Neural-ODE for Pharmacokinetics

Benjamin Maurel
PhD Student, INSERM, Digital Pharmacological Twins

5 february 2025

What is a neural differential equation anyway ?

A differential equation with a neural network vector field.

Canonical example - neural ordinary differential equation :

$$y(0) = y_0, \quad \frac{dy}{dt}(t) = f_\theta(t, y(t))$$

What is a neural differential equation anyway ?

A differential equation with a neural network vector field.

Canonical example - neural ordinary differential equation :

$$y(0) = y_0, \quad \frac{dy}{dt}(t) = f_\theta(t, y(t))$$

where $f_\theta : \mathbb{R} \times \mathbb{R}^d \to \mathbb{R}^d$ is any standard network—today, often a feedforward network. (Rico-Martínez et al., Chem. Eng. Comm. 1992 ; Chen et al., NeurIPS 2018)

What is a neural differential equation anyway ?

A differential equation with a neural network vector field.

Canonical example - neural ordinary differential equation :

$$y(0) = y_0, \quad \frac{dy}{dt}(t) = f_\theta(t, y(t))$$

where $f_\theta : \mathbb{R} \times \mathbb{R}^d \to \mathbb{R}^d$ is any standard network—today, often a feedforward network. (Rico-Martínez et al., Chem. Eng. Comm. 1992 ; Chen et al., NeurIPS 2018) **DE justification : the SIR model.** **DL justification : Euler method on :**

$$\frac{d}{dt}\begin{pmatrix} s(t) \\ i(t) \\ r(t) \end{pmatrix} = \begin{pmatrix} -bs(t)i(t) \\ bs(t)i(t) - ki(t) \\ ki(t) \end{pmatrix}$$

$$\frac{dy}{dt}(t) = f_\theta(t, y(t))$$

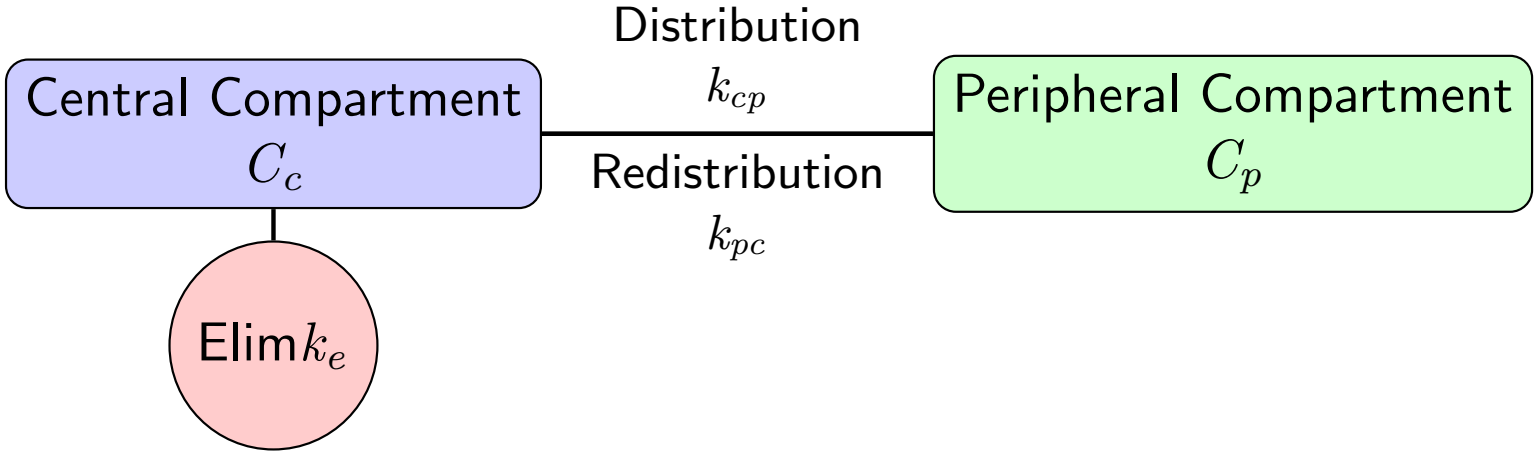
produces a residual network :

$$y_{t_{n+1}} = y_{t_n} + \Delta t\, f_\theta(t_n, y_{t_n}).$$

ODEs everywhere : $y(0) = y_0, \quad \frac{dy}{dt}(t) = f_\theta(t, y(t))$

- Used in physics, biology, economics, and many other fields.

A simple Two-Compartment PK Model



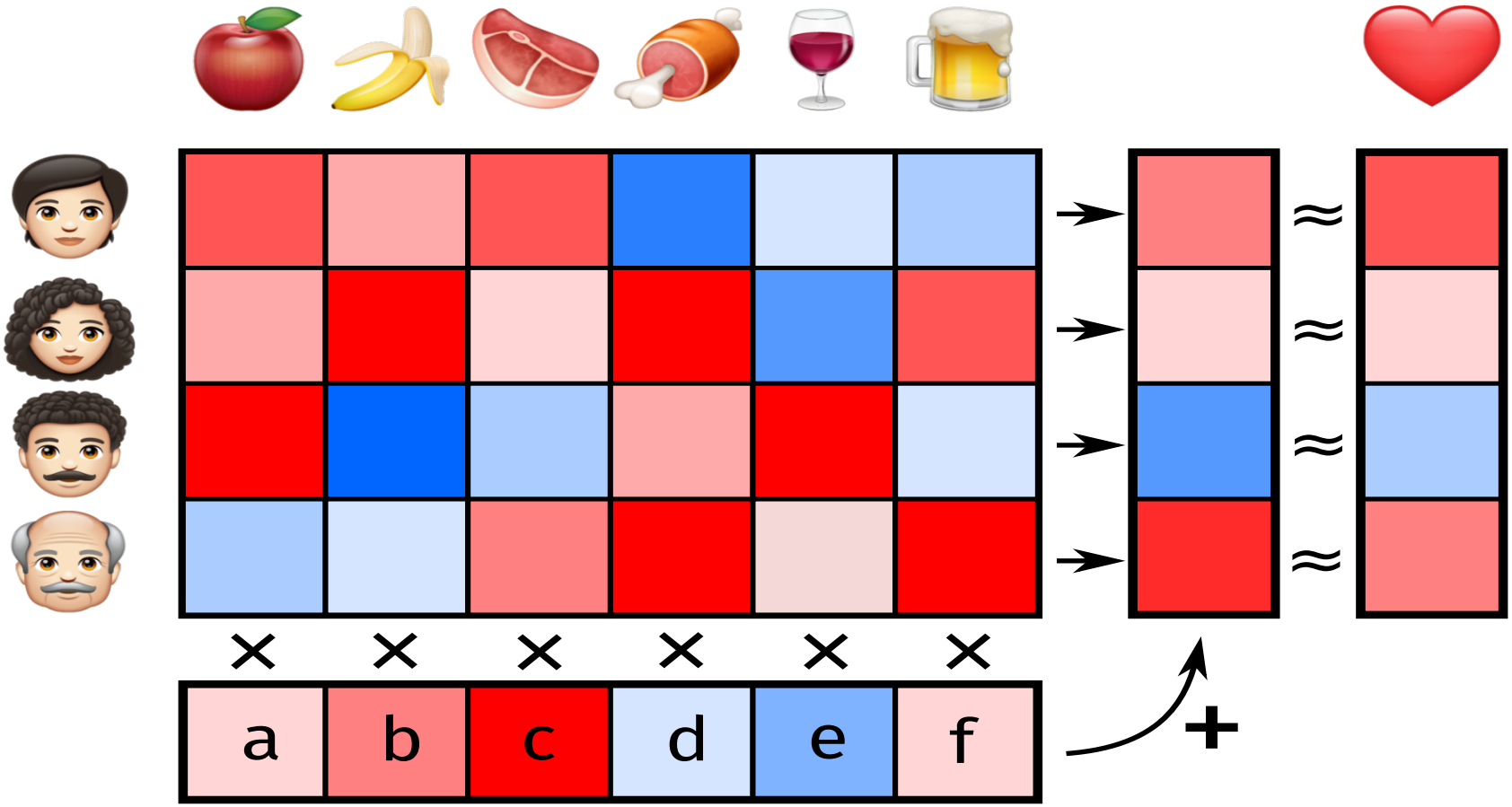
$$\frac{dC_c}{dt} = -k_e\, C_c - k_{cp}\, C_c + k_{pc}\, C_p$$

$$\frac{dC_p}{dt} = k_{cp}\, C_c - k_{pc}\, C_p$$

where C_c and C_p are drug concentrations in the **central** and **peripheral** compartments.

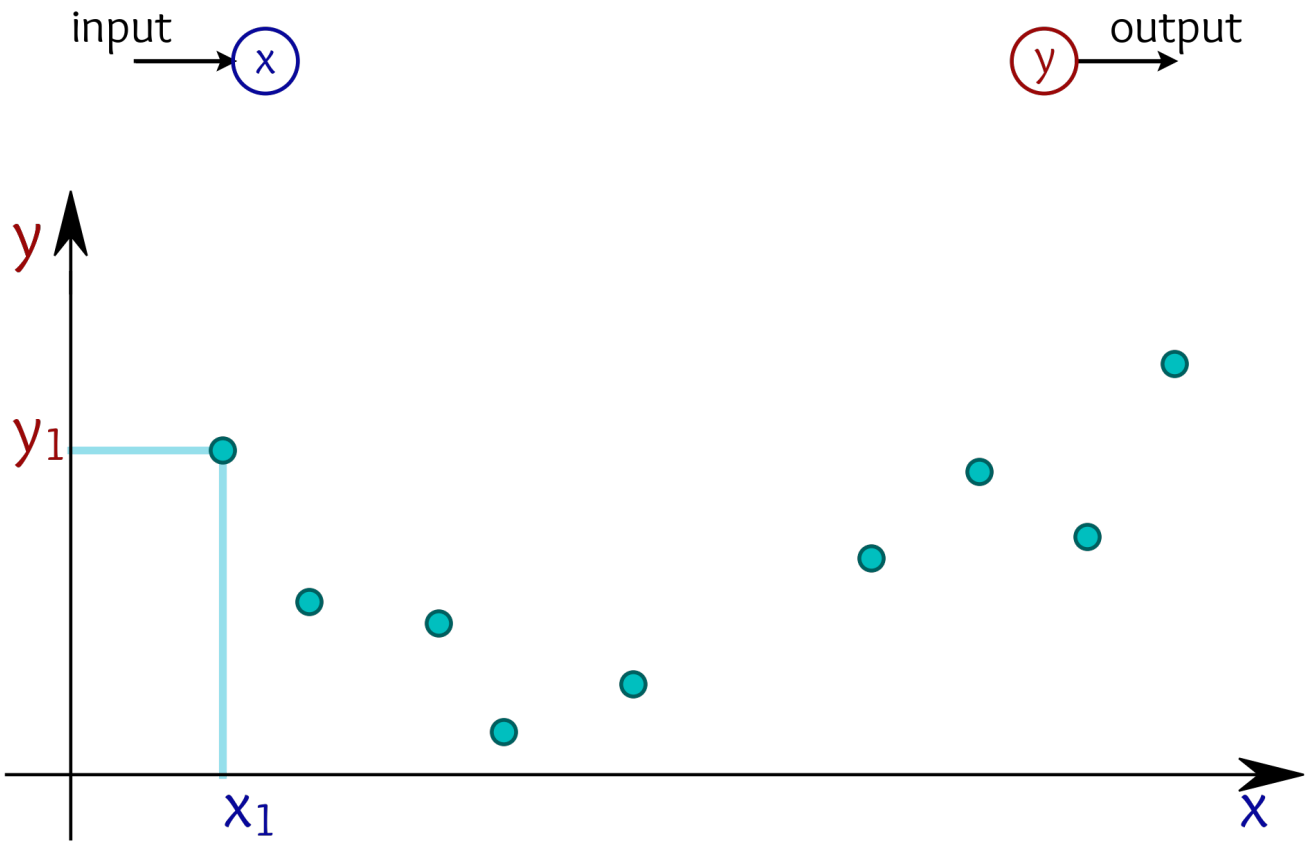
From linear regression to Neural Networks

Big thanks to Jean Feydy (HeKA Team) for the next few images

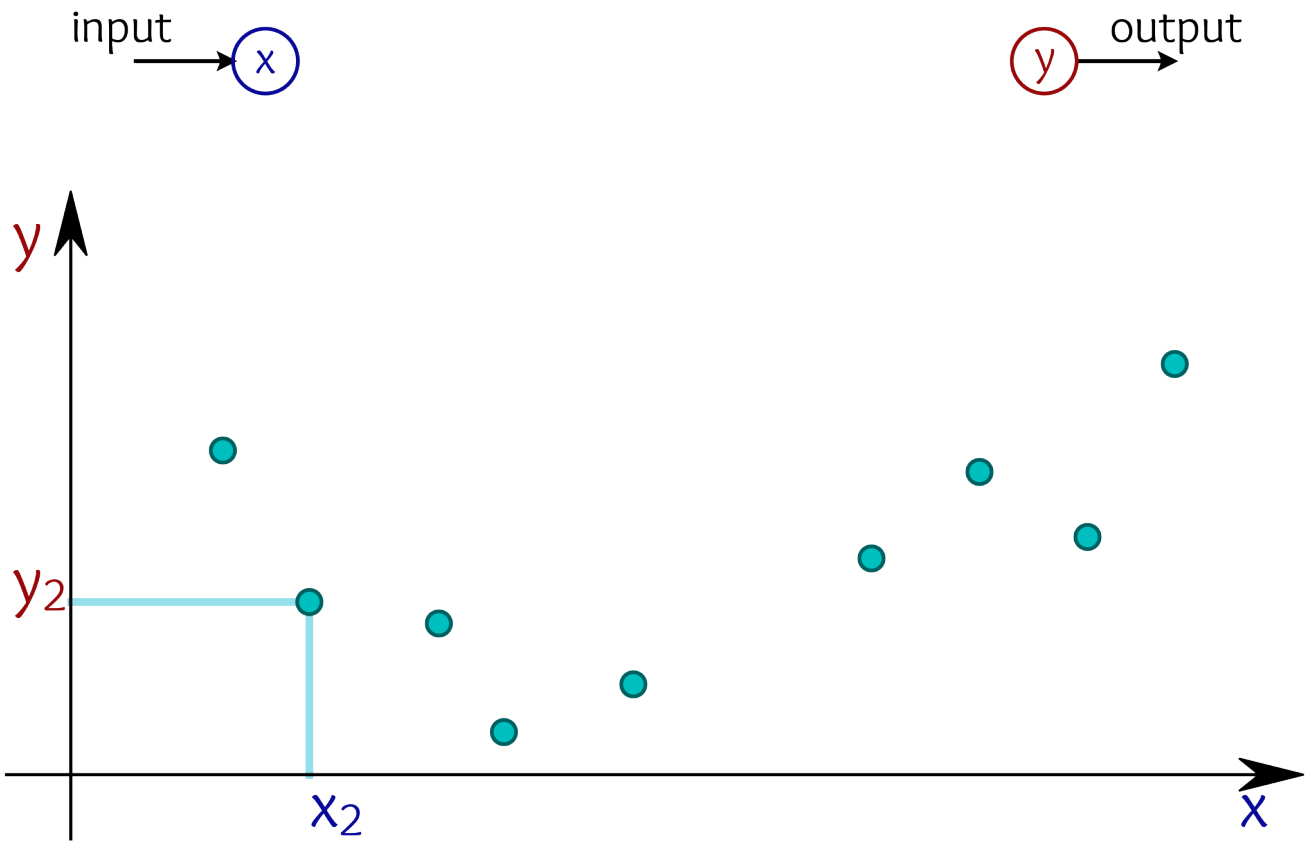


We choose the weights **a**, **b**, . . ., **f**
by minimizing a least squares error.

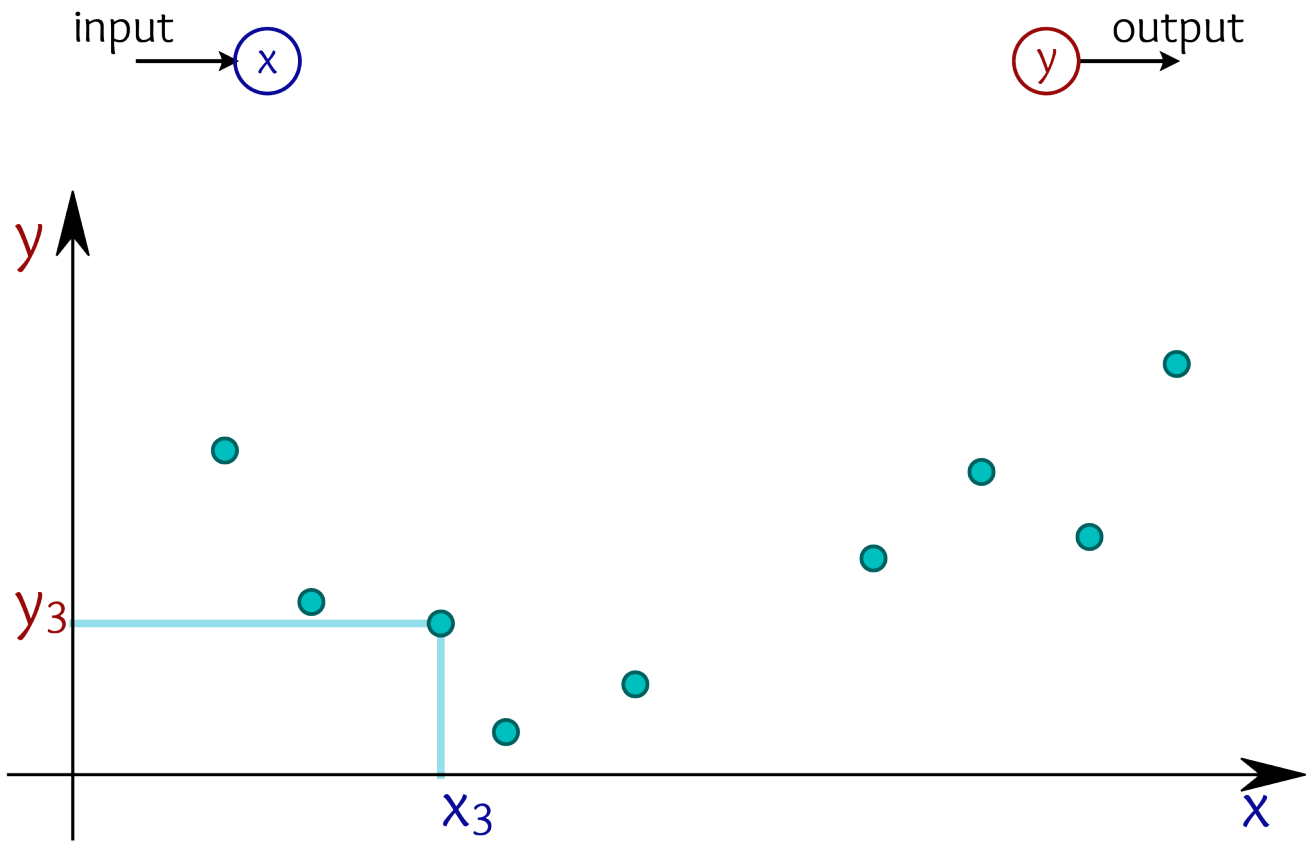
A toy dataset, in dimension 1



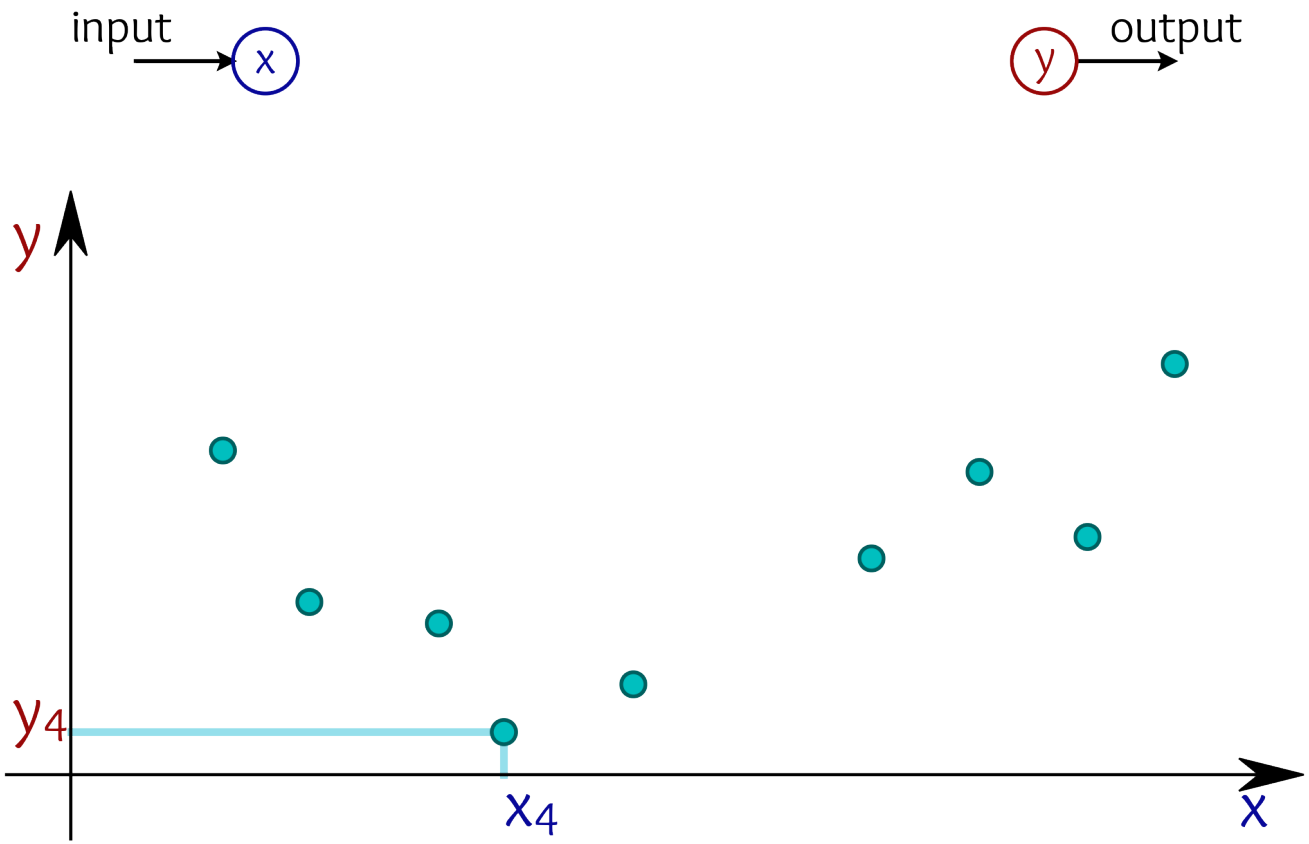
A toy dataset, in dimension 1



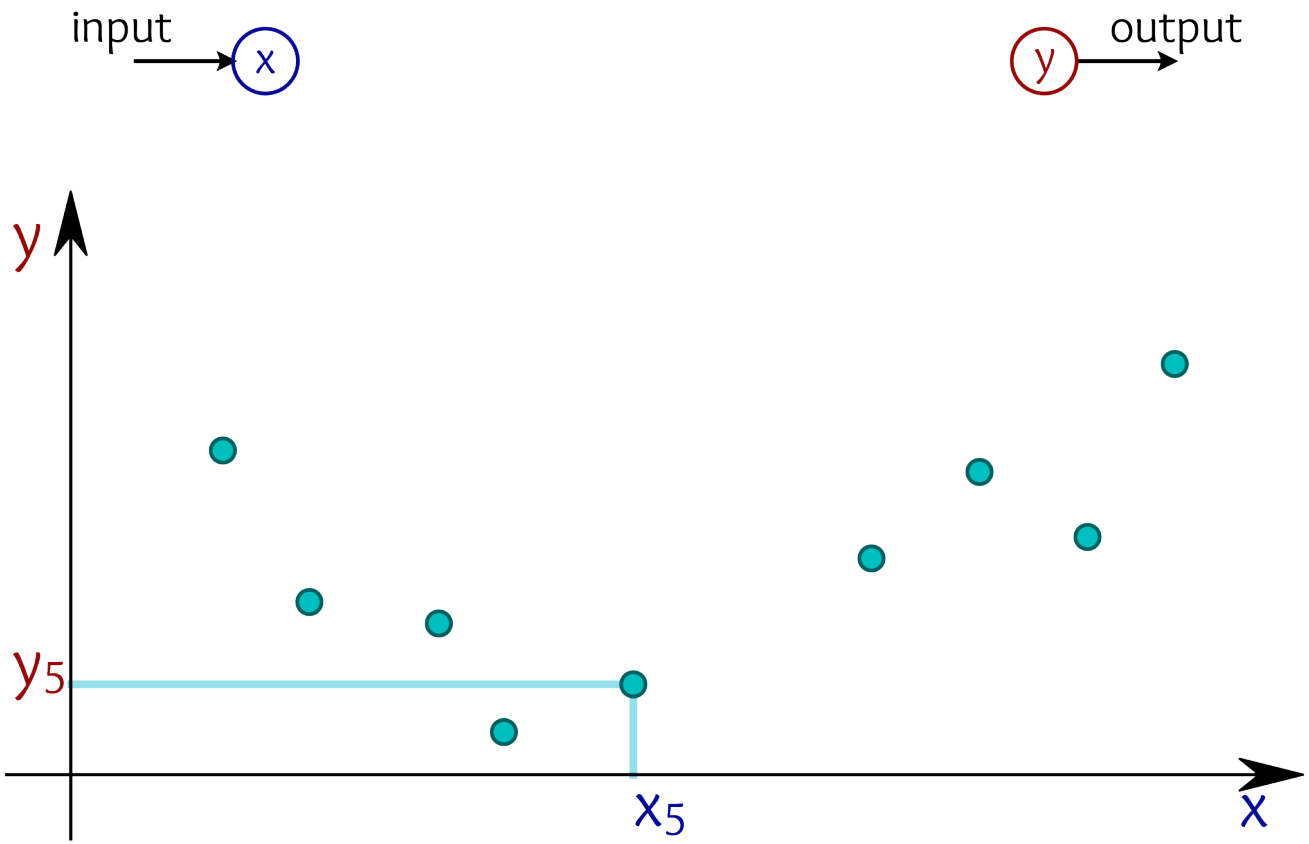
A toy dataset, in dimension 1



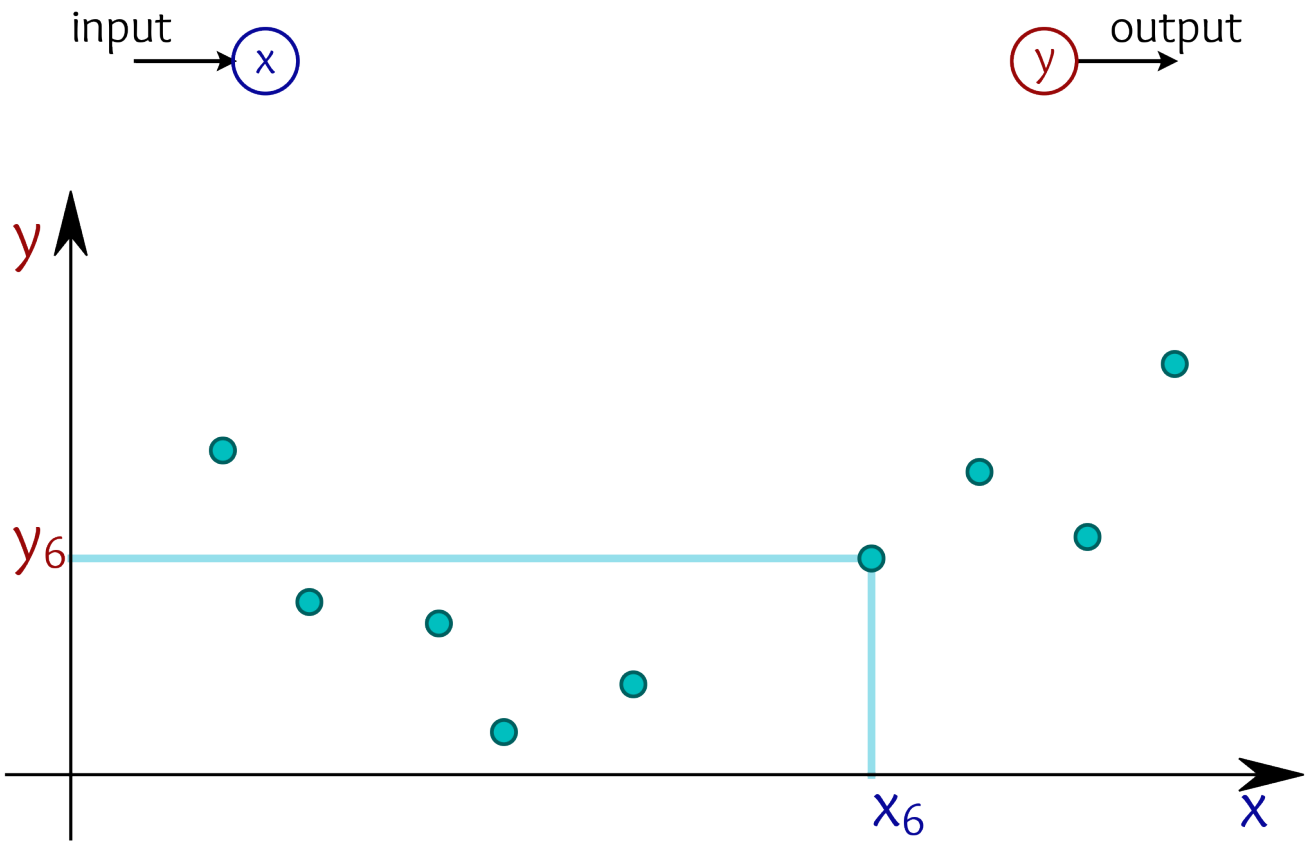
A toy dataset, in dimension 1



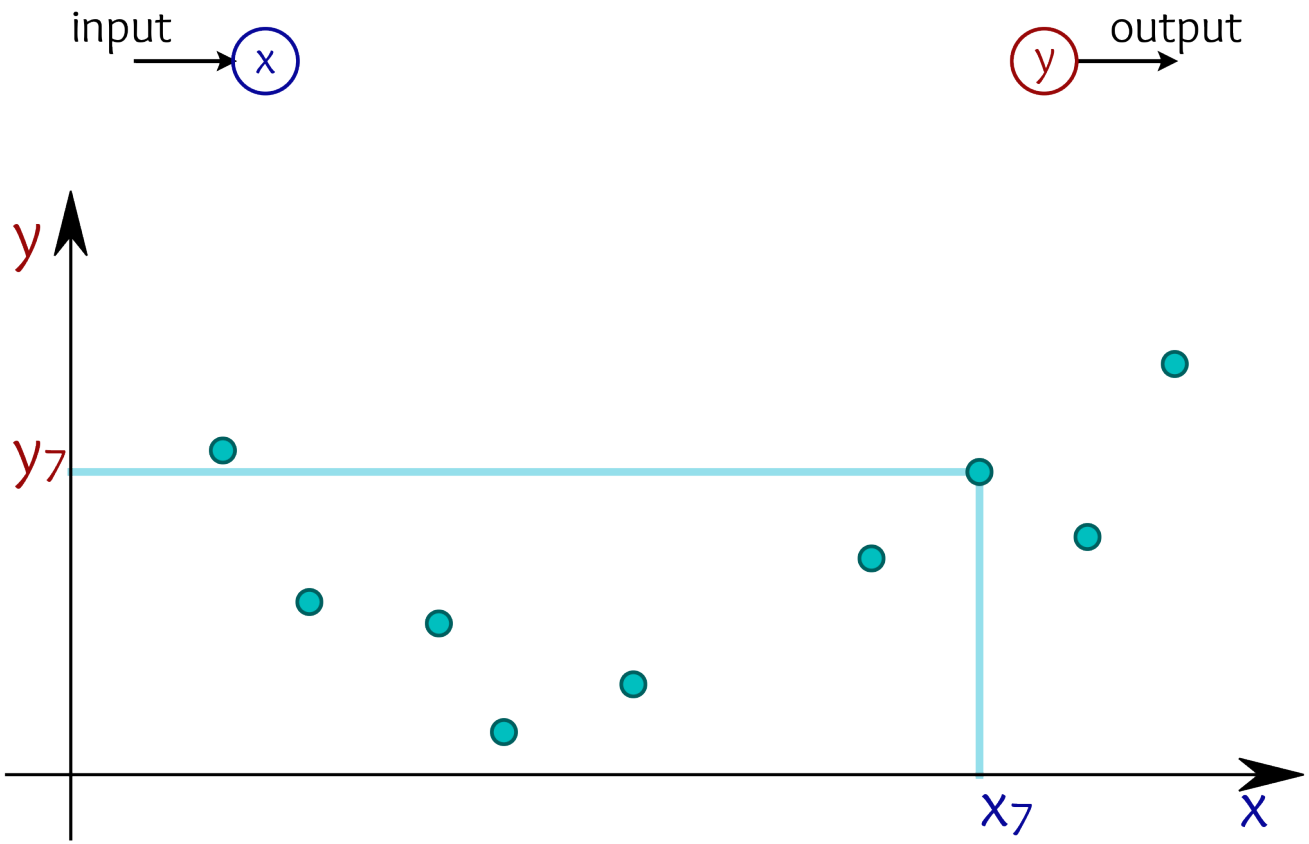
A toy dataset, in dimension 1



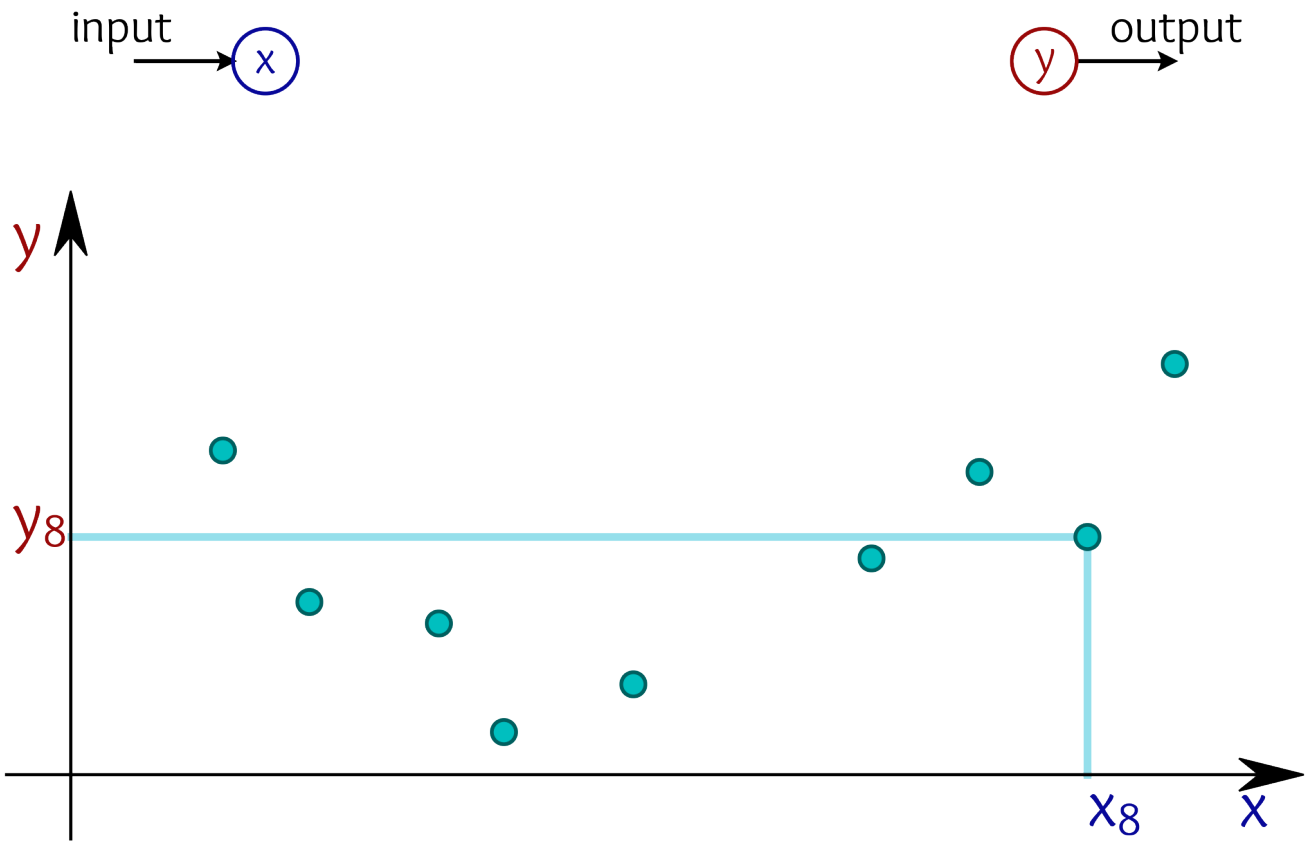
A toy dataset, in dimension 1



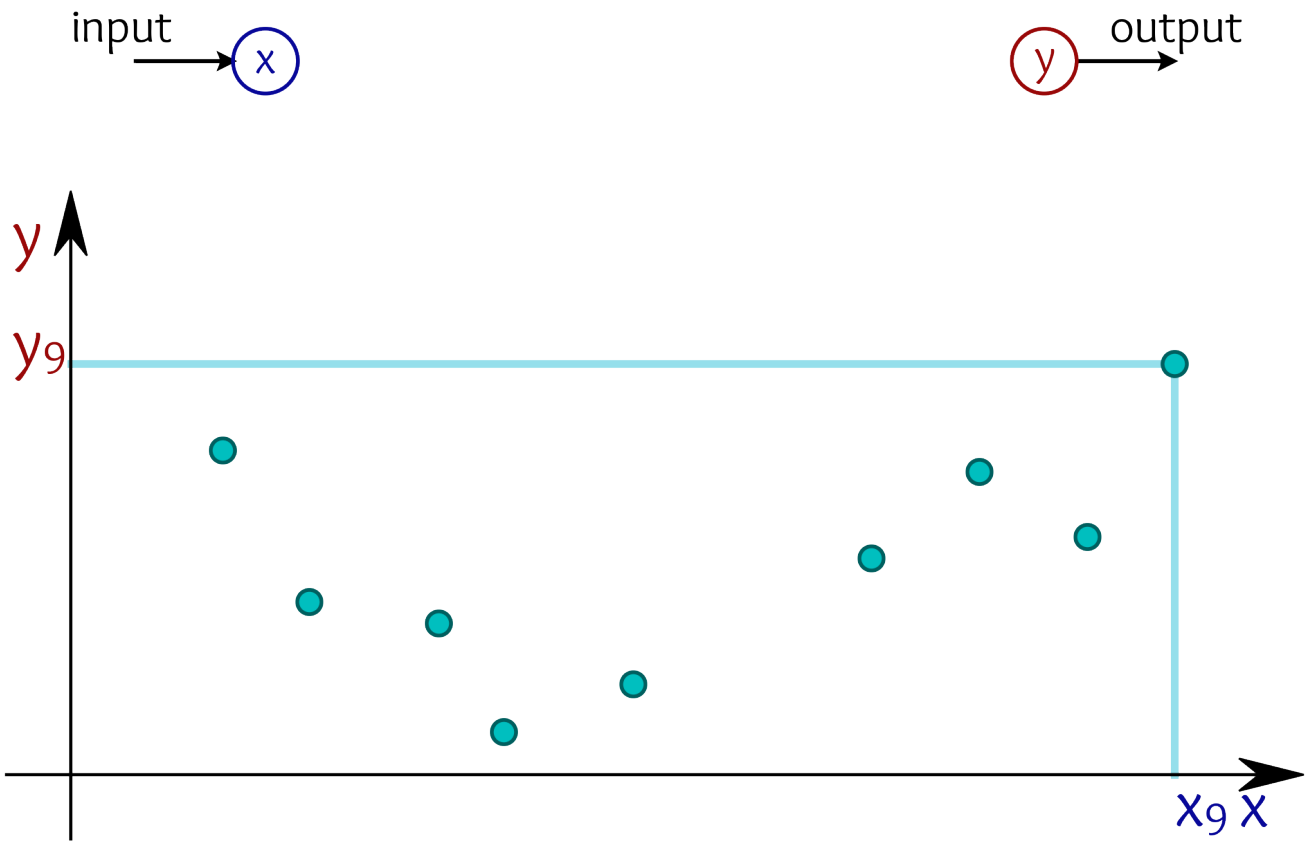
A toy dataset, in dimension 1



A toy dataset, in dimension 1

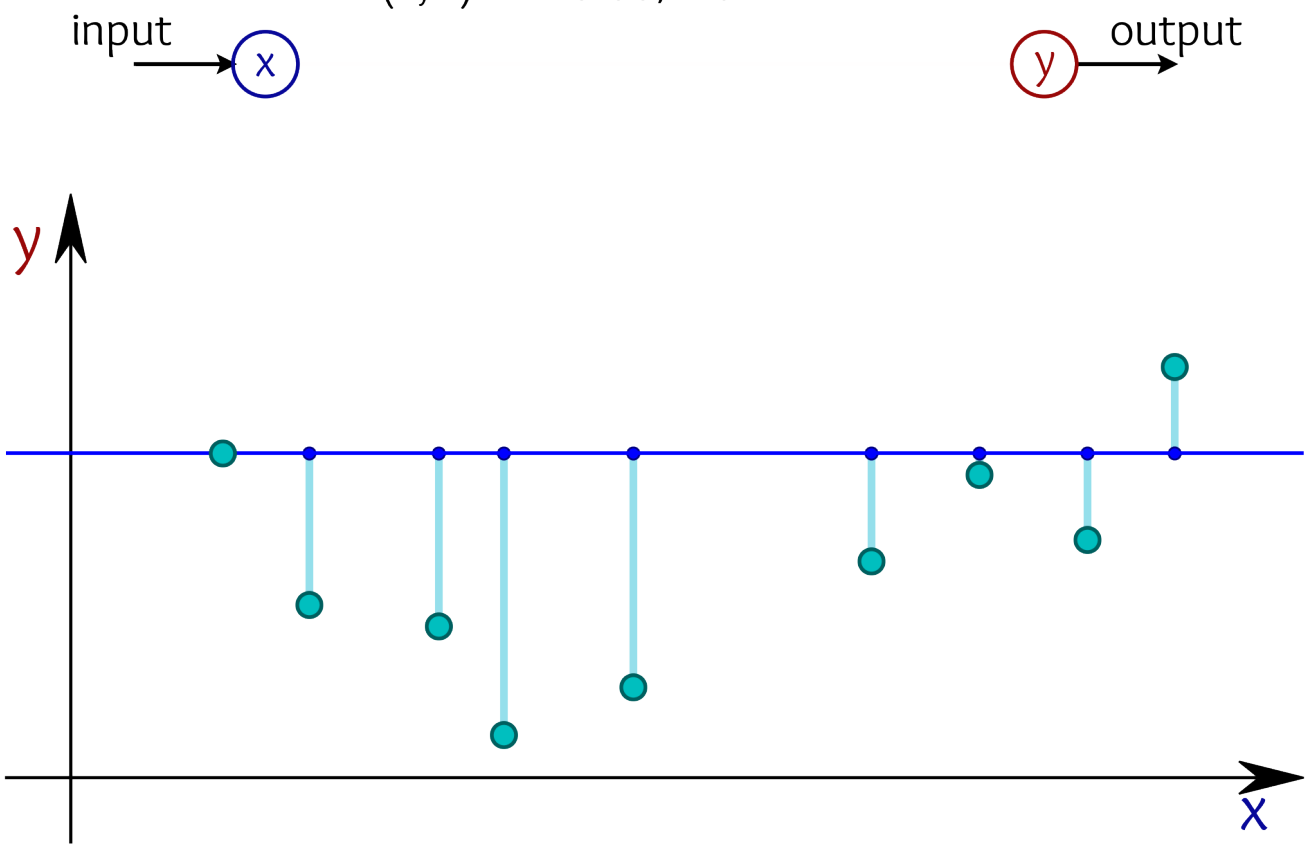


A toy dataset, in dimension 1



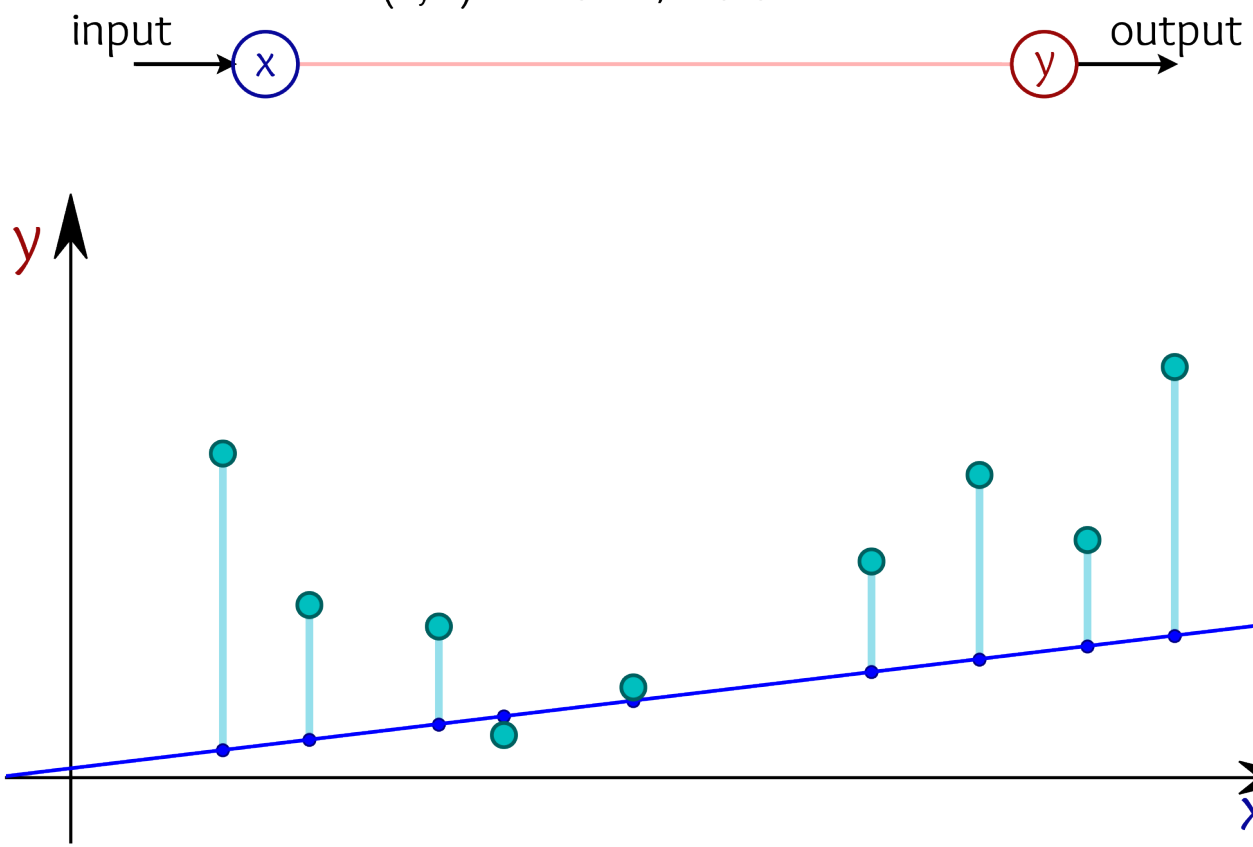
Iterative least squares fitting

$$F(a, b; x) = a \cdot x + b$$
$$(a, b) = +0.00, +0.25$$



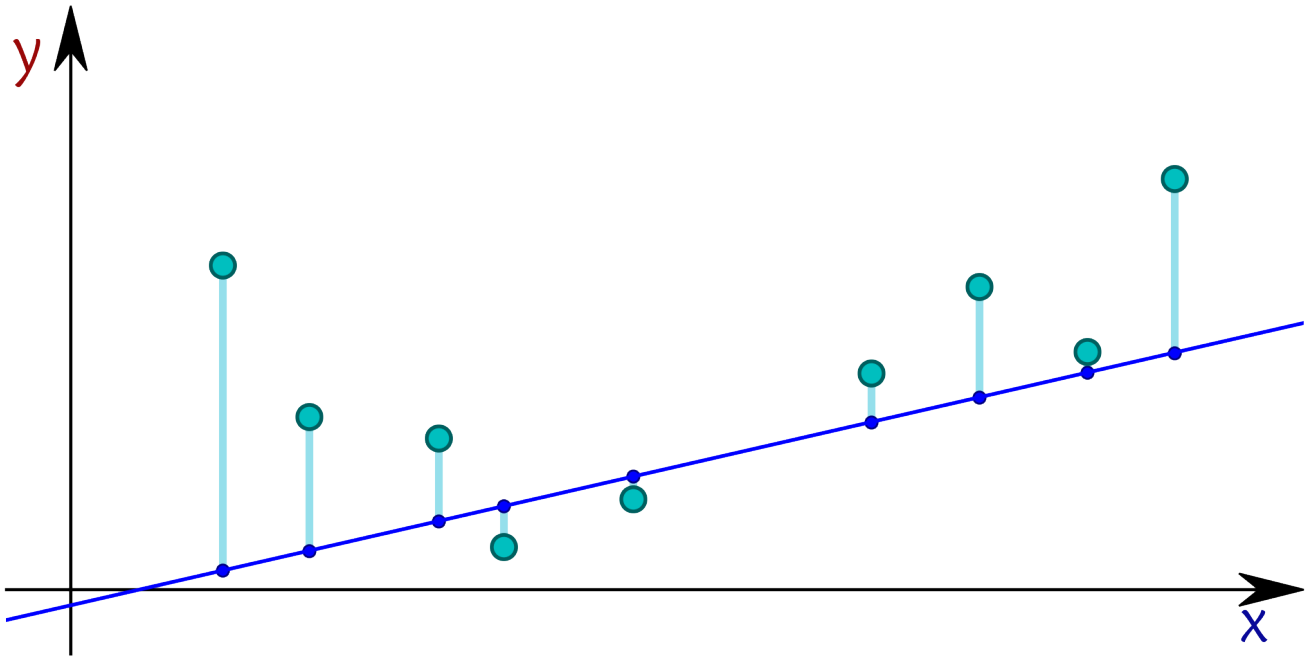
Iterative least squares fitting

$$F(a, b; x) = a \cdot x + b$$
$$(a, b) = +0.12, +0.01$$



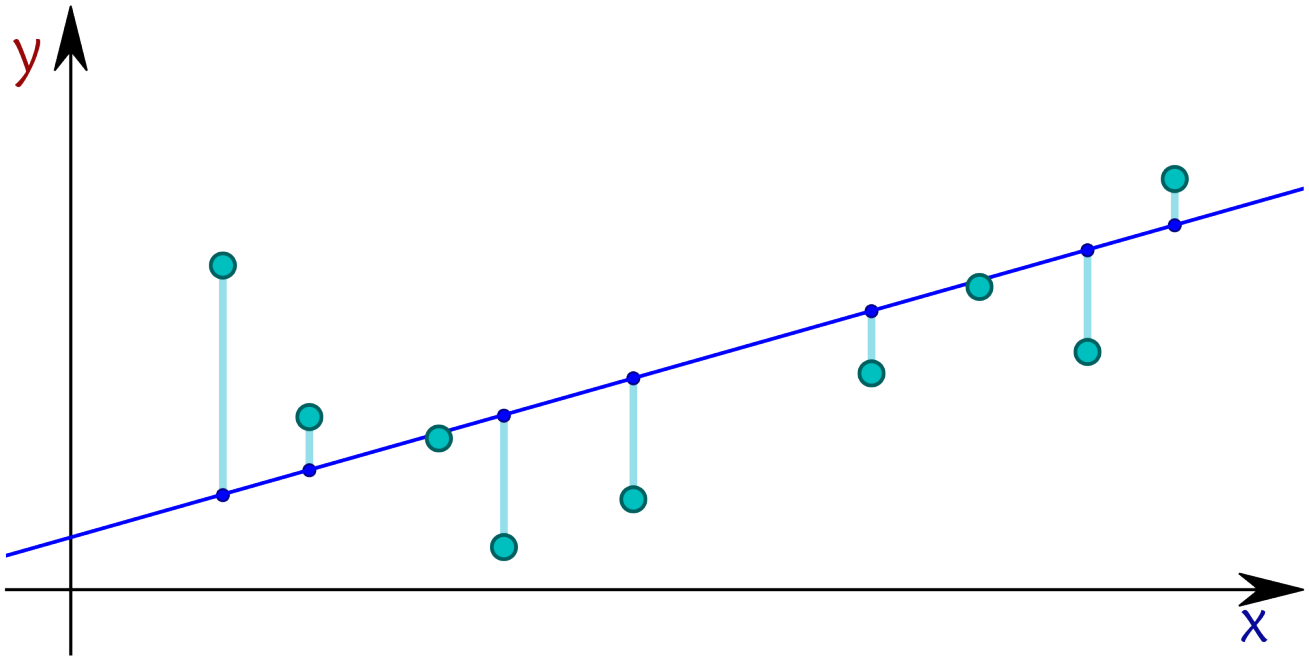
Iterative least squares fitting

$$F(a, b; x) = a \cdot x + b$$
$$(a, b) = +0.23, -0.01$$



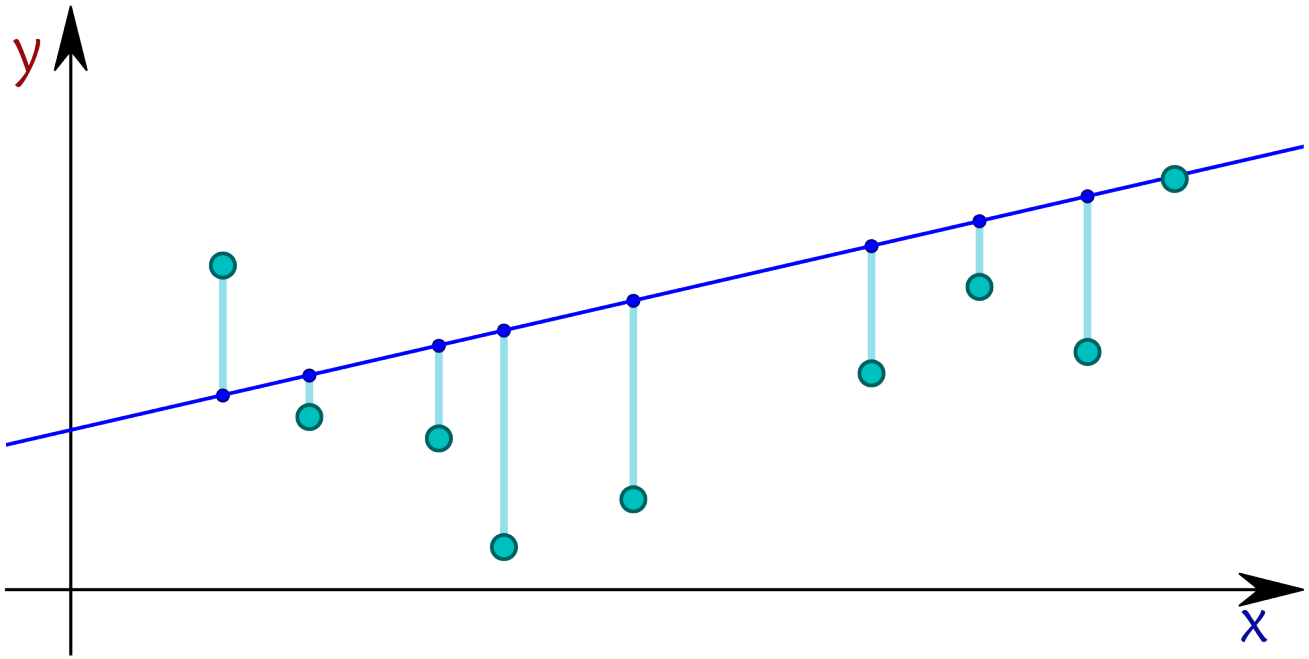
Iterative least squares fitting

$$F(a, b; x) = a \cdot x + b$$
$$(a, b) = +0.28, +0.04$$



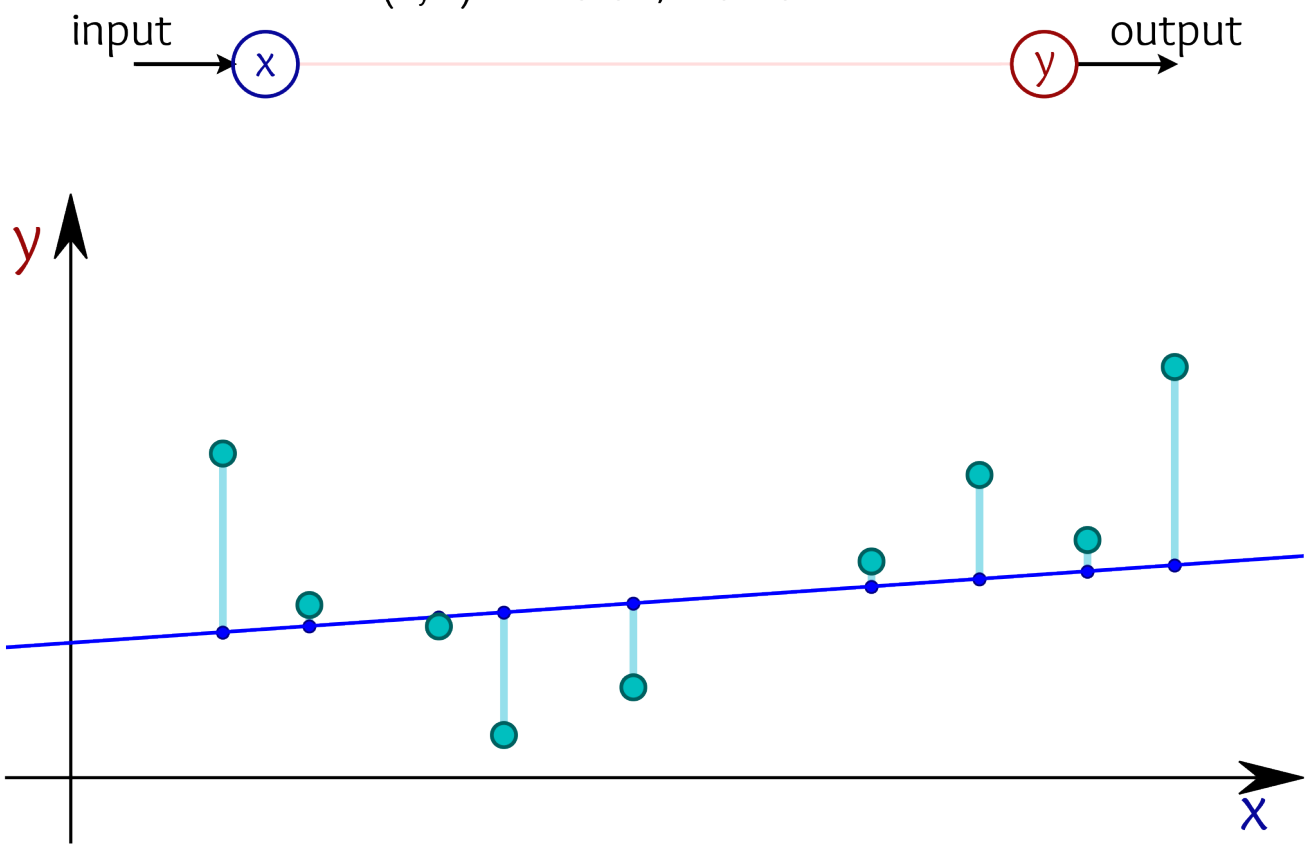
Iterative least squares fitting

$$F(a, b; x) = a \cdot x + b$$
$$(a, b) = +0.23, +0.12$$



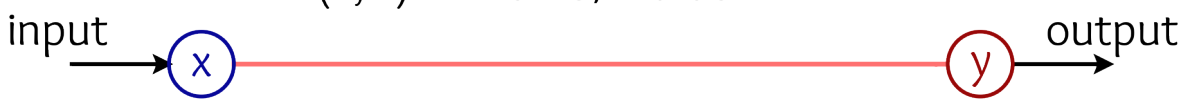
Iterative least squares fitting

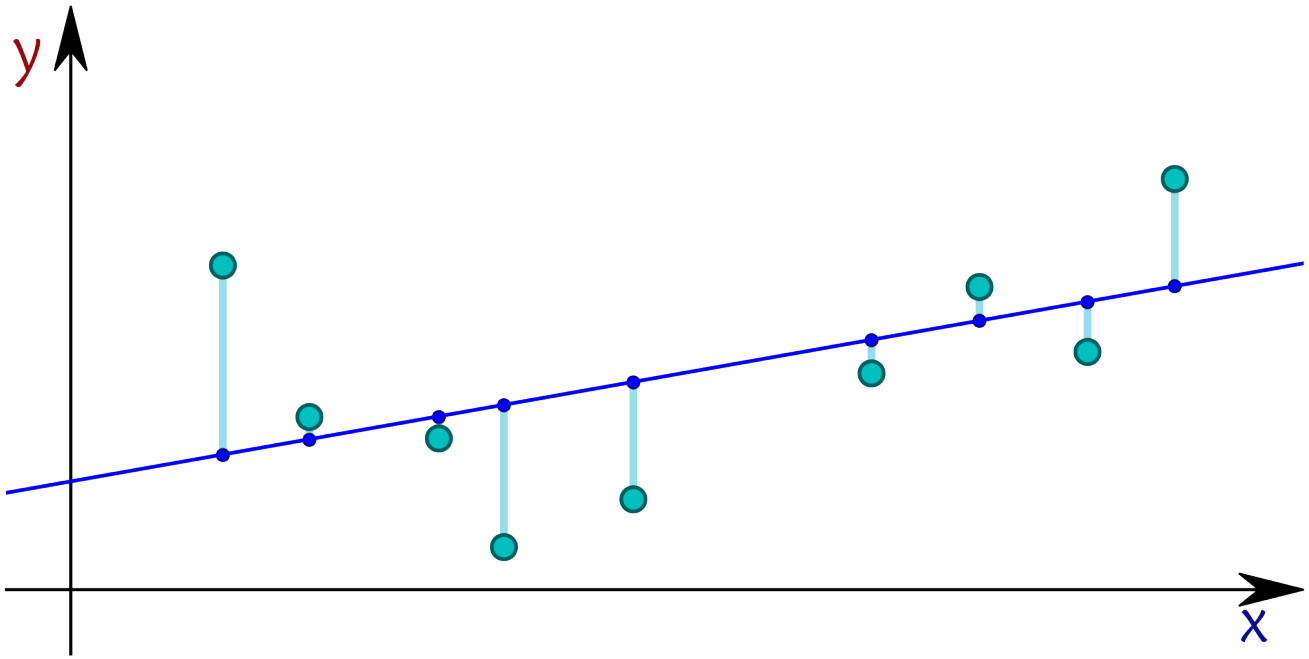
$$F(a, b; x) = a \cdot x + b$$
$$(a, b) = +0.07, +0.10$$



Iterative least squares fitting

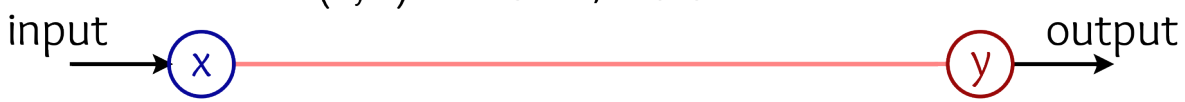
$$F(a, b; x) = a \cdot x + b$$
$$(a, b) = +0.18, +0.08$$

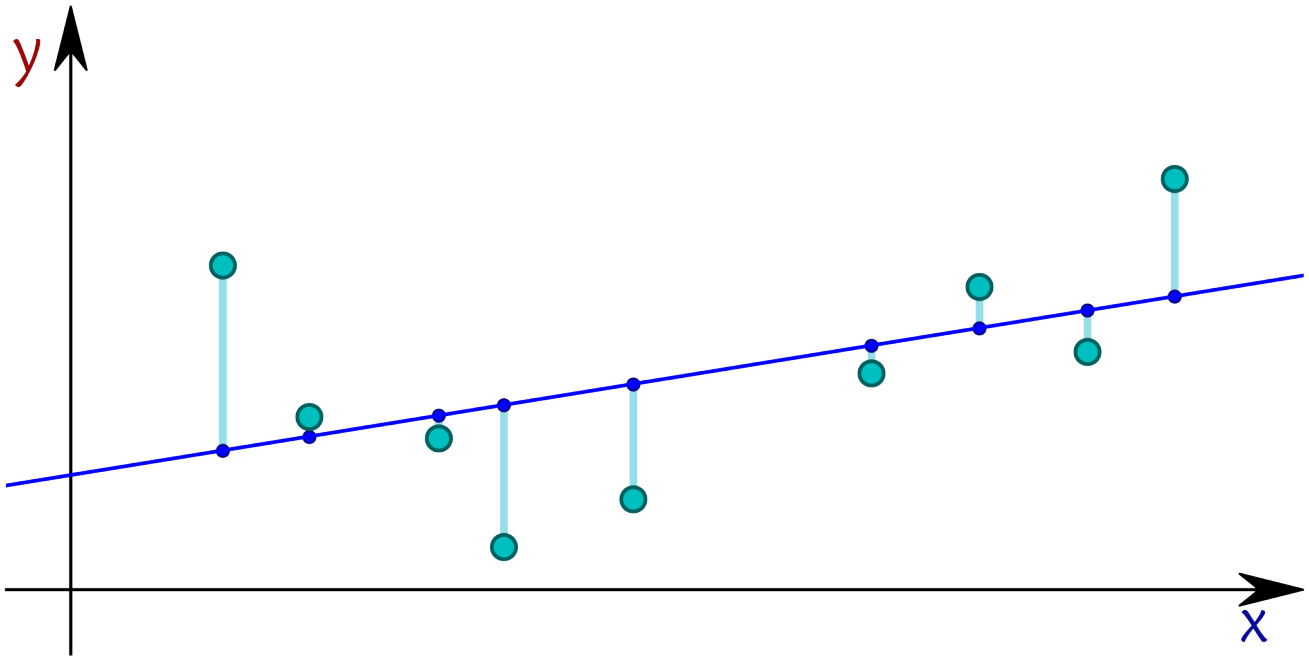




Iterative least squares fitting

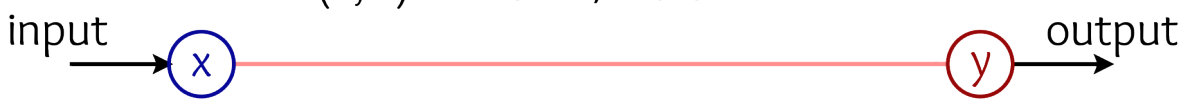
$$F(a, b; x) = a \cdot x + b$$
$$(a, b) = +0.16, +0.09$$

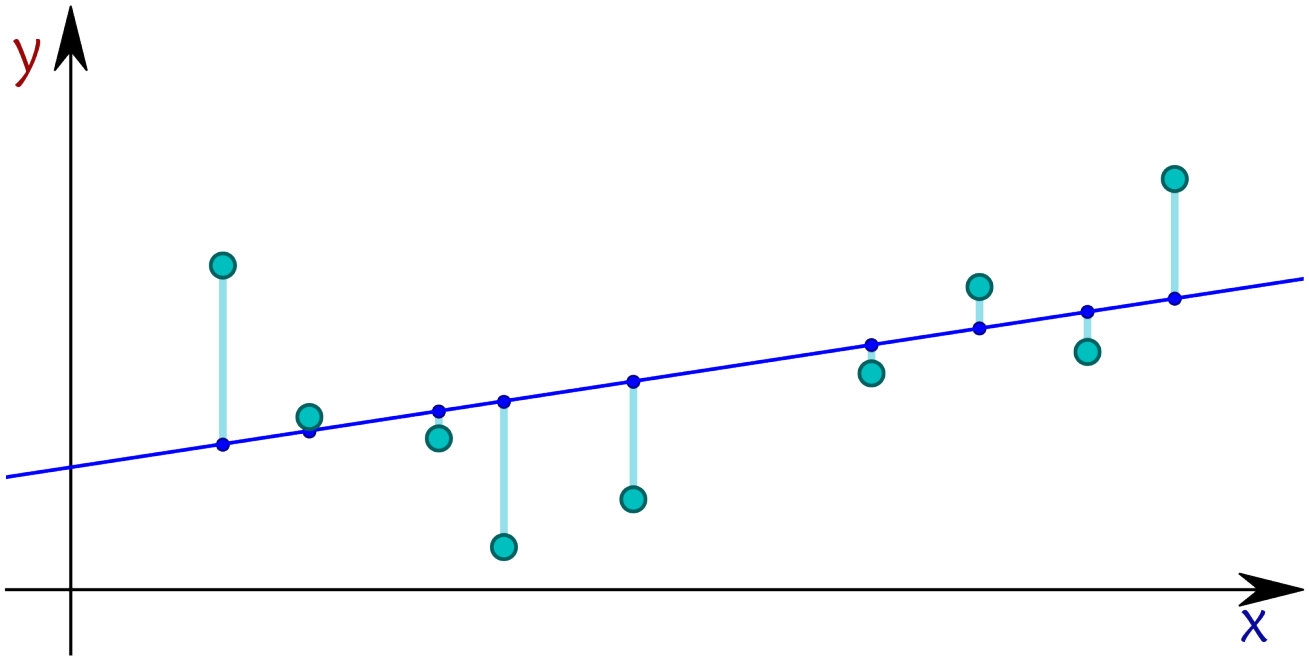




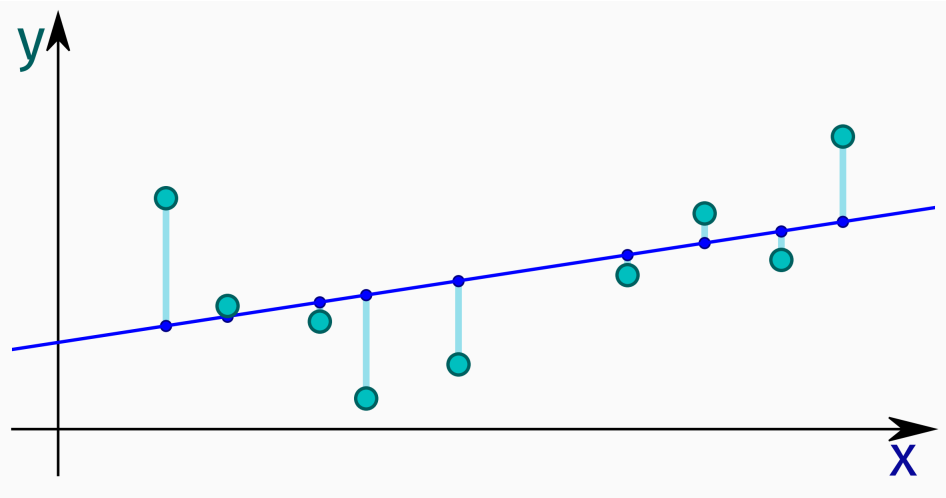
Iterative least squares fitting

$$F(a, b; x) = a \cdot x + b$$
$$(a, b) = +0.15, +0.09$$

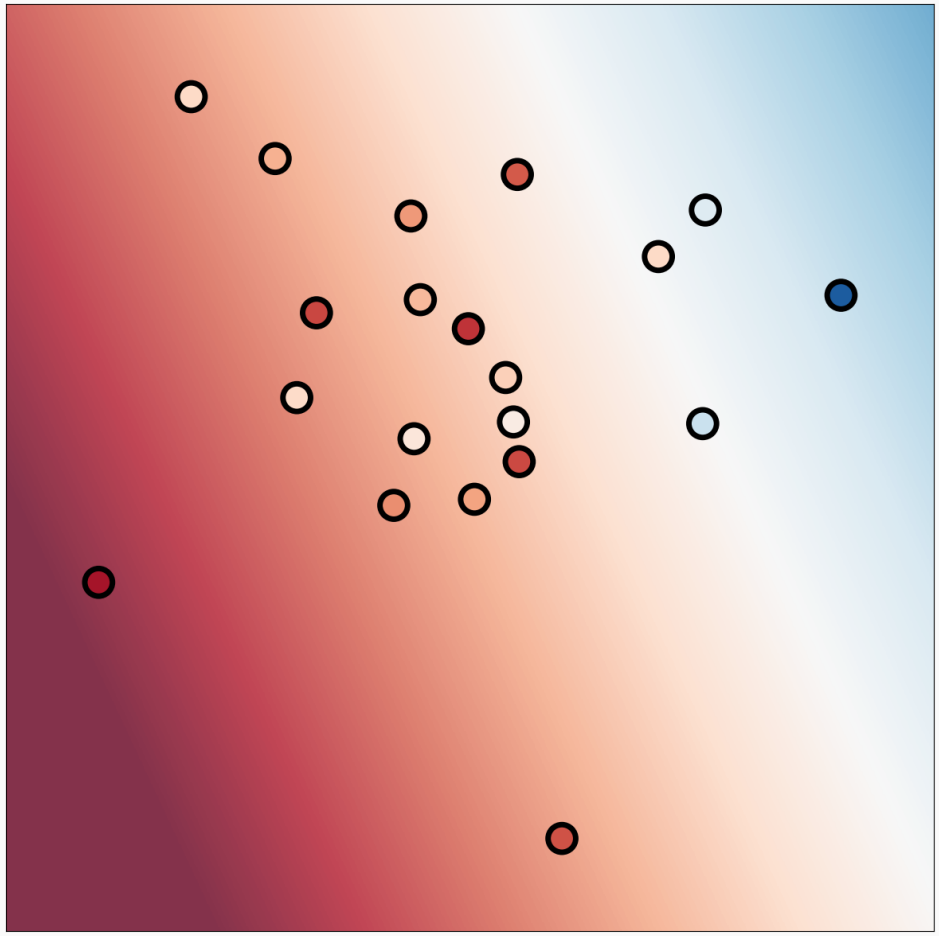




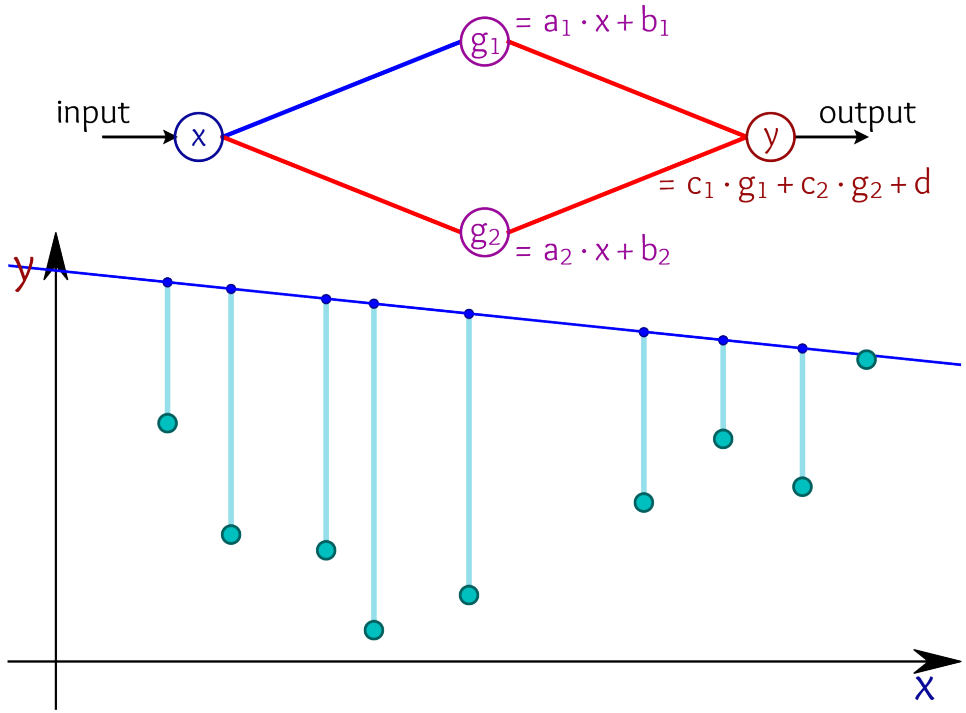
Linear regression



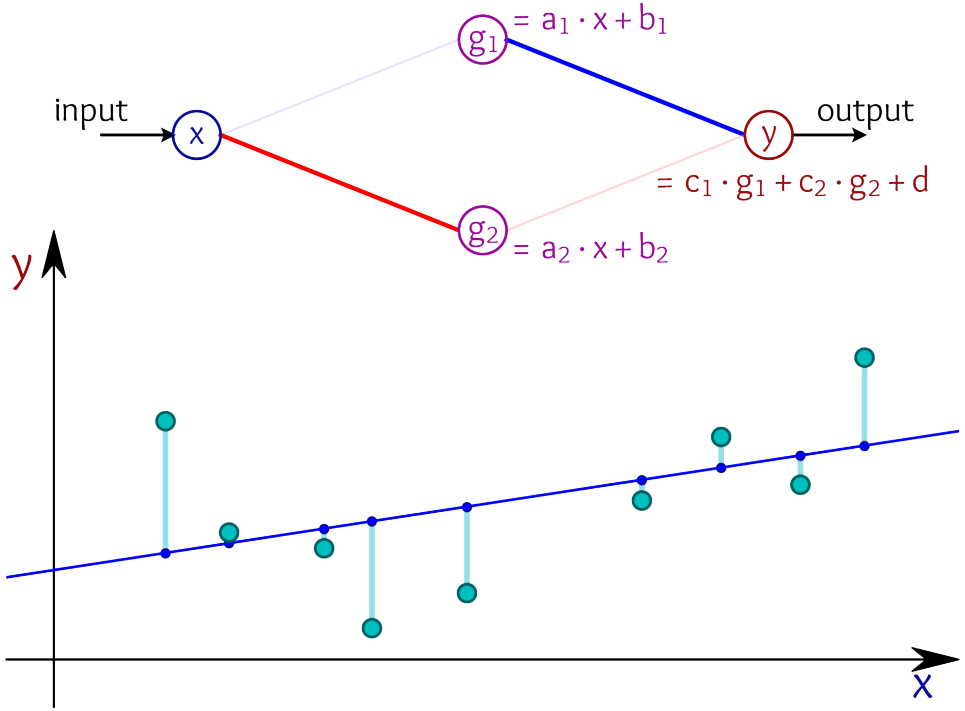
Linear regression models
a **monotonic trend**.
It cannot handle complex relationships
between the input x and the ouput y.



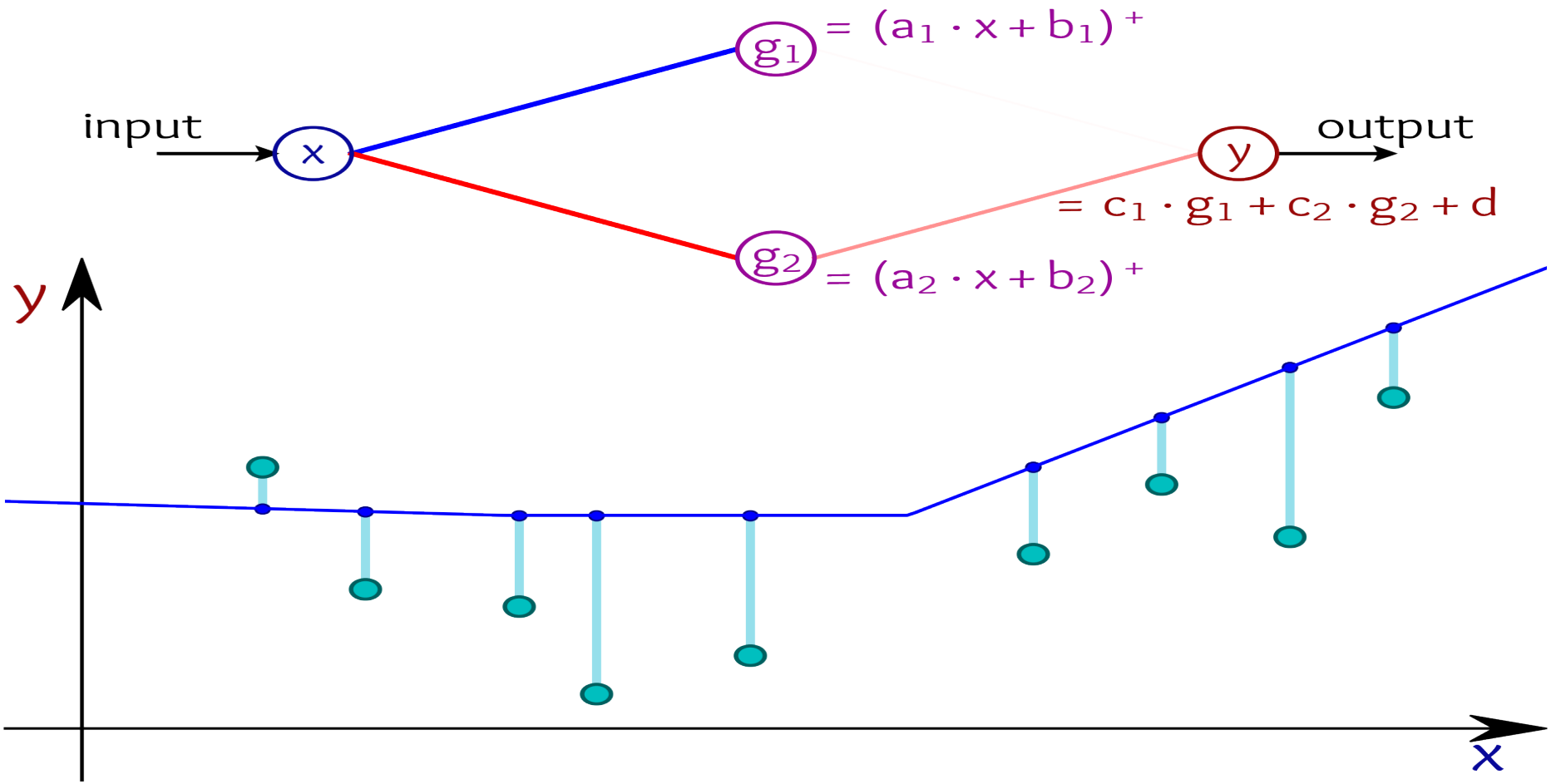
Let's complexify our model with intermediate variables...



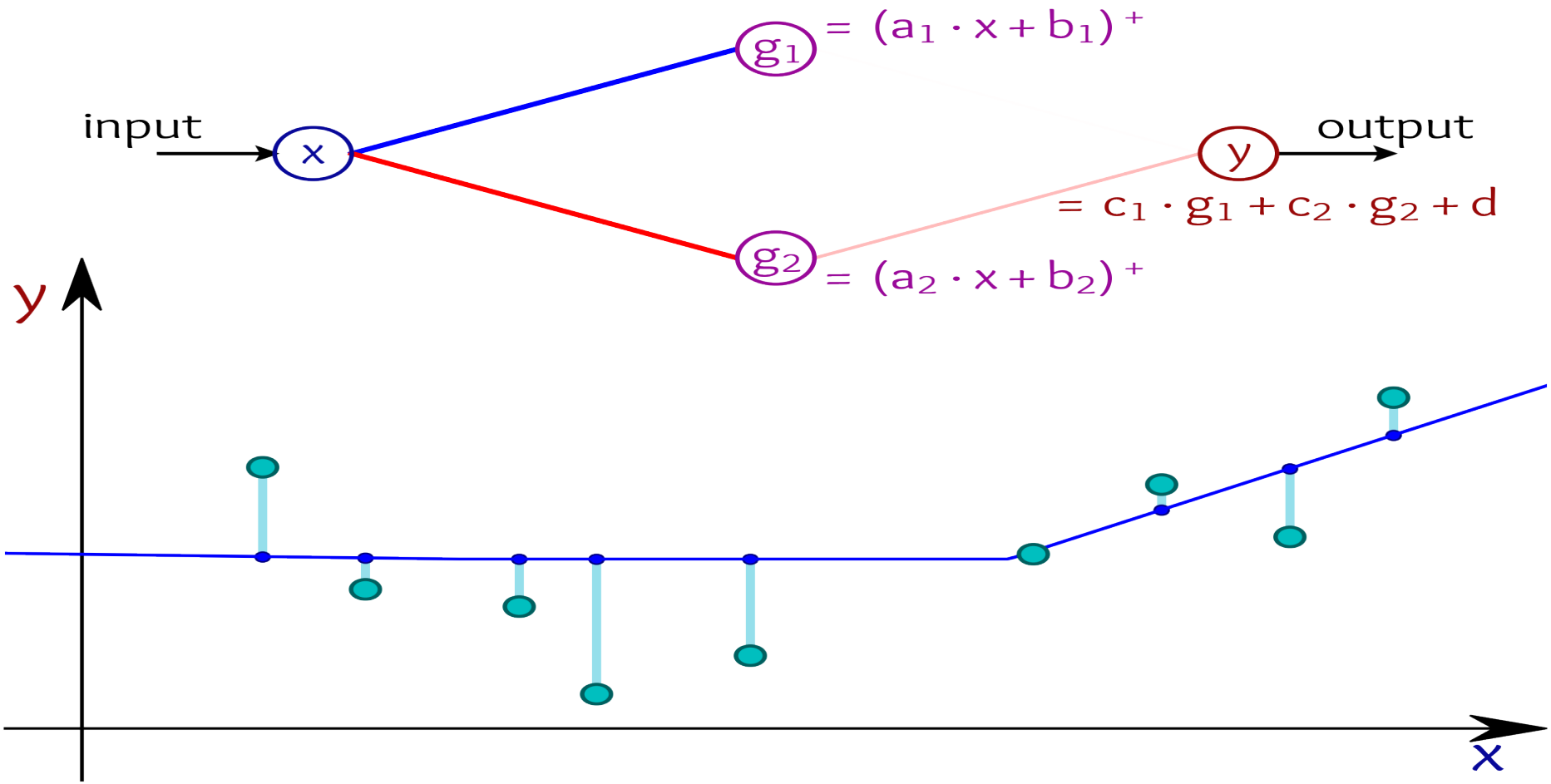
Let's complexify our model with intermediate variables...



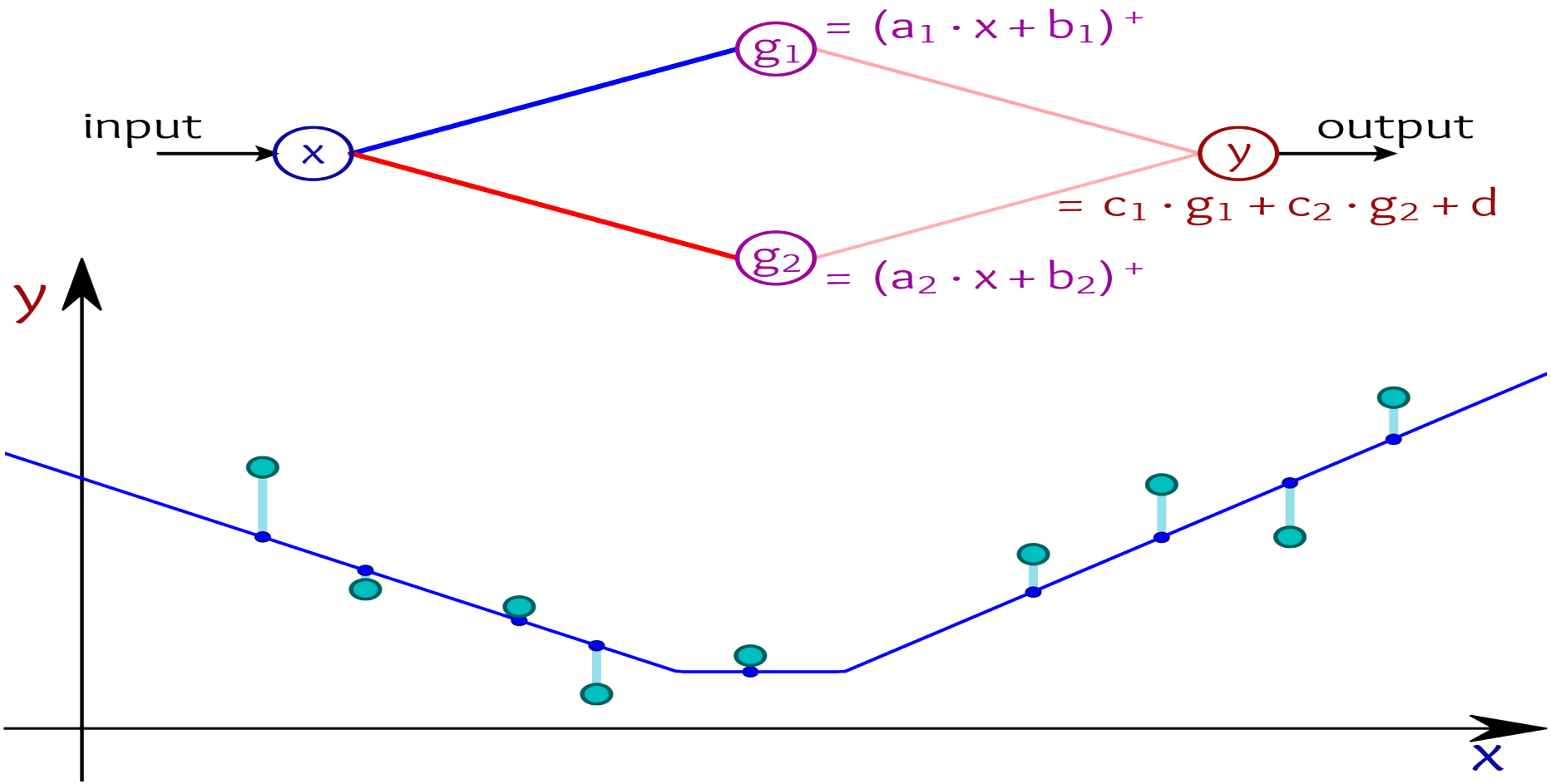
Let's complexify our model with intermediate variables... and non-linearities !



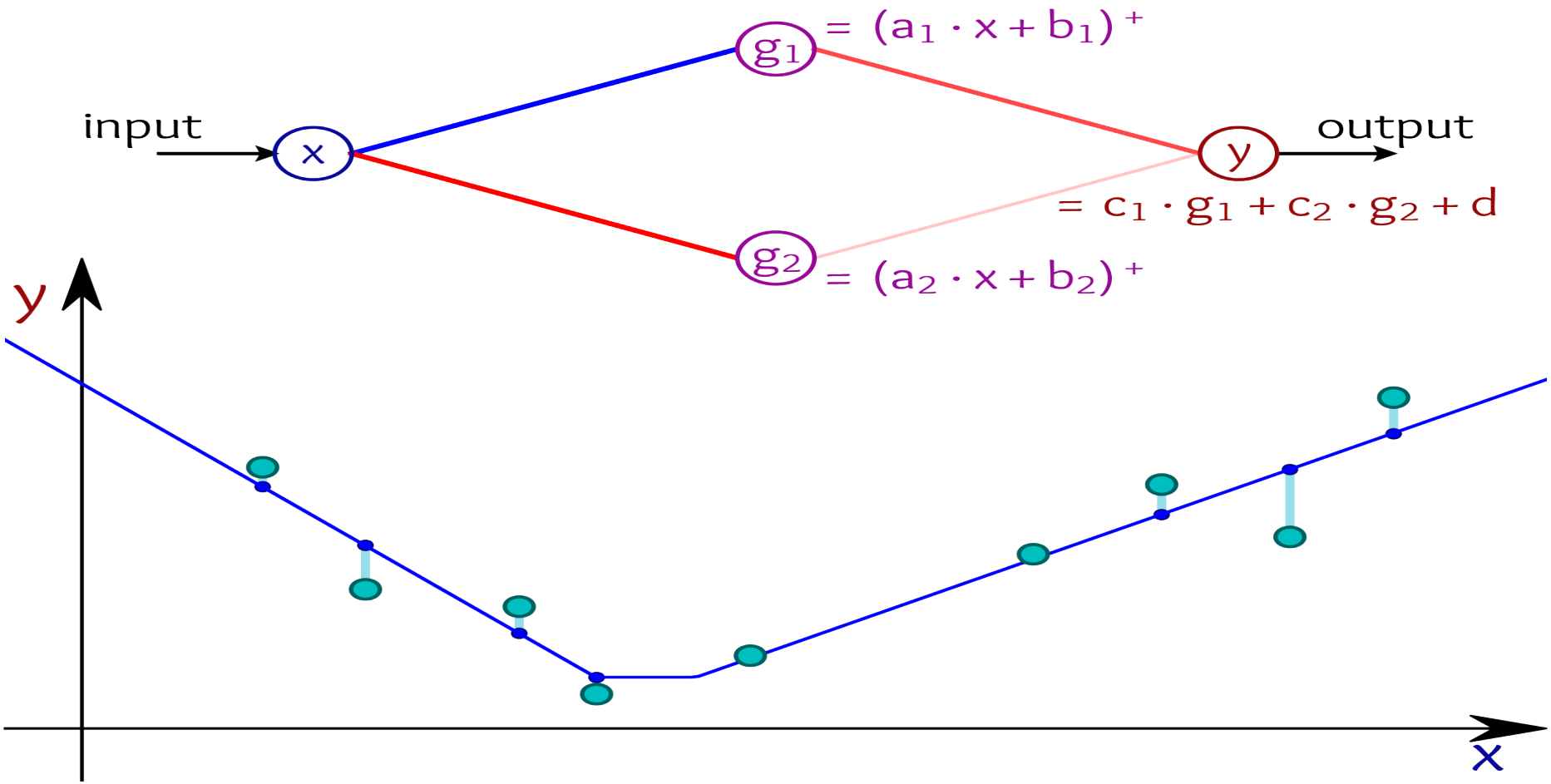
Let's complexify our model with intermediate variables... and non-linearities !



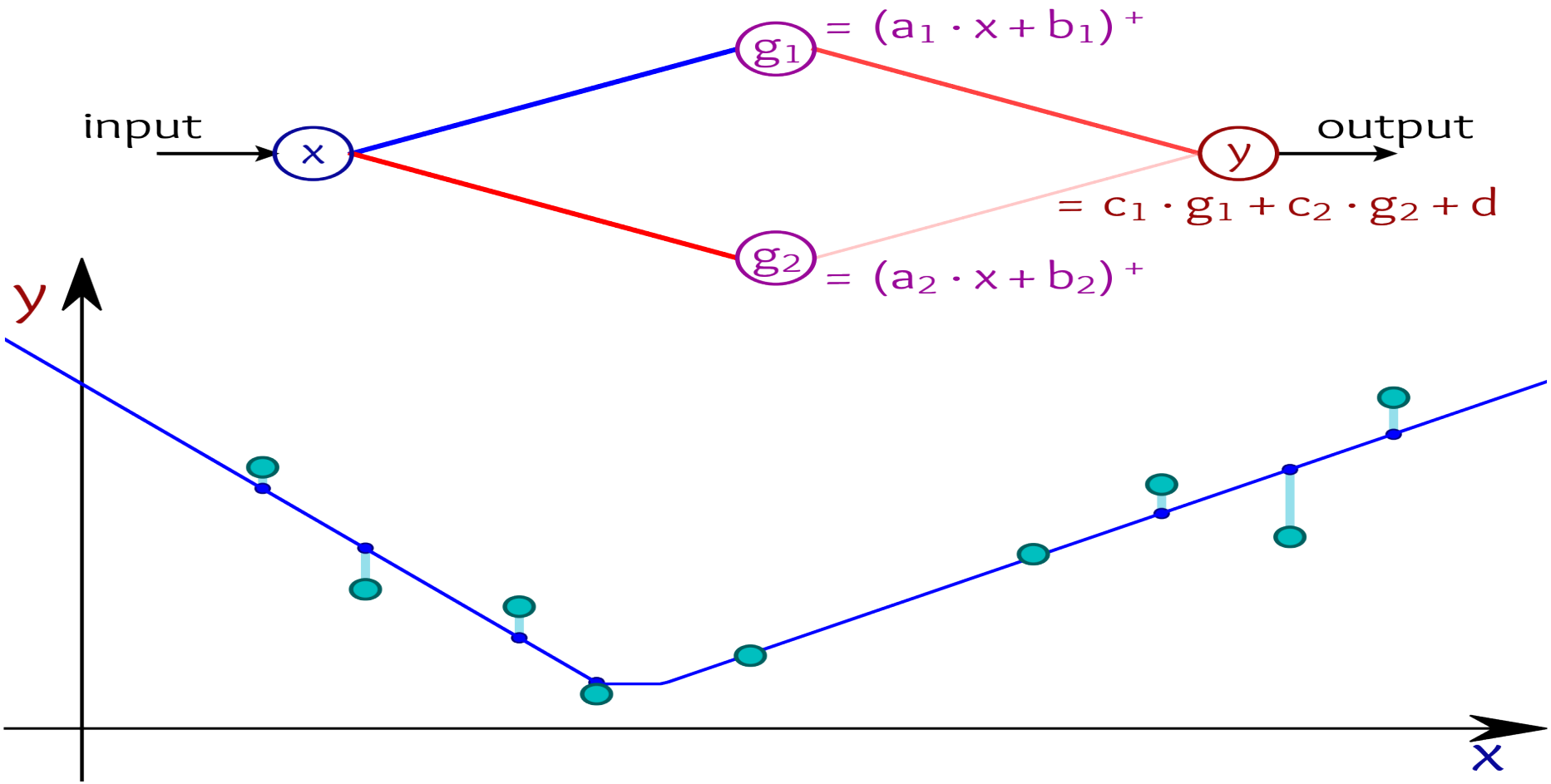
Let's complexify our model with intermediate variables... and non-linearities !



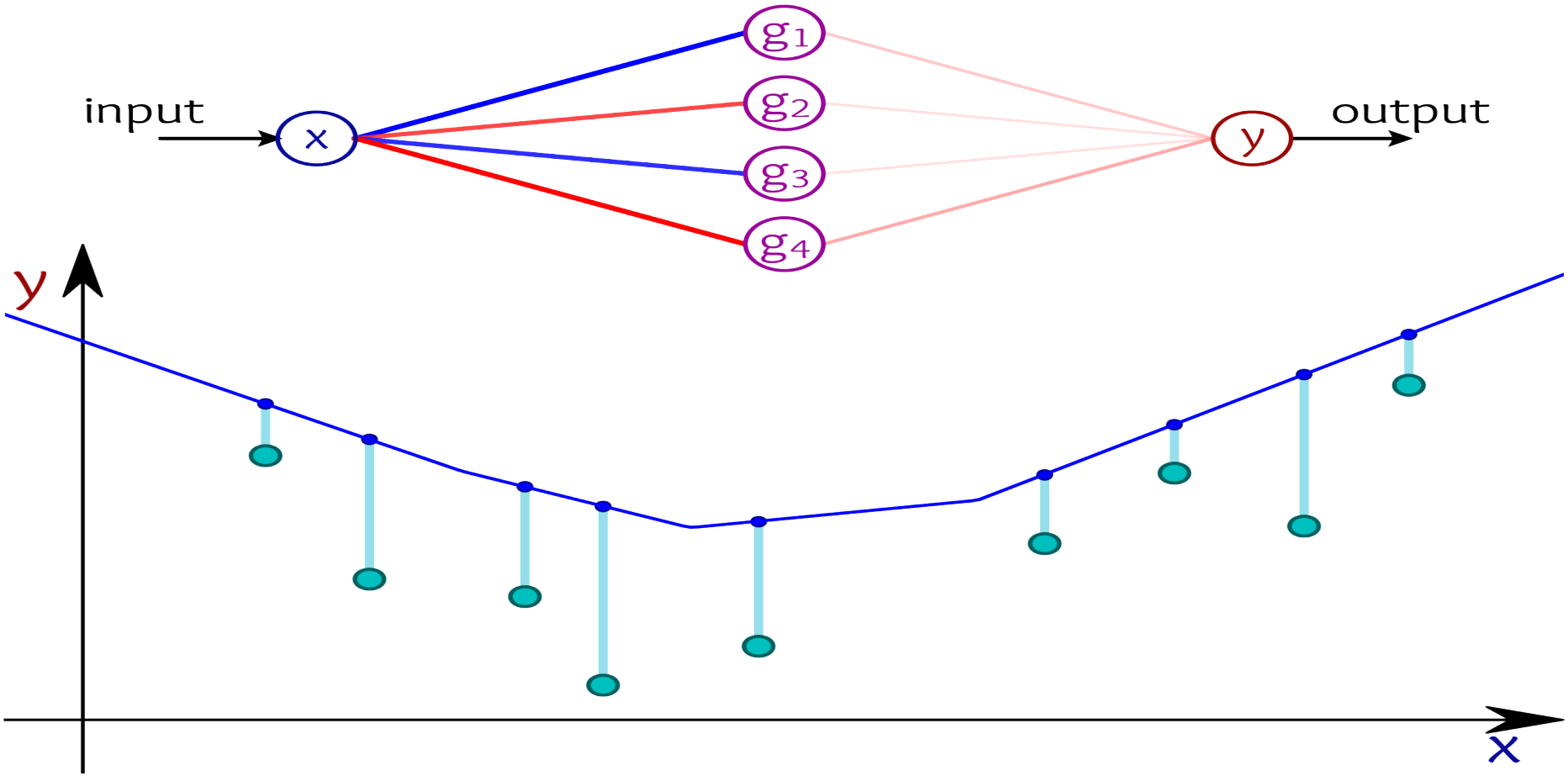
Let's complexify our model with intermediate variables... and non-linearities !



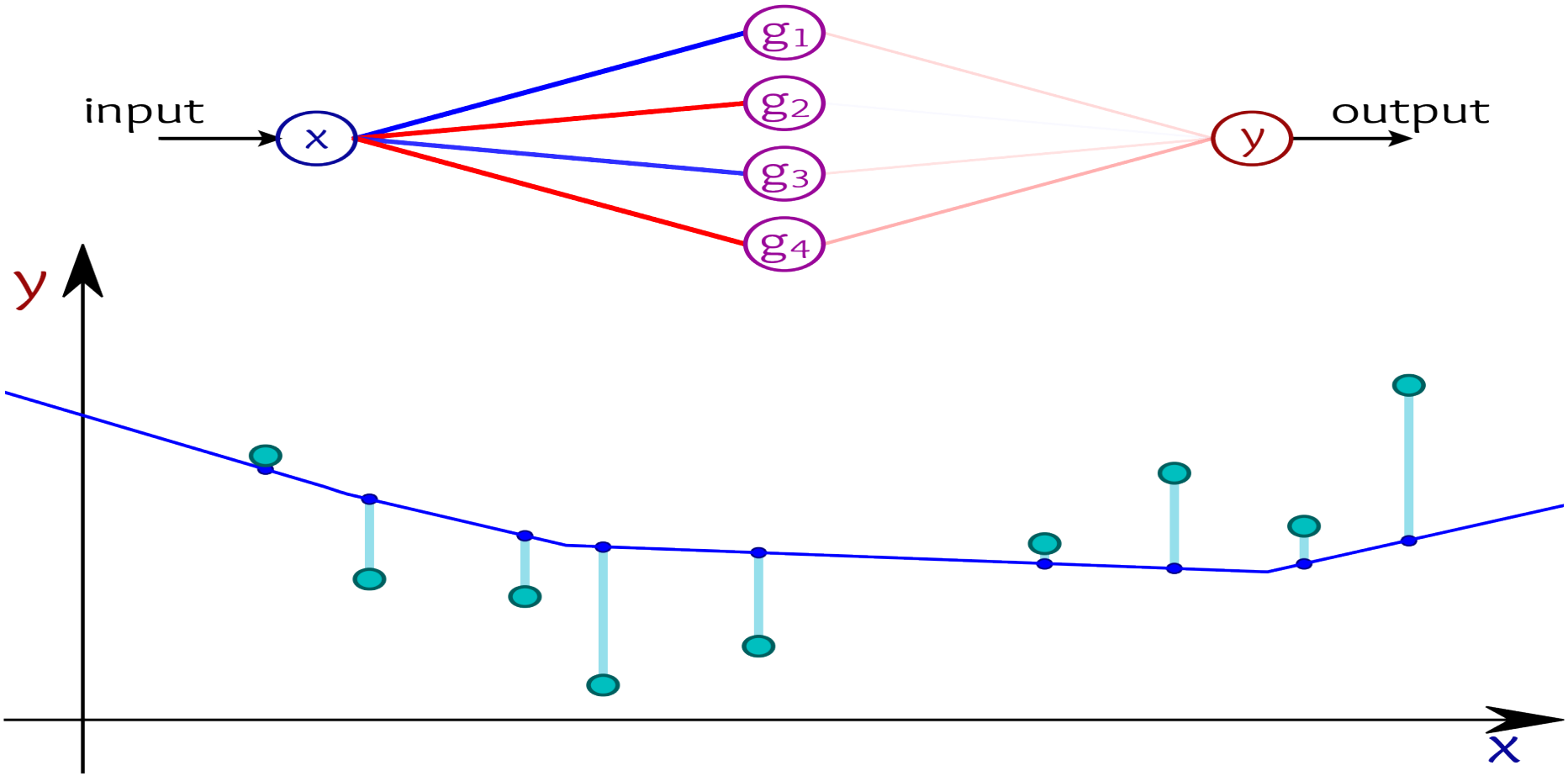
Let's complexify our model with intermediate variables... and non-linearities !



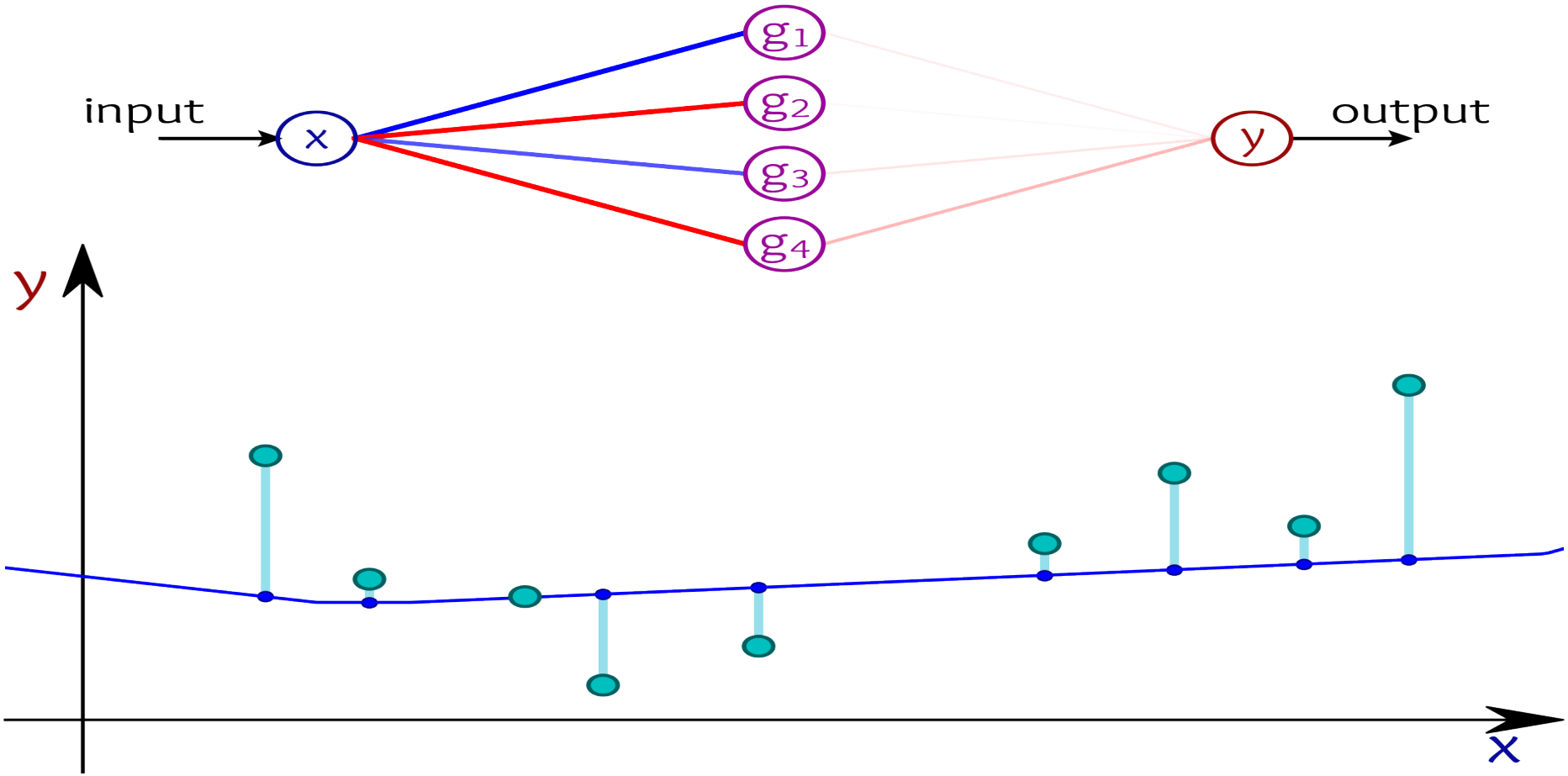
We can then increase the number of “neurons”...



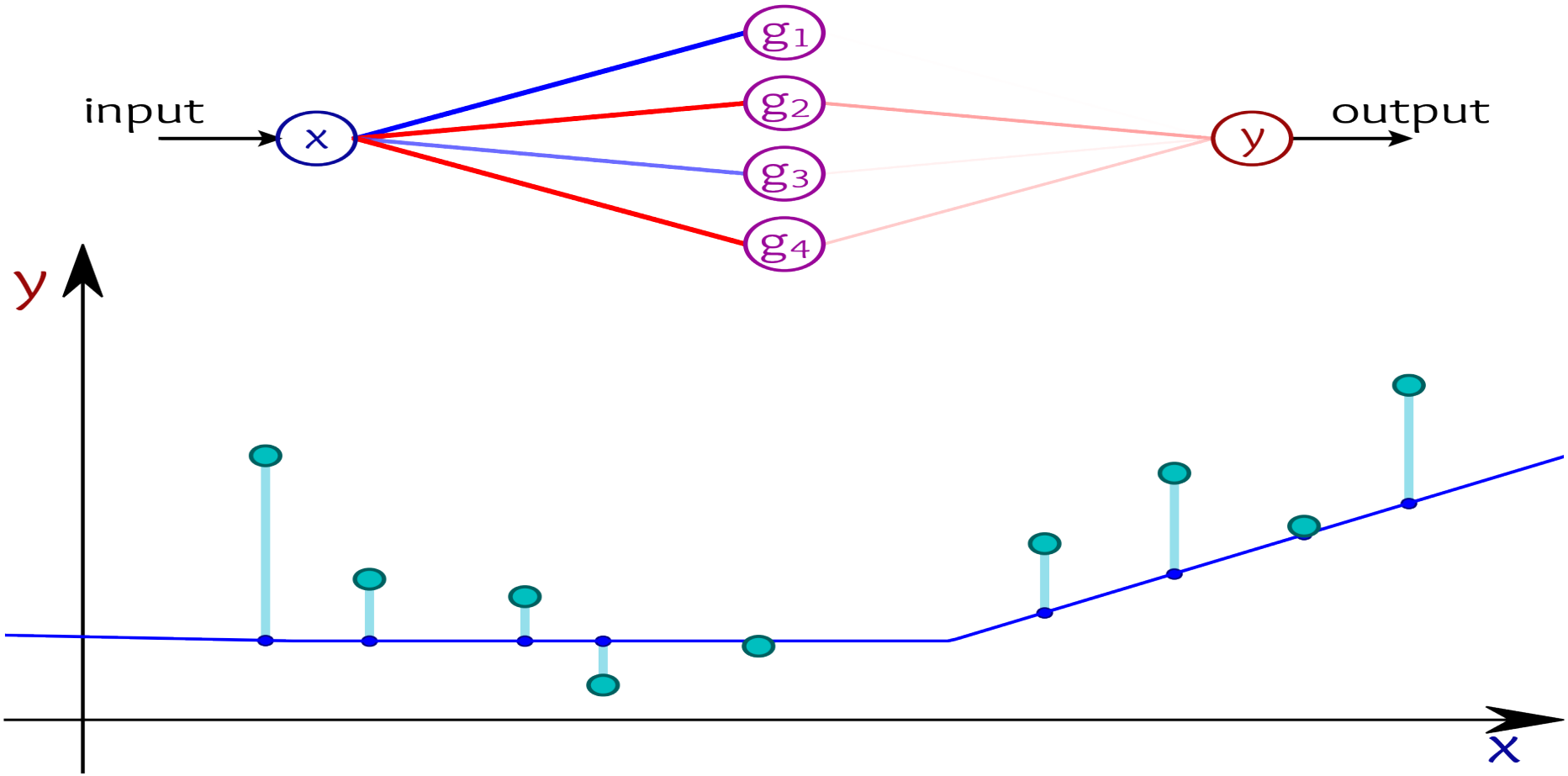
We can then increase the number of “neurons”...



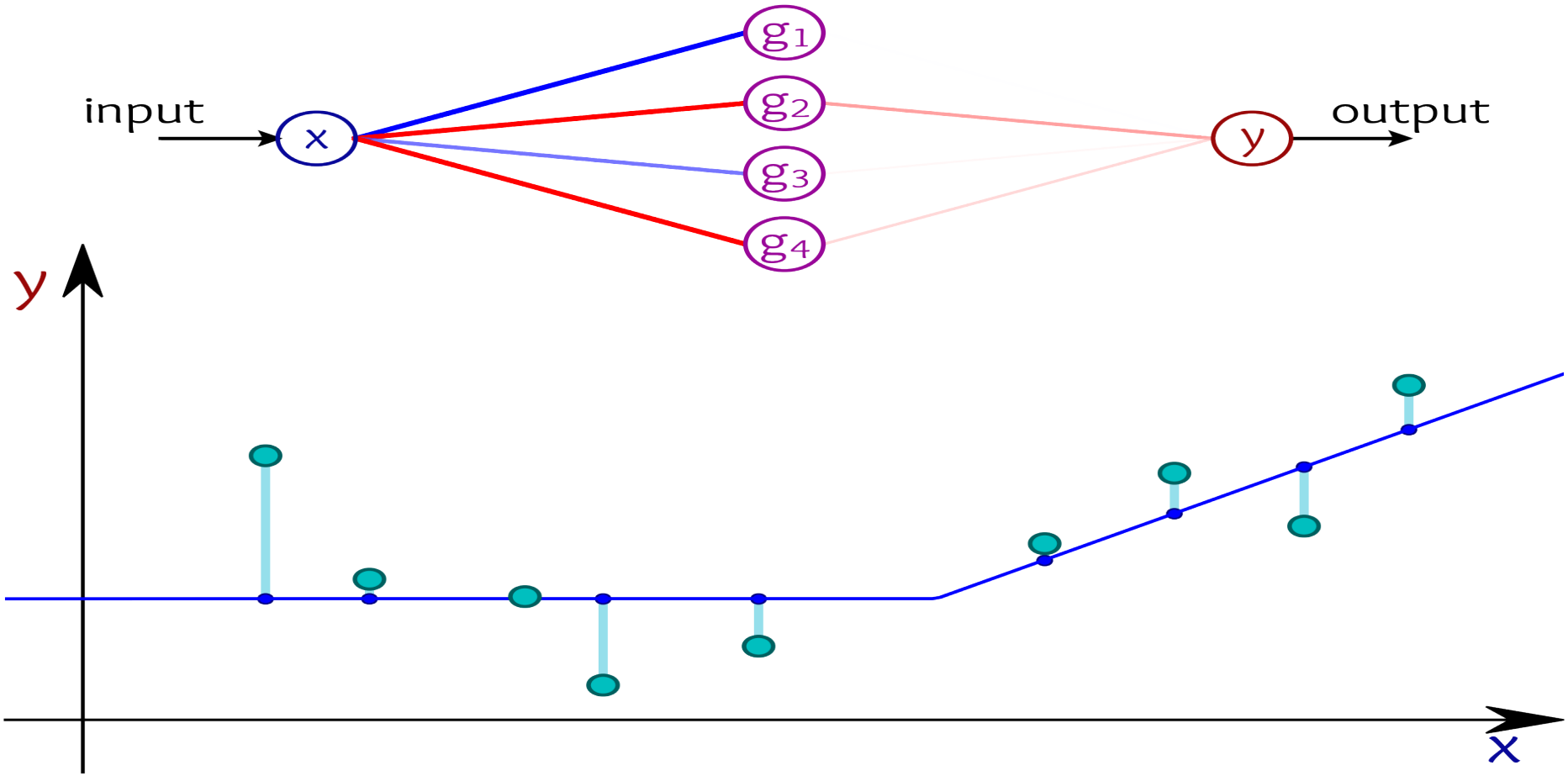
We can then increase the number of “neurons”...



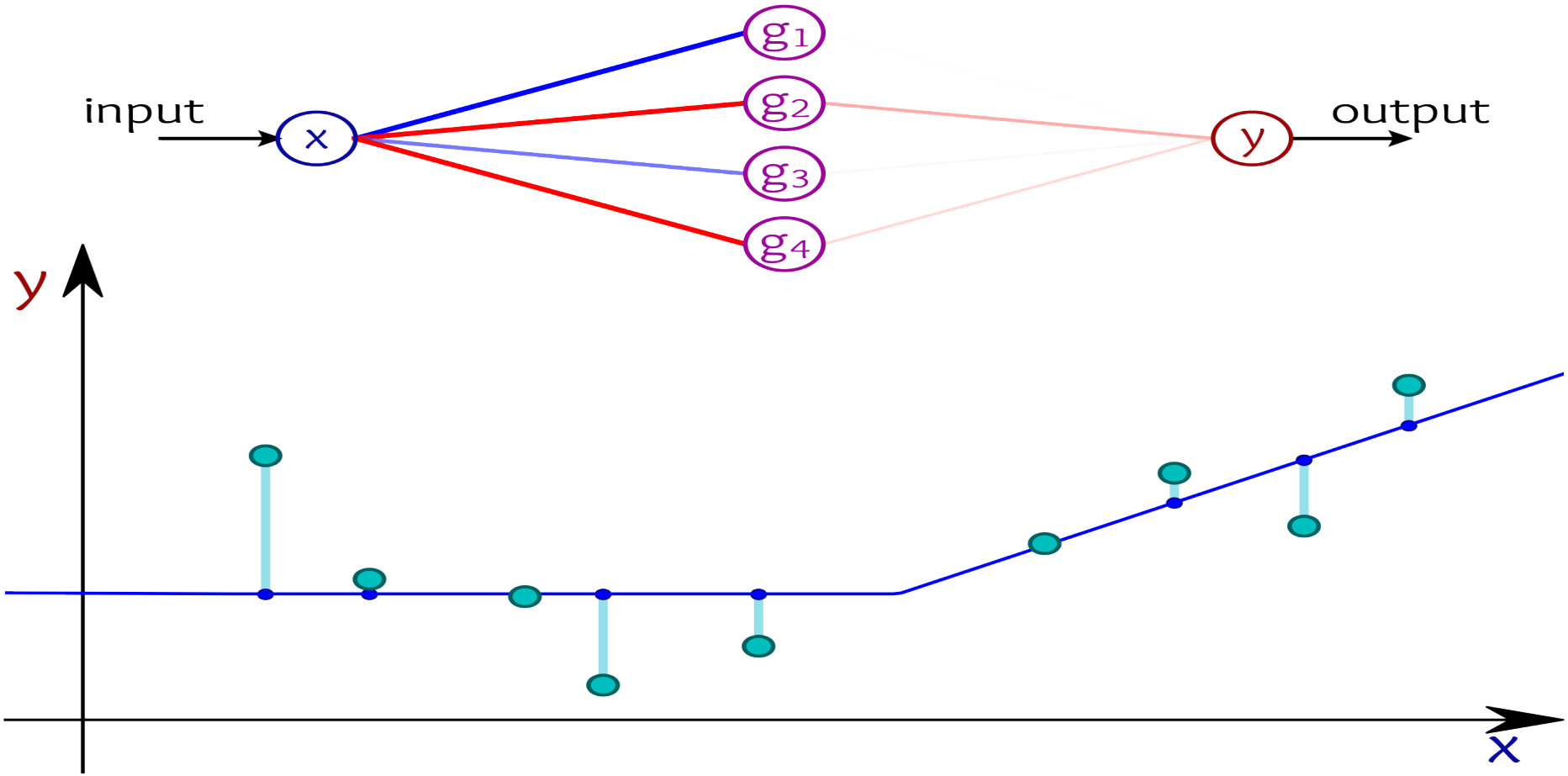
We can then increase the number of “neurons”...



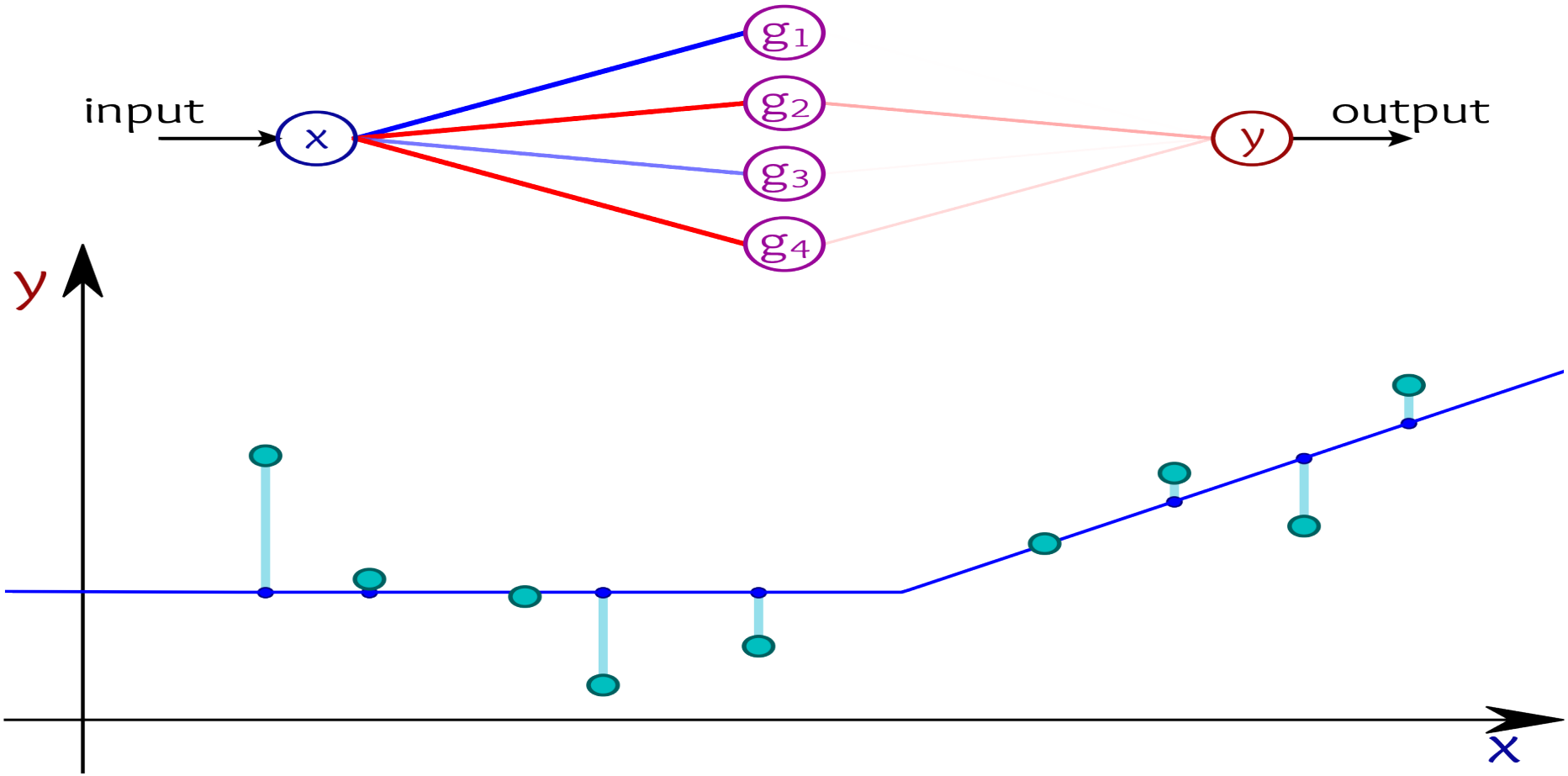
We can then increase the number of “neurons”...



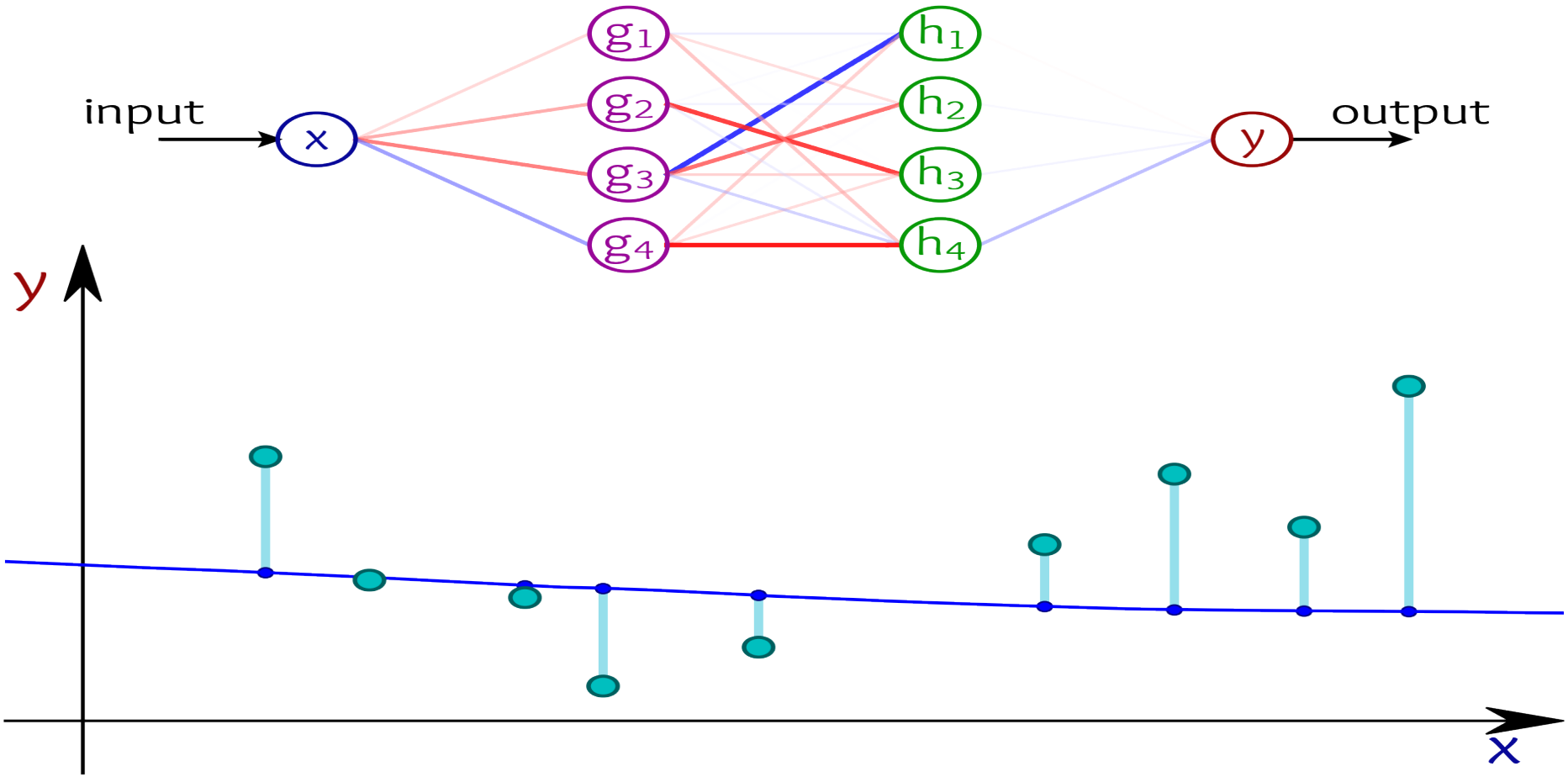
We can then increase the number of "neurons"...



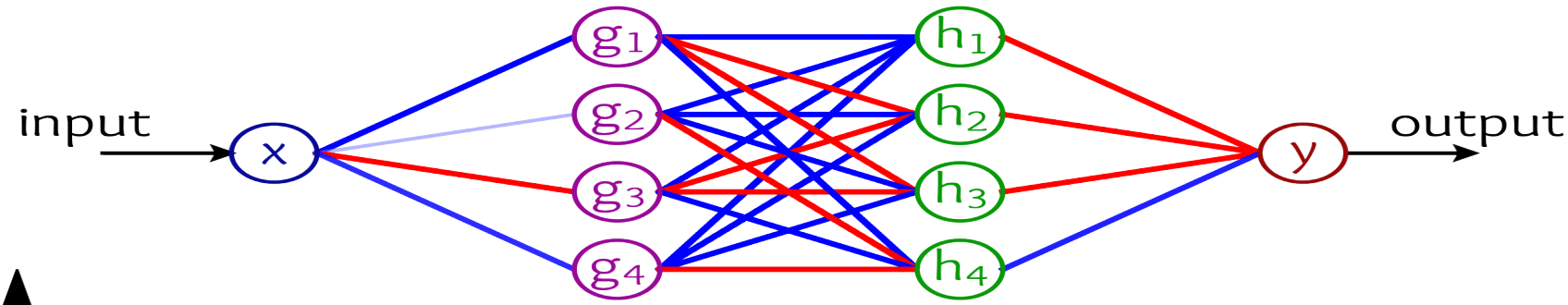
We can then increase the number of “neurons”...

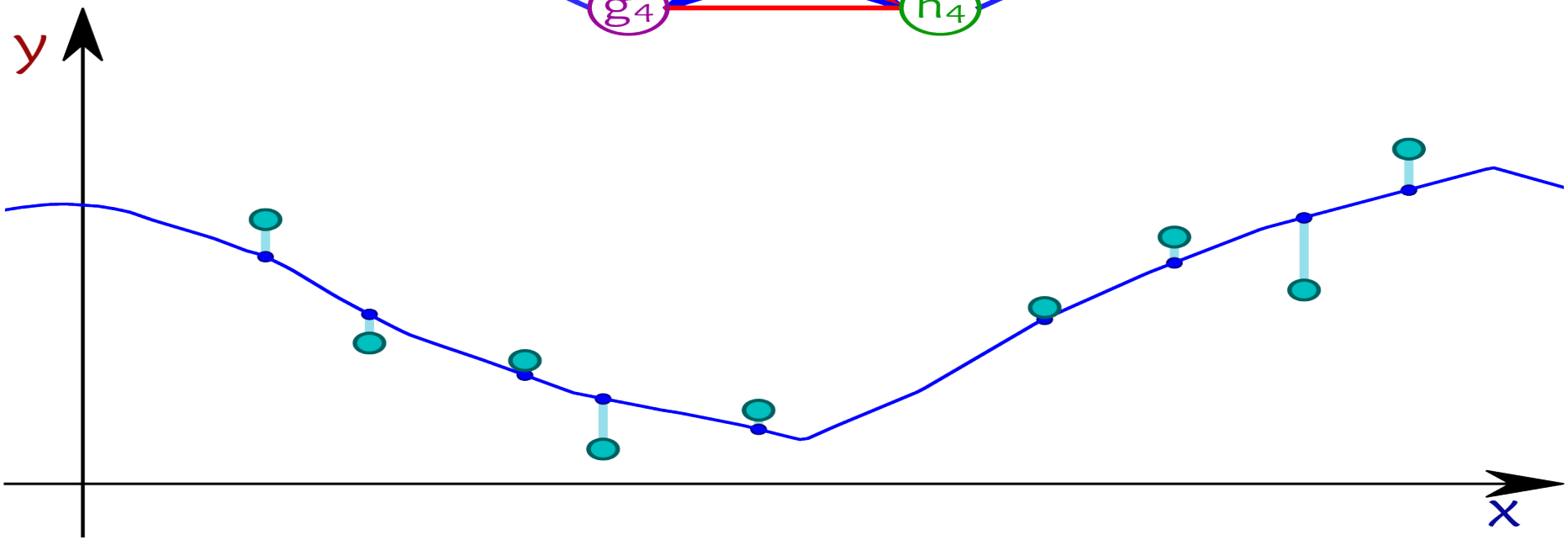


We can then increase the number of "neurons"... and layers !

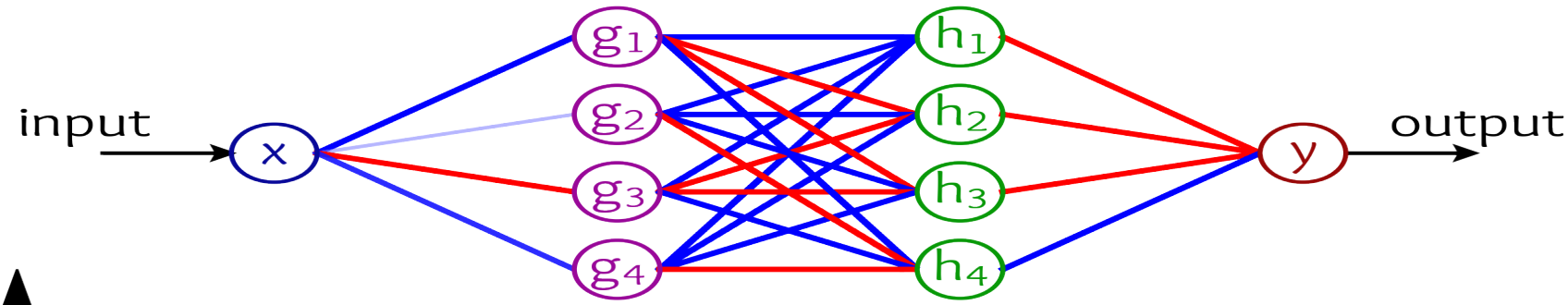


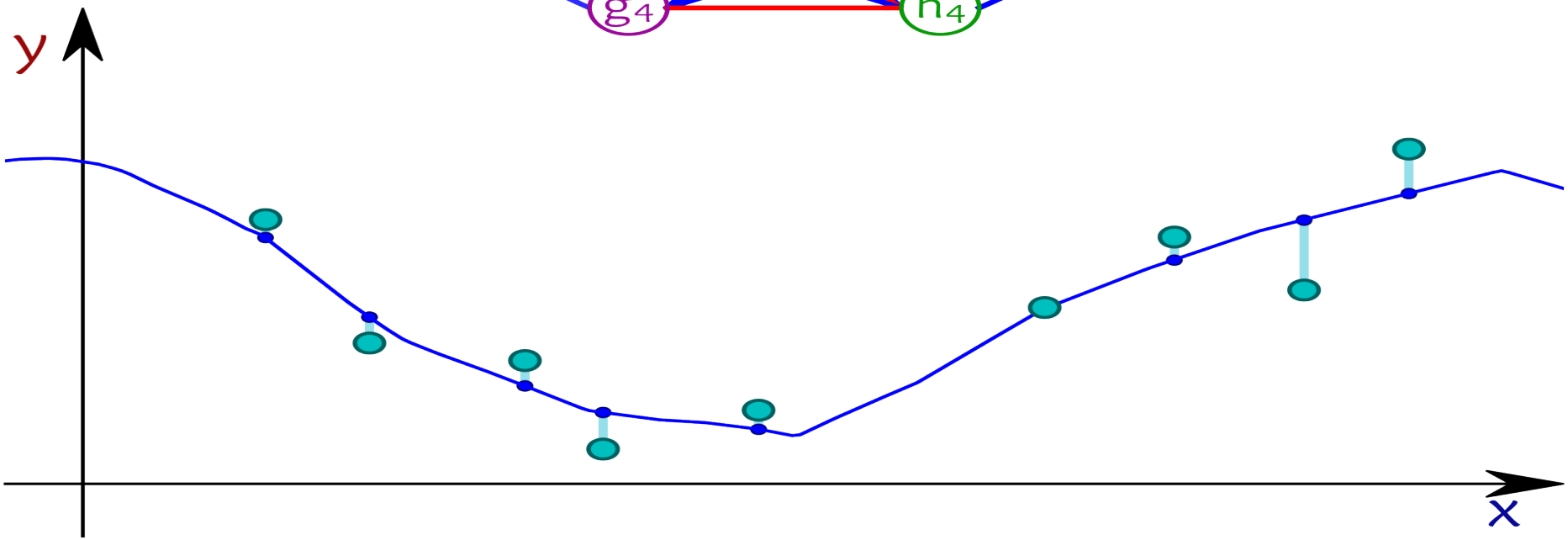
We can then increase the number of "neurons"... and layers !



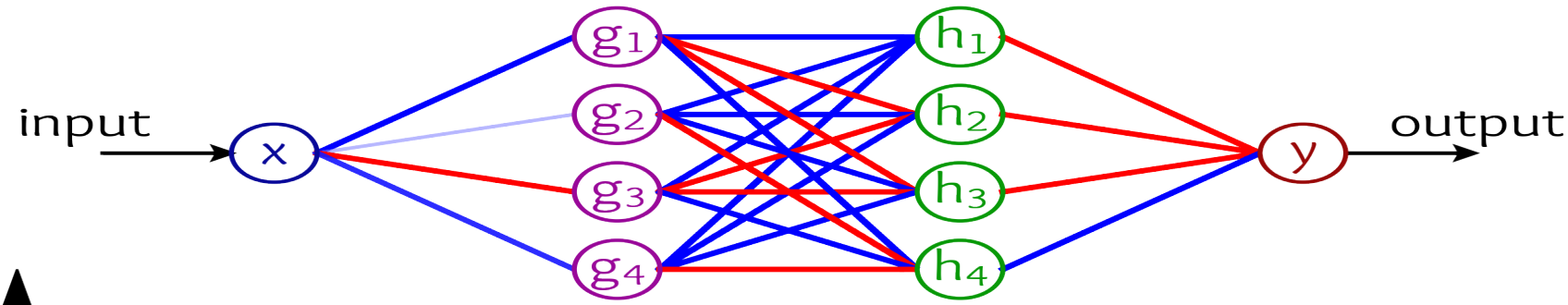


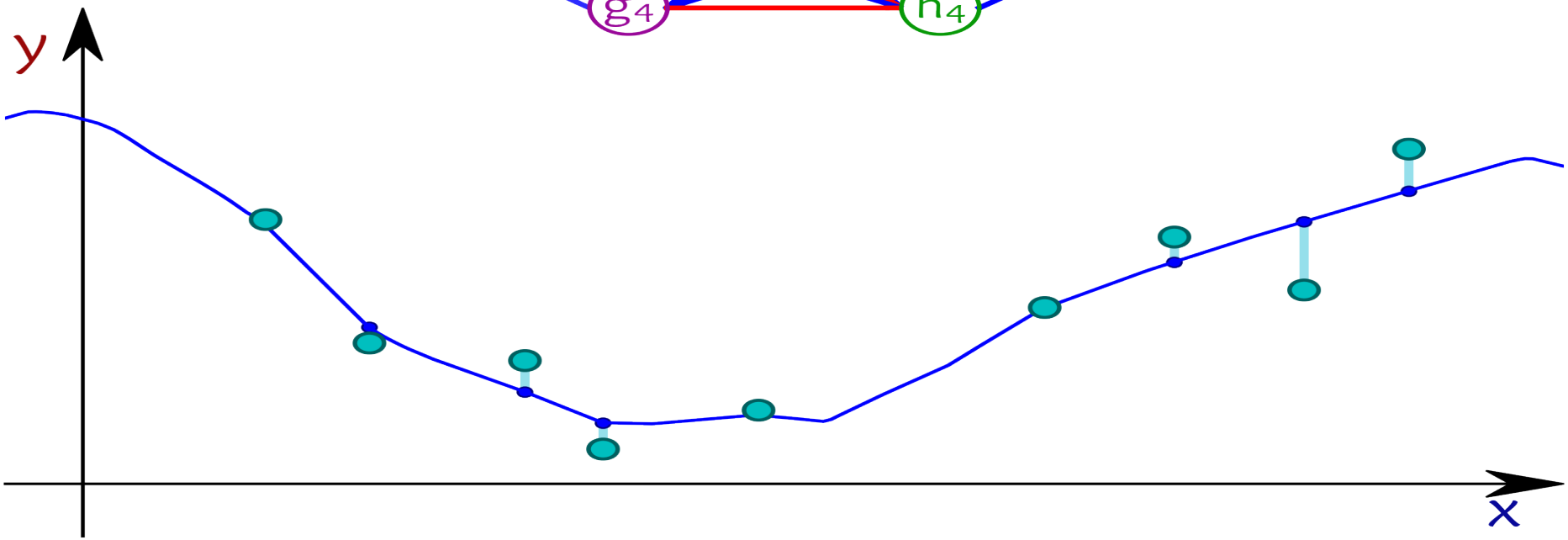
We can then increase the number of "neurons"... and layers !



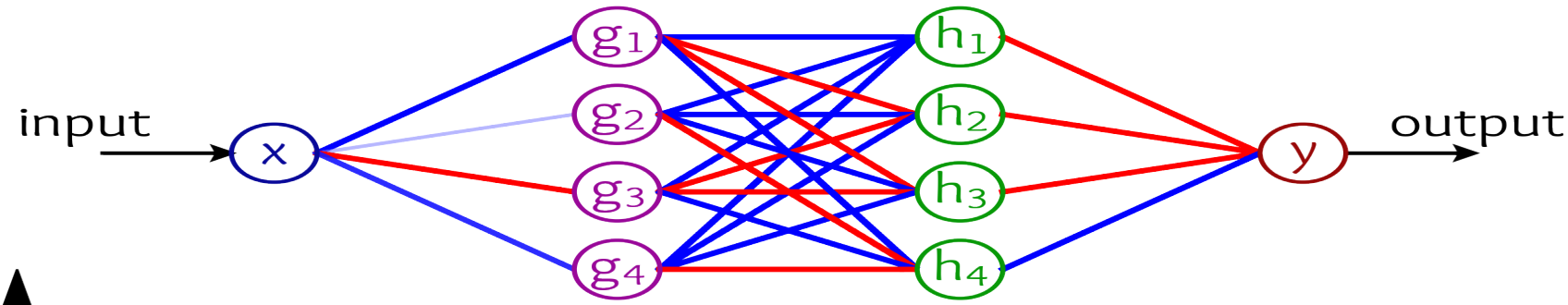


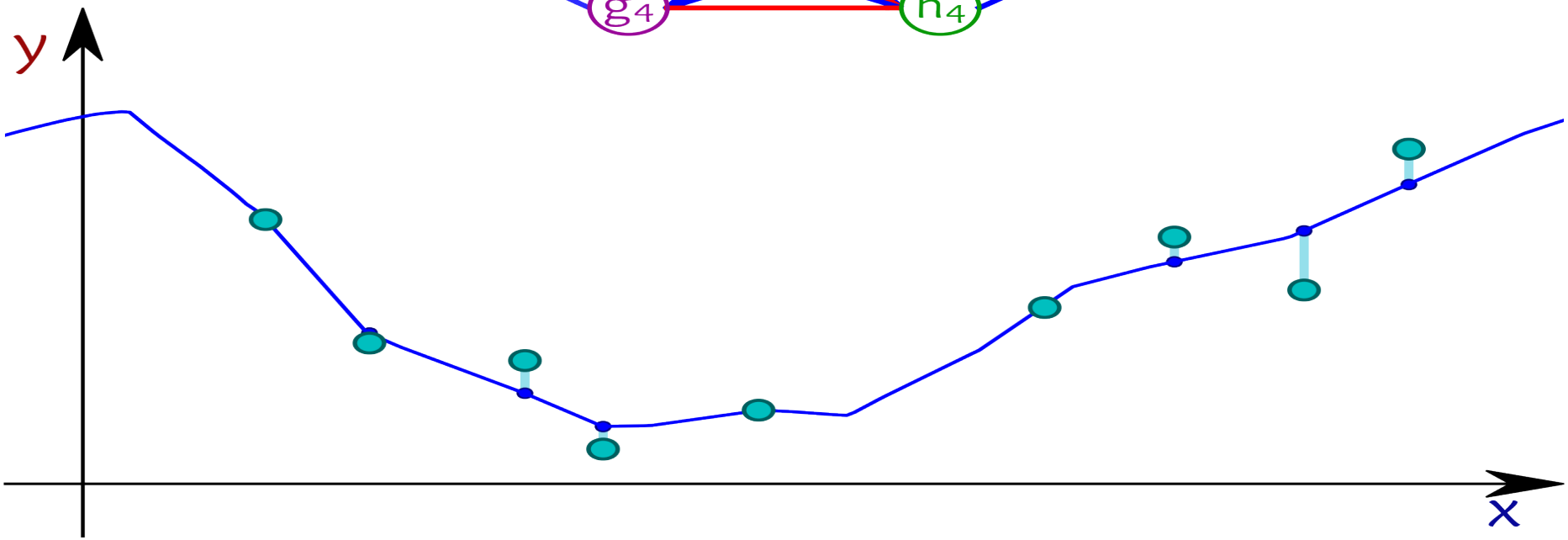
We can then increase the number of "neurons"... and layers !



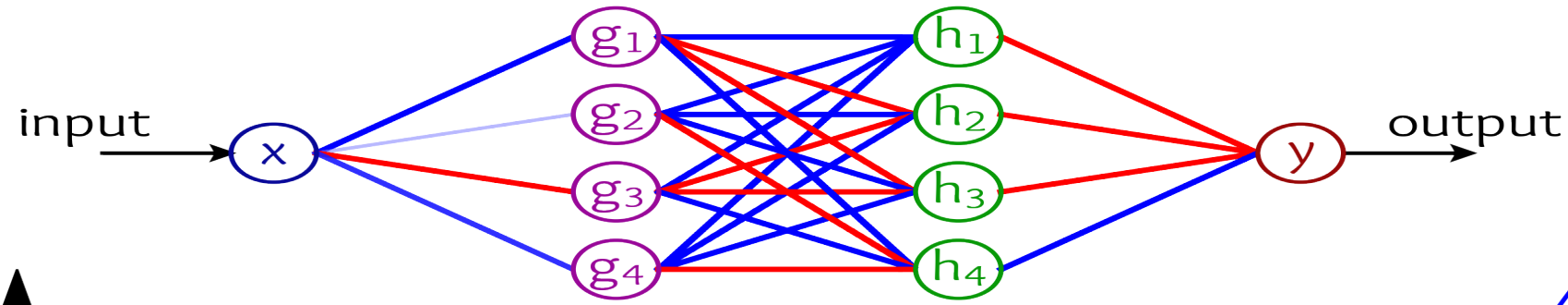


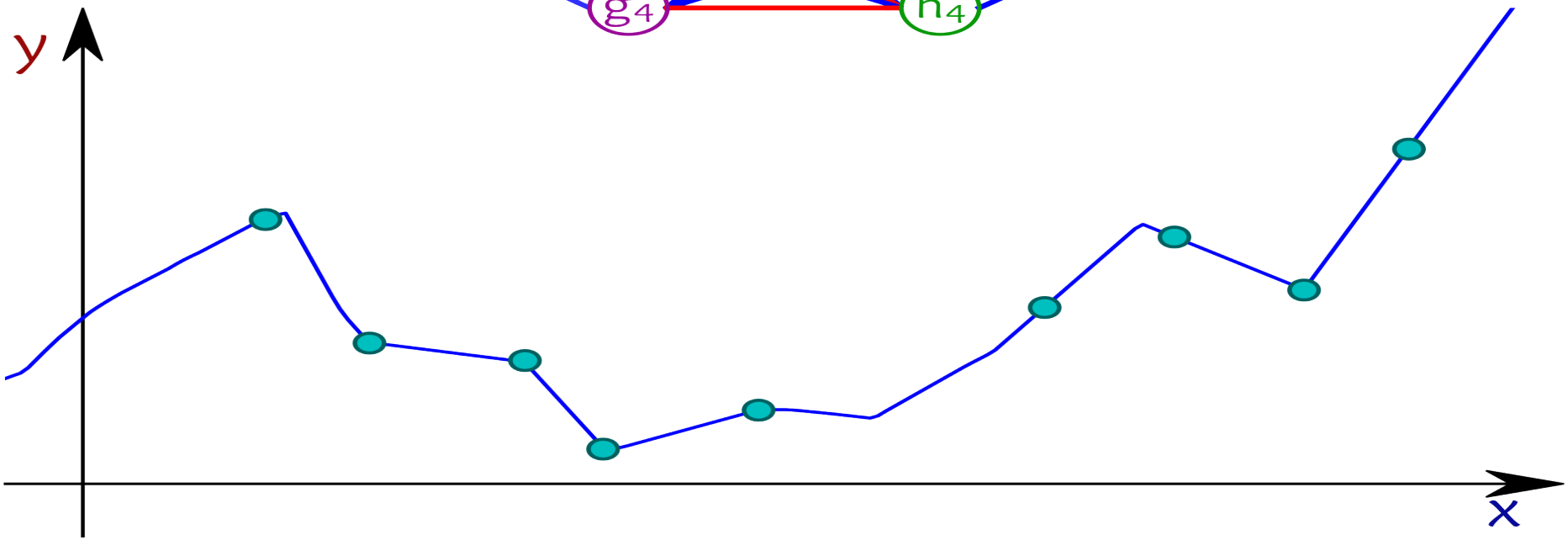
We can then increase the number of "neurons"... and layers !





We can then increase the number of "neurons"... and layers !





(Vanilla, fully connected) neural networks – strengths and weaknesses

- Modular and easy to extend.
- Simplest way of implementing **high-dimensional piecewise linear** models.
- Extremely **well-supported** on CPU and GPU : PyTorch, TensorFlow...

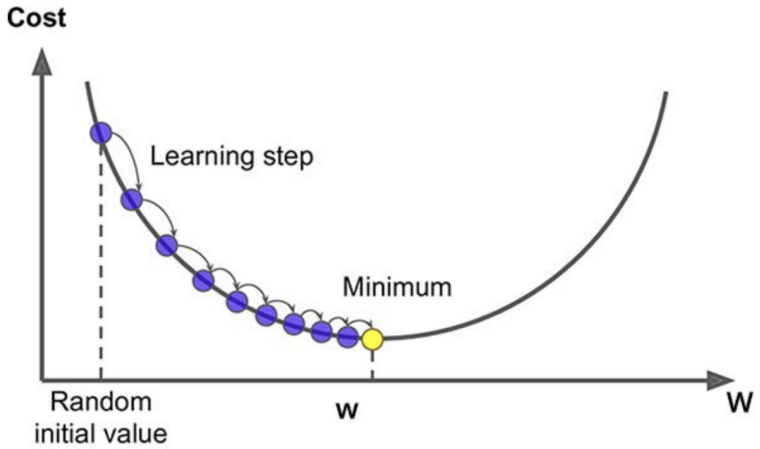
Unfortunately, the optimization of the "neural" weights corresponds to a non-convex optimization problem.

We must rely on **non-deterministic**, stochastic solvers. Performance and smoothness are **not** simply correlated to the number of neurons and layers.

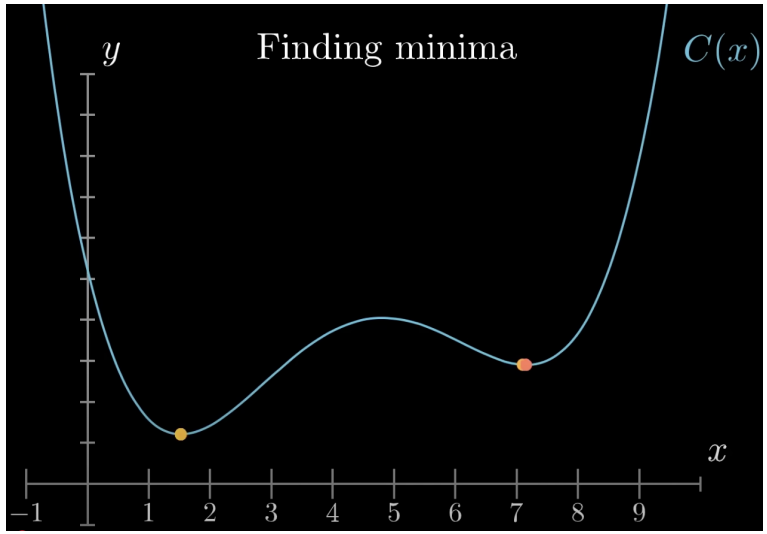
In most applications, this lack of reproducibility and interpretability is a **deal-breaker**.

Let's recap

- In machine learning, the goal is to minimize a loss function (like MSE) to find the optimal model parameters.
- Gradient descent is an iterative optimization algorithm used to find a (local) minimum.
- Key steps in gradient descent :
  1. Compute the gradient of the loss function with respect to parameters.
  2. Update the parameters in the direction of the negative gradient.



Visualization of Gradient Descent. Cost Function vs. Weight (W).
Source: openaccess.uoc.ed



How Neural Networks Learn

- A neural network learns by adjusting its weights (parameters) to minimize an error function.
- neural network $f_\theta : \mathbb{R}^d \to \mathbb{R}^d$ where θ are the learned parameters (weights).
- **Data :** pairs (t_i, y_i), where t_i are time points and y_i are the corresponding observed values of the system.
- **Loss :** A common loss function, the mean squared error (MSE) :

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \|y(t_i) - \hat{y}(t_i)\|^2$$

where $\hat{y}(t_i)$ predicted value, $y(t_i)$ true value.

- **Gradient Descent** :

$$\theta \leftarrow \theta - \eta \frac{\partial L}{\partial \theta}$$

where η learning rate and L loss function.

Gradient Computation for a Simple Neural Network

Network Architecture :

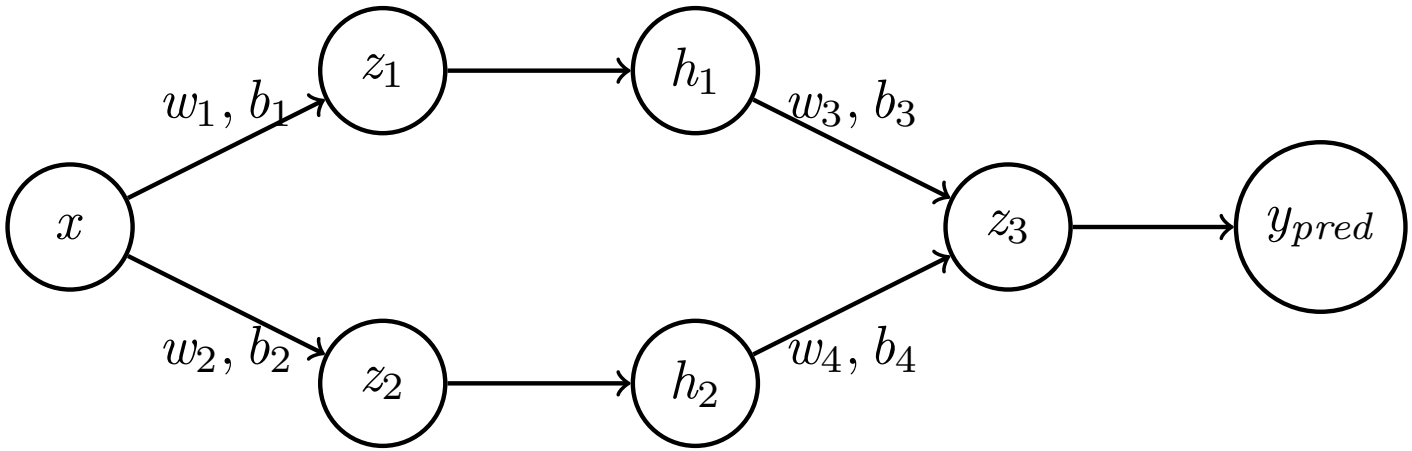
- 1 input node : x 1 hidden layer with 2 neurons : h_1, h_2, 1 output node : y_{pred}
- Activation function : Sigmoid $\sigma(z) = \frac{1}{1+e^{-z}}$

Forward Pass :

$$z_1 = w_1 x + b_1, \quad h_1 = \sigma(z_1)$$
$$z_2 = w_2 x + b_2, \quad h_2 = \sigma(z_2)$$
$$z_3 = w_3 h_1 + w_4 h_2 + b_3 + b_4, \quad y_{pred} = \sigma(z_3)$$

Loss Function : $L = \frac{1}{2}(y - y_{pred})^2$

Network Graph :



Backward Pass : Compute Gradients

Gradient Computation using Chain Rule

Complete Function Composition :

- The neural network can be written as a nested function :

$$
\begin{aligned}
y_{pred} &= \sigma(z_3), \text{ where } z_3 = w_3 h_1 + w_4 h_2 + b_3 \\
h_1 &= \sigma(z_1), \text{ where } z_1 = w_1 x + b_1 \\
h_2 &= \sigma(z_2), \text{ where } z_2 = w_2 x + b_2 \\
y_{pred} &= \sigma(w_3 \sigma(w_1 x + b_1) + w_4 \sigma(w_2 x + b_2) + b_3)
\end{aligned}
$$

Applying Chain Rule for Weight Gradients :

$$
\begin{aligned}
\frac{\partial L}{\partial w_3} &= \frac{\partial L}{\partial y_{pred}} \cdot \frac{\partial y_{pred}}{\partial z_3} \cdot \frac{\partial z_3}{\partial w_3} \\
\frac{\partial L}{\partial w_1} &= \frac{\partial L}{\partial y_{pred}} \cdot \frac{\partial y_{pred}}{\partial z_3} \cdot \frac{\partial z_3}{\partial h_1} \cdot \frac{\partial h_1}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_1}
\end{aligned}
$$

General Rule :

- To compute gradients, propagate from output to input, multiplying derivatives at each step.

Applying Chain Rule for Weight Gradients :

$$\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial y_{pred}} \cdot \frac{\partial y_{pred}}{\partial z_3} \cdot \frac{\partial z_3}{\partial w_3}$$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_{pred}} \cdot \frac{\partial y_{pred}}{\partial z_3} \cdot \frac{\partial z_3}{\partial h_1} \cdot \frac{\partial h_1}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_1}$$

General Rule :

- To compute gradients, propagate from output to input, multiplying derivatives at each step.

Backward Pass : Compute Gradients

$$\frac{\partial L}{\partial y_{pred}} = (y_{pred} - y)$$

$$\frac{\partial y_{pred}}{\partial z_3} = y_{pred}(1 - y_{pred})$$

$$\frac{\partial z_3}{\partial w_3} = h_1$$

$$\frac{\partial L}{\partial w_3} = (y_{pred} - y) y_{pred}(1 - y_{pred}) h_1$$

The Importance of Automatic Differentiation (AutoDiff)

Why is AutoDiff crucial ?

- Neural networks involve millions of parameters—manual differentiation is impractical.
- AutoDiff efficiently computes **exact gradients** without symbolic complexity.
- Enables modern deep learning frameworks like **TensorFlow, PyTorch, and JAX**.

Example : Given $f(x) = x^2 + \sin x$,

- **Symbolic Differentiation :** $f'(x) = 2x + \cos x$ (hardcoded).
- **Numerical Differentiation :** Approximation using finite differences.
- **AutoDiff :** Uses computational graphs to efficiently compute derivatives.

A quick reminder : Euler's Method as a numerical solver

$$y(0) = y_0, \quad \frac{dy}{dt}(t) = f(t, y(t))$$

Definition :

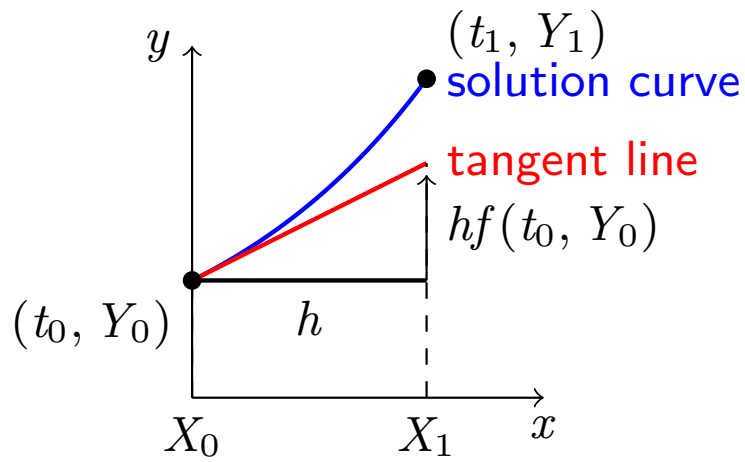
- A simple numerical method to solve first-order ODEs.
- Approximates the solution using **small time steps**.
- Based on the **tangent line** at each step.

Formula :

$$y_{n+1} = y_n + hf(t_n, y_n) \quad \textbf{Exemple with } y(t) = 0.5 * e^t$$

where :

- y_n is the current solution value.
- h is the step size.
- $f(t_n, y_n)$ is the derivative (from the ODE).



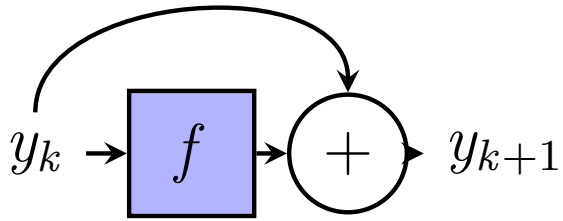
RNNs and Euler's Method : A Connection

Both **Euler's method and **Recurrent Neural Networks (RNNs)** update a state iteratively over time.**

Euler's method :

$$y_{t+1} = y_t + hf(y_t, t)$$

RNNs updates :

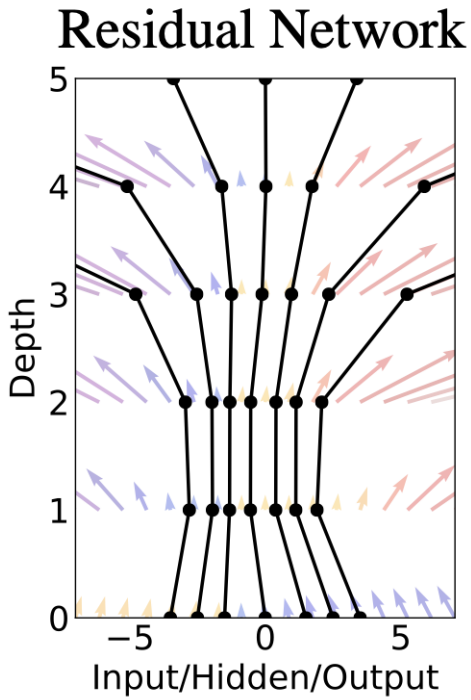


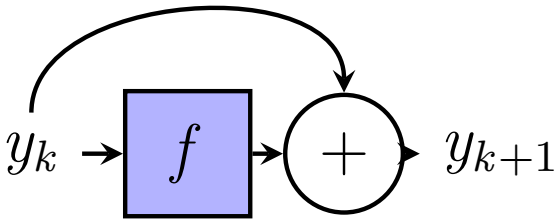
$$y_{k+1} = y_k + f(y_k)$$

Why This Matters :

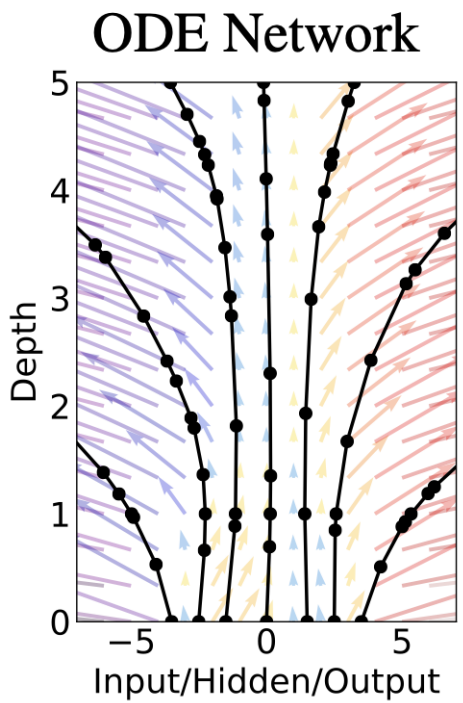
- RNNs can be seen as a discrete approximation to continuous-time dynamical systems.
- This leads to connections between **Neural ODEs** and **Continuous-time RNNs**.

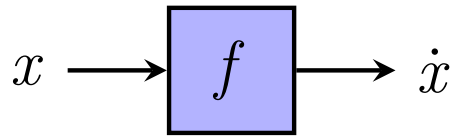
ODE Network vs Recurrent Network





$$y_{k+1} = y_k + f(y_k)$$





$$\dot{x} = f(x)$$

How to train a Neural-ODE

$$\dot{x} = f(x) \quad \Rightarrow \quad x(t_0 + \Delta t) = x(t_0) + \int_{t_0}^{t_0+\Delta t} f(x(\tau), \tau)$$

- Learn f_θ such that $\dot{x} = f_\theta(x)$.
- $\mathcal{L}(x(t_0) + \int_{t_0}^{t_0+\Delta t} f_\theta(x(\tau), \tau)\, d\tau) = \mathcal{L}(ODESolver(x(t_0), f, \theta, t_0, t_1)$
- How to compute $\frac{\partial \mathcal{L}}{\partial \theta}$?
- We cannot use auto-diff on an ODESolver directly.
- There exist a way (adjoint method) to use auto-diff on ODESolver anyway)

The Adjoint Method

- Let $a(t) = \frac{\partial \mathcal{L}}{\partial x(t)}$ at each instant this compute how $x(t)$ influence L
- Then $\frac{da(t)}{dt} = -a(t)^T \frac{\partial f_\theta(x(t),t}{\partial x(t)}$ and $a(t)$ can be solved using an ODESolver.
- Hence we have $\frac{\partial \mathcal{L}}{\partial \theta} = -\int_{t_0}^{t_1} a(t)^T \frac{\partial f(x(t),t,\theta)}{\partial \theta}\, dt$

Pharmaco-dynamics : an example

Two-Compartment PK Model

- Central compartment (plasma) and peripheral compartment (tissue).
- Drug is administered into the **central compartment**.
- Transfers between compartments with rate constants k_{12} and k_{21}.
- Eliminated from the central compartment with rate k_{10}.

ODE System :

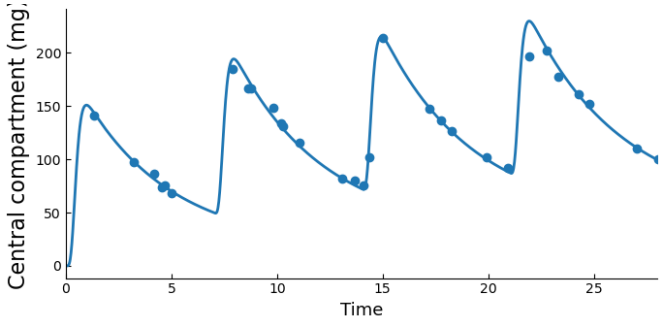
$$\frac{dC_1}{dt} = -k_{10}\,C_1 - k_{12}\,C_1 + k_{21}\,C_2$$

$$\frac{dC_2}{dt} = k_{12}\,C_1 - k_{21}\,C_2$$

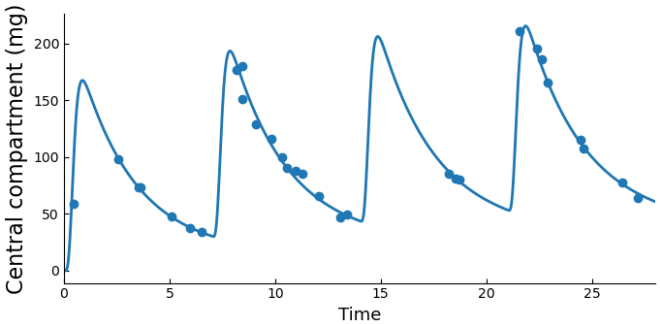
where C_1 and C_2 are drug concentrations in the **central** and **peripheral** compartments.

- Differents possibilities where Neural ODE can be usefull
- Find, for every patient, given some PK data, all k_ij.
- Create an hybrid model.

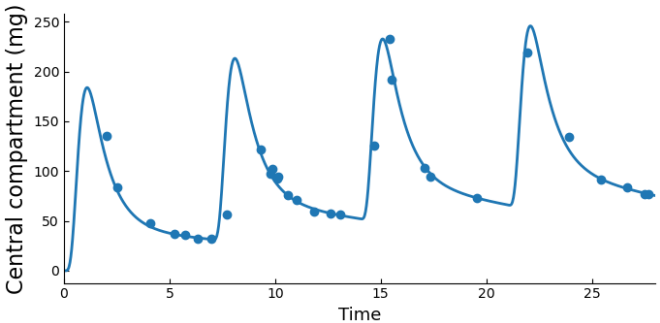
Pharmaco-kinetics example : The data



Patient n°1



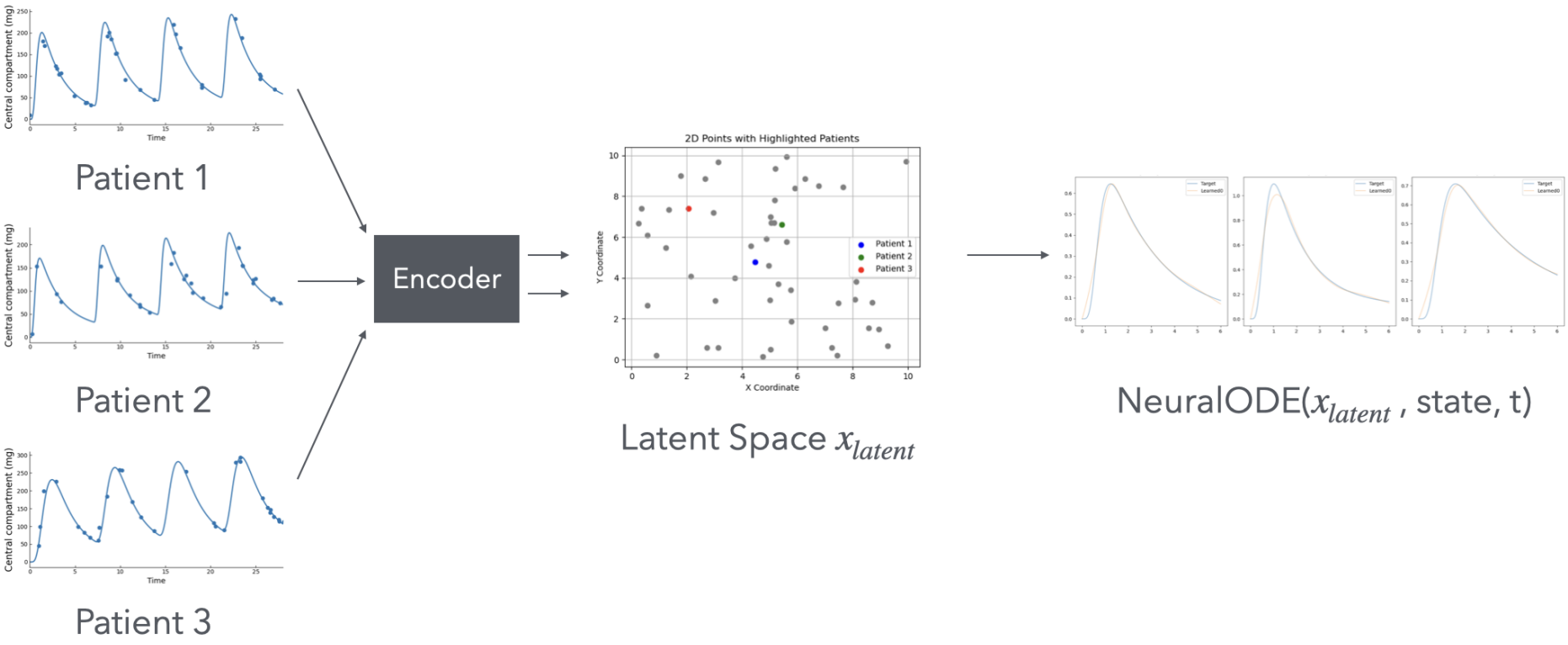
Patient n°2



Patient n°3

- Generated dataset for 50 patients.
- PK model in continuous line
- Noised sample taken at random time (too many)
- Goal : to be able to reconstruct the PK line only from the observed data

Pharmaco-kinetics example : The method



- state $= (C_1(t), C_2(t))$
- Instead of just depending on $C_1(t)$,and $C_2(t)$ my differential equation also depends on the position into the latent space
- It allows to adapt to the variability between patients.

Lotka-Volterra Model based model

- Models the interaction between predator and prey species.
- Equations :

$$\frac{dx}{dt}(t) = \alpha x(t) - \beta x(t) y(t)$$
$$\frac{dy}{dt}(t) = -\gamma y(t) + \delta x(t) y(t)$$

- Data needed :
  - Observations of $x(t)$ (prey population) and $y(t)$ (predator population) over time.
  - Parameters $\alpha, \beta, \gamma, \delta$ governing interactions.
  - Initial conditions $x(0), y(0)$.

Augmenting Lotka-Volterra with Neural Networks

- Adding neural network correction terms :

$$\frac{dx}{dt}(t) = \alpha x(t) - \beta x(t)y(t) + f_\theta(x(t), y(t))$$
$$\frac{dy}{dt}(t) = -\gamma y(t) + \delta x(t)y(t) + g_\theta(x(t), y(t))$$

- This allows for learning residual dynamics from data.
- Loss function for training :

$$\mathcal{L}(\theta) = \frac{1}{NM} \sum_{i=1}^{N} \sum_{j=1}^{M} \left((x_\theta(t_j) - x_i(t_j))^2 + (y_\theta(t_j) - y_i(t_j))^2 \right)$$