Poraquê

User Guide

Predicting charge and kinetic energy densities
from atomistic geometry

Leandro Seixas

Version 26.9.15

Overview

Poraquê predicts the electronic structure of a crystal from its geometry alone. Given a `POSCAR`, an `INCAR` and a `POTCAR`, it produces the valence charge density and the kinetic energy density — with no wavefunctions and no self-consistency cycle.

It does this by chaining two learned operators:

$$\{\texttt{POSCAR},\ \texttt{INCAR},\ \texttt{POTCAR}\} \xrightarrow{\text{analytic}} \texttt{EXTCAR} \xrightarrow{\text{Model 1}} \texttt{CHGCAR} \xrightarrow{\text{Model 2}} \texttt{TAUCAR}$$

The first step is closed-form; only the two field-to-field maps are learned. Every output is written in `CHGCAR` format.

Who this guide is for

This is the practical guide: how to lay out data, train, predict, and read the results. It assumes you can run a plane-wave DFT calculation and want to use Poraquê as a tool.

For the physics and the architecture — why a Fourier neural operator, what the models correspond to in density-functional theory, how the external potential is reconstructed from the pseudopotential tables — see the companion *Technical Guide* in `latex/technical_guide/`.

What you need

Requirement	Note
Python 3.11 or newer	see Chapter 1
A set of DFT calculations	`CHGCAR` and `TAUCAR` per structure
An accelerator (optional)	CUDA or Apple Metal; CPU works
`latexmk` (optional)	only for the automatic PDF reports

Caveat

Poraquê learns from the data you give it. The published results were obtained on seventeen platinum supercells, and they measure interpolation between geometries of one element — plus, weakly, extrapolation across cell size, where the error roughly doubles. Nothing in this guide should be read as evidence that a model trained on one chemistry transfers to another — validate on your own held-out structures before trusting a prediction.

Contents

Overview **1**

1 Installation **4**
1.1 From PyPI . . . 4
1.2 From source . . . 4
1.3 Console commands . . . 4
1.4 Python version . . . 5
1.5 Checking the installation . . . 5
1.6 NVIDIA GPUs . . . 6
1.7 Faster CPU inference . . . 9
1.8 Optional components . . . 10

2 Preparing data **11**
2.1 Directory layout . . . 11
2.2 Grids may differ between materials . . . 11
2.3 Checking your data . . . 11
2.4 Downloading from the Materials Project . . . 12
2.5 The whole database, not a chemical space . . . 13
2.6 What an MP download does and does not contain . . . 15
2.7 The external potential without a POTCAR . . . 15
2.8 Training on several sources at once . . . 16

3 Training **18**
3.1 The short version . . . 18
3.2 One model, all structures . . . 19
3.3 Measuring generalisation . . . 20
3.4 Reading the output . . . 20
3.5 Reference numbers . . . 21
3.6 Options worth knowing . . . 21

4 Predicting a new structure **22**
4.1 Choosing the grid . . . 22
4.2 Matching a VASP grid . . . 22
4.3 Comparing against a reference . . . 24
4.4 Restarting VASP from a predicted density . . . 24
4.5 Sanity checks the script performs . . . 26
4.6 Visualising . . . 26

5 Energies and the ASE calculator **27**
5.1 The energy expression . . . 27
5.2 Choosing the exchange-correlation functional . . . 28
5.3 Computing energies from files . . . 28
5.4 The ASE calculator . . . 29
5.5 What the number is not . . . 32

6 Charges and population analysis **34**
6.1 Charge conservation . . . 34
6.2 Partial charges . . . 35
6.3 What these charges are not . . . 37
6.4 In the reports . . . 37

7 Fine-tuning **38**
7.1 What comes from the checkpoint . . . 38
7.2 Freezing the lifting path . . . 38
7.3 Configuration . . . 39
7.4 LoRA — adapting a model that does not fit . . . 39
7.5 Output, and the `.poraque` format . . . 40

8 Symbolic distillation **42**
8.1 What it can and cannot find . . . 42
8.2 The physical variables . . . 43
8.3 Operator constraints . . . 44
8.4 Figures . . . 44
8.5 Regularization in vacuum . . . 44
8.6 Configuration . . . 45
8.7 Output . . . 46
8.8 The Pareto knee . . . 46
8.9 Rounding . . . 46
8.10 Physical asymptotic compliance . . . 47

9 Configuration reference **49**
9.1 How a value is resolved . . . 49
9.2 Optional features are blocks . . . 49
9.3 Every key is optional . . . 50
9.4 Top level . . . 51
9.5 `data` — where the fields come from . . . 51
9.6 `model` — the operator's shape . . . 55
9.7 Numerical precision . . . 59
9.8 `training` — how it is fitted . . . 61
9.9 `output` — what is written . . . 67
9.10 `symbolic` — distilling a formula . . . 69
9.11 Worked examples . . . 71

10 Troubleshooting **72**
10.1 The run stops immediately . . . 72
10.2 The results look wrong . . . 72
10.3 Grids and shapes . . . 72
10.4 Devices . . . 73
10.5 Reports . . . 73

CHAPTER 1

Installation

1.1 From PyPI

The usual way:

```
pip install poraque
```

1.2 From source

What you want if you intend to change anything — the editable install means an edit to the source takes effect without reinstalling:

```
git clone https://github.com/seixas-research/poraque.git
cd poraque
pip install -e .
```

Either way this installs NumPy, SciPy, ASE, Matplotlib, PyYAML and PyTorch. Symbolic distillation (Chapter 8) needs one optional extra, SymPy, and only to render its result and check its limits — the search itself runs on the NumPy and SciPy installed above:

```
pip install "poraque[symbolic]"
```

1.3 Console commands

Installing also registers five commands, which run from *any* directory once the environment is active — no path to a script, no `cd` into the repository:

Command	Does
`poraque-train`	trains the operators
`poraque-inference`	predicts a new structure
`poraque-mp`	downloads densities from the Materials Project
`poraque-committee`	**calibration**: does ensemble disagreement predict error?
`poraque-active-learning`	**selection**: which structures should the next DFT runs be spent on?
`poraque-vasp`	writes the VASP inputs that read a prediction back: `bands`, `dos`, `energy`; and `chgcar`, which writes a field store out as a `CHGCAR`

The last two are not the same command twice

They are the two halves of one loop, and they differ in the data they run on — which decides everything else.

`poraque-committee` runs on a **labelled** dataset (inputs *and* targets), so the true error is available to correlate the disagreement against. It answers *is this measure worth trusting?* and produces a Spearman coefficient. It costs nothing: the DFT is already done.

`poraque-active-learning` runs on an **unlabelled** pool (inputs only, targets not yet computed). It answers *which structures do I compute next?* and produces a ranking and a transfer into the training set. It costs the DFT runs it selects.

Run the first one first. Both share the same committee, the same divergence and the same ranking table, which is why their output looks alike — what differs is what the number is *for*.

Each is the `main()` of the script of the same name, with the hyphen written as an underscore: `python scripts/poraque_train.py` is equivalent to `poraque-train` and needs nothing installed. Both forms take identical arguments and behave identically.

Note

Relative paths inside a configuration file (`data/vasp`, `models/`) are resolved against the **working directory**, not the installation. Run these commands from the project root, or give absolute paths.

If a command is not found after an upgrade, re-run `pip install -e .`: entry points are registered when the package is installed, not when it is imported.

1.4 Python version

Note

Poraquê requires **Python 3.11 or newer**. Every module is verified against the 3.11 grammar. There is deliberately no upper bound, so newer interpreters are supported.

1.5 Checking the installation

```
pytest
```

```
python -m poraque.ml.device
python -c "from poraque.ml.device import available_devices;
    ↪ print(available_devices())"
```

Without `-check` this reports and exits 0 whatever it finds, which is what a verification step wants: a CPU-only machine is a working installation, not a failure. `-check` is the queue guard of Section 1.6, and it refuses a CPU deliberately.

The device list is ordered by preference, so the first entry is what `device: auto` will select — `cuda`, then `mps`, then `cpu`. `pytest -m "not gpu"` skips the tests that need an accelerator, rather than reporting them as passes on a machine that has none.

1.6 NVIDIA GPUs

`pyproject.toml` asks for `torch>=2.0` and deliberately says nothing about CUDA: which build is right depends on the machine, and pinning one would break Apple Silicon, which wants the newest torch. But the choice has to be made somewhere, because **the wrong one does not raise — it gives a silent CPU run**.

Two things have to line up: **the driver**, which must be new enough for the wheel's CUDA runtime (`nvidia-smi` prints the highest CUDA version it supports, top right), and **the GPU architecture**, which must appear in `torch.cuda.get_arch_list()`.

GPU	capability	wheel
P100	sm_60	`cu118` / `cu121`
V100	**sm_70**	**`cu126`** (CUDA 13 dropped Volta)
T4, RTX 20xx	sm_75	`cu126` or newer
A100	sm_80	any wheel ≥ `cu118`
RTX 30xx	sm_86	any wheel ≥ `cu118`
H100	sm_90	`cu121` or newer
B200, RTX 50xx	sm_100 / sm_120	`cu128` or newer

```
pip install torch --index-url https://download.pytorch.org/whl/cu126
pip install poraque
```

The second row is not hypothetical. `pip install poraque` into a clean environment on Santos Dumont brought `torch 2.13.0+cu130`, which failed twice over: the driver (560.35.03, CUDA 12.6) was too old for a `cu130` runtime, and `sm_70` is not in that build's `arch_list` at all. Even with a newer driver the V100 would have had no kernel to run — *no kernel image is available for execution on the device*, at the first forward pass, after the queue time had been spent.

1.6.1 Check before submitting to a queue

```
python -m poraque.ml.device --check --device cuda || exit 1
```

It prints the torch build, its CUDA runtime, where it was imported from, the architectures it carries kernels for, and one line per GPU saying whether this build can use it — then exits non-zero if the requested device is not usable. Three seconds, against a slot in the GPU queue.

`-device` defaults to `auto`, and `-check` refuses a CPU under `auto` too — a guard that cannot fail is not a guard, and this one could not: it resolved to the CPU and exited 0 on any machine with

a working CPU, which is every machine. Naming the device the scheduler was actually asked for is still the clearer form, because it is the one that also catches `cuda:2` on a node where the job was given one GPU.

The same check belongs in the configuration, where it stops the run rather than the script:

```
training:
  device: cuda
  strict_device: true    # abort instead of falling back to the CPU
```

> **Note**
> Without it, a job that cannot reach its GPU takes its place in the queue, warns into a log nobody reads until afterwards, and trains on the CPU *inside* the GPU allocation until the wall clock ends. `strict_device` is off by default because a workstation wants the fallback — a config written on a machine with a GPU should still run on a laptop — and should be on in every batch job.

1.6.2 `~/.local` outranks the active environment

A `torch` installed in `~/.local/lib/pythonX.Y/site-packages` **wins** over the one in an active conda environment or venv. On a shared cluster that is easy to acquire by accident and hard to notice: the environment a job activated is not necessarily the one it ran.

```
export PYTHONNOUSERSITE=1
```

Poraquê's start-up banner prints torch's version, its CUDA build *and its install directory* for exactly this reason, so a mismatch appears in the run's own log instead of becoming a four-hour CPU run.

1.6.3 Several GPUs, under Slurm

Poraquê trains data-parallel over NCCL when the launcher describes a group, and on one device when it does not. It cannot invent ranks: **the launcher decides the topology and `training.distributed` decides only whether to believe it.** The shipped submission script is `scripts/slurm/poraque_ddp.sbatch`:

```
sbatch scripts/slurm/poraque_ddp.sbatch configs/train.yaml
```

The line that matters is `-ntasks-per-node`, which must equal the number of GPUs:

```
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=4      # one task per GPU. Not optional.
#SBATCH --gres=gpu:4

srun poraque-train --config configs/train.yaml --device cuda \
                   --strict-device --distributed auto
```

Requesting four GPUs and launching *one* task is the failure to watch for. It is not an error — it is a perfectly good single-GPU run — but it leaves `SLURM_NTASKS` at 1 and looks, from inside the process, exactly like the single-GPU run somebody asked for. The run log therefore prints the Slurm variables it saw:

```
  ranks  : nccl rank 0/4 local_rank 0 on sdumont1234:24601 (launched by slurm)
             SLURM_JOB_ID = 4601
```

```
              SLURM_NTASKS = 4
            effective batch = 10 x 4 ranks = 40
```

`MASTER_ADDR` is the first host of `SLURM_STEP_NODELIST` and `MASTER_PORT` is derived from `SLURM_JOB_ID`, so nothing is hard-coded and two jobs sharing a node cannot collide on a port. `torchrun` is understood as well, and is consulted *after* Slurm: a `torchrun` launch inside an allocation carries both sets of variables and its own describe the group it actually formed.

Three things to know before reading a scaling number. The **effective batch size is `batch_size` times the world size**, so a four-rank run is not the same optimiser as the one-rank run it is compared against. **Rank 0 alone writes** the checkpoint, the metrics, the figures and the PDF, and alone prints. And **`num_workers` gets worse, not better**: each rank is already a process with a field cache of its own. One rank per GPU, cache on, no workers — see `data.cache_in_memory` (section 9.5.1), which is worth more here than a second GPU would be, and section 9.8.9 for the two keys themselves.

To bisect a result, run the same allocation single-GPU with `-distributed off` and compare `seconds_per_epoch` in the metrics JSON.

> **Note**
> NCCL only, and no `DataParallel` fallback. Gloo would work on CPU and would be a way to test the plumbing, but it would also let a misconfigured job train distributed across 96 CPU cores at a fraction of one GPU's speed — the same silent waste `strict_device` exists to prevent. Single-process multi-GPU is slower than one GPU for a model of a few megabytes and changes the effective batch size silently. Without CUDA the group is refused with a warning and the run continues on one device.

1.6.4 HPC best practices: build the cache on a CPU node first

A training run starts by building its dataset cache — reading every density, computing the external potential, spectrally downsampling, writing the result under `data.cache` — and none of that touches a GPU. On a cold parallel filesystem it is minutes to hours, and inside a GPU allocation it is minutes to hours of idle accelerators. So split the run into two jobs: a CPU-only job that builds the cache and exits, and the multi-GPU job, released when the first succeeds, with the *same* configuration file.

```
# 1. a CPU-only job: build the cache and exit
cache=$(sbatch --parsable scripts/slurm/poraque_cache.sbatch configs/train.yaml)

# 2. the multi-GPU job, released when the first succeeds; same config
sbatch --dependency=afterok:${cache} scripts/slurm/poraque_ddp.sbatch
    ↪ configs/train.yaml
```

The first script runs `poraque-train -config configs/train.yaml -cache-only`, which builds the cache exactly as a training run would and then stops — before a model, an optimiser, a process group or a training loop exists — with exit status 0 and a summary:

```
==============================================================================
CACHE BUILT -- no training was run (--cache-only)
==============================================================================
  cache       : data/cache/res48_potcar
  storage     : files (one CHGCAR-format text file per field)
  materials   : 97 cached  (ext2chg: 97, chg2tau: 97)
  grid shapes : 24x24x48        11 structures
```

```
                 40x28x48        10 structures
                 40x40x48         9 structures
                 48x48x48        64 structures
                 ...
elements     : Pt
on disk      : 458.9 MiB
decoded      : ~134.2 MiB in RAM per task (data.cache_in_memory: True, budget 4.0
  ↪ GiB)
```

Three things about the mode are worth knowing. **It forces the CPU and forms no group**, whatever the config says: `training.device: cuda`, `strict_device: true` and `distributed: auto` can stay in the file for the GPU job, and the CPU job overrides all three for its own process while touching nothing on disk but the cache. **The GPU job rebuilds nothing** as long as its config keeps the cache fingerprint — `data.resolution`, `data.storage`, `data.compression`, `data.spin`, `data.potcar_dir` and the data paths — as it was; its cache table then reads `cached` on every row and training starts at once. Change one of those keys and the GPU job builds a *different* cache, on the GPU node, which is what the split exists to avoid. And **read the `decoded` line before requesting memory**: it is the size of the fields `data.cache_in_memory` holds in RAM (section 9.5.1), and the number `auto` compares against its budget; a GPU job whose request is below it either declines the cache, at ten times the epoch time, or is killed.

One task, not four: the builder is single-process, and several tasks would each build the whole cache. Re-running `-cache-only` on a cache that already exists is cheap — present materials are left alone — so an interrupted CPU job is resubmitted, not restarted.

1.7 Faster CPU inference

Poraquê ships a small C kernel for the spectral contraction — the one part of a Fourier layer that PyTorch runs poorly at batch 1, which is the shape every prediction has. It is **optional**: without it everything works and falls back to `torch.einsum`.

There is nothing to configure. The kernel is compiled on first use — about half a second, once — and cached in `~/.cache/poraque`. To do that compile now, and check it:

```
python -m poraque.ml.backend --benchmark
```

```
C spectral backend: loaded from ...poraque_spectral_89feecc6.dylib (pthreads)
agreement with torch.einsum: 2.76e-07 relative (float32 rounding is ~1e-7)

one contraction (batch 1, width 32, modes 12^3):
  torch.einsum (4 threads)     4.528 ms
  C, serial                    0.606 ms     7.5x
  C, 4 pthreads                0.200 ms    22.6x
```

All it needs is a C compiler: `xcode-select -install` on macOS, or `build-essential` on Debian/Ubuntu. Add `-rebuild` after editing the kernel.

Whole-model inference	2–3.4× faster at 24^3–32^3, tapering to ∼1× at 96^3 where the FFTs dominate
Training	unaffected: the kernel records nothing on the autograd tape, so it is used only under `torch.no_grad()`
Accuracy	identical to float32 rounding ($\sim 10^{-7}$ relative), checked on every call path by the test-suite
To disable	`PORAQUE_C_BACKEND=0`

`poraque-inference` prints which path it used, so a run that silently fell back is visible rather than merely slow.

1.8 Optional components

Component	Needed for	Install
CUDA build of PyTorch	NVIDIA GPUs	§1.6
`pdflatex` / `latexmk`	automatic PDF reports	TeX Live or MacTeX

Apple Silicon needs nothing extra: the Metal backend ships with the standard PyTorch wheel and is selected automatically.

CHAPTER 2

Preparing data

2.1 Directory layout

One directory per material:

```
data/vasp/
    struct_000/   POSCAR  INCAR  POTCAR  CHGCAR  TAUCAR
    struct_001/   POSCAR  INCAR  POTCAR  CHGCAR  TAUCAR
    ...
```

The prefix is configurable (`data.pattern`); anything matching it and containing the required files is picked up.

> **Implementation**
> **Any standard VASP output works.** Poraquê computes the external ionic potential natively from `POSCAR`, `INCAR` and `POTCAR` using the tabulated pseudopotential, matching a reference potential to a relative 5×10^{-5}. Only `CHGCAR` and `TAUCAR` are needed as targets.

2.2 Grids may differ between materials

They usually do: `ENCUT` and the cell shape fix `NGXF, NGYF, NGZF`, so two structures of the same compound can land on different meshes. This is handled — batches are grouped by grid shape and one model serves them all.

Fields are reduced to a common working resolution (`data.resolution`, the longest axis) by Fourier truncation, which is the exact band-limited projection for a plane-wave field: periodicity and the electron count survive to machine precision.

2.3 Checking your data

The sharpest check on a `CHGCAR` is its integral, which must come back as the exact electron count $\sum_a Z_a^{\mathrm{val}}$:

```
from poraque.fields import ChargeDensity, FieldGrid

path = "data/vasp/struct_000/CHGCAR"
rho = ChargeDensity.read(path, grid=FieldGrid.from_file(path))
print(rho.integrate())   # 297.0 for 27 Pt atoms at ZVAL = 11
```

A result that misses the integer points at a truncated or misread file long before any model is trained.

`poraque-train` does not repeat this check, but it does stop with an error on a missing `CHGCAR`, and it reports how many points a field drives negative when it is band-limited to `data.resolution` — Gibbs ringing around the core peaks, which is an artefact of the truncation rather than a fault in the data.

A missing `TAUCAR` is *not* an error. It simply means the material cannot serve `chg2tau`; it remains a perfectly good `ext2chg` training example, the cache logs `[no TAUCAR]` beside it, and a run asking for `task: all` trains the task the data can support and reports the other as skipped.

2.4 Downloading from the Materials Project

Installing Poraquê registers `poraque-mp`, which turns a *chemical space* — a set of elements — into a local dataset of charge densities. Every material whose composition uses only those elements is fetched, across all stoichiometries and structures.

Size it before committing to the transfer. Over a chemical space the estimate is exact rather than modelled: charge densities are objects in a public S3 bucket, so their sizes are read with `HEAD` requests that move no payload.

```
poraque-mp --elements Pt Pd Ni --estimate
```

```
==================================================================
  CHGCAR ESTIMATE - Pt-Pd-Ni space
==================================================================
  Materials in space           : 27
  FILES TO DOWNLOAD            : 27
  TOTAL DOWNLOAD               : 193.3 MB
  Storage if kept gzipped      : 169.9 MB
  ...
  27 file(s) to download, 193.3 MB total (169.9 MB on disk gzipped).
  Dry run: nothing downloaded, nothing written.
```

`-estimate` is a **pure dry run**: it reports to the console and leaves no file behind — no `summary.csv`, no `summary.json`, no `chgcar_estimate.csv`, not even a directory. Asking how big something would be should not create anything.

Then download. `-output` (equivalently `-outdir`) defaults to the **current directory**, so a command that writes hundreds of megabytes puts them where you ran it rather than somewhere it chose:

```
poraque-mp --elements Pt Pd Ni --output data/MP --max-size-mb 20
```

A download is **one directory per material, named by its id** — the shape of a VASP run, and the shape `data.data_paths` already reads:

```
data/MP/
    summary.csv  manifest.json  manifest.csv
    structures/mp-124.cif
    mp-124/  CHGCAR.gz  INCAR  KPOINTS  POSCAR  mp.json
    mp-81/   CHGCAR.gz  INCAR  KPOINTS  POSCAR  mp.json
    mp-126/  fields.h5  ...                      # with --hdf5
```

The three VASP inputs reconstruct the calculation, so a material directory is one VASP can be pointed at. `INCAR` and `KPOINTS` come from the task record the density belongs to; MP's standard static set fills in when a record cannot be read, and the file says which of the two it is. The `POSCAR` comes from the **density's own header** — that is the geometry it was computed at, and taking it from anywhere else is how a directory ends up with a structure and a density that disagree. `-no-vasp-inputs` skips all three.

> **Implementation**
> **None of them is training data.** The loader records the density and nothing else, and the external potential stays *computed* — exactly as it must be, since at inference time there is no DFT run to read one from. The `mp.json` beside them is what keeps the directory from being taken for a VASP run now that it looks like one; deleting it changes which source reads the data.

Useful filters: `-band-gap MIN MAX` (`0 0` selects metals), `-crystal-system`, `-stable-only`, `-num-sites MIN MAX`, and `-max-sites` / `-max-size-mb` / `-limit` to cap the download. The size cap has **two spellings and they are the same cap**: `-max-size-gb 0.02` is `-max-size-mb 20`, a decimal factor of 1000 apart, matching every size the report prints. A single `CHGCAR` is an MB-scale object and a whole download is a GB-scale one, so the unit that reads naturally depends on which of the two you are thinking about; passing both is refused rather than resolved. The run is resumable and one failure never aborts the batch; `manifest.csv` records what landed, what was cached and what failed.

The API key is read from `$MP_API_KEY`, then a local `.env`, then `~/.env`, so it never has to appear in a config file or a shell history.

> **Implementation**
> **Leave the files gzipped.** Poraquê reads `.gz`, `.bz2`, `.xz` and `.zip` volumetric files in place, streaming them a line at a time. Expanding them buys nothing and costs roughly three times the storage.

2.5 The whole database, not a chemical space

`-all` asks the other question: every material the index says has a charge density, whatever it is made of. It is a single server-side query — `has_props=["charge_density"]` on the summary endpoint — rather than one query per chemical system, and rather than pulling every summary document in the database and discarding the ones without a density.

```
poraque-mp --all --estimate --sample 20   # what would 20 of them cost?
poraque-mp --all --sample 20 --output data/MP    # fetch exactly those 20
```

> **Caveat**
> **It has to be written out.** Omitting `-elements` does not mean *everything*: a command line naming neither it nor `-all` is refused. That flag was required for most of this tool's life, and reading its absence as the whole database would turn a forgotten argument into a multi-terabyte transfer.

Every filter from the previous section still applies, so `-all` is also how a *subset* of the whole archive is chosen — not 150 000 materials but the ones small enough to be worth training on:

```
poraque-mp --all --num-sites 1 12 --max-size-mb 20 \
    --output data/MP --hdf5 --compression gzip
```

-hdf5 keeps everything, and VASP still cannot read it

`-hdf5` writes each density as a chunked, compressed field store rather than a `CHGCAR`. It is a re-encoding, not a conversion — pymatgen already holds $\rho\Omega$, which is the convention the store keeps — and it carries **everything the text file does**: the magnetisation block of a spin-polarised density, and the PAW augmentation records, the one-centre terms `ICHARG = 1` will not restart without.

> **Caveat**
>
> **A store written before 2026-09-01 has neither.** It kept the total block alone and wrote no records, so one download produced two different datasets: `data.spin: auto` resolves against what is on disk, and the same materials trained a two-channel operator as `CHGCAR`s and a one-channel one as stores. Re-fetch the material if you need what the old store dropped.

What no store can do is be opened by VASP, so a run needs the density written back out:

```
poraque-vasp chgcar data/MP/mp-124/fields.h5 --output CHGCAR

# or straight into a deck, which converts rather than copies
poraque-vasp energy data/MP/mp-124/fields.h5 --like <run> --copy-density
```

Every mode of `poraque-vasp` takes a store where it takes a `CHGCAR`; address one field as `fields.h5::CHGCAR` when the store holds several. A text source is copied byte for byte rather than parsed and re-emitted — re-rendering a file that was already correct can only lose digits or shift a column, and the density block is a column-positional Fortran read.

A sample is a selection, not just a sizing trick

`-sample N` draws `N` materials at random and **those become the working set**. With `-estimate` it sizes exactly those `N`; without `-estimate` it downloads them. Everything the run writes into `-output` is about that subset — `summary.csv` and the CIFs included — so a sampled directory never carries an index of a set it does not hold.

> **Caveat**
>
> **Before 2026-09-01 `-sample` reached the dry run alone.** Used without `-estimate` it was silently ignored, and the command fetched the whole matching set — with `-all`, the entire database.

Sizing changes with it. Over a chemical space every object is `HEAD`ed and the total is exact; over the database that is tens of thousands of requests for a number nobody needs to four digits. A sampled estimate measures its `N` exactly — so `TOTAL DOWNLOAD` is a measurement of what a fetch would actually transfer, not an extrapolation — and *also* projects to the whole set, as the sample mean times the number advertised, on its own line and labelled as a projection. Both are printed because they answer different questions, *what will this fetch* and *what would all of it cost*, and a report stating one of them gets read as the other.

`-seed` decides *which* subset, and that is what makes a sampled download resumable: the same seed selects the same materials, so an interrupted run resumes onto the same subset rather than fetching `N` more beside it. Sizes are also strongly right-tailed, so two draws of twenty can differ

by tens of percent, and a projection that moved every time it was asked could not be planned against.

2.6 What an MP download does and does not contain

It contains one compressed `CHGCAR` per material directory and nothing else — no `POSCAR`, `INCAR`, `POTCAR`, `OUTCAR` or `TAUCAR`. (That emptiness is also what tells the directory apart from a run whose inputs are there to build V_{ext} from exactly.) Three consequences, and Poraquê handles the first two for you:

The structure is recoverable. A `CHGCAR` carries its own `POSCAR` in its first lines, so the geometry comes from the density itself.

The valence charges are recoverable. A pseudopotential `CHGCAR` integrates to its cell's valence electron count, which is one linear equation per material in the per-element Z^{val}. A chemical space of any breadth over-determines them. On the Ag–Pt–Pt set this returns 11, 11 and 10 — the `POTCAR` values, read off the data, from three small files.

τ is not. No public archive publishes a kinetic energy density, so `chg2tau` cannot be trained on MP data at all. Use `task: ext2chg`.

2.7 The external potential without a POTCAR

The one thing the density cannot supply is the *pseudopotentials*, and those are what the exact external potential is built from. Since V_{ext} is the model's *input*, this is the single most consequential setting for a run on archive data.

Point `data.potcar_dir` at a `POTCAR` library — one subdirectory per species, as VASP distributes them:

```
data:
  data_paths:
    - data/MP
  potcar_dir: /opt/vasp/potpaw_PBE     # <dir>/Ag/POTCAR, <dir>/Pt/POTCAR, ...
  resolution: 32
```

`.gz` and `.Z` entries are read directly, and a flat `POTCAR.Ag` / `Ag.POTCAR` layout is recognised too. A run that has its own `POTCAR` ignores the library entirely — it is a fallback, not an override, and it also rescues local runs whose `POTCAR` was stripped for licensing.

Pseudopotentials	V_{ext}	residual vs. a reference `EXTCAR`
available	tabulated local pseudopotential	2×10^{-5} relative L^2
none	Gaussian pseudo-ion model	order 10^{-1}

Measured on the Ag–Pt–Pt set, the two constructions differ *from each other* by 0.38 relative L^2. They are different fields, not different roundings of one; the Technical Guide shows why no analytic form factor can close that gap.

> **Caveat**
>
> **Use the library that generated the data.** The Materials Project uses the VASP PBE set (`PAW_PBE`). A `PAW_LDA` library would build a potential the densities were never computed from — worse than the Gaussian fallback, because it looks exact.

Missing or unreadable entries are not fatal. Each such species warns once and falls back to the Gaussian model for the structures containing it, so a library covering four elements of five still buys the exact potential for everything that does not involve the fifth. The run log names the construction per source.

2.7.1 What the cache records, and `strict_potcar`

The cache *directory name* carries `potcar` when a library was configured, and until 26.9.3 that was the whole of the record — which is to say it recorded what was *asked for* and nothing about what was *got*. The fingerprint had the same gap.

It cost a measurement. A control run on Santos Dumont landed on a node where the library's filesystem was not mounted, warned six times on stderr, trained against analytic pseudo-ion potentials, and wrote a `cache_fingerprint.json` identical to one built from real pseudopotentials — in a directory called `res32_potcar`. Every later run reused it silently, on mounted nodes too, and the warning never came back.

The fingerprint now carries `potcar_source`, per element:

```
{ "potcar_dir": "/prj/.../POTCARs",
  "potcar_source": {"Ag": "gaussian", "Pt": "library"} }
```

so a cache built during an outage no longer matches a run that can read the library — it rebuilds rather than being reused — and the resolved mode is in the run log beside the rest of the cache line. It is `null` when no library is configured, so a purely Gaussian dataset's fingerprint is unchanged.

`data.strict_potcar:` `true` turns the fallback into an error naming the elements the library did not serve. Off by default, and meant for queues: the failure shape is `training.strict_device`'s — a job that holds its allocation and quietly computes something other than what was configured.

Without a library the map learned is *model potential* → *DFT density*: well-posed and self-consistent, since inference builds the very same potential, but not VASP's V_{ext}. Say so wherever such a model's numbers are quoted.

2.8 Training on several sources at once

`data.data_paths` is a list of directories, and **every entry has the same shape**: subdirectories, one per material, each holding that material's volumetric files. Nothing in the config says which is which, because on disk they are the same kind of thing:

```
data:
  data_paths:
    - data/vasp/structures  # structure_0000/, structure_0001/, ...
    - data/MP               # mp-124/, mp-81/, ...
    - data/cache/res32      # a cache from an earlier run
  potcar_dir: /opt/vasp/potpaw_PBE
```

What differs is the *content* of a material's directory — inputs beside the density mean V_{ext} is built from them, a density alone means it is built from the structure the `CHGCAR` carries in its own header — and content is read, not declared.

Everything found is pooled into one training set, and identifiers that collide between directories are prefixed with their own so no material is silently dropped. `TAUCAR` is optional **per material**: one run may have written one while its neighbour did not, and the one that did still reaches `chg2tau` while both reach `ext2chg`.

> **Caveat**
> Without `potcar_dir`, mixing materials that carry their own pseudopotentials with materials that carry only a density mixes *two definitions of* V_{ext} under one name, and the operator spends capacity reconciling them. Poraquê warns when it detects this. Setting `potcar_dir` makes both sources use the tabulated construction, which resolves the warning correctly rather than merely silencing it.

CHAPTER 3

Training

3.1 The short version

```
poraque-train --config configs/train.yaml
```

This trains **one** `ext2chg` model and **one** `chg2tau` model on the combined data of every structure, and writes:

Everything it writes lands in *one directory named after the run*, so a trained model and the evidence for it stay together:

Path	Contents
`models/<name>/<name>.poraque`	both trained operators, in one file
`models/<name>/log/`	training log, metrics, resolved configuration
`models/<name>/plots/`	loss curves, cross-sections, parity plots
`models/<name>/report/`	the typeset PDF report

3.1.1 What the report's performance table says

One row per split — `train`, `validation`, `all` — and five columns:

Column	What it answers
rel. L^2	The headline error, and the one every other number in this manual is quoted in.
rel. H^1	Values *and* gradients. A prediction can match a density pointwise and still be rough, and τ_{vW} depends on $\nabla\rho$ — so a small L^2 beside a large H^1 is a model whose energetics will disappoint.
MAE	The typical voxel, in the field's own units, where a squared error is not readable as a quantity.
Max. error	The worst voxel. Means and RMS both hide a single catastrophic site, and near a nucleus is exactly where one occurs.
$\lvert\Delta N\rvert$	Conservation. For `ext2chg` this is the electron-count error in electrons, and it is the metric a pointwise loss controls worst.

Implementation

The `train` and `validation` rows read against each other *are* the generalisation gap. That

is the single most useful thing the table says, and it is why the splits are rows.

Note

The rel. H^1 here is the textbook relative Sobolev norm, which has no free parameter — **not** the H^1 objective `loss: relative_h1` minimises, whose value depends on `sobolev_weight` and so cannot be compared between two runs. Every column is computed without consulting the loss, for exactly that reason.

For a two-channel model every number is the **density** channel; the magnetisation is measured separately, as `magnetisation_relative_l2` in the metrics JSON, and the report says so in its caveats.

The table lists *splits*, not structures: it used to print one row per material, so a 115-material run made six pages of numbers nobody reads individually. The per-structure figures — and `mse`, `rmse`, `r2`, `nrmse_range` and `jsd`, which are not typeset — are all in `log/<name>.json`, which is machine-readable and has no margins.

`<name>` is `task.name` from the configuration, and it is the only thing you set: two runs with different names cannot overwrite each other, and there are no output paths to keep in step by hand. A trained model is not one file — it is weights, the numbers that say how good they are, the figures behind those numbers, and the configuration that produced them — so they arrive and leave together.

3.1.2 What the parity figure shows

The parity plot is drawn over **every structure** of each split, training and validation side by side, each panel titled with how many structures and voxels it holds and labelled with its relative L^2, R^2, MAE and RMSE. A held-out density visibly wider about the identity line is the generalisation gap, read off the figure instead of inferred from two numbers. Under `-kfold` the one panel is every structure as its own fold held it out.

Every structure cannot mean every voxel: a 97-cell set at 48^3 is eleven million pairs. Each structure contributes a random sample of 10 000 voxels (all of them when its grid is smaller), folded into a sparse, fine histogram the moment it is drawn and then discarded, so memory follows the spread of the values rather than the number of structures. The draw is seeded from `training.seed`, so a rerun draws the same voxels. The four numbers on each panel come from running sums over the sampled voxels, so they are exact for that sample — but they pool the split into one set, and are not the table's mean of per-structure errors.

3.2 One model, all structures

A single set of weights is fitted across every training structure at once, and batches mix materials — never one model per material.

Note

There is **one** training protocol and **one** variation on it. By default a fifth of the structures (`valid_fraction`) is held back for validation, so the printed score is genuinely held out and `early_stopping` has something to watch. Nothing else needs selecting.

Caveat

Set `valid_fraction: 0` and nothing is held out, so the metrics become a **training fit**: they show the model can represent the data, not how it will do on a new material. On the reference data the training fit is about 4× better than the cross-validated score, which is the size of the mistake if the two are confused.
That setting is still the right one for the artefact you ship, since it uses all the data — just quote a cross-validated number alongside it.

3.3 Measuring generalisation

Splitting is always at the *structure* level.

3.3.1 The default split

```
poraque-train --config configs/train.yaml \
       --valid-fraction 0.2
```

Structures are shuffled with `seed` and that share is kept back.

3.3.2 K-fold cross-validation

This is the only variation on the protocol; `valid_fraction` is ignored, since the folds define the splits.

```
poraque-train --config configs/train.yaml \
       --kfold --k-folds 5
```

Each fold trains a fresh model on $K-1$ groups and scores it on the held-out group. A consolidated PDF reports the mean and standard deviation of each metric across folds.

Implementation

`k_folds` is capped at the number of structures; setting it equal to that count is leave-one-out. Whole materials are held out — never a subset of voxels from a material that also appears in training, which would score the model on data it had effectively already seen.

3.4 Reading the output

Metric	Read it as
relative L^2	fractional error; 0.03 means 3 %
R^2	1 is perfect; *negative* means worse than the mean
MAE, RMSE	absolute error, in the field's own units
integral error	how well the electron count or T_s is reproduced

For `chg2tau` the log also prints the classical orbital-free functionals (Thomas–Fermi, von Weizsäcker) evaluated on the same input. Beating those is the bar — a learned functional that does not is not adding anything.

3.5 Reference numbers

Seventeen platinum supercells — ten 27-atom and seven 32-atom, four grid shapes — at 32^3 working resolution, 300 epochs with early stopping, a single 80/20 split at `seed=42`:

Model	held out (3 structures)	all structures (training fit)
`ext2chg`	0.0379 ± 0.0027	0.0209
`chg2tau`	0.0511 ± 0.0031	0.0140

Quote the held-out column. The gap to the training fit is the memorisation margin — roughly two- to four-fold here.

Caveat

Read the held-out column knowing what is in it. At `valid_fraction: 0.2` with 17 structures the validation set is three structures, and at `seed=42` all three are 32-atom cells. So these numbers describe the harder cell size only, there is no held-out 27-atom measurement in them, and the $\pm$ is the spread over three structures rather than a cross-validation error bar.

Cross-validation on the earlier 12-structure dataset measured the cell-size effect directly: 0.0189 ± 0.0047 (`ext2chg`) on 27-atom cells against 0.0441 ± 0.0180 on 32-atom ones, so a held-out 32-atom cell was about $2.4\times$ harder. The current single-split figure for that harder subset, 0.0379, is better than the 0.0441 it scored then — which is what having four 32-atom cells in training rather than one should buy.

Expect a similar penalty whenever a new cell size enters the dataset, and use `-kfold` (Section 3.3.2) when you need a figure that covers every structure rather than whichever three the seed selected.

3.6 Options worth knowing

`-pauli-head` For `chg2tau`, predicts $\tau = \tau_{\mathrm{vW}}[\rho] + s\,\mathrm{softplus}(f)$ so the exact bound $\tau \geq \tau_{\mathrm{vW}}$ holds by construction. Improves accuracy and trains slightly faster; recommended.

`-gaussian-blur` Smooths the external potential. Measured to make no meaningful difference at 32^3 and to remove information near the ionic cores; leave it off unless working at higher resolution.

`-device` `auto`, `cuda`, `mps` or `cpu`. An unavailable backend warns and falls back rather than failing.

`-resolution` Working grid. Higher is more faithful and much slower; 32 is a reasonable starting point.

CHAPTER 4

Predicting a new structure

```
poraque-inference new_structure/ \
    --output predictions/new_structure
```

The directory needs only `POSCAR`, `INCAR` and `POTCAR`. The script computes the external potential, predicts the density, then the kinetic energy density, and writes all three in `CHGCAR` format.

4.1 Choosing the grid

Resolved in this order, first hit wins:

1. `-grid NX NY NZ` — explicit;
2. `-like FILE` — adopt an existing volumetric file's grid, which is what you want when comparing against a reference calculation;
3. `-resolution N` — longest axis, preserving the aspect ratio;
4. otherwise `-encut` (default **200 eV**) through the usual `ENCUT`/`PREC` rule.

The default cutoff is Poraquê's own, not the one the reference calculation happened to use: a genuinely new structure has no prior calculation to inherit from, and the same geometry must give the same grid either way. When the two differ the log says so.

> **Implementation**
> 200 eV at `PREC=Normal` lands on 32^3 for the 27-atom reference cells and 28^3 for the 32-atom ones, against a training `resolution` of 32 — so the operator is interpolating rather than extrapolating. Raising the cutoff, or switching to the `Accurate` rule, gives a finer mesh than anything the models have seen.

4.2 Matching a VASP grid

VASP's `PREC` tag sets the multiplier between the wavefunction cutoff and the density FFT grid: `Normal` uses the cheaper 3/2 rule, while `Accurate` and `High` use a factor of 2 so that the density grid is wrap-around free. Two flags expose it.

```
# the Accurate rule at the default cutoff: 42^3 instead of 32^3
poraque-inference struct/ --prec-accurate
```

```
# the same rule at a stated cutoff: 450 eV, Accurate -> 64^3
poraque-inference struct/ --encut 450 --prec-accurate

# write for VASP, from its own INCAR: ENCUT and PREC from the file,
# VASP's grid rule and the PAW records
poraque-inference struct/ --from-incar struct/INCAR
```

Caveat

`-from-incar` takes precedence. When it is given, `-encut` and `-prec-accurate` are ignored entirely, and anything overridden is named in the log. A run described by an input file should reproduce that file's grid; a flag quietly modifying it would make the two disagree while appearing to agree.

`-from-incar` implies `-to-vasp` and `-add-paw`

Both are switched on whether or not they were typed, and the log names the ones it enabled. An INCAR is handed over for one reason — the density is going back into VASP under that file — and each flag, forgotten, fails late and somewhere else: without `-to-vasp` VASP refuses the grid on a restart, without `-add-paw` it restarts from the plane-wave part alone. A `-resolution` beside `-from-incar` is therefore superseded, and the log says so. To size a grid from a cutoff *without* writing for VASP, use `-encut` and `-prec-accurate`.

4.2.1 `-to-vasp`: the grid VASP itself would build

```
poraque-inference struct/ --from-incar struct/INCAR   # implies both flags
poraque-inference struct/ --to-vasp --encut 450       # VASP's grid, no INCAR
```

The sizing above is how a plane-wave code *should* choose a grid. It is not what VASP does, and for a restart that difference matters: the `CHGCAR` declares its own `NGXF NGYF NGZF`, and `ICHARG=1` wants those to be the grid the run itself would build.

VASP derives the density grid in **two stages** (`main.F`):

```
GRID%NGPTAR(1) = XCUTOF*WFACT + 0.5  ! WFACT = 4 for high/accurate/single
CALL FFTCHK(GRID%NGPTAR)             ! rounded here
GRIDC%NGPTAR(1) = GRID%NGPTAR(1)*2   ! then doubled
CALL FFTCHK(GRIDC%NGPTAR)
```

Caveat

The order is the algorithm. For a 27-atom platinum cell at 450 eV the coarse grid is $4 \times 15.25 = 61 \to 64$, so the density grid is 128; computing the density size directly and rounding once gives 64 — a factor of two, and a file VASP would reject.

`-to-vasp` reproduces this exactly. Validated against every reference calculation in the project: **17 of 17**, including the anisotropic $(120, 128, 128)$ and $(108, 108, 112)$ cases.

Implementation

Sizes are rounded by VASP's `FFTCH1` rule — 7-smooth *and* even, so $61 \to 64$ and $109 \to 112$.

Grid selection overall, first match winning: `-grid`, then `-like`, then `-to-vasp`, then `-resolution`, then the cutoff path (`-from-incar`, otherwise `-encut` with `-prec-accurate`).

Caveat

A Fourier neural operator is resolution-*flexible*, not resolution-*indifferent*. Evaluating far from the grid density a model was trained on is extrapolation, and no metric printed here can detect it. The script warns when the requested grid departs markedly from the training resolution; pass `-resolution 32` to evaluate exactly where the models were fitted.

4.3 Comparing against a reference

```
poraque-inference run/ \
    --like run/CHGCAR --compare --plot-dir results/plots/check
```

`-compare` scores the prediction against any reference files present, bringing them onto the prediction grid by spectral truncation. `-plot-dir` adds cross-section and parity figures.

4.4 Restarting VASP from a predicted density

```
poraque-inference struct/ --like struct/CHGCAR --add-paw
```

A `CHGCAR` that VASP will accept for `ICHARG=1` is not just a density on a grid. After the grid block it carries one *augmentation occupancies* record per atom: the one-centre PAW terms, the part of the density inside the augmentation spheres.

Caveat

Those terms are **not representable on the plane-wave grid at all**, so no grid-based model predicts them — ours included. `-add-paw` does not generate them. It copies them verbatim from a reference calculation in the same directory, and the resulting file is a hybrid: interstitial density from the model, core-region occupancies from the reference.

This has a consequence worth stating plainly. `-add-paw` requires a converged `CHGCAR` for the geometry, and if you already have one you do not need a prediction to restart from. The flag earns its place when the reference is a *near-neighbour* — a slightly displaced geometry, a nearby volume — where the on-site occupancies are a good approximation and the interstitial density is what you want the model to supply.

4.4.1 Without a reference calculation

The model carries a fallback. During training the augmentation records are read off the training `CHGCAR`s, averaged per chemical element, and stored in the checkpoint's `paw_profiles`, beside that element's core density:

```
PAW reference: Pt  138 values, averaged over 494 atoms in 17 structure(s)
PAW profile     -> Pt (Z=78): core density + augmentation
```

At inference that table is used whenever the directory has no reference of its own, so a genuinely new structure can still be written as an `ICHARG=1` restart.

Caveat

The averaged table reproduces the true occupancies to about **9 % RMS** on the reference dataset, and the dominant component varies by a factor of two across sites. It is a defensible starting guess, not a converged on-site density — and it covers only the elements the model was trained on. A structure containing anything else gets *no* records rather than a partial block.

A real calculation beside the structure always wins over the table: its records are *that* system's, where the table is an average over others.

4.4.2 And without either: the isolated atom

There is a third source, and `-paw-source atomic` is the only way to reach it. The isolated-atom database that `poraque-atoms` builds carries each free atom's own augmentation record alongside its form factor, so an element the training set never contained still has *something* to write.

It is a last resort, and the gap is not small. Measured on this project's platinum data, a free Pt atom's record is **86.6 % RMS** away from a real bulk Pt site, against **9.9 %** for the training-set average and zero for a reference calculation. A free atom and an atom in a metal have genuinely different on-site occupations; that number is what the difference costs. The three sources rank the same way at every step:

1. a reference calculation beside the structure — exact, because the records are that system's own;
2. an `ext2paw` operator in the bundle (§9.6.3) — this structure's own records, predicted from its potential;
3. the bundle's per-element average — 28 % relative RMS on the platinum set of bulk, slabs and nanoparticles, and only for the elements the model was trained on;
4. the isolated atom — ∼86.6 % RMS, on explicit request only.

`-paw-source auto`, the default, walks that list from the top. Nothing selects the isolated atom silently.

4.4.3 Checks

Three checks run before anything is appended, and each refuses rather than writing a file VASP would reject:

- **No reference.** If nothing in the directory carries augmentation records, the run stops and says why.
- **Wrong atom count.** If the record count does not match the structure, the reference is a different system and its occupancies do not correspond to these positions.
- **Grid mismatch.** The records are on-site and grid-independent, so this is a warning rather than an error — but VASP reads the density on the grid the file declares, and `ICHARG=1` wants that to be the run's own `NGXF`. Use `-like` to match it.

Note

The predicted density integrates to the electron count only approximately (about 1 % out on the current models). VASP renormalises the charge it reads, so this is not fatal for a

> restart, but a density that far from normalised is a poor starting guess and may cost more SCF steps than it saves.

4.5 Sanity checks the script performs

- the predicted electron count against $\sum_a Z_a^{\mathrm{val}}$;
- the number of negative density voxels, if any;
- the bound $\tau \geq \tau_{\mathrm{vW}}[\rho]$ on the chained output, with the minimum margin.

A large electron-count error or a violated bound means the prediction should not be trusted, whatever the field looks like.

4.6 Visualising

All outputs are `CHGCAR`-format. Use `-write-chg` for an additional coarse-formatted `CHG`.

CHAPTER 5

Energies and the ASE calculator

Once ρ and τ have been predicted, the `poraque.physics` module turns them into energies, and the `Poraque` calculator wraps the whole chain behind the ASE calculator protocol.

> **Caveat**
> Read Section 5.5 before comparing any of these numbers to a DFT result. They are **pseudo-valence** energies, and on the current models the error on energy *differences* is larger than the differences themselves.

5.1 The energy expression

$$E = \underbrace{\int \tau \, d^3r}_{T_\mathrm{s}} + \underbrace{\int \rho V_\mathrm{ext} \, d^3r + E_{\alpha Z}}_{E_\mathrm{ext}} + \underbrace{\tfrac{1}{2}\int \rho v_\mathrm{H} \, d^3r}_{E_\mathrm{H}} + E_\mathrm{xc}[\rho] + E_\mathrm{Ewald}$$

Term	How it is obtained
T_s	integral of the predicted `TAUCAR`
E_ext	$\int \rho V_\mathrm{ext}$, with $\mathbf{G} = 0$ excluded
$E_{\alpha Z}$	the finite $\mathbf{G} = 0$ remainder, from the `POTCAR` `PSCORE`
E_H	Poisson solve in reciprocal space, $v_\mathrm{H}(\mathbf{G}{=}0) = 0$
E_xc	PBE by default; LDA available (Section 5.2)
E_Ewald	Ewald sum over the pseudo-ions in a neutralizing background

5.1.1 The $\mathbf{G} = 0$ bookkeeping

Each of the three electrostatic terms diverges at $\mathbf{G} = 0$, and the divergences cancel in a neutral cell. Poraquê follows the standard plane-wave treatment: drop $\mathbf{G} = 0$ from all three, then add back the finite remainder of the electron-ion term,

$$E_{\alpha Z} = \frac{N_\mathrm{elec}}{\Omega} \sum_s N_s \, \mathrm{PSCORE}_s,$$

which is VASP's `alpha Z`. It is of order eV *per atom*, so it is read from the `POTCAR` tables when they are available and reported as `None` when they are not — never silently assumed to be zero.

> **Note**
> `components.missing` lists any term that was skipped, and printing the components emits `incomplete:` when that list is non-empty. A total that is missing a term says so rather

> than looking complete.

5.2 Choosing the exchange-correlation functional

`functional`	Exchange	Correlation
`"pbe"` (**default**)	PBE	PBE
`"lda"`	Dirac	PW92
`"pbe-x"`	PBE	—
`"lda-x"` / `"x-only"`	Dirac	—
`"none"`	—	—

PBE is the default because it matches the reference data: the calculations Poraquê ingests use `PAW_PBE` pseudopotentials with `LEXCH = PE`, so ρ and τ are PBE quantities. Evaluating an LDA E_{xc} on a PBE density is neither a PBE energy nor an LDA one. On the reference Pt supercells the two differ by -0.92 eV/atom (0.65 % of E_{xc}), so the choice is not cosmetic.

Both are validated against the limit that defines them: on a uniform density PBE reduces to LDA *bit-exactly*, since $F_{\mathrm{x}}(0) = 1$ and $H \to 0$. Dirac exchange reproduces $-0.458165293/r_s$ Ha per electron exactly, and PW92 matches the published table to 10^{-6}.

> **Caveat**
> PBE is semilocal and needs $\nabla\rho$. On a *predicted* field that gradient carries the network's noise and the enhancement factor amplifies it; on a band-limited grid it also aliases wherever the density has sharp core peaks. The extra physics is only worth having once the density is good enough for its gradient to mean something — which, on the current models, it is not (Section 5.5).

Set it wherever the energy is computed:

```
EnergyCalculator.from_potential(vext, functional="lda")
Poraque("ext2chg.poraque", "chg2tau.poraque", potcar_dir=..., functional="lda")
```

```
poraque-inference run/ \
    --functional lda
```

5.3 Computing energies from files

```
from poraque.fields import ChargeDensity, ExternalPotential, \
                           FieldGrid, KineticEnergyDensity
from poraque.fields.vasp.potcar import Potcar
from poraque.physics import EnergyCalculator

grid = FieldGrid.from_file("run/CHGCAR")
rho  = ChargeDensity.read("run/CHGCAR", grid=grid)
tau  = KineticEnergyDensity.read("run/TAUCAR", grid=grid)
vext = ExternalPotential.from_calculation("run", grid=grid)
```

```
potcar = Potcar.from_file("run/POTCAR", parse_tables=True)
calc = EnergyCalculator.from_potential(
    vext, pscore={e.element: e.pscore for e in potcar})

print(calc.compute(rho, tau, vext))
```

```
  kinetic      T_s            6207.068300 eV
  external     E_ext        -12634.744762 eV
  alpha Z                     1752.512988 eV
  Hartree      E_H            3230.363304 eV
  xc (pbe)                   -3845.824650 eV
  Ewald                     -25949.345827 eV
  ----------------------------------------
  potential                 -37447.038948 eV
  TOTAL                     -31239.970647 eV
  electrons                    297.000001
```

Implementation

`electrons` is the cheapest available diagnostic. It should equal the total `ZVAL` — 297 for the 27 Pt atoms above — to a few parts in 10^4. A predicted density that has drifted off it invalidates every electrostatic term, since all of them are linear or quadratic in ρ.

Plain arrays are accepted as readily as field objects, so a prediction can be scored without being written to disk first.

5.4 The ASE calculator

`Poraque` behaves like MACE or NequIP at the interface:

```
from ase.build import bulk
from poraque.calculator import Poraque

atoms = bulk("Pt", "fcc", a=3.92, cubic=True)
atoms.calc = Poraque("models/poraque_models.poraque", potcar_dir="POTCARs")

energy = atoms.get_potential_energy()
print(atoms.calc.components)          # the full decomposition
rho = atoms.calc.fields["density"]    # the predicted CHGCAR
```

Each call runs `Atoms` → `Structure` → `FieldGrid` → $V_{\text{ext}} \to \rho \to \tau \to E$. The three fields stay on the calculator afterwards, so a prediction can be written out with `rho.write("CHGCAR")`.

Argument	Meaning
`ext2chg`, `chg2tau`	Checkpoint paths, or loaded `FieldOperators`.
`potcar`	A single `POTCAR` covering the species present.
`potcar_dir`	A `POTCAR` *library*, searched per composition.
`charges`	`{element: Z_val}`, used only without a `POTCAR`.
`resolution`	Longest grid axis; defaults to the checkpoint's training resolution.
`functional`	Exchange-correlation approximation, default `"pbe"`.
`device`	`auto`, `cuda`, `mps` or `cpu`.

5.4.1 Supplying pseudopotentials

`potcar` is right when the composition is fixed. For a calculator that must serve **arbitrary** structures, point `potcar_dir` at a `POTCAR` library: the entries for whatever elements an `Atoms` contains are assembled on demand and cached per composition.

```
atoms.calc = Poraque("models/poraque_models.poraque",
                     potcar_dir="/opt/vasp/potpaw_PBE")
```

Recognised layouts, in preference order — each accepting a `.gz` or `.Z` suffix:

1. `<dir>/Pt/POTCAR` — what VASP ships;
2. `<dir>/Pt_pv/POTCAR` — a variant, used only if it is the *only* candidate, with a warning;
3. `<dir>/POTCAR.Pt` or `<dir>/Pt.POTCAR` — flat.

Note

If several variants match — `Fe_pv` and `Fe_sv`, say — the lookup **raises** rather than picking one. They differ in `ZVAL` and therefore in every energy, so the choice is the user's; name the one you want with an explicit `potcar`.

Caveat

With no `POTCAR` at all the external potential falls back to the *Gaussian* pseudo-ion model, which differs from the tabulated potential by a relative L^2 of about 0.13 — against 2×10^{-5} for the tabulated one. The operators were trained on tabulated potentials, so the Gaussian model feeds them an input outside their training distribution. The calculator warns once and proceeds; treat the result as a smoke test, not a prediction.

5.4.2 Forces and stress

`get_forces()` returns the analytic **Hellmann–Feynman** force: the electron–ion term $-\int \rho\, \partial V_{\text{ext}}/\partial \mathbf{R}$ evaluated in reciprocal space from the same `POTCAR` form factor the potential was built from, plus the analytic Ewald force. Both are differentiated in closed form rather than by finite differences — on a fixed grid those would be dominated by the grid's own discontinuity as atoms cross voxel boundaries.

The implementation is exact: each term agrees with a central finite difference of the energy it differentiates to 10^{-7} eV/Å.

Caveat

The *physics* is incomplete for PAW datasets. Hellmann–Feynman is the whole force only for a *local* pseudopotential; a PAW calculation adds a projector force and a one-centre force, and neither is recoverable from ρ, τ and V_{loc} on a grid. Measured against VASP on platinum, using VASP's own density: MAE 0.83 eV/Å against forces of 1.66 eV/Å. The magnitude is right, the direction is not, so **geometry optimisation and molecular dynamics remain unavailable**.
The cancellation is why it is delicate: the electron–ion and ion–ion terms are each ≈ 100 eV/Å and cancel to ≈ 0.5 eV/Å, so a relative error in ρ arrives in the force amplified roughly 200-fold.

`get_stress()` still raises: the stress needs the energy's response to a strain, which deforms the cell *and* the grid the fields live on.

`implemented_properties` is `["energy", "free_energy", "forces"]`. The first two are the same number: no electronic smearing enters this pipeline, and ASE optimizers request `free_energy` by name.

5.4.3 Cohesive energies

A total energy means nothing on its own. It is referenced to whatever the pseudopotential generator chose, and Poraquê's totals additionally carry an offset of order 10^3 eV per atom from the absent PAW one-centre terms. The **cohesive energy** removes the arbitrary part,

$$\Delta E = E_{\text{total}} - E_{\text{ref}}, \qquad E_{\text{ref}} = \sum_i E_{\text{iso}}(Z_i),$$

with E_{iso} the energy of one isolated atom of that species. What survives is the energy released on assembling the solid from free atoms — the bonding, and nothing else.

Supply a directory holding one subdirectory per element, each an ordinary single-point calculation of one atom in a large box:

```
data/vasp/ref/
    Pt/     POSCAR POTCAR OSZICAR OUTCAR CHGCAR TAUCAR
    N/      ...
```

```
atoms.calc = Poraque("models/poraque_models.poraque",
                     potcar_dir="POTCARs",
                     references="data/vasp/ref")

atoms.get_potential_energy()                    # total, as always
atoms.calc.get_cohesive_energy()                # dE, in eV
atoms.calc.get_cohesive_energy(per_atom=True)   # eV/atom
```

5.4.3.1 Which reference to subtract

`ReferenceEnergies.from_directory` takes a `method`, and the choice matters more than anything else here.

`method`	E_{iso} from	Pt cohesive energy
`"poraque"` (default)	Poraquê's own expression	$-$**1.9** eV/atom
`"code"`	VASP `OUTCAR`/`OSZICAR`	-1157 eV/atom

A cohesive energy is meaningful only when the two energies being subtracted carry the *same* systematic error. Subtracting VASP's atomic energy from Poraquê's total leaves the PAW offset entirely intact; subtracting Poraquê's own atomic energy cancels it, because the same terms are missing from both sides. Hence the default. Use `"code"` when the reference calculations are what you want to compare *against*.

> **Note**
> Referencing changes neither the forces nor energy differences at fixed composition, and both statements are exact identities rather than approximations. E_{ref} depends on composition and not on coordinates, so $\nabla_{\mathbf{R}} E_{\mathrm{ref}} = 0$; and two structures with the same formula share E_{ref}, which cancels bit-for-bit in $\Delta E_1 - \Delta E_2$. Its value is in comparisons *across* compositions — binding energies, cohesive energies per atom, formation energies — where the offset does not cancel and without which the comparison is undefined.

5.4.4 The Hartree potential

v_{H} is **solved, not predicted**. Poisson's equation relates it to ρ exactly, and on a periodic plane-wave grid the reciprocal-space form is the solution rather than an approximation to it:

$$\nabla^2 v_{\mathrm{H}} = -4\pi e^2 \rho \quad \Longleftrightarrow \quad v_{\mathrm{H}}(\mathbf{G}) = \frac{4\pi e^2 \rho(\mathbf{G})}{G^2}, \qquad v_{\mathrm{H}}(\mathbf{G} = 0) = 0.$$

Two FFTs, exact for any band-limited density. A third learned operator would be strictly worse: it would inject error into a quantity that has none, and would not be guaranteed to satisfy the equation that defines it.

```
potential = atoms.calc.get_hartree_potential()
potential.write("LOCPOT")

# v_H + V_ext, the total local potential a plain VASP LOCPOT holds
total = atoms.calc.get_hartree_potential(with_external=True)
```

The $\mathbf{G} = 0$ term is set to zero — the same neutralising-background convention V_{ext} uses (§5.1.1), which is what makes the two potentials addable and what makes E_{H} of a uniform density come out at exactly zero rather than infinite.

Written to `LOCPOT` the field is stored **unscaled**: unlike `CHGCAR`, which holds $\rho\Omega$, a `LOCPOT` holds the potential itself in eV. From the command line:

```
poraque-inference structure/ --output predictions/ --write-locpot
poraque-inference structure/ --output predictions/ --write-locpot --locpot-total
```

The field accessors are symmetric — `get_external_potential()`, `get_charge_density()`, `get_kinetic_energy_density()` and `get_hartree_potential()` — and all four return fields on one shared grid.

5.5 What the number is not

It is not a DFT total energy. `CHGCAR` holds the valence *pseudo* density and `TAUCAR` the valence pseudo kinetic energy density, so the PAW one-centre terms are absent entirely. For

the 27-atom Pt cell above the result is ≈ -31215 eV against a VASP `TOTEN` of order -100 eV. Nothing in this module can close that gap.

Energy differences are not usable yet either. Measured on the current seventeen-structure Pt dataset, within each composition group:

Source of the fields	MAE on ΔE	vs. signal
VASP's own ρ and τ (method ceiling)	0.18–0.24 eV/atom	1.5–2×
The shipped operators	0.87–1.93 eV/atom	7–15×
True spread being resolved	*0.125 eV/atom*	—

The error **exceeds the signal** in both rows and the correlation with the true ordering is consistent with zero, so the predicted ranking of structures carries no information.

The first row is the important one. It is measured with the model removed from the problem entirely, so it is not a training failure — it is the neglected PAW one-centre and non-local energy, which is *not* a per-atom constant. The electrostatics themselves are correct: $E_{\alpha Z}$, Ewald and E_{H} reproduce VASP's `PSCENC`, `TEWEN` and `DENC` to better than 0.01 eV in a difference, and the Ewald sum reproduces exact Madelung constants to 10^{-6}.

The underlying difficulty is scale. The total is a cancellation of terms of order 10^4 eV down to a result of order 10^0 eV, so resolving ΔE to 10 meV/atom needs a relative accuracy of about 10^{-6} in the fields. The operators deliver 10^{-2}.

> **Note**
> The grid is *not* the limiting factor. On reference fields the energy difference changes by less than 0.2 eV between `resolution: 32` and the native 128^3 mesh, and T_{s} and the electron count are preserved exactly — spectral resampling is an exact band-limited projection.

Charge normalisation helps, and is on by default. The shipped `ext2chg` operator predicts densities whose electron count drifts by up to 1.7 % — nearly five electrons out of 297. Every electrostatic term is at least linear in ρ and E_{H} is quadratic, so that drift alone moved totals by tens of eV, differently for each structure, and therefore did not cancel in a difference. Rescaling to the count the pseudopotentials fix *halved* the end-to-end error on ΔE. It is a repair rather than a cure: a density whose integral is wrong by 1.7 % has the wrong shape too, and rescaling does not fix the shape. Disable it with `normalize_density=False` only to diagnose.

CHAPTER 6

Charges and population analysis

Once a density has been predicted, two questions follow: how much charge is there, and where does it sit. This chapter covers the conservation check that must pass before either answer means anything, and the three partitionings that turn a grid into per-atom numbers.

Nothing here is learned. Every quantity is a deterministic functional of a density that has already been predicted, so **the error in a partial charge is inherited from the density and is never smaller than it**.

6.1 Charge conservation

A Kohn–Sham density integrates to the valence count the pseudopotentials fix. That number is an *input* to a DFT calculation rather than an output of one, so it is exactly known — and it is the only property of a predicted density that can be checked without a reference calculation.

```
from poraque.analysis import verify_total_charge

check = verify_total_charge(rho.data, rho.grid.cell, expected_electrons=297.0)
print(check)
# charge check OK: 297.000001 electrons against an expected 297.000000
#                  (+1.918e-09 relative, tolerance 0.001)
```

The integral is the voxel sum times the voxel volume,

$$\int \rho \, d^3r = \sum_{ijk} \rho_{ijk} \, \mathrm{d}V, \qquad \mathrm{d}V = \frac{\Omega}{N_x N_y N_z},$$

which is *exact* for a band-limited field on a uniform periodic mesh: the rectangle and trapezoid rules coincide under periodic boundary conditions, so there is no quadrature error to argue about, only the field itself.

The default tolerance of 10^{-3} is not arbitrary. The electrostatic terms are of order 10^4 eV, so a relative drift of 10^{-3} already moves a total energy by roughly 10 eV — more than any energy difference worth computing.

Caveat

The shipped `ext2chg` operator predicts densities whose electron count drifts by up to 1.7 % — nearly five electrons out of 297. Left uncorrected this moves totals by tens of eV, differently for each structure, so it does not cancel in a difference.

6.1.1 Automatic normalisation

The calculator rescales the density to the exact count by default:

```
atoms.calc = Poraque("models/poraque_models.poraque", potcar_dir="POTCARs",
                     normalize_density=True)   # the default

atoms.calc.verify_charge()          # passes by construction
atoms.calc.raw_electron_drift       # what the operator actually did
```

With normalisation on, the check passes by construction, so the informative number afterwards is `raw_electron_drift` — the drift of the raw prediction, recorded before the repair. Set `normalize_density=False` to measure the operator rather than the repair.

Rescaling is a repair and not a cure: a density whose integral is wrong by 1.7 % has the wrong shape too, and a single global factor does not fix a shape.

6.2 Partial charges

All three schemes share one form,

$$q_A = Z_A^{\mathrm{val}} - \int w_A(\mathbf{r})\,\rho(\mathbf{r})\,d^3r, \qquad \sum_A w_A(\mathbf{r}) = 1,$$

and differ only in the weight w_A. Because the partitions are exhaustive, **all three conserve charge exactly**: the populations sum to $\int \rho$ whatever the weights are.

Method	$w_A(\mathbf{r})$	Cost	Character
`voronoi`	1 for the nearest atom	fast	purely geometric
`hirshfeld`	$\rho_A^{\mathrm{at}} / \sum_B \rho_B^{\mathrm{at}}$	fast	needs a promolecule
`bader`	1 inside A's zero-flux basin	moderate	intrinsic to ρ

6.2.1 From the ASE calculator

```
atoms.calc = Poraque("models/poraque_models.poraque", potcar_dir="POTCARs",
                     references="data/vasp/ref")

atoms.get_charges()                       # standard ASE, Bader by default
atoms.calc.get_charges(method="hirshfeld")
atoms.calc.get_charges(method="voronoi")

print(atoms.calc.charge_analysis)         # the full decomposition
```

`get_charges` returns net charges in units of $+e$, positive for electron-deficient, and is compatible with the standard `atoms.get_charges()` interface. The full analysis — populations, the valence subtracted, which promolecule or Bader backend was used — is left on `charge_analysis`.

6.2.2 Voronoi

Each voxel goes wholly to its nearest atom under the minimum-image convention. In a non-orthogonal cell the surrounding images are searched rather than trusting a wrap of the fractional coordinates — in a skewed lattice the nearest image can be a diagonal neighbour.

Voxels exactly equidistant from several atoms are **shared equally** rather than awarded to the lowest atom index. That is rare on a real density and common on a symmetric cell sampled by a commensurate grid, which is exactly where an index-order tie-break would bias a result that ought to come out symmetric.

Purely geometric: the density enters only through the integral, never through where the boundary is drawn. For atoms of unequal size that is a poor chemical partition, which is why it is a baseline rather than a default.

6.2.3 Hirshfeld

Weights by the free atoms, so the partition is smooth and chemically sensible — but it is *defined by its reference*, and a poor promolecule gives poor charges with no other symptom.

```
partial_charges(rho, method="hirshfeld", valence={"Pt": 11.0},
                references="data/vasp/ref")
```

Each `<references>/<Element>/CHGCAR` is spherically averaged into $\rho^{\mathrm{at}}(r)$. Where no reference exists the fallback is an exponential free atom, $\rho(r) = \frac{N}{8\pi a^3} e^{-r/a}$, normalised to the valence count with the decay length set from the covalent radius through $\langle r \rangle = 3a$. Which was used for each element is reported in `details["promolecule"]` — check it, because the fallback is crude.

> **Note**
> Hirshfeld charges are small in magnitude compared with other schemes. For an elemental solid they come out near zero by construction, since the promolecule is built from identical free atoms; that makes a useful sanity check on the reference handling.

6.2.4 Bader (QTAIM)

The only scheme whose definition is intrinsic to the density: basins are bounded by surfaces of zero flux in $\nabla\rho$. Two backends are provided.

```
partial_charges(rho, method="bader", backend="native")    # pure NumPy
partial_charges(rho, method="bader", backend="external")  # Henkelman `bader`
partial_charges(rho, method="bader", backend="auto")      # external if present
```

Native is a vectorised on-grid steepest ascent: every voxel points at whichever of its 26 neighbours gives the largest density increase *per unit distance*, and those pointers are followed to a local maximum by repeated pointer doubling. The distance weighting matters — without it the diagonal neighbours, $\sqrt{3}$ further away, win too often and the basins acquire a staircase bias along the cell diagonals.

External writes a temporary `CHGCAR`, runs the Henkelman group's `bader` program and parses `ACF.dat`. `backend="external"` raises if the program is absent rather than falling back silently: the two backends are not identical, and a result reported as "Bader (Henkelman)" should not quietly have been something else.

A density may have more maxima than atoms — spurious ones from grid noise, or genuine non-nuclear attractors in a metal. Each maximum is assigned to its nearest atom, so the mapping is many-to-one, and `details["maxima"]` reports how many were found.

6.3 What these charges are not

Caveat

All three partition the *pseudo* valence density. The PAW core is absent, so these are not all-electron populations: a Bader volume here is not the all-electron Bader volume, and the charges are systematically compressed toward zero. This matters most for Bader, whose basin boundaries are set by the density's topology near the nuclei — exactly where the pseudisation is largest.

For a proper Bader analysis VASP's own guidance is to sum `AECCAR0` and `AECCAR2`; Poraquê does not predict those.

Treat the numbers as comparative across a series, not as absolute.

6.4 In the reports

Passing a `PartialCharges` to the report builder typesets the per-atom table, the method and the caveat above into the PDF:

```
from poraque.vis import ModelReport

ModelReport("reports").build(task="ext2chg", per_material=metrics,
                             charges=analysis)
```

CHAPTER 7

Fine-tuning

A model fitted across a broad dataset is a good *starting point* for a specific material family, and a poor substitute for one. Fine-tuning continues that fit on the smaller set at a much lower learning rate: what the base model learned about the general map is kept, and only the specialisation is learned.

```
# 1. the base model, trained across everything you have
poraque-train --config configs/train.yaml

# 2. specialise it on one family
poraque-train --config configs/train.yaml \
    --fine-tune --pretrained models/poraque_models.poraque \
    --root data/oxides --checkpoint-dir models/oxides
```

7.1 What comes from the checkpoint

Two things are taken from the base model rather than from this run's configuration, and both matter.

The architecture, inferred from the stored tensors. A `width` or `modes` left over in the config that disagreed with the weights could only load mismatched tensors, so the tensors are the authority — the `model` section is ignored for the *shape* of a fine-tuned network.

The normalizations. The datasets are re-pointed at the checkpoint's transforms, discarding the ones just fitted to the new data.

> **Caveat**
> Refitting the normalizations would rescale the network's inputs out from under weights trained against the old scale, and the model would spend the fine-tune relearning the scale instead of the chemistry. It also means the new data must fall in a comparable range: fine-tuning on a family whose densities are an order of magnitude away is transfer learning in name only.

7.2 Freezing the lifting path

The lifting map embeds the input field into the network's channel space before any operator acts on it. It is the most general part of the model, so it is the part least in need of specialising — and freezing it removes parameters a small dataset could otherwise overfit. The cell encoder is frozen with it: that embeds the lattice, a property of the input rather than of the mapping.

The **projection head stays trainable**. It decodes to physical units, which is precisely what differs between material families.

```
fine_tuning:
  freeze_lifting_layers: true
```

The run reports what was actually frozen, so it never has to be inferred:

```
      fine-tuning from : models/poraque_models.poraque
      architecture     : inferred from the checkpoint (4,202,001 parameters)
      transforms       : taken from the checkpoint
      frozen           : lifting path, 1,392 parameters;
                         4,200,609 remain trainable
      learning rate    : 1e-05 (fine-tuning; base training uses 0.002)
```

Implementation

Frozen parameters are kept out of the optimiser entirely, not merely denied a gradient. AdamW's decoupled weight decay applies without one, so a frozen weight left in the optimiser would shrink towards zero every step — quietly undoing the pre-training the freeze was meant to preserve.

7.3 Configuration

Key	Default	Meaning
`enable`	`false`	Start from `pretrained_checkpoint` instead of a fresh initialisation.
`pretrained_checkpoint`	`models/poraque_models.poraque`	Base bundle to adapt.
`learning_rate`	`1e-05`	Replaces `training.learning_rate` for this run.
`freeze_lifting_layers`	`false`	Hold the input lifting path fixed. Ignored under `use_lora`.
`use_lora`	`false`	Freeze everything and learn a low-rank correction; see below.
`lora_rank`	`8`	r; cost and capacity are both linear in it.
`lora_alpha`	`16.0`	The update is scaled by `lora_alpha / lora_rank`.
`lora_dropout`	`0.0`	Dropout on the adapter's input.

7.4 LoRA — adapting a model that does not fit

`use_lora: true` freezes every trained weight and learns a rank-r correction beside the model's **dense** $(1 \times 1 \times 1)$ **lifting and projection** convolutions:

$$W' = W + \frac{\alpha}{r} BA, \qquad A \in \mathbb{R}^{r \times d_{\text{in}}}, \quad B \in \mathbb{R}^{d_{\text{out}} \times r}.$$

On the shipped model that is **1 320 trainable parameters against 3 152 353 frozen — 0.042 %**, so the optimiser state and the saved file shrink by three orders of magnitude and the fine-tune fits where a full one does not:

```
      LoRA              : 3 adapted layer(s), rank 8, alpha 16
      trainable         : 1,320 of 3,153,673 parameters (0.042%); the rest is frozen
      NOTE: the checkpoint will hold the ADAPTER only and name this base;
            it cannot be loaded without poraque_models.poraque.
```

B starts at **zero**, so $BA = 0$ and the adapted model *is* the base model until the first optimiser step — a fine-tune that began anywhere else would already have discarded some of what it set out to adapt.

> **Caveat**
> **The spectral weights are not adapted.** They hold ∼99.8 % of the parameters and look like the obvious target, but a rank-r factorisation of a 5-index complex kernel is a *choice of which axes to pair* rather than one decomposition, and adapting them would also stop being cheap.
> What LoRA says here is precise: *keep the learned operator, re-fit how fields enter and leave it.* A family whose operator genuinely differs from the base model's — not merely its input and output scales — is out of its reach, and the honest answer there is a full fine-tune. It buys memory, not generality.

> **Note**
> **A LoRA checkpoint is not self-contained.** It holds the adapter and records where its base lives, which is the whole economy of the method — the frozen tensors are already on disk in the model being adapted. Move or delete that base and the fine-tune cannot be loaded; the error says so and names the file. `freeze_lifting_layers` is ignored under LoRA, which freezes everything anyway, and the run says so rather than appearing to apply both.

The fine-tuning learning rate is much smaller by design: the base rate would walk the weights away from the solution being adapted before a small dataset could constrain them.

7.5 Output, and the `.poraque` format

Poraquê writes its models as `.poraque`, after the project rather than after the architecture inside it:

File	Contents
`models/poraque_models.poraque`	The universal model: both operators in one file.
`models/poraque_finetuned.poraque`	A fine-tune, specialised to one family.

The container is a `torch.save` dictionary of exactly three entries:

```
checkpoint = {
    "model_state_dict": {"ext2chg": ..., "chg2tau": ...},
    "config": {...},                    # the resolved training config
    "paw_profiles": {78: {...}, ...},   # PAW data per atomic number
}
```

`model_state_dict` holds, per task, everything that rebuilds the operator: the weights, the architecture record, both normalisations and the δ-density baseline — weights alone would decode to a field in the wrong units. `config` is the resolved configuration as a plain dictionary. `paw_profiles` holds, per element and keyed by Z, the radial core charge density read from the POTCAR that built V_{ext} (`r` in Å, `core_density` and `pseudo_core_density` in e/Å^3, `core_electrons`) and the `augmentation` occupancies that `-add-paw` writes.

The POTCAR stores the core as $r^2\rho_{00}(r)$ on a mesh in Å, and states neither. Both were pinned by measurement: $\sqrt{4\pi}\int$ of the table is exactly $Z - Z_{\text{val}}$ for every dataset checked — 68.0000 for Pt — and the pseudized core rejoins the all-electron one at `RPACOR` converted to Å.

> **Caveat**
> The core density is derived from the POTCAR, which is licensed with VASP. A checkpoint carrying it carries pseudopotential data; mind that before sharing one outside a group that holds the licence.

The extension is a label, not a different serialisation. Nothing inspects it when loading, so a checkpoint under any name loads fine.

A fine-tune is **never** written over the base model. The two coexist in the same directory, and a run that would genuinely overwrite its own base is refused — before training starts, not after.

The PDF report leads its Configuration section with the fine-tuning facts and carries a caveat at the top, so a reader cannot mistake a specialised model's scores for a general one's. The checkpoint's `config` records `fine_tuning.pretrained_checkpoint`, the learning rate and whether the lifting layers were frozen, so a checkpoint can always be traced back to its parent.

> **Note**
> The extension has been changed twice: `.pth` first became `.pfno`, and `.pfno` became `.poraque` on 2 September 2026. Neither change touched the container. The container itself changed on 16 September 2026, from a `poraque-bundle-1` payload with one top-level entry per task and a metadata block to the three keys above, and **a file in the old layout is not read**: it raises, names the layout, and asks for a retrain. If the `.poraque` file is absent and a `.pfno`, `.pth` or `.pt` of the same stem is present, that one is used and the substitution is announced — an existing trained model should not become invisible because a default filename changed. The search runs the other way too, which matters for a LoRA checkpoint: it stores the path of the base it adapts, and that path may name a file since renamed. Rename the file to silence the notice.

CHAPTER 8

Symbolic distillation

A Fourier Neural Operator reproduces $\rho \mapsto \tau$ to a few percent and explains nothing. Symbolic distillation searches short algebraic expressions for one that reproduces the same mapping — trading accuracy for a formula you can read, publish, and check against known physics.

> **Note**
> Off by default. The one optional package is SymPy, used to render the result and check its limits; the search itself is `poraque.ml.gp`, genetic programming in NumPy with SciPy fitting the constants, and both are already required. Until 2026-09-03 this extra carried PySR and its **Julia** toolchain, fetched on first use — which on a supercomputer means a download from a compute node, a writable depot on a filesystem that may be purged between jobs, and a second language runtime inside an MPI job.

```
pip install -e ".[symbolic]"
poraque-train --config configs/train.yaml --task chg2tau --symbolic
```

Without the extra, a run with distillation enabled reports the missing dependency and keeps its trained model rather than failing.

8.1 What it can and cannot find

The features are evaluated *at a point*, so the search space is exactly the semi-local functionals. Whatever the operator learned that is **non-local** cannot be expressed in that space at all, and appears as irreducible residual.

That makes a poor fit informative rather than disappointing: it puts a number on how much of the learned map is not semi-local. A near-perfect fit would say the operator found nothing a GGA-level functional could not have.

> **Note**
> Only `chg2tau` is attempted. `ext2chg` is the Hohenberg–Kohn map, whose whole content is non-local — a semi-local expression for it would be meaningless rather than merely inaccurate.

8.2 The physical variables

Raw ρ is not enough to build a robust functional. The engine receives the density together with its **dimensionless reduced derivatives**, which is the form a semi-local kinetic functional is actually written in:

$$k_F = (3\pi^2\rho)^{1/3}, \qquad p = \frac{|\nabla\rho|}{2k_F\rho}, \qquad q = \frac{\nabla^2\rho}{4k_F^2\rho}. \tag{8.1}$$

Symbol	In the equation	Meaning
ρ	`rho`	Electron density, atomic units (e/a_0^3).
p	`p`	Reduced gradient — the GGA variable.
q	`q`	Reduced Laplacian — the meta-GGA variable.

Those names are passed to the engine verbatim, so they are the names that appear in the result.

Why the reduced forms rather than $|\nabla\rho|$ and $\nabla^2\rho$ directly: they are dimensionless and invariant under the coordinate scaling $\rho_\lambda(\mathbf{r}) = \lambda^3\rho(\lambda\mathbf{r})$ that fixes T_s, so a functional expressed in them holds at every density scale instead of having to be rediscovered at each. Raw derivatives make every coefficient carry units, which a genetic search spends its budget approximating.

8.2.1 Derivatives are spectral

$\nabla \to i\mathbf{G}$ and $\nabla^2 \to -|\mathbf{G}|^2$, evaluated by FFT. This is *exact* for the band-limited periodic fields a plane-wave grid carries; a finite-difference stencil would introduce an error that the search would then model as physics.

8.2.2 Feature schemes

Two independent knobs: `features` picks the **input variables**, `template` picks how the **target is factorised**.

`features`	Variables given to the engine
`gga` (default)	`rho`, `p`, `q`
`reduced`	`p`, `q`
`raw`	`rho`, `grad_rho`, `lap_rho` (dimensional)

`template`	Fitted target	Reported formula
`none` (default)	τ	$\tau = f(\dots)$
`pauli`	$F = \dfrac{\tau - \tau_{\mathrm{vW}}}{\tau_{\mathrm{TF}}}$	$\tau = \tau_{\mathrm{vW}} + \tau_{\mathrm{TF}}\, f(\dots)$
`thomas_fermi`	$F = \tau/\tau_{\mathrm{TF}}$	$\tau = C_{\mathrm{TF}}\rho^{5/3} f(\dots)$

`pauli` is the physically right choice. $\tau_{\mathrm{vW}} = |\nabla\rho|^2/8\rho$ is known in closed form and $\tau - \tau_{\mathrm{vW}} \geq 0$ by Hoffmann–Ostenhof, so subtracting it leaves exactly the quantity a kinetic functional actually has to model — the Pauli term. Leaving it in means fitting something mostly already known, and near the von Weizsäcker limit it means fitting a near-cancellation between two large numbers.

A template **gives away the part of the physics that is already known**. Thomas–Fermi supplies the density scaling exactly, so the search stops spending its budget rediscovering $\rho^{5/3}$ and works on what is actually unknown — and every constant it must find becomes order unity. It is also the form the literature writes kinetic functionals in, so the answer is directly comparable: Thomas–Fermi is $F = 1$, von Weizsäcker is $F = 5p^2/3$.

The discovered expression is multiplied back into the template before it is reported, so the console,

the JSON and the PDF all show the complete physical formula rather than the factor that was fitted.

> **Note**
> `features: enhancement` is kept as an alias for `features: reduced` plus `template: thomas_fermi`, which is exactly what it used to mean.

8.3 Operator constraints

```
constraints:
  "^": [-1, 1]
```

Per-operator limits on argument complexity, passed straight to the engine. The default holds the **exponent** of `^` to a single node while leaving the base unconstrained (`-1`).

Unconstrained exponents are the main source of nonsense from a power operator: a fractional power of a negative quantity leaves the reals, and an exponent that is itself a subtree is unreadable and almost never physical. Real functionals have simple exponents — 5/3, 4/3, 2.

8.4 Figures

Three are written after a search, all embedded in the report's Symbolic Distillation section:

File	Shows
`<task>_symbolic_pareto.png`	the front, with the lowest-loss candidate and the knee marked
`<task>_symbolic_parity.png`	the **lowest-loss** expression against the DFT reference
`<task>_symbolic_parity_knee.png`	the **knee** expression, on identical voxels

The two parity plots are laid *side by side* in the report, because the question they answer is a comparison: does the shorter formula give anything away? Where the clouds are indistinguishable, the extra nodes bought nothing.

Both are evaluated on the **held-out** structures and compared against the DFT reference — not against whatever was fitted, since with `target: model` the two are different things. With no validation split they fall back to the fitted voxels, and the caption says so.

Read them beside the operator's own parity plot: the gap between them is what the closed form gives up. A formula that tracks the identity line through the bulk and bends away at the high-density end is the usual picture — semi-local features cannot resolve the core peaks.

8.5 Regularization in vacuum

p and q both divide by ρ, which decays exponentially into vacuum. Two things happen, and both are needed:

1. **Every denominator is clamped** at `epsilon`, so k_F, p and q stay finite even where the density underflows.

2. **Voxels with $\rho \leq$ `epsilon` are dropped.** In vacuum p and q are ratios of two vanishing numbers — noise with a plausible magnitude, which corrupts a fit far more quietly than a `NaN` would. Vacuum carries no information about the functional, so nothing is lost by removing it.

`epsilon` defaults to `1e-8`, in atomic units (e/a_0^3). The mask also removes the slightly negative voxels that band-limiting leaves around the core peaks (Gibbs ringing), where a fractional power of ρ would be complex.

> **Implementation**
> For a bulk crystal nothing is dropped — the minimum density is far above the threshold. It matters for slabs, molecules and anything with real vacuum.

8.6 Configuration

```
symbolic:
  enable: false
  target: model          # model | reference
  features: gga          # gga | reduced | raw
  template: none         # none | pauli | thomas_fermi
  epsilon: 1.0e-08       # vacuum threshold, e/a0^3
  constraints:           # per-operator argument-complexity limits
    "^": [-1, 1]
  unary_operations: [exp, log, sqrt, abs]
  binary_operations: ['+', '-', '*', /, ^]
  iterations: 40
  population_size: 33
  populations: 15
  max_depth: 10
  max_size: 30
  parsimony: 0.0032
  n_samples: 4000
  seed: 0
```

`target` selects what is fitted. `model` distils the trained operator's own predictions — what the network learned, faithful to it including its errors. `reference` fits the DFT data directly, which is plain symbolic regression against ground truth and answers a different question.

`unary_operations` and `binary_operations` are passed to the engine untouched. Keep the alphabet small: the search space grows combinatorially, and an operator that cannot appear in the answer only dilutes the population. `/` and `log` are the usual sources of singularities on a density approaching zero.

`n_samples` caps the rows handed to the search. A single 32^3 structure is already 32 768 voxels and the cost is linear in them.

8.7 Output

The expression, its accuracy/complexity front and the fit quality are printed to the terminal, written to the run's JSON summary, and typeset into the PDF report as display mathematics with ρ, p and q rendered properly.

```
expression   : F = (q * 2.454276) + 0.9163395
variables    : p, q  [dimensionless (F = tau / tau_TF)]
complexity   : 5 nodes
fit          : R2 0.9487   relative L2 0.0736
```

Note

Read the front, not just the winner. A single expression hides the trade that produced it; the front shows what each extra node bought.

8.8 The Pareto knee

The front states a trade and refuses to resolve it. The **knee** is where resolving it costs least: the candidate nearest the ideal corner $(0, 0)$ once both axes are rescaled to $[0, 1]$ across the front,

$$d_i = \sqrt{\hat{c}_i^2 + \hat{\mathcal{L}}_i^2}, \qquad \hat{x}_i = \frac{x_i - \min x}{\max x - \min x}.$$

Both rescalings are necessary: a node count and a loss have no common unit, so a distance between them means nothing until they are made comparable.

The loss is compared on a **logarithmic** scale. A front spans orders of magnitude, and on a linear scale every candidate but the most accurate collapses onto $\hat{\mathcal{L}} \approx 1$ — the knee would then be the shortest expression whatever it cost.

The run prints both candidates, and the report marks the knee with • in the front table alongside its distance d:

```
pareto knee  : complexity 7 nodes, loss 0.021  (lowest loss: 18 nodes, 0.0188)
```

Caveat

A knee is a heuristic, not a theorem. With one candidate it is that candidate; where the front is a straight line every point is equidistant. It answers *which expression is worth its length*, not *which expression is right* — that is what the asymptotic limits of the next section are for.

8.9 Rounding

Constants are rounded to **three decimal places** wherever an expression is displayed. A search returns them at full float precision — `0.33333334326744079589843750` is a real example — and three of those in one formula overrun the width of a PDF page.

Three places is also what the numbers are worth: the search is stochastic and its constants move in the third place between seeds, so printing twenty digits states a precision the method does not have.

8.10 Physical asymptotic compliance

A symbolic fit is a numerical statement until it is checked against physics it was never shown. Every candidate on the front is tested against the two limits that pin a kinetic functional, both written for the enhancement factor $F = \tau/\tau_{\mathrm{TF}}$: Both are statements about the *Pauli* enhancement factor $F = (\tau - \tau_{\mathrm{vW}})/\tau_{\mathrm{TF}}$:

$$\underbrace{F(0,0) = 1}_{\text{Thomas–Fermi}}, \qquad \underbrace{F \to 0 \text{ as } p \to \infty}_{\text{von Weizsäcker}}. \tag{8.2}$$

The first is the uniform electron gas: no density variation, τ_{vW} vanishes and the functional must collapse to Thomas–Fermi exactly. The second is the rapidly varying limit — a single orbital, an exponential tail — where $\tau \to \tau_{\mathrm{vW}}$, so the Pauli term added to it must vanish.

Every template is converted to this one convention before checking, using $\tau_{\mathrm{vW}}/\tau_{\mathrm{TF}} = 5p^2/3$ exactly, so a `thomas_fermi` or `none` fit is held to the same physical standard.

The check is analytic, through `sympy.limit` with the symbols bound at parse time, falling back to a converged numerical probe when SymPy cannot resolve a deeply nested expression. Which route was taken is recorded: an analytic limit is a proof, a numerical one is evidence.

> **Caveat**
>
> **Neither textbook functional passes both.** As a Pauli factor Thomas–Fermi is $F = 1 - 5p^2/3$ — correct at the origin, divergent at infinity; von Weizsäcker is $F = 0$ — correct at infinity, wrong at the origin. Failing one limit is not damning on its own. Failing *both* means the expression reproduces the training data without the physics that constrains it outside that range, and it must not be extrapolated.
> A form that passes both, such as $F = e^{-p^2}$ or $F = 1/(1+p^2)$, interpolates between the two regimes — which is what a usable semi-local kinetic functional has to do.

Whether F tends to *any* finite limit is reported apart from whether that limit is zero: a functional settling on a finite non-zero constant stays bounded but never reduces to von Weizsäcker, a repairable failure — unlike one that diverges.

Each candidate carries a `TF/vW` badge in the console, the JSON summary and the PDF report, which gains a dedicated *Physical asymptotic compliance* section:

```
accuracy/complexity front (TF/vW = asymptotic limits satisfied):
    nodes          loss   limits  expression
        1          0.45    TF/--  1.0
        5          0.09    TF/--  1 + 5*p**2/27 + 20*q/9
        9         0.012    TF/vW  1 + 5*p**2/3
2 of 4 candidates satisfy BOTH limits; the simplest is:
    1 + 5*p**2/3
```

The most accurate expression is frequently the least physical, so the compliant candidates are listed separately — a slightly worse expression that obeys both limits is usually the better functional.

> **Note**
>
> The search is stochastic, but a `seed` is now an unconditional promise: it runs in one process, so the same seed gives the same front, and `deterministic` is accepted and ignored

> rather than traded against parallelism. Two *different* seeds will still differ, so treat a single expression as one sample rather than *the* answer.

CHAPTER 9

Configuration reference

A run is defined by `configs/train_config.yaml`: one top-level key and four sections. Generate a copy with every default written out explicitly with

This chapter documents every key that file can contain.

9.1 How a value is resolved

Three sources, highest priority first:

1. an explicit command-line flag,
2. the value in the YAML file,
3. the built-in default.

That ordering is what lets a single committed configuration be swept from the shell without being edited or copied:

```
poraque-train --config configs/train.yaml --epochs 500
```

> **Note**
> Unknown keys are **rejected**, not ignored. A typo such as `learn_rate` would otherwise be dropped silently and the run would quietly use the default — producing results that do not match the file that appears to describe them. The error message names the valid keys for that section.

The *resolved* configuration — defaults, file and command-line overrides merged — is written beside the results as `<json-stem>_config.yaml` (by default `models/<name>/log/<name>_config.yaml`). That is the file worth keeping: it records what actually ran, overrides included.

9.2 Optional features are blocks

Every feature that can be switched off is one key, with its switch **inside** the settings it governs:

```
<group>:
  <feature>:
    enable: <switch>
```

```
    <setting>: <value>
```

There are four: `model.equivariant`, `training.physics_informed`, `symbolic.physics` and `fine_tuning`. A setting inside a block is read **only when `enable` is on**; naming one beside a switch that is off warns rather than acting, and an unknown one raises, naming the block it was in.

> **Note**
> Two of these were a scalar beside a separately named `<feature>_setup` group until 26.9.8, and the two halves drifted in exactly the way that layout invites: `equivariant: false` above a populated `equivariant_setup` reads at a glance as an equivariant run and is not one, and nothing brought the switch and the settings into the same field of view. The old spellings now raise and name their replacement — including the bare `equivariant: true`, which is a *type* error rather than an unknown key and so carries its own message showing the block to write instead.
> `symbolic.enable_symbolic_distillation` became `symbolic.enable` in the same change. It restated its own section in its own name, and at 37 characters was the longest key in the schema — wide enough to set the width of the configuration table in the PDF report.

One `_setup` stays, deliberately: `model.kan_setup`. It is selected by `model.activation`, which is a *choice among five* rather than a switch, so there is no `enable` for it to hold and no boolean it could contradict.

Blocks are validated when the file is **read**, not when the feature is first used. That matters more than it sounds: the model is built after the field cache and the objective after that, so a one-line mistake in either block would otherwise take minutes of downsampling to report.

9.3 Every key is optional

Anything omitted takes its default, so a configuration needs to state only what the run does *differently*. That is what makes a committed file readable as the description of one experiment rather than a dump of every knob: of the eighty settings a full configuration carries, `configs/train.yaml` differs in a handful.

`configs/train_complete_and_commented.yaml` is the reference: it lists the settings worth knowing about, each with the reasoning behind it. Read it there and copy only what you need into your own file. This chapter is the exhaustive version of the same thing.

Eleven configurations ship with the source — three at the top of `configs/` and eight controlled activation comparisons under `configs/kan/`:

File	Purpose
`train.yaml`	The clean starting point. Copy this one.
`train_complete_and_commented.yaml`	The same run, with every choice explained at length — the reference to read, not to copy.
`train_ext2paw.yaml`	The `ext2paw` task on the same data: the pseudo-density and every atom's PAW occupancies.
`kan/compare_kan_*.yaml`	One per learnable activation, base-carrying and pure, identical in everything else — so a difference between two of them is the activation.

> **Note**
> Every one of them is parsed against the current schema by the test suite, and checked for hyphens, camelCase, duplicate keys and retired names. That is not belt-and-braces: the 26.9.8 rename broke all eight `kan/` files and left the ones at the top working, because those were the ones anyone loaded.

9.4 Top level

Key	Default	Meaning
`task`	`all`	Which map to train: `ext2chg`, `chg2tau`, `all` for both in sequence, or `ext2paw` (§9.6.3).

9.5 `data` — where the fields come from

Key	Default	Meaning
`data_paths`	`data/vasp/structures`	The dataset, as a list of directories — see below.
`cache`	`data/cache`	Where downsampled copies are written.
`pattern`	`""`	Prefix filter on the material subdirectories; empty takes all of them.
`format`	`auto`	`vasp`, or `auto` to detect the code from the files present. Those are the only two values.
`resolution`	`32`	Longest grid axis after spectral downsampling.
`potcar_dir`	`null`	`POTCAR` library, used where the data ships no pseudopotentials.
`strict_potcar`	`false`	Refuse the Gaussian fallback when a configured library cannot serve an element (Section 2.7.1). What it *did* serve is in the cache fingerprint either way.
`sigma`	`null`	Gaussian pseudo-ion width in Å, where the Gaussian model is reached.
`gaussian_blur`	`null`	Gaussian blur width in Å applied to the computed potential.
`blur_method`	`spectral`	`spectral` or `ndimage`.
`cache_in_memory`	`auto`	Keep decoded fields in RAM between epochs — see below.
`spin`	`auto`	Train on the magnetisation as a second channel (Section 9.5.5).

9.5.1 cache_in_memory

The largest performance setting in the file, and the only one whose effect is completely invisible from the results.

With it off, **every epoch reopens, decompresses and re-parses every field from disk**. That produces exactly the same numbers as caching does — and takes about ten times as long. Measured on a V100 with 115 structures at 32^3: `__getitem__` was 59 % of the training loop, 78.9 s over 483 calls, while the GPU sat at 2–4 % utilisation waiting on `zlib` and `numpy.array` over a text generator. Turning the cache on took twenty epochs from **168.0 s to 16.3 s**, with the validation error identical to five decimals.

It is not a GPU setting. The parse is the same on Apple Silicon and on the CPU; what an accelerator adds is a fast device left idle by it.

`auto` estimates the decoded size — grid points × itemsize × channels, over the input and target fields — and enables the cache below 4 GiB, logging what it decided and what it costs:

```
  field cache          : in RAM, ~61.3 MiB (data.cache_in_memory: auto)
```

Set it to `false` when the process shares its memory with something the estimate cannot see. What `auto` refuses on its own is the case the old default was right about: 128^3 grids over thousands of materials, where caching silently would turn a slow run into a dead one.

> **Note**
> Parallel loading is **not** the alternative it looks like. See `num_workers` below: added to the cache, workers make training slower.

9.5.2 data_paths

One key names the dataset, and every entry in it has the same shape: a directory of subdirectories, one per material, each holding that material's volumetric files.

```
data:
  data_paths:
    - data/vasp/structures      # structure_0000/, structure_0001/, ...
    - data/MP                   # mp-124/, mp-81/, ...
    - data/cache/res32          # a cache from an earlier run
```

That is what a VASP run tree looks like, what `poraque-mp` writes, and what the cache builder produces — so nothing has to be declared about where a path came from, and no key says so. Every path is pooled into one dataset.

What *differs* is the **content** of a material's directory, and content is read, not configured:

A material's directory holds	Poraquê
inputs beside the density	computes V_{ext} from them — exactly, with a `POTCAR` or `potcar_dir`
a density on its own	computes it from the structure the `CHGCAR` carries in its own header
an `EXTCAR` already	reads it as it stands (a prepared cache)

V_{ext} is **always computed** for the first two, never read from an `EXTCAR` that happens to sit in a run directory: at inference time there is no DFT run to read one from, so the input channel has to be Poraquê's own arithmetic in both places.

`TAUCAR` is **optional, per material**. A directory that has one joins the `chg2tau` set; one that does not still joins `ext2chg` and is simply absent from the other. Nothing is declared for it, and a task with no target anywhere is skipped with a message rather than failing the run — which is what makes `type: all` sensible on a Materials Project download, where no τ exists.

A path holding a **single** material's run directly — its files at the top level rather than one level down — is read as that one material, so a lone calculation needs no wrapper directory.

> **Note**
> Three keys used to answer this one question: `train_paths` (a list), `root` (a single path used when the list was empty) and `source` (which layout each path was). All three were removed on 2026-08-31. The last is the one that mattered: it asked the *config* to declare something the *directory* already answers, which made "a VASP run" and "a Materials Project download" two different things to write down when they are the same thing on disk. A config using any of them now fails and is told what to write instead.

Chapter 2 covers what each kind of directory contains; `potcar_dir` and its Gaussian fallback are in Section 2.7, which is the setting to read before quoting any number from a run on archive data.

9.5.3 `resolution`

Fields are reduced to this many points along their longest axis before training. The reduction is a Fourier truncation — the exact band-limited projection for a plane-wave field — so periodicity is preserved and the electron count is unchanged to machine precision. Interpolation would alias and shift it.

Cost scales as the cube: raising $32 \to 64$ is roughly eight times the memory and time. Grid *shapes* are reduced in proportion, so a $120 \times 128 \times 128$ calculation becomes $30 \times 32 \times 32$ rather than being forced square.

> **Note**
> There is no key for supplying an external potential. `EXTCAR` is **always** computed by `ExternalPotential` from the `POTCAR` tables, which reproduces a reference potential to a relative error of order 10^{-5}. Any `EXTCAR` present in a source directory is ignored — the training input must be exactly what the pipeline produces at inference time, when no such file exists. Standard VASP does not write one anyway.

9.5.4 `gaussian_blur` and `blur_method`

Smooths the computed external potential. The blur is applied *on top of* the tabulated pseudopotential, never in place of it.

> **Caveat**
> `ndimage` uses `scipy.ndimage.gaussian_filter`, which blurs along *grid* axes and is therefore anisotropic in Cartesian space unless the cell is orthogonal — on a face-centred cubic cell the two methods differ by 2–6 %. `spectral` multiplies by $e^{-G^2\sigma^2/2}$ using the true reciprocal metric and stays isotropic on any cell. Prefer the default.

Measured at $\sigma = 0.15$ Å and `resolution: 32`, with one structure held out and everything else fixed, blurring changed the held-out error by 0.7 % and the training time by 0.4 % — neither meaningful, since the grid already band-limits the field. It does remove information near the ionic cores, where the density peaks. Leave it off unless working at higher resolution.

Implementation

The cache directory name encodes these choices, for example `res32_blur0.15spec`, so changing them cannot silently reuse a cache built with different settings.

9.5.5 `spin`

Value	What happens
`auto`	Ask the sources whether their `CHGCAR`s carry a second grid block, and settle the channel count for the whole dataset before the cache is written.
`true`	Demand one: a dataset with no magnetisation **raises**.
`false`	Drop it deliberately, and say so in the log.

This is the switch for spin-polarised training, and there is no other. A `CHGCAR` from an `ISPIN = 2` run stores two grids, the total density ρ and the magnetisation $m = \rho_\uparrow - \rho_\downarrow$; with `spin` resolved on, both reach the cache, `ext2chg` predicts them as channels 0 and 1 of one output, and `chg2tau` reads both. How many channels there are is a property of the data, not of the architecture, which is why the key lives in `data` and why `model` has no second one: two switches could disagree, and one of them would then be quietly ignored. An unpolarised member of a spin set carries $m \equiv 0$, so a mixed archive trains one two-channel operator rather than two datasets. The cache tag says which it holds (`res48_potcar_spin`).

The objective is still one number over (ρ, m) together. The training log says how it divides:

```
        epoch     train loss     charge        mag     val rel L2
    ---------------------------------------------------------------
          3/6        1.23195    0.78913    0.94600        1.17954  *
          6/6        1.16123    0.73227    0.90121        1.11899  *
```

`charge` is the data term with the prediction substituted into the density channel alone, the magnetisation left at its reference; `mag` is the same the other way round. Both keep the objective's norm and **its denominator**, the norm of the whole (ρ, m) target. That choice is forced rather than convenient: a relative error of m on its own divides by m, which is identically zero on every non-magnetic cell, so it would be undefined on precisely the materials most datasets are made of. With a shared denominator the parts of an L^2 objective add in quadrature, data $= \sqrt{\text{charge}^2 + \text{mag}^2}$ per sample, and the columns say where the error lives. An H^1 objective adds its gradient term rather than combining in quadrature, so there the columns rank the channels without summing to anything; the log's legend states which case applies. `train loss` remains the total the optimiser stepped on, physics terms included.

Note

Both parts are in the metrics JSON's `history`, as `train_loss_charge` and `train_loss_magnetisation`, and exist only for a two-channel `ext2chg` target — a one-channel run prints neither column. The magnetisation gets its own parity figure, `<task>_parity_magnetisation`, on linear axes because m changes sign, and its own held-out number, `magnetisation_relative_l2`; the report's performance table stays on the density channel.

9.6 `model` — the operator's shape

Key	Default	Meaning
`width`	`16`	Channel width of the Fourier layers.
`modes`	`8`	Retained Fourier modes per axis.
`n_layers`	`3`	Number of Fourier layers.
`projection_channels`	`64`	Hidden width of the output projection.
`activation`	`silu`	`silu`, `gelu`, `relu`, `tanh` or `kan`.
`projection_activation`	`null`	Read-out nonlinearity; `null` follows `activation`.
`kan_setup`	`null`	The KAN variant and its hyperparameters; read only when `activation: kan`.
`use_coordinates`	`true`	Append three fractional-coordinate input channels.
`cell_conditioning`	`true`	Condition every layer on lattice descriptors (FiLM).
`embedding_dim`	`32`	Width of that cell embedding.
`mode_selection`	`fixed`	`fixed` mode index, or `physical` at constant $G_{\max}$.
`g_max`	`null`	Cutoff wavevector in Å^{-1}; required by `mode_selection: physical`.
`equivariant`	`{enable: false}`	Rotation-equivariant radial kernel — one block, see below.
`paw`	`{enable: false}`	The `ext2paw` operator — one block, see below.
`pauli_residual`	`false`	Structural $\tau \geq \tau_{\mathrm{vW}}$ for `chg2tau`.
`pauli_scale`	`null`	Initial Pauli scale in $\text{eV}/\text{Å}^3$; `null` fits it from the training split.
`learn_pauli_scale`	`true`	Optimise that scale alongside the backbone.
`precision`	`float32`	Dtype the operator computes in; `float64` for checking a result is not a single-precision artefact.

9.6.1 `width`, `modes`, `n_layers`

These three set the capacity. The parameter count is dominated by the spectral weights — $4\,w^2 m^3$ complex numbers per layer for width w and m modes — so `modes` is by far the expensive knob: doubling it multiplies the spectral parameters by eight.

> **Implementation**
> `modes` is a *capacity limit*, not a requirement. On a grid too coarse to supply that many modes, fewer are used automatically, so one configuration serves materials whose grids differ in size.

9.6.2 `equivariant`

One block, following the convention of §9.2:

```
model:
  use_coordinates: false      # required
  equivariant:
```

```
    enable: true
    n_radial: 32
```

Constrains the Fourier multiplier to a **real function of |G| alone**, which is a convolution with a radial kernel and therefore commutes with every rotation. The coefficients being real also makes it commute with inversion, so the operator is equivariant under the full $O(3)$ rather than the $SE(3)$ the construction was asked for.

Key	Default	Meaning
`enable`	`false`	Use the radial kernel in place of the dense one.
`n_radial`	`16`	Radial basis functions — the whole capacity of the kernel.
`g_basis`	`null`	Radius in Å^{-1} the basis spans: `auto`, a number, or `null`.
`spherical_cutoff`	`true`	Mask the retained modes to the inscribed sphere.

`g_basis: auto` — size the basis from the data

The basis spans $[0, g_{\text{basis}}]$; the band a *material* retains is $\min_i 2\pi m_i/L_i$, a property of its own cell. Over 115 Materials Project cells those ran from 1.94 to 20.96 Å^{-1}, so no constant is right for all of them — and the two ways to miss cost very differently.

Missing	What happens	Measured cost
too **narrow**	Modes beyond the basis are clamped onto its outermost function and share one response.	None: 71 % of modes clamped moved the error less than the seed spread.
too **wide**	Radial functions sit at radii no mode reaches.	**14 % to 43 %**, with six of sixteen functions orphaned.

`auto` takes the **median** band the training split retains, resolved before the model is built so the number a checkpoint records is a number. Half the set is then clamped, deliberately: that is the free half. `null` falls back to `g_max` under `mode_selection: physical` and to 8.0 Å^{-1} otherwise — which on this project's own platinum set is 2.3× wider than the band, leaving about 18 of `configs/train.yaml`'s 32 radial functions where no mode exists. That file therefore states `auto`.

Whichever applies, the run says so at startup: what the basis spans, the range and median of the band the training structures retain, and the fraction of retained modes lying beyond it.

Three conditions have to hold together, and each fails silently

The kernel being radial is the one everybody thinks of; the other two were found by measuring.

The retained set must be a ball, not a box — `spherical_cutoff: true`. A radial multiplier over a box of modes is equivariant only under the box's own symmetry group. A cubic cell with equal mode counts hides this completely, because the box *is* invariant under the octahedral group; on a tetragonal cell, switching the cutoff off takes the error from 3×10^{-7} to 5×10^{-2}.

> **`use_coordinates` must be off**, and `enable: true` beside it raises. The three fractional coordinates are not three scalar fields — under a rotation they turn into each other — and being absolute positions they cost translation equivariance too.

The cost is capacity, and it is large. The dense layer holds $4C_{\text{in}}C_{\text{out}}m_1m_2m_3$ complex numbers; this one holds $C_{\text{in}}C_{\text{out}}R$ real ones — 4 096 against 1 048 576 at `width 16`, `modes 8`, `n_radial 16`.

Which does not make `n_radial` the knob to raise *first*, though this guide said so until 2026-09-04. `modes` costs *zero* parameters under equivariance — the coefficient count is C^2R whatever band is retained — so it is a bandwidth knob paid only in arithmetic. On 92 training materials the error rose monotonically with `n_radial` across 8, 16, 32 and 64, while raising `modes` from 4 to 16 improved it 2.7× at *identical* parameter count, for 36 % more time per epoch. On a small training set the radial bank is where the excess capacity accumulates and the bandwidth is free; expect the ordering to invert as the set grows, which is untested.

Whether it pays was measured once, on a V100: 400 epochs on 115 Materials Project structures at 32^3, `width 32 / n_radial 32` reached a validation relative L^2 of 0.047 against 0.173 for a dense model with 28× the parameters — 4.1× better. The 4× *larger* equivariant arm (`width 128, n_radial 64`) was **worse**, at 0.059: capacity is not what buys this, the constraint is. One seed, one dataset, and neither equivariant arm had converged at 400 epochs, so read it as a reason to try the flag rather than as a number.

> **Note**
> An equivariant model is slower per epoch despite having two orders of magnitude fewer parameters, and that is not a defect. The dense layer *stores* one complex number per retained mode per channel pair and reads it; the radial layer stores C^2R real numbers and must **reconstruct** the multiplier at every mode from them — about $R/2$ times the arithmetic at `n_radial: R`. The constraint trades memory for recomputation, so parameters and FLOPs move in opposite directions by construction.

9.6.3 paw

```
task:
  type: all          # the chain, and ext2paw beside it
model:
  use_coordinates: false
  equivariant:
    enable: true     # the whole operator becomes equivariant
  paw:
    enable: true
    width: 64
    modes: 16
    n_layers: 4
```

`ext2paw` maps V_{ext} to two things at once: the pseudo-density on the grid, as `ext2chg` does, and every atom's **augmentation occupancies** $\rho(\ell\ell' LM)$ — the one-centre PAW terms VASP writes after the grid, which with the pseudo-density are what an all-electron density is rebuilt from. They are not a field. There are 138 per atom for `PAW_PBE Pt`, whose projector channels are d, d, s, s, p, p in the POTCAR's order, and one set per density channel, so a spin-polarised dataset keeps the magnetisation's set too.

Key	Default	Meaning
`enable`	`false`	Train `ext2paw`: required by `task.type: ext2paw`, and adds it to an `all` run.
`width`	`64`	The FNO backbone's channels.
`modes`	`16`	Its retained Fourier modes per axis.
`n_layers`	`4`	Its Fourier layers.
`readout_g_max`	`auto`	The band in $Å^{-1}$ read at each atom; `auto` is 95 % of the coarsest training grid's Nyquist frequency.
`occupancy_weight`	`1.0`	The per-atom term against the field objective, and in the validation score.

`ext2paw` is not a link of the chain — `all` still means `ext2chg` then `chg2tau` — and the two switches are checked against each other: `task.type: ext2paw` with `enable: false` raises, and so does `enable: true` beside `ext2chg` or `chg2tau`.

The operator is built as `experiments/paw_occupancies` measured it should be. **An FNO for the grid**, sized by the block; every other `model` setting applies to it unchanged. **A readout at each atom** of the latent field and the predicted density, onto solid-harmonic Gaussians $r^L Y_{LM}(\hat{r})\, e^{-r^2/2s_n^2}$ — the decoder half of a graph neural operator — over one band $|\mathbf{G}| \leq$ `readout_g_max`, resampled to a common spacing so every cell is integrated alike. **An equivariant head per element**: one map per L shared across M, gated by invariants. With `model.equivariant` on, the whole operator rotates its predicted occupancies with the crystal, to 7×10^{-15} in double precision.

> **The band is not optional**
> A readout of the DFT density integrated on each grid as it came scored 3 % on nanoparticles it had seen and 184 % on held-out ones, whose grid spacing differs; over one band it scored 4.9 %. A grid too coarse to supply the band therefore raises, at training and at inference.

The records train with L = 0 centred and every (set, L, pair) block scaled to unit RMS, fitted on the training split and inverted from the checkpoint, and every L block weighs the same in the loss. The log gains a `val occ` column, and the best epoch is chosen on `val rel L2 + occupancy_weight × val occ`. Each element's record layout comes from the POTCAR's `Non local Part`, so a dataset without POTCARs cannot train it; nor, yet, can `-kfold`, fine-tuning or an f-channel element, all refused before the cache. A cache built before 16 September 2026 dropped the records; delete it and let it rebuild.

9.6.4 `use_coordinates` and `cell_conditioning`

`use_coordinates` appends the three fractional coordinates as extra input channels, giving the network a positional reference the field alone does not carry.

`cell_conditioning` feeds lattice descriptors through an embedding of width `embedding_dim` and uses it to modulate each layer's activations (FiLM). Without it the operator sees the field but not the cell it lives in, and cannot distinguish two materials whose grids agree while their lattice vectors do not. Keep both on unless deliberately ablating them.

9.6.5 `mode_selection` and `g_max`

`fixed` keeps the lowest `modes` indices on every material. Because the spacing of reciprocal-lattice points depends on the cell size, that means a *different physical band* for each material.

`physical` instead keeps every mode below a fixed wavevector $G_{\max}$, retaining $\lfloor G_{\max} L_i / 2\pi \rfloor$

modes along axis i, so every material contributes the same band of physics. Prefer it when cell sizes vary widely. It requires `g_max`.

Worth trying rather than only tidier. Measured on 115 Materials Project structures whose cells span 2.29 to 25.87 Å, `physical` with a generous `g_max` beat `fixed` by 20 % in *both* seeds, and was faster — fewer modes are retained. It reproduced in a second regime sharing no parameter with the first. The mechanism looks like uniformity rather than bandwidth: under `fixed`, cells spanning a factor of 11 in length retain physical bands spanning a factor of 11, and `physical` removes that.

It is not the default — one dataset, one architecture — and it is not a free switch either. **Too small is much worse than `fixed`**: `g_max 6` starved the band and lost 55 %. The useful value also moves with the regime, so it belongs to the run rather than to the schema; read the lines `poraque-train` prints at startup rather than copying a number from another config.

> **`g_max` does not size the radial basis under `fixed`**
>
> It did until 2026-09-04, and silently. `g_max` truncates nothing under `mode_selection: fixed`, so a reader concludes the key is inert — but the fallback that gives `g_basis` its default read it under either mode selection. A 90-run architecture study was carried out through a leftover `g_max: 6`, so every equivariant model in it was built with a 6.0 Å^{-1} basis rather than the 8.0 default, and nothing in the log, the resolved config or the run record said so. The measured accuracy cost was inside the seed spread, which is why it survived twenty runs.
>
> Stating a `g_max` beside `fixed` now warns and leaves the basis at its default. The coupling under `physical` is correct and kept: there $m_i = \lfloor G_{\max} L_i / 2\pi \rfloor$, so the inscribed radius $\min_i 2\pi m_i / L_i$ is at most $G_{\max}$ and a basis spanning $G_{\max}$ spans the retained band by construction.

9.6.6 `pauli_residual` and its scale

For `chg2tau`, the model predicts

$$\tau = \tau_{\mathrm{vW}}[\rho] + s\,\mathrm{softplus}\big(f_\theta(\rho)\big),$$

so the Hoffmann–Ostenhof bound $\tau \geq \tau_{\mathrm{vW}}$ holds by construction and the network learns only the Pauli term τ_{P}. Against a matched baseline it improved every fold — 18.3 % in mean relative L^2 — while training 17.8 % *faster*, because the network no longer has to re-derive $|\nabla\rho|^2/8\rho$, roughly 31 % of τ. Recommended.

`pauli_scale` sets the initial magnitude s; `null` fits it from the training split, which is the sensible default. `learn_pauli_scale` lets the optimiser refine it.

> **Caveat**
>
> The bound is a theorem for all-electron densities, whereas `CHGCAR` and `TAUCAR` hold pseudo quantities. Check it on new data before enabling the head — the training log reports any reference points that violate it.

9.7 Numerical precision

Two independent settings, because they control different costs:

Key	Default	Meaning
`data.precision`	`float64`	How the volumetric fields are *stored*: `float16`, `float32` or `float64`. A memory setting — a 160^3 field is 16 MB in double and 8 MB in single.
`model.precision`	`float32`	What the operator *computes* in: `float32` or `float64`.

Keep `data.precision` at `float64` wherever an integral over the cell is the result — the electron count, the energy terms are sums of N^3 values. Drop to `float32` when the field is on its way into a network that computes in single precision anyway.

`model.precision: float64` roughly doubles time and memory and exists to check that a physical result is not a single-precision artefact: an energy difference of a few meV assembled from N^3 voxel sums has no obvious margin in float32.

> **Caveat**
>
> `model.precision: float64` requires `training.device: cpu`. Apple's Metal backend does not implement double precision at all — not slowly, absent — and the run refuses at startup rather than failing an hour in.

9.8 `training` — how it is fitted

Key	Default	Meaning
`valid_fraction`	`0.2`	Fraction of structures held out for validation; `0` uses every structure.
`enable_kfold`	`false`	Run K-fold cross-validation instead; ignores `valid_fraction`.
`k_folds`	`5`	Number of folds, capped at the structure count.
`eval_epoch`	`10`	Evaluate and log every this many epochs.
`early_stopping`	`100`	Stop after this many epochs without validation improvement; `0` disables.
`epochs`	`300`	Passes over the training set.
`batch_size`	`4`	Maximum samples per grid-shape bucket.
`learning_rate`	`0.002`	AdamW step size.
`weight_decay`	`0.0001`	AdamW weight decay.
`scheduler`	`cosine`	`cosine` decay, or `null` for a constant rate.
`grad_clip`	`1.0`	Global gradient-norm clip; `0` disables.
`seed`	`0`	Weight initialisation, batch order and fold shuffling.
`device`	`auto`	`auto`, `cuda`, `mps` or `cpu`.
`strict_device`	`false`	Abort instead of falling back to the CPU; see below.
`distributed`	`auto`	`auto` or `off`; multi-GPU over NCCL when the launcher describes a group.
`distributed_timeout`	`30.0`	Minutes before a collective is declared failed.
`num_workers`	`0`	DataLoader worker processes. Read `data.cache_in_memory` first.
`pin_memory`	`auto`	Page-locked staging; `auto` is true on CUDA and false elsewhere.
`tf32`	`true`	TensorFloat-32 matmul and convolution; CUDA, Ampere and later.
`loss`	`relative_l2`	`absolute_l2`, `relative_l2`, `absolute_h1` or `relative_h1`.
`sobolev_weight`	`0.1`	Gradient-term weight for the two `h1` losses.
`physics_informed`	`{enable: auto}`	The switch and the four constraint weights, in one block; see below.

9.8.1 The validation split

There is **one** training protocol and **one** variation on it.

By default a run trains a single model per task, holding back `valid_fraction` of the structures for validation. That is all there is to it — no mode to select, no structures to name.

`enable_kfold` swaps that for K-fold cross-validation: each fold trains a fresh model on $K - 1$ groups and scores it on the group held out. It answers a different question — whether the

architecture generalises — and produces K models rather than one to deploy. `valid_fraction` is ignored, since the folds define the splits. Setting `k_folds` to the number of structures gives leave-one-out.

Implementation

Splitting is always at the *structure* level: whole materials move together. A voxel-level split would place the same crystal on both sides, and since neighbouring voxels are strongly correlated the score would look excellent while saying nothing about transfer to a new material.

Caveat

`valid_fraction` defaults to **0.2**, so an ordinary run reports a genuine held-out score and `early_stopping` is active. The trade-off is that the model has then seen only 80 % of the data. For the final deployable artefact set `valid_fraction: 0` — accepting that its own metrics become a **training fit** carrying no generalisation claim, about four times better than the cross-validated score on the reference dataset — and quote a separate `-kfold` run for the number.

9.8.2 eval_epoch

Evaluate and log every this many epochs. Validation is computed *only* on those epochs, so raising it on a large validation set is a genuine speed-up rather than only a quieter log. The final epoch always reports, so a run never ends without a current number.

```
progress (every 10 epochs):
  train loss: mean PhysicsInformedLoss per batch   |   val rel L2: held-out
  ↪ error, physical units
        epoch     train loss     val rel L2
  ----------------------------------------
       10/300        0.31745        0.34118
       20/300        0.18902        0.21447  *
```

9.8.3 early_stopping

Stop after this many epochs without an improvement in the **validation** error, and restore the best weights seen:

```
       30/300        0.03173        0.03659  *
       ...
       38/300        0.02249        0.06568
  stopped early at epoch 38: no improvement in 8 epochs
                                (best 0.03659 at epoch 30)

trained 38/300 epochs in 12.0 s   loss 0.8599 -> 0.0225
```

Caveat

It needs a validation split (`valid_fraction > 0`). With nothing held out there is only the training loss, which falls monotonically by construction and so can never signal that training should stop — asking for early stopping anyway **warns** rather than silently doing nothing and leaving you believing the run was protected.
The shipped default holds a fifth of the structures out (`valid_fraction: 0.2`), so early stopping is active out of the box. It goes inactive only if you set `valid_fraction: 0` to

> train on every structure, and the run says so rather than appearing to be protected.

Patience is counted in *epochs* but checked only on the epochs where validation is computed, so a value below `eval_epoch` behaves like `eval_epoch`.

The best weights are restored on exit, so the operator you get is the best one *measured* rather than merely the last one reached — stopping partway down a degrading curve would otherwise hand back the degraded model.

The `*` marks an epoch that improved on the best validation score. Only epochs on which validation was actually measured can do so — a checkpoint is written against a measured score, never an assumed one.

9.8.4 `batch_size`

Capped *per grid-shape bucket*. Materials whose grids differ in shape cannot be stacked into one tensor, so they are grouped by shape and batches drawn within a group. A bucket holding a single material yields batches of one however large this is set. The training log prints the buckets it found.

That cap is a real ceiling, not a formality: **above the largest bucket's size the setting stops meaning anything**. In the 115-structure set measured on a V100 the largest bucket held 70 materials (≈56 after the split), and `bs=64` and `bs=128` were the same run — identical peak memory, identical error, identical time. With the field cache on, throughput responded to the batch up to 16 and was flat thereafter.

`peak_vram_bytes` and `seconds_per_epoch` in the metrics JSON are what a sweep over this should be read from; before they were recorded there, the only route to either was sampling `nvidia-smi` from outside the process. The whole run's cost, stage by stage, closes the log (Section 9.9.1).

9.8.5 `strict_device`

`device: auto` prefers CUDA, then Apple Metal, then the CPU. An explicit backend that is unavailable **warns and falls back to the CPU**, so a configuration written on a machine with a GPU still runs on a laptop.

That is right on a workstation and expensive in a queue. `strict_device: true` turns the fallback into an error, and **every HPC configuration should set it**. The refusal names the probable cause — no GPU visible, an empty `CUDA_VISIBLE_DEVICES`, a driver older than the wheel's CUDA runtime, or a build carrying no kernels for this architecture — and prints the full device report into the run's log.

That last case is worth stating on its own, because `torch.cuda.is_available()` cannot detect it: it answers "is there a driver and a device", not "can this binary generate code for this device". A `+cu130` wheel on a V100 reports available and then aborts at the first kernel launch, since CUDA 13 dropped Volta. Poraquê checks the compute capability against the build's own `arch_list` and refuses before any of that. See §1.6.

`strict_device` applies to `device: auto` as well, which is the default and until 26.9.3 was the case it did *not* cover: the `auto` branch returned the best available backend before `strict` was consulted, so a configuration that set `strict_device: true` and left `device` alone — the exact pairing this section recommends — got no protection at all. It now refuses when `auto` resolves to the CPU, and names `device: cpu` as the way to ask for CPU training deliberately.

9.8.6 num_workers and pin_memory

`num_workers` and `data.cache_in_memory` (§9.5.1) are **alternatives, not complements**, and the measurement is unambiguous. Twenty epochs on 115 structures at 32^3, on a V100:

Variant	Training time	Speedup
As shipped	168.0 s	1.0×
`cache_in_memory: true`	**16.3 s**	**10.3**×
`num_workers: 4`, no cache	120.2 s	1.4×
Both	18.8 s	8.9×

The validation error was identical in all four, as it must be. Each worker is a *process* with a cache of its own, so it re-parses the whole dataset on its first epoch — the parse the cache removes is paid N times instead of once. Raise `num_workers` only when the data genuinely does not fit in RAM, which is the case `cache_in_memory: false` exists for.

`pin_memory` is a CUDA transfer optimisation and nothing else: `auto` is true there and false everywhere, since it does not help on Metal and some PyTorch versions warn when asked for it.

9.8.7 tf32

`tf32` keeps float32's range and drops its mantissa to ten bits inside the tensor cores. It is a real speedup on Ampere and later, and **exactly nothing** before it — a V100 has no TF32 path — so it is on by default and costs nothing where it does not apply. Ignored off CUDA.

9.8.8 There is no compile key

There was one, for a version, and it was removed in 26.9.3 after being measured. Setting `compile`, `compile_mode` or `compile_dynamic` now raises and says why, rather than being accepted and ignored.

Inductor does not generate code for complex operators. Every compiled run said so, in as many words, and `SpectralConv3d`'s weights are complex — it *is* the Fourier neural operator. So the one part of the model compilation was invoked to fuse is the part Inductor hands back to eager, while the wrapper and the graph breaks are still paid for. 150 epochs, two repetitions, one V100:

arm	s/epoch	first epoch (cold → warm)	val rel L^2
off	**0.6289**	3.8 s	0.280563
`mode="default"`	0.6785 (+7.9 %)	199.8 s → 17.6 s	0.284980
`mode="max-autotune"`	0.6879 (+9.4 %)	314.0 s → 20.7 s	0.272624

`TORCH_LOGS=recompiles` emitted nothing, so `dynamic=True` did hold one graph across the 19 shapes: the shape-bucketing worry was unfounded and the gain still does not exist. The validation error also moved, reproducibly and identically across both repetitions, which disqualifies the arms independently of their timing.

On four ranks it did not merely lose. With `-distributed auto` and 60 epochs, in the same allocation as an uncompiled four-rank run that finished in 86.6 s, the compiled run sat **eighteen minutes in the first epoch without printing an epoch line** and was cancelled. A configuration key whose failure mode is a silent deadlock inside a queue allocation is the hazard `strict_device` exists to remove, pointing the other way.

What was *not* measured, and is worth measuring one day: compiling only the real-valued submodules — `CellEncoder`, FiLM, the pointwise convolutions, the projection head, the KAN activations — and leaving `SpectralConv3d` in eager. That is where the 44.4 % of elementwise

GPU time lives. It should come back as an internal decision about which submodules to wrap, not as a switch a config file can throw. The condition for re-opening whole-model compilation is upstream: Inductor gaining complex-operator codegen.

9.8.9 distributed and distributed_timeout

`auto` — the default — forms a `DistributedDataParallel` group over NCCL **when the environment already describes one**, and does nothing otherwise. A Slurm step with more than one task, or a `torchrun` launch, is such an environment; a workstation is not.

It cannot turn a one-process run into a four-GPU one. The launcher decides the topology and this key decides only whether to believe it, so on a cluster the setting that matters is in the submission script — see section 1.6.3, which carries the whole of it.

`off` refuses a group inside a multi-task allocation, which is how four independent single-GPU jobs are run from one submission and how a scaling result is bisected against a single-device baseline.

Three consequences, all visible in the run log. The **effective batch size is `batch_size` times the world size**, so a four-rank run at `batch_size: 10` steps on 40 samples and is not the same optimiser as the one-rank run it is compared against — halve the learning rate or quarter the batch size if the comparison is meant to be of the same optimisation. **Rank 0 alone writes** the checkpoint, the metrics, the figures and the PDF, and alone prints. And **the batches are split, not the samples**: `ShapeBucketSampler` groups materials by grid shape so no padding ever reaches the FFT, and a `DataLoader` takes a `sampler` or a `batch_sampler` and never both — so the bucketing runs first, identically on every rank, and a real `DistributedSampler` partitions the resulting *list of batches*. Every rank gets a unique, non-overlapping subset and the same *number* of batches, which is not tidiness: DDP all-reduces gradients inside each `backward()`, and a rank that runs out of batches first leaves the others in a collective that never completes.

`distributed_timeout` is long on purpose. The first collective happens after every rank has read its prepared cache, and a cold parallel-filesystem read of a few hundred densities is minutes; a timeout that fires there reports itself as a NCCL error and sends the reader looking at the network.

> **Note**
> There is no `DataParallel` fallback and no Gloo backend. Both would let a misconfigured run go quietly slower than a single GPU — the first by replicating in one process, the second by distributing across CPU cores inside a GPU allocation. Without CUDA the group is refused with a warning naming the probable cause, and the run continues on one device.

9.8.10 loss and sobolev_weight

Four objectives, on two named axes:

`loss`	normalised per sample	gradients in the objective
`absolute_l2`	no	no
`relative_l2`	yes	no
`absolute_h1`	no	yes
`relative_h1`	yes	yes

Relative divides each sample's error by its own target norm, so materials whose fields differ by orders of magnitude contribute equally and the value reads directly as a fraction. **Absolute** does not, so every *voxel* counts equally and a denser system contributes proportionally more

gradient — right for a set whose materials are genuinely comparable in magnitude, and usually wrong for a heterogeneous one, which is why `relative_l2` is the default.

H1 adds `sobolev_weight` times the same error on the spatial gradients. Worth enabling when the derivative matters — τ_{vW} depends on $\nabla\rho$, so gradient noise is amplified in low-density regions. Both halves are taken in the *same* norm, so the weight is a pure ratio between values and derivatives rather than also absorbing a change of scale.

The validation column follows the objective. The progress table reports `val abs L2`, `val rel L2`, `val abs H1` or `val rel H1` — whichever is running — and the number behind it is the objective's own data term evaluated on the held-out set:

```
    train loss: mean PhysicsInformedLoss per batch   |   val rel H1: held-out
    ↪ error, physical units
    val rel H1 = rel L2 + 0.1 x the relative L2 of the gradient, matching the
    objective; the final per-structure table below reports plain rel L2
          epoch     train loss     val rel H1
```

That is not only a label. Early stopping and the checkpoint are decided on this number, and while it read `val rel L2` regardless a gradient-constrained run was selecting the model that minimised a functional it was not training on. The four are not comparable with each other — which is why the *per-structure table at the end of a run reports relative L^2 whatever the objective was*. That is the number to quote, and the one that lets two runs be put side by side.

> **Note**
> `loss: sobolev` was renamed on 2026-08-31. It named the gradient term and left the norm implicit, and offered no unnormalised form at all. It now raises, naming `relative_h1` (what it did) and `absolute_h1` as its replacements.

9.8.11 physics_informed

One block, following the convention of §9.2:

```
training:
  physics_informed:
    enable: auto              # auto | true | false
    electron_count_weight: 0.1
```

Key	Constraint it penalises the violation of
`enable`	— decides whether any of the below is evaluated
`electron_count_weight`	$\int \rho \, \mathrm{d}\mathbf{r} = N$ (`ext2chg`)
`positivity_weight`	Non-negativity of the predicted field
`von_weizsacker_weight`	$\tau \geq \tau_{\mathrm{vW}}$ (`chg2tau`)
`euler_lagrange_weight`	Orbital-free stationarity residual (`ext2chg`)

All four weights default to zero, so the objective is the plain supervised baseline until one is enabled deliberately. Each term is dimensionless, so a single weight can serve a heterogeneous dataset.

`enable` has three states, and the middle one is why it exists. `auto`, the default, answers from the weights — the run is physics-informed if and only if one of them is positive, which is what every configuration written before the switch existed already meant. `true` **raises** when no weight is set: a run that declares physics-informed training and silently optimises the supervised baseline has an entirely ordinary loss curve and a report that says otherwise. `false` zeroes the weights and

warns if one was live, and also skips the per-batch decode the constraints act on — measured at 2.4 % of a step on Apple Metal and 6.6 % on a V100, where the copy crosses PCIe.

> **Caveat**
> Not `off`, and not `no`. Anything that is not `auto`, `true` or `false` raises rather than being coerced: `bool("off")` is `True` in Python, so a config saying `off` would have switched the constraints **on**.

> **Caveat**
> Introduce them one at a time against a measured baseline, and only once the data term has converged — a physics residual evaluated at random initialisation is noise. A badly scaled constraint degrades accuracy while looking principled.

Where a constraint can be imposed structurally, prefer that: `pauli_residual` enforces $\tau \geq \tau_{\mathrm{vW}}$ exactly, whereas `von_weizsacker_weight` only discourages violating it.

9.9 output — what is written

Key	Default	Meaning
`root`	`models`	Parent of the run folder. Everything goes in `<root>/<name>/`; `null` disables *all* output.
`checkpoint`	`true`	Write `<name>.poraque`.
`write_log`	`true`	Write `log/`: the log, the metrics JSON and the resolved configuration.
`plot_figures`	`true`	Render `plots/`.
`write_pdf_report`	`true`	Typeset `report/`.
`log`, `json`	`null`	Override the two log paths. `null` derives them from `task.name`; set them only to write outside the run folder.
`plot_format`	`png`	`png`, `pdf` or `svg`.
`dpi`	`200`	Raster resolution for saved figures.
`save_raw_plot_data`	`false`	Write the numbers behind each figure beside it, so a plot can be redrawn without re-running the model.

9.9.1 The resource profile

The last thing `poraque-train` prints is what the run cost:

```
RESOURCE PROFILING SUMMARY
============================================================
  stage                                wall time     peak RSS
  ------------------------------------------------------------
  cache                                    0.0 s    222.6 MiB
  ext2chg: setup                           0.0 s    228.4 MiB
  ext2chg: training                        0.4 s    319.7 MiB
  ext2chg: evaluation and figures          1.6 s    500.5 MiB
  ext2chg: PDF report                      0.0 s    500.5 MiB
  chg2tau: setup                           0.0 s    501.3 MiB
```

```
chg2tau: training                        0.0 s    502.9 MiB
chg2tau: evaluation and figures          0.9 s    552.8 MiB
chg2tau: PDF report                      0.0 s    552.8 MiB
checkpoint                               0.0 s    552.8 MiB
-----------------------------------------------------------
total (wall clock)                       3.1 s    552.8 MiB
device: cpu   outside the stages above: 0.1 s
```

That is a synthetic smoke run; the shape is what matters. Each stage is timed after a device synchronisation, so a GPU stage is charged for its compute rather than for the kernels it queued. The memory columns answer the question an allocation is sized from:

peak RSS is the process's high-water resident set, read from the operating system rather than sampled, so a spike between two samples cannot be missed. It is a running maximum and never falls; the stage where it jumps is the one that needed the memory. DataLoader workers are separate processes and are reported on a line of their own.

peak VRAM , on CUDA, is `torch.cuda.max_memory_allocated` *within* each stage — the peak statistics are reset at every boundary, so the column does fall. The summary adds the most the caching allocator reserved, which is the number `nvidia-smi` shows and the one an out-of-memory error is about.

MPS at end is what the Metal driver held when the stage ended. MPS exposes no peak, so on a Mac this column is a floor, not a maximum.

> **Caveat**
> The summary prints from a `finally`, so a run that raises — an out-of-memory error, a bad file, a Ctrl-C — still reports every stage it finished and marks the one it died in `(interrupted)`: the run whose cost most needs explaining is the one that did not finish. A job killed by a signal is not so lucky. Slurm ends a job at its time limit with `SIGTERM`, which Python does not turn into an exception, so that run leaves no summary; the metrics JSON of a finished run is the record to size the next allocation from.

`-cache-only` prints the summary too, with the cache as its only stage. The same record is in the metrics JSON under `resources`; `-kfold` times the whole cross-validation of each task as one stage.

9.9.2 save_raw_plot_data

A figure is an argument someone will later want to make differently — in a journal's colours, at a journal's size, with two runs on one axis. Re-running the model to get the numbers back is the expensive way to do that, so with this on each figure method writes what it drew, beside what it drew:

```
plots/ext2chg_loss_curves.png       plots/ext2chg_loss_curves.csv
plots/ext2chg_parity.png            plots/ext2chg_parity.csv
                                    plots/ext2chg_parity_bin_edges.csv
plots/ext2chg_s000_field_slice.png  plots/ext2chg_s000_field_slice.npz
```

The sidecar shares the **stem** of its figure, so the pairing is a property of the directory listing rather than of a convention to remember. Tabular figures get a CSV; the slice figure gets a compressed `.npz`, because what it draws are three 2D arrays and a CSV of a 108×108 grid is neither smaller nor easier to read back.

The loss curve is one row per epoch: `epoch`, `train_loss`, and the validation column *named for the norm it holds* (`val_rel_l2`, or `val_abs_h1` for an `absolute_h1` run). Epochs on which validation was not measured leave that cell **empty** rather than interpolated: filling them would put points in the file the run never measured. There is no physics column — `train_loss` is the total the optimiser stepped on, and the per-term breakdown is not recorded.

The parity plot is one row per *occupied* bin: `split`, `reference`, `prediction`, `count`, `density`. Long format, because that pivots straight back into a `pcolormesh`; occupied bins only, because a 200-bin grid is 40 000 cells of which a few thousand carry anything. The bin edges go in their own file, since a log axis makes them unrecoverable from the centres. `count` is what the figure drew, so on log axes it leaves out every sampled voxel at which reference or prediction is not positive; the panel titles give the sample itself. A spin-polarised run writes the same pair again for m, as `<task>_parity_magnetisation`.

The slice figure stores the three panels as arrays — `reference`, `prediction` and `error` — together with the colour limits the figure used. The error is *stored*, not left to be recomputed: a reader who reconstructs it and gets something else then knows the file is inconsistent, rather than trusting their own subtraction.

It needs `plot_figures`. The data is written by the figure methods as they draw, which is what guarantees the file and the image show the same numbers rather than two independent computations of them.

The resolved configuration is written next to `json` with the suffix `_config.yaml`. PDF reports are assembled in a temporary directory that is removed afterwards, so no `.tex`, `.aux` or `.log` files are left behind.

9.10 `symbolic` — distilling a formula

Chapter 8 covers this feature; the keys are listed here for completeness. It is off by default and needs the optional `symbolic` install extra.

Key	Default	Meaning
`enable_symbolic_distillation`	`false`	Search for a closed-form expression after training.
`target`	`model`	`model` distils the operator's predictions; `reference` fits the DFT data.
`features`	`gga`	Variables given to the engine: `gga` (ρ, p, q), `enhancement` (p, q), or `raw`.
`epsilon`	`1e-8`	Vacuum threshold in e/a_0^3; clamps every denominator and drops voxels below it.
`unary_operations`	`[exp, log, sqrt, abs]`	Unary alphabet, passed to the engine untouched.
`binary_operations`	`['+', '-', '*', /, ^]`	Binary alphabet.
`iterations`	`40`	Search iterations — the main quality/time knob.
`population_size`	`33`	Individuals per population.
`populations`	`15`	Populations run in parallel.
`max_depth`	`10`	Ceiling on expression depth.
`max_size`	`30`	Ceiling on node count.
`parsimony`	`0.0032`	Penalty per node; higher gives shorter, worse-fitting expressions.
`n_samples`	`4000`	Voxels sampled across the dataset.
`seed`	`0`	Seeds the sampling and the engine.
`template`	`none`	How the target is factorised: `none`, `thomas_fermi` or `pauli`.
`data_loss`	`mse`	The unpenalised part of the objective; `mae` is more robust on a density whose tails span orders of magnitude.

`symbolic.physics` — constraints on the search

Key	Default	Meaning
`enable`	`true`	Penalise violations *inside* the evolutionary loop rather than filtering the front afterwards.
`positivity_weight`	`100.0`	$F \geq 0$, on the batch.
`thomas_fermi_weight`	`100.0`	$F(0, 0) = 1$: the uniform gas.
`von_weizsacker_weight`	`100.0`	$F \to 0$ as $p \to \infty$: one orbital.
`p_infinity`	`1.0e6`	The reduced gradient standing in for $p \to \infty$ in that probe.

Two blocks of physics weights, and they are not the same

`training.physics_informed` constrains the **neural operator**: charge conservation, positivity of a predicted density, the Euler–Lagrange residual. Its terms are added to a training loss over voxels, and its own `enable` decides whether any of them is evaluated — `auto` answers from the weights, `true` refuses to be an empty claim, `false` also skips the per-batch decode the terms act on.

`symbolic.physics` constrains a **candidate algebraic expression** for the Pauli enhancement factor. Its terms are added to a symbolic-regression fitness over synthetic probe points.

Separate objectives, on separate objects, evaluated at different times by different engines. Two of the key names appear in both — `positivity_weight` and `von_weizsacker_weight` — and mean different things in each, which is exactly why they live in separate blocks rather than sharing a prefix. Both are now the same *shape* — `{enable: ..., weights}` — which makes the collision easier to see and no less real.

Filtering the front after a run only measures how few candidates were physical; by then the populations have spent their budget converging on forms that must be discarded. With `enable` on, the objective is

$$\mathcal{L} = \mathcal{L}_{\text{data}} + w_{+}\,\overline{\min(F,0)^2} + w_{\text{TF}}\,|F(0,0)-1| + w_{\text{vW}}\,|F(p_\infty,0)|.$$

The first term is evaluated on the batch, the other two on probe points — so this needs the engine's *full objective*, not an elementwise loss, which only ever sees `(prediction, target)` pairs and can never evaluate a candidate where the data does not sit.

Note

The `loss` column of the reported front is then this *constrained* objective and is not comparable with an unconstrained run's. The R^2 and relative L^2 are computed from the expression itself and are unaffected.

9.11 Worked examples

```
task: all
model:    {pauli_residual: true}
training: {valid_fraction: 0, epochs: 200}
```

```
task: all
training: {enable_kfold: true, k_folds: 5}
```

```
data:     {resolution: 20}
model:    {width: 8, modes: 4, n_layers: 2, projection_channels: 16}
training: {epochs: 10}
```

```
data:     {resolution: 64}
model:    {width: 32, modes: 12, n_layers: 4}
training: {epochs: 400, batch_size: 2}
```

CHAPTER 10

Troubleshooting

10.1 The run stops immediately

"No struct* directories" Check `data.root` and `data.pattern`. The pattern is a prefix, not a glob.

"Unknown key(s) in section ..." A configuration key was removed. `mode`, `holdout`, `use_vasp_extcar`, `train_fraction` and `split` no longer exist; delete them, or start again from `configs/train.yaml`. The split is now one number, `valid_fraction`, and `EXTCAR` is always computed.

"No valence charge for ..." No `POTCAR` was found. Supply one, or pass `-zval Pt=11`.

"Unknown key(s) in section ..." A typo in the YAML. The message lists the valid keys.

10.2 The results look wrong

R^2 is negative The model is worse than predicting the mean. Usually too few epochs, or a learning rate that diverged — check the loss curve in `models/<name>/plots/`.

The electron count is far off Nothing constrains it by default. A few per cent is normal; tens of per cent means the model is extrapolating, typically because the evaluation grid is far from the training resolution.

The numbers seem too good Check whether anything was held out. With `valid_fraction: 0` they are a training fit. The report's *What these numbers do not show* section states this explicitly.

10.3 Grids and shapes

"grid shape ... does not match the supplied shared grid" Two fields of one material are on different meshes. All three must share a grid; read them with `grid=` from the same source.

"Cannot batch mixed grid shapes" Use the shape-bucketed loader (the training script does) or `batch_size: 1`.

A few negative voxels in `TAUCAR` Band-limiting a strictly positive field with sharp core peaks rings slightly. It is an artefact of the downsampling, not of the data, and is why the pipeline

uses a sign-tolerant asinh normalisation.

10.4 Devices

"CUDA was requested but ..." A warning, not an error — the run continues on CPU.

Training is slower on MPS than CPU At small grids kernel-launch overhead dominates. The advantage grows with size: 2.2× at 32^3, 5.7× at 64^3.

Results differ slightly between devices Expect $\sim 10^{-6}$ relative on a single forward pass. Two *independently trained* runs will differ much more, because optimisation amplifies tiny differences — that is not a bug.

10.5 Reports

A `.tex` appears instead of a PDF No `latexmk` or `pdflatex` on the path. The source is written so it can be compiled elsewhere.

Stray `.aux` or `.log` files Should not happen: reports are built in a temporary directory that is deleted wholesale. If you see them, they came from compiling the guides by hand — use `make` in `latex/technical_guide/` or `latex/user_guide/`, which cleans up.