



User Guide

Predicting charge and kinetic energy densities
from crystal geometry

Leandro Seixas

Version 26.6.4

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 and opens directly in VESTA.

Who this guide is for

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

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

What you need

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

Caveat

Poraquê learns from the data you give it. The published results were obtained on twelve gold 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 Python version	4
1.2 Checking the installation	4
1.3 Optional components	4
2 Preparing data	5
2.1 Directory layout	5
2.2 Grids may differ between materials	5
2.3 Checking your data	5
3 Training	6
3.1 The short version	6
3.2 One model, all structures	6
3.3 Measuring generalisation	7
3.4 Reading the output	7
3.5 Reference numbers	7
3.6 Options worth knowing	8
4 Predicting a new structure	9
4.1 Choosing the grid	9
4.2 Comparing against a reference	9
4.3 Sanity checks the script performs	10
4.4 Visualising	10
5 Energies and the ASE calculator	11
5.1 The energy expression	11
5.2 Choosing the exchange-correlation functional	12
5.3 Computing energies from files	12
5.4 The ASE calculator	13
5.5 What the number is not	15
6 Configuration reference	16
6.1 How a value is resolved	16
6.2 Top level	16
6.3 <code>data</code> — where the fields come from	17
6.4 <code>model</code> — the operator’s shape	18
6.5 <code>training</code> — how it is fitted	20
6.6 <code>output</code> — what is written	22
6.7 Worked examples	23
7 Troubleshooting	24
7.1 The run stops immediately	24

7.2	The results look wrong	24
7.3	Grids and shapes	24
7.4	Devices	25
7.5	Reports	25

CHAPTER 1

Installation

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

This installs NumPy, SciPy, ASE, Matplotlib, PyYAML and PyTorch.

1.1 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.

1.2 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`.

1.3 Optional components

Component	Needed for	Install
CUDA build of PyTorch	NVIDIA GPUs	https://pytorch.org
pdflatex / latexmk	automatic PDF reports	TeX Live or MacTeX

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

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

```
python scripts/validate_vasp_data.py --fit-sigma --form-factor
```

This reads every structure, recomputes the external potential, compares it against a reference EXTCAR if one is present, and runs integrity checks on CHGCAR and TAUCAR — grid agreement, finiteness, sign, and the integrated totals. The electron count is the sharpest check: it should return the exact integer $\sum_a Z_a^{\text{val}}$.

CHAPTER 3

Training

3.1 The short version

```
python scripts/run_train.py --write-config configs/train_config.yaml
python scripts/run_train.py --config configs/train_config.yaml
```

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

Path	Contents
models/poraque_models.pth	both trained operators, in one file
logs/*.log, logs/*.json	metrics and full history
logs/*_config.yaml	the resolved configuration
results/plots/	loss curves, cross-sections, parity plots
reports/*.pdf	a typeset report per model

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 $4\times$ 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

```
python scripts/run_train.py --config configs/train_config.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.

```
python scripts/run_train.py --config configs/train_config.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

Twelve gold supercells — ten 27-atom and two 32-atom, four grid shapes — at 32^3 working resolution, 200 epochs:

Model	5-fold CV	all structures (training fit)
<code>ext2chg</code>	0.0231 ± 0.0127	0.0059
<code>chg2tau</code>	0.0469 ± 0.0273	0.0128

Quote the cross-validated column. The gap to the training fit is the memorisation margin — roughly four-fold here.

Caveat

The cross-validated spread is dominated by *cell size*, not by geometry. Split by supercell:

Subset	ext2chg	chg2tau
27-atom (10 structures)	0.0189 ± 0.0047	0.0379 ± 0.0077
32-atom (2 structures)	0.0441 ± 0.0180	0.0920 ± 0.0415

A held-out 32-atom cell is about $2.4\times$ harder. With only two examples of that size, holding one out leaves a single sibling, so the operator is extrapolating to a grid shape it has barely seen. Expect the same whenever a new cell size enters the dataset — and read an aggregate that mixes cell sizes with that in mind.

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.

Predicting a new structure

```
python scripts/run_eval.py 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 is chosen to land near the grid the shipped models were trained on. On the reference cells it gives 42^3 (27-atom) and 40^3 (32-atom), against a training `resolution` of 32 — close enough that the operator is interpolating rather than extrapolating. Raising it to 400 eV would give 60^3 , a substantially finer mesh than anything the models have seen.

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.

4.2 Comparing against a reference

```
python scripts/run_eval.py 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.

4.3 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.

4.4 Visualising

All outputs are CHGCAR-format and open directly in VESTA. 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

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

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, $v_H(\mathbf{G}=0) = 0$
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. Poraque 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 Au 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.pt", "chg2tau.pt", potcar_dir=..., functional="lda")
```

```
python scripts/run_eval.py 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 Au 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("Au", "fcc", a=4.08, cubic=True)
atoms.calc = Poraquê("models/poraque_models.pth", potcar="POTCAR")

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")` and opened in VESTA.

Argument	Meaning
<code>ext2chg, chg2tau</code>	Checkpoint paths, or loaded <code>FieldOperators</code> .
<code>potcar</code>	A single POTCAR covering the species present.
<code>potcar_dir</code>	A POTCAR <i>library</i> , searched per composition.
<code>charges</code>	{ <code>element: Z_val</code> }, used only without a POTCAR.
<code>resolution</code>	Longest grid axis; defaults to the checkpoint's training resolution.
<code>functional</code>	Exchange-correlation approximation, default "pbe".
<code>device</code>	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.pth",
                    potcar_dir="/opt/vasp/otpaw_PBE")
```

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

1. `<dir>/Au/POTCAR` — what VASP ships;
2. `<dir>/Au_pv/POTCAR` — a variant, used only if it is the *only* candidate, with a warning;
3. `<dir>/POTCAR.Au` or `<dir>/Au.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()` and `get_stress()` raise `NotImplementedError`, so **geometry optimisation and molecular dynamics are unavailable**. Single points, energy-volume scans and ranking of fixed geometries work.

Two independent pieces are missing: the derivative of V_{ext} with respect to the ionic positions (analytic, a Hellmann–Feynman term), and back-propagation of $\partial E / \partial \rho$ through both operators. A finite-difference stand-in is no shortcut here — on a fixed grid it would be dominated by the grid's own discontinuity as atoms cross voxel boundaries.

`implemented_properties` is `["energy", "free_energy"]`. The two are the same number: no

electronic smearing enters this pipeline, and ASE optimizers request `free_energy` by name.

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 Au 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 twelve reference Au supercells, with the models evaluated on their own training data:

Quantity	Value
True spread of E across the twelve structures	0.27 eV/atom
MAE of the predicted differences	0.29 eV/atom
Ratio error / signal	1.06
Correlation of predicted vs. true differences	$r \approx -0.1$

The error is **comparable to the signal** and the correlation is consistent with zero, so the predicted ordering of structures carries no information. That is an improvement on the $3\times$ ratio measured on the earlier, smaller dataset, but the verdict is unchanged.

The cause is cancellation, not a bug: the total is a sum of terms of order 10^4 eV whose physically relevant variation is a fraction of an eV per atom, a relative $\sim 10^{-4}$. A field-level relative L^2 of 2×10^{-2} cannot survive that. Reaching 0.01 eV/atom would need densities accurate to roughly 10^{-5} relative, three orders of magnitude beyond the current models.

Note

The grid is *not* the limiting factor. On reference fields the total energy changes by 0.08 eV out of 31215 (0.003 eV/atom) 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.

Rescaling the predicted density to the known electron count does **not** help: E_{ext} is linear in ρ , so correcting a 0.3% electron-count error shifts a -12690 eV term by about 38 eV. It was measured and rejected, not assumed.

CHAPTER 6

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

```
python scripts/run_train.py --write-config configs/train_config.yaml
```

This chapter documents every key that file can contain.

6.1 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:

```
python scripts/run_train.py --config configs/train_config.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 `logs/fno_training_config.yaml`). That is the file worth keeping: it records what actually ran, overrides included.

6.2 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.

6.3 data — where the fields come from

Key	Default	Meaning
root	data/vasp	Directory holding one subdirectory per material.
cache pattern	data/cache struct	Where downsampled copies are written. Prefix identifying those subdirectories — a prefix, not a glob.
code	auto	DFT code, or <code>auto</code> to detect it from the files present.
resolution	32	Longest grid axis after spectral downsampling.
gaussian_blur	null	Gaussian blur width in Å applied to the computed potential.
blur_method	spectral	<code>spectral</code> or <code>ndimage</code> .

6.3.1 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.

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

Implementation

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

6.4 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.

6.4.1 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.

6.4.2 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.

6.4.3 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`.

6.4.4 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.

6.5 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>	50	Stop after this many epochs without validation improvement; 0 disables.
<code>epochs</code>	200	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.

6.5.1 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.

6.5.2 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
    -----
      10/200      0.31745      0.34118
      20/200      0.18902      0.21447 *
```

6.5.3 early_stopping

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

```
      30/200      0.03173      0.03659 *
      ...
      38/200      0.02249      0.06568
  stopped early at epoch 38: no improvement in 8 epochs
                           (best 0.03659 at epoch 30)

  trained 38/200 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.

With the shipped default `valid_fraction: 0.0`, early stopping is therefore inactive. Set a split to use it.

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.

6.5.4 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.

6.5.5 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.

6.5.6 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.

6.6 output — what is written

Key	Default	Meaning
log	<code>logs/fno_training.log</code>	Human-readable training log.
json	<code>logs/fno_training.json</code>	Metrics and full loss history.
checkpoint_dir	<code>models</code>	Model weights; <code>null</code> disables checkpointing.
plot_dir	<code>results/plots</code>	Figures; <code>null</code> disables plotting.
report_dir	<code>reports</code>	PDF reports; <code>null</code> disables them.
plot_format	<code>png</code>	<code>png</code> , <code>pdf</code> or <code>svg</code> .
dpi	<code>160</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.

6.7 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

7.1 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; regenerate the file with `-write-config`. 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 Au=11`.

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

7.2 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 `results/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.

7.3 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.

7.4 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.

7.5 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.