Introductory Deep Learning Lectures

2025:
4 November
25 November

2026:
20 January
10 February
17 March
21 April
19 May
16 June

Topics:

Neural Networks & Applications
Bayesian Deep Learning
Foundation Models
Self-supervised Learning

Presented by the IACDEEP Research Group

1h lectures
11 AM, IAC Aula

Instituto de Astrofísica de Canarias
C/ Vía Láctea, s/n 38205 La Laguna
Contact: iacdeeplectures@gmail.com

Intro to Neural Networks

Introductory Deep Learning lectures (IACDEEP)

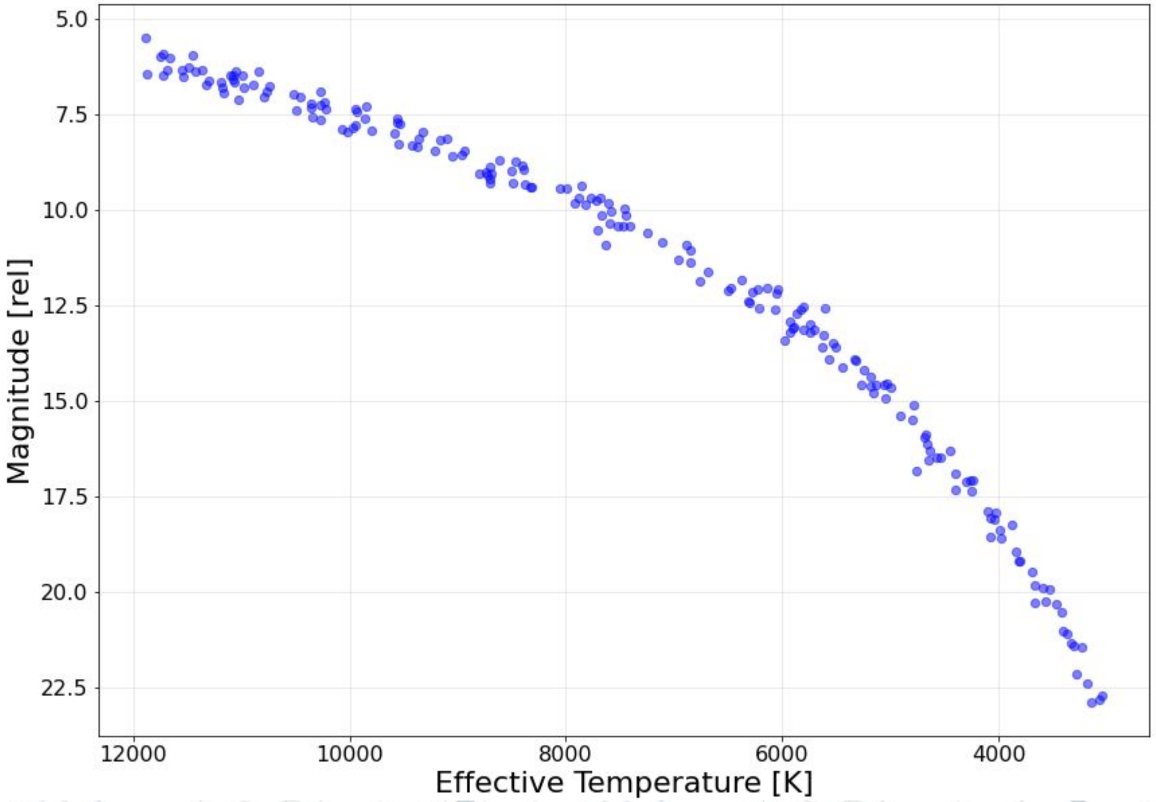
Carlos Westendorp & Marc Huertas

https://github.com/cwestend/IACDEEP_introNN

Course schedule

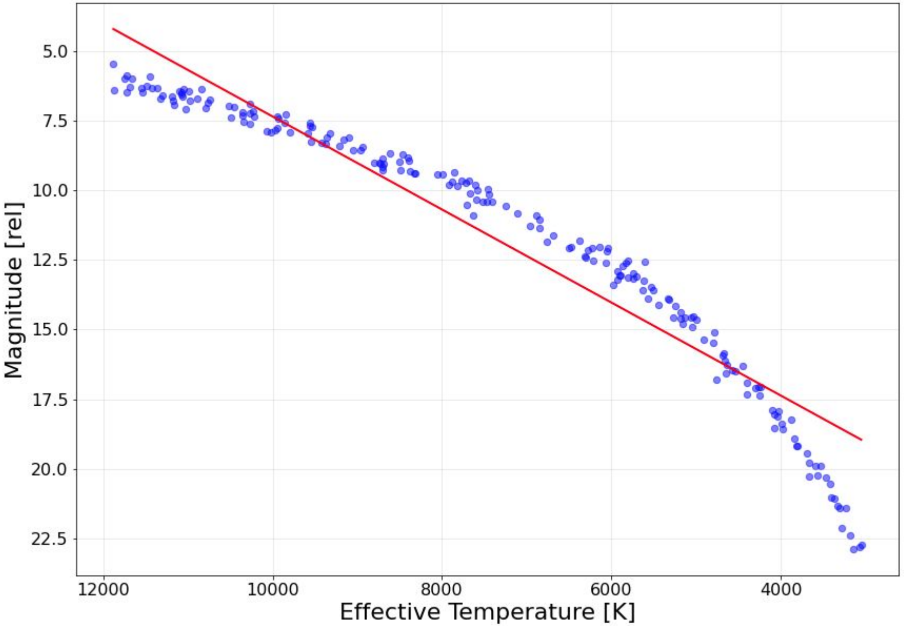
1. **Intro to NN (today)**
2. **Computer Vision using CNNs (25th November)**
3. Statistical deep learning
   a. Bayesian statistics
   b. Neural density estimators
   c. Simulation-based inference
4. NNs for sequences / time series
5. NNs for unstructured data: Graph NNs
6. Foundational Models / self supervised learning

General problem

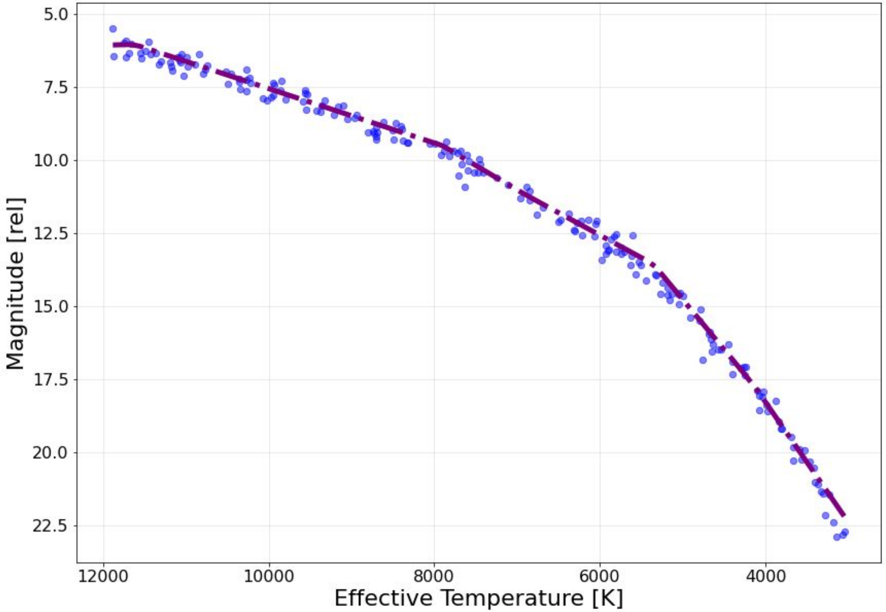


General problem

From physical insight



Data driven



Data driven

- No suitable **physical model** available: **accuracy**
- Physical Model **too complex** or **dataset too large**, minimisation difficulty: **speed**
- Possible **hidden information** in the data (beyond usual summary statistics): **discovery**

→ Astrophysics: **large** and **complex** datasets

Supervised learning

Given a dataset with **known labels** - find a function that can assign (predict) labels for an unlabelled dataset using a set of features (measurements)

Training Set

$$(\vec{x}_1, \vec{x}_2, \vec{x}_3, \ldots, \vec{x}_n)$$

$$(\vec{y}_1, \vec{y}_2, \vec{y}_3, \ldots, \vec{y}_n)$$

Features: colors, fluxes, spectral indexes (**Teff**)

Labels: morphology, object type, redshift (**magnitudes**)

Supervised learning

Given a dataset with **known labels** - find a function that can assign (predict) labels for an unlabelled dataset using a set of features (measurements)

<u>Training Set</u>

$$(\vec{x}_1, \vec{x}_2, \vec{x}_3, \ldots, \vec{x}_n)$$

$$(\vec{y}_1, \vec{y}_2, \vec{y}_3, \ldots, \vec{y}_n)$$

$$f_W(\vec{x}) = \vec{y}$$

Supervised learning

Unlabelled Set

$$(\vec{x_1}', \vec{x_2}', \vec{x_3}', \ldots, \vec{x_n}')$$

$$(\vec{y_1}', \vec{y_2}', \vec{y_3}', \ldots, \vec{y_n}')$$

Training Set

$$(\vec{x_1}, \vec{x_2}, \vec{x_3}, \ldots, \vec{x_n})$$

$$(\vec{y_1}, \vec{y_2}, \vec{y_3}, \ldots, \vec{y_n})$$

$$f_W(\vec{x}) = \vec{y}$$

?

Supervised learning

General Goal: Find a (**non-linear**) function that outputs the correct class / value (y) for a given input object:

$$f_W(\vec{x}) = \vec{y}$$

Features

Parameters (can be large!)

Minimization problem: find W such prediction error is minimal over <u>all unseen vectors</u>

Minimize the loss

1. Define a **Loss funtion** $loss(f_W(), \vec{x}_i, \vec{y}_i)$

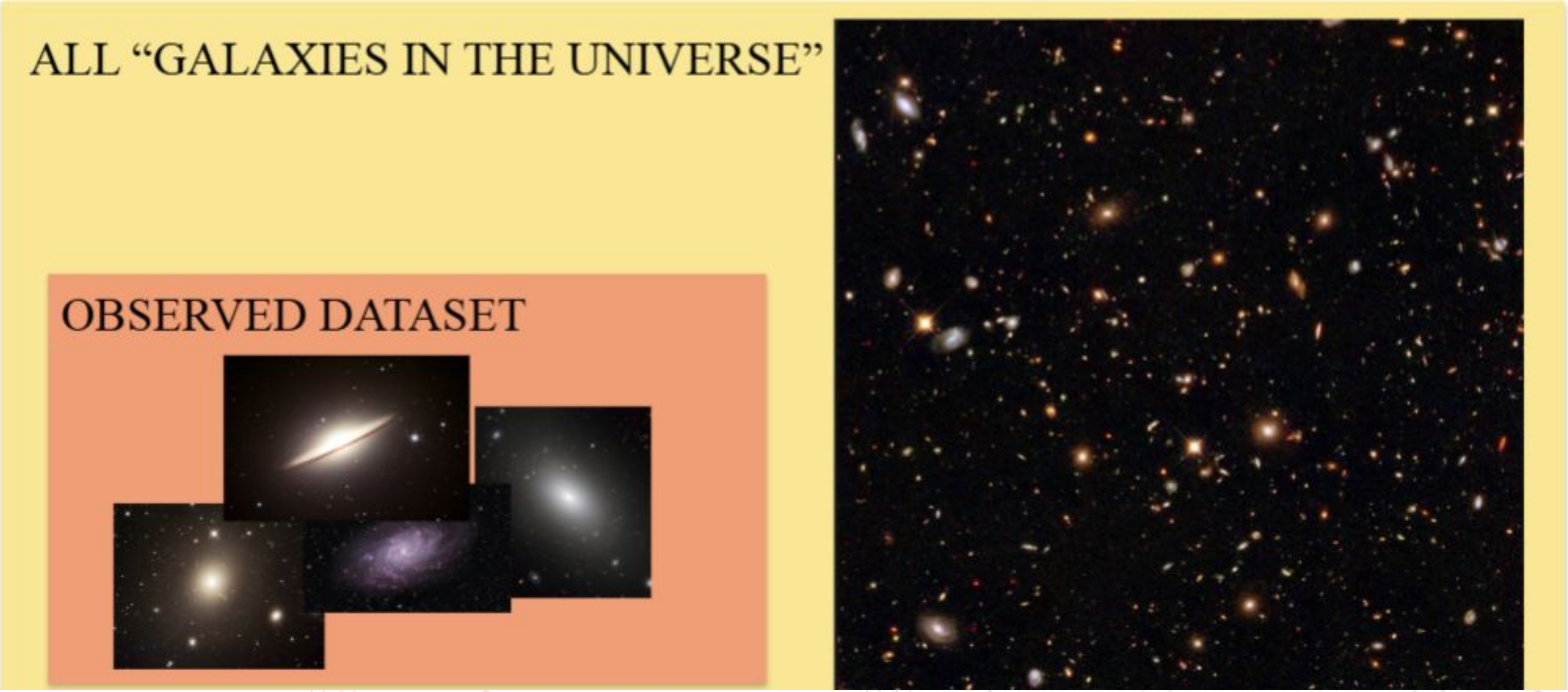
   - for example: **MSE** loss $(f_W(\vec{x}_i) - \vec{y}_i)^2$

2. **Minimize** the **empirical risk** with optimization

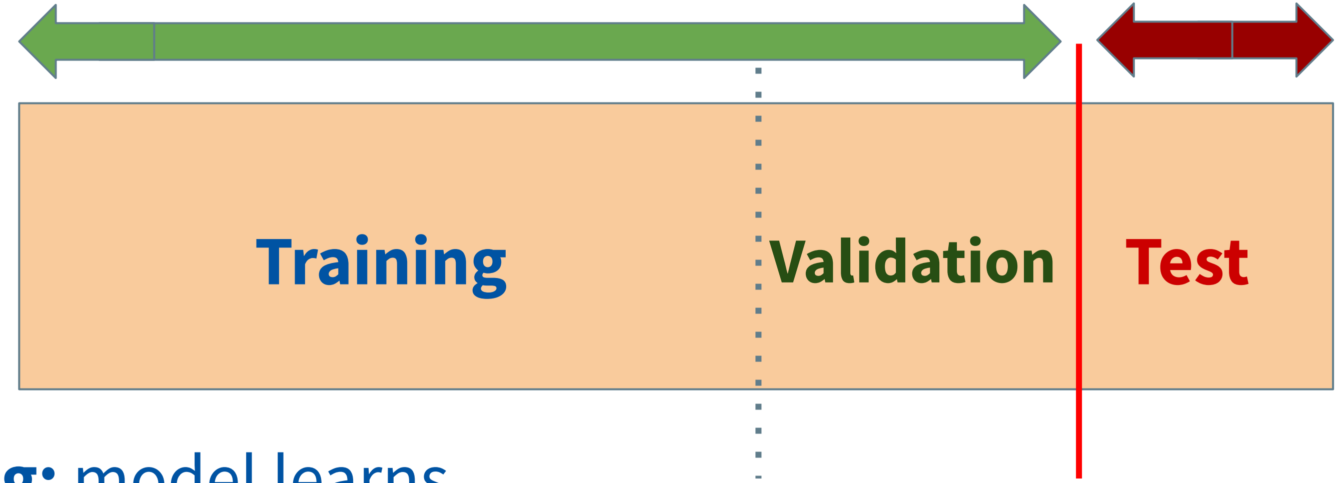
$$R_{empirical}(W) = \frac{1}{N}\sum_i^N loss(f_W(), \vec{x}_i, \vec{y}_i)$$

Minimize the loss

$$R_{empirical}(W) = \frac{1}{N}\sum_{i}^{N} loss(f_W(), \vec{x}_i, \vec{y}_i)$$



In practice: **split data** (need enough!)



Training: model learns

Validation: monitor learning (overfitting)

Test: validity check (not used in training!)

Minimisation problem

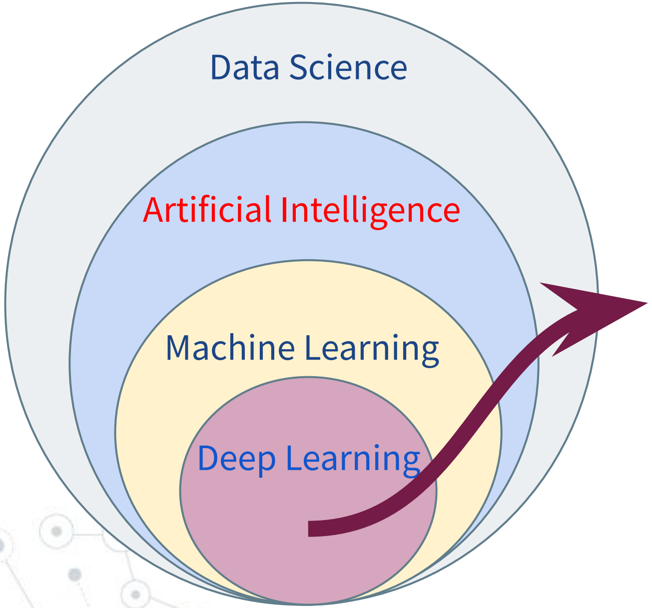
We need:

1. A **Loss function** (something to minimize)
2. Minimization (optimization) **algorithm**

common to **all** Machine Learning algorithms

Machine Learning and Deep Learning

Deep Learning *within* Machine Learning

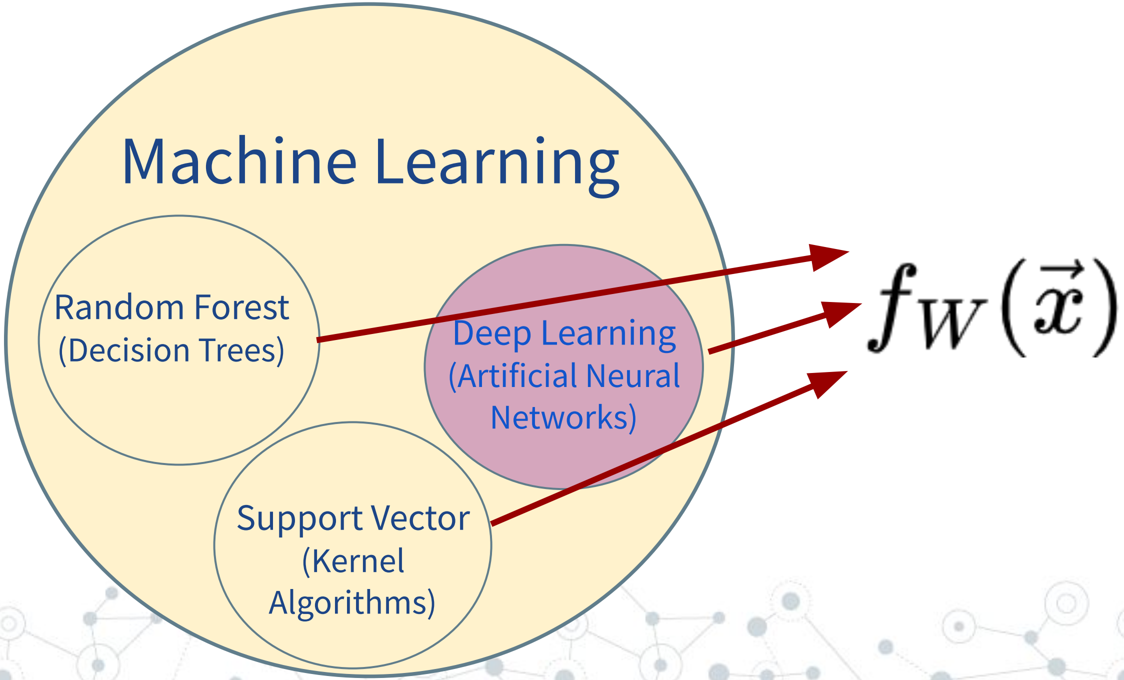


Not only LLMs (ChatGPT, Gemini…)

- Healthcare (cancer detection)
- Autonomous vehicles
- Climate (forecasting, monitor)
- Speech & Audio (translating)
- Finance (fraud detection)
- …

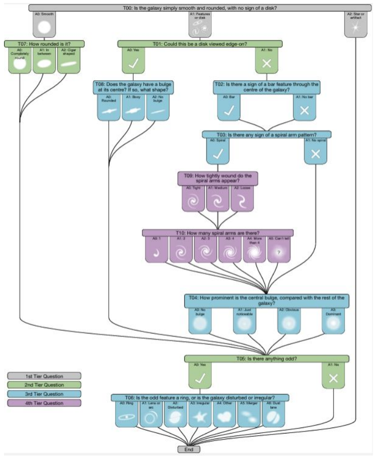
Machine Learning and Deep Learning

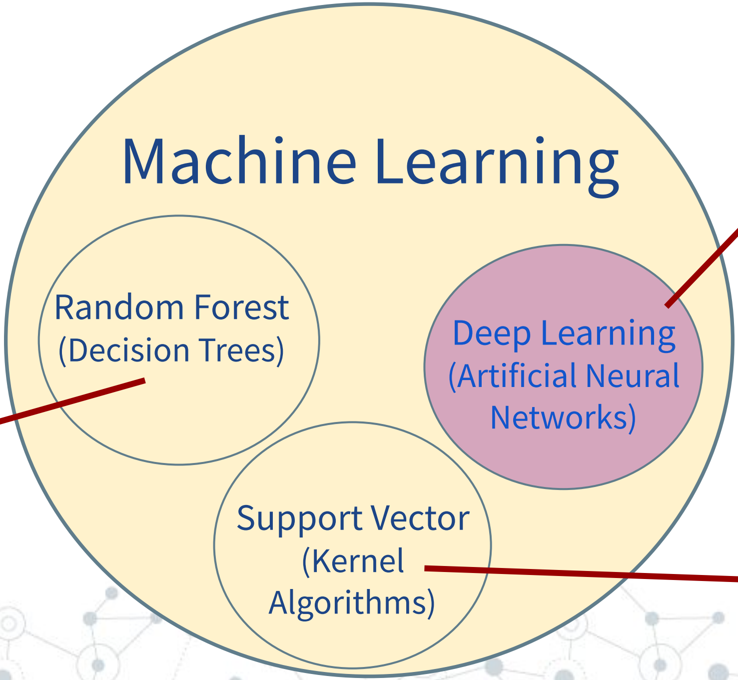
Difference is the function used (sets optimization/loss)

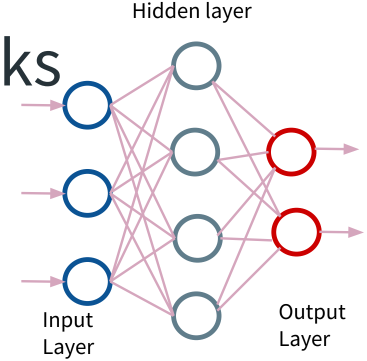


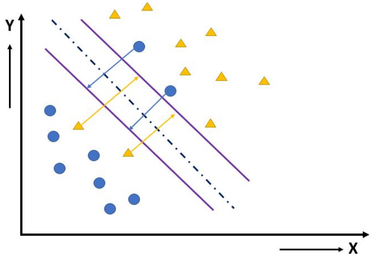
Machine Learning and Deep Learning

Deep Learning uses Artificial Neural Networks









¿Why Deep Learning?

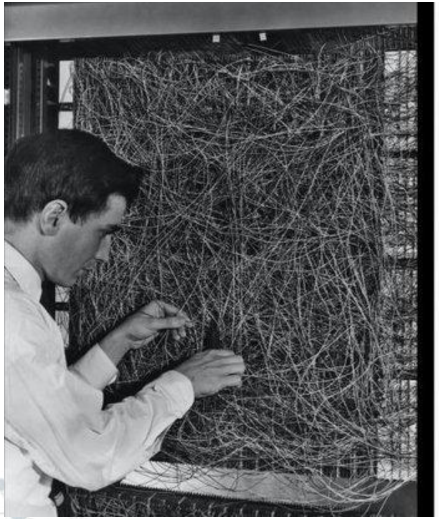
Deep Learning uses **Artificial Neural Networks**

Neural Network origins

- Automata = "that operates by itself" ancient concept (China, Islam, Greece - 300 BC)
- 1950: **Alan Turing** published "Computing Machinery and Intelligence" - Turing Test (called Imitation Game)
- 1956: **John McCarthy** workshop in Dartmouth about "**Artificial Intelligence**"

Neural Network origins

- 1958 **Frank Rosenblatt** (psychologist) proposes the classic **perceptron** (improving on 1943 McCulloch & Pitts neural model)



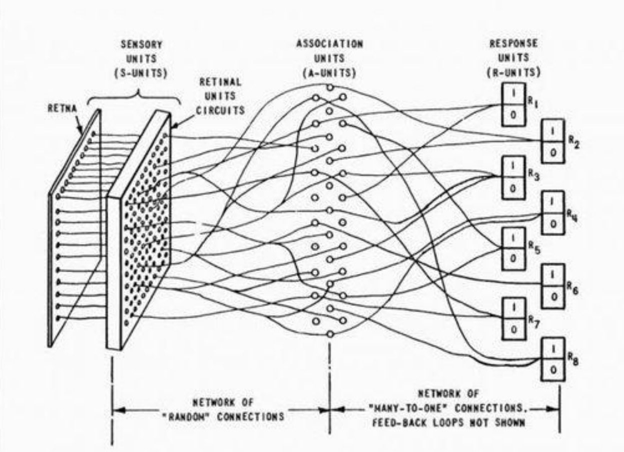


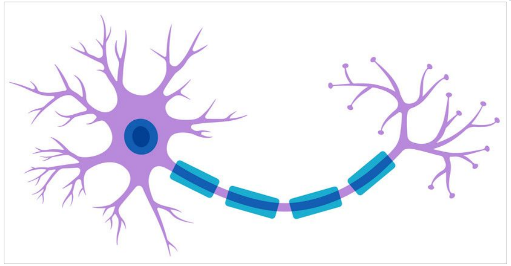
Figure I ORGANIZATION OF THE MARK I PERCEPTRON

Neural Network origins

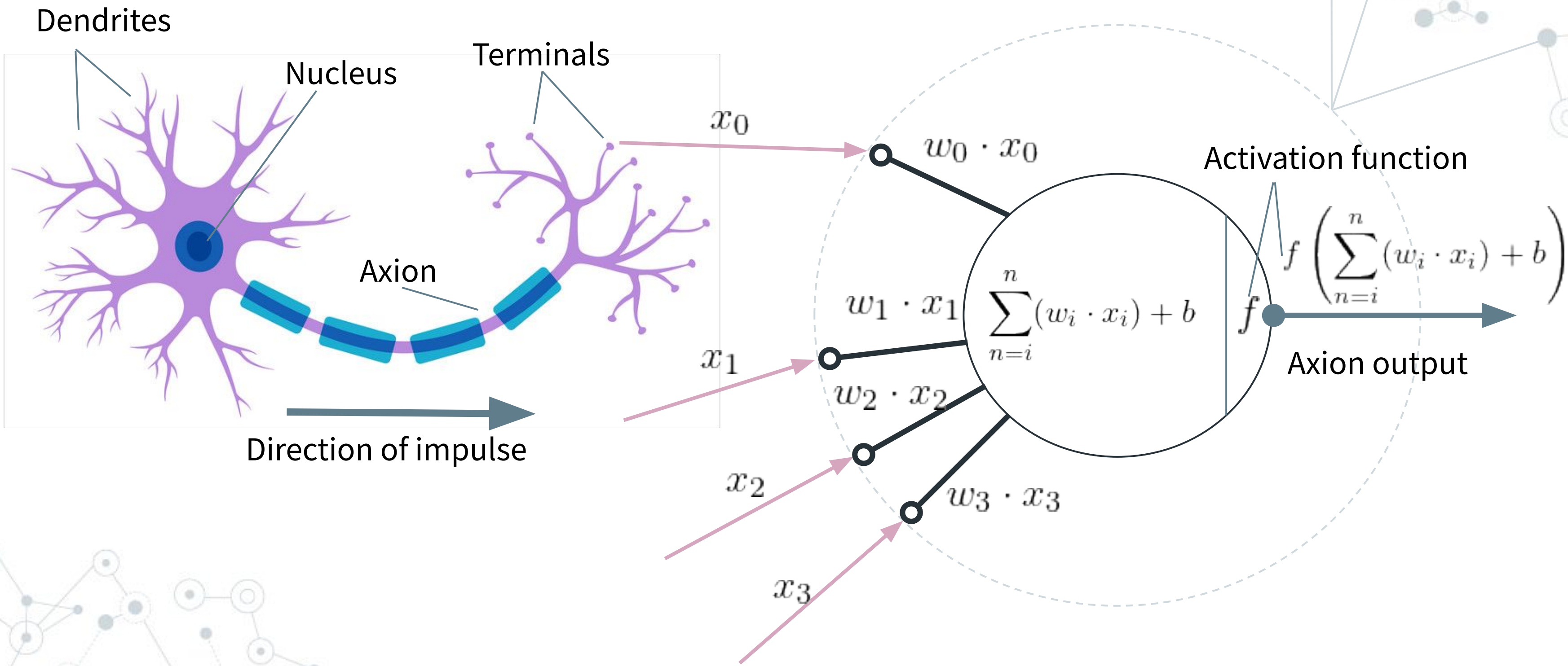
- 1954 **Software perceptron:** **IBM 704** 1st mass produced floating point computer (Fortran, LISP…)

Perceptron: a model neuron

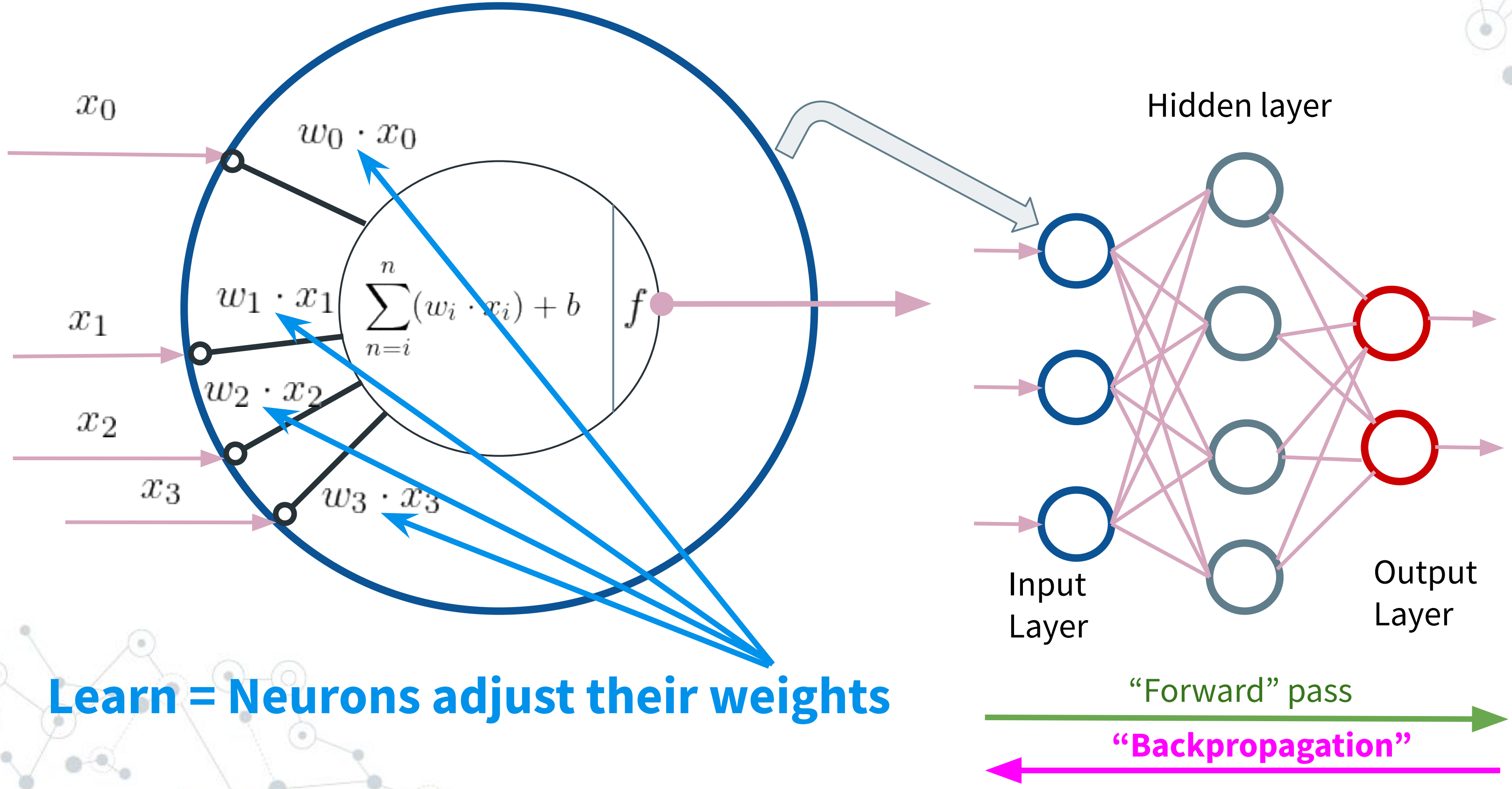
Santiago Ramón y Cajal (1889): Neurons are cells = **individual units** communicate by synapsis



Perceptron: a model neuron



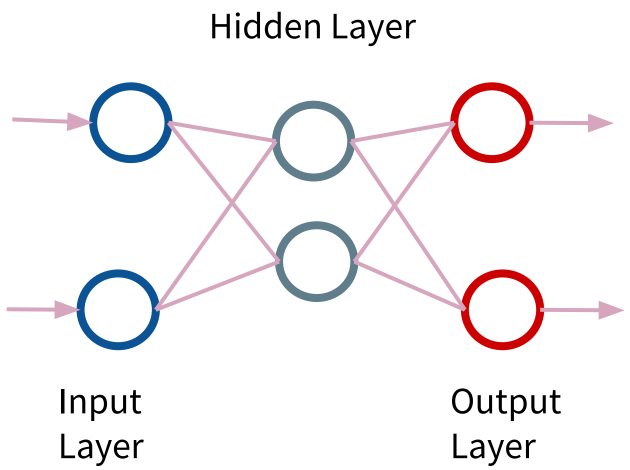
Artificial Neural Network: Multi-layer perceptron



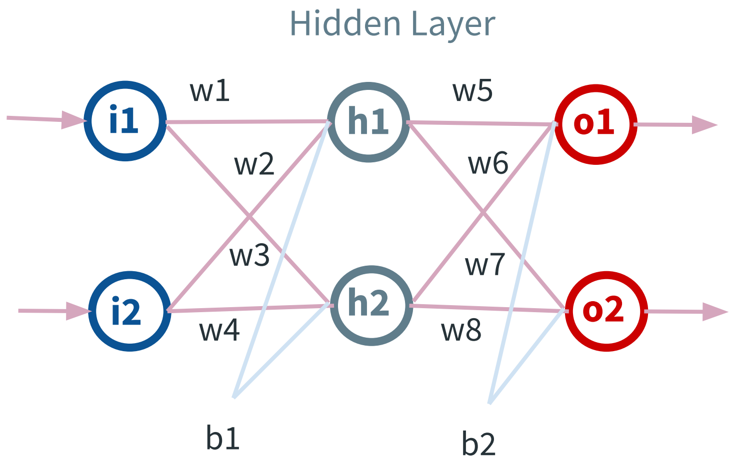
Learn = Neurons adjust their weights

Activation function

- Without a **non-linear activation function (f)** the neural network can only account for linear effects



Activation function (linear)



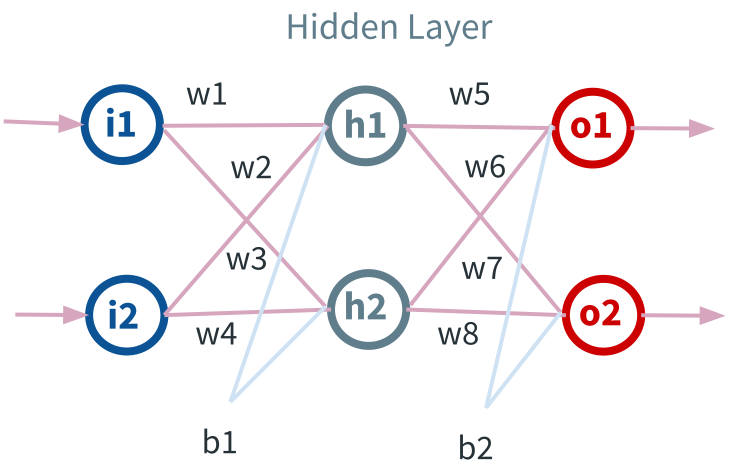
$$h1 = w1 \cdot i1 + w2 \cdot i2 + b1$$

$$h2 = w3 \cdot i1 + w4 \cdot i2 + b1$$

$$o1 = w5 \cdot h1 + w6 \cdot h2 + b2$$

$$o2 = w7 \cdot h1 + w8 \cdot h2 + b2$$

Activation function (linear)



$$o1 = w5 \cdot (w1 \cdot i1 + w2 \cdot i2 + b1) + w6 \cdot (w3 \cdot i1 + w4 \cdot i2 + b1) + b2$$

$$o1 = w5 \cdot w1 \cdot i1 + w5 \cdot w2 \cdot i2 + w5 \cdot b1 + w6 \cdot w3 \cdot i1 + w6 \cdot w4 \cdot i2 + w6 \cdot b1 + b2$$

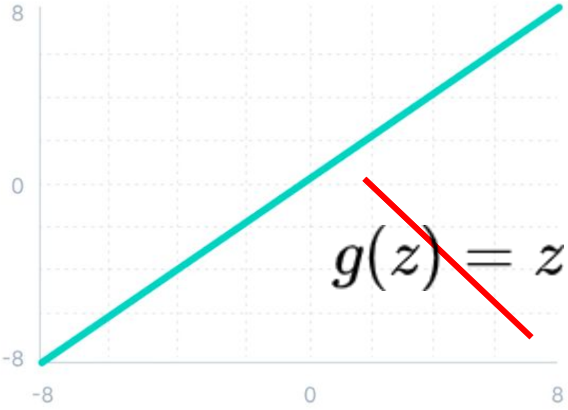
$$o1 = (w5 \cdot w1 + w6 \cdot w3) \cdot i1 + (w5 \cdot w2 + w6 \cdot w4) \cdot i2 + (w5 \cdot b1 + w6 \cdot b1 + b2)$$

$$o1 = A1 \cdot i1 + A2 \cdot i2 + C1$$

Linear again!

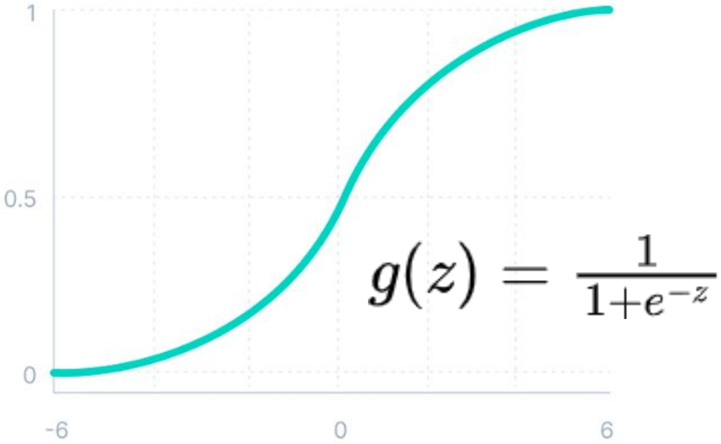
Activation Functions

Linear (no!)



$$\cancel{g(z) = z}$$

Sigmoid / Softmax*

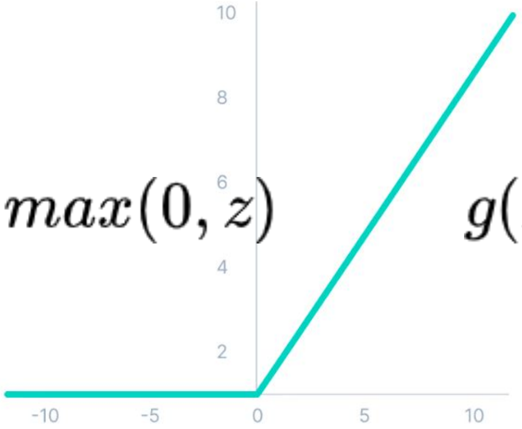


$$g(z) = \frac{1}{1+e^{-z}}$$

$$* \; g(z_i) = \frac{e^{z_i}}{\sum e^{z_j}}$$

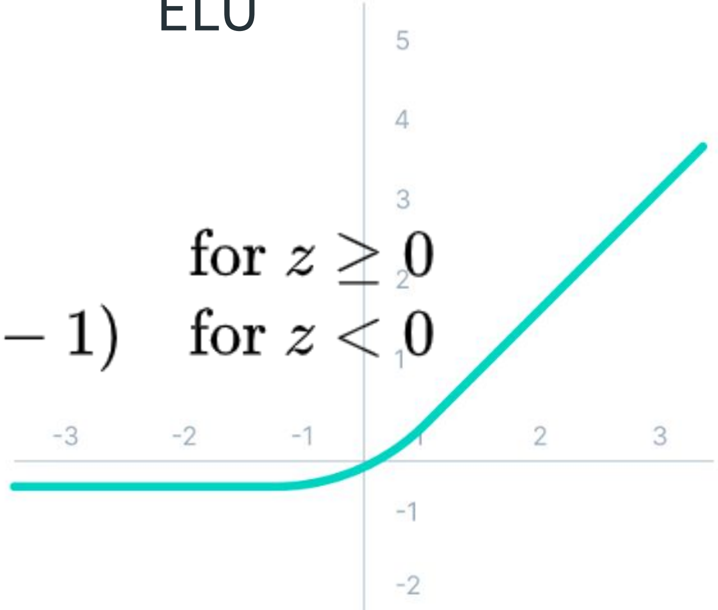
Activation Functions

ReLU



$$g(z) = max(0, z)$$

ELU



$$g(z) = \begin{cases} z & \text{for } z \geq 0 \\ \alpha(e^z - 1) & \text{for } z < 0 \end{cases}$$

Universal Approximation Theorem

Why we **can** use a NN:

"For any **continuous function** for a hypercube [0,1]d to real numbers, and every positive epsilon, there exists a **sigmoid** based 1-**HIDDEN LAYER NEURAL NETWORK** that obtains at most epsilon error in functional space" Cybenko '89

➡ Big enough **NN** can *approximate* (not represent) any smooth function

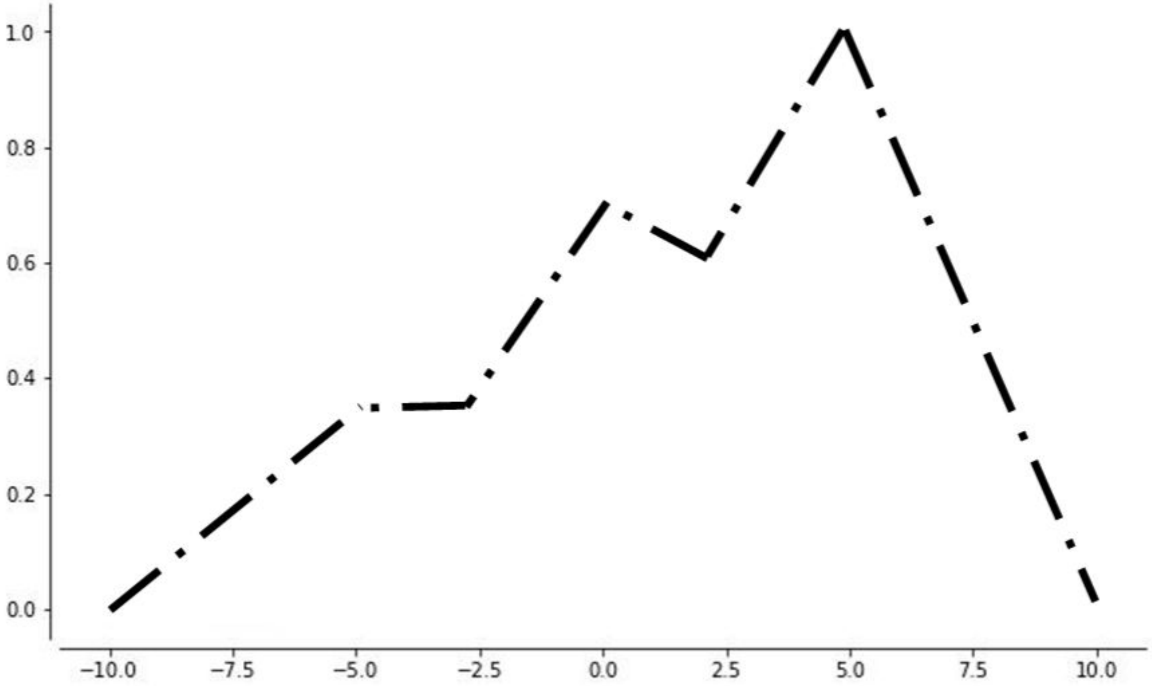
Universal Approximation Theorem

Why we **can** use a NN:

"For any **continuous function** for a hypercube [0,1]d to real numbers, **non-constant, bounded and continuous activation function f**, and every positive epsilon, there exists 1**-HIDDEN LAYER NEURAL NETWORK** using f that obtains at most epsilon error in functional space"
Horvik '91

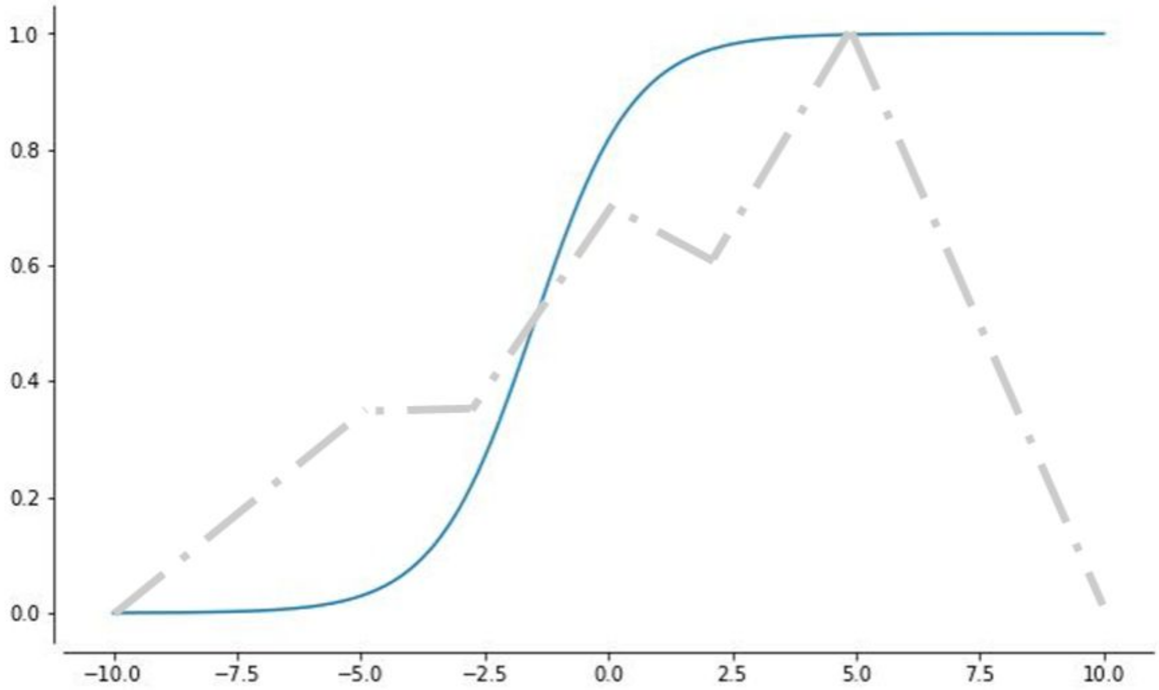
➡ Big enough **NN** can *approximate* (not represent) any smooth function

Universal Approximation Theorem (Intuition)



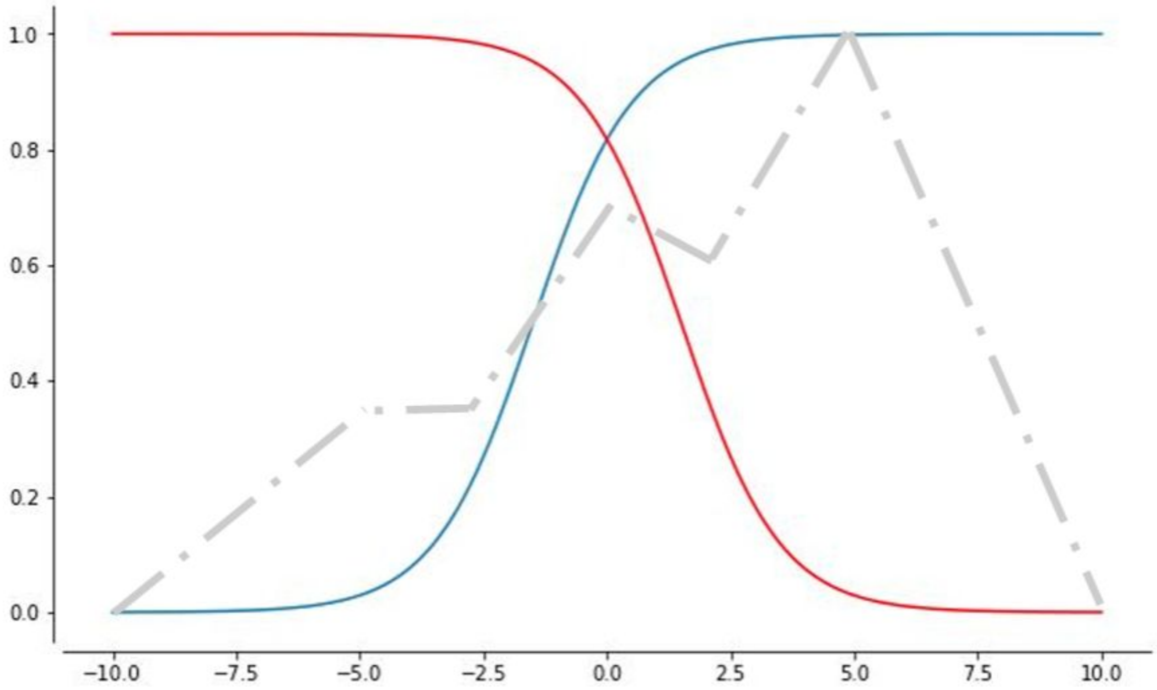
deepmind
@Czarnecki

Universal Approximation Theorem (Intuition)



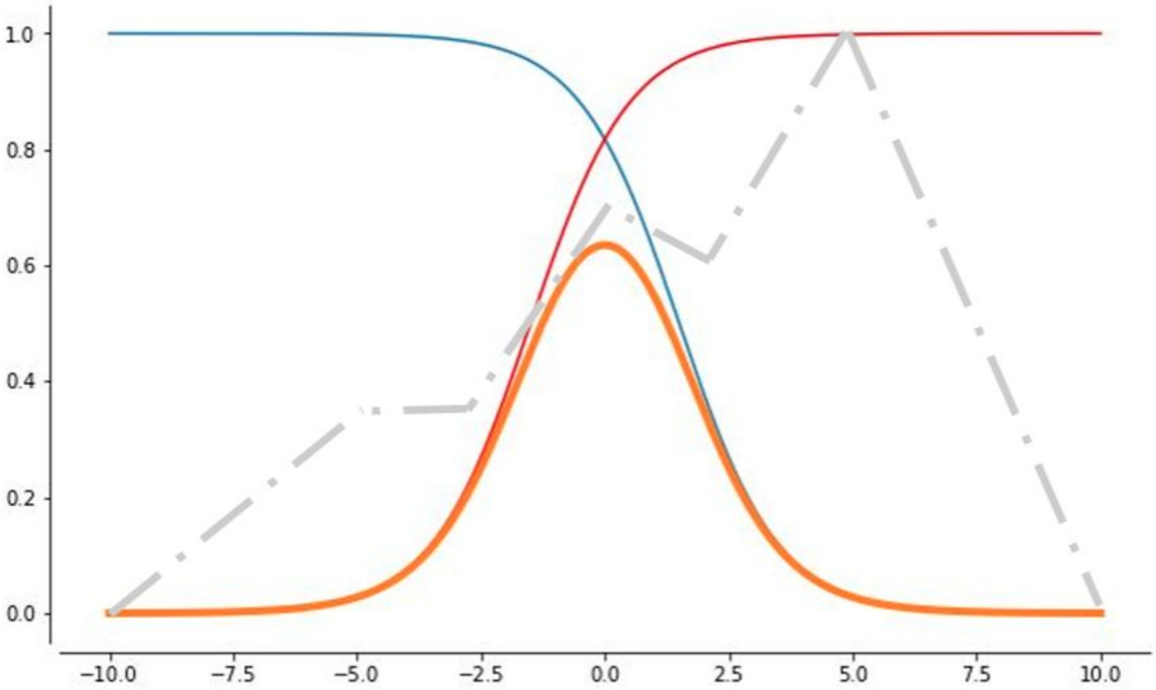
deepmind
@Czarnecki

Universal Approximation Theorem (Intuition)



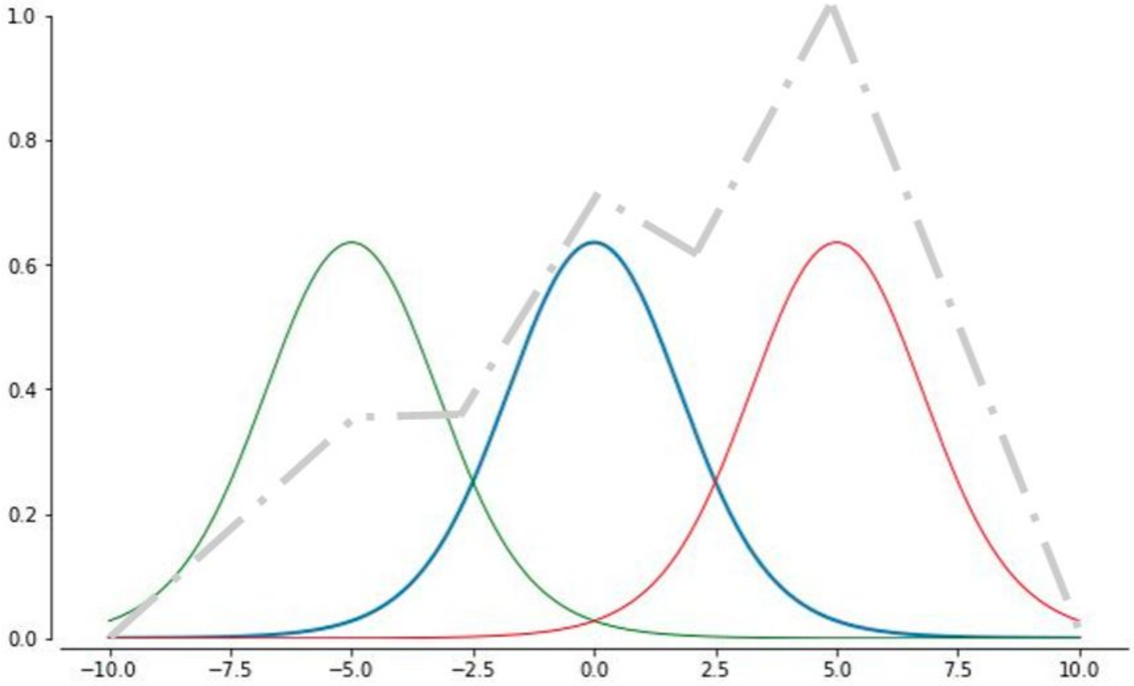
deepmind
@Czarnecki

Universal Approximation Theorem (Intuition)



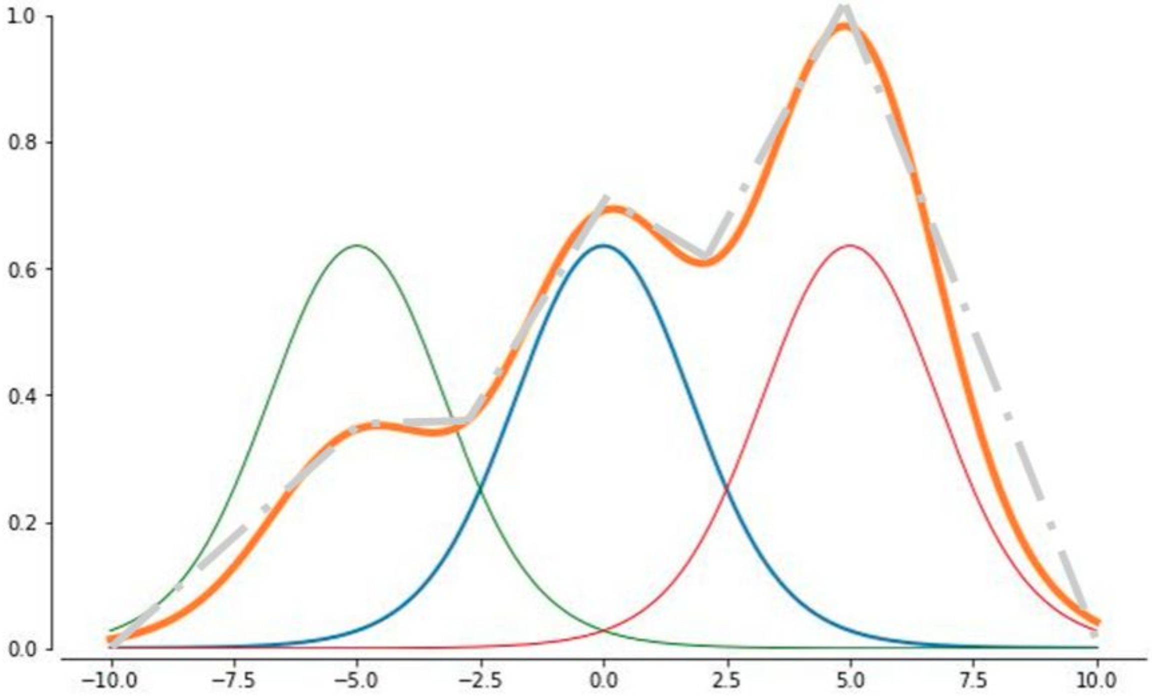
deepmind
@Czarnecki

Universal Approximation Theorem (Intuition)



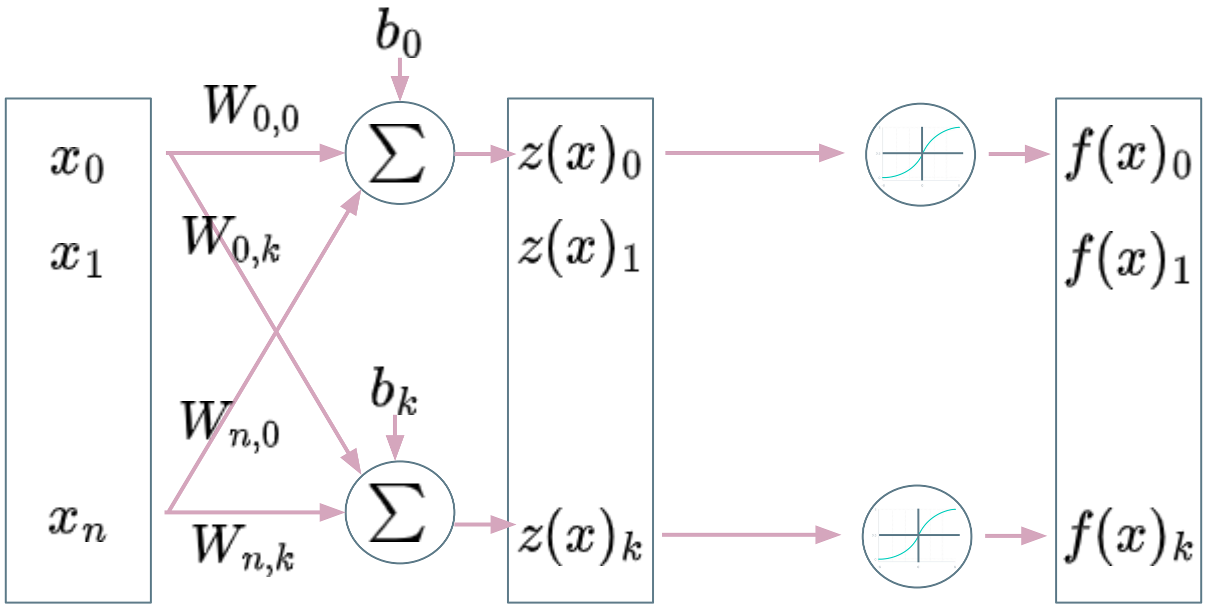
deepmind
@Czarnecki

Universal Approximation Theorem (Intuition)



deepmind
@Czarnecki

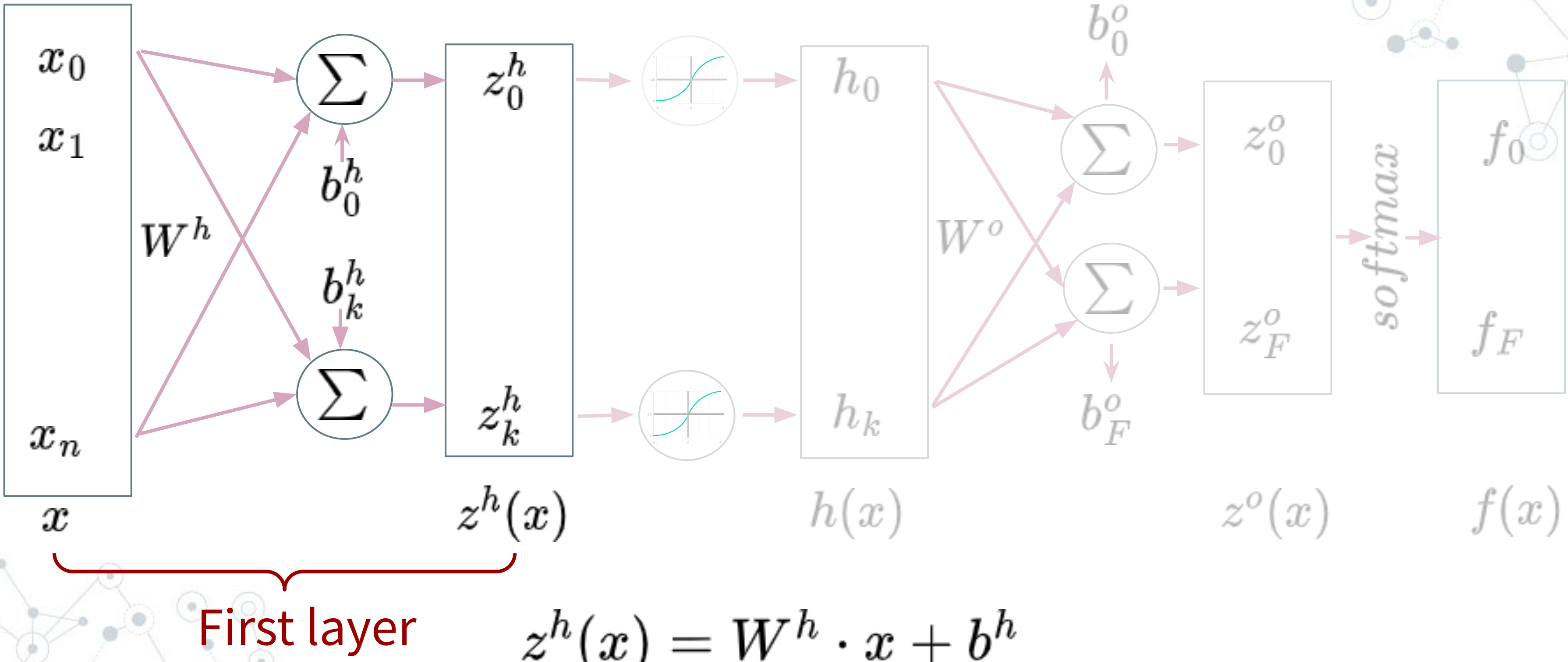
Artificial Neural Network: Multi-layer perceptron



$$f(\vec{x}) = g(W \cdot \vec{x} + \vec{b})$$

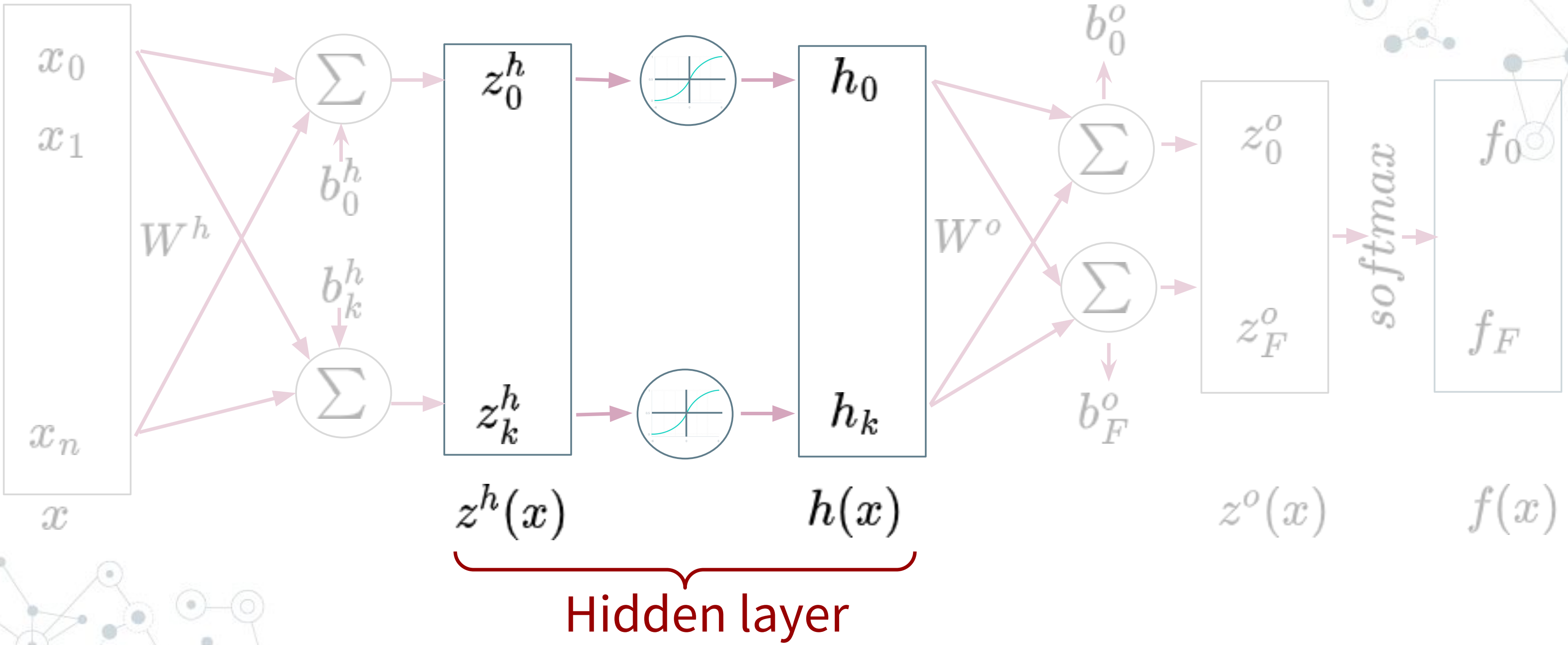
W is an array,
b is a vector

DEEP Learning: many hidden layers



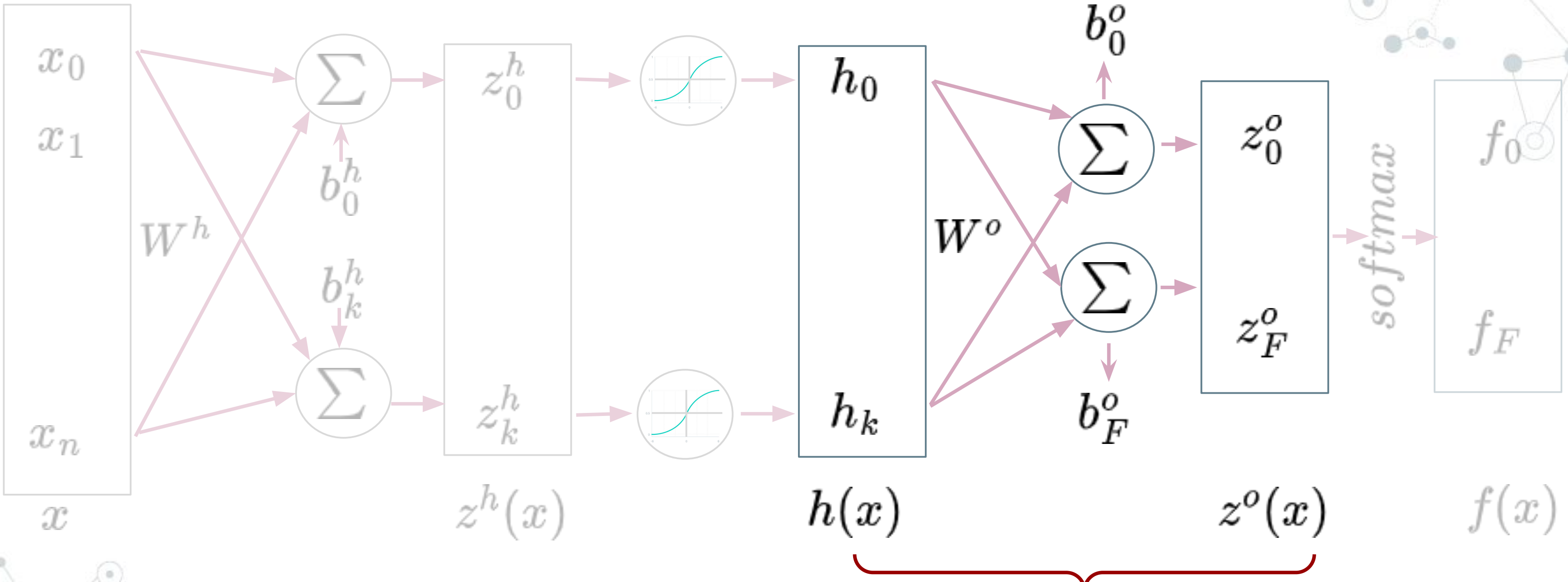
$$z^h(x) = W^h \cdot x + b^h$$

DEEP Learning: many hidden layers



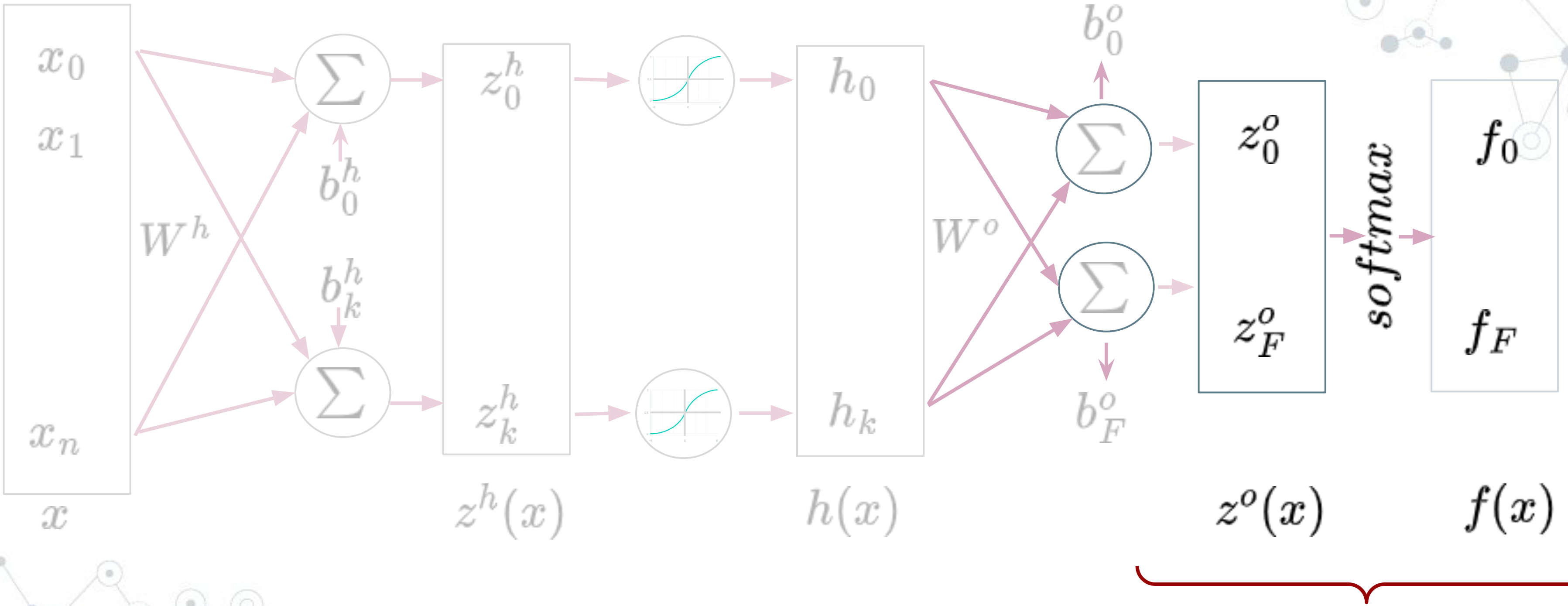
$$h(x) = g(z^h(x)) = g(W^h \cdot x + b^h)$$

DEEP Learning: many hidden layers



$$z^o(x) = W^o \cdot x + b^o$$

DEEP Learning: **many hidden layers**



Prediction layer

Minimize the **loss**

- Find the weights that generate **minimum loss** (an arbitrary multi-dimensional function!)

Simplify: Taylor approximation and use standard algorithms…

Minimize the **loss**

(1st derivative) **Gradient Descent**

$$W_{t+1} = W_t - \lambda_t \nabla f(W_t)$$

Newton (2nd derivative)

$$W_{t+1} = W_t - [Hf(W_t)]^{-1} \nabla f(W_t)$$

hessian

$$H = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1\,\partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1\,\partial x_n} \\ \frac{\partial^2 f}{\partial x_2\,\partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2\,\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n\,\partial x_1} & \frac{\partial^2 f}{\partial x_n\,\partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}.$$

Minimize the **loss**

(1st derivative) **Gradient Descent**

$$W_{t+1} = W_t - \lambda_t \nabla f(W_t)$$

learning rate gradient

Newton is *fast* BUT expensive

- And not always works (smooth functions)

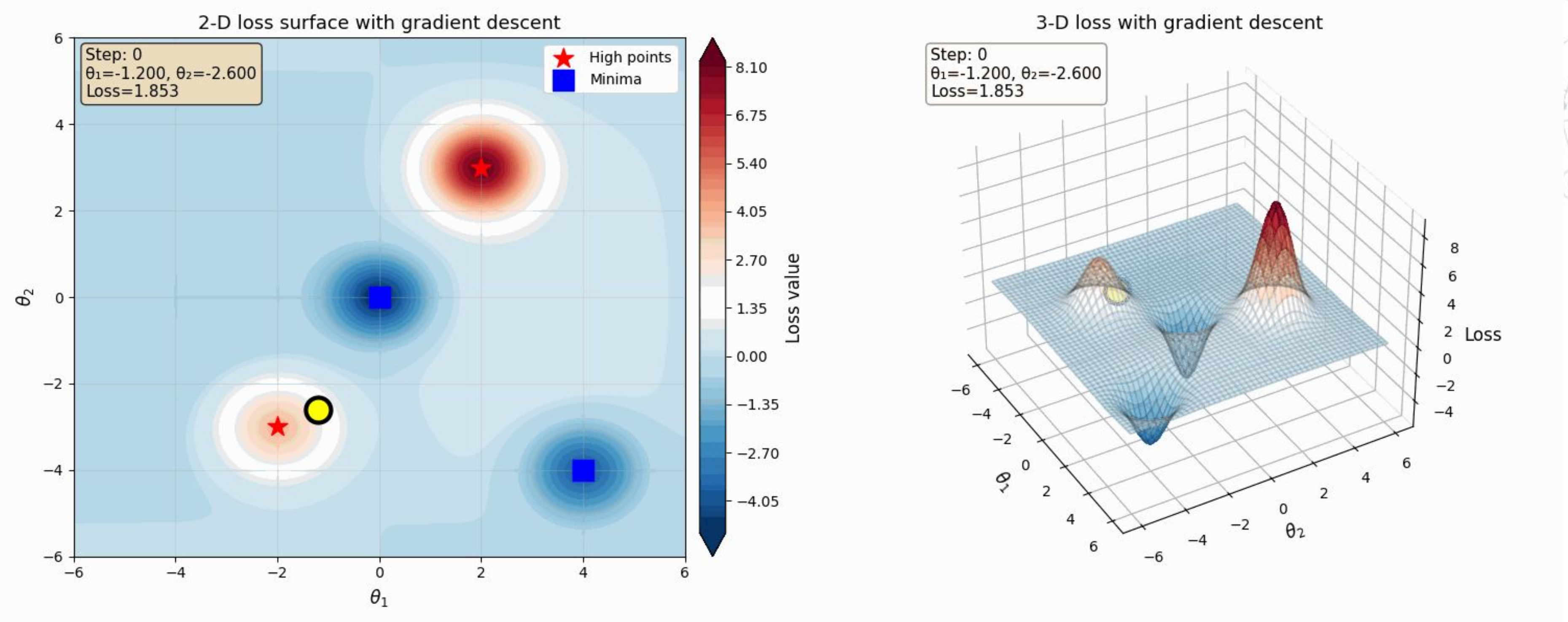
Newton (2nd derivative)

$$W_{t+1} = W_t - [Hf(W_t)]^{-1} \nabla f(W_t)$$

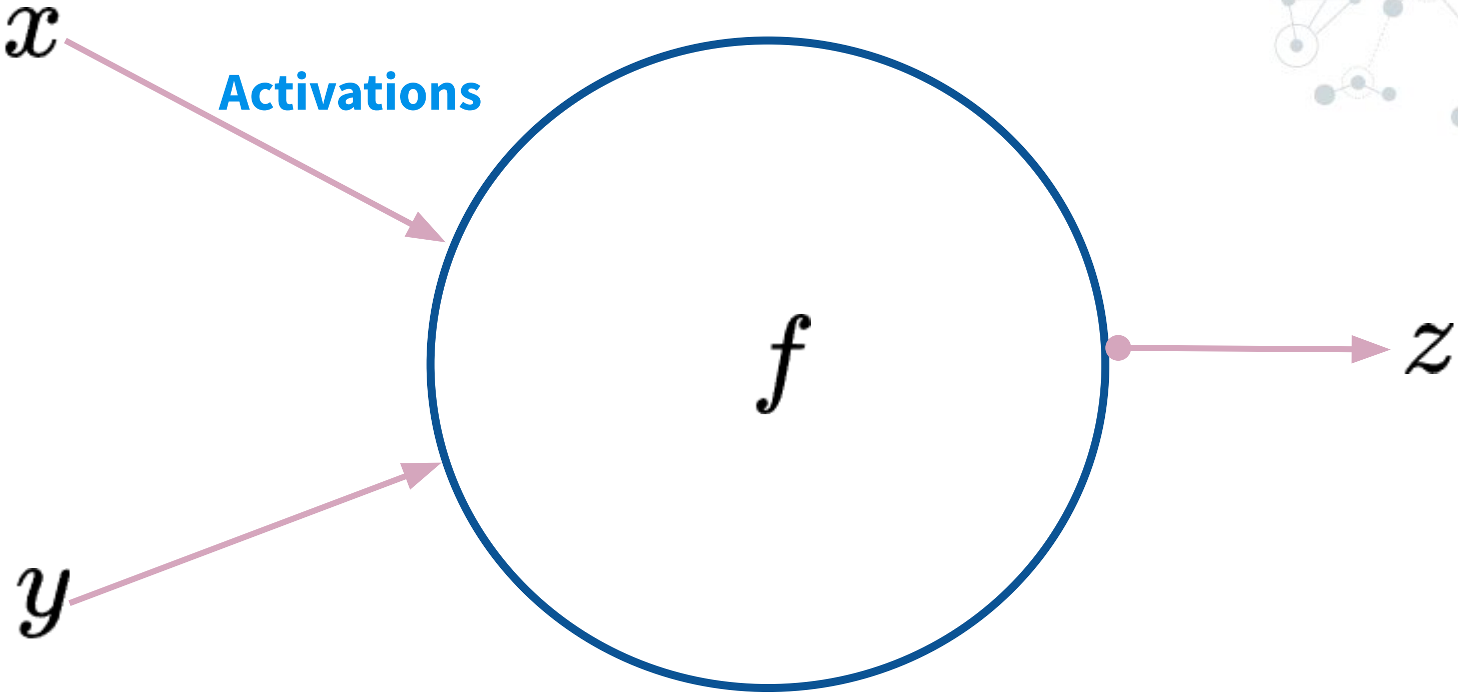
hessian

$$H = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1\,\partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1\,\partial x_n} \\ \frac{\partial^2 f}{\partial x_2\,\partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2\,\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n\,\partial x_1} & \frac{\partial^2 f}{\partial x_n\,\partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}.$$

Gradient Descent

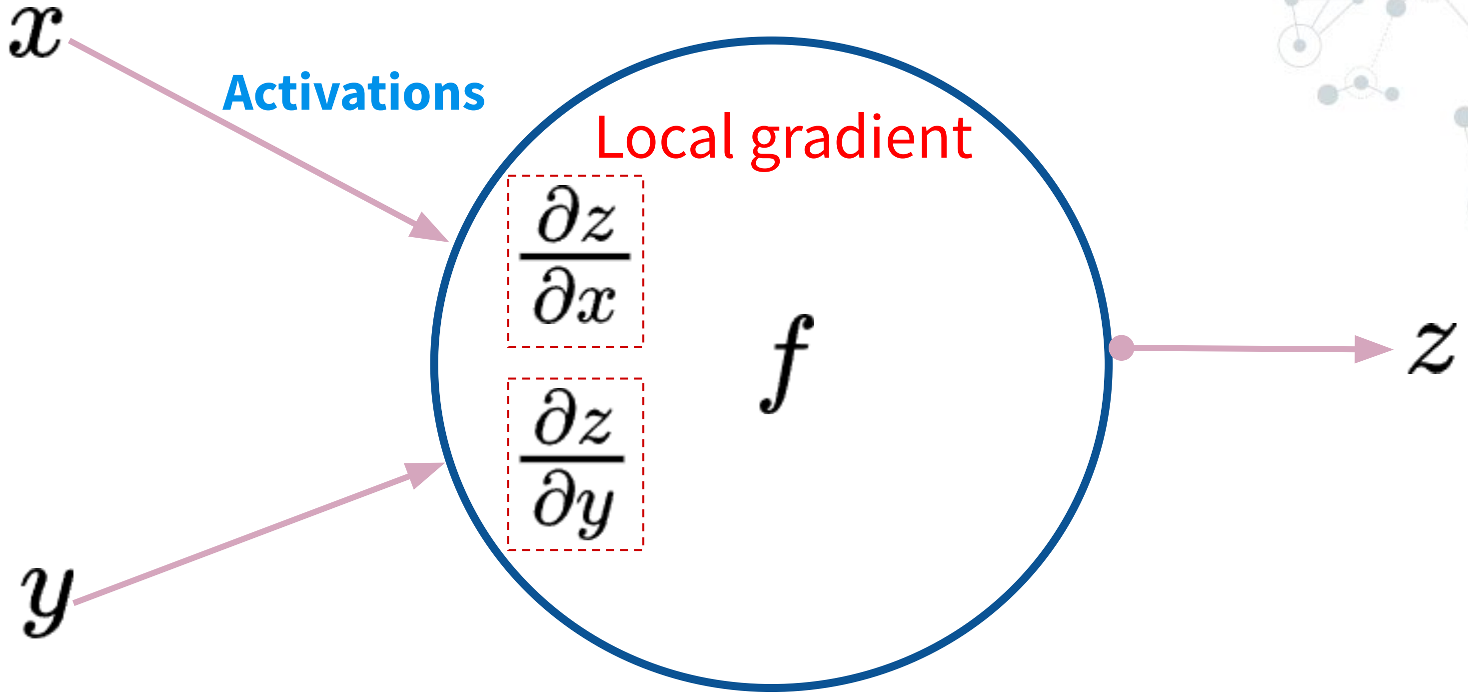


Backpropagation (neuron level)



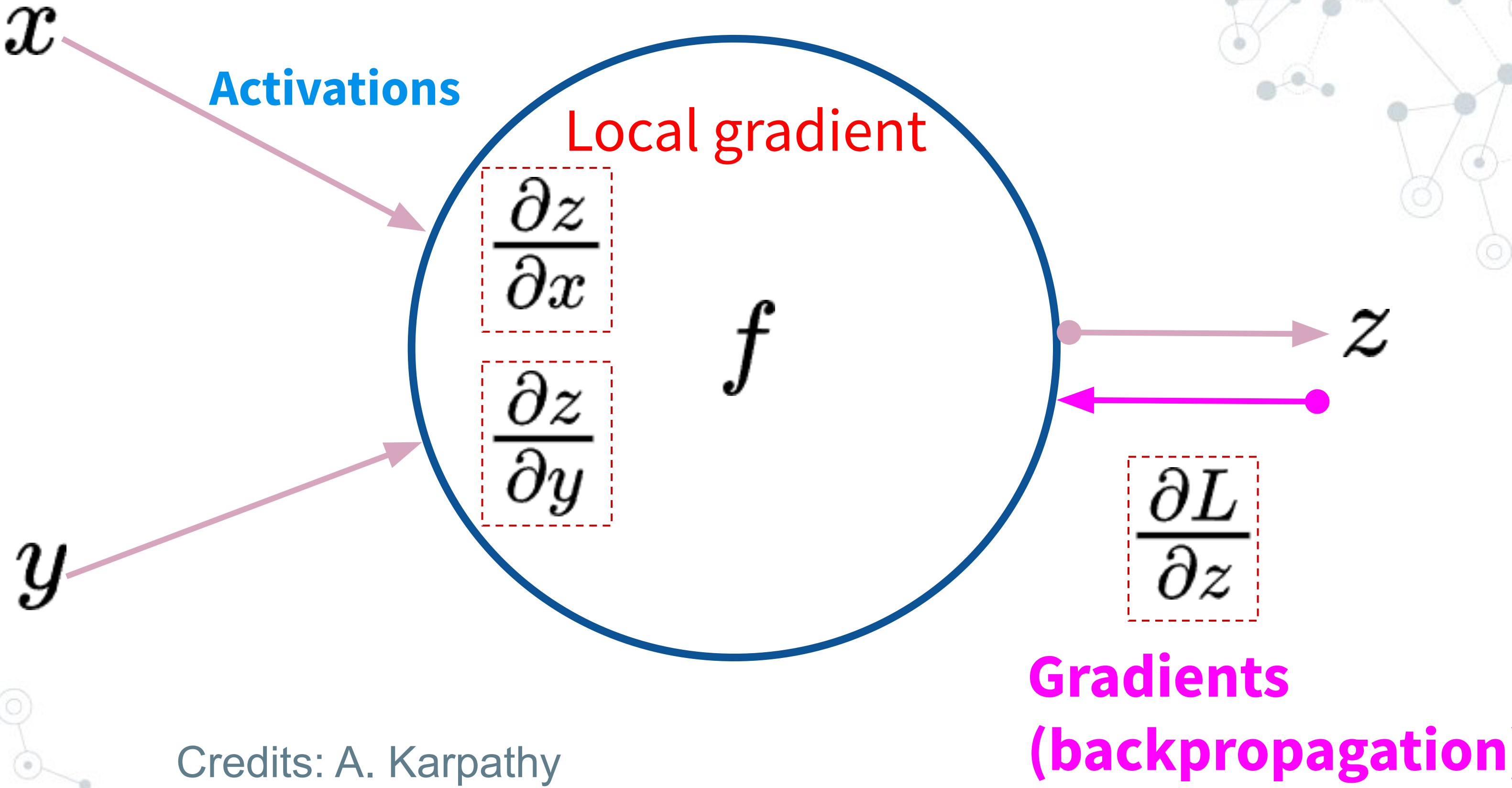
Credits: A. Karpathy

Backpropagation (neuron level)



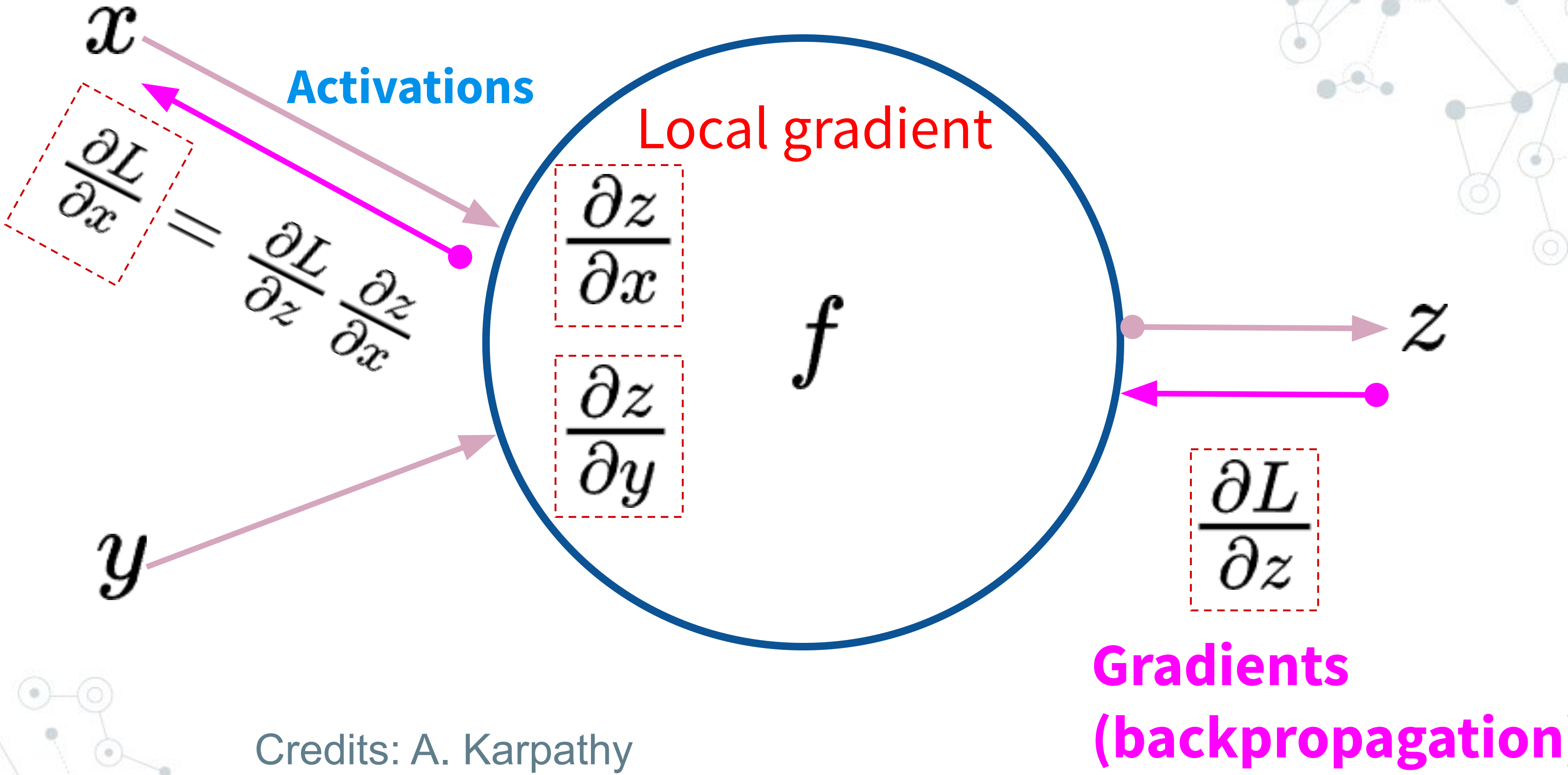
Credits: A. Karpathy

Backpropagation (neuron level)



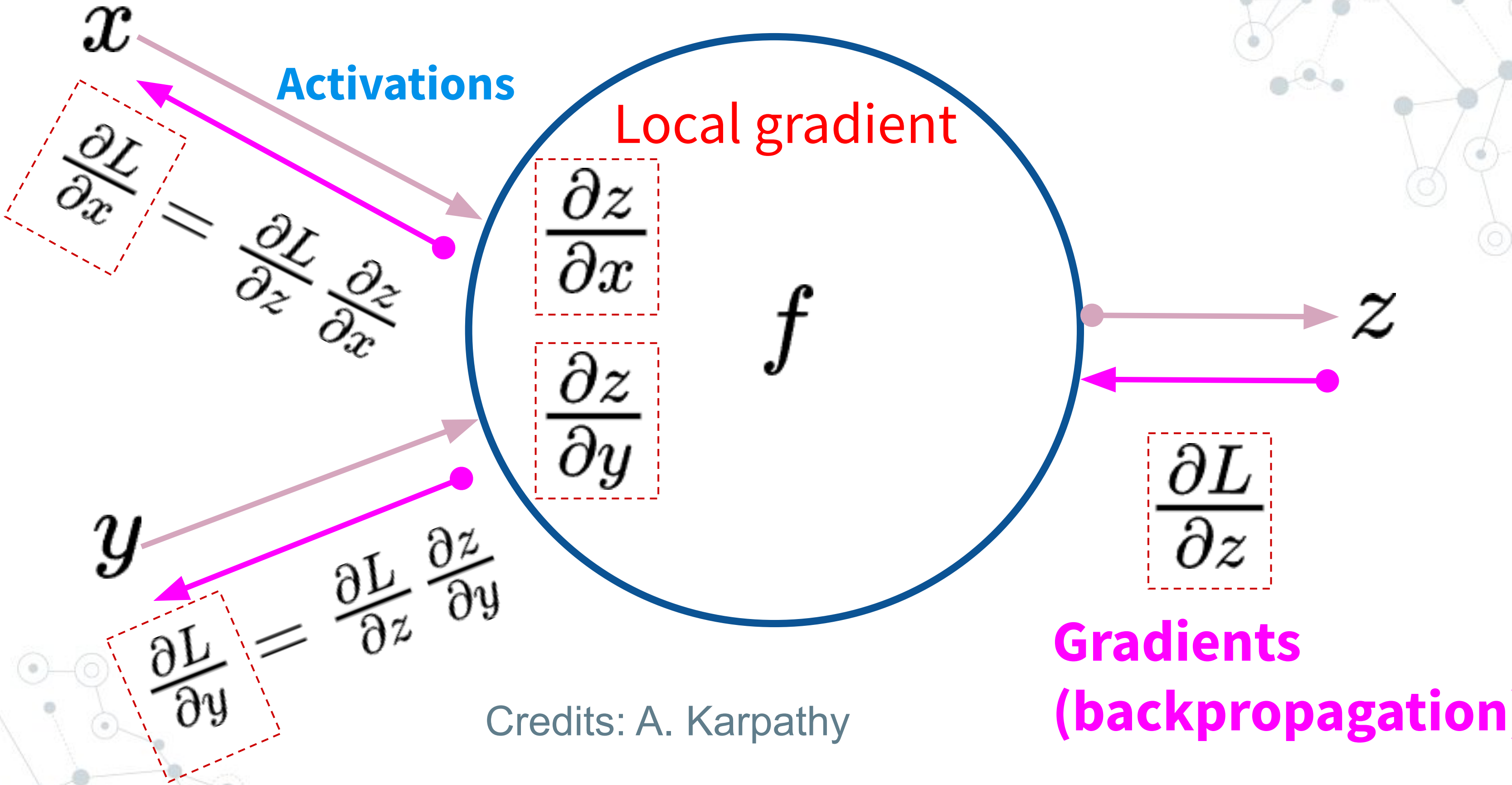
Credits: A. Karpathy

Backpropagation (neuron level)



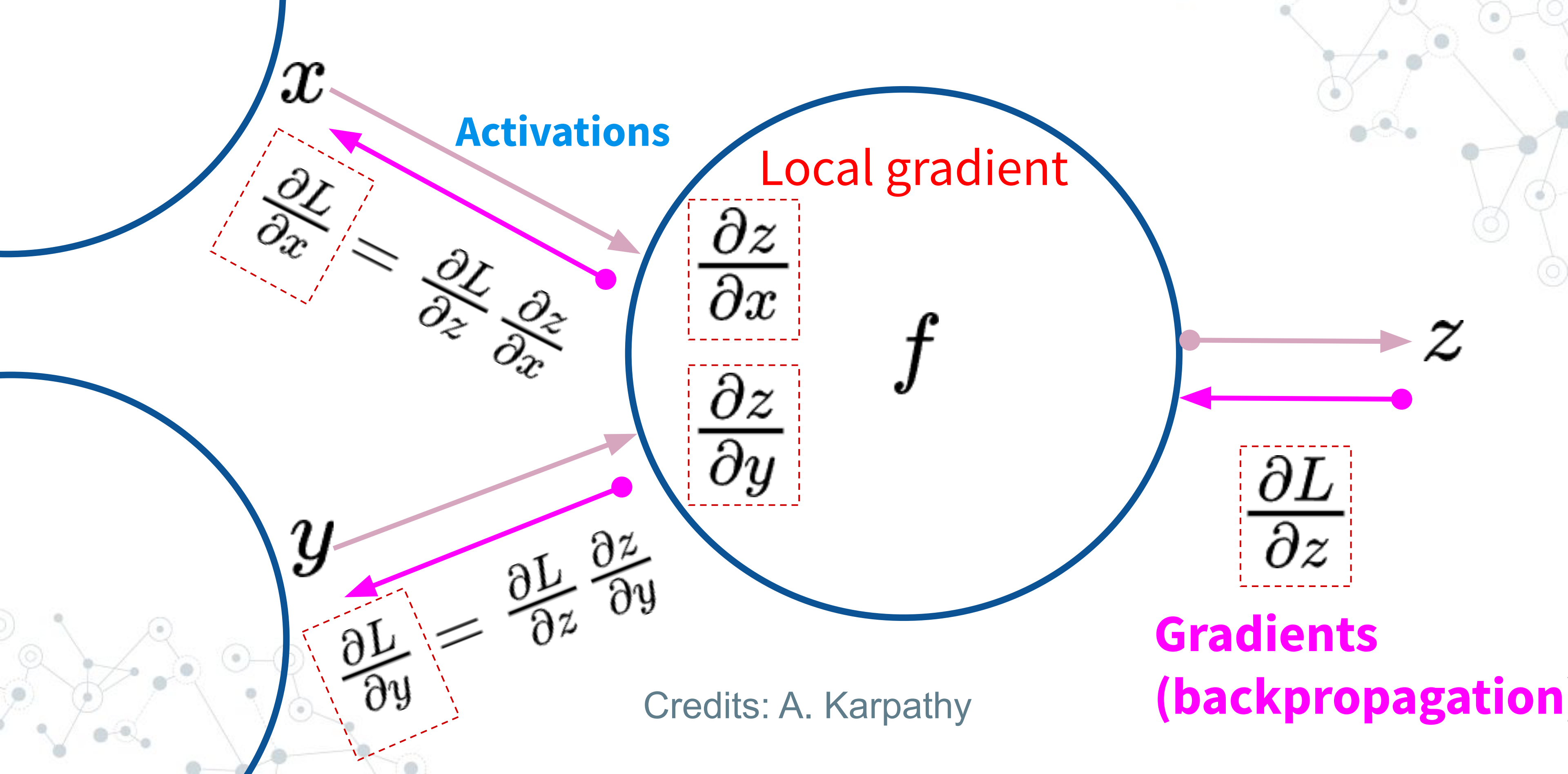
Credits: A. Karpathy

Backpropagation (neuron level)



Credits: A. Karpathy

Backpropagation (neuron level)

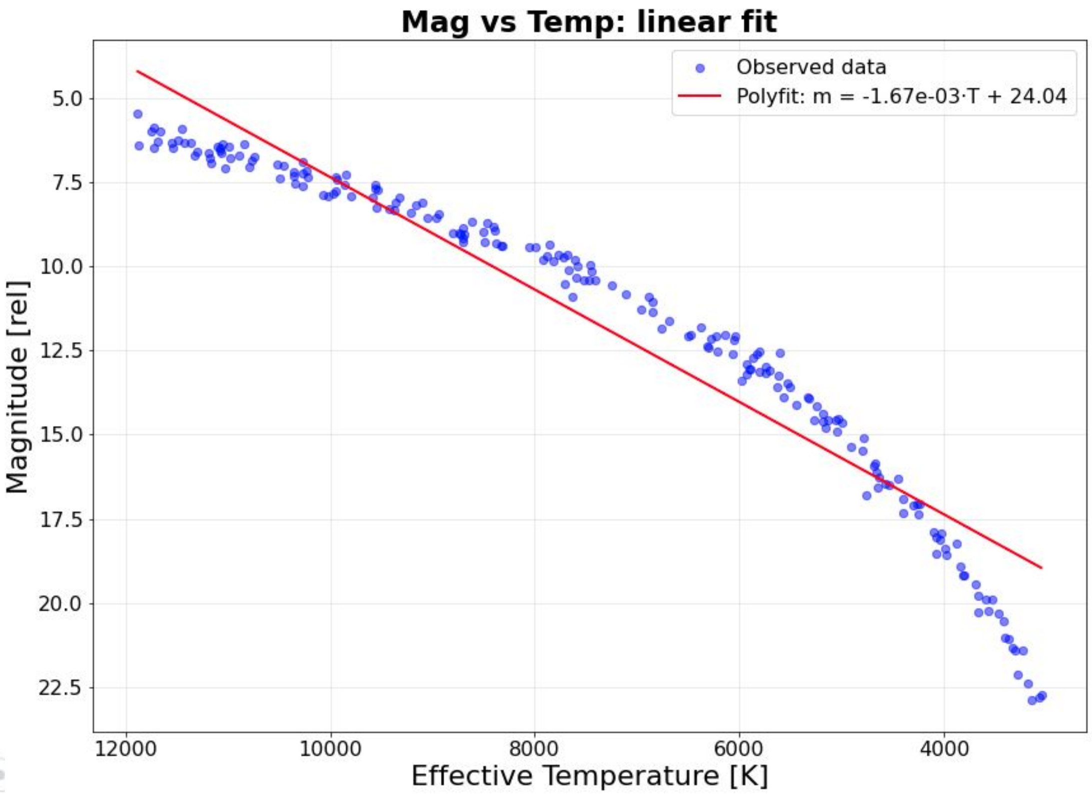


Credits: A. Karpathy

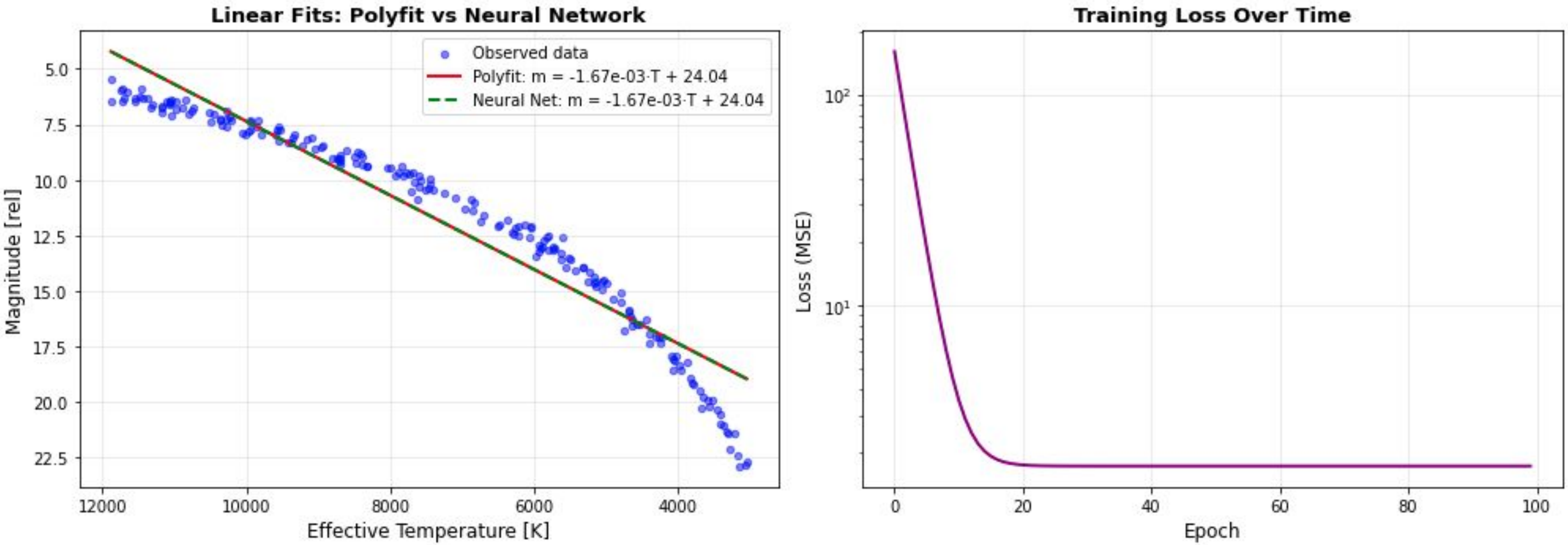
Fit linear NN / non-linear NN

https://github.com/cwestend/IACDEEP_introNN

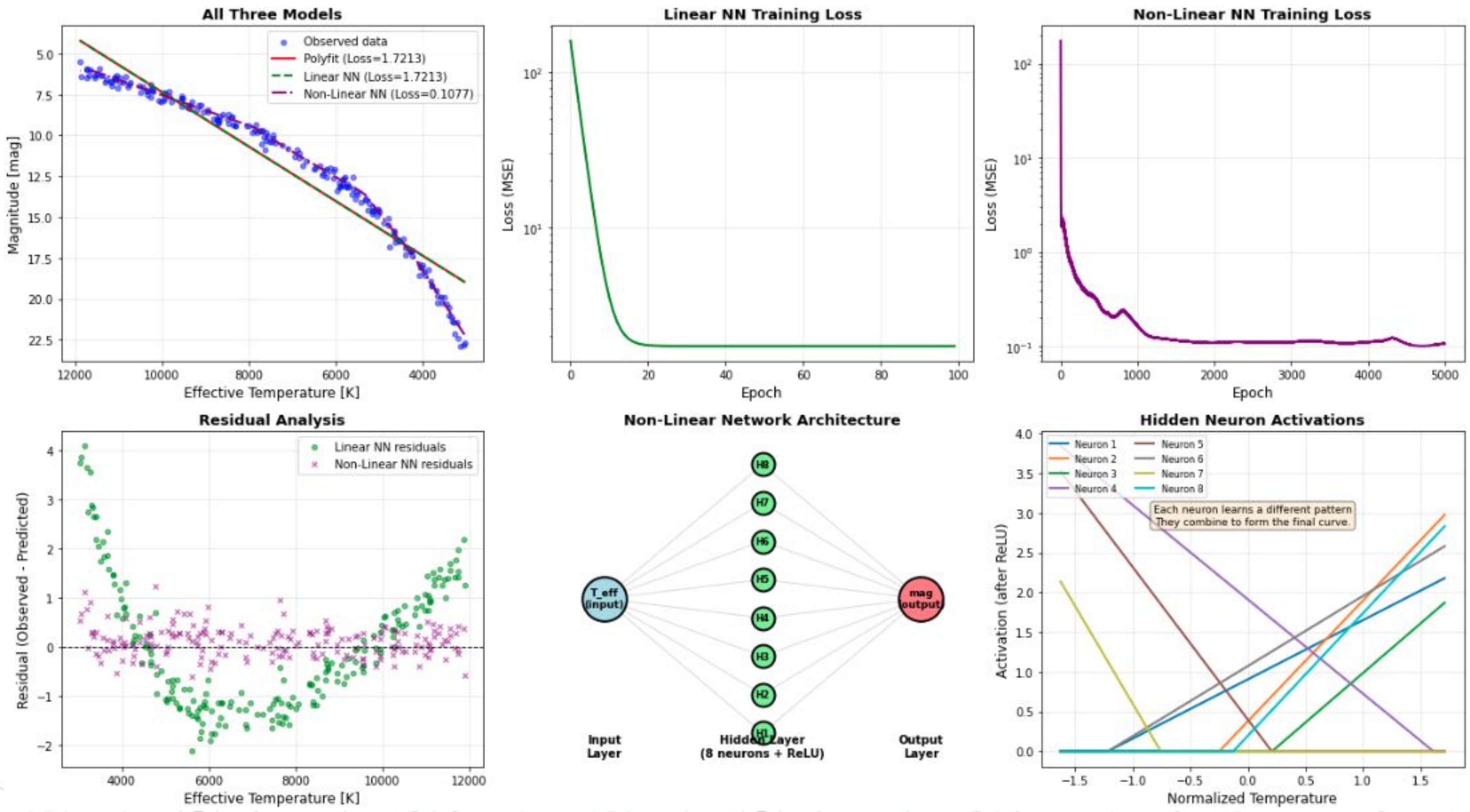
Fit linear NN / non-linear NN



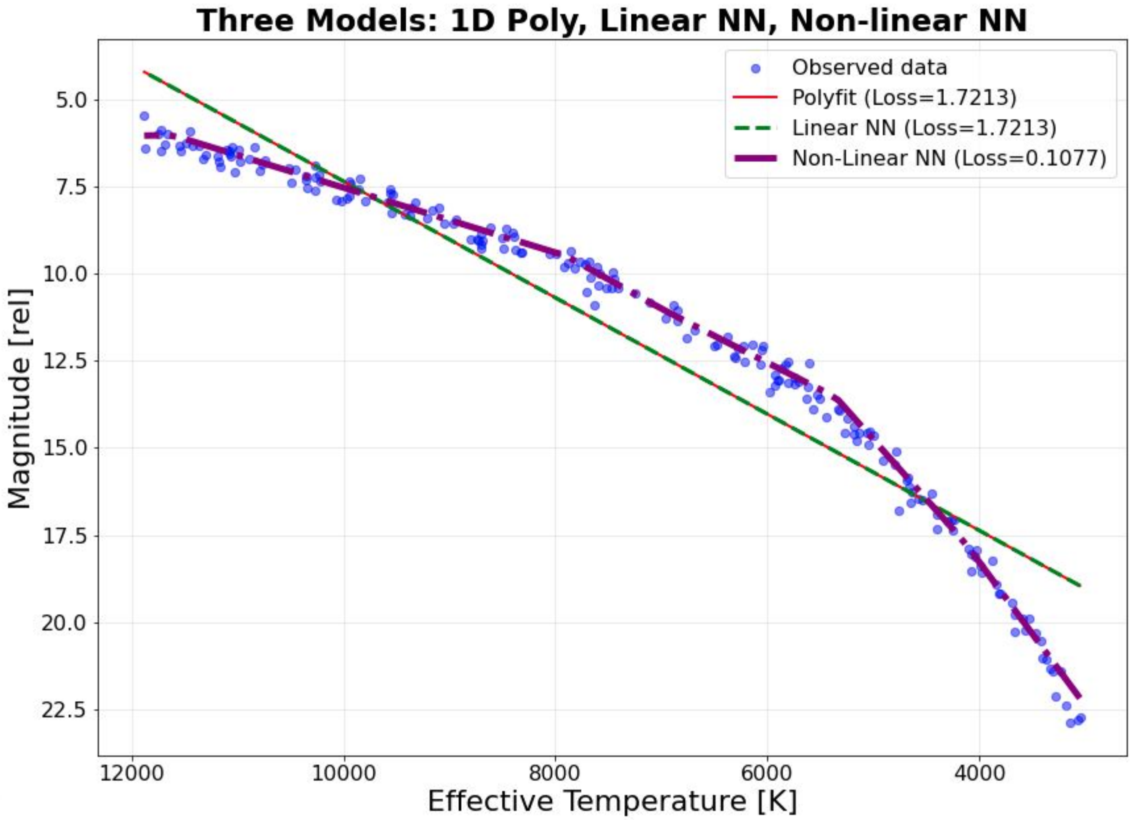
Fit linear NN / non-linear NN



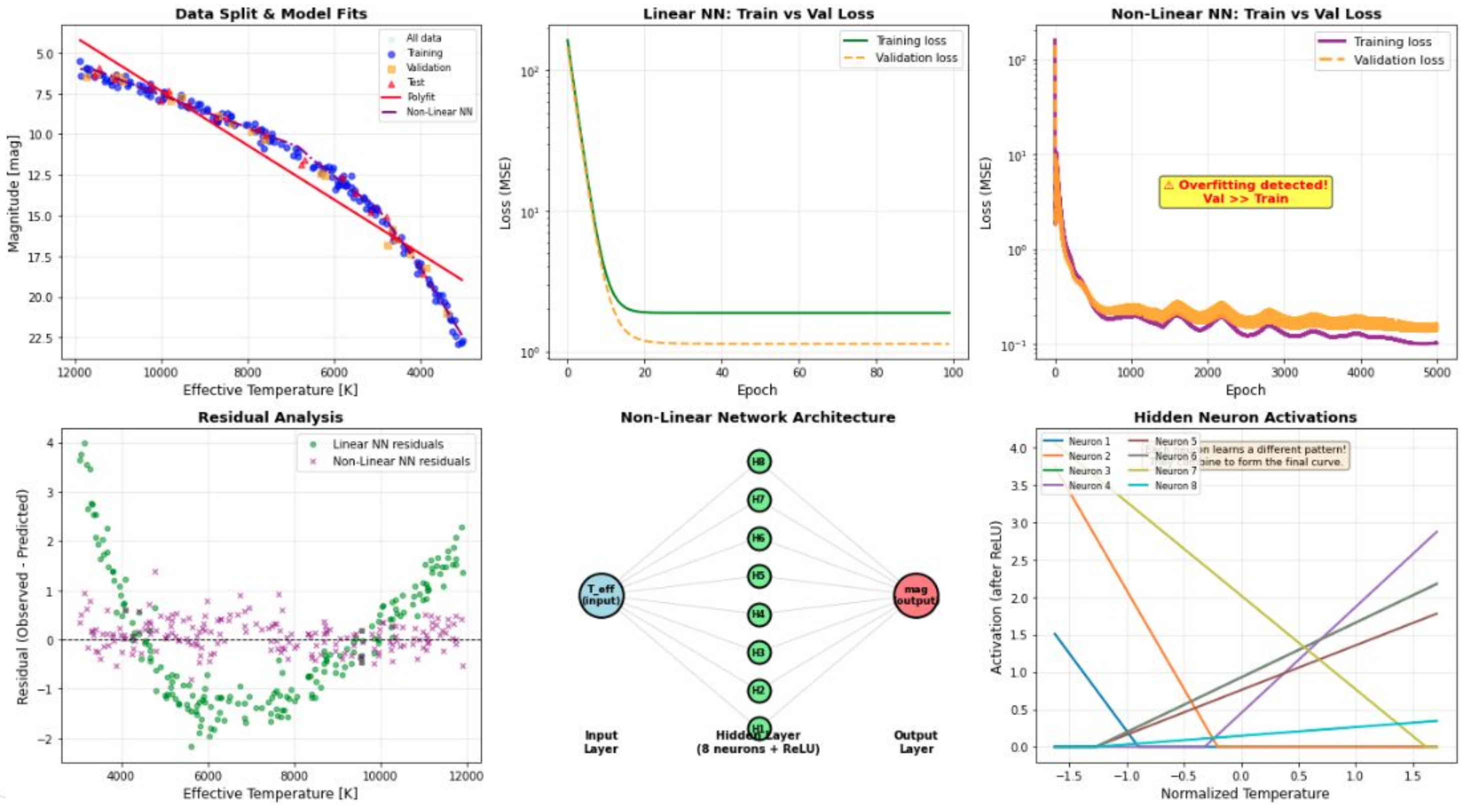
Fit linear NN / non-linear NN



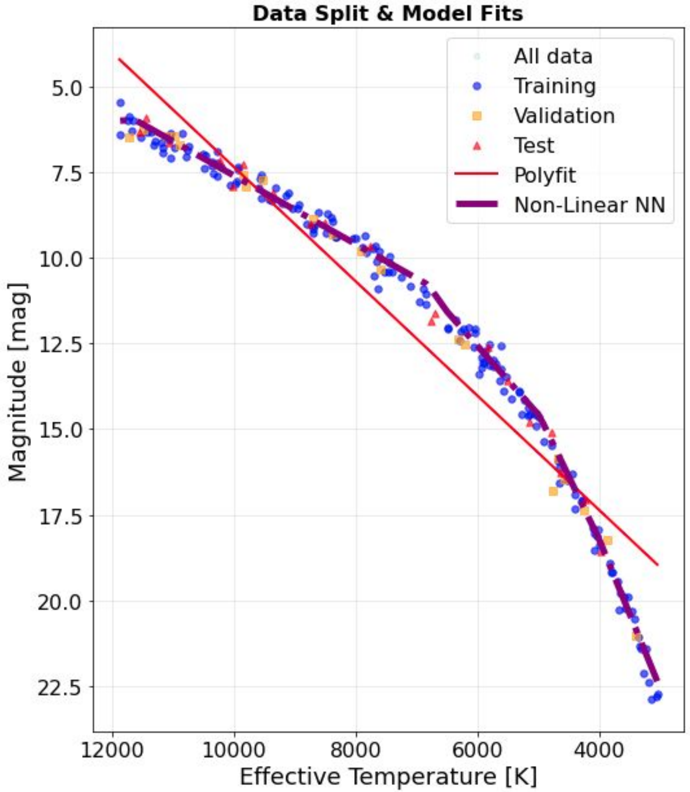
Fit linear NN / non-linear NN

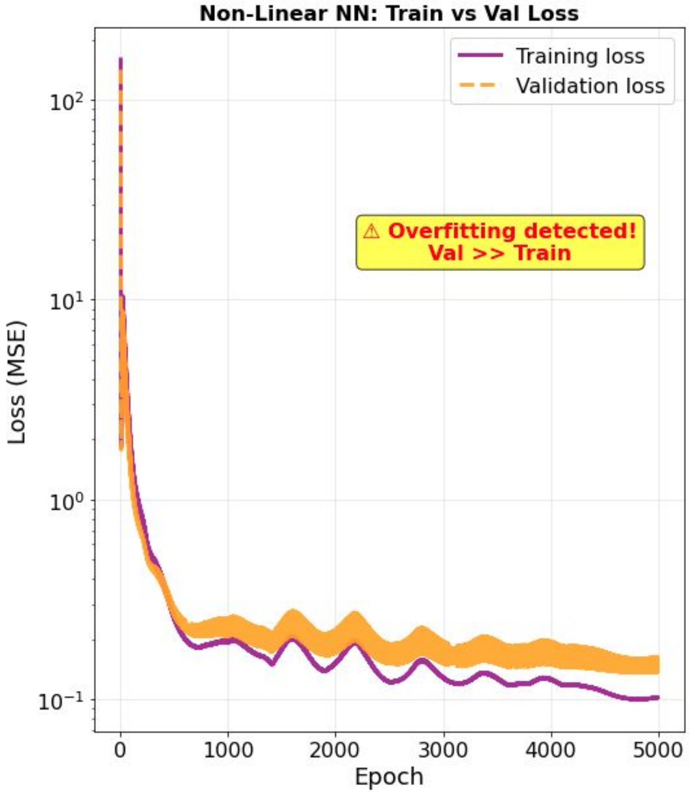


Fit linear NN / non-linear NN with **splits** (80/10/10)



Fit linear NN / non-linear NN with **splits** (80/10/10)





Takeaways:

- **Deep learning:** uses ANN (many hidden), learns from data
- Needs **non-linear** activation functions
- Minimize **loss** (learn):
  - *gradient descent, learning rate, backpropagation*
- **Normalize** data (**must**)
- Needs **lots of data**!

https://github.com/cwestend/IACDEEP_introNN