From the athlete to the computer: computing with a digital twin

A practical introduction to models, computation and Python for sport scientists

Thomas Gomez

Laboratoire de Biologie de l'Exercice pour la Performance et la Santé (LBEPS), Université d'Évry Paris-Saclay
Icam, Grand Paris Sud campus

International master, practical session

Today

Talk

1. The digital twin for sport
2. The mathematical models it needs
3. How a computer computes
4. Programming and Python
5. Artificial intelligence and learning
6. What you will do in the lab

Lab (3 hours)

- In your web browser, nothing to install
- Step-by-step notebooks
- A mini-project: *coach for a day*
- You leave with figures you produced

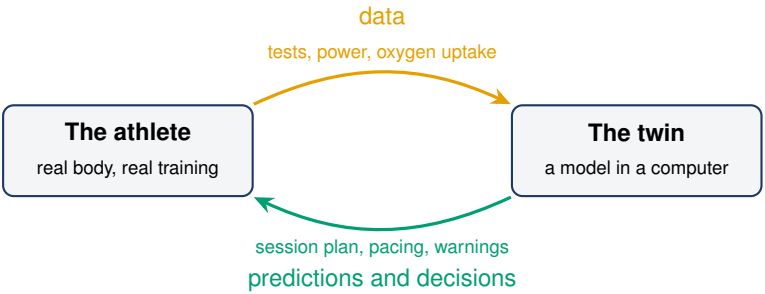
Next sessions (3 hours each)

2. Simulate a 1500 m race and find the best pacing.
3. Learning from data: random forest, neural network, physics-informed network.

No prior knowledge of programming is needed. School mathematics is enough: $+$, $-$, $\times$, $\div$ and a straight line.

The digital twin for sport

What is a digital twin? (reminder)



- The twin is **calibrated** on the data of *this* athlete, not on an average athlete.
- It answers “what if?” questions **before** the athlete tries: how long can I hold 350 W? Which interval session? Which pacing for my race?
- It is **updated** when new data arrive.

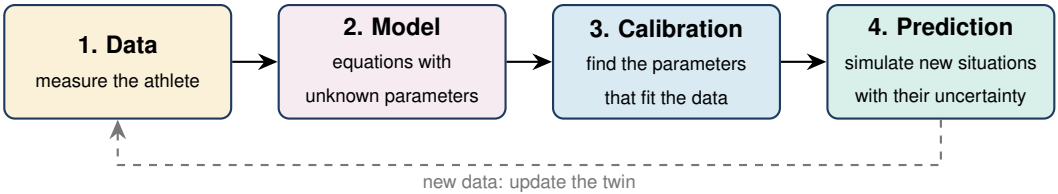
What a twin is used for in sport

Predict	Plan	Optimise
How long can the athlete hold a given power before exhaustion?	Which interval session brings the athlete close to the limit without exhaustion?	How should the effort be spread over a race (pacing)?

Examples from research:

- twins of track cyclists built on a three-vessel energy model, used to plan the team pursuit [3, 4];
- the optimal pacing of a runner computed from a model of speed, force and energy [1].

The four steps of every digital twin



Message

In the lab you will go through the four steps yourself, on a simple model of a cyclist.

Mathematical models

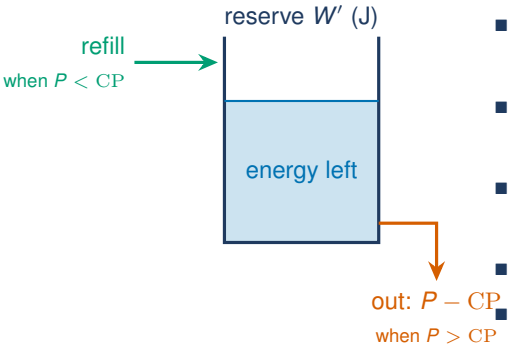
What is a mathematical model?

A model describes the athlete with three ingredients:

	Meaning	**Example (cyclist)**
State	what changes during the effort	energy left in a reserve
Parameters	fixed numbers that describe *this* athlete	critical power CP, reserve size W'
Rules	how the state changes	above CP, the reserve empties

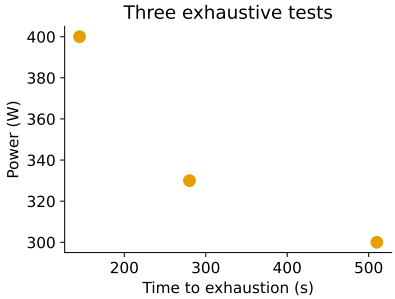
- The rules are the same for everybody; the parameters are personal.
- Calibrating the twin = finding the parameters of the athlete from data.

A simple model: the W' tank

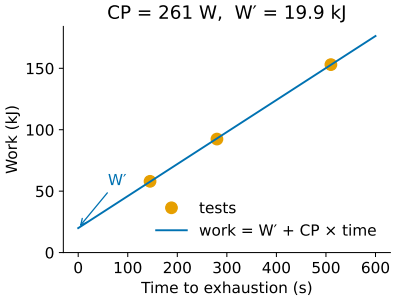


- CP, the **critical power**: the highest power that can be held for a long time [8, 7].
- W': a limited reserve of energy, used only above CP.
- Riding at $P > \mathrm{CP}$ empties the tank by $P - \mathrm{CP}$ joules each second.
- Empty tank $=$ **exhaustion**.
- Below CP the tank refills, as in the "W' balance" models used by coaches [13].

Calibration: from three tests to a straight line



Three rides at constant power until exhaustion.

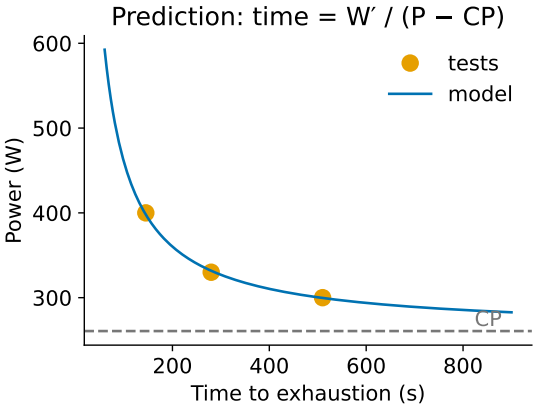


Work = power × time lies on a straight line.

$$\text{work} = W' + \mathrm{CP} \times \text{time} \qquad \Rightarrow \qquad \text{slope} = \mathrm{CP}, \quad \text{intercept} = W'$$

The computer finds the line closest to the points (*least squares*): this is the **calibration**.

Prediction



At a power P above CP the tank empties at $P - \mathrm{CP}$ joules per second, so

$$\text{time to exhaustion} = \frac{W'}{P - \mathrm{CP}}$$

- a prediction for any power, from three tests;
- but no meaning below CP: every model has a **domain of validity**.

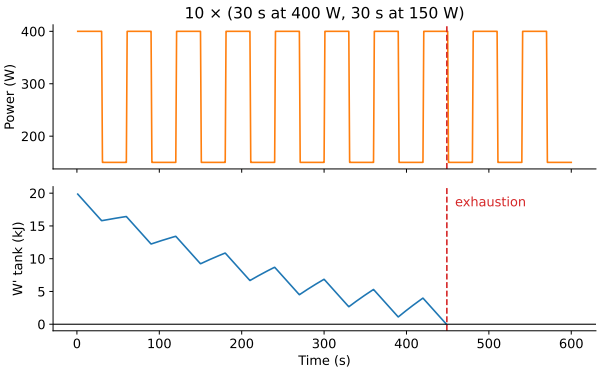
When there is no formula: simulate, one second at a time

Training sessions change power all the time: no simple formula applies. The computer applies the **rule** again and again.

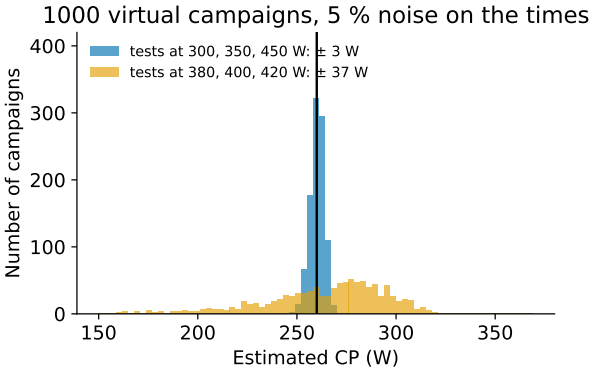
Rule (each second):

$$\text{tank}_{\text{new}} = \text{tank}_{\text{old}} - (P - \text{CP})$$

second	power (W)	tank (J)
0		20 000
1	350	19 910
2	350	19 820
…	…	…
223	350	−70 exhausted



Measurements are never perfect



- A time to exhaustion changes from one day to the next.
- The computer repeats a virtual test campaign 1000 times, with random errors, and recalibrates each time.
- The **spread** of the results is the uncertainty on CP.
- Same athlete, same noise: the **choice of the tests** changes the precision a lot.

Message

A twin must say how sure it is. Which tests teach the most about the athlete is a research question.

Research twins: same four steps, more vessels

Energy models of the cyclist

- Three vessels: oxygen, phosphocreatine, glycolysis (Margaria–Morton model, reinterpreted by [3]).
- 7 parameters, measured through oxygen uptake and times to exhaustion.
- Our work: which parameters can really be determined from such data, and how precisely (two-vessel model [5]; three-vessel model in preparation).

Pacing of the runner

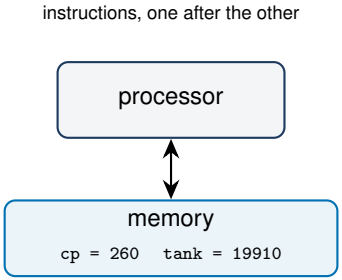
- Speed, propulsive force and anaerobic energy [1].
- The best pacing is found by *optimal control*.
- Next session: you will simulate this runner.

More vessels and more parameters make calibration harder: this is why the computation, and its uncertainty, matter.

How a computer computes

What a computer really does

- A **memory**: boxes that hold numbers.
- A **processor**: very simple operations ($+$, $-$, $\times$, $\div$, compare two numbers).
- Billions of operations per second.
- It does **exactly** what it is told, nothing more: no common sense.



Message

A simulation is not magic: it is a huge number of very simple operations, done very fast.

An algorithm is a recipe

Algorithm: a list of precise steps that solves a problem. The tank simulation in plain English:

1. Fill the tank: tank $= W'$. Set the clock to 0.
2. Repeat as long as the tank is not empty:
   - 2.1 add one second to the clock;
   - 2.2 if the power is above CP, remove $(P - \mathrm{CP})$ from the tank;
   - 2.3 otherwise, refill the tank a little.
3. Write the time on the clock: this is the time to exhaustion.

A computer cannot read English. The recipe must be written in a **programming language**.

Programming and Python

Programming languages

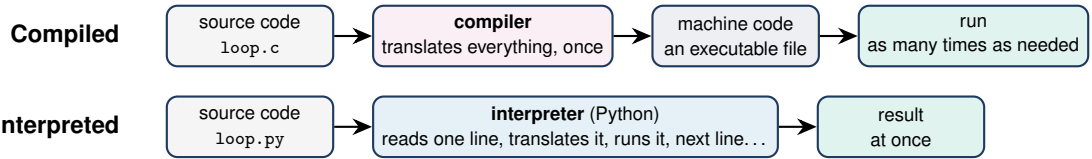
- A **programming language** is a way to write a recipe that a computer can follow: strict words, strict punctuation.
- There are many: Python, R, MATLAB, Julia, C...
- The ideas are the same in all of them: variables, loops, conditions, functions.

Two families

- **compiled** languages: C, C++, Fortran, Rust;
- **interpreted** languages: Python, R, MATLAB, JavaScript.

They differ in *when* the recipe is translated into the processor's own instructions.

Compiled or interpreted?



	Compiled (C, C++, Fortran)	**Interpreted** (Python, R, MATLAB)
Speed	very fast: the processor runs its own instructions	slower: each line is translated again at each pass of a loop
Writing	longer, stricter (types, memory)	short and readable
Trying	compile again after each change	immediate: change a number, run the cell
Errors	many found before running	found only when the line runs

The border is blurred: Python first turns the code into an intermediate “bytecode”; Java and Julia compile while running (*just in time*).

The same computation, three ways

Add $0.5\,i^2$ for $i = 0, 1, \ldots$ up to 10 million.

```
s = 0.0                                  # Python loop
for i in range(10_000_000):
    s = s + 0.5 * i * i
```

```
x = np.arange(10_000_000)                # NumPy
s = np.sum(0.5 * x * x)
```

```
for (long i = 0; i < 10000000; i++)    /* C */
    s = s + 0.5 * i * i;
```

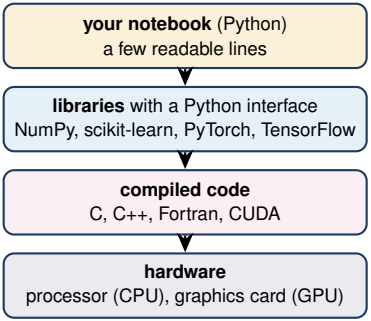
	time
Python loop	0.85 s
C (compiled)	0.035 s
NumPy	0.02 s

Measured on the author's laptop (Apple M1 Max, Python 3.12, best of 3 runs; `tools/benchmark/`).

Message

NumPy is written in C: Python only gives the order, the compiled code does the work. You write simple Python and get compiled speed.

Why Python became the language of science and AI



- **Readable and quick to try**: researchers test an idea in a few lines, in a notebook.
- **A "glue" language**: the heavy computation runs in compiled libraries; Python only drives them. Simplicity *and* speed.
- **The ecosystem**: NumPy (2006) for arrays, scikit-learn for machine learning, TensorFlow (2015) and PyTorch (2016) for neural networks, with automatic gradients and graphics cards.
- **Free and open**, with a huge community: courses, examples, answers.
- **Snowball effect**: the AI tools came out in Python, so people learn Python, so new tools are written for Python.

Limits: pure Python loops are slow, and products (phones, embedded devices, large services) often run in C++ or other languages.

The recipe written in Python

```
cp = 260            # critical power (W)
wp = 20000          # W' (J)
power = 350         # power of the ride (W)

tank = wp           # full tank
second = 0
while tank > 0:     # repeat until empty
    second = second + 1
    tank = tank - (power - cp)

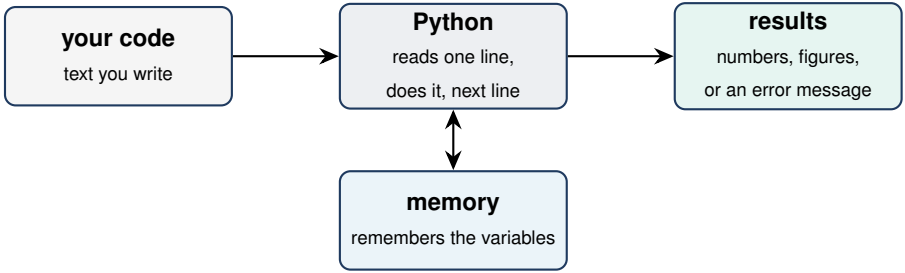
print("Exhausted after", second, "s")
```

```
Exhausted after 223 s
```

- `cp = 260`: put 260 in a box named `cp` (a **variable**).
- `#`: a comment for humans; Python ignores it.
- `while`: **repeat** the shifted lines as long as the condition holds.
- The shift to the right (**indentation**) shows what is inside the loop.
- `print`: show a result.

Formula: $20\,000/(350-260) = 222.2\,\text{s}$.
Same answer.

How Python runs your code



- Python reads **from top to bottom**; a variable must be defined before it is used.
- An **error message** means Python did not understand one line. It tells which line and why: read the **last line** of the message.

Six building blocks are enough for today

Block	Python	Meaning
variable	`power = 250`	a named box
list	`powers = [300, 350, 450]`	several values
loop	`for p in powers:`	repeat for each value
condition	`if p > cp: ... else:`	choose
function	`def work(p, t):  return p * t`	a reusable recipe
figure	`plt.plot(x, y)` then `plt.show()`	draw *y* against *x*

You will meet each of them in Notebook 1, with sport examples.

Notebooks: text and code on the same page

Notebook 2 – The critical power test

A **text cell**: explanations, questions, instructions.

```
slope, intercept = np.polyfit(times, work, 1)
print("CP =", round(slope), "W")
```

```
CP = 261 W
```

Your turn: how long can Alex hold 320 W?

```
t_320 = ...
```

- A **code cell** is run with the ▶ button or `Shift` + `Enter`; its result appears just below.
- Run the cells **in order**. You can change a number and run again: try, observe, understand.

Where we compute: Google Colab

Google Colab

- Notebooks in the web browser; the computation runs on Google's computers.
- Nothing to install. Laptop, tablet, any system.
- Needs a (free) Google account.
- Your copy is saved in your Google Drive.
- Figures are downloaded as images and inserted in Word, Google Docs or PowerPoint.

Backup: JupyterLite

- Same notebooks, Python runs *inside* the browser.
- No account. Address: `thomas-gomez.fr/python-lab`.

No AI assistant today

Today the aim is that *you* understand each line. AI tools can come later, once you can check what they write.

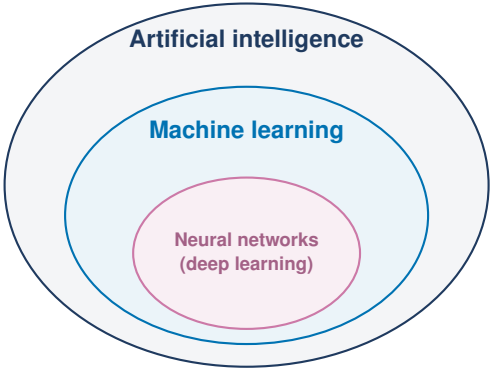
Errors are normal

- Programmers see error messages all day long.
- Read the **last line** of the red message.
- Most errors are typos, a missing bracket or a cell not run.
- If everything is strange: *Runtime* → *Restart session*, then *Run all*.
- Ask your neighbour, then the teacher.

Last line	Usually
`NameError`	typo, missing quotes, cell above not run
`SyntaxError`	missing `)`, `,` or `:`
`IndentationError`	wrong spaces at the start of a line

Artificial intelligence and learning

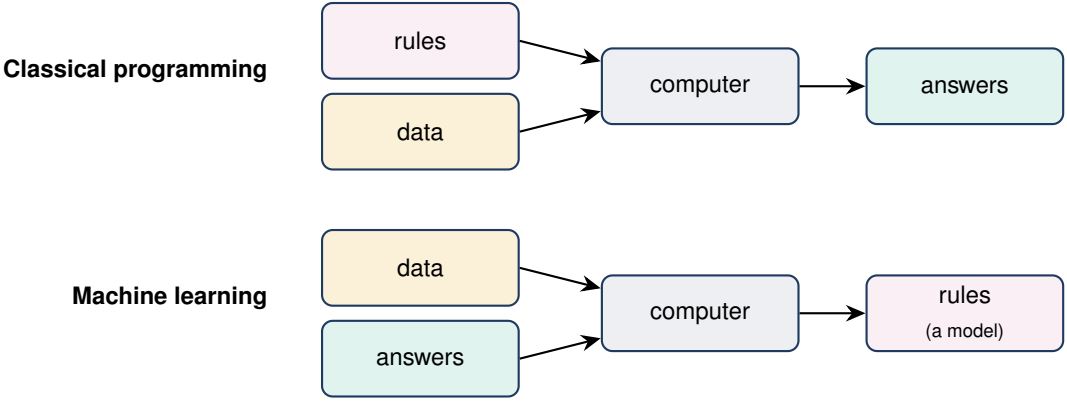
What is artificial intelligence?



- **Artificial intelligence (AI)**: programs that do tasks we thought needed human intelligence (play chess, recognise a face, translate, answer questions).
- **Machine learning**: the program is not given the rules; it **learns** them from examples.
- **Neural networks**: one family of learning methods; very large ones are behind image recognition and chatbots such as ChatGPT.

In sport: recognising the activity from the sensors of a watch, detecting events in match videos, predicting a performance from training data.

Writing the rules, or learning them



- Our W' tank and runner: the rules were written by physiologists and mathematicians.
- Machine learning: we give examples (power $\rightarrow$ time to exhaustion) and the computer finds a rule that reproduces them.

Supervised learning in three steps

Examples: an input and the right answer.

input: power (W)	answer: time (s)
300	505
350	228
410	140
...	...

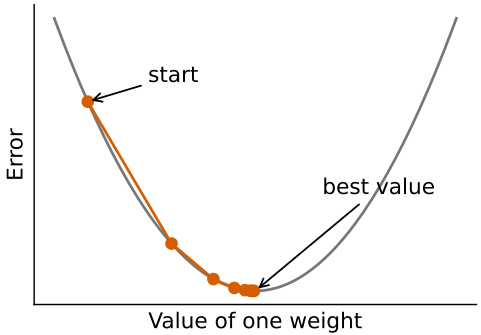
Keep some examples **apart** to test the model, like an exam with new questions.

```
model = make_forest()                    # 1. create
model.fit(train_power, train_time)       # 2. learn
model.predict([320, 450])                # 3. predict
```

- `fit`: the computer adjusts the model so that its answers are close to the right answers.
- `predict`: answers for inputs it has never seen.
- The same three steps for every method of the library `scikit-learn` [9].

What does “learning” mean? Reducing an error

Gradient descent: small steps downhill



1. A model has **knobs** (parameters, or *weights*).
2. The **error** measures how far its answers are from the right answers.
3. **Learning** = turning the knobs to make the error as small as possible.
4. Gradient descent: look at which direction goes down, take a small step, repeat thousands of times.

Message

You already did it: calibrating CP and W' with the least squares line is learning with 2 knobs. A neural network has hundreds or billions.

The learning loop in mathematics (1): model and loss

Example: learn CP and W' of Alex from 3 tests, as a network would.

Model, with 2 knobs a and b

$$\hat{y}_i = b + a\,t_i$$

t_i: time of test i; $\hat{y}_i$: predicted work; $a = \text{CP}$, $b = W'$.

Loss: the mean squared error

$$L(a,b) = \frac{1}{n}\sum_{i=1}^{n} \big(\underbrace{\hat{y}_i - y_i}_{e_i,\ \text{error}}\big)^2$$

y_i: measured work; $n = 3$ tests.

- $L \geq 0$, and $L = 0$ only if the line passes through every point.
- Learning = finding (a, b) that make L as small as possible: $(a^*, b^*) = \arg\min L(a,b)$.
- Squares: big errors count a lot, and L is a smooth "valley" (easy to differentiate).

The learning loop in mathematics (2): the gradient

Derivative $=$ slope of the loss when one knob moves a little:

$$\frac{\partial L}{\partial a} \approx \frac{L(a+h,\, b) - L(a,\, b)}{h} \qquad (h \text{ small})$$

With the school rule $(u^2)' = 2\,u\,u'$ and $\partial \hat{y}_i / \partial a = t_i$, $\partial \hat{y}_i / \partial b = 1$:

$$\frac{\partial L}{\partial a} = \frac{2}{n} \sum_i e_i\, t_i \qquad \frac{\partial L}{\partial b} = \frac{2}{n} \sum_i e_i$$

- The **gradient** $\nabla L = \left(\partial L/\partial a,\ \partial L/\partial b\right)$ points **uphill**.
- $\partial L/\partial a > 0$: increasing a increases the error $\Rightarrow$ decrease a.
- $\partial L/\partial a < 0$: increase a.
- $\nabla L = 0$ at the bottom of the valley: nothing more to learn.

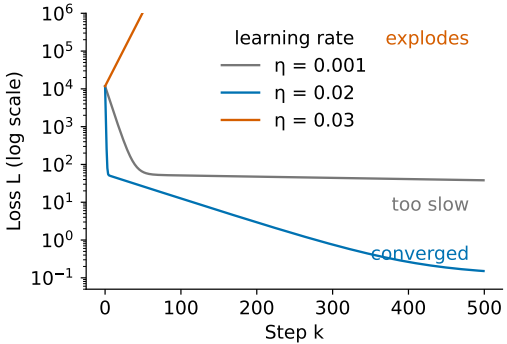
The learning loop in mathematics (3): the update rule

Repeat, for $k = 0, 1, 2, \ldots$

$$a_{k+1} = a_k - \eta \frac{\partial L}{\partial a}(a_k, b_k)$$
$$b_{k+1} = b_k - \eta \frac{\partial L}{\partial b}(a_k, b_k)$$

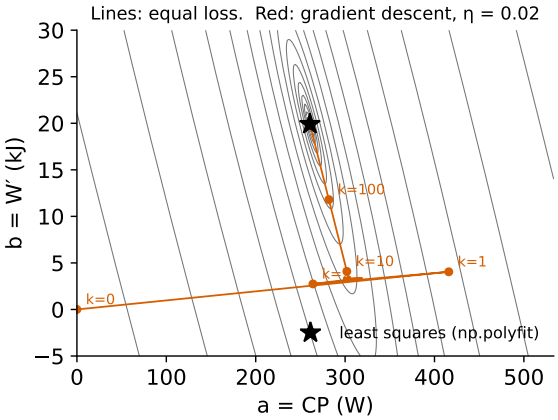
η (eta) is the **learning rate**: the size of the step.

- η too small: correct but very slow.
- η too large: each step jumps over the bottom of the valley, higher and higher: it explodes.
- Choosing η is part of the craft (in notebook 8 you try it).



Alex's three tests, 500 steps, start $a_0 = b_0 = 0$.

Gradient descent finds CP and W'



step k	loss L	CP (W)	W' (kJ)
0	11 770.3	0	0.0
1	1 608.0	416	4.0
2	259.6	264	2.7
10	47.5	302	4.1
100	12.6	282	11.8
1000	0.1	261	19.9

After 1000 steps: the same CP and W' as the least squares line of notebook 2. Here a formula exists; for a neural network there is none, and the loop is the only way.

The learning loop in Python

```
a, b = 0.0, 0.0                    # start anywhere
eta = 0.02                         # learning rate
for step in range(2000):           # repeat
    error = (b + a * t) - y        # e_i
    loss = np.mean(error ** 2)     # L(a, b)
    grad_a = 2 * np.mean(error * t) # dL/da
    grad_b = 2 * np.mean(error)    # dL/db
    a = a - eta * grad_a           # step downhill
    b = b - eta * grad_b
```

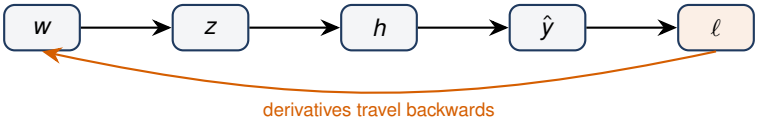
Every training has these four parts:

1. **predict** with the current knobs;
2. measure the **loss**;
3. compute the **gradient**;
4. **update** the knobs.

In a library, `model.fit(...)` runs this loop for you. In notebook 8 you write it yourself.

Many weights: the chain rule (back-propagation)

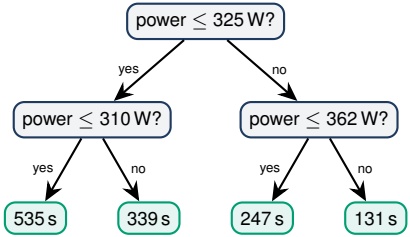
One neuron: $z = w\,x + b, \quad h = \tanh(z), \quad \hat{y} = c + a\,h, \quad \ell = (\hat{y} - y)^2$



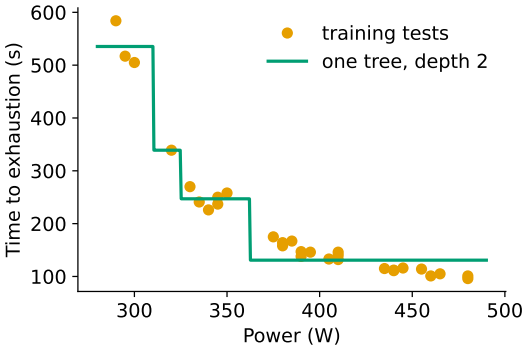
$$\frac{\partial \ell}{\partial w} = \underbrace{\frac{\partial \ell}{\partial \hat{y}}}_{2(\hat{y}-y)} \times \underbrace{\frac{\partial \hat{y}}{\partial h}}_{a} \times \underbrace{\frac{\partial h}{\partial z}}_{1-h^2} \times \underbrace{\frac{\partial z}{\partial w}}_{x}$$

- Multiply the local derivatives along the path from the loss back to the weight: **back-propagation** [12]. PyTorch and TensorFlow do it automatically for millions of weights.
- Vocabulary: an **epoch** is one pass over all examples; a **mini-batch** computes the gradient on a few examples at a time (stochastic gradient descent); **Adam** adapts the step of each weight (used in our notebooks).

A decision tree asks yes/no questions

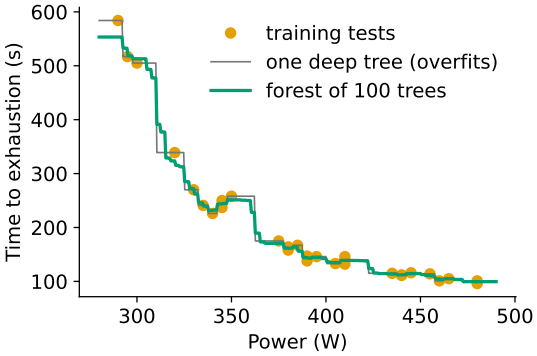


The questions and the answers (mean time of the tests in each box) are **learned** from the 30 training tests.



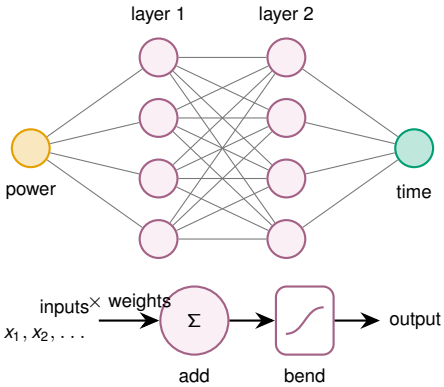
2 questions → 4 answers → a staircase.

Random forest: many trees vote



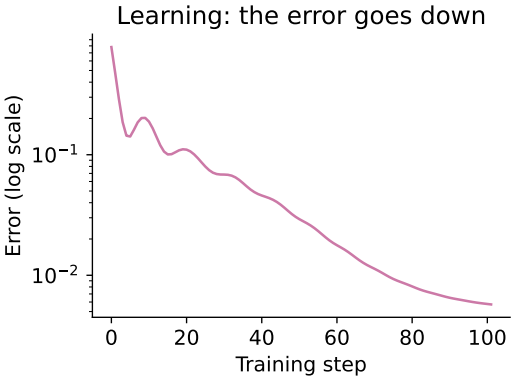
- A very deep tree passes through **every** training point, noise included: it learns by heart (overfitting) and fails on new tests.
- A **random forest** [2] grows 100 trees, each on a random part of the examples, and **averages** their answers.
- The errors of the trees partly cancel: smoother, more reliable.

A neural network: neurons in layers



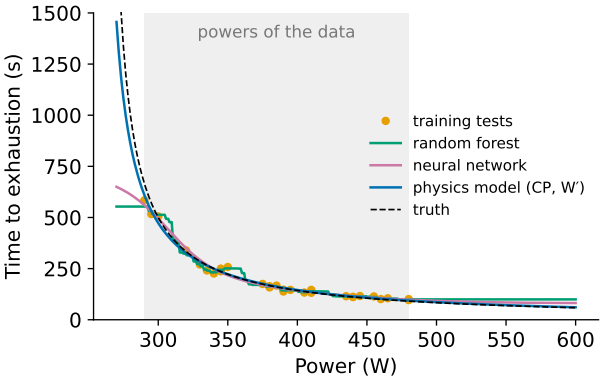
- A **neuron**: multiply each input by a **weight**, add, then **bend** the result with a smooth curve.
- Many neurons in **layers** can draw almost any curve.
- Our network: 2 layers of 20 neurons = about 500 weights.
- Training: gradient descent on all the weights at once (*back-propagation* [12]).

Training a network



- At the start the weights are random: the network answers anything.
- At each step the weights move a little downhill: the error decreases.
- It stops when the error no longer improves.
- Same principle, at a gigantic scale, for the chatbots: billions of weights, trained on texts.

Forest, network, or physics?

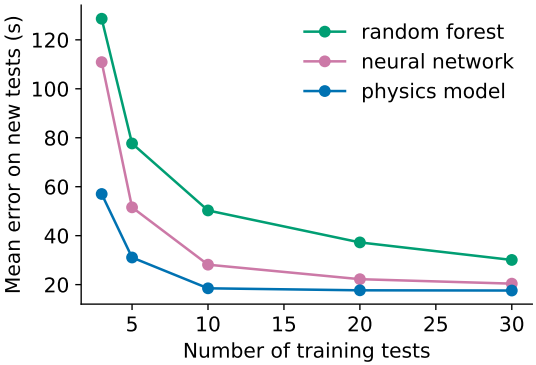


- **Inside** the data (grey band): all three are close.
- **Outside**: the forest repeats its last step; the network continues its curve anyhow.
- The physics model ($W'/(P - \mathrm{CP})$) keeps the right shape: it carries **knowledge**, not only data.

Message

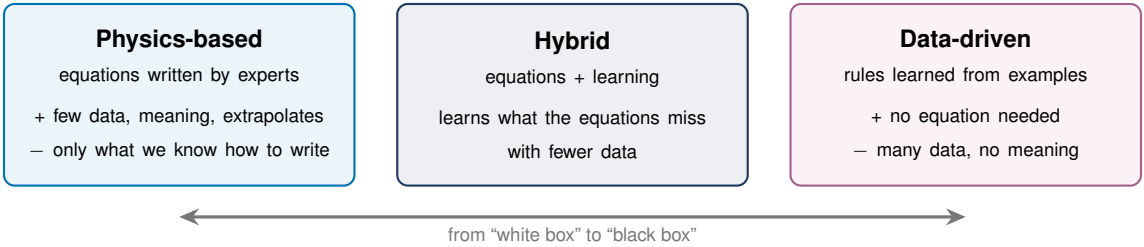
A learned model is reliable only where it has seen examples.

How much data does each method need?



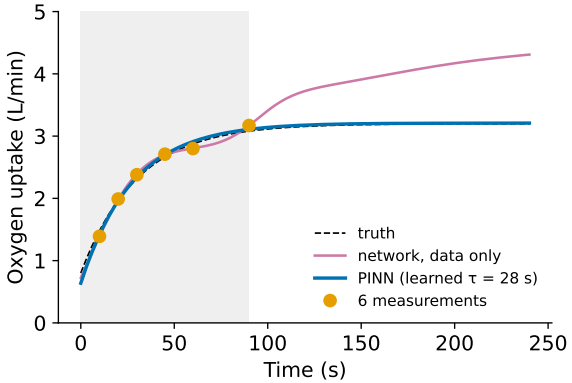
- Each model trained on 3 to 30 tests chosen at random, tested on 10 other tests.
- The physics model is good with **a few tests**: it only has 2 parameters to find.
- The forest and the network need many more examples to reach the same error.
- In sport, data are **expensive**: an exhaustion test per day, per athlete.

Hybrid learning: put the physics into the learning



- One hybrid method: the **physics-informed neural network** (PINN) [11, 6]. The network learns from the data *and* must respect an equation.

A physics-informed neural network (PINN)



Oxygen uptake at the start of an exercise [10]:

$$\tau \frac{dV}{dt} = V_{\text{end}} - V$$

Only 6 measurements, in the first 90 s.

$$\text{error} = \underbrace{\text{error on the data}}_{\text{as usual}} + \lambda \times \underbrace{\text{error on the equation}}_{\text{physics}}$$

- Data only: the network invents the end of the curve.
- With the equation: right curve, and the athlete's τ and V_{end} are learned.

The PINN learning loop: same loop, richer loss

Network $V_\theta(t)$ (θ: all its weights); unknown physiology: τ, V_{end}.

$$L(\theta, \tau, V_{\text{end}}) = \underbrace{\frac{1}{n}\sum_{i=1}^{n}\left(V_\theta(t_i) - y_i\right)^2}_{\text{data: 6 measurements}} + \lambda \underbrace{\frac{1}{m}\sum_{j=1}^{m}\left(\tau\,\frac{dV_\theta}{dt}(t_j) - V_{\text{end}} + V_\theta(t_j)\right)^2}_{\text{physics: checked at } m=60 \text{ times, no measurement needed}}$$

1. predict V_θ and its derivative dV_θ/dt (chain rule again: exact, no measurement);
2. loss $=$ data $+ \lambda\times$ physics;
3. gradient with respect to θ, τ *and* V_{end};
4. one step downhill for all of them; repeat 4000 times.

- The equation acts as **free extra data** between and after the measurements.
- λ balances data and physics: too small, the physics is ignored; too large, the data are.
- τ and V_{end} come out of the training: parameters with a physiological meaning.

What to remember about AI for a digital twin

Machine learning is strong when

- there are many examples;
- the rules are unknown or too complex to write;
- predictions stay inside the range of the data.

Physics is strong when

- data are few and expensive;
- we must predict new situations;
- the parameters must mean something for the coach.

Message

Hybrid twins combine both. Whatever the method, test it on new data and say how sure it is. A chatbot is also a learned model: it can be wrong, and you must be able to check it.

The lab

Session 1: the cyclist's twin (3 hours)

	Notebook	You learn	Time
0	First steps in Colab	run a cell, read an error, save	15 min
1	Python basics	variables, lists, loops, functions, figures	40 min
2	The critical power test	data $\rightarrow$ model $\rightarrow$ calibration $\rightarrow$ prediction	35 min
		break	
3	A simple digital twin	simulate a session second by second	40 min
4	Uncertainty	noise, Monte Carlo, choice of the tests	25 min
5	Mini-project	coach for a day	the rest

Work in pairs. Every notebook says what to run, what to change and what you should see.

Mini-project: coach for a day

Each pair coaches one of four athletes (A, B, C, D), who did three exhaustive tests.

1. **Calibrate**: CP and W' (Figure 1).
2. **Predict**: time to exhaustion at three powers.
3. **Plan**: an interval session that brings the reserve below 10 % without exhaustion (Figure 2).
4. **Doubt**: the uncertainty on CP (Figure 3).

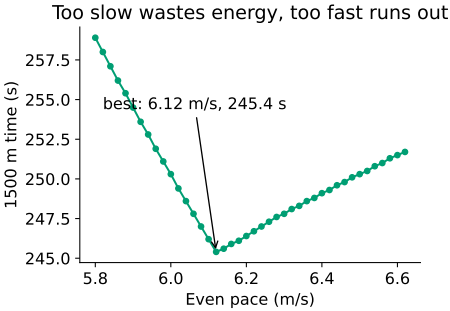
To hand in

A report of 1 to 2 pages (Word or Google Docs) with your three figures and short answers. Template in the student guide.

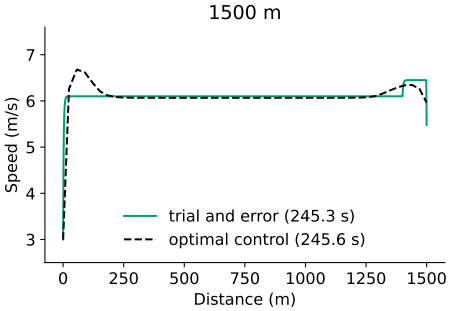
Figures

`save_figure("fig1_calibration")` downloads the image; then *Insert* → *Image* in your document.

Session 2: pacing a 1500 m



Try many even paces with a loop: there is a best one.



Your strategy against the optimal one computed by mathematicians.

Runner model of [1]: speed, propulsive force, anaerobic energy refilled by the aerobic power. A strategy is a Python function; the simulator turns it into a race.

Session 3: learning from data (3 hours)

	Notebook	You learn	Time
8	Learning from examples	decision tree, random forest; a learning loop written by hand; neural network; training and test sets; overfitting	90 min
		break	
9	A physics-informed network	a network that respects the oxygen uptake equation	45 min
10	Mini-project	*physics or data?* how many data does each method need	the rest

The report has three figures: error against the number of tests, error against the number of measurements, and the time constant τ learned by the PINN.

Let’s start

1. Open the link given by the teacher (Google Drive folder, or `thomas-gomez.fr/python-lab`).
2. Sign in with your Google account (create one if needed: it is free).
3. Open `00_first_steps_in_colab`, then *File* $\rightarrow$ *Save a copy in Drive*.
4. Follow the student guide, page by page.

Message

Do not try to understand everything at once. Run, change one number, run again, observe. This is how everybody learns to program.

References

References i

- Aftalion A, Trélat E (2021). Pace and motor control optimization for a runner. *J Math Biol* 83:9. doi:10.1007/s00285-021-01632-z
- Breiman L (2001). Random forests. *Mach Learn* 45:5–32. doi:10.1023/A:1010933404324
- Boillet A, Messonnier LA, Cohen C (2024). Individualized physiology-based digital twin model for sports performance prediction: a reinterpretation of the Margaria–Morton model. *Sci Rep* 14:5470. doi:10.1038/s41598-024-56042-0
- Boillet A, Noble M, Sachet I, Messonnier LA, Cohen C (2024). Individualized optimal strategy in team pursuit for track cycling. *Sci Rep* 14:25308. doi:10.1038/s41598-024-75963-4
- Gomez T (2026). Which quantities of a bioenergetic digital twin of an athlete can be individualized? Identifiability of a two-compartment model of critical power: a simulation study. Manuscript submitted to PeerJ; code and data doi:10.5281/zenodo.23600981

References ii

- Karniadakis GE, Kevrekidis IG, Lu L, Perdikaris P, Wang S, Yang L (2021). Physics-informed machine learning. *Nat Rev Phys* 3:422–440. doi:10.1038/s42254-021-00314-5
- Jones AM, Vanhatalo A, Burnley M, Morton RH, Poole DC (2010). Critical power: implications for determination of $\dot{V}O_2$max and exercise tolerance. *Med Sci Sports Exerc* 42:1876–1890. doi:10.1249/MSS.0b013e3181d9cf7f
- Monod H, Scherrer J (1965). The work capacity of a synergic muscular group. *Ergonomics* 8:329–338. doi:10.1080/00140136508930810
- Pedregosa F, Varoquaux G, Gramfort A, et al. (2011). Scikit-learn: machine learning in Python. *J Mach Learn Res* 12:2825–2830
- Poole DC, Jones AM (2012). Oxygen uptake kinetics. *Compr Physiol* 2:933–996. doi:10.1002/cphy.c100072

References iii

- Raissi M, Perdikaris P, Karniadakis GE (2019). Physics-informed neural networks: a deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *J Comput Phys* 378:686–707. doi:10.1016/j.jcp.2018.10.045
- Rumelhart DE, Hinton GE, Williams RJ (1986). Learning representations by back-propagating errors. *Nature* 323:533–536. doi:10.1038/323533a0
- Skiba PF, Chidnok W, Vanhatalo A, Jones AM (2012). Modeling the expenditure and reconstitution of work capacity above critical power. *Med Sci Sports Exerc* 44:1526–1532. doi:10.1249/MSS.0b013e3182517a80