Machine learning principles with applications in physics

A crash course on Machine Learning

Tony Bonnaire

Image generated with Midjourney « A representation of deep learning merging with physics »

5th September, École Normale Supérieure, 2023-2024

LPENS ENS | PSL

Organisation of the course

Given by: Myself and Jorge Fernandez-de-Cossío Díaz.

Format: 3 lectures + projects.

Exam: Oral presentation of your "solution" to the projects.

Aim: Introduce you to the basics of ML through applications in physics.

Quick syllabus:

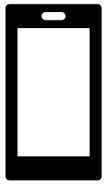
1. **(today 05/09, Tony) Crash course on ML: introduction, basic models, optimisation, generalisation**
2. (next week 12/09, Jorge) More on DL (VAEs, GANs, CNNs)
3. (in two weeks 19/09, Jorge + Tony) Introduction to Python for ML and DL (hands-on with PyTorch)

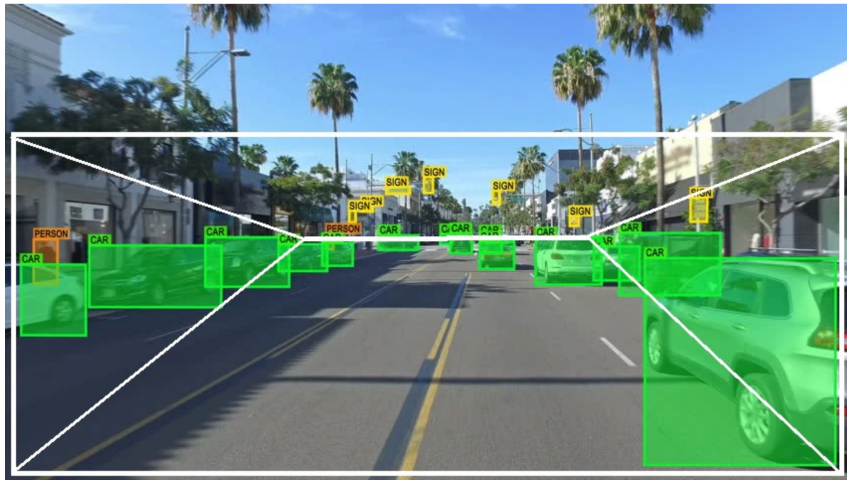
Some references:

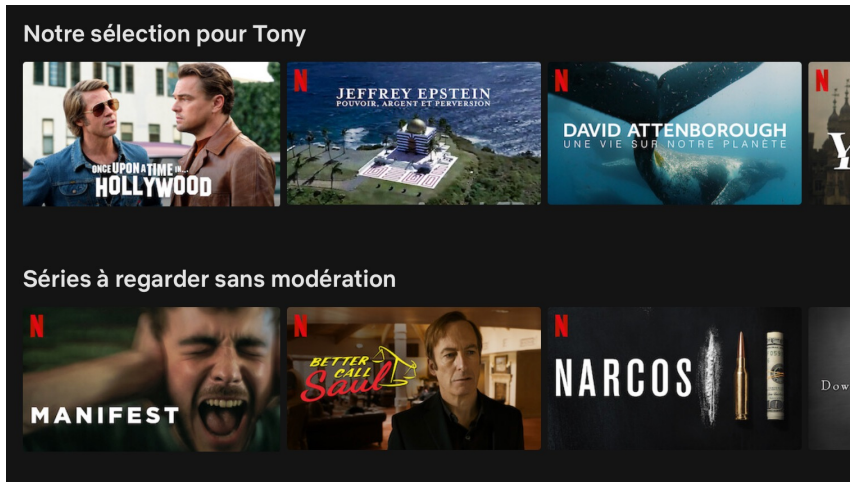
- Pattern recognition and Machine Learning, Cristopher Bishop, 2006
- Deep Learning, Goodfellow, Bengio and Courville, 2016
- https://challengedata.ens.fr: a bank of data science challenges to apply all the things we will learn in this course

Generalities

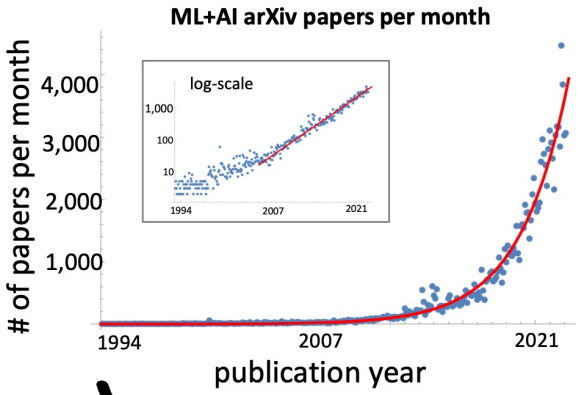
AI, data science and machine learning are now embedded in our daily lives





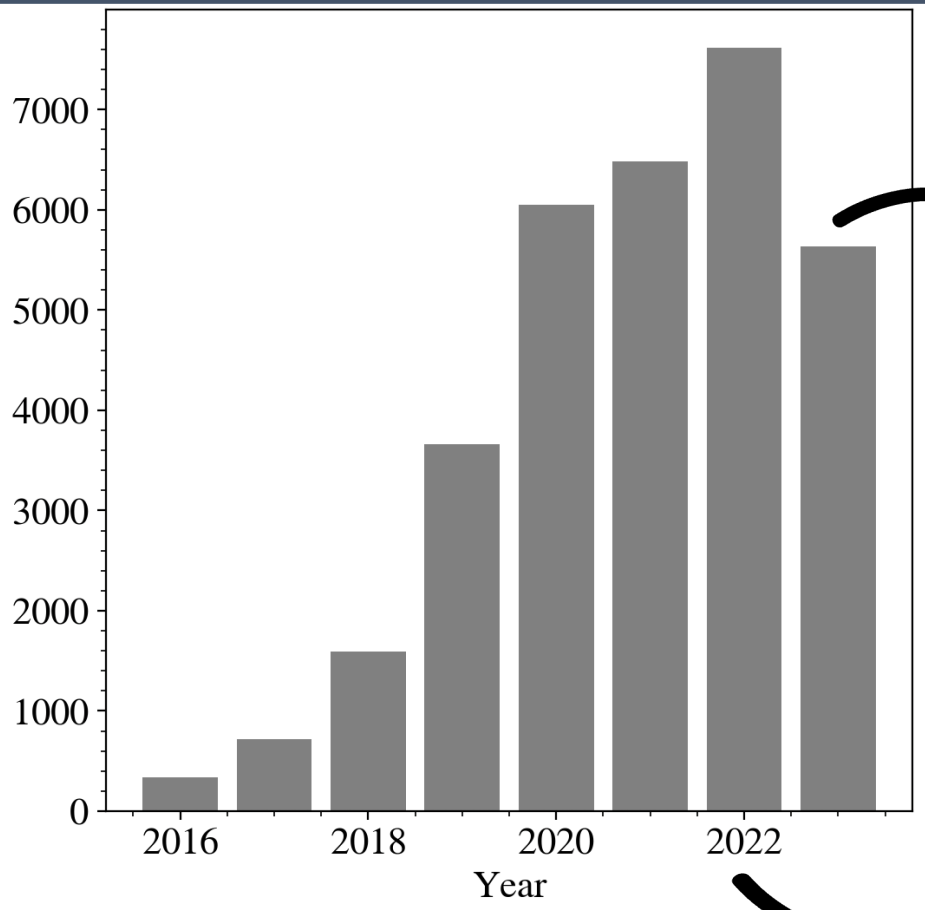


But also in science



Sidenote: ML papers from stat and cs filters grow exponentially fast! (Kren et al., 2022)

arXiv papers query with **physics**, **quantitative biology** and **electrical engineering** filters containing ML, AI or DL in abstract



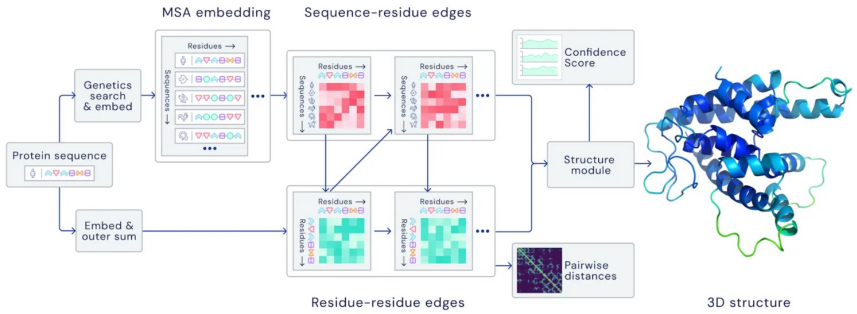
2023 not over yet!

20 papers a day in average in 2022

But also in science

Healthcare

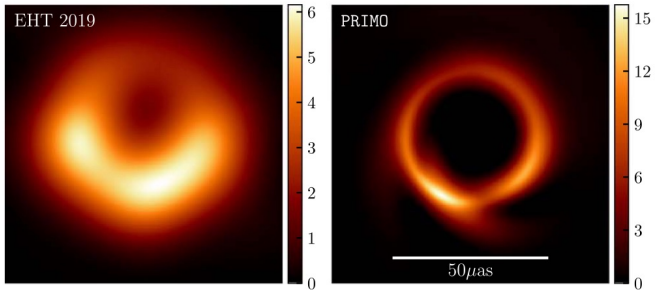
- Drug discovery
- Protein structure reconstruction



Jumper et al., 2021

Astrophysics and cosmology

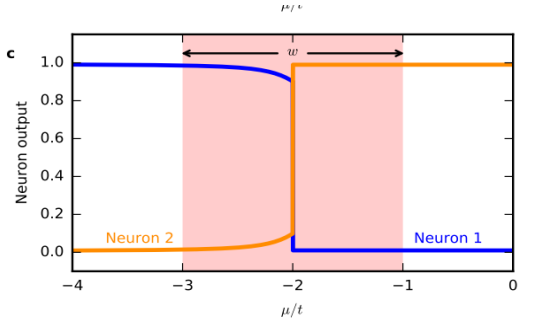
- Galaxy deblending
- Image restoration
- Source separation



Medeiros et al., 2023

Theoretical physics

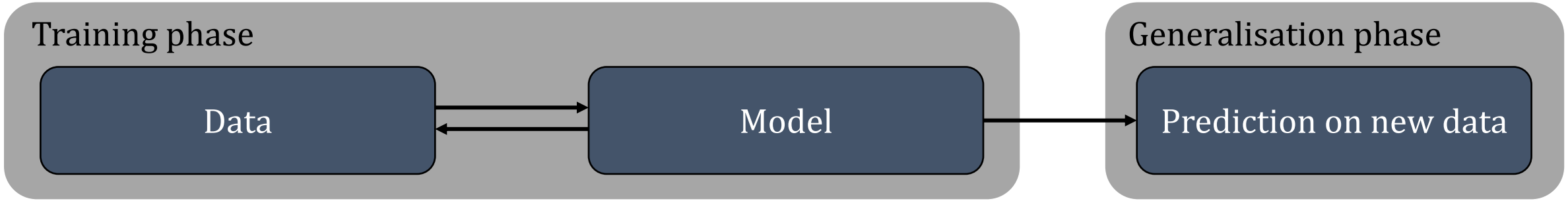
- Study phase transitions
- Discover experiments and equations



Van Nieuwenburg et al., 2017

...

What is “learning”?



Images

Tabular

Sound

Text

....

Random forest

Neural networks

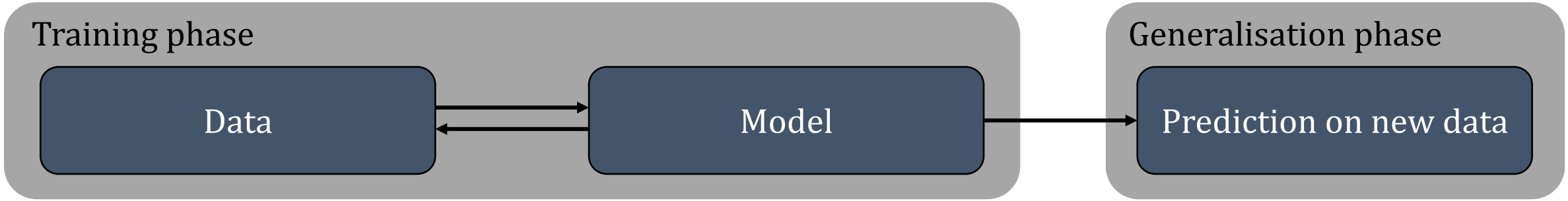
CNN

GAN

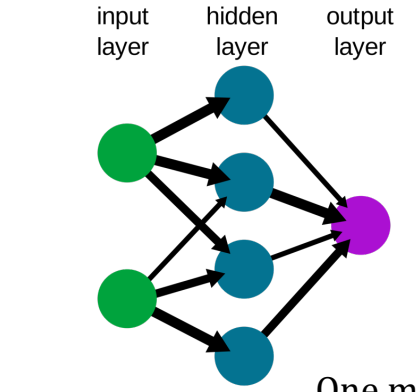
...

All these will be introduced in this set of lectures

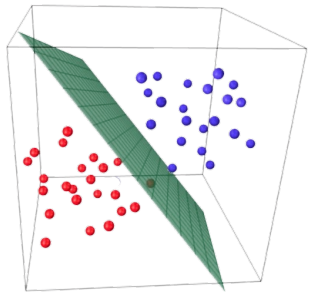
What is “learning”? An example



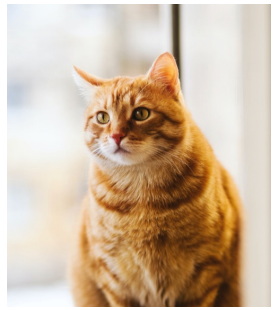
Images of a ”cat” or “dog”



One model

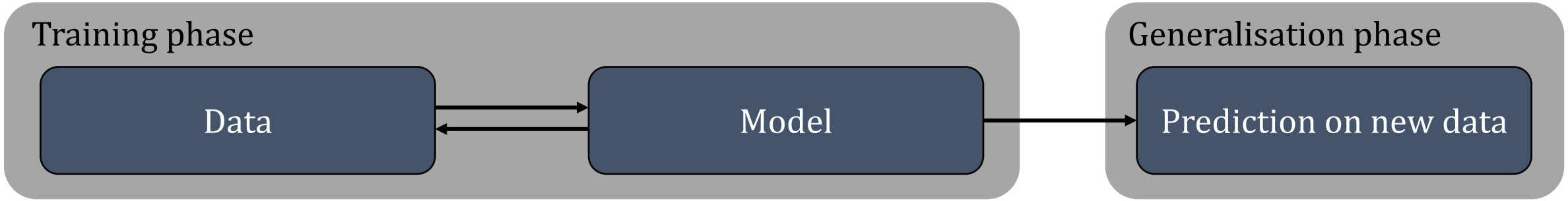


Another model



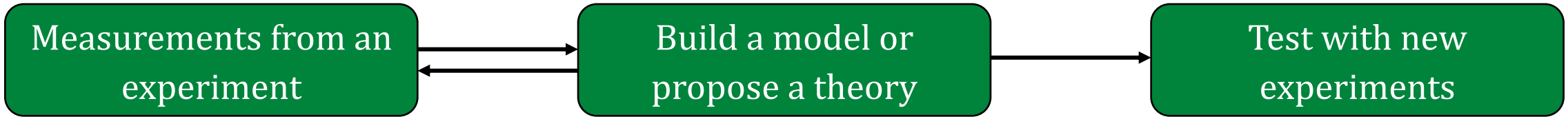
”cat” or “dog” ?

ML and the scientific method



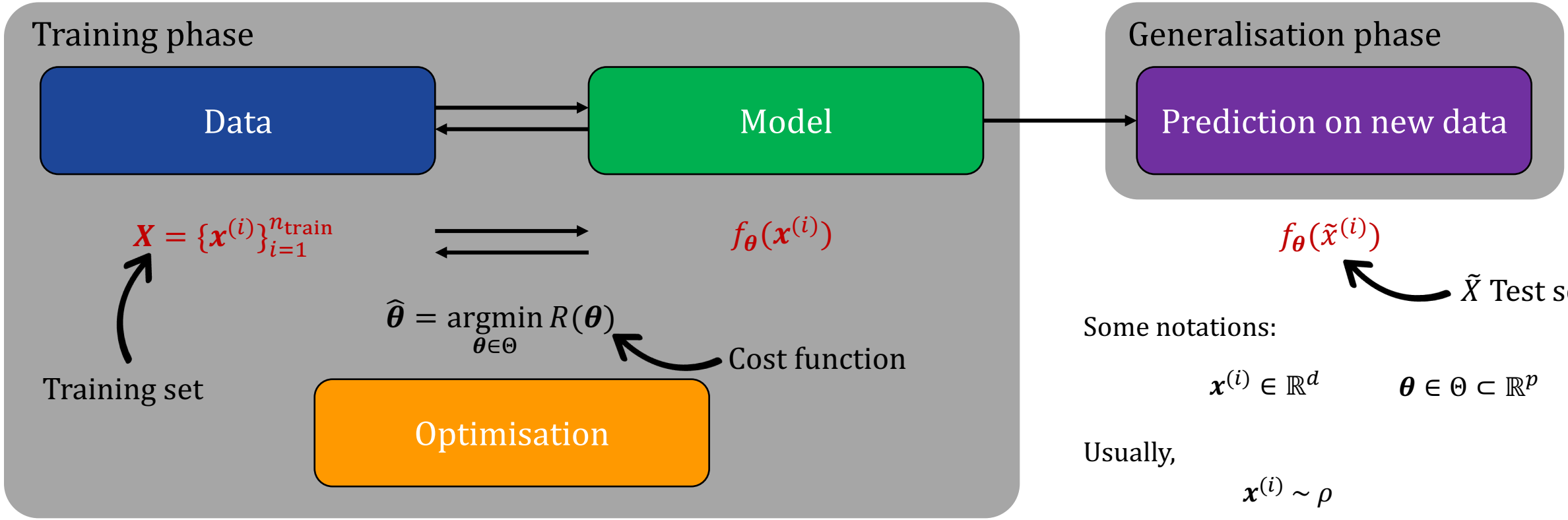
...In fact, all this is close to what you know!

The scientific method



Validate/invalidate your model

ML building blocks



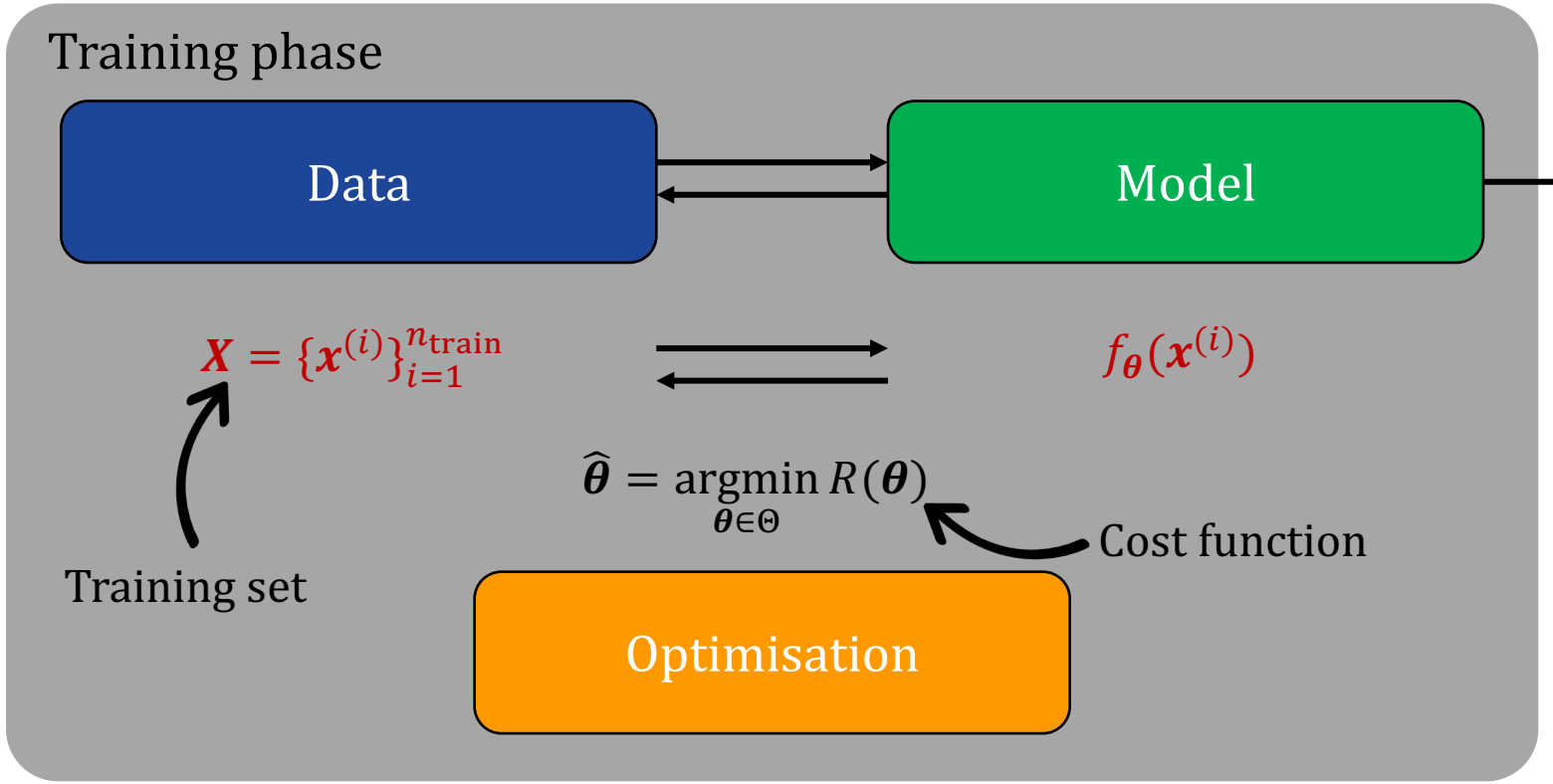
Some notations:

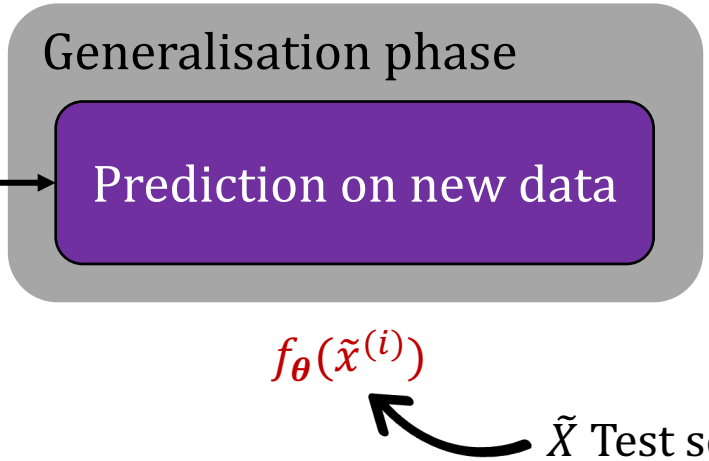
$x^{(i)} \in \mathbb{R}^d$ $\quad$ $\theta \in \Theta \subset \mathbb{R}^p$

Usually,

$x^{(i)} \sim \rho$

While it make take different forms depending on the problem and the model, the **cost function** measures how well (or more precisely *how bad*) your model is performing on the entire training set





Some notations:

$\boldsymbol{x}^{(i)} \in \mathbb{R}^d$ $\quad$ $\boldsymbol{\theta} \in \Theta \subset \mathbb{R}^p$

Usually,

$\boldsymbol{x}^{(i)} \sim \rho$

Three main building blocks of *most* ML systems: some **data** feed a **parametrised model**. The best set of parameters are obtained through the **optimisation** of a cost function. The model can then be assessed on **new data**.

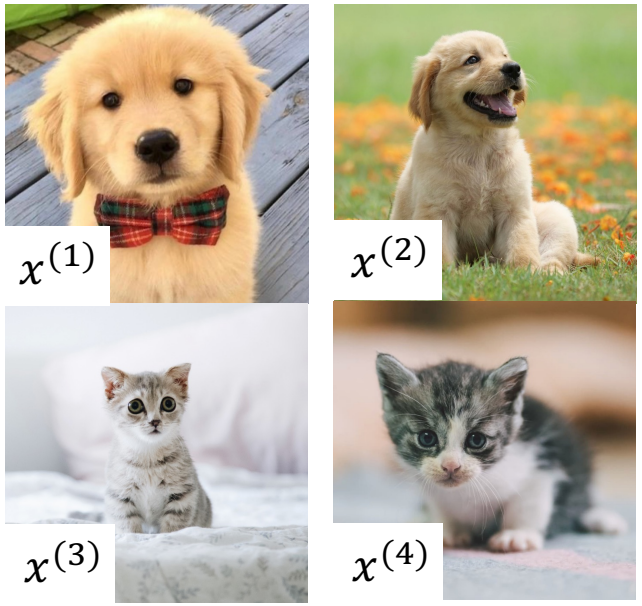
Supervised learning

$\boldsymbol{x}^{(i)}$ is the ith **feature vector** of the training base, and $y^{(i)}$ is called the **target (or predicted) variable**

- Data are actually X and Y coming as pairs

$$\{(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})\}_{i=1}^{n}, \qquad (\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}) \in \mathbb{R}^d \times \mathbb{Y}, \qquad \mathbb{Y} \subset \mathbb{R}^q$$

- Model $f_\theta : X \to Y$ and usually $f_\theta = p(y|x)$

$X =$ 

Aim: Know if an image encodes a cat or a dog (called a classification task)

$Y = \{1, 1, 0, 0\}$

$f_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) = \hat{\boldsymbol{y}}^{(i)}$ ⟶ $f_{\hat{\theta}}$

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, R(Y, \hat{Y}, \boldsymbol{\theta})$$

$$\text{with } R(Y, \hat{Y}, \boldsymbol{\theta}) = \frac{1}{n_{\text{train}}} \sum_{i=1}^{n_{\text{train}}} \ell(y^{(i)}, \hat{y}^{(i)})$$

Loss function: measures how bad your model is on a single example

Data | Model | Optimisation

Families of ML problems

Supervised learning

- Data are actually X and Y coming as pairs

$$\left\{\left(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}\right)\right\}_{i=1}^{n}, \qquad \left(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}\right) \in \mathbb{R}^d \times \mathbb{Y}, \qquad \mathbb{Y} \subset \mathbb{R}^q$$

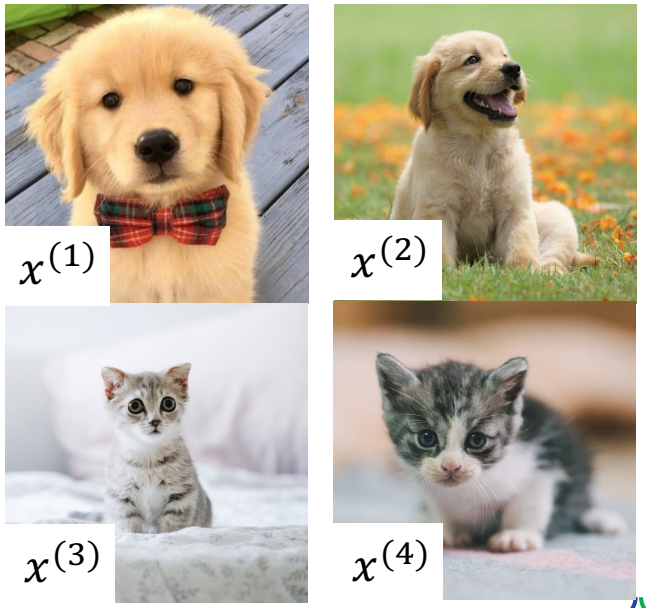
- Model $f_\theta : X \to Y$ and usually $f_\theta = p(y|x)$

Examples of tasks: Classification | Regression | Timeseries prediction | Segmentation

Examples of models: Artificial Neural network | Random forest | Linear regression | Logistic regression | Naïve Bayes | Nearest neighbours

Unsupervised learning

- Training data are the set of $\boldsymbol{x}^{(i)}$'s only; no known results to predict
- In unsupervised learning, one seeks **patterns or structures** in X without prior labels

$X =$ 

Aim: find two relevant classes to separate your dataset (called a clustering task)

$$f_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right) = \hat{y}^{(i)} \longrightarrow f_{\hat{\theta}}$$

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, R(\hat{Y}, \boldsymbol{\theta})$$

Data | Model | Optimisation

Families of ML problems

Unsupervised learning

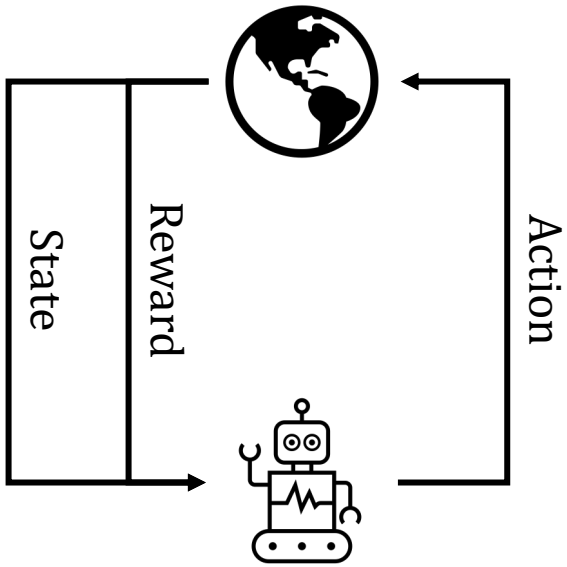
- Training data are the set of $\boldsymbol{x}^{(i)}$'s only; no known results to predict
- In unsupervised learning, one seeks **patterns or structures** in X without prior labels

Examples of tasks: Clustering | Data augmentation | Dimensionality reduction | Sampling

Examples of models: Autoencoder | Boltzmann Machine | Gaussian mixture model | Generative Adversarial network

Reinforcement learning

- The philosophy is different: the model does not try to "imitate" like in supervised learning nor to find patterns but "tries" things
- It is based on an **agent** interacting with an **environment**
- The agent tries to find the best possible sequence of states and actions to **maximise a reward**

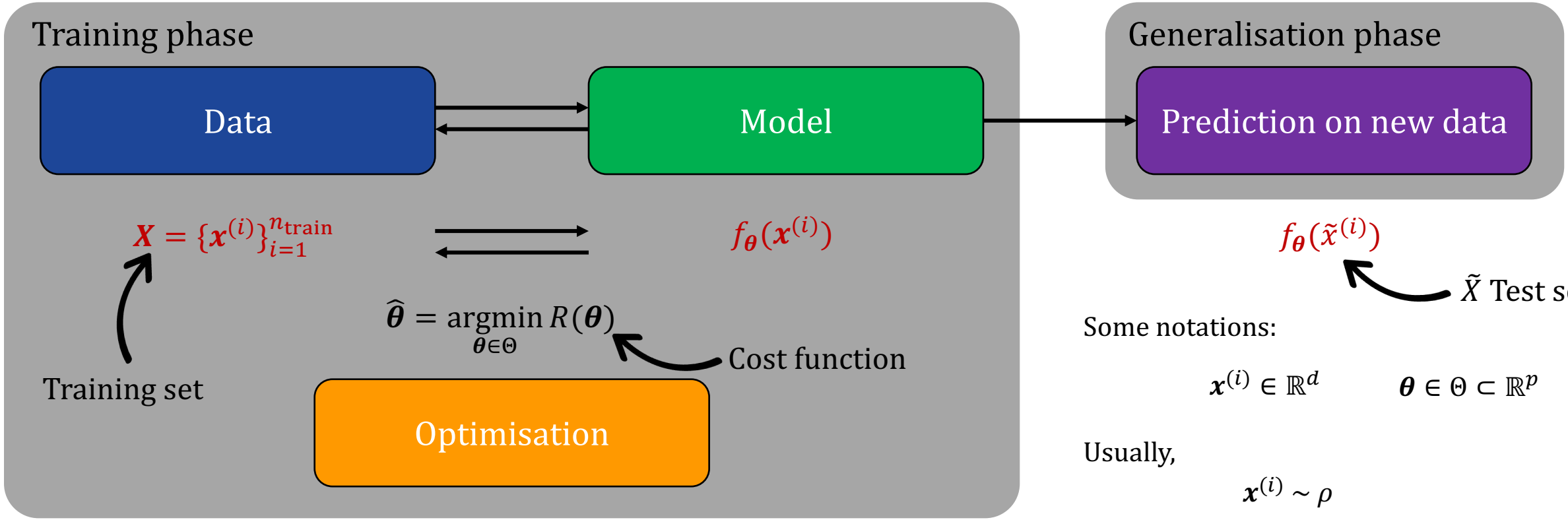


Examples of tasks: Game theory | Robotics | Autonomous driving

Examples of models: Markov decision processes | Q-networks | Deep policy gradient

Not discussed in this course, but a very good reference is Reinforcement Learning – An introduction, Sutton and Barto, 2018

ML building blocks



Some notations:

$x^{(i)} \in \mathbb{R}^d$ $\quad$ $\theta \in \Theta \subset \mathbb{R}^p$

Usually,

$x^{(i)} \sim \rho$

Three main building blocks of *most* ML systems: some **data** feed a **parametrised model**. The best set of parameters are obtained through the **optimisation** of a cost function. The model can then be assessed on **new data**.

The bias-variance trade-off

- The whole aim of training supervised models is to generalise well on unseen data. In practice, we would like to minimise $\mathbb{E}_{\boldsymbol{x},y}\big(\ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}), y)\big)$ that we approximate by $\frac{1}{n_{\text{train}}}\sum_{i=1}^{n_{\text{train}}} \ell\left(\left(f_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right), y^{(i)}\right)\right)$
- In a regression context, suppose there exists f such that $y^{(i)} = f\left(\boldsymbol{x}^{(i)}\right) + \epsilon^{(i)}$, with $\mathbb{E}\left[\epsilon^{(i)}\right] = 0, \mathbb{E}\left[\epsilon^{(i)^2}\right] = \sigma_\epsilon^2$
- We build a model f_θ of f minimising the squared error $\ell\left(\left(f_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right), y^{(i)}\right)\right) = \left(y^{(i)} - f_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right)\right)^2$
- We can show that the expected error on a test example $\tilde{\boldsymbol{x}}$ decomposes as

$$\mathbb{E}\left[\left(y - f_{\boldsymbol{\theta}}(\tilde{\boldsymbol{x}})\right)^2\right] = \mathbb{E}[f_{\boldsymbol{\theta}}(\tilde{\boldsymbol{x}}) - f(\tilde{\boldsymbol{x}})]^2 + \mathbb{E}[(f(\tilde{\boldsymbol{x}}) - \mathbb{E}[f_{\boldsymbol{\theta}}(\tilde{\boldsymbol{x}})])^2] + \sigma_\epsilon^2$$

$$\mathbb{E}\left[\left(y - f_{\boldsymbol{\theta}}(\tilde{\boldsymbol{x}})\right)^2\right] = \text{Bias}[f_{\boldsymbol{\theta}}(\tilde{\boldsymbol{x}})]^2 + \text{Var}[f_{\boldsymbol{\theta}}(\tilde{\boldsymbol{x}})] + \sigma_\epsilon^2$$

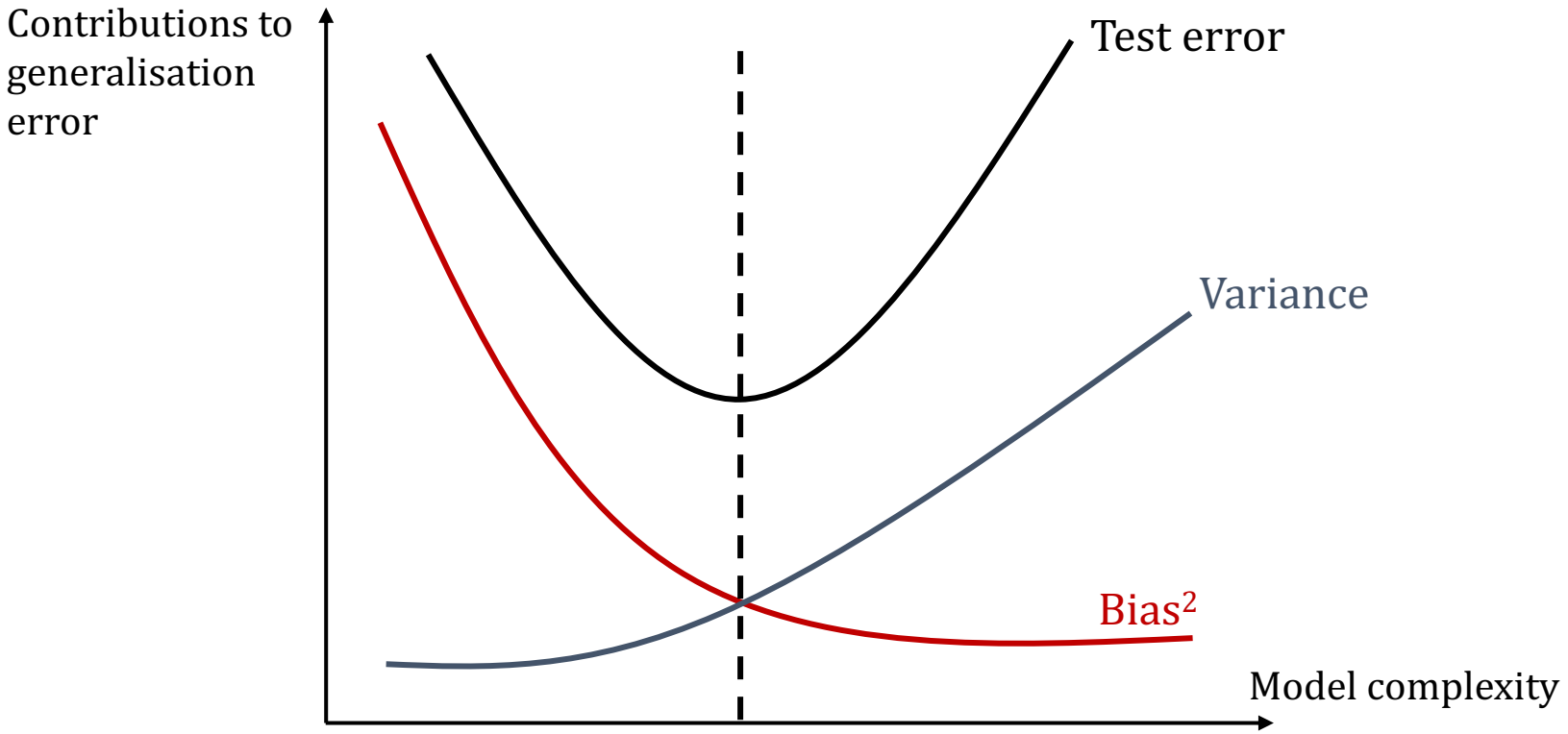
Modelling error | Model variability | Irreducible error

Note: Every terms are >0

- Note that a similar expression holds for classification

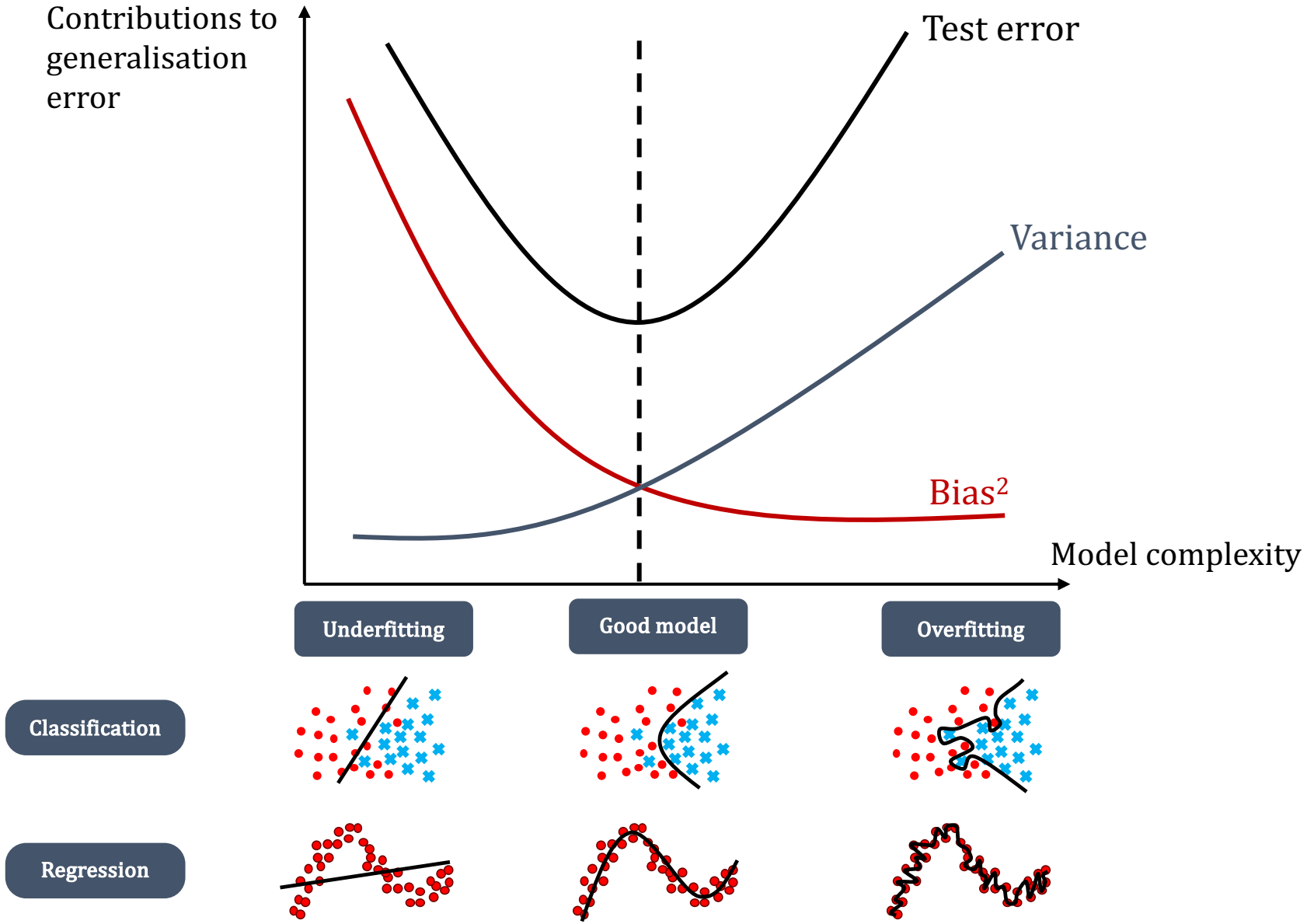
Model complexity and generalisation error

- Usually, **“simple” models have large bias and low variance while “complex” ones have large variance and small bias**



Models need to be built such that they are not too flexible to fit the noise in the data but also not too restrictive to avoid bias

Model complexity and generalisation error



Linear regression

- Example: salary prediction based on the years of experience
- Data are $n = 373$ couples $\left(x^{(i)}, y^{(i)}\right) \Longrightarrow$ **Supervised learning**
- The target variable $y \in \mathbb{R}$ is continuous $\Longrightarrow$ **Regression**
- In linear regression, the model is

$$f_{\boldsymbol{\theta}}\left(x_1^{(i)}\right) = \theta_0 + \theta_1 x_1^{(i)},$$

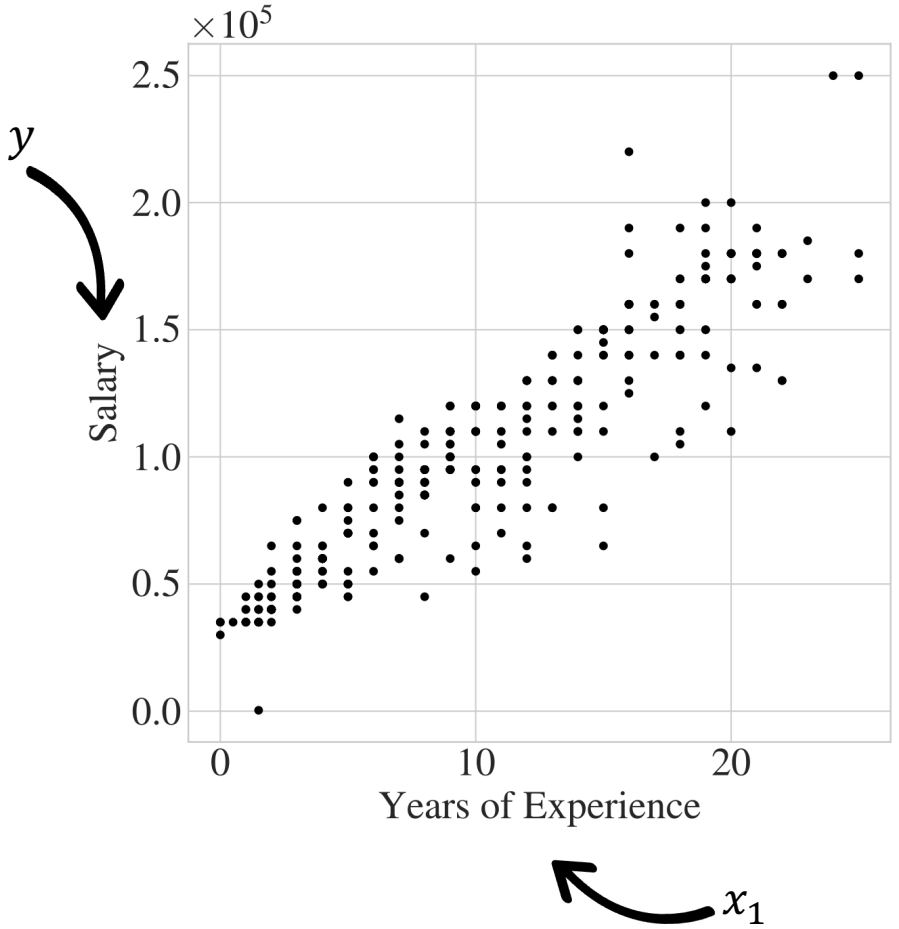
where $x_1^{(i)}$ is the nb. of years of experience of the i^{th} training example

- It can be extended to additional features and functions of the features

$$f_{\boldsymbol{\theta}}\left(x_1^{(i)}, x_2^{(i)}, \cdots\right) = \theta_0 + \theta_1 x_1^{(1)} + \theta_2 x_2^{(i)} + \cdots$$

$$f_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right) = \boldsymbol{\theta}^{\mathrm{T}} \boldsymbol{x}^{(i)},$$

where $\boldsymbol{\theta} = [\theta_0, \theta_1, \theta_2, \cdots]^{\mathrm{T}}$, $\boldsymbol{x}^{(i)} = \left[1, x_1^{(i)}, x_2^{(i)}, \cdots\right]^{\mathrm{T}}$

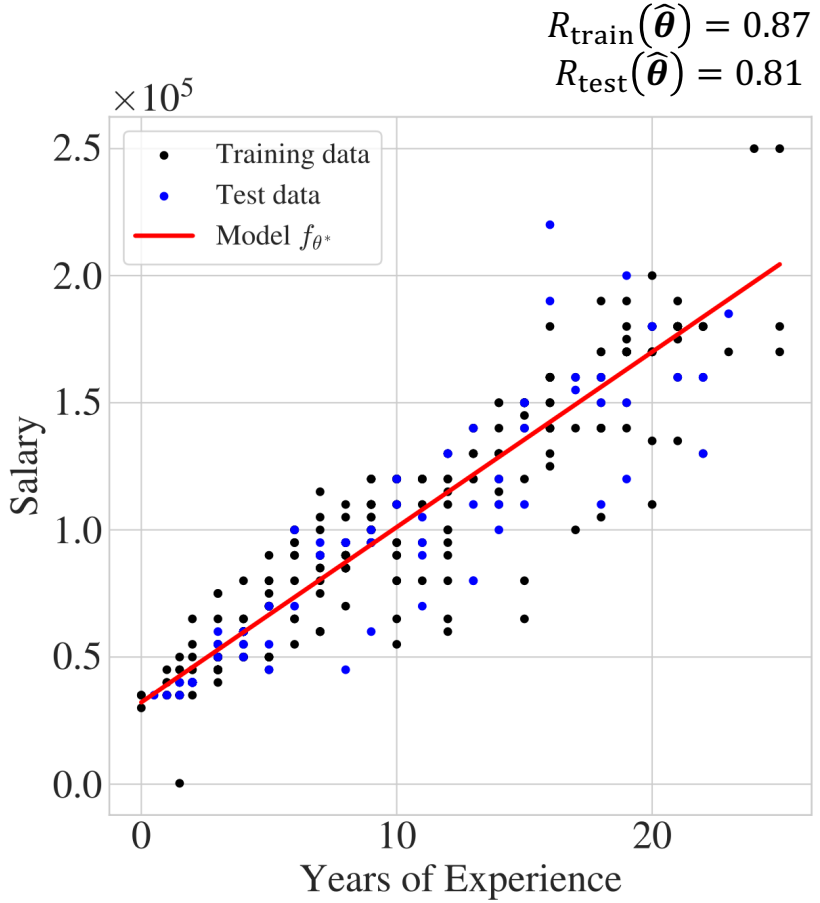


Linear regression

- Now the model is fixed, how to find $\widehat{\boldsymbol{\theta}}$, the best possible parameters for our model and data?
- A natural way is assuming a **squared loss function**

$$\widehat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, R(\boldsymbol{\theta}) = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, \frac{1}{n}\sum_{i=1}^{n}\left(\hat{y}^{(i)} - y^{(i)}\right)^2$$

- Note that this is the maximum likelihood estimator, assuming Gaussian error between $\hat{y}^{(i)} = f_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right)$ and y
- ***Here***, the optimisation problem can be solved analytically in closed-form giving (usually not the case): $\widehat{\boldsymbol{\theta}} = \left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\right)^{-1}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{y}$



I separated the dataset into **training and test sets**, $n_{\text{train}} = 0.8n$ and $n_{\text{test}} = 0.2n$ chosen randomly

\+ Exactly solvable, low variance

\− Cannot represent local relationships, may be biased

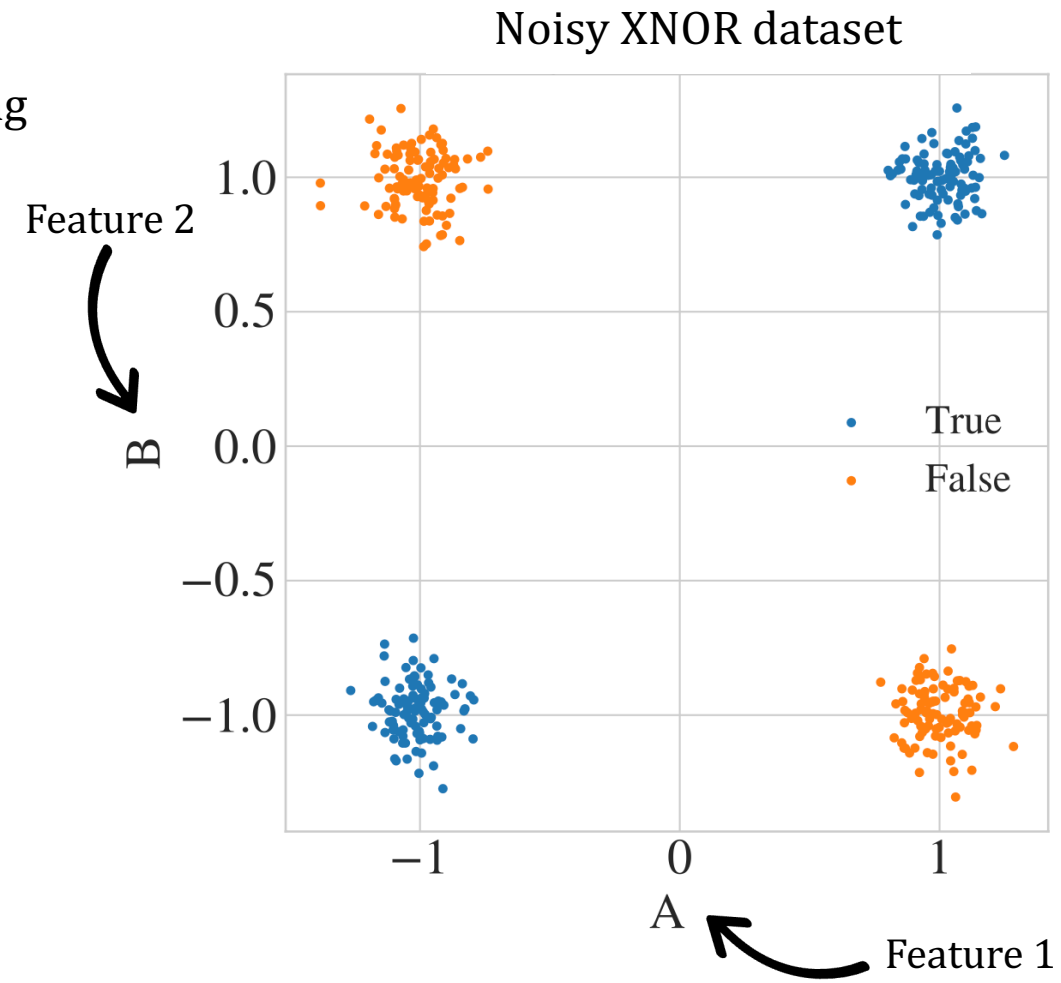
A classification task: the XOR dataset

Decision trees

- Consider a **classification task** on an artificial dataset replicating the XNOR function

A	B	XNOR
True	True	True
True	False	False
False	True	False
False	False	True

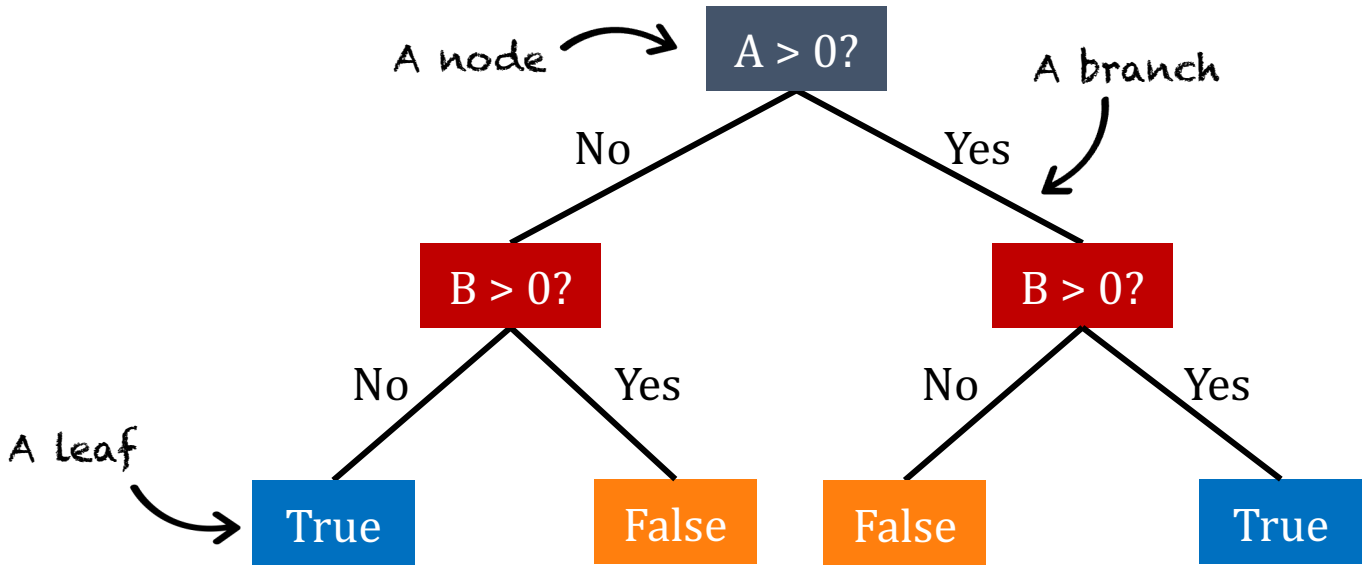
- Data are $n = 500$ couples $\left(\boldsymbol{x}^{(i)}, y^{(i)}\right) \Longrightarrow$ **Supervised learning**
- The target variable $y \in \{-1,1\}$ is discrete $\Longrightarrow$ **Classification**
- A linear classification would not be able to learn such a function*
- Decision trees to the rescue!

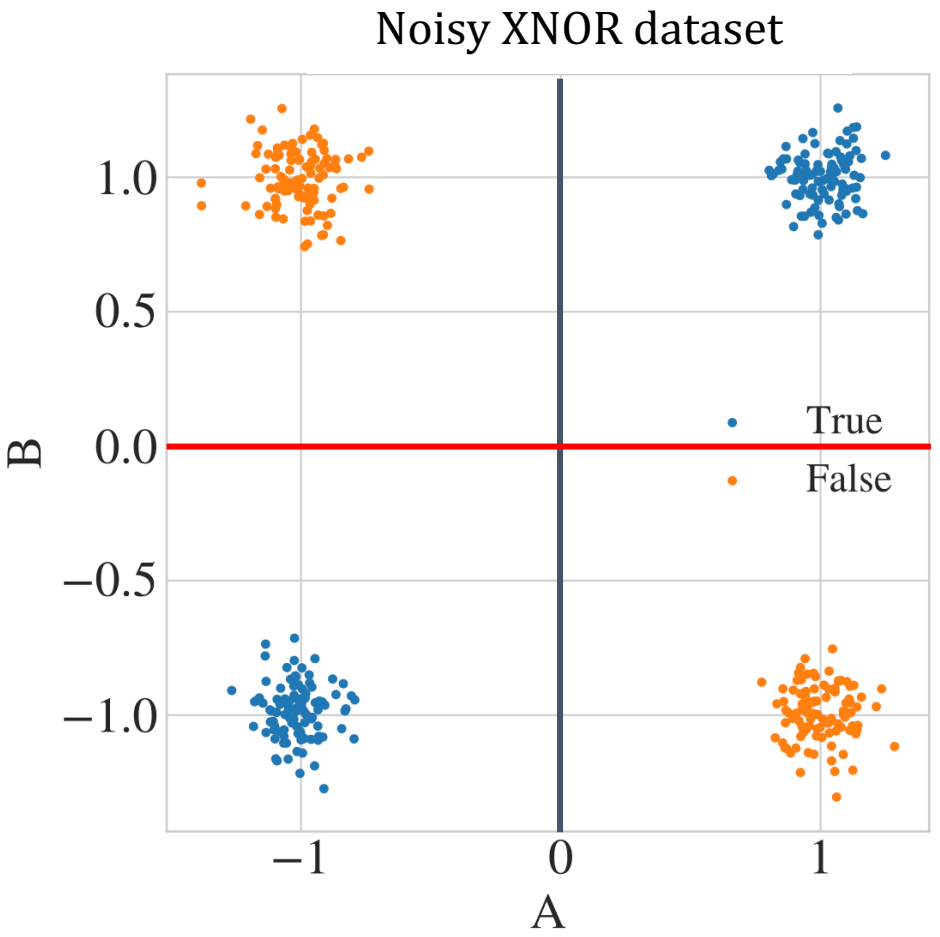


*In fact, an alternative (not discussed here) would be to use **kernels** to find a higher dimensional space in which the data separates linearly and then use a linear classifier.

Decision trees

- Decision trees incrementally ask questions about the features





- All root and inner nodes question the value of a feature, and branches split the dataset into **different regions to which a datapoint can belong uniquely**

A more formal view of decision trees

Decision trees

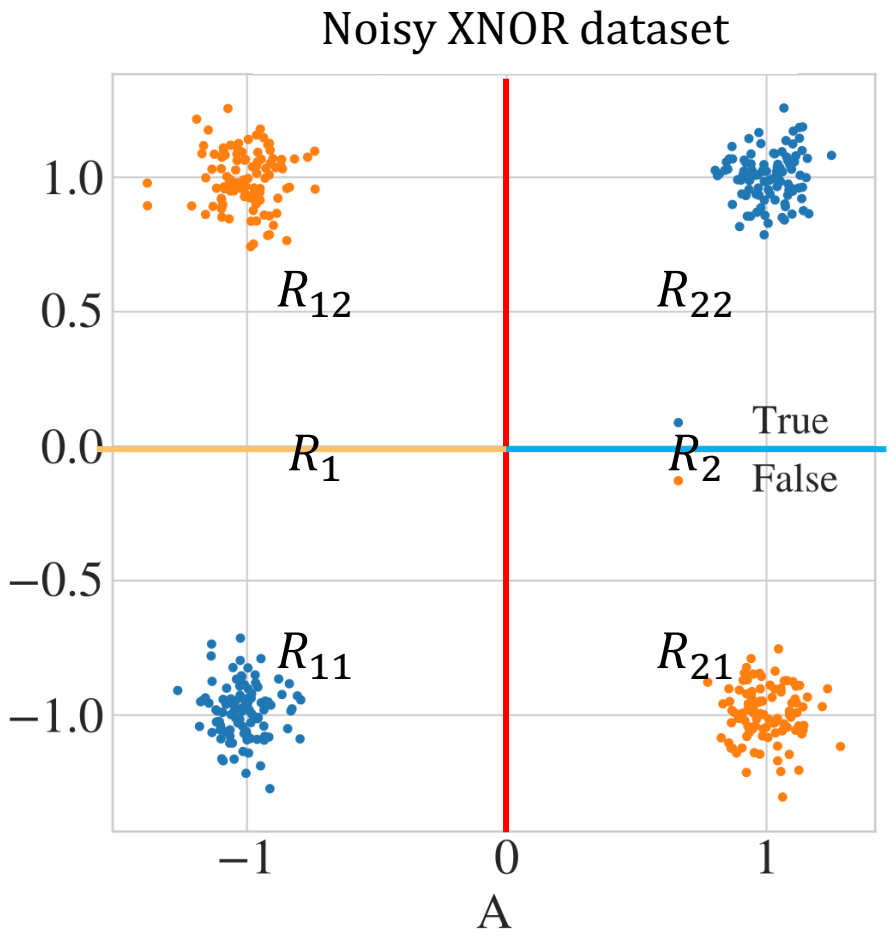
- More formally, at a given node in parent region R_t asking a question about the j^{th} feature, we create two regions :

$$R_1 = \{\boldsymbol{x} \mid x_j < \alpha_{t1}, \boldsymbol{x} \in R_t\}$$

$$R_2 = \{\boldsymbol{x} \mid x_j \geq \alpha_{t2}, \boldsymbol{x} \in R_t\}$$

- The parameters $\boldsymbol{\theta}$ of decision trees are the threshold values at each nodes (the sequence of α)
- To find the optimal values, decision tree algorithms minimise a loss function in each region, and **a good choice is the cross-entropy**

$$L(R_i) = -\sum_{k=1}^{q} \rho_k^i \log_2 \rho_k^i, \qquad \rho_k^i = \frac{\left|\{\boldsymbol{x}^{(j)} \mid \boldsymbol{x}^{(j)} \in R_i,\ y^{(j)} = k\}\right|}{\left|\{\boldsymbol{x}^{(j)} \mid \boldsymbol{x}^{(j)} \in R_i\}\right|}$$



+ can handle categorical values, easy to interpret, fast to compute, both regression and classification

− Shallow trees: weak learners (underfit) and high bias, deep trees: high variance estimators (overfit)

Decision trees

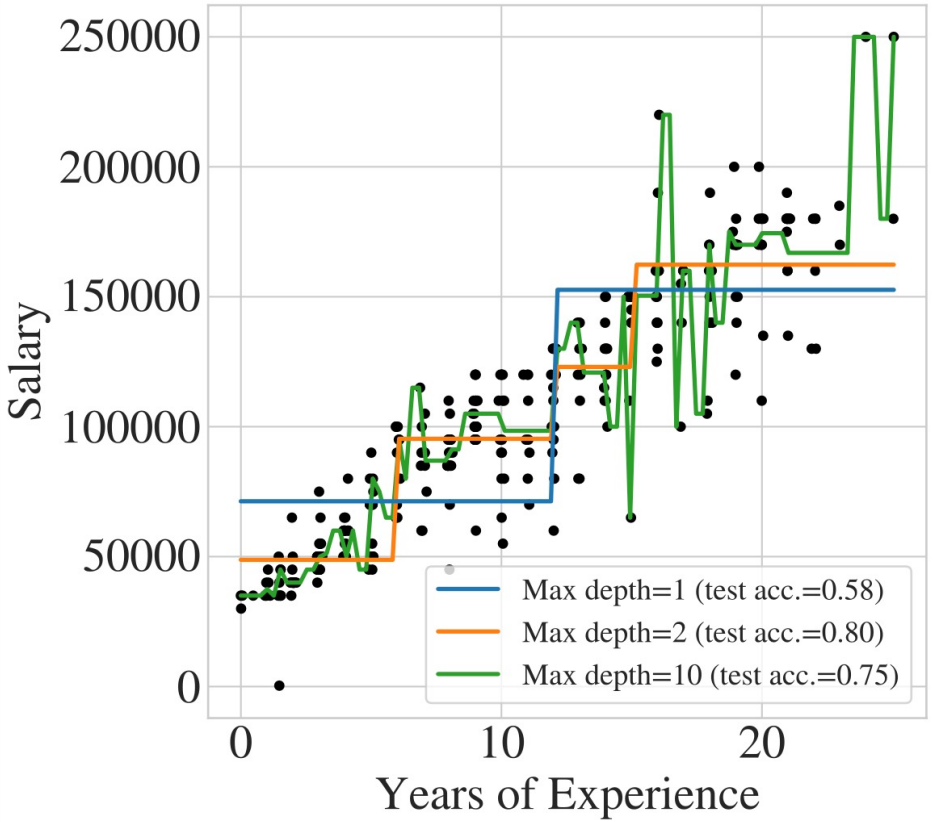
- More formally, at a given node in parent region R_t asking a question about the j^{th} feature, we create two regions :

$$R_1 = \{\boldsymbol{x} \mid x_j < \alpha_{t1}, \boldsymbol{x} \in R_t\}$$

$$R_2 = \{\boldsymbol{x} \mid x_j \geq \alpha_{t2}, \boldsymbol{x} \in R_t\}$$

- The parameters $\boldsymbol{\theta}$ of decision trees are the threshold values at each nodes (the sequence of α)
- To find the optimal values, decision tree algorithms minimise a loss function in each region, and **a good choice is the cross-entropy**

$$L(R_i) = -\sum_{k=1}^{q} \rho_k^i \log_2 \rho_k^i \,, \qquad \rho_k^i = \frac{\left|\{\boldsymbol{x}^{(j)} \mid \boldsymbol{x}^{(j)} \in R_i,\ y^{(j)} = k\}\right|}{\left|\{\boldsymbol{x}^{(j)} \mid \boldsymbol{x}^{(j)} \in R_i\}\right|}$$

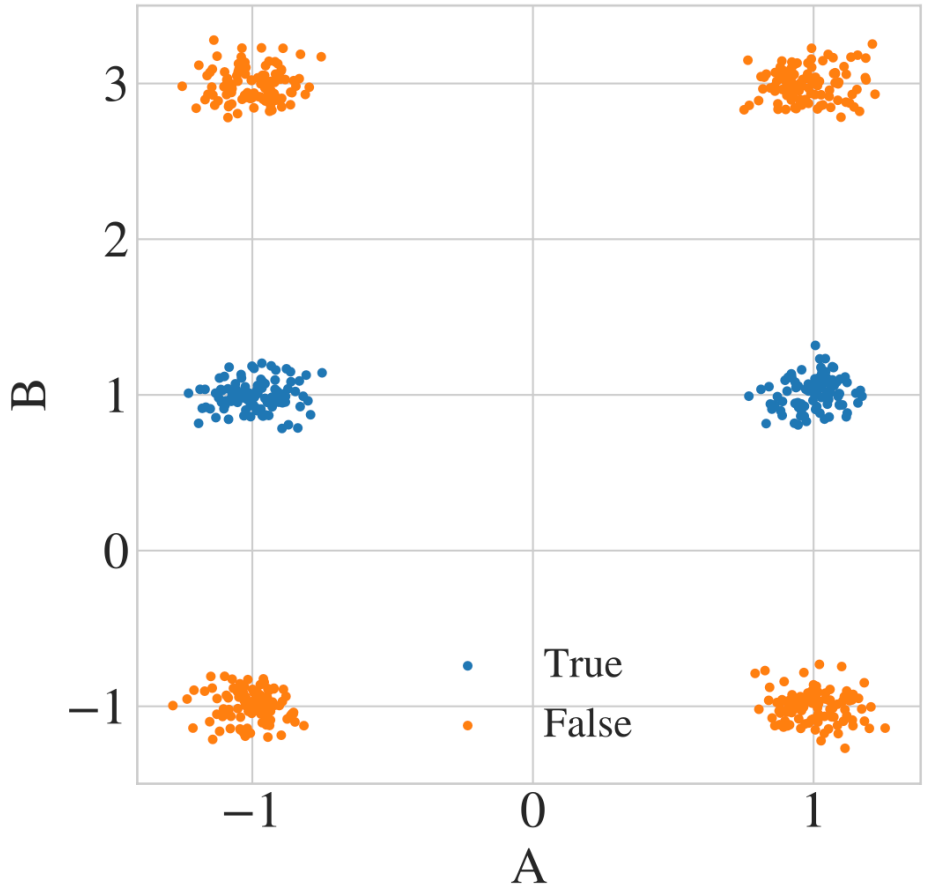


+ can handle categorical values, easy to interpret, fast to compute, both **regression** and classification

− Shallow trees: weak learners (underfit) and high bias, deep trees: high variance estimators (overfit)

A more formal view of decision trees

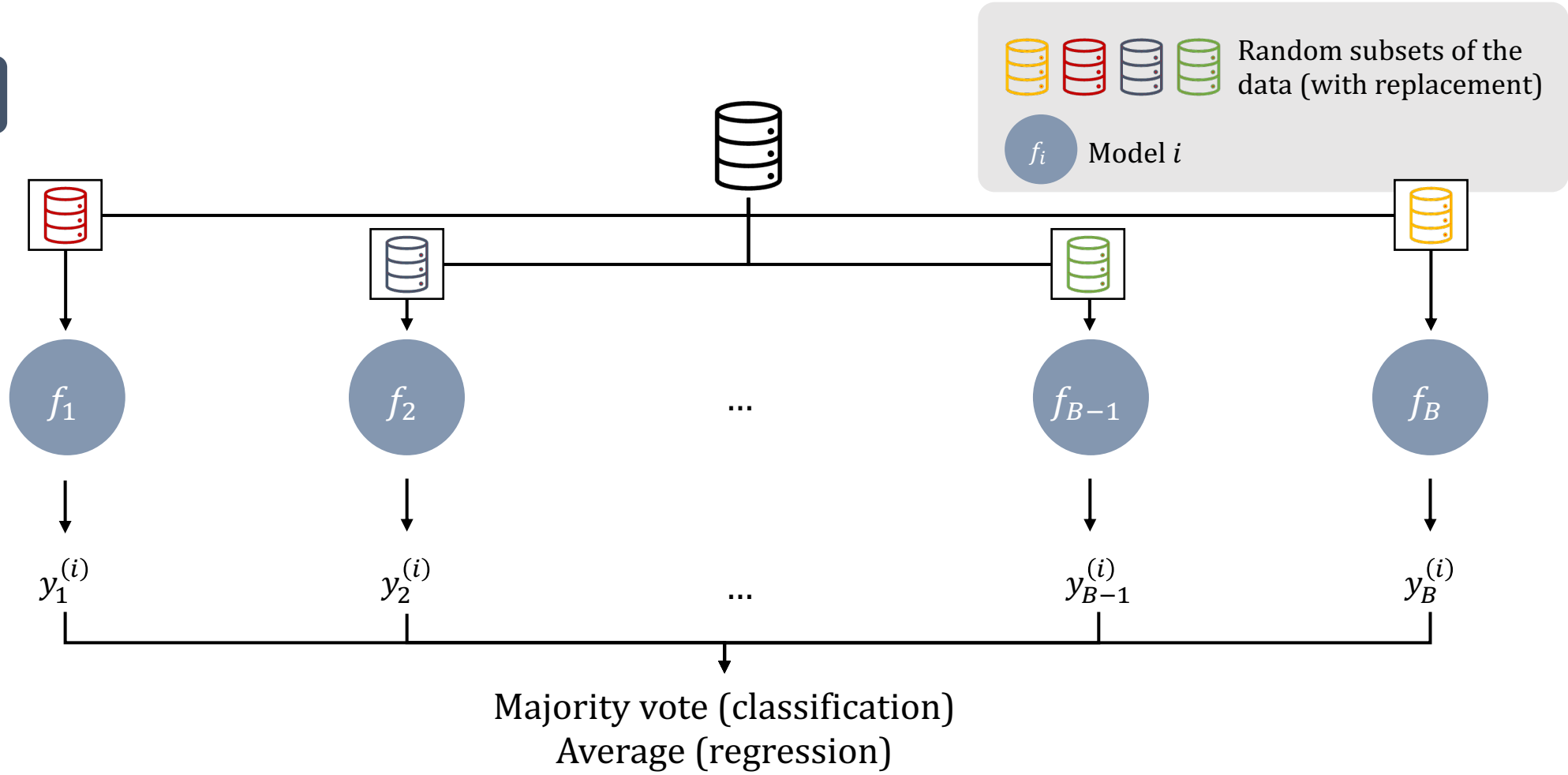
Decision trees



Practice: Can you build a decision tree solving the binary classification problem?

- To circumvent the problems that can have weak learners like decision trees, **ensembling methods** were proposed

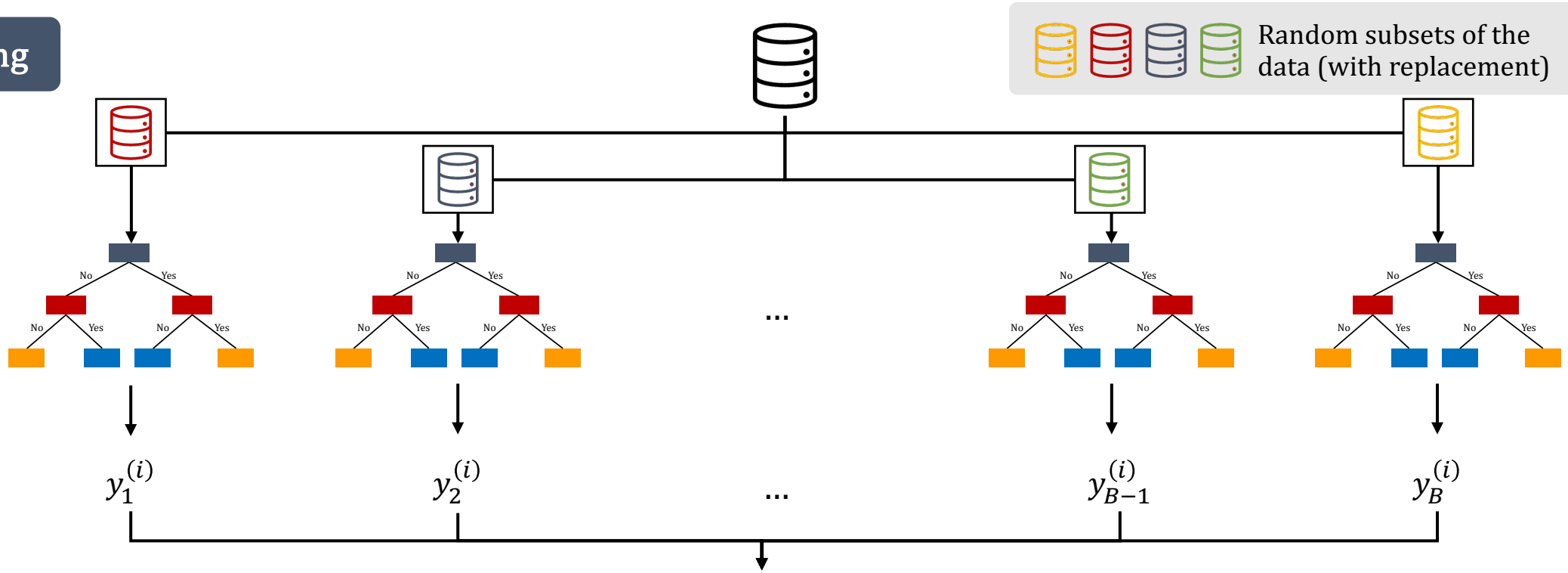
Bagging



- To reduce the variance, models need to be **uncorrelated**: this is achieved by using **random sampling of the dataset**

Ensembling methods: random forest

Bagging



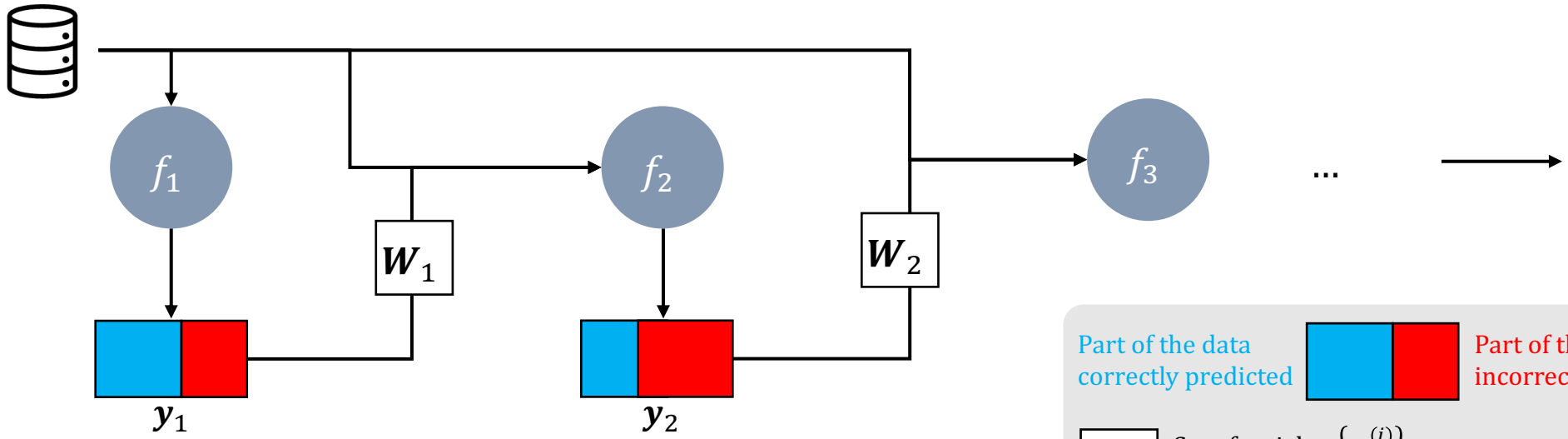
Random Forest = Bagging of decision trees + feature bagging

\+ Can still handle categorical values, both regression and classification, **reduced variance**

− More expansive to compute (need to train B trees instead of one), harder to interpret

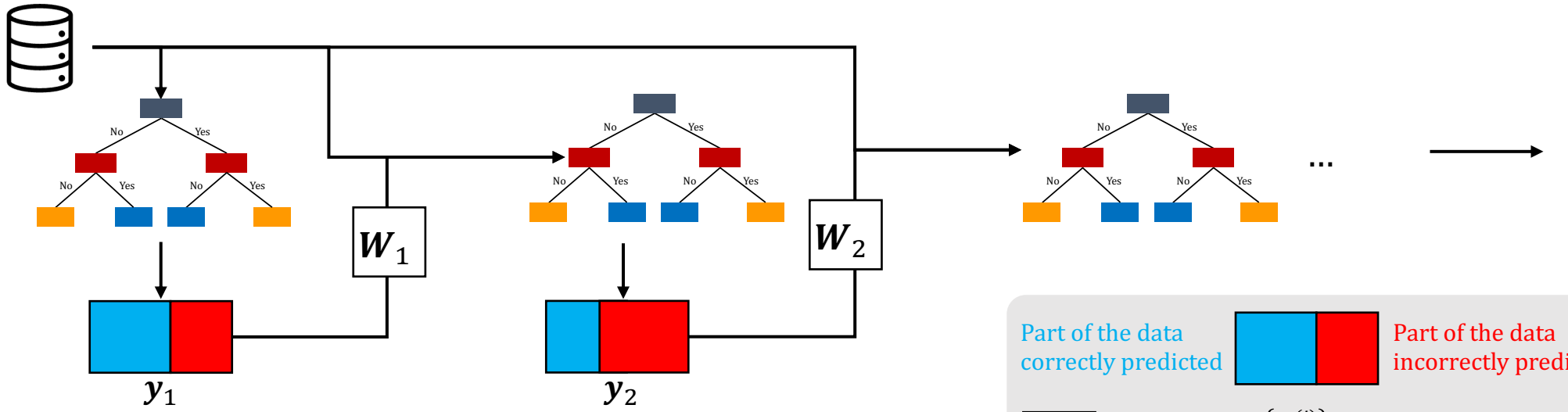
- While bagging trains high-variance models in parallel to reduce the variance of the combined estimate, **boosting trains high-bias models sequentially to reduce the overall bias**

Boosting



- Successive learners f_i are fed by data $\boldsymbol{X}_i$, a weighted version of the initial dataset $\boldsymbol{X}$, **giving more weights to the errors committed by the previous model $\boldsymbol{f_{i-1}}$**
- The output is, as in bagging, a linear combination of all the learners
- The choice of weighting and training depends on the algorithm and context (see Adaboost or gradient boosting)

Boosting



Part of the data correctly predicted

Part of the data incorrectly predicted

W_i Set of weights $\left\{w_j^{(i)}\right\}_{j=1,\ldots,n_{\text{train}}}$ computed from the mistakes made by the model $i-1$

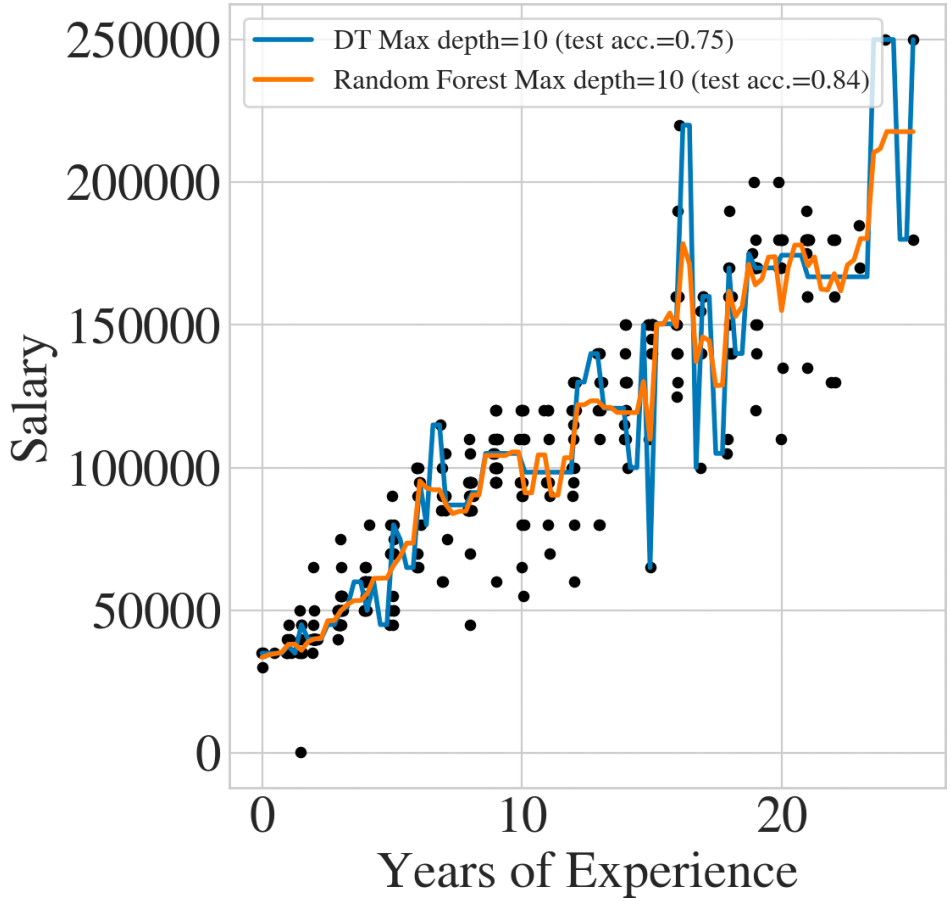
Boosted trees = Boosting of decision trees

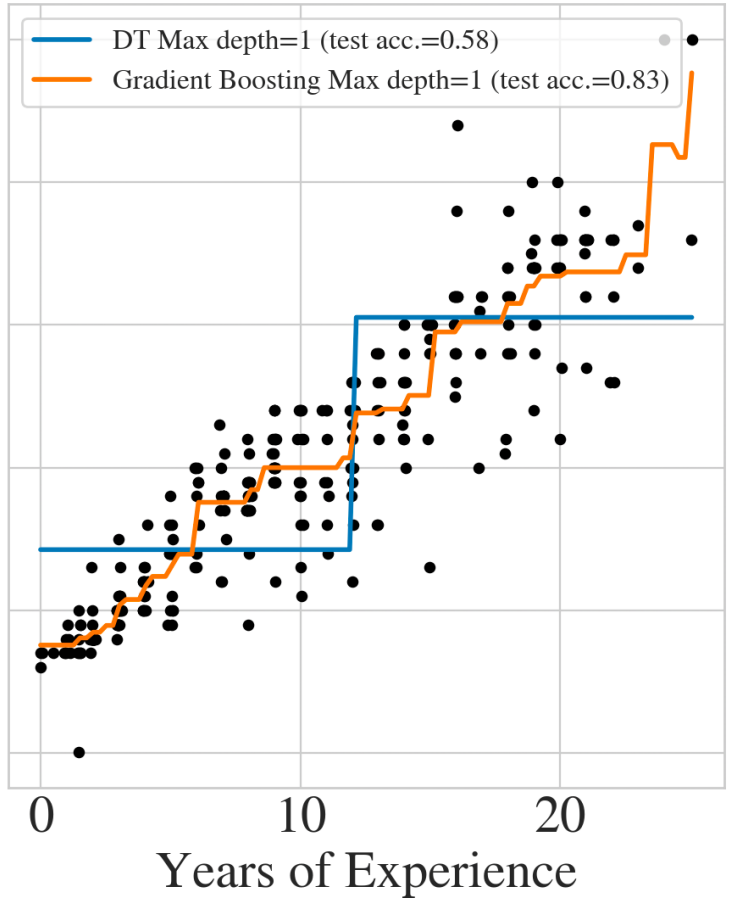
\+ Both regression and classification (residuals or weighted classification error), **reduced bias**, good performances

− More expansive to compute, increased variance, subject to overfitting

Comparison between DT, RF and GB

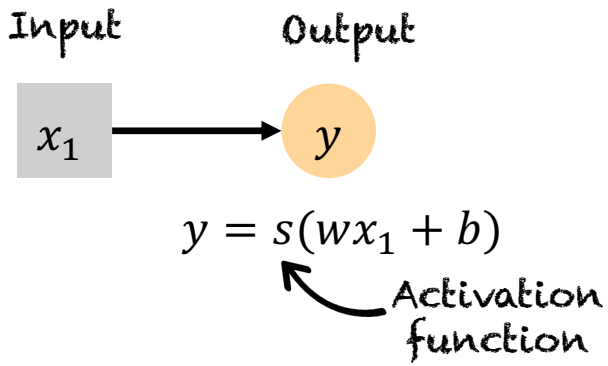
Comparison on our regression problem



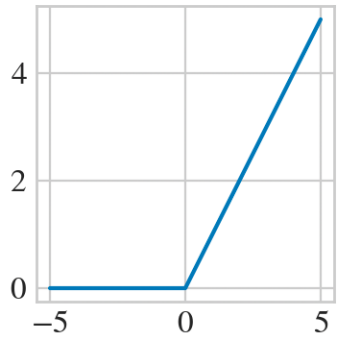


Artificial neural networks

- Building block of neural networks: the **neuron**
- Made of three operations: it first multiplies the input by a **weight**, then adds a **bias**, and finally applies an **activation function** s to the result
- If s is the identity, you recognize the linear regression model with $\theta_0 = b$ and $\theta_1 = w$. To represent non-linear functions, s must be non-linear

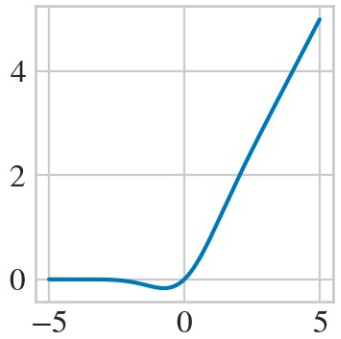


ReLU



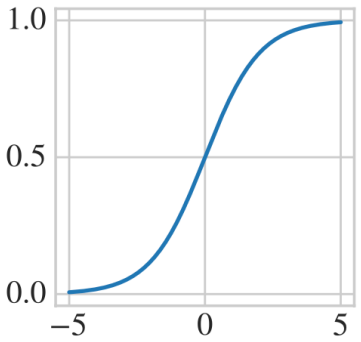
$s(x) = \max(0, x)$

GeLU



$s(x) = \frac{x}{2}\left(1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right)$

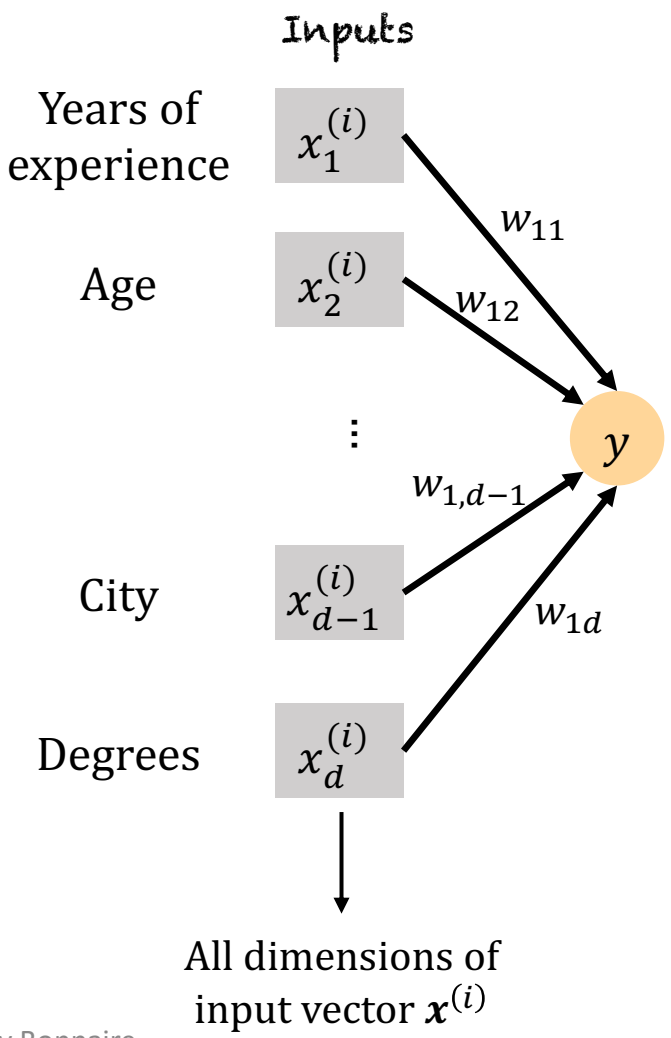
Sigmoid



$s(x) = \frac{1}{1 + \exp(-x)}$

- This model is motivated by biological neurons and the activation function mimics the activation or inhibition through **non-linear (and differentiable) functions**
- Can take different forms but some examples include the **ReLU or sigmoid functions**

Artificial neural networks



Notation: w_{ij} is the weight linking the unit j of the first layer to the unit i of the next layer

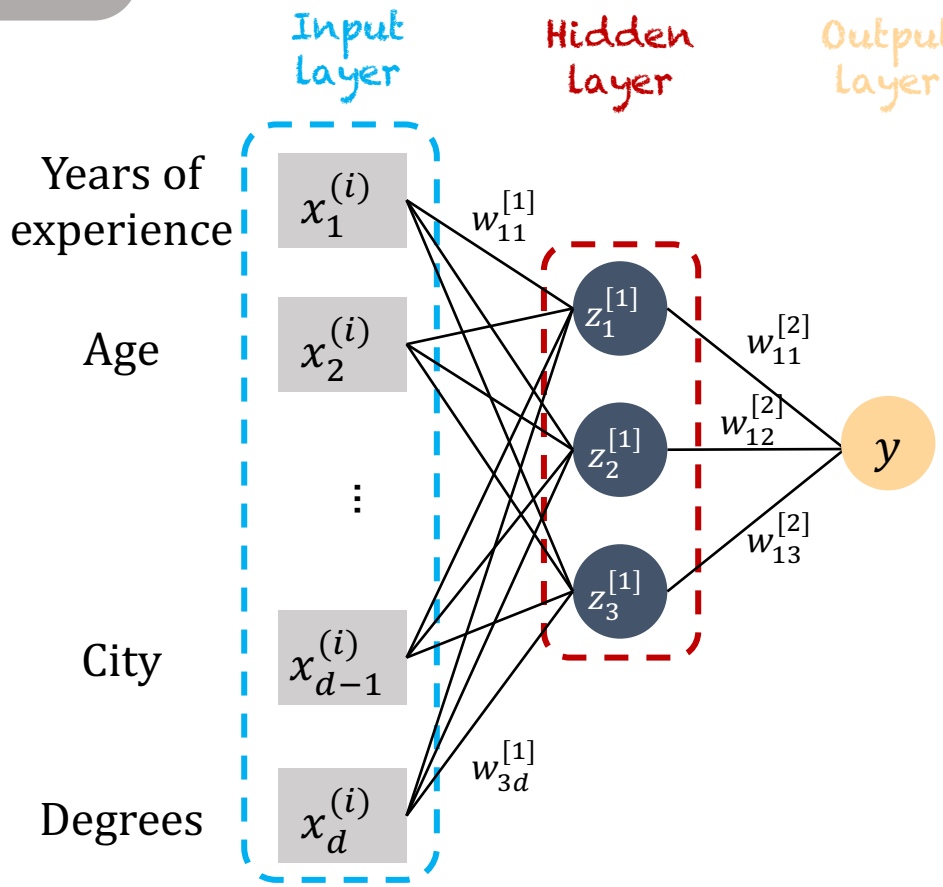
Output

$$y = s\left(\sum_{j=1}^{d} w_{1j} x_j^{(i)} + b\right)$$

$$y = s\left(\boldsymbol{w}_1 \cdot \boldsymbol{x}^{(i)} + b\right)$$

Such a network has $p = d + 1$ parameters (as in multi-feature linear regression)

Artificial neural networks



- Units in a same layer do not interact
- Data go from input to output in a **feed-forward way**
- The width of the output layer corresponds to the number of classes/values that you want to predict
- The **activation** of the unit j in layer one is given by

Layer 1 — Weights of layer 1 for unit j (vectorized)

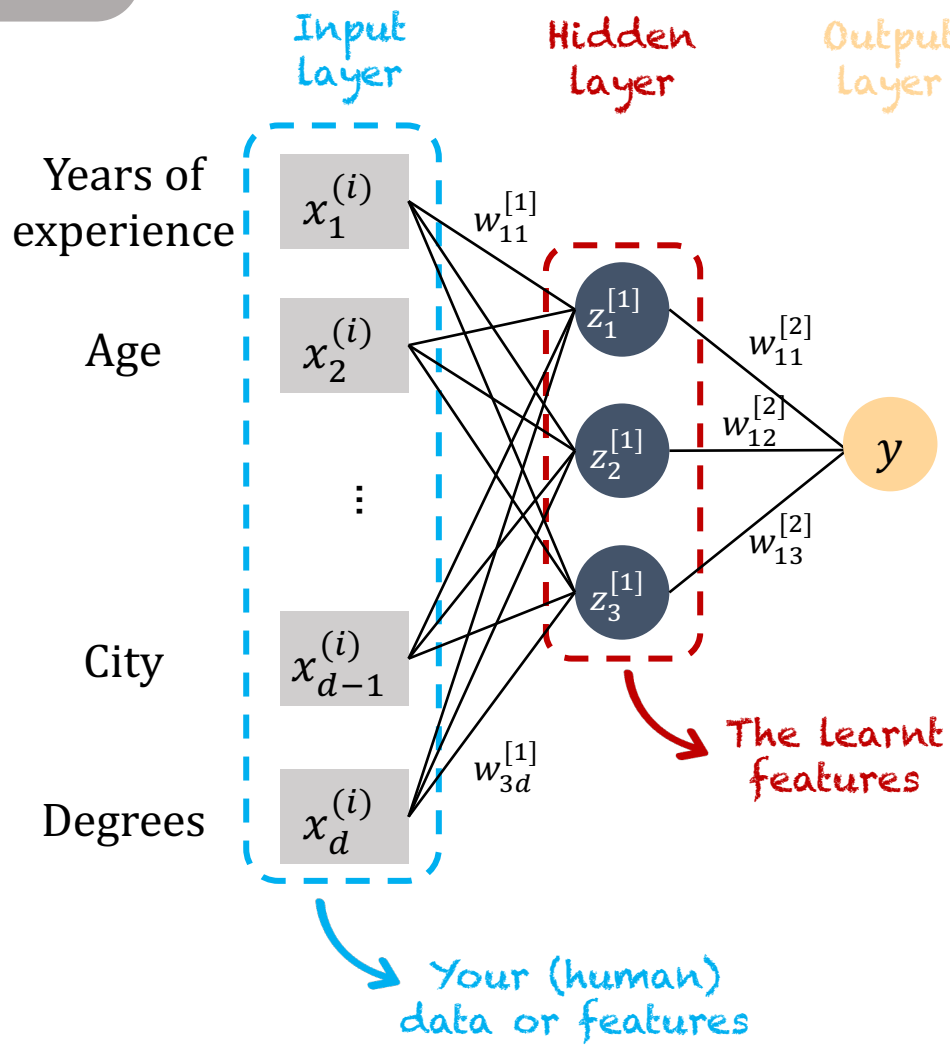
$$z_j^{[1]} = s\left(\boldsymbol{w}_j^{[1]} \cdot \boldsymbol{x}^{(i)} + b_j^{[1]}\right)$$

Unit j — Bias of the unit j in layer 1

- We also define the **pre-activation**

$$u_j^{[1]} = \boldsymbol{w}_j^{[1]} \cdot \boldsymbol{x} + b_j^{[1]}$$

Artificial neural networks



Let's have a look at the output of this network

$$y = s\left(\sum_{j=1}^{d} w_{1j}^{[2]} z_j + b^{[2]}\right)$$

$\{z_j\}_{j=1,\dots,3}$ act as **new features** to predict y

Artificial neural networks

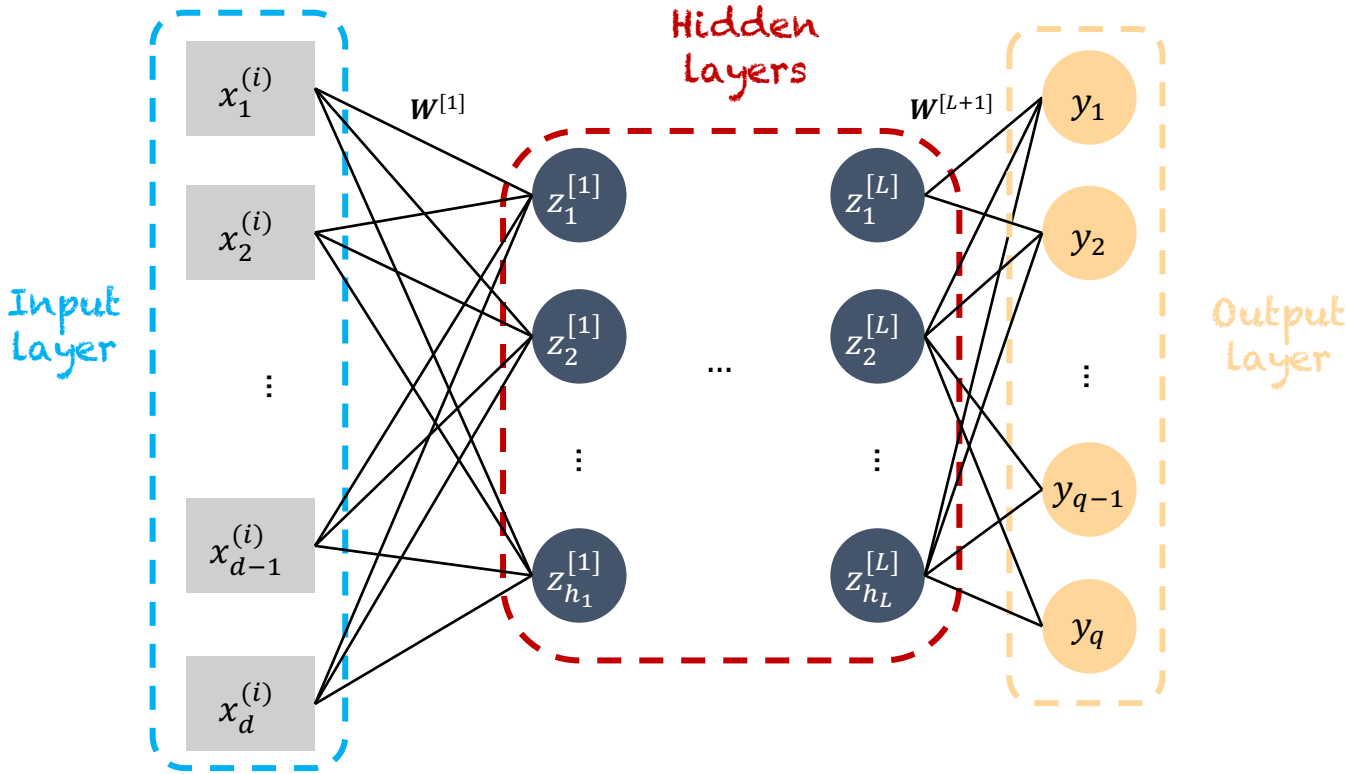
- A fully-connected neural network with d inputs, L hidden layers of width $h_1, h_2, \dots, h_L$ and an output layer of size q
- The jth output is computed as

$$y_j = s^{[L+1]}\left(\boldsymbol{W}^{[L+1]}\boldsymbol{z}^{[L]}\right)$$

$$y_j = s^{[L+1]}\left(\boldsymbol{W}^{[L+1]}s^{[L]}\left(\boldsymbol{W}^{[L]}\dots s^{[1]}\left(\boldsymbol{W}^{[1]}\boldsymbol{x}\right)\right)\right)$$

$$\boldsymbol{w}_j^{[l]} = \left[b_j^{[l]}, w_{1j}^{[l]}, w_{2j}^{[l]}, \dots, w_{h_l j}^{[l]}\right]^{\mathrm{T}}$$

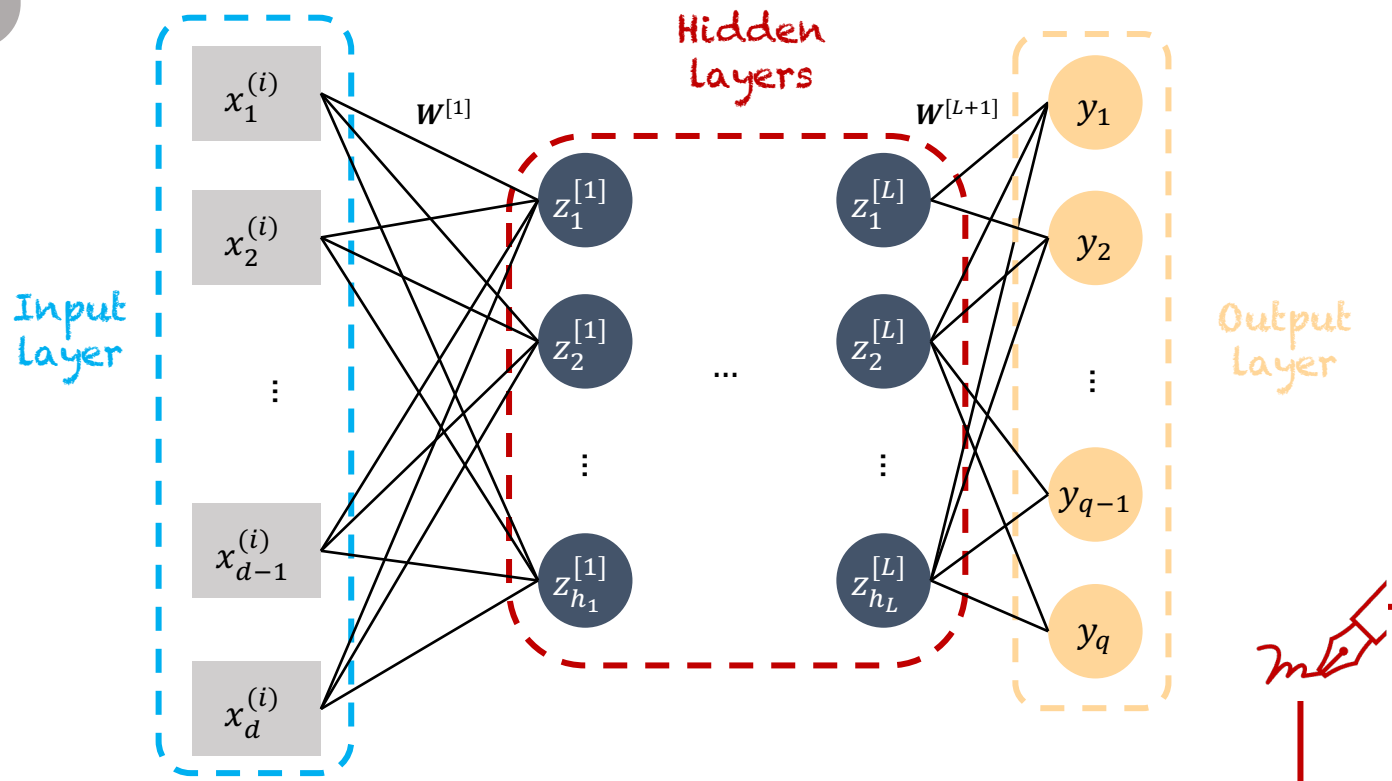
$$\boldsymbol{W}^{(l)} = \left[\boldsymbol{w}_1^{[l]}, \boldsymbol{w}_2^{[l]}, \dots, \boldsymbol{w}_{h_l}\right] \in \mathbb{R}^{h_l \times (h_{l-1}+1)}$$



- In the end, a **neural network** is a function $f_{\boldsymbol{\theta}}: X \to Y$ of some parameters ($\boldsymbol{\theta}$ = **weights and biases**)
- $f_{\boldsymbol{\theta}}$ is a composition of non-linear function when s is non-linear, allowing to build non-linear estimators
- The cascade of layers learn successive features to predict the outputs

General view of deep neural networks

Artificial neural networks



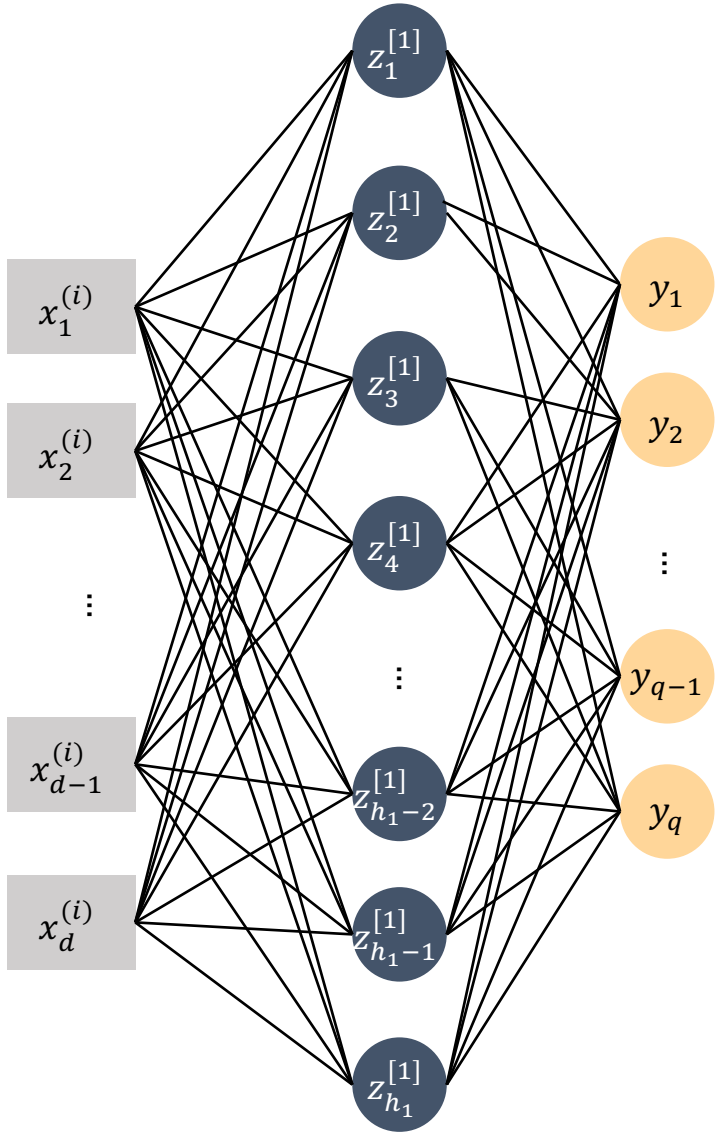
How many parameters does this network have? (don't forget biases!)

- Both regression and classification, learns features, good performances when enough data
- Need a lot of data to train, subject to overfitting, uninterpretable, targeted-purpose architectures usually work better

ANN as universal approximators

Artificial neural networks

- Some of the reasons why neural networks became later so popular "recently":
  1. Data accessibility
  2. Efficient algorithms and hardware evolution
  3. Automatic learning of features
- But they are also universal approximators (see Cybenko 1989)



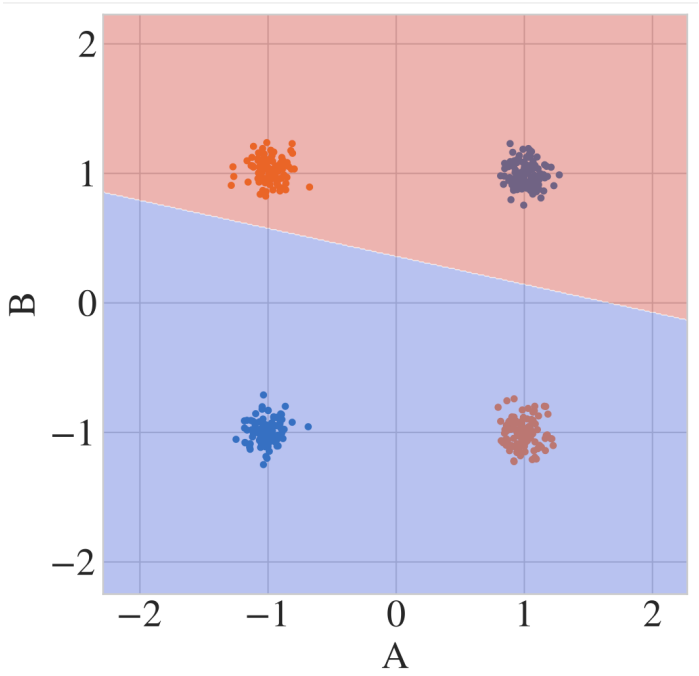
Theorem (informal)

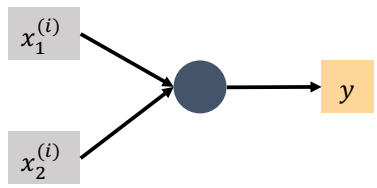
A fully-connected neural network with a single hidden layer $(L = 1)$ *with enough neurons* $(h_1$ *large*$)$ *can fit* ***any arbitrary smooth function.***

ANN: example on the XOR dataset

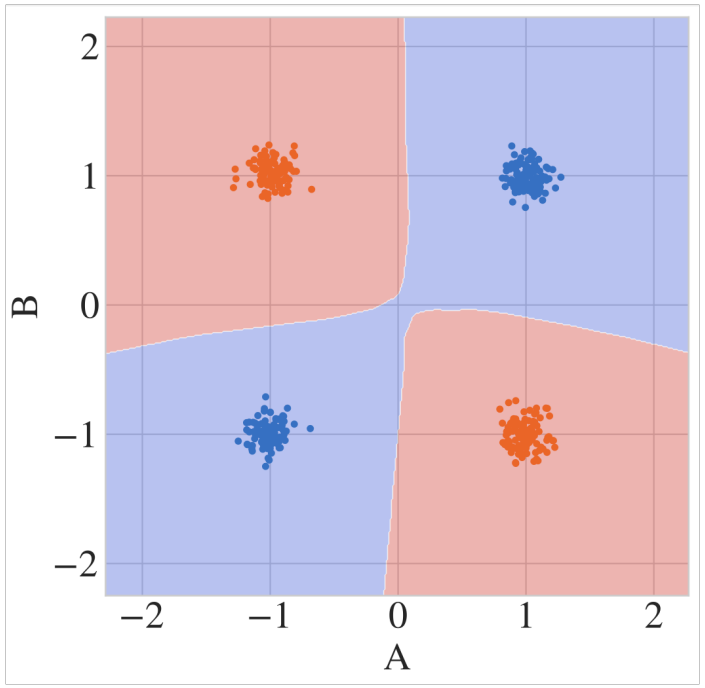
Artificial neural networks

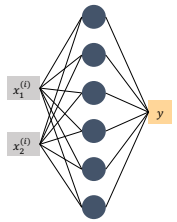
1 hidden layer of 1 unit



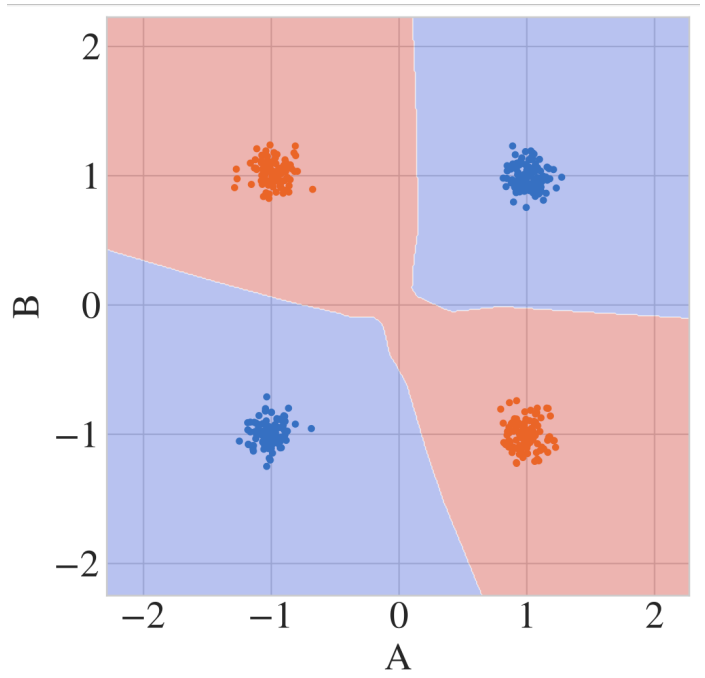


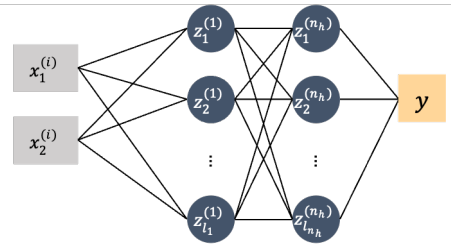
1 hidden layer of 100 units



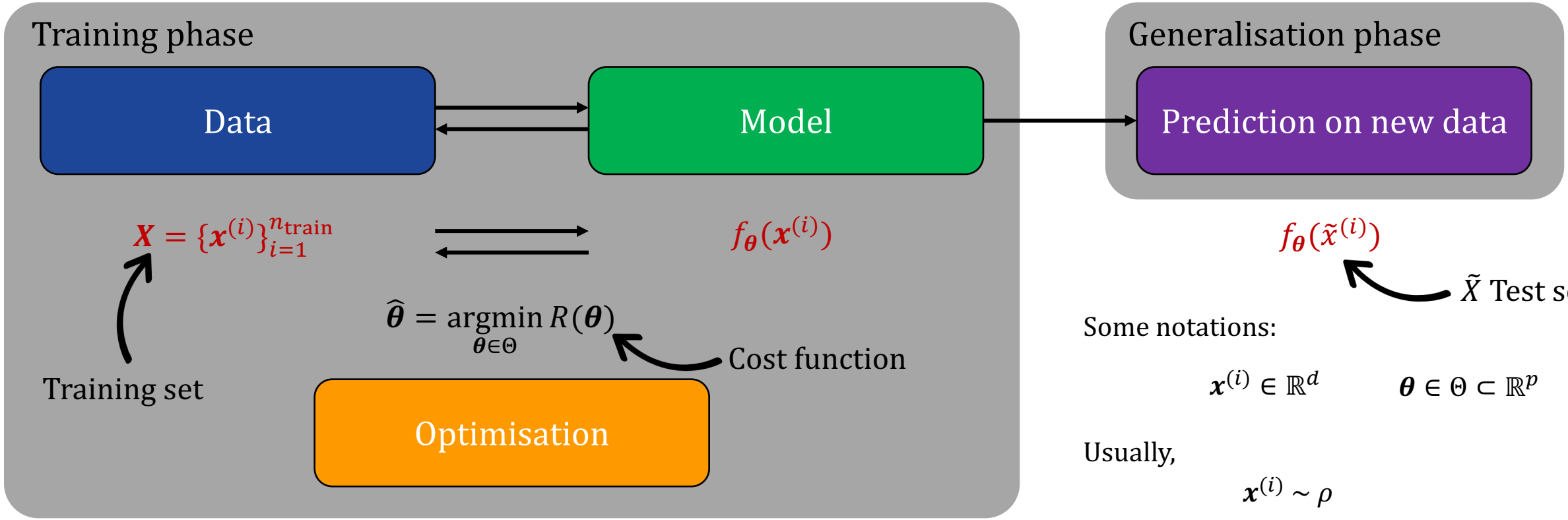


2 hidden layers of 10 units





ML building blocks



Some notations:

$x^{(i)} \in \mathbb{R}^d$ $\quad$ $\theta \in \Theta \subset \mathbb{R}^p$

Usually,

$x^{(i)} \sim \rho$

Three main building blocks of *most* ML systems: some **data** feed a **parametrised model**. The best set of parameters are obtained through the **optimisation** of a cost function. The model can then be assessed on **new data**.

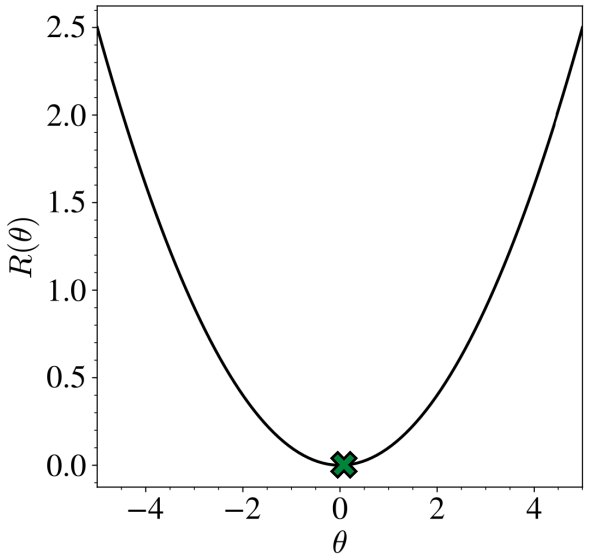
How to perform ERM in general?

- **Empirical risk minimisation** (ERM) provides a generic way to choose $\widehat{\boldsymbol{\theta}}$ in supervised learning

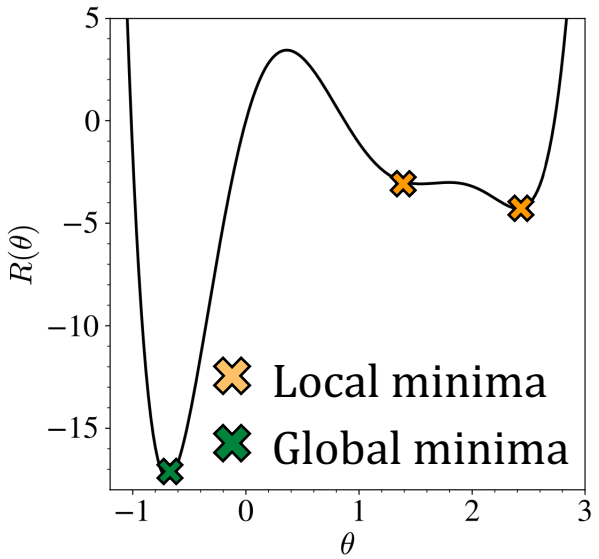
$$\widehat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, R(\boldsymbol{\theta}) = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, \frac{1}{n}\sum_{i=1}^{n} \ell\left(f_{\boldsymbol{\theta}}\left(x^{(i)}\right), y^{(i)}\right)$$

- The aim is to minimize a function of (many) parameters under a loss function encoding how well the model approximates the target variable
- In linear regression, we could directly optimise the empirical risk in closed-form, but **in general it is not possible**

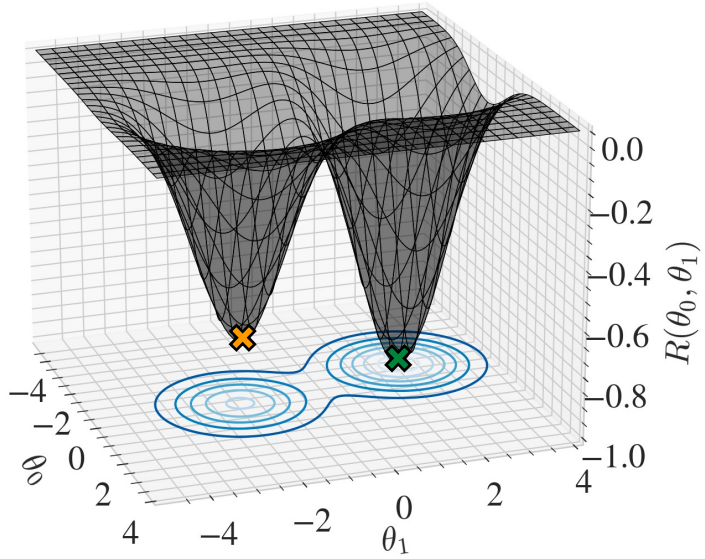
1D convex



1D non-convex

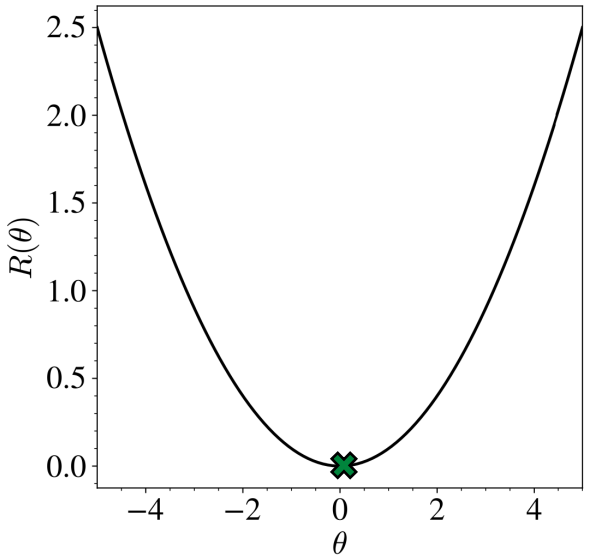


2D non-convex

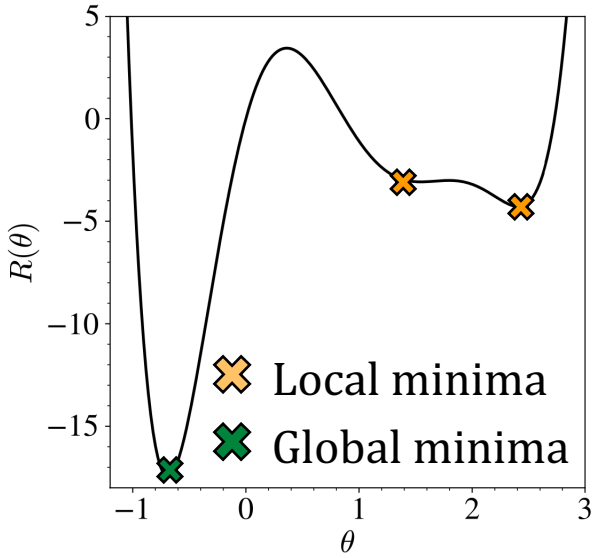


Naïve view and curse of dimensionality

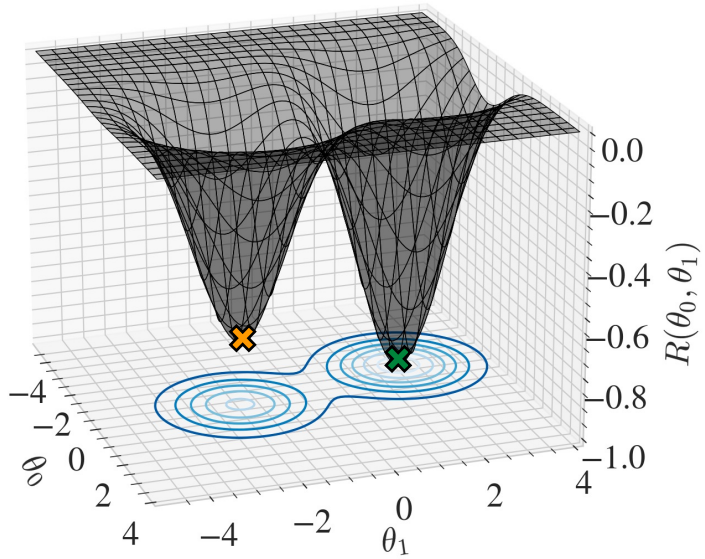
1D convex



1D non-convex



2D non-convex



- A naïve way of minimizing such functions could be to uniformly pave the parameter space and choose the one with the smallest value as being the minimum: this is **global optimization (grid search optimization)**
- This is where the **curse of dimensionality** kicks in. In general, we optimize models over many parameters $d \gg 1$ (imagine the pixels of an image) and all the points are far away from each other in high dimensions
- Sampling uniformly $[0, 1]^{10}$ with a step of 0.01 requires 10^{20} evaluations (think of GPT-3 and its 175 billion parameters!)

Gradient descent algorithm

- A solution: local, directed search to navigate through the landscape

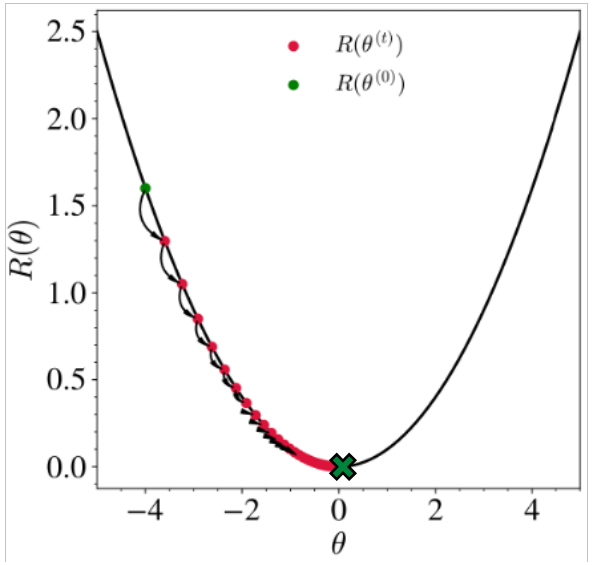
  ↳ Numerical optimisation by **gradient descent**

Algorithm: Gradient descent

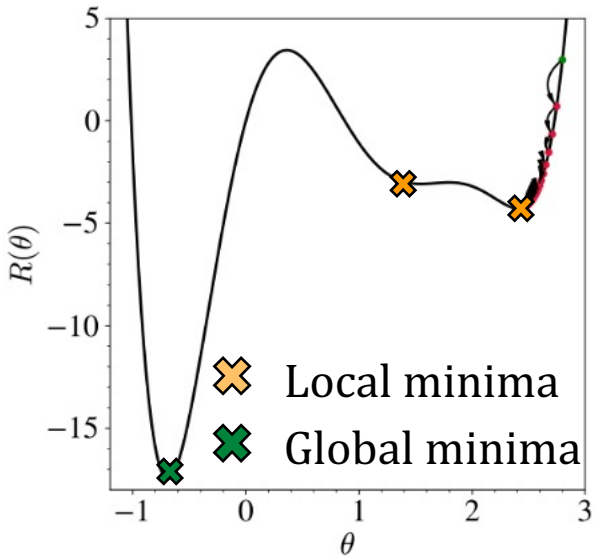
1. Initialise $\boldsymbol{\theta}_0$ randomly
2. Compute $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta \boldsymbol{\nabla}_{\boldsymbol{\theta}} R(\boldsymbol{\theta}^{(t)})$
3. Repeat 2 until $\left\| \boldsymbol{\theta}^{(t+1)} - \boldsymbol{\theta}^{(t)} \right\|_2^2 \leq \epsilon$

Note: the superscript does not have to do with the training example here but with the time step ($\boldsymbol{\theta}$ is a parameter).

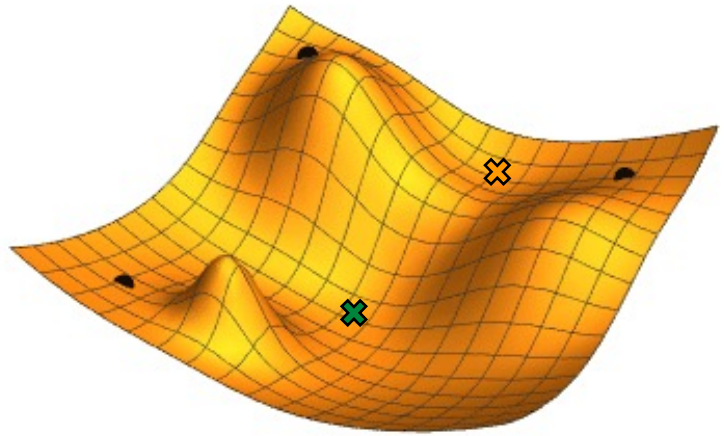
1D convex



1D non-convex



2D non-convex

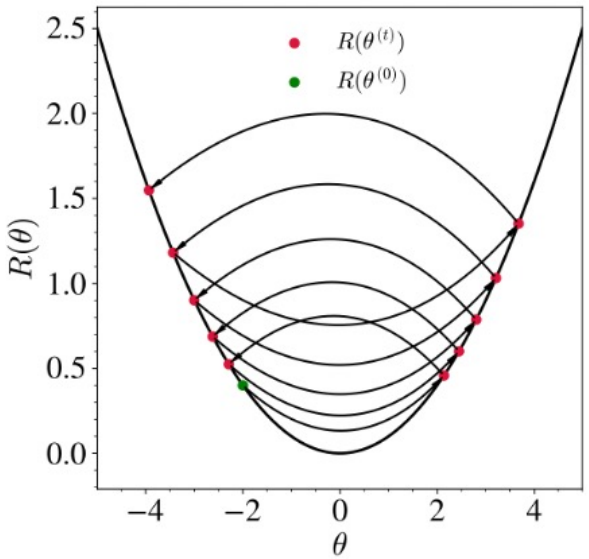


A word about hyperparameters

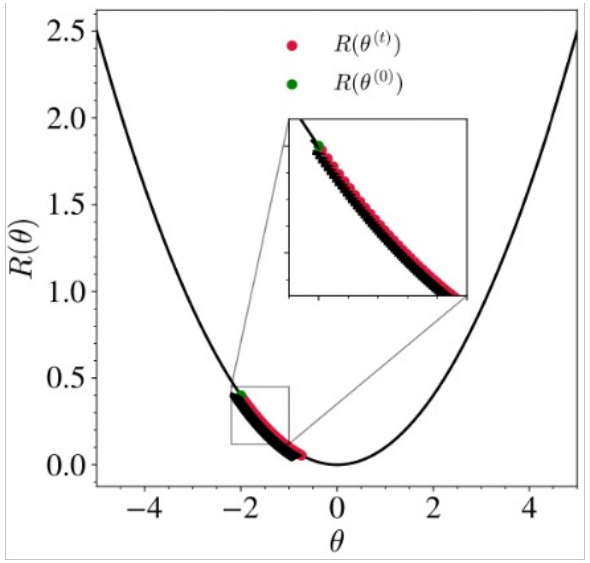
- One **hyperparameter**: the learning rate η
- A value that is too large can lead to divergence while, when too small, the computational cost explodes (+ stuck in small asperities)

Hyperparameter: parameter that is **not learned** during the optimisation. Ex: depth of tree in DTs, # of trees in RFs, learning rates, etc.

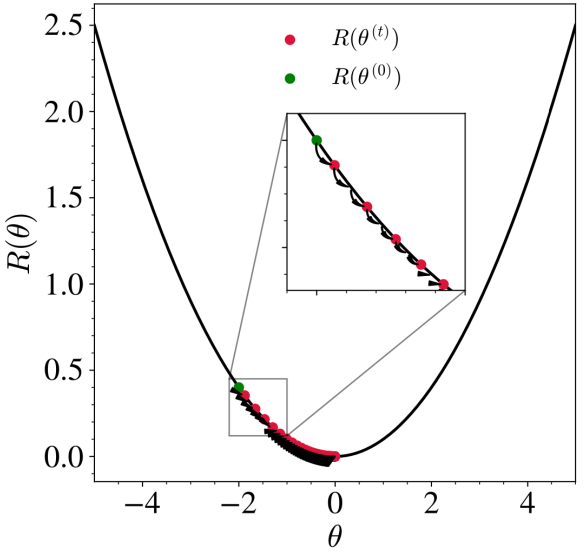
η too large



η too small



appropriate η



- Usually, we use **grid search** to find the hyperparameter performing best on a third dataset: the **validation set (or use cross-validation)**

Example: our linear regression

- In the linear regression, the risk is

$$R(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}\left(\theta_0 + \theta_1 x_1^{(i)} - y^{(i)}\right)^2 \qquad \boldsymbol{\theta} = [\theta_0, \theta_1]^{\mathrm{T}}$$

- Let's apply the gradient descent algorithm starting from $\boldsymbol{\theta}^{(0)}$ random
- Then, we need to compute the gradient $\nabla_{\boldsymbol{\theta}} R(\boldsymbol{\theta})$

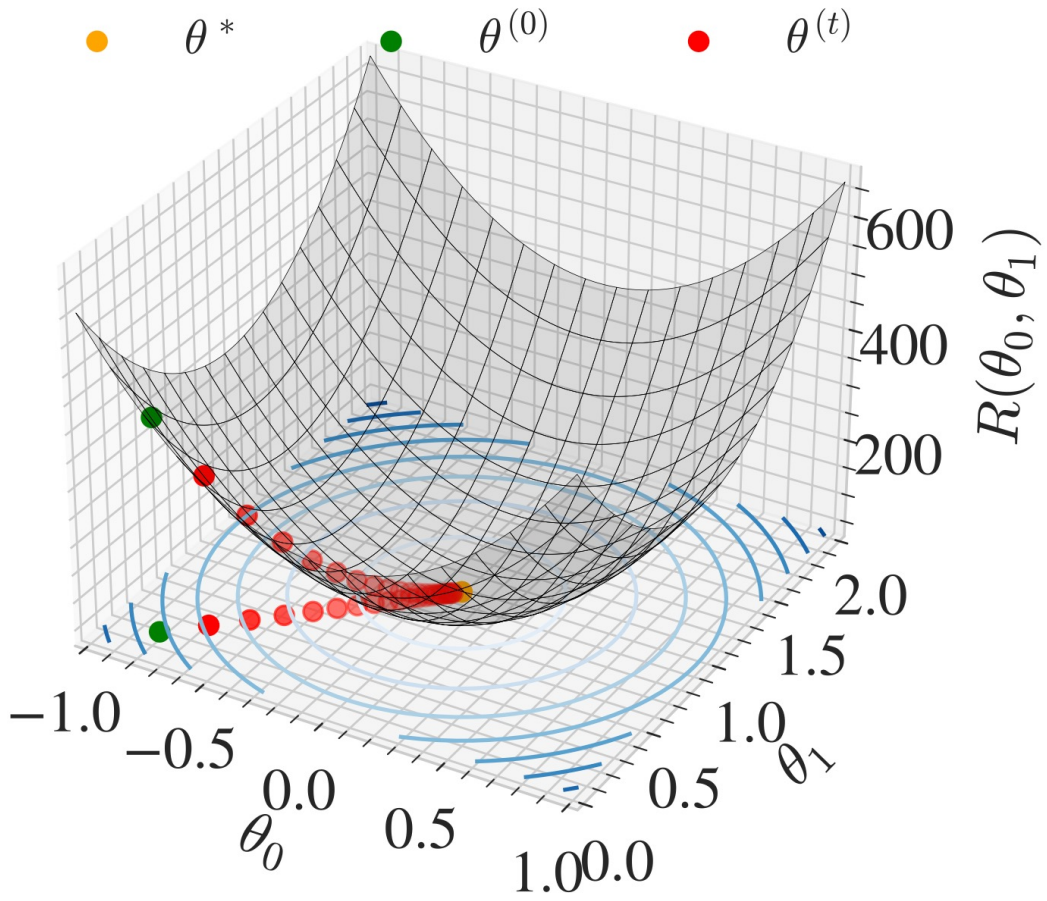
$$\frac{\partial R(\boldsymbol{\theta})}{\partial \theta_0} = \frac{2}{n}\sum_{i=1}^{n}\left(\theta_0 + \theta_1 x_1^{(i)} - y^{(i)}\right),$$

$$\frac{\partial R(\boldsymbol{\theta})}{\partial \theta_1} = \frac{2}{n}\sum_{i=1}^{n} x_1^{(i)}\left(\theta_0 + \theta_1 x_1^{(i)} - y^{(i)}\right)$$

- Therefore, the update is

$$\theta_j^{(t+1)} = \theta_j^{(t)} - \frac{\eta}{n}\sum_{i=1}^{n} x_j^{(i)}\left(\theta_0 + \theta_1 x_1^{(i)} - y^{(i)}\right)$$

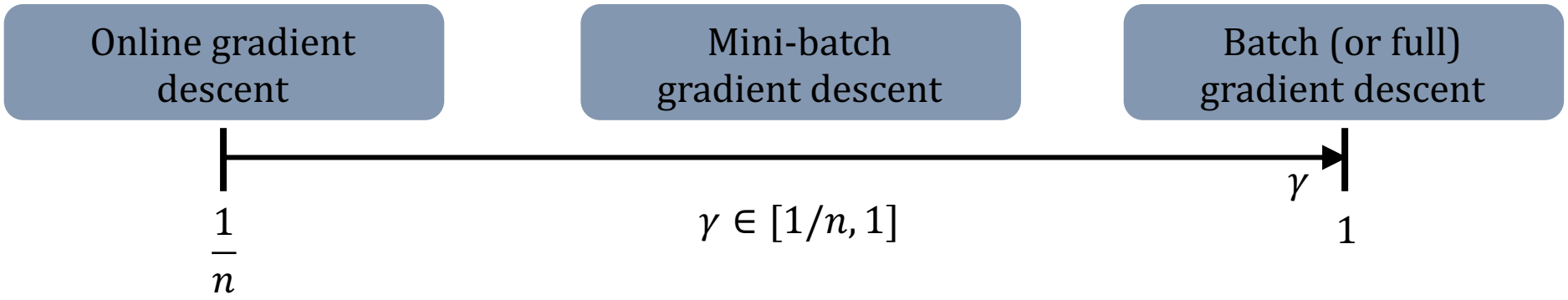
where $\forall\, i \in [1, \cdots, n]\ x_0^{(i)} = 1$



Note that I **standardized** the features. This is sometimes required when optimising some models. In GD, it allows faster convergence.

Stochastic gradient descent algorithm

- Problem of gradient descent: we **need the entire dataset** to compute $\nabla_{\boldsymbol{\theta}} R(\boldsymbol{\theta}^{(t)})$
- Solution: what about using only a fraction γ of the dataset chosen randomly?



Algorithm: Stochastic gradient descent

1. Initialise $\boldsymbol{\theta}_0$ randomly
2. For $e \in [1, \cdots, E]$
   - 2.1. Shuffle the dataset
   - 2.2. For $i \in [1, \cdots, \lfloor \gamma n \rfloor]$
     - 2.2.1. Compute $\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} - \eta \, \widehat{\boldsymbol{\nabla}}_{\boldsymbol{\theta}} R(\boldsymbol{\theta}^{(i)})$

where $\widehat{\boldsymbol{\nabla}} R(\boldsymbol{\theta}) = \sum_{j=i\gamma n}^{i\gamma n+\gamma n} \boldsymbol{\nabla} R_j(\boldsymbol{\theta})$

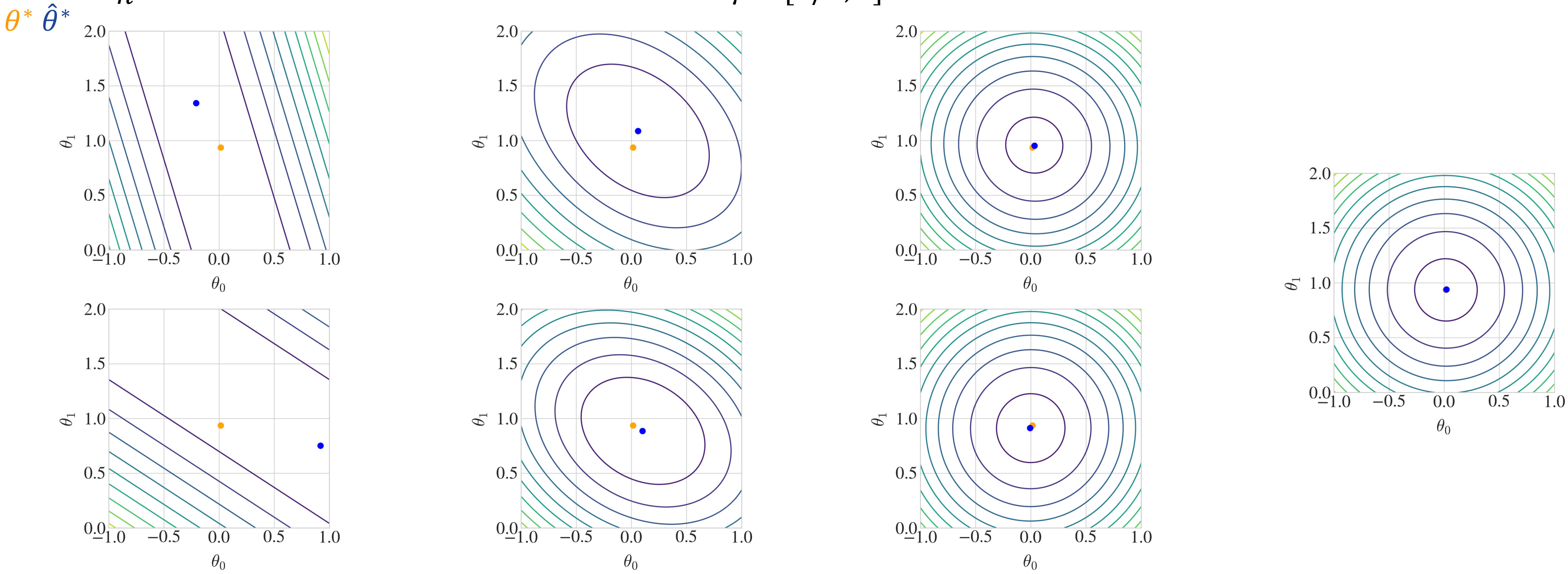
- e is called an **epoch** and a set of γn training examples is called a **mini-batch**
- Usually**, SGD often converges faster than full-batch** GD
- It may however oscillate around the true minimum
- Under some technical assumptions, **SGD provides an almost sure convergence** to a local (resp. global) in non-convex (resp. convex) landscapes

SGD landscapes on our linear regression

Online gradient descent | **Mini-batch gradient descent** | **Batch (or full) gradient descent**

$\frac{1}{n}$ ⟶ 1

$\gamma \in [1/n, 1]$



$\gamma = 1/n$ | $\gamma = 0.1$ | $\gamma = 0.5$ | $\gamma = 1$

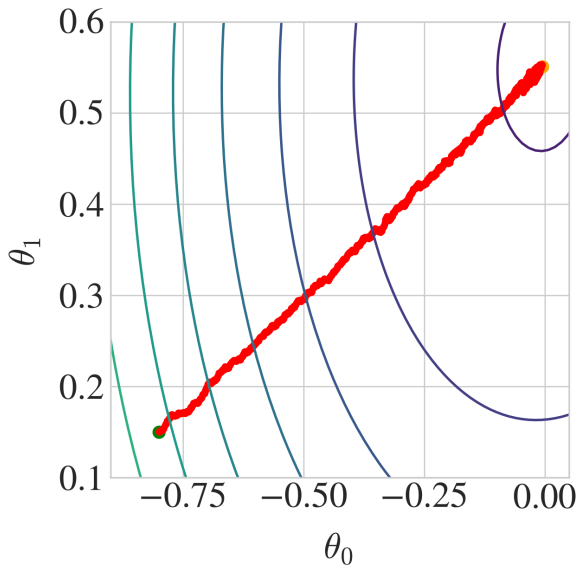
SGD on our linear regression

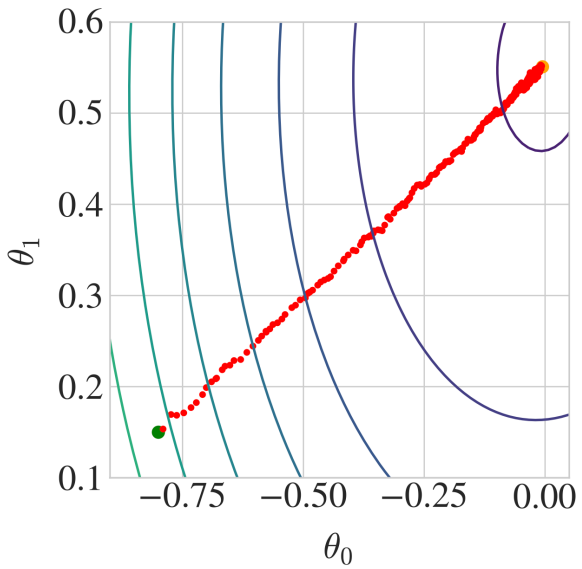
Online gradient descent

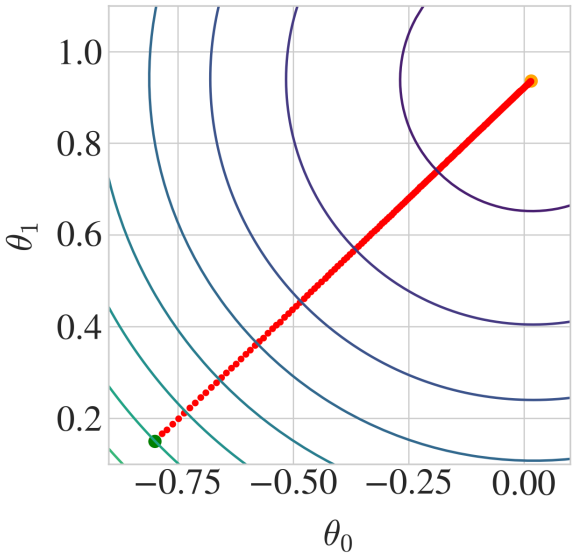
Mini-batch gradient descent

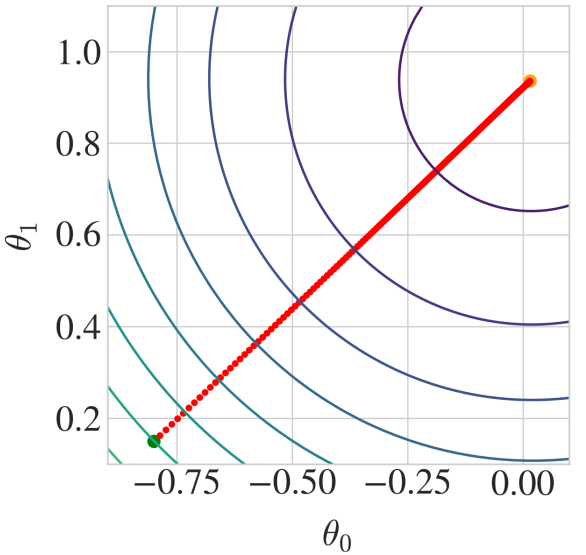
Batch (or full) gradient descent

$\frac{1}{n}$ — $\gamma \in [1/n, 1]$ — 1









$\gamma = 1/n$

$\gamma = 0.1$

$\gamma = 0.5$

$\gamma = 1$

- The problem is that neural networks are compositions of non-linear functions

$$y_j = s^{[L+1]}\left(\boldsymbol{W}^{[L+1]} s^{[L]}\left(\boldsymbol{W}^{[L]} \dots s^{[1]}\left(\boldsymbol{W}^{[1]}\boldsymbol{x}\right)\right)\right)$$

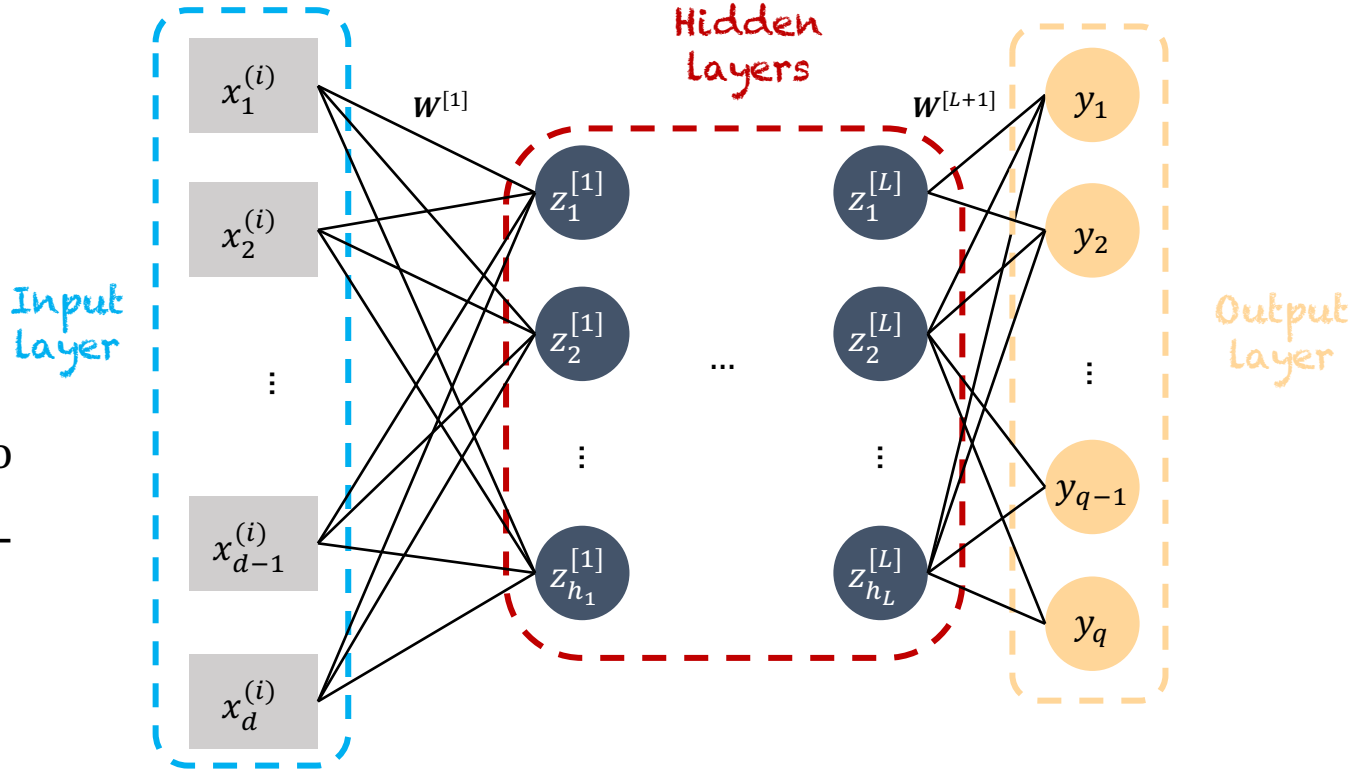
$$\boldsymbol{W}^{(l)} = \left[\boldsymbol{w}_1^{[l]}, \boldsymbol{w}_2^{[l]}, \dots, \boldsymbol{w}_{h_l}\right] \in \mathbb{R}^{h_l \times h_{l-1}}$$

$$\boldsymbol{w}_j^{[l]} = \left[b_j^{[l]}, w_{1j}^{[l]}, w_{2j}^{[l]}, \dots, w_{h_l j}^{[l]}\right]^{\mathrm{T}}$$

- We however need to optimize the cost function to obtain the "best" values of $\boldsymbol{W}$ producing the closer-to-optimal target values

- How to compute $\frac{\partial \ell(\boldsymbol{W}^{[0]},\dots,\boldsymbol{W}^{[L+1]})}{\partial w_{ij}^{[l]}}$?

- **The backpropagation of errors**: an application of the **chain rule!**



- Take the example of a fully-connected network with $d = 3$, one hidden layer, and two output neurons: $\hat{\boldsymbol{y}} = s\left(\boldsymbol{W}^{[2]} s\left(\boldsymbol{W}^{[1]}\boldsymbol{x}\right)\right)$

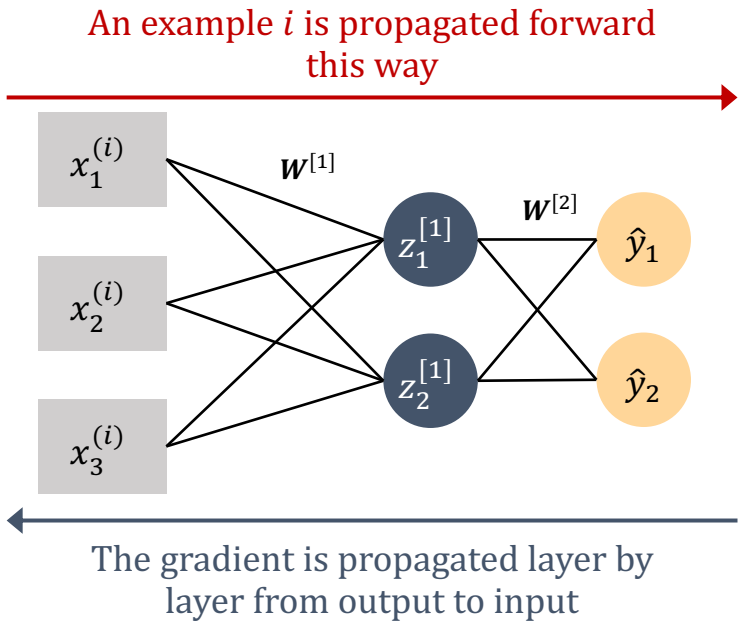
$$\boldsymbol{W}^{[1]} = \begin{bmatrix} b_1^{[1]} & w_{11}^{[1]} & w_{21}^{[1]} & w_{31}^{[1]} \\ b_2^{[1]} & w_{12}^{[1]} & w_{22}^{[1]} & w_{32}^{[1]} \end{bmatrix} \qquad \boldsymbol{W}^{[2]} = \begin{bmatrix} b_1^{[2]} & w_{11}^{[2]} & w_{21}^{[2]} \\ b_2^{[2]} & w_{12}^{[2]} & w_{22}^{[2]} \end{bmatrix}$$

- Consider the squared error loss function for each training example i

$$\ell\left(\boldsymbol{W}^{[1]}, \boldsymbol{W}^{[2]}\right) = \frac{1}{2}\sum_{j=1}^{2}\left(y_j - \hat{y}_j\right)^2$$

- From the equation of $\hat{\boldsymbol{y}}$, we see the contribution of $\boldsymbol{W}^{[2]}$ is "closer" to the output than $\boldsymbol{W}^{[1]}$

- Let's compute $\dfrac{\partial \ell\left(\boldsymbol{W}^{[1]}, \boldsymbol{W}^{[2]}\right)}{\partial w_{11}^{[2]}}$



- Using the chain rule

$$\frac{\partial \ell(\boldsymbol{W}^{[1]}, \boldsymbol{W}^{[2]})}{\partial w_{11}^{[2]}} = \frac{\partial \ell}{\partial u_1^{[2]}} \times \frac{\partial u_1^{[2]}}{\partial w_{11}^{[2]}}$$

where $u_1^{[2]}$ is the pre-activation of the unit 1 in layer 2 (here output layer)

$$u_1^{[2]} = w_{11}^{[2]} z_1^{[1]} + w_{21}^{[2]} z_2^{[1]} + b_1^{[2]}$$

- The second term is then easy to compute as $\frac{\partial u_1^{[2]}}{\partial w_{11}^{[2]}} = z_1^{[1]}$
- For the first term, use the chain rule again leads to

$$\delta_1^{[2]} = \frac{\partial \ell}{\partial u_1^{[2]}} = \frac{\partial \ell}{\partial \hat{y}_1} \frac{\partial \hat{y}_1}{\partial u_1^{[2]}}$$

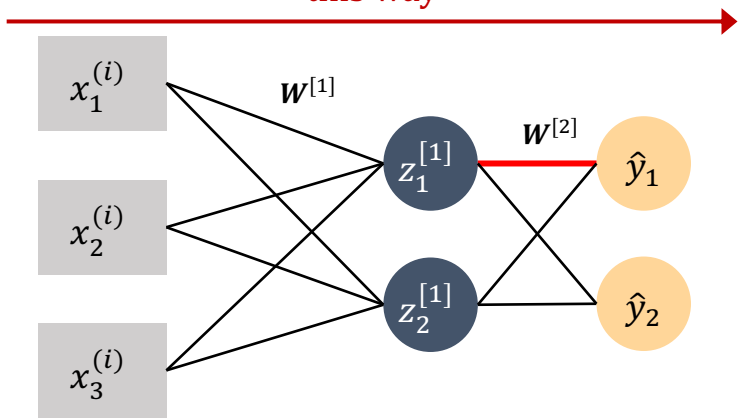
with $\hat{y}_1 = s\left(u_1^{[2]}\right) = s(w_{11}^{[2]} z_1^{[1]} + w_{21}^{[2]} z_2^{[1]} + b_1^{[2]})$

Finally,

$$\frac{\partial \ell(\boldsymbol{W}^{[1]}, \boldsymbol{W}^{[2]})}{\partial w_{11}^{[2]}} = (y_1 - \hat{y}_1) s'\left(u_1^{[2]}\right) z_1^{[1]}$$

$$\ell(\boldsymbol{W}^{[1]}, \boldsymbol{W}^{[2]}) = \frac{1}{2} \sum_{j=1}^{2} \left(y_j - \hat{y}_j^{(i)}\right)^2$$

An example i is propagated forward this way



The gradient is propagated layer by layer from output to input

$$\hat{y} = s\left(\boldsymbol{W}^{[2]} s(\boldsymbol{W}^{[1]} \boldsymbol{x})\right)$$

$$\boldsymbol{W}^{[1]} = \begin{bmatrix} b_1^{[1]} & w_{11}^{[1]} & w_{11}^{[1]} & w_{31}^{[1]} \\ b_2^{[1]} & w_{12}^{[1]} & w_{22}^{[1]} & w_{32}^{[1]} \end{bmatrix}$$

$$\boldsymbol{W}^{[2]} = \begin{bmatrix} b_1^{[2]} & w_{11}^{[2]} & w_{21}^{[2]} \\ b_2^{[2]} & w_{12}^{[2]} & w_{22}^{[2]} \end{bmatrix}$$

- You can proceed the same for all the weights linked to the output layer
- What about parameters in the hidden layers?
- Let's compute

$$\frac{\partial \ell(\boldsymbol{W}^{[1]}, \boldsymbol{W}^{[2]})}{\partial w_{11}^{[1]}} = \frac{\partial \ell}{\partial u_1^{[1]}} \times \frac{\partial u_1^{[1]}}{\partial w_{11}^{[1]}}$$

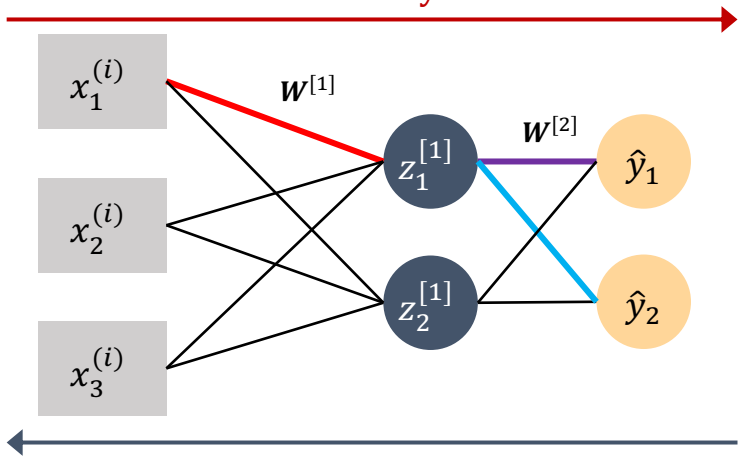
- The second term is still easy to compute
- For the first term, we have $u_1^{[1]} = w_{11}^{[1]} x_1 + w_{21}^{[1]} x_2 + b_1^{[1]}$ and observe that there are two paths to reach the weight $w_{11}^{[1]}$:

$$\delta_1^{[1]} = \frac{\partial \ell}{\partial u_1^{[2]}} \frac{\partial u_1^{[2]}}{\partial u_1^{[1]}} + \frac{\partial \ell}{\partial u_2^{[2]}} \frac{\partial u_2^{[2]}}{\partial u_1^{[1]}}$$

Already computed in the previous layer (hence the name "Backpropagation")

$$\ell(\boldsymbol{W}^{[1]}, \boldsymbol{W}^{[2]}) = \frac{1}{2} \sum_{j=1}^{2} \left(y_j - \hat{y}_j^{(i)}\right)^2$$

An example i is propagated forward this way



The gradient is propagated layer by layer from output to input

$$\hat{y} = s\left(\boldsymbol{W}^{[2]} s\left(\boldsymbol{W}^{[1]} \boldsymbol{x}\right)\right)$$

$$\boldsymbol{W}^{[1]} = \begin{bmatrix} b_1^{[1]} & w_{11}^{[1]} & w_{11}^{[1]} & w_{31}^{[1]} \\ b_2^{[1]} & w_{12}^{[1]} & w_{22}^{[1]} & w_{32}^{[1]} \end{bmatrix}$$

$$\boldsymbol{W}^{[2]} = \begin{bmatrix} b_1^{[2]} & w_{11}^{[2]} & w_{21}^{[2]} \\ b_2^{[2]} & w_{12}^{[2]} & w_{22}^{[2]} \end{bmatrix}$$

$$\delta_1^{[1]} = \frac{\partial \ell}{\partial u_1^{[2]}} \frac{\partial u_1^{[2]}}{\partial u_1^{[1]}} + \frac{\partial \ell}{\partial u_2^{[2]}} \frac{\partial u_2^{[2]}}{\partial u_1^{[1]}}$$

Already computed in the previous layer (hence the name "Backpropagation")

- To compute $\frac{\partial u_1^{[2]}}{\partial u_1^{[1]}}$ and $\frac{\partial u_2^{[2]}}{\partial u_1^{[1]}}$, we can use the same chain rule again giving

$$\frac{\partial u_1^{[2]}}{\partial u_1^{[1]}} = \frac{\partial u_1^{[2]}}{\partial z_1^{[1]}} \frac{\partial z_1^{[1]}}{\partial u_1^{[1]}} = w_{11}^{[2]} s'\left(u_1^{[1]}\right) \qquad \frac{\partial u_2^{[2]}}{\partial u_1^{[1]}} = \frac{\partial u_2^{[2]}}{\partial z_1^{[1]}} \frac{\partial z_1^{[1]}}{\partial u_1^{[1]}} = w_{21}^{[2]} s'\left(u_1^{[1]}\right)$$

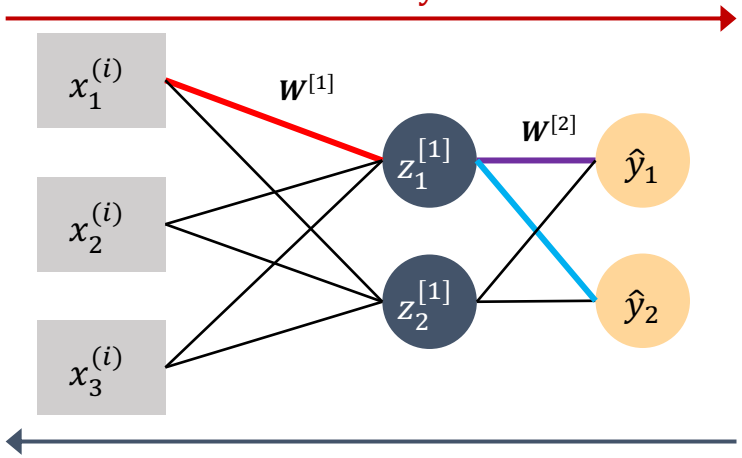
- Hence,

$$\frac{\partial \ell\left(\boldsymbol{W}^{[1]}, \boldsymbol{W}^{[2]}\right)}{\partial w_{11}^{[1]}} = \left(\delta_1^{[2]} w_{11}^{[2]} + \delta_2^{[2]} w_{21}^{[2]}\right) s'\left(u_1^{[1]}\right) x_1$$

- Where everything is known!

$$\ell\left(\boldsymbol{W}^{[1]}, \boldsymbol{W}^{[2]}\right) = \frac{1}{2} \sum_{j=1}^{2} \left(y_j - \hat{y}_j^{(i)}\right)^2$$

An example i is propagated forward this way



The gradient is propagated layer by layer from output to input

$$\hat{y} = s\left(\boldsymbol{W}^{[2]} s\left(\boldsymbol{W}^{[1]} \boldsymbol{x}\right)\right)$$

$$\boldsymbol{W}^{[1]} = \begin{bmatrix} b_1^{[1]} & w_{11}^{[1]} & w_{11}^{[1]} & w_{31}^{[1]} \\ b_2^{[1]} & w_{12}^{[1]} & w_{22}^{[1]} & w_{32}^{[1]} \end{bmatrix}$$

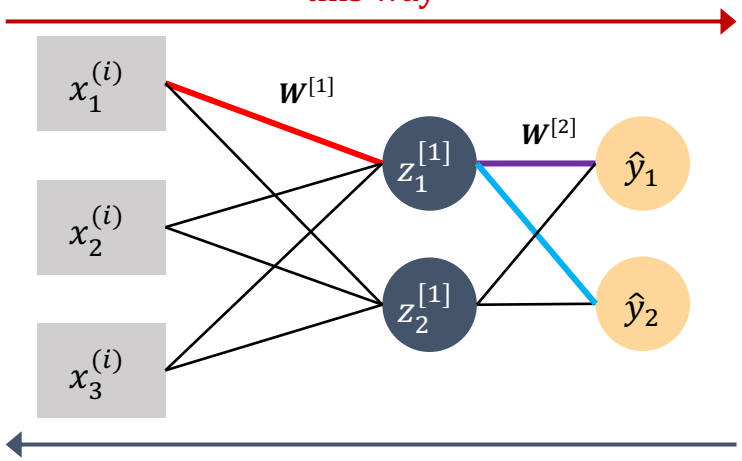
$$\boldsymbol{W}^{[2]} = \begin{bmatrix} b_1^{[2]} & w_{11}^{[2]} & w_{21}^{[2]} \\ b_2^{[2]} & w_{12}^{[2]} & w_{22}^{[2]} \end{bmatrix}$$

Backprop: summary and conclusions

$$\ell(\boldsymbol{W}^{[1]}, \boldsymbol{W}^{[2]}) = \frac{1}{2}\sum_{j=1}^{2}\left(y_j - \hat{y}_j^{(i)}\right)^2$$

- Writing those equations **propagating the errors from output to input recursively** for both weights and biases leads to the **backpropagation algorithm**
- This method grants an **efficient computation of the gradient** in neural networks
- Together with the SGD algorithm they allow to train efficiently networks with many parameters

An example i is propagated forward this way



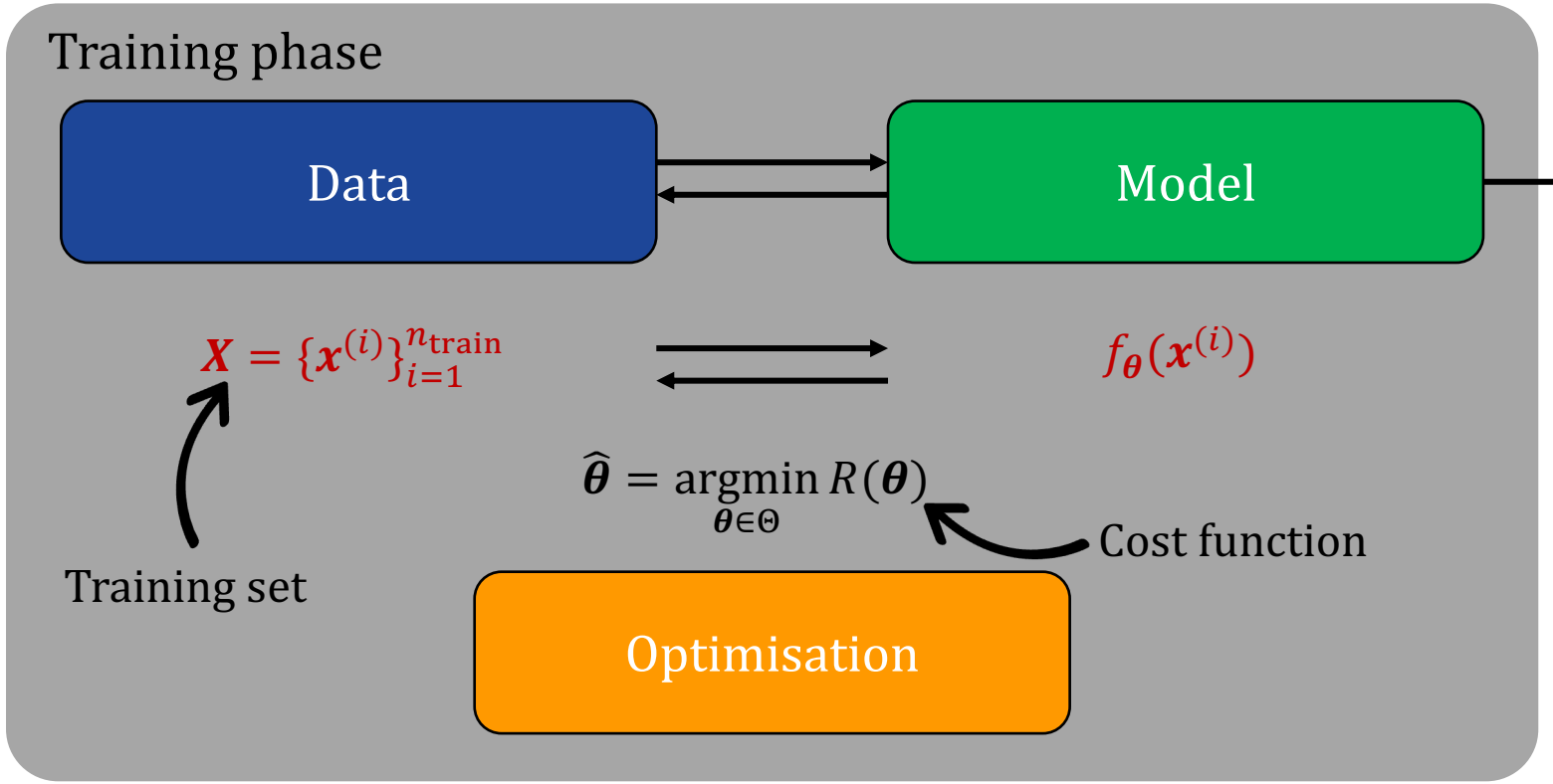
The gradient is propagated layer by layer from output to input

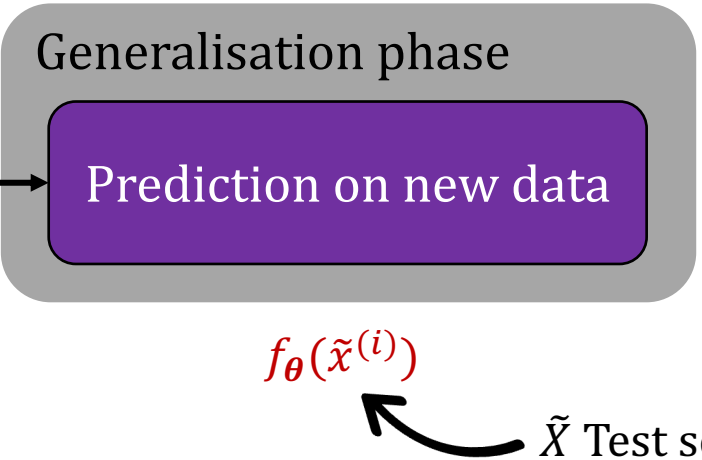
$$\hat{y} = s\left(\boldsymbol{W}^{[2]}s(\boldsymbol{W}^{[1]}\boldsymbol{x})\right)$$

$$\boldsymbol{W}^{[1]} = \begin{bmatrix} b_1^{[1]} & w_{11}^{[1]} & w_{11}^{[1]} & w_{31}^{[1]} \\ b_2^{[1]} & w_{12}^{[1]} & w_{22}^{[1]} & w_{32}^{[1]} \end{bmatrix}$$

$$\boldsymbol{W}^{[2]} = \begin{bmatrix} b_1^{[2]} & w_{11}^{[2]} & w_{21}^{[2]} \\ b_2^{[2]} & w_{12}^{[2]} & w_{22}^{[2]} \end{bmatrix}$$

ML building blocks





Some notations:

$x^{(i)} \in \mathbb{R}^d$ $\quad$ $\theta \in \Theta \subset \mathbb{R}^p$

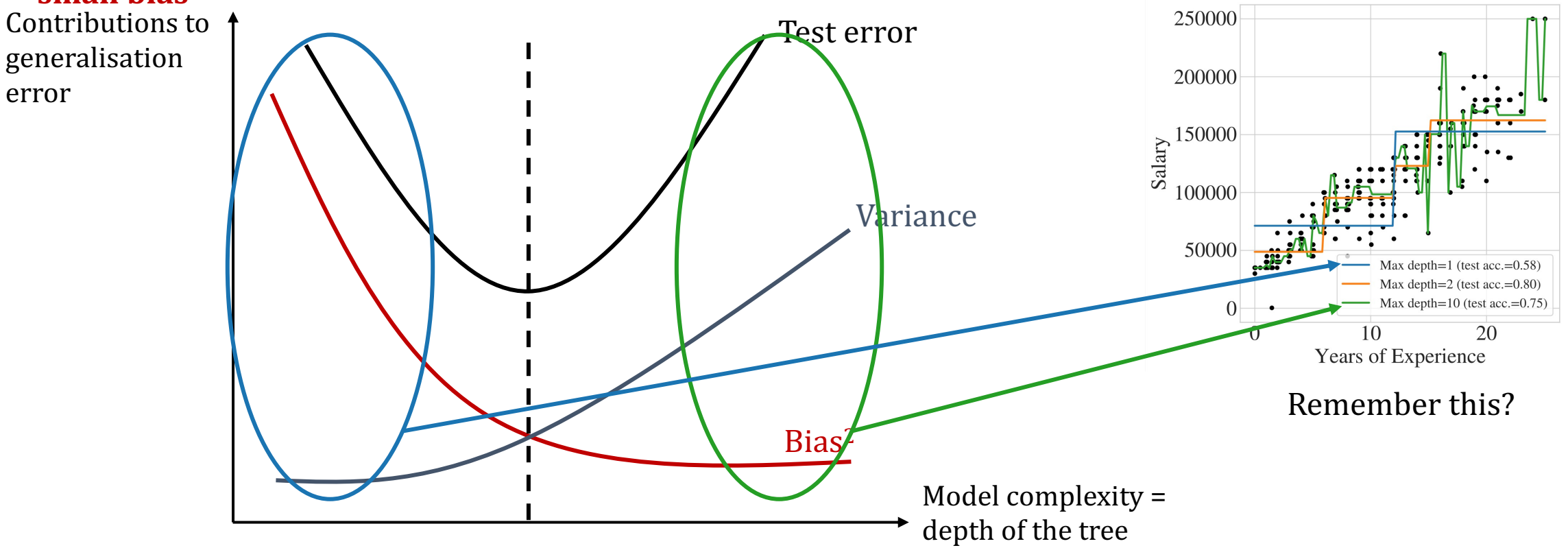
Usually,

$x^{(i)} \sim \rho$

Three main building blocks of *most* ML systems: some **data** feed a **parametrised model**. The best set of parameters are obtained through the **optimisation** of a cost function. The model can then be assessed on **new data**.

Example on decision trees

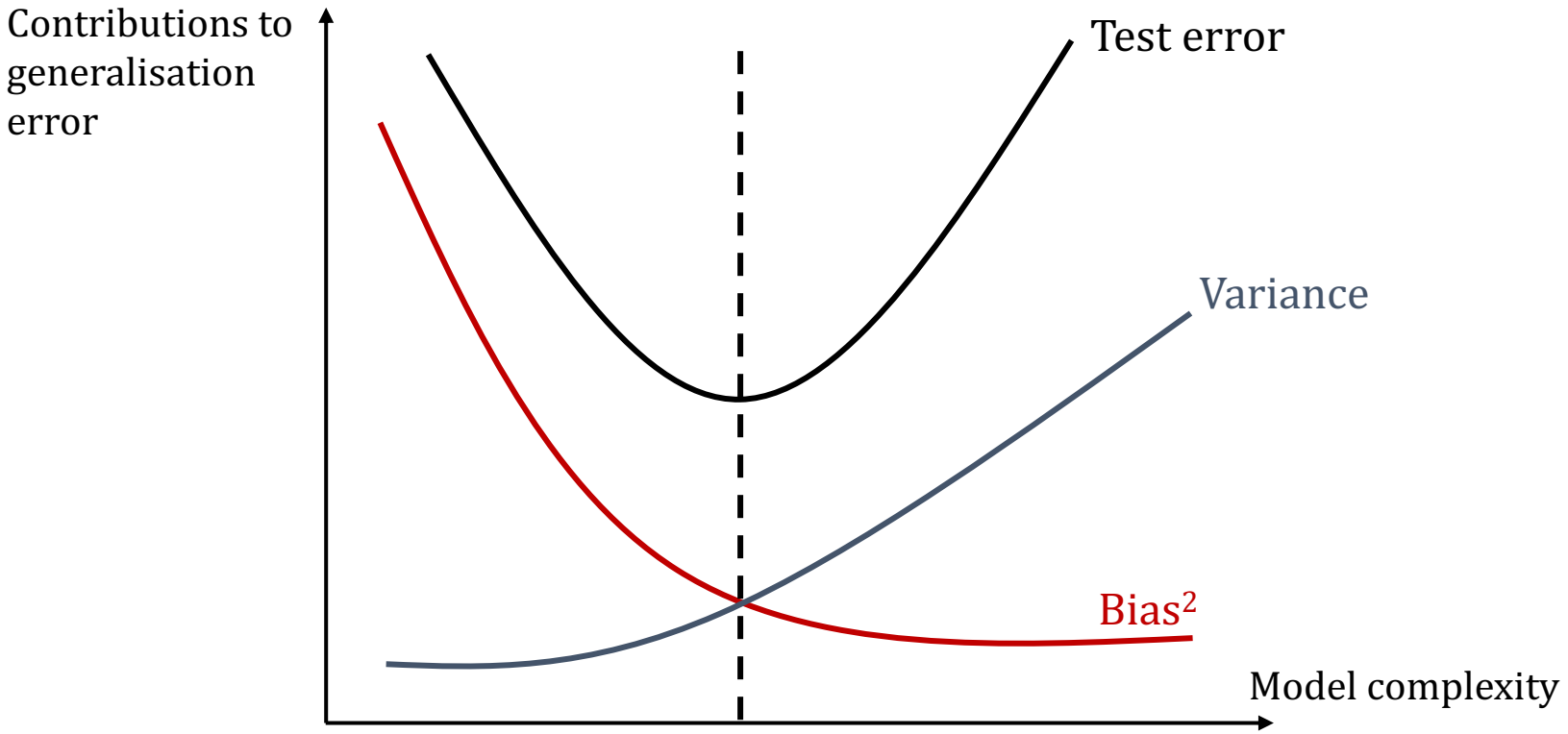
- Usually, **“simple” models have large bias and low variance while “complex” ones have large variance and small bias**



Remember this?

Model complexity and generalisation error

- Usually, **“simple” models have large bias and low variance while “complex” ones have large variance and small bias***

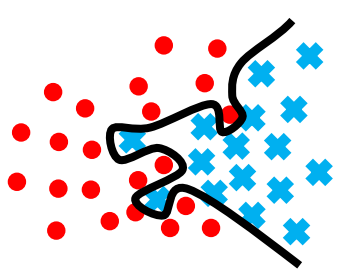
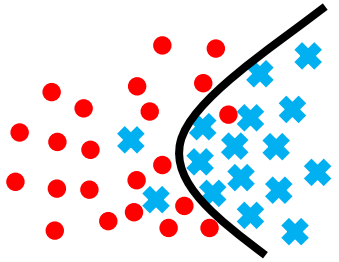
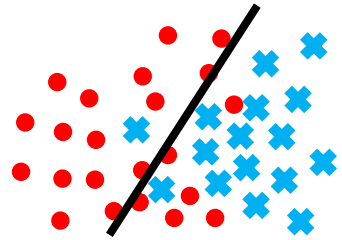
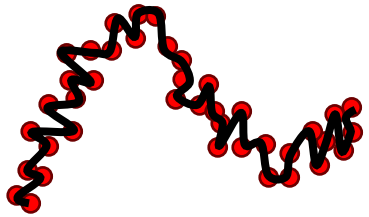
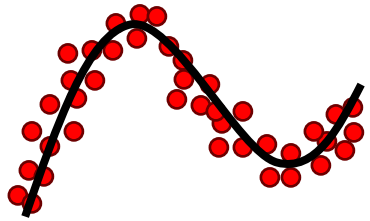
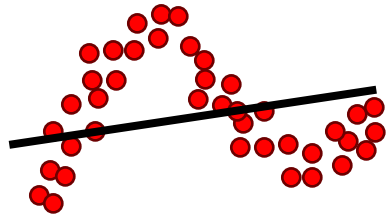


Models need to be built such that they are not too flexible to fit the noise in the data but also not too restrictive to avoid bias

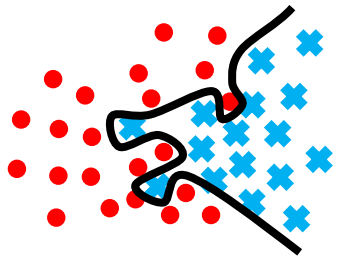
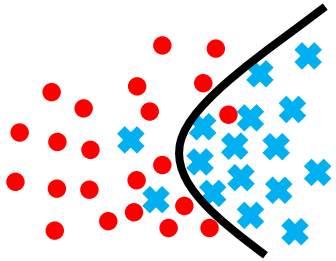
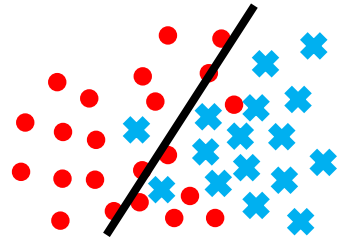
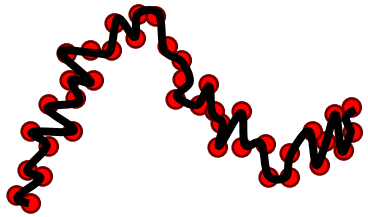
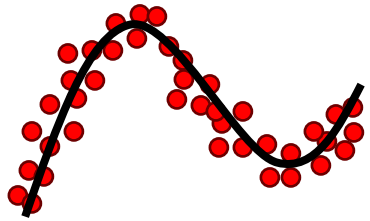
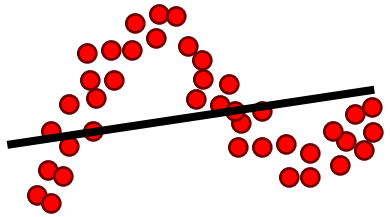
* Heavily overparametrized models in fact go beyond this view and the test error decreases again in a **double descent** phenomenon (see Belkin+18 and Nakkiran+19)

Generalisation in practice

- To measure if we really learnt something useful in supervised learning in practice, we use a **test dataset** that the model has never seen but for which we know the labels and check that $R_{\text{train}}(\hat{\theta})$ and $R_{\text{test}}(\hat{\theta})$ are of the same order
- Based on the previous view, there are two regimes where things could go wrong: high variance and low bias models vs high bias and low variance models, respectively defining **overfitting** and **underfitting**

	Overfitting	Good model	Underfitting
Classification			
Regression			
	$R_{\text{train}}(\hat{\theta}) \gg R_{\text{test}}(\hat{\theta})$	$R_{\text{train}}(\hat{\theta}) \lesssim R_{\text{test}}(\hat{\theta})$	$R_{\text{train}}(\hat{\theta}) \approx R_{\text{test}}(\hat{\theta})$

Some fixes to under and over fitting

	Overfitting	Good model	Underfitting
Classification			
Regression			
	$R_{\text{train}}(\hat{\boldsymbol{\theta}}) \gg R_{\text{test}}(\hat{\boldsymbol{\theta}})$	$R_{\text{train}}(\hat{\boldsymbol{\theta}}) \lesssim R_{\text{test}}(\hat{\boldsymbol{\theta}})$	$R_{\text{train}}(\hat{\boldsymbol{\theta}}) \approx R_{\text{test}}(\hat{\boldsymbol{\theta}})$
Possible fixes	• Add more data • Remove features • Stop the training earlier • Add regularisation	• You did a good job!	• Try a more complex model • Train your model longer

- One generic way to deal with **overfitting** is by incorporating some idea of **regularity** about the parameters: this is called explicit **regularization** and appears in the optimization problem as a constraint on parameter space

$$\hat{\theta} = \min_{\boldsymbol{\theta} \in \Theta} R(\boldsymbol{\theta})$$

Unregularized optimization

$$\hat{\theta} = \min_{\boldsymbol{\theta} \in \Theta} R(\boldsymbol{\theta}) \ \ s.t. P(\boldsymbol{\theta}) \leq \epsilon$$

Regularized optimization

- The function $P(\boldsymbol{\theta})$ is called **penalty function** and the regularized problem can in fact be written equivalently as

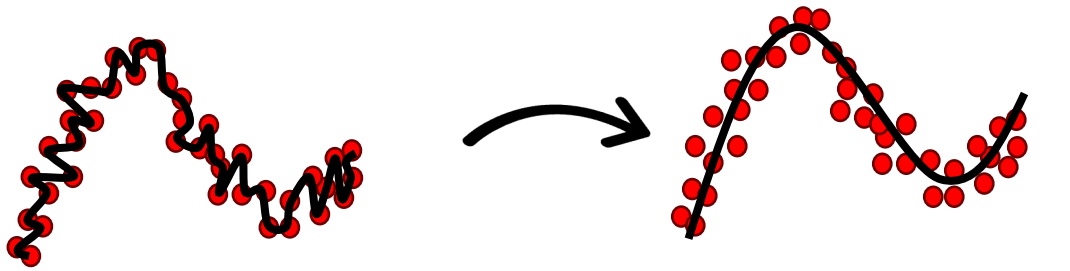
$$\hat{\theta} = \min_{\boldsymbol{\theta} \in \Theta} R(\boldsymbol{\theta}) + \lambda P(\boldsymbol{\theta})$$

1. λ is a **regularization** (hyper)**parameter**
2. $P(\boldsymbol{\theta})$ can take different forms depending on the penalty we chose to impose on the set of parameters

- Common penalty functions are the L_p-norms with $p = 0, 1$ or 2

$$P(\boldsymbol{\theta}) = \|\boldsymbol{\theta}\|_p$$

- By solving instead the regularized optimization with an appropriate value of λ, we can reduce the overfitting



$$\hat{\theta} = \min_{\boldsymbol{\theta} \in \Theta} R(\boldsymbol{\theta})$$

Unregularized optimization

$$\hat{\theta} = \min_{\boldsymbol{\theta} \in \Theta} R(\boldsymbol{\theta}) + \lambda \|\boldsymbol{\theta}\|_2^2$$

Regularized optimization

- **The model on the left lacks smoothness** which is introduced by the L_2 penalty
- The additional term here penalizes large weight values and **reduce the variance** of the estimator (even though increasing the bias)
- Regularization can also be viewed in terms of Bayesian statistics as **a prior distribution** you impose of the set of parameters