NEURAL NETWORKS FROM SCRATCH
LECTURE 02 - NEURAL NETWORKS

Hendrik Borras, Xiao Wang
{hendrik.borras, xiao.wang}@ziti.uni-heidelberg.de,
HAWAII Lab, Institute of Computer Engineering
Heidelberg University

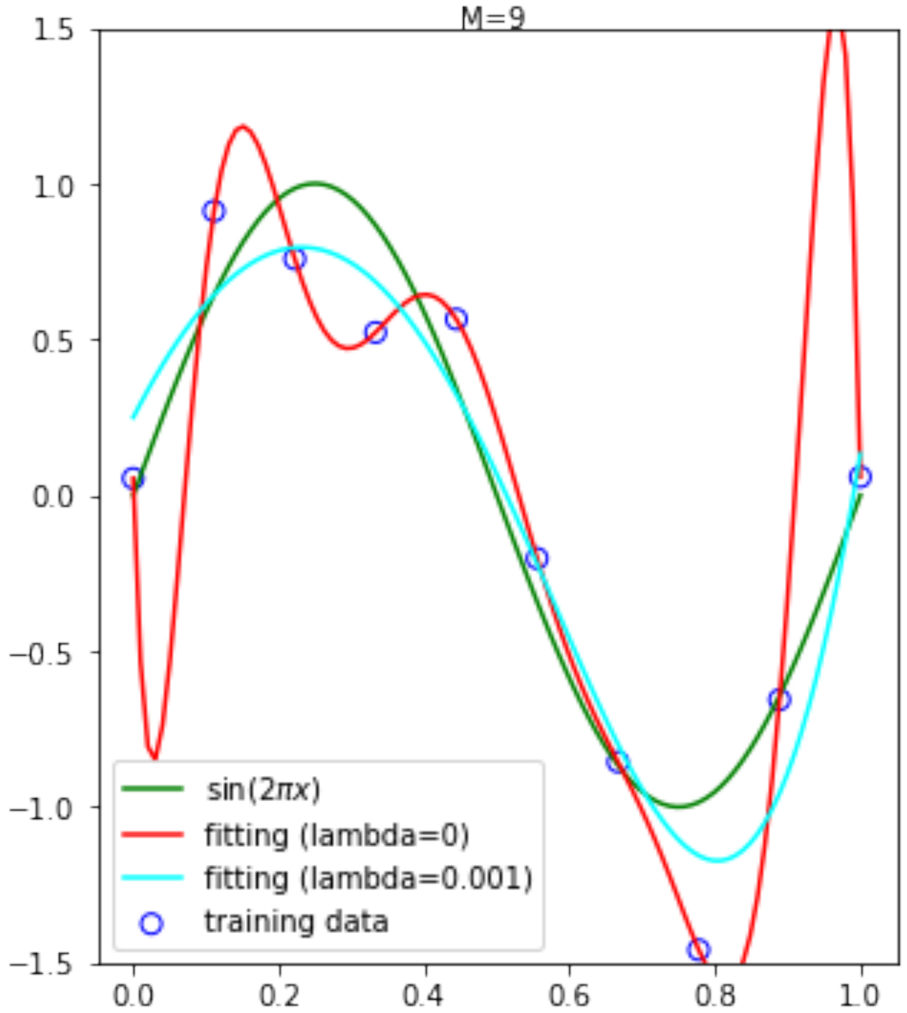
RECAP

Example: Polynomial Fit

- Parameters and training
- Model selection
- Overfitting and regularization

Polynomials are not universal approximators

- But Artificial Neural Networks (ANNs) are
- Subject of today



GLOSSARY

(Mini-)Batch

Number of samples processed before each update

SGD

Stochastic Gradient Descent: Iterative gradient-based optimization using random batches

Hyperparameters

Settings you choose before training (e.g. learning rate, batch size)

Loss

Measure of how wrong the model's predictions are ("Error"/"Objective")

Overfitting

Model learns specifics of the training set (noise) instead of general patterns

Regularisation

Methods to keep models simpler and prevent overfitting

Train/Val/Test split

Separate data for fitting, tuning, and evaluating

ARTIFICIAL NEURAL NETWORKS

ARTIFICIAL NEURAL NETWORKS (ANNS)

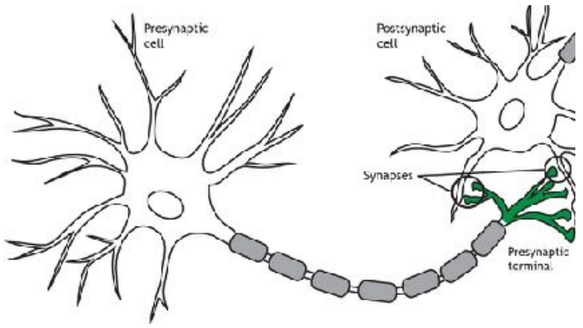
ANNs ?= Machine Learning

Historically no, today kinda yes

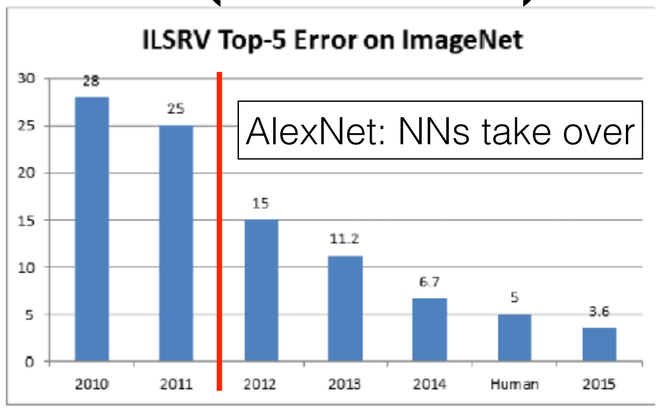
Kind of inspired by biology

!= spiking neural networks

c.f. “non-differentiable”



Biological Synapse Structure

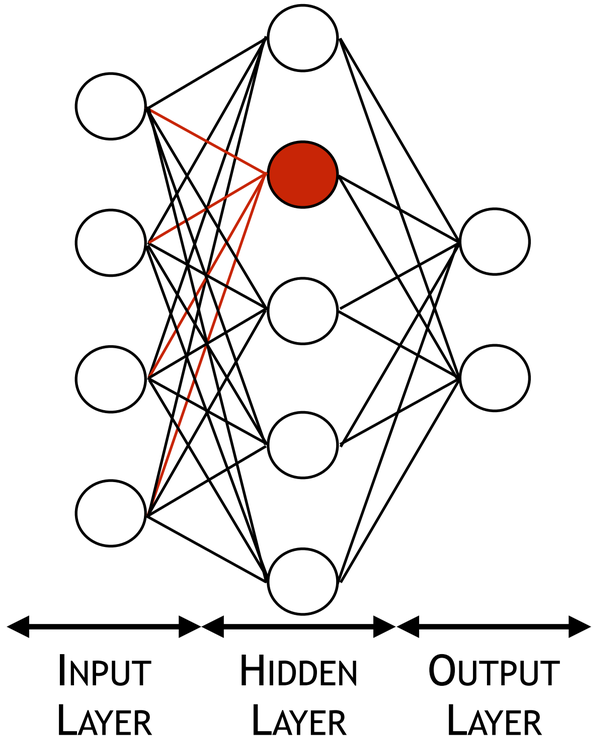


More complex problems require more complex models

Informal term of “model capacity” (or expressivity)

ANNs are universal function approximators

Universal approximation theorem(s):
Neural networks can represent a wide variety of interesting functions with appropriate weights



MULTI-LAYER PERCEPTRON (MLP)

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, …)

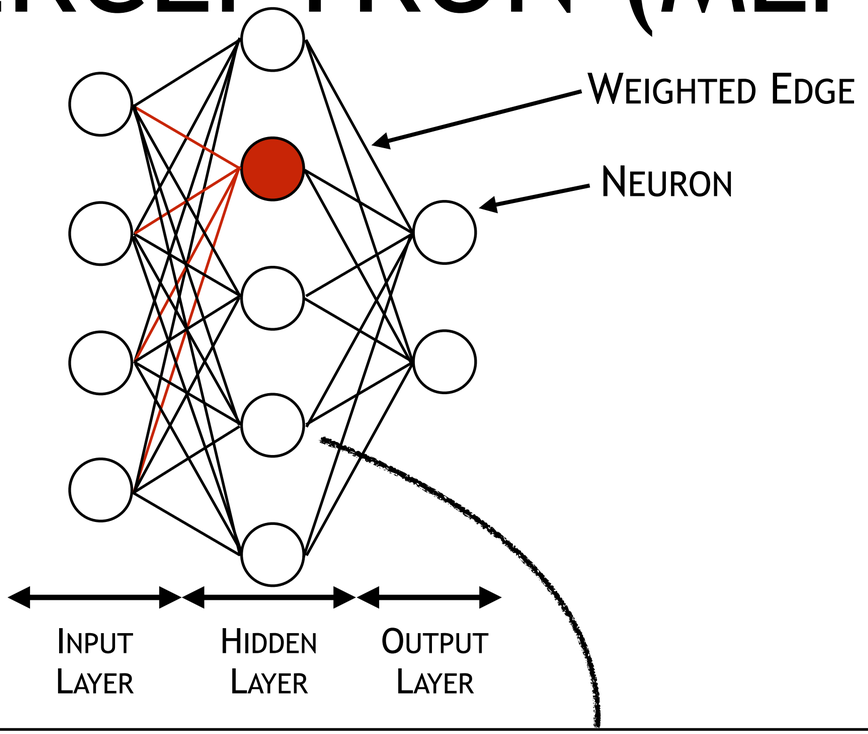
$\mathbf{W}$: weight matrix

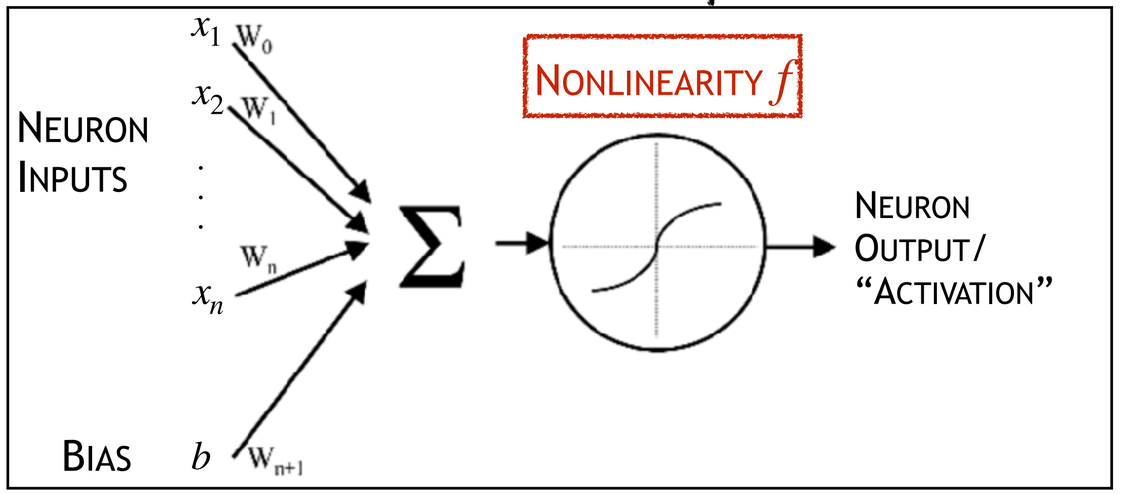
$\mathbf{x}$: activation vector

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

Vector notation for layer l

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





INTERLUDE: INTUITIVE VIEW

Learning logical operations

AND: $x_1 + x_2 > 1.5 \rightarrow x_1 + x_2 - 1.5 = 0$

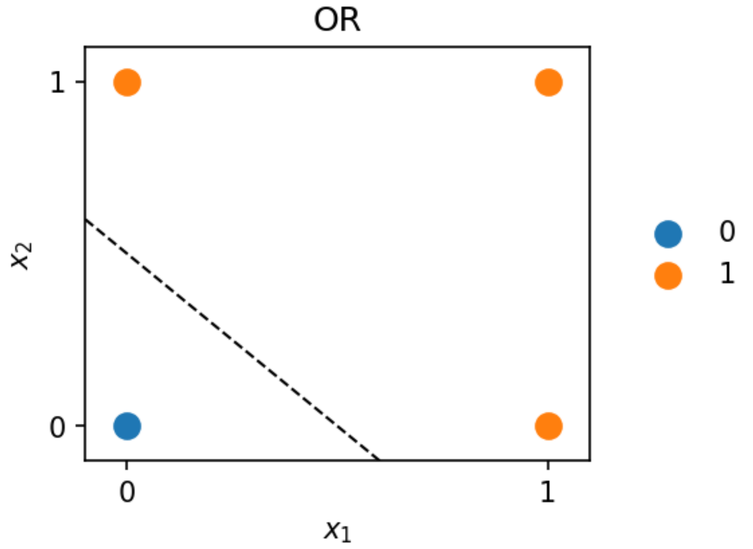
OR: $x_1 + x_2 > 0.5 \rightarrow x_1 + x_2 - 0.5 = 0$

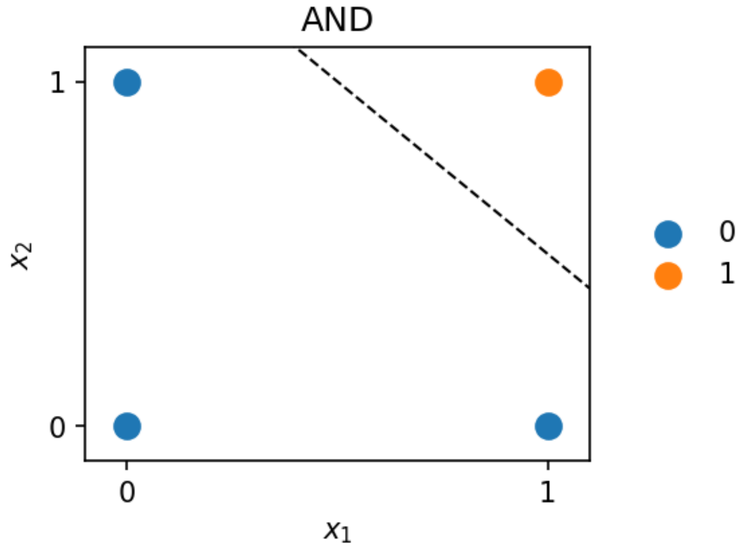
A single neuron can do the same

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

AND: $w_1 = w_2 = 1, b = -1.5$

OR: $w_1 = w_2 = 1, b = -0.5$





INTERLUDE: INTUITIVE VIEW

Learning XOR

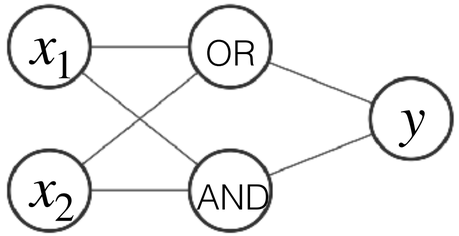
XOR: Not Linearly Separable

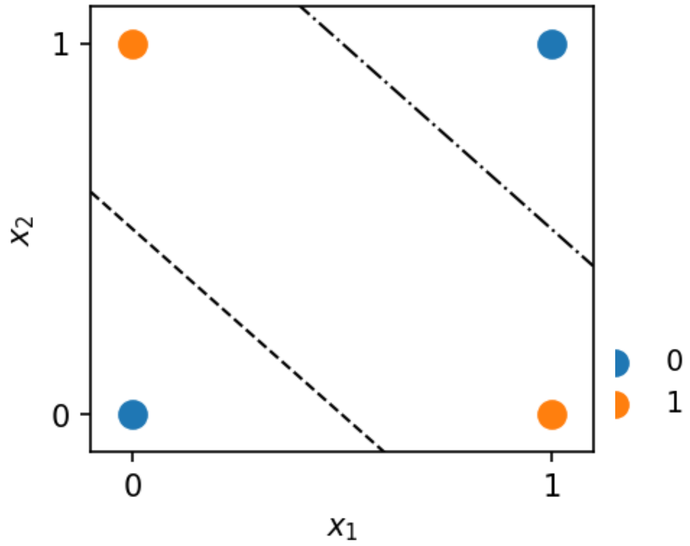
But two (three) neurons can still solve the problem: XOR=OR-AND

$$y_{\text{OR}} = f\left(w_{11}\, x_1 + w_{12}\, x_2 + b_1\right)$$

$$y_{\text{AND}} = f\left(w_{21}\, x_1 + w_{22}\, x_2 + b_2\right)$$

$$y = f\left(w_{\text{OR}}\, y_{\text{OR}} + w_{\text{AND}}\, y_{\text{AND}} + b\right)$$





	OR	AND	XOR
0,0	0/-0.5	0/-1.5	0/1.0
0,1	1/0.5	0/-0.5	1/1.0
1,0	1/0.5	0/-0.5	1/1.0
1,1	1/1.5	1/0.5	0/1.0

Nonlinearities are essential

Without them, this is just a sum of linear functions (still linear)

$$\mathbf{W_1} \cdot (\mathbf{W_2} \cdot \mathbf{x}) = \mathbf{W} \cdot \mathbf{x}$$

EXAMPLE NONLINEARITIES

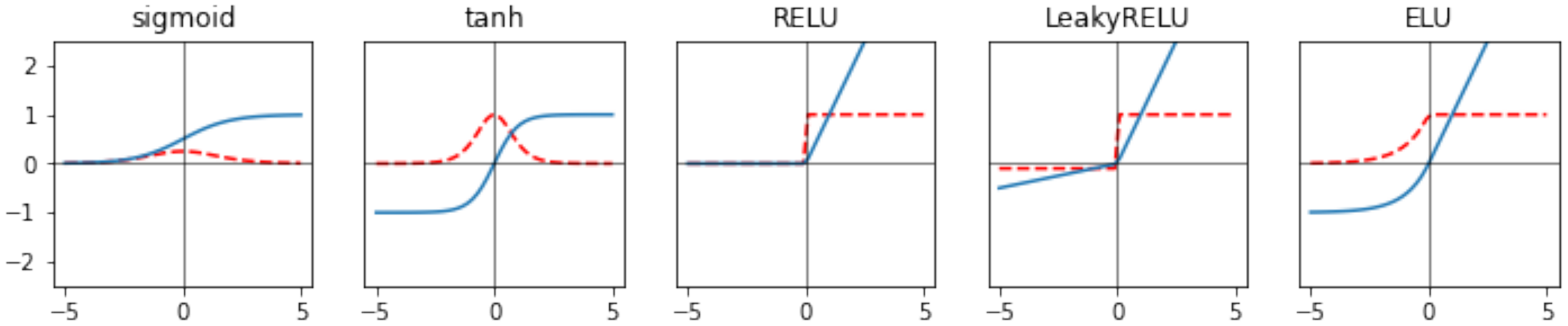
sigmoid: $f(x) = \frac{1}{1+e^{-x}}$ => output in range $[0,1]$

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



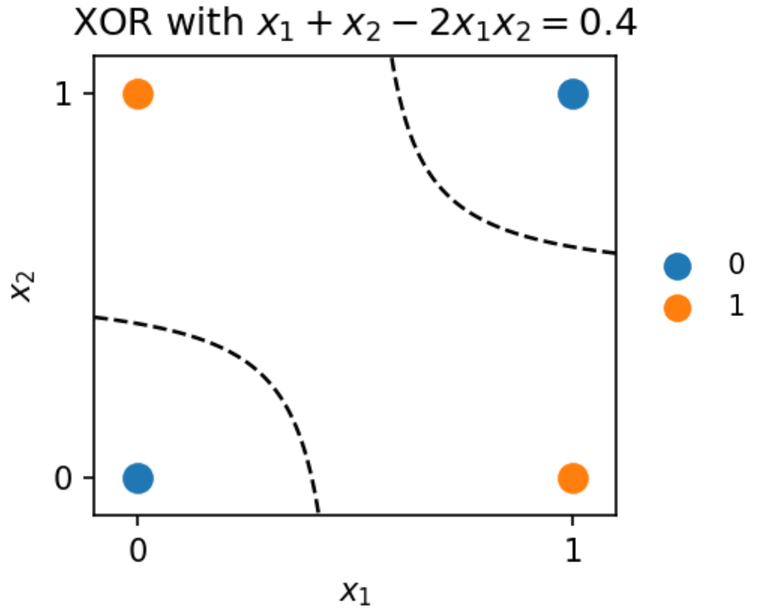
Basically any non-linear function can be used

INTERLUDE: INTUITIVE VIEW

XOR: Why not use a polynomial again?

E.g.: $x_1 + x_2 - 2x_1x_2 + 0x_1^2 + 0x_2^2 = 0.4$ works!

But: Combinatorial scaling, $N_{\text{terms}} \sim \binom{n_{\text{data}} + d_{\text{poly}}}{d_{\text{poly}}}$



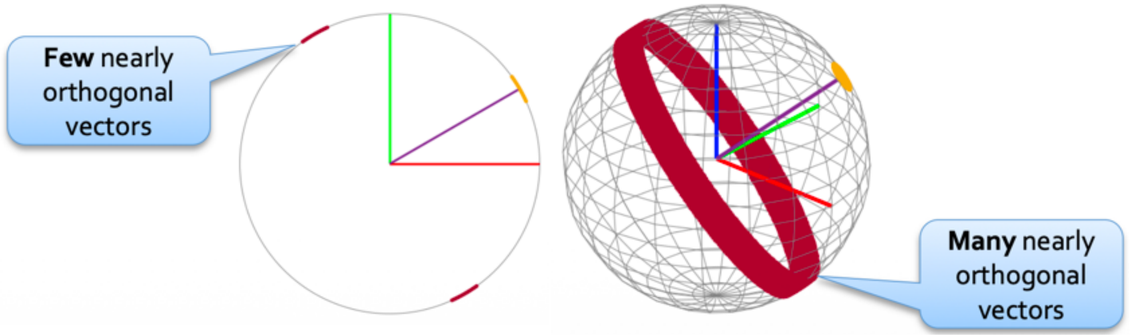
Curse of dimensionality

One pixel = one dimension

MNIST: $28 \times 28px \rightarrow n_{data} = 784$

Dims	Terms
n=3, d=3	20
n=5, d=3	56
n=10, d=3	286
n=100, d=3	~160k

INTERLUDE: INTUITIVE VIEW

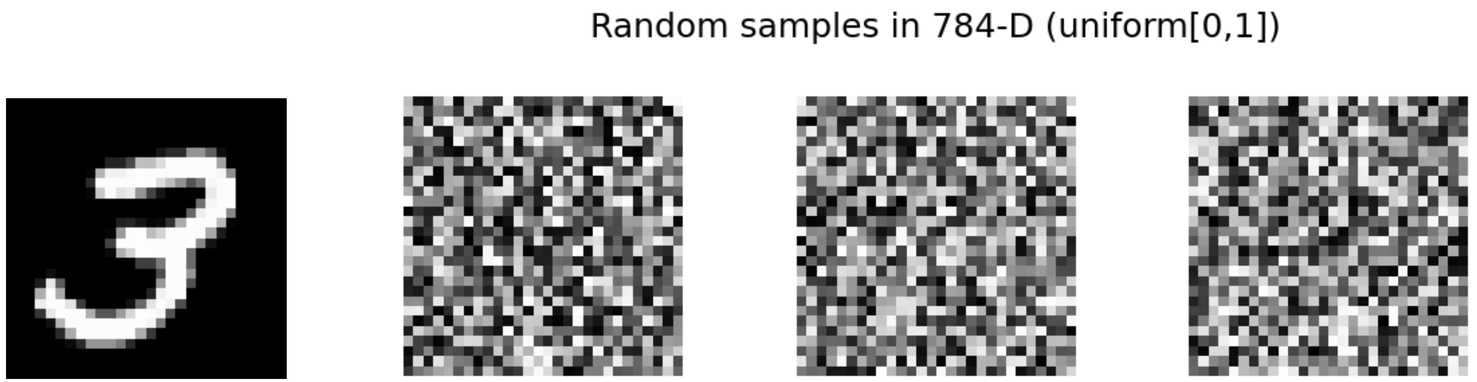


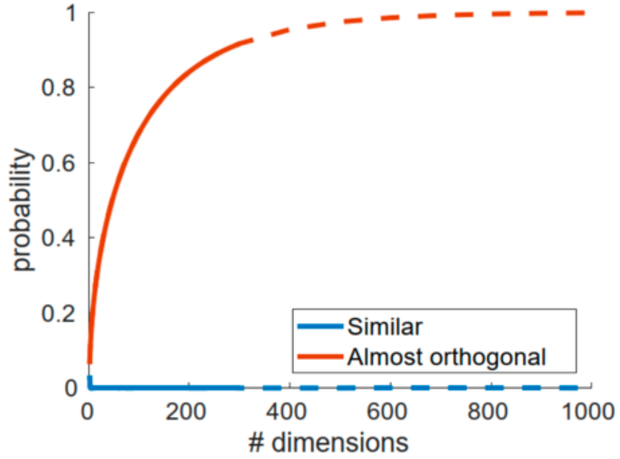
High dimensions are weird: In high d…

- almost all of the volume of a hypersphere/cube is close to its surface
- All points are equally “far away”, approximately orthogonal

Our only hope: Lower-dimensional manifolds

TL;DR: Neural networks are “smart mappers”



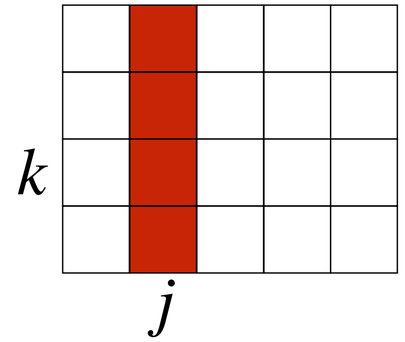


VECTOR AND MATRIX NOTATION

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

Matrix = bold uppercase

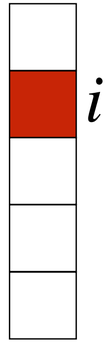
Matrix element $w_{k,j}$ has row k, column j



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

Vector = bold lowercase

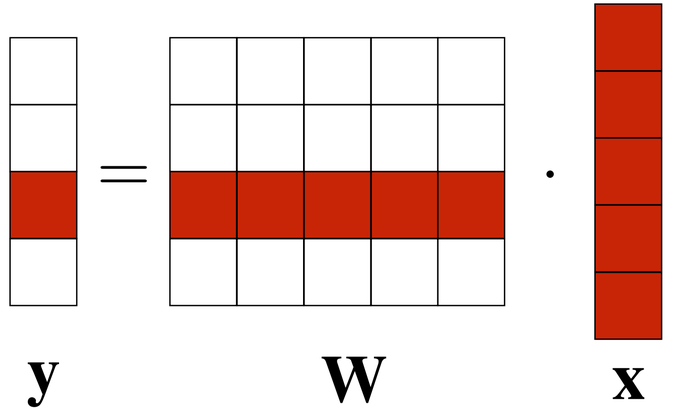
Vectors are vertical, use $\mathbf{x}^T$ for horizontal vectors



Matrix-vector multiplication

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

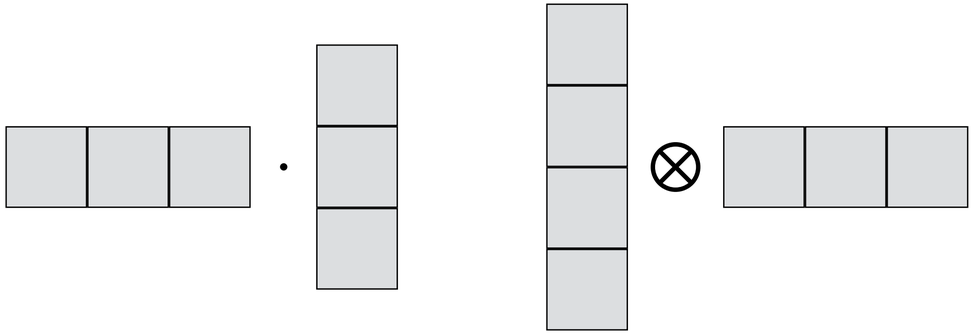
$y_k = \sum_j (w_{k,j} \cdot x_j)$, resp. $\mathbf{y} = \mathbf{W} \cdot \mathbf{x}$



Vector-vector multiplication

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

Outer product: $\mathbf{A} = \mathbf{c} \otimes \mathbf{b} = \mathbf{b}\mathbf{c}^T$



MULTI-LAYER PERCEPTRON (MLP)

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, …)

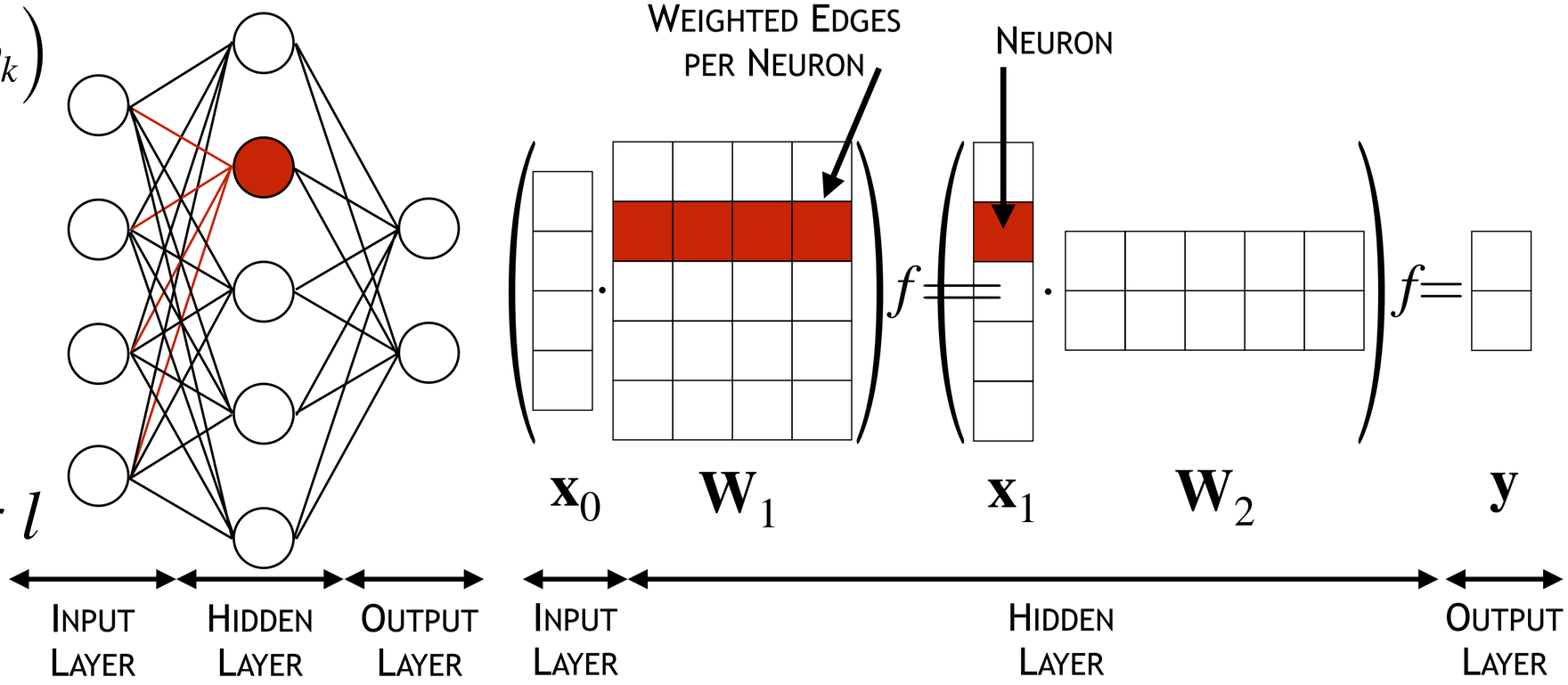
$\mathbf{W}$: weight matrix

$\mathbf{x}$: activation vector

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

Vector notation for layer l

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



FORWARD PROP ON ONE SLIDE

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

$\mathbf{x}_l = f(\mathbf{W}_l \oplus \mathbf{x}_{l-1} + \mathbf{b}_l)$, for nonlinear activation function $f(\cdot)$, linear operation $\oplus$, 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 constant activation element (e.g., $x_0 = 1$)

Then a complete MLP with L layers is

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

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: $s(x) = \frac{e^{x_i}}{\Sigma_{j=1}^{K} e^{x_j}}$, 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)

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

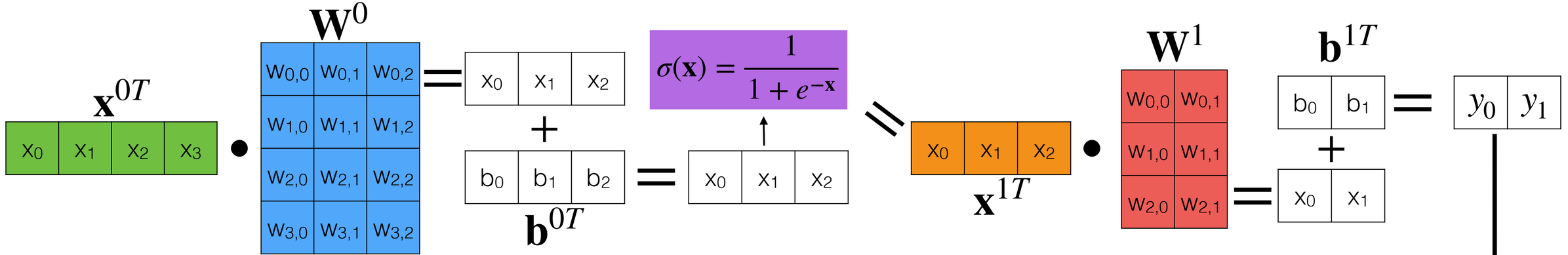
See also log-softmax

Why softmax instead of argmax?

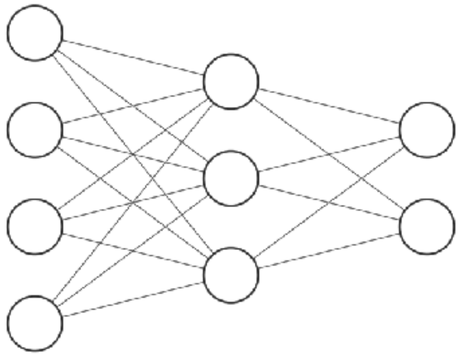
y	x	argmax(x)	softmax(x)	norm(x)
0	1	0,0 %	7,0 %	13,1 %
0	0,1	0,0 %	2,9 %	1,3 %
0	3	100,0 %	51,9 %	39,2 %
0	0,1	0,0 %	2,9 %	1,3 %
0	0,4	0,0 %	3,9 %	5,2 %
1	2	0,0 %	19,1 %	26,1 %
0	1	0,0 %	7,0 %	13,1 %
0	0,05	0,0 %	2,7 %	0,7 %
0	0	0,0 %	2,6 %	0,0 %

CALCULATING A NEURAL NETWORK

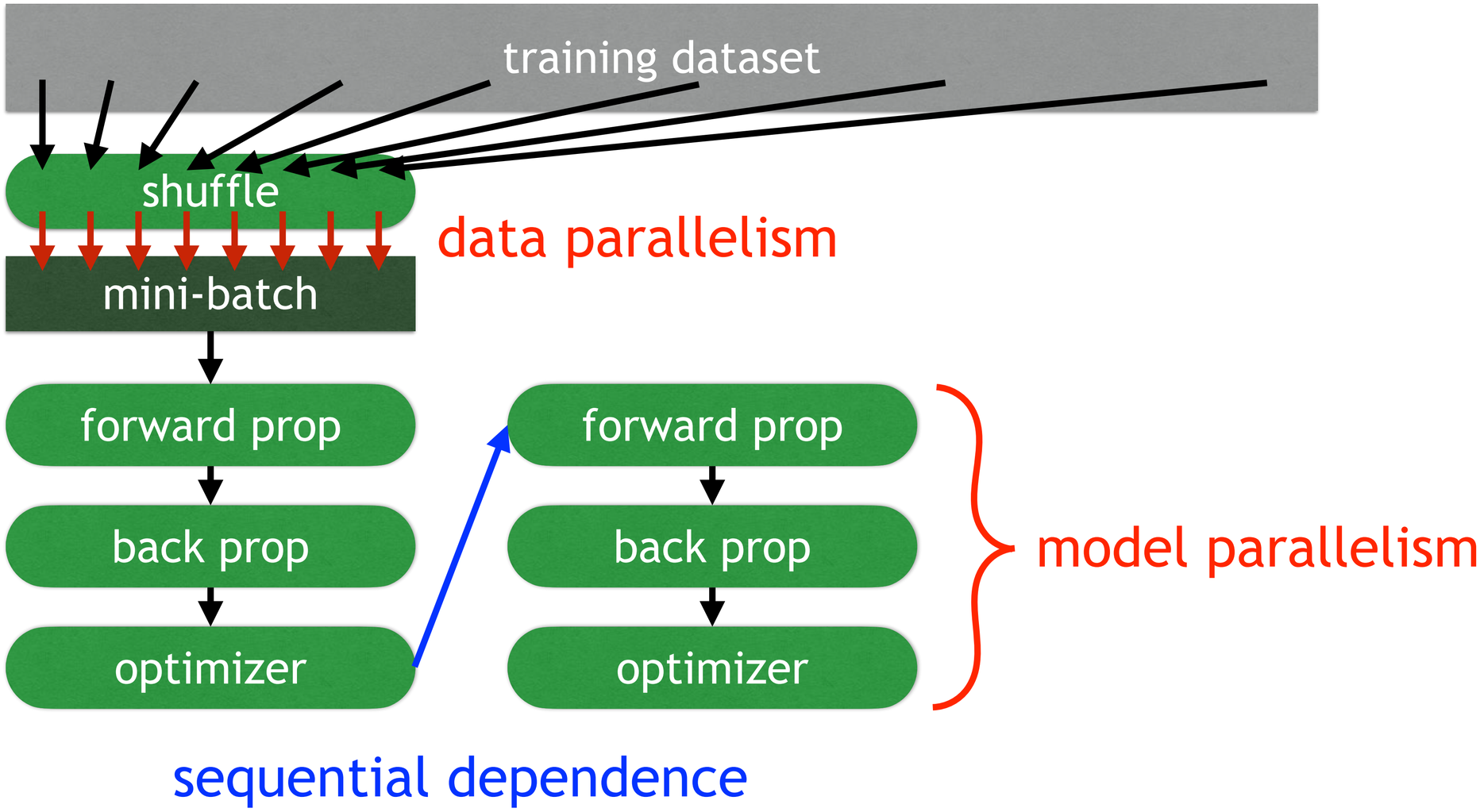
Forward pass



$$\mathbf{y} = s(\mathbf{W}^1(\sigma(\mathbf{W}^0\mathbf{x}^0 + \mathbf{b}^0)) + \mathbf{b}^1)$$



TRAINING OF DEEP NEURAL NETWORKS



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

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

data term l that penalizes wrong prediction (error function)

regularizer $r(\mathbf{W})$ (e.g. ℓ^2–norm) penalizes large weights, trade-off hyperparameter λ

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

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

PARTIAL DERIVATIVES SOUGHT

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

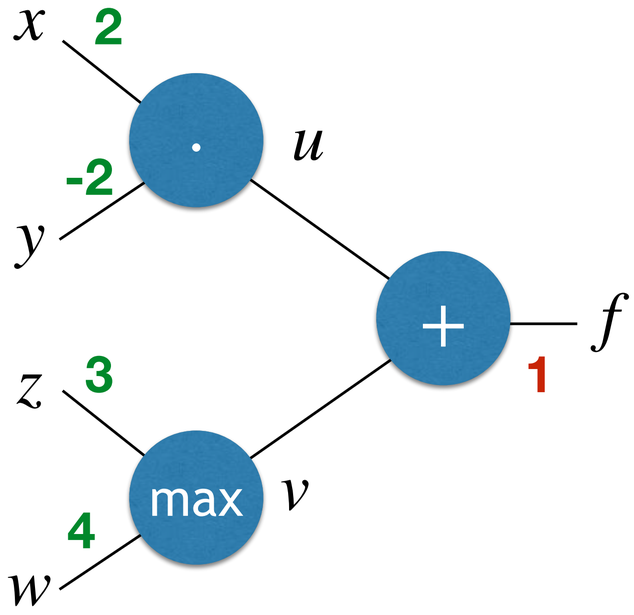
$$\frac{\partial}{\partial x} f, \frac{\partial}{\partial y} f, \frac{\partial}{\partial z} f, \frac{\partial}{\partial w} f \quad ?$$

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: $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial u}\frac{\partial u}{\partial x}$

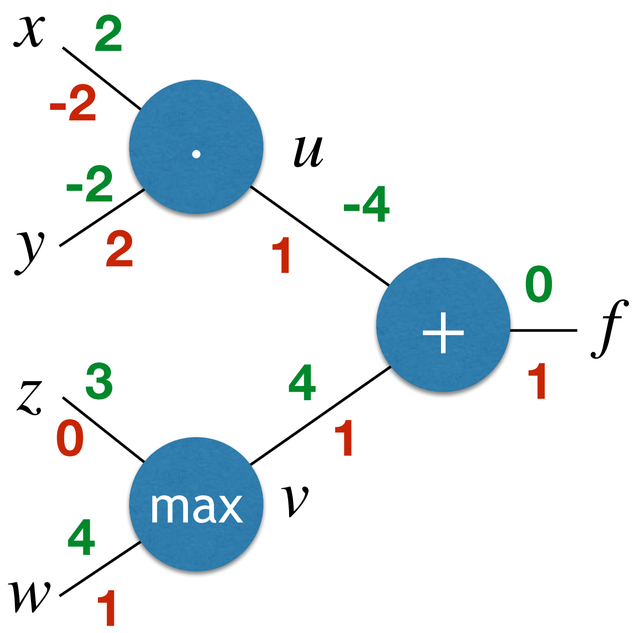
$$\frac{\partial f}{\partial u} = \frac{\partial add(u,v)}{\partial u} = 1; \frac{\partial u}{\partial x} = \frac{\partial mul(x,y)}{\partial x} = y$$

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

Others in an analogous manner

Potential problem: vanishing gradient

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



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

Kaiming He, Deep Residual Learning for Image Recognition, CVPR 2016

GRADIENT BEHAVIOR

add gate: gradient distributor

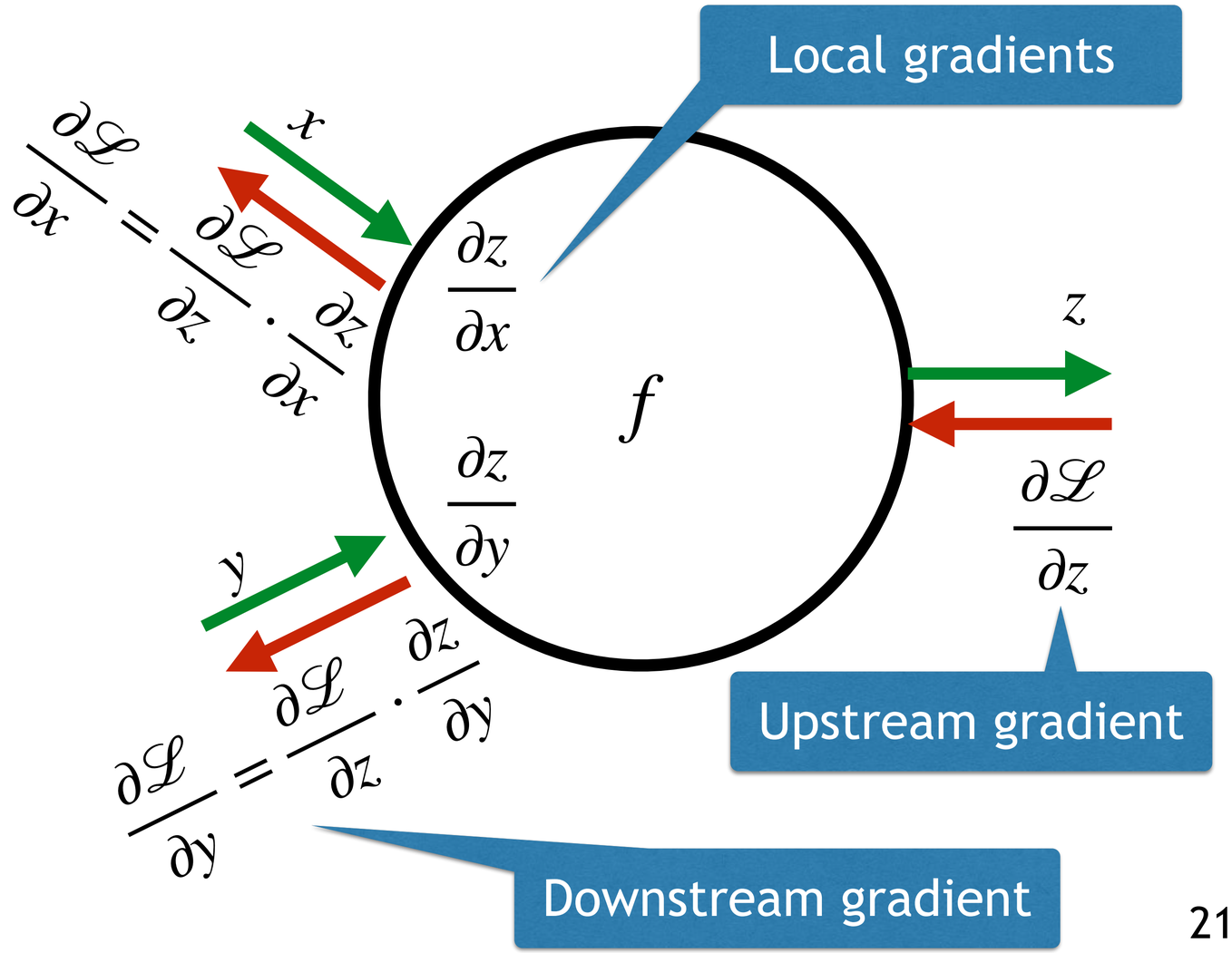
Both downstream gradients equal upstream gradient

mul gate: gradient switcher

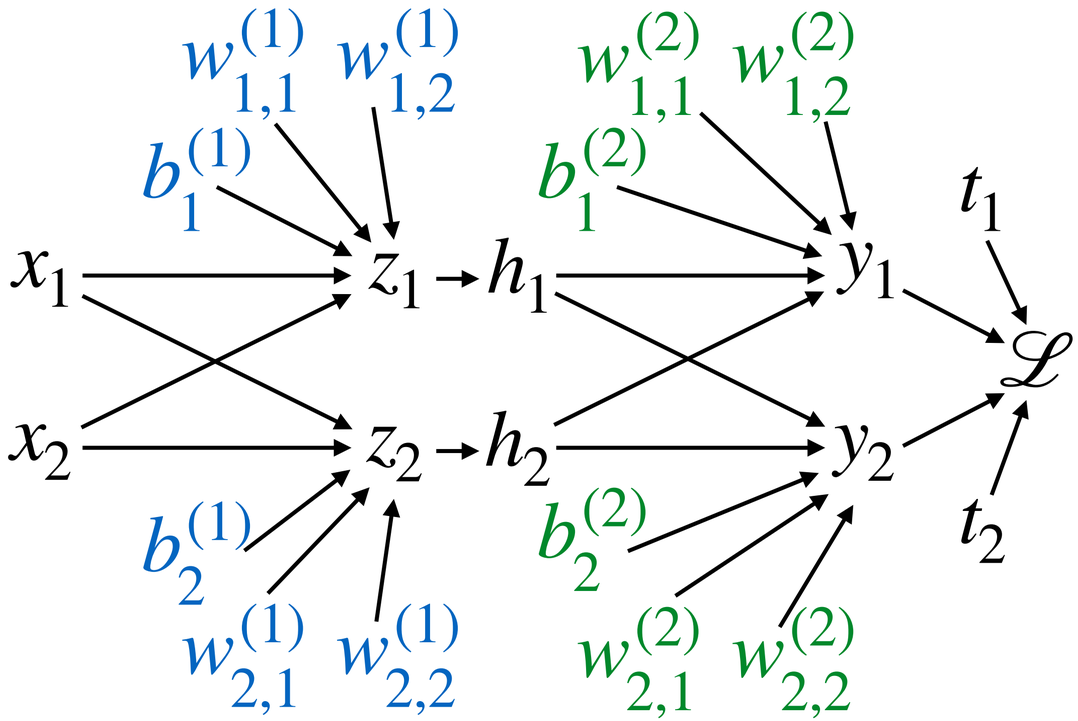
Downstream gradients multiplied with other input

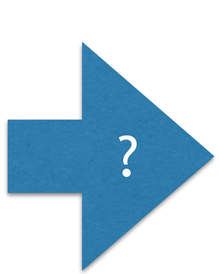
max gate: gradient router

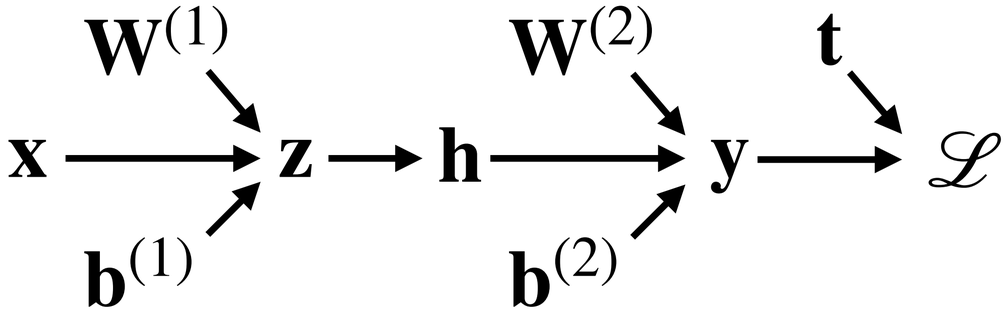
Downstream gradients depend on comparison of inputs



VECTOR CALCULUS







VECTOR CALCULUS

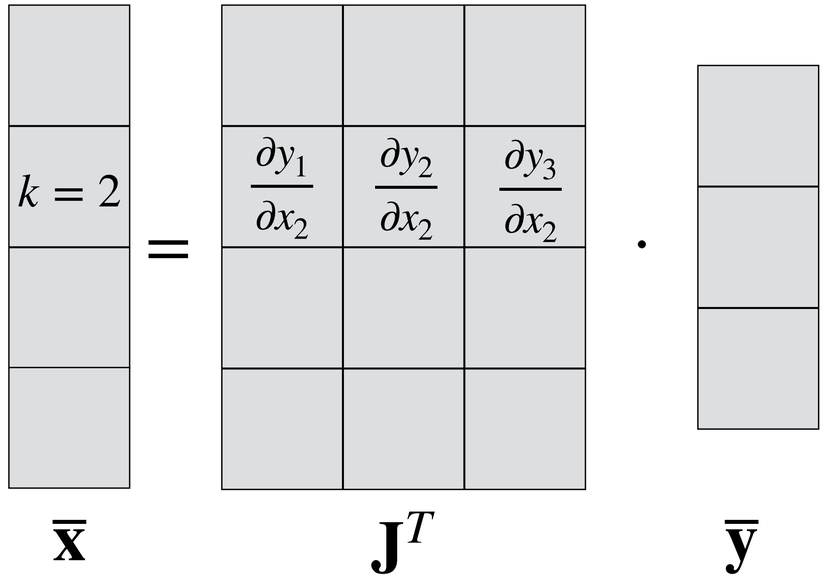
Consider a vector-valued function $(y_1 \dots y_m)^T = f(x_1 \dots x_n)^T$

Or $\mathbf{y} = f(\mathbf{x})$

Then the partial derivatives are $\overline{x_k} = \sum_i \overline{y_i} \frac{\partial y_i}{\partial x_k}, \ \forall k$

Based on a Jacobian matrix (in numerator layout)

$$\mathbf{J} = \left(\frac{\partial \mathbf{y}}{\partial x_1} \dots \frac{\partial \mathbf{y}}{\partial x_n} \right) = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$



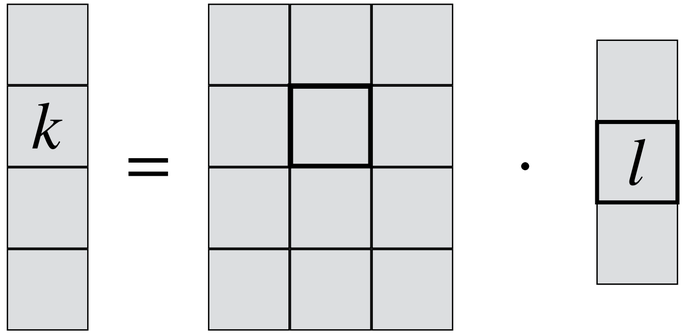
the partial derivatives can be formulated as $\overline{\mathbf{x}} = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}} \right)^T \overline{\mathbf{y}} = \mathbf{J}^T \overline{\mathbf{y}}$

VECTOR CALCULUS - PARTIAL DERIVATIVE TO X

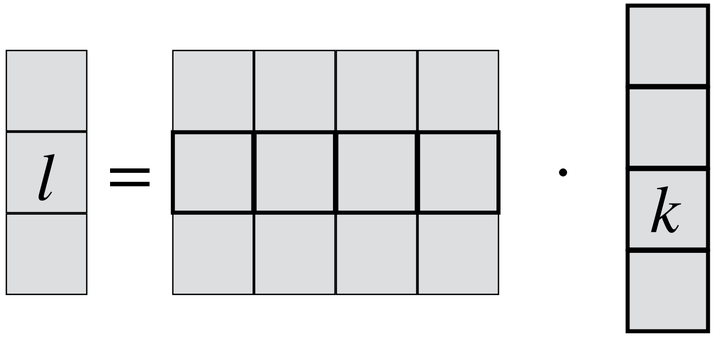
Matrix-vector multiplication $\mathbf{y} = \mathbf{W}\mathbf{x}$ resp. $y_k = \sum_i w_{k,i} x_i$

$$\frac{\partial y_k}{\partial x_l} = \frac{\partial}{\partial x_l} \sum_i w_{k,i} x_i = w_{k,l} \Rightarrow \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \mathbf{W} \Rightarrow \bar{\mathbf{x}} = \mathbf{W}^T \bar{\mathbf{y}} \text{ (chain rule)}$$

$$\Rightarrow \mathbf{J} = \mathbf{W}$$



$\mathbf{y}$ $\mathbf{W}$ $\mathbf{x}$

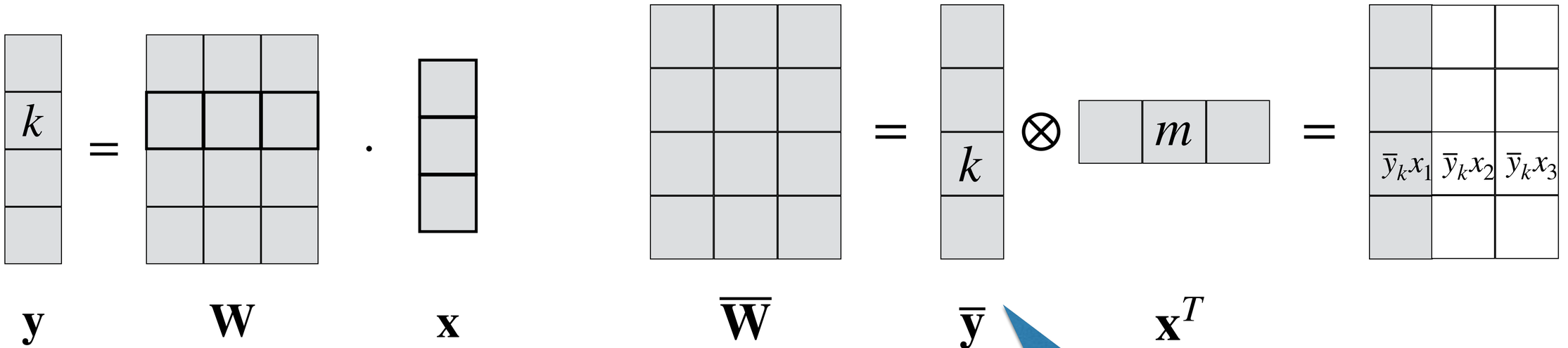


$\bar{\mathbf{x}}$ $\mathbf{W}^T$ $\bar{\mathbf{y}}$

VECTOR CALCULUS - PARTIAL DERIVATIVE TO W

Matrix-vector multiplication $\mathbf{y} = \mathbf{W}\mathbf{x}$ resp. $y_k = \sum_i w_{k,i} x_i$

$$\frac{\partial y_k}{\partial w_{l,m}} = \frac{\partial}{\partial w_{l,m}} \sum_i w_{k,i} x_i = \begin{cases} x_m; k = l \\ 0; \text{else} \end{cases} \Rightarrow \overline{\mathbf{W}} = \overline{\mathbf{y}} \otimes \mathbf{x} \text{ (chain rule)}$$



VECTOR CALCULUS - PARTIAL DERIVATIVE TO Y

Elementwise operation $\mathbf{y} = e^{\mathbf{x}}$, resp. $y_k = e^{x_k}$

e.g. sigmoid activation function

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} e^{x_0} & & 0 \\ & \ddots & \\ 0 & & e^{x_n} \end{bmatrix}$$ and thus $\overline{\mathbf{x}} = e^{\mathbf{x}} \circ \overline{\mathbf{y}}$ (chain rule)

Only for $k = l$, k being an element of $\mathbf{y}$ and l an element of $\mathbf{x}$, the partial derivate is nonzero

Hadamard product, $\circ$ (elementwise multiplication)

MULTILAYER NETWORKS - VECTORIZED VIEW

$$\overline{y_k} = \frac{\partial \mathscr{L}}{\partial y_k} = \frac{\partial}{\partial y_k}\left(\frac{1}{2}\sum_i (y_i - t_i)^2\right) = y_k - t_k \qquad \overline{\mathbf{y}} = (\mathbf{y} - \mathbf{t})$$

$$\overline{w_{k,j}^{(2)}} = \frac{\partial \mathscr{L}}{\partial y_k} \cdot \frac{\partial y_k}{\partial w_{k,j}^{(2)}} = \overline{y_k} \cdot \frac{\partial}{\partial w_{k,j}^{(2)}}\left(\sum_i (w_{k,i}^{(2)} h_i) + b_k^{(2)}\right) = \overline{y_k} \cdot h_j \qquad \overline{\mathbf{W}^{(2)}} = \overline{\mathbf{y}}\mathbf{h}^T$$

$$\overline{b_k^{(2)}} = \frac{\partial \mathscr{L}}{\partial y_k} \cdot \frac{\partial y_k}{\partial b_k^{(2)}} = \overline{y_k} \cdot \frac{\partial}{\partial b_k^{(2)}}\left(\sum_i (w_{k,i}^{(2)} h_i) + b_k^{(2)}\right) = \overline{y_k} \qquad \overline{\mathbf{b}^{(2)}} = \overline{\mathbf{y}}$$

$$\overline{h_k} = \sum_j \frac{\partial \mathscr{L}}{\partial y_j} \cdot \frac{\partial y_j}{\partial h_k} = \sum_j \overline{y_j} \cdot \frac{\partial}{\partial h_k}\left(\sum_i (w_{j,i}^{(2)} h_i) + b_j^{(2)}\right) = \sum_j \overline{y_j} \cdot w_{j,k}^{(2)} \qquad \overline{\mathbf{h}} = \mathbf{W}^{(2)T}\overline{\mathbf{y}}$$

$$\overline{z_k} = \frac{\partial \mathscr{L}}{\partial h_k} \cdot \frac{\partial h_k}{\partial z_k} = \overline{h_k} \cdot \frac{\partial}{\partial z_k}\sigma(z_k) = \overline{h_k}\sigma'(z_k) \qquad \overline{\mathbf{z}} = \overline{\mathbf{h}} \circ \sigma'(\mathbf{z})$$

$$\overline{w_{k,j}^{(1)}} = \frac{\partial \mathscr{L}}{\partial z_k} \cdot \frac{\partial z_k}{\partial w_{k,j}^{(1)}} = \overline{z_k} \cdot \frac{\partial}{\partial w_{k,j}^{(1)}}\left(\sum_i (w_{k,i}^{(1)} x_i) + b_k^{(1)}\right) = \overline{z_k} \cdot x_j \qquad \overline{\mathbf{W}^{(1)}} = \overline{\mathbf{z}}\mathbf{x}^T$$

$$\overline{b_k^{(1)}} = \frac{\partial \mathscr{L}}{\partial z_k} \cdot \frac{\partial z_k}{\partial b_k^{(1)}} = \overline{z_k} \cdot \frac{\partial}{\partial b_k^{(1)}}\left(\sum_i (w_{k,i}^{(1)} x_i) + b_k^{(1)}\right) = \overline{z_k} \qquad \overline{\mathbf{b}^{(1)}} = \overline{\mathbf{z}}$$

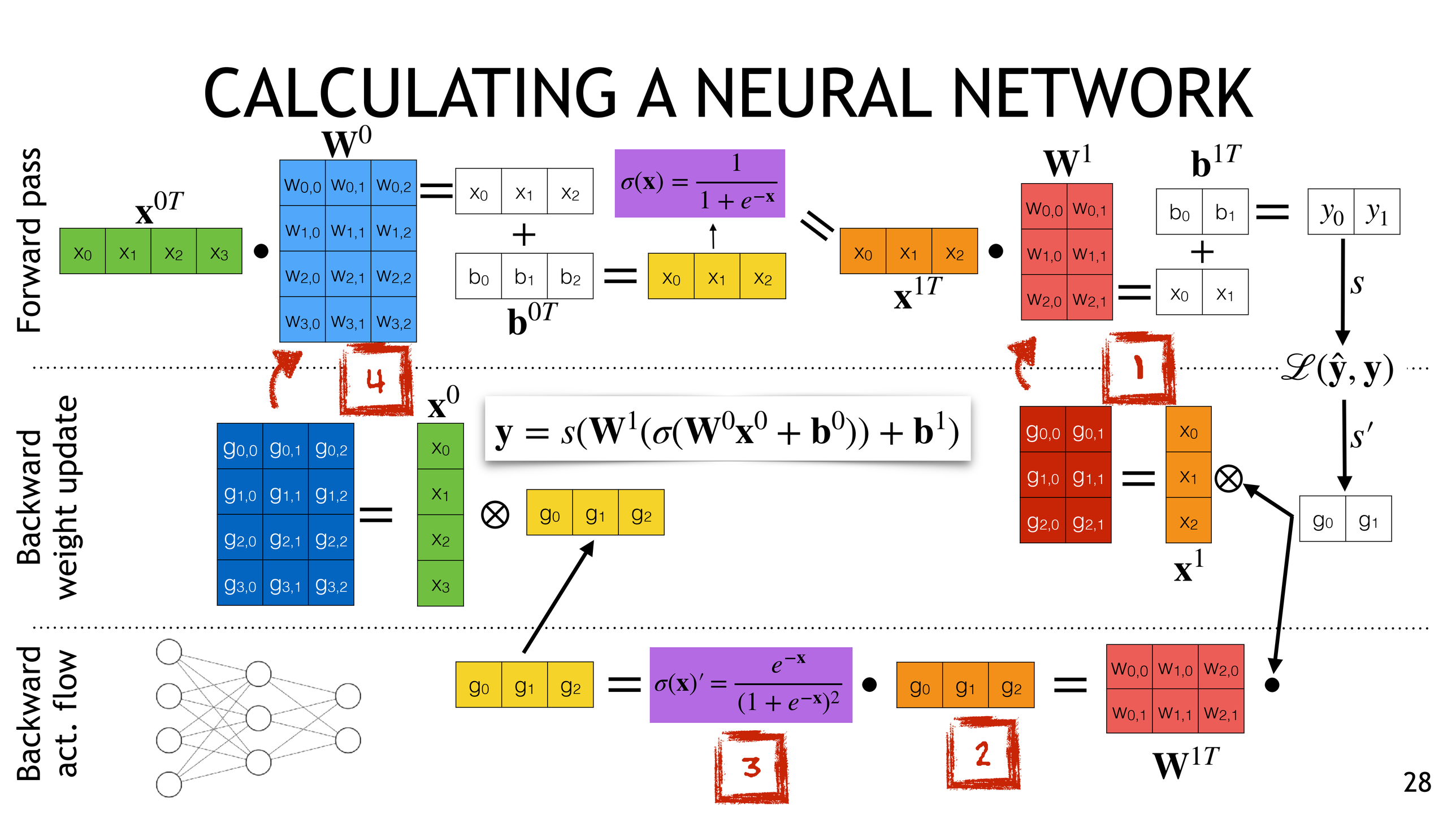
Transpose or not transpose, check the shape.

CALCULATING A NEURAL NETWORK

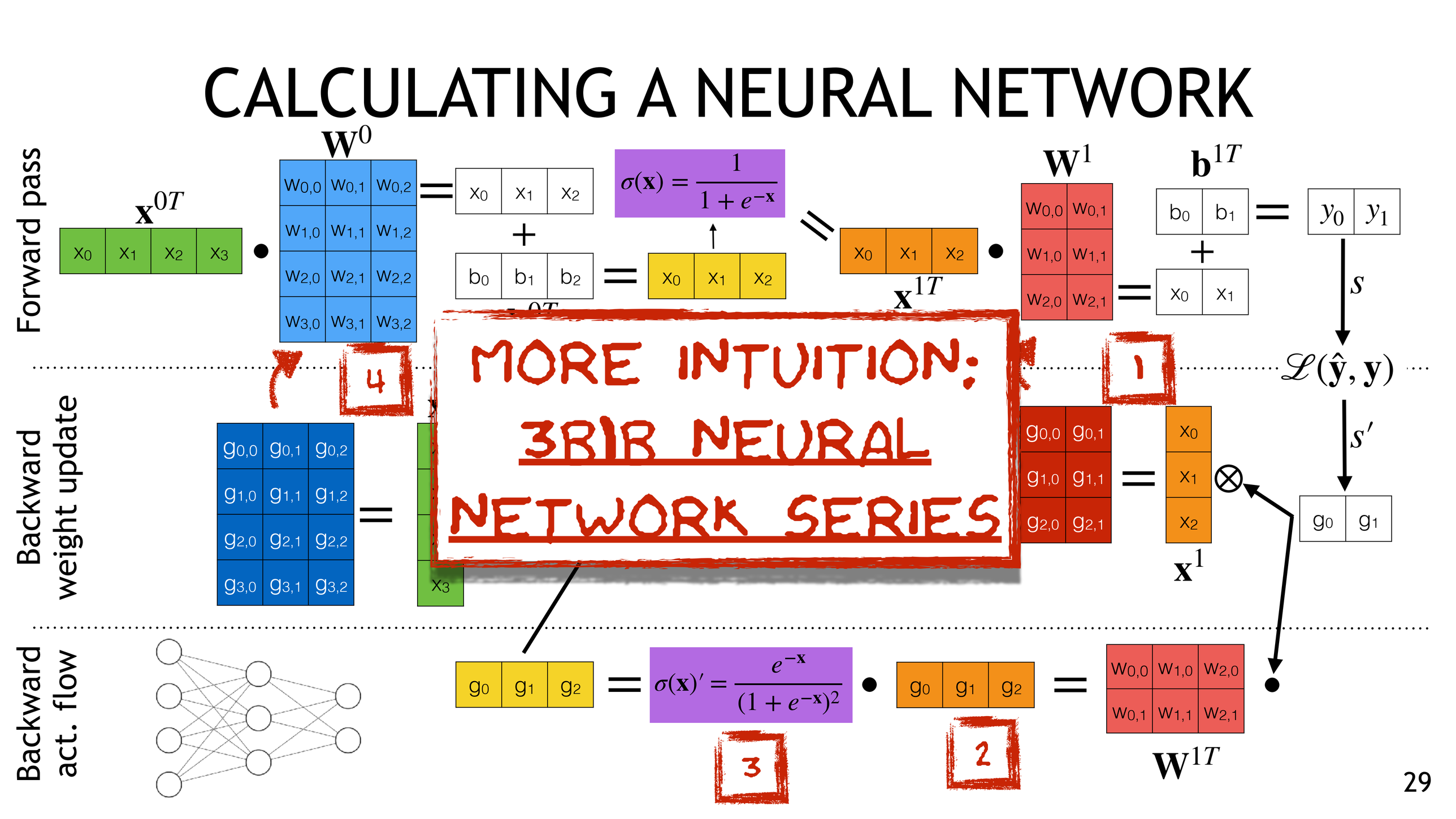
Forward pass

Backward weight update

Backward act. flow

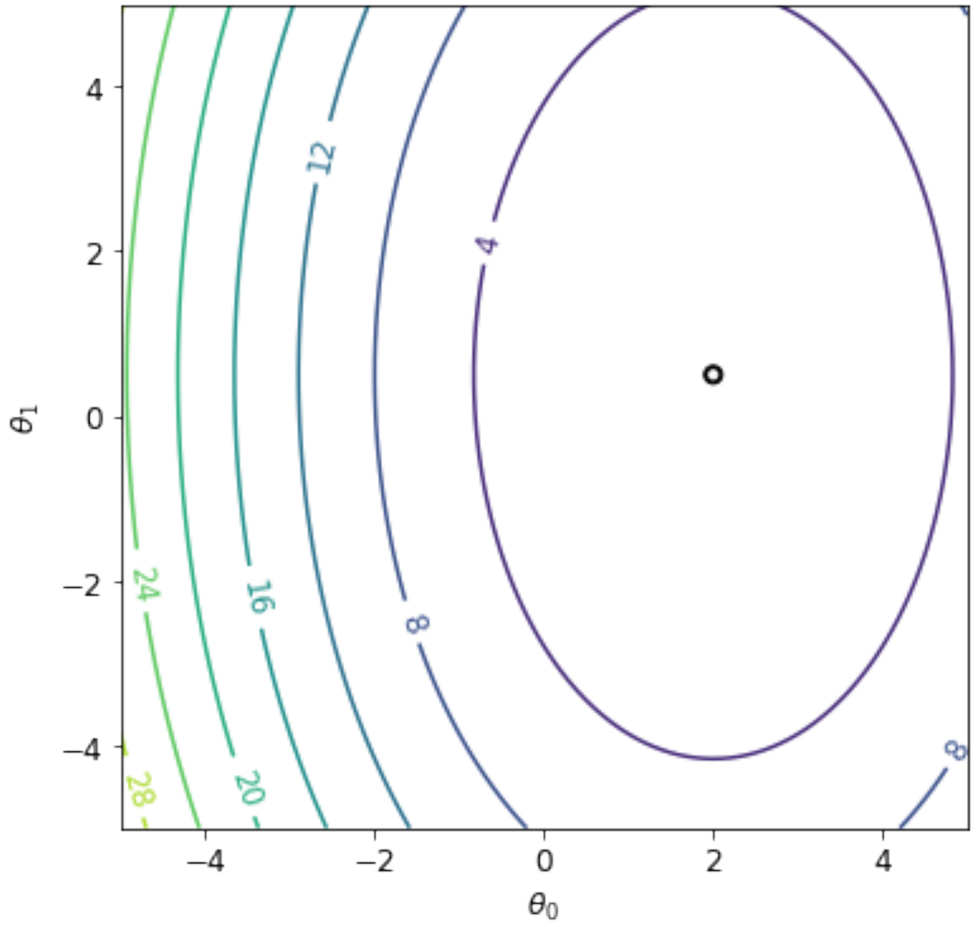


CALCULATING A NEURAL NETWORK

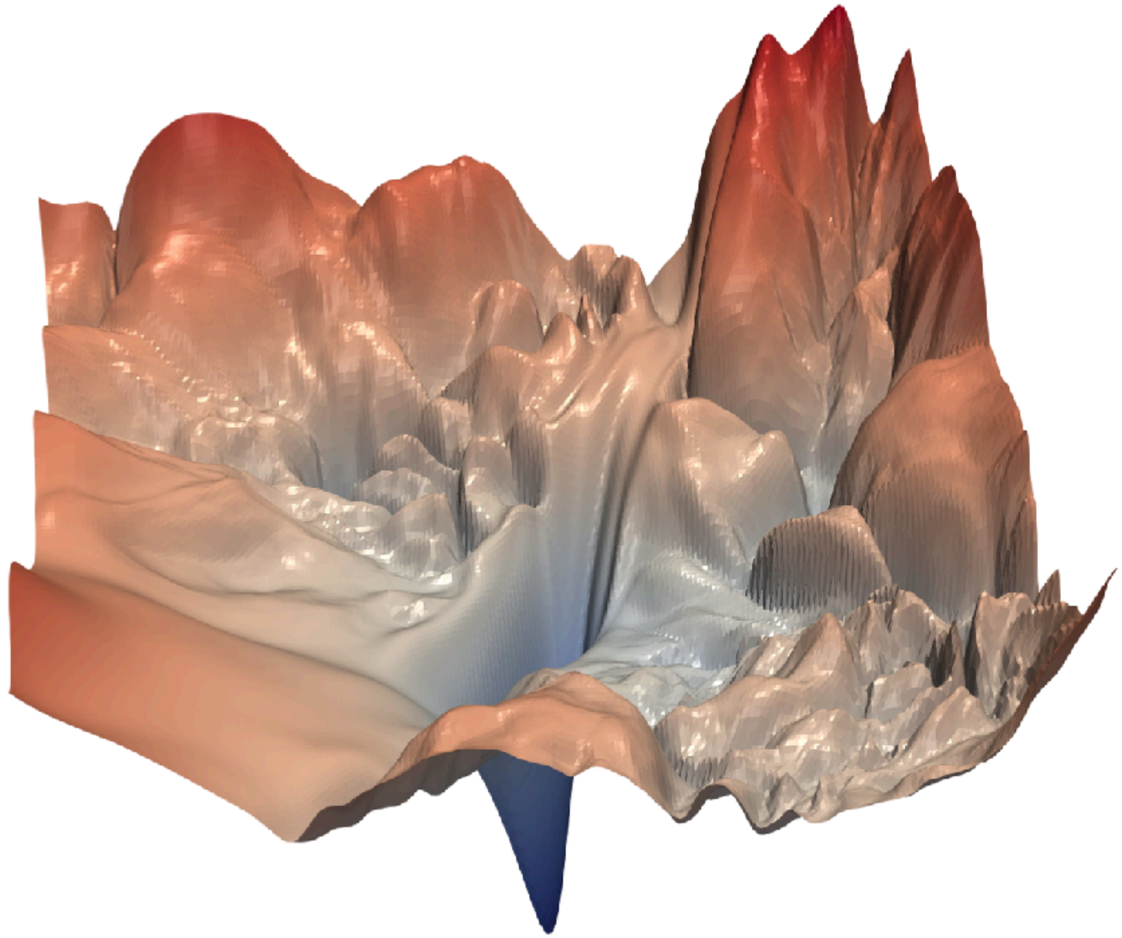


EXAMPLE LOSS LANDSCAPES IN MODERN ANNS

Convex problem



Highly non-convex problem



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)

WRAPPING UP

SUMMARY

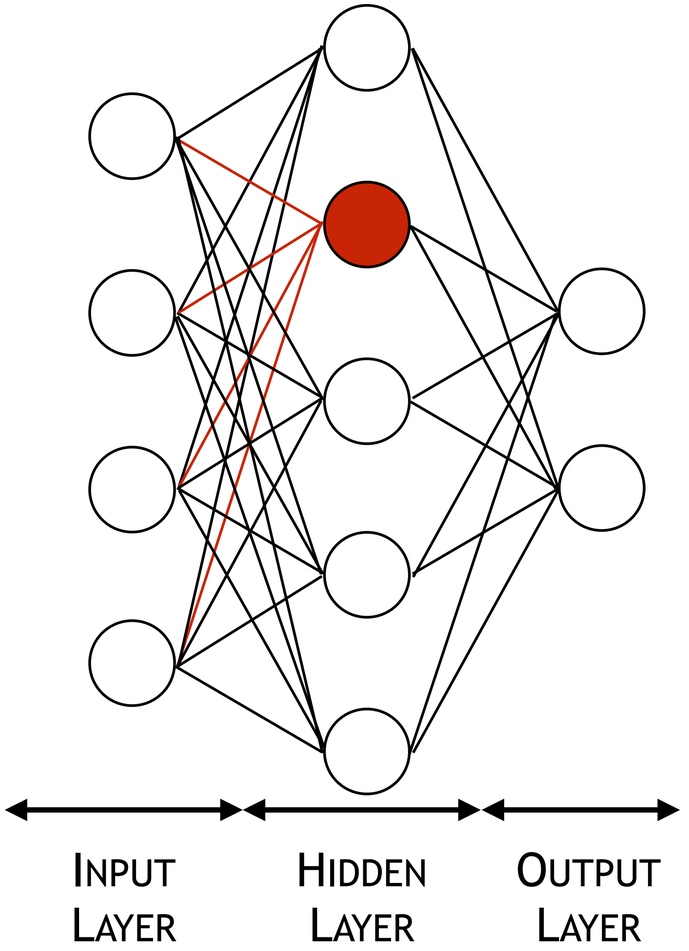
MLP: The prototypical neural network

ANNs are smart mappers

- Universal approximation capability
- Nonlinearities are crucial
- Manifold learning to beat the curse of dimensionality

NN training

- Forward pass: Matmuls + nonlinearities
- Backward pass: Partial derivatives of weight w.r.t. loss
- Iterative weight updates



EXERCISE 02

Online at the materials page

- Neural network from scratch: Forward pass, backward pass, weight update
- Either completable in Numpy or JAX
- Training and investigation of the LR

To be submitted via e-mail by:

- Wednesday 9:00

Discussion after the next lecture

https://hawaii.ziti.uni-heidelberg.de/teaching/ap_nn_from_scratch_materials_sose2026/