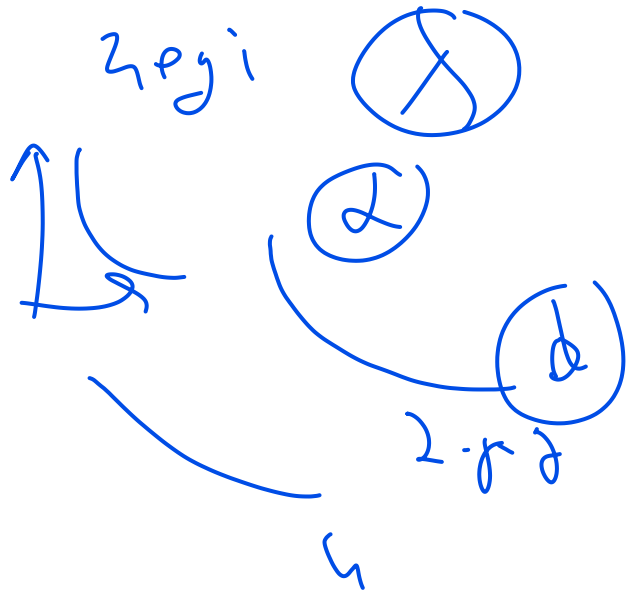
Outline



Parameters vs. hyperparameters

The simple test: can you *compute* it from the data alone, or do you have to *choose* it? Compute $\Rightarrow$ parameter. Choose $\Rightarrow$ hyperparameter.

Parameters θ are what the model *learns* from data.

- Linear regression: the coefficients (one per feature + intercept).
- Decision tree: the split points and feature choices at each node.
- Neural net: all the weights and biases.

Hyperparameters λ are what the model *doesn't* learn - you have to choose them.

- Ridge regression: `alpha` (regularization strength).
- Decision tree: `max_depth`, `min_samples_leaf`.
- Neural net: learning rate, number of layers, batch size, weight decay.

Different hyperparameter choices $\Rightarrow$ different fitted models $\Rightarrow$ different generalization.
Tuning = systematically picking λ to optimize a validation/CV score.

The bilevel optimization view

Tuning is an *outer* optimization wrapping the model's own *inner* optimization.

Outer loop (tuning): pick best λ^*

Inner loop (training): given $\boldsymbol{\lambda}$, solve
$\hat{\boldsymbol{\theta}}(\boldsymbol{\lambda}) = \arg\min_{\boldsymbol{\theta}} \mathcal{R}_{\text{emp}}(\boldsymbol{\theta}; \boldsymbol{\lambda})$

pass $\boldsymbol{\lambda}$ in $\rightarrow$ get back $\hat{\boldsymbol{\theta}}(\boldsymbol{\lambda})$, score it on validation/CV $\rightarrow$ outer search picks next $\boldsymbol{\lambda}$

In plain English: the outer loop tries hyperparameter values; the inner loop fits the model with those values. *One outer step = one full model training.*

Choices on outer search strategy:

- **Manual** - you propose $\boldsymbol{\lambda}$ values from intuition.
- **Grid** - exhaustive over a Cartesian product.
- **Random** - sample uniformly from distributions.
- **Bayesian** - build a surrogate model of score vs. $\boldsymbol{\lambda}$, query smartly.

Manual tuning

We know tuning means searching for the best $\boldsymbol{\lambda}$. The simplest search needs no tooling at all - a human proposes a value, reads the validation score, and adjusts. We start here to build intuition before automating.

Manual tuning: "grad student descent"

The simplest strategy: a human proposes one $\boldsymbol{\lambda}$, evaluates, then proposes another based on what they saw.

Typical workflow:

1. Start with the library's defaults. Get a baseline.
2. Identify the most-likely-impactful hyperparameter (domain knowledge or quick experiments).
3. Sweep it across ~ 5 values, plot validation score, pick the best.
4. Fix that value, move to the next hyperparameter, repeat.
5. Stop when changes don't improve validation score.

Also called *manual coordinate descent* - you adjust one hyperparameter at a time.

Manual is fast to start and slow to scale. It works for 1-2 HPs and an experienced practitioner. For everything else, the automated methods coming up are usually better. Treat the rest of this deck as upgrades, not replacements.

When manual works (and when it doesn't)

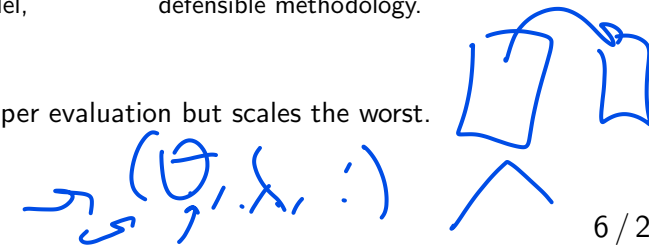
Manual is best when:

- You have strong intuition about the model (you've trained 50 Ridge regressions already).
- The search space is small (1-2 hyperparameters).
- Each evaluation is slow (you don't want to launch 100 jobs by accident).
- You're *developing* a sense for the model, not just optimizing.

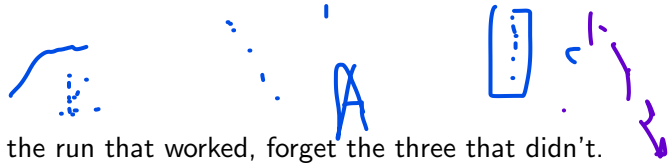
Manual breaks down when:

- More than ~ 3 hyperparameters interact (you can't visualize the surface).
- You can't afford to be inconsistent across runs.
- Multiple people work on the same model (reproducibility).
- You're shipping to production and need a defensible methodology.

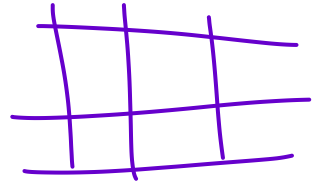
Manual tuning teaches you the most per evaluation but scales the worst.



Pitfalls of manual tuning



1. **Confirmation bias.** You notice the run that worked, forget the three that didn't. Mitigation: write down every run, including failures.
2. **Validation-set abuse.** The 50th hyperparameter setting you tried still gets evaluated on the same validation set. You're effectively overfitting validation. Mitigation: cross-validation, separate test set, count your evaluations.
3. **No reproducibility.** "I'll try `alpha=0.3` next... or was it 0.03?" Mitigation: a notebook log, or skip manual and use a configured automated search.
4. **Local optimum.** Coordinate descent gets stuck. The best Ridge `alpha` *given the current data preprocessing* may differ from the best after you change features. Mitigation: redo the whole sweep occasionally.



Grid search

Manual tuning doesn't scale and isn't reproducible - two people get two answers. Let's hand the search to the computer: list candidate values for each hyperparameter and evaluate *every* combination, systematically.

Grid search: exhaustive over a Cartesian product

Pick a finite list of candidate values for each hyperparameter. Evaluate *every* combination.

Example for Ridge with polynomial features:

Hyperparameter	Candidate values
`alpha`	$\{10^{-3}, 10^{-2}, 10^{-1}, 1, 10, 100\}$
`poly_degree`	$\{1, 2, 3, 4, 5\}$

Total: $6 \times 5 = 30$ configurations. With 5-fold CV, $30 \times 5 = 150$ model fits.

Exhaustive, reproducible, embarrassingly parallel. Also potentially *very* expensive.

Grid search: combinatorial blowup

Suppose you tune a model with 4 hyperparameters, 5 candidate values each, using 5-fold CV. **Try to predict** before reading on:

- How many model fits will grid search require?
- If each fit takes 30 seconds, how long does the whole search take (single-threaded)?

Answers:

- $5^4 \times 5 = 625 \times 5 = \mathbf{3125}$ model fits.
- $3125 \times 30\text{s} \approx 26$ hours single-threaded.

Add one more hyperparameter (5 values): **15**,**625** fits, **5.4** days. The curse of dimensionality strikes early.

Grid search is impractical above 3 hyperparameters with non-trivial ranges. The next section introduces random search, which solves this.

Code: GridSearchCV with Pipeline

```python
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.linear_model import Ridge
from sklearn.pipeline import Pipeline

pipe = Pipeline([("poly", PolynomialFeatures()),
                 ("scale", StandardScaler()),
                 ("ridge", Ridge())])

grid = {
    "poly__degree": [1, 2, 3, 4, 5],
    "ridge__alpha": [1e-3, 1e-2, 1e-1, 1, 10, 100],
}

gs = GridSearchCV(pipe, grid, cv=5,
                  scoring="neg_mean_squared_error", n_jobs=-1)
gs.fit(X_train, y_train)
print(gs.best_params_, -gs.best_score_)
```

n_jobs=-1 uses all CPU cores. Parameters are accessed as step_name__param_name.

When grid is OK

- $\leq 2-3$ hyperparameters AND each has ≤ 6 candidate values.
- Each model fit is cheap (small data, simple model).
- Reproducibility / publication: "we evaluated all X configurations."
- **Discrete, narrow, well-known values** (e.g., `max_depth` $\in \{3, 5, 7\}$ for trees) - there's nothing to randomize over.
- Inputs have clearly preferred discrete values from prior knowledge.

Once you have 4+ hyperparameters or 10+ values each, grid stops being practical. Random search is almost always better.

Random search

Grid search explodes combinatorially and spends equal budget on axes that barely matter. What if, instead of a rigid lattice, we sampled each hyperparameter at random - so *every* trial probes a fresh value on *every* axis?

Why grid wastes compute in high dimensions

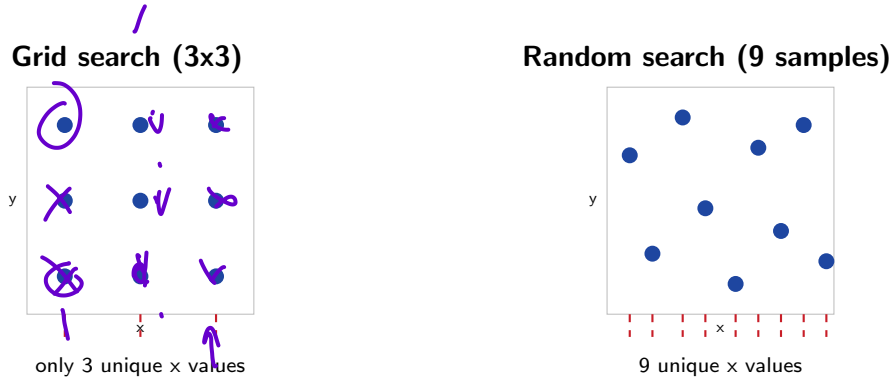
Most hyperparameters don't matter equally. For a typical Ridge + polynomial model:

- `alpha` matters a lot.
- `poly_degree` matters a lot.
- Other small dials (e.g., `tol`, `max_iter`) barely affect the score.

Grid search spends *equal* compute on each axis. If 9 out of 10 axes are flat, you've used 90% of your budget exploring flat directions.

Random search samples each hyperparameter from a distribution. Every trial visits a new value on *every* axis - including the ones that matter.

Bergstra–Bengio: random > grid in high dimensions



Random search gets 9 distinct samples along the important axis; grid gets 3. The wider the gap between effective dimensionality and total dimensionality, the bigger the random-search win.

Default: random search for ≥ 3 hyperparameters. With ~ 60 trials, random search typically matches a 10^d grid. Reach for grid only when you have specific known values.

Code: RandomizedSearchCV

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import loguniform, randint

dist = {
    "ridge__alpha":   loguniform(1e-4, 1e3),    # continuous, log-
        scale
    "poly__degree":   randint(1, 7),            # discrete 1..6
}

rs = RandomizedSearchCV(pipe, dist, n_iter=60, cv=5,  # n_iter =
    budget
        scoring="neg_mean_squared_error",
        random_state=509, n_jobs=-1)
rs.fit(X_train, y_train)
print(rs.best_params_, -rs.best_score_)
```

`loguniform`/`uniform`/`randint` for log-scale / bounded / discrete HPs. **Why random is nice:** stop anytime, bound the budget with `n_iter`, mixes with continuous distributions. Modern alt: `HalvingRandomSearchCV`.

Bayesian optimization & beyond

Random search throws *independent* darts: each trial ignores everything the previous trials revealed. We'd rather choose each new experiment *using the results so far* - spending trials where the score already looks promising, not blindly.

Bayesian optimization (and other smart searches)

Random search treats every trial as independent. **Bayesian optimization (BO)** instead *learns* from the scores seen so far: it fits a probabilistic **surrogate** of score-vs-$\boldsymbol{\lambda}$, then picks the next $\boldsymbol{\lambda}$ that best trades off *exploit* (where the mean looks high) against *explore* (where it's still uncertain). Every trial uses information from all the previous ones.

- **Tools:** Optuna (TPE sampler - the modern default), Hyperopt, Ray Tune. They "just work" across mixed (float / int / categorical) spaces.
- **Worth it when** each evaluation is expensive ($\gtrsim 1$ min) *and* you have ≥ 4 interacting HPs. Cheap evals, or a big parallel cluster $\rightarrow$ plain random search is simpler and nearly as good.

We already built this machinery in the optimization module. The full theory - surrogate models, Gaussian processes, acquisition functions (EI / UCB / PI) - is in **Math 14: Bayesian Optimization**.

Evolutionary algorithms (CMA-ES, genetic algorithms) are another derivative-free way to tune: population-based, gradient-free, strong on rugged / non-differentiable spaces - see **Math 13: Evolutionary Algorithms**.

In practice

We now have four search methods. But which one you pick matters less than *how* you use it: how you shape the search space, how you avoid burning compute, and what a real run actually looks like.

Designing the search space

The method matters less than the *space* you hand it. Three rules:

1. **Right scale.** *Log-uniform* for anything spanning orders of magnitude (learning rate, regularization λ); *uniform* for bounded ratios (`l1_ratio`, momentum). Linear-spacing λ as $\{0.001, 0.01, 0.1\}$ wastes almost the whole range.
2. **Right type.** Integer for counts (`max_iter` iterations, polynomial degree); float for continuous; categorical for unordered choices (penalty L1/L2, solver).
3. **Bracket the optimum.** Ranges wide enough that the best value lands *inside*, not at an endpoint.

Hyperparameter kind	Distribution
learning rate, regularization λ (`alpha`)	log-uniform
`l1_ratio` (ElasticNet), momentum	uniform (bounded)
iterations (`max_iter`), polynomial degree	integer
penalty (L1 / L2), solver	categorical

The #1 search mistake: a range that doesn't contain the optimum – if your best value sits at a boundary, widen and re-run. **Naming gotcha:** sklearn calls λ `alpha`, and you search it on a *log* scale. *Conditional* HPs (`l1_ratio` only matters for ElasticNet, not Ridge) need conditional spaces - Optuna handles them naturally.

Specifying the space: Optuna vs Hyperopt

Same job, two APIs – tuning a regularized linear model on λ (`alpha`) and the iteration budget (`max_iter`). **Optuna** is *define-by-run* (sample inside the objective); **Hyperopt** is *define-then-run* (a space dict up front).

Optuna - `trial.suggest_*`

```
def objective(trial):
  alpha = trial.suggest_float(
      "alpha", 1e-4, 1e3, log=True)
  n_iter = trial.suggest_int(
      "max_iter", 100, 5000)
  model = Lasso(alpha=alpha,
                max_iter=n_iter)
  return cv_score(model)
```

Hyperopt - `hp.*`

```
space = {
 "alpha": hp.loguniform(
    "alpha", log(1e-4), log(1e3)),
 "max_iter": hp.uniformint(
    "max_iter", 100, 5000),
}
# fmin(objective, space,
#      algo=tpe.suggest, max_evals
    =60)
```

Gotcha: Optuna's `log=True` takes the *raw* bounds; Hyperopt's `hp.loguniform` takes them in *log space* (`np.log(low)`, `np.log(high)`). Mixing these up is a classic silent bug. Drive them with `study.optimize(objective, n_trials=60)` or `fmin(fn, space, algo=tpe.suggest, max_evals=60)`.

Pruning: stop losing trials early

Most trials are clearly losing well before they finish. **Pruning** kills them early and reallocates the budget. Two levels:

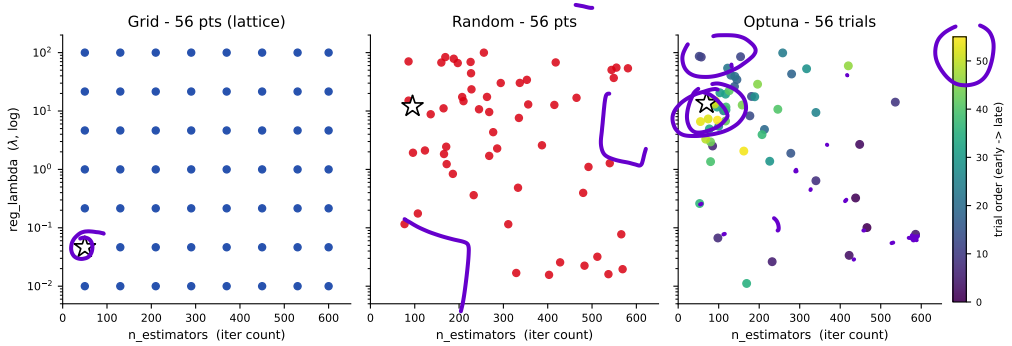
- **Model-level early stopping.** For boosted trees / NNs, stop adding rounds / epochs once a validation metric stalls (`early_stopping_rounds`, Keras `EarlyStopping`). Cuts *each* trial's cost.
- **Search-level pruning.** Kill an underperforming *trial* mid-training. Optuna: `MedianPruner` (drop trials below the running median), `HyperbandPruner` (multi-fidelity).

```
study = optuna.create_study(pruner=optuna.pruners.MedianPruner())
def objective(trial):
    for step in range(100):                # epochs / boosting rounds
        val = train_one_more_step()
        trial.report(val, step)
        if trial.should_prune():           # below the median so far?
            raise optuna.TrialPruned()
    return val
```

Pruning + early stopping commonly cut search time 2-5× - biggest win when training has a natural "step" (epochs, rounds).

Benchmark: which points does each method try?

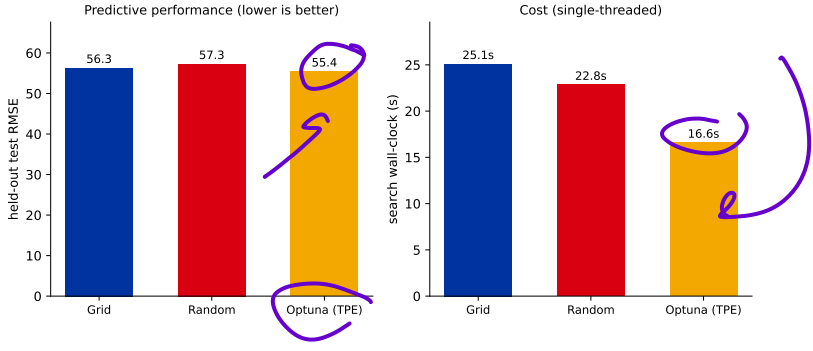
Tune 2 knobs of a boosted-tree model (a black box; trees come in ch.3) on `diabetes`: `n_estimators` (iteration count) and `reg_lambda` (λ). Same *56-trial* budget, 5-fold CV; $\star$ = best.



The sampling strategy is the whole story. **Grid** fills a rigid lattice; **random** scatters uniformly; **Optuna** is *adaptive* - early trials (dark) explore widely, late trials (yellow) cluster on the good region. Here it also reached the best score (test RMSE 55.4 vs 56.3 / 57.3) and the shortest search time (next slide).

Benchmark: score and cost

Same 56-trial runs, summarized: best held-out test RMSE (predictive performance) and total search time per method.



Equal budget, different payoff. All three do the same 280 fits and land close on accuracy (RMSE 55-57) - on a small 2-HP space no method works magic. Optuna is fastest: it spends most trials on cheap, small-`n_estimators` models and skips the expensive big ones grid and random keep evaluating. (Wall-clock is single-threaded and machine-dependent - the *ordering* is the point, not the exact seconds.)

A practical workflow

1. **Pick a few HPs.** Tune the 2-3 that matter most - not everything at once.
2. **Get a feel by hand.** Try a handful of values manually to see roughly where the good region is and what each knob does.
3. **Set ranges from that.** Use what you learned to pick (log-)ranges that bracket the good region.
4. **Then automate.** Launch a Bayesian search and let it run while you have lunch.
5. **Save progress.** Persist the study so you can pause, resume, and keep optimizing overnight.

```
study = optuna.create_study(
    study_name="ridge_tuning",
    storage="sqlite:///optuna.db",  # persist trials to disk
    load_if_exists=True)            # resume the saved study
study.optimize(objective, n_trials=50)  # rerun anytime to add more
```

New to a model? Do lots of manual search first. You learn far more per trial by hand - what each knob does, what "good" looks like - than by watching an automated search you don't yet understand. Automate once you have the intuition.

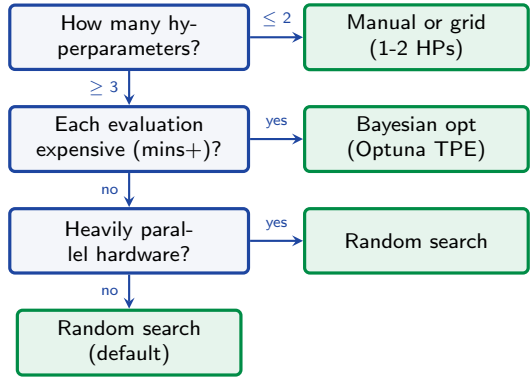
The compute budget reality

Hyperparameter tuning consumes most of an ML project's compute. Practical patterns:

- **Subsample.** Tune on 10% of training data, retrain best config on full data. Often within 1% of fully-tuned.
- **Early stopping + pruning.** Stop weak trials before they finish - within-model early stopping and search-level pruning (see the pruning frame). Cuts each trial's cost 2-5x.
- **Multi-fidelity methods.** Hyperband / Successive Halving: run many trials cheaply, gradually allocate more budget to the survivors.
- **Stop when score plateaus.** Track best-so-far; if it hasn't improved in 20 trials, you're done.

Honest reporting: the score you tuned to optimize is *optimistically biased*. Report the test-set score after tuning is complete (or use nested CV - L05b). *Conceptually: HP tuning navigates the bias-variance tradeoff (L01d2). Each HP setting moves the model along the U-shape; CV finds the sweet spot empirically.*

Decision flowchart



Defaults: ≤ 2 HPs $\rightarrow$ grid is fine; otherwise random search; if compute is the bottleneck, Bayesian opt.

Whatever method: use subsampling, early stopping, and pruning to multiply your effective budget (see compute-budget frame).

Recap

- **Parameters** are learned from data. **Hyperparameters** are chosen by you. Tuning = outer optimization wrapping the inner fit.
- **Manual.** Cheap, intuition-building, doesn't scale. Watch for confirmation bias and validation-set abuse.
- **Grid.** Exhaustive over Cartesian product. Fine for ≤ 2 HPs; explodes in high dimensions.
- **Random.** Default for ≥ 3 HPs. Each trial is a fresh sample on every axis. Easy to budget and parallelize. Beats grid when effective dimensionality is low.
- **Bayesian & beyond.** Models the score-vs-HP surface and queries smartly; best when each evaluation is expensive. Optuna's TPE is the modern default. Full theory in **Math 14** (Bayesian opt) and **Math 13** (evolutionary algorithms).
- **In practice.** Design the space well (log-uniform, bracket the optimum), prune weak trials, and remember equal-budget benchmarks show the win is often *speed*, not accuracy.
- **Bias-variance lens (L01d2):** HP tuning navigates the bias-variance U. Each candidate setting moves the model along the U; CV finds the sweet spot. Without this lens HP tuning feels arbitrary; with it, it's purposeful.
- **Honest reporting.** Tuning inflates the score; report on a held-out test set or use nested CV (L05b).