Universität Konstanz

03 | Multilayer Perceptron

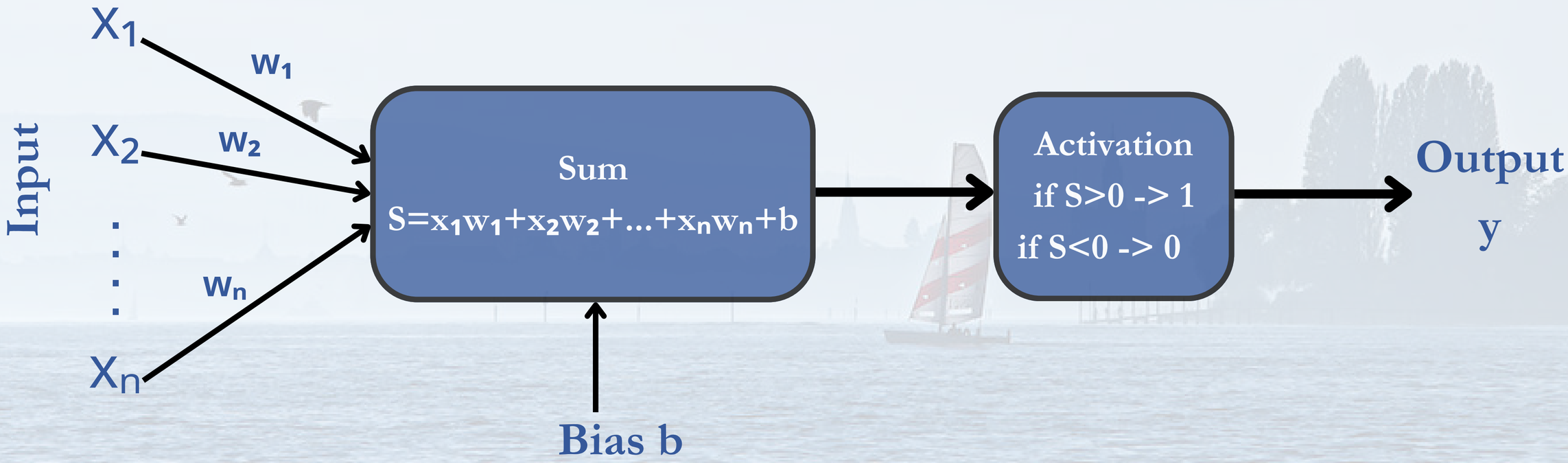
Giordano De Marzo

https://giordano-demarzo.github.io/

Deep Learning for the Social Sciences

Structure of the Perceptron

The Perceptron combines weighted inputs and activation:



Mathematical Notation

Mathematically we can represent the perceptron using a vector product

$y=a[\mathbf{wx}+b]$

Here

- $\mathbf{w}=(w_1, w_2, ..., w_n)$ is the vector containing the n weights
- $\mathbf{x}=(x_1, x_2, ..., x_n)$ is the vector containing the n-dimensional input
- a is the activation function. In our case
  - $a(x)=1$ if $x>0$
  - $a(x)=0$ if $x<0$
- b is the bias

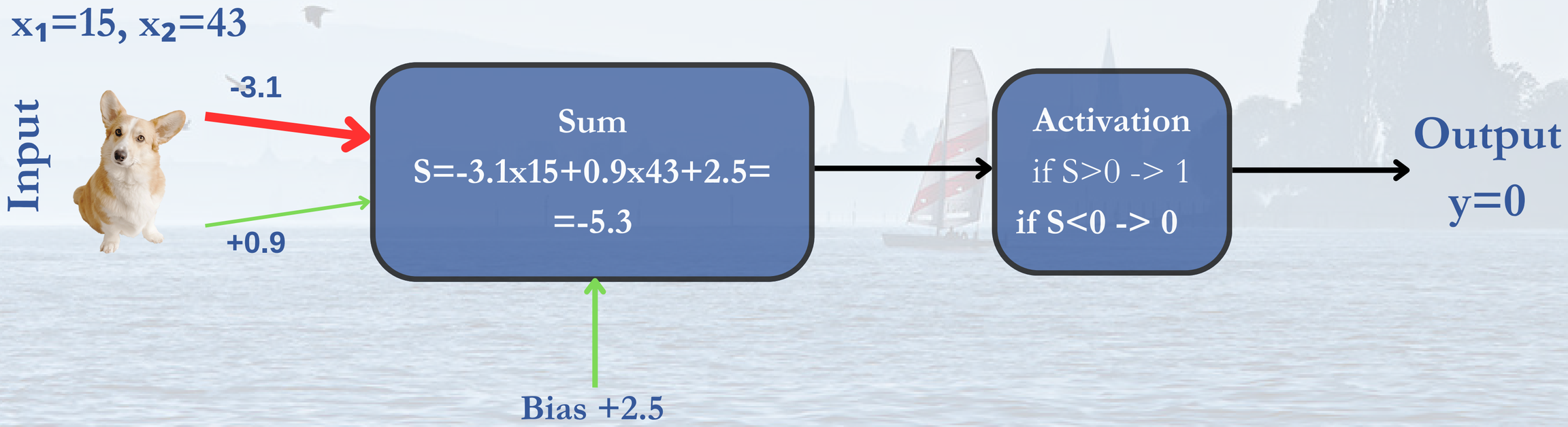
With **wx** we denote the scalar product of the two vectors

$\mathbf{wx}=x_1w_1+x_2w_2+...+x_nw_n$

Perceptron with 2 Inputs

Let's see a practical example plugging some numbers in the perceptron

- we use as parameters w_1=-3.1, w_2=0.9 and b=2.5
- the input is **x**=(15, 43)



Universität Konstanz

Outline

1. Limits of the Perceptron
2. Shallow Neural Networks
3. Multilayer Perceptron Architecture

Universität
Konstanz

Limits of the Perceptron

Understanding the Perceptron

The Perceptron creates a linear decision boundary to separate categories.

Cat: $w_1x_1 + w_2x_2 + b > 0$

Dog: $w_1x_1 + w_2x_2 + b < 0$

The decision boundary itself is defined by:

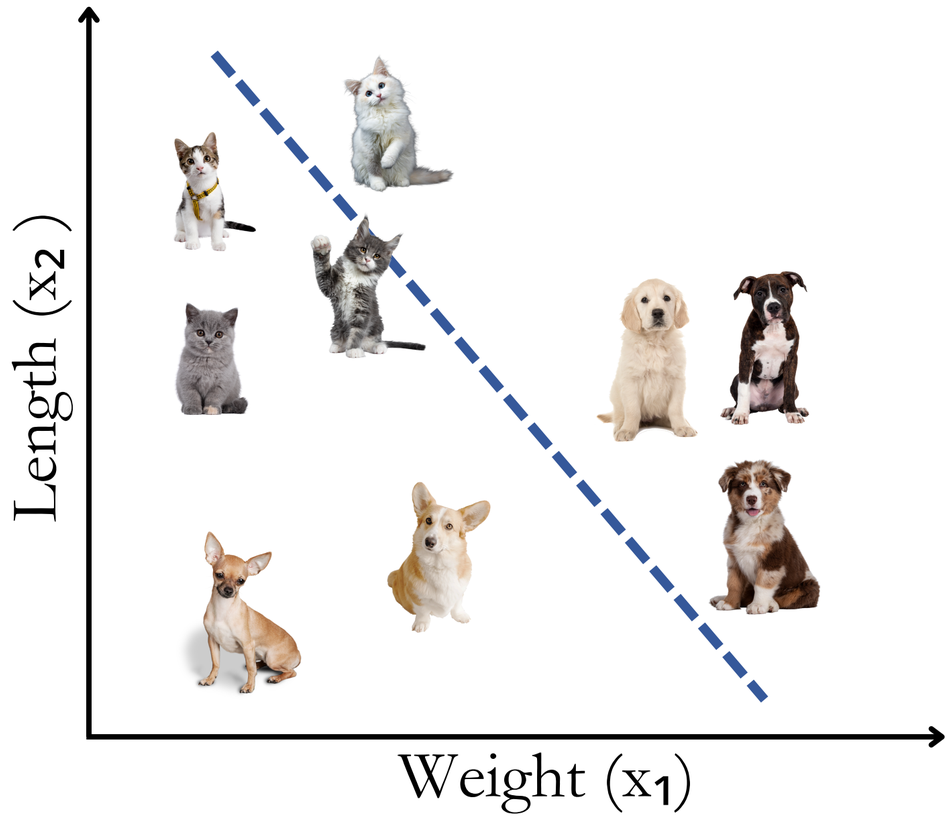
$w_1x_1 + w_2x_2 + b = 0$

Which can be rewritten as:

$x_2 = -(w_1/w_2)x_1 - (b/w_2)$

This demonstrates that the Perceptron:

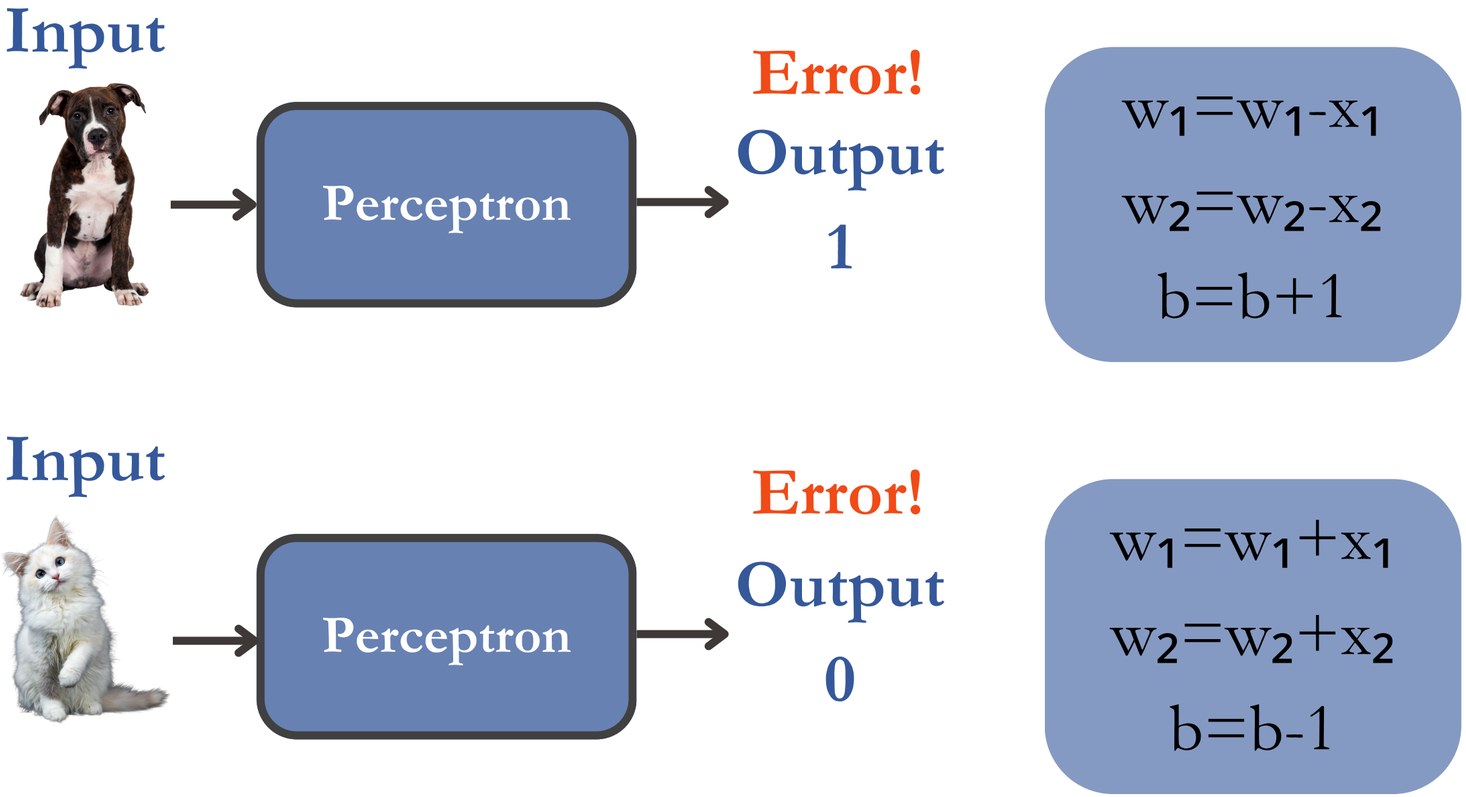
- Draws a straight line in a 2D feature space
- Classifies points above the line as cats
- Classifies points below the line as dogs



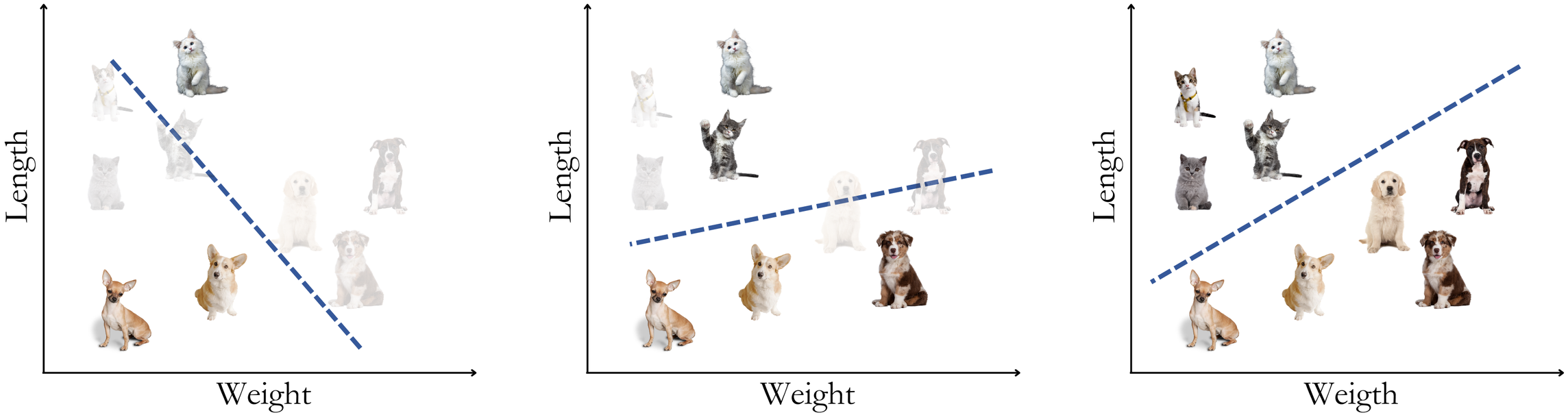
https://giordano-demarzo.github.io/teaching/deep-learning-25/perceptron/

Training Rule

During training the perceptron is shown labelled data and its weights are adjusted when it produces wrong classifications



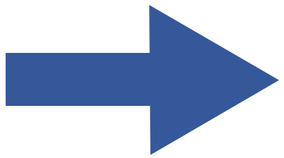
Visualizing Training

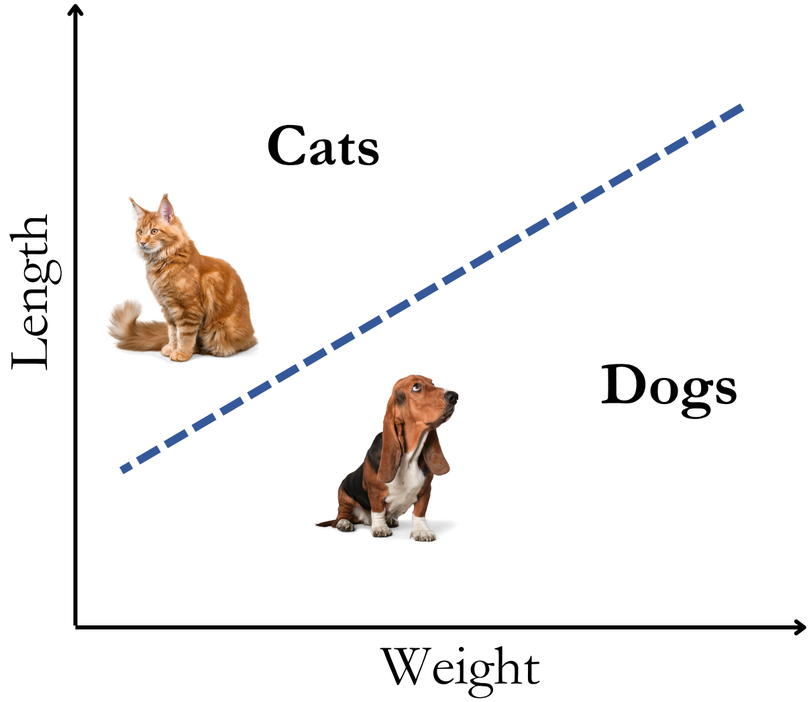


As training progresses, the decision boundary moves to better separate the classes.

Classification

Once trained, the Perceptron can classify new animals from their weight and length





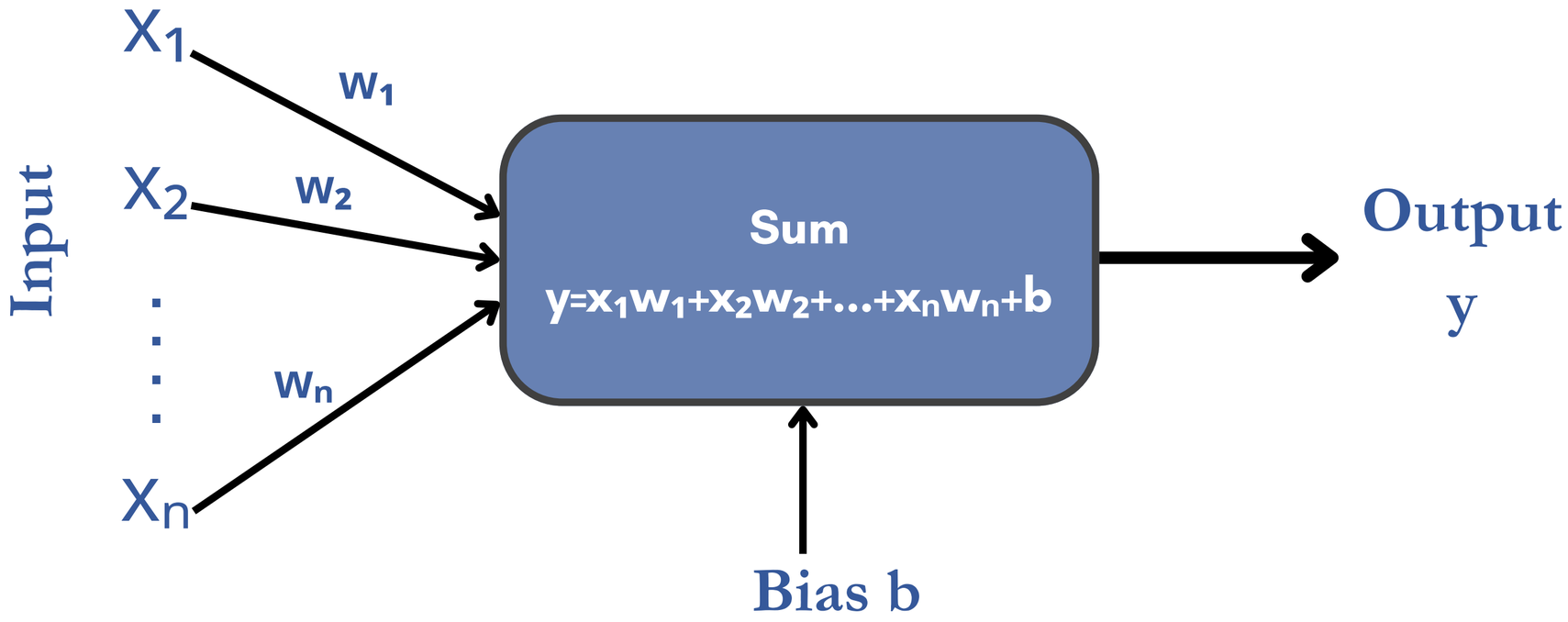
Cat

Dog

Perceptron and Regression

The Perceptron can also perform regression

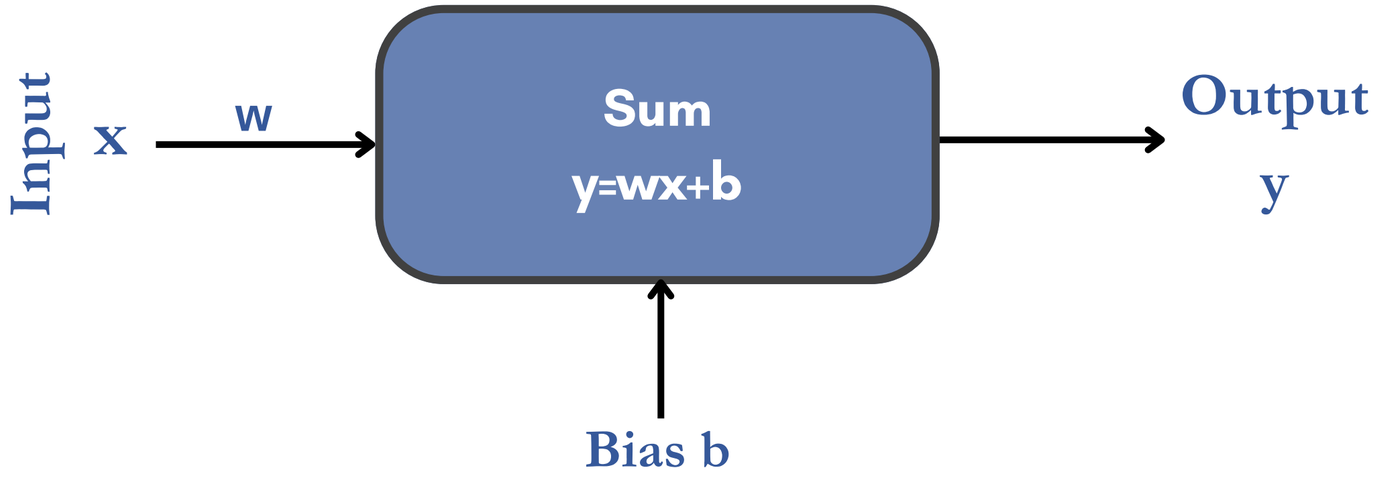
- Predicts a continuous numerical value instead of a category
- Skips the thresholding step in the activation function

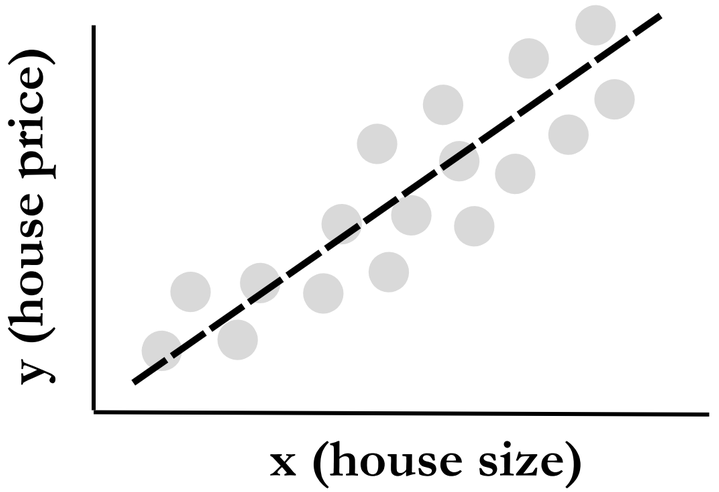


Single Input Case

In the most simple case we have a single input

- the model output is y=wx+b
- during training the model learns w and b to fit the data
- this is equivalent to a linear regression





Activation Functions

Step Function

- Used as output for the classification task
  - a(x)=1 if x>0
  - a(x)=0 if x<0

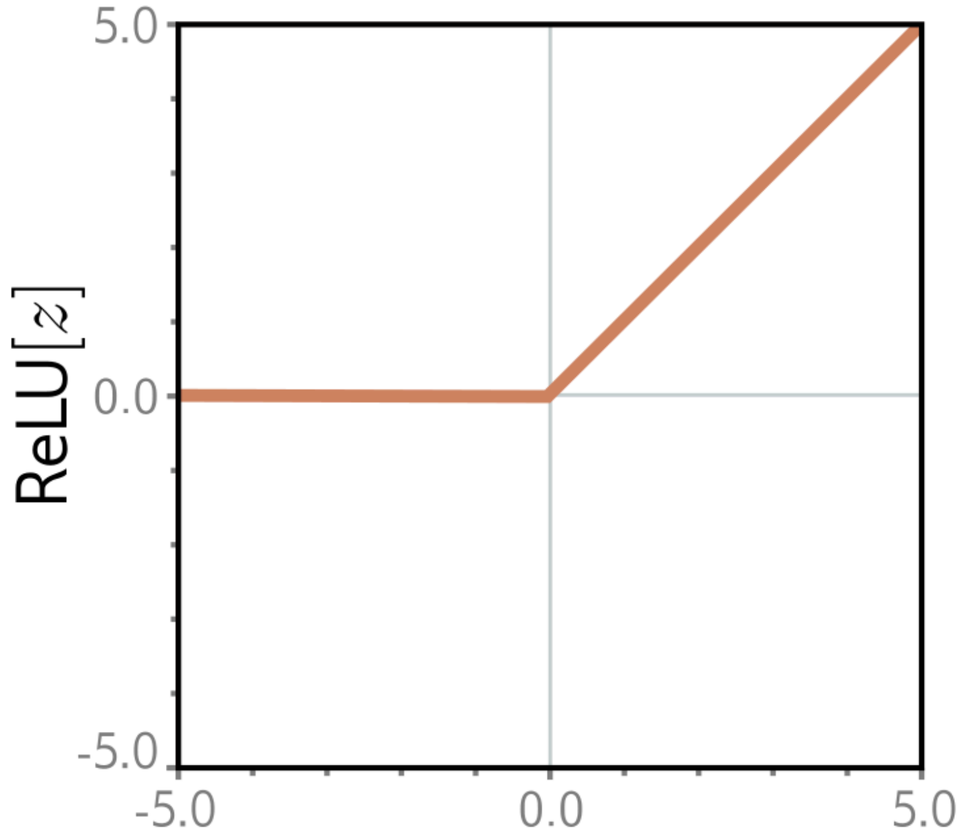
Linear Activation

- Used as output for the regression task
  - a(x)=x

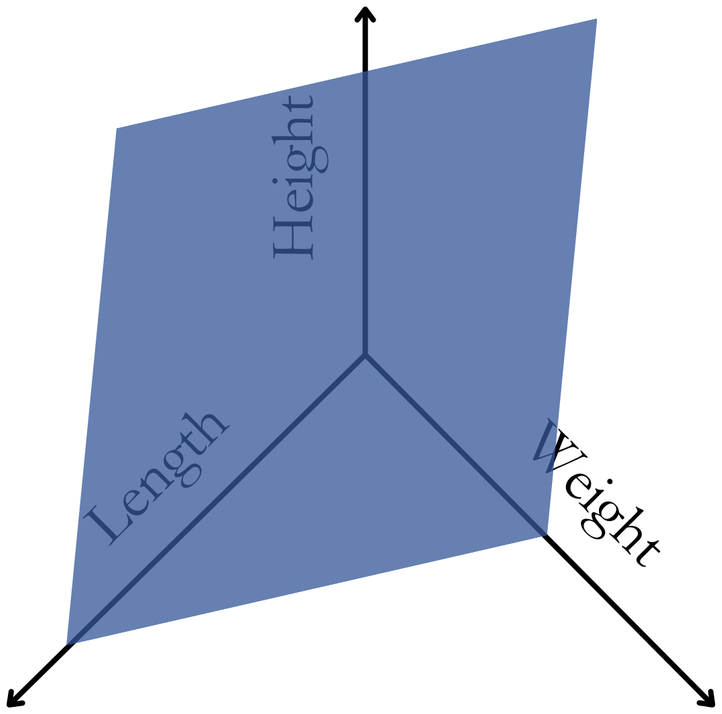
The Rectified Linear Unit (ReLU) is another example

ReLU

- Used in hidden layers of deep neural networks
  - a(x)=x if x>0
  - a(x)=0 if x<0



Higher Dimensions



The Perceptron's principles extend to higher dimensions:

- 2 Dimensions: The Perceptron uses a line to separate categories (e.g., dogs and cats based on weight and length)
- 3 Dimensions: Adding another feature (e.g., height) creates a 3D space where the Perceptron uses a plane as separator

What happens in higher dimensions?

Limits of the Perceptron

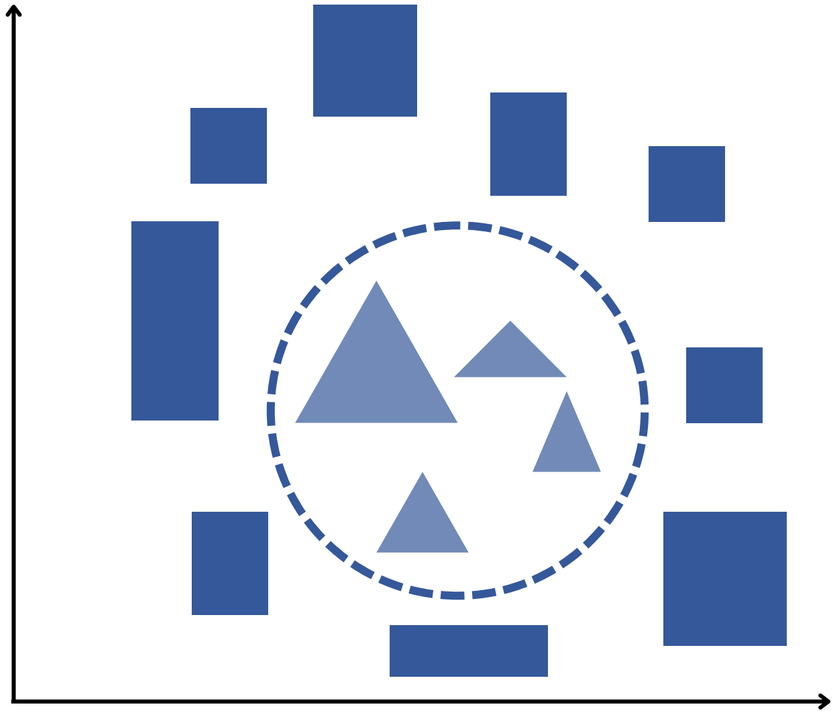
The Perceptron works well when data categories can be separated by a line (in 2D), a plane (in 3D), or a hyperplane (in higher dimensions).
However, many real-world problems aren't linearly separable:

- If data forms patterns like circles or spirals
- If categories are intermingled in complex ways

In these cases, a single Perceptron is insufficient!

- Problems like XOR cannot be solved by a single Perceptron

This limitation led to the so called **AI Winter**



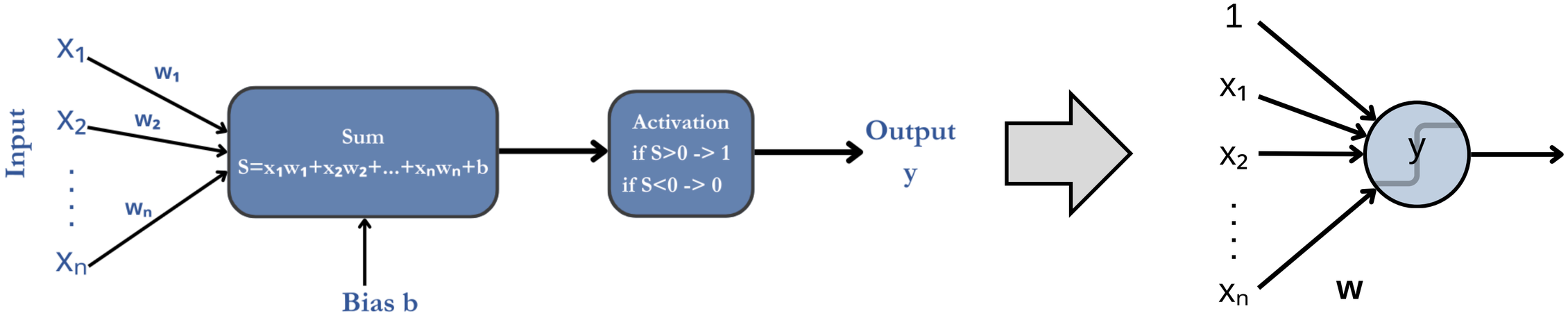
Universität
Konstanz

Shallow Neural Networks

Another Representation

In the following we will use a more simple representation for the perceptron

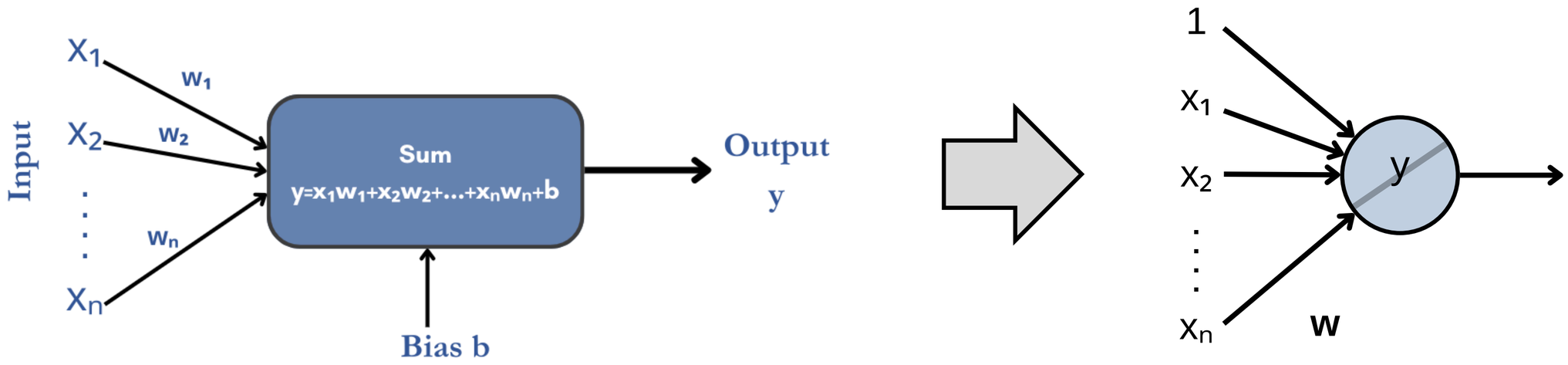
- we combine the weights and the bias in the same vector
  - $w=(b, w_1, w_2, ..., w_n)$
- we add a dummy input that is always 1 and that gets multiplied by the bias
- we write on the neuron the name of its output and we plot its activation on it



Another Representation

In the following we will use a more simple representation for the perceptron

- we combine the weights and the bias in the same vector
  - $w=(b, w_1, w_2, ..., w_n)$
- we add a dummy input that is always 1 and that gets multiplied by the bias
- we write on the neuron the name of its output and we plot its activation on it

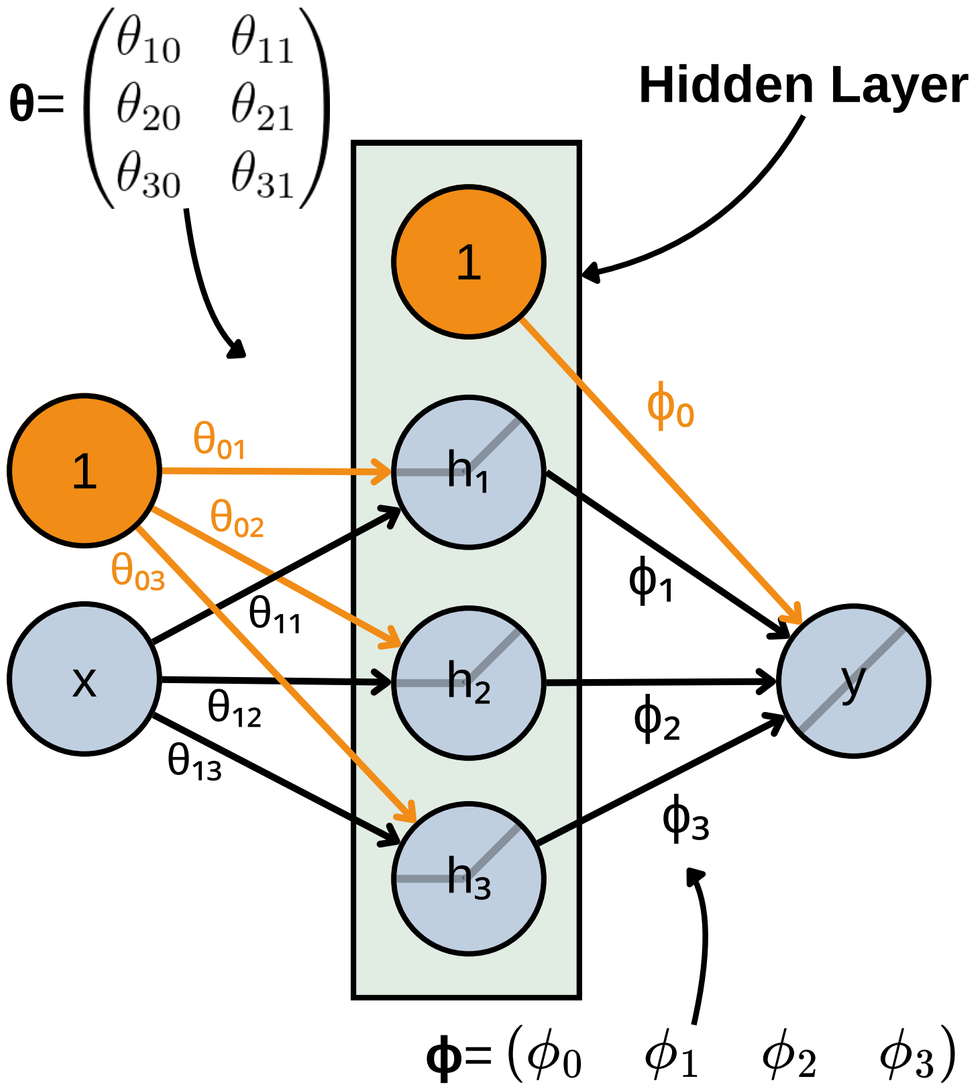


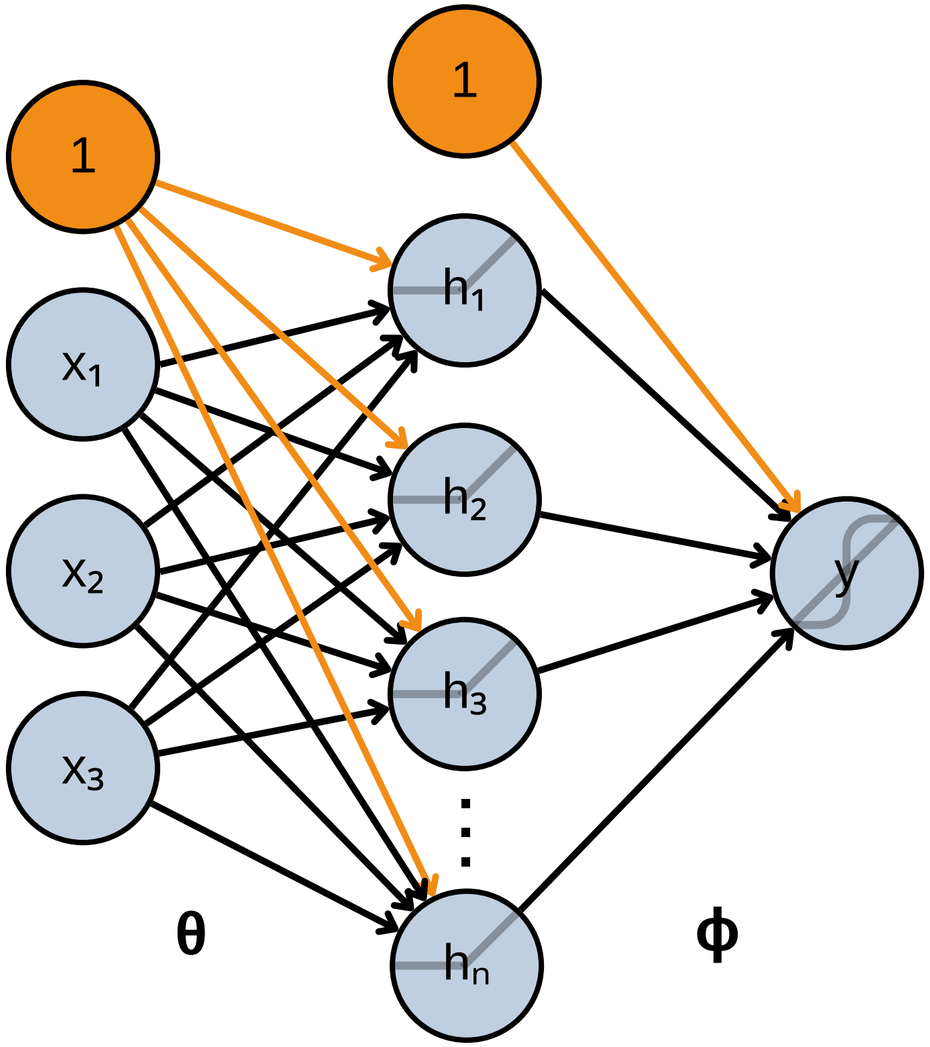
Combining Perceptrons

We consider again the simple regression problem with a single input x and output y

- we can apply more than one single perceptron to the input (and dummy)
- each of these perceptrons will produce a different output h_i
- we can then use these outputs as input for another perceptron that produce the output y

In this way we are adding an **hidden layer** to the neural network





Shallow Neural Networks

More generally we can have neural networks with

- as many inputs as we want
- as many hidden neurons as we want

The parameters of this neural network will be contained in two weight matrices

- **θ** connecting the input to the hidden layer
- **ϕ** connecting the hidden layer to the output

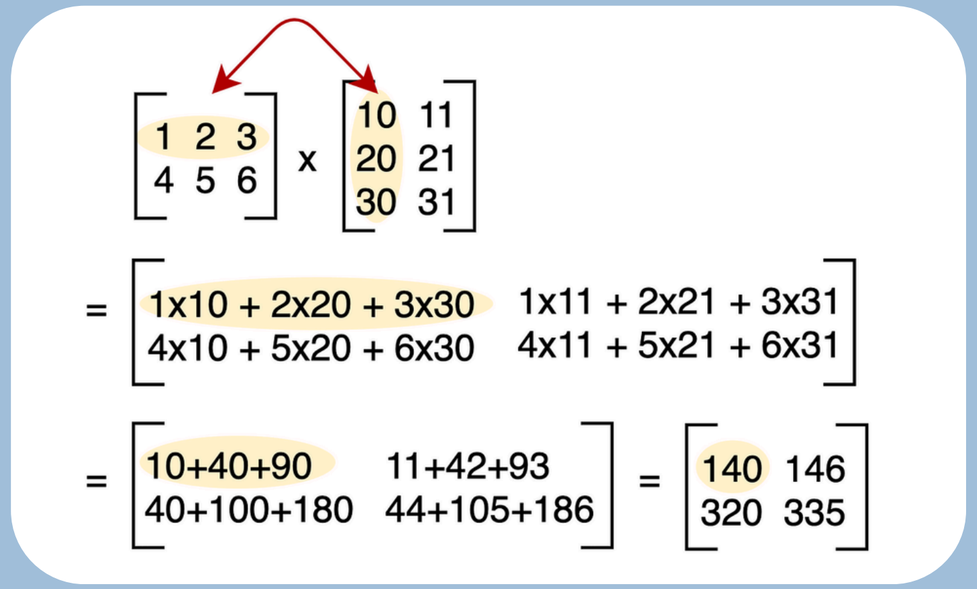
This type of neural network with a single hidden layer is called **Shallow Neural Network**

Math Recap

Matrix Multiplication

Matrix Multiplication consists in row by column multiplications:

- the elements (i, j) of the product matrix is obtained starting from the row i of the first matrix and the column j of the second matrix
- in general matrix multiplication is non commutative **A** x **B** ≠ **B** x **A**
- the number of columns of the first matrix must be equal to the number of rows of the second matrix



Mathematical Representation

We denote by **x** input vector with the dummy

x=(1, x)

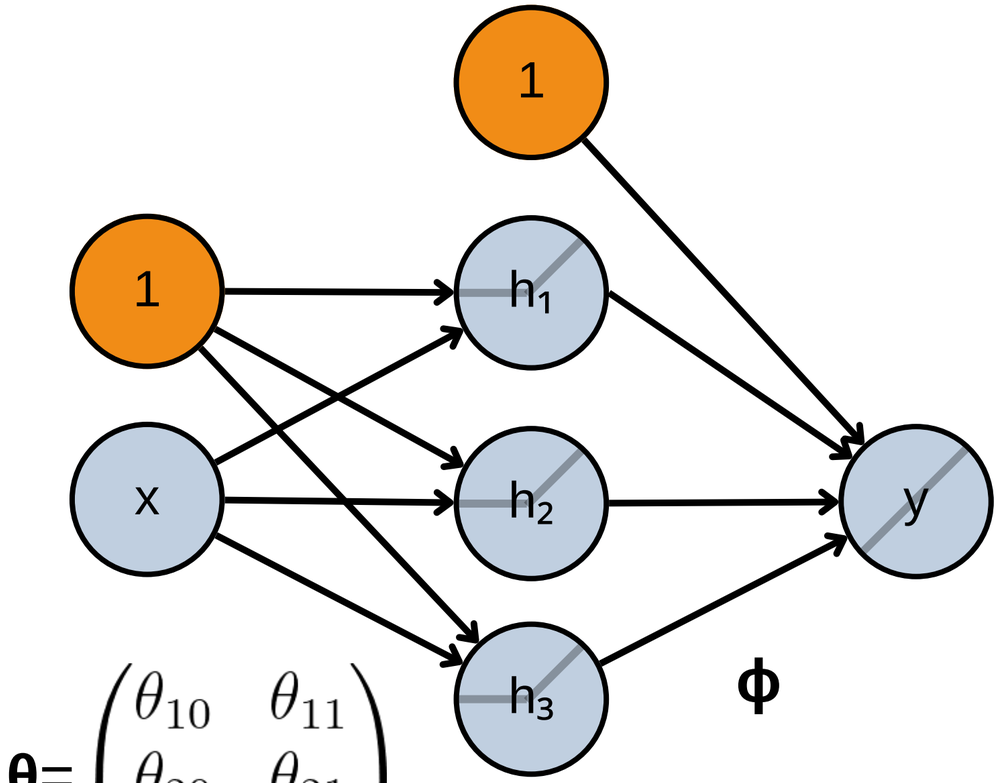
The hidden neurons **h'**=(h_1, h_2, h_3) satisfy

h'=a[**θx**]

Where a is the ReLU activation.

Hidden units

$$\begin{cases} h_1 = \mathrm{a}[\theta_{10} + \theta_{11}x] \\ h_2 = \mathrm{a}[\theta_{20} + \theta_{21}x] \\ h_3 = \mathrm{a}[\theta_{30} + \theta_{31}x] \end{cases}$$



$$\boldsymbol{\theta} = \begin{pmatrix} \theta_{10} & \theta_{11} \\ \theta_{20} & \theta_{21} \\ \theta_{30} & \theta_{31} \end{pmatrix}$$

Mathematical Representation

We denote by **x** input vector with the dummy

x=(1, x)

The hidden neurons **h'**=(h_1, h_2, h_3) satisfy

h'=a[**θx**]

Where a is the ReLU activation.

We can then define the hidden unit vector as

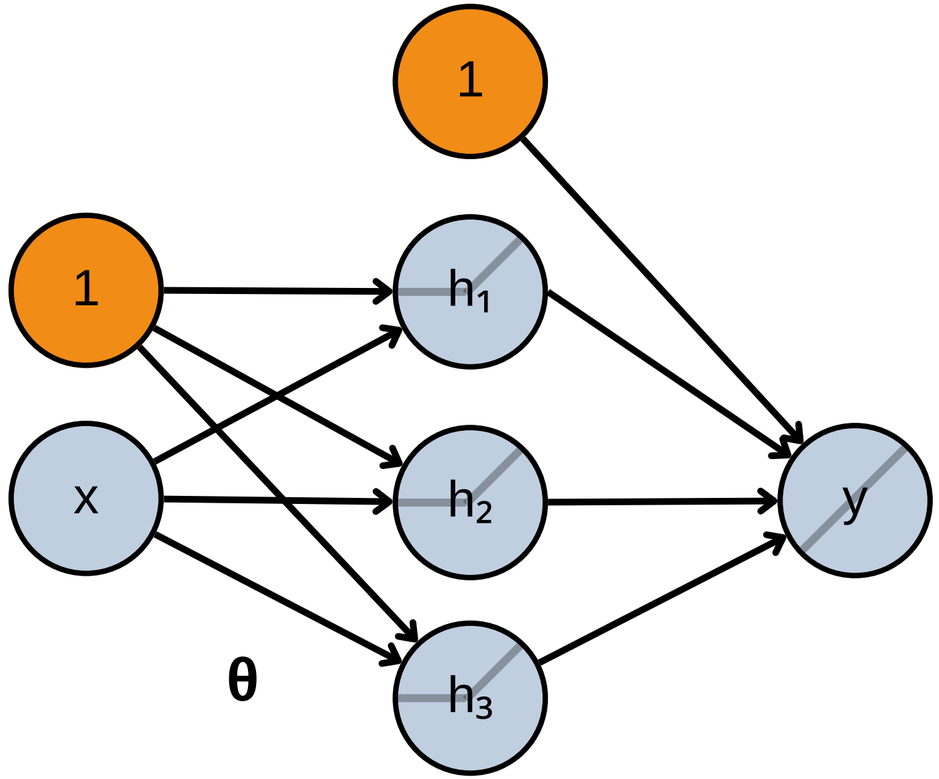
h=(1, h_1, h_2, h_3)=(1, **h'**)

and get the output y as

y=a[**ϕh**]

Where this time a is a linear activation

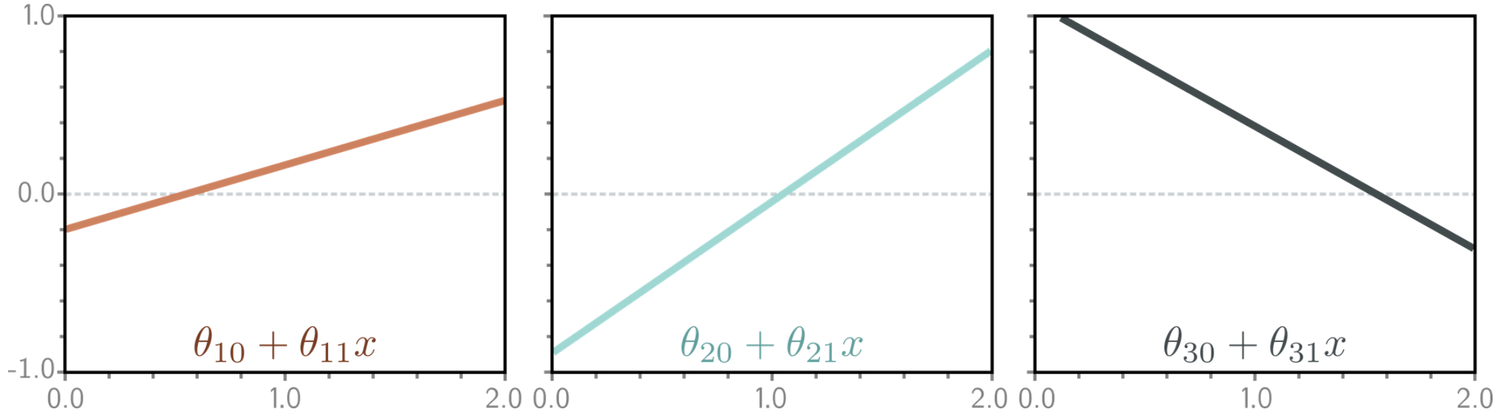
$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$



ϕ= $(\phi_0 \quad \phi_1 \quad \phi_2 \quad \phi_3)$

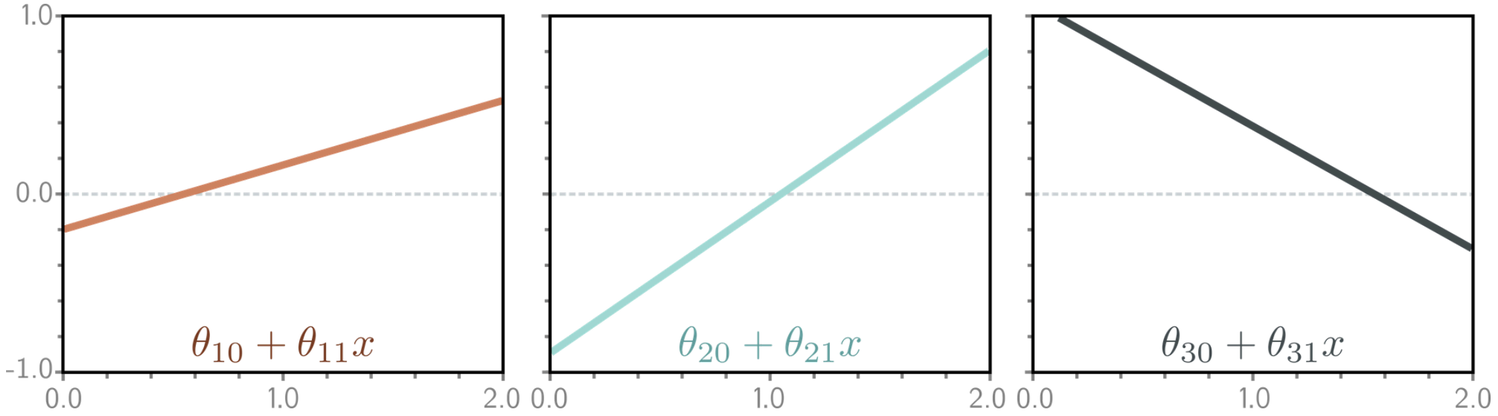
Computing the Hidden Layer

First we need to compute the product between the input and the first weights vector **θ**

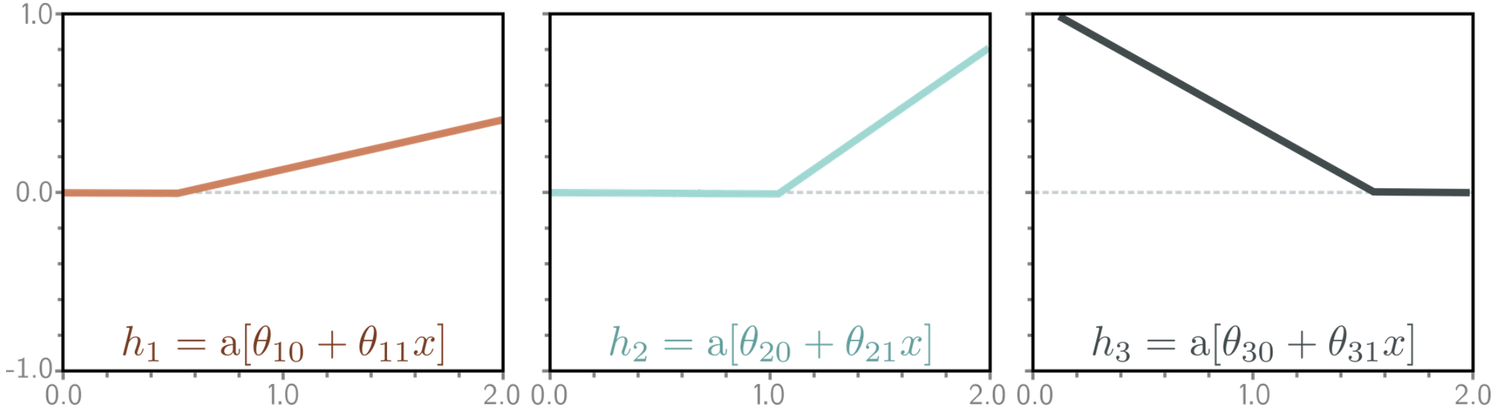


Computing the Hidden Layer

First we need to compute the product between the input and the first weights vector **θ**

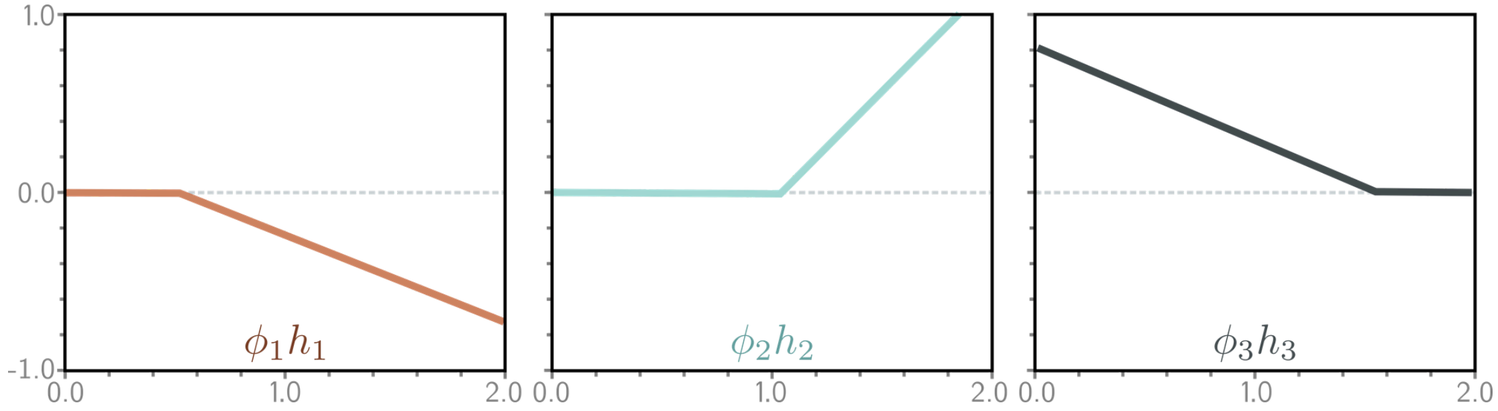


Then we apply the activation function (ReLU) to get the hidden state vector **h**



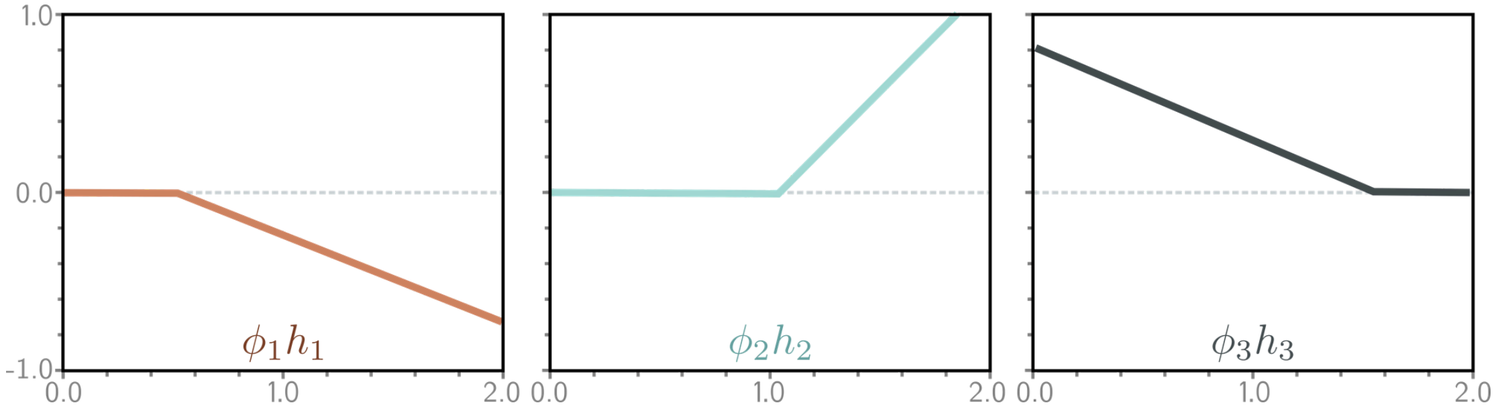
Computing the Output

We then multiply the hidden vector state for the second weight vector **ϕ**

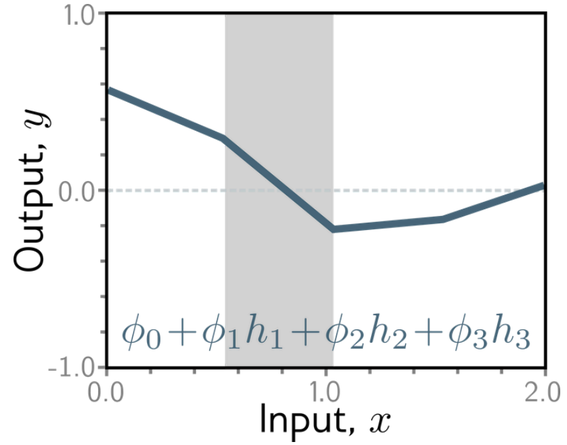


Computing the Output

We then multiply the hidden vector state for the second weight vector **ϕ**

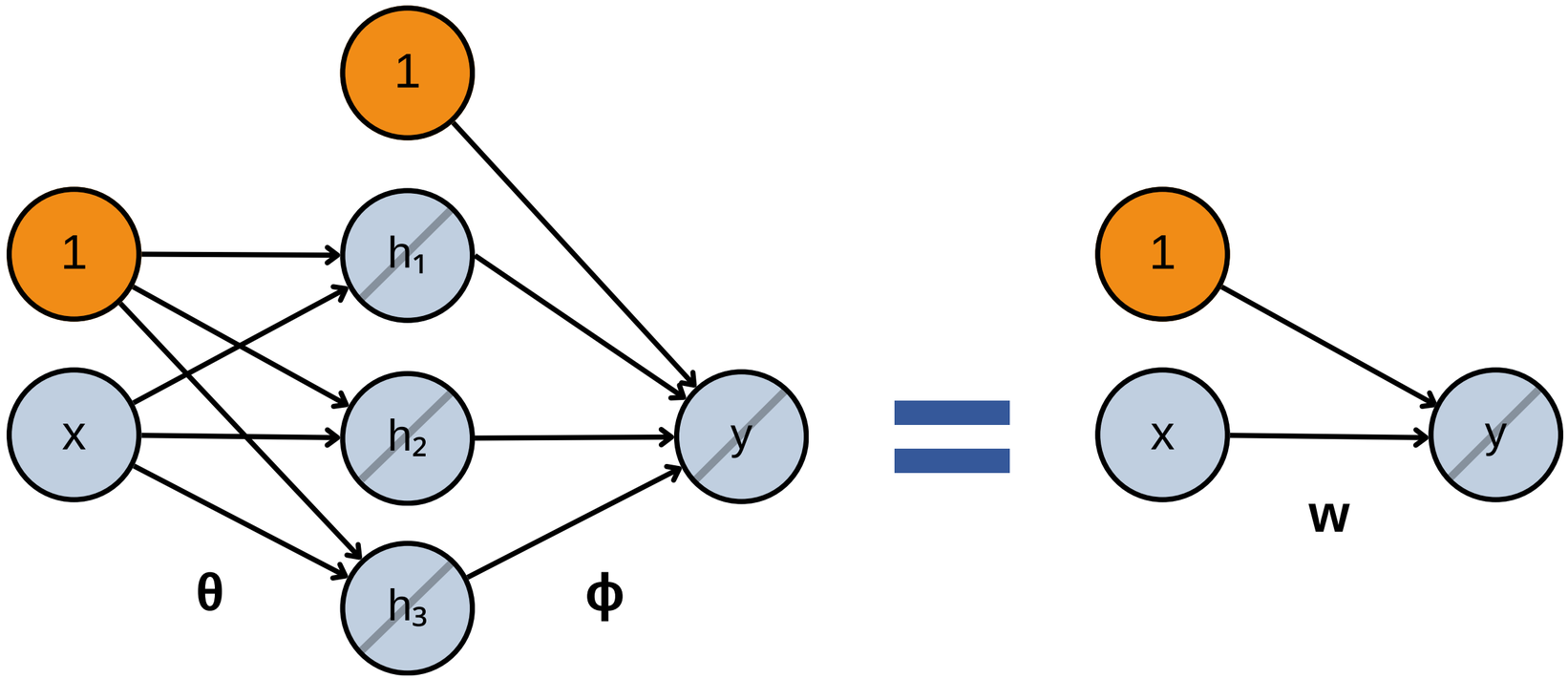


Finally we sum the weighted hidden units to get the output y



Why is the ReLU Important?

Non-linear activation functions like the ReLU are crucial in Deep Learning. A shallow neural network with linear activation functions is equivalent to a simple perceptron

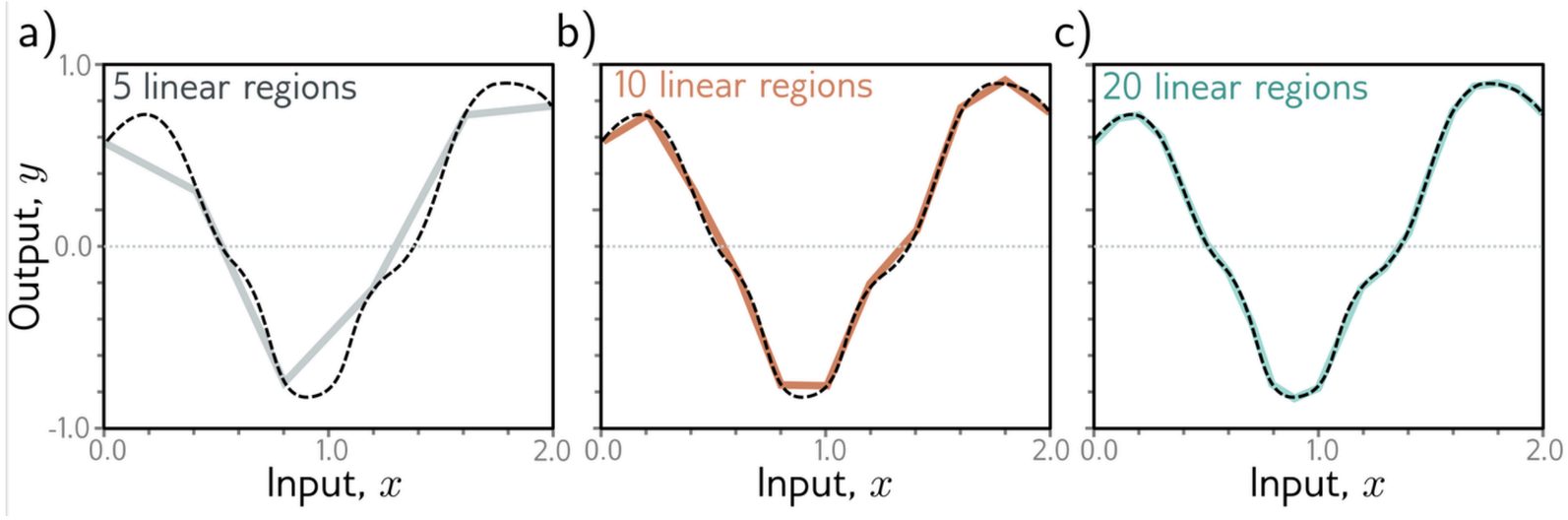


Universal Approximation Theorem

The **Universal Approximation theorem** states that

A shallow neural network with a single hidden layer containing a finite number of neurons can approximate any continuous function

This explains why even relatively simple networks can model complex phenomena, though it does not provide a method to find the optimal network parameters.



https://giordano-demarzo.github.io/teaching/deep-learning-25/shallow_nn/

Universität
Konstanz

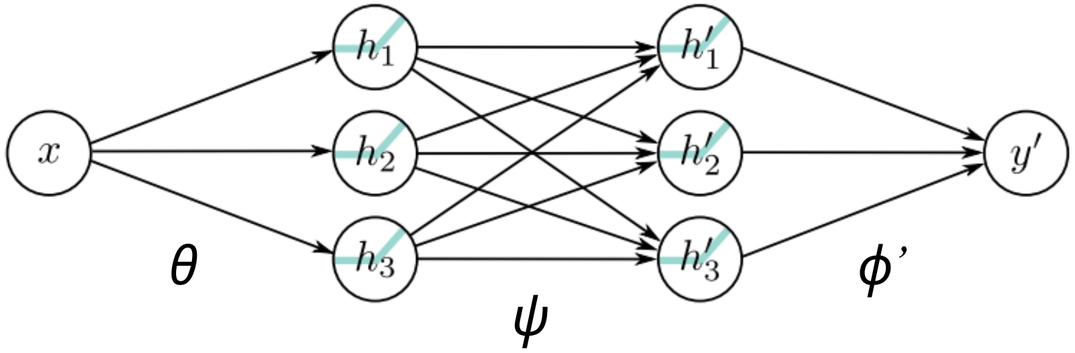
Multilayer Perceptron Architecture

Deep Neural Networks: Multilayer Perceptron

A **deep neural network** is a **feed-forward** neural network with at least **two hidden layers.**

The multilayer perceptron (MLP) is the most simple example of deep neural network:

- it is composed of an input layer (x), an output layer (y) and at least 2 hidden layers (h, h')
- each neuron
  - takes the outputs of the neurons of the previous layer
  - weights (θ, ψ, ϕ) and then sums these outputs, also including the bias
  - computes and then outputs the result of the (non-linear) activation function (a)



Mathematical Representation of MLP

The first hidden layer processes the input

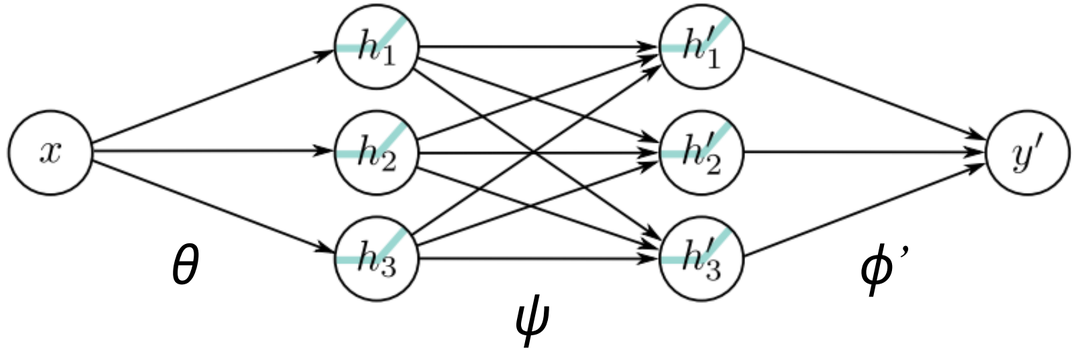
$$
\begin{aligned}
h_1 &= \mathrm{a}[\theta_{10} + \theta_{11}x] \\
h_2 &= \mathrm{a}[\theta_{20} + \theta_{21}x] \\
h_3 &= \mathrm{a}[\theta_{30} + \theta_{31}x],
\end{aligned}
$$

The second hidden layer processes the output of the first hidden layer

$$
\begin{aligned}
h'_1 &= \mathrm{a}[\psi_{10} + \psi_{11}h_1 + \psi_{12}h_2 + \psi_{13}h_3] \\
h'_2 &= \mathrm{a}[\psi_{20} + \psi_{21}h_1 + \psi_{22}h_2 + \psi_{23}h_3] \\
h'_3 &= \mathrm{a}[\psi_{30} + \psi_{31}h_1 + \psi_{32}h_2 + \psi_{33}h_3],
\end{aligned}
$$

The output layer computes the output processing the result of the second hidden layer

$$y' = \phi'_0 + \phi'_1 h'_1 + \phi'_2 h'_2 + \phi'_3 h'_3.$$



Mathematical Representation of MLP

The first hidden layer processes the input

$$
\begin{aligned}
h_1 &= \mathrm{a}[\theta_{10} + \theta_{11}x] \\
h_2 &= \mathrm{a}[\theta_{20} + \theta_{21}x] \\
h_3 &= \mathrm{a}[\theta_{30} + \theta_{31}x].
\end{aligned}
$$

$$
\begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix} = \mathbf{a}\left[\begin{bmatrix} \theta_{10} \\ \theta_{20} \\ \theta_{30} \end{bmatrix} + \begin{bmatrix} \theta_{11} \\ \theta_{21} \\ \theta_{31} \end{bmatrix} x\right]
$$

The second hidden layer processes the output of the first hidden layer

$$
\begin{aligned}
h'_1 &= \mathrm{a}[\psi_{10} + \psi_{11}h_1 + \psi_{12}h_2 + \psi_{13}h_3] \\
h'_2 &= \mathrm{a}[\psi_{20} + \psi_{21}h_1 + \psi_{22}h_2 + \psi_{23}h_3] \\
h'_3 &= \mathrm{a}[\psi_{30} + \psi_{31}h_1 + \psi_{32}h_2 + \psi_{33}h_3],
\end{aligned}
$$

$$
\begin{bmatrix} h'_1 \\ h'_2 \\ h'_3 \end{bmatrix} = \mathbf{a}\left[\begin{bmatrix} \psi_{10} \\ \psi_{20} \\ \psi_{30} \end{bmatrix} + \begin{bmatrix} \psi_{11} & \psi_{12} & \psi_{13} \\ \psi_{21} & \psi_{22} & \psi_{23} \\ \psi_{31} & \psi_{32} & \psi_{33} \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix}\right]
$$

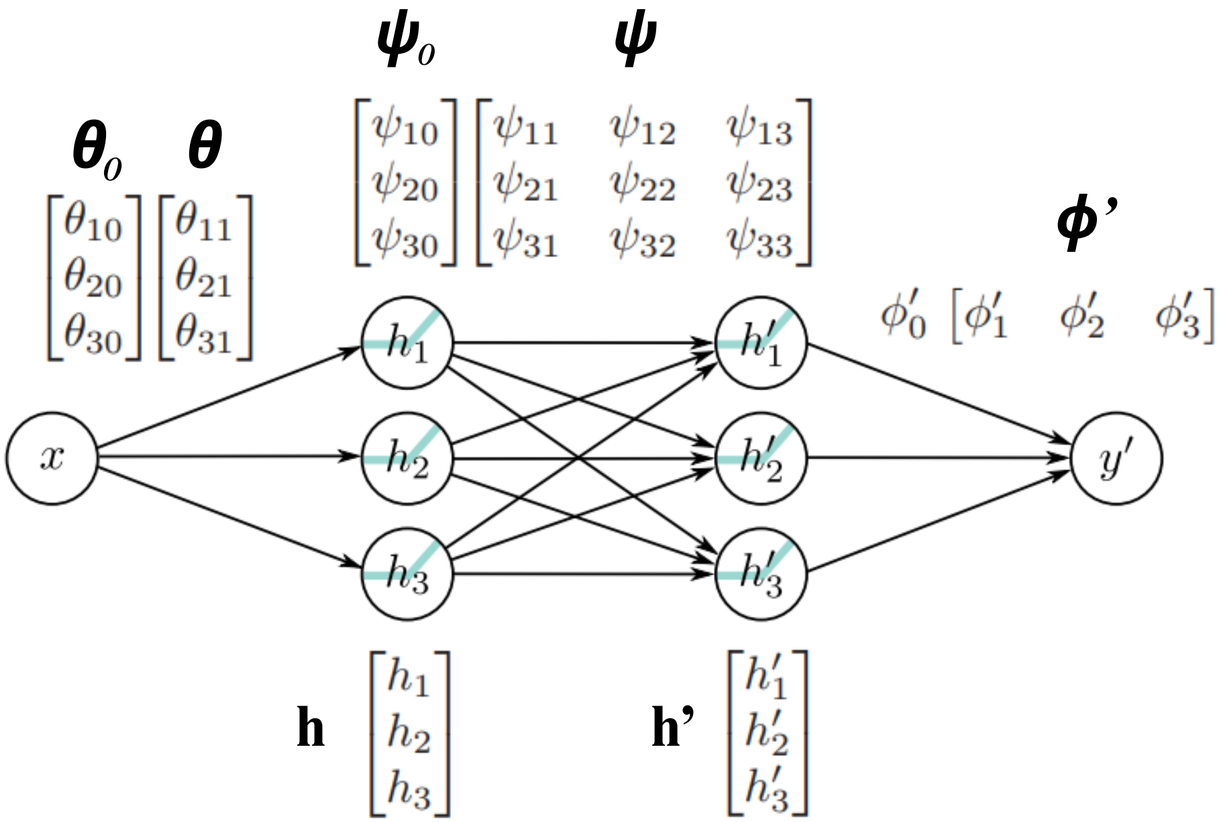
The output layer computes the output processing the result of the second hidden layer

$$
y' = \phi'_0 + \phi'_1 h'_1 + \phi'_2 h'_2 + \phi'_3 h'_3.
$$

$$
y' = \phi'_0 + \begin{bmatrix} \phi'_1 & \phi'_2 & \phi'_3 \end{bmatrix} \begin{bmatrix} h'_1 \\ h'_2 \\ h'_3 \end{bmatrix},
$$

Matrix Notation

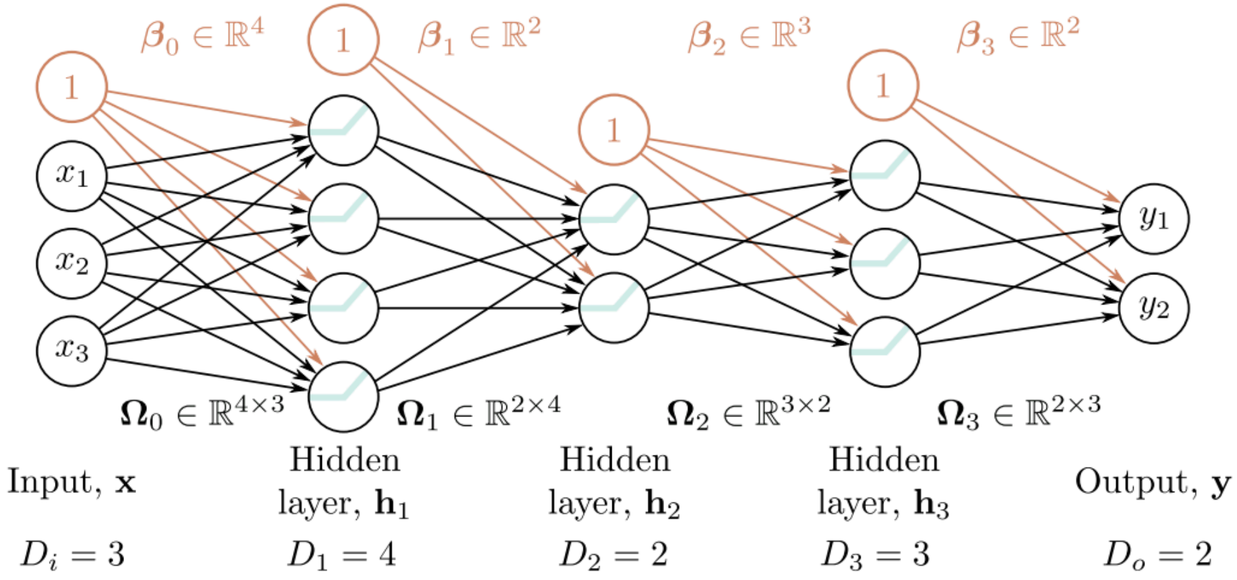
We can write this same neural network in matrix notation. Bold letters denote vectors and mtatrices.



$$\begin{aligned}\mathbf{h} &= \mathbf{a}\left[\boldsymbol{\theta}_0 + \boldsymbol{\theta}x\right]\\ \mathbf{h}' &= \mathbf{a}\left[\boldsymbol{\psi}_0 + \boldsymbol{\Psi}\mathbf{h}\right]\\ y' &= \phi'_0 + \boldsymbol{\phi}'\mathbf{h}',\end{aligned}$$

General Formulation

In general we will have K hidden layers $\mathbf{h_1}...\mathbf{h_k}$ and K+1 weight matrices $\mathbf{\Omega_0}...\mathbf{\Omega_k}$ and bias vectors $\mathbf{\beta_0}...\mathbf{\beta_k}$
Matrix notation is useful to represent large neural networks in a compact notation.

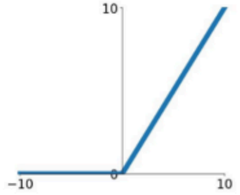


$$
\begin{aligned}
\mathbf{h}_1 &= \mathbf{a}[\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0\mathbf{x}] \\
\mathbf{h}_2 &= \mathbf{a}[\boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1\mathbf{h}_1] \\
\mathbf{h}_3 &= \mathbf{a}[\boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2\mathbf{h}_2] \\
&\vdots \\
\mathbf{h}_K &= \mathbf{a}[\boldsymbol{\beta}_{K-1} + \boldsymbol{\Omega}_{K-1}\mathbf{h}_{K-1}] \\
\mathbf{y} &= \boldsymbol{\beta}_K + \boldsymbol{\Omega}_K\mathbf{h}_K.
\end{aligned}
$$

Activation Functions for Hidden Layers

ReLU

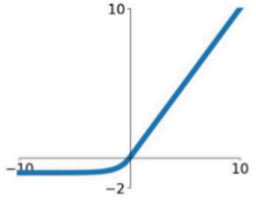
$\max(0, x)$



- Standard option
- Fast computation
- No gradient vanishing

ELU

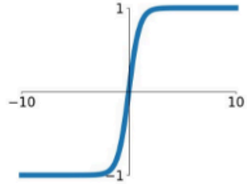
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



- Possible replacement for ReLU
- No gradient sparsity

tanh

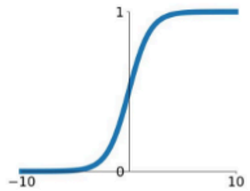
$\tanh(x)$



- Not suitable for dense and convolutional layers:
- Gradient vanishing
- Used in RNN and GAN

Sigmoid

$\sigma(x) = \frac{1}{1+e^{-x}}$

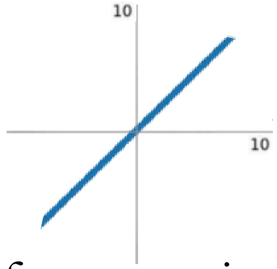


- Similar uses and problems of tanh

Activation Functions for Output Layer

Linear

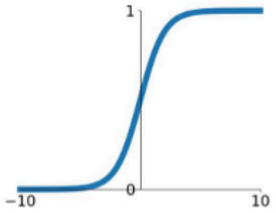
x



- Standard option for regression tasks

Sigmoid

$\sigma(x) = \frac{1}{1+e^{-x}}$



- used in binary classification problems (2 classes)
- single output neuron

Softmax

$$\text{SoftMax}(x_i) = \frac{e^{x_i}}{\sum_j^N e^{x_j}}$$

- Used in multi-class classification problems
- N output neurons
- Output of each neuron is in (0,1) and is interpretale as a probability

Let's Make it Easy

This may seems very complex, but the concept is very easy:

- a **DNN** is just a very **complex function with many parameters** (weights and biases)
- this function takes an input, typically an high-dimension vector, and returns an output

Then why using all those neurons and matrices? Can't we just use a standard function with a million parameter and fit this function to the data like we would do with a linear fit?

- possible in theory
- impossible in practice
- DNN are just a very **computationally efficient** way to represent and fit arbitrary complex functions

Our goal is thus to adjust the weights and biases of the neural network to make it represent the function we want (that better fitting real data).

Summary

Limits of the Perceptron

The perceptron can only solve linear problems: in the case of classification it draws a linear decision boundary, while in the case of regression it can only perform a linear regression

Shallow Neural Networks

The output of a perceptron can be fed into another perceptron, leading to a shallow neural network. Shallow Neural Networks can approximate any arbitrarily complex functions with enough hidden units.

Multilayer Perceptron Architecture

Deep Neural Networks are neural networks with at least two hidden layers. There is nothing conceptually different with respect to shallow neural networks.

Next Lectures

Tomorrow NO Coding Lab (23/04)

Next coding lab will be on April 30

Next week Lecture (29/04)

We will cover how to train deep neural networks and techniques to improve the training of neural networks

Following Lectures

Then for the next 2 weeks we will focus on two conceptually similar architectures: Convolutional Neural Networks and Graph Neural Networks

First Assignment

On April 30 I will upload on my website the first assignment (Multlayer Perceptron)