



Poraquê

User Guide

Predicting charge and kinetic energy densities
from atomistic geometry

Leandro Seixas

Version 26.8.40

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:

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

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
<code>latexmk</code> (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 Faster CPU inference	5
1.7 Optional components	6
2 Preparing data	7
2.1 Directory layout	7
2.2 Grids may differ between materials	7
2.3 Checking your data	7
2.4 Downloading from the Materials Project	8
2.5 What an MP download does and does not contain	9
2.6 The external potential without a POTCAR	9
2.7 Training on several sources at once	10
3 Training	11
3.1 The short version	11
3.2 One model, all structures	11
3.3 Measuring generalisation	12
3.4 Reading the output	12
3.5 Reference numbers	13
3.6 Options worth knowing	13
4 Predicting a new structure	14
4.1 Choosing the grid	14
4.2 Matching a VASP grid	14
4.3 Comparing against a reference	15
4.4 Restarting VASP from a predicted density	16
4.5 Sanity checks the script performs	17
4.6 Visualising	17
5 Energies and the ASE calculator	18
5.1 The energy expression	18
5.2 Choosing the exchange-correlation functional	19
5.3 Computing energies from files	19
5.4 The ASE calculator	20
5.5 What the number is not	23

6	Charges and population analysis	25
6.1	Charge conservation	25
6.2	Partial charges	26
6.3	What these charges are not	28
6.4	In the reports	28
7	Fine-tuning	29
7.1	What comes from the checkpoint	29
7.2	Freezing the lifting path	29
7.3	Configuration	30
7.4	Output, and the <code>.pfno</code> format	30
8	Symbolic distillation	32
8.1	What it can and cannot find	32
8.2	The physical variables	32
8.3	Operator constraints	34
8.4	Figures	34
8.5	Regularization in vacuum	34
8.6	Configuration	35
8.7	Output	35
8.8	The Pareto knee	36
8.9	Rounding	36
8.10	Physical asymptotic compliance	36
9	Configuration reference	38
9.1	How a value is resolved	38
9.2	Every key is optional	38
9.3	Top level	39
9.4	<code>data</code> — where the fields come from	39
9.5	<code>model</code> — the operator’s shape	41
9.6	Numerical precision	42
9.7	<code>training</code> — how it is fitted	43
9.8	<code>output</code> — what is written	46
9.9	<code>symbolic</code> — distilling a formula	46
9.10	Worked examples	48
10	Troubleshooting	49
10.1	The run stops immediately	49
10.2	The results look wrong	49
10.3	Grids and shapes	49
10.4	Devices	50
10.5	Reports	50

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 an optional extra, which pulls in PySR and its Julia toolchain:

```
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
<code>poraque-train</code>	trains the operators
<code>poraque-inference</code>	predicts a new structure
<code>poraque-mp</code>	downloads densities from the Materials Project
<code>poraque-committee</code>	calibration: does ensemble disagreement predict error?
<code>poraque-active-learning</code>	selection: which structures should the next DFT runs be spent on?
<code>poraque-vasp</code>	writes the VASP inputs that read a prediction back: bands, dos, energy

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

The device list is ordered by preference, so the first entry is what `device:` `auto` will select — `cuda`, then `mps`, then `cpu`.

1.6 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 (pthread)
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 $\sim 1\times$ at 96^3 where the FFTs dominate
Training	unaffected: the kernel records nothing on the autograd tape, so it is used only under <code>torch.no_grad()</code>
Accuracy	identical to float32 rounding ($\sim 10^{-7}$ relative), checked on every call path by the test-suite
To disable	<code>PORAQUE_C_BACKEND=0</code>

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

1.7 Optional components

Component	Needed for	Install
CUDA build of PyTorch	NVIDIA GPUs	https://pytorch.org
<code>pdflatex</code> / <code>latexmk</code>	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.

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 NGX_F, NGY_F, NGZ_F, 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 $\sum_a Z_a^{\text{val}}$:

```
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. 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
```

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 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 What an MP download does and does not contain

It contains one compressed `CHGCAR` per material and nothing else — no `POSCAR`, `INCAR`, `POTCAR`, `OUTCAR` or `TAUCAR`. 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`.

2.6 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:
  train_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 <code>EXTCAR</code>
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, and the cache directory records it (`res32_potcar`), so a tabulated cache and a Gaussian one can never be mistaken for each other.

Without a library the map learned is *model potential* $\rightarrow$ *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.

2.7 Training on several sources at once

`data.train_paths` is a list, and each entry is detected from its contents — a directory of DFT runs, an archive of standalone CHGCARs, or a prepared cache from an earlier run:

```
data:
  train_paths:
    - data/vasp          # your own calculations
    - data/MP            # a Materials Project download
  potcar_dir: /opt/vasp/potpaw_PBE
```

Everything found is pooled into one training set, and identifiers that collide between archives are prefixed with their directory so no material is silently dropped.

Caveat

Without `potcar_dir`, mixing a calculation archive with a bulk archive 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>.pfno	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

<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.

3.2 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; <i>negative</i> 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)
<code>ext2chg</code>	0.0379 ± 0.0027	0.0209
<code>chg2tau</code>	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 $\tau = \tau_{vW}[\rho] + s \text{softplus}(f)$ so the exact bound $\tau \geq \tau_{vW}$ 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
```

```
# take ENCUT and PREC from a real INCAR: 450 eV, Accurate -> 64^3
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.

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

```
poraque-inference struct/ --to-vasp --from-incar struct/INCAR --add-paw
```

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 NGX_F NGY_F NGZ_F, 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) = XCUTOFF*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 $4 \times 15.25 = 61 \rightarrow 64$, 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 $61 \rightarrow 64$ and $109 \rightarrow 112$.

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 CHGCARs, averaged per chemical element, and stored in the `.pfno` bundle:

```
PAW reference: Pt 138 values, averaged over 494 atoms in 17 structure(s)
PAW reference -> stored in the bundle (Pt)
```

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 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 NGX. 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 $\sum_a Z_a^{\text{val}}$;
- the number of negative density voxels, if any;
- the bound $\tau \geq \tau_{\text{vW}}[\rho]$ 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.

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

$$E = \underbrace{\int \tau d^3r}_{T_s} + \underbrace{\int \rho V_{\text{ext}} d^3r + E_{\alpha Z}}_{E_{\text{ext}}} + \underbrace{\frac{1}{2} \int \rho v_{\text{H}} d^3r}_{E_{\text{H}}} + E_{\text{xc}}[\rho] + E_{\text{Ewald}}$$

Term	How it is obtained
T_s	integral of the predicted TAUCAR
E_{ext}	$\int \rho V_{\text{ext}}$, 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, $v_{\text{H}}(\mathbf{G}=0) = 0$
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,

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

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 $F_x(0) = 1$ and $H \rightarrow 0$. 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.pfno", "chg2tau.pfno", 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

Poraquê behaves like MACE or NequIP at the interface:

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

atoms = bulk("Pt", "fcc", a=3.92, cubic=True)
atoms.calc = Poraquê("models/poraque_models.pfno", 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` $\rightarrow$ `Structure` $\rightarrow$ `FieldGrid` $\rightarrow$ $V_{\text{ext}} \rightarrow \rho \rightarrow \tau \rightarrow E$. 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 <code>FieldOperators</code> .
potcar	A single POTCAR covering the species present.
potcar_dir	A POTCAR <i>library</i> , 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.pfno",
                    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,

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

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.pfno",
                    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
"poraquê" (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 $\nabla_{\mathbf{R}} E_{\text{ref}} = 0$; 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:

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

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×
<i>True spread being resolved</i>	<i>0.125 eV/atom</i>	—

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.

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,

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

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.pfno", 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,

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

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	$\rho_A^{\text{at}} / \sum_B \rho_B^{\text{at}}$	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.pfno", 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 $\rho^{\text{at}}(r)$. 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)
```

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.pfno \
  --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.pfno
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
<code>enable</code>	<code>false</code>	Start from <code>pretrained_checkpoint</code> instead of a fresh initialisation.
<code>pretrained_checkpoint</code>	<code>models/poraque_models.pfno</code>	Base bundle to adapt.
<code>learning_rate</code>	<code>1e-05</code>	Replaces <code>training.learning_rate</code> for this run.
<code>freeze_lifting_layers</code>	<code>false</code>	Hold the input lifting path fixed.

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.

7.4 Output, and the .pfno format

Poraquê writes its models as `.pfno` — *Poraquê Fourier Neural Operator*:

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

The container is a `torch.save` payload keyed by task, with a format tag and metadata; the extension is a label, not a different serialisation. Nothing inspects it when loading, so a bundle 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 bundle metadata records `fine_tuned_from`, the learning rate and whether the lifting layers were frozen, so a checkpoint can always be traced back to its parent.

Note

Models written before this rename carry `.pth`. If the `.pfno` file is absent and a `.pth` of the same stem is present, it is used and the substitution is announced — an existing trained model should not become invisible because a default filename changed. Rename it to silence the notice.

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, and an optional install. PySR carries a **Julia** toolchain, which it fetches the first time a search runs, so it is not pulled in by a plain installation.

```
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:

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

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

$\nabla \rightarrow i\mathbf{G}$ and $\nabla^2 \rightarrow -|\mathbf{G}|^2$, 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	$F = \frac{\tau - \tau_{\text{vW}}}{\tau_{\text{TF}}}$	$\tau = \tau_{\text{vW}} + \tau_{\text{TF}} f(\dots)$
thomas_fermi	$F = \tau / \tau_{\text{TF}}$	$\tau = C_{\text{TF}} \rho^{5/3} f(\dots)$

pauli is the physically right choice. $\tau_{\text{vW}} = |\nabla\rho|^2/8\rho$ is known in closed form and $\tau - \tau_{\text{vW}} \geq 0$ 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 $\wedge$ 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
<code><task>_symbolic_pareto.png</code>	the front, with the lowest-loss candidate and the knee marked
<code><task>_symbolic_parity.png</code>	the lowest-loss expression against the DFT reference
<code><task>_symbolic_parity_knee.png</code>	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 $\rho \leq \text{epsilon}$ 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 $1\text{e-}8$, 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_symbolic_distillation: 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,

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

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 $F = \tau/\tau_{\text{TF}}$: Both are statements about the *Pauli*

enhancement factor $F = (\tau - \tau_{\text{vW}})/\tau_{\text{TF}}$:

$$\underbrace{F(0,0) = 1}_{\text{Thomas-Fermi}}, \quad \underbrace{F \rightarrow 0 \text{ as } p \rightarrow \infty}_{\text{von Weizsäcker}}. \quad (8.2)$$

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 $\tau \rightarrow \tau_{\text{vW}}$, so the Pauli term added to it must vanish.

Every template is converted to this one convention before checking, using $\tau_{\text{vW}}/\tau_{\text{TF}} = 5p^2/3$ 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.

Caveat

The search is stochastic. Two runs with the same `seed` can still differ unless PySR is also put in deterministic, serial mode — it warns about this itself. Treat a single expression as one sample, not as *the* answer.

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 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.

Four configurations ship with the source:

File	Purpose
<code>configs/train.yaml</code>	The clean starting point. Copy this one.
<code>configs/train_complete_and_commented.yaml</code>	The same run, with every choice explained at length — the reference to read, not to copy.
<code>configs/train_materialsproject.yaml</code>	Training on a Materials Project density download.
<code>configs/train_mixed.yaml</code>	One operator from several datasets at once.

9.3 Top level

Key	Default	Meaning
<code>task</code>	<code>all</code>	Which map to train: <code>ext2chg</code> , <code>chg2tau</code> , or <code>all</code> for both in sequence.

9.4 data — where the fields come from

Key	Default	Meaning
<code>train_paths</code>	<code>null</code>	List of dataset directories, which may mix layouts. Falls back to <code>root</code> .
<code>root</code>	<code>data/vasp</code>	Single dataset directory, used when <code>train_paths</code> is <code>null</code> .
<code>source</code>	<code>auto</code>	Layout of each path: <code>auto</code> , <code>vasp</code> , <code>bulk</code> or <code>prepared</code> .
<code>cache</code>	<code>data/cache</code>	Where downsampled copies are written.
<code>pattern</code>	<code>struct</code>	Prefix identifying subdirectories of a <code>vasp</code> path — a prefix, not a glob.
<code>code</code>	<code>auto</code>	DFT code, or <code>auto</code> to detect it from the files present.
<code>resolution</code>	<code>32</code>	Longest grid axis after spectral downsampling.
<code>potcar_dir</code>	<code>null</code>	POTCAR library, used where the data ships no pseudopotentials.
<code>sigma</code>	<code>null</code>	Gaussian pseudo-ion width in Å, where the Gaussian model is reached.
<code>gaussian_blur</code>	<code>null</code>	Gaussian blur width in Å applied to the computed potential.
<code>blur_method</code>	<code>spectral</code>	<code>spectral</code> or <code>ndimage</code> .

`source` and `train_paths` are covered in Chapter 2; `potcar_dir` and its Gaussian fallback in Section 2.6, which is the setting to read before quoting any number from a run on archive data.

9.4.1 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 $32 \rightarrow 64$ 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.

9.4.2 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 $\sigma = 0.15 \text{ \AA}$ 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 model — the operator’s shape

Key	Default	Meaning
<code>width</code>	16	Channel width of the Fourier layers.
<code>modes</code>	8	Retained Fourier modes per axis.
<code>n_layers</code>	4	Number of Fourier layers.
<code>projection_channels</code>	48	Hidden width of the output projection.
<code>activation</code>	<code>gelu</code>	<code>gelu</code> , <code>relu</code> , <code>silu</code> or <code>tanh</code> .
<code>use_coordinates</code>	<code>true</code>	Append three fractional-coordinate input channels.
<code>cell_conditioning</code>	<code>true</code>	Condition every layer on lattice descriptors (FiLM).
<code>embedding_dim</code>	32	Width of that cell embedding.
<code>mode_selection</code>	<code>fixed</code>	<code>fixed</code> mode index, or <code>physical</code> at constant $G_{\max}$.
<code>g_max</code>	<code>null</code>	Cutoff wavevector in $\text{\AA}^{-1}$; required by <code>mode_selection: physical</code> .
<code>pauli_residual</code>	<code>false</code>	Structural $\tau \geq \tau_{\text{vW}}$ for <code>chg2tau</code> .
<code>pauli_scale</code>	<code>null</code>	Initial Pauli scale in $\text{eV}/\text{\AA}^3$; <code>null</code> fits it from the training split.
<code>learn_pauli_scale</code>	<code>true</code>	Optimise that scale alongside the backbone.

9.5.1 `width`, `modes`, `n_layers`

These three set the capacity. The parameter count is dominated by the spectral weights — $4w^2m^3$ 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.

9.5.2 `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.

9.5.3 `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`.

9.5.4 pauli_residual and its scale

For `chg2tau`, the model predicts

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

so the Hoffmann–Ostenhof bound $\tau \geq \tau_{\text{vW}}$ 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.

9.6 Numerical precision

Two independent settings, because they control different costs:

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

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.

9.7 training — how it is fitted

Key	Default	Meaning
<code>valid_fraction</code>	0.2	Fraction of structures held out for validation; 0 uses every structure.
<code>enable_kfold</code>	false	Run K-fold cross-validation instead; ignores <code>valid_fraction</code> .
<code>k_folds</code>	5	Number of folds, capped at the structure count.
<code>eval_epoch</code>	10	Evaluate and log every this many epochs.
<code>early_stopping</code>	100	Stop after this many epochs without validation improvement; 0 disables.
<code>epochs</code>	300	Passes over the training set.
<code>batch_size</code>	4	Maximum samples per grid-shape bucket.
<code>learning_rate</code>	0.002	AdamW step size.
<code>weight_decay</code>	0.0001	AdamW weight decay.
<code>scheduler</code>	cosine	cosine decay, or null for a constant rate.
<code>grad_clip</code>	1.0	Global gradient-norm clip; 0 disables.
<code>seed</code>	0	Weight initialisation, batch order and fold shuffling.
<code>device</code>	auto	auto, cuda, mps or cpu.
<code>loss</code>	relative_l2	relative_l2 or sobolev.
<code>sobolev_weight</code>	0.1	Gradient-term weight when loss: sobolev.
<code>physics</code>	all 0.0	Physics-informed loss weights; see below.

9.7.1 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.

9.7.2 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 *
```

9.7.3 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.

9.7.4 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.

9.7.5 loss and sobolev_weight

relative_l2 normalises the error per sample, so materials whose fields differ by orders of magnitude contribute equally and the reported value reads directly as a fraction.

sobolev adds a relative gradient term weighted by **sobolev_weight**. Worth enabling when the derivative matters — τ_{vW} depends on $\nabla\rho$, so gradient noise is amplified in low-density regions.

9.7.6 physics

Weight	Constraint it penalises the violation of
electron_count_weight	$\int \rho \, d\mathbf{r} = N$ (ext2chg)
positivity_weight	Non-negativity of the predicted field
von_weizsacker_weight	$\tau \geq \tau_{vW}$ (chg2tau)
euler_lagrange_weight	Orbital-free stationarity residual (ext2chg)

All four 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.

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 $\tau \geq \tau_{vW}$ exactly, whereas **von_weizsacker_weight** only discourages violating it.

9.8 output — what is written

Key	Default	Meaning
<code>root</code>	<code>models</code>	Parent of the run folder. Everything goes in <code><root>/<name>/</code> ; <code>null</code> disables <i>all</i> output.
<code>checkpoint</code>	<code>true</code>	Write <code><name>.pfno</code> .
<code>write_log</code>	<code>true</code>	Write <code>log/</code> : the log, the metrics JSON and the resolved configuration.
<code>plot_figures</code>	<code>true</code>	Render <code>plots/</code> .
<code>write_pdf_report</code>	<code>true</code>	Typeset <code>report/</code> .
<code>log, json</code>	<code>null</code>	Override the two log paths. <code>null</code> derives them from <code>task.name</code> ; set them only to write outside the run folder.
<code>plot_format</code>	<code>png</code>	<code>png</code> , <code>pdf</code> or <code>svg</code> .
<code>dpi</code>	<code>200</code>	Raster resolution for saved figures.

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.

9.9 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
<code>enable_symbolic_distillation</code>	<code>false</code>	Search for a closed-form expression after training.
<code>target</code>	<code>model</code>	<code>model</code> distils the operator's predictions; <code>reference</code> fits the DFT data.
<code>features</code>	<code>gga</code>	Variables given to the engine: <code>gga</code> (ρ, p, q), <code>enhancement</code> (p, q), or <code>raw</code> .
<code>epsilon</code>	<code>1e-8</code>	Vacuum threshold in e/a_0^3 ; clamps every denominator and drops voxels below it.
<code>unary_operations</code>	<code>[exp, log, sqrt, abs]</code>	Unary alphabet, passed to the engine untouched.
<code>binary_operations</code>	<code>['+', '-', '*', '/', '^]</code>	Binary alphabet.
<code>iterations</code>	<code>40</code>	Search iterations — the main quality/time knob.
<code>population_size</code>	<code>33</code>	Individuals per population.
<code>populations</code>	<code>15</code>	Populations run in parallel.
<code>max_depth</code>	<code>10</code>	Ceiling on expression depth.
<code>max_size</code>	<code>30</code>	Ceiling on node count.
<code>parsimony</code>	<code>0.0032</code>	Penalty per node; higher gives shorter, worse-fitting expressions.
<code>n_samples</code>	<code>4000</code>	Voxels sampled across the dataset.
<code>seed</code>	<code>0</code>	Seeds the sampling and the engine.
<code>template</code>	<code>none</code>	How the target is factorised: <code>none</code> , <code>thomas_fermi</code> or <code>pauli</code> .
<code>data_loss</code>	<code>mse</code>	The unpenalised part of the objective; <code>mae</code> is more robust on a density whose tails span orders of magnitude.

symbolic.physics — constraints on the search

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

Two blocks named physics, and they are not the same

`training.physics` 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.

`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.

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.

9.10 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}
```

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: $2.2\times$ at 32^3 , $5.7\times$ 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.