Machine Learning for Physics and Astronomy

Juan Rojo & Tanjona Rabemananjara

VU Amsterdam & Theory group, Nikhef

Natuur- en Sterrenkunde BSc (Joint Degree), Honours Track

Lecture 1, 6/9/2022

Today's lecture

- Why Machine Learning? Basic concepts and terminology
- Supervised Learning: model fitting and polynomial regression
- The need for regularisation in ML (cross-validation)
- The bias/variance tradeoff

Why Machine Learning?

Why this course?

Machine Learning rightly deserves to be part of the **toolbox of a modern scientist**

- Essential for building **modern models and algorithms** in various areas of physics and astronomy

- Very **fast developments** both in algorithms and in computing platforms have significantly extended the breadth of problems that can be tackled with ML

- Deep **physical connections** with many problems in physics and astronomy, *e.g.* quantum computation, condensed matter systems, conservation laws, ….

- Applied to problems even in very **formal fields,** *e.g.* string theory

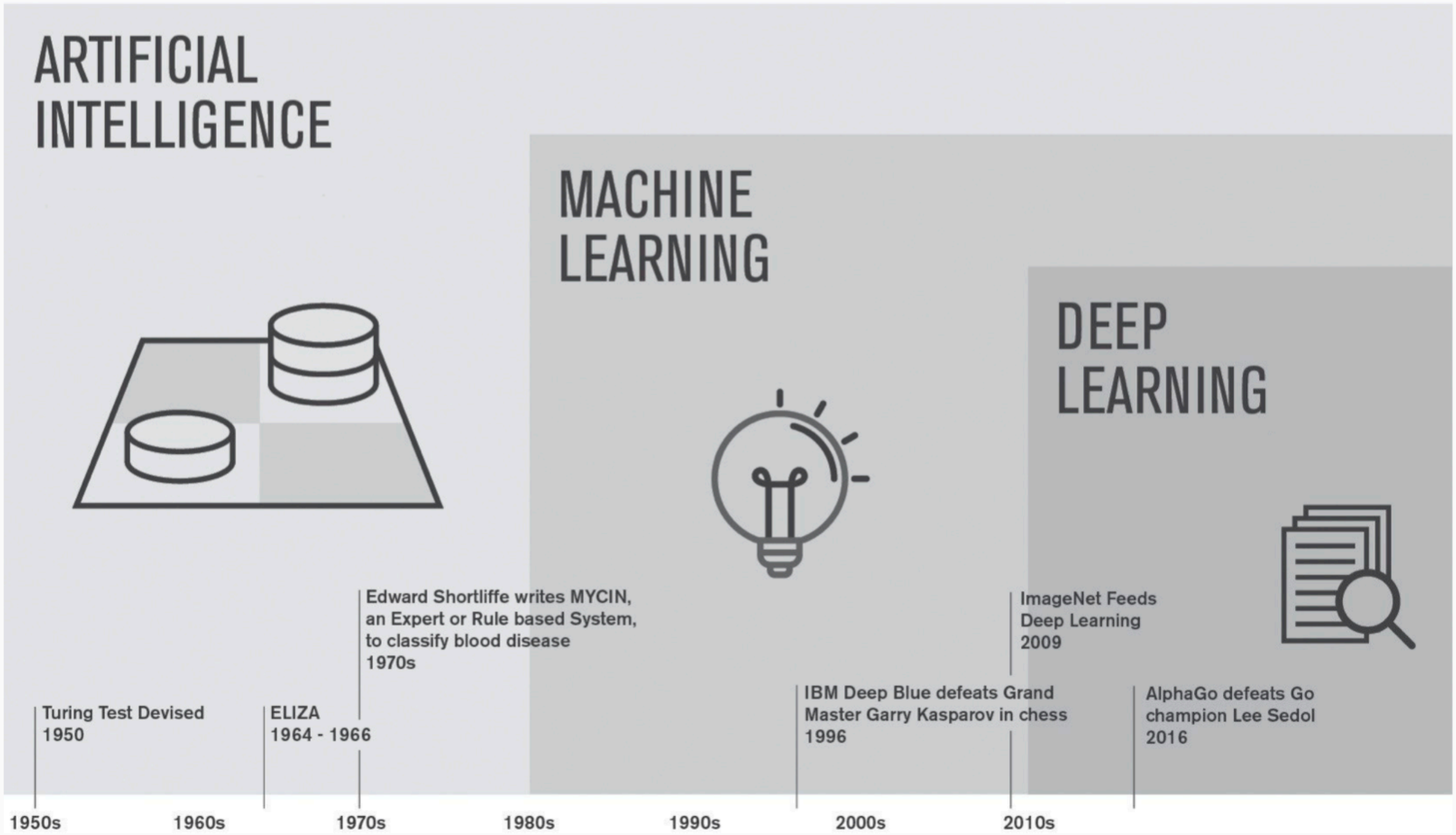
- Large interested in the community, **societal implications** (AI hype)

Furthermore, **expertise in ML/AI** is powerful asset for also for careers outside academia

AI vs ML

Machine Learning is part of the **Artificial Intelligence paradigm**

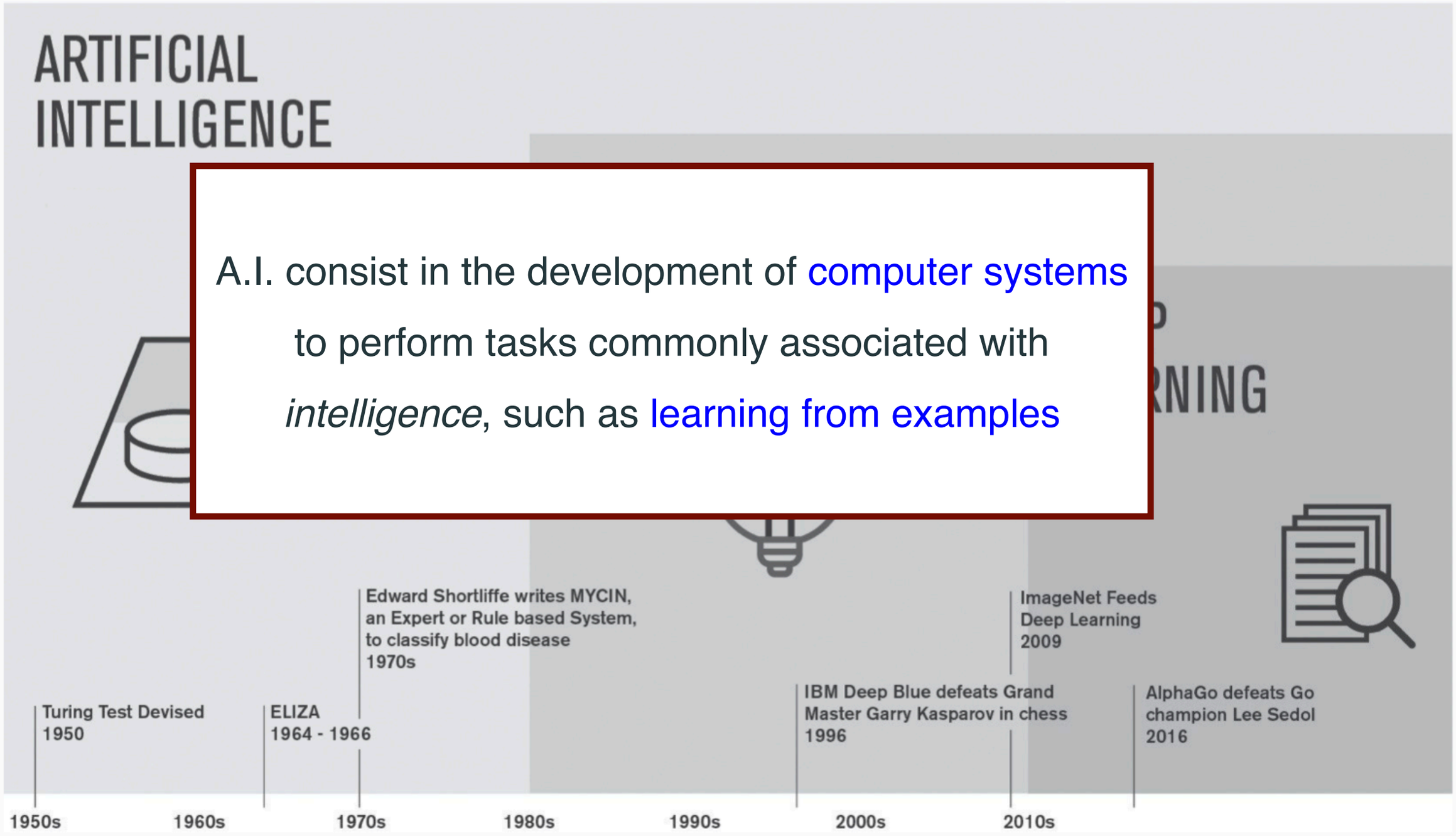
AI is the science and engineering of making intelligent machines (McCarthy '56)



AI vs ML

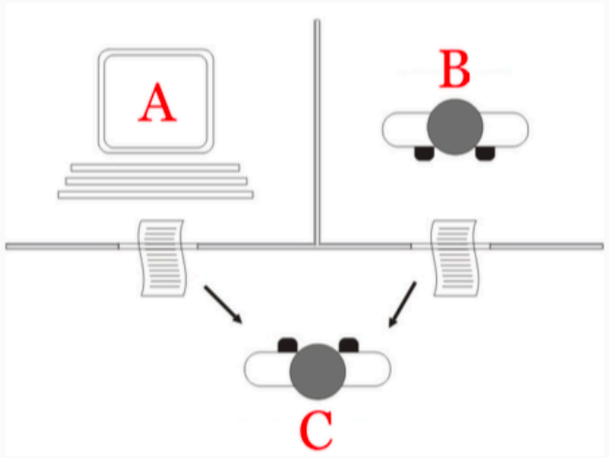
Machine Learning is part of the **Artificial Intelligence paradigm**

AI is the science and engineering of making intelligent machines (McCarthy '56)



Problems in AI

Most problems tackled with Artificial Intelligence fall in **two categories**

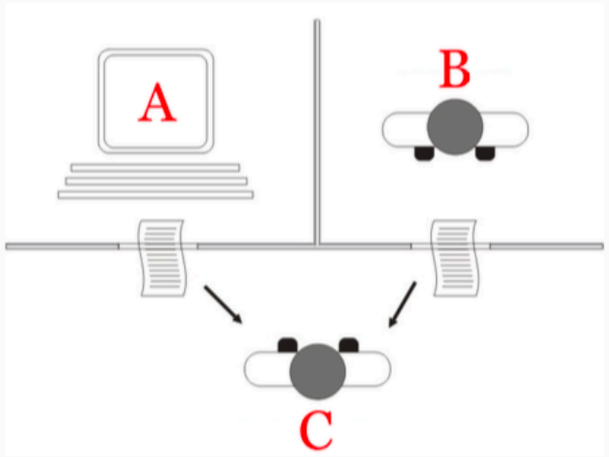


Problems in AI

Most problems tackled with Artificial Intelligence fall in **two categories**

(1) abstract and formal: *easy for computers* but *difficult for humans*

knowledge-based approach



e.g. chess (DeepBlue)

(chess is deterministic game with finite number of options)

(2) intuitive, hard to formalize: *easy for humans* but *difficult for machines*

concept capture and generalisation

e.g. pattern recognition

Problems in AI

Most problems tackled with Artificial Intelligence fall in **two categories**

To excel in tasks which are intuitive to humans but difficult to machines, an A.I. system needs to **acquire its own knowledge**: the Machine is Learning

unlike in chess, where there is no new knowledge to acquire once rules are spelled out

Machine Learning algorithms allow computers the ability to **carry out a task without being explicitly programmed how to do it** by **learning from examples**

(2) intuitive, hard to formalize: *easy for humans* but *difficult for machines*

concept capture and generalisation

e.g. pattern recognition

ML is everywhere

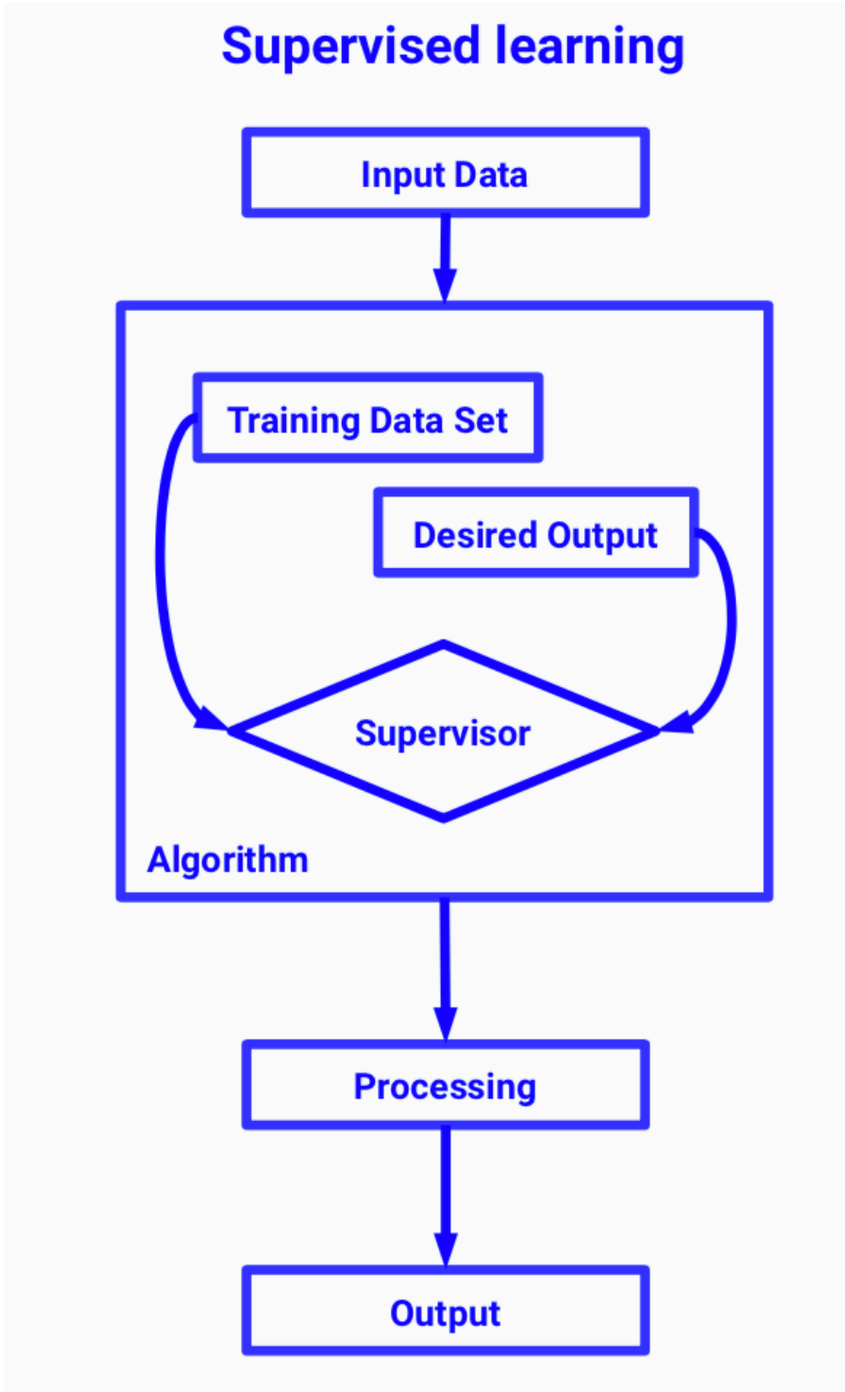
- **Database mining:**
  - Search engines
  - Spam filters
  - Medical and biological records
- **Intuitive tasks for humans:**
  - Autonomous driving
  - Natural language processing
  - Robotics (reinforcement learning)
  - Game playing (DQN algorithms)
- **Human learning:**
  - Concept/human recognition
  - Computer vision
  - Product recommendation

Amazon, Netflix, Google, ...

Learning to learn

Machine Learning algorithms can be divided into several classes, including

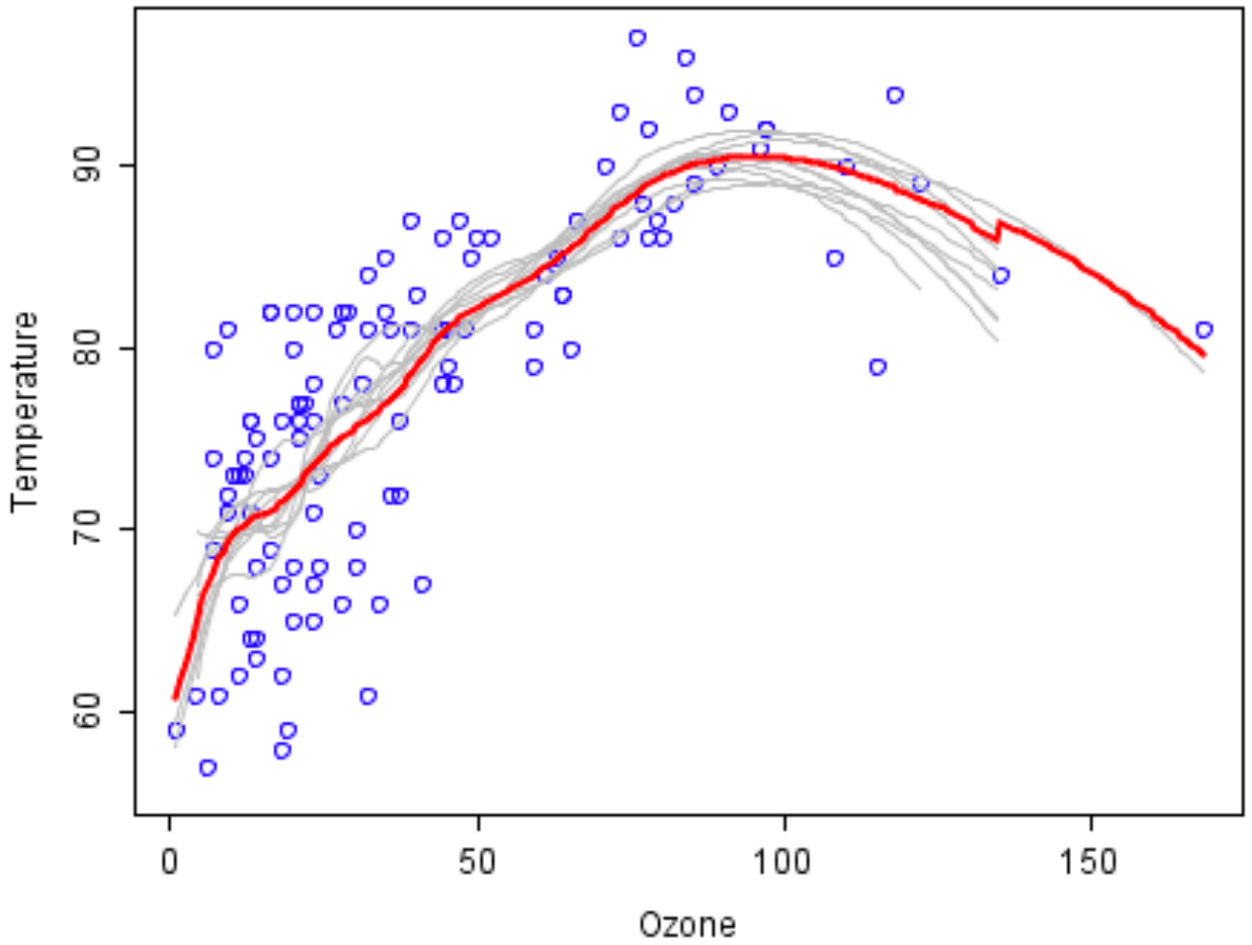
- **Supervised Learning**:
  regression, classification, …

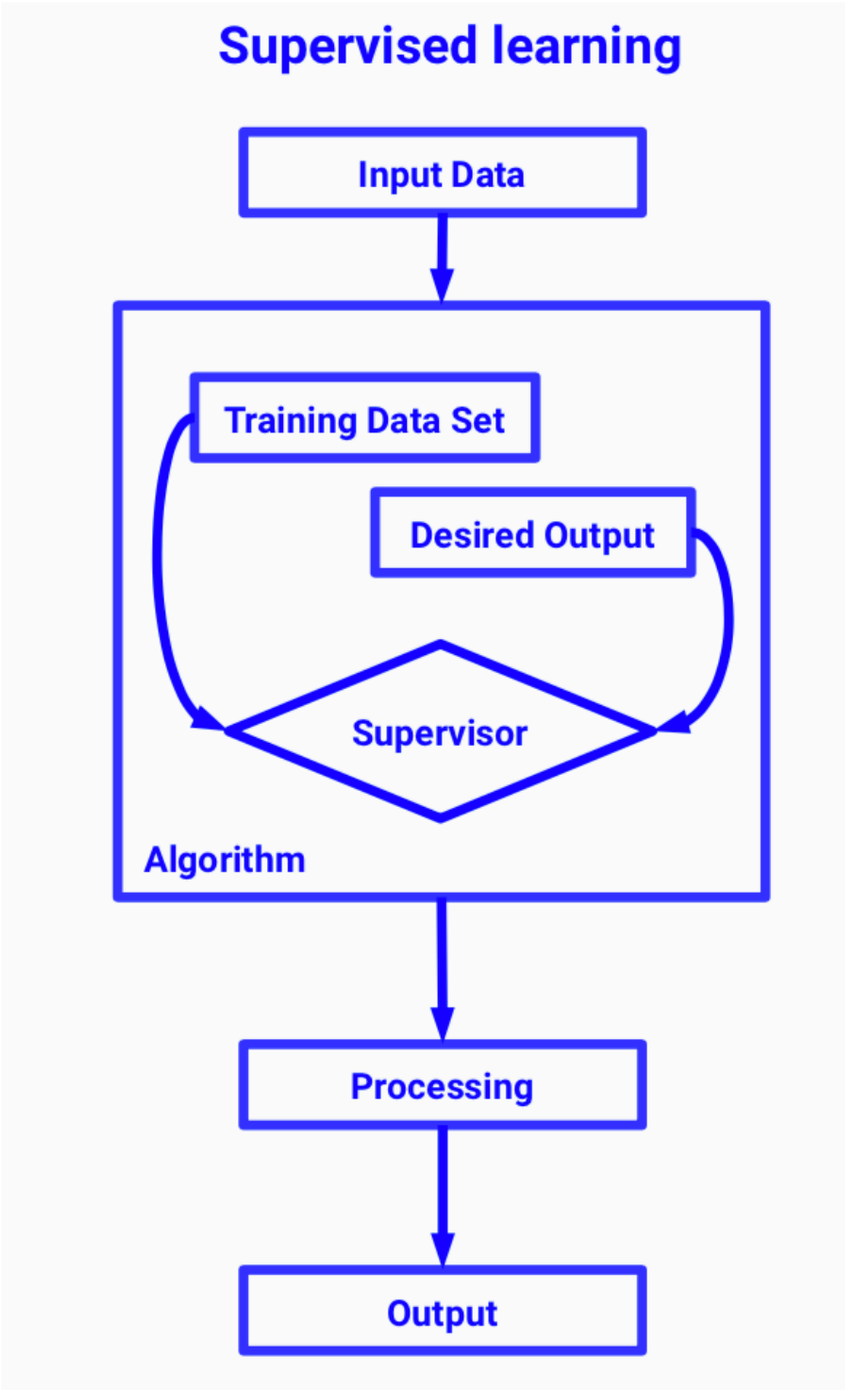


Learning to learn

Machine Learning algorithms can be divided into several classes, including

- **Supervised Learning**:
  regression, classification, …

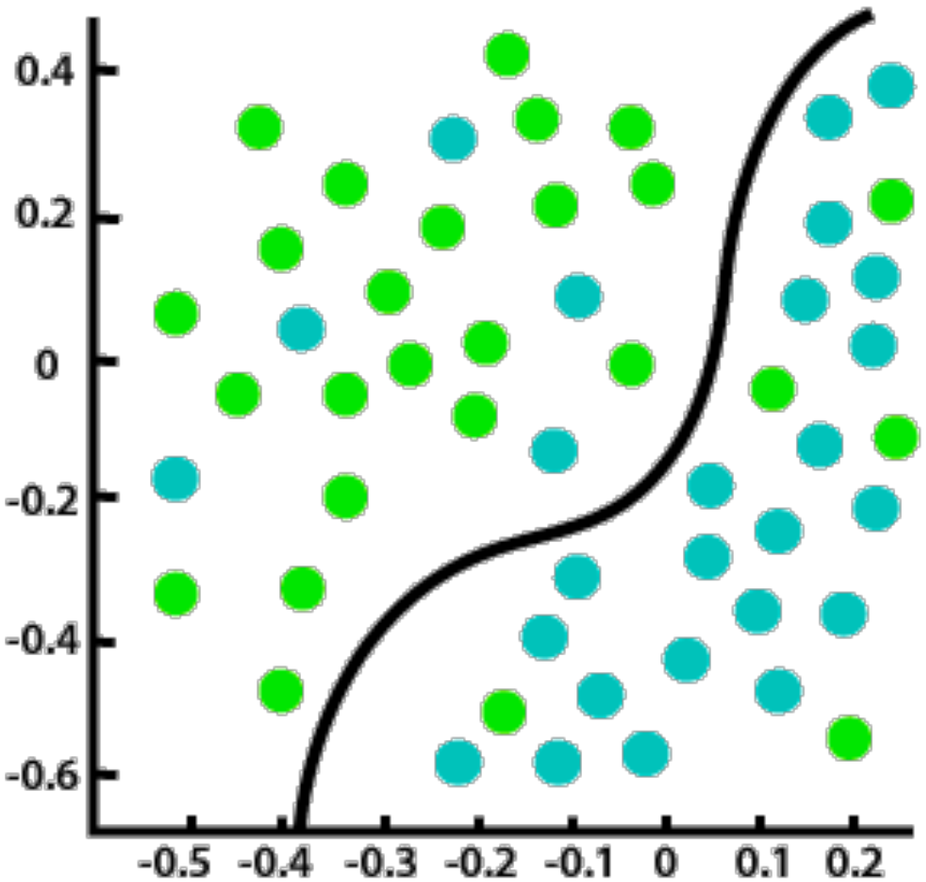




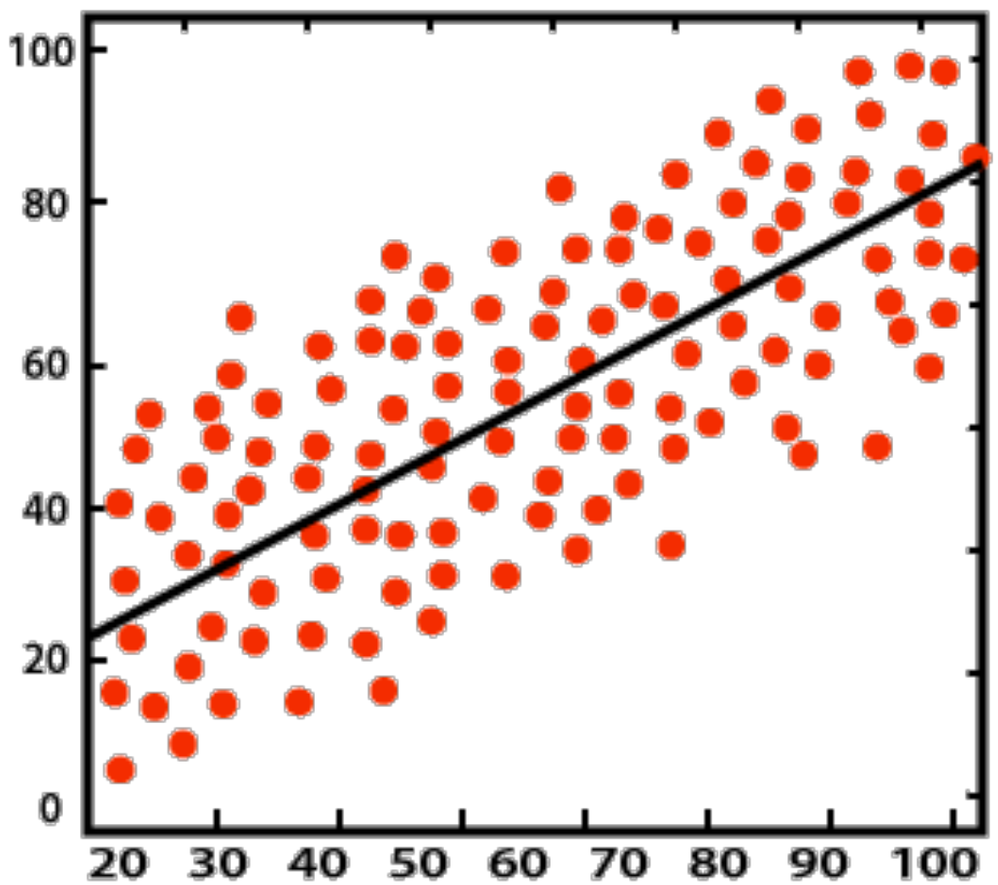
Learning to learn

Machine Learning algorithms can be divided into several classes, including

- **Supervised Learning**:
  regression, classification, …



Classification



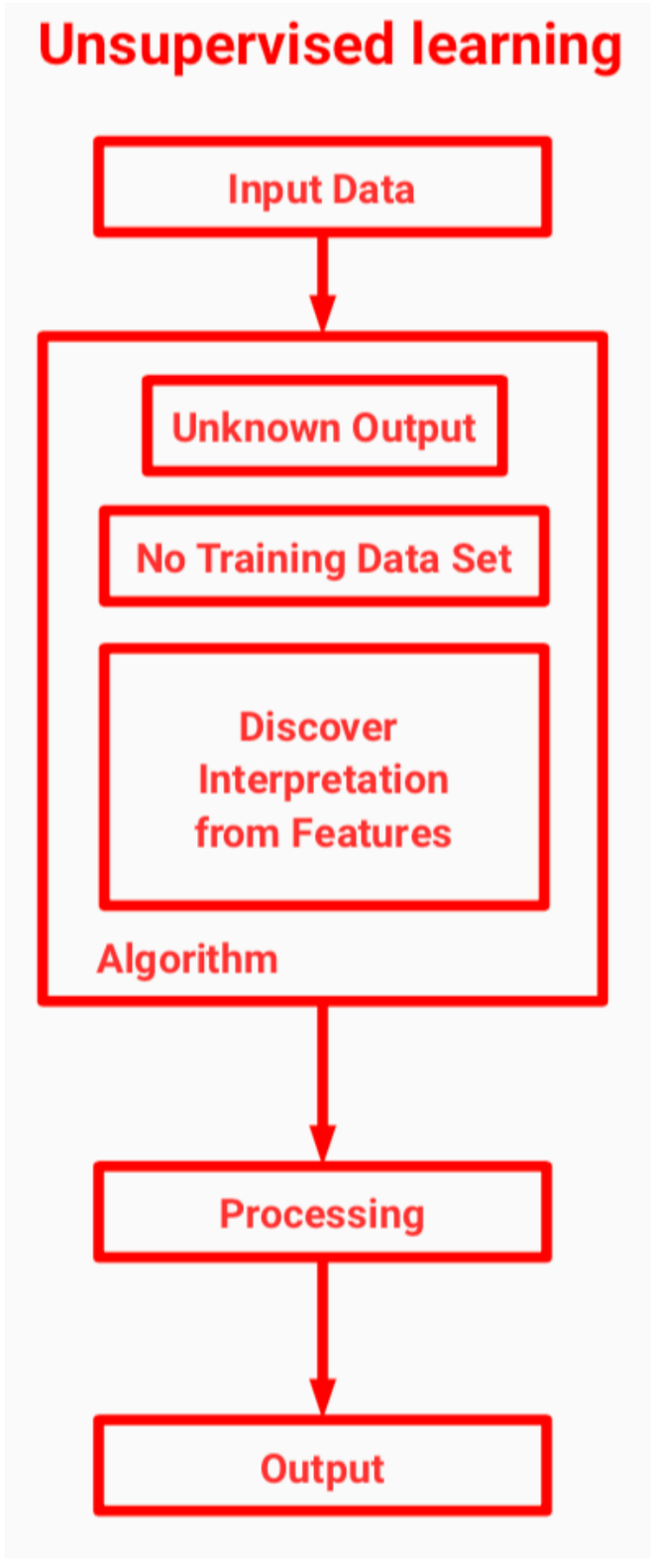
Regression

Learning to learn

Machine Learning algorithms can be divided into several classes, including

- **Supervised Learning**:
  regression, classification, …

- **Unsupervised Learning**:
  clustering, data dimensional reduction, ….

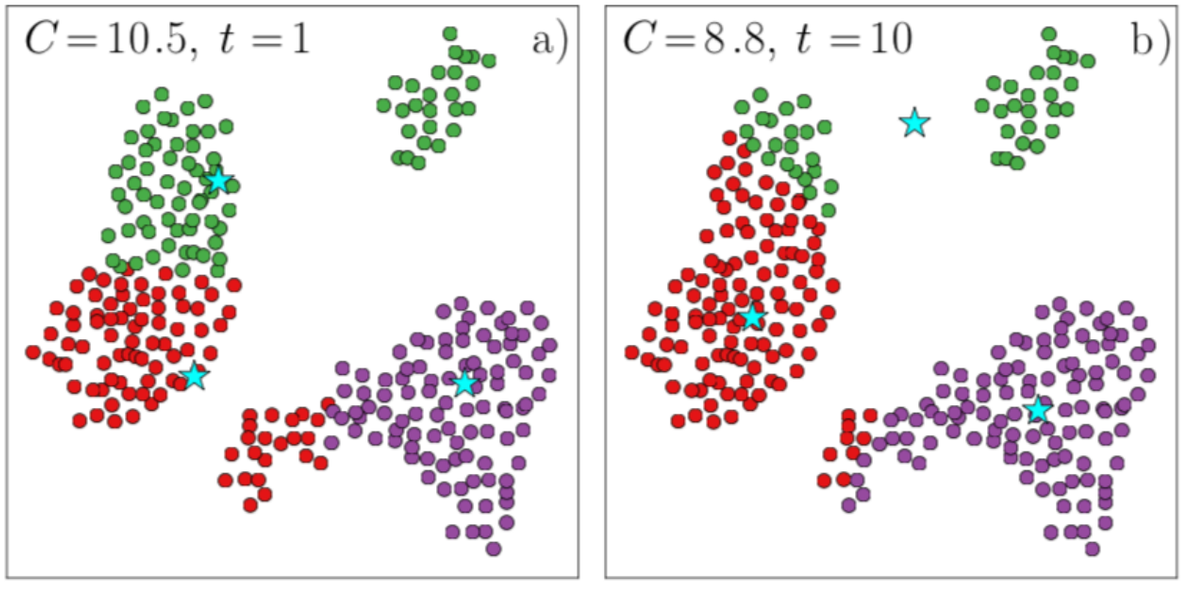


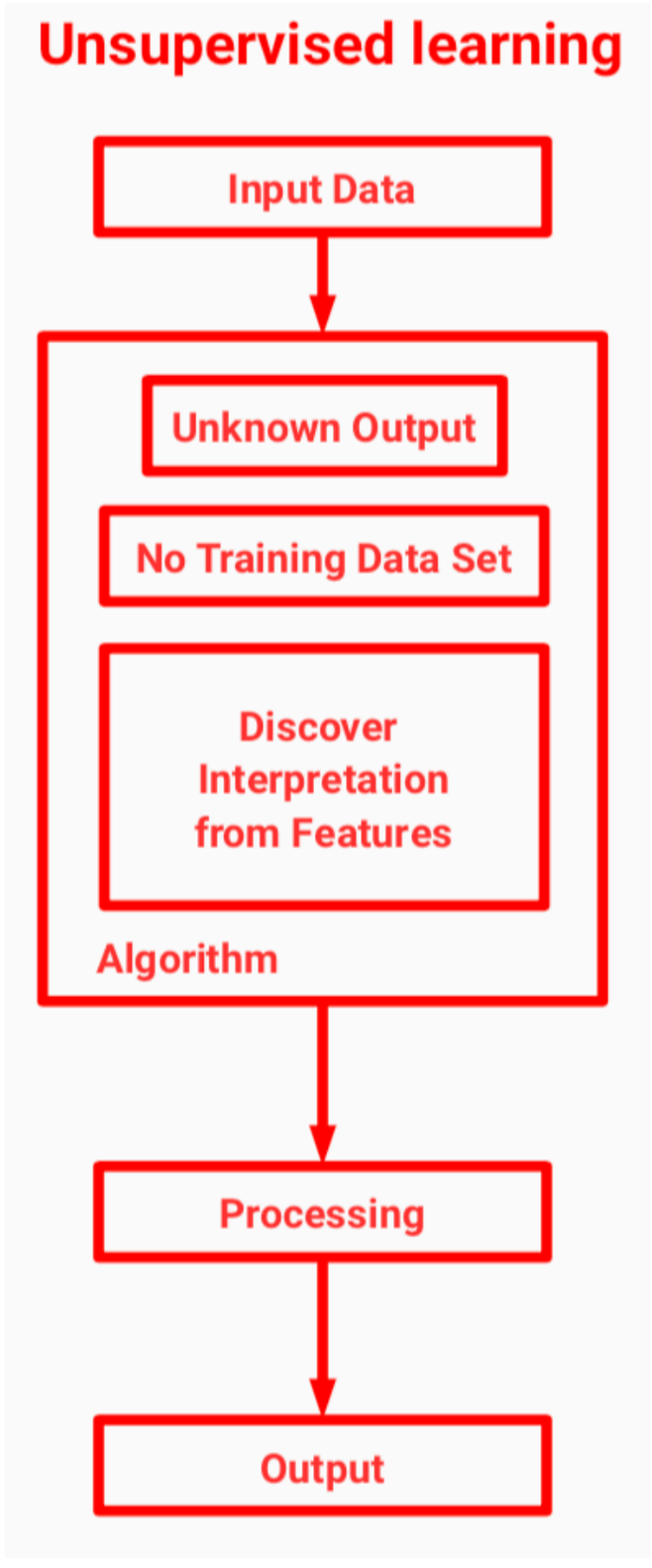
Learning to learn

Machine Learning algorithms can be divided into several classes, including

- **Supervised Learning**:
  regression, classification, …

- **Unsupervised Learning**:
  clustering, data dimensional reduction, ….





Learning to learn

Machine Learning algorithms can be divided into several classes, including

- **Supervised Learning**:
  regression, classification, …

- **Unsupervised Learning**:
  clustering, data dimensional reduction, ….

- **Reinforcement learning**:
  efficiently react to changing environment

Learning to learn

Machine Learning algorithms can be divided into several classes, including

- **Reinforcement learning**: efficiently react to changing environment

The Machine Learning Galaxy II

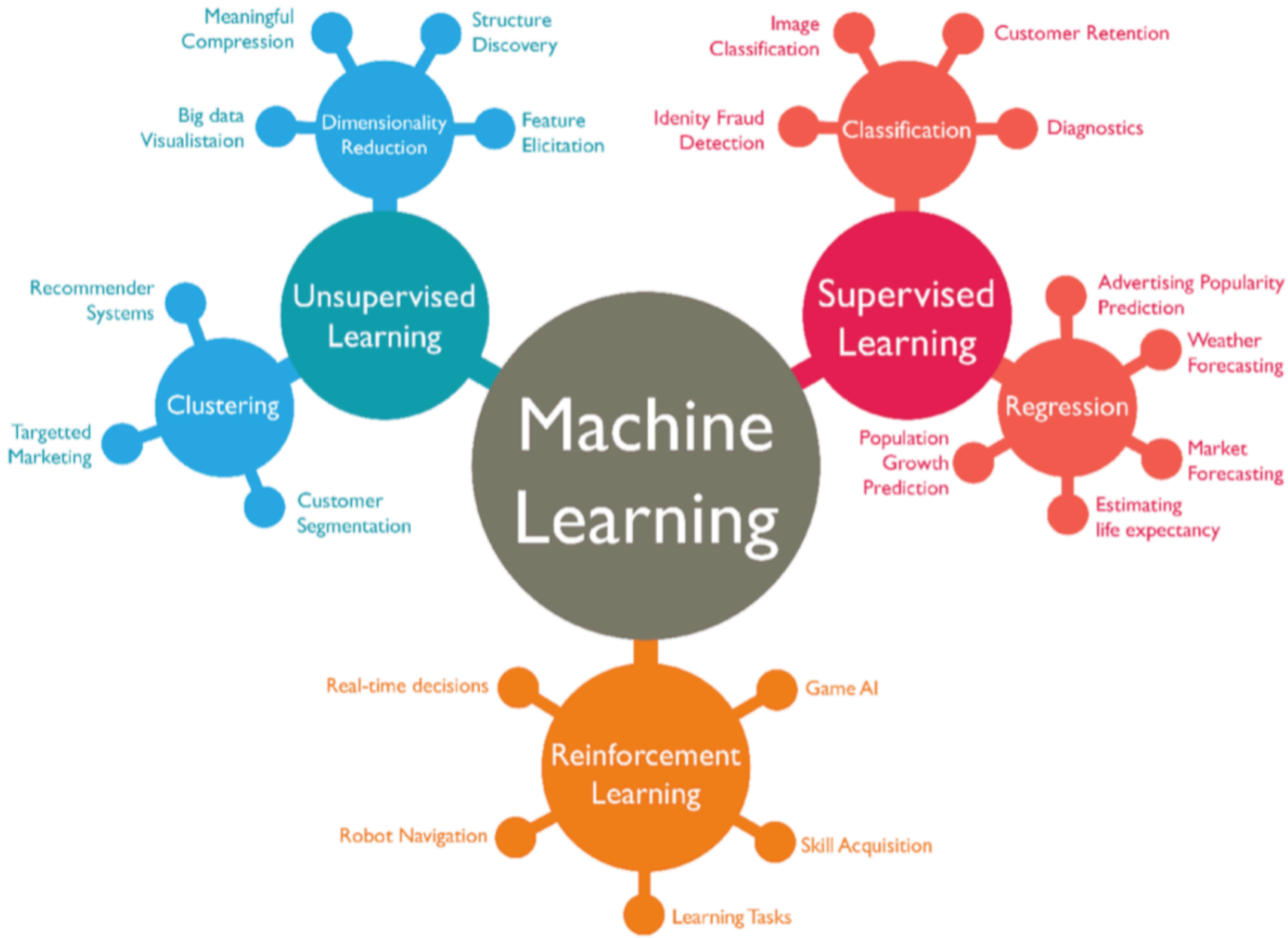


Image via Abdul Rahid

In this course we will have time to cover only subset of ML algorithms and applications!

Supervised Learning: Model Fitting and Regression

Supervised learning

We denote as **supervised learning** the ML task of **learning a function** that maps a **vector of inputs** to a **vector of outputs** from a finite set of training example

note that some assumptions will be needed: a function is an **infinite-dimensional object** but learning takes place from a **finite number of examples**

main property of supervised learning: the **training samples are labeled**

Supervised learning

We denote as **supervised learning** the ML task of **learning a function** that maps a **vector of inputs** to a **vector of outputs** from a finite set of training example

note that some assumptions will be needed: a function is an **infinite-dimensional object** but learning takes place from a **finite number of examples**

main property of supervised learning: the **training samples are labeled**

$$\boldsymbol{x}_i = \left(x_{i,1}, x_{i,2}, \ldots, x_{i,p} \right) \quad \rightarrow \quad y_i$$

data point (with p features) ***label***

the label can be **discrete** (signal/noise, cat/dog) or **continuous** (output of function)

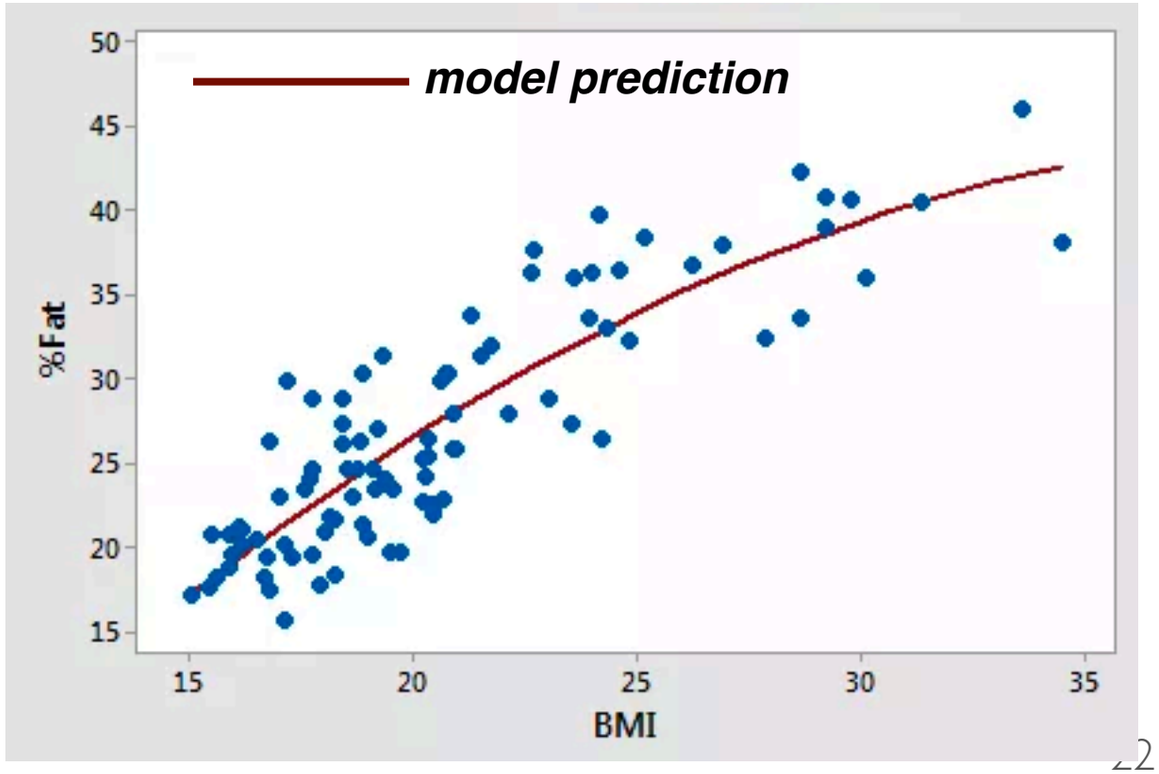
Supervised learning

We denote as **supervised learning** the ML task of **learning a function** that maps a **vector of inputs** to a **vector of outputs** from a finite set of training example

note that some assumptions will be needed: a function is an **infinite-dimensional object** but learning takes place from a **finite number of examples**

main property of supervised learning: the **training samples are labeled**

continuous outputs: regression



data point: BMI

label: % Fat

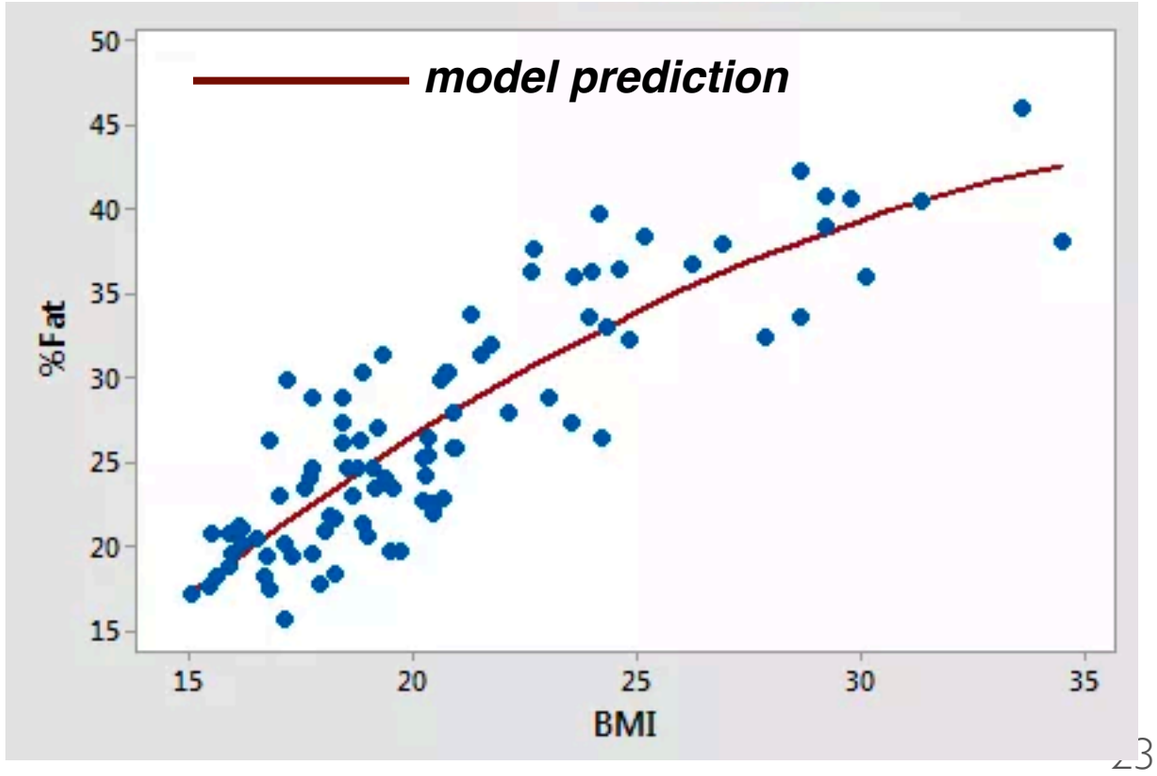
Supervised learning

We denote as **supervised learning** the ML task of **learning a function** that maps a **vector of inputs** to a **vector of outputs** from a finite set of training example

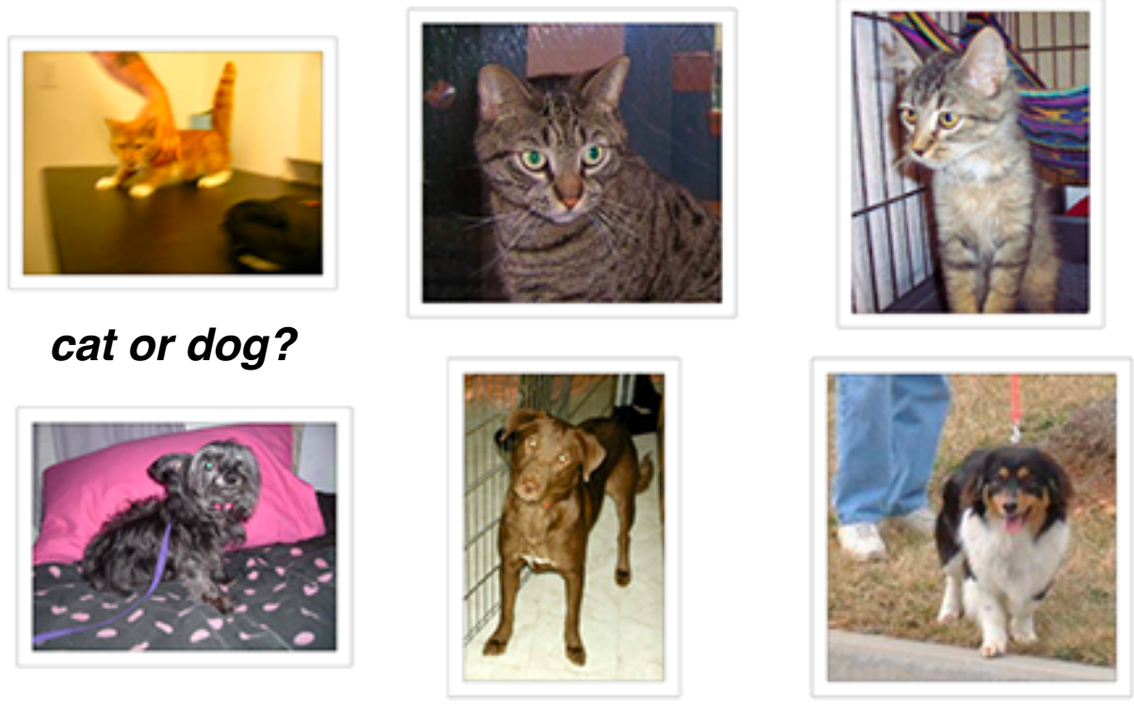
note that some assumptions will be needed: a function is an **infinite-dimensional object** but learning takes place from a **finite number of examples**

main property of supervised learning: the **training samples are labeled**

continuous outputs: regression



discrete outputs: classification



Supervised learning

We denote as **supervised learning** the ML task of **learning a function** that maps a **vector of inputs** to a **vector of outputs** from a finite set of training example

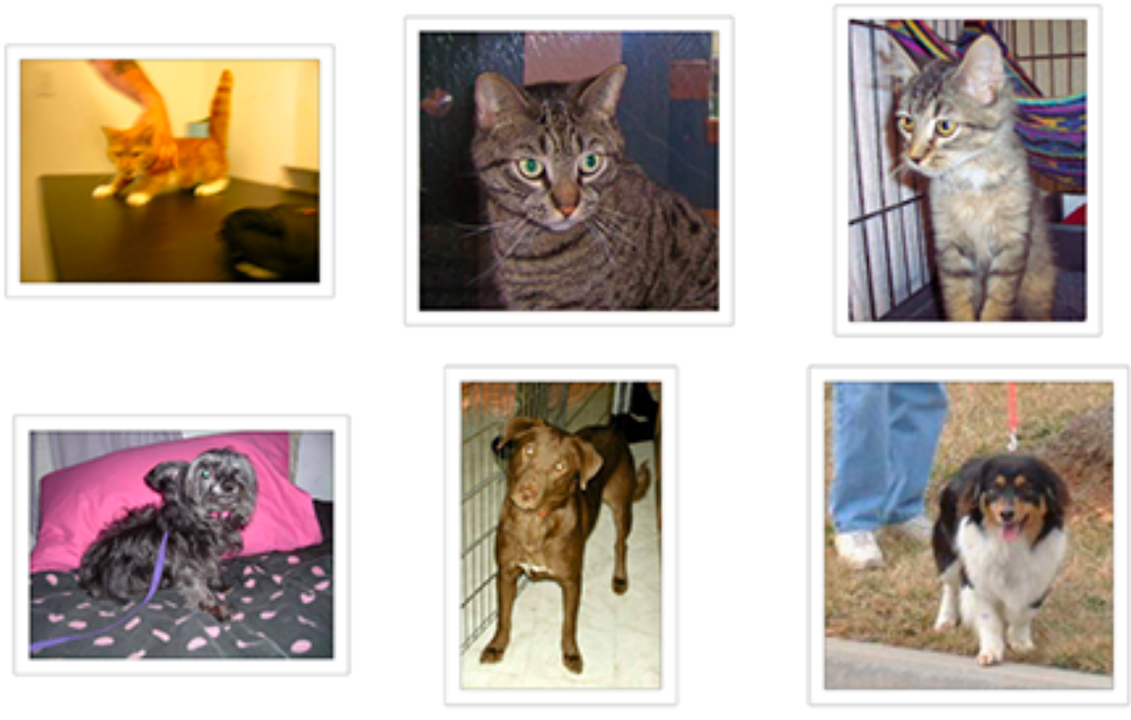
note that some assumptions will be needed: a function is an **infinite-dimensional object** but learning takes place from a **finite number of examples**

main property of supervised learning: the **training samples are labeled**

data point: RGB values of each pixel

label: car or dog

discrete outputs: classification



Setting up the problem

problems in **Supervised Machine Learning** are defined by the following ingredients:

(1) Input dataset: $\mathscr{D} = (\boldsymbol{X}, \boldsymbol{Y})$

*array of **independent** variables*

*array of **dependent** variables*

$$\boldsymbol{X} = \left(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N\right) \qquad \boldsymbol{Y} = \left(y_1, y_2, \ldots, y_N\right)$$

$$\boldsymbol{x}_i = \left(x_{i,1}, x_{i,2}, \ldots, x_{i,p}\right)$$

each independent variable contains p features

Setting up the problem

problems in **Supervised Machine Learning** are defined by the following ingredients:

(1) Input dataset: $\mathscr{D} = (\boldsymbol{X}, \boldsymbol{Y})$

*array of **independent** variables*

$$\boldsymbol{X} = \left(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n\right)$$

*array of **dependent** variables*

$$\boldsymbol{Y} = \left(y_1, y_2, \ldots, y_n\right)$$

$$\boldsymbol{x}_i = \left(x_{i,1}, x_{i,2}, \ldots, x_{i,p}\right)$$

each independent variable contains p features

(2) Model: $f\left(\boldsymbol{X}, \boldsymbol{\theta}\right)$

mapping between dependent and independent variables

$$f : \boldsymbol{X} \rightarrow \boldsymbol{Y}$$

model parameters

$$\boldsymbol{\theta} = \left(\theta_1, \theta_2, \ldots, \theta_m\right)$$

the more complex the problem, the more flexible the model

Setting up the problem

problems in **Supervised Machine Learning** are defined by the following ingredients:

(1) Input dataset: $\mathscr{D} = (\boldsymbol{X}, \boldsymbol{Y})$

(2) Model: $f\left(\boldsymbol{X}, \boldsymbol{\theta}\right)$

(3) Cost function: $C\left(\boldsymbol{Y}; f(\boldsymbol{X}; \boldsymbol{\theta}\right)$

The cost function measures how well the model (for a specific choice of its parameters) is able to **describe the input dataset**

example of cost function for single dependent variable: sum of residuals squared

$$C\left(\boldsymbol{Y}; f(\boldsymbol{X}; \boldsymbol{\theta}\right) = \frac{1}{n} \sum \left(y_i - f(\boldsymbol{x}_i, \boldsymbol{\theta})\right)^2$$

Setting up the problem

problems in **Supervised Machine Learning** are defined by the following ingredients:

(1) Input dataset: $\mathscr{D} = (\boldsymbol{X}, \boldsymbol{Y})$

(2) Model: $f\left(\boldsymbol{X}, \boldsymbol{\theta}\right)$

(3) Cost function: $C\left(\boldsymbol{Y}; f(\boldsymbol{X}; \boldsymbol{\theta}\right)$

The cost function measures how well the model (for a specific choice of its parameters) is able to describe the input dataset

Fitting the model means determining the values of its parameters which **minimise the cost function**

$$\left.\frac{\partial C\left(\boldsymbol{Y}; f(\boldsymbol{X}; \boldsymbol{\theta}\right)}{\partial \theta_i}\right|_{\theta=\theta_{\text{opt}}} = 0$$

Model fitting

before dealing with more sophisticated samples of Machine Learning, let's illustrate the main aspects of model fitting with a simple example: **polynomial regression in 1D**

First of all we can generate data following a **known underlying law** and then adding **stochastic noise**, to emulate a realistic situation

$$y_i = f(x_i) + \eta_i\,, \qquad i = 1,\ldots,n$$

dependent *variable*

*the **physical law** (might be known)*

independent *variable*

uncorrelated Gaussian variable

$$\langle \eta_i \rangle = 0$$

$$\langle \eta_i \eta_j \rangle = \sigma^2 \delta_{ij}$$

noise strength

The larger the value of the **noise *σ*** compared to the variations of the **law *f,*** the noisier the data

Model fitting

in the real world we would have only the information about the ``measurements''

$$\mathscr{D} = (\boldsymbol{X}, \boldsymbol{Y})$$

and our goal is to extract the **underlying physical law** from this data

We need to define **classes of models** that might provide a good description of the data, in a way that its parameters encode the main features of the physical law

$$f_\alpha(x;\, \boldsymbol{\theta}_\alpha)$$

model prediction — *label of model class* — *parameters of model class α*

we want to compare the performance of models with **different complexities:** different functional forms, number of parameters, intrinsic flexibility, ….

Model fitting

The simplest possible model is a polynomial: **polynomial regression**

$$f_\alpha(x;\boldsymbol{\theta}_\alpha) = \sum_{j=0}^{N_\alpha} \theta_{\alpha,j} x^j$$

for example the model class that contains all possible cubic polynomials is

$$f_3(x;\boldsymbol{\theta}_3) = \sum_{j=0}^{3} \theta_{3,j} x^j$$

a more complex model does not necessarily imply a more predictive one: the appropriate amount of complexity depends on the **features of the data sample** (*e.g.* size, variability)

in polynomial regression the model parameter are given by **least-squares method**

$$\hat{\theta} = \arg\ \min_\theta \left\{ \sum_{i=1}^{n} \left(y_i - f_\alpha(x_i;\boldsymbol{\theta}_\alpha) \right)^2 \right\}$$

minimise sum of residuals squared

Model fitting

Simples case: **least-squares method** for order-one polynomials

$$f_q(x;\boldsymbol{\theta}_1) = \sum_{j=0}^{1} \theta_{\alpha,j} x^j = \theta_{1,0} + \theta_{1,1} x$$

which is nothing but the **linear regression** taught in first-year statistics

$$\widehat{\theta} = \arg\min_{\theta} \left\{ \sum_{i=1}^{n} \left(y_i - \theta_{1,0} - \theta_{1,1} x \right)^2 \right\}$$

where you can compute analytically the **best-fit values of the coefficients**

Model fitting

Simples case: **least-squares method** for order-one polynomials

$$\left.\frac{\partial}{\partial\theta_0}\sum_{i=1}^{n}\left(y_i-\theta_0-\theta_1 x_i\right)^2\right|_{\widehat{\theta_0}}=0$$

$$\left.\frac{\partial}{\partial\theta_1}\sum_{i=1}^{n}\left(y_i-\theta_0-\theta_1 x_i\right)^2\right|_{\widehat{\theta_1}}=0$$

and by solving this linear system of equations one obtains the **model parameters**

$$\widehat{\theta_1}=\frac{\langle xy\rangle-\langle x\rangle\langle y\rangle}{\langle x^2\rangle-\langle x\rangle^2}$$

$$\widehat{\theta_0}=\langle y\rangle-\widehat{\theta_1}\,\langle x\rangle$$

averages computed ever the n elements of the data sample

The need for regularisation

Best-fit model

What is the best strategy to **determine the model parameters**?

Best-fit model

What is the best strategy to **determine the model parameters**?

Seems a silly question, surely those are simply **minimum of cost function?**

$$\hat{\theta} = \arg\ \min_{\theta}\left\{ \sum_{i=1}^{n} \left(y_i - f_\alpha(x_i; \boldsymbol{\theta}_\alpha)\right)^2 \right\}$$

Best-fit model

What is the best strategy to **determine the model parameters**?

Seems a silly question, surely those are simply **minimum of cost function?**

$$\hat{\theta} = \arg\min_{\theta}\left\{ \sum_{i=1}^{n} \left(y_i - f_\alpha(x_i; \boldsymbol{\theta}_\alpha)\right)^2 \right\}$$

However this is in general **not the case**, because:

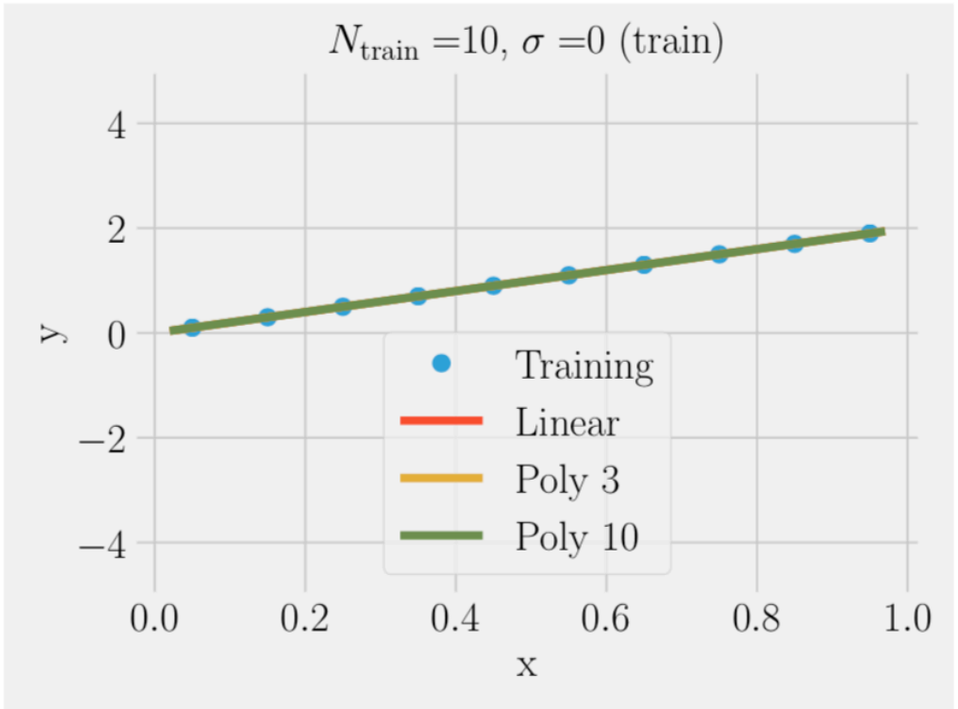
- Real world data is **noisy**: we want to **learn the underlying law**, not the statistical fluctuations

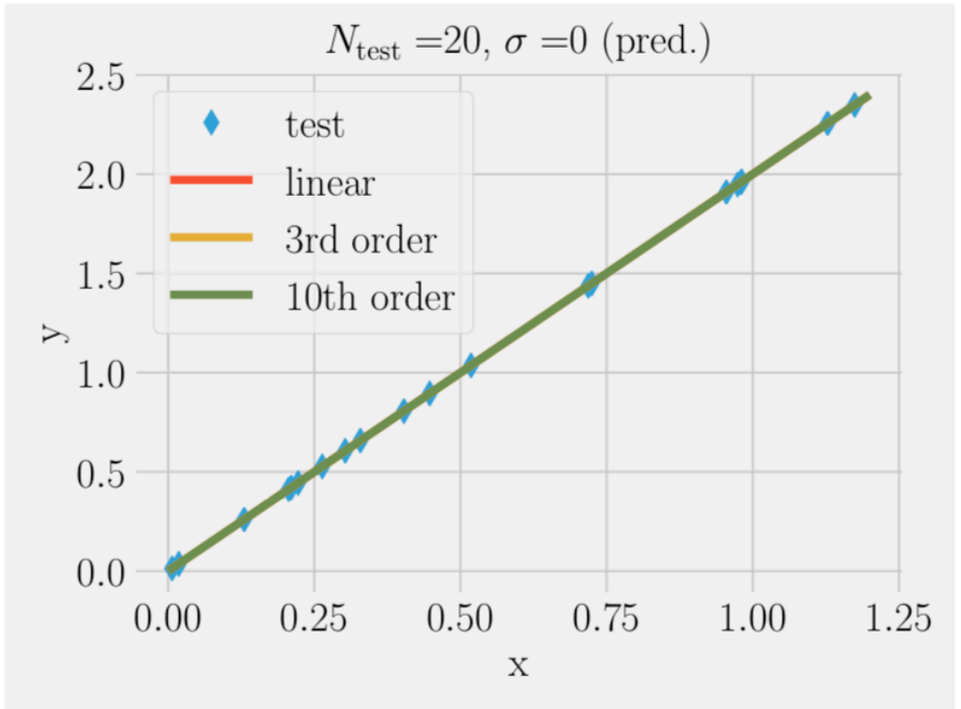
- More than fitting the data, our real goal is to create a model that **predicts future/ different data:** we need figures of merit outside the training dataset!

- To ensure that our model describes the underlying law (and thus one can safely generalise) rather than the noise, a **regularisation procedure** needs to be used

Model fitting

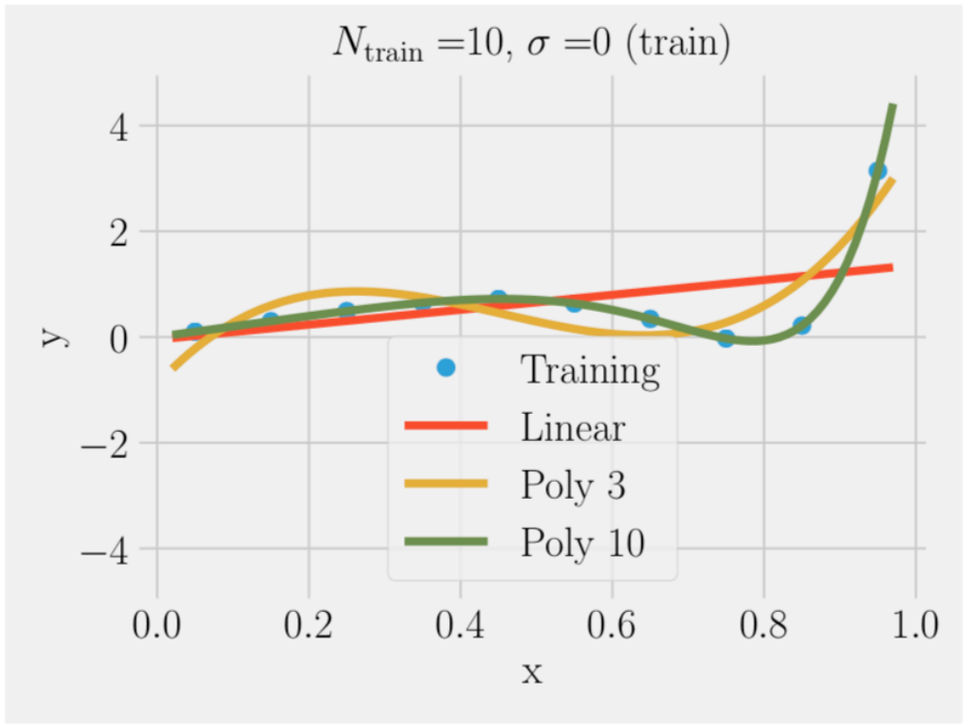
$f(x) = 2x\,, \quad x \in [0,1]$

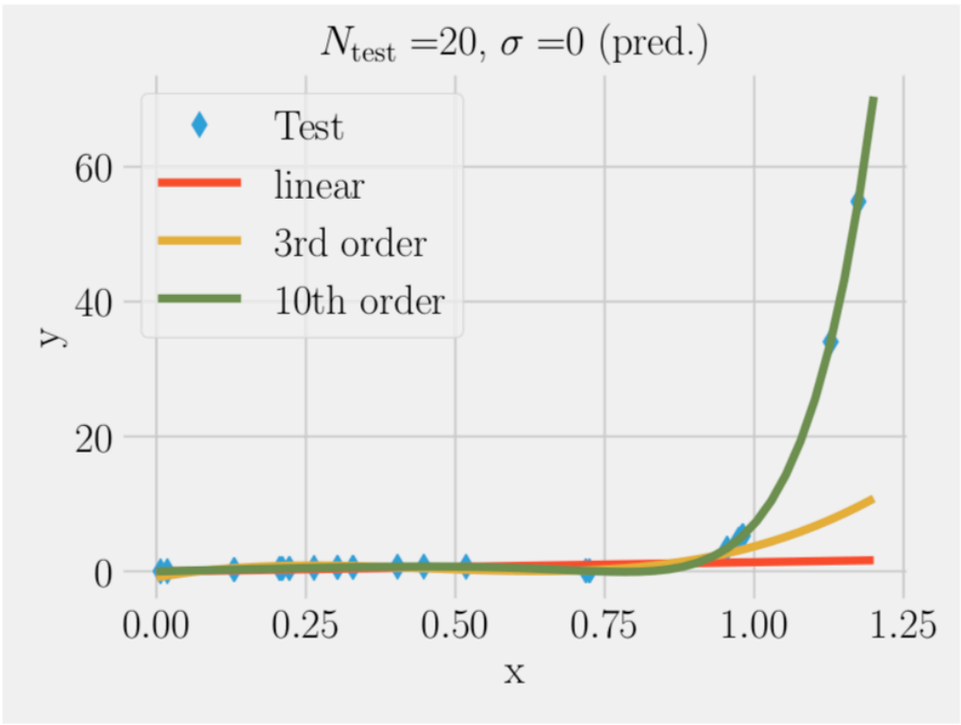




$f(x) = 2x - 10x^5 + 15x^{10}\,, \quad x \in [0,1]$

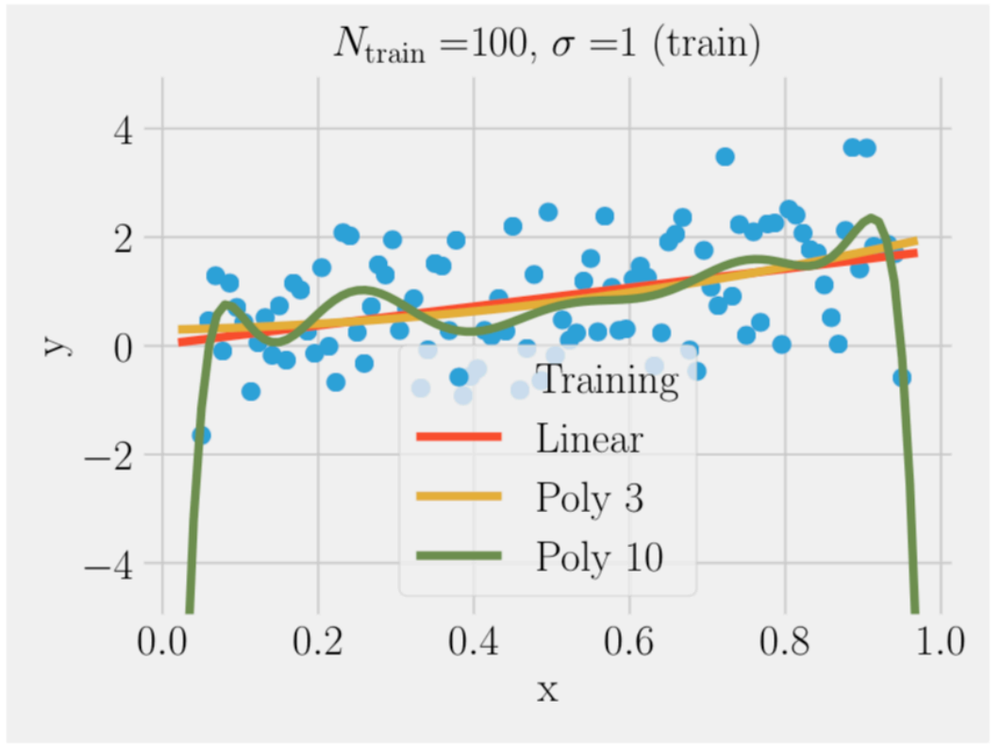
in the absence of noise, fitting and predicting are identical, provided model has enough flexibility

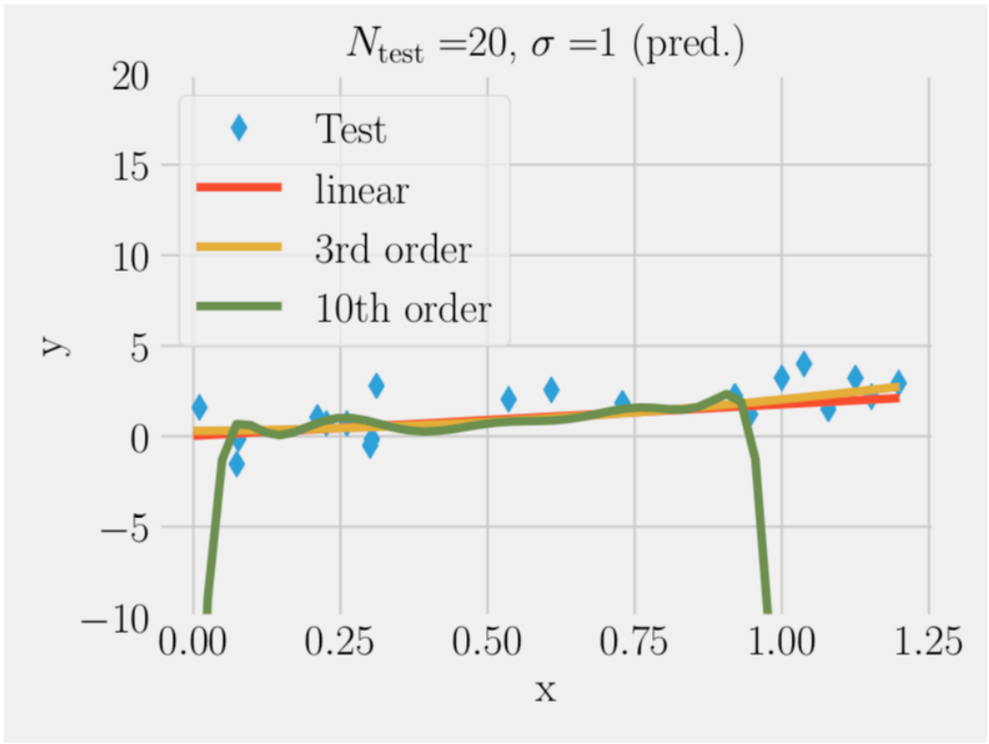




Model fitting

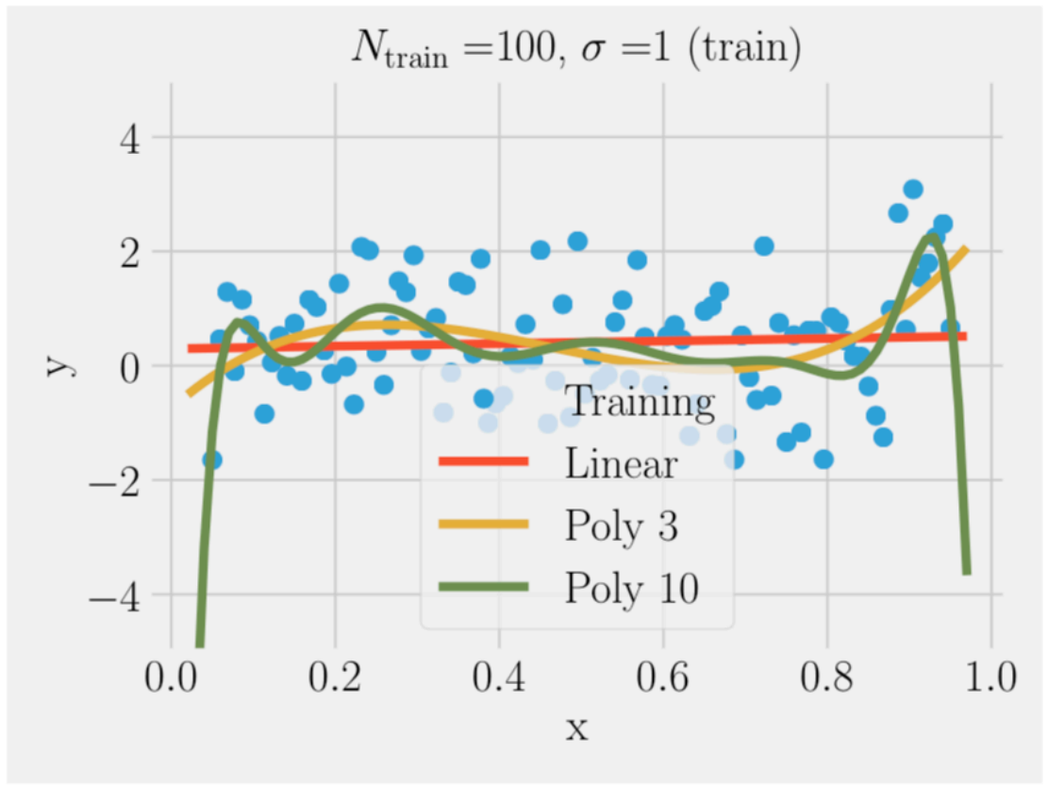
$f(x) = 2x\,, \quad x \in [0,1]$

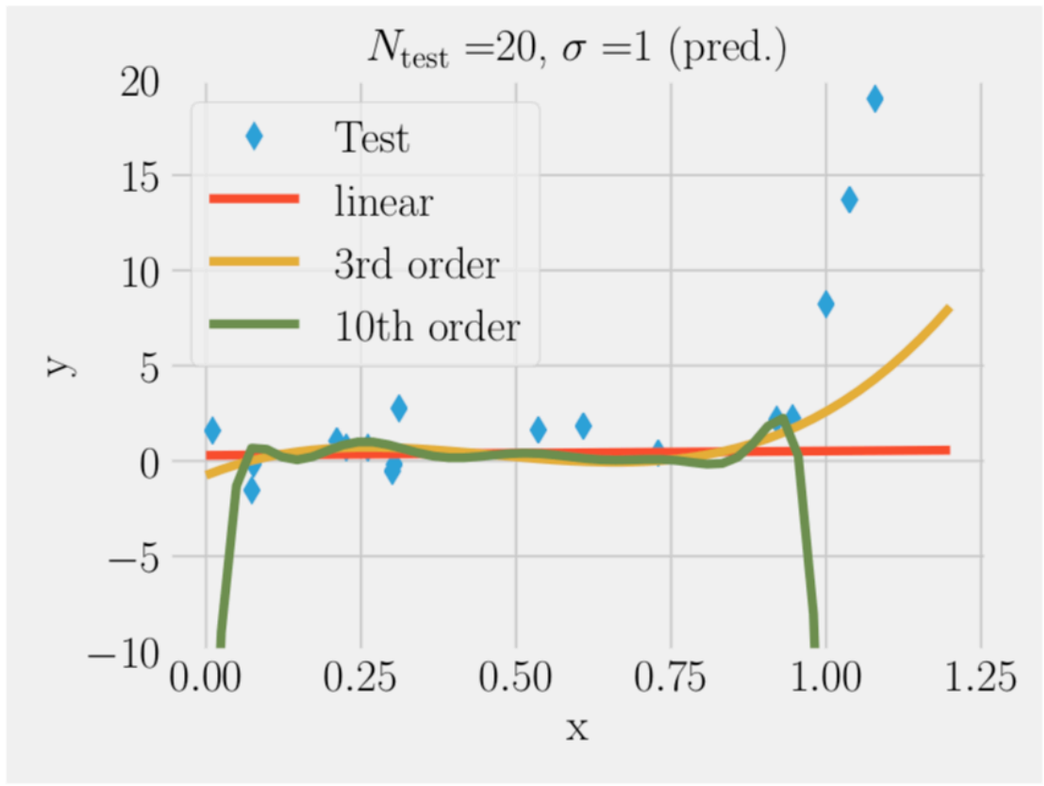




$f(x) = 2x - 10x^5 + 15x^{10}\,, \quad x \in [0,1]$

in the presence of noise, models with less complexity can exhibit improved predictive power





Cross-validation: regularisation

What is the best strategy to determine the model parameters?

first of all, divide input dataset into **two disjoint sets**

$$\mathscr{D} = \mathscr{D}_{\text{tr}} + \mathscr{D}_{\text{val}}$$

training dataset:

used to extract model parameters

validation dataset:

used to monitor generalisation power

the model parameters are determined using information from training dataset

$$\widehat{\theta} = \arg\ \min_{\boldsymbol{\theta}} \{ C\left(\boldsymbol{Y}_{\text{th}}, f(\boldsymbol{X}_{\text{tr}}; \boldsymbol{\theta}) \right) \}$$

while the **performance of the model** (generalisation, extrapolation) is evaluated by computing the cost function on the validation dataset

$$C\left(\boldsymbol{Y}_{\text{val}}, f(\boldsymbol{X}_{\text{val}}; \widehat{\theta}) \right)$$

Cross-validation (regularisation)

In-sample (training) error ⟶ $$E_{\text{tr}} \equiv C\left(\boldsymbol{Y}_{\text{tr}}, f(\boldsymbol{X}_{\text{tr}}; \hat{\boldsymbol{\theta}})\right)$$

Out-of-sample (validation) error ⟶ $$E_{\text{val}} \equiv C\left(\boldsymbol{Y}_{\text{val}}, f(\boldsymbol{X}_{\text{val}}; \hat{\boldsymbol{\theta}})\right)$$

Splitting the data into mutually exclusive training and validation sets provides an unbiased estimate for the **predictive performance of the model**

In ML problems one should select the model that **minimises** the out-of-sample error $\boldsymbol{E_{val,}}$ since this is the model that generalises in the most efficient way

note that choices on how to partition the dataset will affect the conclusion about which is best model

Cross-validation (regularisation)

In-sample (training) error $\longrightarrow$ $$E_{\text{tr}} \equiv C\left(\boldsymbol{Y}_{\text{tr}}, f(\boldsymbol{X}_{\text{tr}}; \hat{\boldsymbol{\theta}})\right)$$

Out-of-sample (validation) error $\longrightarrow$ $$E_{\text{val}} \equiv C\left(\boldsymbol{Y}_{\text{val}}, f(\boldsymbol{X}_{\text{val}}; \hat{\boldsymbol{\theta}})\right)$$

Splitting the data into mutually exclusive training and validation sets provides an unbiased estimate for the **predictive performance of the model**

In ML problems one should select the model that **minimises** the out-of-sample error $\boldsymbol{E_{val,}}$ since this is the model that generalises in the most efficient way

Fitting is not predicting: in general the model that describes better a given set of data will not be the one that generalises and predicts better related datasets

Regularisation

In ML, the common goal of regularisation strategies is to modify the learning algorithms to **reduce its generalisation error without increasing its training error**

$$C(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - f(\boldsymbol{x}_i;\boldsymbol{\theta})\right)^2$$

unregularised cost function

$$C(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - f(\boldsymbol{x}_i;\boldsymbol{\theta})\right)^2 + \lambda\boldsymbol{\theta}^T\boldsymbol{\theta}$$

regularised cost function

regularisation hyperparameter, to be tuned

weight-decay regularisation: model parameters with smaller L2-norms are preferred

Regularisation

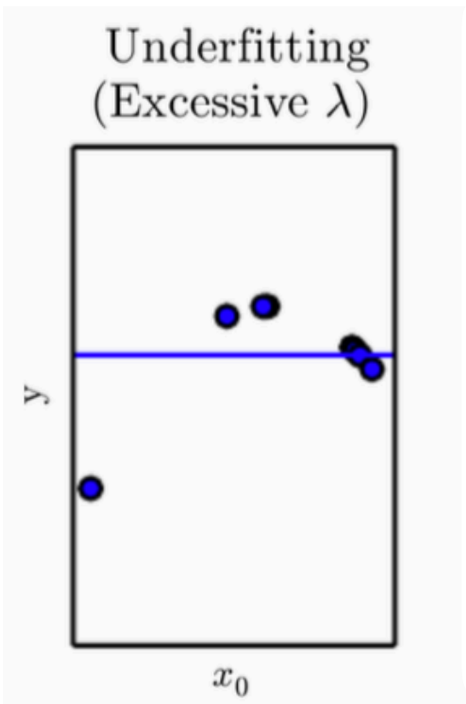
In ML, the common goal of regularisation strategies is to modify the learning algorithms to **reduce its generalisation error without increasing its training error**

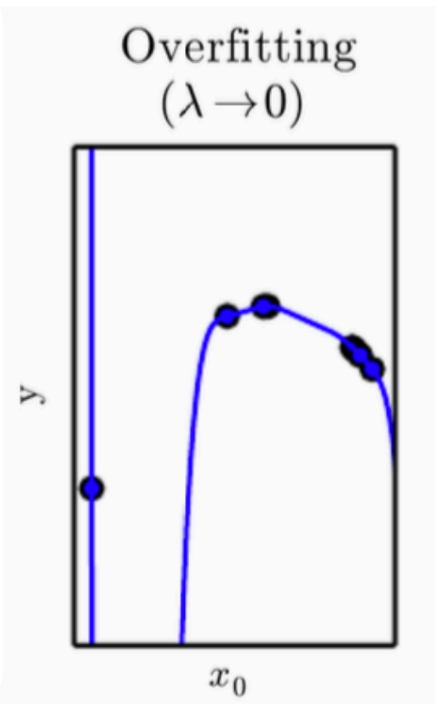
$$C(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - f(\boldsymbol{x}_i;\boldsymbol{\theta})\right)^2$$

unregularised cost function

$$C(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - f(\boldsymbol{x}_i;\boldsymbol{\theta})\right)^2 + \lambda\boldsymbol{\theta}^T\boldsymbol{\theta}$$

regularised cost function





no free lunch, careful tuning of regularisation required

Regularisation

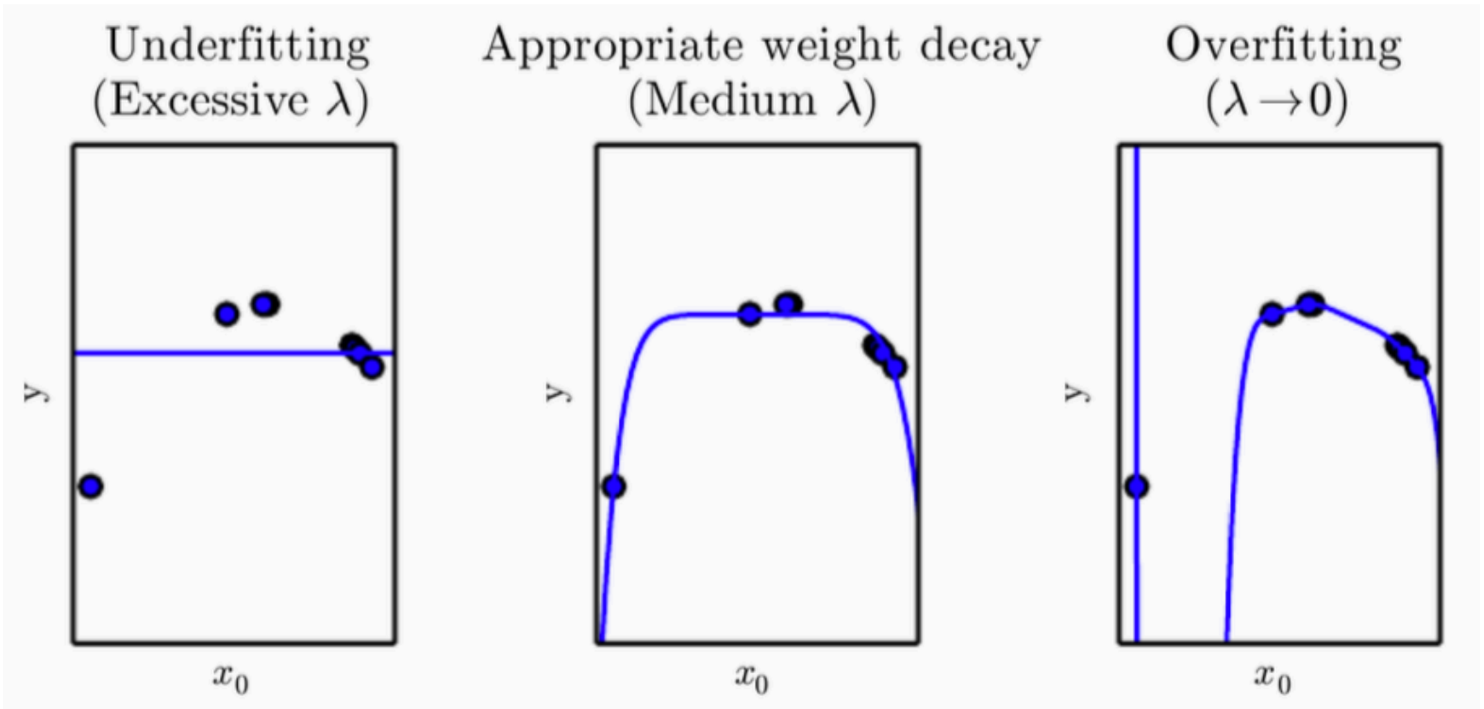
In ML, the common goal of regularisation strategies is to modify the learning algorithms to **reduce its generalisation error without increasing its training error**

$$C(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - f(\boldsymbol{x}_i;\boldsymbol{\theta})\right)^2$$

unregularised cost function

$$C(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - f(\boldsymbol{x}_i;\boldsymbol{\theta})\right)^2 + \lambda\boldsymbol{\theta}^T\boldsymbol{\theta}$$

regularised cost function



no free lunch, careful tuning of regularisation required

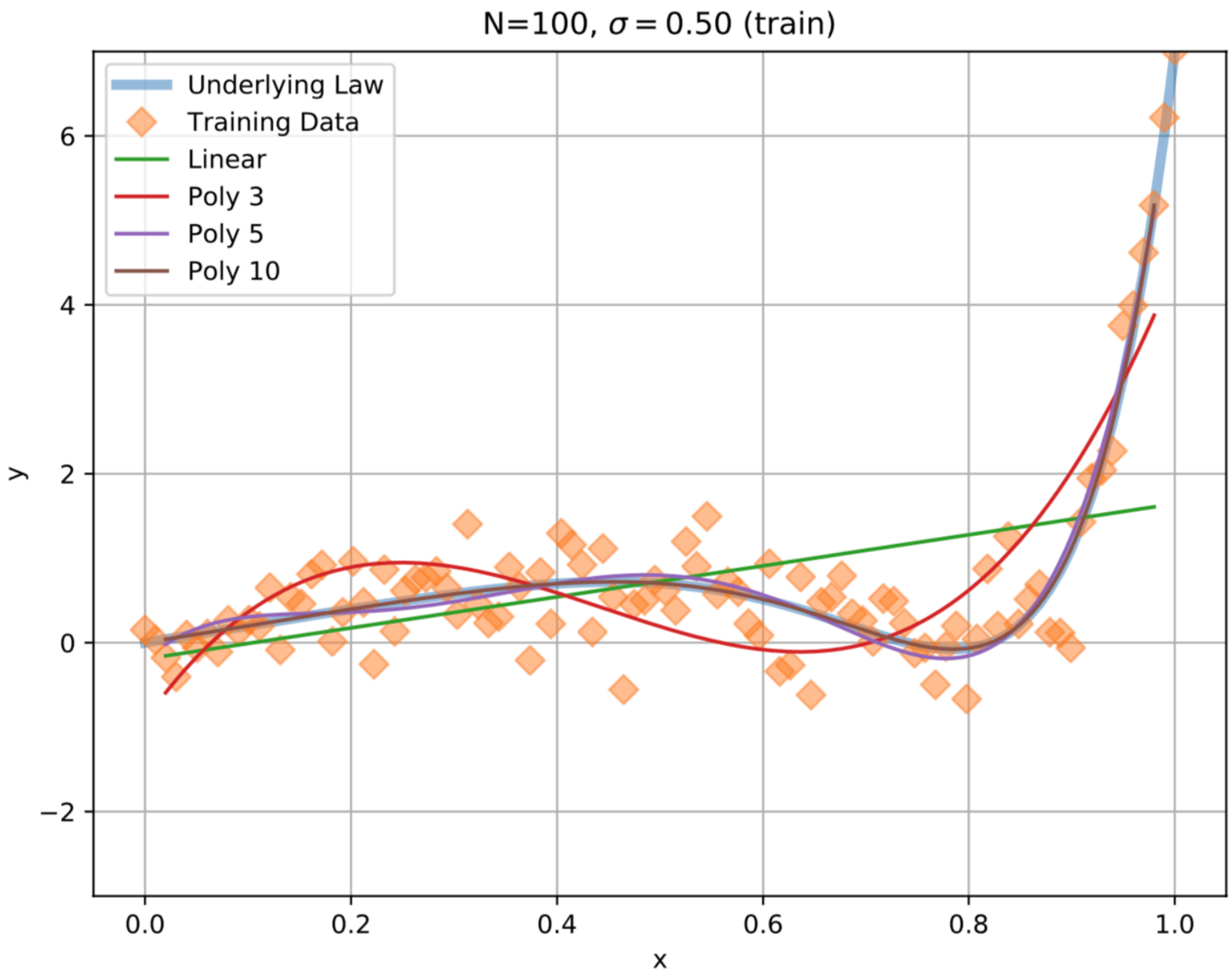
Tutorial 1

- Generate data with different underlying laws and try to model it using **polynomial regression**

- Study how the cost function to the training data depends on *i)* the complexity of the underlying law, *ii)* the flexibility of the model, *iii)* the number of training points

- Now generate a validation dataset, either within the training data range [0,1] or outside it. Study which model is **more predictive** as a function of the same parameters as before in the two cases

- Can you **identify overfitting**? Can you think of a way of avoiding it?



Why Machine Learning is difficult

- **Fitting existing data** is conceptually different from **making predictions about new data**

- Increasing the model complexity can improve the description of the training data but **reduce the predictive power** of the model due to overfitting, unless a suitable regularisation strategy is implemented

- For complex and/or small datasets, simple models can be better at prediction than complex models. The ``**right" amount of complexity** cannot in general be determined from first principles

- It is difficult to **generalise** beyond the situations encountered in the training set: the model cannot learn what it has not seen

Why Machine Learning is difficult

- **Fitting existing data** is conceptually different from **making predictions about new data**

- Increasing the model complexity can improve the description of the training data but **reduce the predictive power** of the model due to overfitting, unless a suitable regularisation strategy is implemented

- For complex and/or small datasets, simple models can be better at prediction than complex models. The ``**right" amount of complexity** cannot in general be determined from first principles

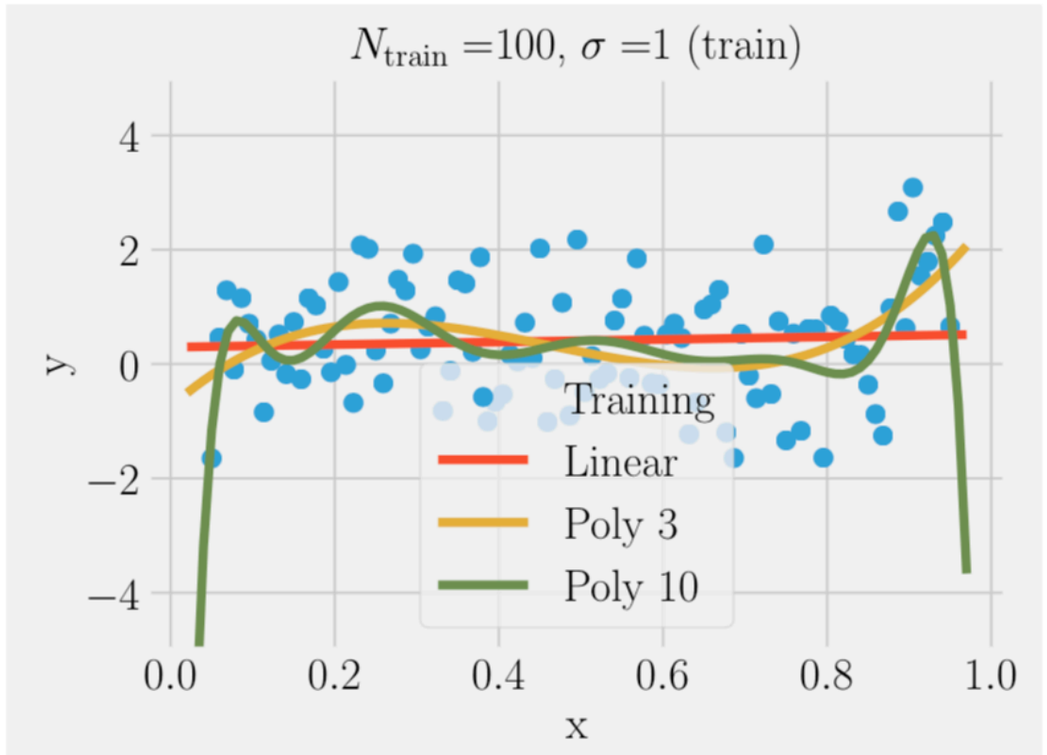
- It is difficult to **generalise** beyond the situations encountered in the training set: the model cannot learn what it has not seen

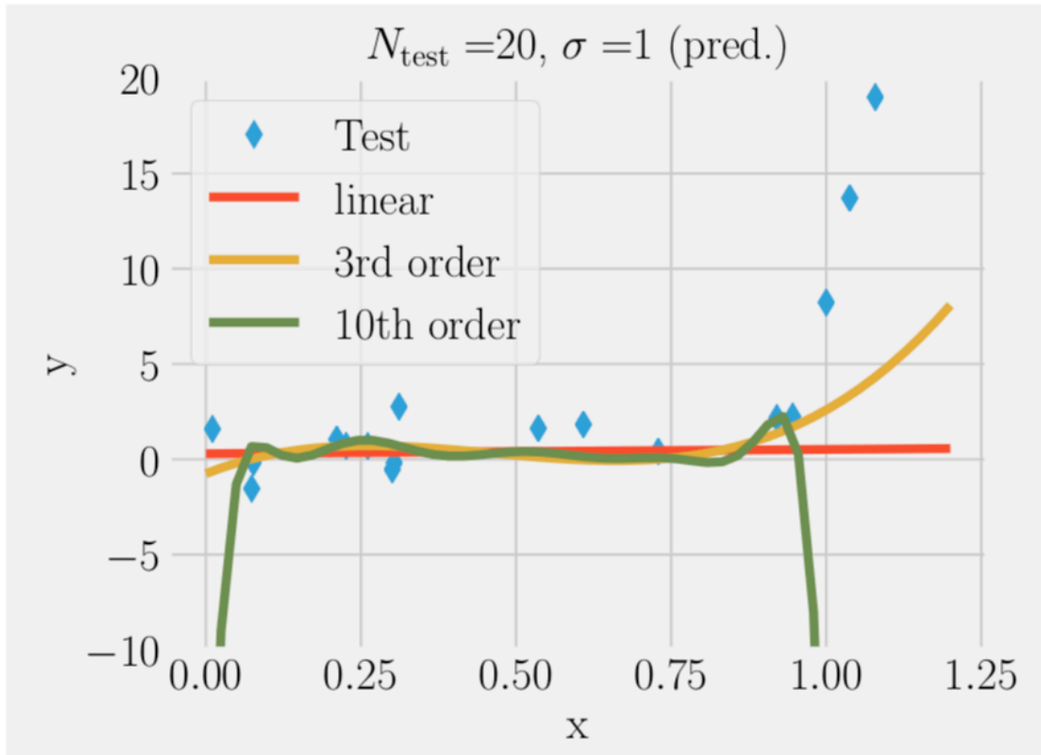
- Many problems that are **approachable in principle** can become **unfeasible in practice**, e.g. due to computational limitations, lack of convergence, instability ……

Deep Learning successes boosted by hardware developments

The bias-variance tradeoff

The bias-variance trade-off





increasing the model complexity can improve the description of the training data but **reduce the predictive power** of the model due to overfitting

The bias-variance trade-off

Our starting point is the **underlying law** ***y=f(x)*** which we aim to learn from a dataset (x_i,y_i), $i=1,\ldots,N$. We will also need an **hypothesis set** ***H*** containing all functions that we consider to be good candidates for the underlying law

The goal of **Statistical Learning Theory** is to determine a function from the hypothesis set *H* that approximates *f(x)* as best as possible, ideally in a strict mathematical limit

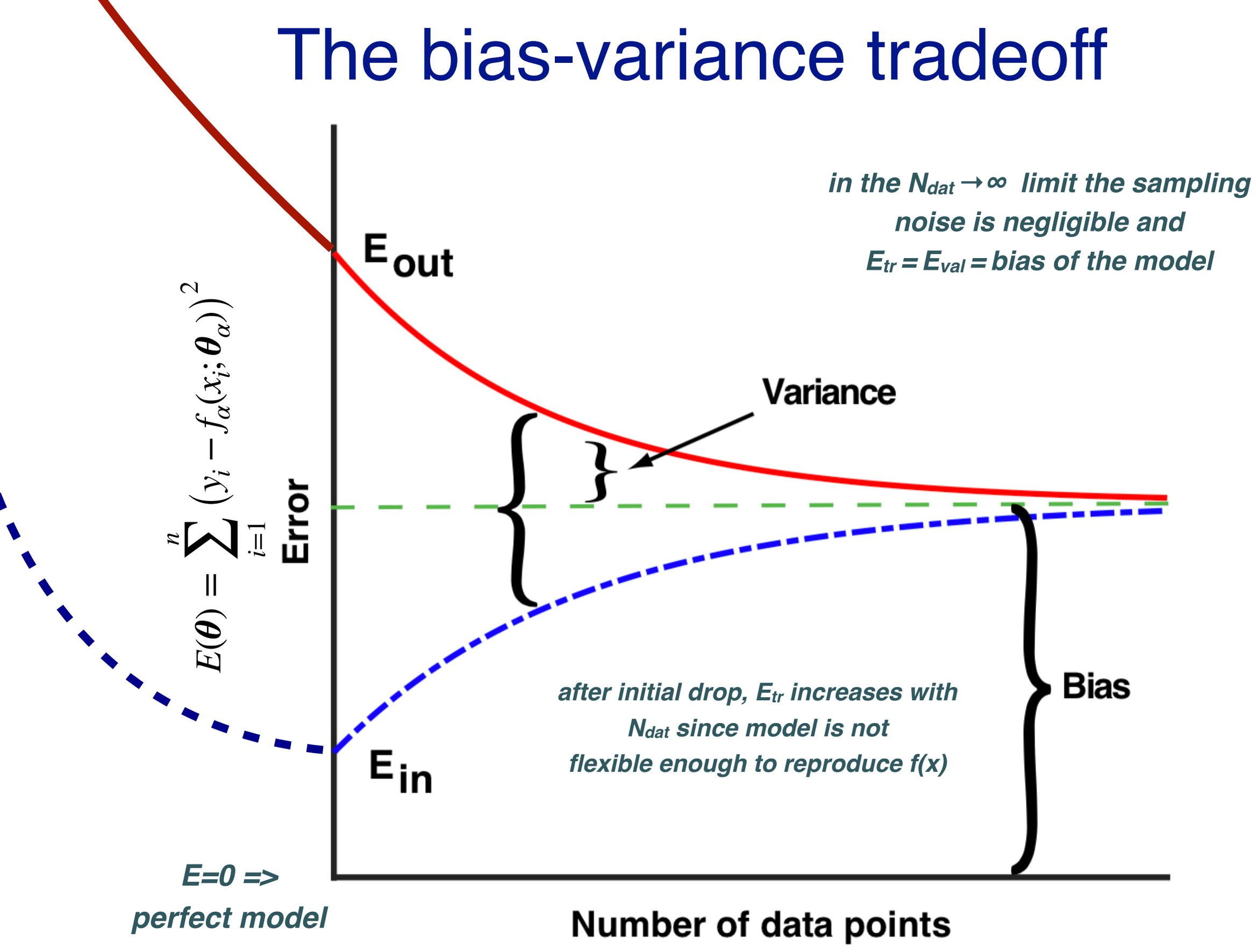
Here we will provide an **intuitive picture** of how Statistical Learning works

Consider the following situation:

- The **underlying law** is so complex that we cannot aim to exactly reproduce *f(x)*
- We want to study the dependence of E_{tr} and E_{val} with the number of data points

This setting allows us to present one of the most important concepts in the theory of Machine Learning: **the bias-variance tradeoff**

The bias-variance tradeoff

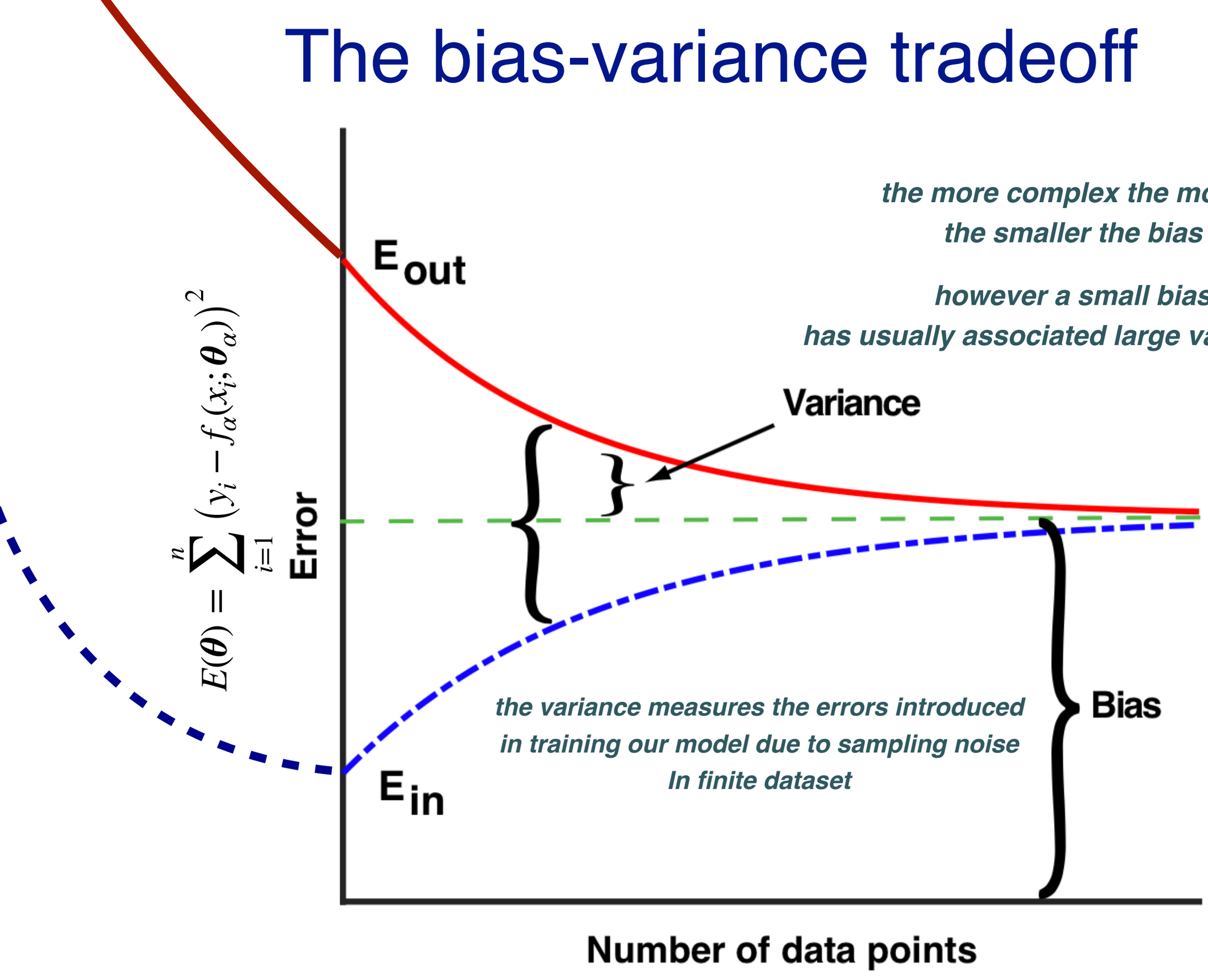


in the $N_{dat} \to \infty$ limit the sampling noise is negligible and $E_{tr} = E_{val}$ = bias of the model

after initial drop, E_{tr} increases with N_{dat} since model is not flexible enough to reproduce f(x)

E=0 => perfect model

The bias-variance tradeoff



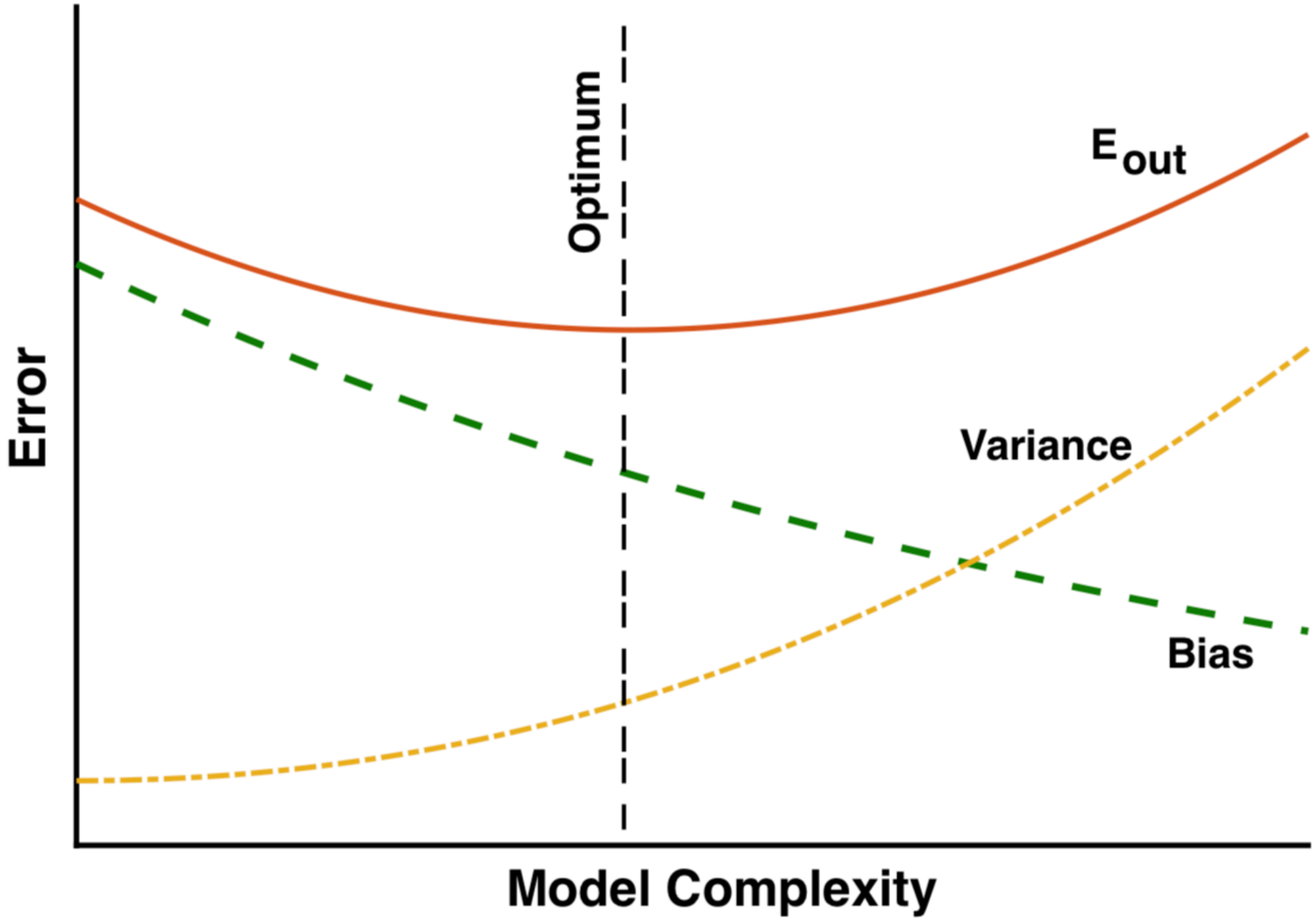
the more complex the model,
the smaller the bias

however a small bias
has usually associated large variance ….

the variance measures the errors introduced
in training our model due to sampling noise
In finite dataset

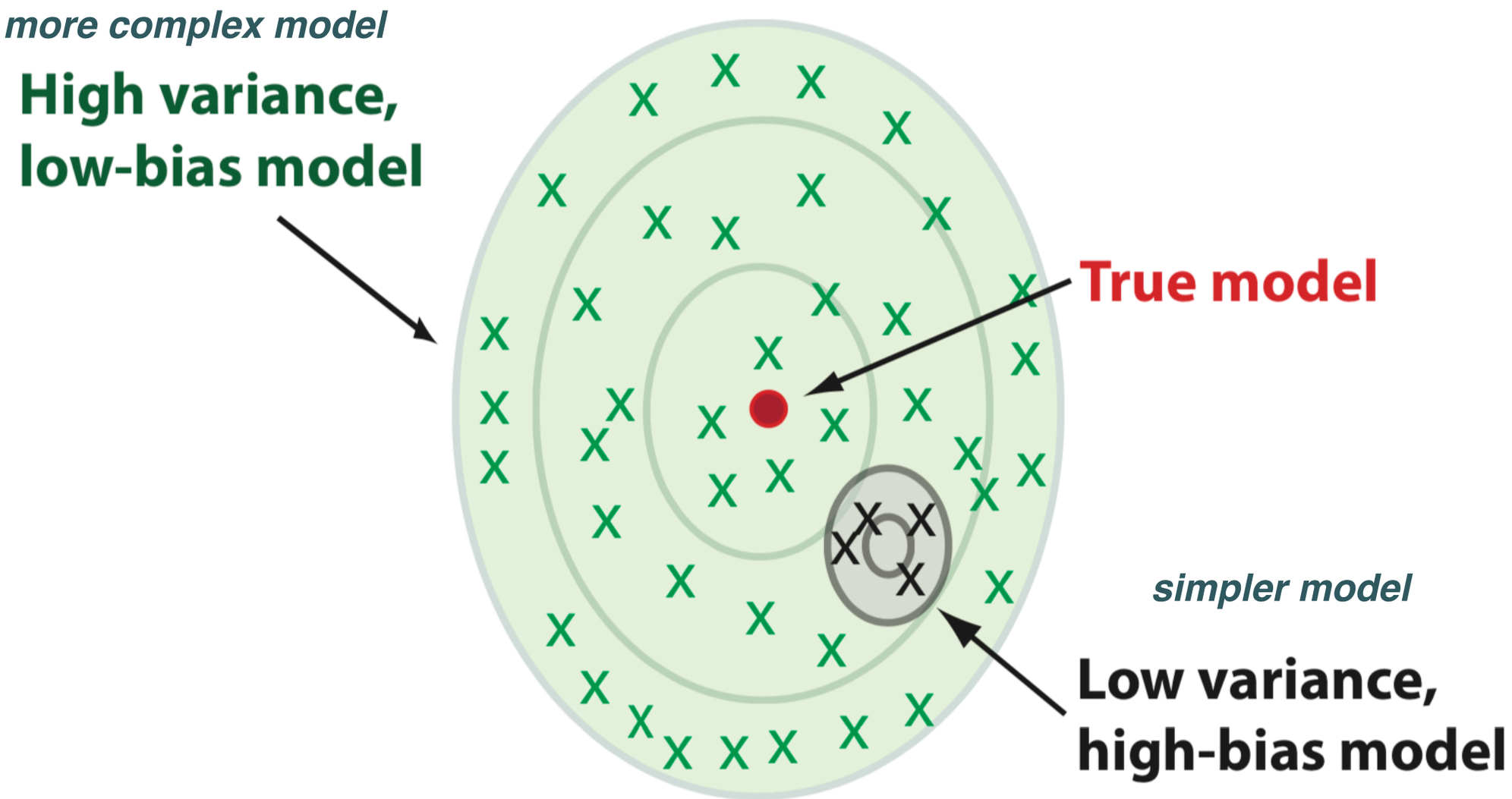
The bias-variance tradeoff

Optimal model performance (measured by minimising the generalisation error E_{val}) is typically achieved at intermediate levels of model complexity: the **bias-variance tradeoff**



The bias-variance tradeoff

Optimal model performance (measured by minimising the generalisation error E_{val}) is typically achieved at intermediate levels of model complexity: the **bias-variance tradeoff**



The bias-variance tradeoff

we can provide further insight on the bias-variance tradeoff with a simple scenario

assume a **model** extracted from a **training set** $\longrightarrow$ $\hat{y}(\boldsymbol{x}_{\rm tr})$

and that the true underlying law is given by

$$Y = y(\boldsymbol{X}) + \epsilon\,, \qquad y(\boldsymbol{x}_{\rm tr}) = E(Y, \boldsymbol{X} = \boldsymbol{x}_{\rm tr})$$

zero-mean, fixed variance noise

take a data point **outside the training set, *(x_0,y_0)***, and compute **its variance** in our model

$$E[\left(Y(x_0) - \hat{y}(x_0)\right)^2] = \left(y_0^2 - 2y_0 E[\hat{y}(x_0)] + E[\hat{y}^2(x_0)] + E[\epsilon^2]\right)$$

underlying law *model*

where we set to zero terms linear in the stochastic noise

The bias-variance tradeoff

we can provide further insight on the bias-variance tradeoff with a simple scenario

assume a **model** extracted from a **training set** $\longrightarrow$ $\hat{y}(\boldsymbol{x}_{\text{tr}})$

and that the true underlying law is given by

$$Y = y(\boldsymbol{X}) + \epsilon\,, \qquad y(\boldsymbol{x}_{\text{tr}}) = E(Y, \boldsymbol{X} = \boldsymbol{x}_{\text{tr}})$$

zero-mean, fixed variance noise

take a data point **outside the training set, *(x_0,y_0)***, and compute **its variance** in our model

$$E[\left(Y(x_0) - \hat{y}(x_0)\right)^2] = \left(\text{Bias}[\hat{y}(x_0)]\right)^2 + \text{Var}[\hat{y}(x_0))] + \text{Var}(\epsilon)$$

$$\text{Bias}[\hat{y}(x_0))] = E[\hat{y}(x_0)] - y_0$$

$$\text{Var}[\hat{y}(x_0))] = E[\hat{y}^2(x_0)] - E[\hat{y}(x_0)]^2$$

the **model generalisation power** decreases both with its bias and its variance

Supervised Learning:

Deterministic Optimisation Strategies

Optimisation strategies

problems in **Supervised Machine Learning** are defined by the following ingredients:

(1) Input dataset: $\mathscr{D} = (\boldsymbol{X}, \boldsymbol{Y})$

(2) Model: $f\left(\boldsymbol{X}, \boldsymbol{\theta}\right)$

(3) Cost function: $C\left(\boldsymbol{Y}; f(\boldsymbol{X}; \boldsymbol{\theta}\right)$

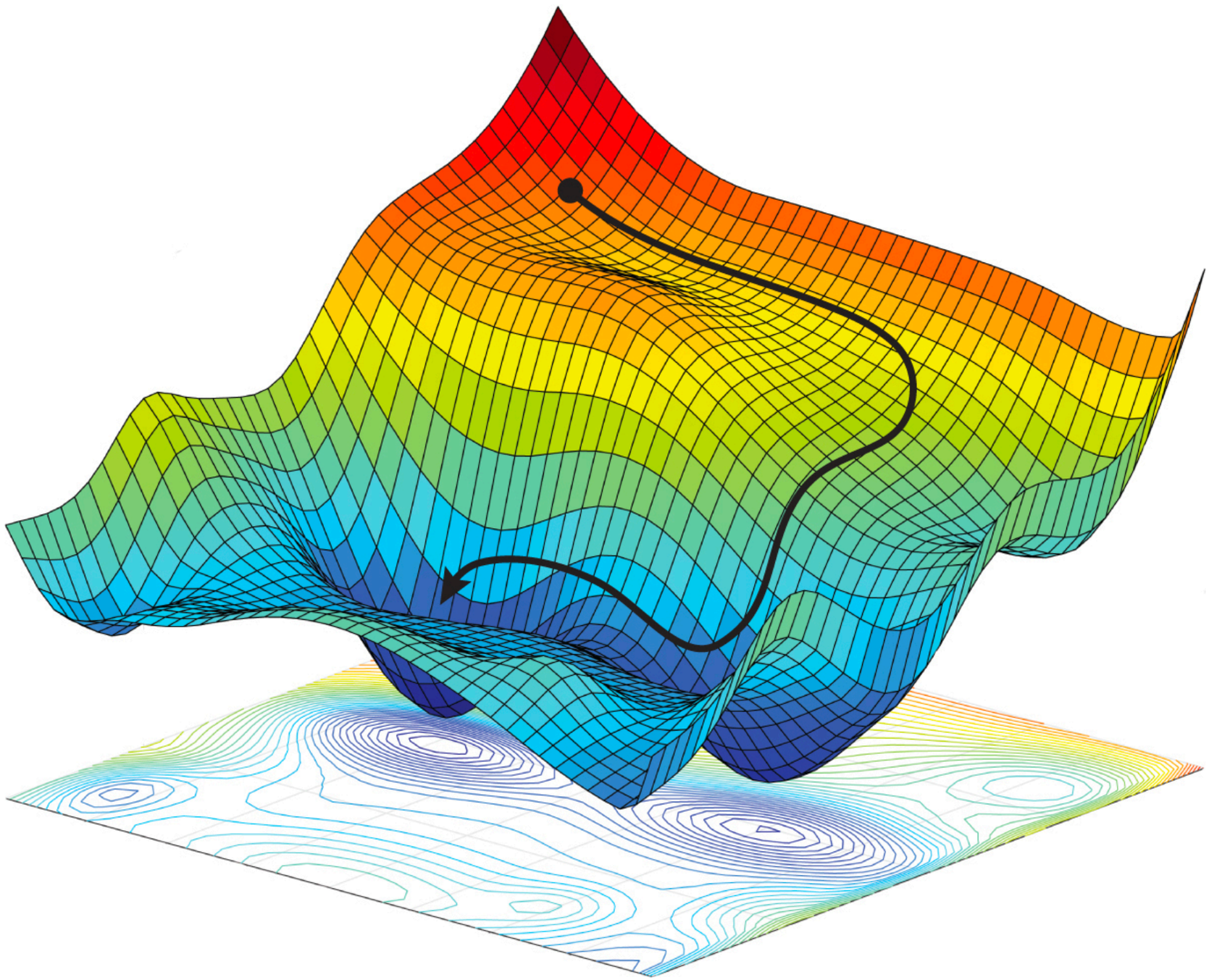
the model parameters are then found by **minimising the cost function**

for simple models (*e.g.* polynomial regression), the solution of this minimisation problem can be found analytically. This is not possible with complex models in realistic applications. In the context of ML studies, there exist two main classes of **minimisers (optimisers)** that are frequently deployed:

- **Gradient Descent** and its generalisations
- **Genetic Algorithms** and its generalisations

Gradient Descent

basic idea: iteratively adjust the model parameters in the direction where the **gradient of the cost function** is large and negative (**steepest descent direction**)



Gradient descent

basic idea: iteratively adjust the model parameters in the direction where the **gradient of the cost function** is large and negative (**steepest descent direction**)

Our goal is thus to **minimise an error (cost) function** that can usually be expressed as

$$E\left(\boldsymbol{\theta}\right) = \sum_{i=1}^{n} e_i(x_i;\boldsymbol{\theta})$$

e.g. in polynomial regression we had that

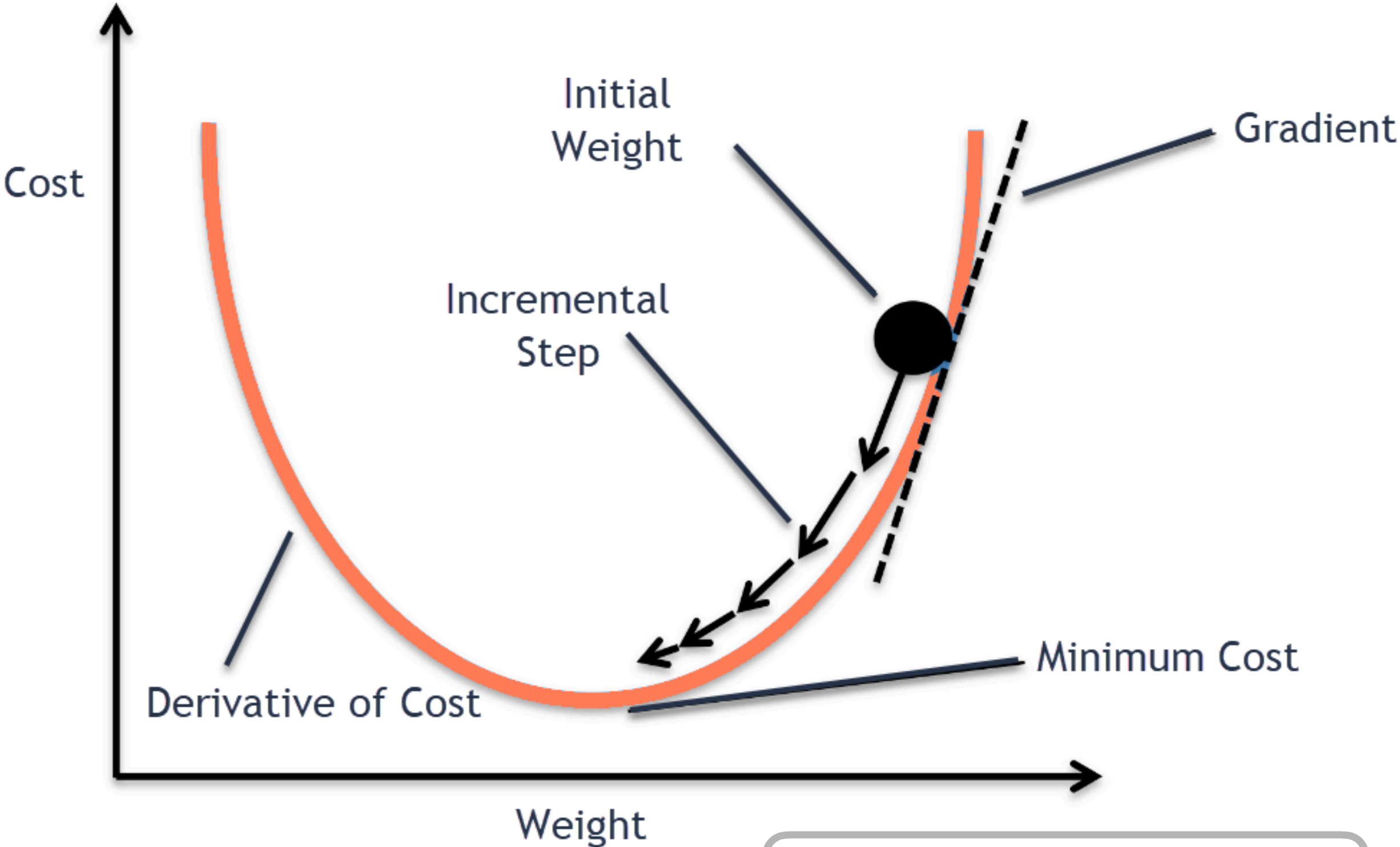
$$E(\boldsymbol{\theta}) = \sum_{i=1}^{n} \left(y_i - f_\alpha(x_i;\boldsymbol{\theta}_\alpha)\right)^2 \quad \text{so} \quad e_i = \left(y_i - f_\alpha(x_i;\boldsymbol{\theta}_\alpha)\right)^2$$

and we will see that in **logistic regression** (classification problems)

$$E(\boldsymbol{\theta}) = \sum_{i=1}^{n} \left(-y_i \ln \sigma(\boldsymbol{x}_i^T\boldsymbol{\theta}) - (1-y_i)\ln[1-\sigma(\boldsymbol{x}_i^T\boldsymbol{\theta}]\right)$$

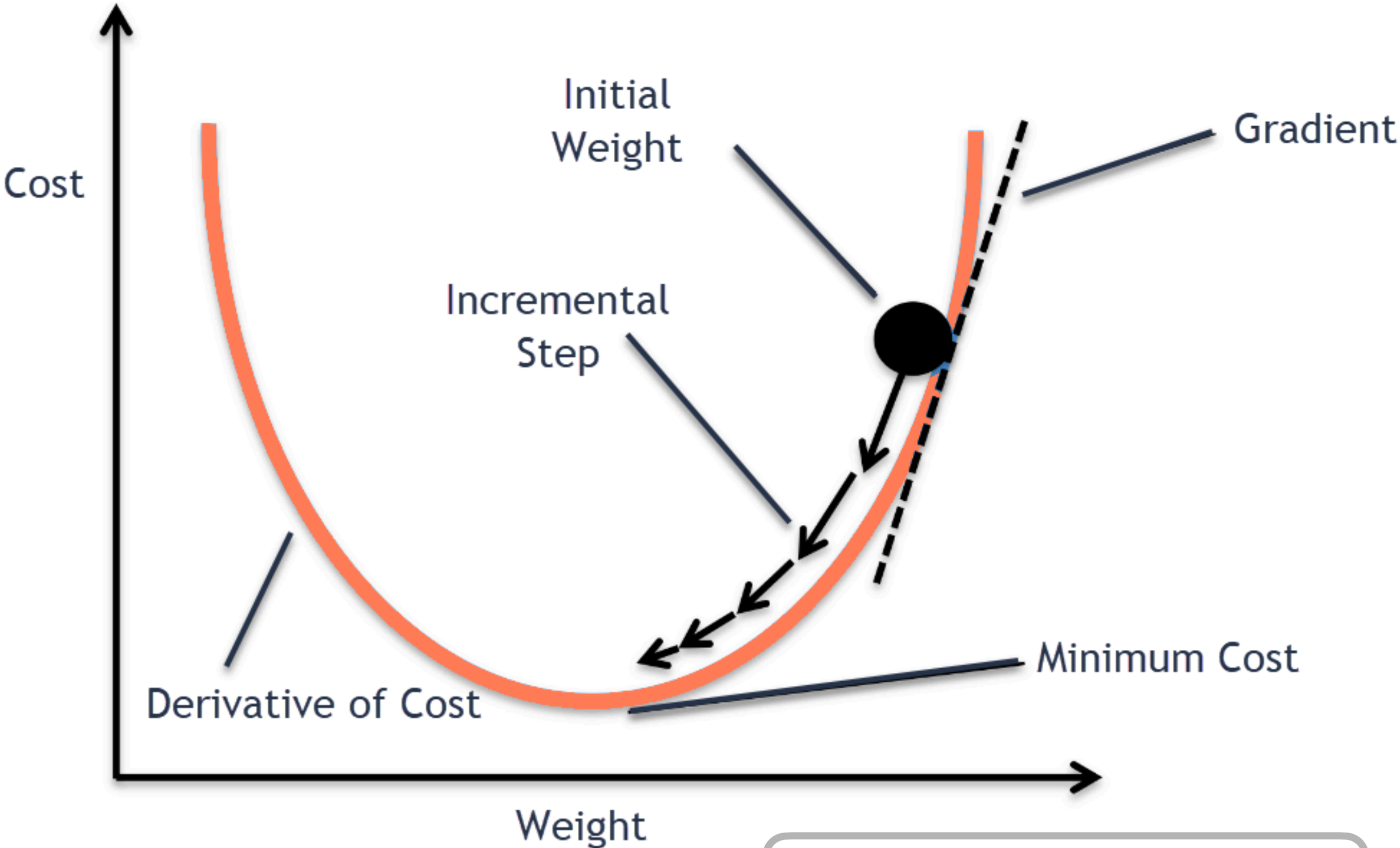
aka the cross-entropy, relevant for categorisation

Gradient descent



What do you think is key benefit of GD?

Gradient descent



Only local information (gradient) required!

Gradient descent

starting from a suitable initial condition, GD **iteratively updates** the model parameters:

$$\mathbf{v}_t = \eta_t \nabla_\theta E(\boldsymbol{\theta}_t)\,, \qquad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t$$

learning rate

gradient of cost function

*update of model parameters between iterations **t** and **t+1***

the learning rate determines the size of the step in the direction of the gradient

What could go wrong when choosing the learning rate?

Gradient descent

starting from a suitable initial condition, GD **iteratively updates** the model parameters:

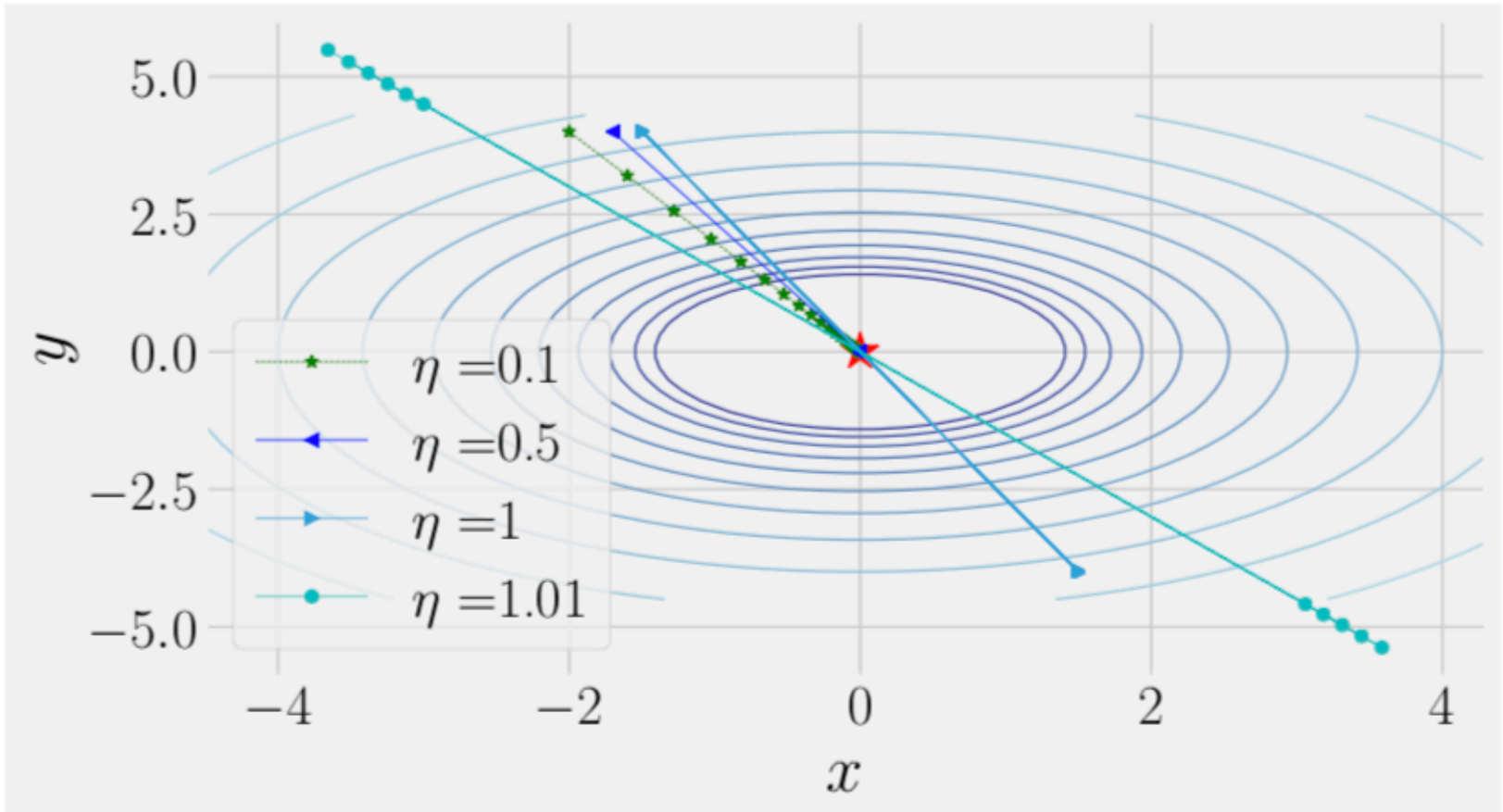
$$\mathbf{v}_t = \eta_t \nabla_\theta E(\boldsymbol{\theta}_t)\,, \qquad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t$$

learning rate

gradient of cost function

*update of model parameters between iterations **t** and **t+1***

the learning rate determines the size of the step in the direction of the gradient



small η: guaranteed to find local minimum, at price of large number of iterations

large η: can overshoot minimum, problems of convergence

nb evaluating gradient can be very cpu-intensive!

Gradient descent

starting from a suitable initial condition, GD **iteratively updates** the model parameters:

$$\mathbf{v}_t = \eta_t \nabla_\theta E(\boldsymbol{\theta}_t)\,, \qquad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t$$

learning rate

gradient of cost function

*update of model parameters between iterations **t** and **t+1***

compare GD with **Newton's method**: first Taylor-expand of cost function for a small change of the model parameters

$$E\left(\boldsymbol{\theta}+\mathbf{v}\right) \simeq E\left(\boldsymbol{\theta}\right) + \nabla_\theta E\left(\boldsymbol{\theta}\right)\cdot\mathbf{v} + \frac{1}{2}\mathbf{v}^T H(\boldsymbol{\theta})\mathbf{v}$$

$$H_{ij}(\boldsymbol{\theta}) = \left\{ \frac{\partial^2 E(\boldsymbol{\theta})}{\partial\theta_i \partial\theta_j} \right\}$$

Hessian matrix (2nd derivatives)

Gradient descent

starting from a suitable initial condition, GD **iteratively updates** the model parameters:

$$\mathbf{v}_t = \eta_t \nabla_\theta E(\boldsymbol{\theta}_t)\,, \qquad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t$$

learning rate

gradient of cost function

*update of model parameters between iterations **t** and **t+1***

compare GD with **Newton's method**: first Taylor-expand of cost function for a small change of the model parameters

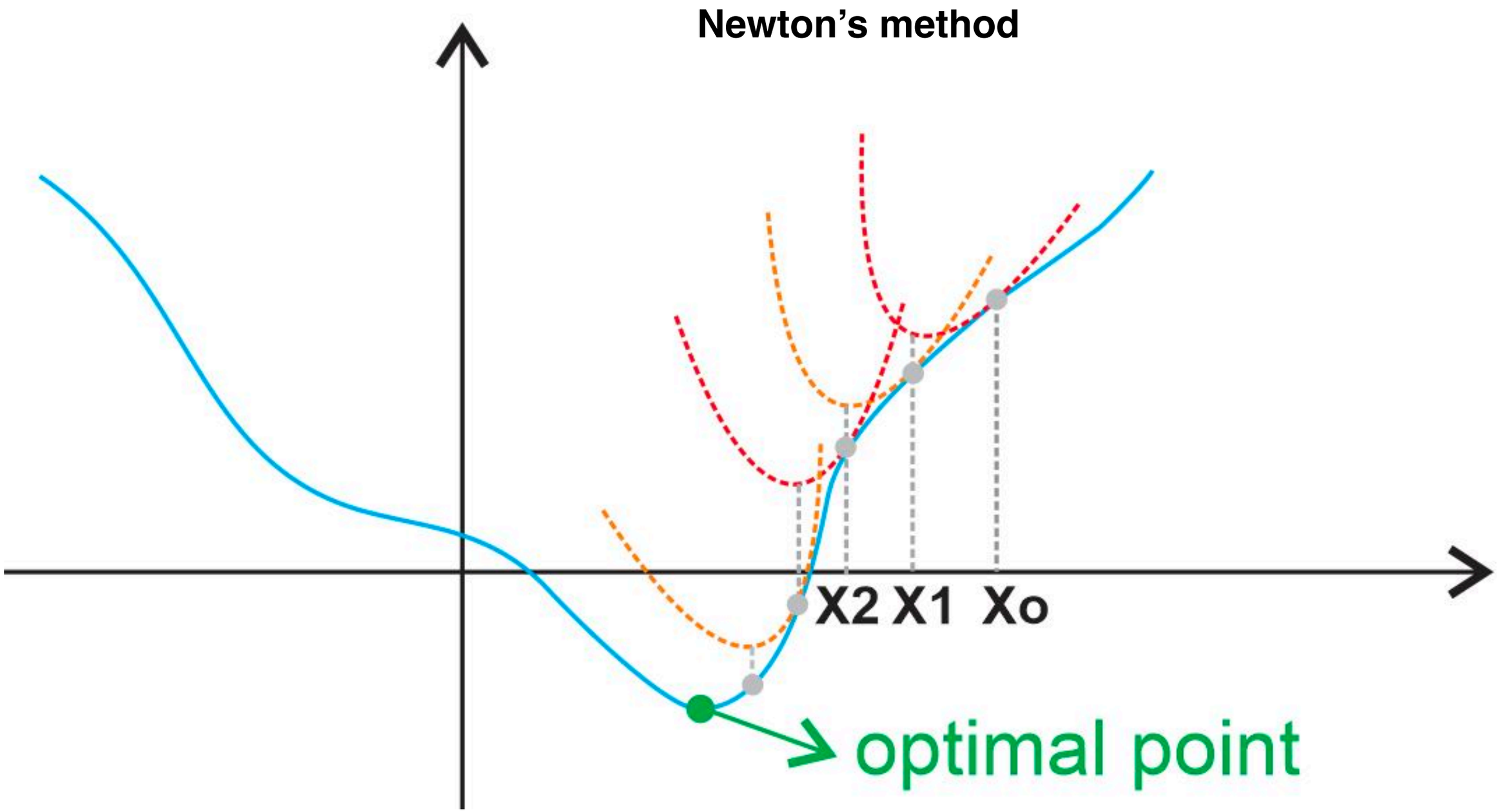
$$E\left(\boldsymbol{\theta} + \mathbf{v}\right) \simeq E\left(\boldsymbol{\theta}\right) + \nabla_\theta E\left(\boldsymbol{\theta}\right) \cdot \mathbf{v} + \frac{1}{2}\mathbf{v}^T H(\boldsymbol{\theta})\mathbf{v}$$

and then differentiate with respect to the step requiring that the **linear term vanishes**

$$\left.\frac{\partial E\left(\boldsymbol{\theta} + v\right)}{\partial \mathbf{v}}\right|_{\mathbf{v}_{\text{opt}}} = 0 \longrightarrow \nabla_\theta E\left(\boldsymbol{\theta}\right) + H(\boldsymbol{\theta})\mathbf{v}_{\text{opt}} = \mathbf{0}$$

$$\mathbf{v}_t = H^{-1}(\boldsymbol{\theta}_t) \cdot \nabla_\theta E(\boldsymbol{\theta}_t)\,, \qquad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t$$

Gradient descent



Gradient descent

Gradient Descent → $\mathbf{v}_t = \eta_t \nabla_\theta E(\boldsymbol{\theta}_t)\,, \qquad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t$

Newton's method → $\mathbf{v}_t = H^{-1}(\theta_t) \cdot \nabla_\theta E(\boldsymbol{\theta}_t)\,, \qquad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t$

Why Newton's method not suitable for machine learning problems?

Gradient descent

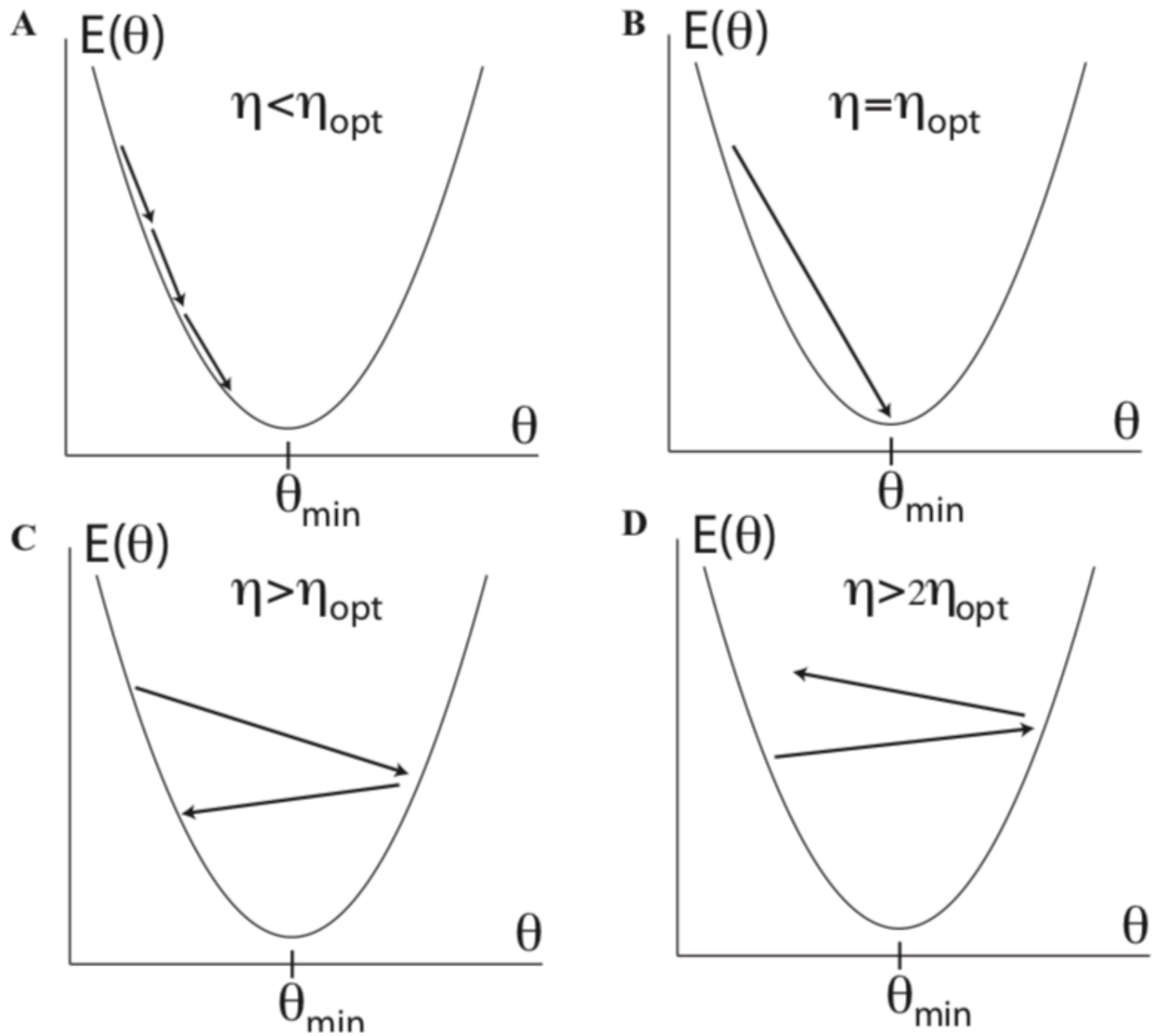
Gradient Descent $\longrightarrow \mathbf{v}_t = \eta_t \nabla_\theta E(\boldsymbol{\theta}_t), \qquad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t$

Newton's method $\longrightarrow \mathbf{v}_t = H^{-1}(\theta_t) \cdot \nabla_\theta E(\boldsymbol{\theta}_t), \qquad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t$

- Newton's method not practical for ML applications: it involves **evaluation and inversion of Hessian matrices with M^2 entries**, with M = number of model parameters

- But it provides useful ideas about how to improve GD, for example by **adapting the learning rate to the local curvature** of region of the parameter space

suggest improvements of GD!



Advanced gradient descent

the simplest implementation of GD, unsurprisingly, has several limitations

- It converges to **local, rather than global**, minima of the cost function

poor performance for realistic applications

Advanced gradient descent

the simplest implementation of GD, unsurprisingly, has several limitations

- It converges to **local, rather than global**, minima of the cost function

poor performance for realistic applications

- Evaluating gradients is **computationally expensive** for large datasets

$$E(\boldsymbol{\theta}) = \sum_{i=1}^{n} \left(y_i - f_\alpha(x_i; \boldsymbol{\theta}_\alpha)\right)^2$$

involves sum over all n data points

Advanced gradient descent

the simplest implementation of GD, unsurprisingly, has several limitations

- It converges to **local, rather than global**, minima of the cost function

poor performance for realistic applications

- Evaluating gradients is **computationally expensive** for large datasets

$$E(\boldsymbol{\theta}) = \sum_{i=1}^{n} \left(y_i - f_\alpha(x_i; \boldsymbol{\theta}_\alpha)\right)^2$$

involves sum over all n data points

- Very sensitive to choice of **learning rates**

Ideally one would like an adaptive learning rate

Advanced gradient descent

the simplest implementation of GD, unsurprisingly, has several limitations

- It converges to **local, rather than global**, minima of the cost function

poor performance for realistic applications

- Evaluating gradients is **computationally expensive** for large datasets

$$E(\boldsymbol{\theta}) = \sum_{i=1}^{n} \left(y_i - f_\alpha(x_i; \boldsymbol{\theta}_\alpha)\right)^2$$

involves sum over all n data points

- Very sensitive to choice of **learning rates**

Ideally one would like an adaptive learning rate

- Treats **uniformly all directions** in the parameter space

and why this is a problem??

Advanced gradient descent

the simplest implementation of GD, unsurprisingly, has several limitations

- It converges to **local, rather than global**, minima of the cost function

poor performance for realistic applications

- Evaluating gradients is **computationally expensive** for large datasets

$$E(\boldsymbol{\theta}) = \sum_{i=1}^{n} \left(y_i - f_\alpha(x_i; \boldsymbol{\theta}_\alpha)\right)^2$$

involves sum over all n data points

- Very sensitive to choice of **learning rates**

Ideally one would like an adaptive learning rate

- Treats **uniformly all directions** in the parameter space

we would like to use info also on curvature (as in Newton's method)

Generalised Gradient Descent methods have been developed to address these shortcomings and are at the basis of **modern deep learning methods**

Stochastic gradient descent

Stochasticity can be added to GD by approximating the gradient on a subset of the training data, called a **mini-batch**

$$\nabla_\theta E(\boldsymbol{\theta}) = \sum_{i=1}^{n} \nabla_\theta e_i(\boldsymbol{x}_i;\boldsymbol{\theta}) \simeq \nabla_\theta E^{\mathrm{MB}}(\boldsymbol{\theta}) \equiv \sum_{i\in B_k} \nabla_\theta e_i(\boldsymbol{x}_i;\boldsymbol{\theta})$$

$k = 1,\ldots,n/K$ ← *# points per batches*

 # batches

We then **cycle over all mini-batches**, updating the model parameters at each step *k*

$$\mathbf{v}_t = \eta_t \nabla_\theta^{\mathrm{MB}} E(\boldsymbol{\theta}_t)\,, \qquad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t$$

A full iteration over all *n* data points (over the *n/K* batches) is called an **epoch**

benefits of SGD: stochasticity prevents getting stuck in local minima, the calculation of the gradient is speed up & stochasticity acts as natural regulariser

Adding momentum to SGD

SGD can be used with a **momentum term** that provides some **memory** on the direction in which one is moving in the parameter space

$$\mathbf{v}_t = \gamma \mathbf{v}_{t-1} + \eta_t \nabla_\theta^{\mathrm{MB}} E(\boldsymbol{\theta}_t)\,, \qquad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t$$

momentum parameter (0 < ɣ < 1)

$$\Delta\boldsymbol{\theta}_{t+1} = \gamma \Delta\boldsymbol{\theta}_t - \eta_t \nabla_\theta E(\boldsymbol{\theta}_t), \qquad \Delta\boldsymbol{\theta}_t = \boldsymbol{\theta}_t - \boldsymbol{\theta}_{t-1}$$

proposed parameter change in iteration t+1

proposed parameter change in iteration t

momentum in SGD helps to **gain speed in directions with persistent but small gradients**, while suppressing oscillations in high-curvature directions.

Adaptative Gradient Descent

Adaptative GD varies the learning rate to reflect the **local curvature** of the parameter space without the need to evaluate the Hessian matrix

RMSprop: keep track also of the **second moment of gradient**, similar as how the momentum term is a running average of previous gradients

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \frac{\boldsymbol{g}_t}{\sqrt{\boldsymbol{s}_t + \epsilon}}$$

regulator to avoid numerical divergences

$$\boldsymbol{g}_t = \nabla_\theta E(\boldsymbol{\theta})$$

gradient at iteration t

$$\boldsymbol{s}_t = \beta \boldsymbol{s}_{t-1} + (1-\beta)\boldsymbol{g}_t^2$$

controls averaging time of 2nd moment of gradient

$$\boldsymbol{s}_t = \mathbb{E}[\boldsymbol{g}_t^2]$$

2nd moment of gradient (averaged over iterations)

$$\beta = 1 \rightarrow \boldsymbol{s}_t = \boldsymbol{s}_{t-1}$$
$$\beta = 0 \rightarrow \boldsymbol{s}_t = \boldsymbol{g}_t^2$$

effective learning rate reduced in directions where gradient is consistently large

Adaptative Gradient Descent

Adaptative GD varies the learning rate to reflect the **local curvature** of the parameter space without the need to evaluate the Hessian matrix

ADAM: adaptively change the parameters from information on **running averages** of first and second moments of the gradient

$$\boldsymbol{g}_t = \nabla_\theta E(\boldsymbol{\theta})$$

$$\boldsymbol{m}_t = \beta_1 \boldsymbol{m}_{t-1} - (1-\beta_1)\boldsymbol{g}_t \qquad \boldsymbol{s}_t = \beta_2 \boldsymbol{s}_{t-1} + (1-\beta_2)\boldsymbol{g}_t^2$$

$$\widehat{\boldsymbol{m}}_t = \frac{\boldsymbol{m}_t}{1-(\beta_1)^t} \qquad \widehat{\boldsymbol{s}}_t = \frac{\boldsymbol{s}_t}{1-(\beta_2)^t}$$

sets lifetime of first moment

sets lifetime of second moment

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \frac{\widehat{\boldsymbol{m}}_t}{\sqrt{\widehat{\boldsymbol{s}}_t + \epsilon}}$$

Adaptative Gradient Descent

Adaptative GD varies the learning rate to reflect the **local curvature** of the parameter space without the need to evaluate the Hessian matrix

ADAM has two main advantages: *(i)* adapting step size to cut off large gradient directions (prevent oscillations) and *(ii)* measuring gradients in a natural length scale

to see this, express the ADAM update rules in terms of the **variance in parameter space**

$$\boldsymbol{\sigma}_t = \widehat{\boldsymbol{s}}_t - \left(\widehat{\boldsymbol{m}}_t\right)^2$$

and we can see that the update rule for one of the parameters reads now

$$\Delta\theta_{t+1} = -\eta_t \frac{\widehat{m}_t}{\sqrt{\sigma_t^2 + \widehat{m}_t^2 + \epsilon}}$$

small fluctuations of gradient

$$\Delta\theta_{t+1} \to -\eta_t$$

large fluctuations of gradient

$$\Delta\theta_{t+1} = -\eta_t \widehat{m}_t / \sigma_t^2$$

learning rate promotional to signal-to-noise ratio

Tutorial 2 (2a & 2b)

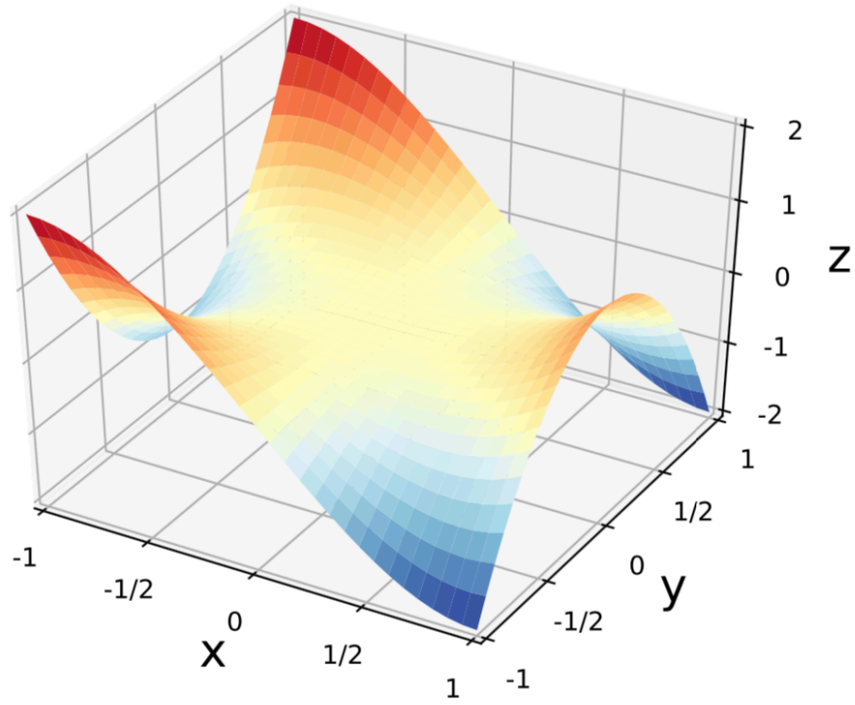
- Study different optimisation algorithms based on GD for simple **1D** (easy) and **2D** functions (more difficult)

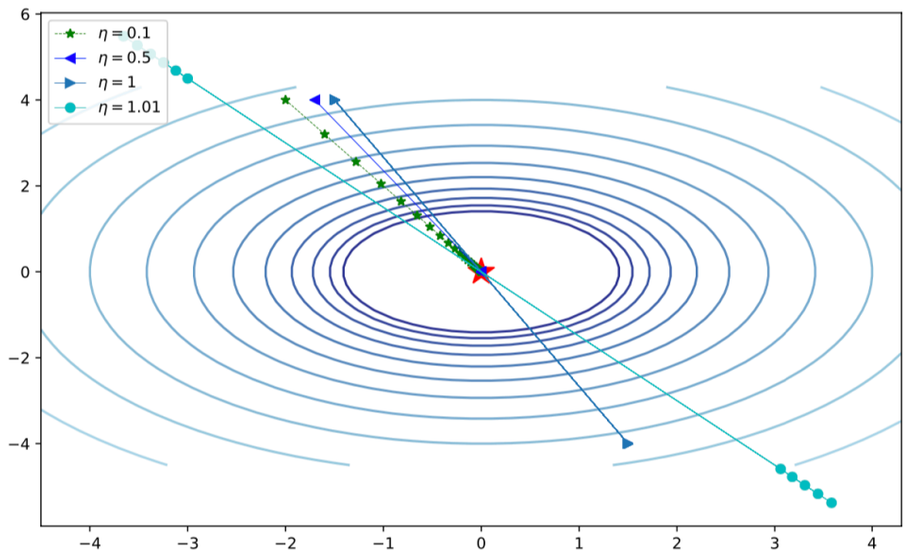
- Visualise the **path of GD algorithms in parameter space**

- Determine which choices of the parameters allow reaching the minima quicker

- Given new 2D functions, show that you can **tune GD to find all the minima**, and compare with the analytical results

challenge: find the most efficient optimiser settings for a given function!





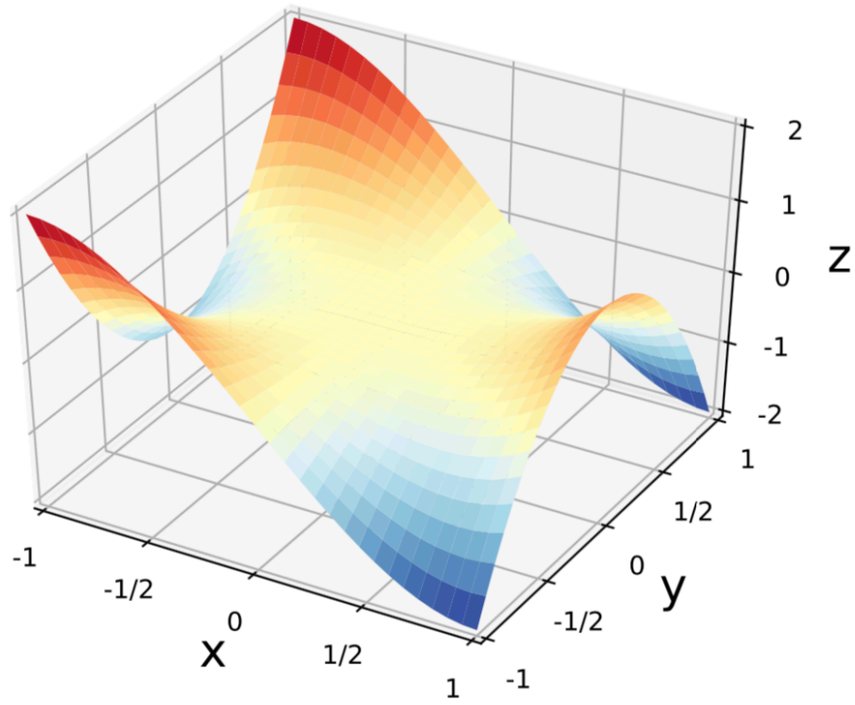
Tutorial 2

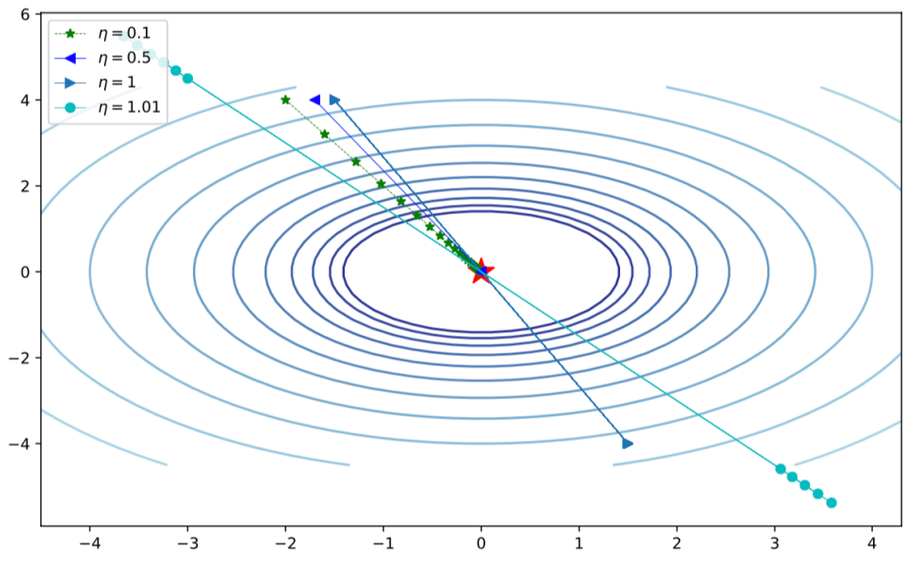
- Study different optimisation algorithms based on GD for simple **1D** (easy) and **2D** functions (more difficult)

- Visualise the **path of GD algorithms in parameter space**

- Determine which choices of the parameters allow reaching the minima quicker

- Given new 2D functions, show that you can **tune GD to find all the minima**, and compare with the analytical results





Question: how could one go beyond hyperparameter setting by trial and error?

Lecture 3!

Extra Material

The Machine Learning Galaxy I

In this course we will have time to cover only subset of ML algorithms and applications!