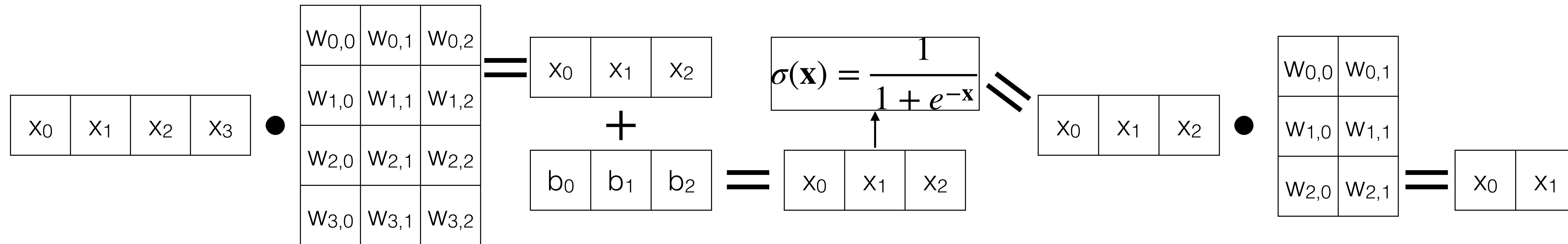
CALCULATING A NEURAL NETWORK



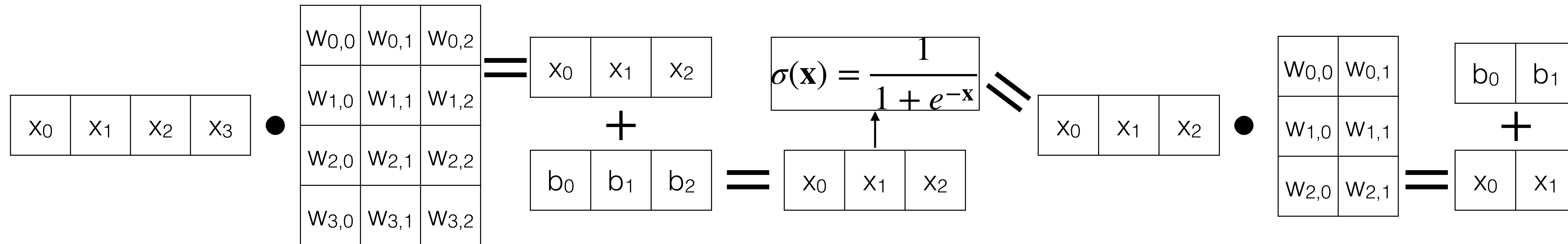
CALCULATING A NEURAL NETWORK



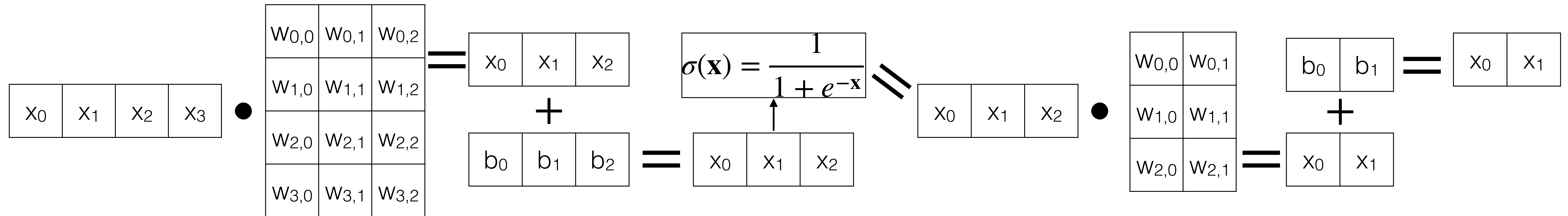
CALCULATING A NEURAL NETWORK



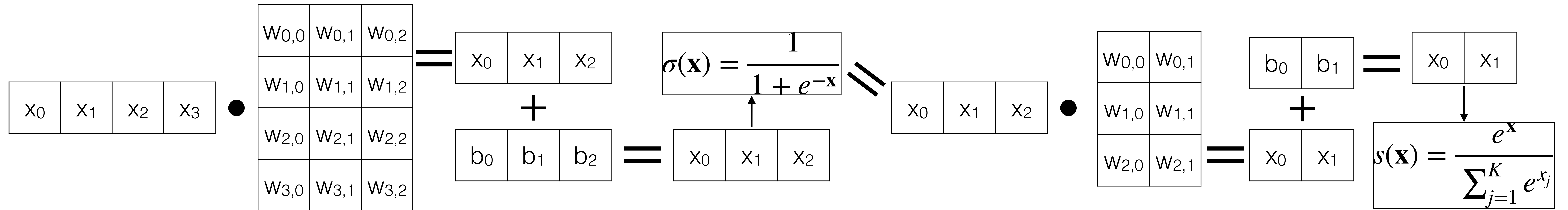
CALCULATING A NEURAL NETWORK



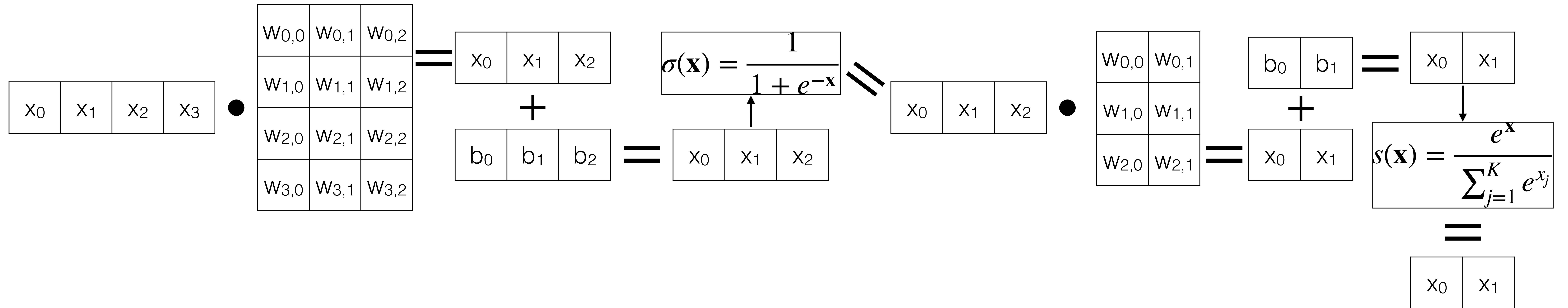
CALCULATING A NEURAL NETWORK



CALCULATING A NEURAL NETWORK



CALCULATING A NEURAL NETWORK



PURE PYTHON NEURAL NET

```
#!/usr/bin/env python3

import math
import random
from typing import Callable, List, Tuple

if __name__ == '__main__':

    model = XORNet(input_size=2,
                   hidden_size=2,
                   output_size=1)

    X = [[1, 1], [0, 1], [1, 0], [0, 0]]
    Y = [[0], [1], [1], [0]]

    model.train(X, Y, epochs=100, lr=0.5)

    model.pred(X, Y)
```

PURE PYTHON NN: DATA STRUCTURE

```
class Matrix():
    def __init__(self,
                 cols: int,
                 rows: int,
                 random_init=False) -> None:

        self._cols = cols
        self._rows = rows

        self.data = [0 for _ in range(cols*rows)]

        if random_init:
            for y in range(rows):
                for x in range(cols):
                    rand_val = random.uniform(-1, 1)
                    self[y, x] = rand_val

    def __getitem__(self, key: Tuple[int, int]) -> float:
        return self.data[key[0]*self._cols + key[1]]

    def __setitem__(self, key: Tuple[int, int], val: float) -> None:
        self.data[key[0]*self._cols + key[1]] = val
```

```
…

def __repr__(self) -> str:
    repr_string = ''

    for y in range(self._rows):
        for x in range(self._cols):
            repr_string += f'{self.data[y*self._cols + x]:.3f}\t'
        repr_string += '\n'

    return repr_string

def elementwise(self, func: Callable):
    out = Matrix(self._cols, self._rows)
    for count in range(len(self.data)):
        out.data[count] = func(self.data[count])
    return out

def shape(self) -> Tuple[int, int]:
    return (self._cols, self._rows)
```

PURE PYTHON NN: FULLY-CONNECTED OPERATION

```
def linear(x: Matrix,
           w: Matrix,
           b: Matrix) -> Matrix:

    I, _ = x.shape()
    J, K = w.shape()

    out = Matrix(I, K)

    for k in range(K):
        for j in range(J):
            for i in range(I):
                out[k, i] += w[k, j]*x[j, 0]

    for i in range(I):
        for k in range(K):
            out[k, i] += b[0, k]

    return out
```

PURE PYTHON NN: ACTIVATION FUNCTIONS

```
def sigmoid_(x: float):
    return 1/(1 + math.exp(-x))

def sigmoid(input_: Matrix):
    return input_.elementwise(sigmoid_)

def sigmoid_grad(input_: Matrix):
    return input_.elementwise(lambda x: sigmoid_(x)*(1 - sigmoid_(x)))

def tanh(input_: Matrix):
    return input_.elementwise(math.tanh)

def tanh_grad(input_: Matrix):
    return input_.elementwise(lambda x: 1 - math.tanh(x)*math.tanh(x))
```

PURE PYTHON NN: NETWORK - INITIALIZATION

```python
class XORNet():

    def __init__(self,
                    input_size: int,
                    hidden_size: int,
                    output_size: int):

        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        self.w_h = Matrix(input_size, hidden_size, random_init=True)
        self.b_h = Matrix(1, hidden_size, random_init=True)

        self.w_o = Matrix(hidden_size, output_size, random_init=True)
        self.b_o = Matrix(1, output_size, random_init=True)

        self.act_h = sigmoid
        self.act_o = tanh

        # Activations for grad computation

        self.z_h = None
        self.z_o = None
```

PURE PYTHON NN: NETWORK - FORWARD PROP

```
    def forward(self, x: Matrix):
        self.z_h = linear(x, self.w_h, self.b_h)
        self.z_o = linear(self.act_h(self.z_h), self.w_o, self.b_o)

        return self.act_o(self.z_o)
```

PURE PYTHON NN: NETWORK - FORWARD PROP

```
    def forward(self, x: Matrix):
        self.z_h = linear(x, self.w_h, self.b_h)
        self.z_o = linear(self.act_h(self.z_h), self.w_o, self.b_o)

        return self.act_o(self.z_o)
```

BACK PROP ON ONE SLIDE

Data set containing N input-target pairs: $\mathscr{D} = \{(\mathbf{x}_1, t_1), \ldots, (\mathbf{x}_N, t_N)\}$

Training ANNs: adjust randomly initialized weights $\mathbf{W}$ to solve a given task by minimizing a loss function $\mathscr{L}$ using gradient-based optimization

$$\mathscr{L}(\mathbf{W}; \mathscr{D}) = \sum_{n=1}^{N} l(y(\mathbf{W}, \mathbf{x}_n), t_n) + \lambda r(\mathbf{W});$$

based on a data term l that penalizes wrong prediction (error function); and

for a regularizer $r(\mathbf{W})$ such as ℓ^1–norm or ℓ^2–norm and a trade-off hyperparameter λ

Backpropagation: compute gradient for input-target pair and minimize the loss function by iteratively calculating

$\mathbf{W} := \mathbf{W} - \eta \nabla_{\mathbf{W}} \mathscr{L}(\mathbf{W}; \mathscr{D})$, for $\nabla_{\mathbf{x}} = \left(\frac{\partial}{\partial x_1}, \ldots, \frac{\partial}{\partial x_n}\right)$ and learning rate η

Key operations: partial derivative and all-reduce

PARTIAL DERIVATIVES SOUGHT

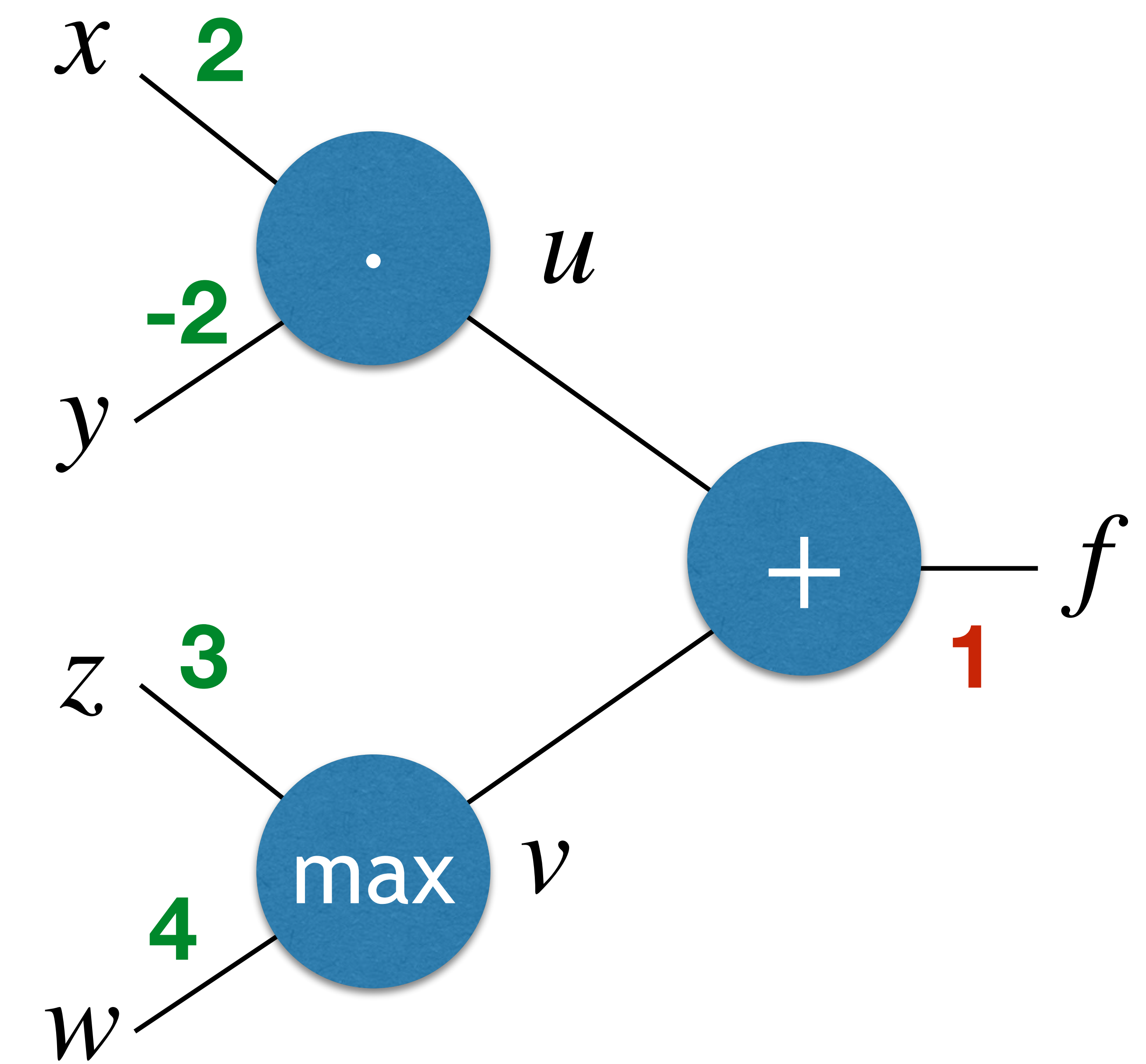
Consider exemplary function $f(x, y, z, w) = xy + max(z, w)$

$$\frac{\partial}{\partial x} f, \frac{\partial}{\partial y} f, \frac{\partial}{\partial z} f, \frac{\partial}{\partial w} f \quad ?$$

PARTIAL DERIVATIVES SOUGHT

Consider exemplary function $f(x, y, z, w) = xy + max(z, w)$

$$\frac{\partial}{\partial x}f, \frac{\partial}{\partial y}f, \frac{\partial}{\partial z}f, \frac{\partial}{\partial w}f \quad ?$$

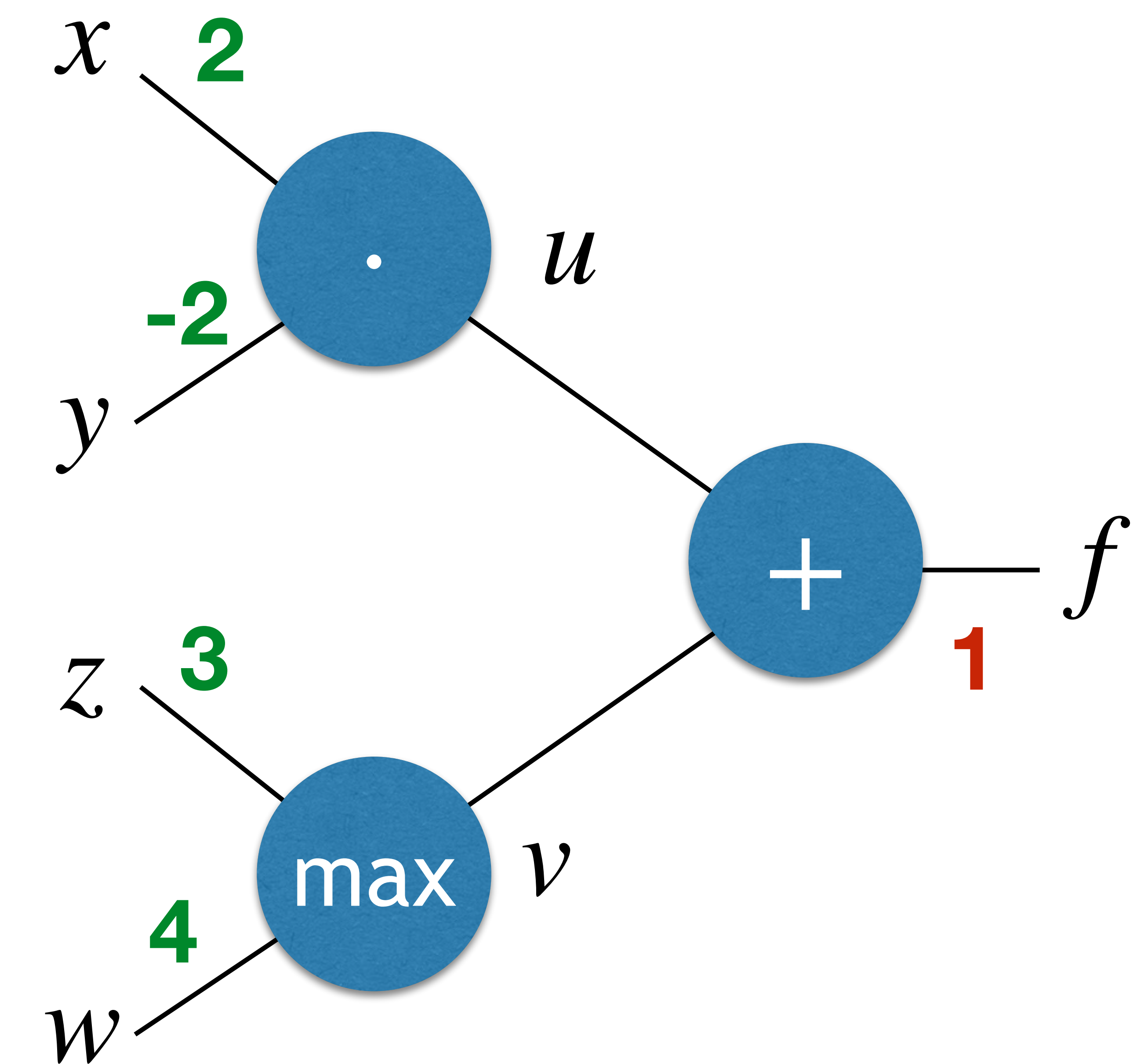


PARTIAL DERIVATIVES SOUGHT

Consider exemplary function $f(x, y, z, w) = xy + max(z, w)$

$$\frac{\partial}{\partial x} f, \frac{\partial}{\partial y} f, \frac{\partial}{\partial z} f, \frac{\partial}{\partial w} f \quad ?$$

Consider $f(x, y, z, w) = add(mul(x, y), max(z, w))$



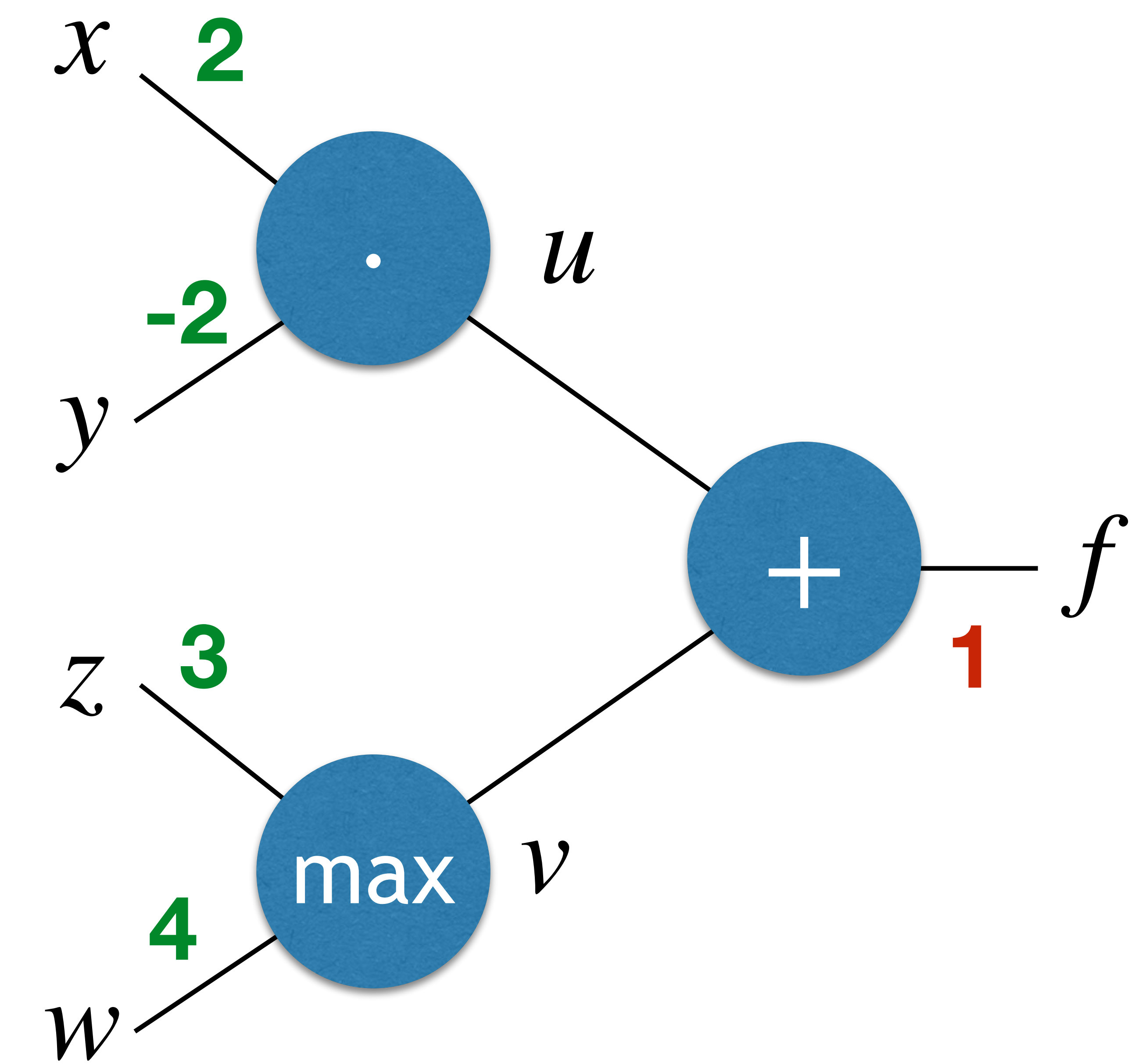
PARTIAL DERIVATIVES SOUGHT

Consider exemplary function $f(x, y, z, w) = xy + max(z, w)$

$$\frac{\partial}{\partial x} f, \frac{\partial}{\partial y} f, \frac{\partial}{\partial z} f, \frac{\partial}{\partial w} f \quad ?$$

Consider $f(x, y, z, w) = add(mul(x, y), max(z, w))$

$$add(u, v) = u + v; \frac{\partial add}{\partial u} = \frac{\partial add}{\partial v} = 1$$



PARTIAL DERIVATIVES SOUGHT

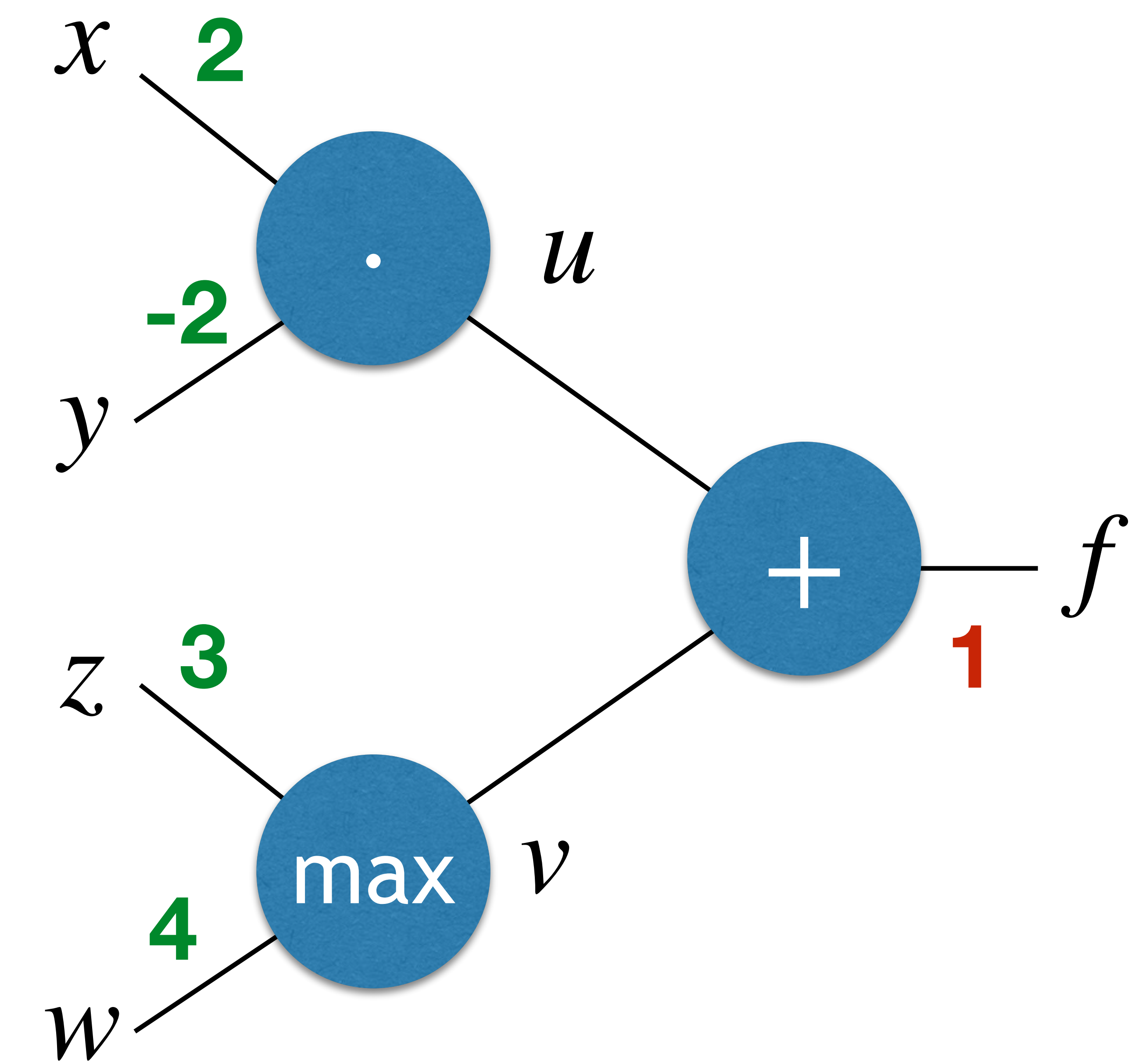
Consider exemplary function $f(x, y, z, w) = xy + max(z, w)$

$$\frac{\partial}{\partial x} f, \frac{\partial}{\partial y} f, \frac{\partial}{\partial z} f, \frac{\partial}{\partial w} f \quad ?$$

Consider $f(x, y, z, w) = add(mul(x, y), max(z, w))$

$$add(u, v) = u + v; \frac{\partial add}{\partial u} = \frac{\partial add}{\partial v} = 1$$

$$mul(x, y) = xy; \frac{\partial mul}{\partial x} = y; \frac{\partial mul}{\partial y} = x$$



PARTIAL DERIVATIVES SOUGHT

Consider exemplary function $f(x, y, z, w) = xy + max(z, w)$

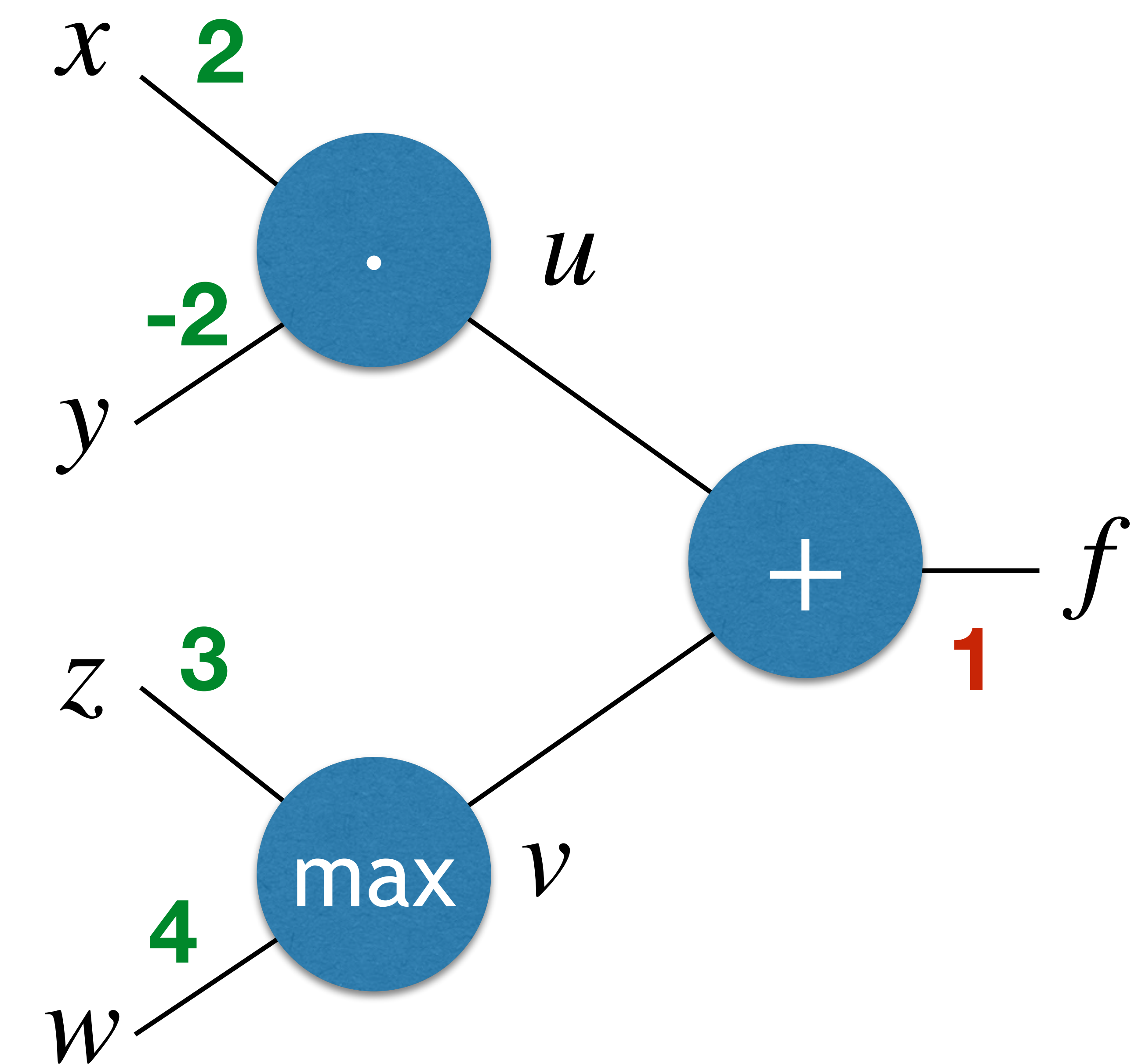
$$\frac{\partial}{\partial x} f, \frac{\partial}{\partial y} f, \frac{\partial}{\partial z} f, \frac{\partial}{\partial w} f \quad ?$$

Consider $f(x, y, z, w) = add(mul(x, y), max(z, w))$

$$add(u, v) = u + v; \frac{\partial add}{\partial u} = \frac{\partial add}{\partial v} = 1$$

$$mul(x, y) = xy; \frac{\partial mul}{\partial x} = y; \frac{\partial mul}{\partial y} = x$$

$$max(z, w) = \begin{cases} z; \text{if } z \geq w \\ w; \text{else} \end{cases}; \frac{\partial max}{\partial z} = \begin{cases} 1; \text{if } z \geq w \\ 0; \text{else} \end{cases}; \frac{\partial max}{\partial w} = \begin{cases} 0; \text{if } z \geq w \\ 1; \text{else} \end{cases}$$



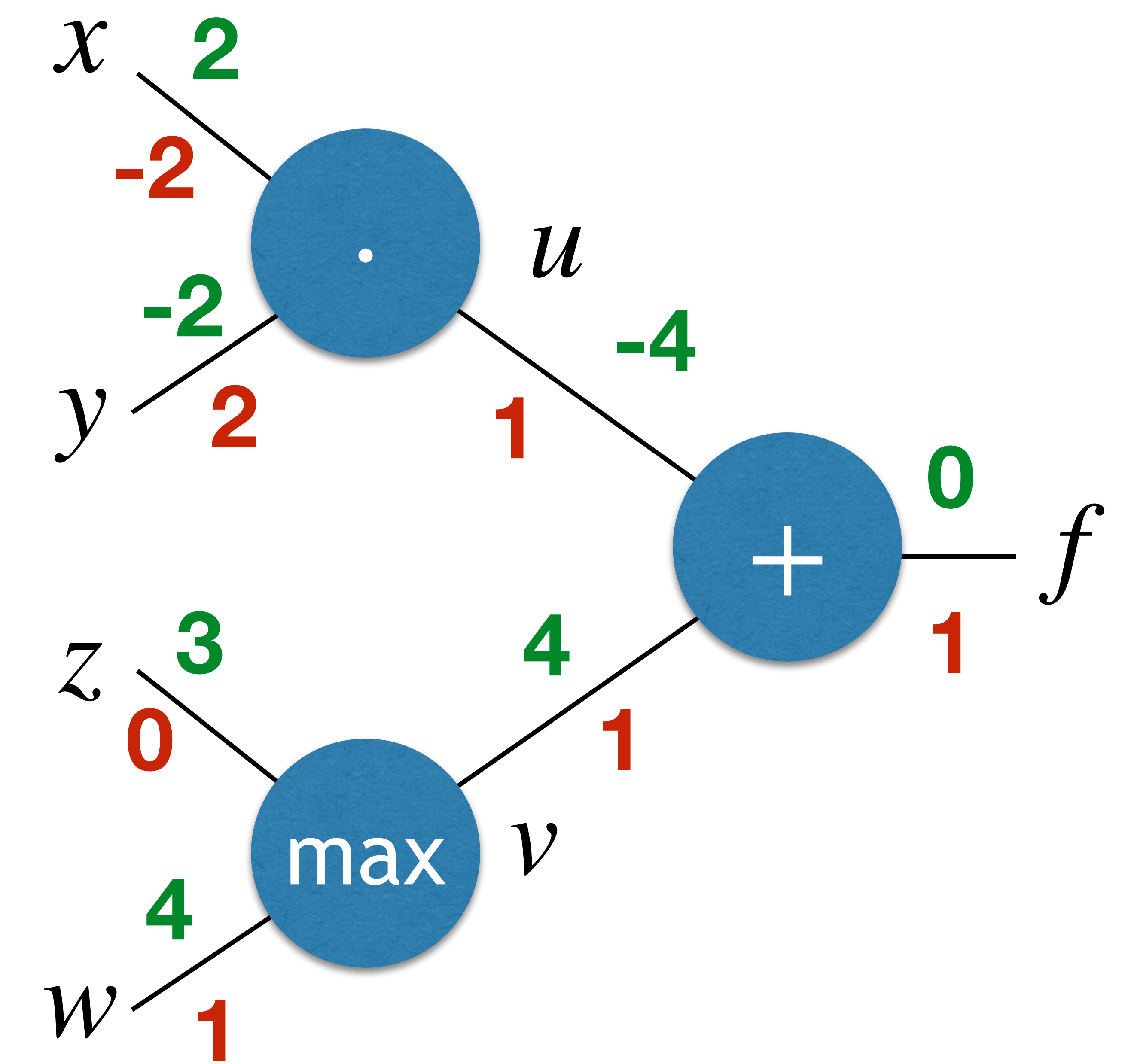
CHAIN RULE TO THE RESCUE

Chain rule of calculus: $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial u}\frac{\partial u}{\partial x}$

$$\frac{\partial f}{\partial u} = \frac{\partial add(u,v)}{\partial u} = 1; \frac{\partial u}{\partial x} = \frac{\partial mul(x,y)}{\partial x} = y$$

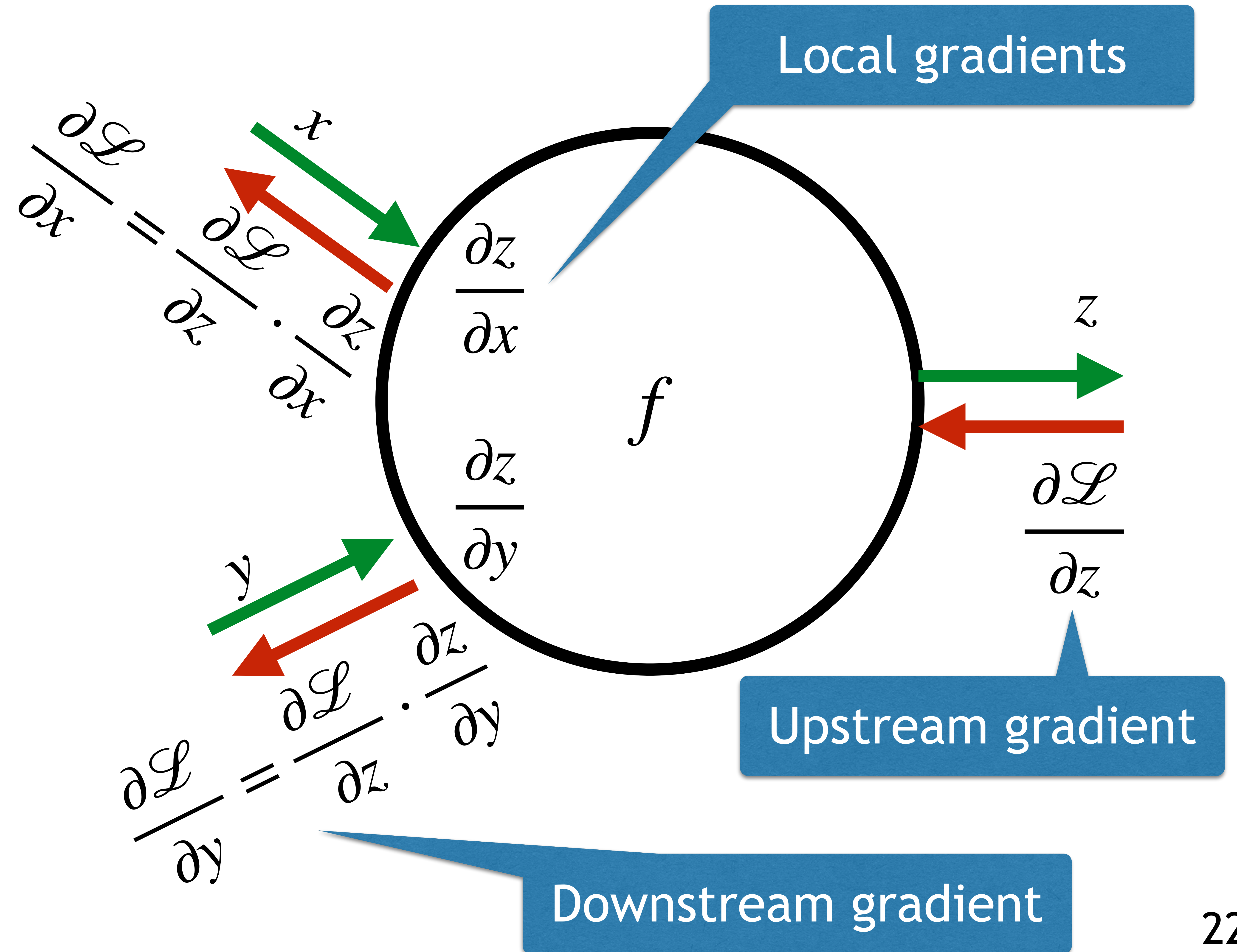
=> $\frac{\partial f}{\partial x} = 1 \cdot y$

Others in an analogous manner



$$\theta := \theta - \eta \cdot \frac{\partial \mathscr{L}}{\partial \theta}$$

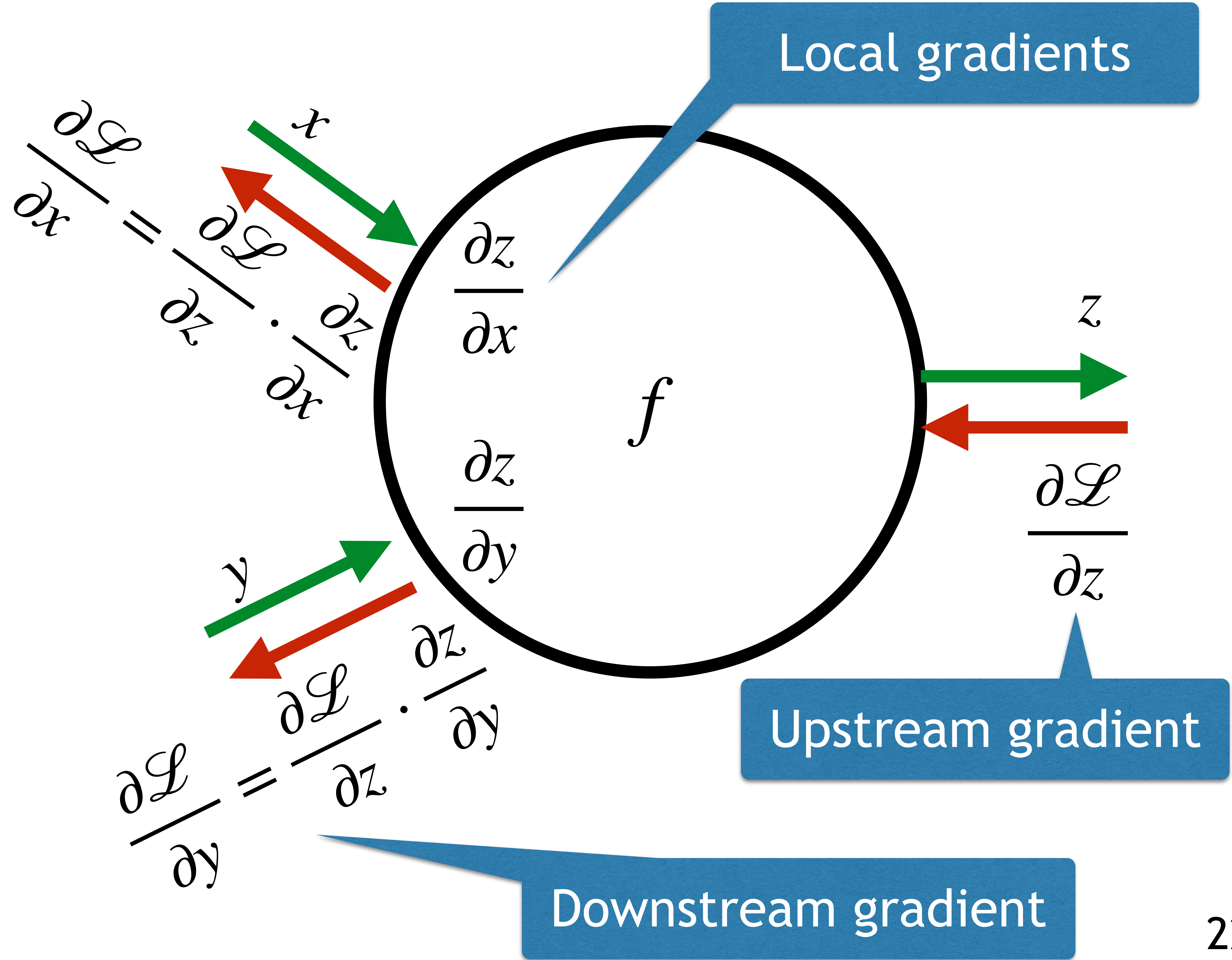
GRADIENT BEHAVIOR



GRADIENT BEHAVIOR

add gate: gradient distributor

Both downstream gradients equal upstream gradient



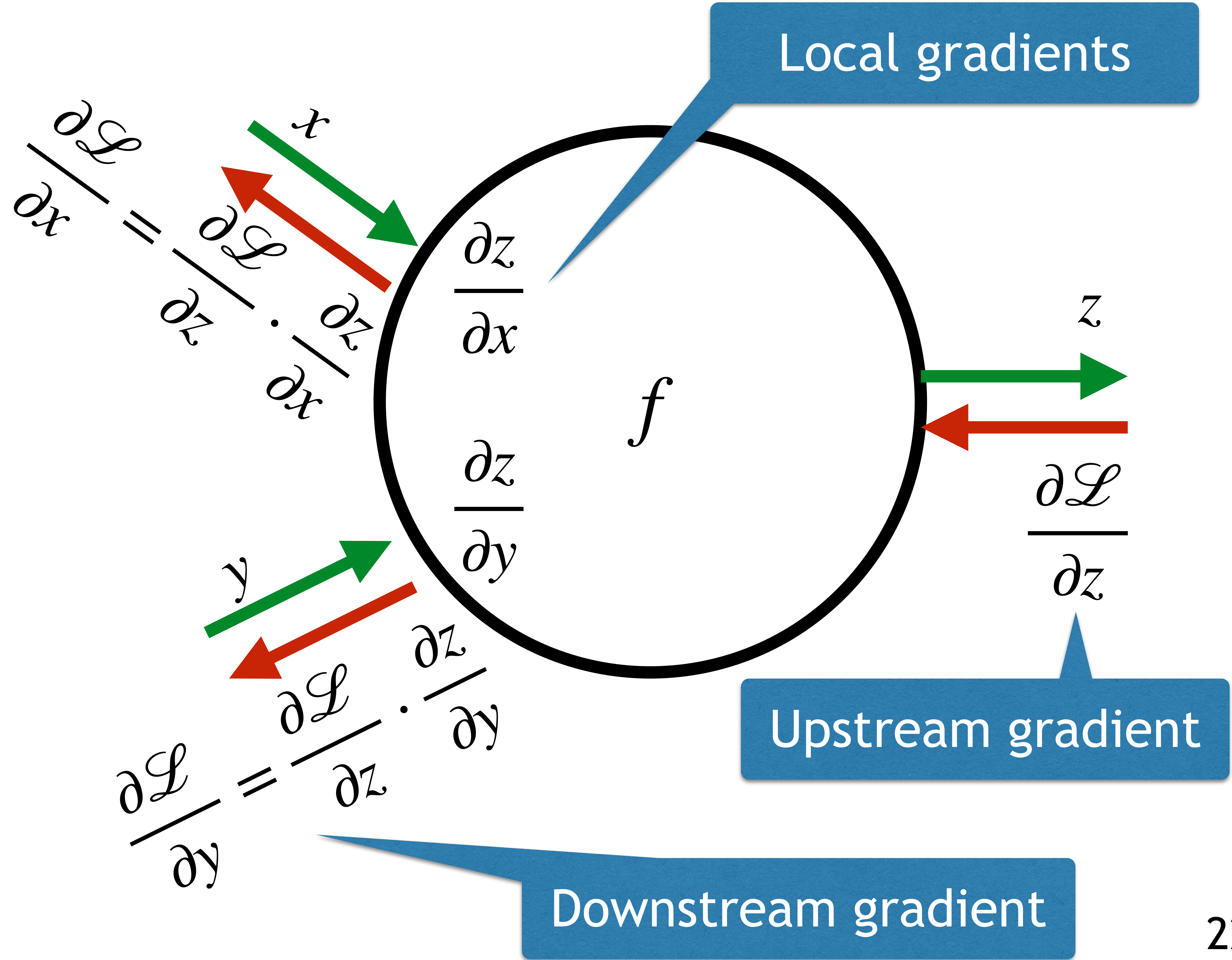
GRADIENT BEHAVIOR

add gate: gradient distributor

Both downstream gradients equal upstream gradient

mul gate: gradient switcher

Downstream gradients multiplied with other input



GRADIENT BEHAVIOR

add gate: gradient distributor

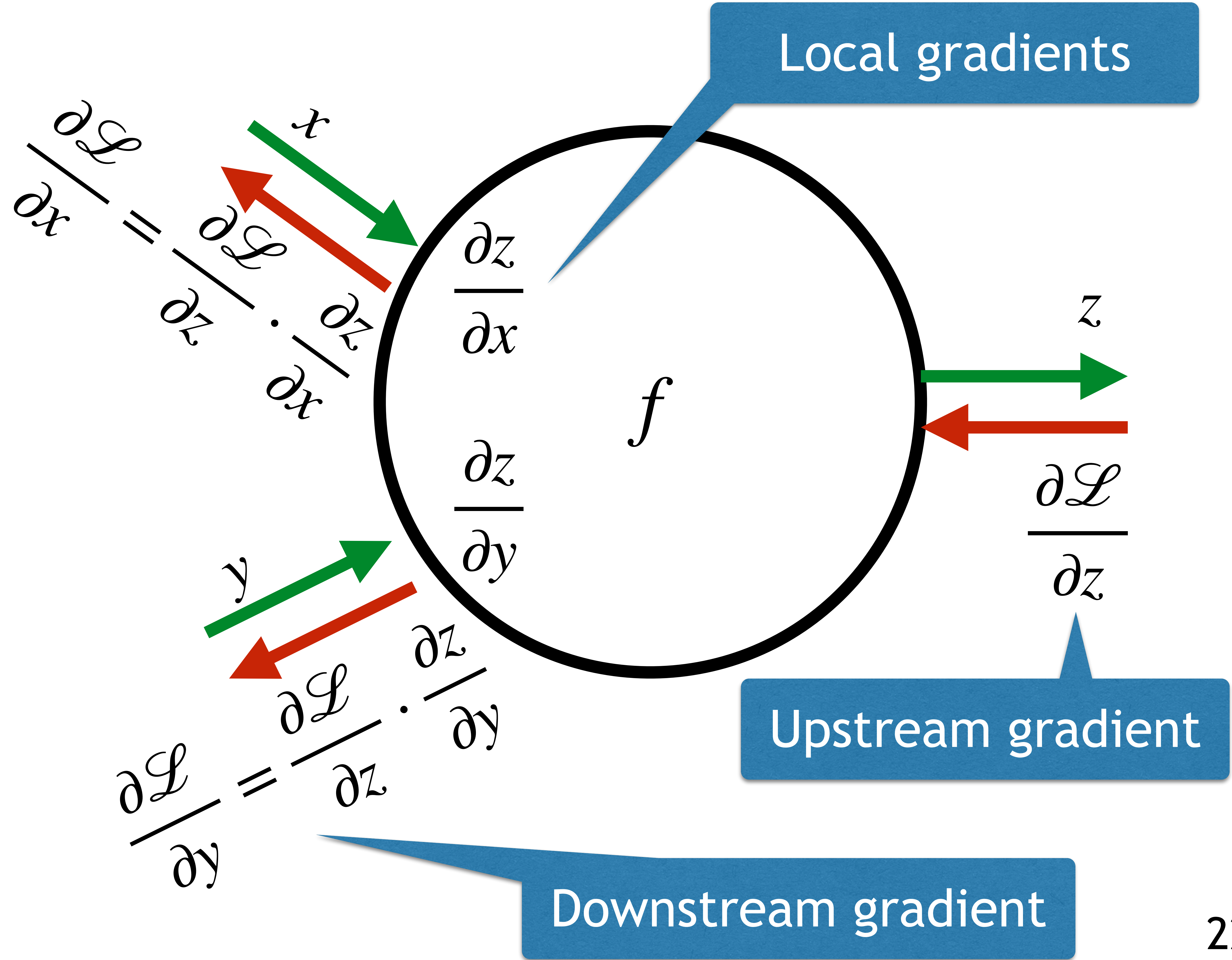
Both downstream gradients equal upstream gradient

mul gate: gradient switcher

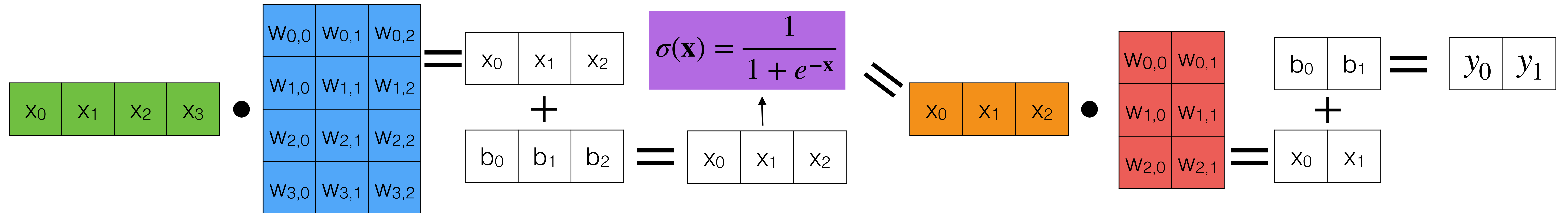
Downstream gradients multiplied with other input

max gate: gradient router

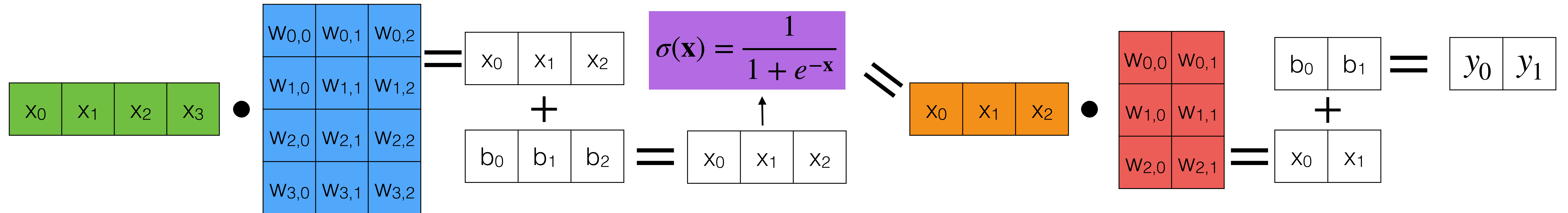
Downstream gradients depend on comparison of inputs

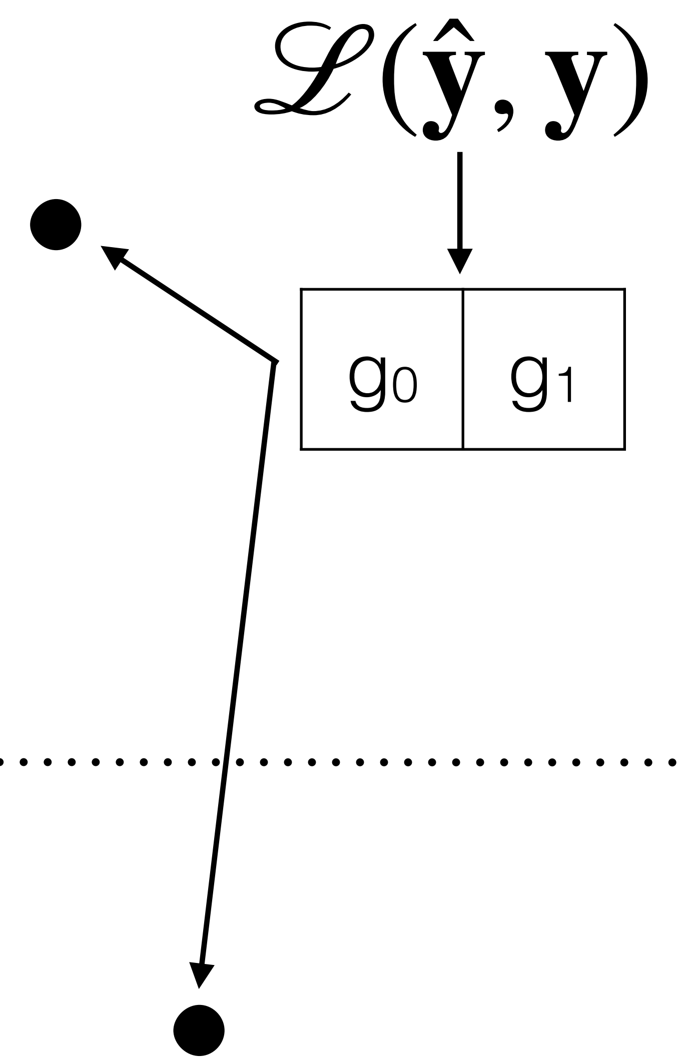


CALCULATING A NEURAL NETWORK

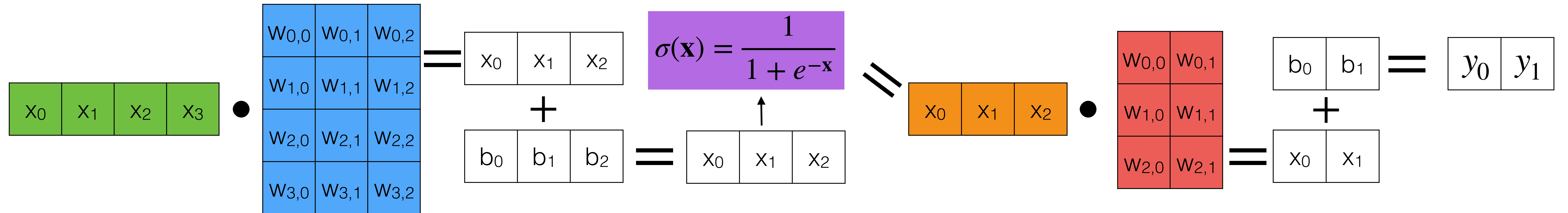


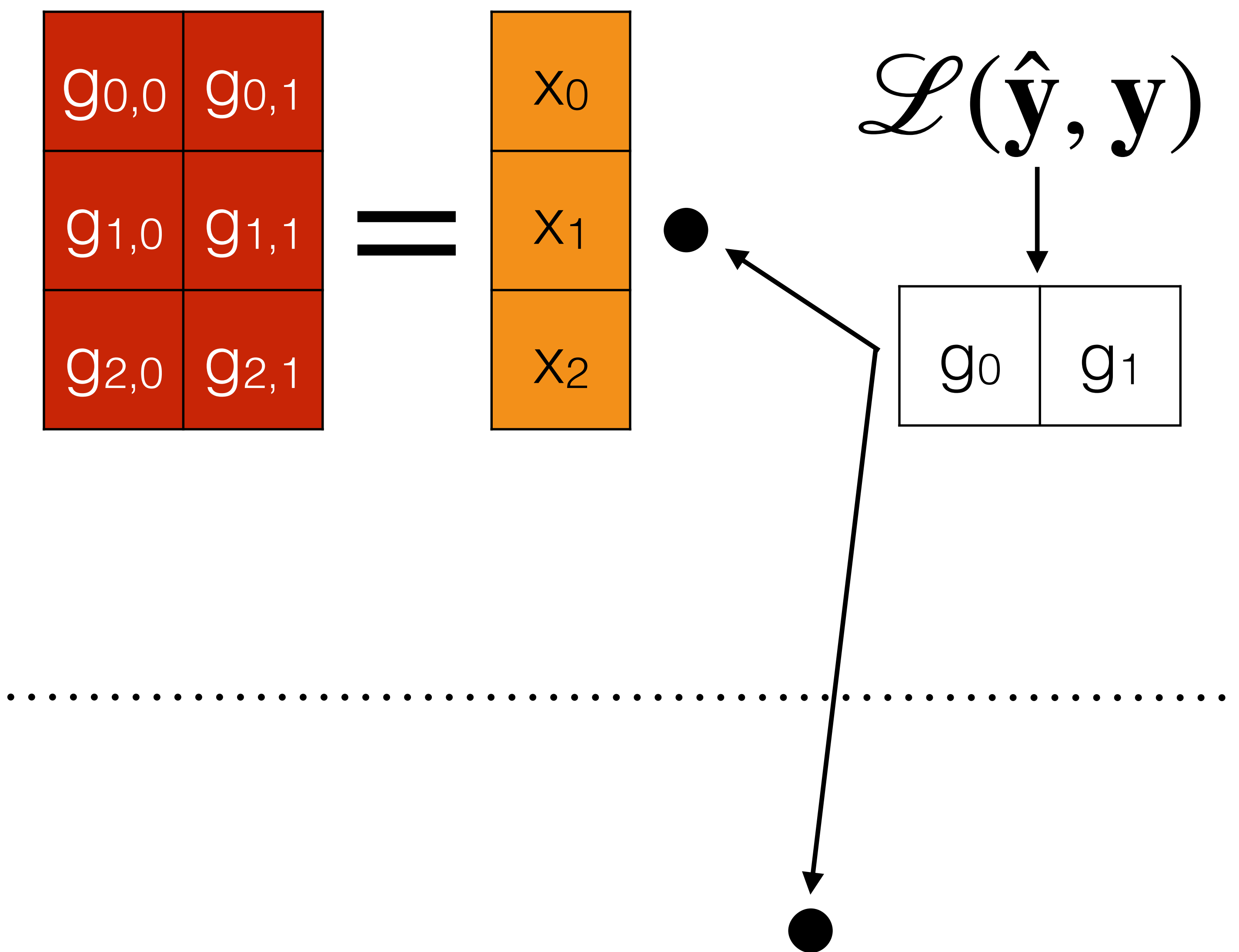
CALCULATING A NEURAL NETWORK



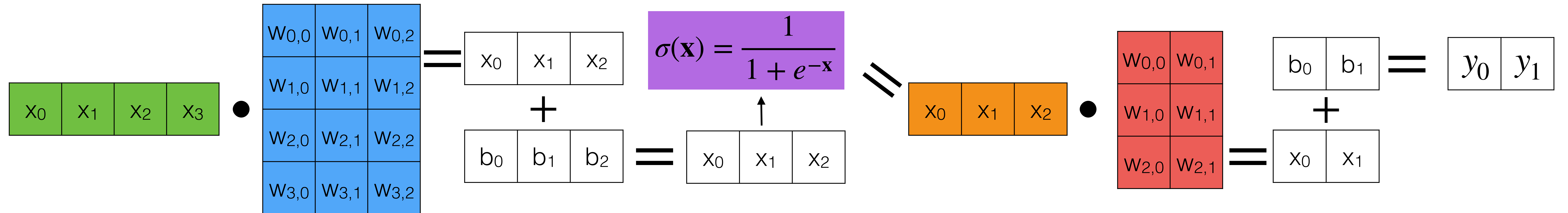


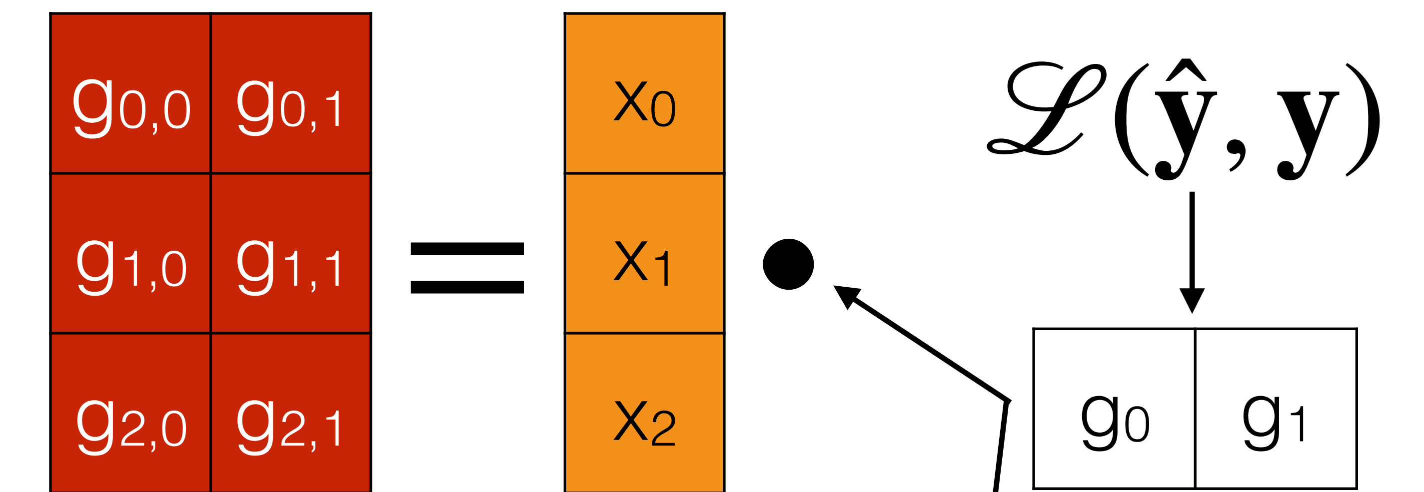
CALCULATING A NEURAL NETWORK

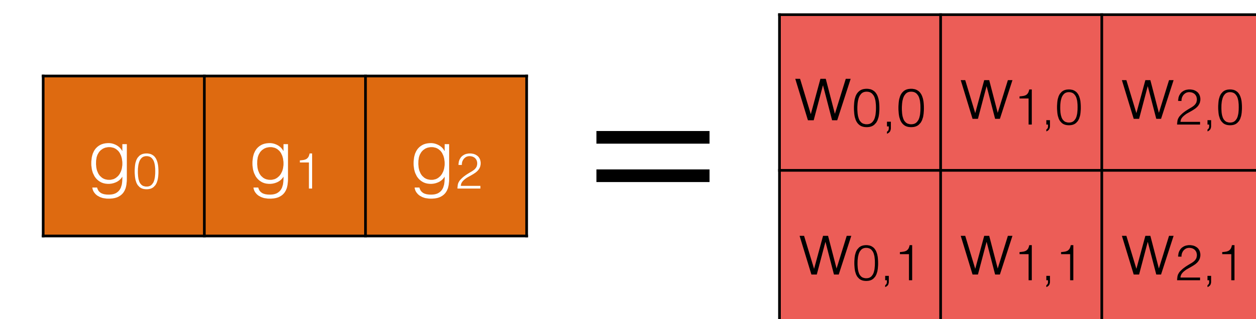




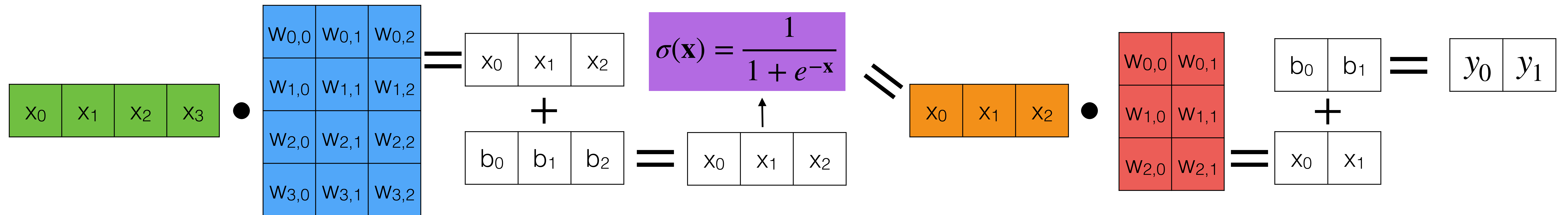
CALCULATING A NEURAL NETWORK

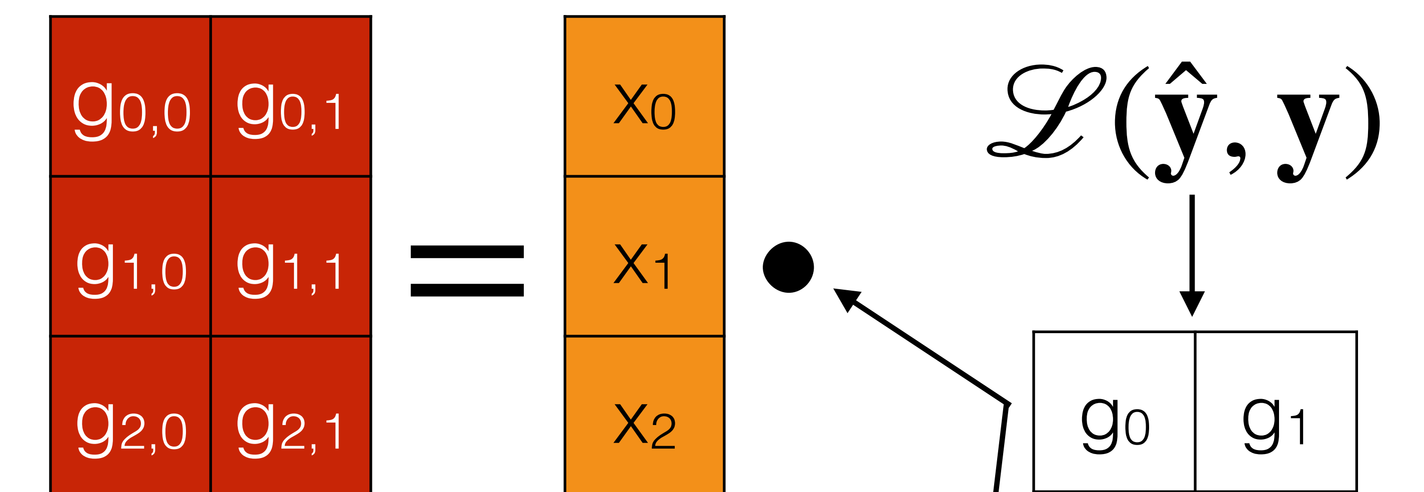




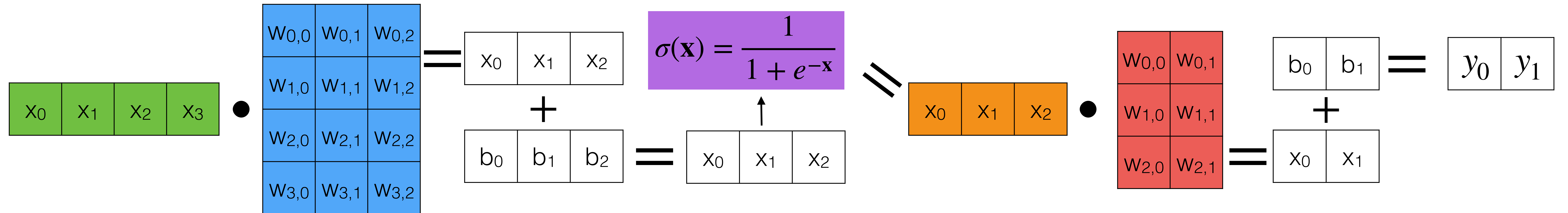


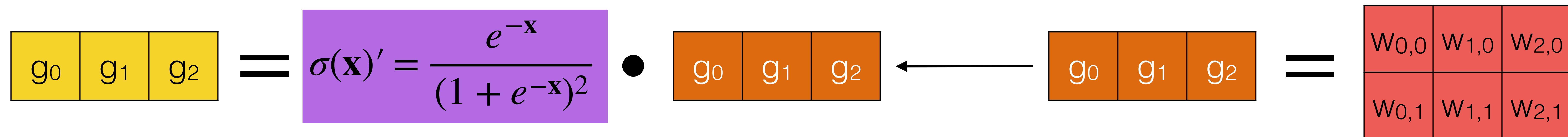
CALCULATING A NEURAL NETWORK



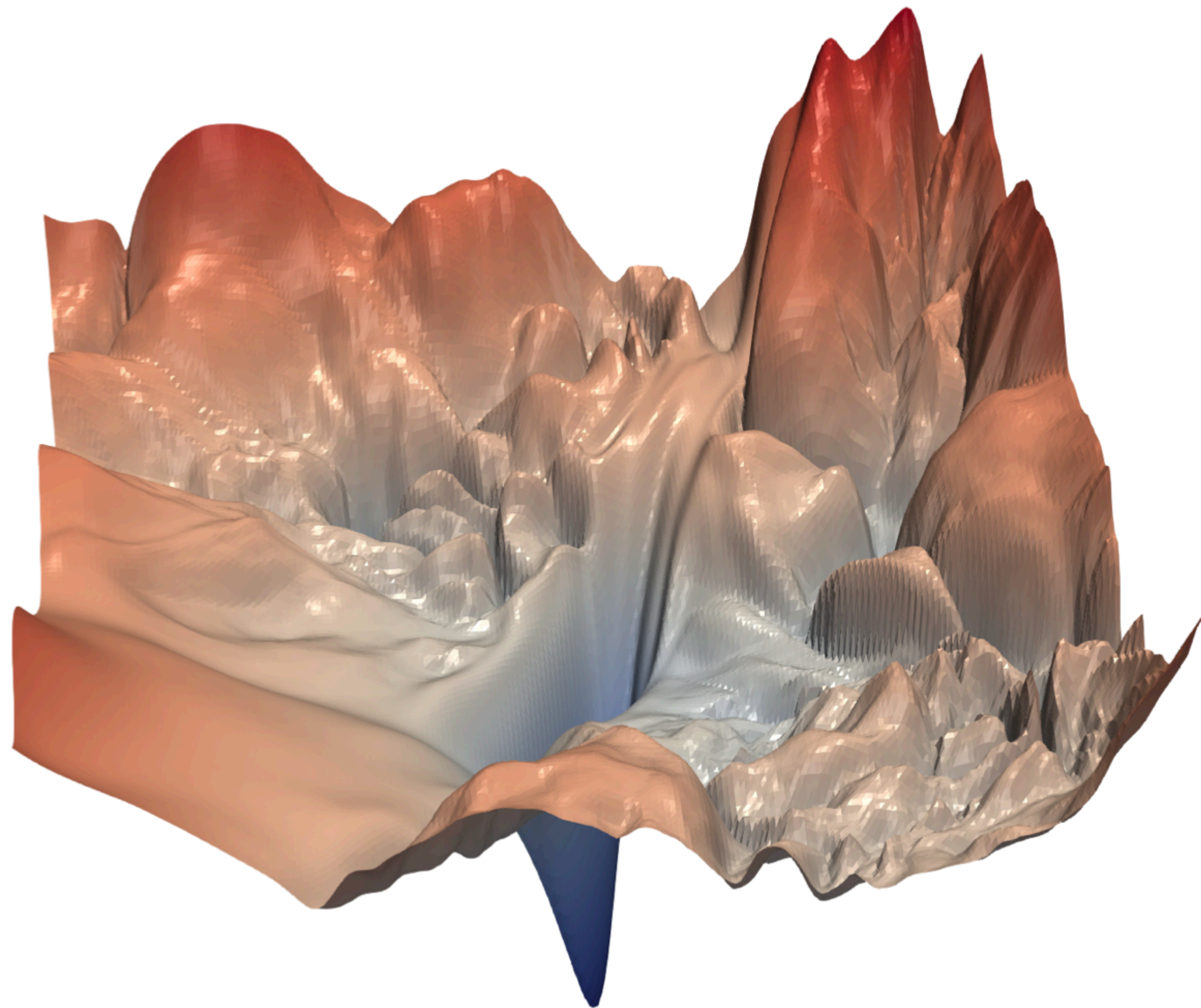


CALCULATING A NEURAL NETWORK

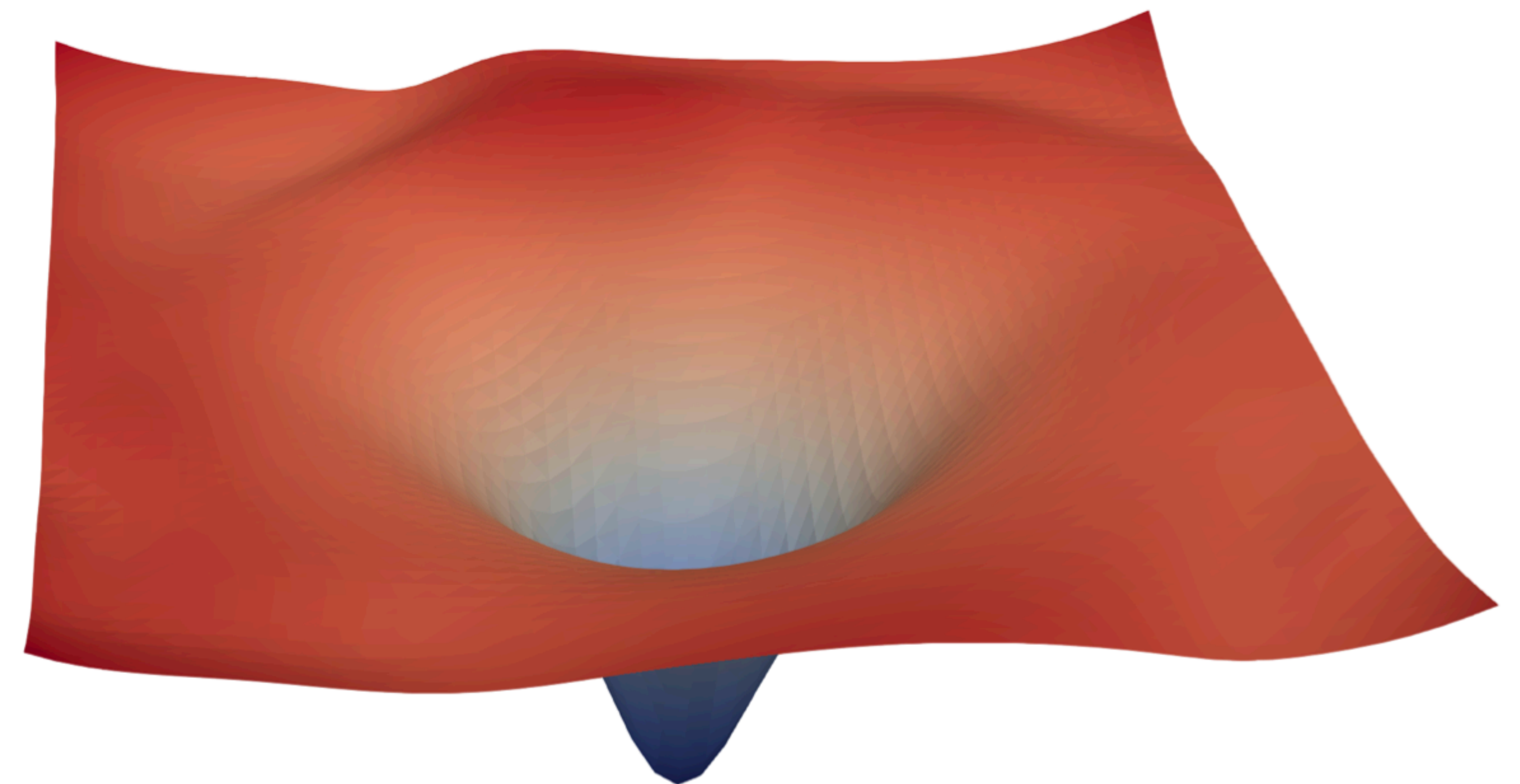




EXAMPLE LOSS LANDSCAPES IN MODERN ANNS



(a) without skip connections



(b) with skip connections

Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. 2018. Visualizing the loss landscape of neural nets. In 32nd International Conference on Neural Information Processing Systems (NIPS'18)

PURE PYTHON NN: NETWORK - BACKPROP

```python
def backward(self, x: Matrix,
                   y: Matrix,
                   y_hat: Matrix,
                   lr: float):

    e_o = Matrix(*y.shape())
    d_o = Matrix(*y.shape())

    act_grad = tanh_grad(self.z_o)

    for k in range(y.shape()[0]):
        e_o[k, 0] = y[k, 0] - y_hat[k, 0]
        d_o[k, 0] = e_o[k, 0]*act_grad[k, 0]

    I, _ = self.w_h.shape()
    K, J = self.w_o.shape()

    e_h = Matrix(*(self.z_h.shape()))
    d_h = Matrix(*(self.z_h.shape()))
```

```python
…

for k in range(K):
    for j in range(J):
        e_h[k, 0] += d_o[j, 0]*self.w_o[j, k]
     d_h[k, 0] = e_h[j, 0]*act_grad[j, 0]

H = self.act_h(self.z_h)

for j in range(J):
    for k in range(K):
        self.w_o[j, k] -= lr*d_o[j, 0]*H[k, 0]
        self.b_o[0, j] -= lr*d_o[j, 0]

for i in range(I):
    for k in range(K):
        self.w_h[i, k] -= lr*d_h[k, 0]*x[i, 0]
        self.b_h[0, i] -= lr*d_h[k, 0]
```

PURE PYTHON NN: TRAINING & PREDICTION

```
def train(self,
          X: List[List[float]] | List[List[int]],
          Y: List[List[float]] | List[List[int]],
          epochs: int,
          lr: float) -> None:

    for epoch in range(epochs):
        for x, y in zip(X, Y):
            y_hat = self.forward(to_matrix(x))
            self.backward(to_matrix(x), to_matrix(y), y_hat, lr)


def pred(self,
          X: List[List[float]] | List[List[int]],
          Y: List[List[float]] | List[List[int]]) -> None:

    for x in X:
            y_hat = self.forward(to_matrix(x))
            print(f'x: {x}\ty_hat: {y_hat}')
```

PERCEPTRON DEMO

WEIGHTS AND BIASES

Or more general: Experiment tracking

EXPERIMENT TRACKING SERVICES

General use case:

- Live monitoring, with built in plotting
- Easy hyperparamter overview
- Artifact storage
- No more log parsers

Many options out there:

- Weights & Biases
- Neptune
- determined.ai
- …

SETUP

See: https://docs.wandb.ai/quickstart/

General setup for all these services:

- Sign up
- Install library & login
- Instrument code:
  - Initialization and registration of hyper-parameters
  - Log running variables
  - Insights into system utilization

Tracking usually via web interface

```
# train.py
import wandb
import random  # for demo script

wandb.login()

epochs = 10
lr = 0.01

run = wandb.init(
    # Set the project where this run will be logged
    project="my-awesome-project",
    # Track hyperparameters and run metadata
    config={
        "learning_rate": lr,
        "epochs": epochs,
    },
)

offset = random.random() / 5
print(f"lr: {lr}")

# simulating a training run
for epoch in range(2, epochs):
    acc = 1 - 2**-epoch - random.random() / epoch - offset
    loss = 2**-epoch + random.random() / epoch + offset
    print(f"epoch={epoch}, accuracy={acc}, loss={loss}")
    wandb.log({"accuracy": acc, "loss": loss})

# run.log_code()
```

WANDB DEMO

WRAPPING UP

SUMMARY

ANNs are universal function approximators

- Excellent prediction capabilities, but difficult/impossible to understand (black boxes)

Brandon Reagen; Robert Adolf; Paul Whatmough; Gu-Yeon Wei; David Brooks; Margaret Martonosi, "Deep Learning for Computer Architects", Morgan & Claypool, 2017, doi:10.2200/S00783ED1V01Y201706CAC041

SUMMARY

ANNs are universal function approximators

- Excellent prediction capabilities, but difficult/impossible to understand (black boxes)

Backpropagation for learning

- Differentiability mandatory for all layers, activation functions, and further features to come

Brandon Reagen; Robert Adolf; Paul Whatmough; Gu-Yeon Wei; David Brooks; Margaret Martonosi, "Deep Learning for Computer Architects", Morgan & Claypool, 2017, doi:10.2200/S00783ED1V01Y201706CAC041

SUMMARY

ANNs are universal function approximators

- Excellent prediction capabilities, but difficult/impossible to understand (black boxes)

Backpropagation for learning

- Differentiability mandatory for all layers, activation functions, and further features to come

Rich ecosystem around ANN training

- Tracking services such as WandB, but open-source solutions also exist

Brandon Reagen; Robert Adolf; Paul Whatmough; Gu-Yeon Wei; David Brooks; Margaret Martonosi, "Deep Learning for Computer Architects", Morgan & Claypool, 2017, doi:10.2200/S00783ED1V01Y201706CAC041

SUMMARY

ANNs are universal function approximators

- Excellent prediction capabilities, but difficult/impossible to understand (black boxes)

Backpropagation for learning

- Differentiability mandatory for all layers, activation functions, and further features to come

Rich ecosystem around ANN training

- Tracking services such as WandB, but open-source solutions also exist

Simplicity wall: ANNs spend most of their time in matrix multiplications

- Predictability, static loop-trip counts, little control overhead

Brandon Reagen; Robert Adolf; Paul Whatmough; Gu-Yeon Wei; David Brooks; Margaret Martonosi, "Deep Learning for Computer Architects", Morgan & Claypool, 2017, doi:10.2200/S00783ED1V01Y201706CAC041

SUMMARY

ANNs are universal function approximators

- Excellent prediction capabilities, but difficult/impossible to understand (black boxes)

Backpropagation for learning

- Differentiability mandatory for all layers, activation functions, and further features to come

Rich ecosystem around ANN training

- Tracking services such as WandB, but open-source solutions also exist

Simplicity wall: ANNs spend most of their time in matrix multiplications

- Predictability, static loop-trip counts, little control overhead

Brandon Reagen; Robert Adolf; Paul Whatmough; Gu-Yeon Wei; David Brooks; Margaret Martonosi, "Deep Learning for Computer Architects", Morgan & Claypool, 2017, doi:10.2200/S00783ED1V01Y201706CAC041

ANY QUESTIONS?

THIS WEEKS EXERCISE

Review by Group 7 (Liam, Daniela, Marianna)
Curve fitting solution by Group 3 (Paul, Linus, Maximilian)

PLOTTING IN THE EXERCISES

Please include the plots, not just the code for the plots.

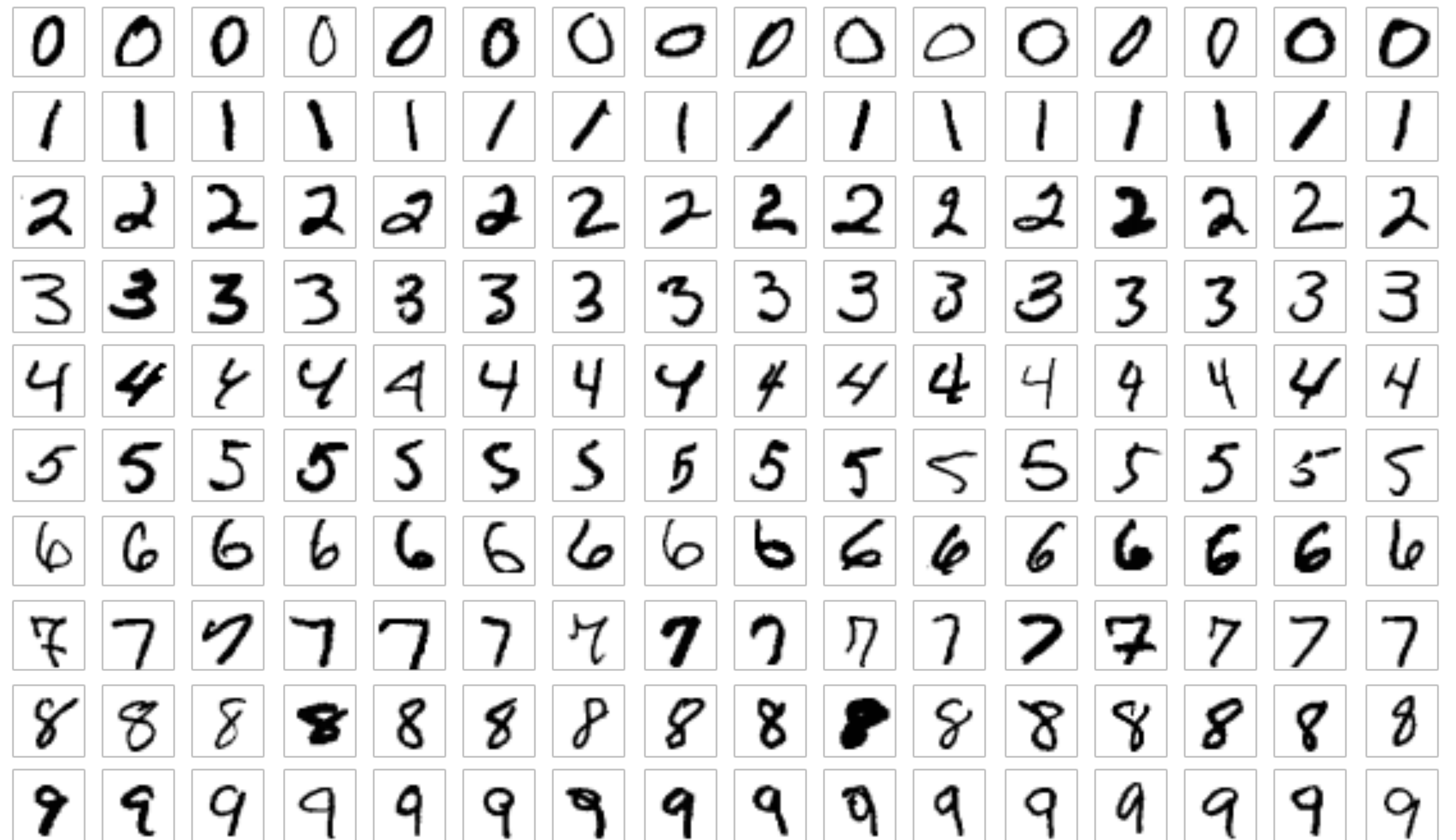
When you submit a plot, add:

- Short explanation of what you expected
- What you actually observed
- How you would explain the observation

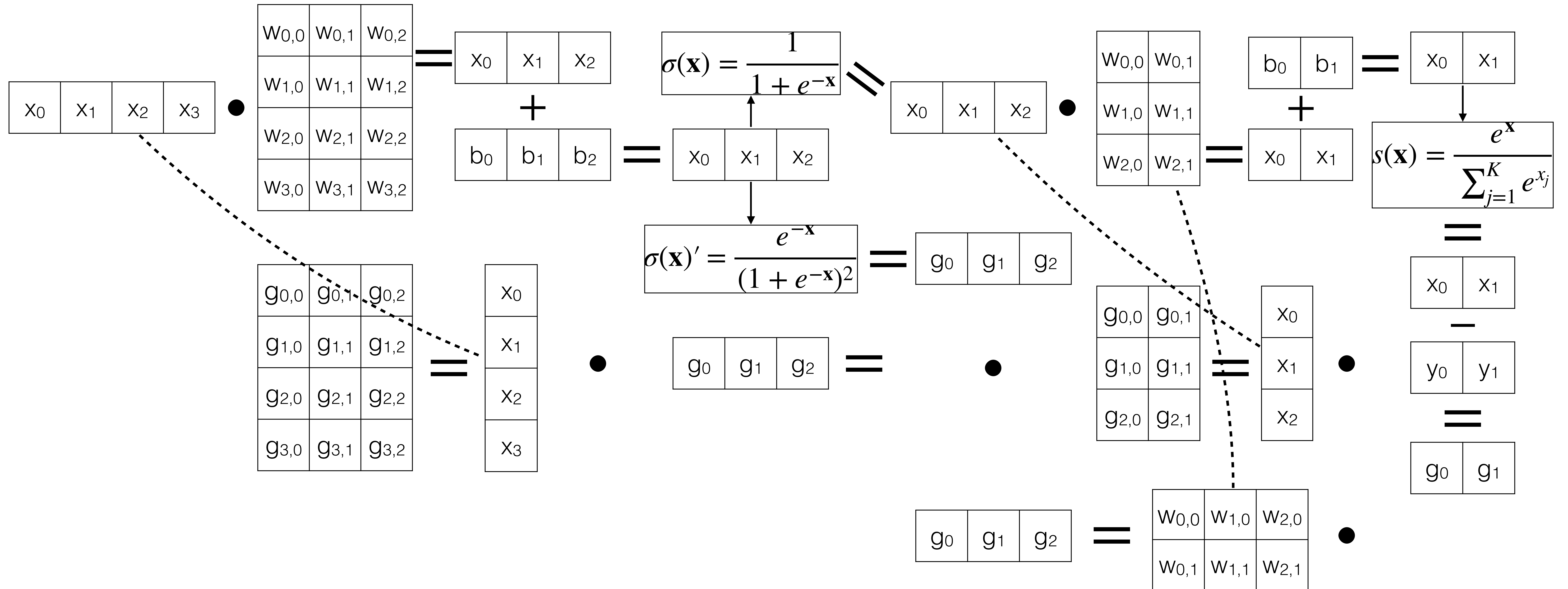
NEXT WEEKS EXERCISE

MNIST DATASET

- 28x28 grey scale images
- 10 classes (digits 0-9)
- 60k training
- 10k testing



EXERCISE 1



Implement your own NN from scratch using exercise02_template.py!

NEXT WEEKS EXERCISE

Implementing an MLP from scratch

Implement forward, backward and update routines

Some small experiments around the learning rate

Submission deadline: Tuesday 09:00 am

https://csg.ziti.uni-heidelberg.de/teaching/ap_nn_from_scratch_materials/

ANY QUESTIONS?