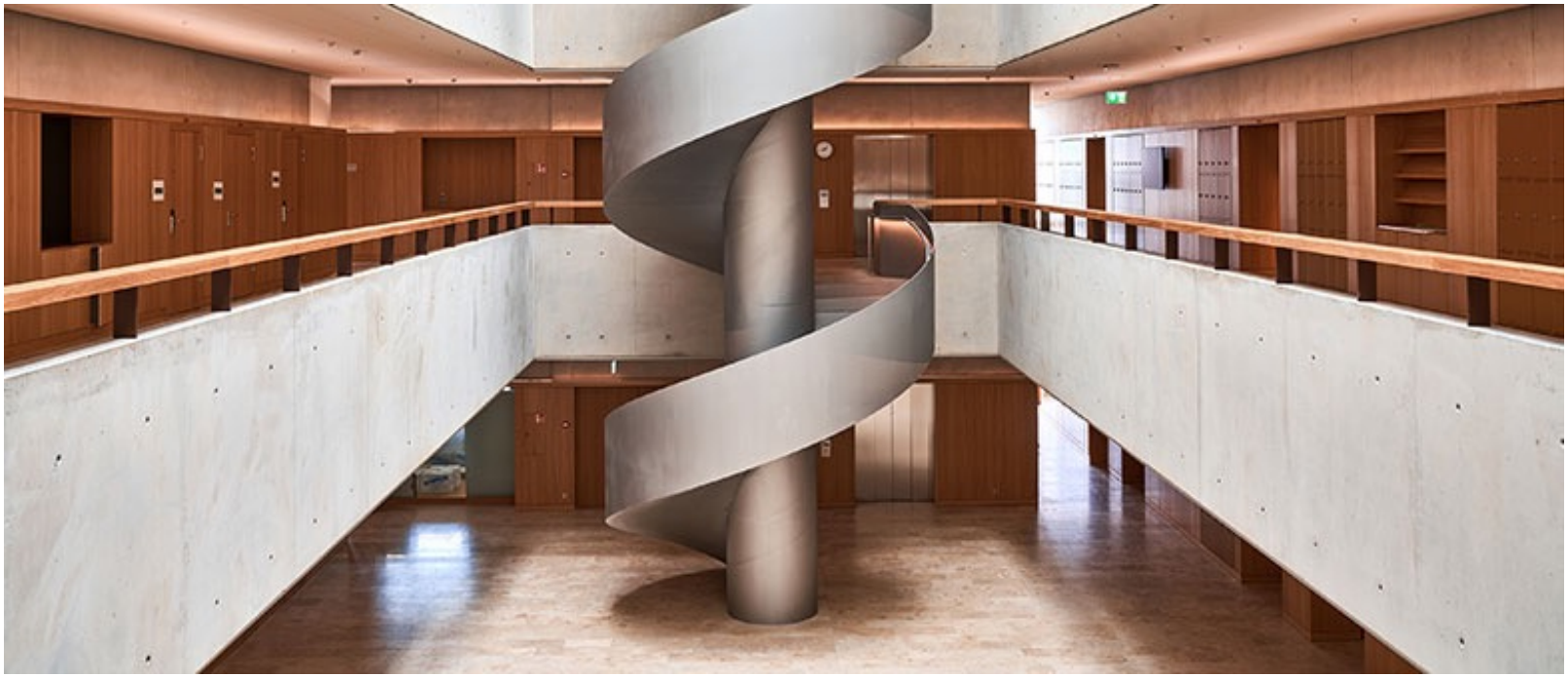
HSLU Hochschule Luzern



CAS Machine Learning

Deep Learning: From Core Foundations to Applications

Javier Montoya, Dr. sc. ETH
javier.montoya@hslu.ch

Core Foundation

Artificial Intelligence

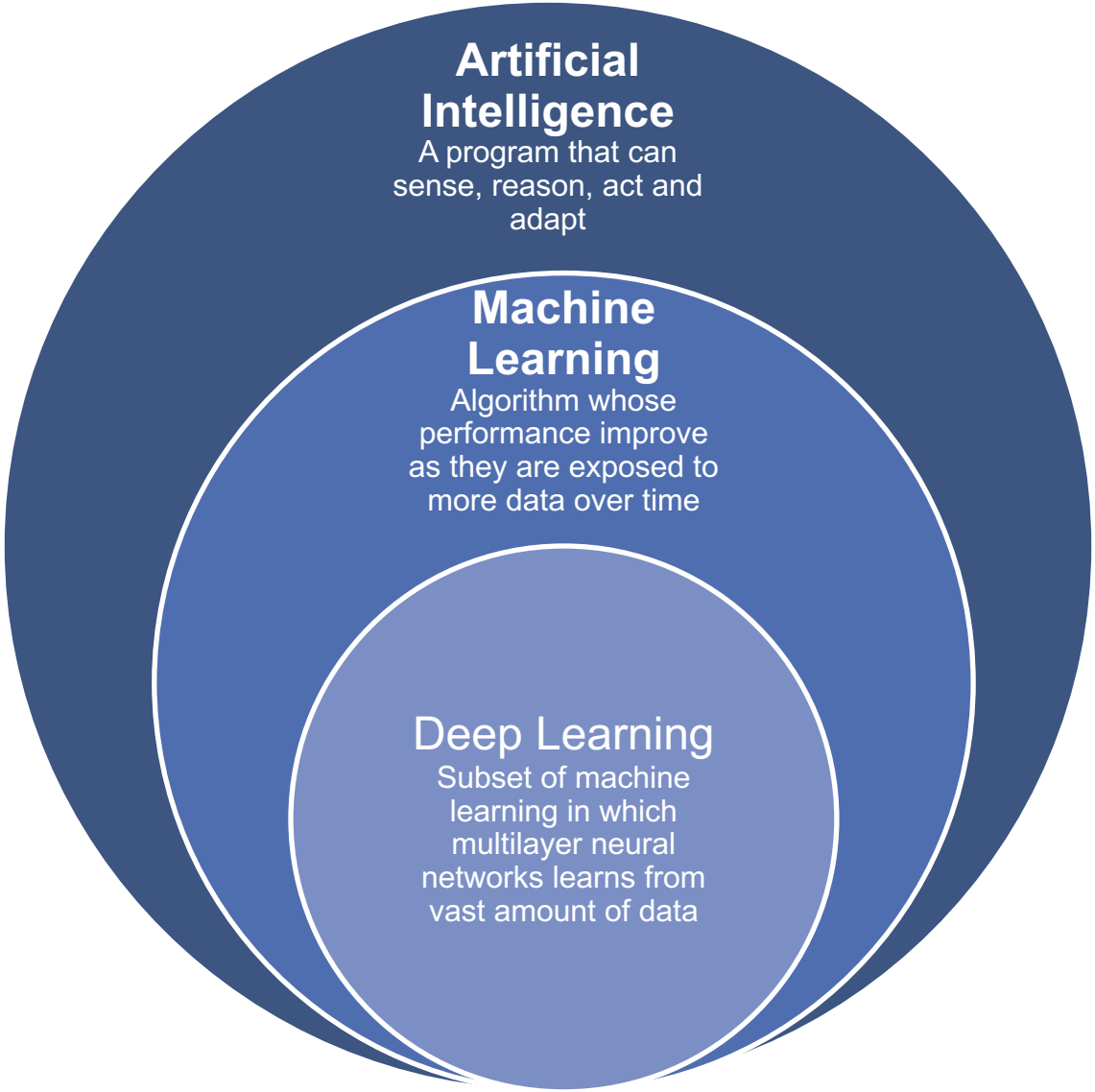
- Overview
- Perceptron
- Neurons and Layers
- Activation Functions

Neural Networks

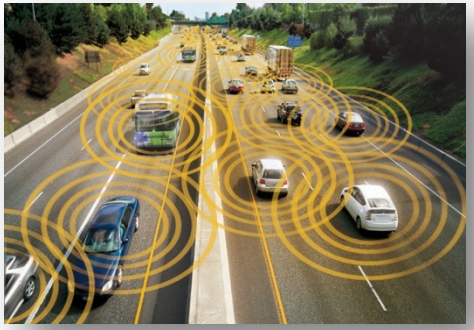
- Architectures
- Optimization: Forward/Backprop.
- Loss Functions

Training in Practice

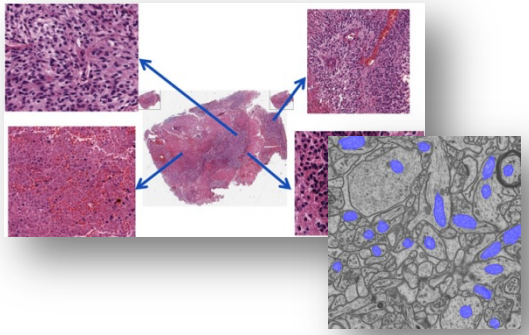
- Regularization
- Train/Validation/Test


Artificial Intelligence
A program that can sense, reason, act and adapt
Machine Learning
Algorithm whose performance improve as they are exposed to more data over time
Deep Learning
Subset of machine learning in which multilayer neural networks learns from vast amount of data

Applications of Deep Learning



Transportation

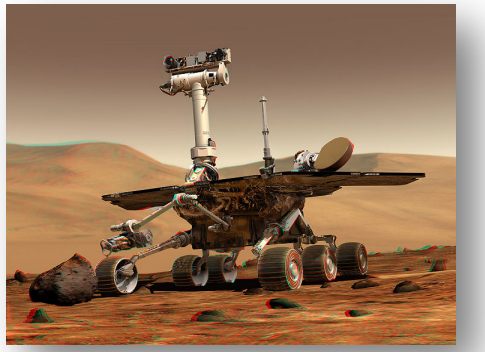


Medicine

Security

Smart-home

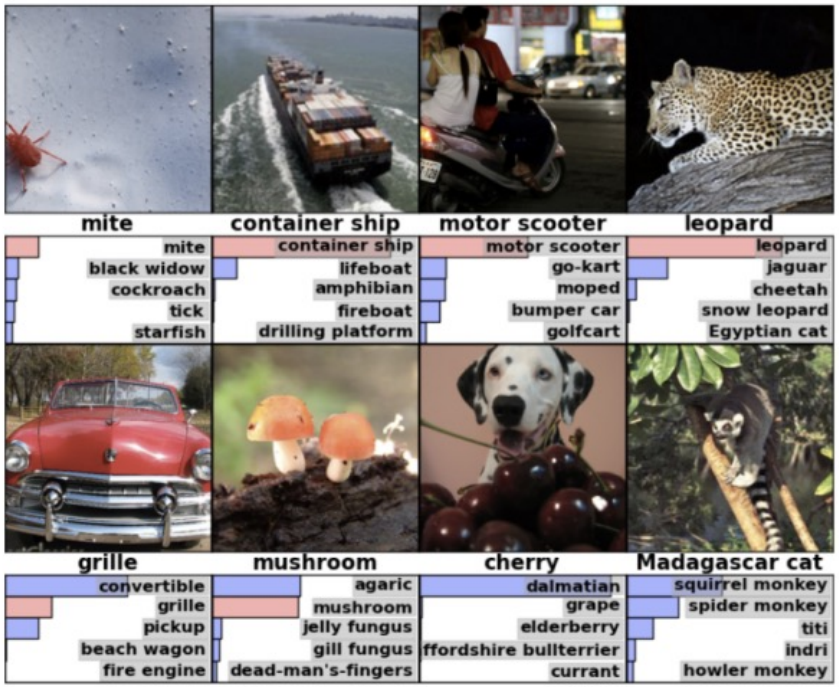
Entertainment



High Tech

Successes of AI: ImageNet Challenge

ILSVRC



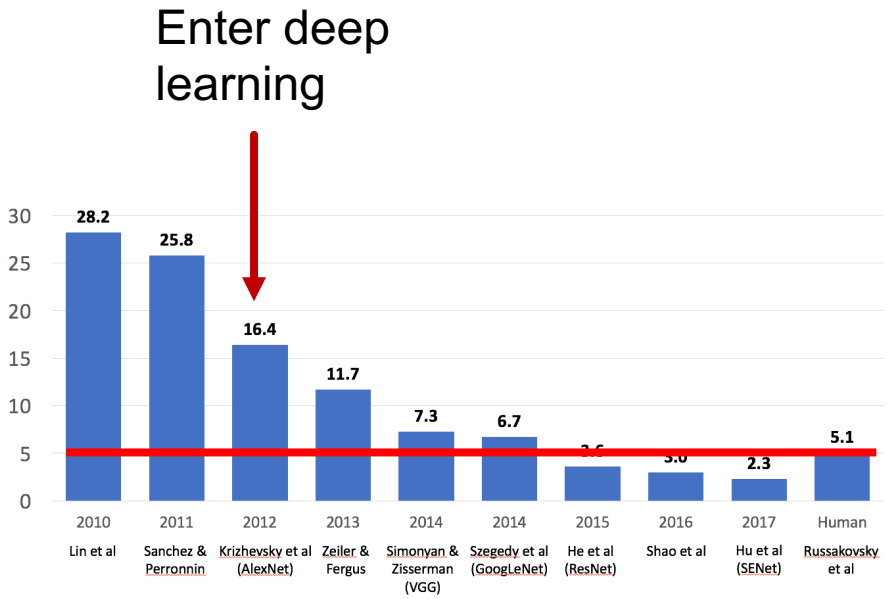
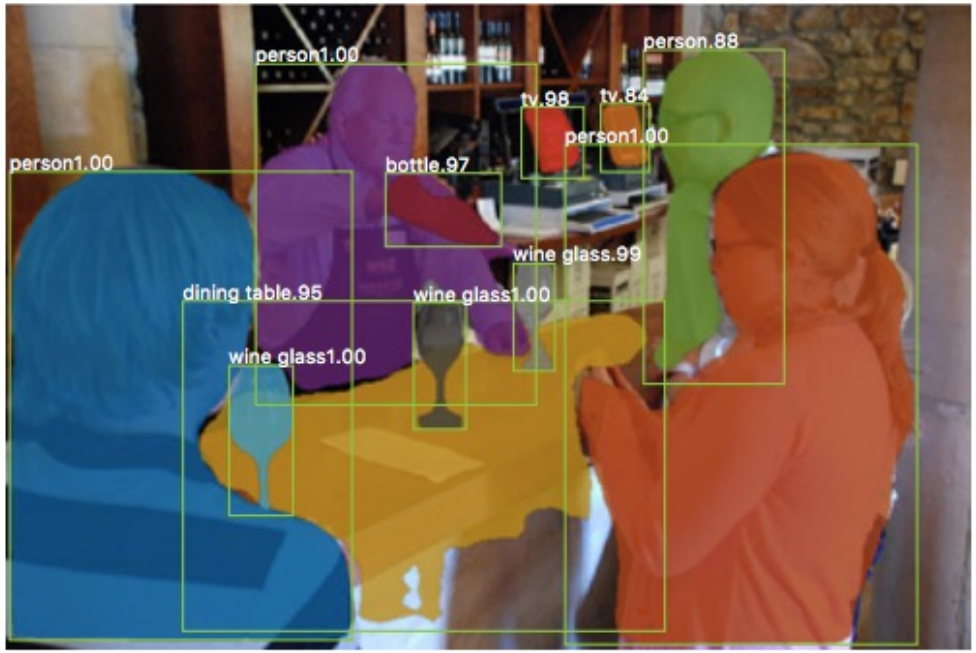


Figure source

Successes of AI: Detection, Segmentation



K. He, G. Gkioxari, P. Dollar, and R. Girshick, Mask R-CNN,
ICCV 2017 (Best Paper Award)

Successes of AI: Detection, Segmentation

Bounding box prediction, dense prediction

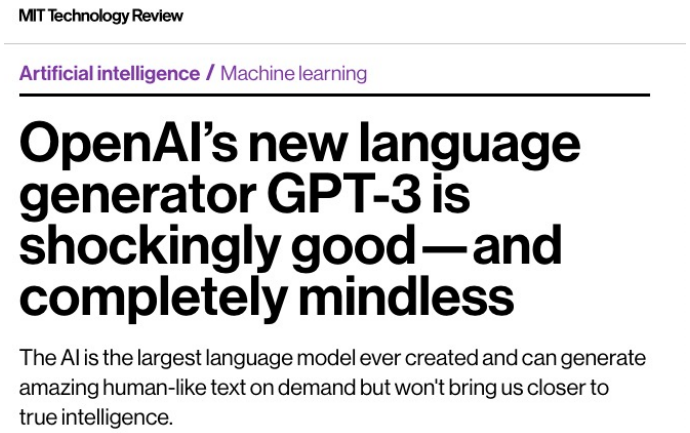
Keypoint prediction

K. He, G. Gkioxari, P. Dollar, and R. Girshick, Mask R-CNN, ICCV 2017 (Best Paper Award)

Successes of AI: Image Generation

Successes of AI: Natural Language

- Neural machine translation
  - The Great AI Awakening – New York Times Magazine, 12/14/2016
- Language models: e.g., GPT-3



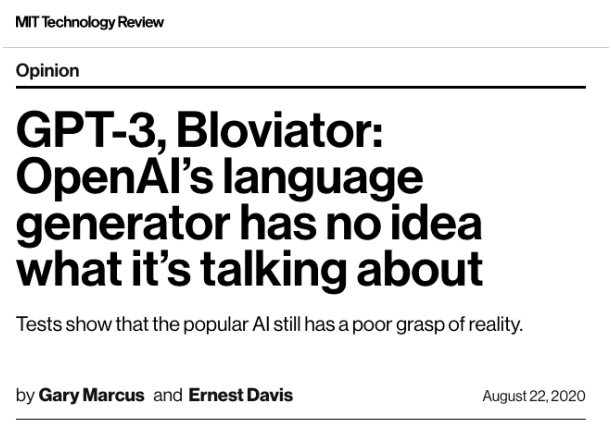
MIT Technology Review

Artificial intelligence / Machine learning

OpenAI's new language generator GPT-3 is shockingly good—and completely mindless

The AI is the largest language model ever created and can generate amazing human-like text on demand but won't bring us closer to true intelligence.

https://www.technologyreview.com/2020/07/20/1005454/openai-machine-learning-language-generator-gpt-3-nlp/



MIT Technology Review

Opinion

GPT-3, Bloviator: OpenAI's language generator has no idea what it's talking about

Tests show that the popular AI still has a poor grasp of reality.

by **Gary Marcus** and **Ernest Davis** August 22, 2020

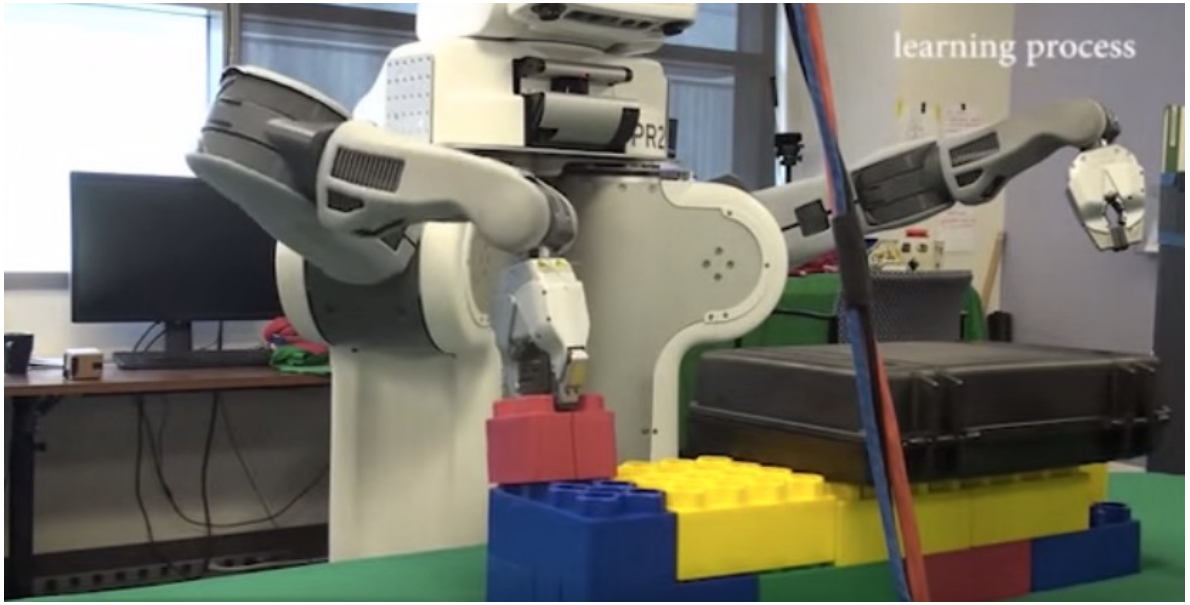
https://www.technologyreview.com/2020/08/22/1007539/gpt3-openai-language-generator-artificial-intelligence-ai-opinion/

Successes of AI: Natural Language

- Neural machine translation
  - The Great AI Awakening – New York Times Magazine, 12/14/2016
- Language models: e.g., GPT-3
- Vision and language models: DALL-E, CLIP

Successes of AI: Robotics

- Sensorimotor Learning



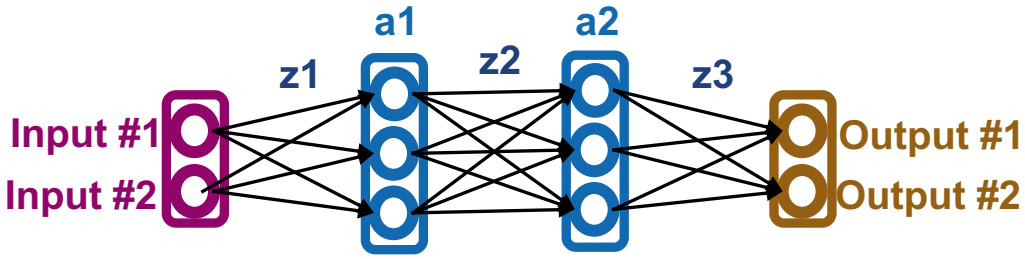
Overview video, training video

S. Levine, C. Finn, T. Darrell, P. Abbeel, End-to-end training of deep visuomotor policies, JMLR 2016

Why is A.I now?

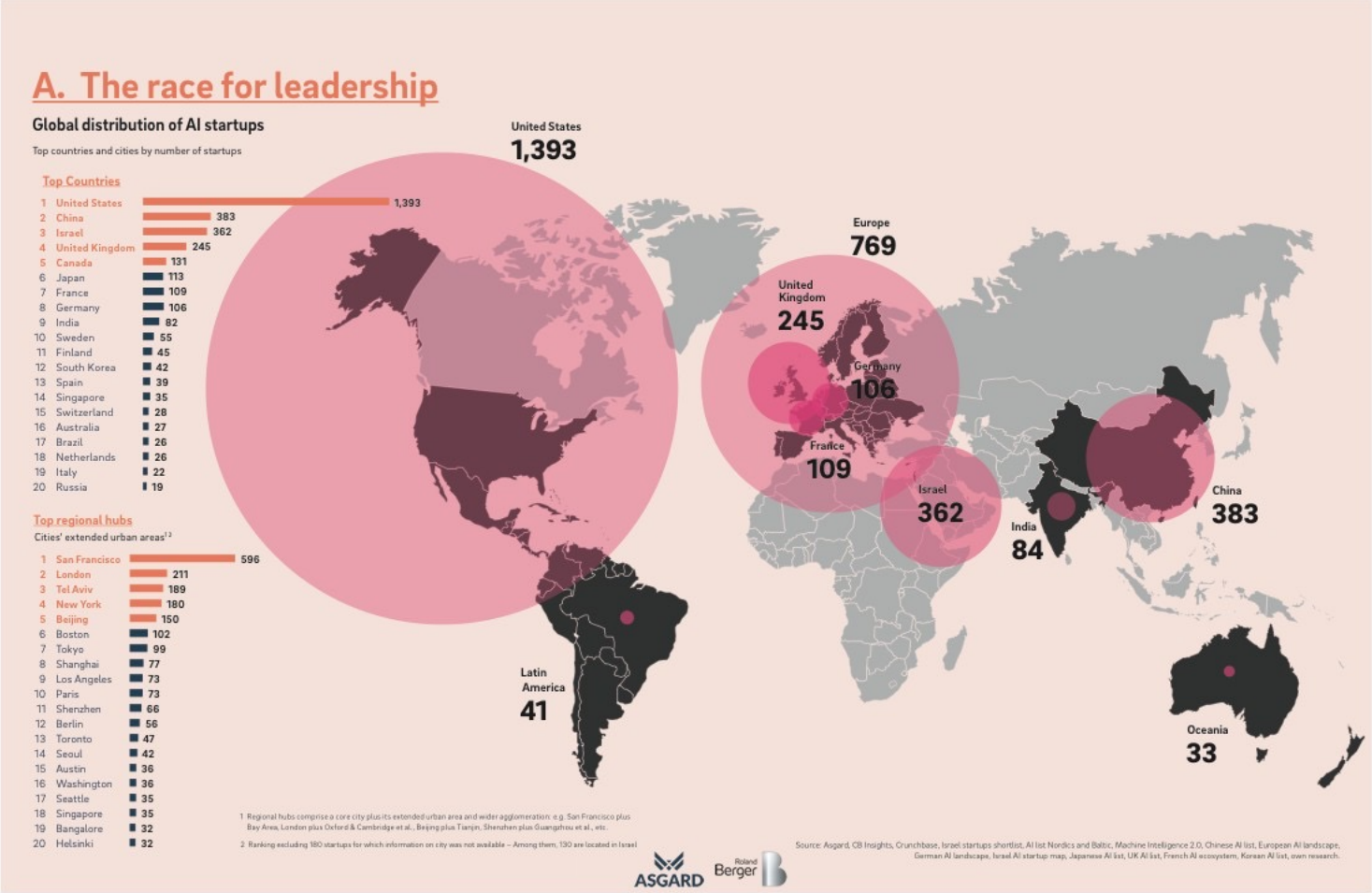
(i) Data

(ii) Software (Algorithms)



(iii) Hardware (GPU)

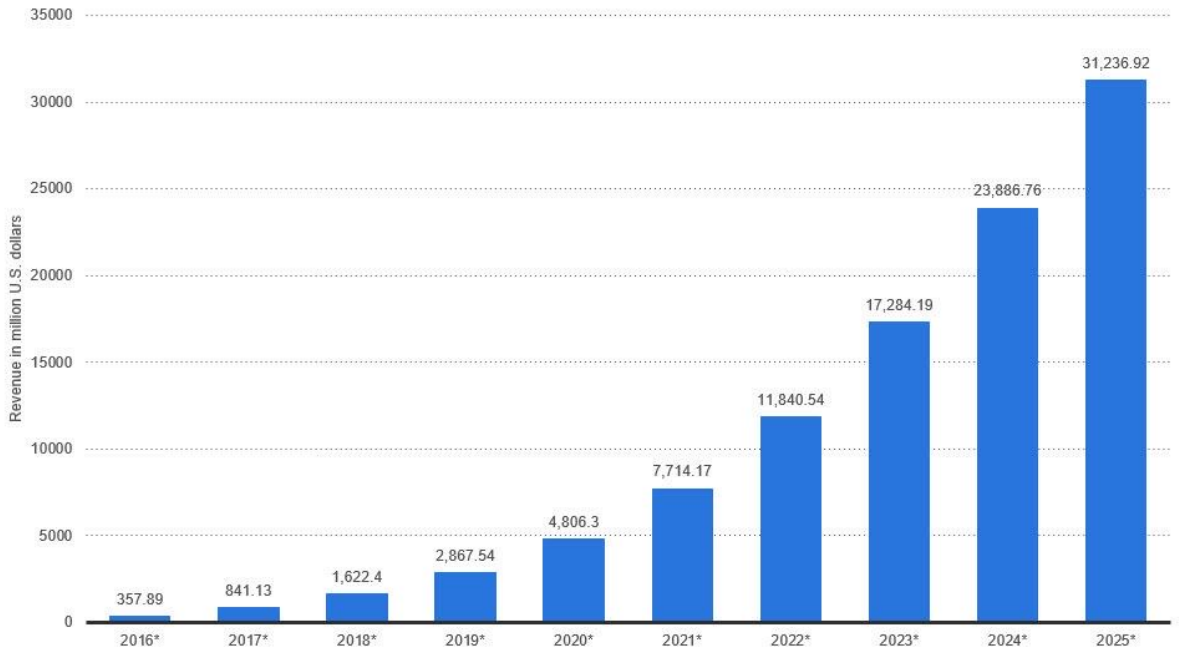
AI: Some recent statistics



AI: Some recent statistics

Enterprise artificial intelligence market revenue worldwide 2016-2025

Revenues from the artificial intelligence for enterprise applications market worldwide, from 2016 to 2025 (in million U.S. dollars)



statista

Machine Learning

- The goal of ML is to learn from data
  - It relies on statistics and computational tools (optimization)
- Example (supervised learning) tasks

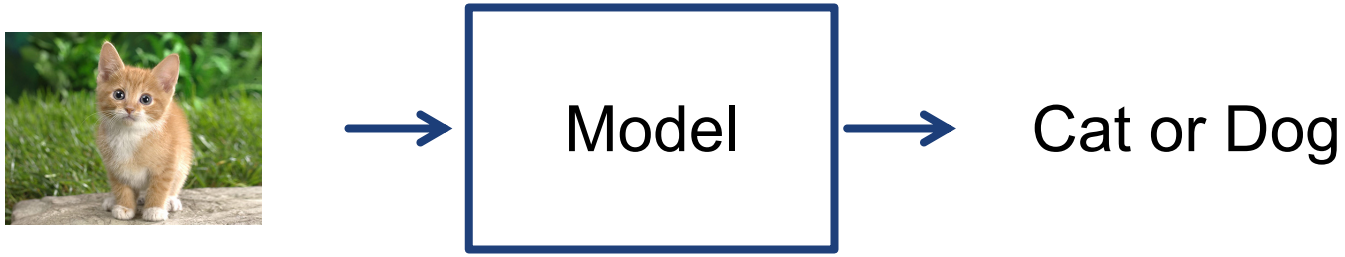
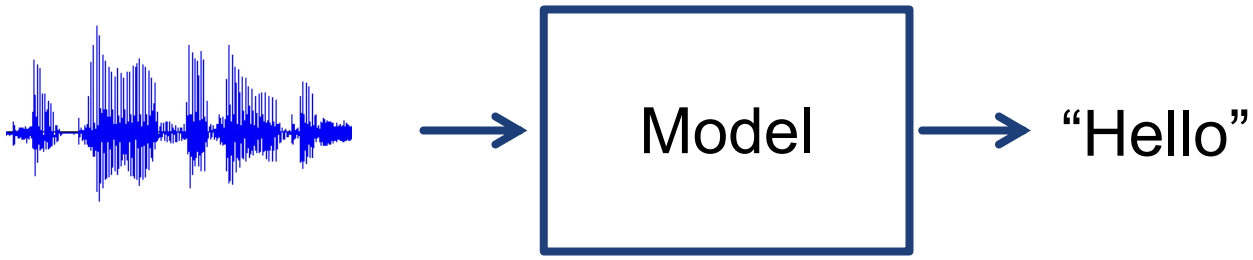


Image recognition / classification

Machine Learning

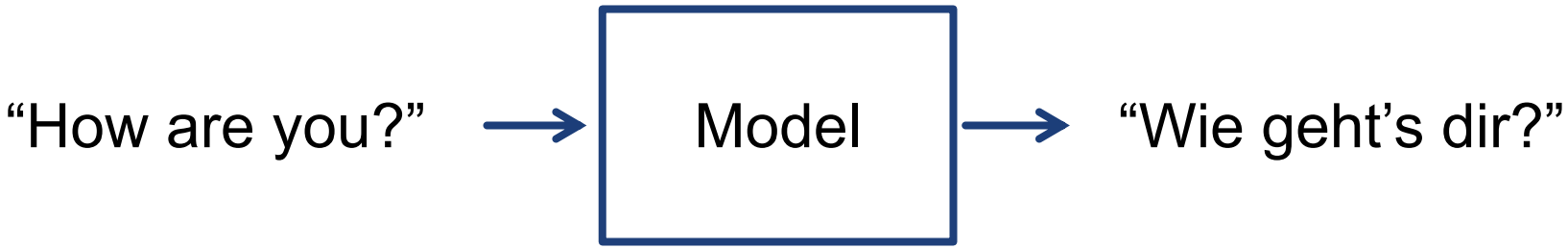
- The goal of ML is to learn from data
  - It relies on statistics and computational tools (optimization)
- Example (supervised learning) tasks



Speech recognition

Machine Learning

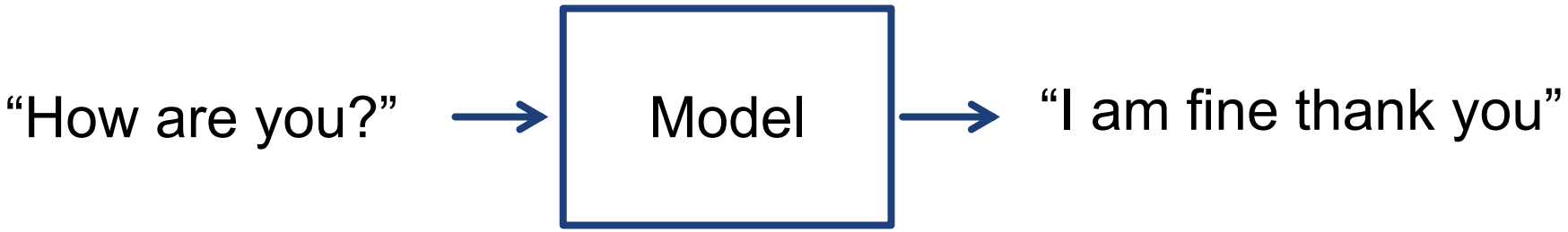
- The goal of ML is to learn from data
  - It relies on statistics and computational tools (optimization)
- Example (supervised learning) tasks



Machine Translation

Machine Learning

- The goal of ML is to learn from data
  - It relies on statistics and computational tools (optimization)
- Example (supervised learning) tasks



Conversational agent / chatbot

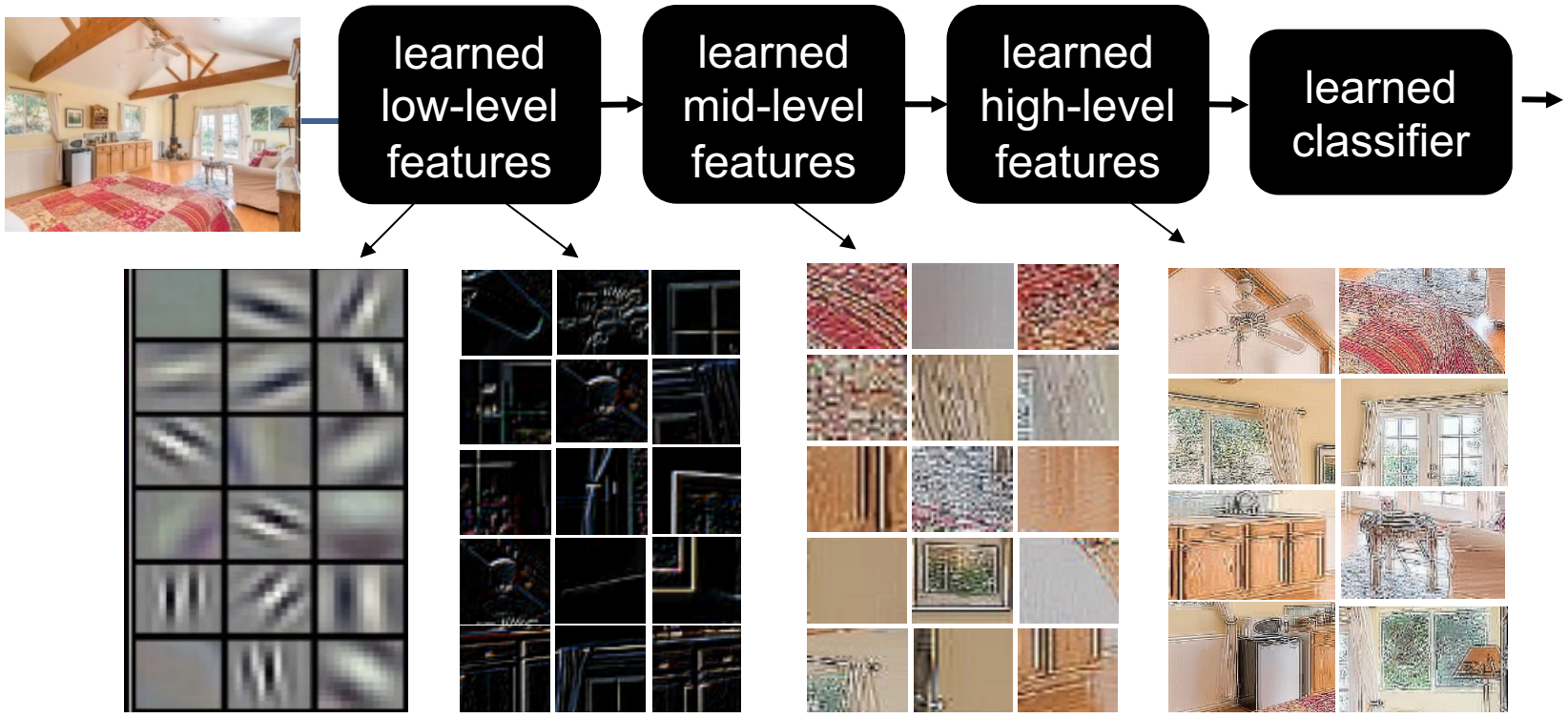
“Traditional” machine learning:

handcrafted features → learned classifier → **cat**

Deep, “end-to-end” learning:

learned low-level features → learned mid-level features → learned high-level features → learned classifier → **cat**

- DL applies a multi-layer process for learning rich hierarchical features (i.e., data representations)
  - Input image pixels → Edges → Textures → Parts → Objects



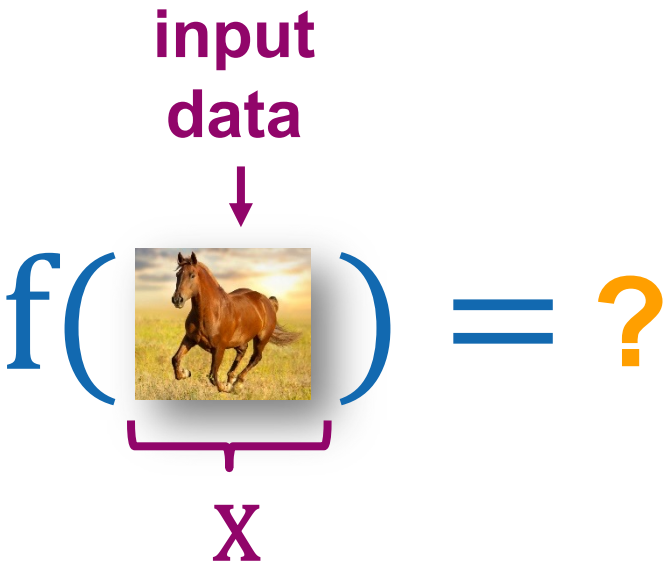
Goal of AI/ML models

- Learn a function f( ) to some data X to predict a **result**.

$$f(x) = ?$$

Goal of AI/ML models

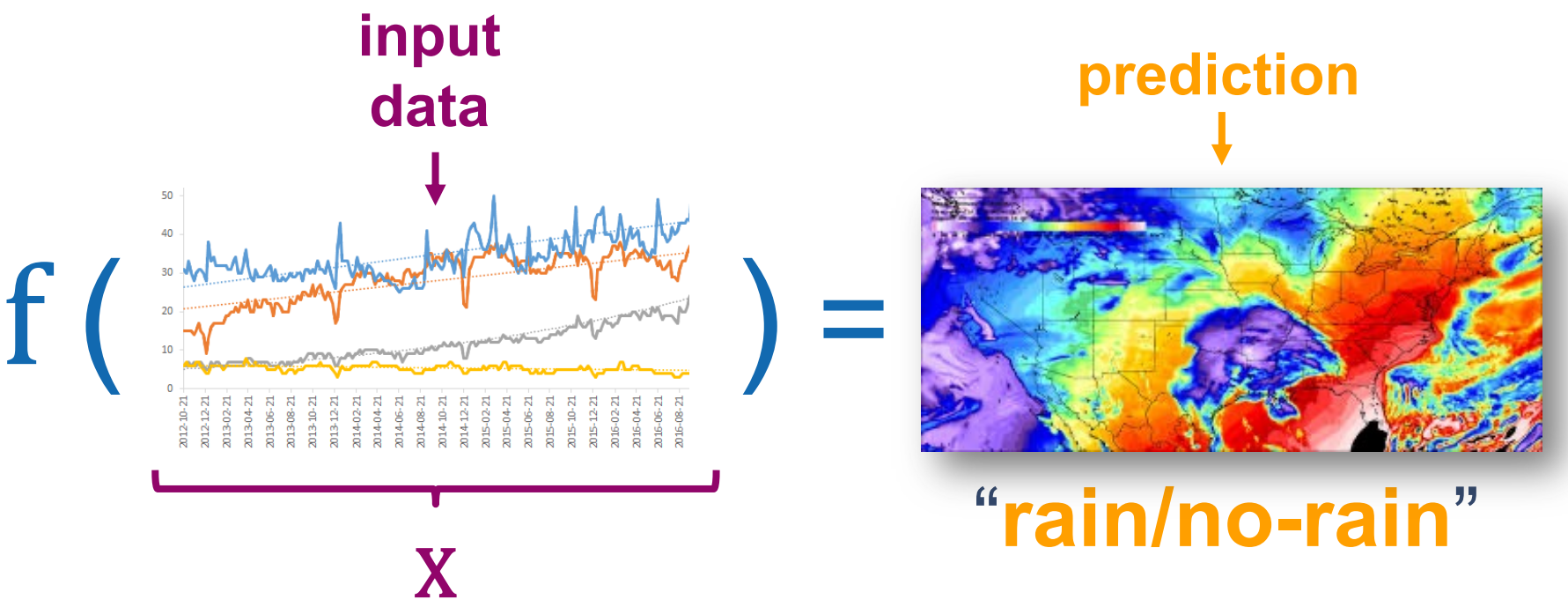
- Learn a function f( ) to some data X to predict a **result**.
- X = images



Goal of AI/ML models

- Learn a function f( ) to some data X to predict a result.
- X = sensor data

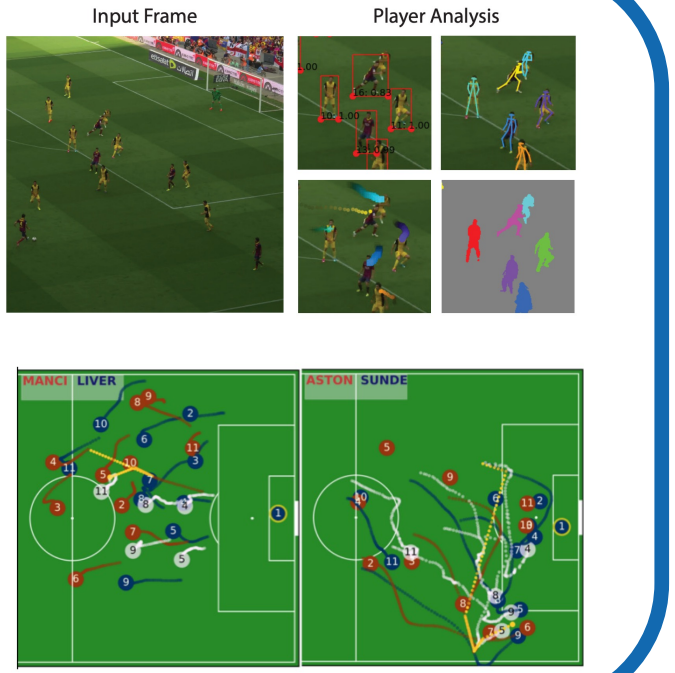
input data



Goal of AI/ML models

- Learn a function f( ) to some data X to predict a result.
- X = {video, sensor data}

f



=

prediction

Season
2018

Torneo de Verano		Apertura							Clausura
Club		MP	W	D	L	GF	GA	GD	Pts	Last 5
1	Melgar	15	9	3	3	21	13	8	30	
2	Alianza Lima	15	8	3	4	20	14	6	27	
3	Ayacucho	15	7	5	3	30	25	5	26	
4	Universitario	15	8	2	5	20	17	3	25	
5	Sporting Cristal	15	7	3	5	37	14	23	24	

"champion 2023"

X

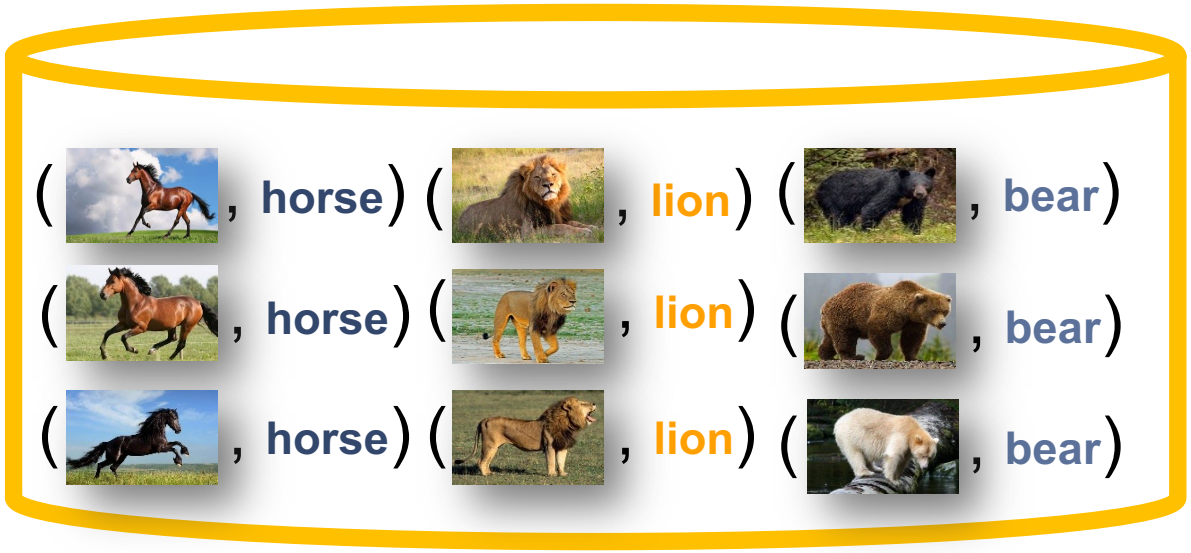
Goal of AI/ML models

$$y = f(x)$$

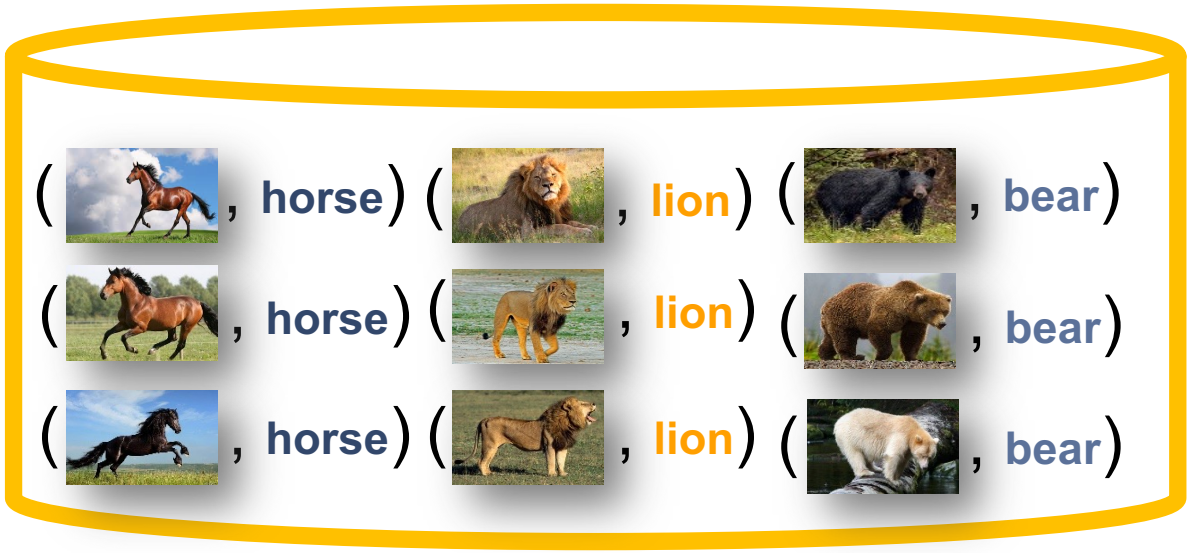
output — prediction function — input

- **Training** (or **learning**): given a *training set* of labeled examples $\{(x_1, y_1), \ldots, (x_N, y_N)\}$, instantiate a predictor $f(\,)$
- **Testing** (or **inference**): apply $f(\,)$ to a new *test example* x and output the predicted value $y = f(x)$

How A.I works?


(, horse) (, lion) (, bear)
(, horse) (, lion) (, bear)
(, horse) (, lion) (, bear)
Training set
training
f( )
Prediction function

How A.I works?



Training set

training

f( )

Prediction function

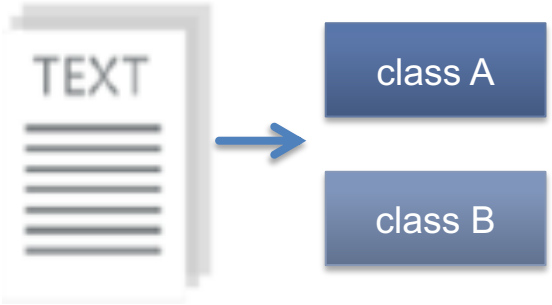
$\hat{y}$ =f( )

Prediction

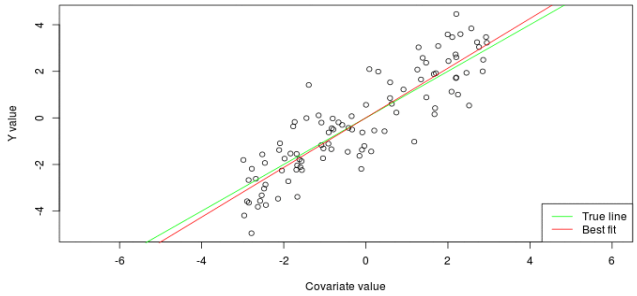
Unseen/new image

Types of A.I models

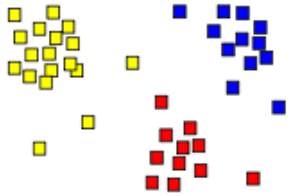
- ***Supervised***: learning with labeled data
  - Example: email classification, image classification
  - Example: regression for predicting real-valued outputs
- ***Unsupervised***: discover patterns in unlabeled data
  - Example: cluster similar data points
- ***Reinforcement learning***: learn to act based on feedback/reward
  - Example: learn to play Go



Classification



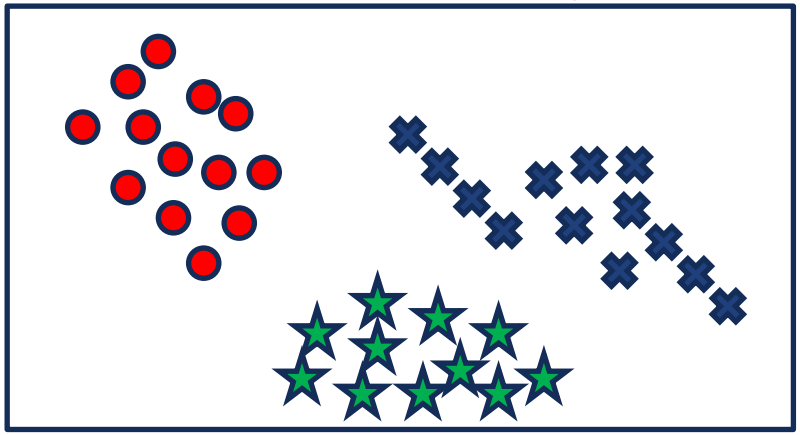
Regression



Clustering

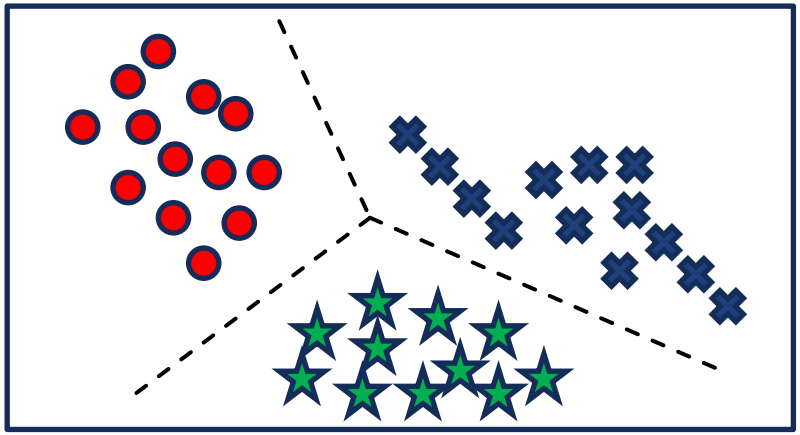
Types of A.I models

Supervised learning



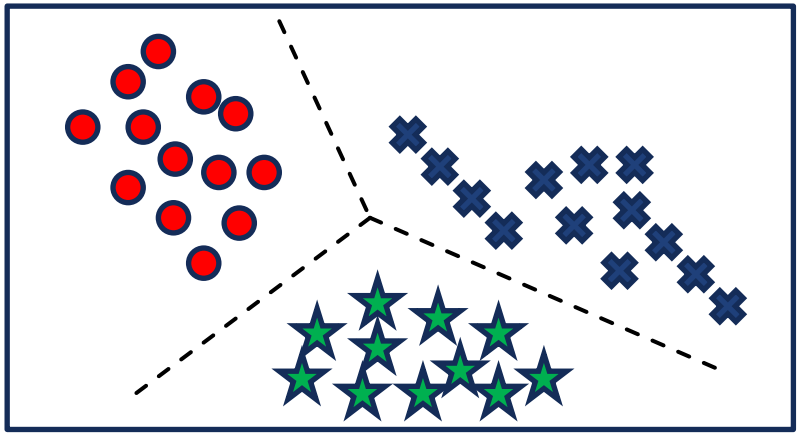
Types of A.I models

$$\{x_n \in R^d, y_n \in R\}_{n=1}^{N}$$

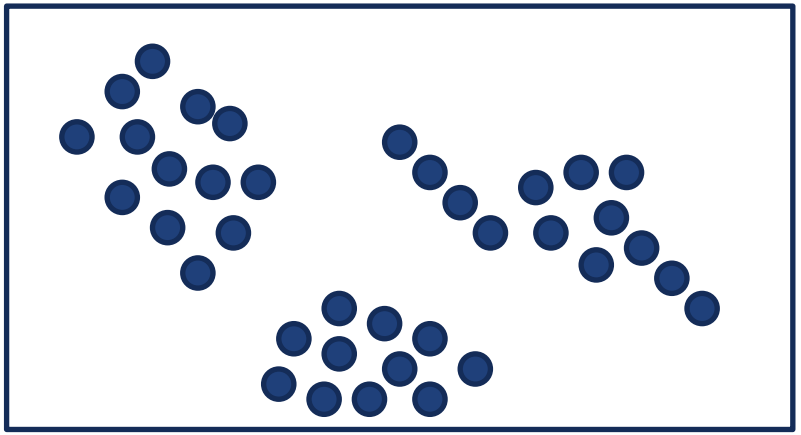


Types of A.I models

$\{x_n \in R^d, y_n \in R\}_{n=1}^{N}$

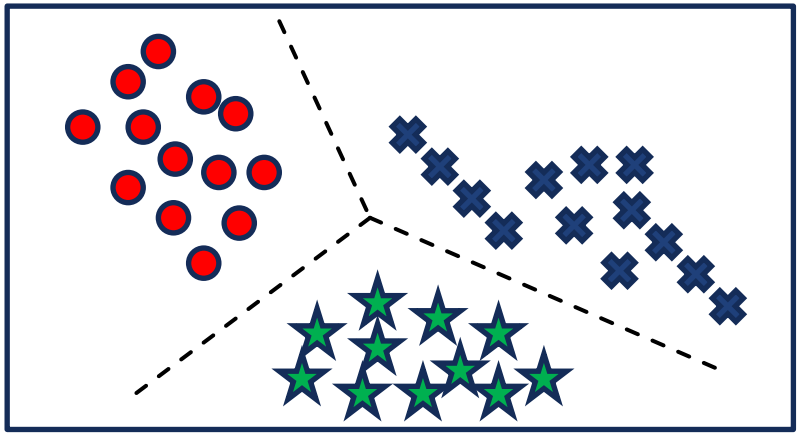


Unsupervised learning

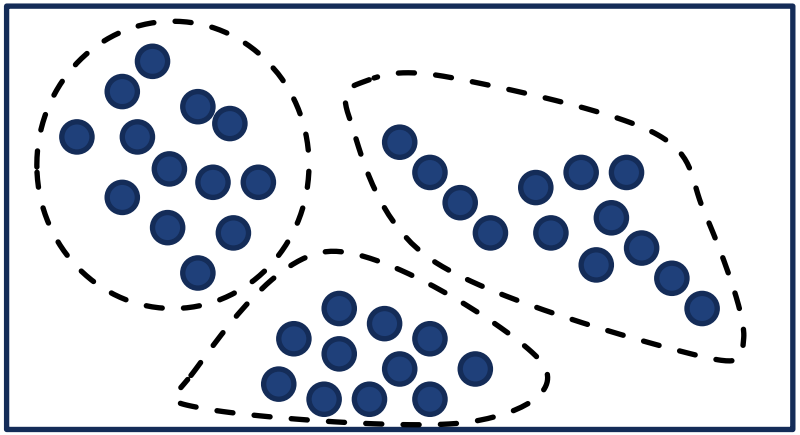


Types of A.I models

$\{x_n \in R^d, y_n \in R\}_{n=1}^N$

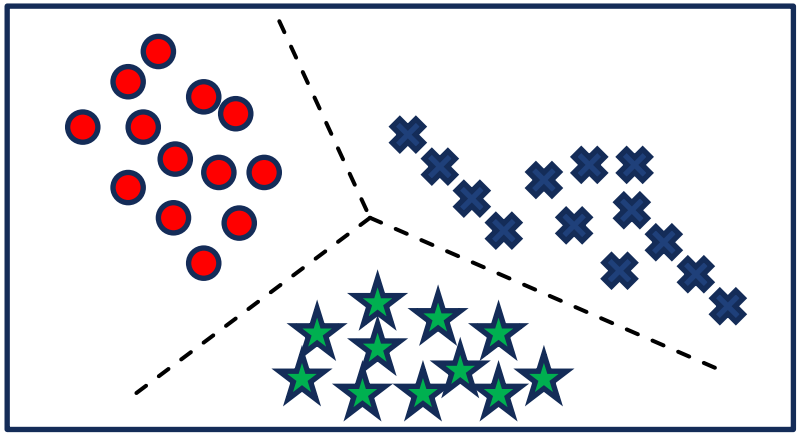


$\{x_n \in R^d\}_{n=1}^N$

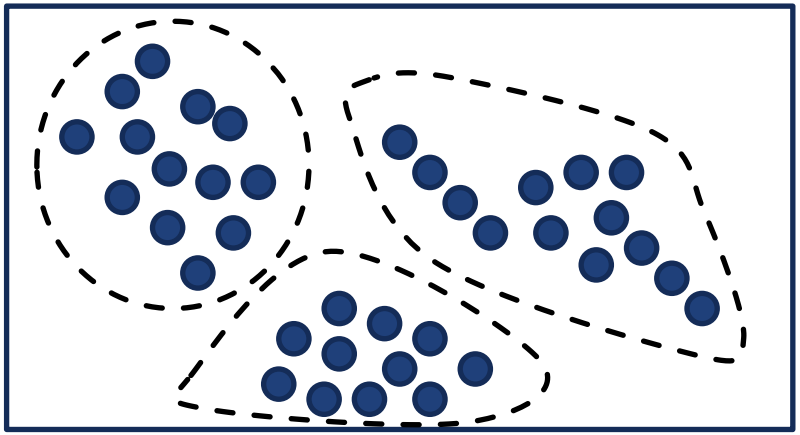


Types of A.I models

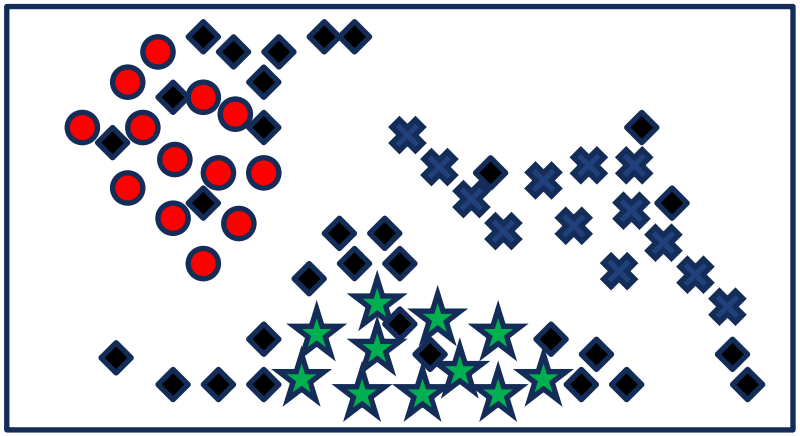
$\{x_n \in R^d, y_n \in R\}_{n=1}^N$



$\{x_n \in R^d\}_{n=1}^N$

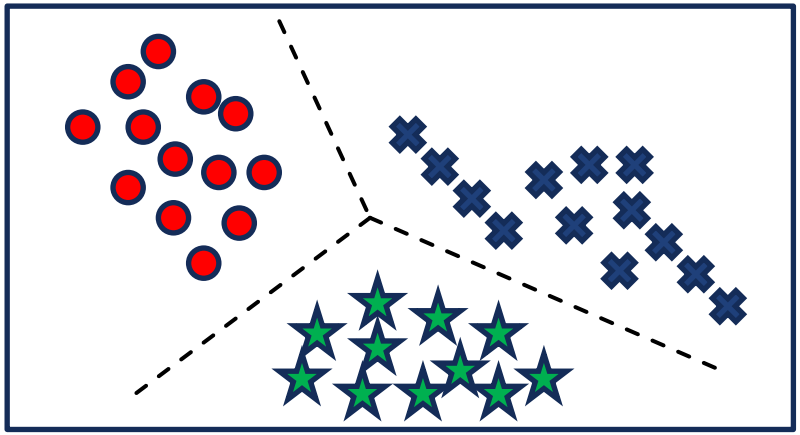


Semi-supervised learning

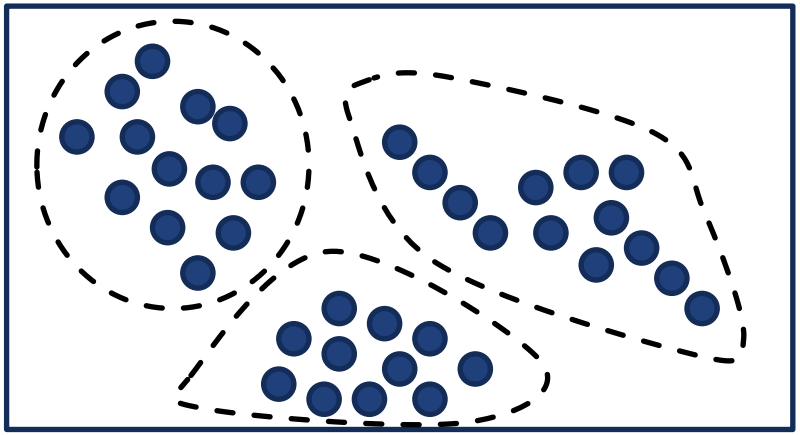


Types of A.I models

$\{x_n \in R^d, y_n \in R\}_{n=1}^N$

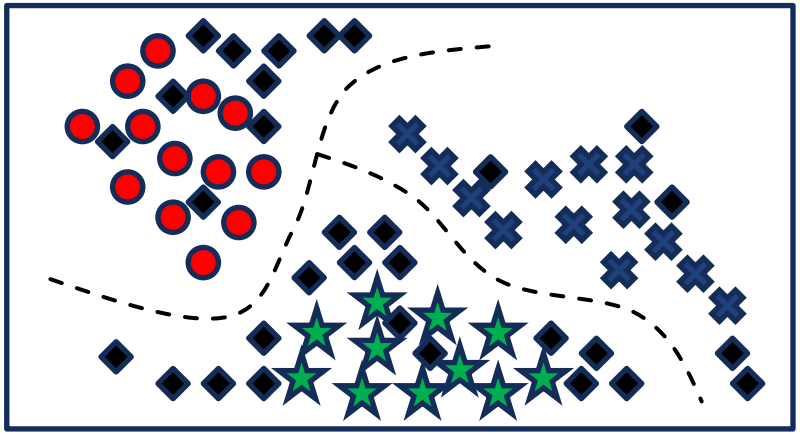


$\{x_n \in R^d\}_{n=1}^N$



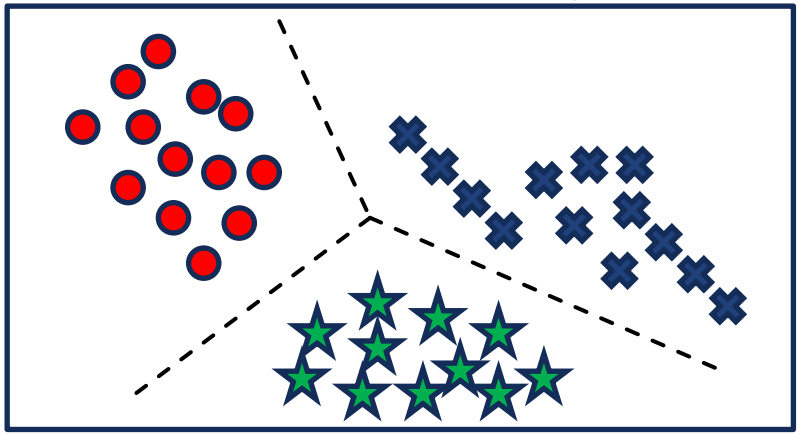
$\{x_n \in R^d, y_n \in R\}_{n=1}^N$

$\{x_n \in R^d\}_{n=1}^N$

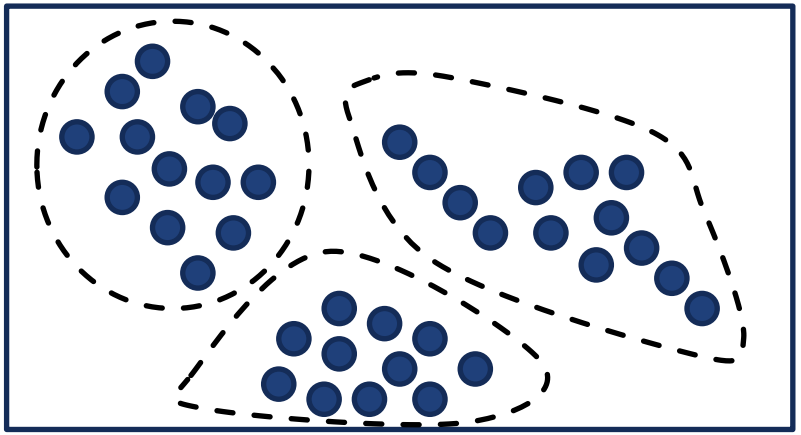


Types of A.I models

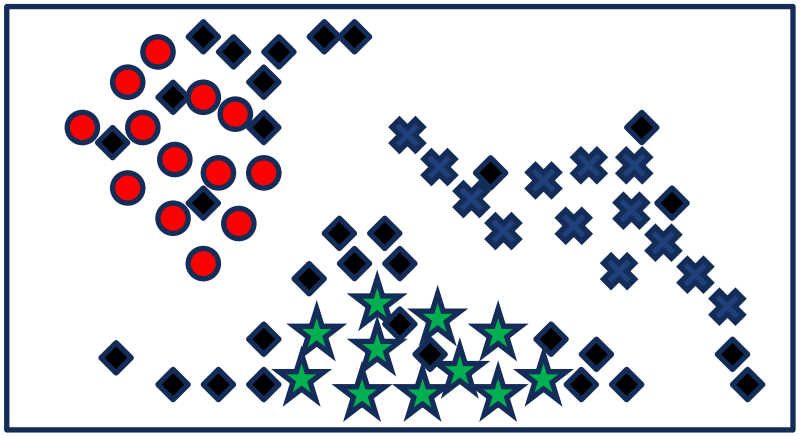
Supervised learning



Unsupervised learning



Semi-supervised learning



Types of A.I models

Semi-supervised
(labels for a small portion
of training data)

Unsupervised
(no labels)

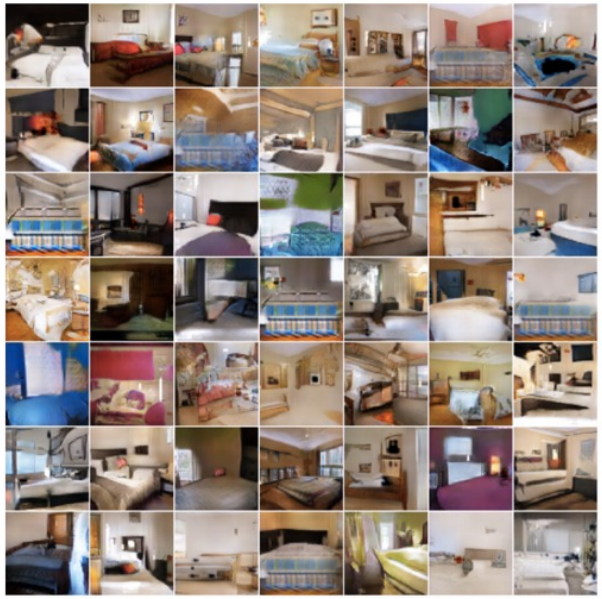
Weakly supervised (noisy
labels, labels not exactly
for the task of interest)

Supervised
(clean, complete
training labels
for the task of
interest)

Generative Learning

- Learning the data distribution
  - **Learning to sample:** Produce samples from a data distribution that mimics the training set

Generative adversarial networks



"Bedroom"

"Face"

Self-supervised or Predictive Learning

- Use part of the data to predict other parts of the data
  - Example: Image colorization

R. Zhang et al., Colorful Image Colorization, ECCV 2016

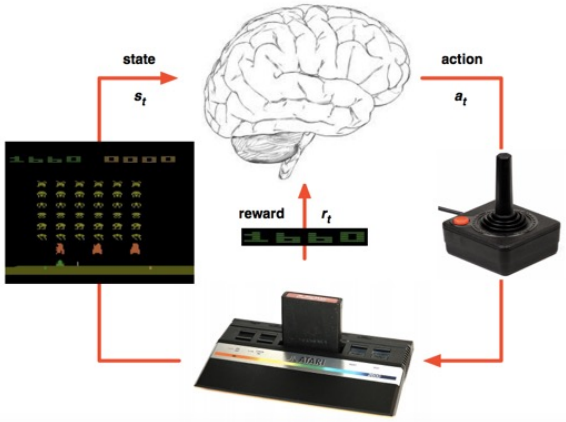
Reinforcement Learning

- Learn from rewards in a *sequential* environment

Arthur Samuel's checkers system

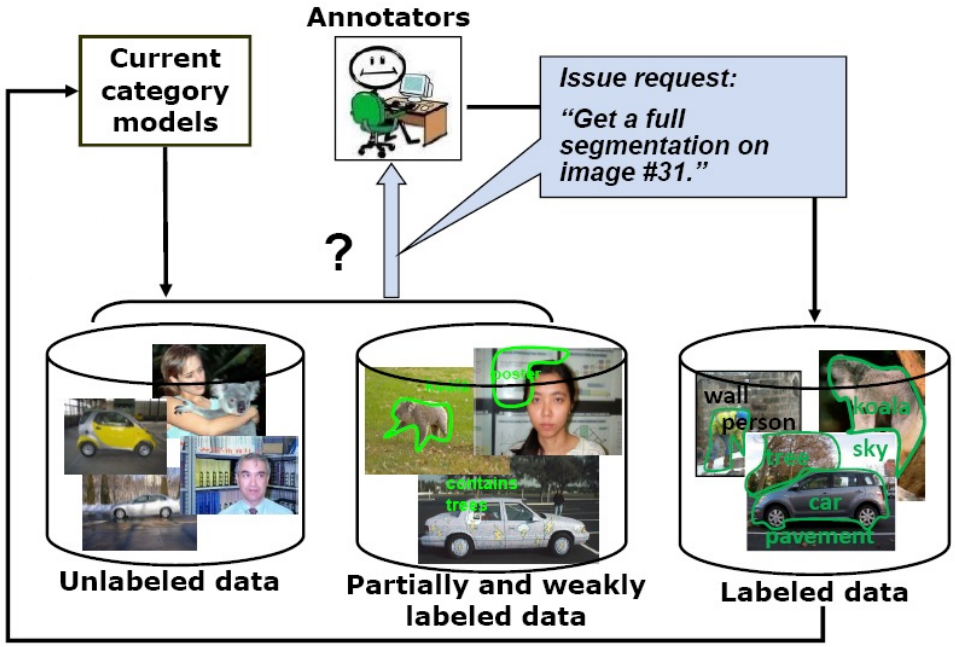
DeepMind's AlphaGo

DeepMind's Atari system



Active Learning

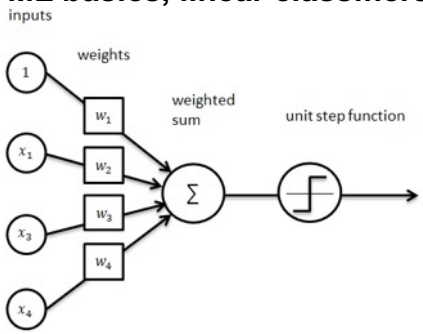
- The learning algorithm can choose its own training examples, or ask a “teacher” for an answer on selected inputs



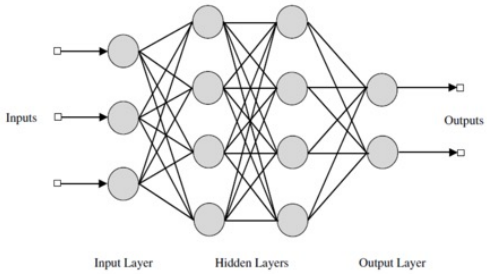
S. Vijayanarasimhan and K. Grauman. Cost-Sensitive Active Visual Category Learning. IJCV 2010

Class Overview

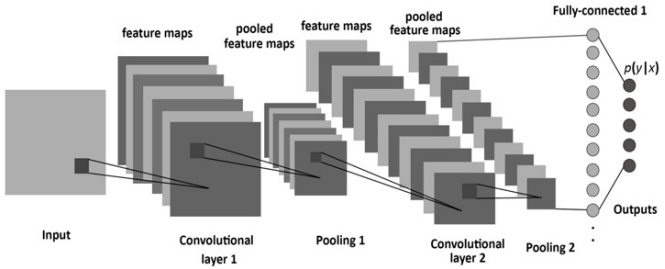
ML basics, linear classifiers



Multilayer neural networks, backpropagation



Convolutional networks for classification

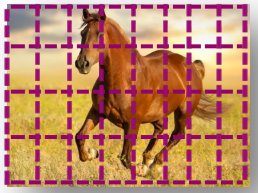


The A.I Training Pipeline

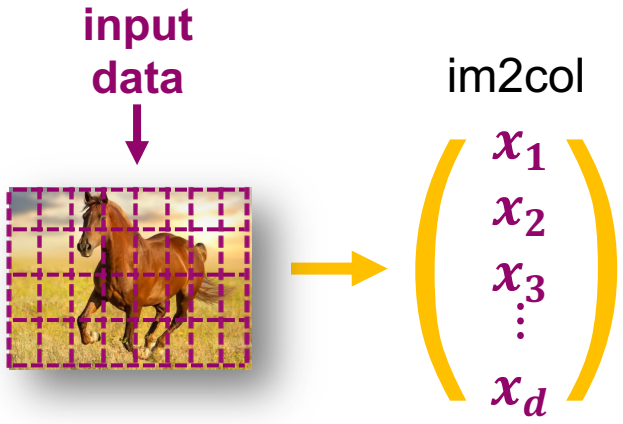
input data

The A.I Training Pipeline

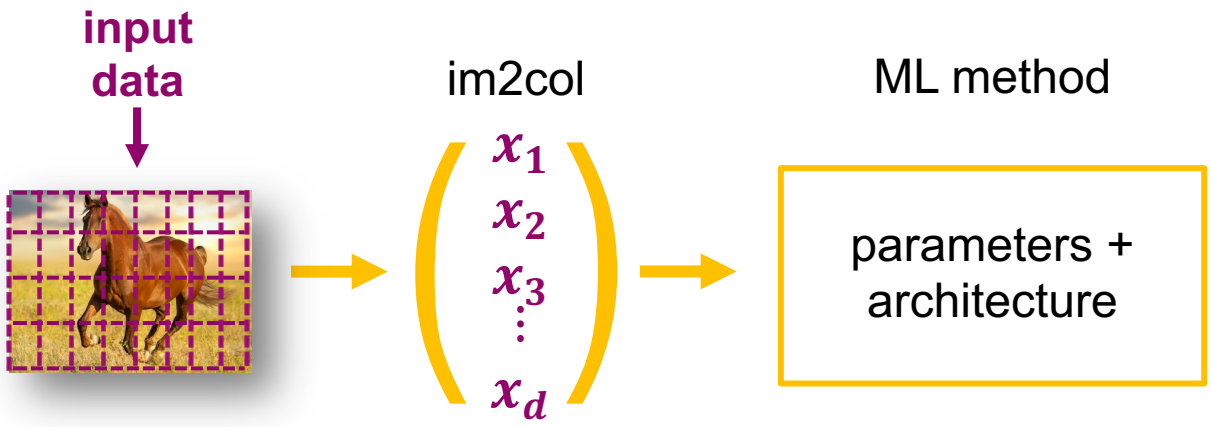
input data



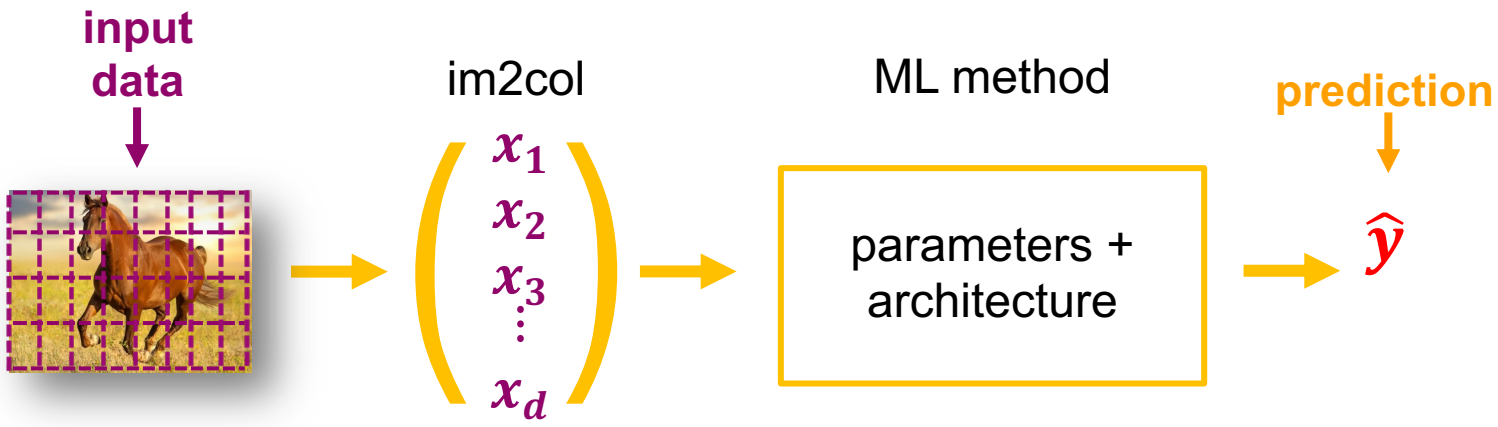
The A.I Training Pipeline



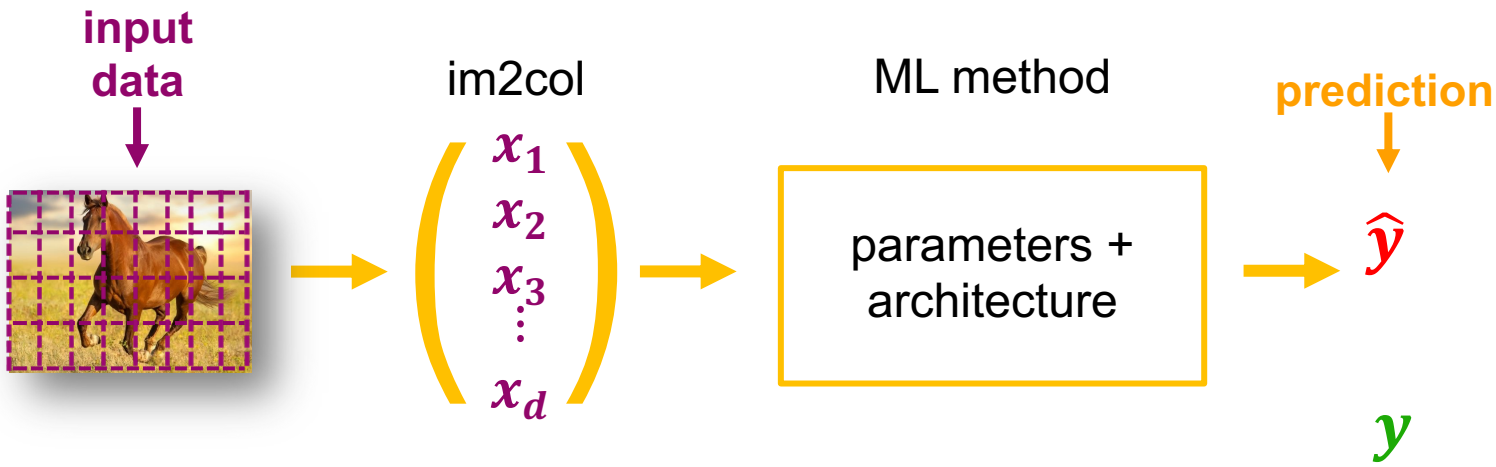
The A.I Training Pipeline



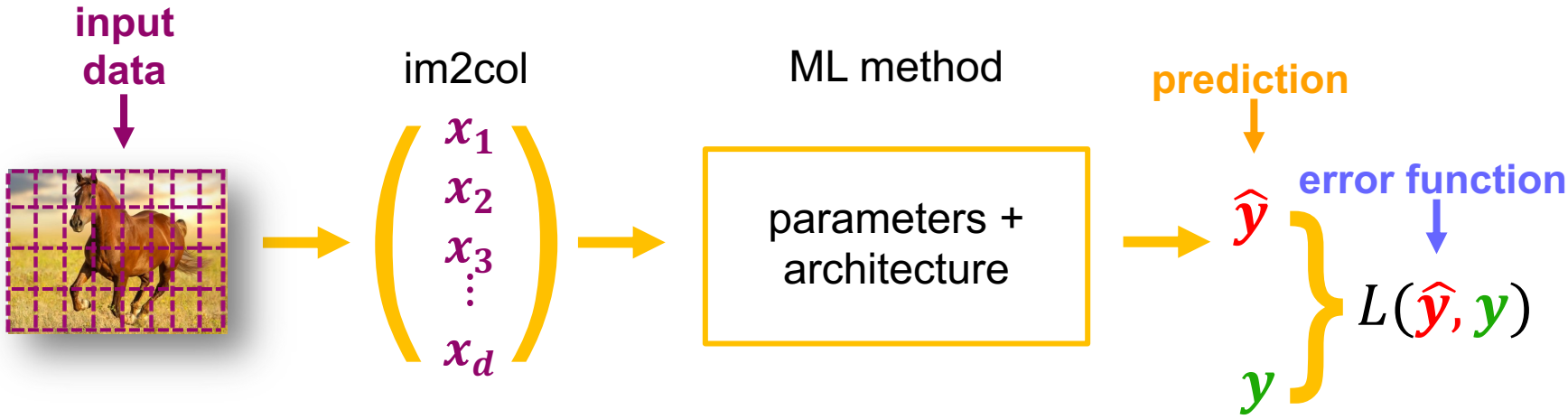
The A.I Training Pipeline



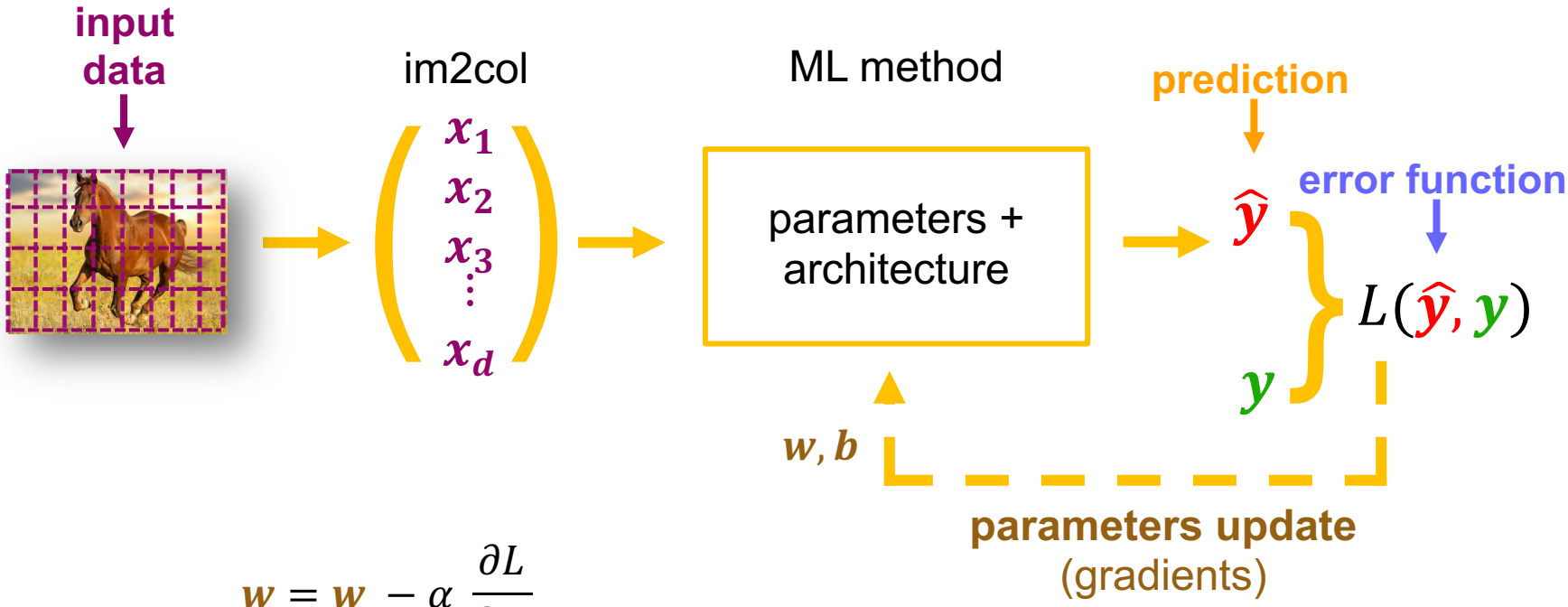
The A.I Training Pipeline



The A.I Training Pipeline



The A.I Training Pipeline

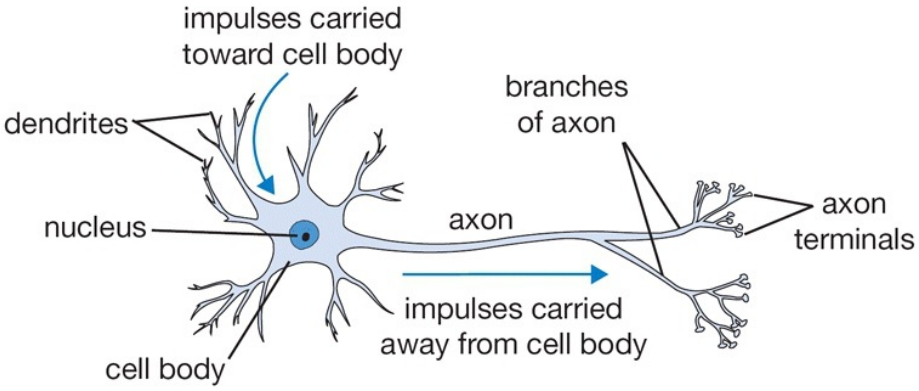


$$w = w - \alpha \frac{\partial L}{\partial w}$$

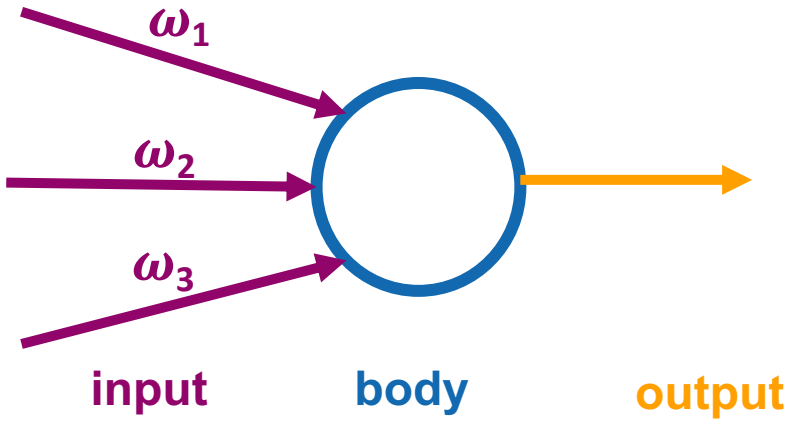
$$b = b - \alpha \frac{\partial L}{\partial b}$$

Building Deep Learning Models: The Perceptron Activation Functions

The Perceptron

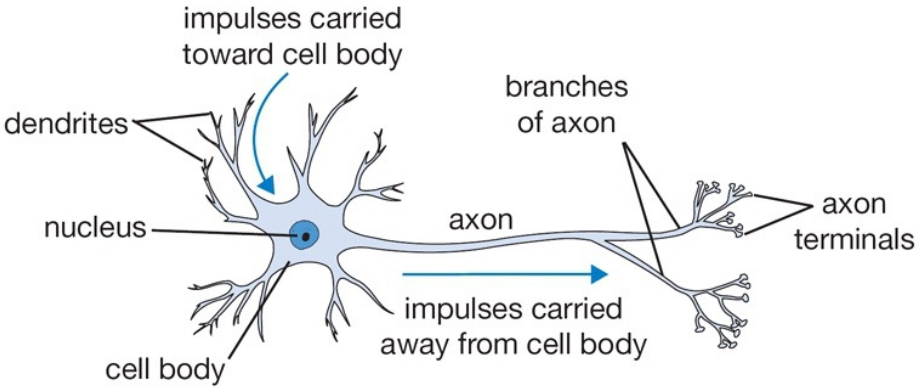


A biological neuron

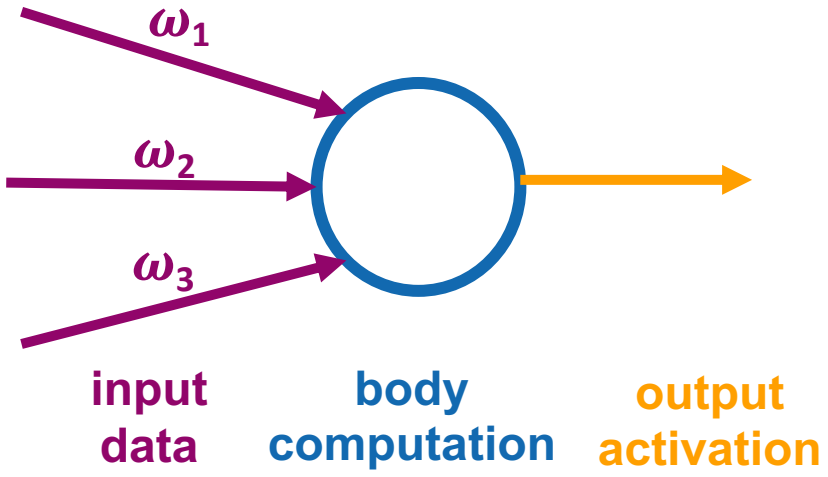


An artificial neuron

The Perceptron

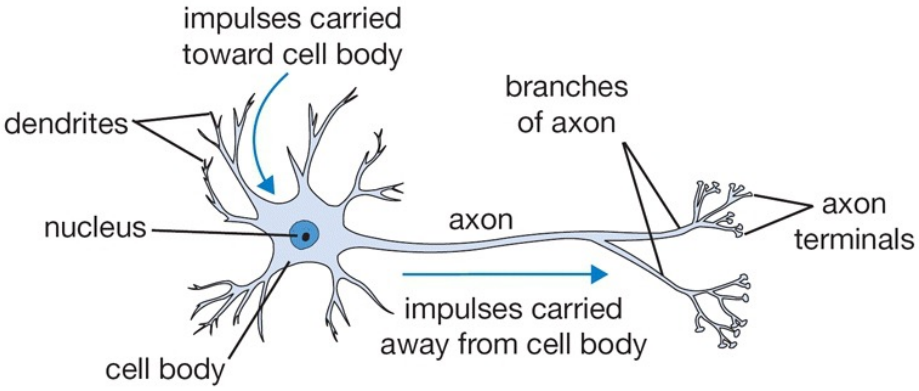


A biological neuron

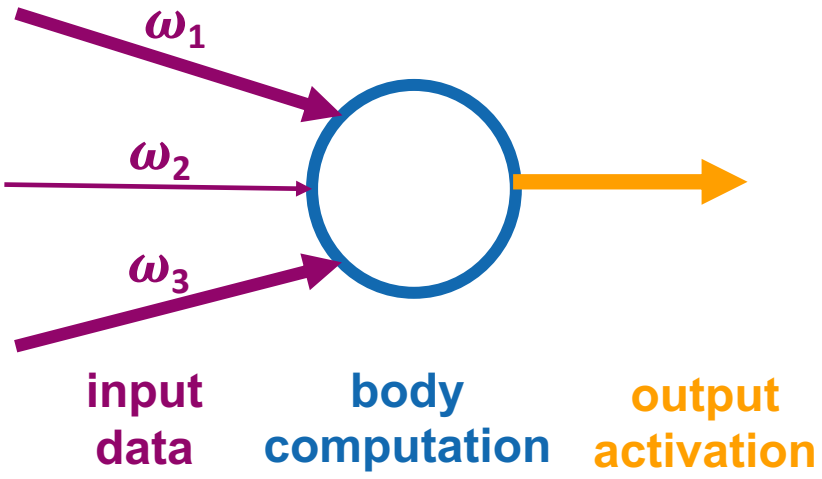


An artificial neuron

The Perceptron

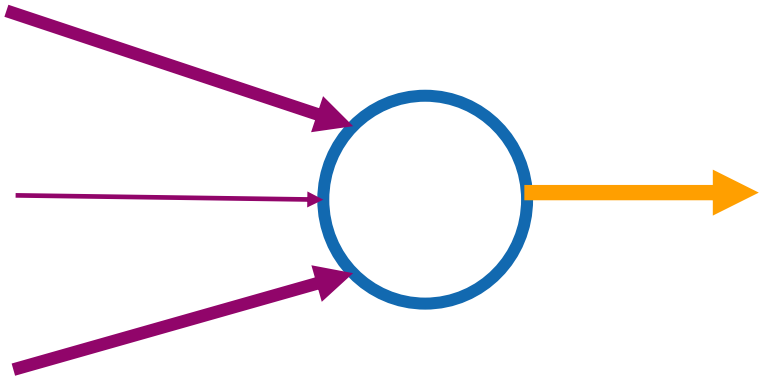


A biological neuron

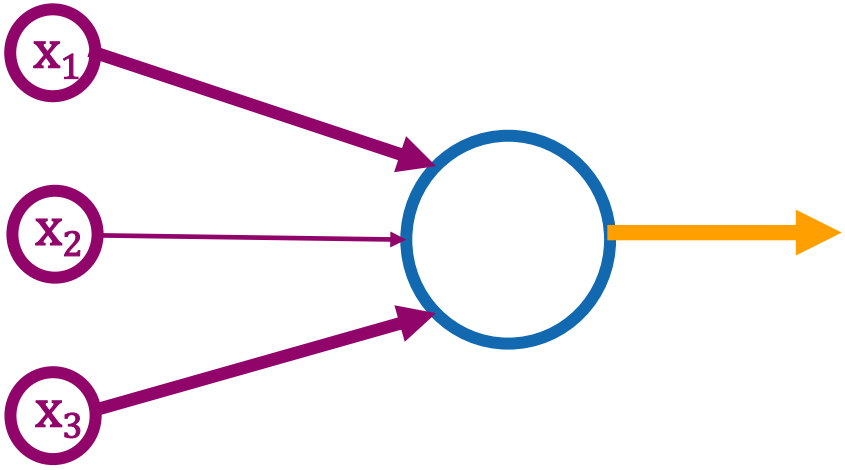


An artificial neuron

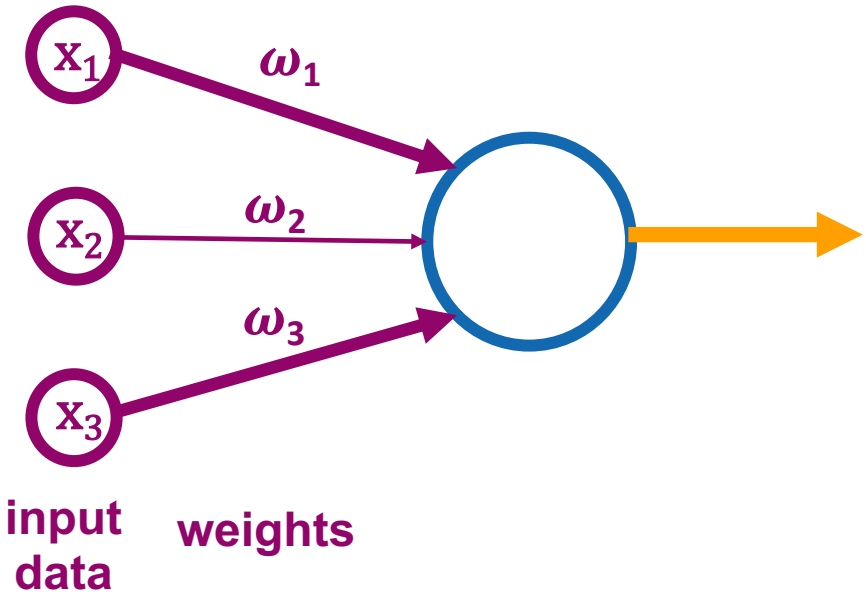
The Perceptron



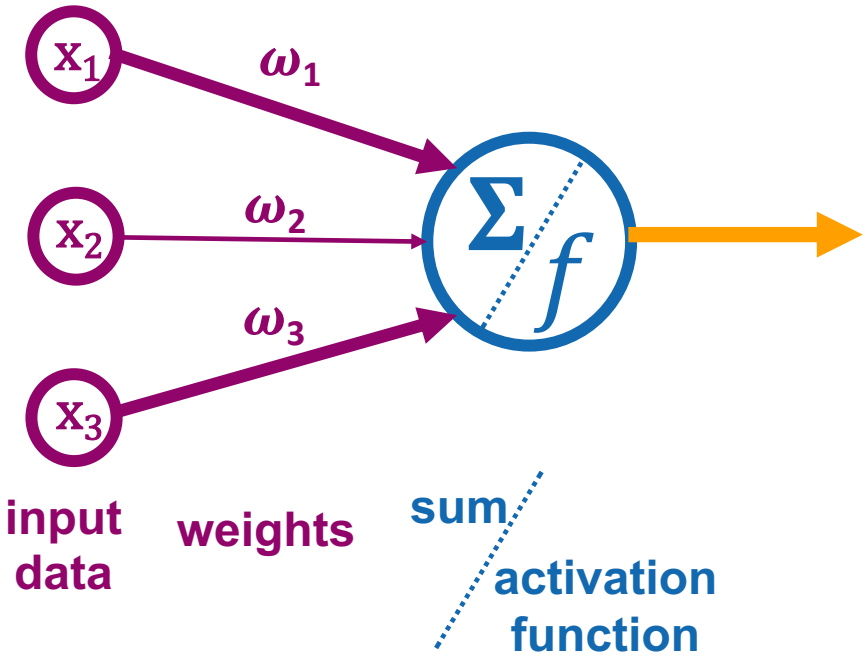
The Perceptron



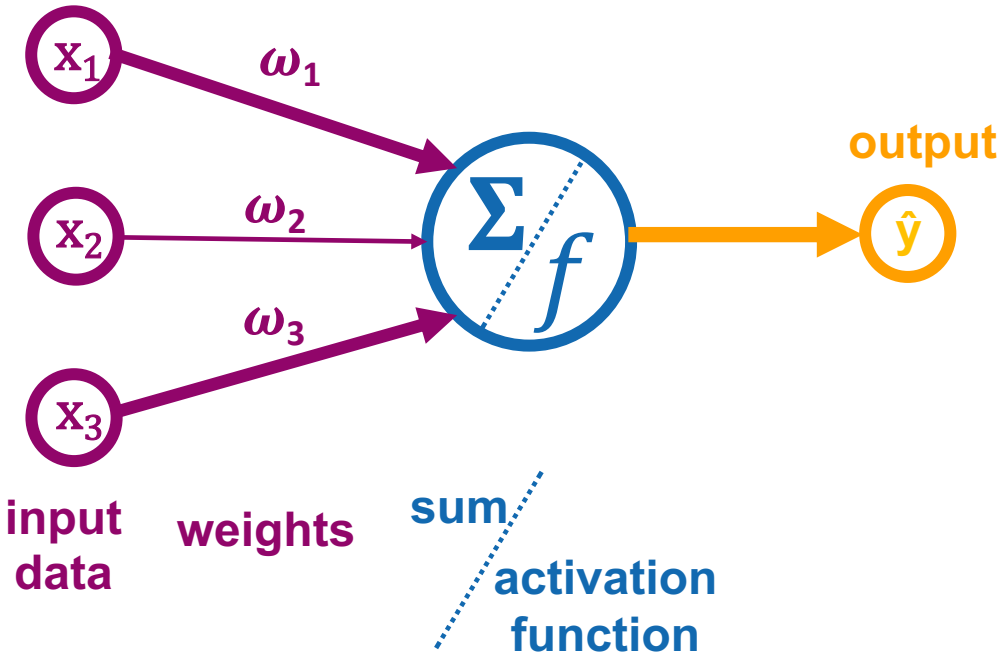
The Perceptron



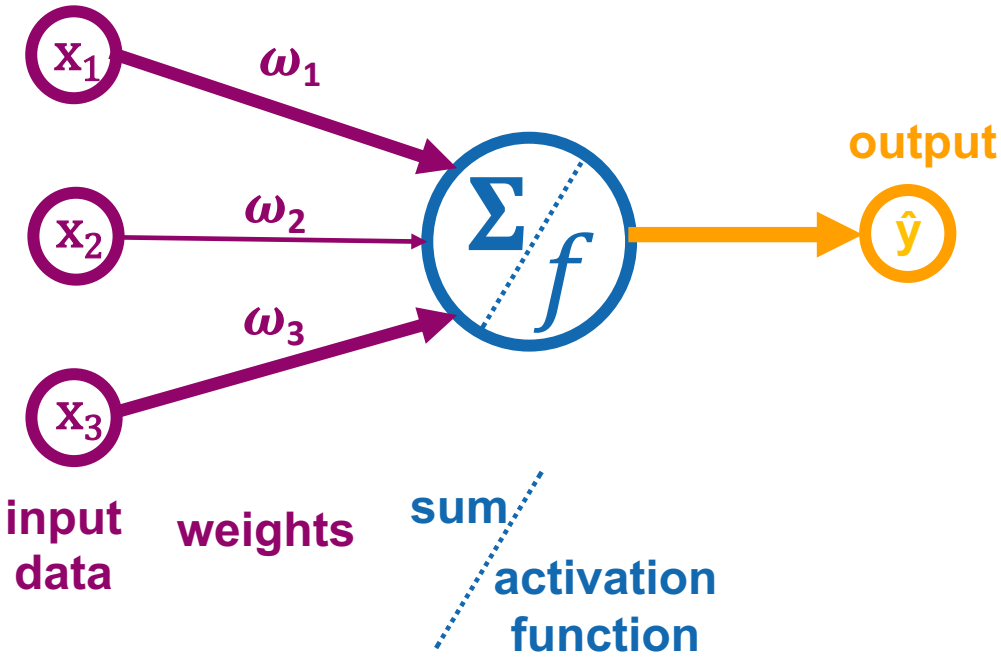
The Perceptron



The Perceptron



The Perceptron

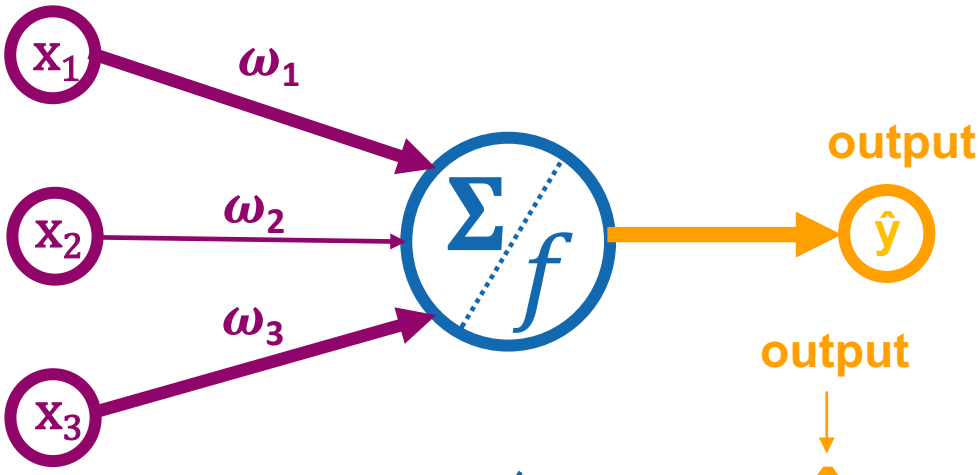


output
input data
weights

$$\hat{y} = f(x, \omega)$$

computation

The Perceptron



input data

weights

sum

activation function

output

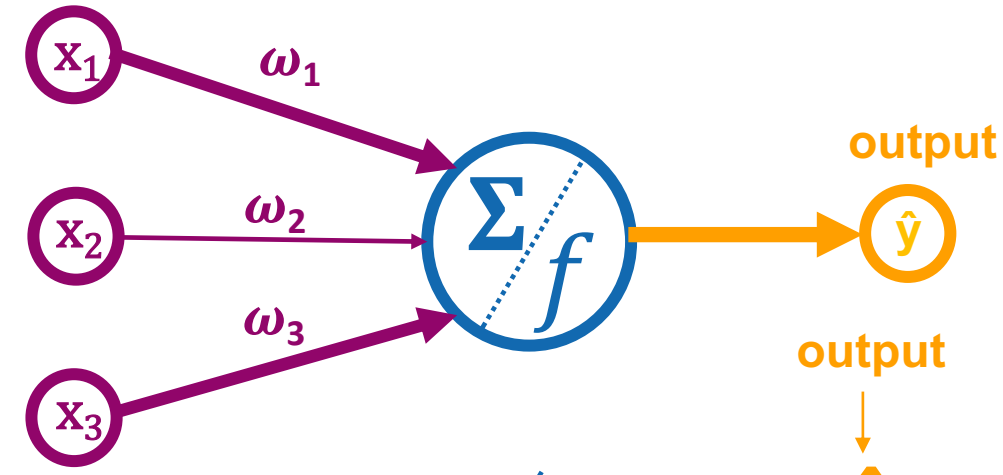
$$\hat{y} = f\left((x_1, x_2 \cdots x_d), (\omega_1, \omega_2 \cdots \omega_d)\right)$$

input data

weights

computation

The Perceptron



input data

weights

sum

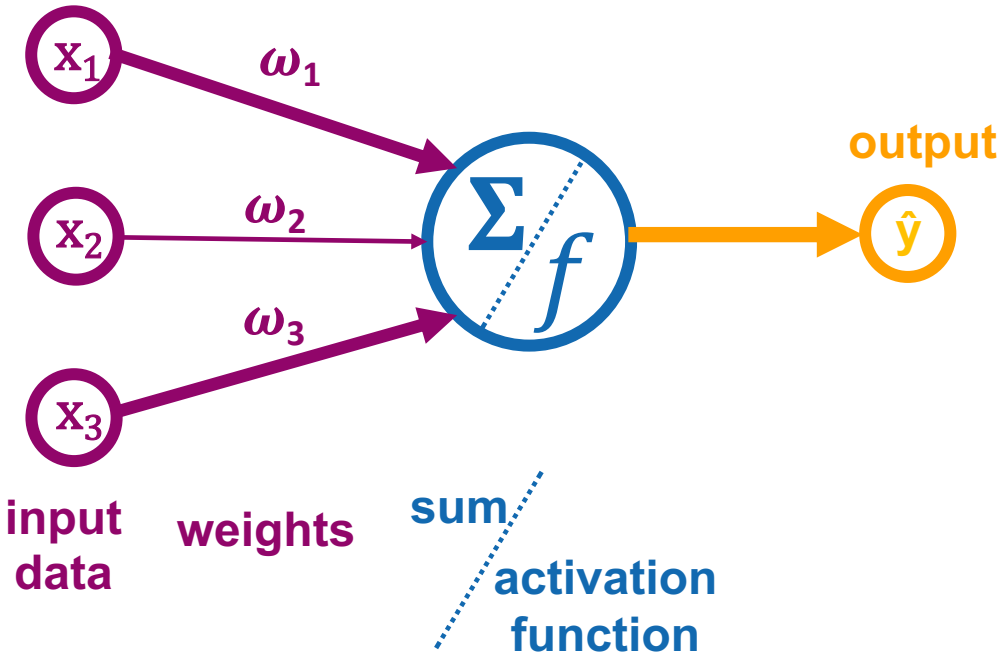
activation function

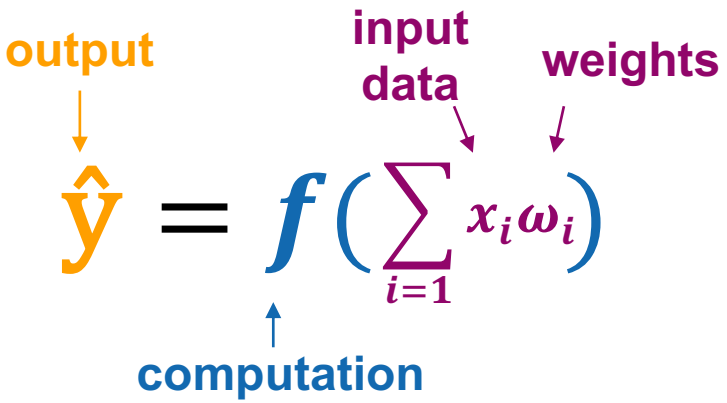
output

$$\hat{y} = f\left(\underbrace{x_1\omega_1 + x_2\omega_2 + \cdots + x_d\omega_d}_{\text{input data x weights}}\right)$$

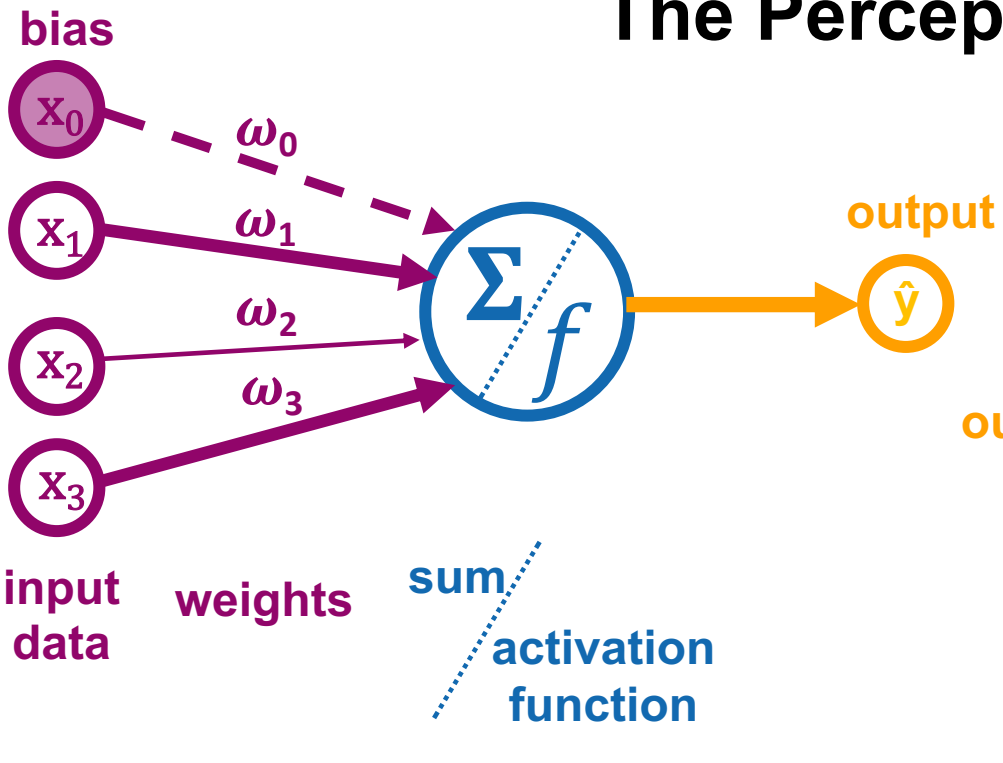
computation

The Perceptron





The Perceptron

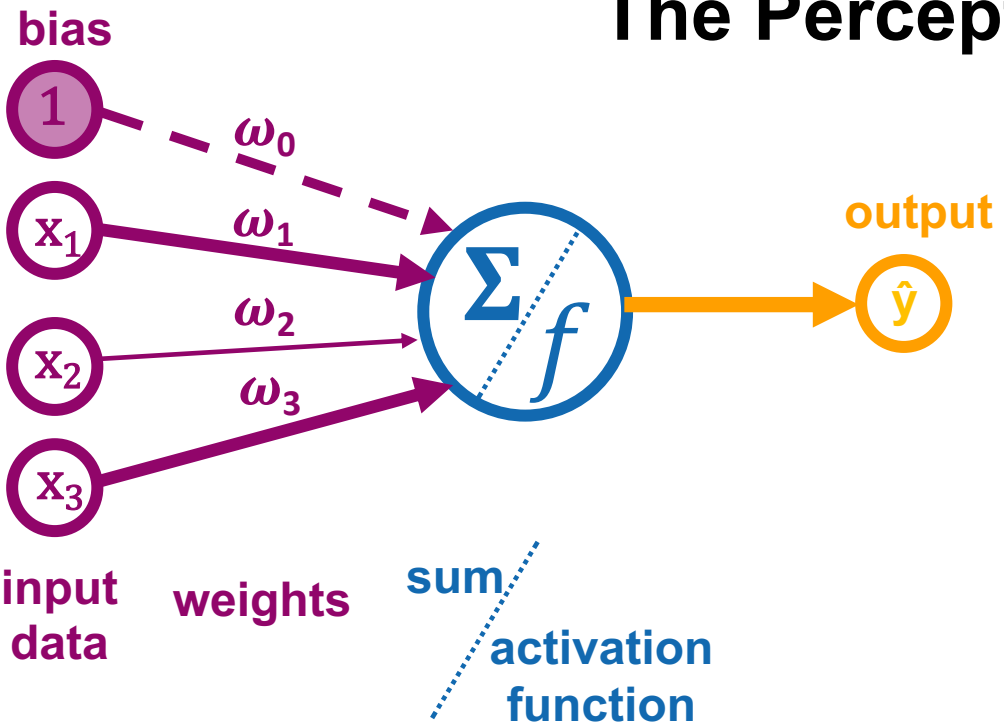


output
input data
weights

$$\hat{y} = f\left(\omega_0 + \sum_{i=1} x_i \omega_i\right)$$

computation

The Perceptron

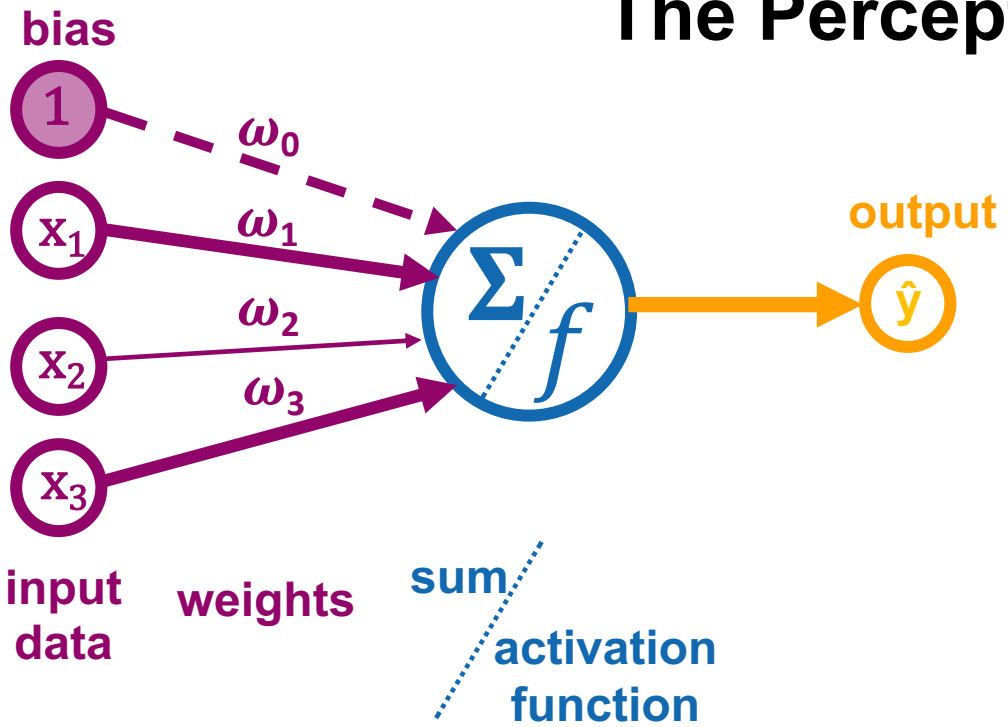


output input data weights

$$\hat{y} = f\left(\sum_{i=0} x_i \omega_i\right)$$

activation function

The Perceptron



output

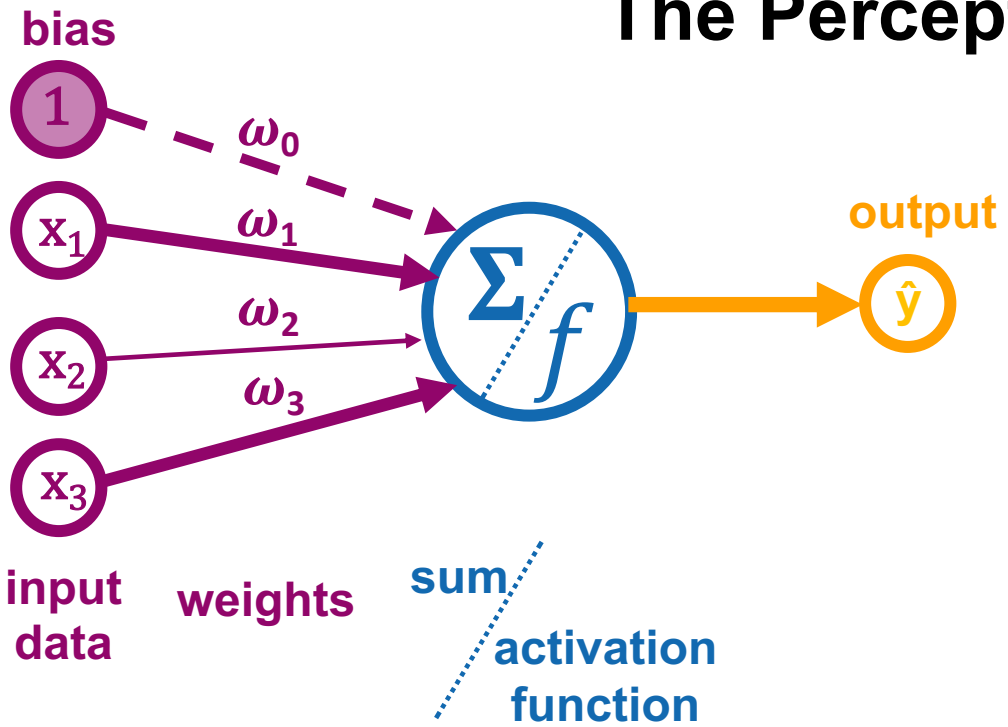
input data

weights

$$\hat{y} = f\left(\sum_{i=0} x_i \omega_i\right)$$

$$\hat{y} = f\left(X^T W\right)$$

The Perceptron



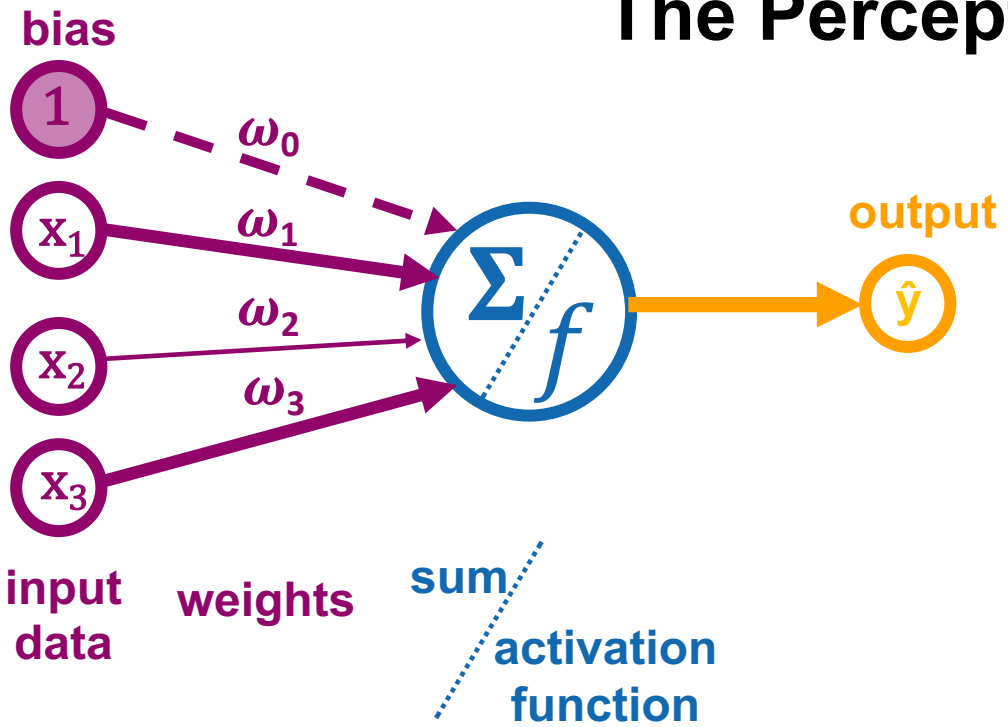
output $\hat{y}$ input data x_i weights ω_i

$$\hat{y} = f\left(\sum_{i=0} x_i \omega_i\right)$$

$$\hat{y} = f\left(X^T W\right)$$

where $X = \begin{bmatrix} X_0 \\ \vdots \\ X_d \end{bmatrix}$ and $W = \begin{bmatrix} \omega_0 \\ \vdots \\ \omega_d \end{bmatrix}$

The Perceptron

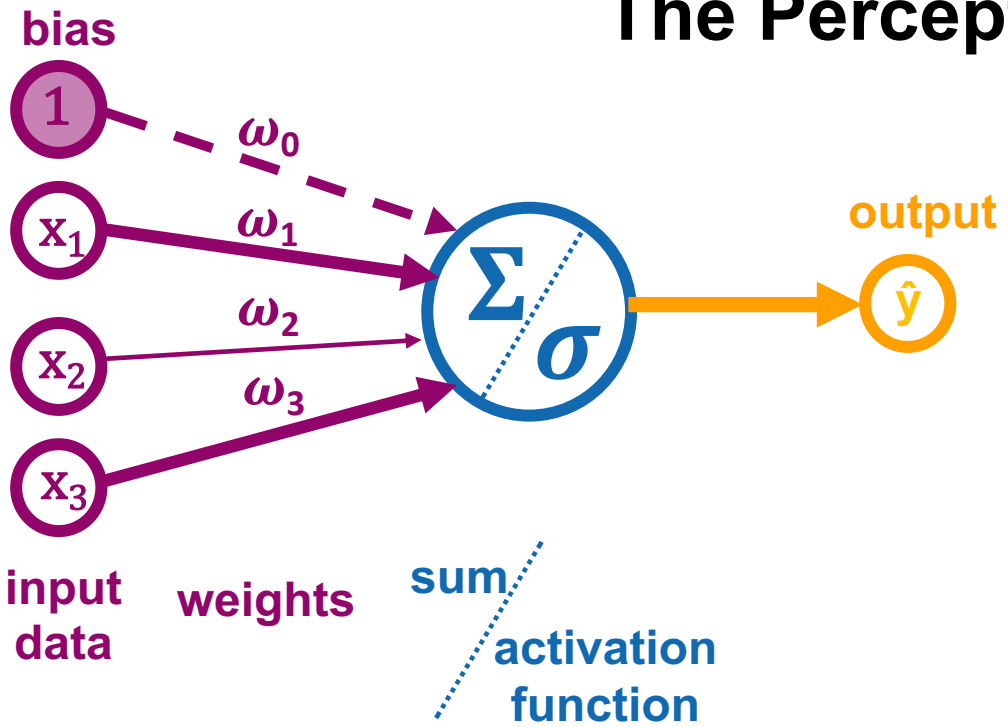


output
input data
weights

$$\hat{y} = f(x^T w)$$

activation function

The Perceptron

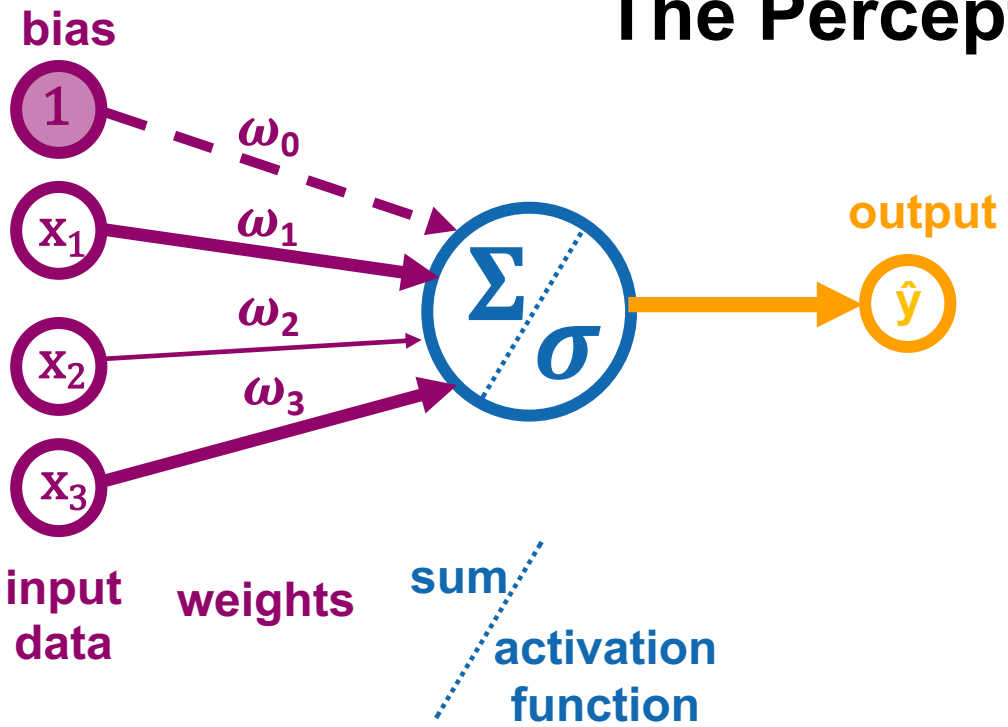


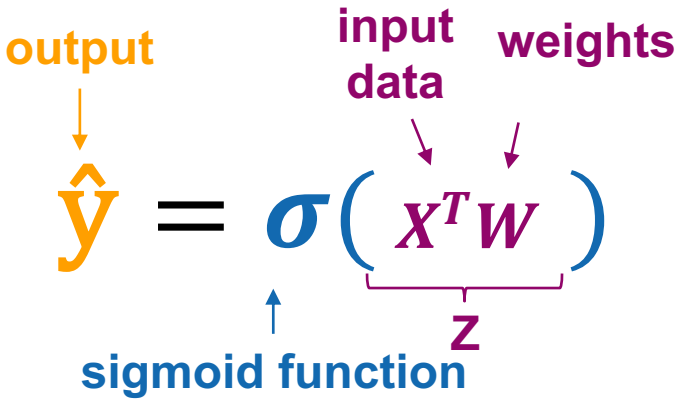
output
input data
weights

$$\hat{y} = \sigma\left(X^T W \right)$$

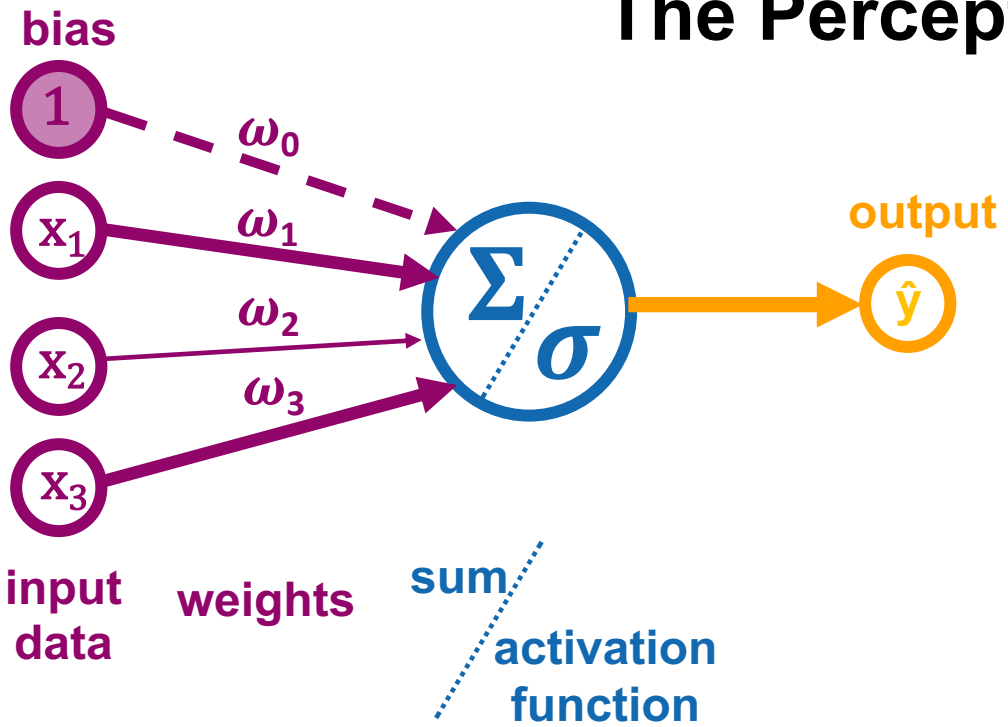
sigmoid function

The Perceptron





The Perceptron

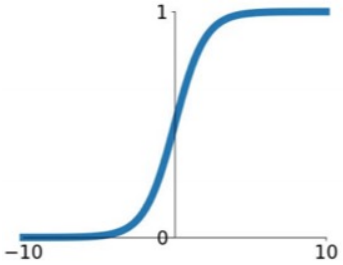


output input data weights

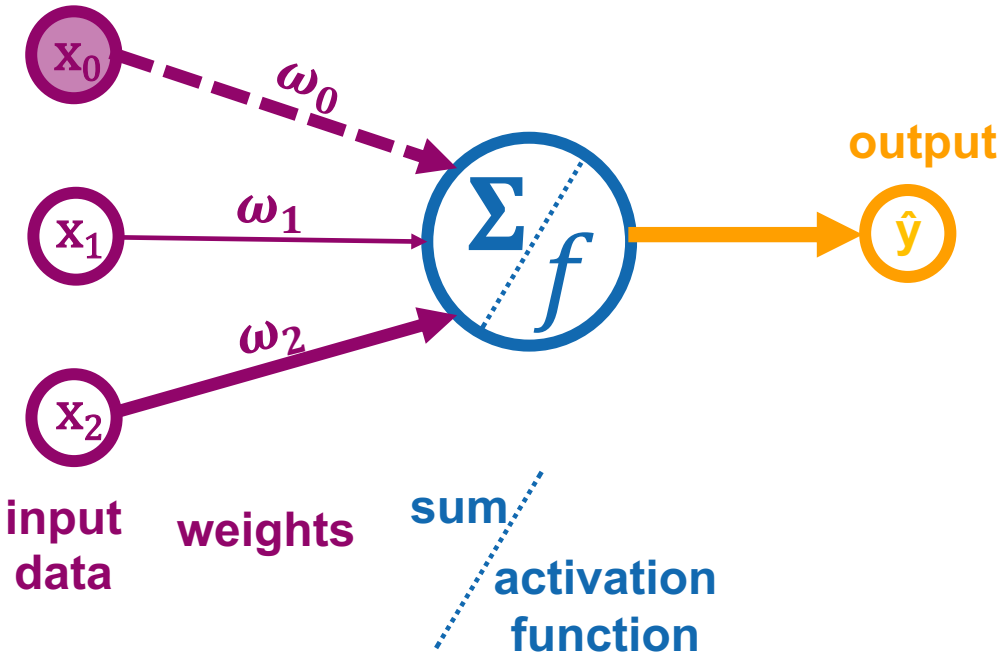
$$\hat{y} = \sigma(\underbrace{X^T W}_{z})$$

Sigmoid function:

$$f(z) = \sigma(z) = \frac{1}{1+e^{-z}} = \frac{1}{1+e^{-X^T W}}$$



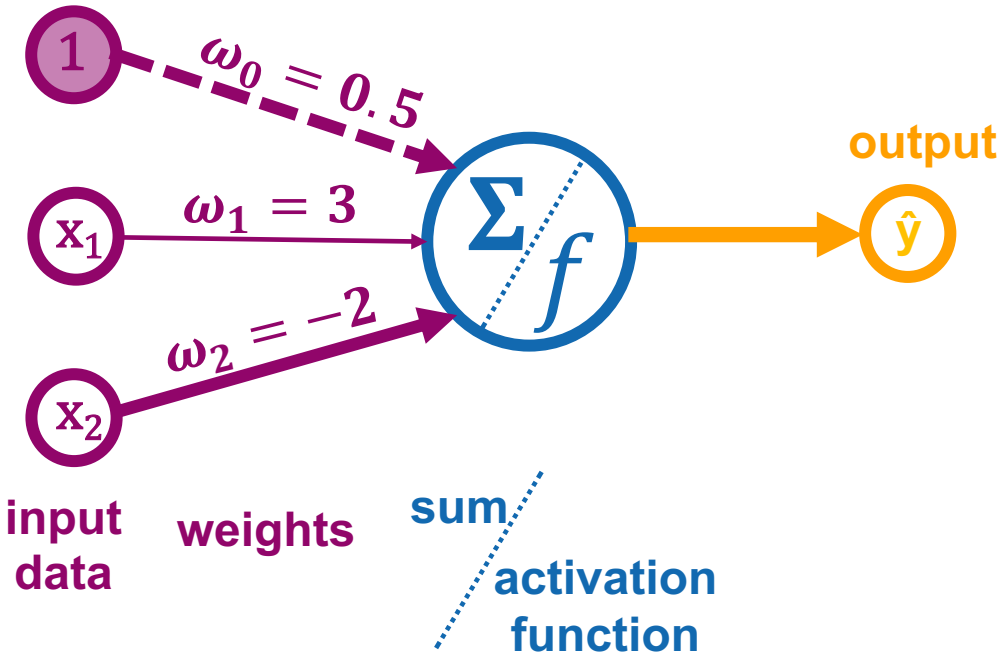
The Perceptron: Example



$$\hat{y} = f(X^T W)$$

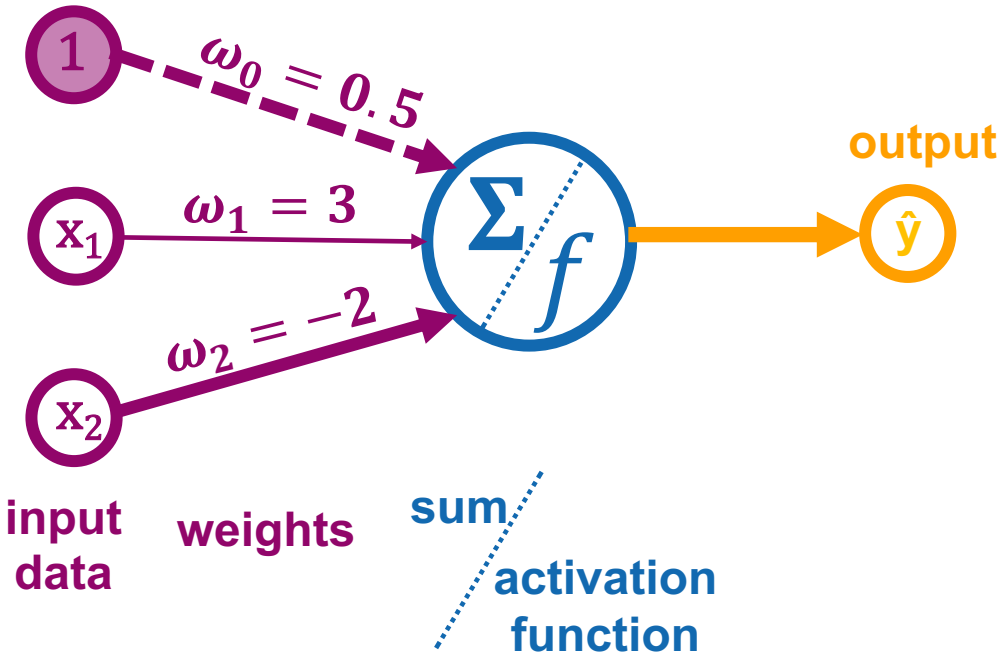
output: $\hat{y}$; input data: X; weights: W

The Perceptron: Example



output $\hat{y}$ = f (input data X^T weights W)

The Perceptron: Example

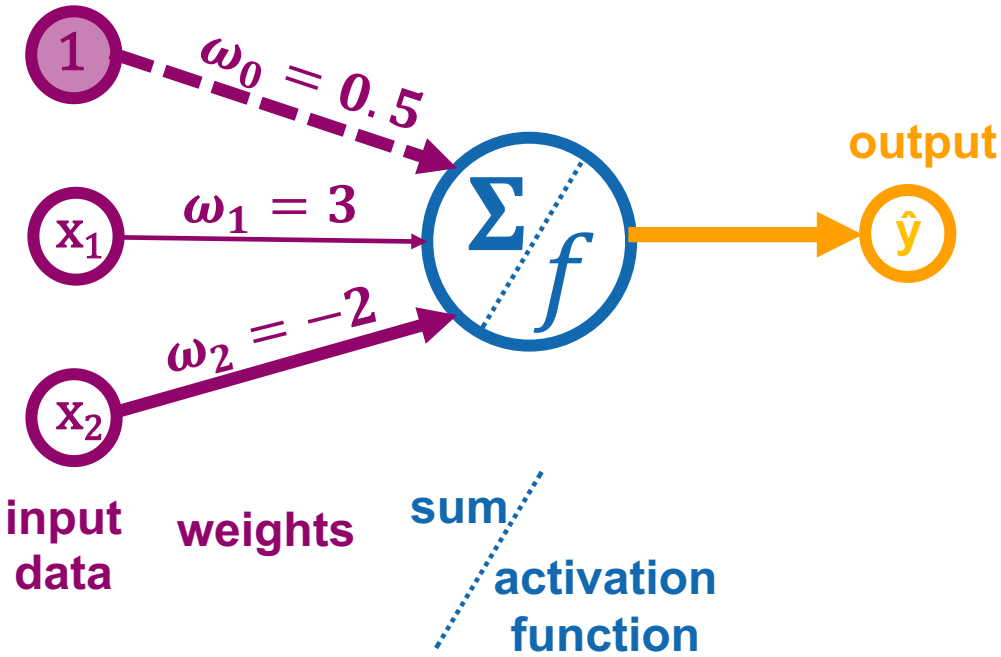


output ŷ, input data X, weights W

$$\hat{y} = f\left(X^T W\right)$$

$$\hat{y} = f\left(\begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 0.5 \\ 3 \\ -2 \end{bmatrix}\right)$$

The Perceptron: Example



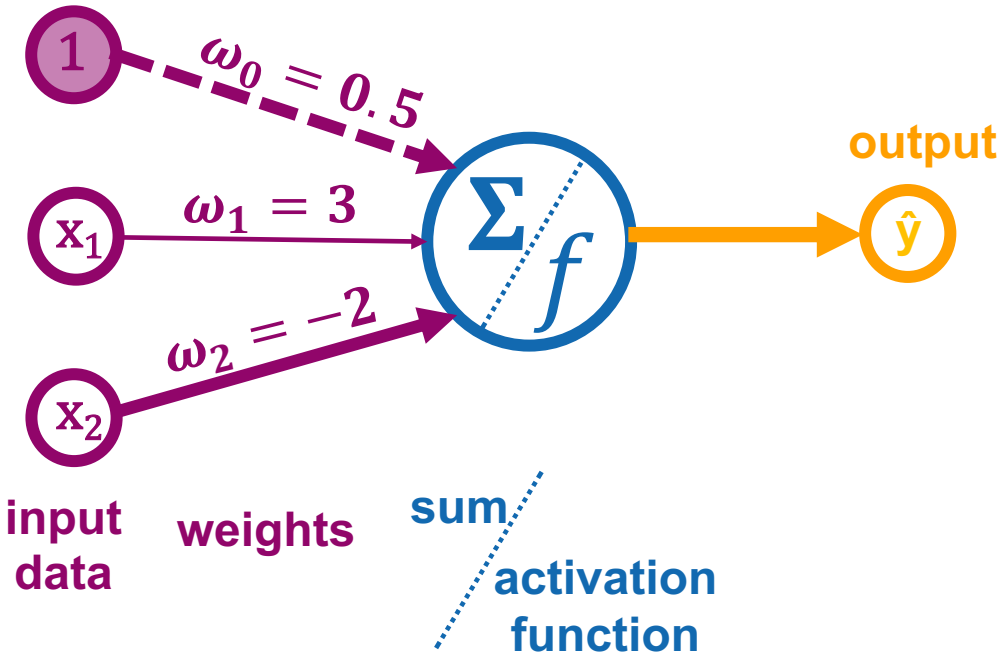
output ŷ, input data X, weights W

$$\hat{y} = f\left(X^T W\right)$$

$$\hat{y} = f\left(\begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 0.5 \\ 3 \\ -2 \end{bmatrix}\right)$$

$$\hat{y} = f\left(\underbrace{0.5 + 3x_1 - 2x_2}_{\text{this is a line in } R^2}\right)$$

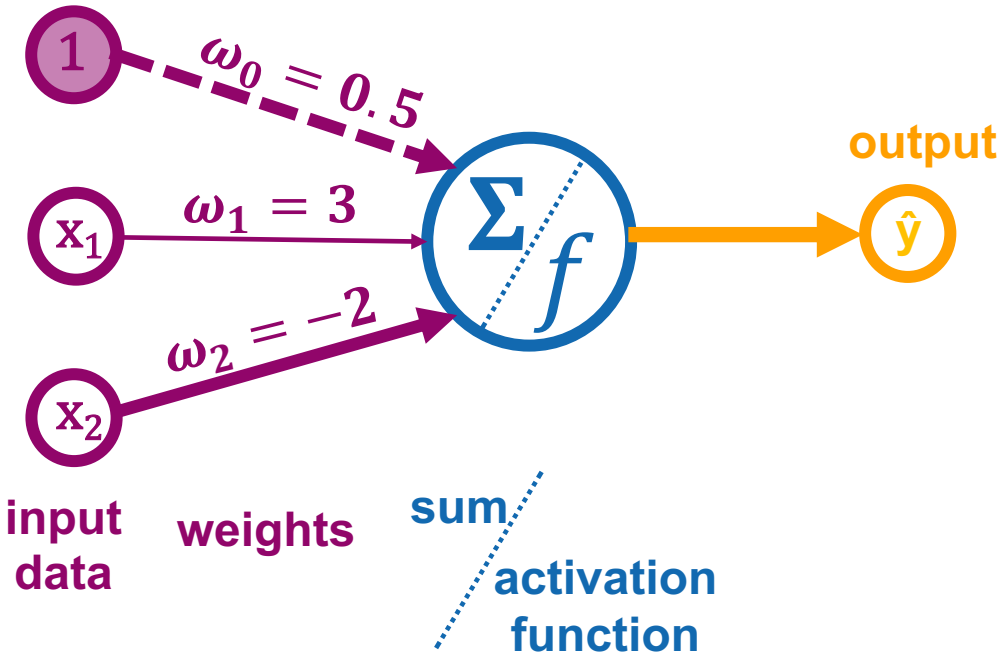
The Perceptron: Example



output input data x weights

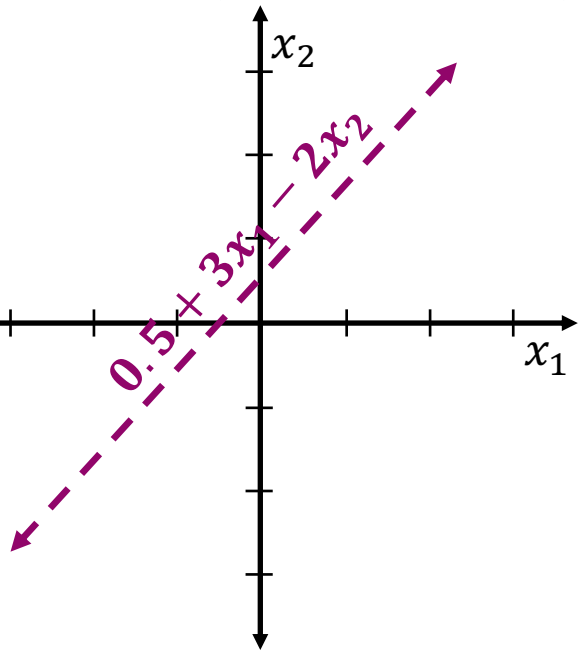
$$\hat{y} = f\left(0.5 + 3x_1 - 2x_2\right)$$

The Perceptron: Example

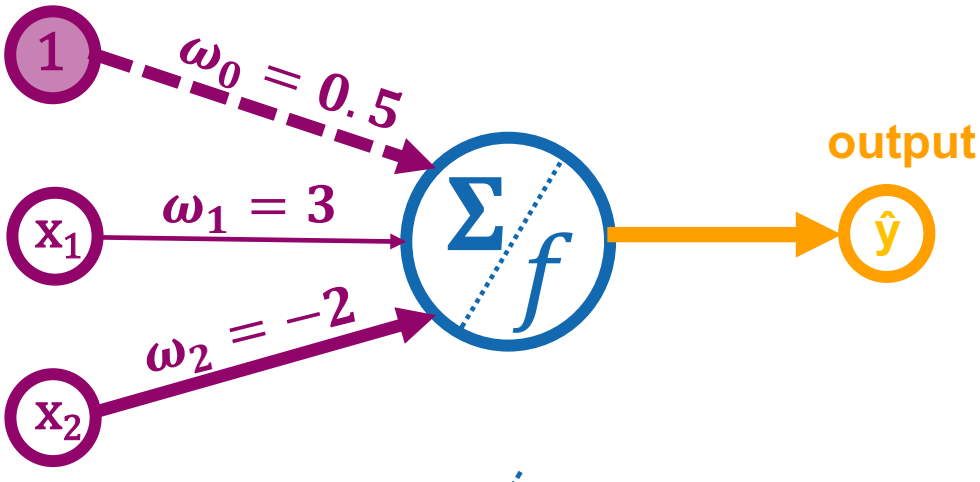


output input data x weights

$$\hat{y} = f\left(0.5 + 3x_1 - 2x_2\right)$$



The Perceptron: Example

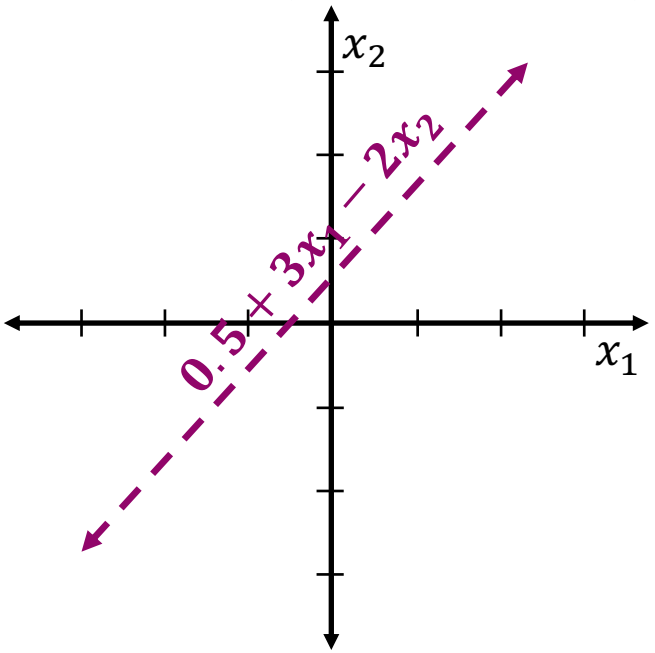


input data | weights | sum | activation function

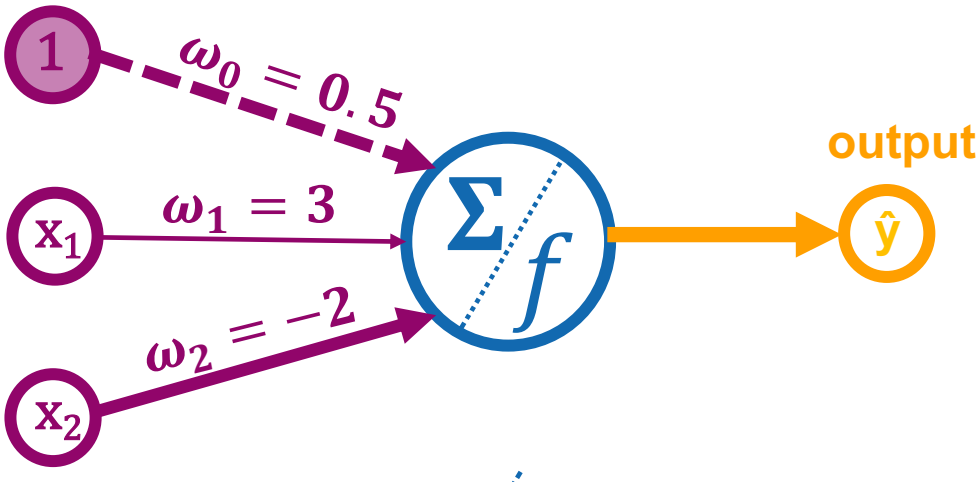
output | input data x weights

$$\hat{y} = f(0.5 + 3x_1 - 2x_2)$$

Given $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -2 \\ 3 \end{bmatrix} \in \boldsymbol{R^2}$



The Perceptron: Example

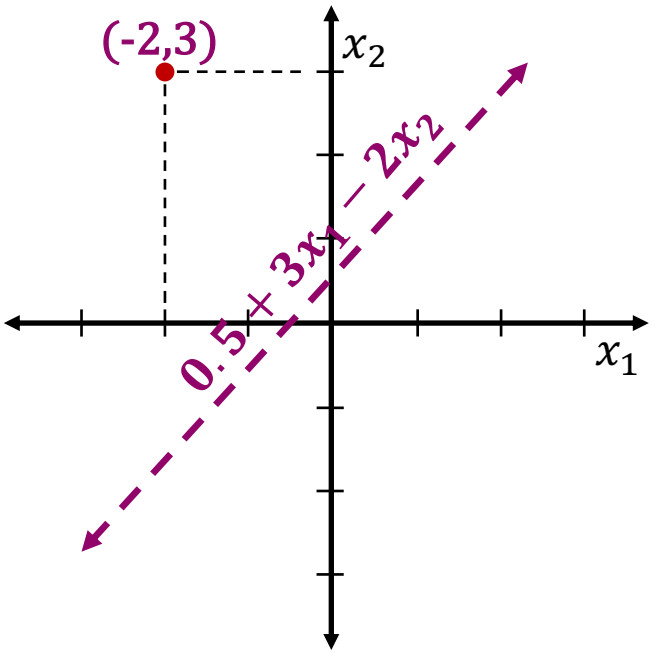


input data | weights | sum | activation function

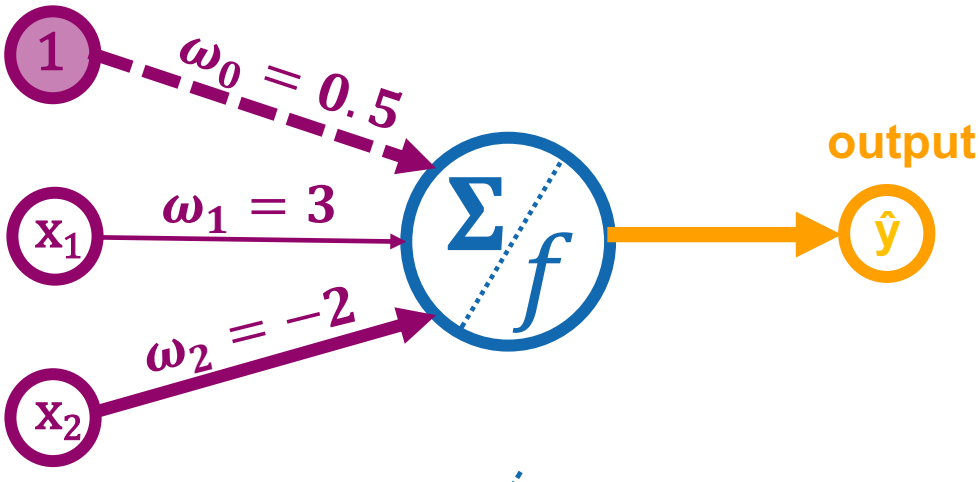
Given $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -2 \\ 3 \end{bmatrix} \in \boldsymbol{R^2}$

output | input data x weights

$$\hat{y} = f\left(0.5 + 3x_1 - 2x_2\right)$$



The Perceptron: Example



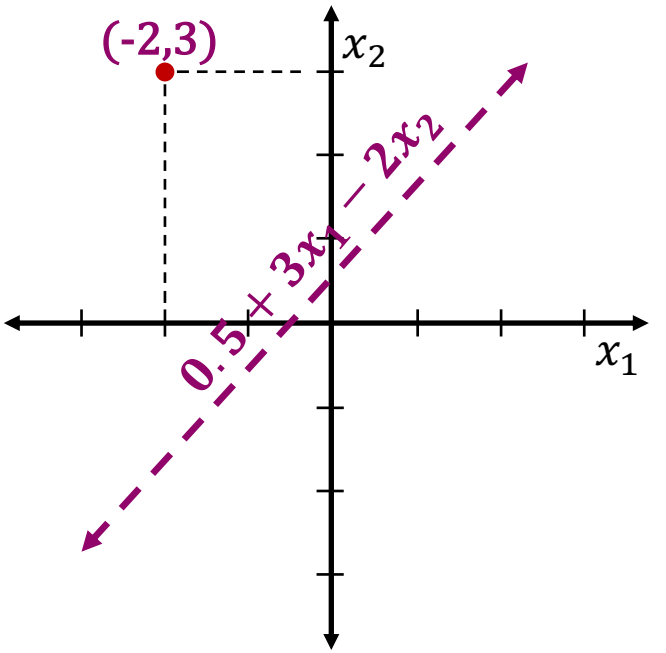
input data | weights | sum | activation function

$$\text{output} \quad \text{input data} \times \text{weights}$$

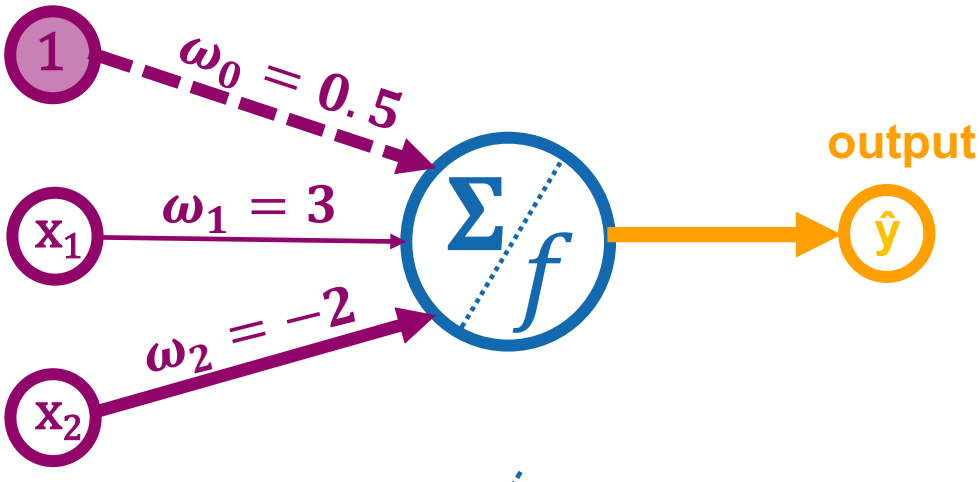
$$\hat{y} = f\left(0.5 + 3x_1 - 2x_2\right)$$

Given $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -2 \\ 3 \end{bmatrix} \in R^2$

$$\hat{y} = f\left(0.5 + 3(x_1) - 2(x_2)\right)$$



The Perceptron: Example



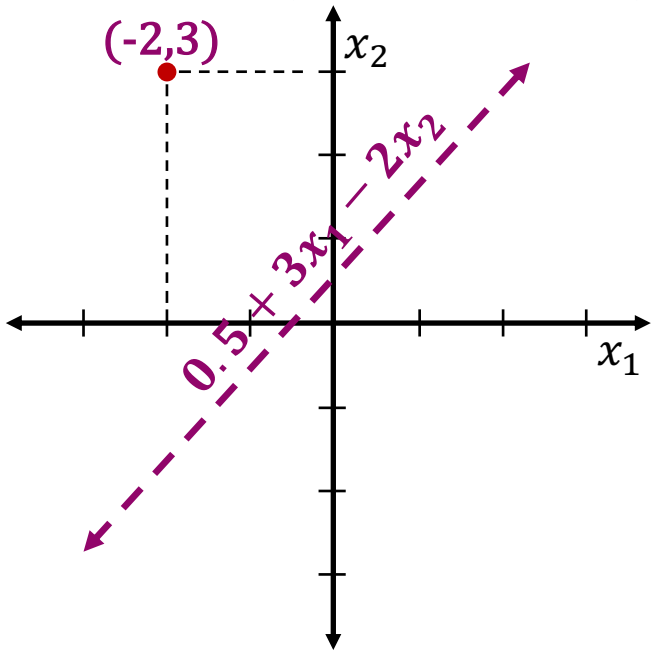
output input data x weights

$$\hat{y} = f\left(0.5 + 3x_1 - 2x_2\right)$$

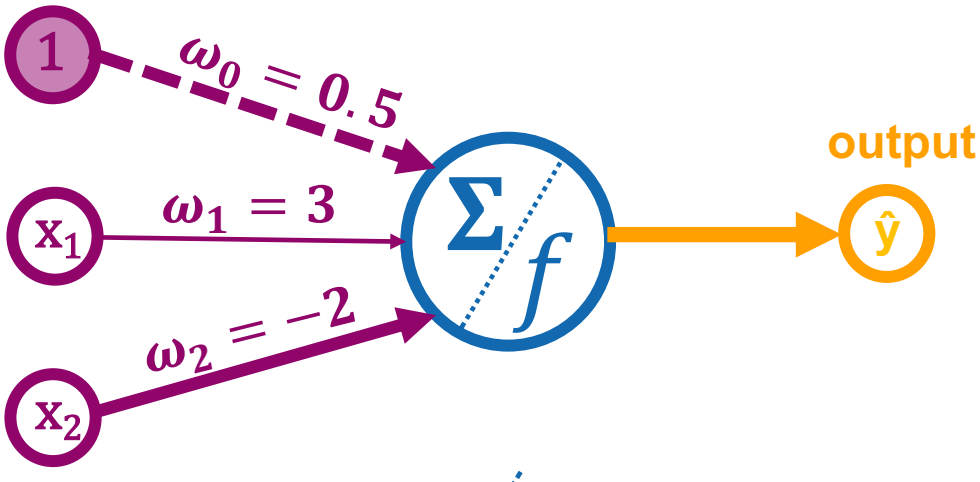
Given $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -2 \\ 3 \end{bmatrix} \in R^2$

$$\hat{y} = f\left(0.5 + 3(x_1) - 2(x_2)\right)$$

$$\hat{y} = f\left(0.5 + 3(-2) - 2(3)\right)$$



The Perceptron: Example



output input data x weights

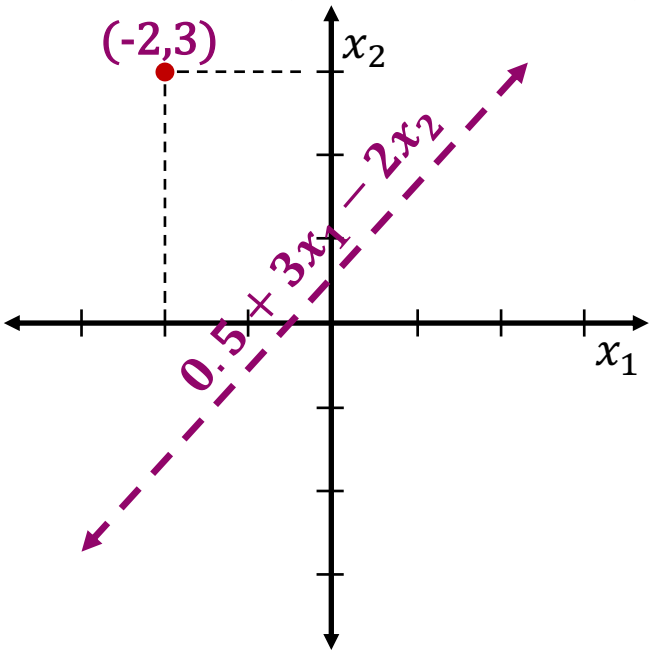
$$\hat{y} = f(0.5 + 3x_1 - 2x_2)$$

Given $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -2 \\ 3 \end{bmatrix} \in R^2$

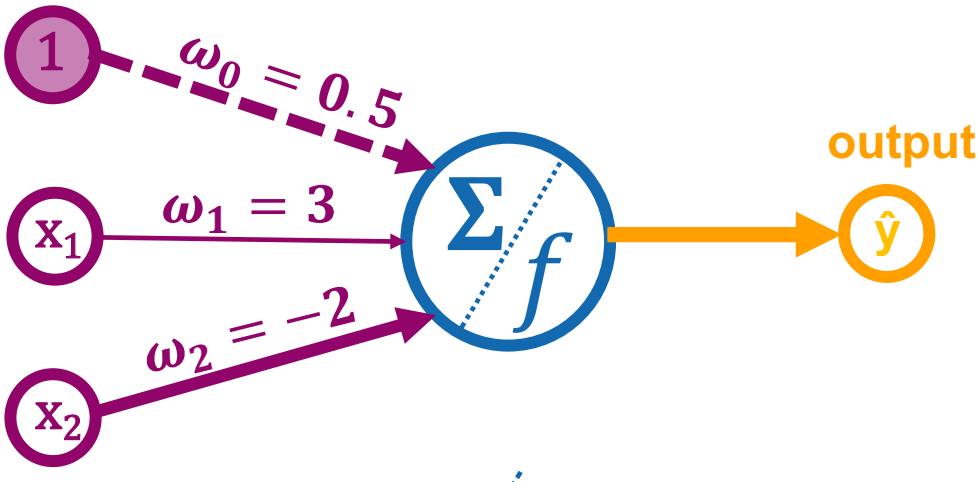
$$\hat{y} = f(0.5 + 3(x_1) - 2(x_2))$$

$$\hat{y} = f(0.5 + 3(-2) - 2(3))$$

$$\hat{y} = f(0.5 - 6 - 6) = f(-11.5)$$



The Perceptron: Example



output input data x weights

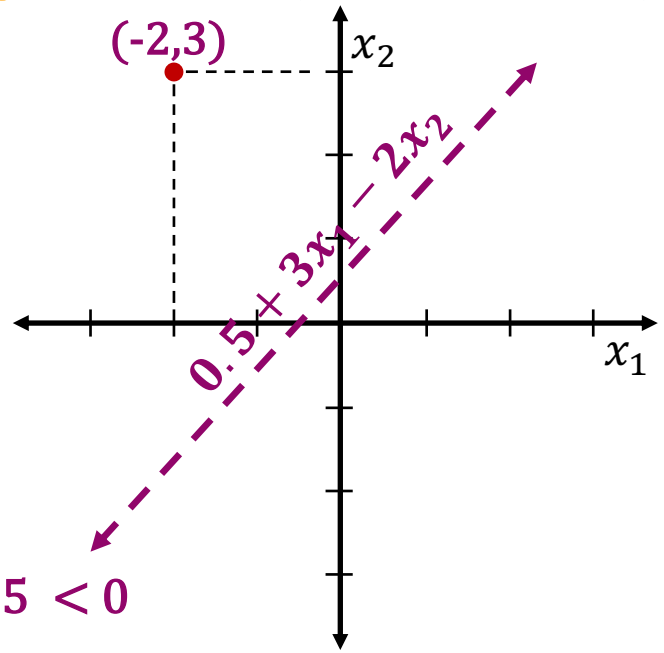
$$\hat{y} = f\left(0.5 + 3x_1 - 2x_2\right)$$

Given $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -2 \\ 3 \end{bmatrix} \in R^2$

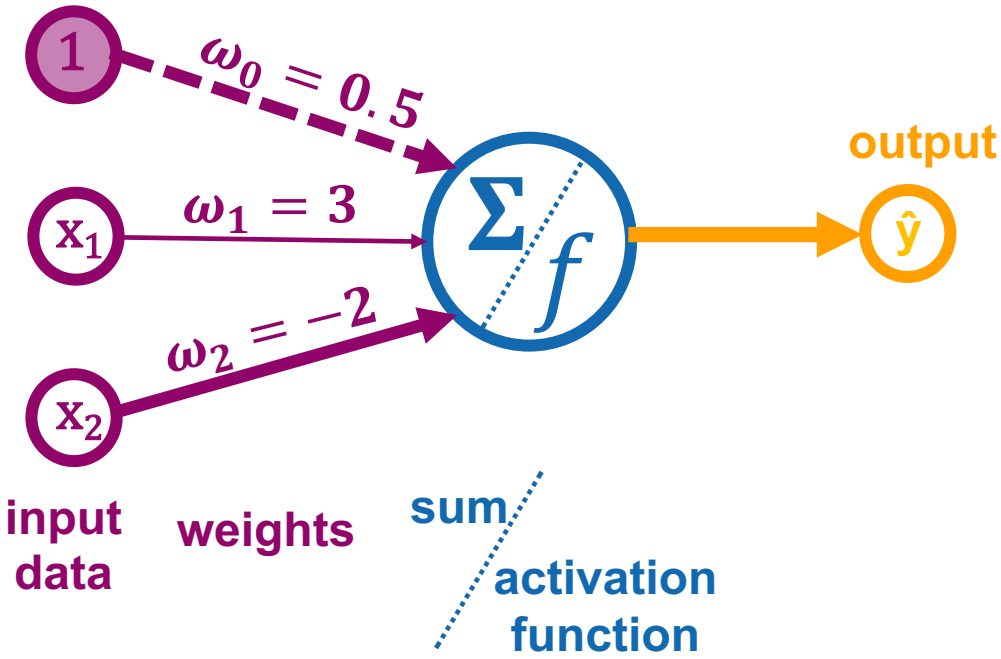
$$\hat{y} = f\left(0.5 + 3(x_1) - 2(x_2)\right)$$

$$\hat{y} = f\left(0.5 + 3(-2) - 2(3)\right)$$

$$\hat{y} = f\left(0.5 - 6 - 6\right) = f(-11.5), \quad -11.5 < 0$$



The Perceptron: Example



output input data x weights

$$\hat{y} = f\underbrace{(0.5 + 3x_1 - 2x_2)}_{z = -11.5}$$

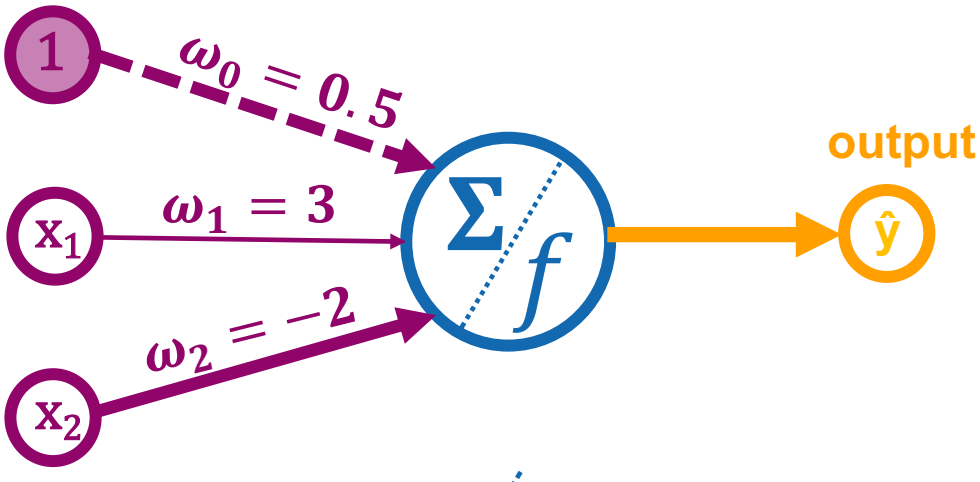
Given $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -2 \\ 3 \end{bmatrix} \in R^2$

$$\hat{y} = f(0.5 + 3(x_1) - 2(x_2))$$

$$\hat{y} = f(0.5 + 3(-2) - 2(3))$$

$$\hat{y} = f(0.5 - 6 - 6) = f(-11.5), \text{ -11.5} < 0$$

The Perceptron: Example



output ŷ = f ($0.5 + 3x_1 - 2x_2$)

input data x weights

$z = -11.5$

sign function

$$f(z) = \begin{cases} -1, if\ z < 0 \\ 0, if\ z = 0 \\ 1, if\ z > 0 \end{cases}$$

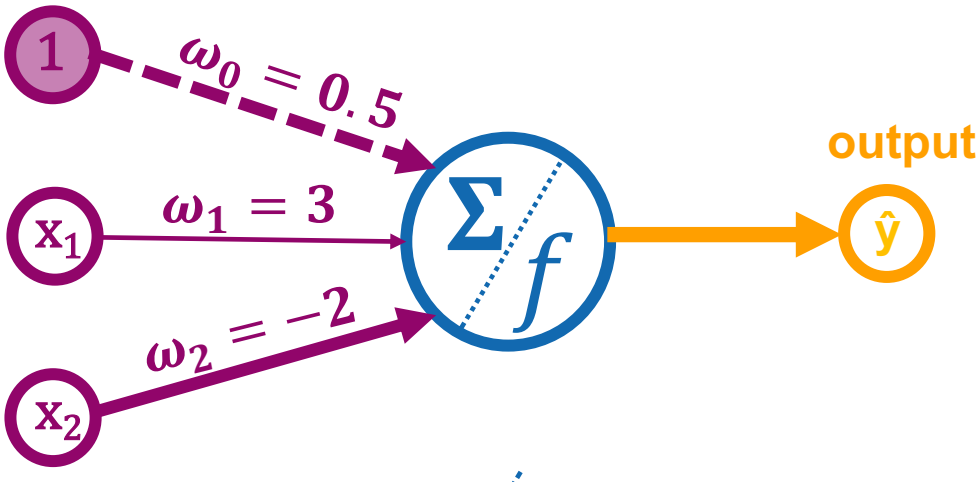
Given x = $\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -2 \\ 3 \end{bmatrix} \in R^2$

$$\hat{y} = f(0.5 + 3(x_1) - 2(x_2))$$

$$\hat{y} = f(0.5 + 3(-2) - 2(3))$$

$$\hat{y} = f(0.5 - 6 - 6) = f(-11.5), \text{ -11.5 } < 0$$

The Perceptron: Example



input data | weights | sum | activation function

output | input data x weights

$$\hat{y} = f\underbrace{(0.5 + 3x_1 - 2x_2)}_{z = -11.5}$$

sign function

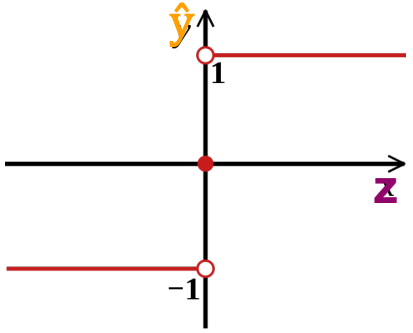
$$f(z) = \begin{cases} -1, if\ z < 0 \\ 0, if\ z = 0 \\ 1, if\ z > 0 \end{cases}$$

Given $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -2 \\ 3 \end{bmatrix} \in R^2$

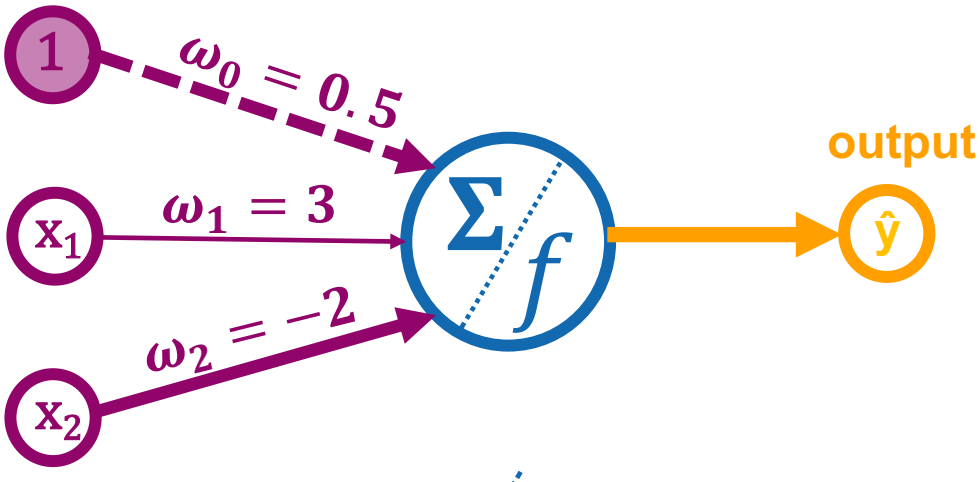
$$\hat{y} = f(0.5 + 3(x_1) - 2(x_2))$$

$$\hat{y} = f(0.5 + 3(-2) - 2(3))$$

$$\hat{y} = f(0.5 - 6 - 6) = f(-11.5), \text{ -11.5} < 0$$

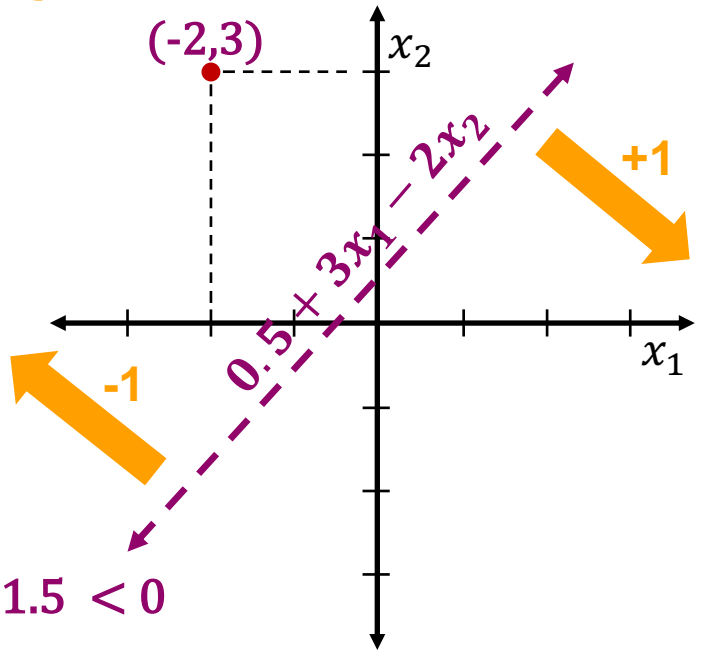


The Perceptron: Example



output input data x weights

$$\hat{y} = f\left(0.5 + 3x_1 - 2x_2\right)$$



Given $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -2 \\ 3 \end{bmatrix} \in R^2$

$$\hat{y} = f\left(0.5 + 3(x_1) - 2(x_2)\right)$$

$$\hat{y} = f\left(0.5 + 3(-2) - 2(3)\right)$$

$$\hat{y} = f\left(0.5 - 6 - 6\right) = f(-11.5), \quad -11.5 < 0$$

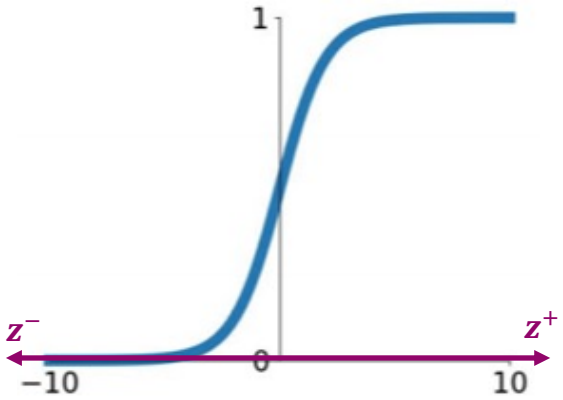
Building Deep Learning Models: The Perceptron

Building Deep Learning Models: The Perceptron Activation Functions

Activation Functions

Sigmoid

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



output — input data X weights

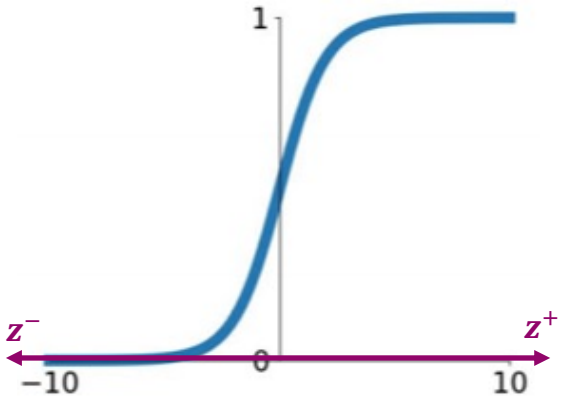
$$\hat{y} = \sigma\underbrace{(0.5 + 3x_1 - 2x_2)}_{z = -11.5}$$

$$\hat{y} = \sigma(-11.5) \approx 0$$

Activation Functions

Sigmoid

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

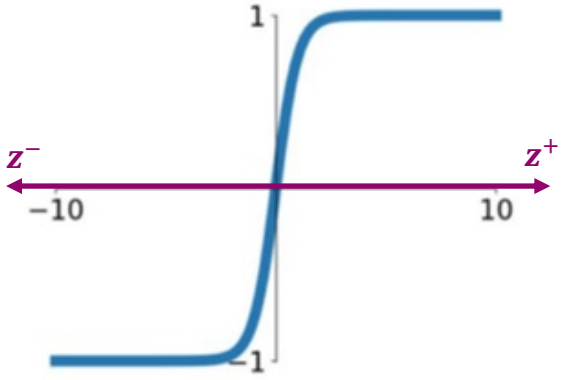


output $\hat{y} = \sigma(\underbrace{0.5 + 3x_1 - 2x_2}_{z = -11.5})$

$\hat{y} = \sigma(-11.5) \approx 0$

tanh

$$\tanh(x)$$



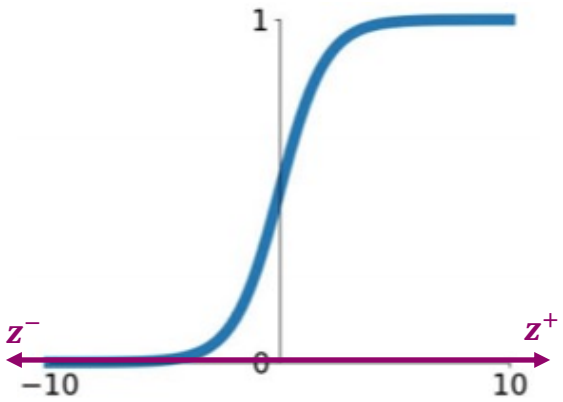
output $\hat{y} = \tau(\underbrace{0.5 + 3x_1 - 2x_2}_{z = -11.5})$

$\hat{y} = \tau(-11.5) \approx -1$

Activation Functions

Sigmoid

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

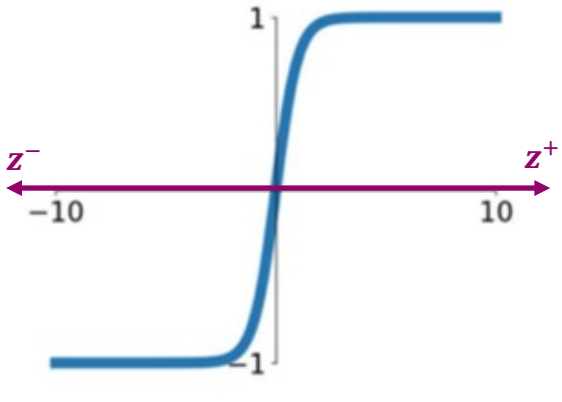


output $\hat{y} = \sigma(\underbrace{0.5 + 3x_1 - 2x_2}_{z = -11.5})$

$\hat{y} = \sigma(-11.5) \approx 0$

tanh

$$\tanh(x)$$

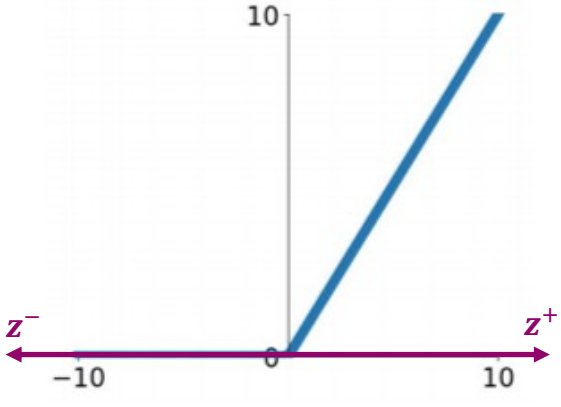


output $\hat{y} = \tau(\underbrace{0.5 + 3x_1 - 2x_2}_{z = -11.5})$

$\hat{y} = \tau(-11.5) \approx -1$

ReLU

$$\max(0, x)$$



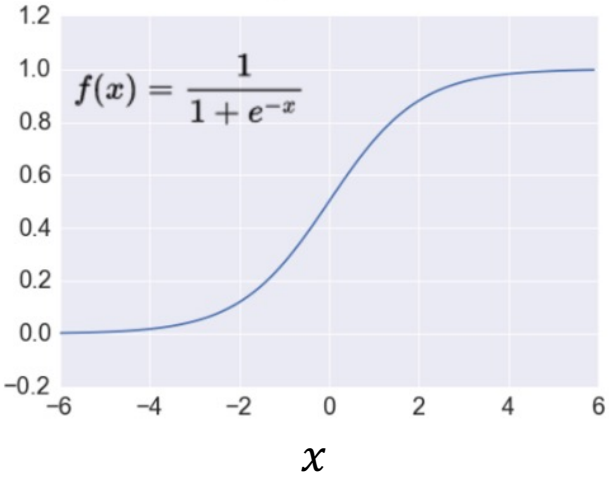
output $\hat{y} = m(\underbrace{0.5 + 3x_1 - 2x_2}_{z = -11.5})$

$\hat{y} = m(-11.5) = 0$

Activation Functions

- ***Sigmoid function*** σ: takes a real-valued number and "squashes" it into the range between 0 and 1
  - The output can be interpreted as the firing rate of a biological neuron
    - Not firing = 0; Fully firing = 1
  - When the neuron's activation are 0 or 1, sigmoid neurons saturate
    - Gradients at these regions are almost zero (almost no signal will flow)
  - Sigmoid activations are less common in modern NNs

$f(x)$

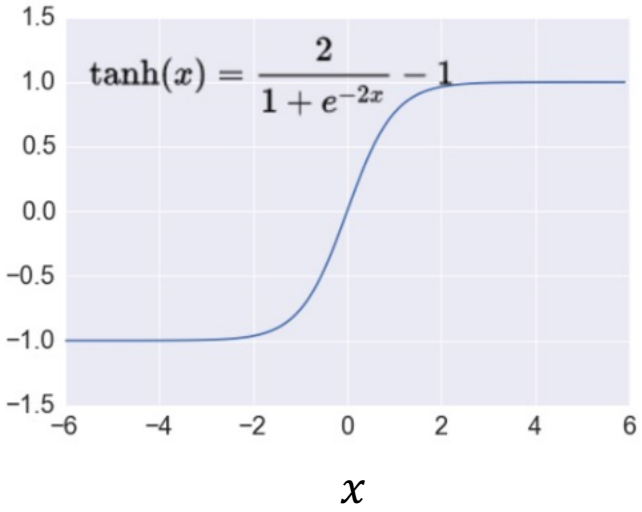


$\mathbb{R}^n \rightarrow [0,1]$

Activation Functions

- ***Tanh function***: takes a real-valued number and "squashes" it into range between -1 and 1
  - Like sigmoid, tanh neurons saturate
  - Unlike sigmoid, the output is zero-centered
    - It is therefore preferred than sigmoid
  - Tanh is a scaled sigmoid: $\tanh(x) = 2 \cdot \sigma(2x) - 1$

$f(x)$



$\mathbb{R}^n \rightarrow [-1,1]$

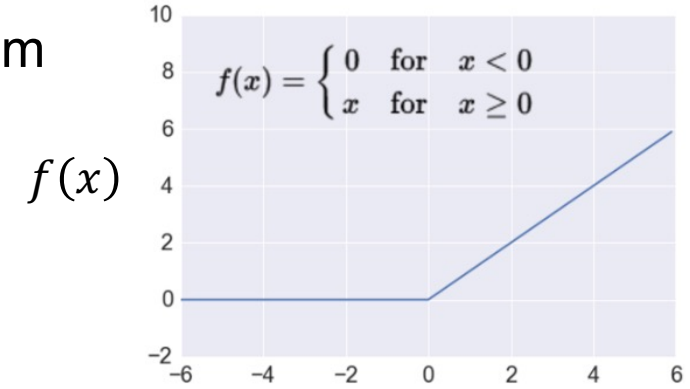
Activation Functions

- ***ReLU*** (Rectified Linear Unit): takes a real-valued number and thresholds it at zero

$$f(x) = \max(0, x)$$

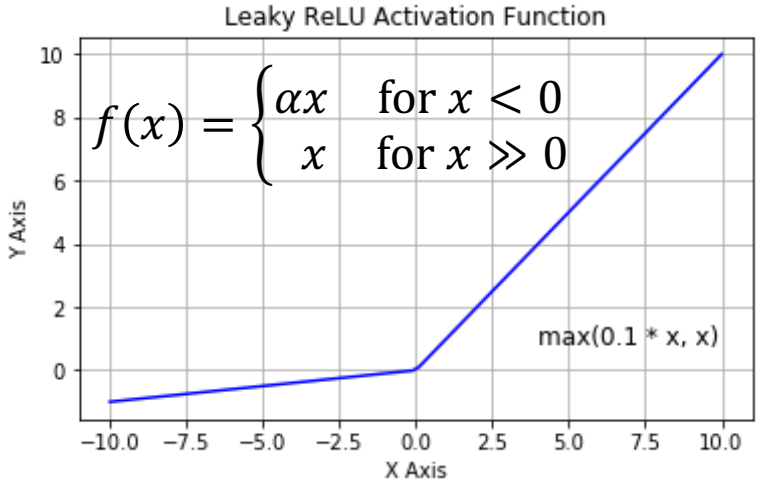
  - Most modern deep NNs use ReLU activations
  - ReLU is fast to compute
    - Compared to sigmoid, tanh
    - Simply threshold a matrix at zero
  - Accelerates the convergence of gradient descent
    - Due to linear, non-saturating form
  - Prevents the gradient vanishing problem

$\mathbb{R}^n \rightarrow \mathbb{R}^n_+$



Activation Functions

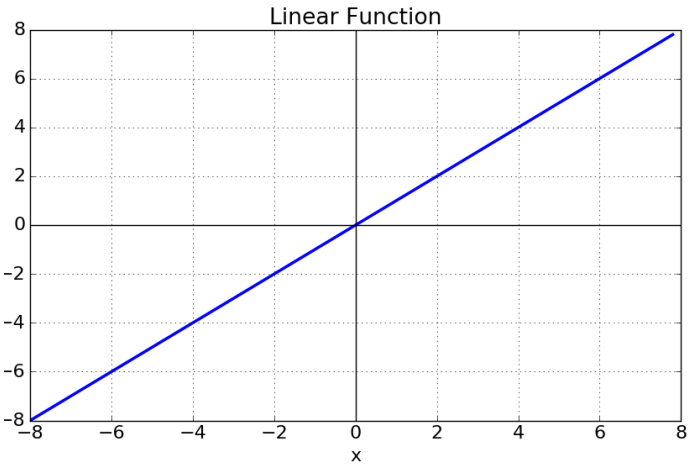
- ***Leaky ReLU*** activation function is a variant of ReLU
  - Instead of the function being 0 when $x < 0$, a leaky ReLU has a small negative slope (e.g., α = 0.01, or similar)
  - This resolves the dying ReLU problem
  - Most current works still use ReLU
    - With a proper setting of the learning rate, the problem of dying ReLU can be avoided



Activation Functions

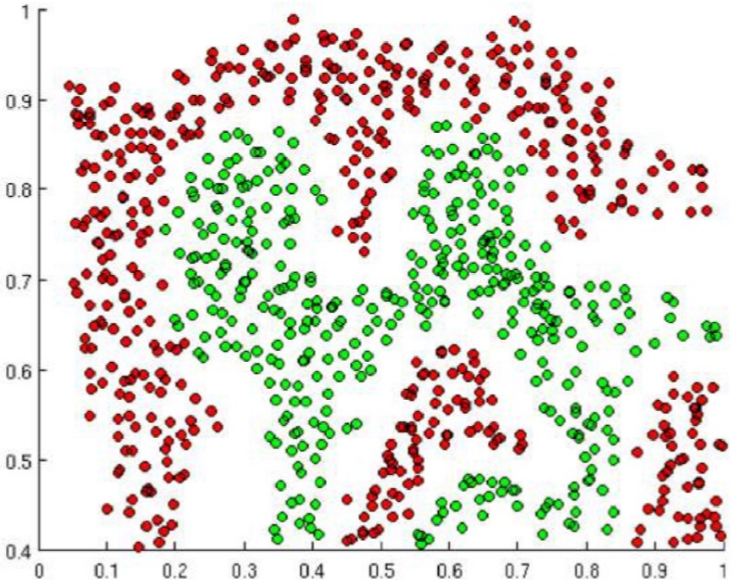
- ***Linear function*** means that the output signal is proportional to the input signal to the neuron
  - If the value of the constant *c* is 1, it is also called identity activation function
  - This activation type is used in regression problems
    - E.g., the last layer can have linear activation function, in order to output a real number (and not a class membership)

$$\mathbb{R}^n \rightarrow \mathbb{R}^n$$



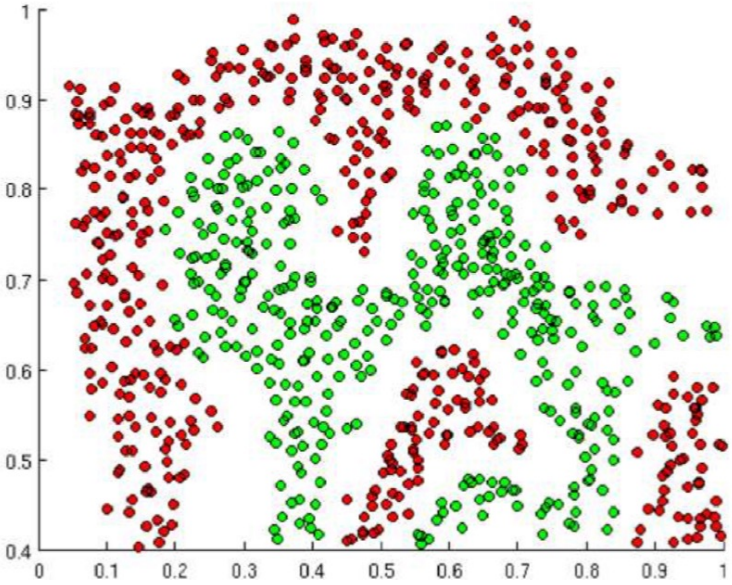
Importance of Activation Functions

How to distinguish between **red** and **green** data samples?



Importance of Activation Functions

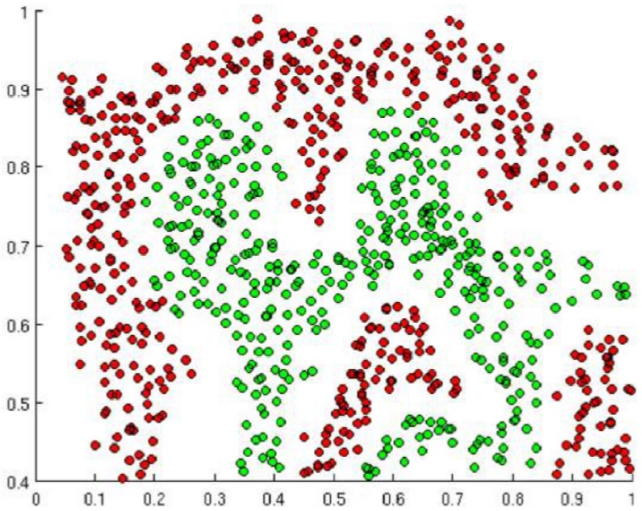
How to distinguish between **red** and **green** data samples?



The activation functions introduce **non-linearities** into the network

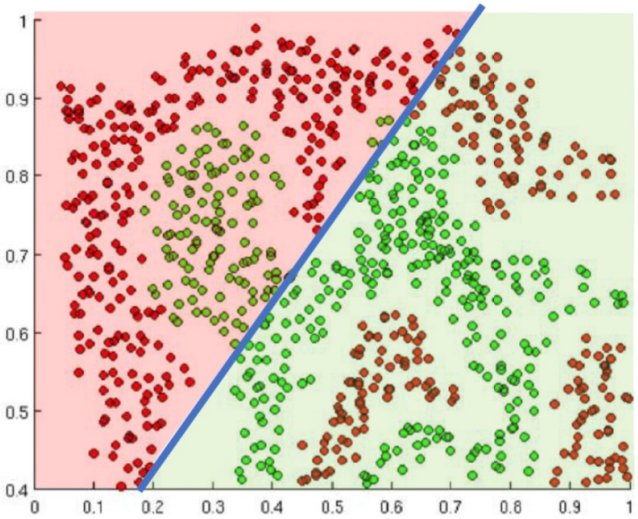
Importance of Activation Functions

input data



Importance of Activation Functions

input data

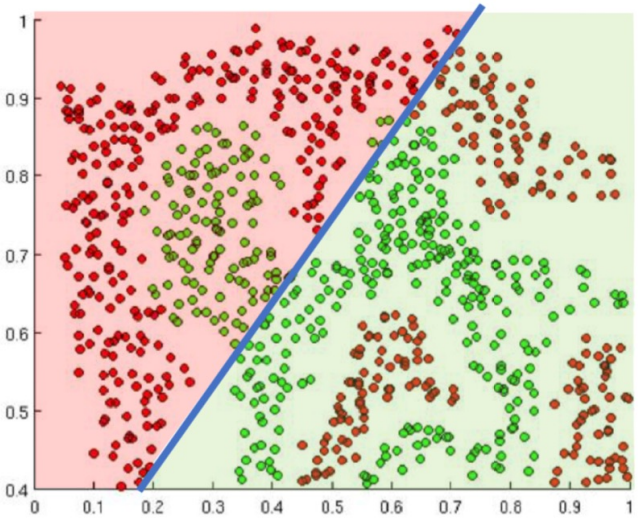


$$\hat{y} = f(z) = z$$

linear function: produces linear hyperplanes

Importance of Activation Functions

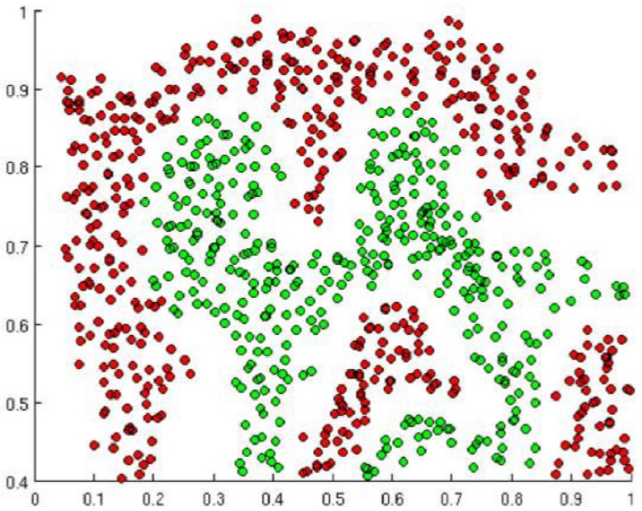
input data



$$\hat{y} = f(z) = z$$

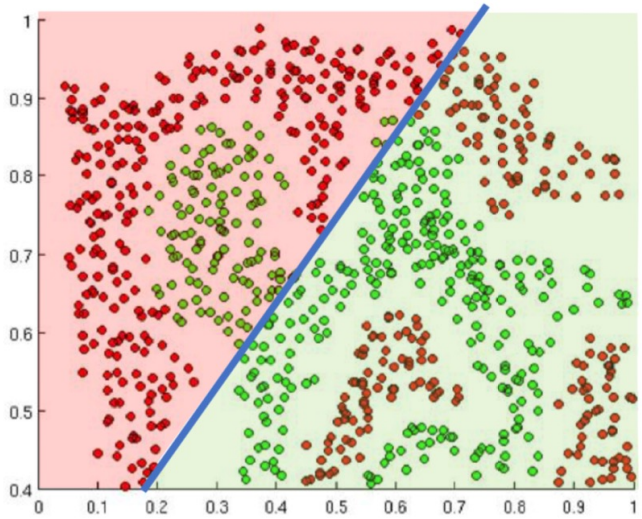
linear function: produces linear hyperplanes

input data



Importance of Activation Functions

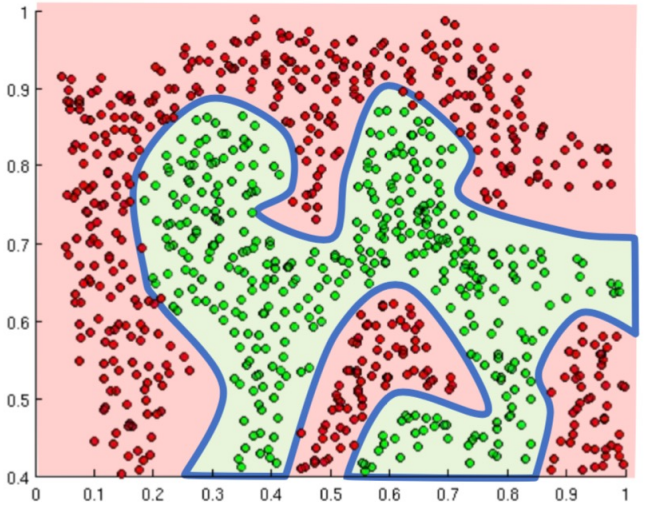
input data



$$\hat{y} = f(z) = z$$

linear function: produces linear hyperplanes

input data

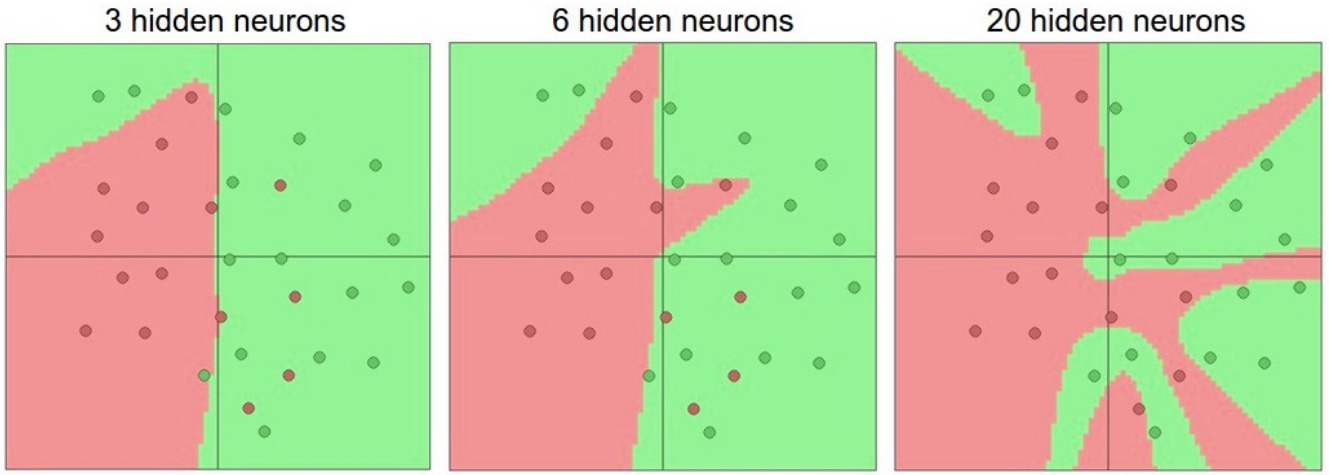


$$\hat{y} = \sigma(z) = z$$

non-linear function: produces **non-linear** hyperplanes

Importance of Activation Functions

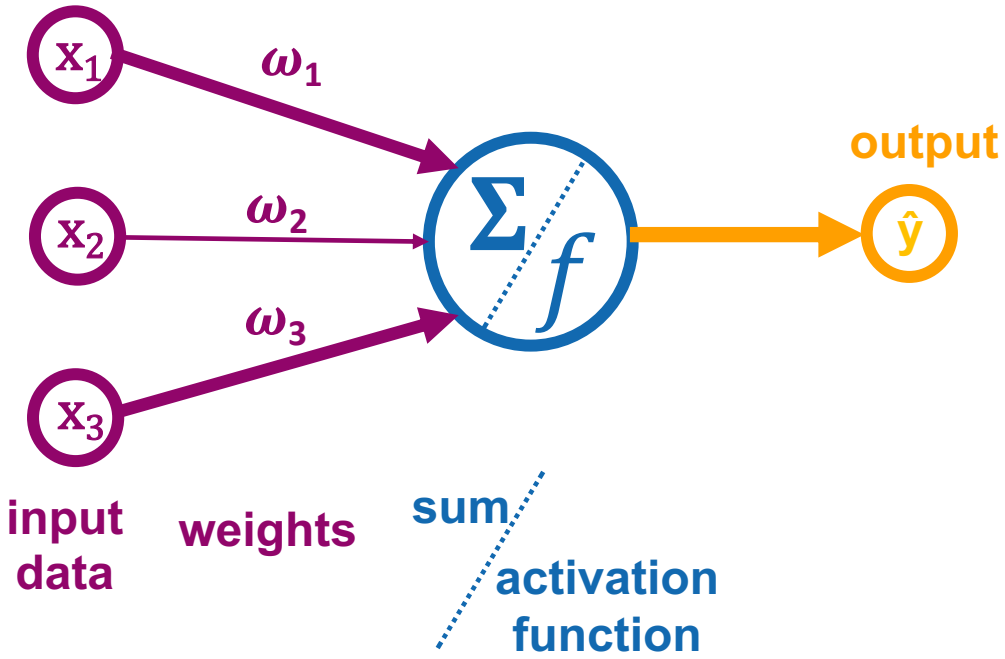
- Non-linear activations are needed to learn complex (non-linear) data representations
  - Otherwise, NNs would be just a linear function (such as $W_1 W_2 x = Wx$)
  - NNs with large number of layers (and neurons) can approximate more complex functions
    - Figure: more neurons improve representation (but, may overfit)



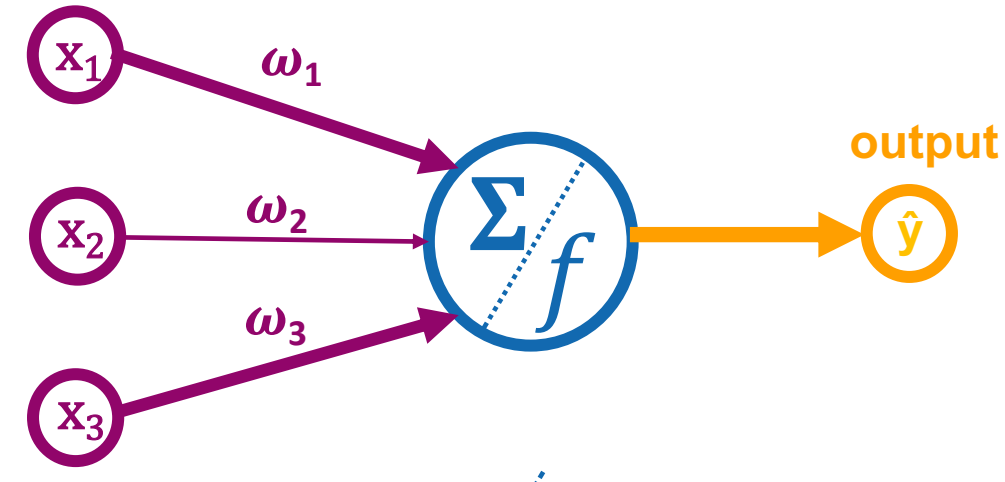
Building Deep Learning Models: The Perceptron Activation Functions

Building Deep Learning Models:
The Perceptron
Activation Functions
Multilayer Perceptron

Neural Networks: Architectures



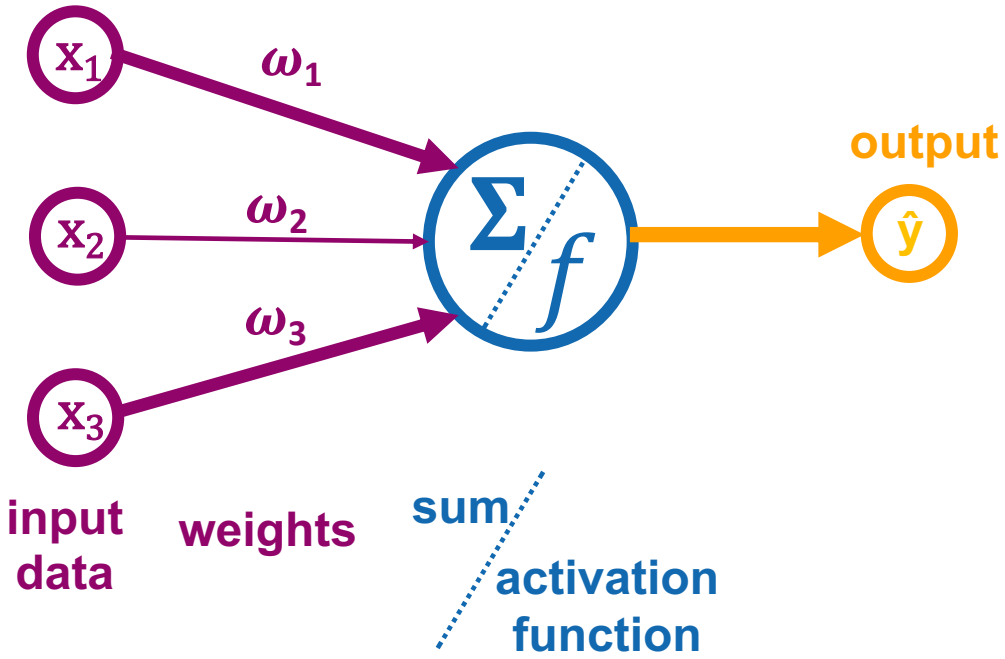
Neural Networks: Architectures



$$\hat{y} = f\left(\sum_{i=0} x_i \omega_i\right)$$

output: $\hat{y}$ — input data: x_i — weights: ω_i

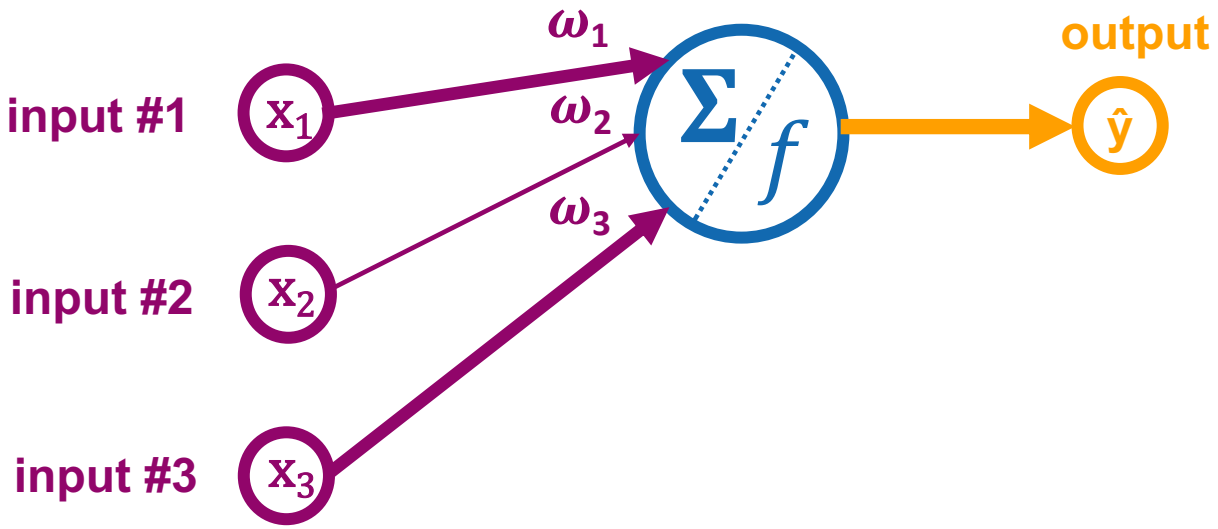
Neural Networks: Architectures



$$\hat{y} = f\left(\sum_{i=0} x_i \omega_i\right)$$

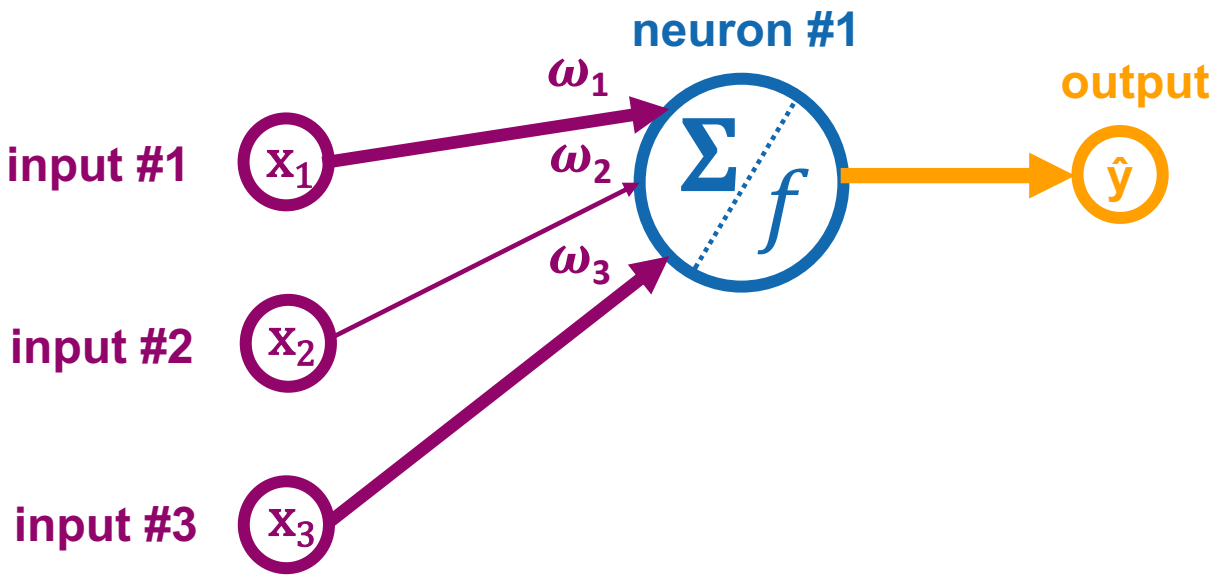
$$\hat{y} = f\left(X^T W\right)$$

Neural Networks: Architectures



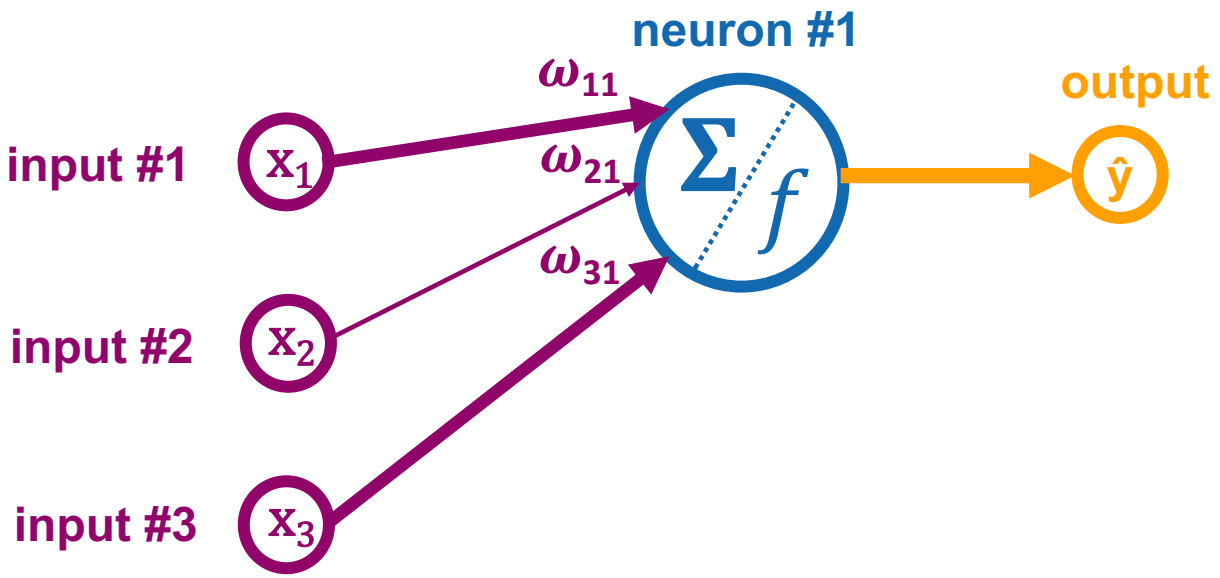
$$\hat{y} = f\left(\sum_{i=0} x_i \omega_i\right)$$

Neural Networks: Architectures



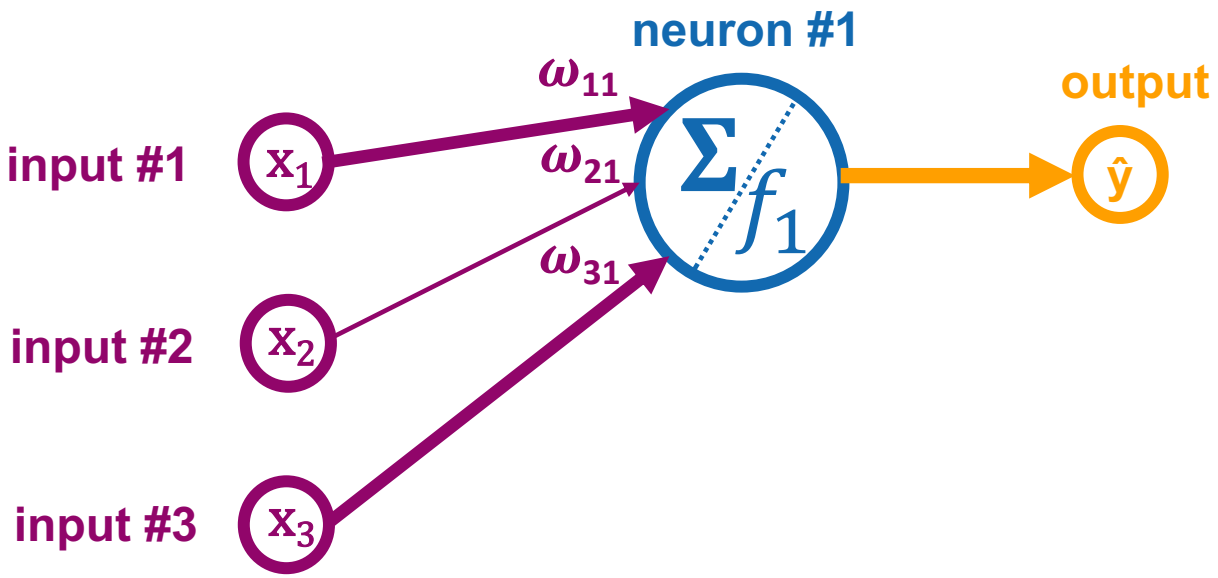
$$\hat{y} = f\left(\sum_{i=0} x_i \omega_i\right)$$

Neural Networks: Architectures



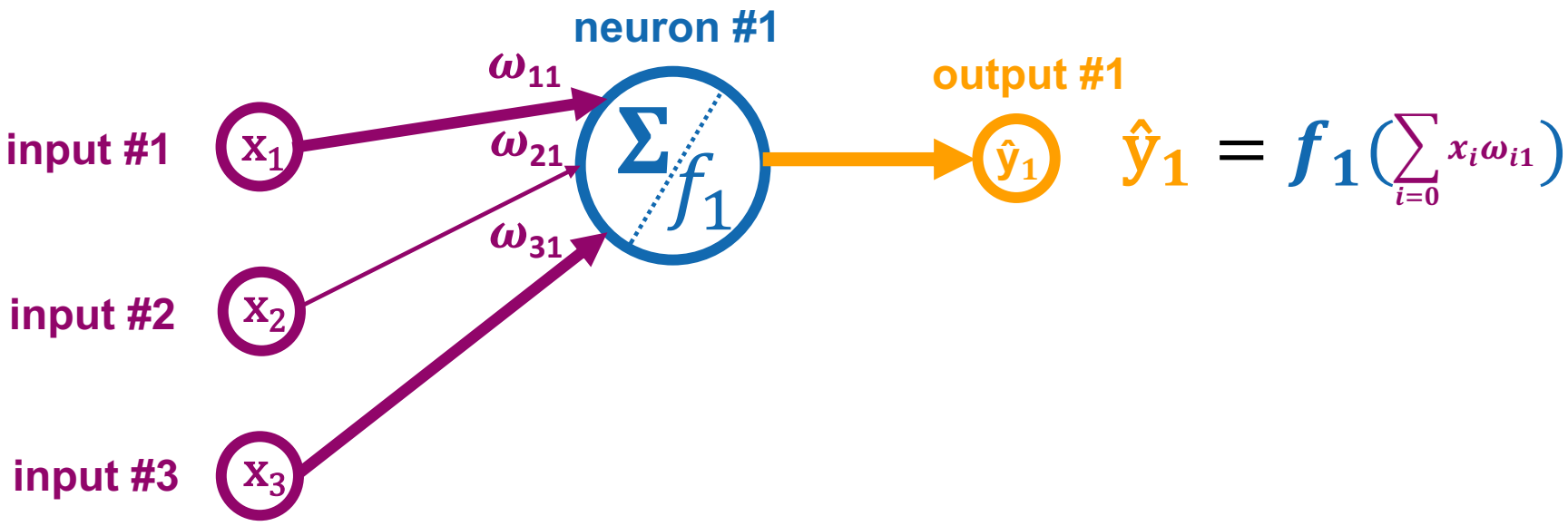
$$\hat{y} = f\left(\sum_{i=0} x_i \omega_{i1}\right)$$

Neural Networks: Architectures

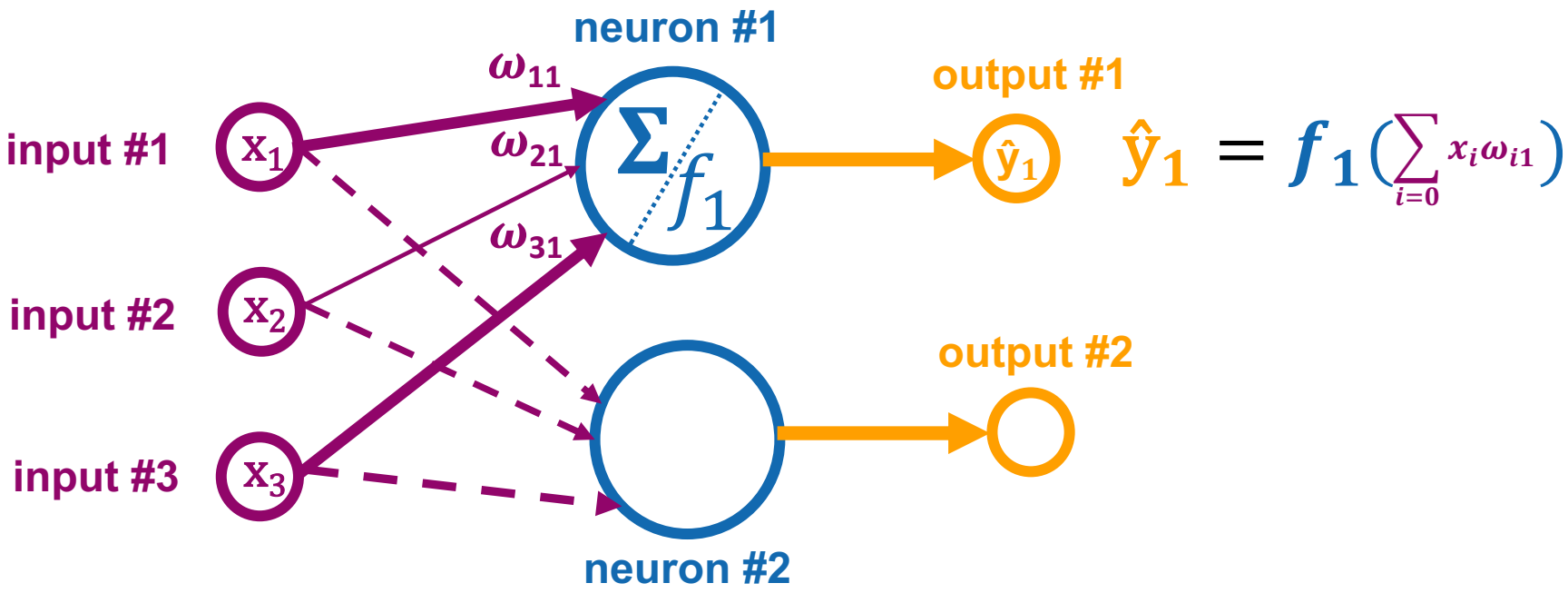


$$\hat{y} = f_1\left(\sum_{i=0} x_i \omega_{i1}\right)$$

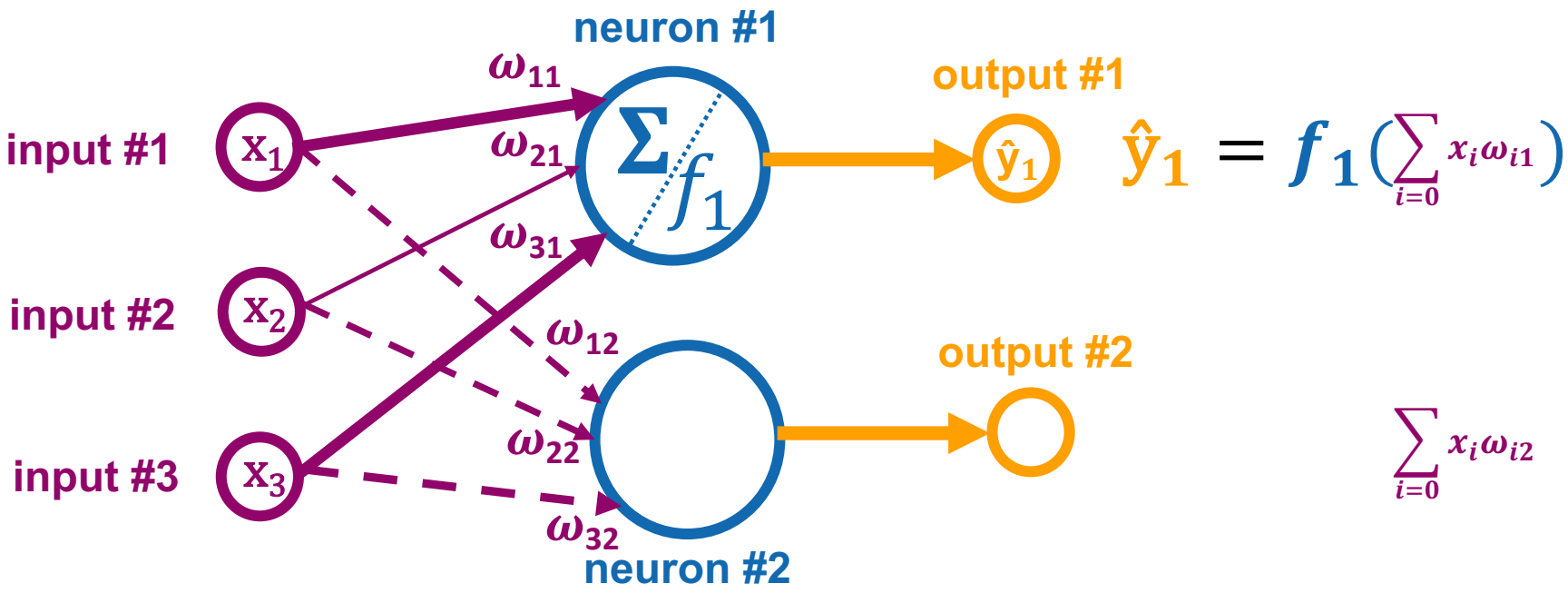
Neural Networks: Architectures



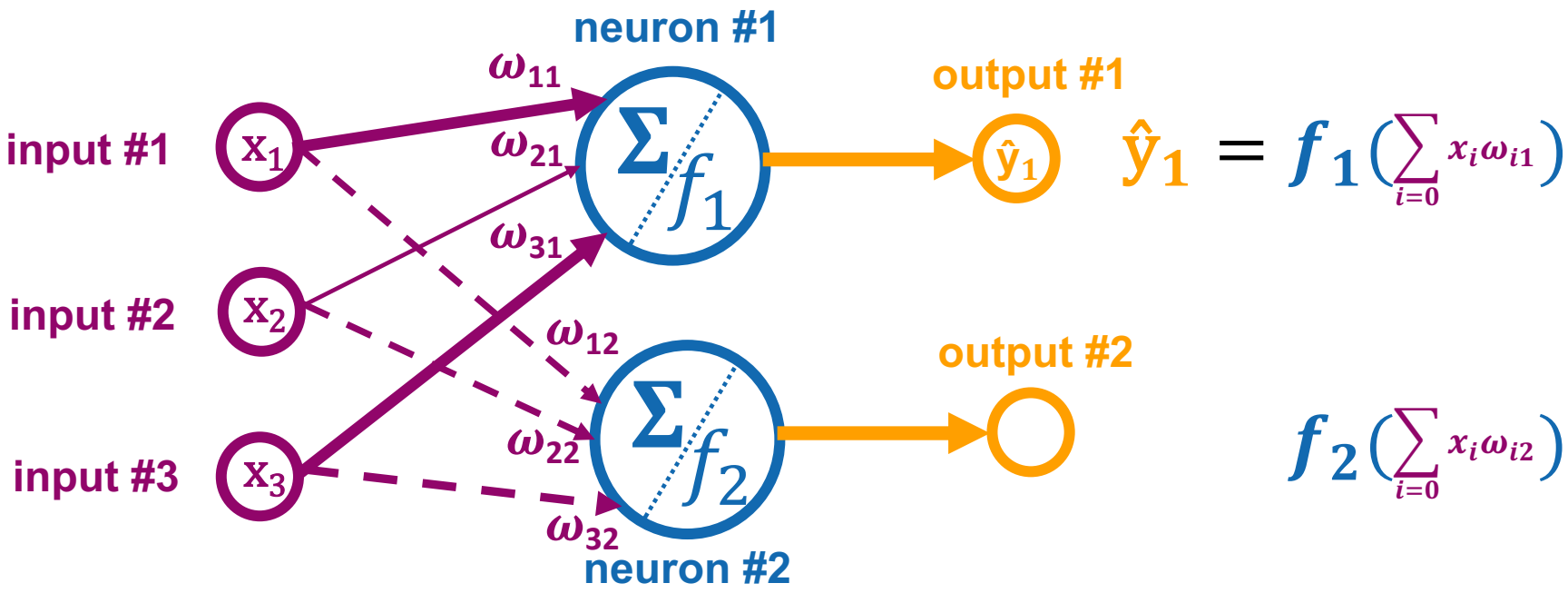
Neural Networks: Architectures



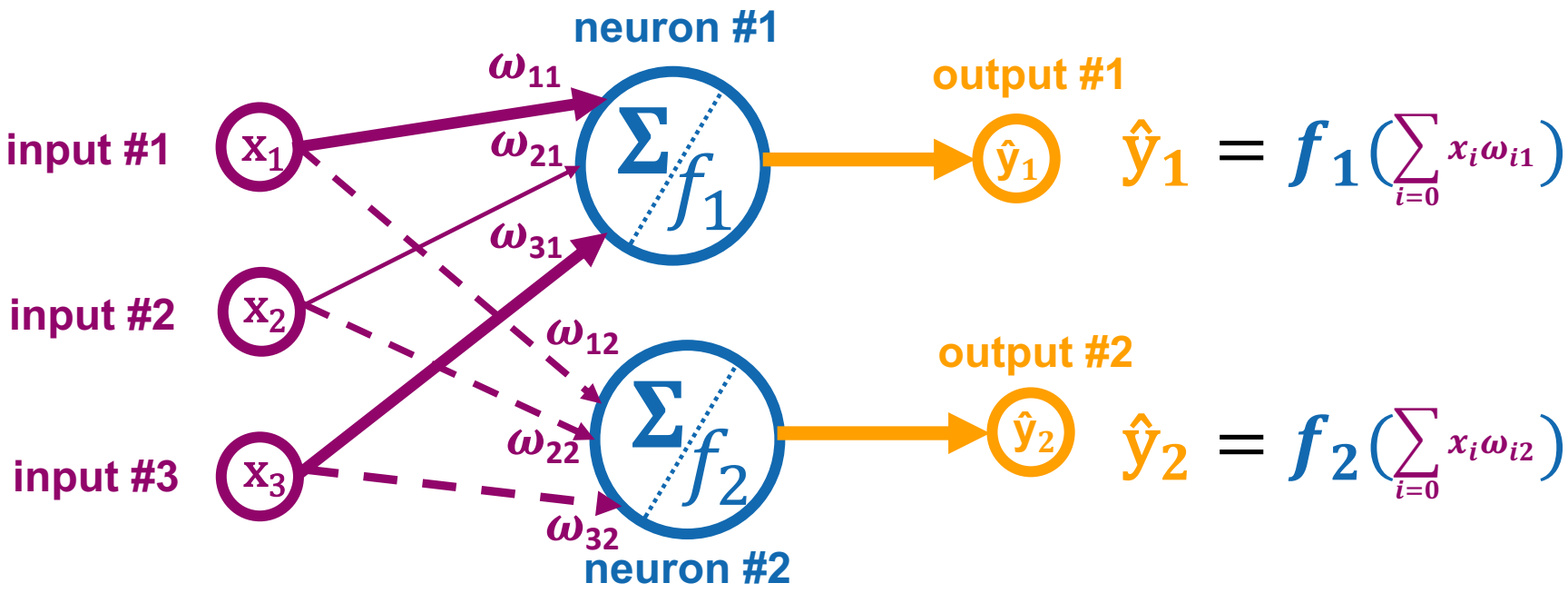
Neural Networks: Architectures



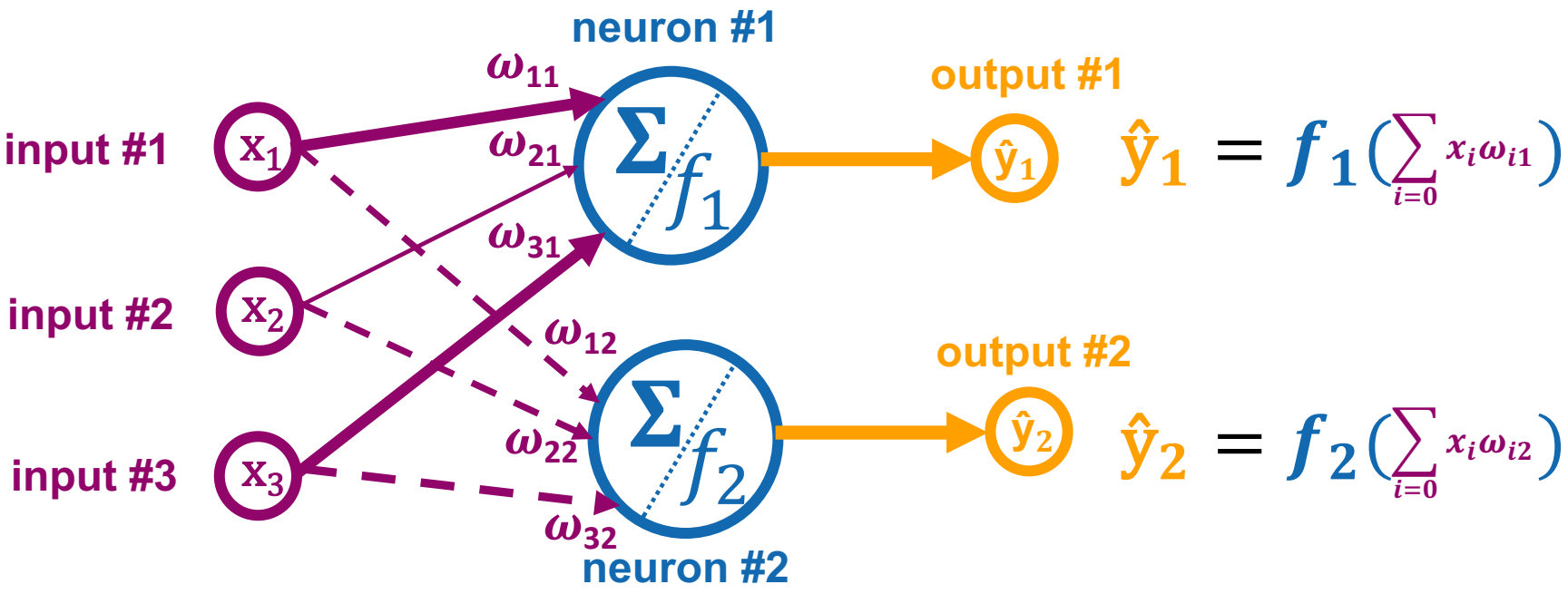
Neural Networks: Architectures



Neural Networks: Architectures

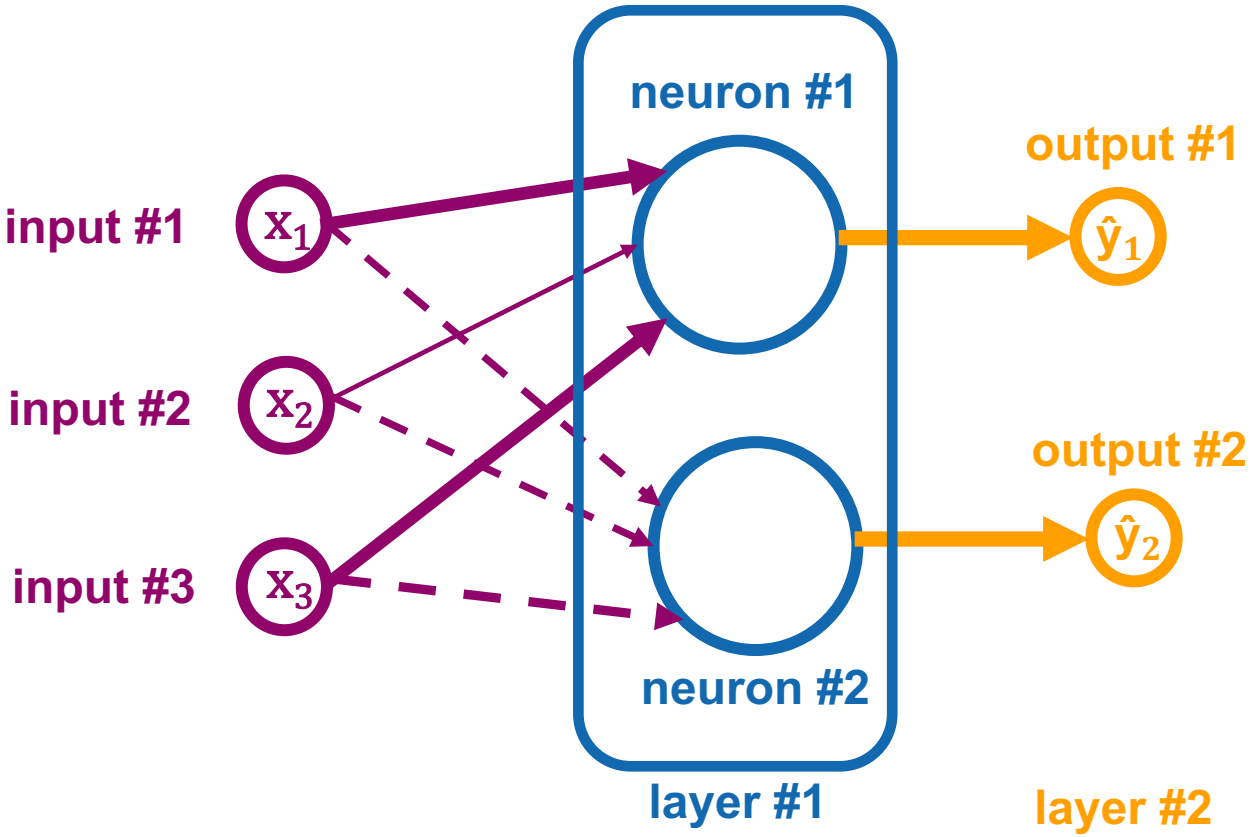


Neural Networks: Architectures



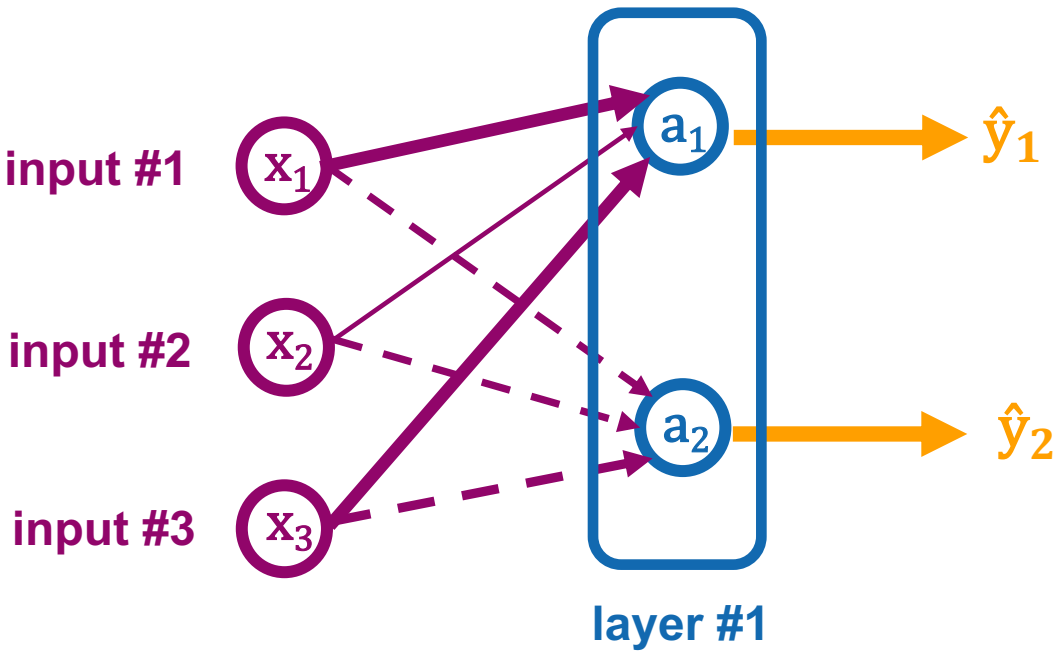
$$\hat{y}_j = f_j\left(\sum_{i=0} x_i \omega_{ij}\right)$$

Neural Networks: Architectures



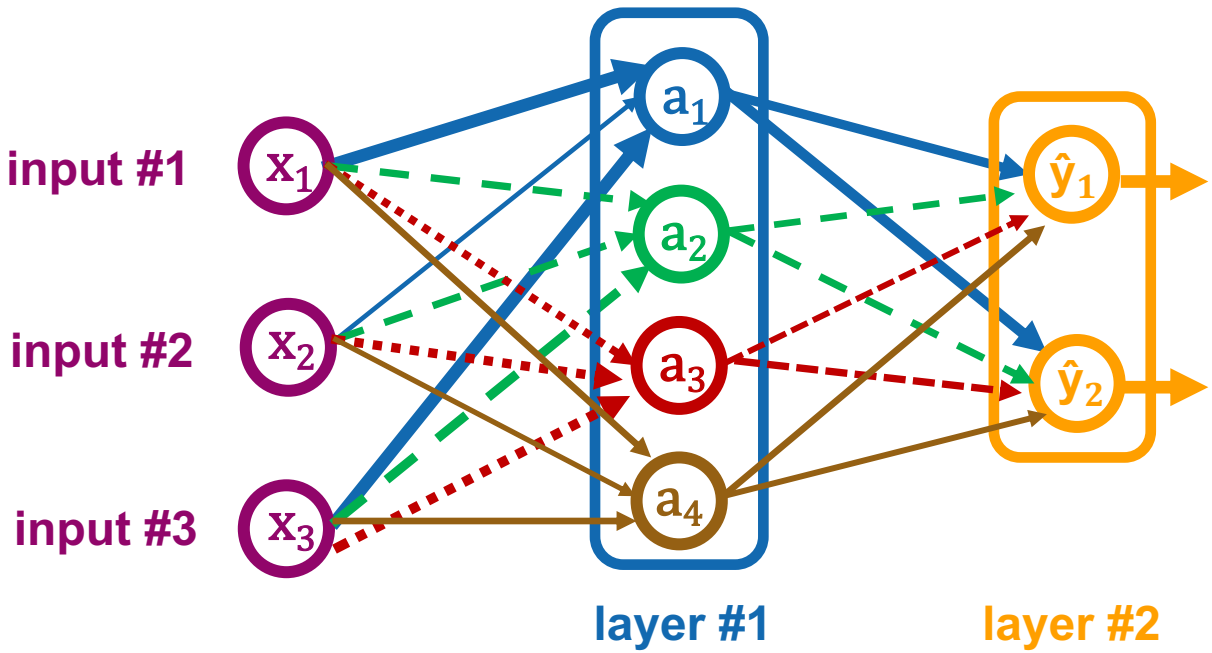
$$\hat{y}_j = f_j\left(\sum_{i=0} x_i \omega_{ij}\right)$$

Neural Networks: Architectures



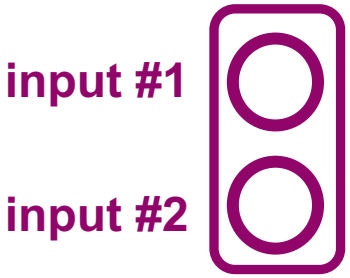
$$\hat{y}_j = f_j\left(\sum_{i=0} x_i \omega_{ij}\right)$$

Neural Networks: Architectures



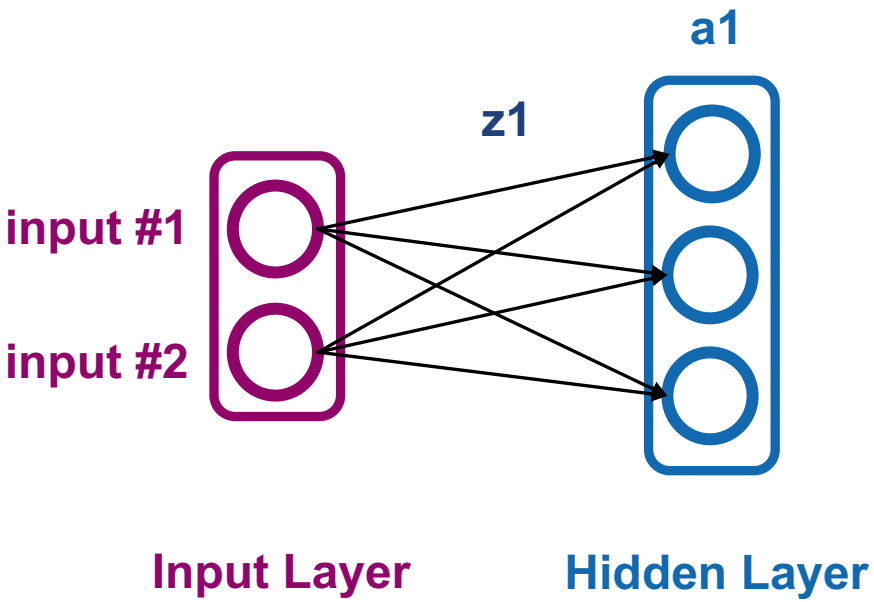
$$z_j = f_j\left(\sum_{i=0} x_i \omega_{ij}\right)$$

Neural Networks: Architectures

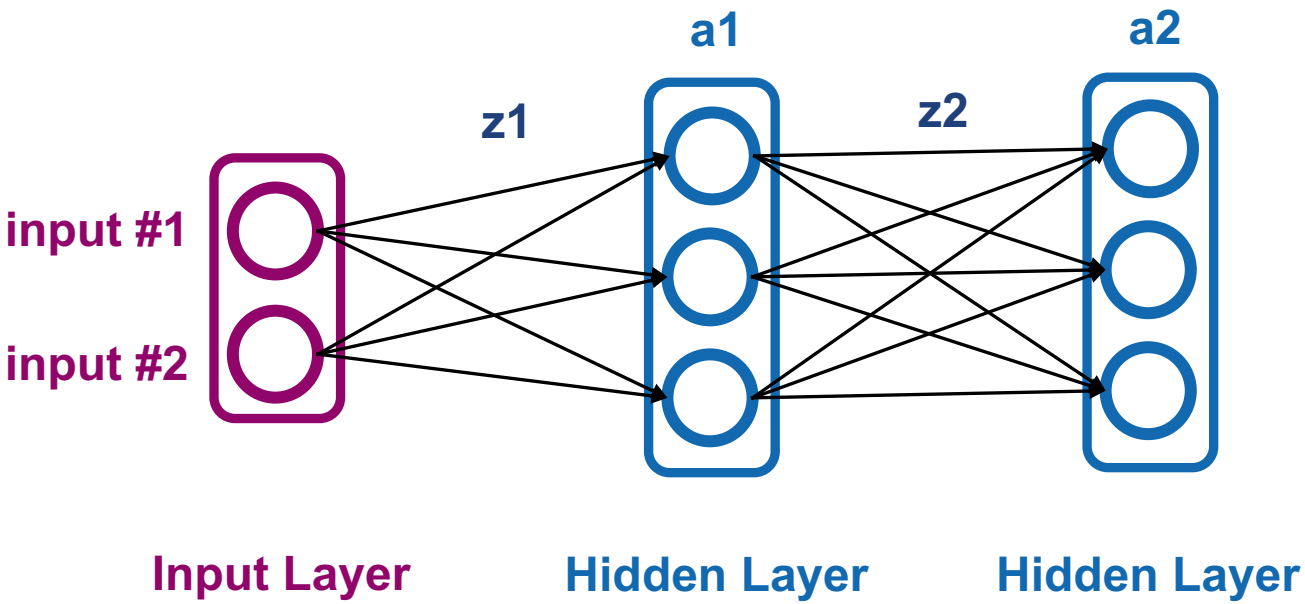


Input Layer

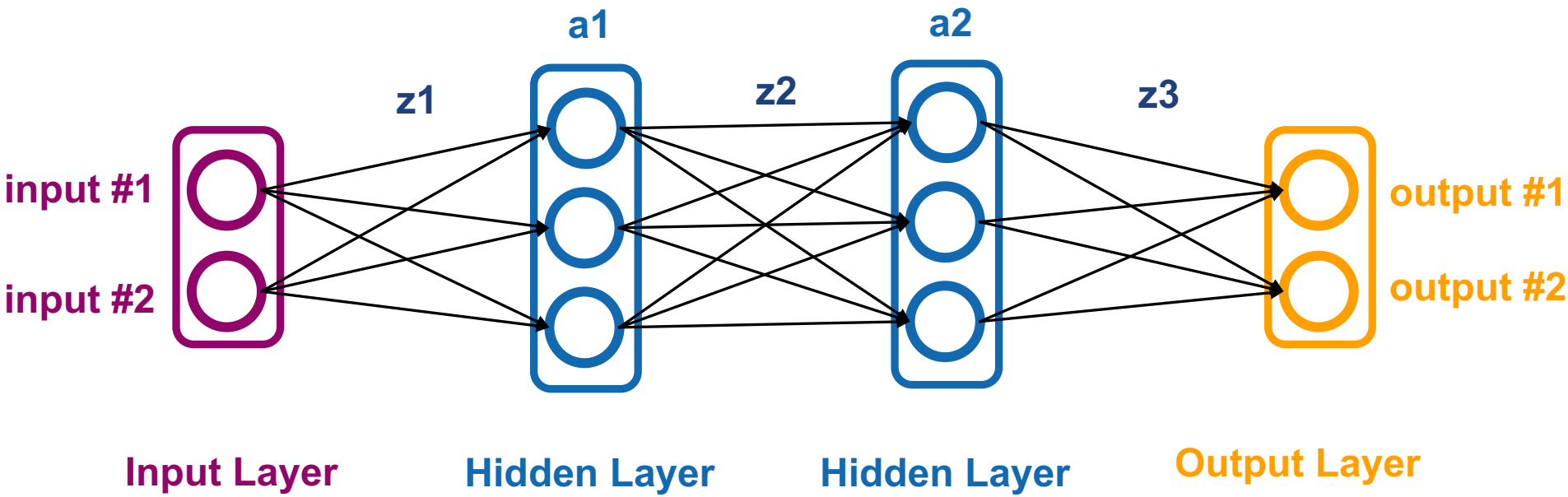
Neural Networks: Architectures



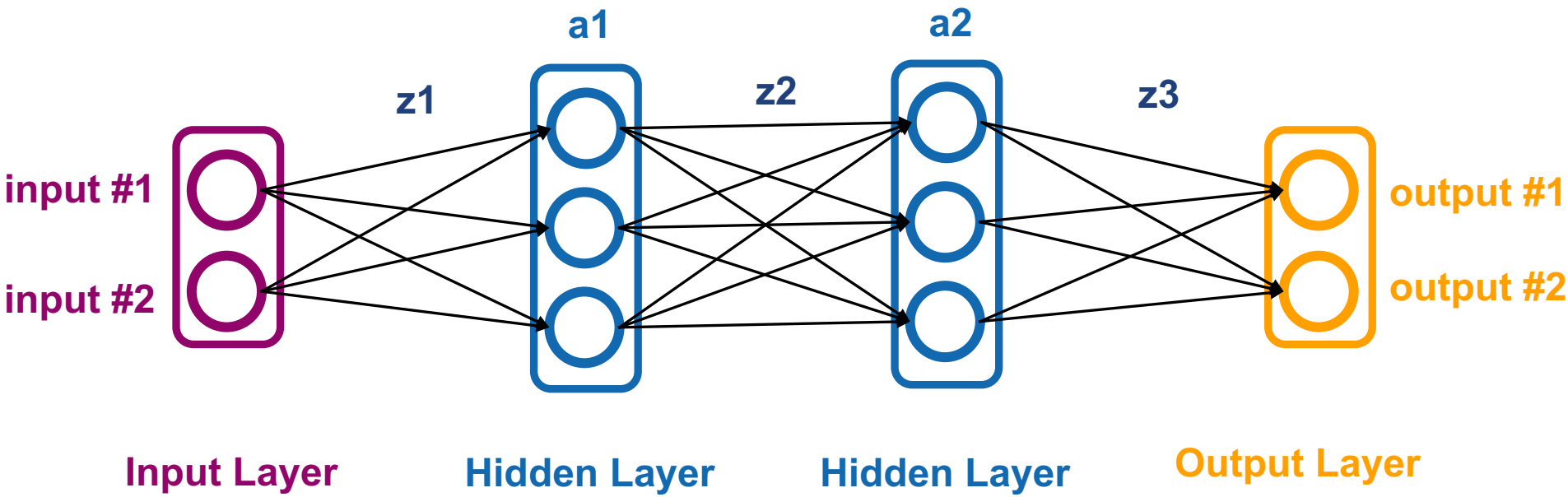
Neural Networks: Architectures



Neural Networks: Architectures



Neural Networks: Architectures

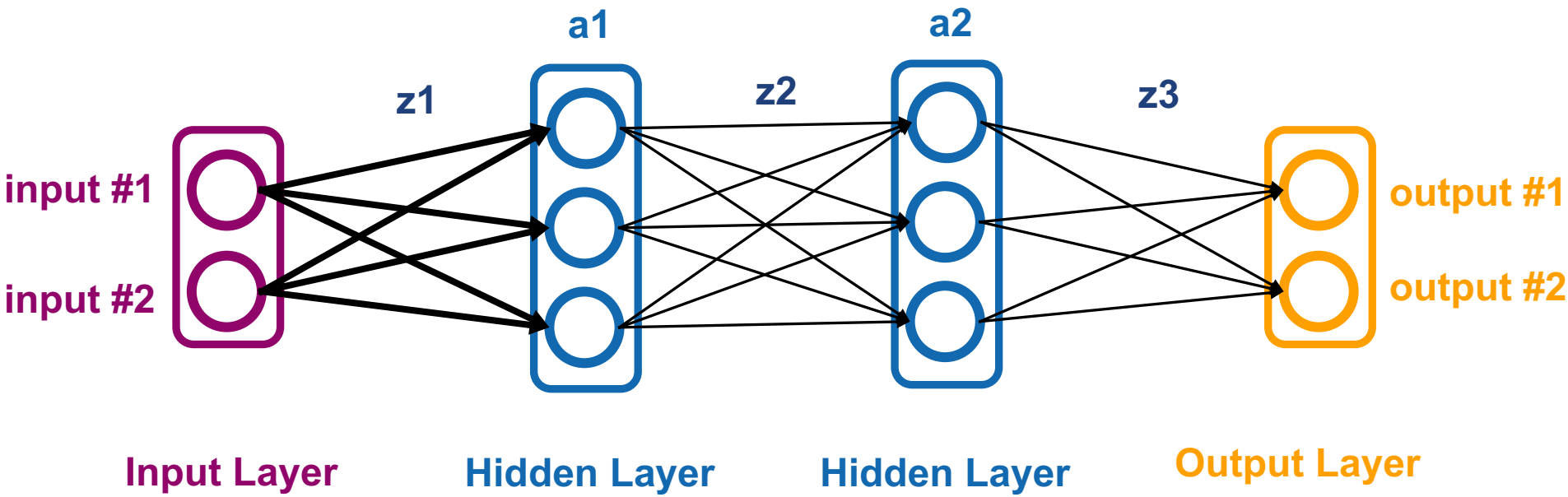



```
# Forward propagation
z1 = X.dot(W1) + b1
a1 = sigmoid(z1)
z2 = a1.dot(W2) + b2
a2 = sigmoid(z2)
z3 = a2.dot(W3) + b3
exp_scores = np.exp(z3)
probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
```

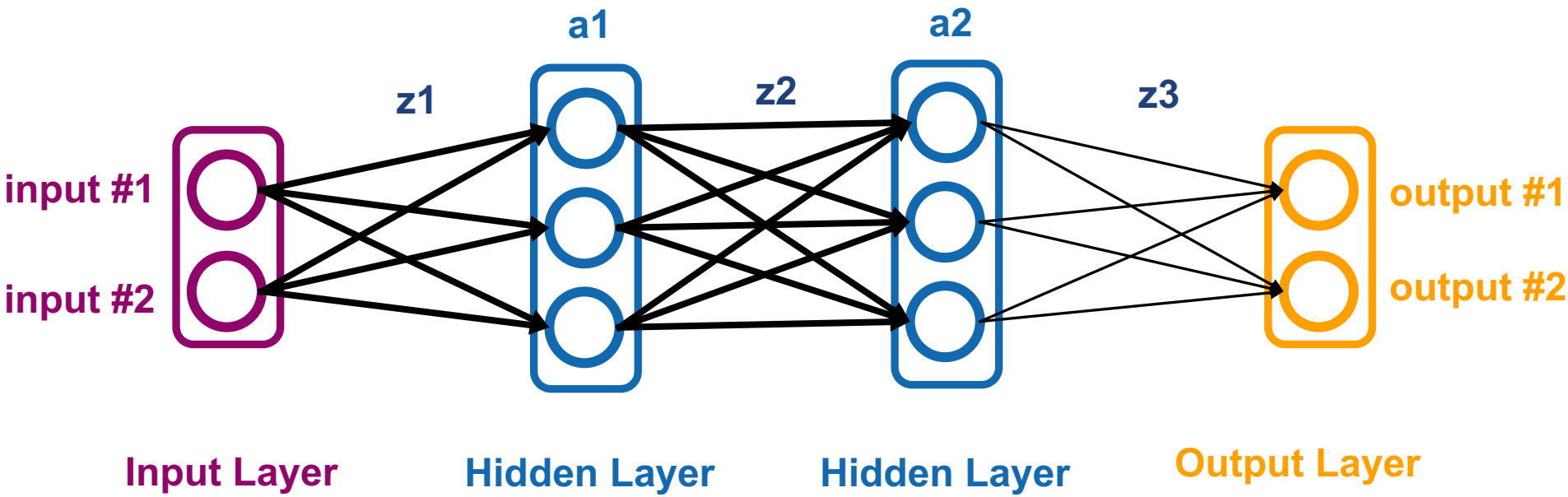
Building Deep Learning Models: The Perceptron Activation Functions Multilayer Perceptron

Building Deep Learning Models:
The Perceptron
Activation Functions
Multilayer Perceptron
Training

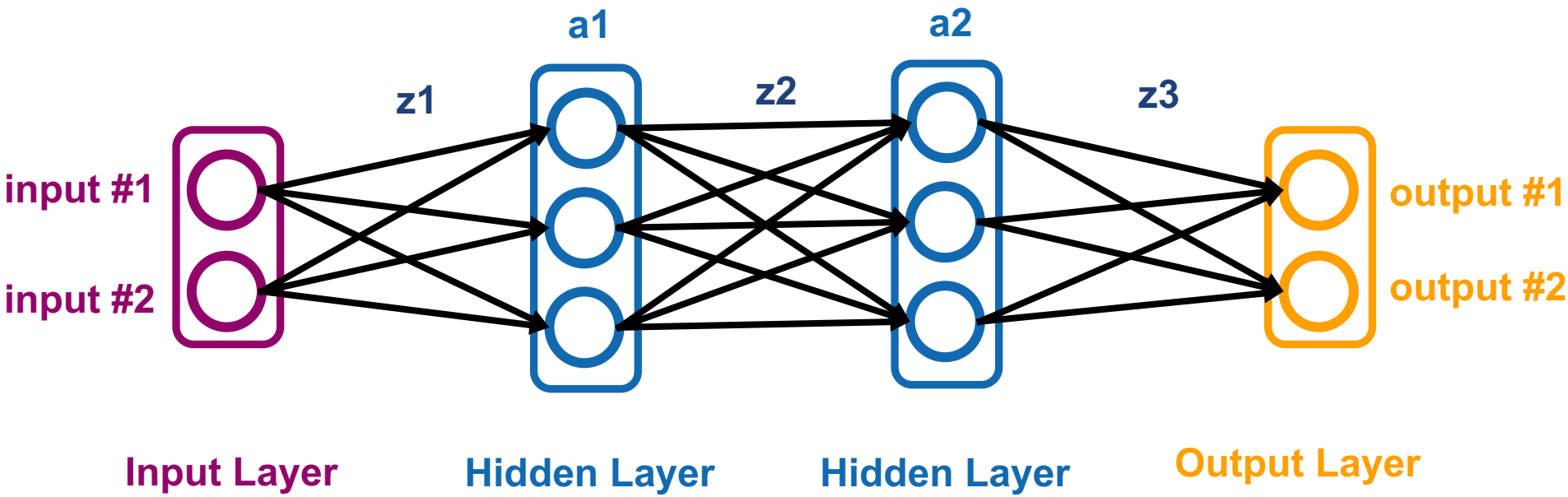
Neural Networks: Feedforward Propagation



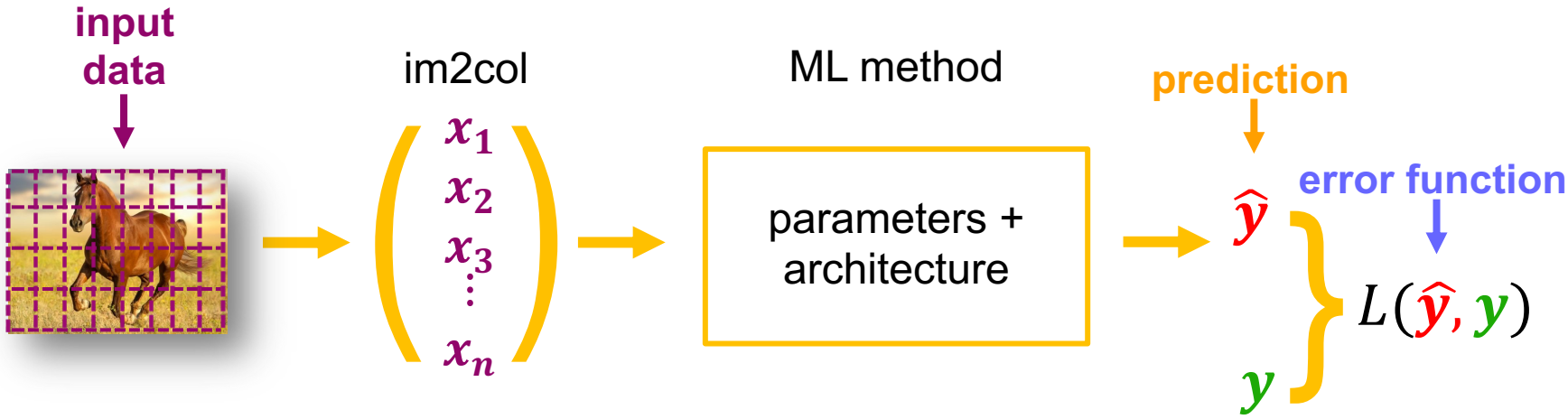
Neural Networks: Feedforward Propagation



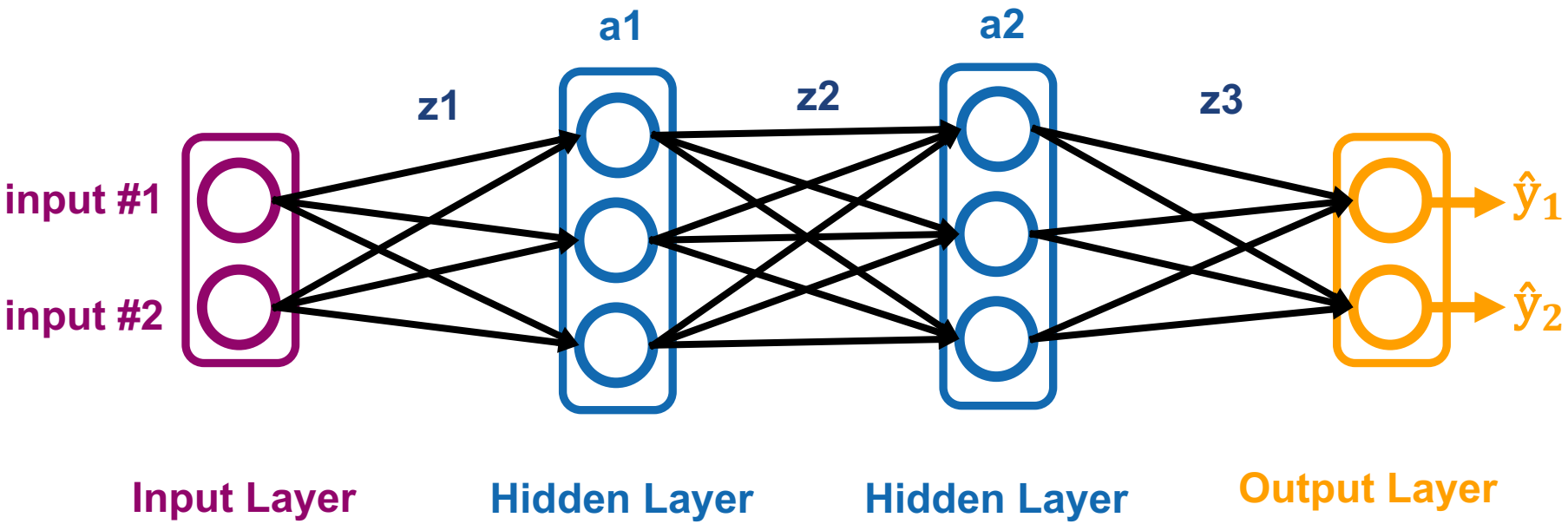
Neural Networks: Feedforward Propagation



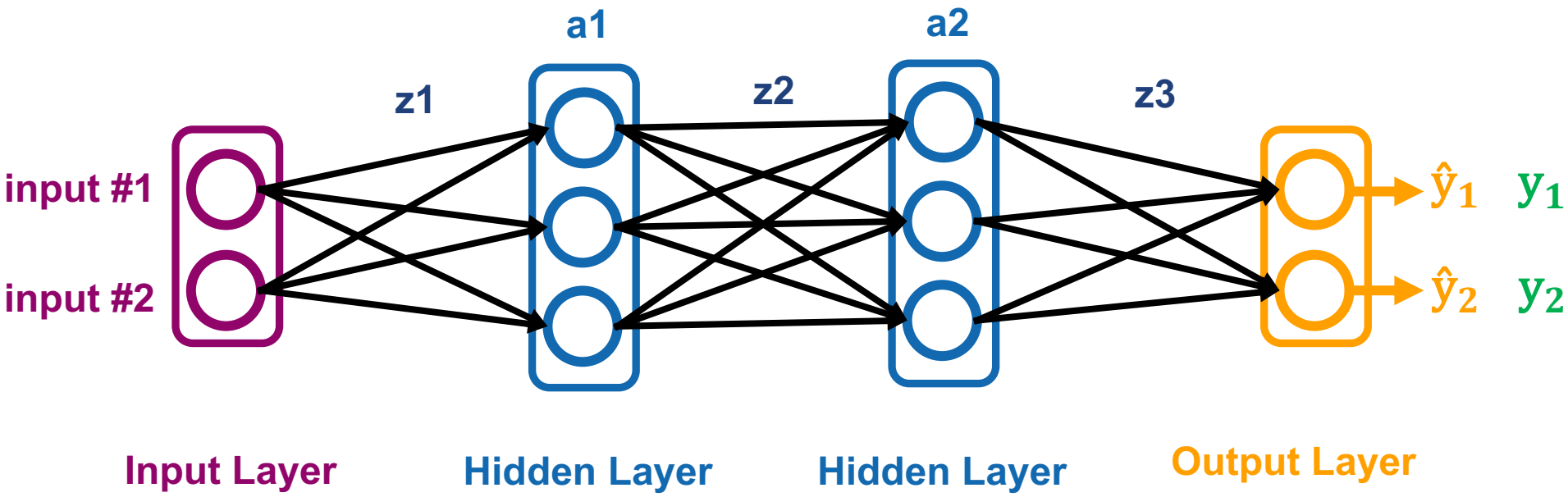
Neural Networks: Error Analysis



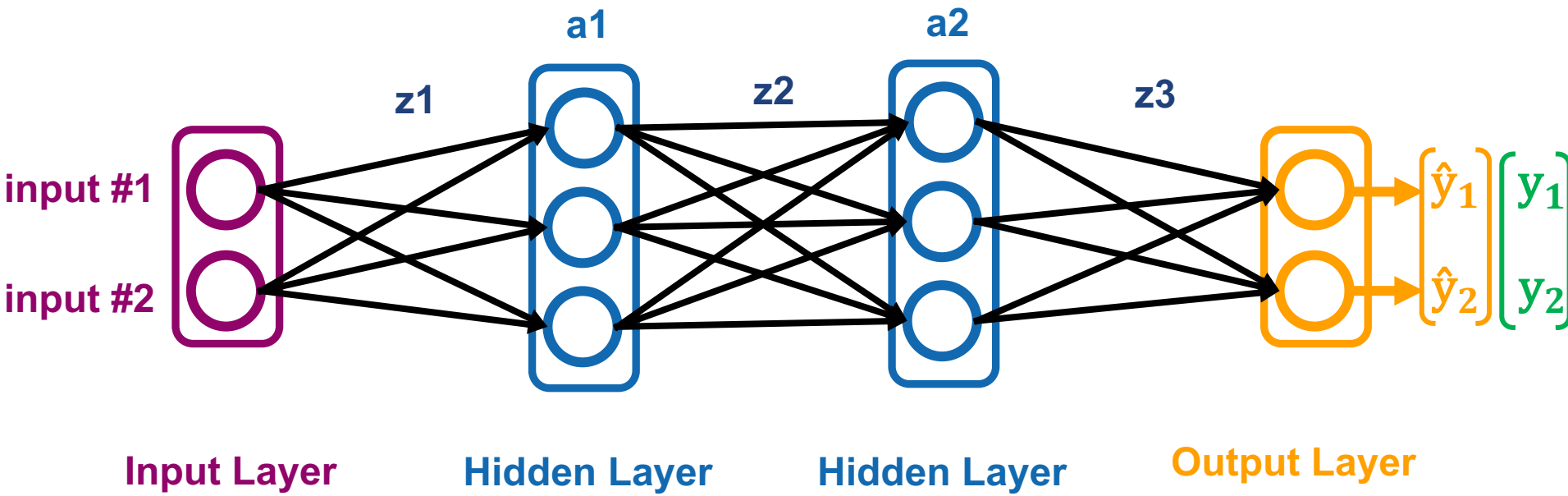
Neural Networks: Error Analysis



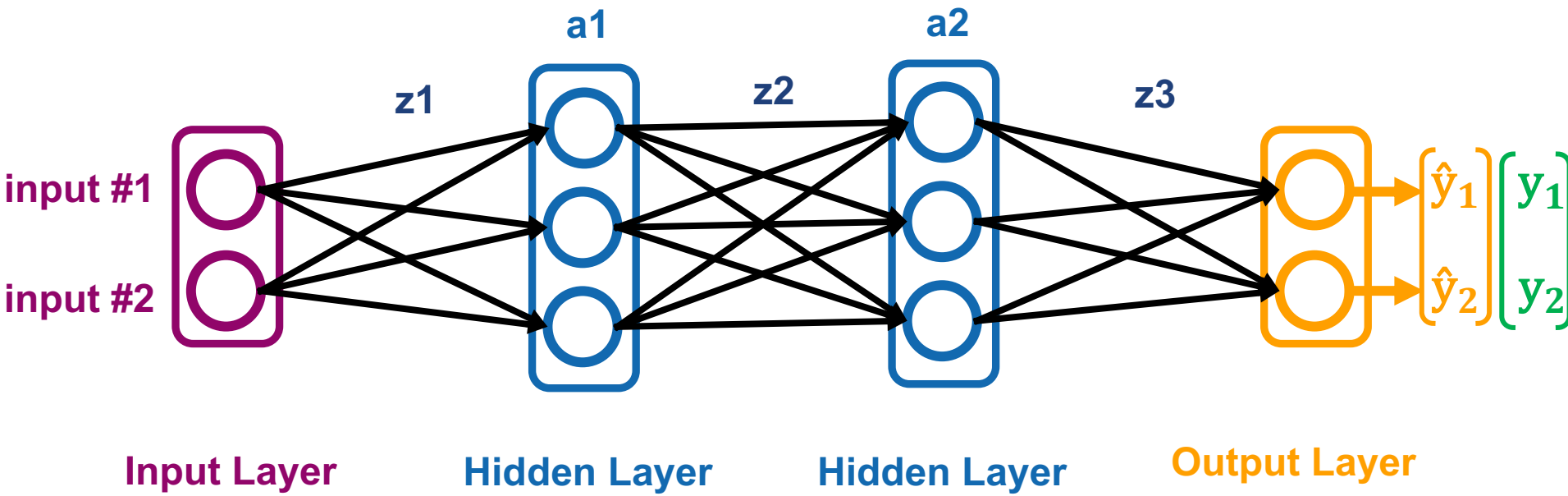
Neural Networks: Error Analysis



Neural Networks: Error Analysis



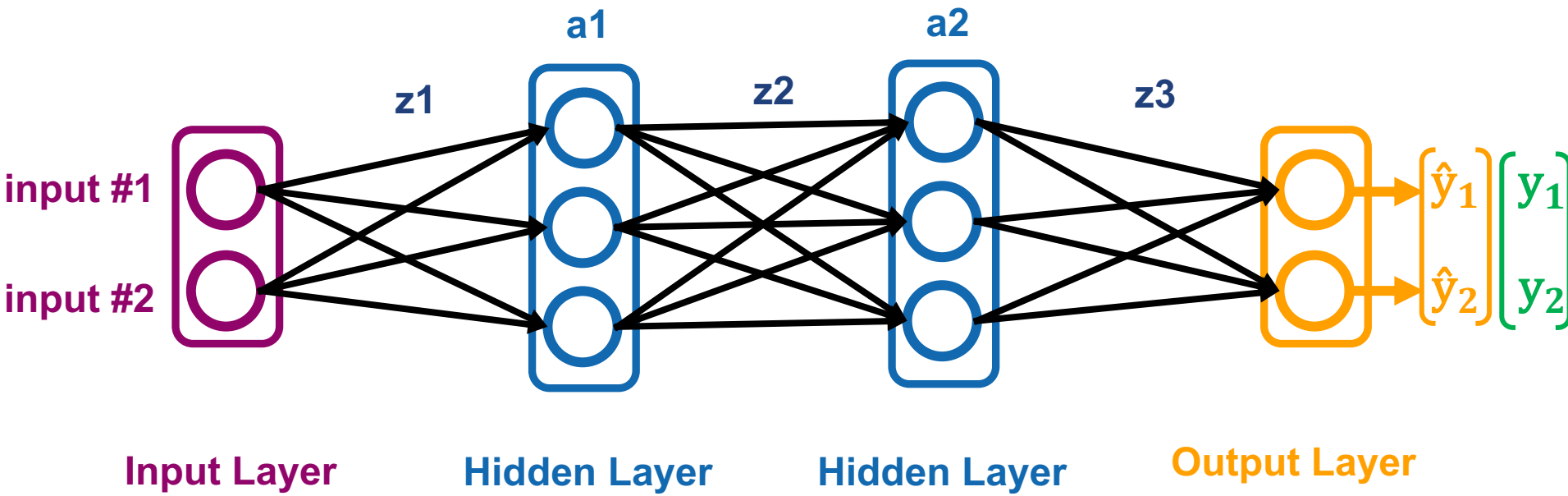
Neural Networks: Error Analysis



Error Function/ Loss Function

$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(\, \boldsymbol{f}(\boldsymbol{x_i};\, \boldsymbol{W}), \boldsymbol{y_i})$$

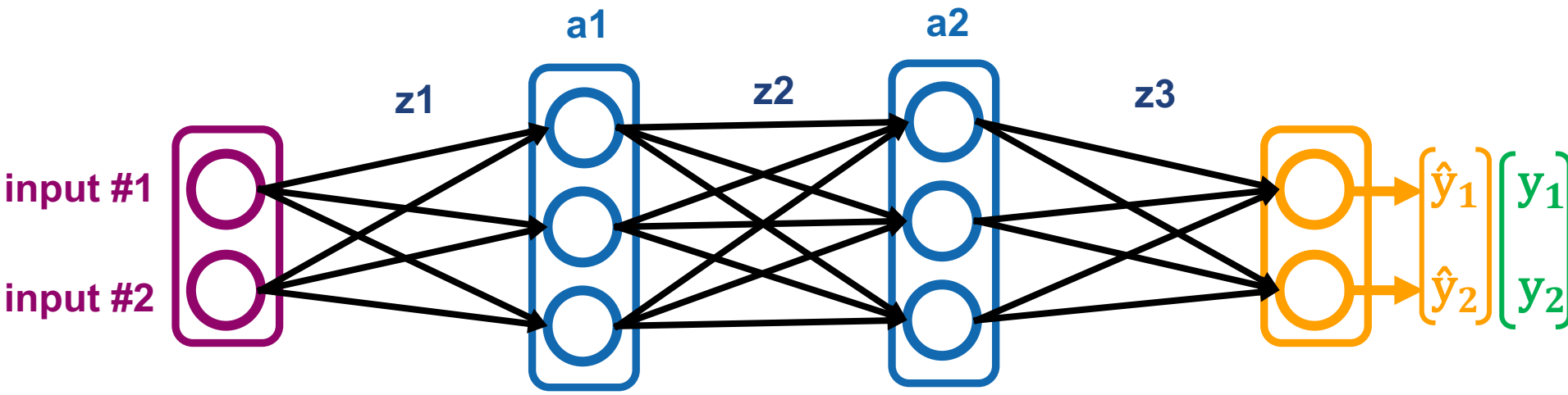
Neural Networks: Error Analysis



Error Function/ Loss Function

$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(\underbrace{f(\boldsymbol{x}_i;\, \boldsymbol{W})}_{\hat{y}_i}, \boldsymbol{y}_i)$$

Neural Networks: Error Analysis



Error Function/ Loss Function

$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\left(\underbrace{f(\boldsymbol{x_i};\, \boldsymbol{W})}_{\hat{y}_i}, \boldsymbol{y_i}\right)$$

$$\boldsymbol{W} = \{\boldsymbol{W_0}, \boldsymbol{W_1}, \cdots\}$$

Loss Functions: Classification Tasks

Training examples

Pairs of N inputs x_i and ground-truth class labels y_i

Output Layer

Softmax Activations
[maps to a probability distribution]

$$P(y = j \mid \mathbf{x}) = \frac{e^{\mathbf{x}^\top \mathbf{w}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}^\top \mathbf{w}_k}}$$

Loss function

Cross-entropy

$$\mathcal{L}(\theta) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{K} \left[y_k^{(i)} \log \hat{y}_k^{(i)} + \left(1 - y_k^{(i)}\right) \log \left(1 - \hat{y}_k^{(i)}\right) \right]$$

Ground-truth class labels y_i and model predicted class labels $\hat{y}_i$

Loss Functions: Regression Tasks

Training examples

Pairs of N inputs x_i and ground-truth output values y_i

Output Layer

Linear (Identity) or Sigmoid Activation

Loss function

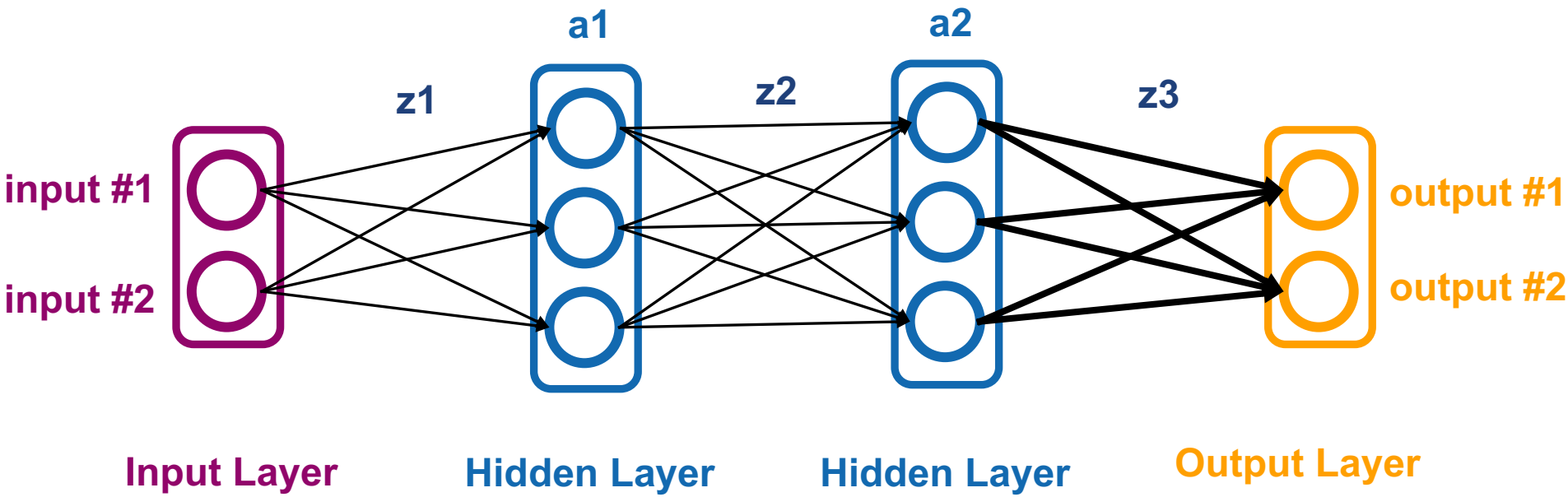
Mean Squared Error

$$\mathcal{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

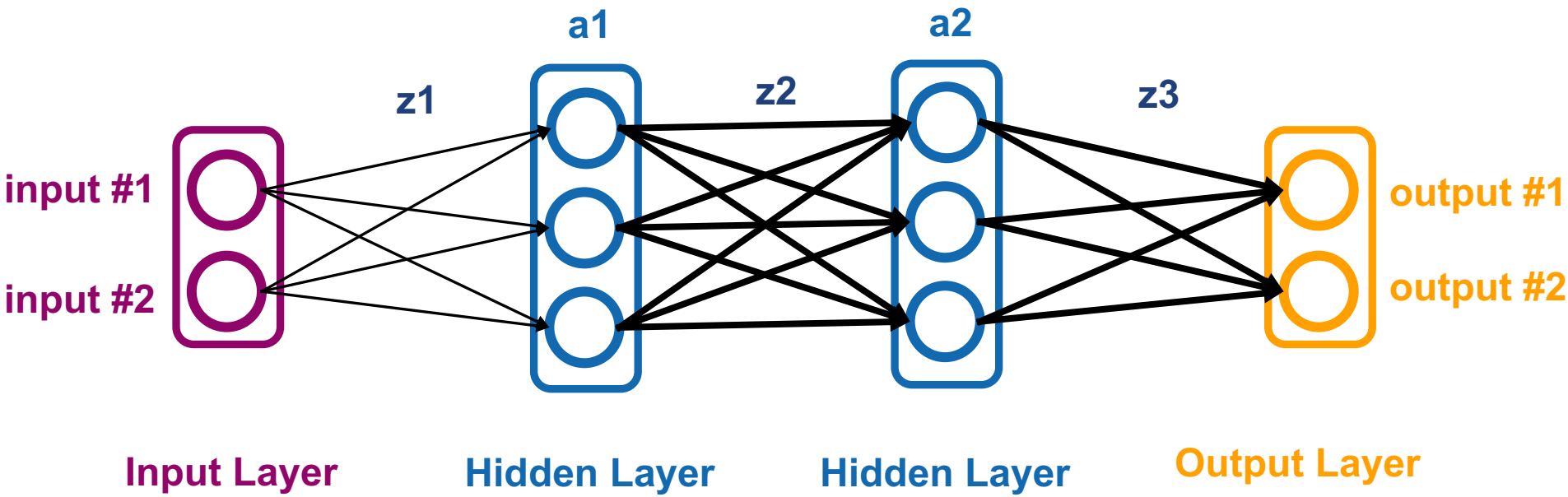
Mean Absolute Error

$$\mathcal{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left|y^{(i)} - \hat{y}^{(i)}\right|$$

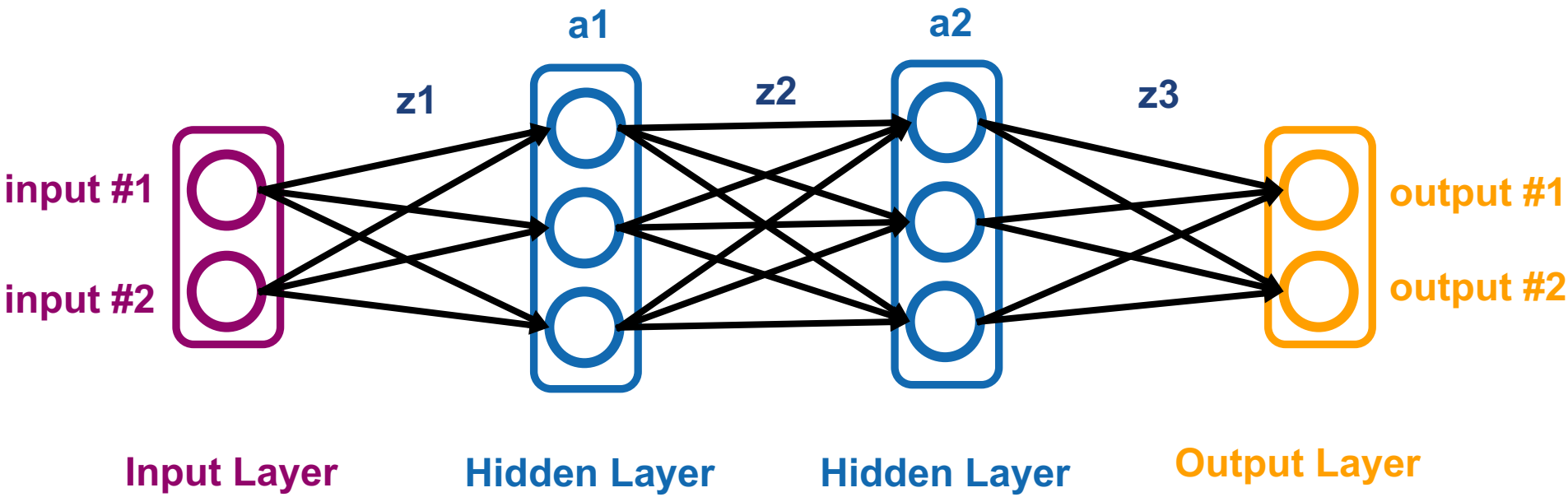
Neural Networks: Backpropagation



Neural Networks: Backpropagation



Neural Networks: Backpropagation



Neural Networks: Backpropagation

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{argmin} \frac{\mathbf{1}}{\boldsymbol{n}} \sum_{i=1}^{n} \mathcal{L}(\boldsymbol{f}(\boldsymbol{x_i}; \boldsymbol{W}), \boldsymbol{y_i})$$

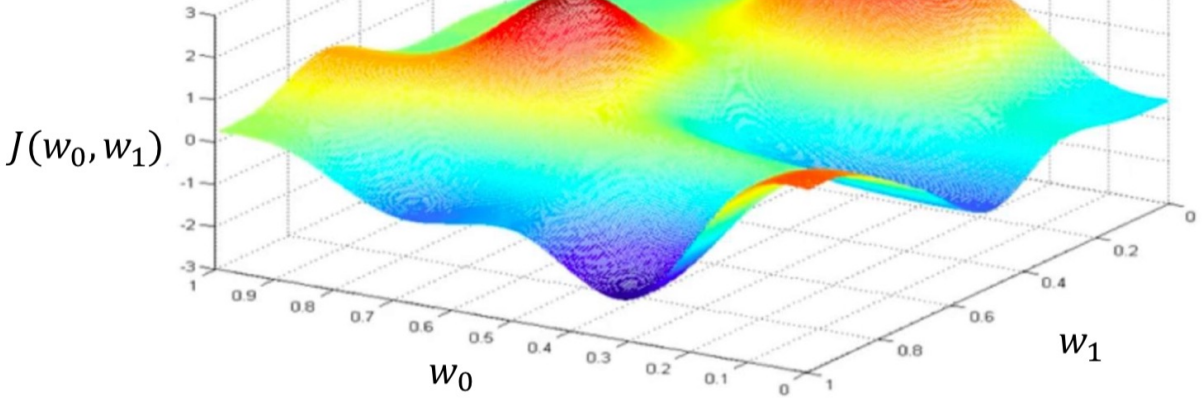
$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{argmin}\, J(\boldsymbol{W})$$

$$\boldsymbol{W} = \{W_0, W_1, \cdots\}$$

Neural Networks: Backpropagation

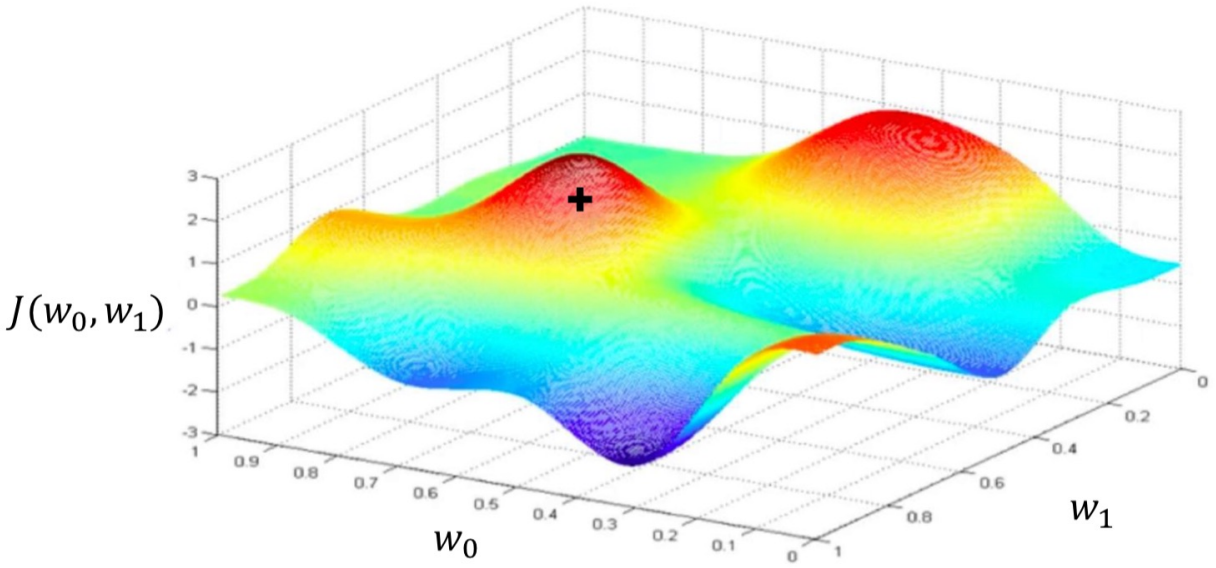
$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} J(\boldsymbol{W})$$

Remember:
Our loss is a function of the network weights!



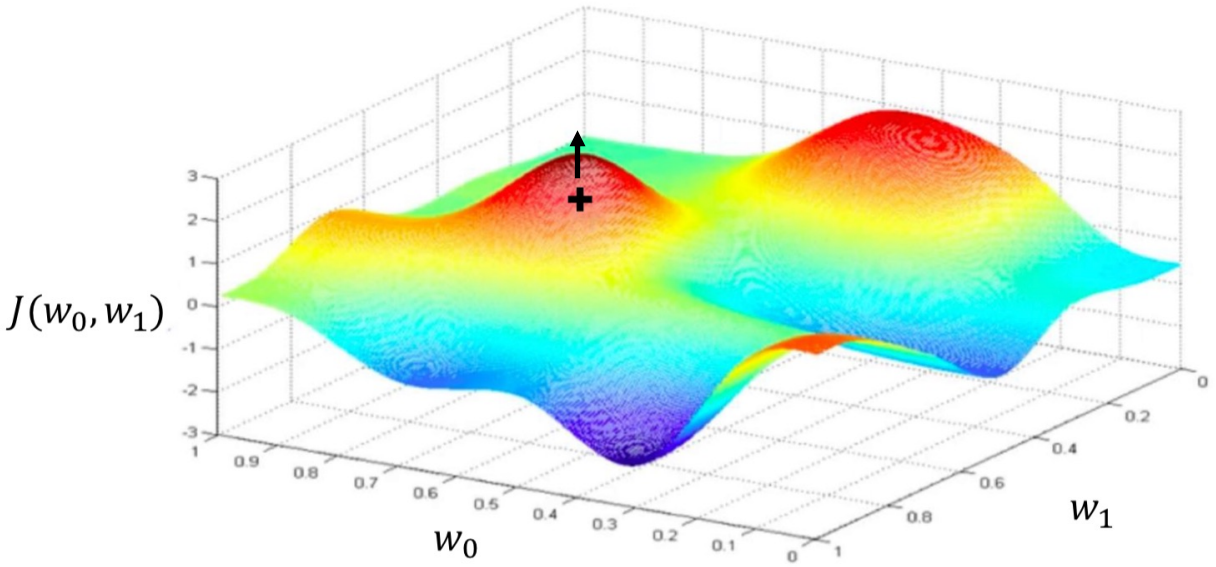
Neural Networks: Backpropagation

1. Random pick initial weights: $(\boldsymbol{w_0}, \boldsymbol{w_1})$



Neural Networks: Backpropagation

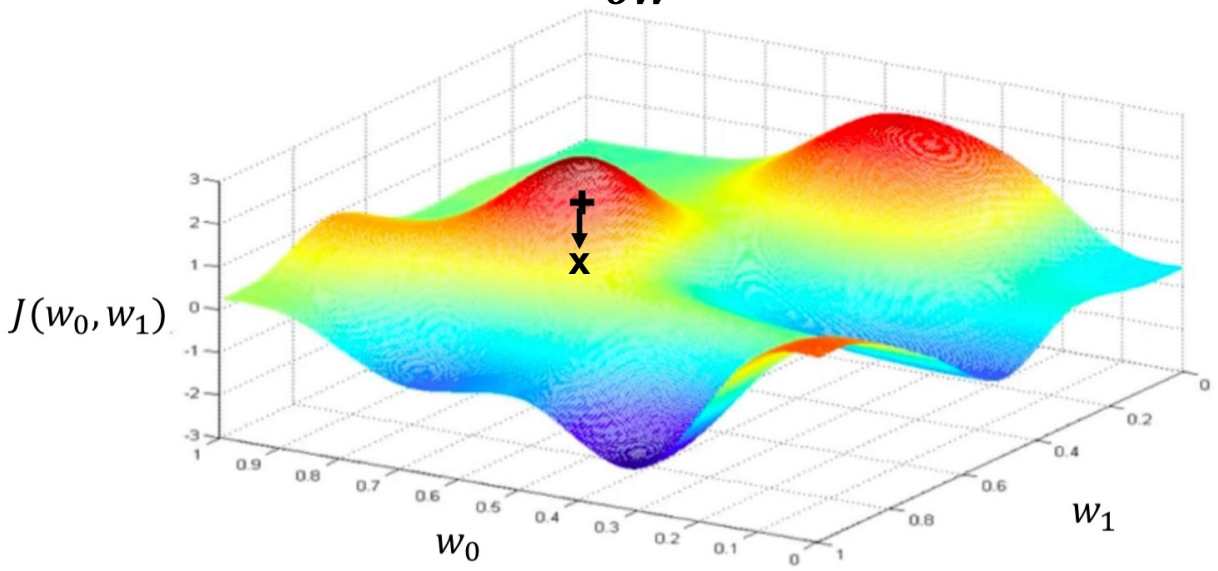
2. Compute gradients: $\frac{\partial J(W)}{\partial W}$



Neural Networks: Backpropagation

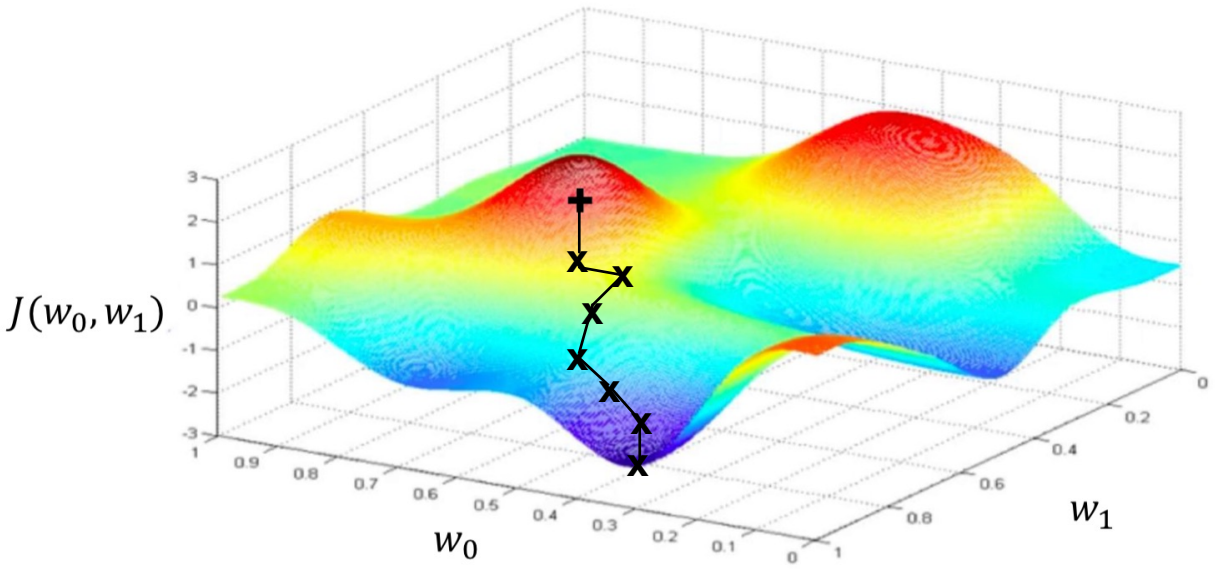
3. Take small step $\boldsymbol{\eta}$ in opposite direction of the gradient:

$$-\boldsymbol{\eta}\frac{\partial \boldsymbol{J}(\boldsymbol{W})}{\partial \boldsymbol{W}}$$

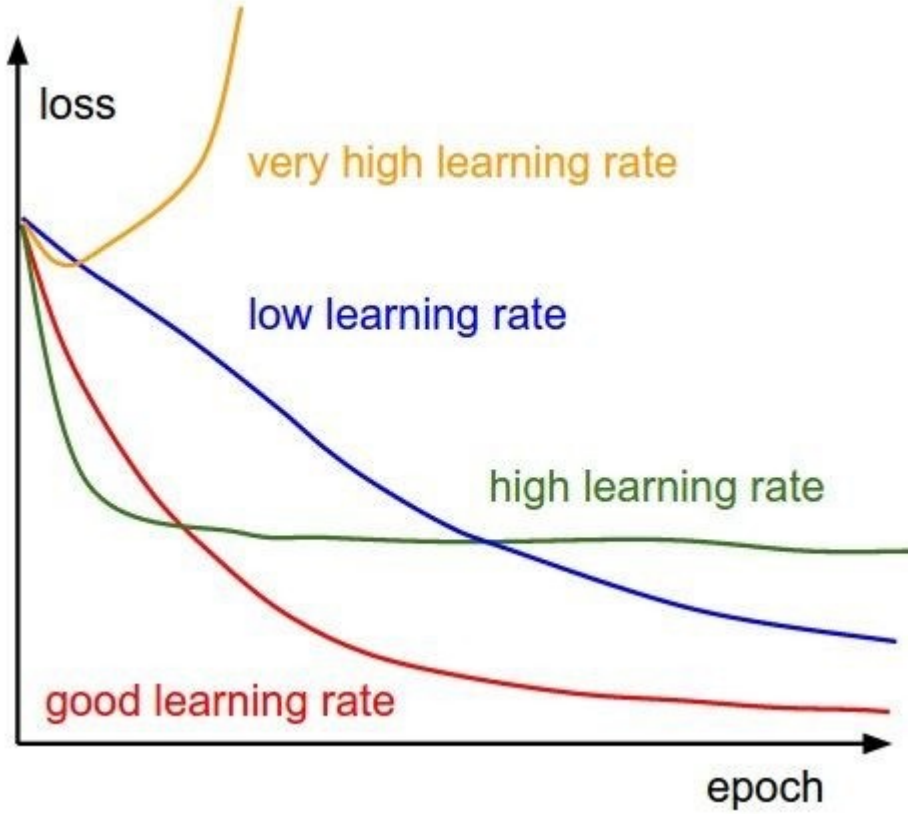


Neural Networks: Backpropagation

4. Repeat until convergence



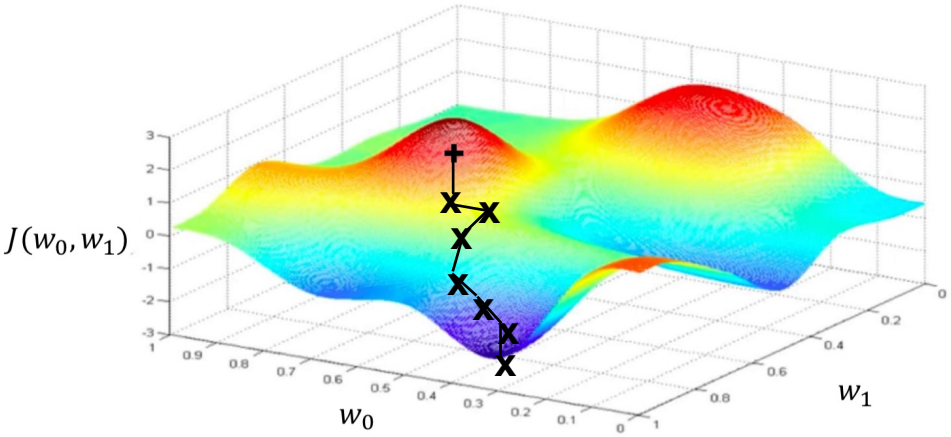
Neural Networks: Training



Neural Networks: Backpropagation

Algorithm:

1. Initialize weights randomly $\sim \mathcal{N}(\mu, \sigma^2)$
2. Repeat until convergence:
   1. Compute gradient: $\frac{\partial J(W)}{\partial W}$
   2. Update weights: $W = W - \eta \frac{\partial J(W)}{\partial W}$
3. Return weights

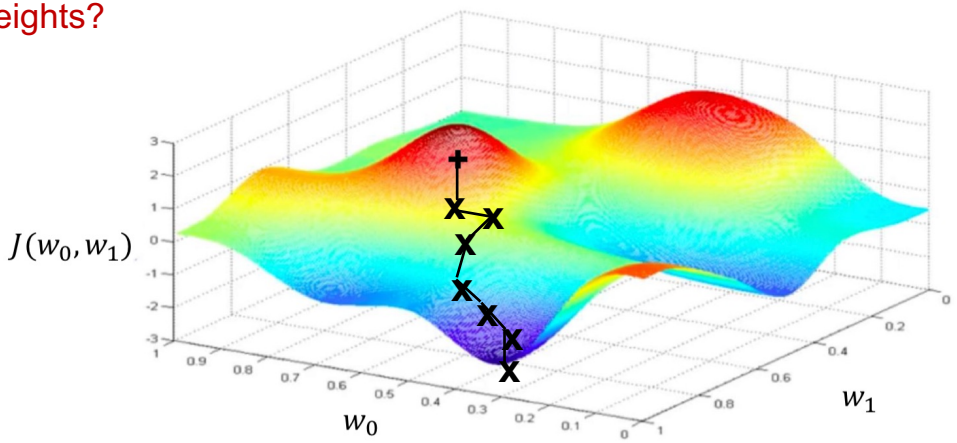


Neural Networks: Backpropagation

Algorithm:

how to initialize the weights?

1. Initialize weights randomly $\sim \mathcal{N}(\mu, \sigma^2)$
2. Repeat until convergence:
   1. Compute gradient: $\frac{\partial J(W)}{\partial W}$
   2. Update weights: $W = W - \eta \frac{\partial J(W)}{\partial W}$
3. Return weights



Weight Initialisation

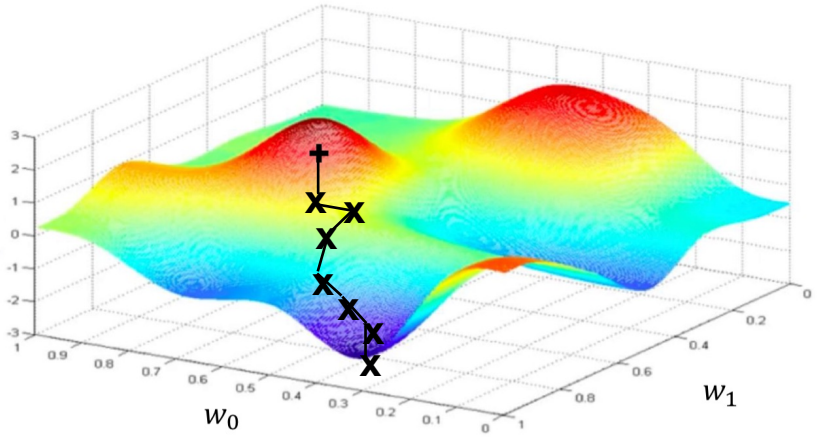
- Typically: initialize to random values sampled from zero-mean Gaussian: $w \sim \mathcal{N}(0, \sigma^2)$
  - Standard deviation matters!
  - Key idea: avoid reducing or amplifying the variance of layer responses, which would lead to vanishing or exploding gradients
- Common heuristics:
  - Xavier initialization: $\sigma^2 = 1/n_{\text{in}}$ or $\sigma^2 = 2/(n_{\text{in}} + n_{\text{out}})$, where n_{in} and n_{out} are the numbers of inputs and outputs to a layer (Glorot and Bengio, 2010)
  - Kaiming initialization (for ReLU): $\sigma^2 = 2/n_{\text{in}}$ (He et al., 2015)
- Initializing biases: just set them to 0

Neural Networks: Backpropagation

Algorithm:

how to initialize the weights?

1. Initialize weights randomly $\sim \mathcal{N}(\mu, \sigma^2)$
2. Repeat until convergence:
   1. Compute gradient: $\frac{\partial J(W)}{\partial W}$ activation functions/ architectures
   2. Update weights: $W = W - \eta \frac{\partial J(W)}{\partial W}$
3. Return weights



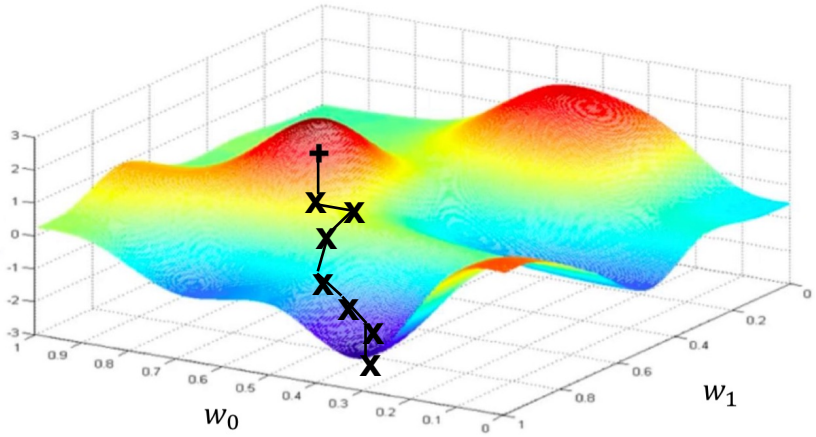
Neural Networks: Backpropagation

Algorithm:

how to initialize the weights?

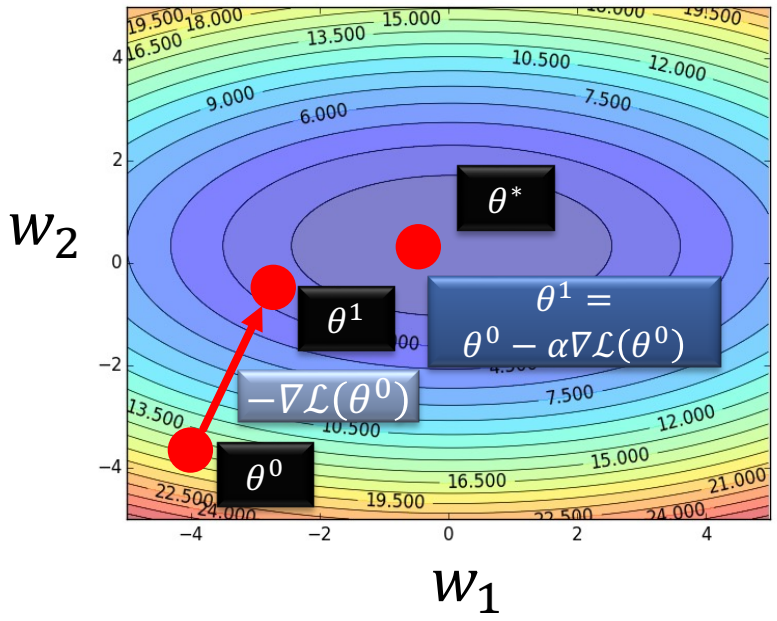
1. Initialize weights randomly $\sim \mathcal{N}(\mu, \sigma^2)$
2. Repeat until convergence:
   1. Compute gradient: $\frac{\partial J(W)}{\partial W}$ activation functions/ architectures
   2. Update weights: $W = W - \eta \frac{\partial J(W)}{\partial W}$
3. Return weights

how to update weights?



Gradient Descent Algorithm

- Example: a NN with only 2 parameters w_1 and w_2, i.e., $\theta = \{w_1, w_2\}$
  - The different colors represent the values of the loss (minimum loss θ^* is ≈ 1.3)

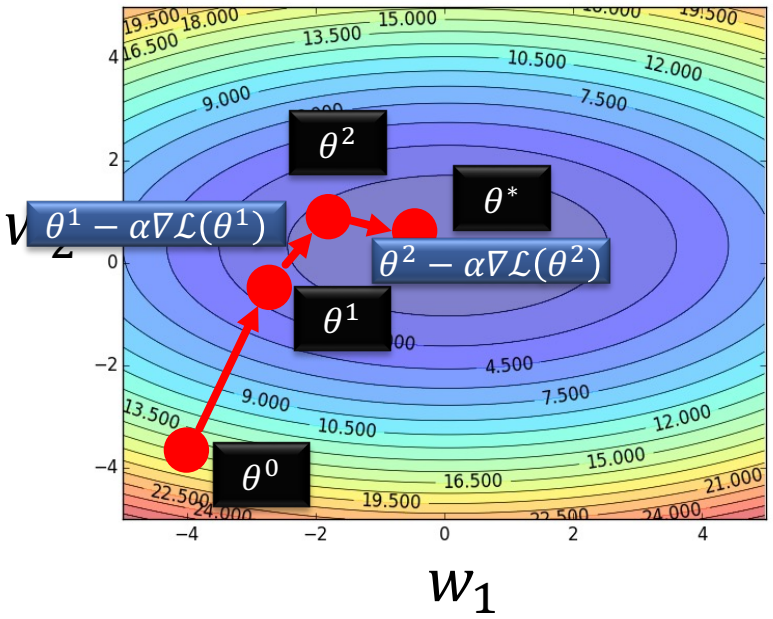


1. Randomly pick a starting point θ^0
2. Compute the gradient at θ^0, $\nabla\mathcal{L}(\theta^0)$
3. Times the learning rate η, and update θ, $\theta^{new} = \theta^0 - \alpha\nabla\mathcal{L}(\theta^0)$
4. Go to step 2, repeat

$$\nabla\mathcal{L}(\theta^0) = \begin{bmatrix} \partial\mathcal{L}(\theta^0)/\partial w_1 \\ \partial\mathcal{L}(\theta^0)/\partial w_2 \end{bmatrix}$$

Gradient Descent Algorithm

- Example: a NN with only 2 parameters w_1 and w_2, i.e., $\theta = \{w_1, w_2\}$
  - The different colors represent the values of the loss (minimum loss θ^* is ≈ 1.3)

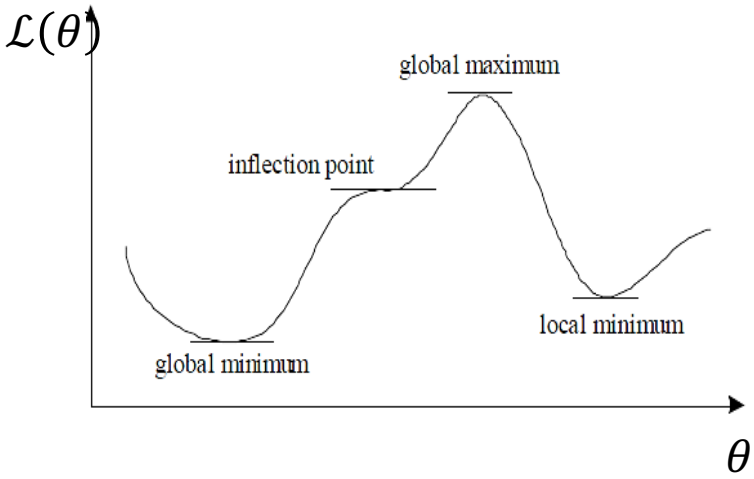


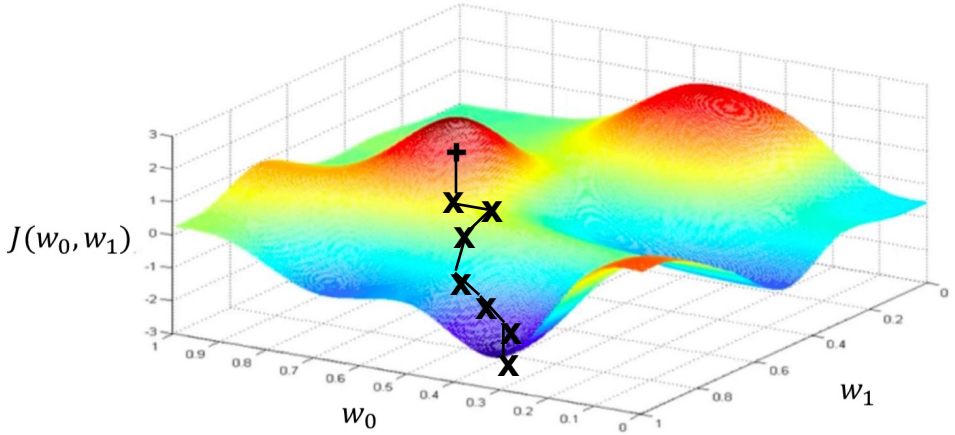
1. Randomly pick a starting point θ^0
2. Compute the gradient at θ^0, $\nabla\mathcal{L}(\theta^0)$
3. Times the learning rate η, and update θ, $\theta^{new} = \theta^0 - \alpha\nabla\mathcal{L}(\theta^0)$
4. Go to step 2, repeat

$$\nabla\mathcal{L}(\theta^0) = \begin{bmatrix} \partial\mathcal{L}(\theta^0)/\partial w_1 \\ \partial\mathcal{L}(\theta^0)/\partial w_2 \end{bmatrix}$$

Gradient Descent Algorithm

- Gradient descent algorithm stops when a local minimum of the loss surface is reached
  - GD does not guarantee reaching a global minimum
  - However, empirical evidence suggests that GD works well for NNs

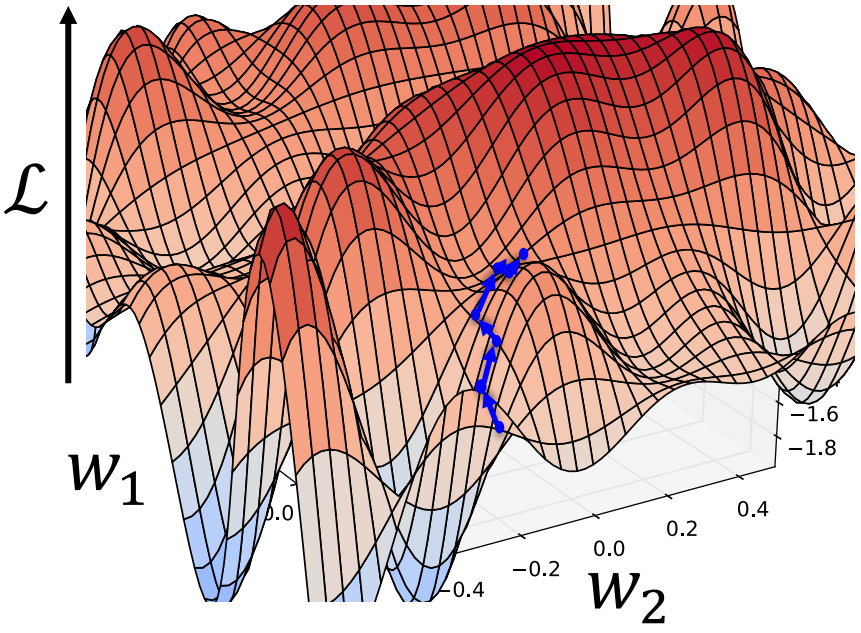




Picture from: https://blog.paperspace.com/intro-to-optimization-in-deep-learning-gradient-descent/

Gradient Descent Algorithm

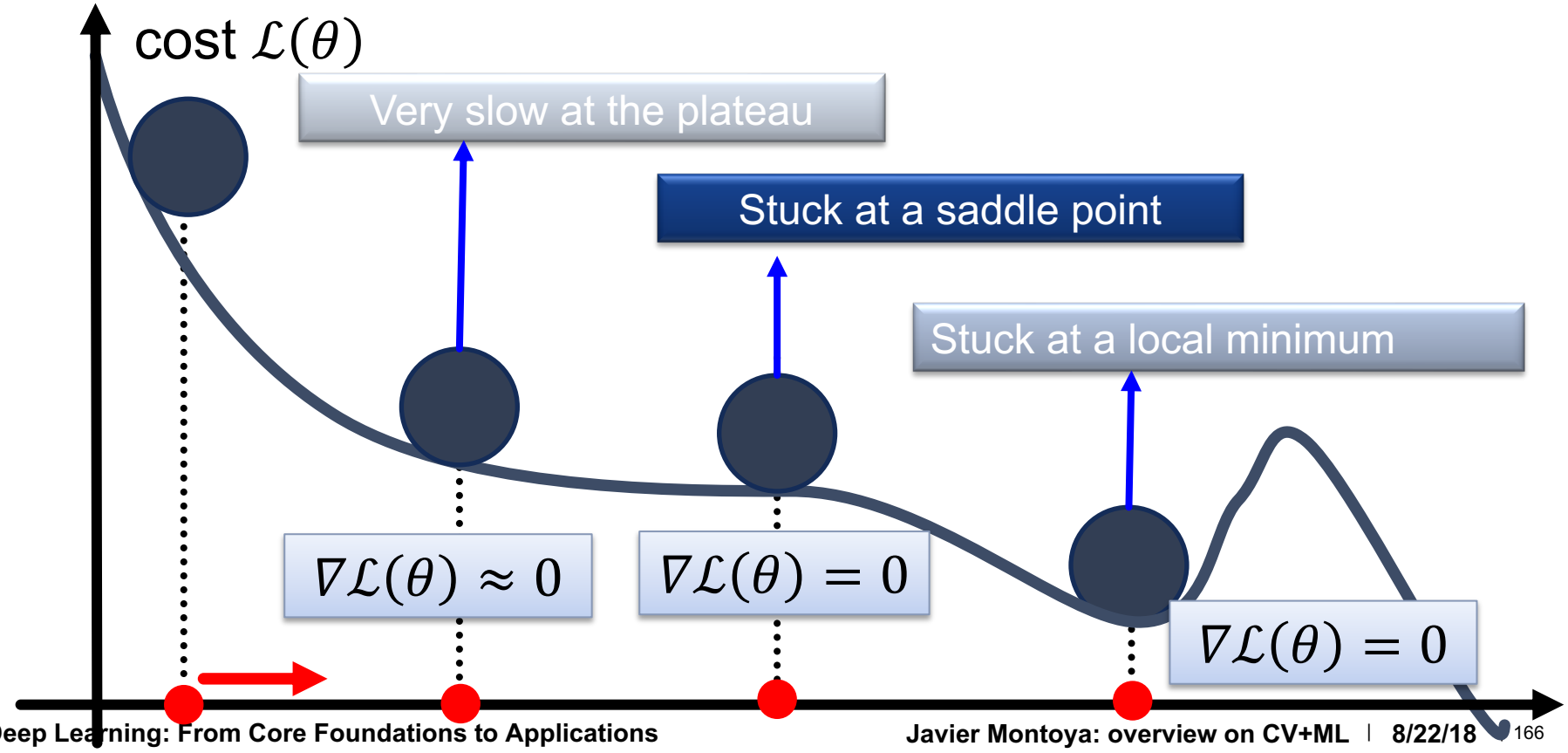
- For most tasks, the loss surface $\mathcal{L}(\theta)$ is highly complex (and non-convex)



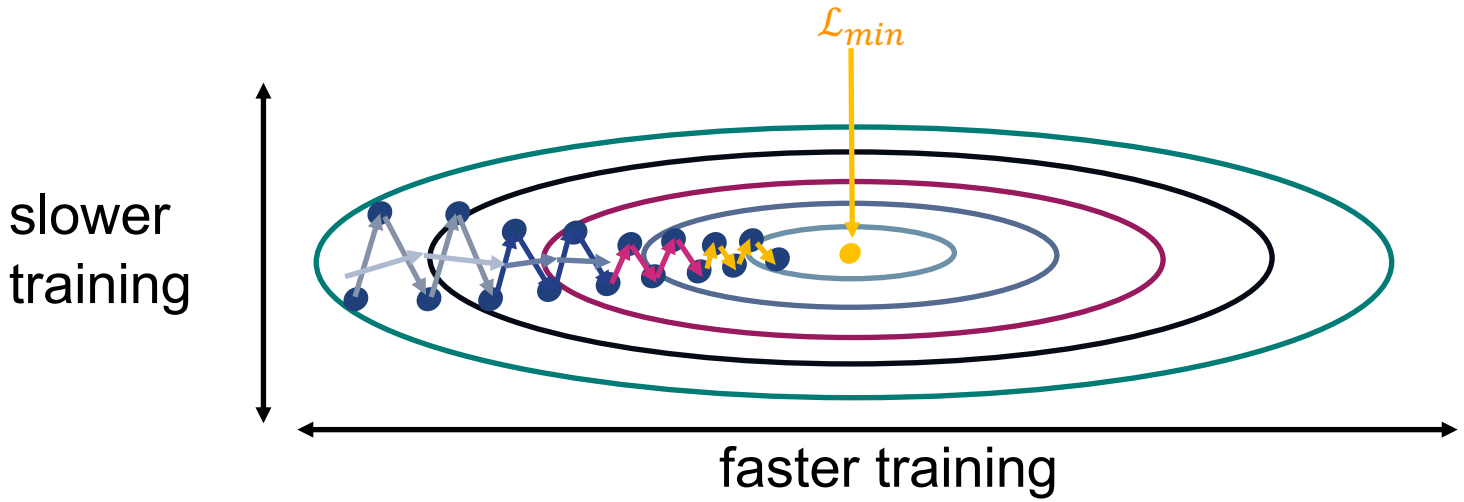
- Random initialization in NNs results in different initial parameters θ^0 every time the NN is trained
  - Gradient descent may reach different minima at every run
  - Therefore, NN will produce different predicted outputs
- In addition, currently we don't have algorithms that guarantee reaching a global minimum for an arbitrary loss function

Gradient Descent Algorithm

- Besides the local minima problem, the GD algorithm can be very slow at plateaus, and it can get stuck at saddle points



Momentum



Remarks:

- In momentum, we accelerate the training by dampening the oscillations during the weight updates.
- The updates are not only done on the opposite direction of the $-\frac{\partial L}{\partial w}$, but also on the average direction on the last updates.

Momentum

SGD

- For every training epoch e:
  - For every minibatch $\{X^{[b]}, Y^{[b]}\}$:

$$\vdots$$

$$W = W - \eta \frac{\partial L}{\partial W}$$

SGD with Momentum

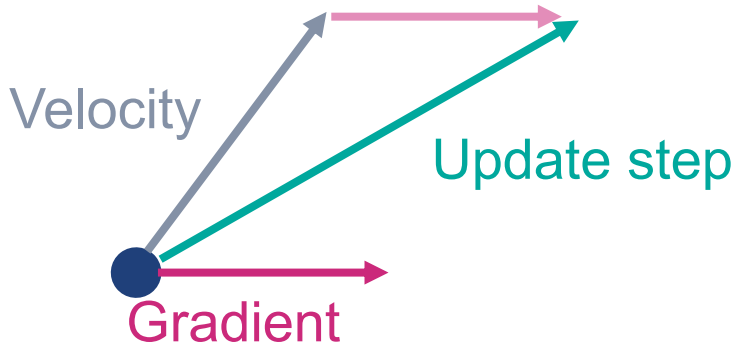
- For every training epoch e:
  - For every minibatch $\{X^{[b]}, Y^{[b]}\}$:

Velocity prev. iter

$$v_{dw} = \beta v_{dw} + (1 - \beta) \frac{\partial L}{\partial W}$$

Velocity

$$W = W - \eta v_{dw}$$



Momentum

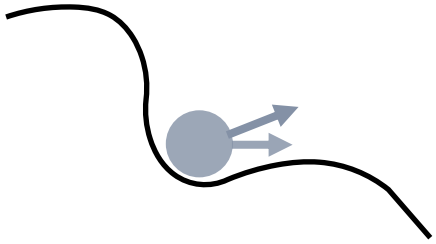
SGD

- For every training epoch e:
  - For every minibatch $\{X^{[b]}, Y^{[b]}\}$:

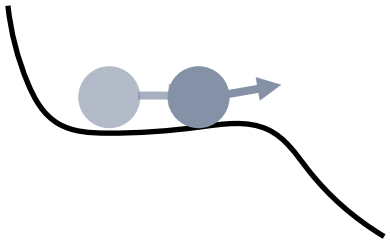
$$\vdots$$

$$W = W - \eta \frac{\partial L}{\partial W}$$

Local minima



Saddle points



SGD with Momentum

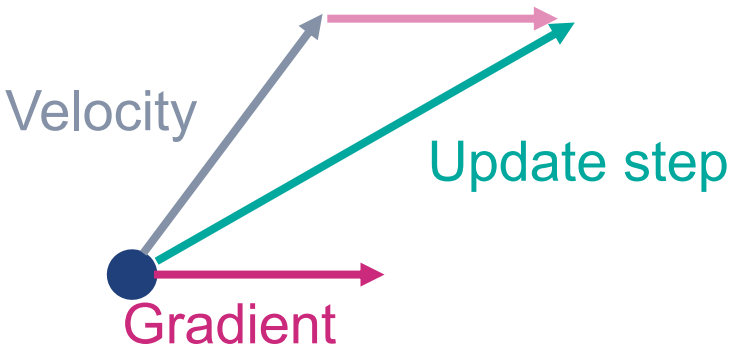
- For every training epoch e:
  - For every minibatch $\{X^{[b]}, Y^{[b]}\}$:

Velocity prev. iter

$$v_{dw} = \beta v_{dw} + (1 - \beta) \frac{\partial L}{\partial W}$$

Velocity

$$W = W - \eta v_{dw}$$



Momentum

- Introduce a “momentum” variable m and associated “friction” coefficient β:

$$v_{dw} = \beta v_{dw} + (1 - \beta)\frac{\partial L}{\partial W}$$
$$W = W - \eta v_{dw}$$

  - Move faster in directions with consistent gradient
  - Avoid oscillating in directions with large but inconsistent gradients

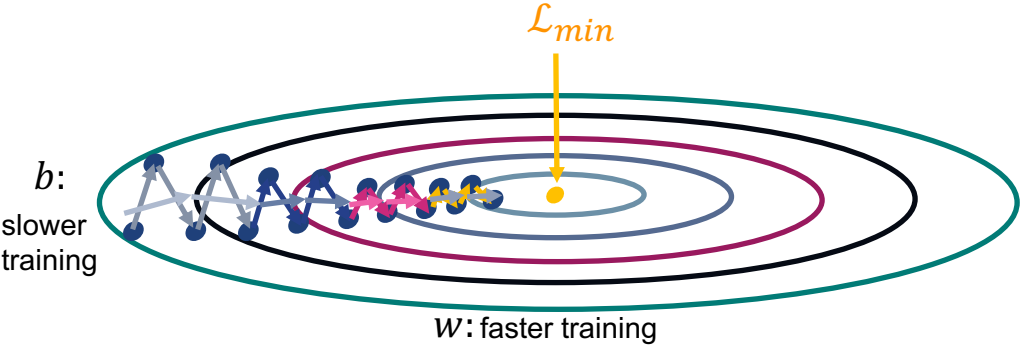
Root Mean Square Prop (RMSProp)

SGD

- For every training epoch e:
  - For every minibatch $\{X^{[b]}, Y^{[b]}\}$:

$$\vdots$$

$$W = W - \eta \frac{\partial L}{\partial W}$$



RMSProp

- For every training epoch e:
  - For every minibatch $\{X^{[b]}, Y^{[b]}\}$:

$$\vdots$$

$$s_{dw} = \beta s_{dw} + (1-\beta)\left(\frac{\partial L}{\partial W}\right)^2$$

$$W = W - \eta \frac{\frac{\partial L}{\partial W}}{\sqrt{s_{dw} + \epsilon}}$$

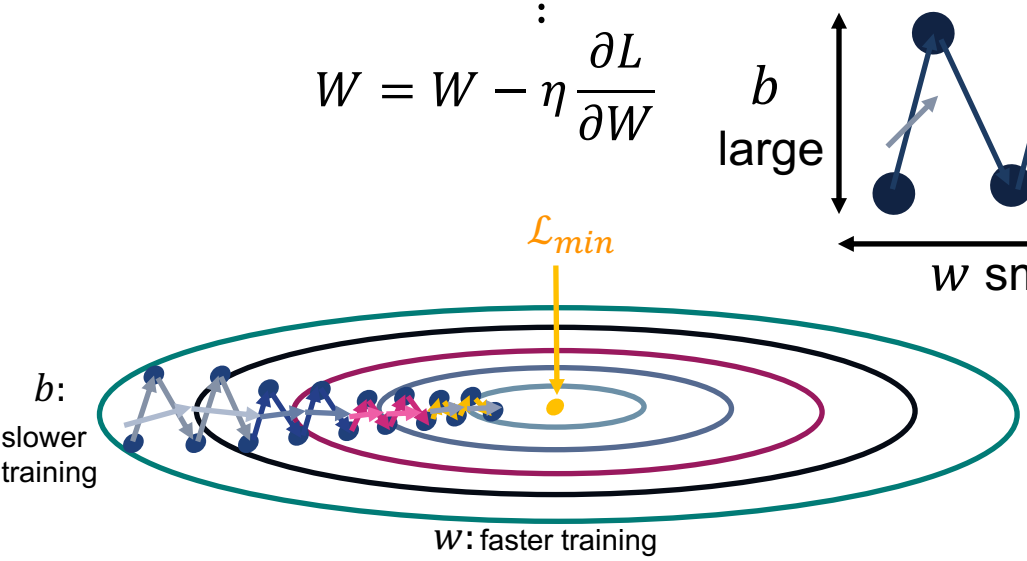
Root Mean Square Prop (RMSProp)

SGD

- For every training epoch e:
  - For every minibatch $\{X^{[b]}, Y^{[b]}\}$:

$$\vdots$$

$$W = W - \eta \frac{\partial L}{\partial W}$$



RMSProp

- For every training epoch e:
  - For every minibatch $\{X^{[b]}, Y^{[b]}\}$:

$$\vdots$$

$$s_{dw} = \beta s_{dw} + (1-\beta)\left(\frac{\partial L}{\partial W}\right)^2$$

large

$$W = W - \eta \frac{\frac{\partial L}{\partial W}}{\sqrt{s_{dw} + \epsilon}}$$

small

large

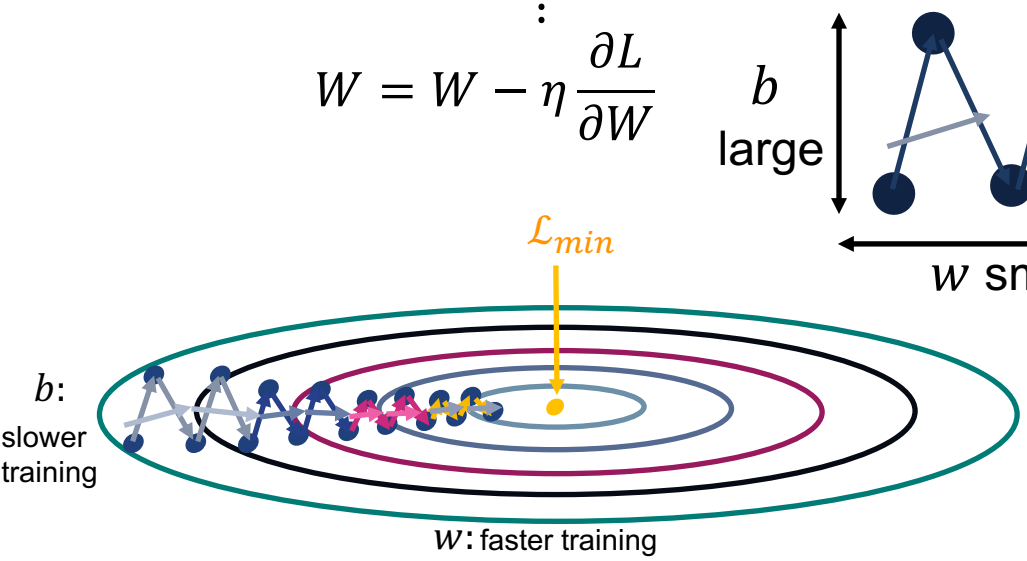
Root Mean Square Prop (RMSProp)

SGD

- For every training epoch e:
  - For every minibatch $\{X^{[b]}, Y^{[b]}\}$:

$$\vdots$$

$$W = W - \eta \frac{\partial L}{\partial W}$$



RMSProp

- For every training epoch e:
  - For every minibatch $\{X^{[b]}, Y^{[b]}\}$:

$$\vdots$$

$$s_{dw} = \beta s_{dw} + (1-\beta)\left(\frac{\partial L}{\partial W}\right)^2$$

small

$$W = W - \eta \frac{\frac{\partial L}{\partial W}}{\sqrt{s_{dw} + \epsilon}}$$

large

small

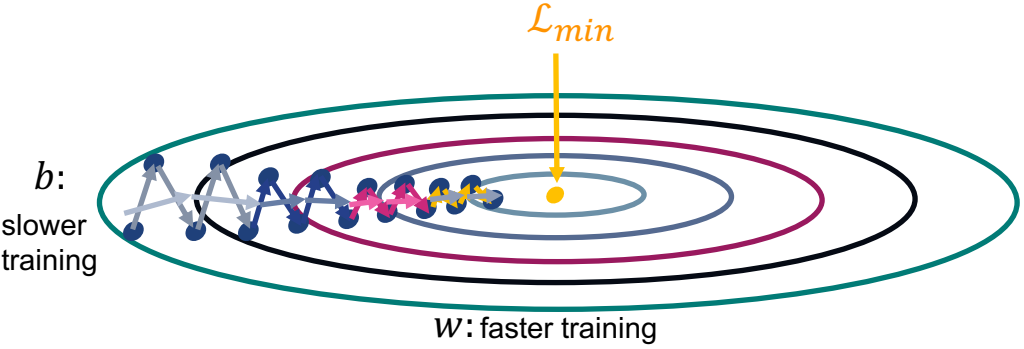
Root Mean Square Prop (RMSProp)

SGD

- For every training epoch e:
  - For every minibatch $\{X^{[b]}, Y^{[b]}\}$:

$$\vdots$$

$$W = W - \eta \frac{\partial L}{\partial W}$$



RMSProp

- For every training epoch e:
  - For every minibatch $\{X^{[b]}, Y^{[b]}\}$:

$$\vdots$$

$$s_{dw} = \beta s_{dw} + (1-\beta)\left(\frac{\partial L}{\partial W}\right)^2$$

$$W = W - \eta \frac{\frac{\partial L}{\partial W}}{\sqrt{s_{dw} + \epsilon}}$$

Root Mean Square Prop (RMSProp)

- Introduce a decay factor β (typically ≥ 0.9) to downweight past history exponentially:

$$s_{dw} = \beta s_{dw} + (1-\beta)(\frac{\partial L}{\partial W})^2$$

$$W = W - \eta \frac{\frac{\partial L}{\partial W}}{\sqrt{s_{dw} + \epsilon}}$$

Adam: Combining RMSProp with Momentum

- Update momentum:

$$m \leftarrow \beta_1 m + (1 - \beta_1)\nabla L$$

- For each dimension k of the weight vector:

$$v^{(k)} \leftarrow \beta_2 v^{(k)} + (1 - \beta_2)\left(\frac{\partial L}{\partial w^{(k)}}\right)^2$$

$$w^{(k)} \leftarrow w^{(k)} - \frac{\eta}{\sqrt{v^{(k)}} + \epsilon} m^{(k)}$$

- Full algorithm includes *bias correction* to account for m and v starting at 0: $\hat{m} = \frac{m}{1-\beta_1^t}$, $\hat{v} = \frac{v}{1-\beta_2^t}$ (t is the timestep)
- Default parameters from paper are reputed to work well for many models: $\beta_1 = 0.9, \beta_2 = 0.999, \eta = 1e-3, \epsilon = 1e-8$

D. Kingma and J. Ba, Adam: A method for stochastic optimization, ICLR 2015

Mini-batch Gradient Descent Algorithm

- Iterate over epochs
  - Group data into mini-batches of size b
    - Compute gradient of the loss for the mini-batch $(x_1, y_1), \ldots, (x_b, y_b)$:

$$\nabla \hat{L} = \frac{1}{b} \sum_{i=1}^{b} \nabla l(w, x_i, y_i)$$

    - Update parameters:

$$w \leftarrow w - \eta \nabla \hat{L}$$

  - Check for convergence, decide whether to decay learning rate

- What are the hyperparameters?
  - Mini-batch size, learning rate decay schedule, deciding when to stop

Building Deep Learning Models:
The Perceptron
Activation Functions
Multilayer Perceptron
Training
Practical Aspects

Dataset split

Idea #1: choose hyperparameters that best work on the set

Dataset split

Idea #1: choose hyperparameters that best work on the set

Idea #2: split data into **train** and **test** sets

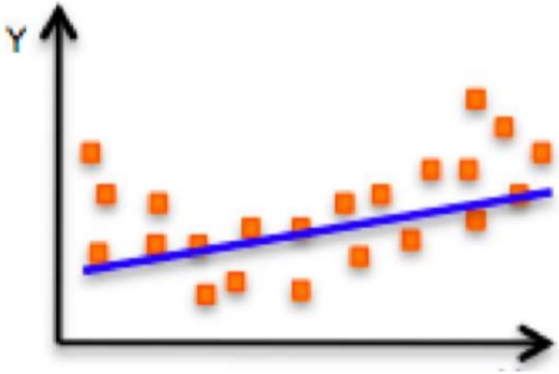
Dataset split

Idea #1: choose hyperparameters that best work on the set

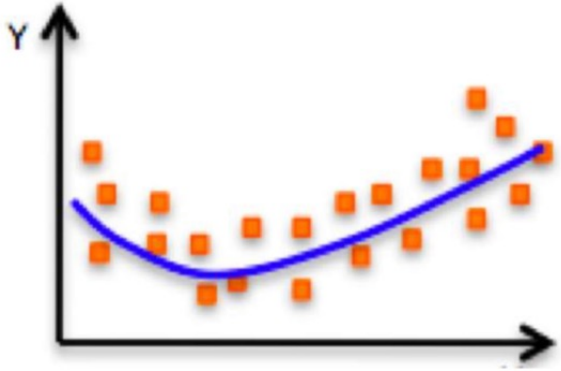
Idea #2: split data into **train** and **test** sets

Idea #3: split data into **train, val,** and **test** sets

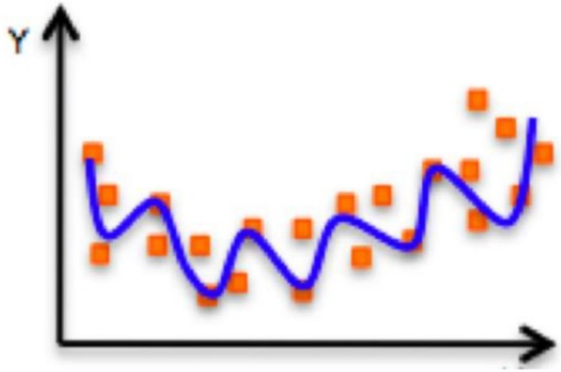
Regularisation



Underfitting

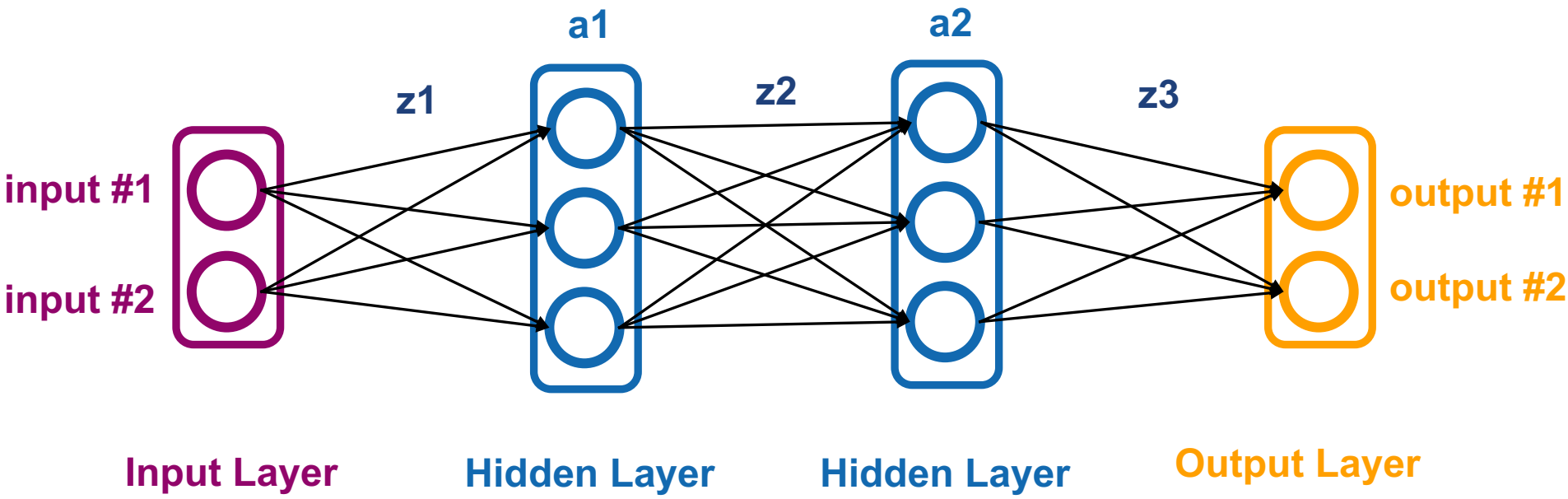


Good fit



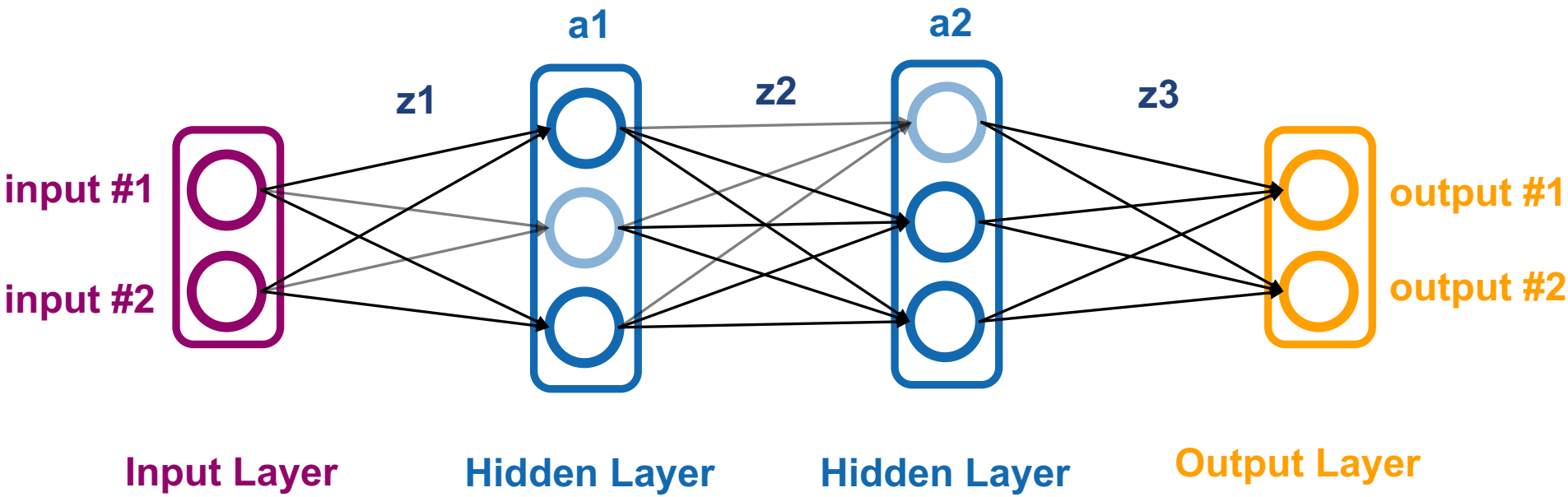
Overfitting

Regularisation: Dropout



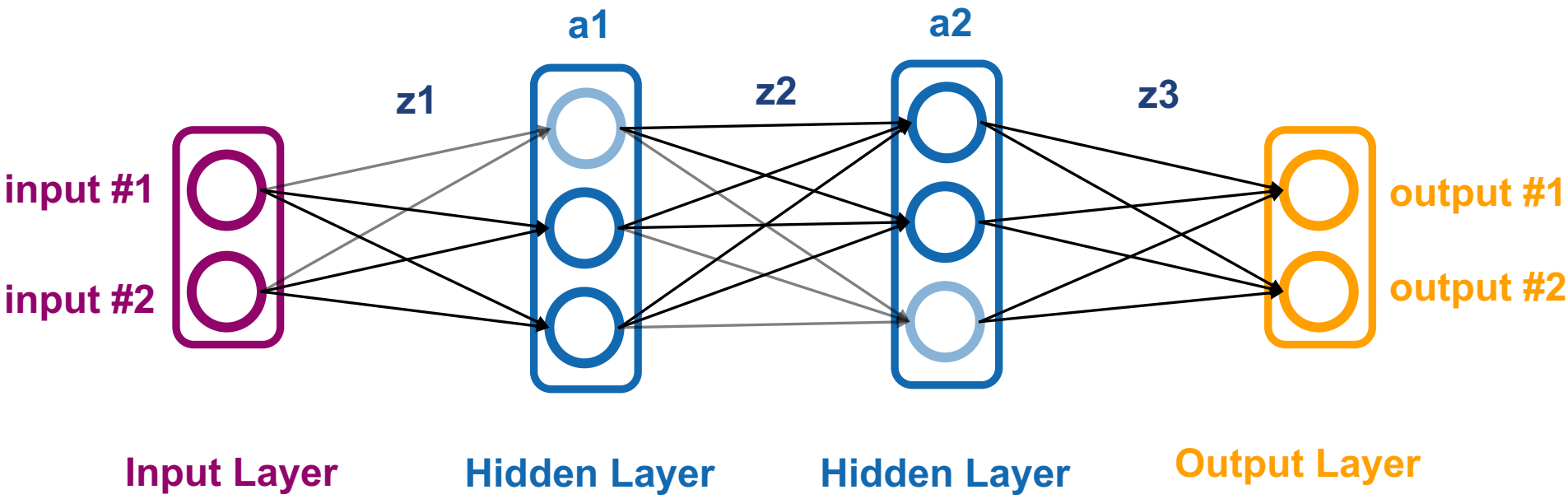
During training set randomly some activations to **0**

Regularisation: Dropout



During training set randomly some activations to **0**

Regularisation: Dropout



During training set randomly some activations to **0**

Regularisation: Early Stopping

Data augmentation

- Introduce transformations not adequately sampled in the training data
  - Geometric: flipping, rotation, shearing, multiple crops

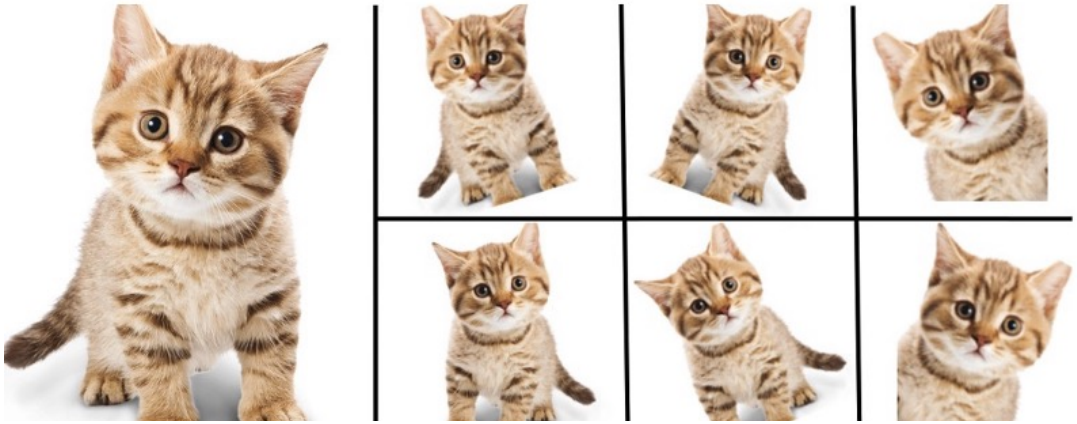


Image source

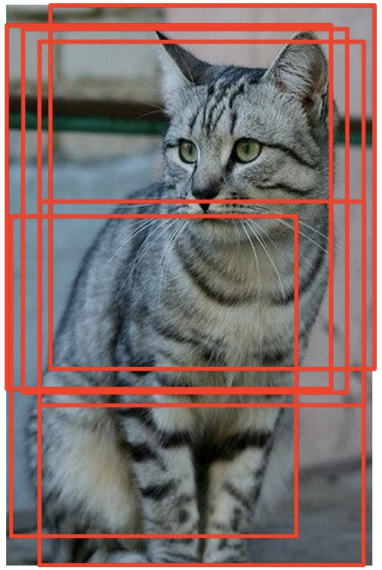


Image source

Data augmentation

- Introduce transformations not adequately sampled in the training data
  - Geometric: flipping, rotation, shearing, multiple crops
  - Photometric: color transformations

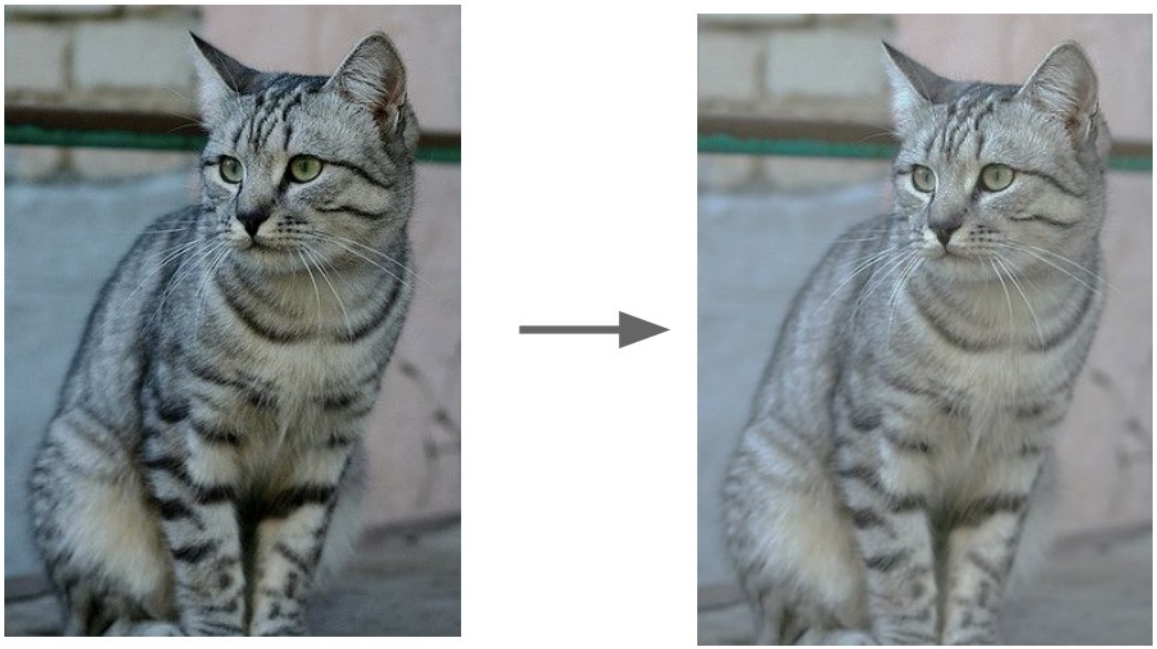
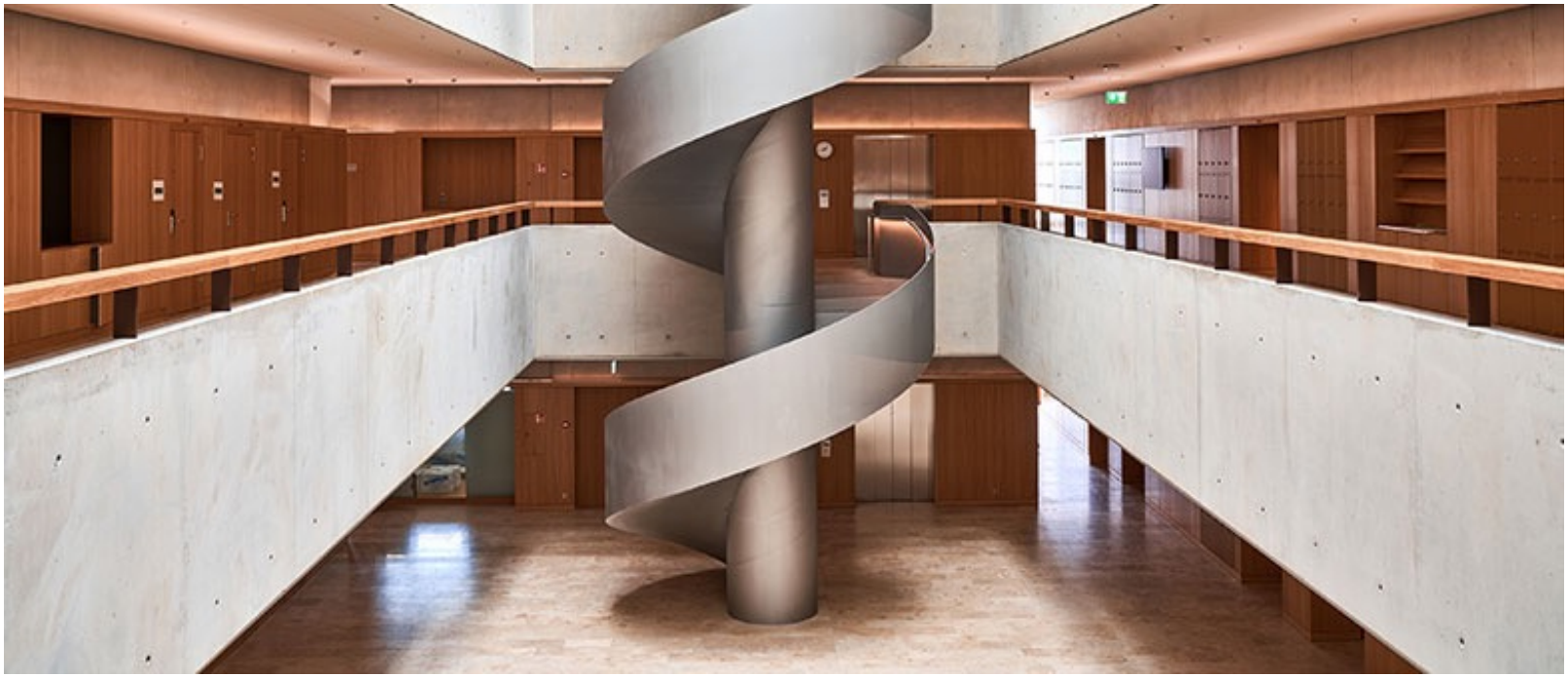


Image source

Data augmentation

- Introduce transformations not adequately sampled in the training data
  - Geometric: flipping, rotation, shearing, multiple crops
  - Photometric: color transformations
  - Other: add noise, compression artifacts, lens distortions, etc.
  - Automatic augmentation strategies: AutoAugment, RandAugment



CAS Machine Learning

Deep Learning: From Core Foundations to Applications

Javier Montoya, Dr. sc. ETH
javier.montoya@hslu.ch