Neural Networks: Basic Structure, Representation Power

Machine Learning Course - CS-433
Nov 1, 2023
Nicolas Flammarion

EPFL

Deep Learning: ML Breakthrough

State of the art performance across a broad range of useful tasks

From Linear Models to NNs

Supervised learning: we observe some data $S_{\text{train}} = \{x_n, y_n\}_{n=1}^N \in \mathcal{X} \times \mathcal{Y}$

➡ given a new x, we want to predict its label y

Linear prediction (with augmented features): $y = f_{\text{Lin}}(x) = \phi(x)^\top w$

Features are given

Prediction with a neural network (NN):

$$y = f_{\text{NN}}(x) = f(x)^\top w$$

Function implemented by the NN parameters

First layers transform the input into a good representation

Last layer is performing a linear prediction

Why Neural Networks?

Linear models vs. neural networks:

$$y = f_{\text{Lin}}(x) = \phi(x)^{\top} w \text{ (fixed } \phi) \quad \text{vs.} \quad y = f_{\text{NN}}(x) = f(x)^{\top} w \text{ (learned } f)$$

Classical machine learning:
— Relatively simple models on top of **features handcrafted by domain experts**
— Only works well when used with **good handcrafted features**

Deep learning:
— Large neural networks that **learn features directly from the data**
— Can be viewed as a complicated feature extractor + linear prediction
— **Requires large amounts of data and compute to train**
— Quality often continues to **improve substantially** with more data and larger models

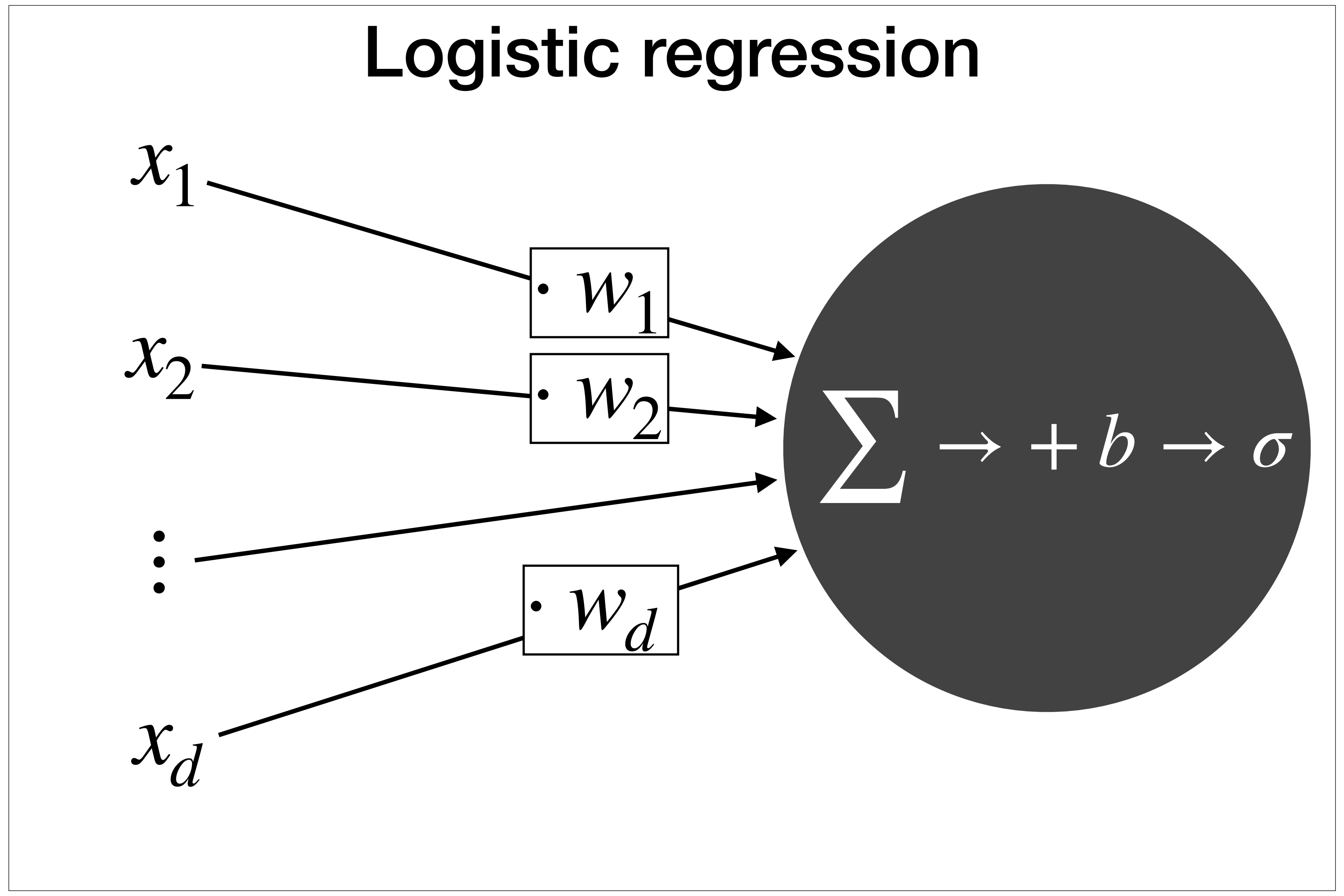
The Basic Structure of Neural Networks

Recap: Logistic Regression

<u>Logistic Regression</u>:

$$f_{LR}(x) = p(1\,|\,x) = \sigma(x^\top w + b) = \sigma\left(\sum_{i=1}^{d} x_i w_i + b\right)$$

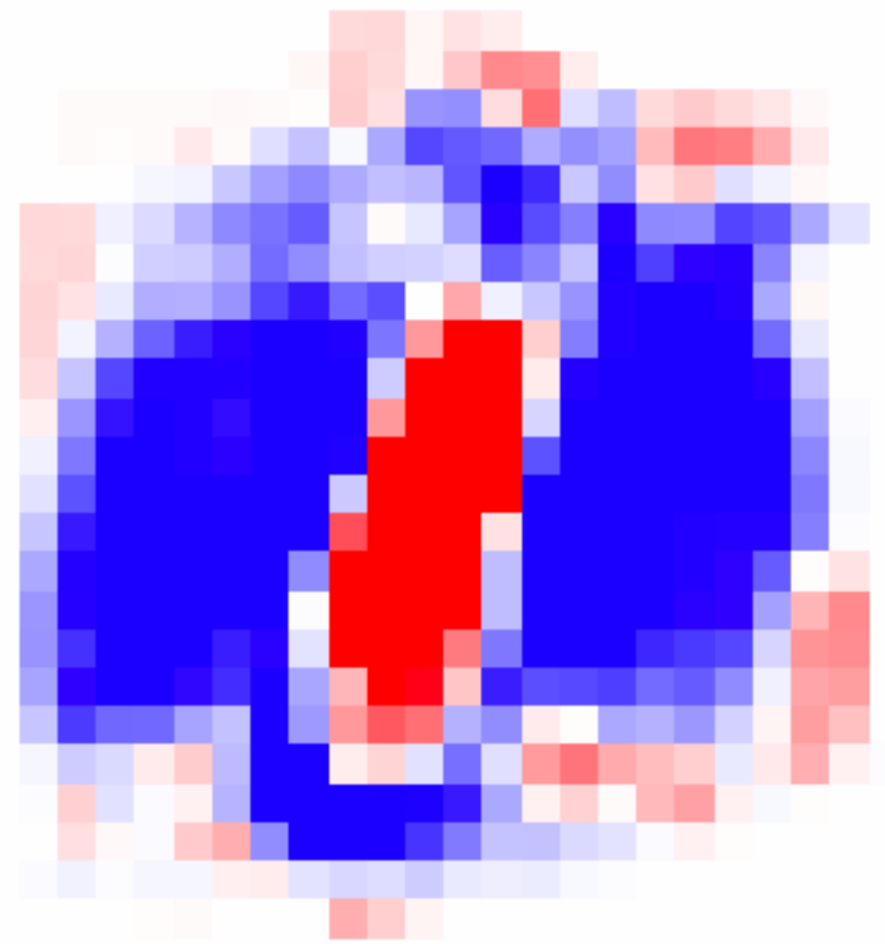
where $\sigma(z) = (1 + \exp(-z))^{-1}$ is the *sigmoid*



<u>Pattern-matching perspective</u>:

- **Task**: classify digit 1 vs. digit 0 with logistic regression
- We learn a **single pattern** w that we apply elementwise to each input x (very restrictive!)
- We classify the digit as 1 if $p(1\,|\,x) \geq 0.5$ or, equivalently, if $\sum_{i=1}^{d} x_i w_i \geq -b$

learned w

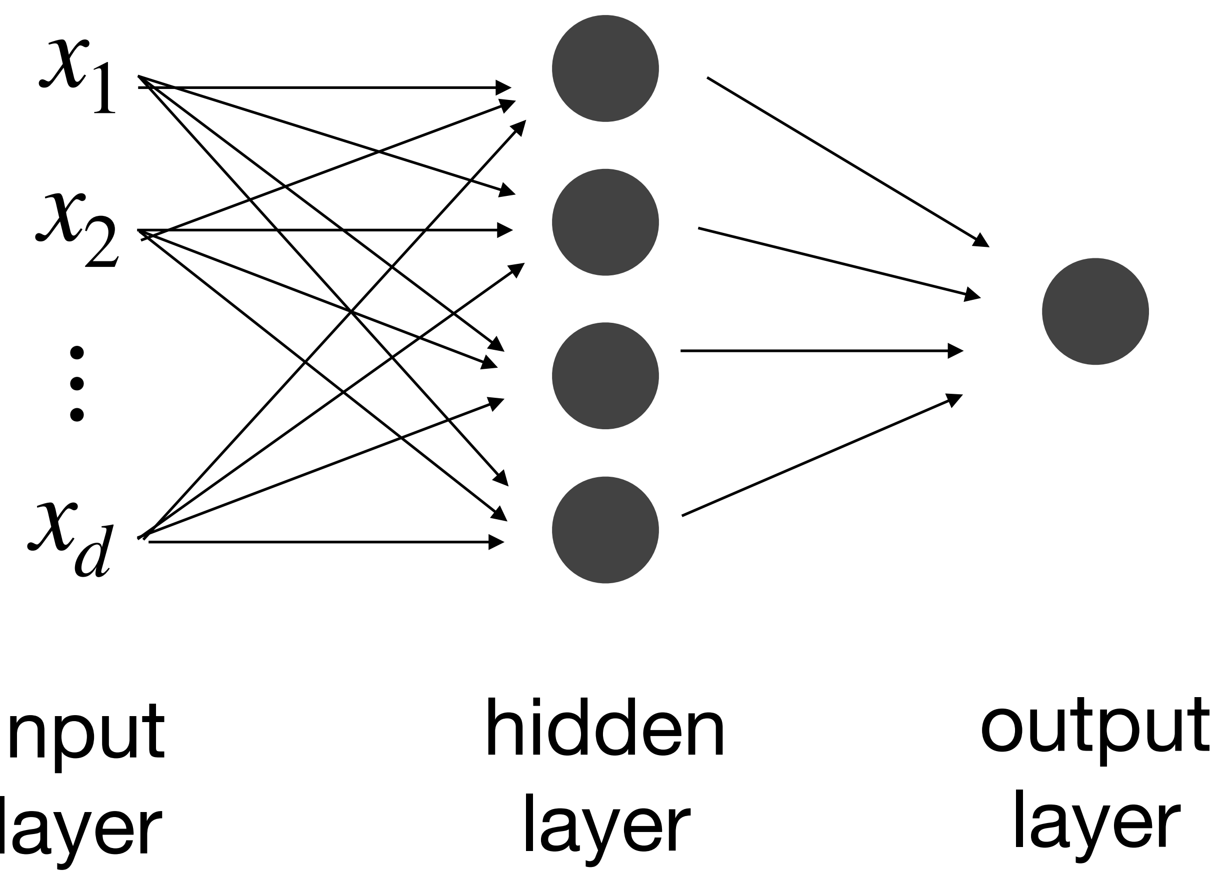


Two-Layer Neural Networks

Two-layer neural network (aka two-layer perceptron):

- Stack logistic regression units $\phi(x)_j = \sigma\left(\sum_{i=1}^{d} x_i w_{i,j}^{(1)} + b_j\right)$ (aka *hidden units or neurons*) in a *hidden layer*
- Aggregate the hidden units with a linear function $\phi(x)^\top w^{(2)}$

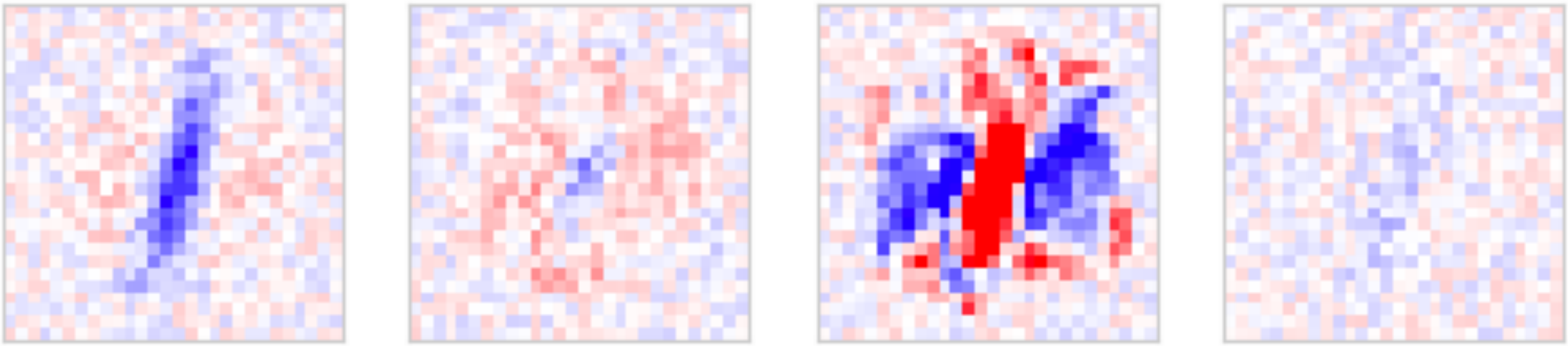
Two-layer neural network



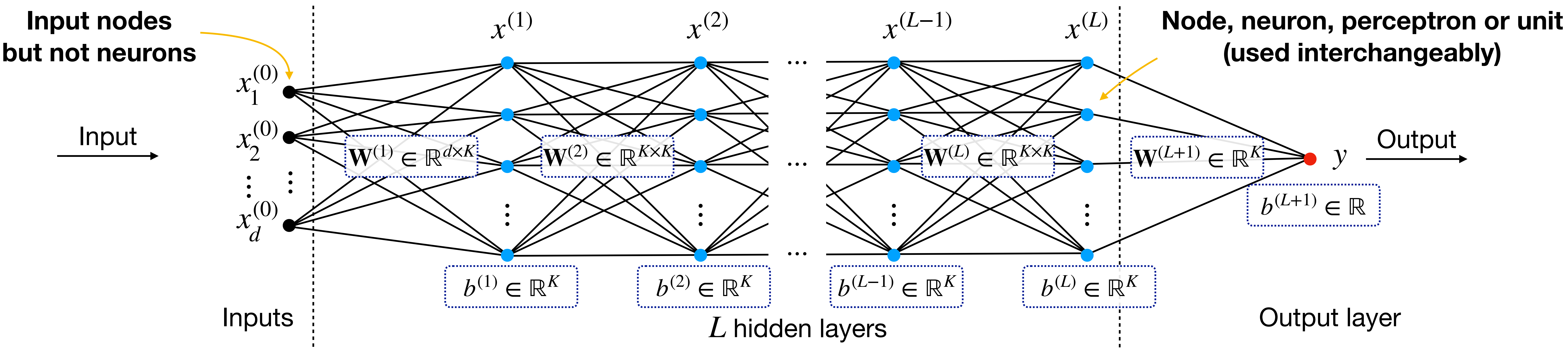
Pattern-matching perspective:

- **Task**: classify digit 1 vs. digit 0 with a two-layer neural network
- **Each** hidden unit learns a **different pattern** (not necessarily interpretable!)
- We classify based on a linear combination of these patterns ⇒ much more flexible

learned $w_{:,1}^{(1)}$ **learned** $w_{:,2}^{(1)}$ **learned** $w_{:,3}^{(1)}$ **learned** $w_{:,4}^{(1)}$



Multi-layer Neural Networks



Assume L hidden layers with K neurons each + output layer with single node

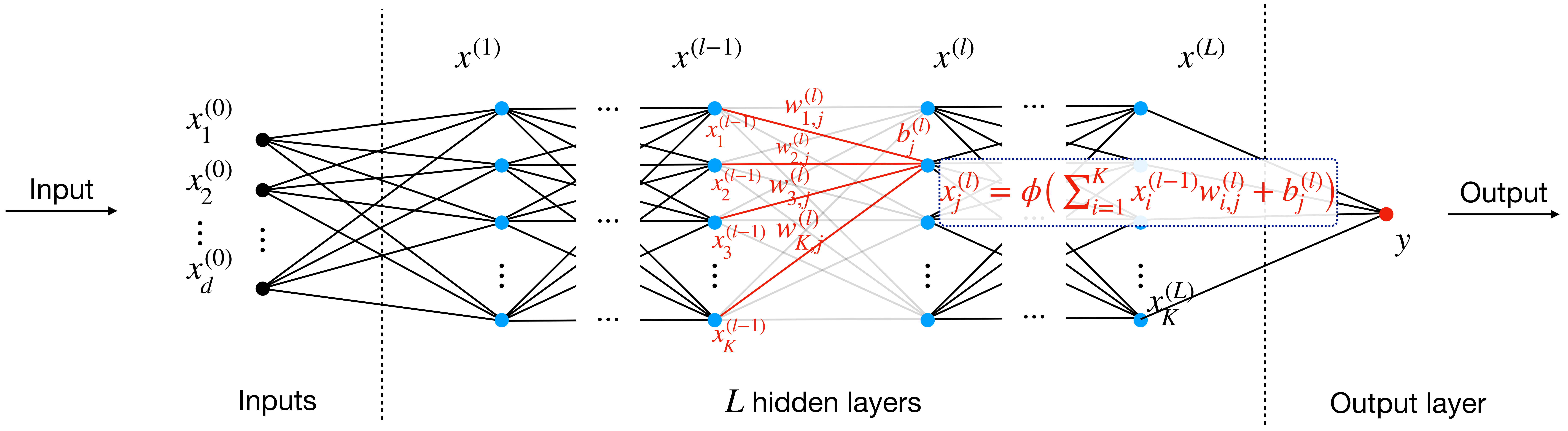
Outputs of hidden layer l given by vector: $x^{(l)} = f^{(l)}(x^{(l-1)}) := \phi\left((\mathbf{W}^{(l)})^\top x^{(l-1)} + b^{(l)}\right)$

Elementwise

Learnable Parameters: Weight matrices $\mathbf{W}^{(l)}$ and bias vectors $b^{(l)}$ for $1 \leq l \leq L+1$

— Each column of $\mathbf{W}^{(l)}$ corresponds to the weights of one perceptron

Single Neuron View



$$x_j^{(l)} = \phi\left(\sum_{i=1}^{K} x_i^{(l-1)} w_{i,j}^{(l)} + b_j^{(l)} \right)$$

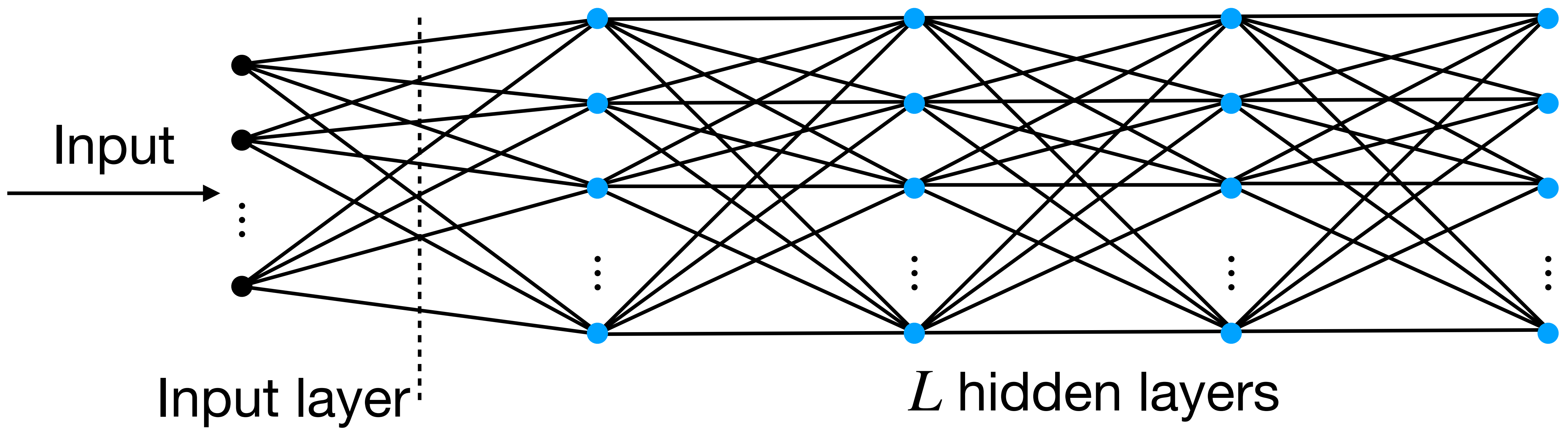
Important: ϕ is non-linear
otherwise we can only
represent linear functions

weight of the edge going from node i in layer $l-1$ to node j in layer l

bias term associated with node j in layer l

NNs extract suitable features from the input

A NN can be decomposed into a feature extractor and the output layer.
Feature extractor from $\mathbb{R}^d$ to $\mathbb{R}^K$:



$\Longleftrightarrow$

$$\begin{aligned} &\mathbb{R}^d \to \mathbb{R}^K \\ f: \quad &x \mapsto \begin{pmatrix} f_1(x) \\ \vdots \\ f_K(x) \end{pmatrix} \end{aligned}$$

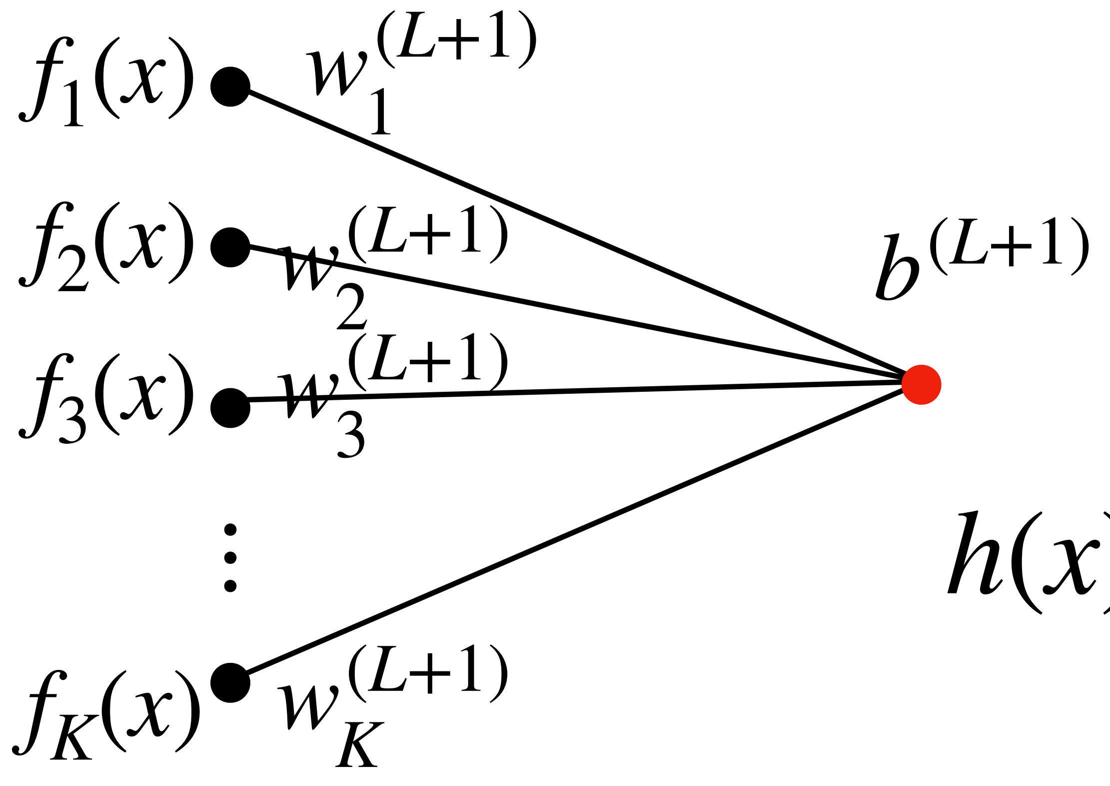
This function is defined by

- The biases $\{b^{(l)}\}_{l\in[L]}$ and weights $\{\mathbf{W}^{(l)}\}_{l\in[L]}$ so we learn
  ➡ $O(dK + K^2L) \approx O(K^2L)$ parameters
- The activation function ϕ we pick

In practice: both L and K are large - overparametrized NNs

Neural Network: Training Time

Input



Data representation

Output

$$h(x) = f(x)^\top w^{(L+1)} + b^{(L+1)}$$

Linear prediction

Regression

$$\ell(y, h(x)) = (h(x) - y)^2$$

Binary Classification with $y \in \{-1,1\}$

$$\ell(y, h(x)) = log(1 + exp(-yh(x)))$$

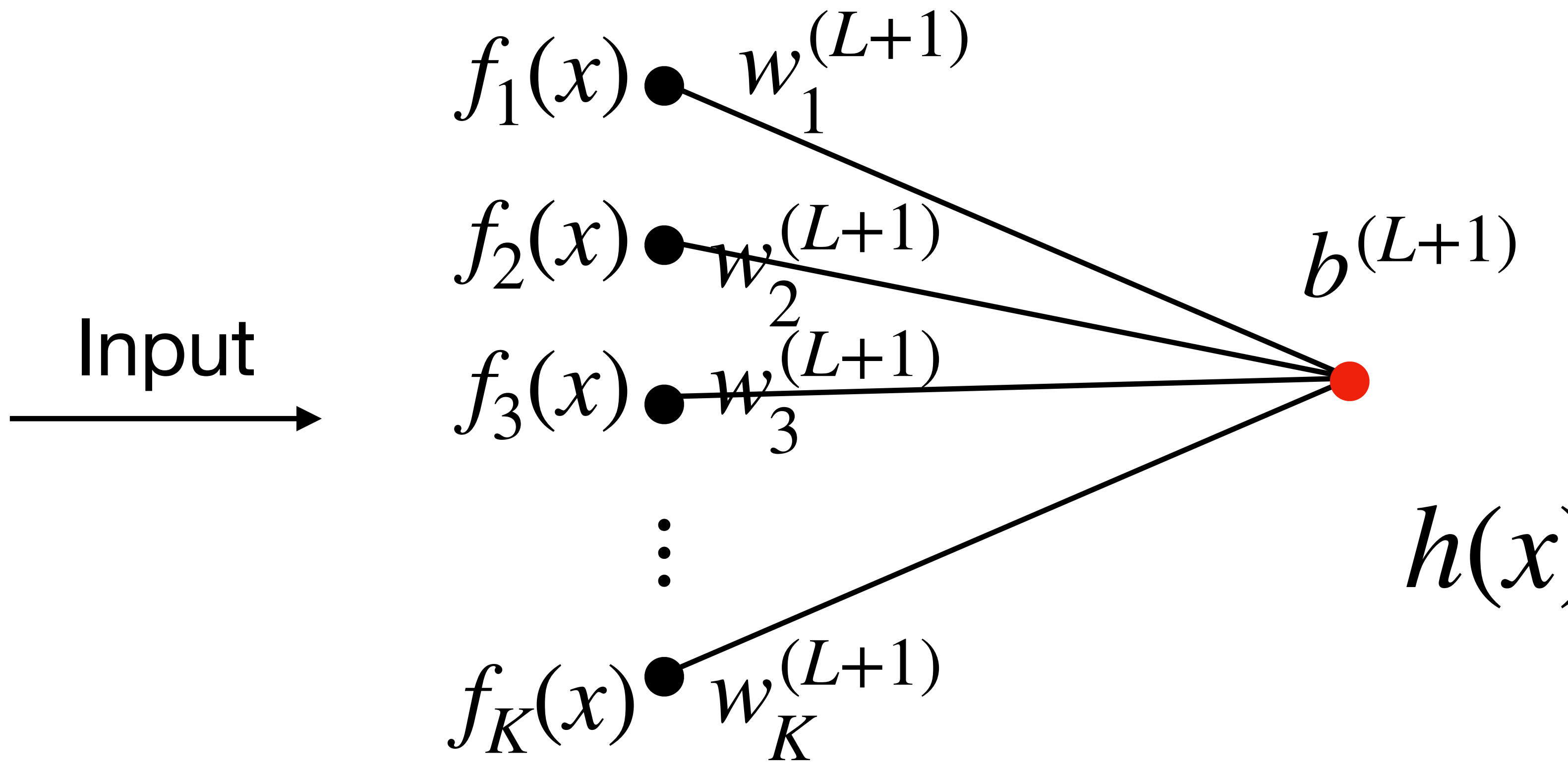
Multi-Class Classification with $y \in \{1,\ldots,K\}$
(see the exercise!)

$$\ell(y, h(x)) = -\,log \frac{e^{h(x)_y}}{\sum_{i=1}^{K} e^{h(x)_i}}$$

ML task

With a suitable representation of the data $f(x)$, the last layer only performs a linear regression or classification step

Neural Network: Inference Time



Output $\longrightarrow$

$$h(x) = f(x)^\top w^{(L+1)} + b^{(L+1)}$$

Regression

$h(x)$

Binary Classification with $y \in \{-1, 1\}$

$sign\big(h(x)\big)$

Multi-Class Classification with $y \in \{1, \ldots, K\}$
(see the exercise!)

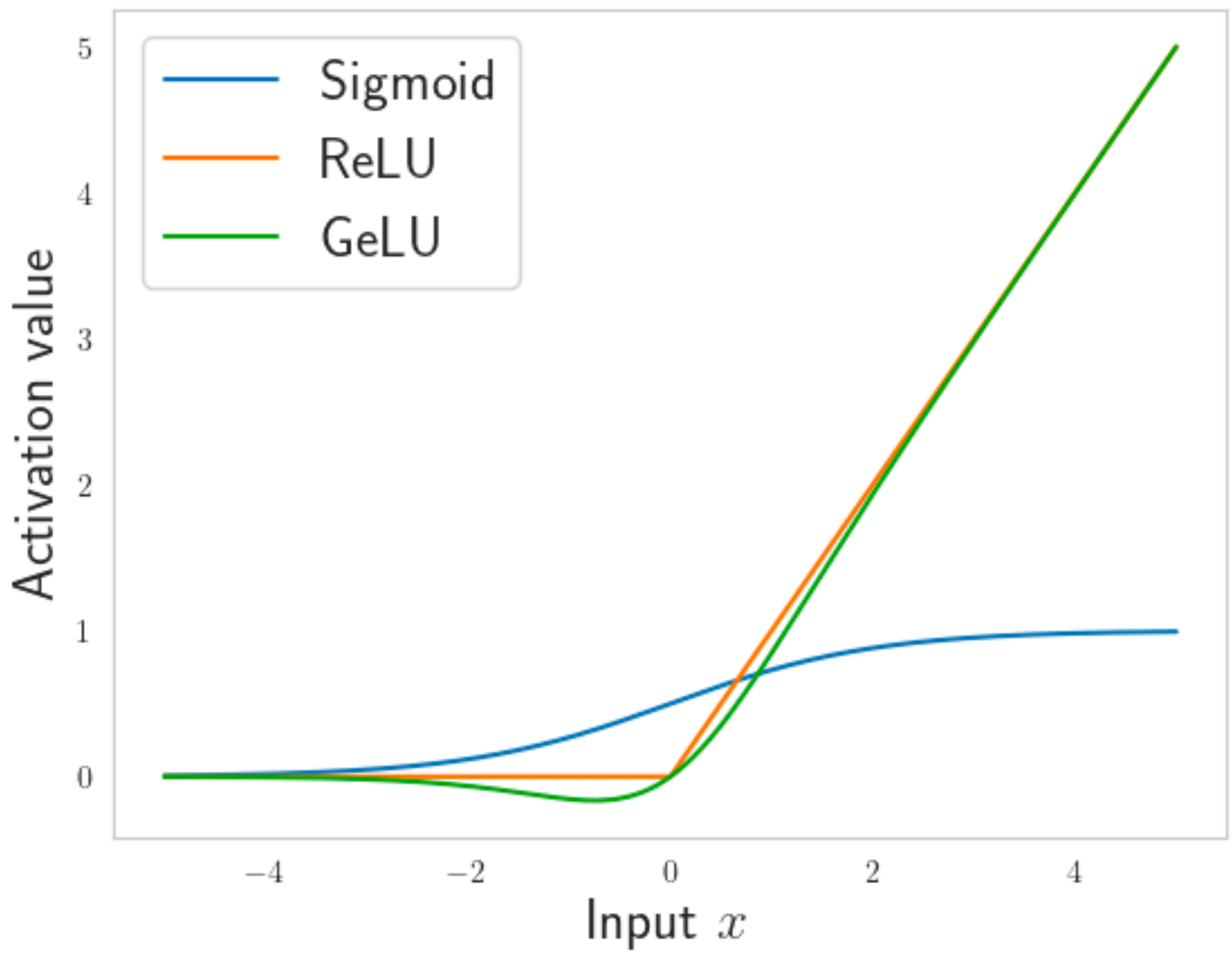
$argmax_{c \in \{1,\ldots,K\}}\ h(x)_c$

Data representation | Linear prediction | ML task

With a suitable representation of the data $f(x)$, the last layer only performs a linear regression or classification step

Popular activation functions

- Sigmoid: $\phi(x) = \sigma(x) = \dfrac{1}{1 + e^{-x}}$
- ReLU: $\phi(x) = (x)_+ = \max\{0, x\}$
- GELU: $\phi(x) = x \cdot \Phi(x) \approx x \cdot \sigma(1.702x)$



- In practice: ReLU and GeLU are the most widely used activation functions

Representational Power of NNs

Three theoretical questions in Deep Learning

- **Expressive power** of NNs: Why are the functions we are interested in so **well approximated** by NNs?
- **Success of naive optimization**: Why does **gradient descent** lead to a good local minimum?
- **Generalization miracle**: Why is there **no overfitting** with so many parameters?

L_2 Approximation: Barron's result

Let $f : \mathbb{R}^d \to \mathbb{R}$ and define $\hat{f}(\omega) = \int_{\mathbb{R}^d} f(x) e^{-i\omega^\top x} dx$, its Fourier transform

<u>Assumption</u>: $\int_{\mathbb{R}^d} |\omega| |\hat{f}(\omega)| d\omega \le C$ (smoothness assumption)

<u>Claim</u>: For all $n \ge 1$ and $r > 0$, there exists a function f_n of the form

$$f_n(x) = \sum_{j=1}^{n} c_j \phi(x^\top w_j + b_j) + c_0$$

such that

$$\int_{|x| \le r} \left(f(x) - f_n(x) \right)^2 dx \le \frac{(2Cr)^2}{n}$$

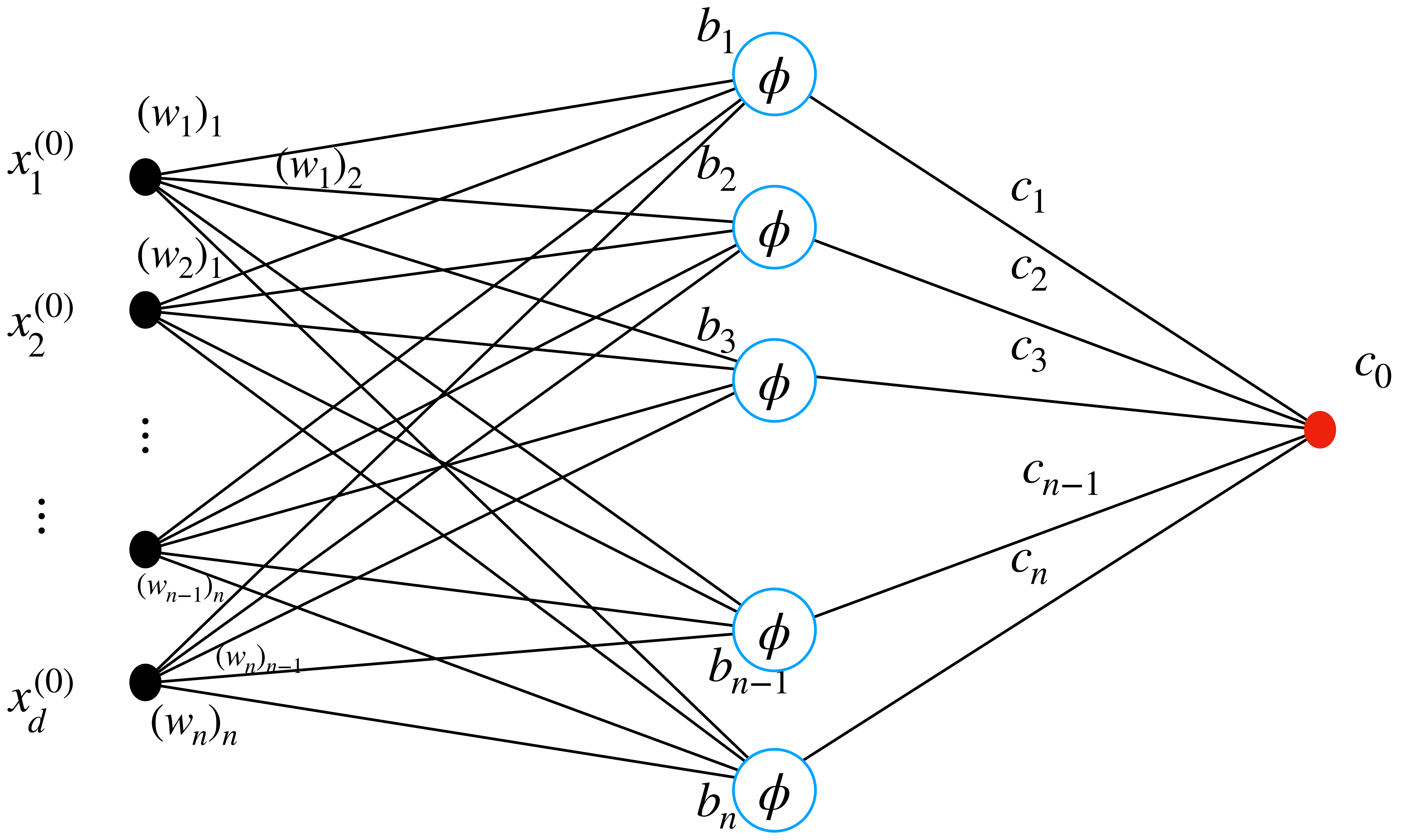
All sufficiently smooth function can be approximated by a one-hidden-layer NN

$$\int_{|x|\le r} \left(f(x) - f_n(x)\right)^2 dx \le \frac{(2Cr)^2}{n}$$

- The more neurons allowed, the smaller the error.
- The smoother the function (the smaller C), the smaller the error
- The larger the domain (the larger r), the greater the error
- Approximation is in average (in ℓ_2-norm)
- Applicable for any “sigmoid-like” activation function

The function f_n is a one-hidden-layer NN with n nodes

$$f_n(x) = \sum_{j=1}^{n} c_j \phi(x^\top w_j + b_j) + c_0 = c^\top \phi(W^\top x + b) + c_0$$



L_1 Approximation: Proof by picture

Simple and intuitive explanation of a slightly different result:

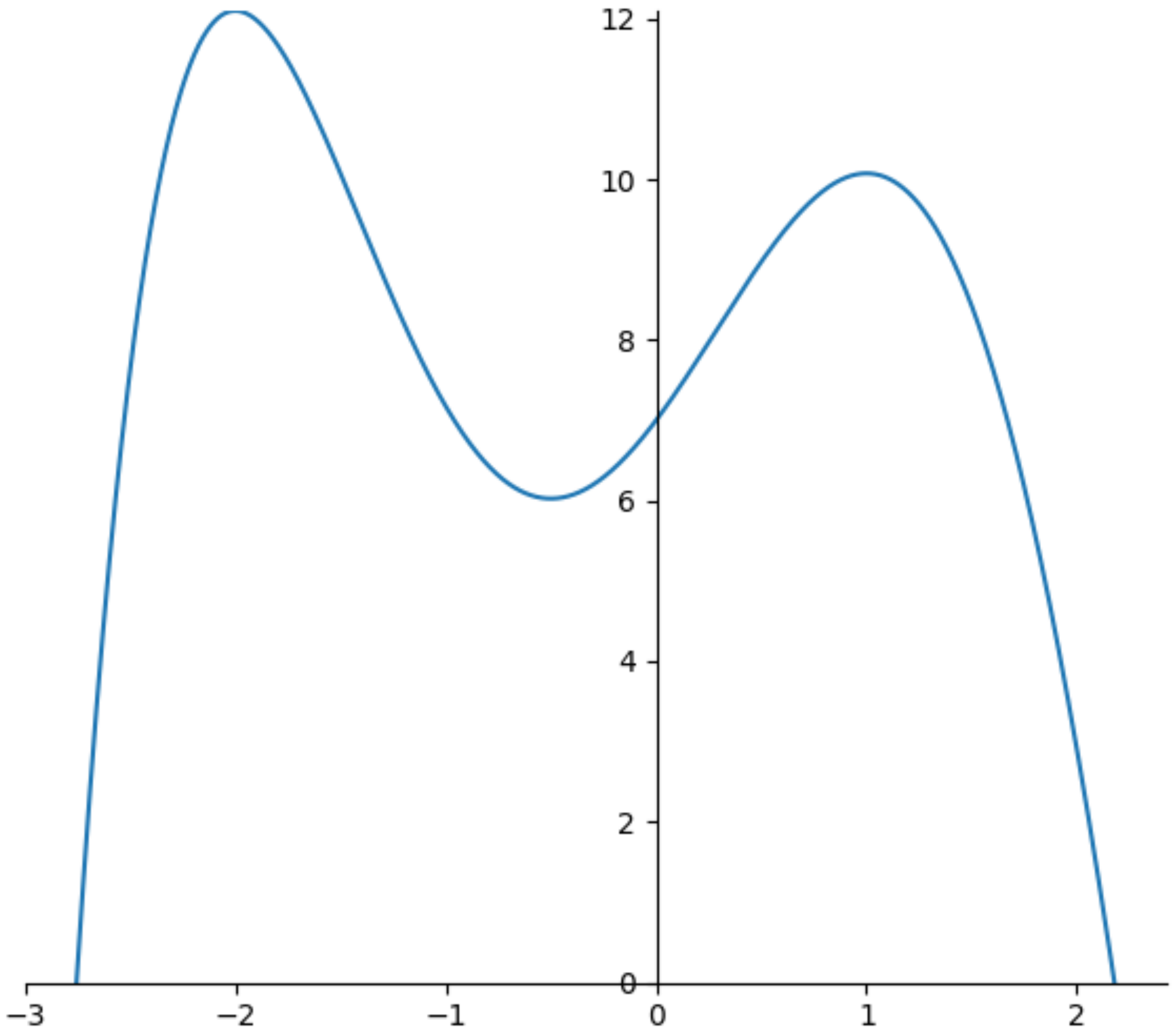
"A NN with sigmoid activation and at most two hidden layers can approximate well a smooth function in ℓ_1-norm"

Approximation in ℓ_1-norm:

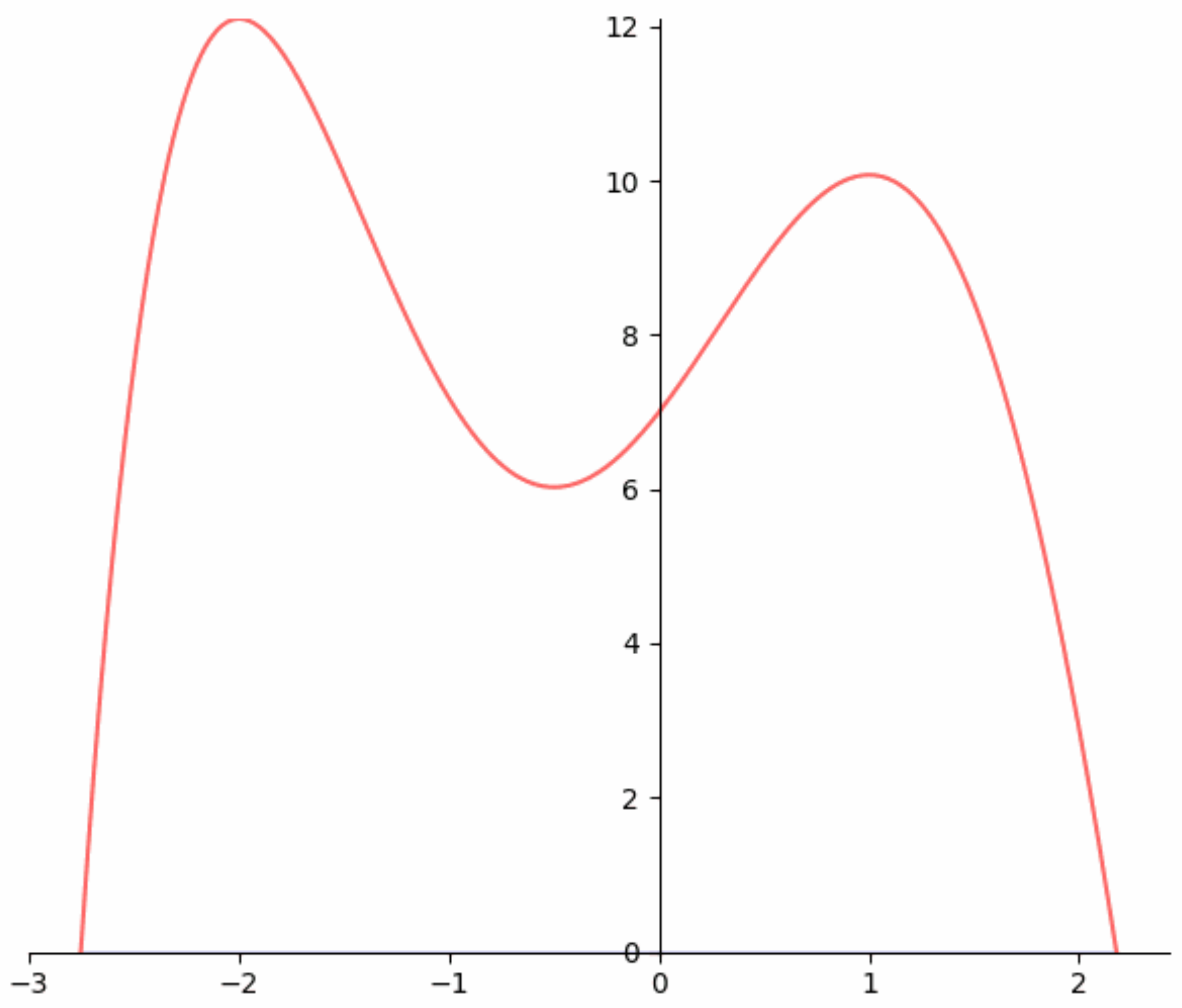
$$\int_{|x| \leq r} |f(x) - f_n(x)| \, dx \leq \text{Something small}$$

L_1 Approximation: Proof by picture

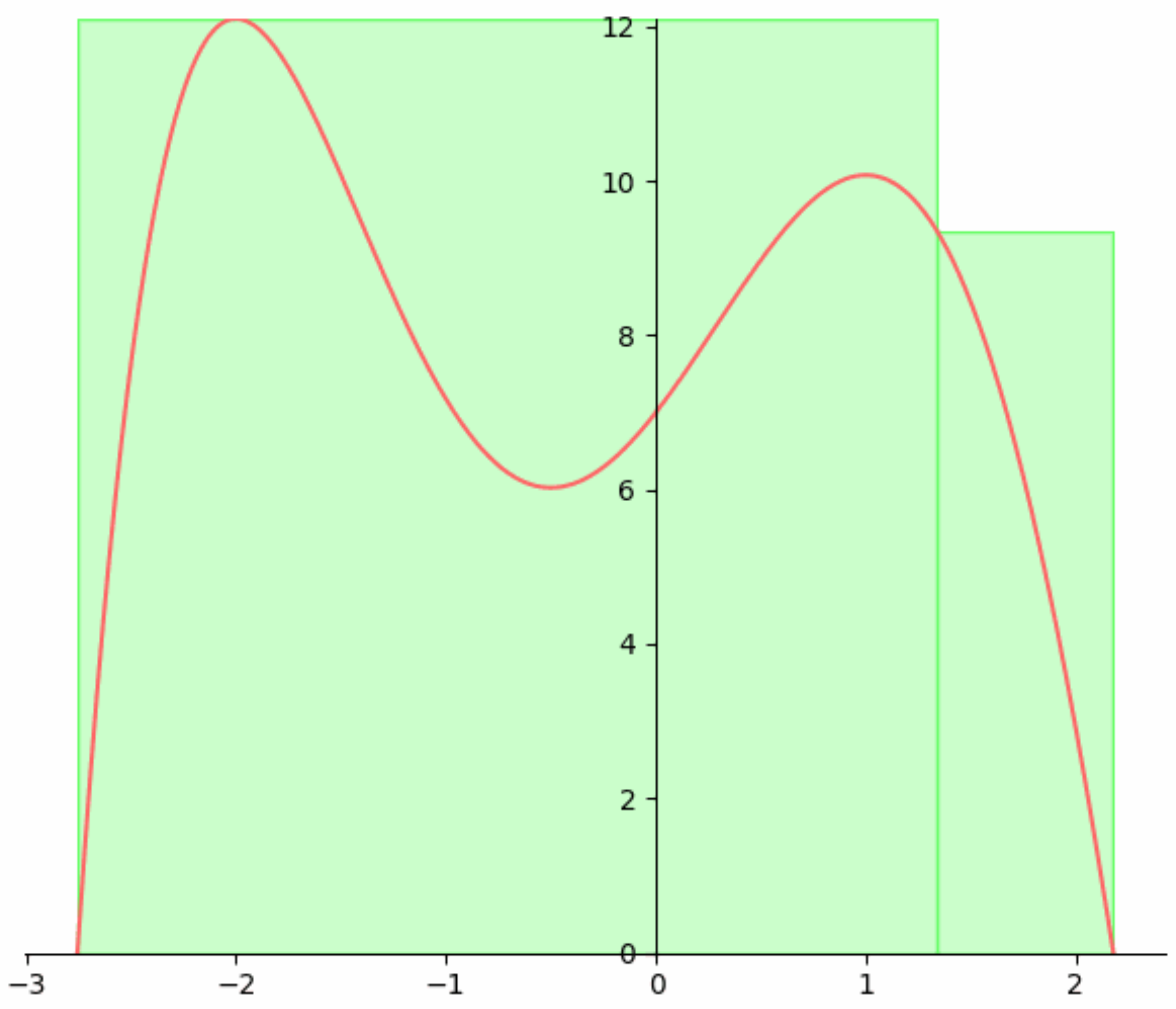
Consider a function $f : \mathbb{R} \to \mathbb{R}$ on a bounded domain



Approximation of the function by a sum of rectangle functions



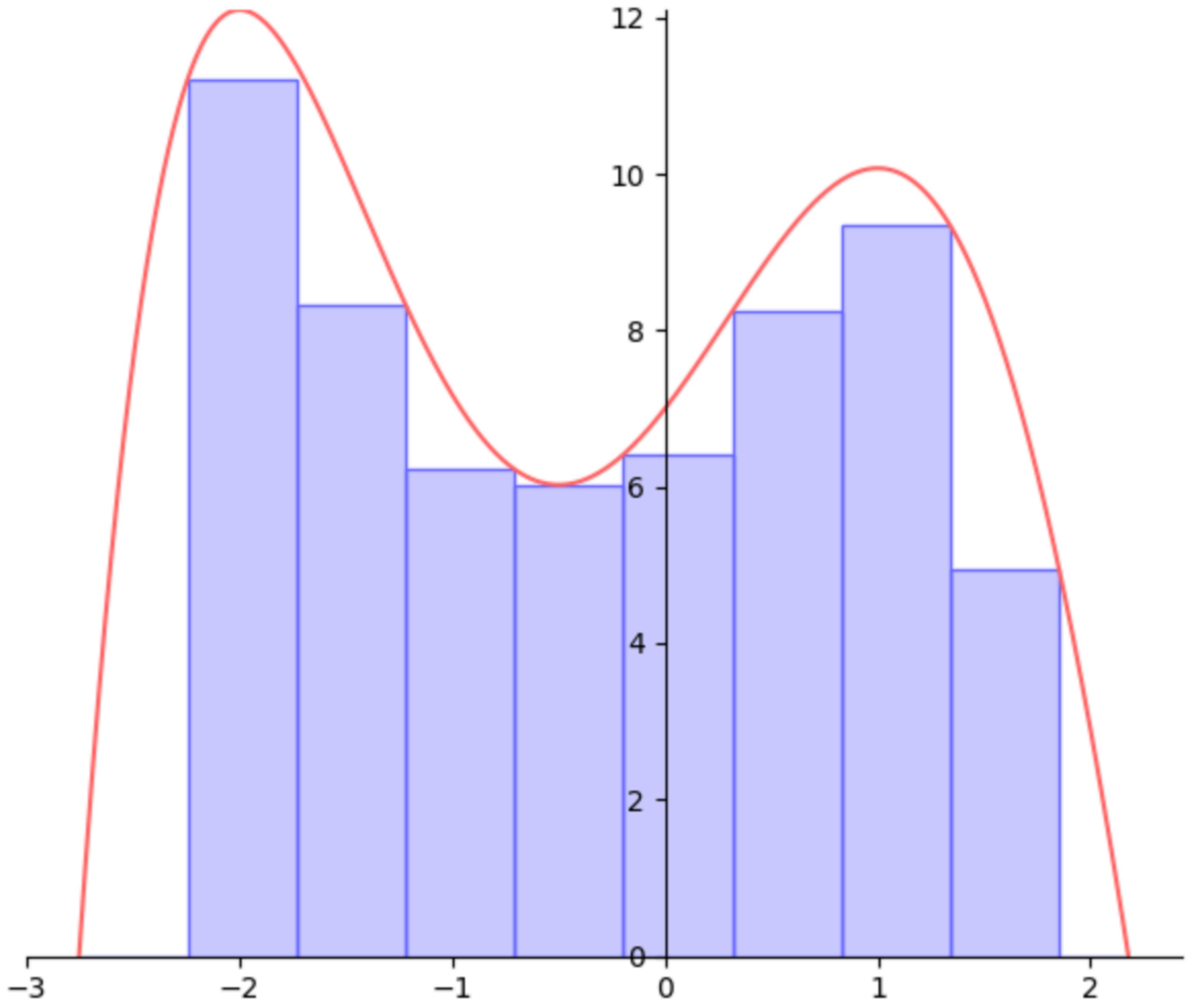
From below



From above

The function is Riemann integrable - its integral can be approximated to any desired accuracy using “lower” and “upper” sums of the area of rectangles

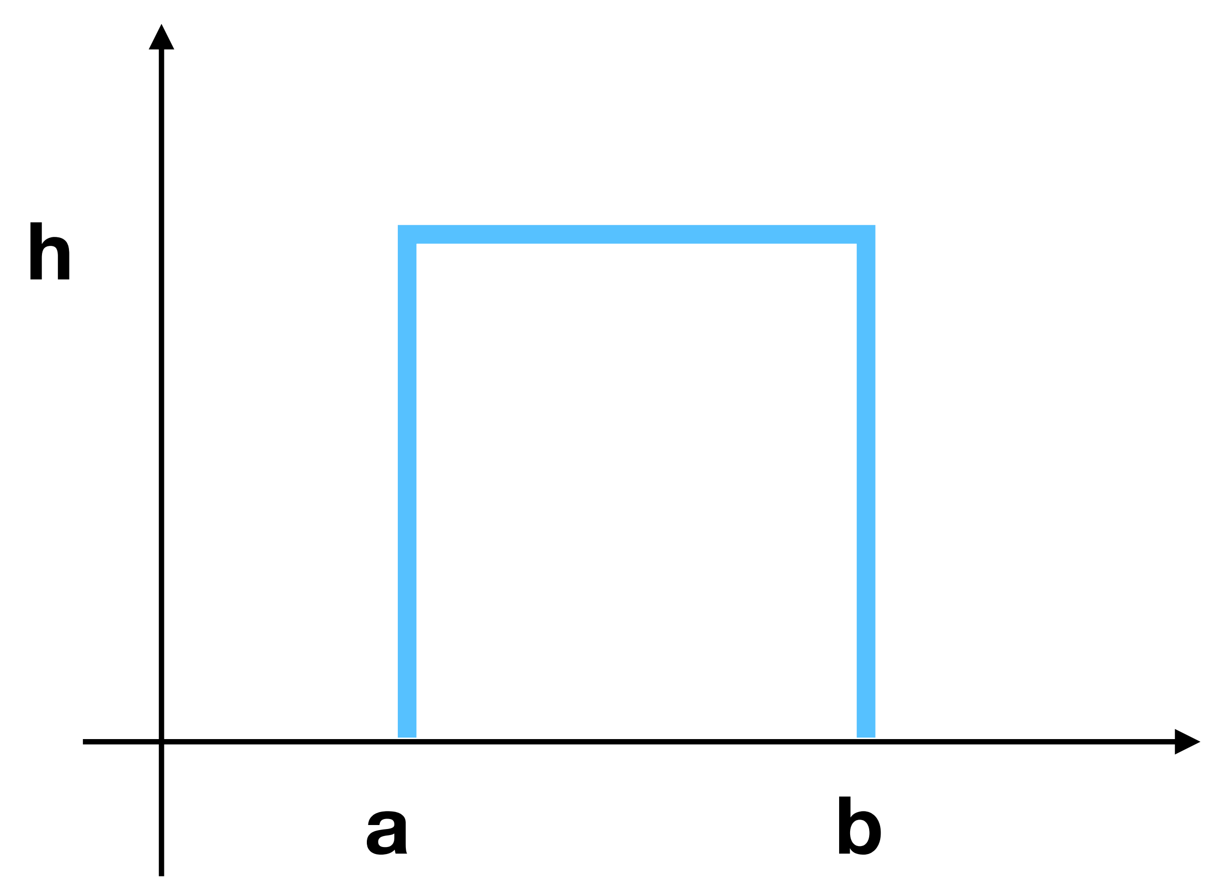
Approximation of the function by a sum of rectangle functions



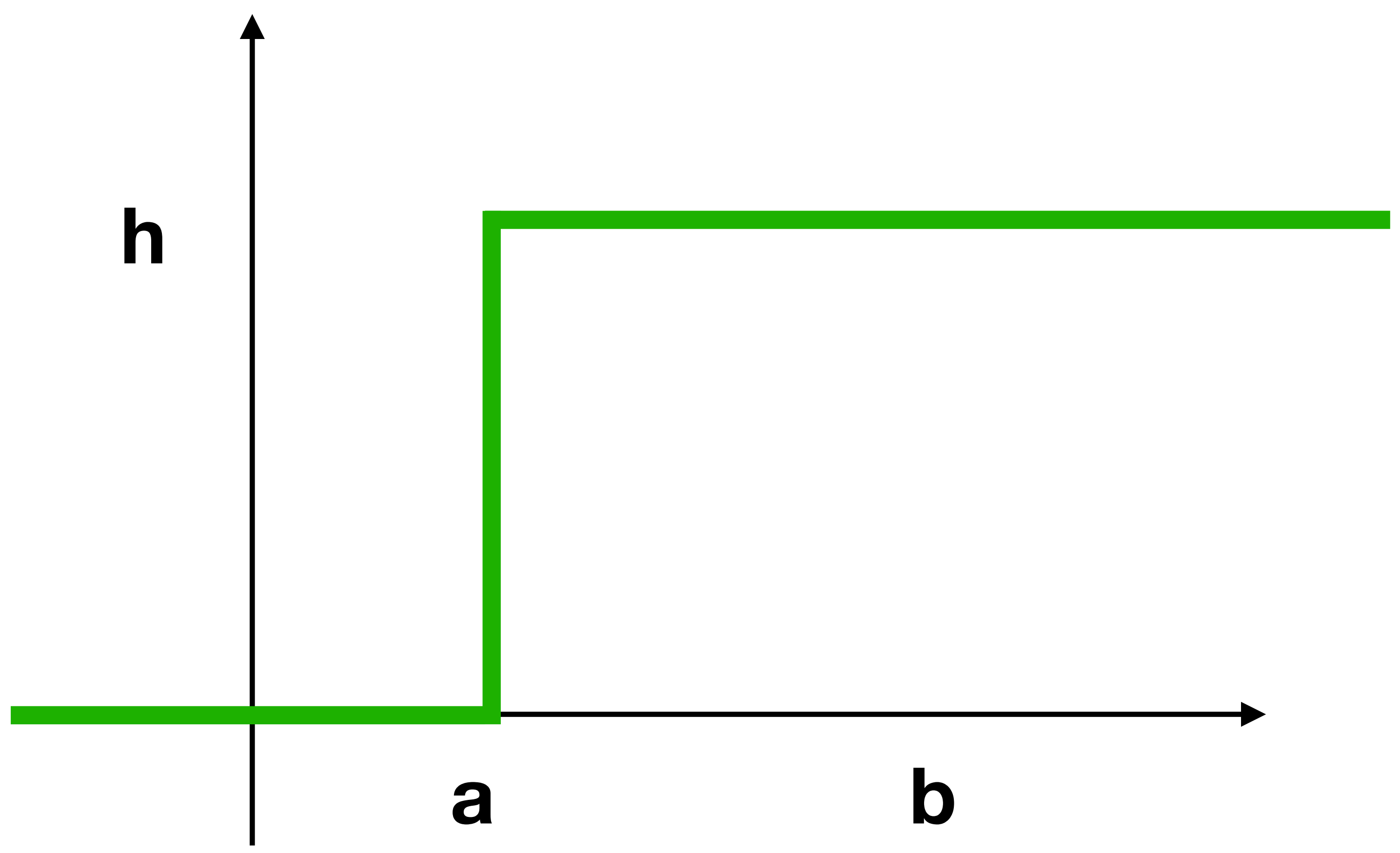
Approximation in ℓ_1-norm:

$$\begin{aligned}\int_{|x|\le r} |f(x) - f_n(x)|\,dx &= \int_{|x|\le r} (f(x) - f_n(x))dx \\ &= \int_{|x|\le r} f(x)dx - \int_{|x|\le r} f_n(x)dx \\ &= \int_{|x|\le r} f(x) - \sum_i \text{Area}(\text{Rectangle}_i)\end{aligned}$$

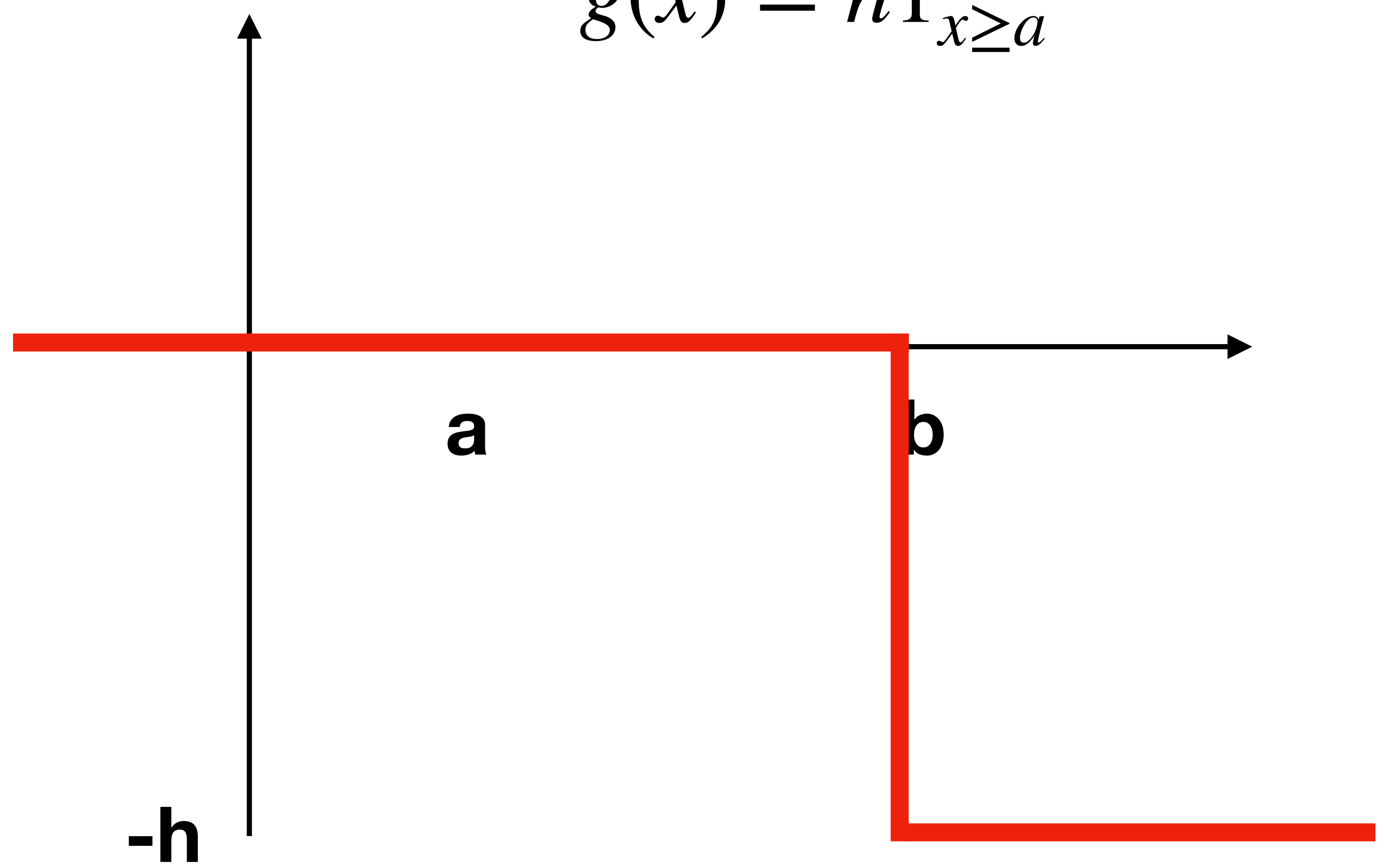
A rectangle function is equal to the sum of two step functions



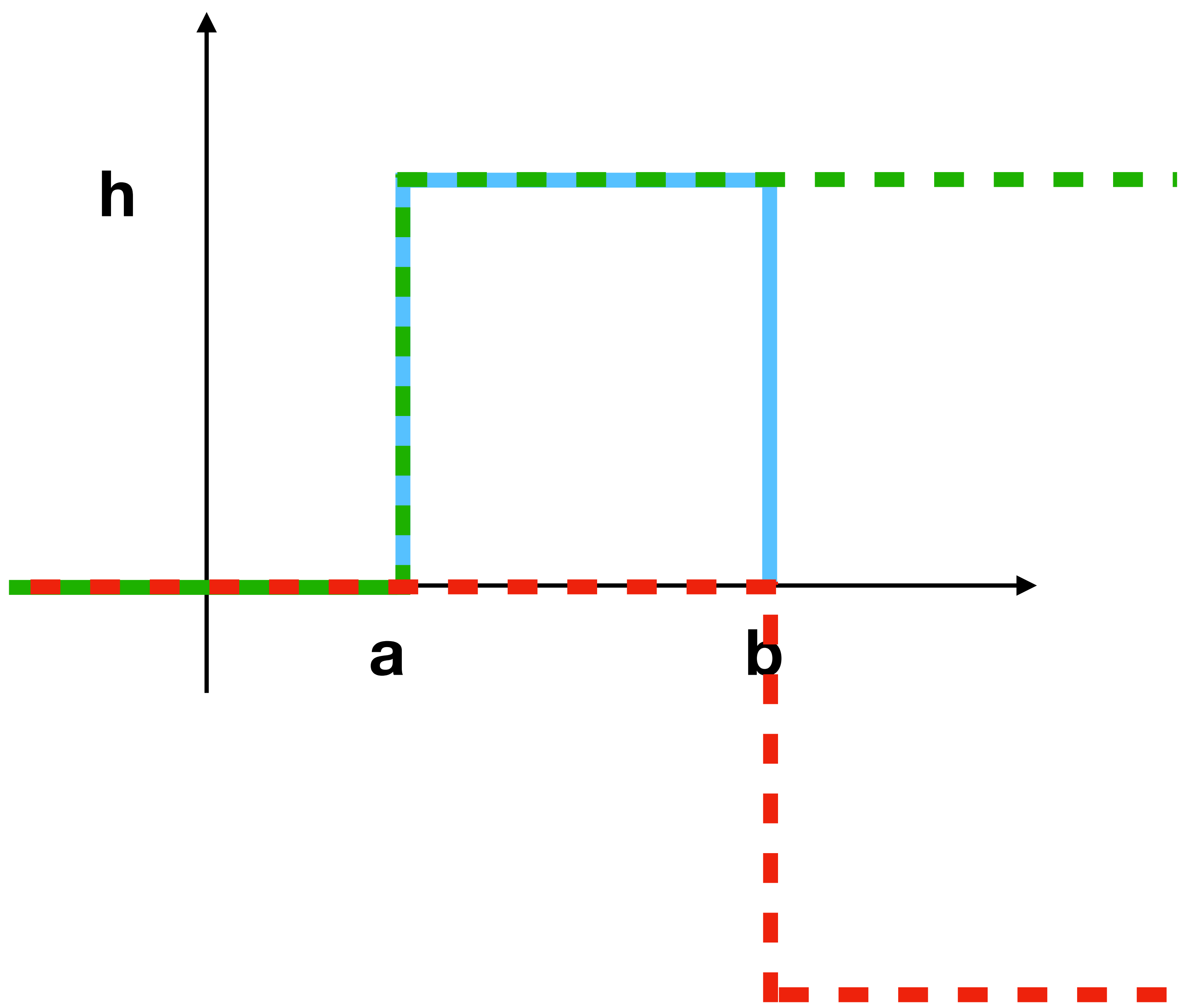
$f(x) = h1_{x \in [a,b]}$



$g(x) = h1_{x \geq a}$



$h(x) = -h1_{x \geq b}$



$f(x) = g(x) + h(x)$

Approximate a step function with a sigmoid

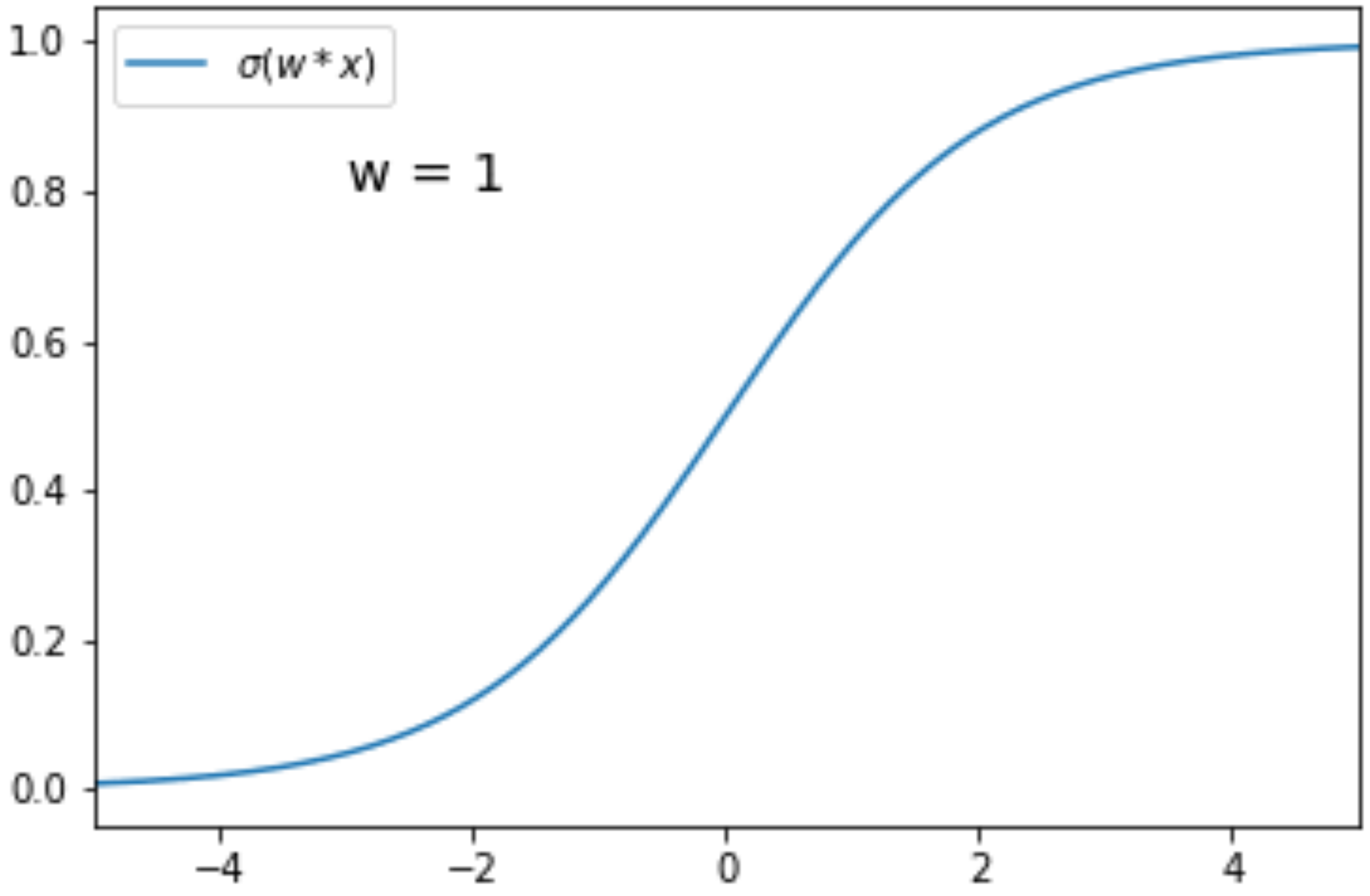
$$\tilde{\phi}(x) = \phi\big(w(x - b)\big)$$

By setting:

- b: where the transition happens
- w: makes the transition steeper

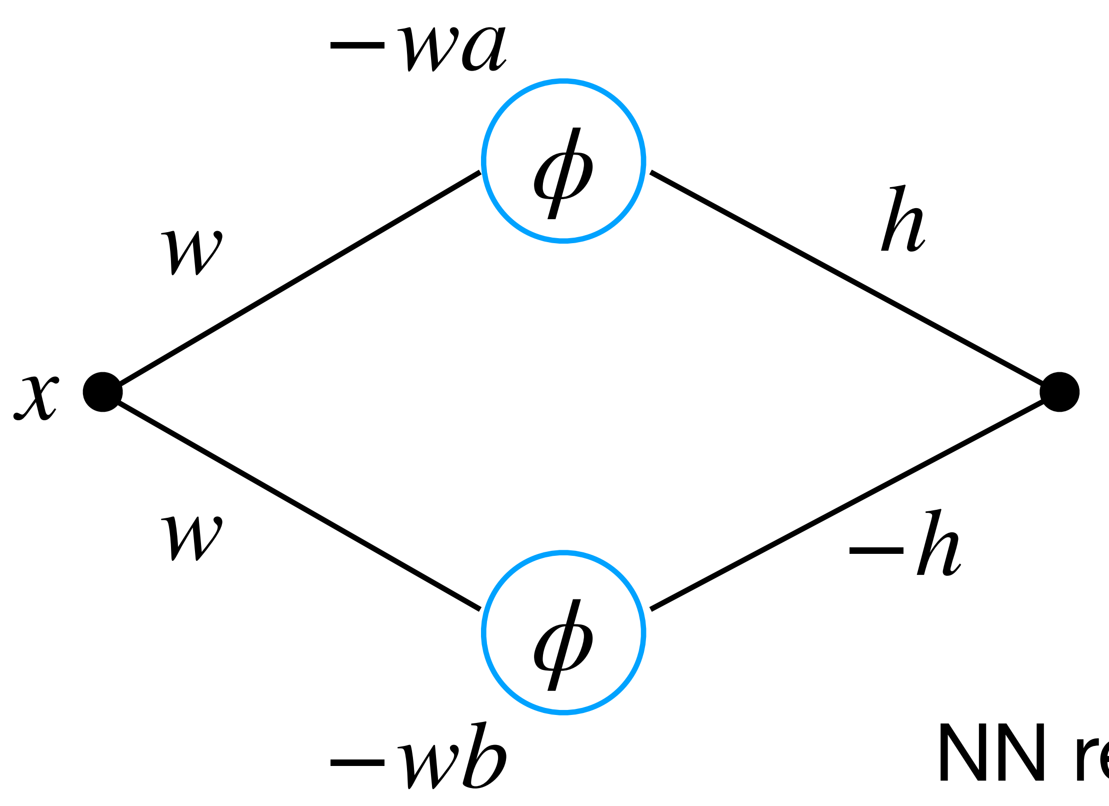
Derivative: $\tilde{\phi}'(b) = w/4$

➡ The width of the transition is $O(4/w)$



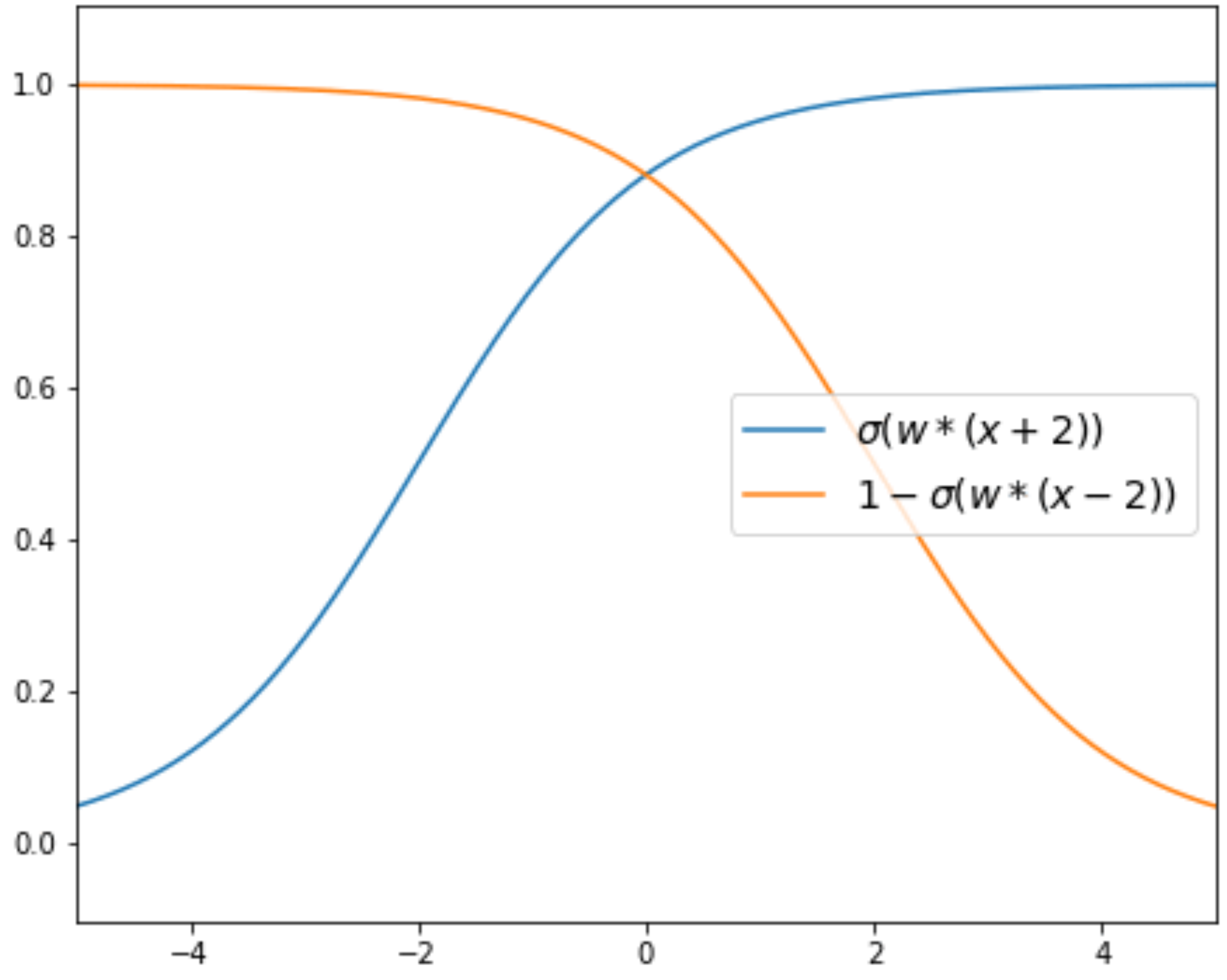
Approximation of the rectangle

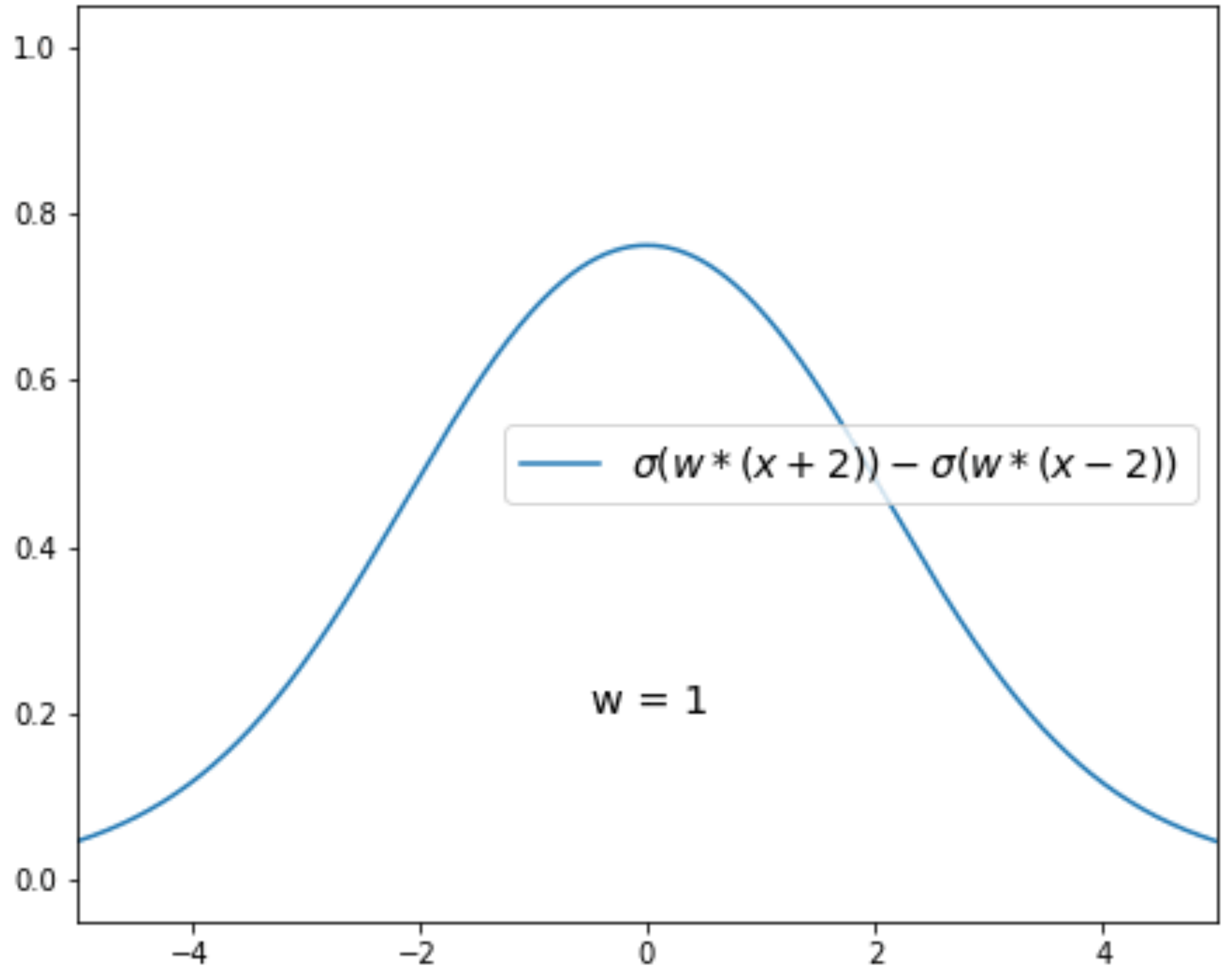
$$h\Big(\phi(w(x-a)) - \phi(w(x-b))\Big)$$



NN representation

$a = -2$
$b = 2$
$h = 1$





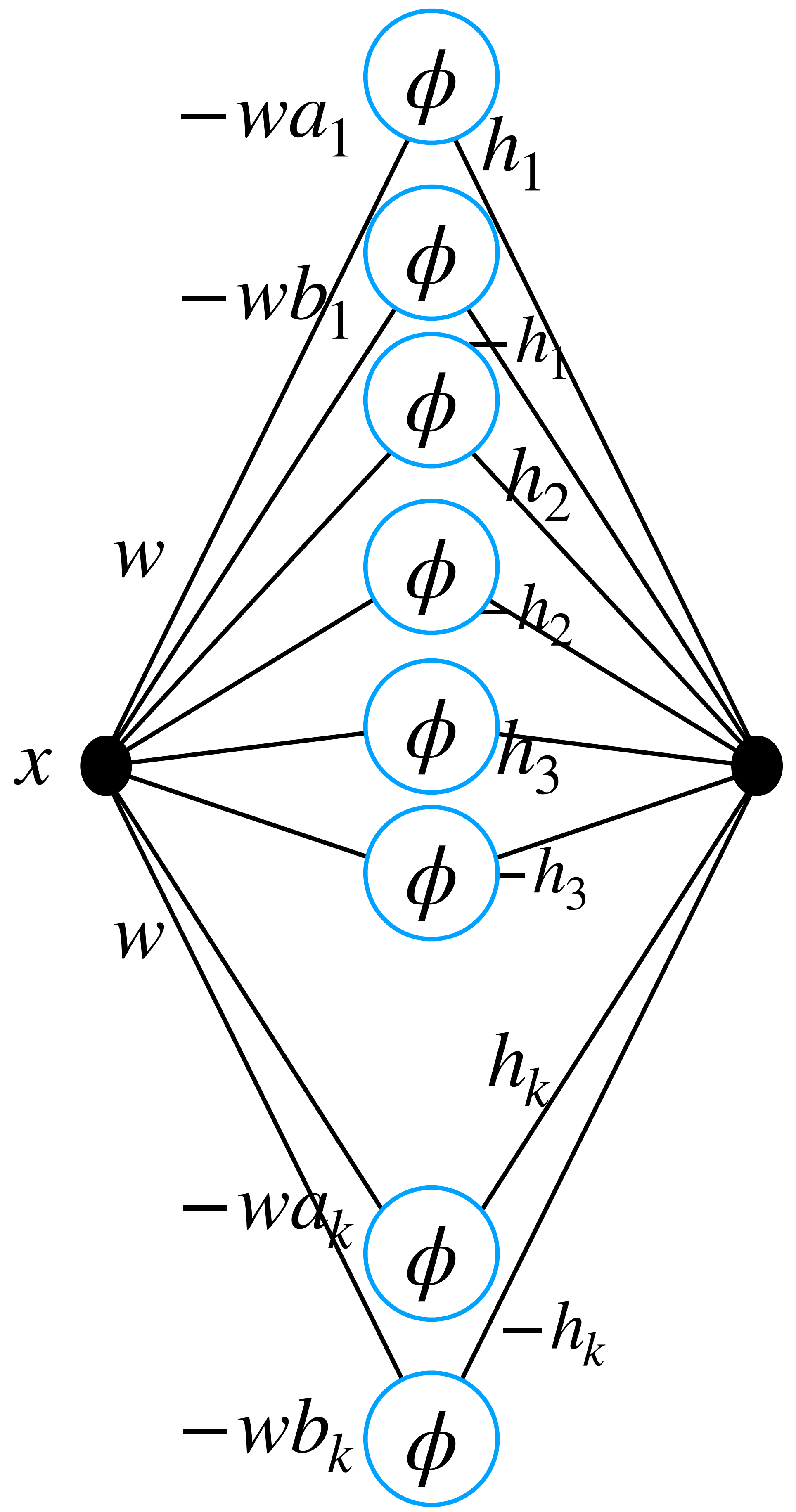
Conclusion in the 1D case

1. Approximate the function in the Riemann sense by a sum of k rectangles
2. Represent each rectangle using two nodes in the hidden layer of a neural network
3. Compute the sum of all nodes in the hidden layer (considering appropriate weights and signs) to get the final output

   ➡ NN with one hidden layer containing $2k$ nodes for a Riemann sum with k rectangles

<u>Remarks:</u>

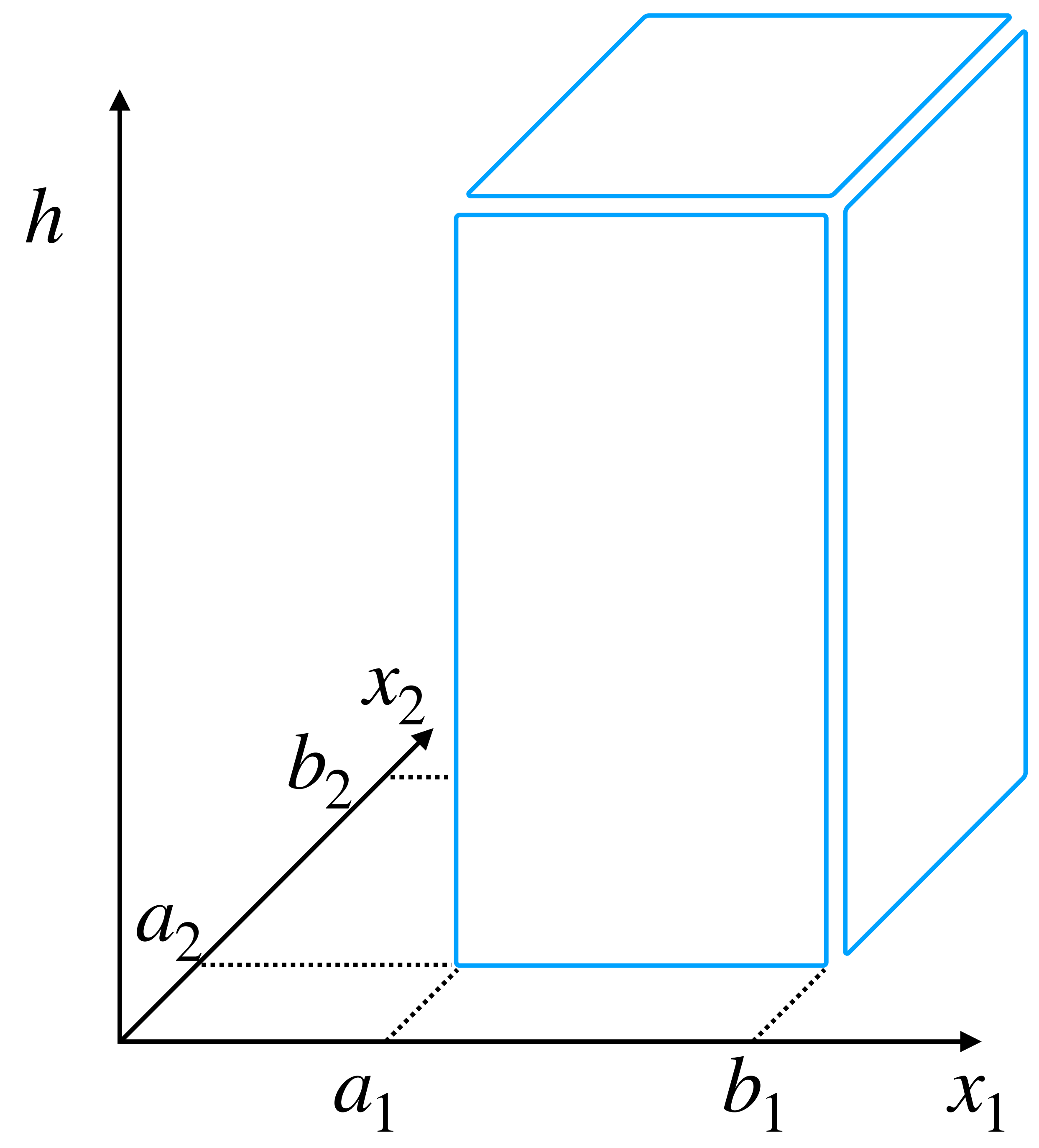
- The same intuition applies to any sigmoid-like function
- This is an intuitive explanation, not a quantitative one
- The weights w must be large



Larger dimension: $d = 2$

Same idea:

- Approximate the function by 2D rectangle functions
- Approximate a 2D rectangle function by sigmoids

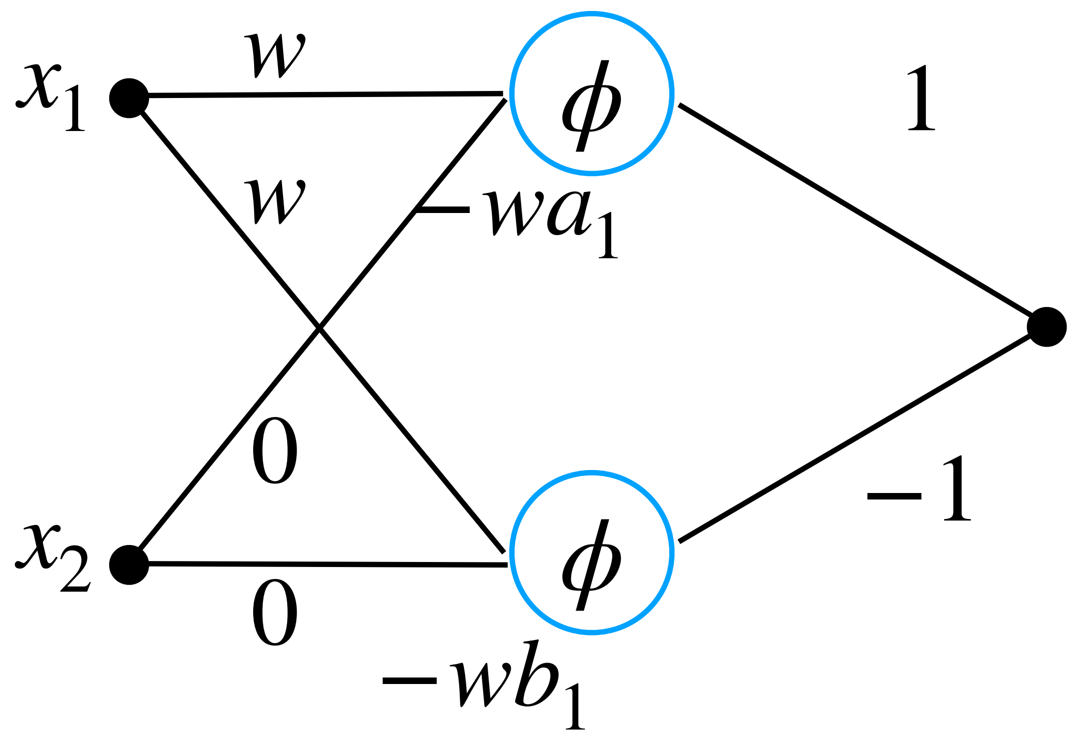


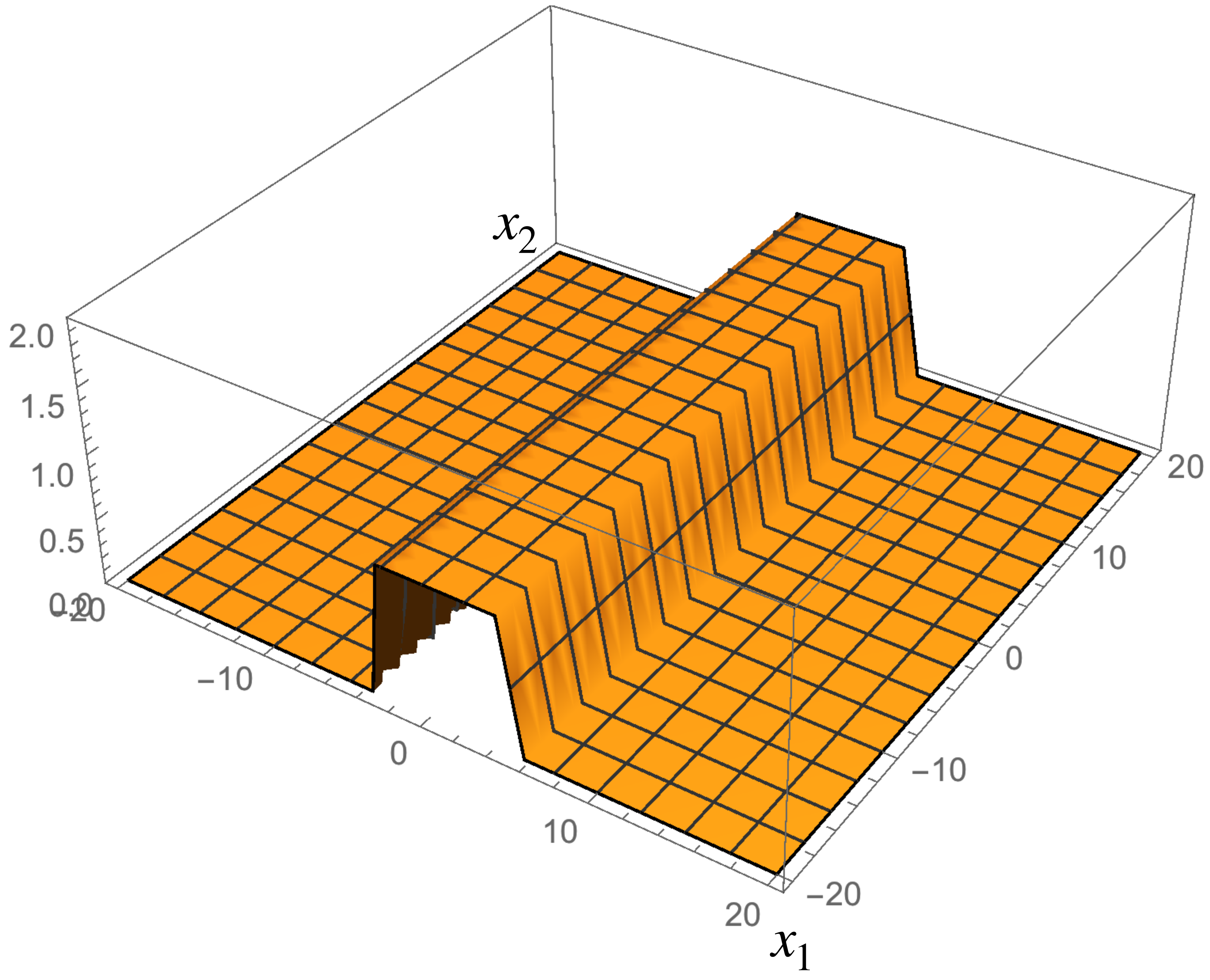
Two sigmoids can approximate an infinite rectangle function

$$(x_1, x_2) \mapsto \phi(w(x_1 - a_1)) - \phi(w(x_1 - b_1))$$

The rectangle:

- ranges from a_1 to b_1 in the x_1 direction
- is unbounded in the x_2 direction



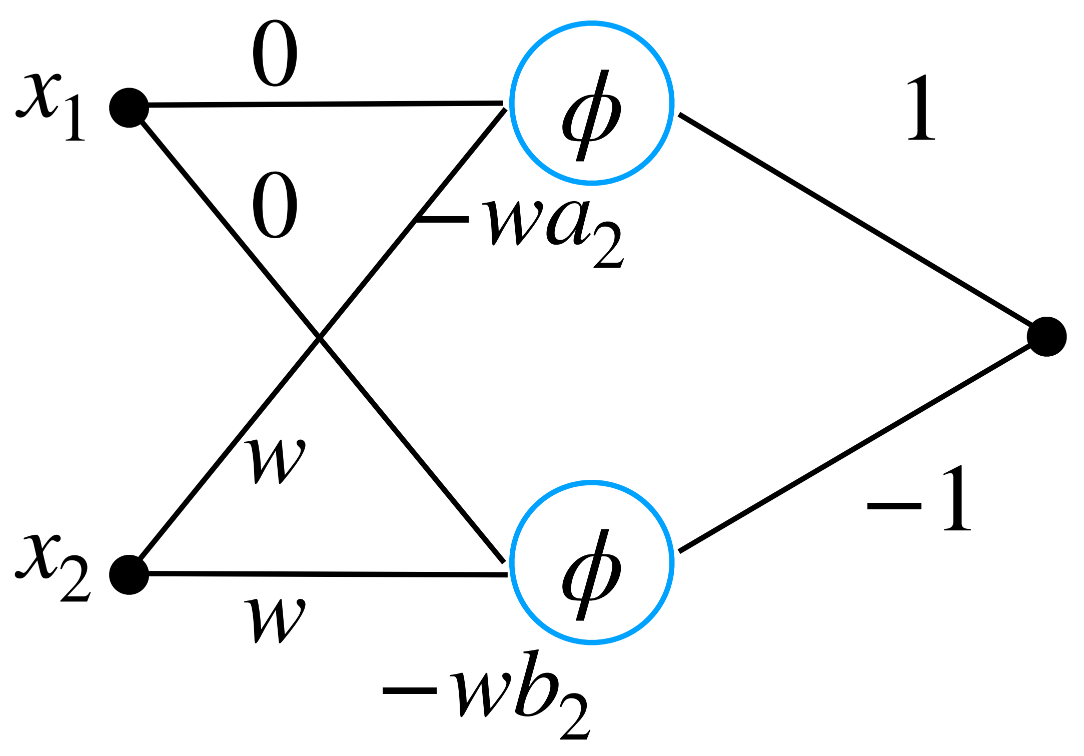


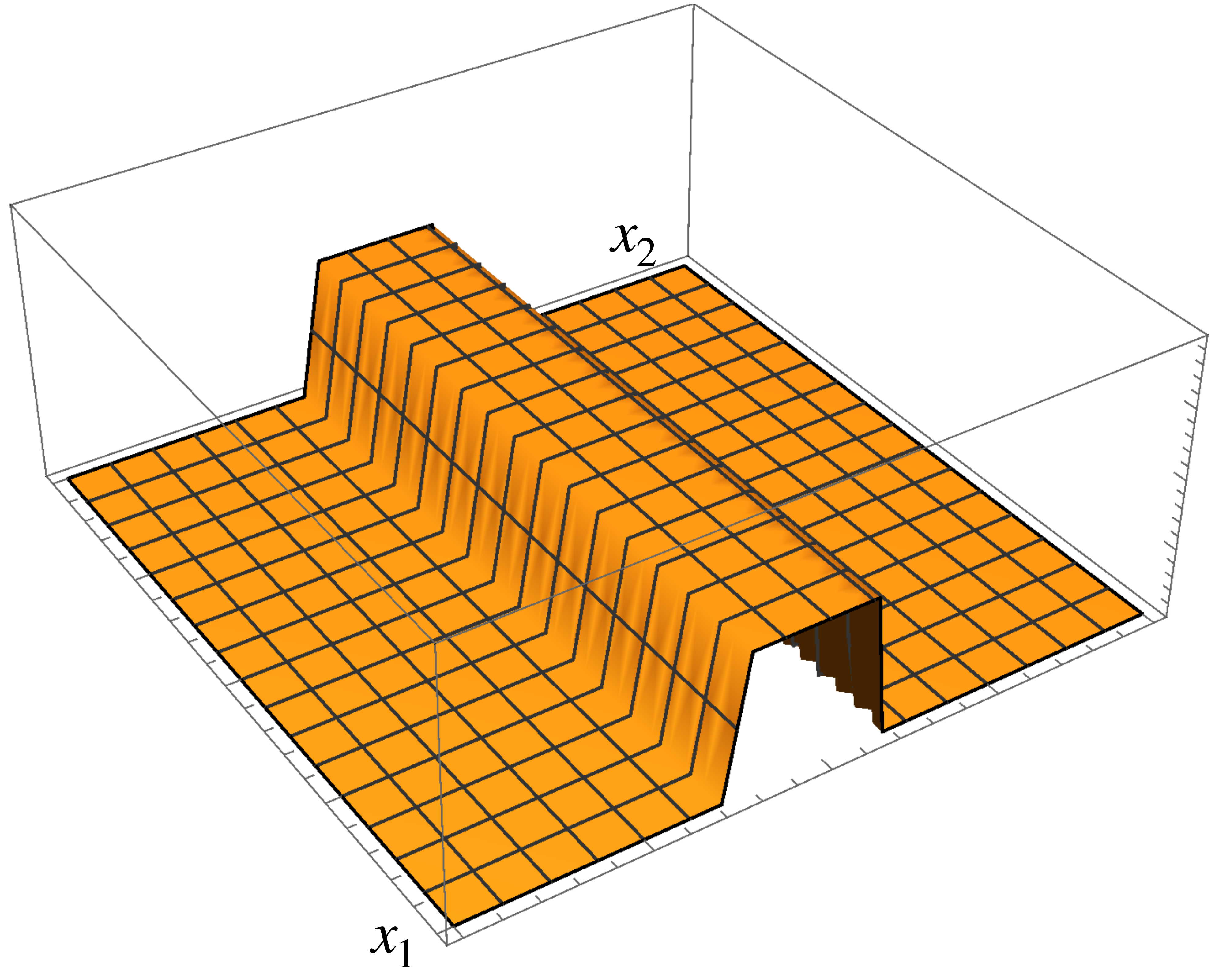
Two sigmoids can approximate an infinite rectangle function

$$(x_1, x_2) \mapsto \phi(w(x_2 - a_2)) - \phi(w(x_2 - b_2))$$

The rectangle:

- ranges from a_2 to b_2 in the x_2 direction
- is unbounded in the x_1 direction

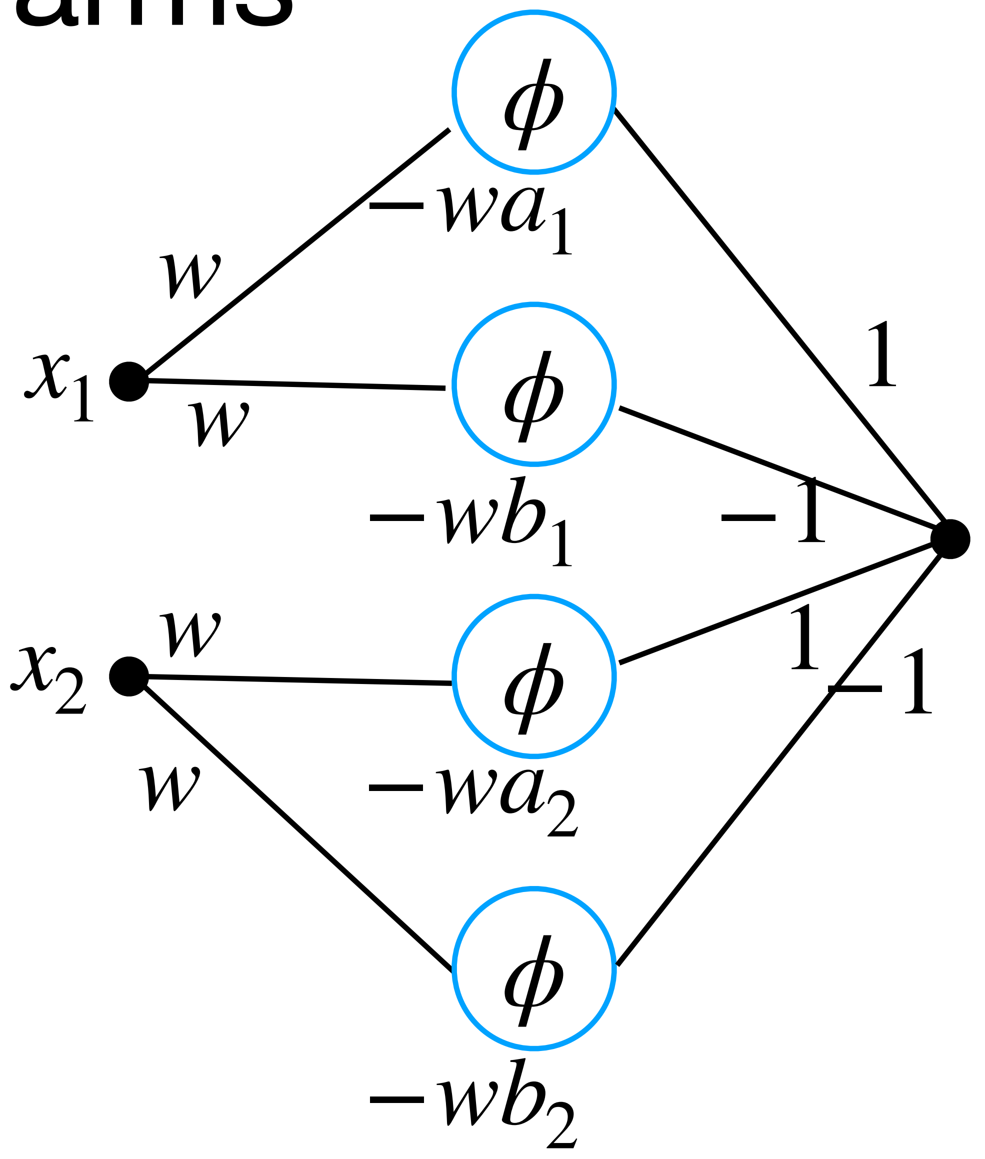


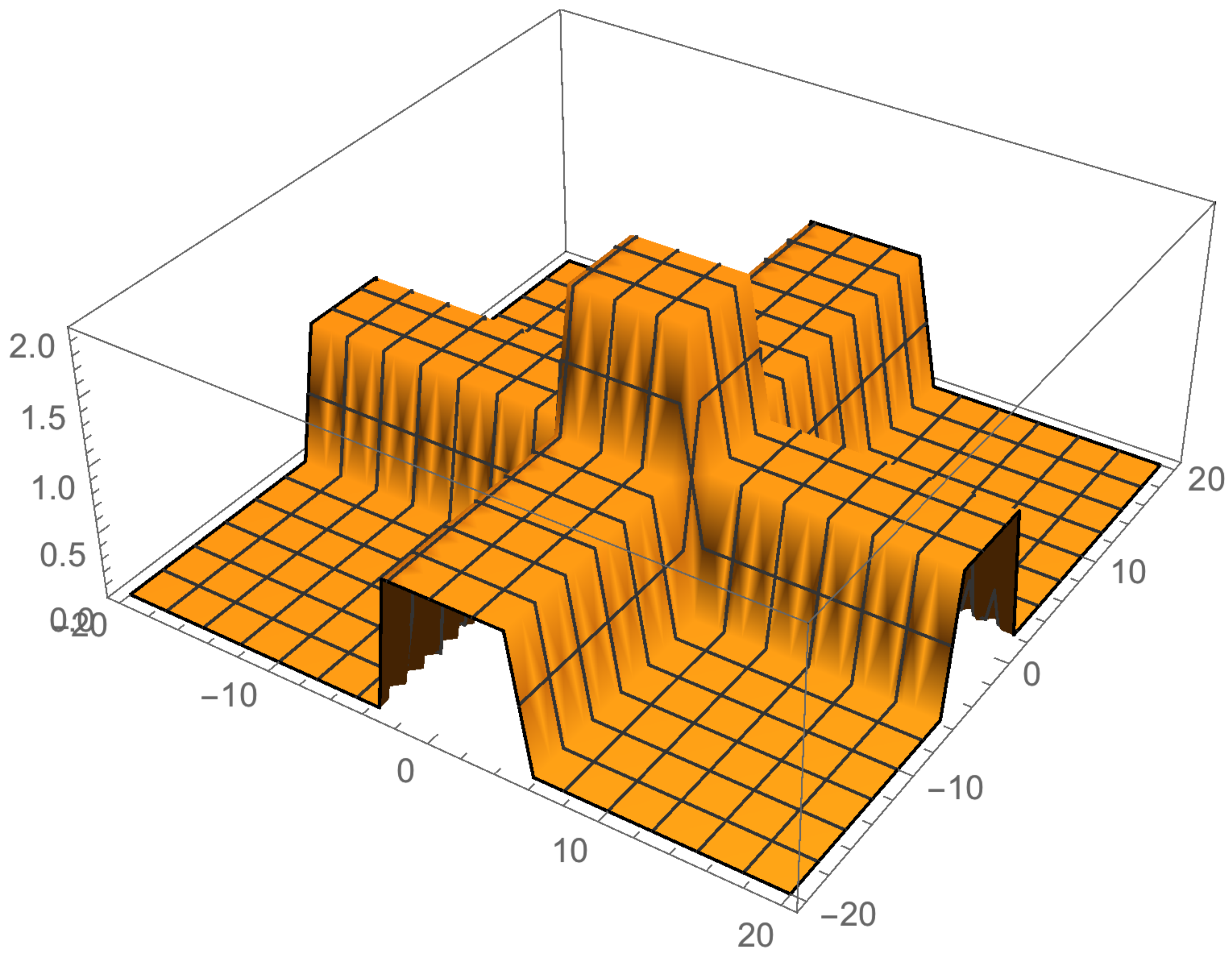


Four sigmoids approximate a cross

$$(x_1, x_2) \mapsto \phi(w(x_1 - a_1)) - \phi(w(x_1 - b_1)) + \phi(w(x_2 - a_2)) - \phi(w(x_2 - b_2))$$

➡ This approximation is close to our objective, with the exception of the two infinite “arms”





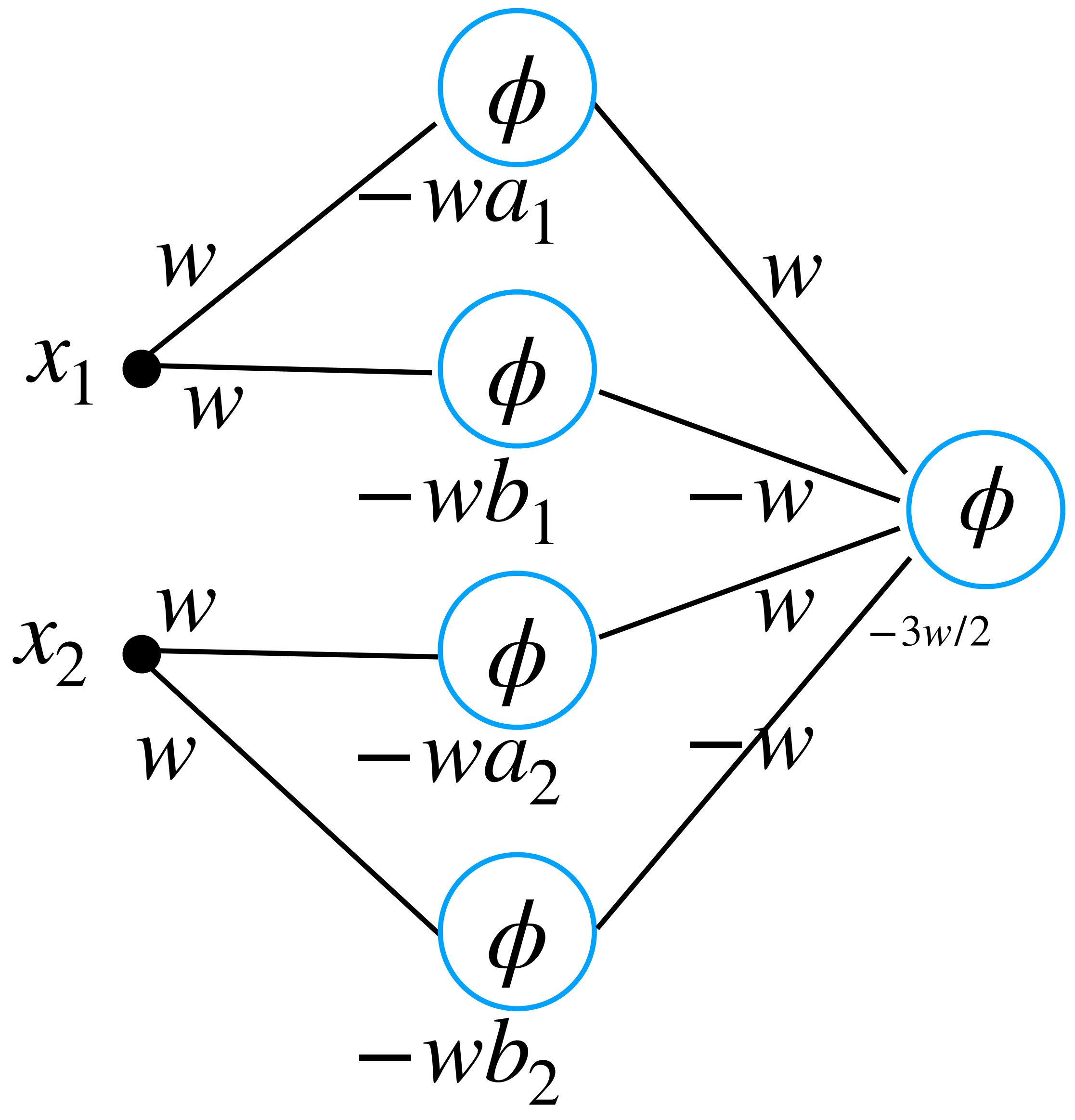
How can we eliminate the crossed arms?

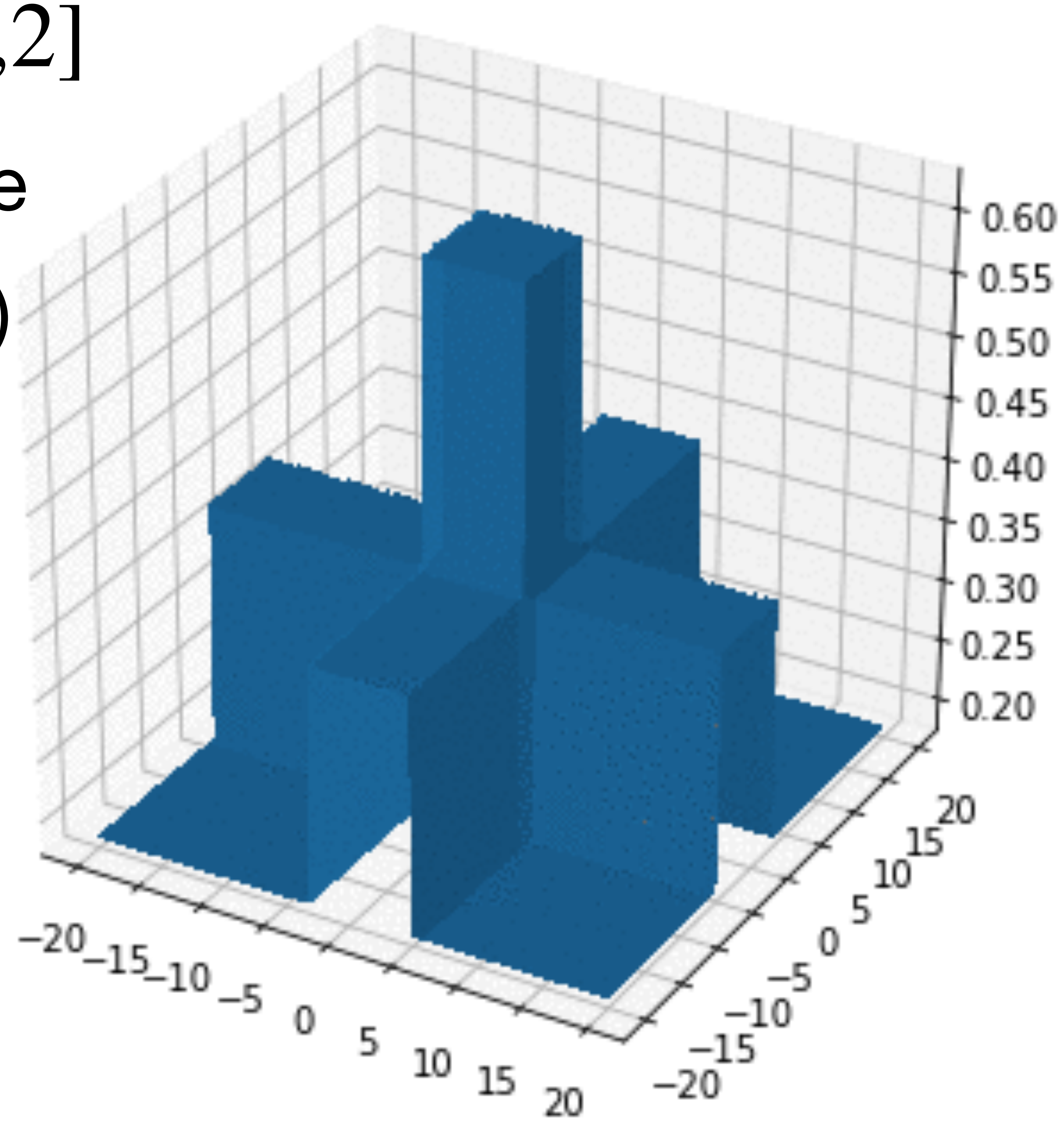
Using the sigmoid to threshold unwanted infinite arms

Thresholding the function will eliminate the arms

It is equivalent to composing it with $1_{y \geq c}$ for $c \in (1,2]$

- ➡ Approximate $1_{y \geq c}$ using a sigmoid with a large weight w and an appropriate bias (e.g., $3w/2$)





Point-wise approximations

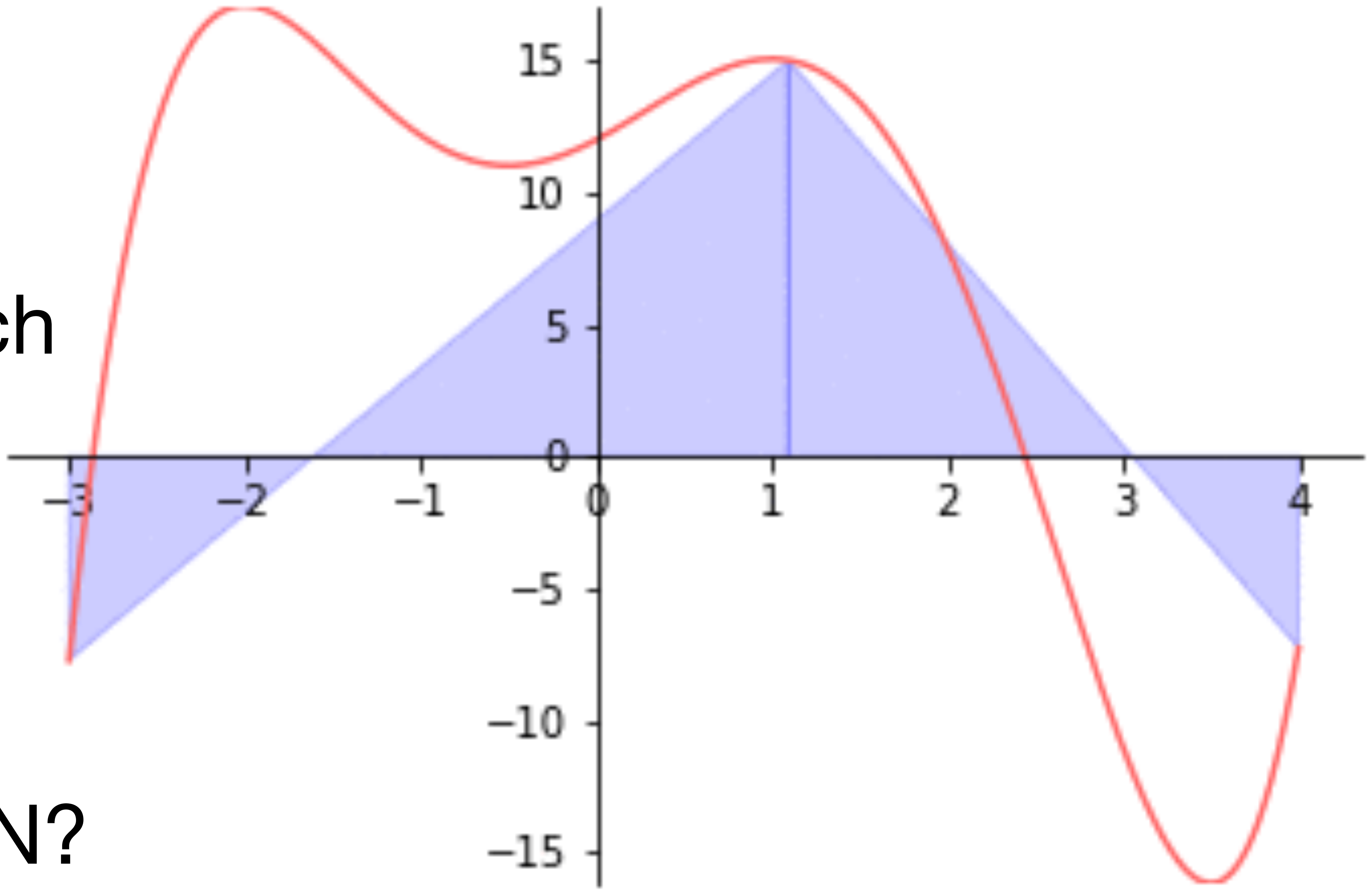
<u>Def</u>: piecewise linear (PWL) function:

$$q(x) = \sum_{i=1}^{m} (a_i x + b_i) 1_{r_{i-1} \leq x < r_i} \text{ with } a_i r_i + b_i = a_{i+1} r_i + b_{i+1}$$

<u>ℓ_∞-approximation result</u> (Shektman, 1982):

Let f be a continuous function on $[c, d]$. For all $\varepsilon > 0$, it exists a piecewise linear function q such that:

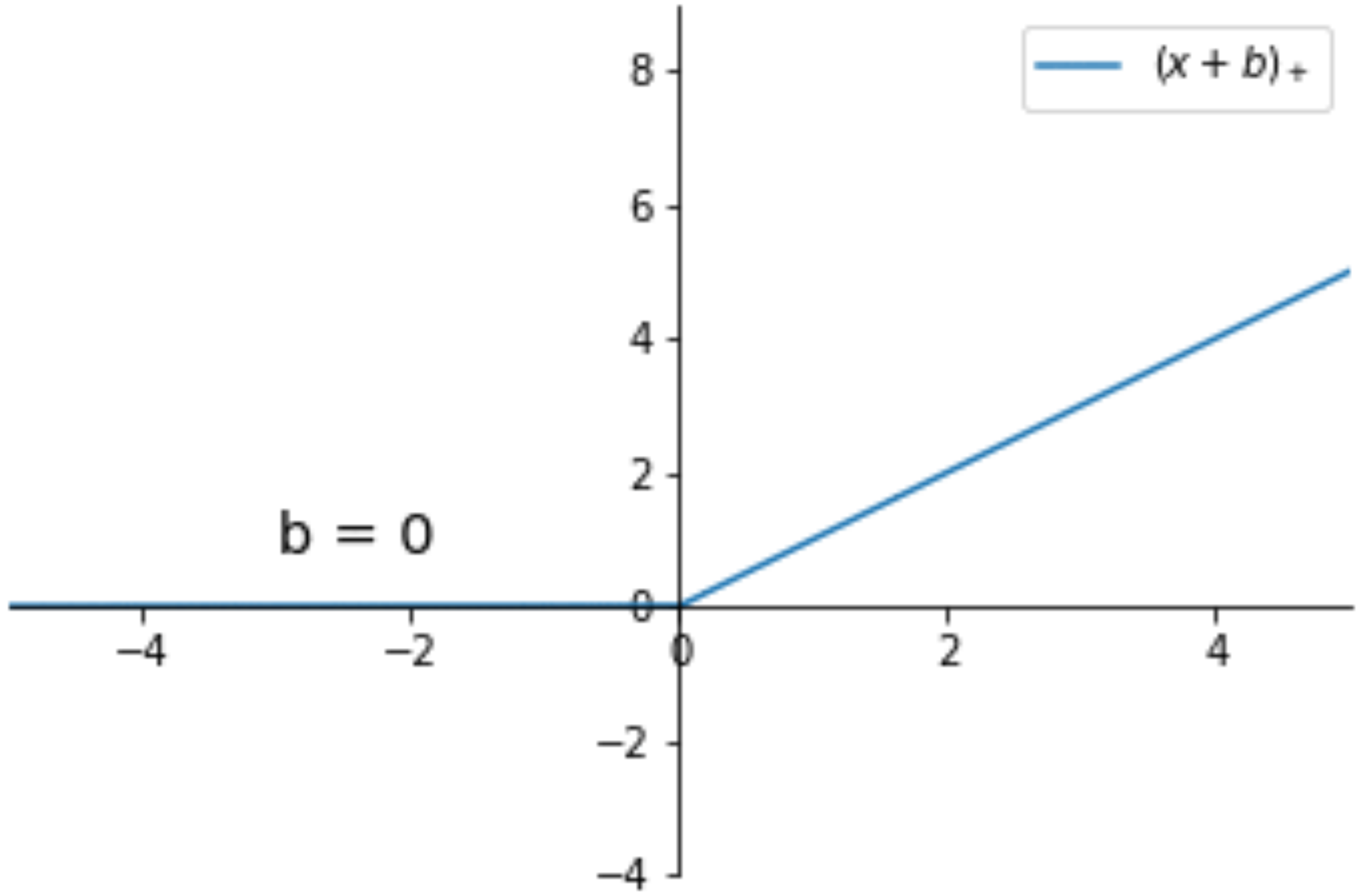
$$\sup_{x \in [c,d]} |f(x) - q(x)| \leq \varepsilon$$



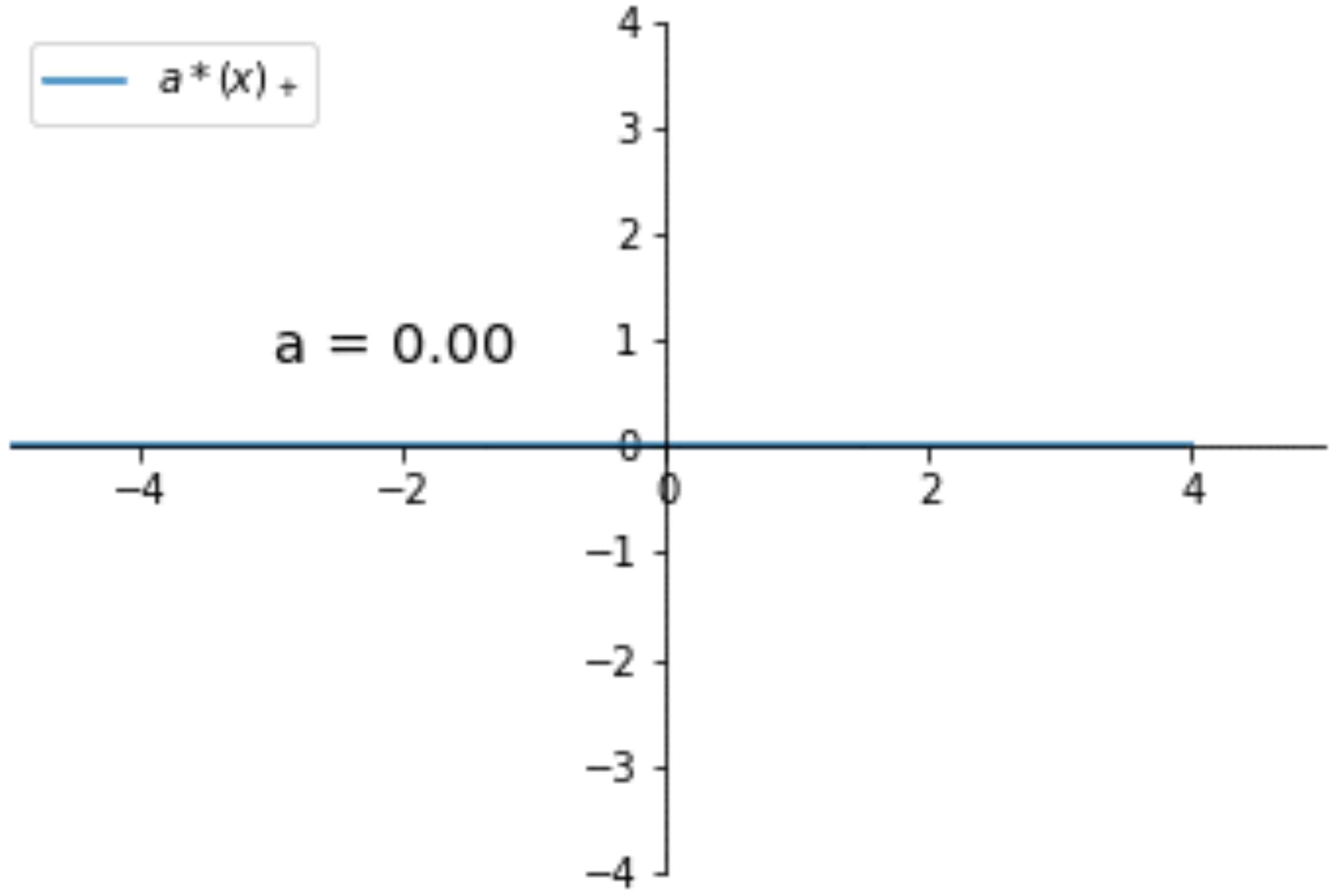
➡ How to approximate PWL functions with a NN?

Understanding the role of bias and weight in ReLU functions

$$(ax + b)_+ = \max\{0, ax + b\}$$



The bias b determines where the kink is



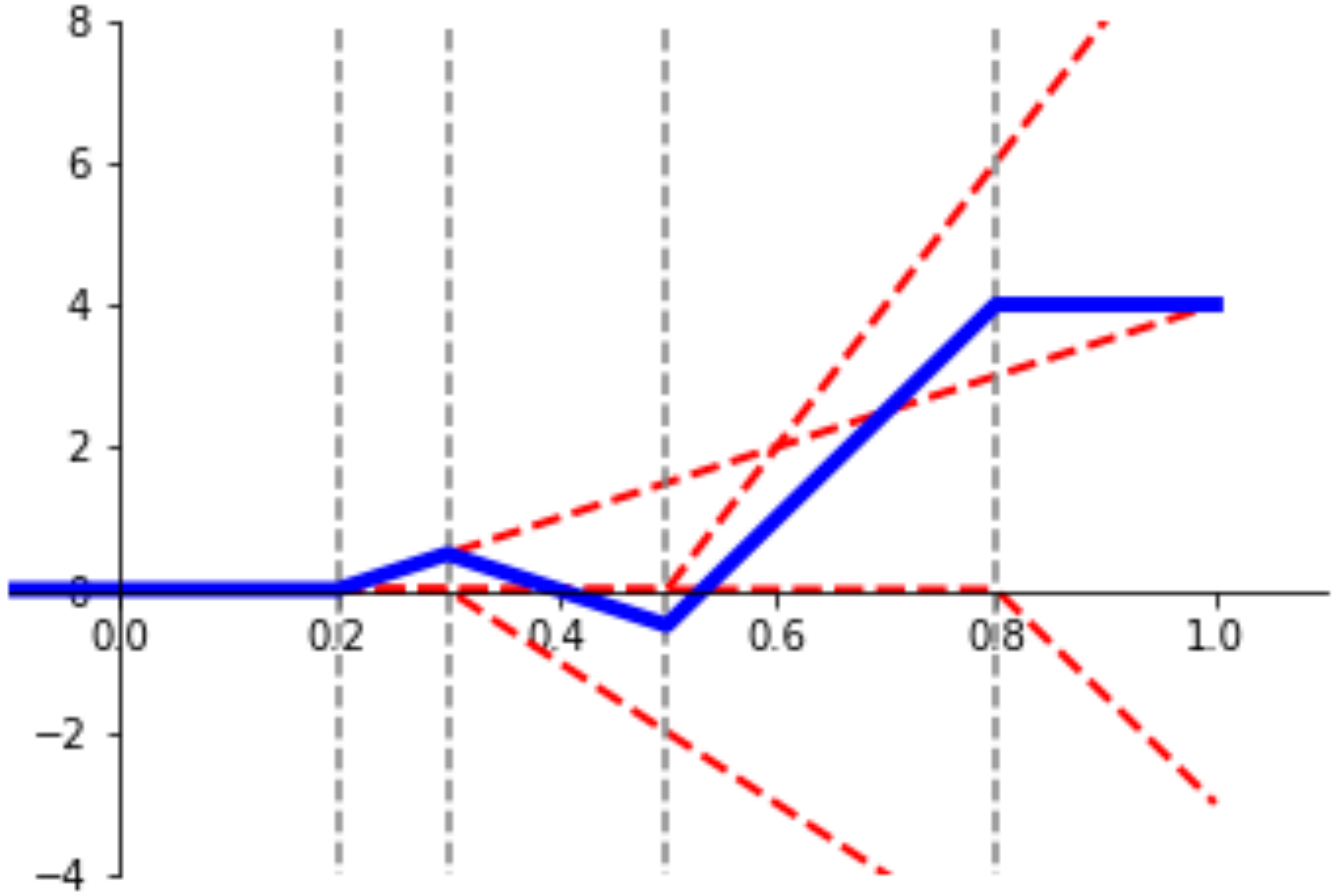
The weight a determines the slope

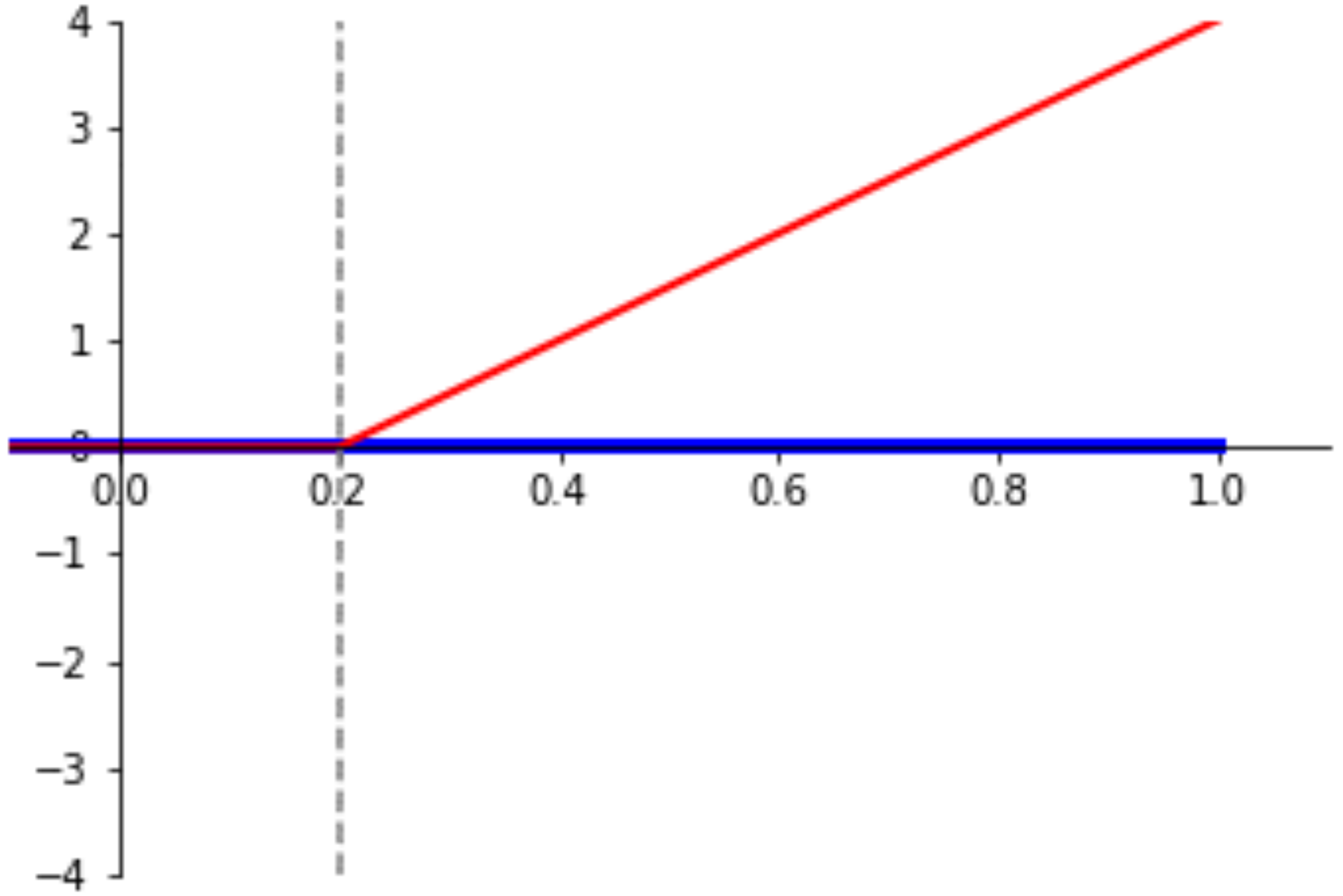
Linear combinations of RELUs and PWL functions

$\sum_{i=1}^{m} \tilde{a}_i (x - \tilde{b}_i)_+$ is a piecewise linear function

How do we get a new segment with slope a starting at $r > \max_i(\tilde{b}_i)$?

Intuition: Get the kink at r by setting $\tilde{b}_{i+1} = r$ and slope by additionally canceling existing slope i.e. $\tilde{a}_{i+1} = a - \sum_i \tilde{a}_i$





Formal: Piecewise linear functions can be written as combination of RELU

Claim 1: Any PWL q can be rewritten as

$$q(x) = \tilde{a}_1 x + \tilde{b}_1 + \sum_{i=2}^{m} \tilde{a}_i (x - \tilde{b}_i)_+$$

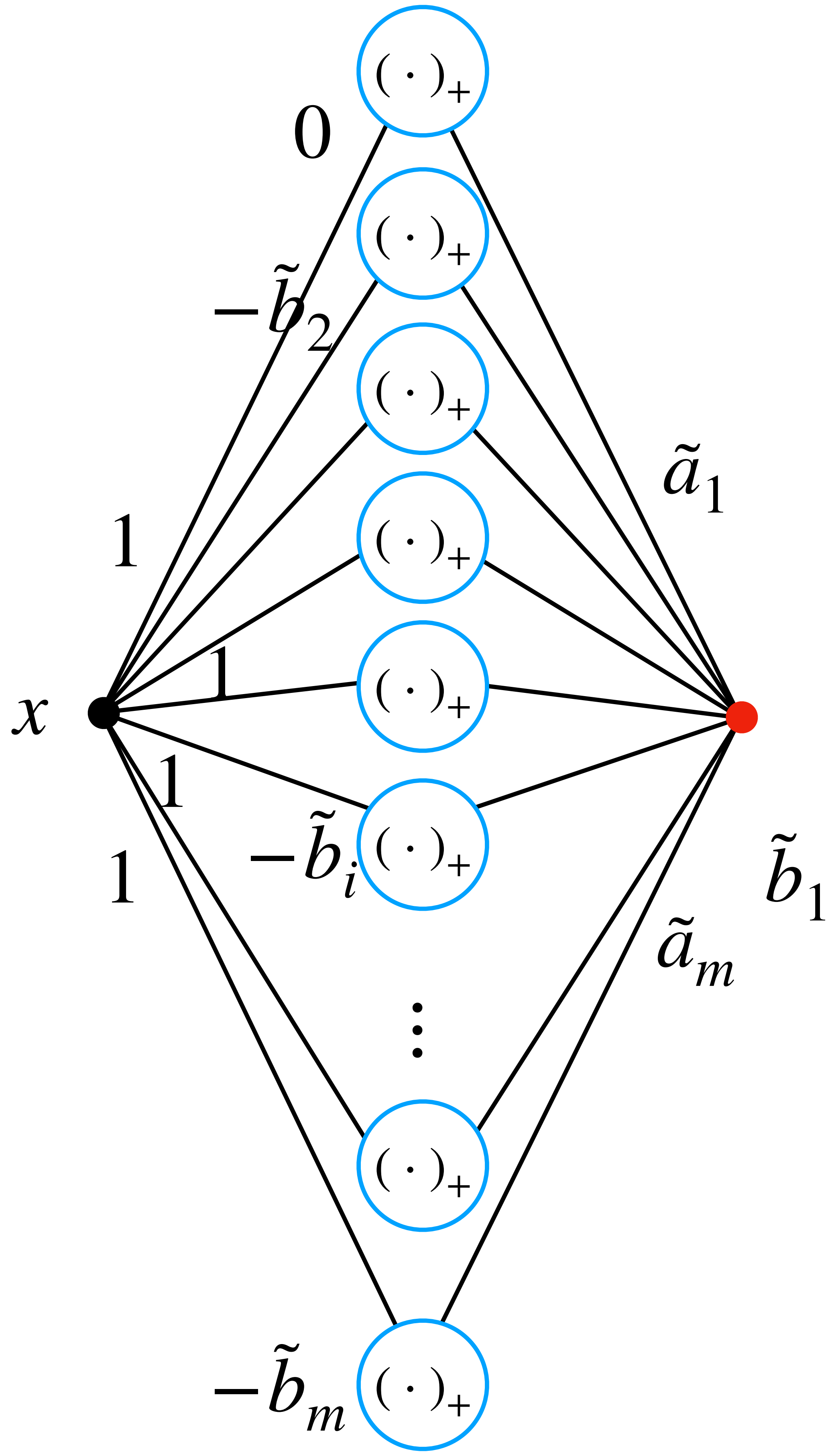
where $\tilde{a}_1 = a_1, \tilde{b}_1 = b_1, \; a_i = \sum_{j=1}^{i} \tilde{a}_i$ and $\tilde{b}_i = r_{i-1}$

Claim 2: q can be implemented as a one-hidden-layer NN with RELU activation. Each term corresponds to one node:

- Bias $-\tilde{b}_i$
- Output weight $\tilde{a}_i$

The term $\tilde{a}_1 x + \tilde{b}_1$ also corresponds to one node:

- Bias $\tilde{b}_1$: bias of the output node
- Term $\tilde{a}_1 x = \tilde{a}_1 (x)_+$ since $x \in [0,1]$



Proof of the equivalent formulation

$$q(x) = \sum_{i=1}^{m} (a_i x + b_i) 1_{r_{i-1} \le x < r_i} \quad r(x) = \tilde{a}_1 x + \tilde{b}_1 + \sum_{i=2}^{m} \tilde{a}_i (x - \tilde{b}_i)_+$$

$$\tilde{a}_1 = a_1, \tilde{b}_1 = b_1 \text{ and } a_i = \sum_{j=1}^{i} \tilde{a}_j \text{ and } \tilde{b}_i = r_{i-1}$$

- For $x \in [0, r_1]$

  $(\tilde{a}_1, \tilde{b}_1) = (a_1, b_1) \implies q(x) = a_1 x + b_1 = \tilde{a}_1 x + \tilde{b}_1 = r(x)$ because $\tilde{b}_2 = r_1$

- For $x \in [r_1, r_2]$, $r(x) = \tilde{a}_1 x + \tilde{b}_1 + (a_2 - a_1)(x - r_1)_+$

  $= a_1 x + b_1 + (a_2 - a_1)(x - r_1) = a_2 x + b_1 - (a_2 - a_1) r_1$

  $r'(x) = a_2$ and $r(r_1) = q(r_1)$ as shown above

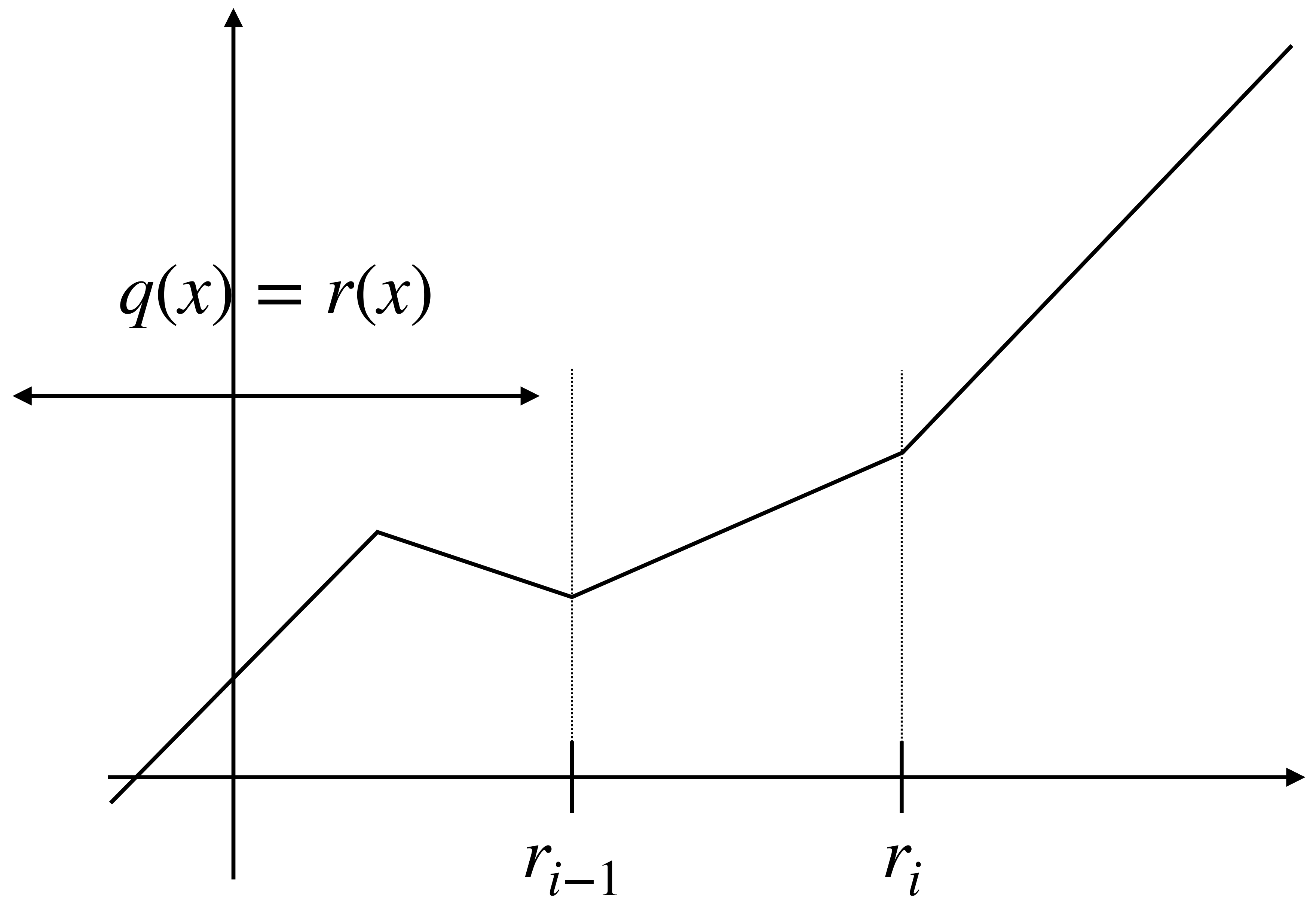
  $\implies r(x) = q(x)$ for $x \in [r_1, r_2]$

Proof by induction

Let's assume that $r(x) = q(x)$ for $x \in [0, r_{i-1}]$

For $x \in [r_{i-1}, r_i]$

$$r(x) = \tilde{a}_1 x + \tilde{b}_1 + \sum_{j=2}^{m} \tilde{a}_j (x - \tilde{b}_j)_+$$
$$= \tilde{a}_1 x + \tilde{b}_1 + \sum_{j=2}^{i} \tilde{a}_j (x - \tilde{b}_j)$$
$$= \sum_{j=1}^{i} \tilde{a}_j x + \tilde{b}_1 - \sum_{j=2}^{j} \tilde{a}_j \tilde{b}_j$$



Thus

- $r'(x) = \sum_{j=1}^{i} \tilde{a}_j = a_i$ good slope
- $r(r_{i-1}) = q(r_{i-1})$ good starting point

$\implies r(x) = q(x)$ for $x \in [r_{i-1}, r_1]$

Why: two affine functions with the same starting point and the same slope are equal

Recap

- Neural networks consist of **linear layers** stacked together with **non-linearities**

- A neural network can be seen as a **learned feature extractor** + **linear prediction**

- Neural networks typically require large amounts of **data** and **compute** to learn good features

- Neural networks have very high **representational power** (i.e., the universal approximation result) in contrast to simple models like linear regression