



Poraquê

User Guide

Predicting charge and kinetic energy densities
from atomistic geometry

Leandro Seixas

Version 26.10.7

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
latexmk (optional)	only for the automatic PDF reports

Caveat

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

Contents

Overview	1
1 Installation	4
1.1 From PyPI	4
1.2 From source	4
1.3 Console commands	4
1.4 Python version	5
1.5 Checking the installation	5
1.6 NVIDIA GPUs	6
1.7 Optional C kernels	9
1.8 Optional components	10
2 Preparing data	11
2.1 Directory layout	11
2.2 Grids may differ between materials	11
2.3 Checking your data	11
2.4 Downloading from the Materials Project	12
2.5 The whole database, not a chemical space	13
2.6 What an MP download does and does not contain	15
2.7 The external potential without a POTCAR	15
2.8 Training on several sources at once	16
3 Training	18
3.1 The short version	18
3.2 One model, all structures	19
3.3 Measuring generalization	20
3.4 Reading the output	20
3.5 Reference numbers	21
3.6 Options worth knowing	21
4 Predicting a new structure	22
4.1 Choosing the grid	22
4.2 Matching a VASP grid	22
4.3 Comparing against a reference	24
4.4 Restarting VASP from a predicted density	24
4.5 Sanity checks the script performs	26
4.6 Visualizing	26
5 Energies and the ASE calculator	27
5.1 The energy expression	27
5.2 Choosing the exchange-correlation functional	28
5.3 Computing energies from files	28
5.4 The ASE calculator	29
5.5 What the number is not	32

6	Charges and population analysis	34
6.1	Charge conservation	34
6.2	Partial charges	35
6.3	What these charges are not	37
6.4	In the reports	37
7	Fine-tuning	38
7.1	What comes from the checkpoint	38
7.2	Freezing the lifting path	39
7.3	Configuration	40
7.4	LoRA — adapting a model that does not fit	40
7.5	Output, and the .poraquê format	41
8	Configuration reference	43
8.1	How a value is resolved	43
8.2	Operators and blocks: presence is the switch	44
8.3	What is fixed	45
8.4	run — what the run is called	45
8.5	data — where the fields come from	46
8.6	ext2chg — the density operator	51
8.7	model — the network of each operator	52
8.8	ext2chg.occupancies — the per-atom PAW head	55
8.9	chg2tau — the kinetic operator	57
8.10	training — how it is fitted	59
8.11	fine_tuning — start from a trained checkpoint	64
8.12	runtime — the hardware	65
8.13	report — what is written	69
8.14	Worked examples	72
9	Troubleshooting	74
9.1	The run stops immediately	74
9.2	The results look wrong	74
9.3	Grids and shapes	75
9.4	Devices	75
9.5	Reports	75

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.

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 ; and chgcar , which writes a field store out as a CHGCAR

The last two are not the same command twice

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

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

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

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

Note

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

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

1.4 Python version

Note

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

1.5 Checking the installation

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

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

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

1.6 NVIDIA GPU_s

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

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

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

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

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

1.6.1 Check before submitting to a queue

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

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

`-device` defaults to `auto`, and `-check` refuses a CPU under `auto` too — a guard that cannot fail is not a guard, and this one could not: it resolved to the CPU and exited 0 on any machine with a working CPU, which is every machine. Naming the device the scheduler was actually asked for is still the clearer form, because it is the one that also catches `cuda:2` on a node where the job was given one GPU.

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

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

Note

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

1.6.2 ~/.local outranks the active environment

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

```
export PYTHONNOUSERSITE=1
```

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

1.6.3 Several GPUs, under Slurm

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

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

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

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

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

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

```
ranks : nccl rank 0/4 local_rank 0 on sdumont1234:24601 (launched by slurm)
        SLURM_JOB_ID = 4601
        SLURM_NTASKS = 4
        effective batch = 10 x 4 ranks = 40
```

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

Three things to know before reading a scaling number. The **effective batch size is `batch_size` times the world size**, so a four-rank run is not the same optimizer as the one-rank run

it is compared against. **Rank 0 alone writes** the checkpoint, the metrics, the figures and the PDF, and alone prints. And **loader_workers gets worse, not better**: each rank is already a process with a field cache of its own. One rank per GPU, cache on, no workers — see `data.cache.in_memory` (section 8.5.6), which is worth more here than a second GPU would be, and section 8.12.7 for the two keys themselves.

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

Note

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

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

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

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

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

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

```
[CACHE SUMMARY]
status      : built; no training was run (--cache-only)
cache       : data/cache/res48_potcar
storage     : files (one CHGCAR-format text file per field)
materials   : 97 cached (ext2pseudo: 97, pseudo2tau: 97)
grid shapes : 24x24x48      11 structures
              40x28x48      10 structures
              40x40x48       9 structures
              48x48x48      64 structures
              ...
elements    : Pt
on disk     : 458.9 MiB
decoded     : ~134.2 MiB in RAM per task (data.cache.in_memory: True, budget 4.0
↪ GiB)
```

Three things about the mode are worth knowing. **It forces the CPU and forms no group**,

whatever the config says: `runtime.device: cuda, strict_device: true` and `distributed: auto` can stay in the file for the GPU job, and the CPU job overrides all three for its own process while touching nothing on disk but the cache. **The GPU job rebuilds nothing** as long as its config keeps the cache fingerprint — `data.grid.resolution`, `data.grid.max_spacing`, `data.cache.storage`, `data.cache.compression`, `data.spin`, `data.potcar_dir` and the data path — as it was; its cache table then reads *cached* on every row and training starts at once. Change one of those keys and the GPU job builds a *different* cache, on the GPU node, which is what the split exists to avoid. And **read the decoded line before requesting memory**: it is the size of the fields `data.cache.in_memory` holds in RAM (section 8.5.6), and the number `auto` compares against its budget; a GPU job whose request is below it either declines the cache, at ten times the epoch time, or is killed.

One task, not four: the builder is one task with several processes (`runtime.workers: auto` reads `-cpus-per-task`), and several tasks would each build the whole cache. Re-running `-cache-only` on a cache that already exists is cheap — present materials are left alone — so an interrupted CPU job is resubmitted, not restarted.

1.7 Optional C kernels

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`. A second kernel, the text codec of volumetric grid blocks, makes dataset cache builds 3.2–3.4× faster and is built the same way. Installation compiles nothing. To compile both now, check each against its Python reference and see what loaded — in a cluster setup step, say, so that no job pays for the compile — run:

```
poraque-compile
```

Its options are `-rebuild` (recompile even if cached), `-clean` (delete cached libraries no current kernel loads) and `-only spectral` or `-only codec`. The exit status is 0 when every requested kernel loaded and 1 otherwise; everything still runs without them. `PORAQUE_C_BACKEND=0` disables both, and `PORAQUE_CACHE_DIR` moves the cache. The codec uses portable compiler flags only, so a library built on a login node cannot crash a compute node with a different CPU; the spectral kernel still tries `-march=native` first, so build it on the node type that will run inference. The spectral kernel alone, with a benchmark:

```
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.

1.8 Optional components

Component	Needed for	Install
CUDA build of PyTorch	NVIDIA GPUs	§1.6
<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.

CHAPTER 2

Preparing data

2.1 Directory layout

One directory per material:

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

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

Implementation

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

2.2 Grids may differ between materials

They usually do: ENCUT and the cell shape fix 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.grid.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.grid.resolution` — Gibbs ringing around the core peaks, which is an artifact of the truncation rather than a fault in the data.

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

2.4 Downloading from the Materials Project

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

Size it before committing to the transfer. Over a chemical space the estimate is exact rather than modeled: 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 `chgcар_estimate.csv`, not even a directory. Asking how big something would be should not create anything.

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

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

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

```
data/MP/
  summary.csv manifest.json manifest.csv
  structures/mp-124.cif
  mp-124/ CHGCAR.gz INCAR KPOINTS POSCAR mp.json
```

```
mp-81/  CHGCAR.gz  INCAR  KPOINTS  POSCAR  mp.json
mp-126/ fields.h5  ...                # with --hdf5
```

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

Implementation

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

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

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

Implementation

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

2.5 The whole database, not a chemical space

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

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

Caveat

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

multi-terabyte transfer.

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

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

`-hdf5` keeps everything, and VASP still cannot read it

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

Caveat

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

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

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

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

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

A sample is a selection, not just a sizing trick

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

Caveat

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

Sizing changes with it. Over a chemical space every object is **HEAded** and the total is exact; over the database that is tens of thousands of requests for a number nobody needs to four digits. A sampled estimate measures its N exactly — so **TOTAL DOWNLOAD** is a measurement of what a fetch would actually transfer, not an extrapolation — and *also* projects to the whole set, as the sample mean times the number advertised, on its own line and labeled as a projection. Both are

printed because they answer different questions, *what will this fetch* and *what would all of it cost*, and a report stating one of them gets read as the other.

`-seed` decides *which* subset, and that is what makes a sampled download resumable: the same seed selects the same materials, so an interrupted run resumes onto the same subset rather than fetching N more beside it. Sizes are also strongly right-tailed, so two draws of twenty can differ by tens of percent, and a projection that moved every time it was asked could not be planned against.

2.6 What an MP download does and does not contain

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

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

The valence charges are recoverable. A pseudopotential **CHG**CAR 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 the kinetic operator cannot be trained on MP data at all: write no `chg2tau:` section. Nor do MP downloads carry a **POTCAR**, so the density operator trains on the grid alone, with `occupancies: false` in its `ext2chg:` section; the per-atom PAW records need the pseudopotential to lay them out.

2.7 The external potential without a POTCAR

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

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

```
data:
  path: data/MP
  potcar_dir: /opt/vasp/potpaw_PBE      # <dir>/Ag/POTCAR, <dir>/Pt/POTCAR, ...
  grid:
    resolution: 32
  ext2chg:
    occupancies: false
```

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

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

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

Caveat

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

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

2.7.1 What the cache records

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

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

The fingerprint now carries `potcar_source`, per element:

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

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

(A `data.strict_potcar` key that turned the fallback into an error was removed on 2026-10-13: what the library served is recorded in the cache either way, and the run log names it.)

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.

2.8 Training on several sources at once

`data.path` is **one directory**: subdirectories, one per material, each holding that material’s volumetric files. A VASP run tree (`structure_0000/`, ...), a download (`mp-124/`, ...) and a cache from an earlier run are all this shape, and nothing in the config says which is which,

because on disk they are the same kind of thing. To train across several, gather the material directories under one parent (symbolic links do):

```
data:
  path: data/pooled          # structure_0000/, mp-124/, res32_0001/, ...
  potcar_dir: /opt/vasp/potpaw_PBE
```

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

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

Caveat

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

Training

3.1 The short version

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

An operator is trained when its section is present in the configuration, and this file has two: it 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
<code>models/<name>/<name>.poraque</code>	both trained operators, in one file
<code>models/<name>/log/</code>	training log, metrics, resolved configuration
<code>models/<name>/plots/</code>	loss curves, cross-sections, parity plots
<code>models/<name>/report/</code>	the typeset PDF report

3.1.1 What the report's performance table says

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

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

Implementation

The **train** and **validation** rows read against each other *are* the generalization gap. That is the single most useful thing the table says, and it is why the splits are rows.

Note

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

For a two-channel model every number is the **density** channel; the magnetization is measured separately, as `magnetization_relative_l2` in the metrics JSON, and the report says so in its caveats.

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

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

3.1.2 What the parity figure shows

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

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

3.2 One model, all structures

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

Note

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

Caveat

Set `validation.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 artifact you ship, since it uses all the data — just quote a cross-validated number alongside it.

3.3 Measuring generalization

Splitting is always at the *structure* level.

3.3.1 The default split

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

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

3.3.2 K-fold cross-validation

This is the only variation on the protocol; `validation.fraction` is ignored, since the folds define the splits. It is not yet available to an operator with its occupancies on, so a cross-validated run says `occupancies: false` in its operator sections.

```
poraque-train --config configs/train_simple.yaml \
  --kfold 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

The number of folds (`validation.kfold`, or the argument of `-kfold`; a bare `-kfold` means five) 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 the kinetic operator 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>ext2pseudo</code>	0.0379 ± 0.0027	0.0209
<code>pseudo2tau</code>	0.0511 ± 0.0031	0.0140

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

Caveat

Read the held-out column knowing what is in it. At `validation.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 (`ext2pseudo`) 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

chg2tau.head: pauli For the kinetic operator, predicts $\tau = \tau_{\text{vW}}[\rho] + s \text{softplus}(f)$ so the exact bound $\tau \geq \tau_{\text{vW}}$ holds by construction. Improves accuracy and trains slightly faster; recommended. For now it needs **occupancies: false** beside it. It is a setting of the file, not a flag.

-task Trains only one of the operator sections the file describes (`ext2chg` or `chg2tau`).

-seed Seeds the validation draw, the folds, the batch order and the initial weights; vary it across runs to build a committee.

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

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

CHAPTER 4

Predicting a new structure

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

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

4.1 Choosing the grid

Resolved in this order, first hit wins:

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

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

Implementation

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

4.2 Matching a VASP grid

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

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

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

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

Caveat

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

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

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

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

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

The sizing above is how a plane-wave code *should* choose a grid. It is not what VASP does, and for a restart that difference matters: the CHGCAR declares its own 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-center 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-neighbor* — 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 checkpoint's `paw_profiles`, beside that element's core density:

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

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

Caveat

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

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

4.4.2 And without either: the isolated atom

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

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

1. a reference calculation beside the structure — exact, because the records are that system's own;
2. an `ext2chg` operator in the bundle (§8.8) — this structure's own records, predicted from its potential;
3. the bundle's per-element average — 28 % relative RMS on the platinum set of bulk, slabs and nanoparticles, and only for the elements the model was trained on;
4. the isolated atom — ~86.6 % RMS, on explicit request only.

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

4.4.3 Checks

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

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

Note

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

restart, but a density that far from normalized 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 Visualizing

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

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

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. Poraqu  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("ext2pseudo.poraque", "pseudo2tau.poraque", potcar_dir=...,
    ↪ functional="lda")
```

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

5.3 Computing energies from files

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

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

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

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

kinetic	T_s	6207.068300 eV
external	E_ext	-12634.744762 eV
alpha Z		1752.512988 eV
Hartree	E_H	3230.363304 eV
xc (pbe)		-3845.824650 eV
Ewald		-25949.345827 eV

potential		-37447.038948 eV
TOTAL		-31239.970647 eV
electrons		297.000001

Implementation

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

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

5.4 The ASE calculator

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.poraquê", 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
<code>ext2pseudo</code> , <code>pseudo2tau</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.poraque",
                    potcar_dir="/opt/vasp/potpaw_PBE")
```

Recognized 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-center 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 optimization 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-center 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.poraque",
                    potcar_dir="POTCARs",
                    references="data/vasp/ref")

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

5.4.3.1 Which reference to subtract

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

method	E_{iso} from	Pt cohesive energy
"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 neutralizing-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-center 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-center 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 normalization helps, and is on by default. The shipped `ext2pseudo` 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 `ext2pseudo` 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 normalization

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

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

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

With normalization 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.poraque", potcar_dir="POTCARs",
                    references="data/vasp/ref")

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

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

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

6.2.2 Voronoi

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

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, $\rho(r) = \frac{N}{8\pi a^3} e^{-r/a}$, normalized 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 vectorized on-grid steepest ascent: every voxel points at whichever of its 26 neighbors 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 neighbors, $\sqrt{3}$ 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 pseudization 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="ext2pseudo", 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 specialization is learned.

Fine-tuning is switched on by the *presence* of a `fine_tuning:` section in the configuration. The flags `-pretrained` and `-fine-tune-lr` override the section's values; they do not create it, and without the section they are refused. It is not yet available to an operator with its PAW occupancies on (a pretrained bundle has no occupancy layer to adapt), so the sections below say `occupancies: false`.

```
# base.yaml: the model trained across everything you have
run:
  name: general
ext2chg:
  occupancies: false
chg2tau:
  occupancies: false
```

```
# oxides.yaml: specialize it on one family
run:
  name: oxides
data:
  path: data/oxides
ext2chg:
  occupancies: false
chg2tau:
  occupancies: false
fine_tuning:
  checkpoint: models/general/general.poraque
```

```
poraque-train --config base.yaml
poraque-train --config oxides.yaml
poraque-train --config oxides.yaml --fine-tune-lr 5e-6
```

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 specializing — 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: true
```

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

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

Implementation

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

7.3 Configuration

Key	Default	Meaning
<code>fine_tuning</code>	<code>off</code>	The section's presence starts from <code>checkpoint</code> instead of a fresh initialization; <code>false</code> turns it off, <code>true</code> takes every default.
<code>checkpoint</code>	<code>models/poraque_models_base</code>	Base bundle to adapt.
<code>learning_rate</code>	<code>1.0e-05</code>	Replaces <code>training.optimizer.learning_rate</code> for this run.
<code>freeze_lifting</code>	<code>false</code>	Hold the input lifting path fixed. Ignored under <code>lora</code> .
<code>lora</code>	<code>false</code>	A block: freeze everything and learn a low-rank correction; see below.
<code>lora.rank</code>	<code>8</code>	r ; cost and capacity are both linear in it.
<code>lora.alpha</code>	<code>16.0</code>	The update is scaled by <code>alpha / rank</code> .
<code>lora.dropout</code>	<code>0.0</code>	Dropout on the adapter's input.

7.4 LoRA — adapting a model that does not fit

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

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

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

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

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

Caveat

The spectral weights are not adapted. They hold $\sim 99.8\%$ of the parameters and look like the obvious target, but a rank- r factorization of a 5-index complex kernel is a *choice of which axes to pair* rather than one decomposition, and adapting them would also stop being cheap.

What LoRA says here is precise: *keep the learned operator, re-fit how fields enter and leave it*. A family whose operator genuinely differs from the base model's — not merely its input and output scales — is out of its reach, and the honest answer there is a full fine-tune. It buys memory, not generality.

Note

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

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

7.5 Output, and the .poraque format

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

File	Contents
<code>models/<name>/<name>.poraque</code>	The universal model: both operators in one file.
<code>models/<name>/<name>_finetuned.poraque</code>	A fine-tune, specialized to one family.

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

```
checkpoint = {
    "model_state_dict": {"ext2pseudo": ..., "pseudo2tau": ...},
    "config": {...},          # the resolved training config
    "paw_profiles": {78: {...}, ...}, # PAW data per atomic number
}
```

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

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

Caveat

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

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

A fine-tune is **never** written over the base model: it is `<name>_finetuned.poraque` in its own run 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 specialized model's scores for a general one's. The checkpoint's `config` records `fine_tuning.checkpoint`, the learning rate and whether the lifting layers were frozen, so a checkpoint can always be traced back to its parent.

Note

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

Configuration reference

A run is described by a YAML file of eight sections, each about one thing:

```
run:          # what the run is called, where it writes, its seed
data:         # where the materials are, and how they are cached
ext2chg:      # the density operator: V_ext -> pseudo-density + occupancies
chg2tau:      # the kinetic operator: pseudo-density + occupancies -> tau
training:     # the optimization, shared by both operators
fine_tuning:  # start from a trained checkpoint (off unless present)
runtime:      # the hardware: device, processes, distribution
report:       # figures and the PDF
```

Every key is optional: a file states only what the run does differently, and anything omitted takes its default. Two files in `configs/` are the starting points. `configs/train_simple.yaml` is the short one, the platinum set with both operators, forty-odd lines, only what the run decides. `configs/train_complete_and_commented.yaml` is the map: every key at its default, with a line saying what it means. This chapter documents every key either can contain, and the measurements behind the defaults that were measured.

8.1 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 one committed config be swept from the shell without being edited or copied:

```
poraque-train --config configs/train_simple.yaml --epochs 500 --seed 1
```

Each flag overrides one dotted path (`--epochs` is `training.epochs`, `--cache-workers` is `runtime.workers`); `poraque-train --help` lists them under the section they belong to.

Note

Unknown keys are **rejected**, not ignored. A typo such as `learn_rate` would otherwise be silently dropped 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 keys and sections of the schema this one replaced (2026-10-13) are rejected too, and each error names where its setting went: `training.valid_fraction` is now `training.validation.fraction`, `data.cache_workers` is `runtime.workers`, and so on.

The *resolved* configuration, with defaults, file and overrides merged, is written beside the results as `<name>_config.yaml`. That is the file worth keeping: it records what actually ran.

8.2 Operators and blocks: presence is the switch

An operator is trained when its section is present. `ext2chg:` and `chg2tau:` are the two; leave one out and it is not trained. A configuration with neither raises, since there is nothing to do.

```
ext2chg:          # trained
chg2tau:          # trained, with its defaults
```

Each operator carries its own `model`, because the two maps need different networks and a single shared block forced the larger one into a corner of the other.

A feature that can be switched off is a block whose presence is the switch. `false` turns it off, `true` (or the bare key) turns it on with the defaults, a mapping turns it on with those settings. There is no `enable` key anywhere:

```
ext2chg:
  model:
    equivariant: false      # off
```

```
ext2chg:
  model:
    equivariant: true       # on, with the defaults
```

```
ext2chg:
  model:
    equivariant:            # on, with these
    n_radial: 32
```

The old layout put a switch beside the settings it governed, and a block could be switched off while still listing settings that read as if they applied; with the presence as the switch, what the file says is what runs. Blocks are validated when the file is **read**, not when the feature is first used. That matters more than it sounds: the model is built after the field cache and the objective after that, so a one-line mistake in either would otherwise take minutes of downsampling to report.

The blocks are `ext2chg.structure_input`, `ext2chg.occupancies`, `chg2tau.occupancies`, `model.equivariant`, `training.physics`'s weights (zero is off), `fine_tuning` and `fine_tuning.lora`.

The two names are the vocabulary of configuration; four are the vocabulary of the code. `occupancies` is the switch between them. With it on (the default) `ext2chg` predicts the pseudo-density and every atom's PAW augmentation occupancies, and `chg2tau` reads them. With `occupancies: false` the operator trains on the grid alone, which internally is the `ext2pseudo`

task (`V_ext` to pseudo-density) or the `pseudo2tau` task (pseudo-density to τ). Those two names remain in logs, figure and file names and checkpoints, and `poraque-train --task` takes only the configuration's two. The grid-alone pair needs no POTCAR, and a bundle needs exactly that pair to run `poraque-inference` and the ASE calculator.

8.3 What is fixed

Some settings stopped being choices because one value won every measurement or the alternative never worked. They are not keys: the cosine schedule, a gradient-norm clip of 1, the weight initialization seeded by `run.seed`, the read-out activation following `model.activation`, the Pauli scale fitted on the training split and learned, an Euler-Lagrange weight of 0, fields held in float64, the isolated atom's PAW records for a prediction, gzip level 4, pinned memory on CUDA and not elsewhere, a 30 minute collective timeout, and an asinh scale fitted rather than fixed. Naming one in a file raises.

8.4 run — what the run is called

Key	Default	Meaning
<code>name</code>	<code>poraque_models</code>	the run's name; everything goes in <code><output>/<name>/</code>
<code>output</code>	<code>models</code>	parent directory; <code>null</code> writes nothing (no checkpoint, log or figure)
<code>seed</code>	<code>42</code>	the validation draw, the folds, the batch order and the initial weights

```
run:
  name: mp_ag_au_pt
  output: models
  seed: 42
```

Everything a run writes lives under `<output>/<name>/`: the weights (`<name>.poraque`), `log/` (the log, the metrics JSON and the resolved configuration), `plots/` and `report/`. `output` is the only path setting and `name` is the only thing that distinguishes two runs, so "delete this experiment" and "send me that model" are one directory each rather than a collection of fragments by filename.

A run that trains both operators writes two reports, `<name>_ext2pseudo_report.pdf` and `<name>_pseudo2tau_report.pdf` (or `ext2chg` and `chg2tau`, by whether the occupancies are on): one name cannot serve both. Cross-validation writes `<name>_kfold_report.pdf`, and a fine-tune writes its weights to `<name>_finetuned.poraque` rather than over the general model it specializes.

Caveat

Two runs that share a `name` share their output files, and the second wins. Change it for anything worth keeping beside the last run: a different chemical space, resolution, or set of physics weights.

`seed` is one number for everything that is random. Earlier versions had a second one for the initial weights, so that a committee of models could differ in initialization alone; now a different

seed also draws a different validation split and a different batch order, which is a stronger, and different, source of disagreement. The ensemble tools take it as it is (see `docs/notes/committee.md`).

8.5 data — where the fields come from

Key	Default	Meaning
<code>path</code>	<code>data/vasp/structures</code>	the dataset: one directory, one subdirectory per material — see below
<code>potcar_dir</code>	<code>null</code>	POTCAR library, used where a material ships none
<code>atoms</code>	<code>data/vasp/isolated_atoms</code>	isolated-atom reference: a <code>poraque-atoms</code> JSON, or a directory of one-atom-in-a-box runs
<code>spin</code>	<code>auto</code>	train on the magnetization as a second channel — see below (Section 8.5.5)
<code>grid.resolution</code>	<code>32</code>	longest grid axis after spectral downsampling
<code>grid.max_spacing</code>	<code>null</code>	largest grid spacing in Å after downsampling — see below (Section 8.5.3)
<code>cache.dir</code>	<code>data/cache</code>	where downsampled copies are written
<code>cache.storage</code>	<code>text</code>	<code>text</code> (a CHGCAR-format file per field) or <code>hdf5</code> (one <code>fields.h5</code> per material)
<code>cache.compression</code>	<code>null</code>	<code>null</code> , <code>gzip</code> or <code>lzf</code> ; <code>hdf5</code> only, and it raises on a text cache
<code>cache.in_memory</code>	<code>auto</code>	keep decoded fields in RAM between epochs — see below (Section 8.5.6)

```
data:
  path: data/vasp/structures
  potcar_dir: /opt/vasp/potpaw_PBE
  grid:
    resolution: 48
  cache:
    storage: hdf5
    compression: gzip
```

8.5.1 path

One key names the dataset: a directory of subdirectories, one per material, each holding that material’s volumetric files.

```
data/vasp/structures/
  structure_0000/      CHGCAR  TAUCAR  POSCAR  POTCAR  INCAR ...
  structure_0001/
  ...
```

That is what a VASP run tree looks like, what `poraque-mp` writes, and what the cache builder produces, so nothing has to be declared about where a path came from. A material is a directory with a density in it: a CHGCAR, compressed or not, or a `fields.h5`. To pool several sources, gather them under one directory (symbolic links do); the set is the directory’s content.

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

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

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

TAUCAR is **optional, per material**. A directory that has one joins the kinetic operator’s set; one that does not still joins the density operator’s and is simply absent from the other. Nothing is declared for it, and an operator with no target anywhere is skipped with a message rather than failing the run, which is what makes both sections sensible on a Materials Project download, where no τ exists.

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

See Chapter 2 for what each kind of directory contains and for training across a mixture of them.

8.5.2 `grid.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.

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

Note

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

8.5.3 `grid.max_spacing`

resolution caps the **longest** axis, and the others are reduced in proportion. That is right for a compact cell and wrong for a long, thin one. On the CHONPS set a $3 \times 4 \times 35$ Å cell kept 6 and 8 points on its short axes, 0.6 Å spacing, where the median cell had 0.2 Å, and such cells were the worst-predicted of the validation split by a factor of five. **max_spacing** is the second bound, in Å: an axis that **resolution** would leave coarser than it is refined to the smallest FFT-friendly count that meets it, but **never past its native grid**, since a truncation cannot invent coefficients the calculation never had. **null**, the default, is **resolution** alone.

The value is part of the cache tag (`res32_h0.25_potcar`) and of the fingerprint, so a cache built with one bound is never read under another, and like **resolution** it belongs to the set of keys a `--cache-only` job and the GPU job after it must agree on.

8.5.4 potcar_dir and the Gaussian fallback

The external potential needs pseudopotentials, and not every dataset ships them. A Materials Project download has a structure and a density and nothing else; archived local runs often have their POTCARs stripped for licensing. `potcar_dir` points at a **library**, one subdirectory per species, as VASP distributes them:

```
data:
  potcar_dir: /opt/vasp/potpaw_PBE      # <dir>/Ag/POTCAR, <dir>/Pt/POTCAR, ...
```

.gz and .Z entries are read directly, and a flat POTCAR.Ag / Ag.POTCAR layout is recognized too. A run that has its own POTCAR ignores the library entirely: it is a fallback, not an override.

Which construction results is the **most consequential single property of a training set**, because the two are different physical quantities rather than different approximations to one:

	V_{ext}	residual vs. a reference EXTCAR
pseudopotentials available	tabulated local pseudopotential (VASP's POTION)	2×10^{-5} relative L_2
none available	Gaussian pseudo-ion model, $f_s(G) = e^{-G^2\sigma^2/2}$	order 10^{-1}

Measured on the Ag–Pt–Pt Materials Project set, the two constructions differ from each other by **0.38 relative L_2** .

Without the library the map learned is *model potential* $\rightarrow$ *DFT density*. That is well-posed and self-consistent, since inference builds the very same potential, but it is not VASP's V_{ext} , and a model trained that way is not comparable with one trained on tabulated potentials. It also means a mixed dataset spanning both constructions is training on two quantities under one name; `MixedFieldDataset` warns when that happens, and setting `potcar_dir` is what makes the warning go away *correctly*.

Caveat

Point at 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, which is 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 that contain it, so a library covering four elements of five still buys the exact potential for everything that does not involve the fifth. The width is derived per species from the pseudopotential core radius where one is known and is 0.5 Å otherwise (an MP download supplies no RCORE).

The cache records which construction it holds. A library that cannot be read falls back per element, and what was wrong until 26.9.3 is that nothing downstream recorded it. A control run on Santos Dumont landed on a node where the library's filesystem was not mounted, warned six times, trained against analytic pseudo-ion potentials, and wrote a fingerprint byte-identical to one built from real pseudopotentials. Every later run reused it silently, including on nodes where the library was mounted. The fingerprint now carries `potcar_source`, per element:

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

so a cache built during an outage no longer matches a run that can read the library; it rebuilds rather than being reused. 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 confused for each other.

Caveat

The occupancies need the POTCAR. The layout of each element's PAW record (which projector channels, so how many values) is read from the pseudopotential. With `ext2chg.occupancies` on, the default, and no POTCAR anywhere, the run stops before training and says to give the data one. A dataset without them, such as a Materials Project download, trains the grid alone: write `occupancies: false` in the operator's section.

8.5.5 spin

Value	What happens
auto	ask the sources whether their CHGCARs carry a second grid block, and settle the channel count for the whole dataset before the cache is written
true	demand one: a dataset with no magnetization raises
false	drop it deliberately, and say so in the log

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

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

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

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

Both parts are in the metrics JSON's `history`, as `train_loss_charge` and `train_loss_magnetization`, and exist only for a two-channel density target; a one-channel run prints neither column. The magnetization gets its own parity figure, `<task>_parity_magnetization`, on linear axes because m changes sign, and its own held-out number, `magnetization_relative_12`; the report's performance table stays on the density channel. How the two channels enter the

objective is `training.spin` (Section 8.10.5).

8.5.6 `cache.in_memory`

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

With it off, **every epoch reopens, decompresses and re-parses every field from disk**. That produces exactly the same numbers as caching does, and takes about ten times as long. Measured on a V100 with 115 structures at 32³: `__getitem__` was 59 % of the training loop, 78.9 s over 483 calls, while the GPU sat at 2–4 % utilization waiting on `zlib` and `numpy.array` over a text generator. Turning the cache on took twenty epochs from **168.0 s to 16.3 s**, with the validation error identical to five decimals.

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

The cache keeps the normalized sample, not only the field. Removing the parse left the transforms: every epoch still ran the input and target normalizations (an `asinh` over every voxel of every channel) on the CPU, and on a fast GPU that, not the operator, was the training loop. Measured on an RTX 5060 Ti with the CHONPS set at 48³, the GPU sat at about 2 % utilization with every decoded field already in RAM; memoizing the normalized samples took an epoch from ~60 s to ~8 s and the GPU to ~99 %, for the same arithmetic. The memo cannot go stale, because a dataset’s transforms are properties: assigning either one empties it, so no sample normalized by a superseded transform survives the change.

`auto` estimates what the cache will hold (grid points $\times$ itemsize $\times$ channels over the input and target fields, three times over: the decoded fields, the tensors made from them, and the normalized samples), plus the atomic baseline in δ -density mode, and enables the cache below 4 GiB, logging what it decided and what it costs:

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

Set it to `false` when the process shares its memory with something the estimate cannot see. What `auto` refuses on its own is the case the old default was right about: 128³ grids over thousands of materials, where caching silently would turn a slow run into a dead one.

Parallel loading is not the alternative it looks like. See `runtime.loader_workers` (Section 8.12.3).

8.5.7 `cache.storage` and `cache.compression`

`text` writes each field as a CHGCAR-format file, readable by anything that reads VASP; `hdf5` writes one chunked `fields.h5` per material, 1.38 $\times$ smaller at `gzip` and exact to 10⁻¹⁶ relative (text round-trips to 10⁻¹¹). HDF5 stores carry the magnetization block and the PAW records, but VASP cannot open one; `poraquê-vasp chgcar` writes it back to text. `compression` applies to HDF5 only (`gzip` at level 4, or `lzf`, with byte-shuffle on) and raises on a text cache. Both are part of the cache fingerprint.

8.5.8 `atoms`

The isolated-atom reference, as a `poraquê-atoms` JSON or as a directory of one-atom-in-a-box runs. It supplies three things: the baseline ρ_{sup} of `ext2chg.target: delta`, the structure channels of `structure_input` (Section 8.6.2), and the PAW records a prediction is written with. An element the library does not cover fails before the first epoch. The shipped library covers platinum only.

8.6 ext2chg — the density operator

$V_{\text{ext}} \rightarrow \tilde{\rho}$ and, with the occupancies on, every atom’s augmentation occupancies. Present: trained.

Key	Default	Meaning
<code>target</code>	<code>delta</code>	<code>delta</code> learns $\rho - \rho_{\text{sup}}$ and adds ρ_{sup} back on output; <code>full</code> learns ρ
<code>structure_input</code>	<code>false</code>	channels describing the structure, appended after V_{ext} — see below (Section 8.6.2)
<code>model</code>	width 64, layers 4, modes 16	the network — see below (Section 8.7)
<code>occupancies</code>	<code>on</code>	the per-atom PAW head — see below (Section 8.8); <code>false</code> trains the grid alone

8.6.1 target

`delta`, the default, trains on the residual over a superposition of isolated atoms ($\delta\rho = \rho - \rho_{\text{sup}}$; the residual is 0.057 in L^2 on the Pt cells). The operator still returns the **full** density: prediction adds ρ_{sup} back, in a fixed order (decode, add the baseline, clip, normalize), and every physics term sees the absolute density. The atomic library is copied into the checkpoint. It needs `data.atoms` (Section 8.5.4) to cover every element. `full` trains on ρ itself and needs no library.

8.6.2 structure_input

A block, off by default:

```
ext2chg:
  structure_input:
    superposition: true      # rho_sup, the delta-density baseline, as a channel
    elements: auto          # one superposition per element of the data, or a list
```

The operator is handed V_{ext} and must work out from it, voxel by voxel, *which* atoms surround a point and how far away they are before it can say anything about the bonding charge there. The potential carries that information, but only implicitly, in the shape of each ion’s local pseudopotential, and a network of a few layers spends much of its depth decoding it. This block states it outright, as extra input channels after the potential:

Key	Default	Meaning
<code>superposition</code>	<code>true</code>	lead with ρ_{sup} , the superposition of isolated atoms the δ -density baseline already builds
<code>elements</code>	<code>auto</code>	one channel per element (that element’s atoms alone, superposed); <code>auto</code> is every element of the training data, a list names them, an empty list gives none

The per-element channels sum to ρ_{sup} , and an element absent from a structure is a channel of zeros. All of them share one asinh scale, the median of $|\rho_{\text{sup}}|$, because a scale fitted to a channel that is mostly zeros would be its noise floor.

Every channel is a scalar field built from radial form factors of the isolated-atom library, so the operator that reads them stays equivariant under rotations, inversion and translations, and every one is a function of the geometry alone: at inference they are rebuilt from the structure exactly as the potential is, and a caller still hands over a potential and nothing else. The library is therefore required (`data.atoms`, with `target: delta`; the combination with `full` raises). The

checkpoint records the resolved block, element list included, because an input-channel count alone cannot say which channels the weights expect; a structure holding an element outside that list is refused at inference rather than having its atoms silently vanish from the per-element channels while remaining in the potential.

Measured on 785 CHONPS cells with the same backbone, the channels took the held-out relative L^2 of ρ from 4.4 % to 3.7 % at epoch 10: one comparison, early in training, so a reason to try the block rather than a settled number.

8.7 model — the network of each operator

`ext2chg.model` and `chg2tau.model` have the same keys; they differ in their defaults, because the two maps differ in difficulty:

Key	<code>ext2chg</code>	<code>chg2tau</code>	Meaning
<code>width</code>	64	16	channel width of the Fourier layers
<code>layers</code>	4	3	number of Fourier layers
<code>modes</code>	16	8	retained Fourier modes per axis (under <code>band</code> , the ceiling)
<code>band</code>	<code>null</code>	<code>null</code>	<code>null</code> : <code>modes</code> on every axis; a number in $\text{\AA}^{-1}$: the same physical band in every cell
<code>projection_channels</code>	64	64	hidden width of the read-out head
<code>activation</code>	<code>silu</code>	<code>silu</code>	<code>silu</code> , <code>gelu</code> , <code>relu</code> or <code>tanh</code> , in every layer and in the read-out
<code>cell_conditioning</code>	32	32	width of the lattice embedding that modulates every layer (FiLM); <code>false</code> drops it
<code>equivariant</code>	<code>false</code>	<code>false</code>	rotation-equivariant radial kernel — a block, see below
<code>precision</code>	<code>float32</code>	<code>float32</code>	<code>float32</code> or <code>float64</code> , what the operator computes in

8.7.1 width, modes, 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.

`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.

8.7.2 activation

`silu`, `gelu`, `relu` and `tanh` are stateless: one fixed function, no parameters, nothing to learn, and the same function serves the read-out. `silu` is the default because it won: learnable Kolmogorov–Arnold activations were tried against it on this project’s Pt data and none beat it. The KAN code, its tests and its comparison configs now live in `experiments/kan_symbolic`; `model.kan_setup` raises and names the archive.

8.7.3 cell_conditioning

The lattice, through an embedding of this width, modulates 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 it on unless deliberately ablating it. The three fractional coordinates are appended to the input as extra channels exactly when the model is *not* equivariant; there is no key for it, since an equivariant operator cannot take them (below).

8.7.4 band

`null` 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.

A number keeps every mode below that wavevector, retaining $\lfloor \text{band } L_i / 2\pi \rfloor$ modes along an axis of length L_i (never more than `modes`), so every material contributes the same band of physics. Prefer it when cell sizes vary widely.

Worth trying rather than only tidier. Measured at LNCC on 115 Materials Project structures whose cells span 2.29 to 25.87 Å, a fixed band with a generous value beat the per-axis count by 20 % in **both** seeds, and was *faster*, since fewer modes are retained. It reproduced in a second regime sharing no parameter with the first (32^3 / `modes` 8 / `band` 16, and 64^3 / `modes` 24 / `band` 24; 20.5 % and 20.3 %). The mechanism looks like uniformity rather than bandwidth: with a fixed count, cells spanning a factor of 11 in length retain physical bands spanning a factor of 11, and `band` removes that.

It is not the default (one dataset, one architecture), and it is not a free switch either. **Too small is much worse than null:** a band of 6 starved the model and lost 55 %. The useful value also moves with the regime (16 in the first study, 24 in the second, with the margin collapsing to 3 % at 32), so it belongs to the run rather than to the schema. `poraquê-train` reports the modes retained per structure at startup and, for an equivariant run, the band they span; read those rather than trusting a number copied from another config.

`band` does not size the radial basis of an equivariant model when it is `null`: it did until 2026-09-04, silently, and a 90-run architecture study at LNCC was run through a leftover value because nothing in the log said so. The basis is `equivariant.g_basis`, and only that (below).

8.7.5 equivariant

A block, off by default:

```
ext2chg:
  model:
    equivariant:
      n_radial: 32
```

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

Key	Default	Meaning
<code>n_radial</code>	16	radial basis functions: the whole capacity of the kernel
<code>g_basis</code>	auto	radius in Å ⁻¹ the basis spans: <code>auto</code> or a number

g_basis: auto — size the basis from the data

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

missing	what happens	measured cost
too narrow	modes beyond the basis are clamped onto its outermost function and share one response	none: 71 % of modes clamped moved the error less than the seed spread
too wide	radial functions sit at radii no mode reaches	14 % to 43 % : a basis of 48 against a median band of 31 orphaned six of sixteen functions

auto takes the **median** band the training split retains, resolved before the model is built so that the number a checkpoint records is a number. Half the set is then clamped, deliberately: that is the free half. The median is also what fits the measurements: the four configurations that performed well had **g_basis** within a few percent of the median band edge, and the one that did not was 54 % above it. The run says at startup what the basis spans, the range and median of the band the training structures retain, and the fraction of retained modes lying beyond the basis.

Three conditions have to hold together, and each fails silently. The kernel being radial is the one everybody thinks of; the other two were found by measuring, and the schema now guarantees them rather than asking.

The retained set must be a ball, not a box. A radial multiplier over a box of modes is equivariant only under the box's own symmetry group. A cubic cell with equal mode counts hides this completely, because the box *is* invariant under the octahedral group; on a tetragonal cell a box takes the error from 3e-7 to 5e-2. The modes of an equivariant model are always a ball (there used to be a **spherical_cutoff** key; it is gone, as nobody wanted the broken setting).

No coordinate channels. The three fractional coordinates are not three scalar fields (under a rotation they turn into each other) and being absolute positions they cost translation equivariance too. An equivariant model does not append them.

The coefficients must be real, which the layer enforces.

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

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

Whether it pays was measured once, on a V100: 400 epochs on 115 Materials Project structures at 32³, **width 32 / n_radial 32** reached a validation relative L^2 of 0.047 against 0.173 for a dense model with 28× the parameters, 4.1× better. The 4× *larger* equivariant arm (**width 128, n_radial 64**) was **worse**, at 0.059: capacity is not what buys this, the constraint is. One seed, one dataset, and neither equivariant arm had converged at 400 epochs, so read it as a reason to try the block rather than as a number.

Note

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

Caveat

NVIDIA’s cuequivariance is deliberately **not** used and is not a dependency. It accelerates arithmetic on irreducible representations of $O(3)$, and a scalar-field-to-scalar-field operator carries only $\ell = 0$: every tensor product in it is a scalar multiply, and the Fourier layer is a complex `einsum` diagonal in the mode index, which Clebsch–Gordan cannot express. Profiled by kernel class on a V100 its share is 0.0 %. Equivariance here is a constraint on the kernel, not a kernel library.

8.8 `ext2chg.occupancies` — the per-atom PAW head

A block, **on by default**:

```
ext2chg:
  occupancies:
    weight: 3.0
    neighbor_min: 0.6
```

`false` trains the grid alone and needs no POTCAR; `true` or the bare key is the defaults.

Key	Default	Meaning
<code>weight</code>	1.0	the per-atom term against the field objective, in training and in the validation score alike
<code>magnetization_weight</code>	1.0	weight of the magnetization record set in the occupancy loss and score, against the charge set’s
<code>readout_band</code>	auto	the band in $\text{\AA}^{-1}$ read at each atom; <code>auto</code> is 95 % of the coarsest Nyquist frequency over the training and validation grids
<code>neighbor_min</code>	2.2	first of ten neighbor-kernel centers in $\text{\AA}$, below the fixed last center at 5.8 $\text{\AA}$; 0.6 also covers short CHON bonds

(`runtime.readout_cache_mib` sets the device memory for the read-out’s kernels; it is a hardware setting, below.)

With the head on, the operator maps V_{ext} to two things at once: the pseudo-density on the grid and every atom’s **augmentation occupancies** $\rho(\ell\ell'LM)$, the one-center PAW terms VASP writes after the grid, which together with the pseudo-density are what an all-electron density is reconstructed from. They are not a field. There are 138 per atom for **PAW_PBE Pt** (projector channels d, d, s, s, p, p, in the POTCAR’s order) and one set per density channel, so a spin-polarized dataset keeps the magnetization’s set as well.

The operator is built as `experiments/paw_occupancies` measured it should be (`paw`):

An FNO for the grid, sized by the section’s `model`. Every other `model` setting (activation, equivariance, band) applies to it unchanged.

A readout at each atom, of the latent field and the predicted density, onto solid-harmonic Gaussians $r^L Y_{LM}(\hat{r}) e^{-r^2/2s_n^2}$. It is the decoder half of a graph neural operator, with one constraint that was not optional. The fields are first restricted to one band $|\mathbf{G}| \leq \text{readout_band}$ and resampled to a spacing of $\pi/2G_{\max}$. A readout of the DFT density integrated on each grid as it came scored 3 % on nanoparticles it had seen and 184 % on held-out ones, whose grid spacing differs; over one band, 4.9 %. A grid too coarse to supply the band **raises**, at training and at inference.

`auto` takes the band from the validation grids as well as the training ones. The band is a property of the geometry, not of any target, and a held-out cell coarser than every training cell would otherwise stop the run at its first validation pass rather than before the first epoch.

The geometric neighbor features retain ten Gaussian channels of width 0.3 Å, under a 6 Å cutoff. `neighbor_min` changes their first center, preserving the original 2.2–5.8 Å range by default. The experiment in `experiments/chon_ext2chg` tests 0.6–5.8 Å for shorter covalent bonds. Their actual centers are checkpoint buffers; loading an older checkpoint preserves its features.

An equivariant head per element. Each L block of the record is one map shared across M, gated by an MLP over the invariants, with a linear path beside it. With `model.equivariant` on, the whole operator rotates its predicted occupancies with the crystal, measured to 7e-15 in double precision. The layout of each element’s record comes from the POTCAR’s `Non local Part`, so a dataset without POTCARs cannot train it.

An invertible target. The records train with $L = 0$ centered and every (set, L, pair) block scaled to unit RMS, both fitted on the training split, stored in the checkpoint and inverted at prediction, and every L block weighs the same in the loss. On raw records the loss would be 99 % the $L = 0$ block.

One weight per record set. A spin-polarized density carries a second set, the magnetization’s, and with every (set, L) block weighed alike its $L > 0$ blocks (0.005 % of the occupancy norm on the CHONPS set, and as unpredictable from the potential as the field magnetization is: 98 % held-out error at $L = 1$) were most of the occupancy loss and of the score the best epoch is chosen on. `magnetization_weight` scales that set against the charge set’s; below one, the charge blocks drive both. The weights are normalized to their mean, so the default of 1.0 is the unweighted loss to the bit. A one-channel dataset has no second set and ignores the key. It is not `training.spin.magnetization_weight` (Section 8.10.5), which weights the magnetization *channel* of the field objective.

The training log gains two columns: `val occ`, the relative RMS of the held-out occupancies in physical units, and `occ loss`, the training term on the held-out split. The best epoch is chosen on `val rel L2 + weight x occ loss`, the objective itself, on held-out data. `val occ` is reported rather than selected on because 99 % of the records’ squared norm is the charge set’s $L = 0$ block: it reads a few percent while the $L > 0$ blocks are tens of percent wrong. After training, each structure’s `occupancy_relative_rms` is written to the metrics JSON beside a baseline (the training-mean $L = 0$ record with every $L > 0$ block zero), and each split is broken down by record set and L, in the log and under `occupancies`.

A bundle holding this operator gives `poraquê-inference --add-paw` a fourth PAW source, `model`; `auto` prefers it to the averaged table.

The cache carries the records; a cache built before 16 September 2026 dropped them, so delete it and let it rebuild. `FieldPairDataset.site_records(i)` returns one structure’s (occupancies, record lengths, atomic numbers and fractional positions, in the file’s atom order) for both

operators that carry them: as the target of `ext2chg` and as the input of `chg2tau` (Section 8.9). A dataset built for either yields, beside the usual `input` and `target`:

Key	Shape	
<code>paw</code>	(B, A, S, L)	occupancies: batch, atoms, record sets, values
<code>paw_mask</code>	(B, A, L)	a real value of a real atom's record
<code>atom_mask</code>	(B, A)	a real atom
<code>species</code>	(B, A)	atomic numbers, 0 for padding
<code>positions</code>	(B, A, 3)	fractional coordinates

Caveat

Not yet supported, and refused before the cache: `training.validation.kfold` (the transform is fitted per split, not per fold), `fine_tuning` (a pretrained bundle has no occupancy head to adapt), and an f-channel element, whose records reach $L = 6$. Each is fine with `occupancies: false`.

8.9 chg2tau — the kinetic operator

The kinetic energy density from the pseudo-density **and** every atom's augmentation occupancies, the on-site part the grid cannot hold, entering as input. Present: trained, on the materials that carry a TAUCAR.

Key	Default	Meaning
<code>head</code>	<code>plain</code>	<code>plain</code> predicts τ ; <code>pauli</code> predicts $\tau_{\text{vW}}[\rho] + s \text{softplus}(f)$ — see below (Section 8.9.1)
<code>model</code>	width 16, layers 3, modes 8	the network — see above (Section 8.7)
<code>occupancies</code>	<code>on</code>	the per-atom input; <code>false</code> trains on the grid alone
<code>occupancies.channels</code>	8	field channels each atom's records are written into

```
chg2tau:
  occupancies:
    channels: 8
```

What `ext2chg` predicts is exactly what `chg2tau` reads, which is the route this project takes to a total energy: $V_{\text{ext}} \rightarrow (\tilde{\rho}, \rho_{ij}^a) \rightarrow \tau$. The data is the same as for the grid-alone kinetic map: a material needs a CHGCAR with its records and a TAUCAR, and the records come from the input density, read by the same `site_records`.

The operator is the ordinary FNO with more input channels. A `SiteWriter`, the transpose of the `ext2chg` readout, writes each atom's normalized record as a field: atom-centered solid-harmonic Gaussians $r^L Y_{LM}(\hat{r}) e^{-r^2/2s^2}$ at three widths (0.25, 0.45 and 0.8 Å), with coefficients from one learned map per element and L, **shared across M**. A rotation of the atom's occupancies therefore rotates the kernel it writes, the written field is a scalar field, and the operator reading it stays equivariant. The field is built in reciprocal space, where it is exact: by Hobson's theorem the Fourier transform of a solid harmonic times a Gaussian is the same solid harmonic

of $\mathbf{G}$ times a Gaussian, so each atom is a phase $e^{-i\mathbf{G}\cdot\mathbf{R}_a}$, the periodic sum over images comes free with the inverse FFT, and nothing depends on the grid but the band it can hold. The `channels` written fields are concatenated after the density, and an `OccupancyInputModel` hands the whole stack to `FN03d`. The records are normalized as `ext2chg` normalizes its target (the same `OccupancyTransform`, fitted on the training split and stored in the checkpoint) and their layouts come from the POTCAR, so here too a dataset without POTCARs cannot train it.

The operator cannot predict from a density by itself: `FieldOperator.predict(field, sites=...)` needs the structure's records, a dataset's `site_records(i)` or occupancies, species and positions assembled by hand, and raises without them.

Caveat

`chg2tau` with its occupancies is refused under `training.validation.kfold` and under `fine_tuning`, as `ext2chg` is and for the same reasons. The ASE calculator does not run it yet.

`poraquê-inference` chains the density operator into the kinetic one. The two operators are usually trained apart (the density on whatever archive is at hand, the kinetic energy density on calculations that wrote a TAUCAR), so `--models` may be repeated, and each task is taken from the first bundle that holds it:

```
poraquê-inference Pt2/ --models models/chonps_v2/chonps_v2.poraquê \
                    --models models/pt_chg2tau/pt_chg2tau.poraquê
```

`--kinetic auto` (the default) runs `chg2tau` whenever a bundle holds it, and then prefers `ext2chg` for the density, so the pseudo-density and the occupancies `chg2tau` reads come from one model; `--kinetic pseudo2tau` keeps the density route. The occupancies are the ones the `ext2chg` operator predicts for the structure in front of it; with no such operator in `--models`, those of a reference CHGCAR beside the structure; with neither, the run stops and says what to add.

8.9.1 head: pauli

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. The scale s is fitted on the training split and then learned alongside the backbone; neither is a key. Against a matched baseline it improved every fold, by 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, with one limitation of the present version: it is implemented for the grid alone, so it needs `occupancies: false`, and with the occupancies on it raises.

```
chg2tau:
  head: pauli
  occupancies: false
```

Caveat

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

8.10 training — how it is fitted

One section for both operators.

Key	Default	Meaning
<code>epochs</code>	500	passes over the training set
<code>batch_size</code>	4	maximum samples per grid-shape bucket
<code>samples_per_step</code>	0	structures per optimizer step, accumulated across buckets; 0 steps once per batch
<code>optimizer.name</code>	<code>adamw</code>	<code>adamw</code> , <code>adam</code> or <code>sgd</code> , on a cosine schedule with the gradient norm clipped at 1
<code>optimizer.learning_rate</code>	0.002	step size
<code>optimizer.weight_decay</code>	0.0001	weight decay
<code>loss.norm</code>	<code>relative_l2</code>	<code>absolute_l2</code> , <code>relative_l2</code> , <code>absolute_h1</code> or <code>relative_h1</code>
<code>loss.sobolev_weight</code>	0.1	gradient-term weight; read by the two <code>h1</code> norms only
<code>spin.mode</code>	<code>joint</code>	<code>joint</code> or <code>per_channel</code> : how the two channels of a spin-polarized target enter the data term
<code>spin.magnetization_weight</code>	1.0	weight of the magnetization term; <code>per_channel</code> only
<code>spin.magnetization_floor</code>	0.01	floor on its denominator, as a fraction of $\ \rho\ $; <code>per_channel</code> only
<code>validation.fraction</code>	0.2	fraction of structures held out; 0 uses every structure
<code>validation.every</code>	10	evaluate and log every N epochs
<code>validation.patience</code>	300	stop after N epochs without improvement; 0 never stops
<code>validation.kfold</code>	<code>null</code>	K: cross-validate over K folds instead of one split
<code>physics.electron_count</code>	0.0	weight of $\int \rho = N$ (density operator)
<code>physics.positivity</code>	0.0	weight of $\rho, \tau \geq 0$
<code>physics.von_weizsacker</code>	0.0	weight of $\tau \geq \tau_{\text{vW}}$ (kinetic operator)

8.10.1 validation: the split, the folds, the stopping

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

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

kfold: K swaps that for K-fold cross-validation (on the command line, `--kfold K`, or a bare `--kfold` for five): 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 generalizes, and produces K models rather than one to deploy. **fraction** is ignored, since the folds define the splits. Setting K to the number of structures gives leave-one-out; it is capped there. It cannot be combined with **fine_tuning** (each fold trains from scratch) and is not available to an operator with its occupancies on.

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 neighboring voxels are strongly correlated the score would look excellent while saying nothing about transfer to a new material.

Caveat

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 artifact set **fraction: 0**, accepting that its own metrics become a training fit, and quote a separate **kfold** run for the generalization number.

Caveat

With **fraction: 0** the reported metrics are **training fit**, carrying no generalization claim. On the reference dataset they are about four times better than the cross-validated score. Quote the cross-validated numbers.

every is how often to evaluate and log. 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
  -----
      10/500      0.31745      0.34118
      20/500      0.18902      0.21447 *
```

patience stops the run after this many epochs without an improvement in the **validation** error, and restores the best weights seen:

```
      30/500      0.03173      0.03659 *
      ...
      38/500      0.02249      0.06568
  stopped early at epoch 38: no improvement in 8 epochs (best 0.03659 at epoch
  ↪ 30)

  trained 38/500 epochs in 12.0 s   loss 0.8599 -> 0.0225
```

Caveat

It needs a validation split (**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 patience anyway **warns** rather than silently doing nothing and leaving you believing the run was protected.

Patience is counted in *epochs* but checked only on the epochs where validation is computed, so a value below **every** behaves like **every**. 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. **history** records **best_epoch**, **best_error** and **stopped_early**. 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.

8.10.2 batch_size

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

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

`peak_vram_bytes` and `seconds_per_epoch` in the metrics JSON are what a sweep over `batch_size` should be read from. The whole run's cost, stage by stage, closes the log (the resource profile (Section 8.13.2)).

8.10.3 samples_per_step

One optimizer step per batch weighs every *batch* alike, and so a structure by the inverse of its batch's size. On a dataset whose grid shapes are mostly unique that is far from neutral: a structure alone in its bucket moves the weights as far as four structures sharing the commonest shape do together, at the default `batch_size: 4`, and the cells that happen to have odd shapes are fitted at the expense of the rest.

`samples_per_step: N` accumulates gradients over consecutive shape-bucketed batches until at least `N` **structures** have been seen, then takes one step. Each batch's loss is weighted by its size, so every structure carries the same weight whatever bucket it fell in. Bucketing is untouched (no batch ever mixes shapes); it is only the step that spans several of them. The gradient of a step is the mean over its structures, and the epoch's remainder, fewer than `N`, takes a step of its own rescaled to remain a mean. `0`, the default, keeps the one-step-per-batch behavior. The optimizer then takes about one step per `N` structures rather than one per batch, which is a change of schedule as well as of weighting; compare against a baseline at matched steps, not only at matched epochs.

8.10.4 loss

Four norms, on two named axes:

norm	normalized per sample	gradients in the objective
<code>absolute_l2</code>	no	no
<code>relative_l2</code>	yes	no
<code>absolute_h1</code>	no	yes
<code>relative_h1</code>	yes	yes

Relative divides each sample's error by its own target norm, so materials whose fields differ by orders of magnitude contribute equally and the value reads directly as a fraction. **Absolute** does not, so every *voxel* counts equally and a denser system contributes proportionally more gradient: right for a set whose materials are genuinely comparable in magnitude, and usually wrong for a heterogeneous one, which is why `relative_l2` is the default.

H1 adds `sobolev_weight` times the same error on the spatial gradients. Worth using when the derivative matters: τ_{vW} depends on $\nabla \rho$, so gradient noise is amplified in low-density regions.

Both halves are taken in the *same* norm, so the weight is a pure ratio between values and derivatives rather than also absorbing a change of scale.

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

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

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

8.10.5 spin: the two channels of a spin target

A CHGCAR written with `ISPIN = 2` carries (ρ, m) , and there are two ways to score it. The block is read only for a two-channel target.

`joint`, the default, is one norm over both channels: each channel's error divided by the whole (ρ, m) target's norm and combined in quadrature, which is identically $\|\delta(\rho, m)\|/\|(\rho, m)\|$. `per_channel` divides each channel by **its own** norm and sums them,

$$\mathcal{L} = \mathcal{L}[\rho] + w_m \mathcal{L}[m],$$

with $w_m = \text{magnetization_weight}$. Because $m \equiv 0$ exactly on a large fraction of a real set, the magnetization's denominator is floored: $\sqrt{\|m\|^2 + (\eta\|\rho\|)^2}$ with $\eta = \text{magnetization_floor}$. A single-channel target is identical in both modes, bit for bit, and under `joint magnetization_weight` **does nothing**: there is one term for it to scale.

The default is the measured one, and the arithmetic misleads about which should win. Under `joint` the magnetization's share of the objective is 3×10^{-8} in the *median* on the 432-cell CHON set, which is the observation that makes `per_channel` look obviously correct. Measured head to head on that cache (two seeds, 60 epochs, magnetic cells scored by predicted over reference $\|m\|$ and by m 's relative L^2):

training set	joint	per_channel
30 magnetic + 30 noise cells	1.00 / 0.084	1.00 / 0.109
12 magnetic + 100 noise cells (the real 11 % imbalance)	0.96 / 0.166	0.001 / 1.000

What trains the magnetization is the *mean* share, not the median: the magnetic cells carry real weight under `joint`, and that is enough. `per_channel` instead hands each of the many noise cells a floored term of order one, and a hundred cells saying "drive m to zero" bury the dozen saying anything else, so it collapses the magnetization it was introduced to rescue. Reach for it only when magnetic cells are not a small minority, and measure rather than assume.

The progress table's `charge` and `mag` columns are the two parts in whichever mode is running, and the held-out error early stopping reads carries the same mode, so the checkpoint is selected

on the objective rather than on a third quantity. Not to be confused with `ext2chg.occupancies.magnetization_weight`, which weights the magnetization *record set* of the per-atom occupancy target.

8.10.6 physics

Weighted penalties on the predicted fields, each dimensionless, so a single weight can serve a heterogeneous dataset:

Key	Constraint it penalizes the violation of	Default
<code>electron_count</code>	$\int \rho d\mathbf{r} = N$ (density operator)	0.0
<code>positivity</code>	non-negativity of the predicted field	0.0
<code>von_weizsacker</code>	$\tau \geq \tau_{\text{vW}}$ (kinetic operator)	0.0

```
training:
  physics:
    electron_count: 0.1
```

All three default to zero, so the objective is the plain supervised baseline until one is chosen deliberately, and there is no switch beside them: **all zero is off**. A run with every weight at zero skips the machinery altogether, which is not merely a change of objective. Every constraint acts on *decoded* fields, so with one live the loop must copy the reference field to the device on every batch, invert the target transform over the prediction inside the autograd graph, invert the input transform too, and in delta-density mode copy the baseline across and add it back twice. Off, all of that is work whose result is discarded. Measured: 2.4 % of a step on Apple Metal and **6.6 % on a V100**, where the per-batch copy crosses PCIe.

(The orbital-free stationarity residual, which used to be a fourth weight, is not configurable: training never read `xc`, so the term ran without v_{xc} . `experiments/euler_lagrange` is where it lives.)

Charge conservation

`electron_count` is the first constraint to reach for:

$$\mathcal{L}_N = \left\langle \left(\frac{\int \hat{\rho} d^3r - \int \rho d^3r}{\int \rho d^3r} \right)^2 \right\rangle,$$

the squared relative error between the integral of the **predicted** density and the integral of the **reference** one, the valence electron count. Two things make it the first constraint to reach for:

It needs no labels. The reference density is in every batch, so N is its own integral. That is what lets it work on an archive that publishes ρ and nothing else.

For a density operator with the occupancies on, this constrains the total charge channel of the grid density. The **VASP CHGCAR convention** defines its integral as `NELECT`; the PAW occupancy records are separate targets and are not summed into that integral. Magnetization is not constrained by this term. The loss uses physical units after the inverse transform and restoration of the atomic baseline in **delta** mode. Using the reference integral also preserves the electron count of charged training structures.

This is a **soft penalty**, so a positive weight does not guarantee exact conservation. The validation score includes the weighted mean squared count error; the log reports `val |dN/N|` before output normalization. The JSON history contains `train_electron_count_loss`, `val_`

`electron_count_loss`, and `val_electron_count_error` (a relative fraction, not a percentage). Conservation is exact at inference, where the density is renormalized to the valence count (`--no-normalize` opts out).

It fixes what the data term cannot. A relative L^2 is indifferent to a percent of charge spread thinly through the interstitial region, and a total energy is not. Across a chemical space where N runs from ten to ninety electrons per cell, nothing else pins the scale per material.

A weight of 0.1 sits an order of magnitude below the data term, which is the intended balance: the physics guides, the data decides. The configuration under `experiments/chon_ext2chg/configs/` that used a weight of 1.0 is in the previous schema and a historical record; its weight has not been selected by a hyperparameter sweep. The training log reports one number, `train loss`: the **total** objective the optimizer stepped on, data fidelity plus every weighted constraint. That total is the only quantity comparable between two runs; the individual terms are reported by the loss unweighted, so they do not sum to it.

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 initialization is noise. A badly scaled constraint degrades accuracy while looking principled.

Where a constraint can be imposed structurally, prefer that: `chg2tau.head: pauli` enforces $\tau \geq \tau_{vW}$ exactly, whereas `von_weizsacker` only discourages violating it.

8.11 fine_tuning — start from a trained checkpoint

A block, off unless present. Fine-tuning needs the section in the configuration (the command-line flags `--pretrained` and `--fine-tune-lr` override its values and do not switch it on):

Key	Default	Meaning
<code>checkpoint</code>	<code>models/poraque_models/poraque_models.poraque</code>	the base bundle to adapt
<code>learning_rate</code>	<code>1.0e-5</code>	much lower than a fresh fit, which is the point
<code>freeze_lifting</code>	<code>false</code>	hold the input embedding fixed and adapt the rest
<code>lora</code>	<code>false</code>	freeze every weight and learn low-rank adapters — a block
<code>lora.rank</code>	<code>8</code>	adapter rank
<code>lora.alpha</code>	<code>16.0</code>	the update is scaled by α / rank
<code>lora.dropout</code>	<code>0.0</code>	adapter dropout

```
fine_tuning:
  checkpoint: models/mp_general/mp_general.poraque
  learning_rate: 1.0e-5
  lora:
    rank: 8
```

The normalizations come from the checkpoint and are never refitted. It cannot be combined with `training.validation.kfold`, and it is refused for an operator with its occupancies on (a

pretrained bundle has no occupancy layer to adapt). See Chapter 7.

8.12 runtime — the hardware

Settings that change how fast a run is, never what it computes.

Key	Default	Meaning
<code>device</code>	<code>auto</code>	<code>auto</code> , <code>cuda</code> , <code>cuda:1</code> , <code>mps</code> or <code>cpu</code>
<code>strict_device</code>	<code>false</code>	abort instead of falling back to the CPU — see below
<code>distributed</code>	<code>auto</code>	<code>auto</code> or <code>off</code> ; multi-GPU over NCCL when the launcher describes a group
<code>workers</code>	<code>auto</code>	processes that <i>build</i> the cache, one material each — see below (Section 8.12.2)
<code>loader_workers</code>	<code>0</code>	DataLoader worker processes — read <code>data.cache.in_memory</code> first
<code>tf32</code>	<code>true</code>	TensorFloat-32 matmul and convolution; CUDA, Ampere and later
<code>readout_cache_mib</code>	<code>1024</code>	device memory for the <code>ext2chg</code> read-out's kernels

8.12.1 device and strict_device

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

That is right on a workstation and expensive in a queue. `strict_device:` `true` turns the fallback into an error:

```
runtime:
  device: cuda
  strict_device: true
```

Every HPC configuration should set it. Without it, a job that cannot reach its GPU takes its place in the queue, warns into a log nobody reads until afterwards, and trains on the CPU *inside* the GPU allocation until the wall clock ends. The refusal names the probable cause (no GPU visible, an empty `CUDA_VISIBLE_DEVICES`, a driver older than the wheel's CUDA runtime, or a build carrying no kernels for this architecture) and prints the full device report into the run's log.

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

`strict_device` applies to `device:` `auto` as well, which is the default: it refuses when `auto` resolves to the CPU, and names `device:` `cpu` as the way to ask for CPU training deliberately.

8.12.2 workers

How many processes **build** the cache, one material each. `auto` (the default), a count, or `0` for the single-process build (`--cache-workers` on the command line). Not to be confused with `loader_`

workers (Section 8.12.3), which loads an already-built cache during training and is the opposite trade.

The build is not the FFT work it looks like. Measured per material on the 97-cell platinum set at **resolution: 48**:

stage	share of the clock
Fortran E17.11 text reading and writing	86 %
computing <code>V_ext</code> (POTCAR tables + one FFT)	8 %
spectral downsampling: the FFT	2.2 %

So a thread count cannot help: there is no BLAS call anywhere in the path, and every transform goes through `np.fft`, which is single-threaded. A cache job that asks for eight cores and sets `OMP_NUM_THREADS=8` uses one of them. What scales is **materials: _build_one** is independent per material, and it already writes every field to `.partial-<pid>` and moves it into place with `os.replace`, so a pool needs no lock and the result is byte-identical to the single-process build.

Measured on eight cores, 16 materials at 108^3 :

workers	text storage	HDF5 storage
1	2.764 s/material	1.515 s/material
2	1.462 s (1.89×)	0.832 s (1.82×)
4	0.868 s (3.18×)	0.518 s (2.92×)
8	0.684 s (4.04×)	0.440 s (3.44×)

On the full 97-cell set the `--cache-only` cache stage went from **656.9 s to 246.1 s** at the 4 workers `auto` picks there, $2.67\times$, not $4\times$, because per-material time rises $1.49\times$ under memory-bandwidth contention on the 288^3 grids. Against the 961.8 s the build cost before the two serial redundancies (reading only the header for the grid shape, and splitting the text block in chunks) were removed, it is $4.0\times$.

`auto` takes the smallest of three numbers and says which bound it hit:

```
workers: 4 (auto: 8 usable core(s); 4 within the 8 GiB budget at ~1995 MiB
each, sized on the largest grid at 23.89 M voxels; 97 material(s))
```

- the cores this process **may** use (`sched_getaffinity`, which is what Slurm's `--cpus-per-task` restricts a task to; the node's core count is the wrong number on a shared node);
- a memory budget divided by an estimate taken from the **largest** native grid in the set. Measured by building one material per process: a 108^3 density peaked at 498 MiB and a 288^3 one at 1558 MiB, so the model is 400 MiB fixed plus 70 B per voxel. The largest rather than the average, because every worker may be handed the nanoparticle and sizing on the mean is how a build goes into swap on the last eleven materials. The budget is **80 % of the job's Slurm allocation** where there is one (`SLURM_MEM_PER_NODE`, or `SLURM_MEM_PER_CPU × SLURM_CPUS_PER_TASK`) and 8 GiB otherwise; a constant would pick four workers on a `--cpus-per-task=8 --mem=32G` node and idle half the allocation, which is the exact job this exists for;
- the number of materials.

Below **1 M native voxels in total** `auto` stays single-process whatever the core count: a **spawned** pool costs 0.49–0.60 s for each worker to import the package, and the build runs at 1.03 s per million voxels, so a pool on a toy set is pure loss. An explicit count ignores that floor.

Materials are handed out **largest first**: their native grids span a factor of 19 in this set and the cost is linear in voxels, so in discovery order the biggest cells form a tail with idle cores behind them. Rows therefore print in completion order rather than alphabetically, and i/N counts materials finished. Raise the job's memory request along with its core count: a 288^3 density needs ~1.6 GiB per worker.

8.12.3 loader_workers

`loader_workers` and `data.cache.in_memory` (Section 8.5.6) are **alternatives, not complements**, and the measurement is unambiguous. Twenty epochs on 115 structures at 32^3 , on a V100:

variant	training time	speedup
as shipped	168.0 s	1.0×
<code>cache.in_memory: true</code>	16.3 s	10.3×
<code>loader_workers: 4</code> , no cache	120.2 s	1.4×
both	18.8 s	8.9×

The validation error was identical in all four, as it must be. Each worker is a *process* with a cache of its own, so it re-parses the whole dataset on its first epoch; the parse the cache removes is paid N times instead of once. Raise `loader_workers` only when the data genuinely does not fit in RAM, which is the case `cache.in_memory: false` exists for. Pinned memory is a CUDA transfer optimization and nothing else, so it is on there and off elsewhere, and is not a key.

8.12.4 tf32

`tf32` keeps float32's range and drops its mantissa to ten bits inside the tensor cores. It is a real speedup on Ampere and later, and **exactly nothing** before it (a V100 has no TF32 path), so it is on by default and costs nothing where it does not apply. Ignored off CUDA. An equivariance check under it reads ~1e-3 and is not a bug.

8.12.5 readout_cache_mib

The `ext2chg` read-out's kernels depend only on a cell and its grid, so they are built once per geometry (on the CPU, with SciPy's Bessel functions) and kept on the device, least recently used evicted first, within this budget. A training split whose kernels exceed it rebuilds the evicted ones every epoch, and that rebuild then dominates the epoch: 64 MiB was far below the ~360 MB the 785 CHONPS geometries need at a $6 \text{ \AA}^{-1}$ band, and the 1024 MiB default holds them all. The setting changes how often a kernel is computed, never what it is, so it is not part of the checkpoint.

8.12.6 There is no compile key

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

Inductor does not generate code for complex operators. Every compiled run said so:

```
torch/_inductor/lowering.py:1917: UserWarning: Torchinductor does not support
code generation for complex operators. Performance may be worse than eager.
```

`SpectralConv3d`'s weights are complex, and it *is* the Fourier neural operator. So the one part of the model compilation was invoked to fuse is the part Inductor hands back to eager, while the wrapper and the graph breaks are still paid for. That is the mechanism behind the numbers: 150 epochs, two repetitions, one V100:

arm	s/epoch	first epoch (cold → warm)	val rel L2
off	0.6289	3.8 s	0.280563
mode="default"	0.6785 (+7.9 %)	199.8 s → 17.6 s	0.284980
mode="max-autotune"	0.6879 (+9.4 %)	314.0 s → 20.7 s	0.272624

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

On four ranks it did not merely lose. With `--distributed auto` and 60 epochs, in the same allocation as an uncompiled four-rank run that finished in 86.6 s, the compiled run sat **eighteen minutes in the first epoch without printing an epoch line** and was canceled. A configuration key whose failure mode is a silent deadlock inside a queue allocation is the hazard `strict_device` (Section 8.12.1) exists to remove, pointing the other way.

What was *not* measured, and is worth measuring one day: compiling only the real-valued submodules (CellEncoder, FiLM, the pointwise convolutions, the projection head) and leaving SpectralConv3d in eager. That is where the 44.4 % of elementwise GPU time lives. It should come back as an internal decision about which submodules to wrap, not as a switch a config file can throw. The condition for re-opening whole-model compilation is upstream: Inductor gaining complex-operator codegen.

8.12.7 distributed

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

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

```
#SBATCH --ntasks-per-node=4          # one task per GPU
#SBATCH --gres=gpu:4
srun poraque-train --config configs/train_simple.yaml --distributed auto
```

See Several GPUs, under Slurm (Section 1.6.3) for the full script and for how `MASTER_ADDR` and `MASTER_PORT` are derived from the allocation.

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

Three consequences, all of them visible in the run log:

- The **effective batch size is `batch_size` × the world size**. A four-rank run at `batch_size: 10` steps on 40 samples, so it is not the same optimizer as the one-rank run it is compared against: halve the learning rate or quarter the batch size if the comparison is meant to be of the same optimization.
- **Rank 0 alone writes** the checkpoint, the metrics, the figures and the PDF, and alone prints.
- **The batches are split, not the samples**. `ShapeBucketSampler` groups materials by

grid shape so no padding ever reaches the FFT, and a `DataLoader` takes a `sampler` or a `batch_sampler` and never both, so the bucketing runs first, identically on every rank, and a real `DistributedSampler` partitions the resulting *list of batches*. Each rank gets a unique, non-overlapping subset, and every rank gets the same *number* of batches, which is not tidiness: DDP all-reduces gradients inside each `backward()`, and a rank that runs out of batches first leaves the others in a collective that never completes.

The collective timeout is long (30 minutes) on purpose, and is not a key. The first collective happens after every rank has read its prepared cache, and a cold parallel-filesystem read of a few hundred densities is minutes; a timeout that fires there reports itself as a NCCL error and sends the reader looking at the network.

Note

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

8.13 report — what is written

Key	Default	Meaning
<code>figures</code>	<code>true</code>	render <code>plots/</code> : loss curves, parity over every structure, field slices
<code>pdf</code>	<code>true</code>	typeset <code>report/</code> ; needs a LaTeX toolchain
<code>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
<code>raw_data</code>	<code>false</code>	write the numbers behind each figure beside it

The checkpoint, the log, the metrics JSON and the resolved configuration are always written when `run.output` is set; `run.output: null` is the one switch that silences them all. `--no-report` and `--no-plots` turn the PDF and the figures off from the shell.

8.13.1 What the report's performance table says

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

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

The **train** and **validation** rows read against each other **are** the generalization gap. That is the single most useful thing the table says, and it is why the splits are rows.

Note

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

For a two-channel model every number is the **density** channel; the magnetization is measured separately, as `magnetization_relative_l2` in the metrics JSON, and the report says so in its caveats.

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

8.13.2 The resource profile

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

```
[RESOURCES]
stage                wall time    peak RSS
-----
cache                0.0 s      222.6 MiB
ext2pseudo: setup    0.0 s      228.4 MiB
ext2pseudo: training 0.4 s      319.7 MiB
ext2pseudo: evaluation and figures 1.6 s      500.5 MiB
ext2pseudo: PDF report 0.0 s      500.5 MiB
pseudo2tau: setup    0.0 s      501.3 MiB
pseudo2tau: training 0.0 s      502.9 MiB
pseudo2tau: evaluation and figures 0.9 s      552.8 MiB
pseudo2tau: PDF report 0.0 s      552.8 MiB
checkpoint           0.0 s      552.8 MiB
-----
total (wall clock)   3.1 s      552.8 MiB
device: cpu   outside the stages above: 0.1 s
peak RSS: the process's high-water resident set so far (getrusage), so it never
↳ falls between rows
```

(That is a synthetic smoke run of the grid-alone operators; the shape is what matters. Like every block of the log it opens with its [NAME] header; see the Manual.) Each stage is timed after a device synchronization, so a GPU stage is charged for its compute rather than for the kernels it queued. The memory columns answer the question an allocation is sized from:

- **peak RSS** is the process's high-water resident set, read from the operating system rather than sampled, so a spike between two samples cannot be missed. It is a running maximum and never falls; the stage where it jumps is the one that needed the memory. DataLoader workers are separate processes and are reported on a line of their own.
- **peak VRAM**, on CUDA, is `torch.cuda.max_memory_allocated` *within* each stage (the peak statistics are reset at every boundary, so the column does fall). The summary adds the most the caching allocator reserved, which is the number `nvidia-smi` shows and the one an out-of-memory error is about.
- **MPS at end** is what the Metal driver held when the stage ended. MPS exposes no peak, so on a Mac this column is a floor, not a maximum.

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

8.13.3 raw_data

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

<code>plots/ext2pseudo_loss_curves.png</code>	<code>plots/ext2pseudo_loss_curves.csv</code>
<code>plots/ext2pseudo_parity.png</code>	<code>plots/ext2pseudo_parity.csv</code>
	<code>plots/ext2pseudo_parity_bin_edges.csv</code>
<code>plots/ext2pseudo_s000_field_slice.png</code>	<code>plots/ext2pseudo_s000_field_slice.npz</code>

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

- **The loss curve** is one row per epoch: `epoch`, `train_loss`, and the validation column *named for the norm it holds* (`val_rel_l2`, or `val_abs_h1` for an **absolute_h1** run). Epochs on which validation was not measured leave that cell **empty** rather than interpolated; filling them would put points in the file the run never measured. There is no physics column: `train_loss` is the total the optimizer stepped on, and the per-term breakdown is not recorded.
- **The parity plot** is one row per *occupied* bin: `split`, `reference`, `prediction`, `count`, `density`. Long format, because that pivots straight back into a `pcolormesh`; occupied bins only, because a 200-bin grid is 40 000 cells of which a few thousand carry anything.

The bin edges go in their own file, since a log axis makes them unrecoverable from the centers. `count` is what the figure drew, so on log axes it leaves out every sampled voxel at which reference or prediction is not positive; the panel titles give the sample itself. A spin-polarized run writes the same pair again for m , as `<task>_parity_magnetization`.

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

It needs **figures**. The data is written by the figure methods as they draw, which is what guarantees the file and the image show the same numbers rather than two independent computations of them. PDF reports are assembled in a temporary directory that is removed afterwards, so no `.tex`, `.aux` or `.log` files are left behind.

8.14 Worked examples

The short form of the platinum run, both operators, is `configs/train_simple.yaml`. The ones below change one thing each.

Train the deployable models: nothing held out, the Pauli head for the kinetic operator (which, for now, means the grid alone):

```
run:
  name: poraque_models
ext2chg:
  occupancies: false
chg2tau:
  head: pauli
  occupancies: false
training:
  epochs: 200
  validation:
    fraction: 0
```

Measure generalization:

```
run:
  name: poraque_models_cv
ext2chg:
  occupancies: false
training:
  validation:
    kfold: 5
```

Two runs over the same data with different physics, kept side by side. The baseline leaves the weights at zero, which also skips the per-batch decode the constraints need:

```
run:
  name: agaupt_baseline
ext2chg:
  occupancies: false
```

```
run:
  name: agaupt_charge_conserving
ext2chg:
  occupancies: false
training:
  physics:
    electron_count: 0.1
```

A Materials Project download, which ships no POTCAR and no TAUCAR: the density operator on the grid alone, with the library that generated the data.

```
run:
  name: mp_ag_au_pt
data:
  path: data/MP
  potcar_dir: /opt/vasp/potpaw_PBE
ext2chg:
  occupancies: false
```

A fast smoke test:

```
data:
  grid:
    resolution: 20
ext2chg:
  occupancies: false
model:
  width: 8
  modes: 4
  layers: 2
  projection_channels: 16
training:
  epochs: 10
```

Higher fidelity, if the hardware allows:

```
data:
  grid:
    resolution: 64
ext2chg:
  model:
    width: 32
    modes: 12
    layers: 4
training:
  epochs: 400
  batch_size: 2
```

Troubleshooting

9.1 The run stops immediately

“No material with a charge density found under ...” Check `data.path`: it is one directory holding one subdirectory per material, and a material is a directory with a density (`CHGCAR` or `fields.h5`) in it.

“Removed configuration section(s)”, or “Replaced: ...” The configuration is written for the schema that preceded 2026-10-13. The message names where each setting went (`training.valid_fraction` is `training.validation.fraction`, `data.data_paths` is `data.path`, `model.paw` is the `ext2chg` section and its `occupancies` block); or start again from `configs/train_simple.yaml`. `EXTCAR` is always computed.

“Nothing to train” The file has neither an `ext2chg`: nor a `chg2tau`: section. An operator is trained when its section is present.

“The PAW occupancies need each element’s projector channels” The occupancies are on (the default) and no `POTCAR` was found to lay the records out from. Give the data one with `data.potcar_dir`, or write `occupancies: false` in the operator’s section to train the grid alone.

“No pseudopotential file found and no -zval given” At inference, the structure directory has no `POTCAR`. Supply one, or pass `-zval Pt=11`.

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

9.2 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 `validation.fraction: 0` they are a training fit. The report’s *What these numbers do not show* section states this explicitly.

9.3 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 `training.batch_size: 1`.

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

9.4 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 optimization amplifies tiny differences — that is not a bug.

9.5 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.