```python
#!/usr/bin/env python
# src/mantra/grn/config.py
"""
Configuration and result containers for the GRN GNN.

GRNModelConfig:
  - architecture choices for GRNGNN + optional trait head
GRNTrainConfig:
  - training loop and optimizer hyperparameters
GRNLossConfig:
  - lambda weights for composite loss terms
GRNResults:
  - minimal container for training history and best-checkpoint metadata
"""

from __future__ import annotations

from dataclasses import dataclass
from typing import Optional

import numpy as np


@dataclass
class GRNModelConfig:
    """
    Architecture config for the GRN GNN (GRNGNN).

    Covers the number of message-passing layers, embedding widths, dropout,
    and whether to include dose inputs and a trait head.
    """

    # Core GNN
    n_layers: int = 3           # number of message-passing layers
    gene_emb_dim: int = 64      # dimensionality of per-gene embeddings
    hidden_dim: int = 128       # hidden width inside GNN layers
    dropout: float = 0.1        # dropout rate applied in GNN layers

    # Input augmentation
    use_dose: bool = False      # if True, model expects a dose covariate per sample

    # Optional trait head on top of program representation
    n_traits: int = 0           # 0 = no trait head; >0 = predict this many traits
    trait_hidden_dim: int = 64 # hidden width inside the trait head MLP


@dataclass
class GRNTrainConfig:
    batch_size: int = 16
    lr: float = 2e-4
    weight_decay: float = 0.0
    max_epochs: int = 50
    grad_clip: float = 1.0
    early_stop_patience: int = 10
    early_stop_min_delta: float = 0.0

    # NEW: cosine LR schedule
    use_cosine_lr: bool = False
    cosine_eta_min: float = 1e-6  # minimum LR



@dataclass
class GRNLossConfig:
    """
    Lambda weights for each loss term in GRN training.

    All lambdas are >= 0; setting a lambda to 0.0 effectively disables that term.
```

```
    """

    lambda_geo: float = 0.0     # geometry prior term (EGGFM energy)
    lambda_prog: float = 0.0    # program-level term (Î\224P_obs via W)
    lambda_trait: float = 0.0   # trait-level supervision term


@dataclass
class GRNResults:
    """
    Container for GRN training history and best-checkpoint metadata.

    This is intentionally minimal and can be extended as needed with
    per-loss curves (expr/geo/prog/trait) or additional diagnostics.
    """

    # Training curves
    train_loss: np.ndarray                  # shape: [n_epochs]
    val_loss: Optional[np.ndarray] = None   # shape: [n_epochs] or None if no val

    # Index of epoch with best validation loss (or final epoch if no val)
    best_epoch: int = -1

    # Configs used for this run (useful for manifesting)
    model_cfg: GRNModelConfig = GRNModelConfig()
    train_cfg: GRNTrainConfig = GRNTrainConfig()
    loss_cfg: GRNLossConfig = GRNLossConfig()
```

```python
# src/mantra/grn/dataset.py

from __future__ import annotations

from pathlib import Path
from typing import Dict, Optional

import numpy as np
import torch
from torch.utils.data import Dataset


class K562RegDeltaDataset(Dataset):
    """
    NPZ-backed dataset for GRN training.

    Expected keys in the .npz:
        - reg_idx:    [N]         int64, regulator index per sample
        - deltaE:     [N, G]      float32, gene-level Î\224E_obs
        - deltaP_obs: [N, K]      float32 (optional)
        - deltaY_obs: [N, T]      float32 (optional)
        - dose:       [N] or [N,1] float32 (optional)

    Notes:
        - n_genes inferred from deltaE.shape[1]
        - n_regulators inferred as max(reg_idx) + 1
    """

    def __init__(self, npz_path: Path) -> None:
        npz_path = Path(npz_path)
        data = np.load(npz_path, allow_pickle=False)

        self.reg_idx = data["reg_idx"].astype(np.int64)
        self.deltaE = data["deltaE"].astype(np.float32)

        self.deltaP_obs = (
            data["deltaP_obs"].astype(np.float32)
            if "deltaP_obs" in data.files
            else None
        )
        self.deltaY_obs = (
            data["deltaY_obs"].astype(np.float32)
            if "deltaY_obs" in data.files
            else None
        )
        self.dose = (
            data["dose"].astype(np.float32)
            if "dose" in data.files
            else None
        )

        self.n_samples = self.reg_idx.shape[0]
        self.n_genes = self.deltaE.shape[1]
        self.n_regulators = int(self.reg_idx.max()) + 1

    def __len__(self) -> int:
        return self.n_samples

    def __getitem__(self, idx: int) -> Dict[str, torch.Tensor]:
        batch: Dict[str, torch.Tensor] = {
            "reg_idx": torch.as_tensor(self.reg_idx[idx], dtype=torch.long),
            "deltaE": torch.from_numpy(self.deltaE[idx]),  # [G]
        }
        if self.deltaP_obs is not None:
            batch["deltaP_obs"] = torch.from_numpy(self.deltaP_obs[idx])
        if self.deltaY_obs is not None:
            batch["deltaY_obs"] = torch.from_numpy(self.deltaY_obs[idx])
        if self.dose is not None:
```

```
            batch["dose"] = torch.as_tensor(self.dose[idx], dtype=torch.float32)
        return batch
```

```python
# src/mantra/grn/inference.py

from __future__ import annotations

from typing import Optional, Dict

import torch
from torch import nn, Tensor

from mantra.grn.models import GRNGNN, TraitHead

class GRNInference:
    """
    Inference wrapper for a trained GRN + energy prior.

    Given:
      - grn_model
      - optional trait_head
      - A (gene graph), x_ref, W (cNMF loadings)
      - energy_prior (HVG or embedding prior)

    Provides:
      - predict_batch(batch): uses same batch dict interface as training
      - predict(reg_idx, dose=None): minimalist convenience for (r,d) â\206\222 Î\224E, Î
\224P, Î\224y, energy
    """
    def __init__(
        self,
        grn_model: GRNGNN,
        A: Tensor,
        x_ref: Tensor,
        W: Tensor,
        energy_prior: nn.Module,
        trait_head: Optional[TraitHead] = None,
        device: Optional[str] = None,
    ) -> None:
        self.device = device or ("cuda" if torch.cuda.is_available() else "cpu")

        self.grn_model = grn_model.to(self.device).eval()
        self.trait_head = trait_head.to(self.device).eval() if trait_head is not None else
None

        self.A = A.to(self.device)
        self.x_ref = x_ref.to(self.device)
        self.W = W.to(self.device)
        self.energy_prior = energy_prior.to(self.device).eval()

        for p in self.grn_model.parameters():
            p.requires_grad_(False)
        if self.trait_head is not None:
            for p in self.trait_head.parameters():
                p.requires_grad_(False)
        for p in self.energy_prior.parameters():
            p.requires_grad_(False)

    @torch.no_grad()
    def predict_batch(self, batch: Dict[str, Tensor]) -> Dict[str, Tensor]:
        """
        batch keys:
            reg_idx: [B]
            dose:    [B] or None
        """
        reg_idx = batch["reg_idx"].to(self.device)
        dose = batch.get("dose", None)
        if dose is not None:
            dose = dose.to(self.device)

        deltaE_pred = self.grn_model(reg_idx=reg_idx, dose=dose, A=self.A)  # [B, G]
```

```
        x_hat = self.x_ref.unsqueeze(0) + deltaE_pred                          # [B, G]
        energy = self.energy_prior(x_hat)                                        # [B]

        deltaP_pred = deltaE_pred @ self.W                                     # [B, K]

        out: Dict[str, Tensor] = {
            "deltaE_pred": deltaE_pred.cpu(),
            "deltaP_pred": deltaP_pred.cpu(),
            "energy": energy.cpu(),
        }

        if self.trait_head is not None:
            deltaY_pred = self.trait_head(deltaP_pred)                         # [B, T]
            out["deltaY_pred"] = deltaY_pred.cpu()

        return out

    @torch.no_grad()
    def predict(
        self,
        reg_idx: Tensor,       # [B] or scalar long
        dose: Optional[Tensor] = None,  # [B] or scalar float, optional
    ) -> Dict[str, Tensor]:
        """
        Convenience wrapper around predict_batch.
        """
        if reg_idx.dim() == 0:
            reg_idx = reg_idx.view(1)
        batch = {"reg_idx": reg_idx}

        if dose is not None:
            if dose.dim() == 0:
                dose = dose.view(1)
            batch["dose"] = dose

        return self.predict_batch(batch)
```

```python
#!/usr/bin/env python
# src/mantra/grn/make_adj.py
from __future__ import annotations

from pathlib import Path
from typing import Any, Dict, Optional

import numpy as np
import scanpy as sc  # type: ignore
import torch


def make_grn_adj(
    ad_path: Path,
    energy_ckpt_path: Path,
    out_path: Path,
    *,
    corr_thresh: float = 0.2,
    topk: Optional[int] = 10,
) -> Dict[str, Any]:
    """
    Build a geneâ\200\223gene adjacency matrix A [G, G] in the EGGFM HVG space.

    The adjacency is constructed from geneâ\200\223gene Pearson correlations
    across cells in `ad`, aligned to the HVG gene ordering in the
    EGGFM checkpoint:

        1) Load EGGFM checkpoint to get HVG gene list / ordering.
        2) Align the AnnData var_names to that ordering.
        3) Compute geneâ\200\223gene Pearson correlations across cells.
        4) Take |corr| as similarity, threshold by corr_thresh.
        5) Optionally keep only top-k neighbors per gene (row-wise).
        6) Row-normalize so each row sums to 1.0, with self-loops
           for isolated genes.

    Parameters
    ----------
    ad_path:
        Path to QC'd unperturbed AnnData (e.g. data/interim/k562_gwps_unperturbed_qc.h5ad)
        that lives in the same gene space as the EGGFM HVGs.
    energy_ckpt_path:
        Path to EGGFM .pt checkpoint with 'var_names' or 'feature_names'
        defining the HVG gene list and ordering.
    out_path:
        Path to .npy file to write adjacency matrix A [G, G].
    corr_thresh:
        Minimum absolute Pearson correlation to keep an edge.
    topk:
        If not None and >0, keep at most top-k non-zero neighbors per
        gene (per row) before row-normalization.

    Returns
    -------
    dict with basic stats:
        - G_eff: number of genes in adjacency (after mapping)
        - corr_thresh
        - topk
        - n_edges: number of non-zero entries in A
        - out_path: path to saved .npy
    """
    ad_path = Path(ad_path)
    energy_ckpt_path = Path(energy_ckpt_path)
    out_path = Path(out_path)
    out_path.parent.mkdir(parents=True, exist_ok=True)

    print(f"[adj] loading AnnData: {ad_path}", flush=True)
    ad = sc.read_h5ad(str(ad_path))
    print(
```

```
        f"[adj] AnnData loaded: n_obs={ad.n_obs}, n_vars={ad.n_vars}",
        flush=True,
    )

    # ---------- 1) Load EGGFM checkpoint to get HVG genes ----------
    print(f"[adj] loading energy checkpoint: {energy_ckpt_path}", flush=True)
    ckpt = torch.load(energy_ckpt_path, map_location="cpu")
    if "var_names" in ckpt:
        hvg_genes = np.array(ckpt["var_names"])
    elif "feature_names" in ckpt:
        hvg_genes = np.array(ckpt["feature_names"])
    else:
        raise KeyError(
            "Checkpoint missing 'var_names'/'feature_names'; "
            "cannot infer HVG gene list for adjacency."
        )

    G_ckpt = hvg_genes.shape[0]
    print(f"[adj] n_HVG from checkpoint: G_ckpt={G_ckpt}", flush=True)

    # ---------- 2) Map HVG genes into AnnData var_names ----------
    var_full = np.array(ad.var_names)
    gene_to_idx: Dict[str, int] = {g: i for i, g in enumerate(var_full)}

    hvg_idx_full = []
    missing_genes = []
    for g in hvg_genes:
        idx = gene_to_idx.get(g)
        if idx is None:
            missing_genes.append(g)
        else:
            hvg_idx_full.append(idx)

    if missing_genes:
        print(
            f"[adj][warn] {len(missing_genes)} HVG genes from checkpoint not in AnnData.var
_names. "
            f"Examples: {missing_genes[:10]}",
            flush=True,
        )

    hvg_idx_full_np = np.array(hvg_idx_full, dtype=int)
    G_eff = int(hvg_idx_full_np.shape[0])
    print(
        f"[adj] using G_eff={G_eff} genes after mapping into AnnData",
        flush=True,
    )
    if G_eff == 0:
        raise RuntimeError(
            "No HVG genes from checkpoint found in AnnData var_names!"
        )

    ad_view = ad[:, hvg_idx_full_np].copy()  # now var_names aligned to EGGFM genes

    # ---------- 3) Materialize expression matrix X [N_cells, G_eff] ----------
    X = ad_view.X
    if not isinstance(X, np.ndarray):
        X = X.toarray()
    X = np.asarray(X, dtype=np.float32)
    n_cells, G = X.shape
    print(f"[adj] X shape for correlation: {X.shape}", flush=True)

    # ---------- 4) Compute geneâ\200\223gene Pearson correlation ----------
    # corr[g1, g2] = corr across cells (columns are genes)
    print("[adj] computing Pearson correlations...", flush=True)
    corr = np.corrcoef(X, rowvar=False)  # [G, G]
    corr = np.nan_to_num(corr, nan=0.0)
```

```
    # ---------- 5) Build adjacency by |corr| + threshold + top-k ----------
    A = np.abs(corr)             # similarity in [0,1]
    np.fill_diagonal(A, 0.0)     # remove self-edge from raw corr

    # threshold
    thresh = float(corr_thresh)
    if thresh > 0.0:
        A[A < thresh] = 0.0
    print(
        f"[adj] applied corr_thresh={thresh:.3f}; "
        f"non-zeros pre-topk={int((A > 0).sum())}",
        flush=True,
    )

    # optional top-k per row
    if topk is not None and int(topk) > 0:
        k = int(topk)
        print(f"[adj] applying top-k sparsification with k={k}", flush=True)
        for i in range(G):
            row = A[i]
            # indices where row > 0
            nz_mask = row > 0
            if nz_mask.sum() > k:
                # kth largest value among non-zero entries
                nz_vals = row[nz_mask]
                cutoff = np.partition(nz_vals, -k)[-k]
                # zero out anything below cutoff
                row[row < cutoff] = 0.0
                A[i] = row

    # ---------- 6) Row-normalize and ensure self-loops for isolated ----------
    row_sums = A.sum(axis=1, keepdims=True)
    isolated = (row_sums[:, 0] == 0.0)

    if isolated.any():
        n_iso = int(isolated.sum())
        print(
            f"[adj] {n_iso} genes were isolated; adding self-loops.",
            flush=True,
        )
        idx_iso = np.where(isolated)[0]
        A[idx_iso, idx_iso] = 1.0  # set diagonal entries for isolated genes
        row_sums = A.sum(axis=1, keepdims=True)

    A = A / row_sums


    n_edges = int((A > 0).sum())
    print(
        f"[adj] final adjacency shape={A.shape}, non-zeros={n_edges}",
        flush=True,
    )

    np.save(out_path, A.astype(np.float32))
    print(f"[adj] saved adjacency to {out_path}", flush=True)

    return {
        "G_eff": G_eff,
        "corr_thresh": thresh,
        "topk": int(topk) if topk is not None else None,
        "n_edges": n_edges,
        "out_path": out_path,
    }
```

```python
# src/mantra/grn/make_npz.py
from __future__ import annotations

from pathlib import Path
from typing import Any, Dict, Optional

import numpy as np
import scanpy as sc  # type: ignore
from scipy import sparse
import torch


def make_grn_npz(
    ad_raw_path: Path,
    energy_ckpt_path: Path,
    out_dir: Path,
    *,
    reg_col: str = "gene",
    dose_col: str = "gem_group",
    control_value: str = "non-targeting",
    max_pct_mt: float = 0.2,
    min_umi: float = 2000.0,
    min_cells_per_group: int = 10,
    val_frac: float = 0.2,
    seed: int = 7,
    cnmf_W_path: Optional[Path] = None,
    traits_dim: int = 3,
) -> Dict[str, Any]:
    """
    Stream K562 GWPS .h5ad and build train/val NPZs in the EGGFM HVG space.

    Parameters
    ----------
    ad_raw_path:
        Path to big backed AnnData (e.g. data/raw/k562_gwps.h5ad).
    energy_ckpt_path:
        Path to EGGFM checkpoint with `var_names` / `feature_names`.
    out_dir:
        Directory where `train.npz` and `val.npz` will be written.
    reg_col:
        obs column with perturbed target gene / regulator.
    dose_col:
        obs column with dose / gem group (ignored here, but kept for API).
    control_value:
        Value in `reg_col` denoting non-targeting controls.
    max_pct_mt:
        Max allowed mitochondrial fraction for QC.
    min_umi:
        Min UMI_count per cell for QC.
    min_cells_per_group:
        Min # cells for a regulator to be kept.
    val_frac:
        Fraction of regulator-level samples for validation.
    seed:
        RNG seed for train/val split.
    cnmf_W_path:
        Optional path to W.npy [G,K]; if None, uses Î\224P_obs = Î\224E.
    traits_dim:
        Dimensionality of Î\224Y_obs stub (e.g. 3 for MCH, RDW, IRF).

    Returns
    -------
    dict with basic stats: N_train, N_val, G, n_regulators_used, etc.
    """
    out_dir = Path(out_dir)
    out_dir.mkdir(parents=True, exist_ok=True)

    # ---------- 1) Get HVG genes from EGGFM checkpoint ----------
```

```
    print(f"[ckpt] loading energy checkpoint: {energy_ckpt_path}", flush=True)
    ckpt = torch.load(energy_ckpt_path, map_location="cpu")
    if "var_names" in ckpt:
        hvg_genes = np.array(ckpt["var_names"])
    elif "feature_names" in ckpt:
        hvg_genes = np.array(ckpt["feature_names"])
    else:
        raise KeyError(
            "Checkpoint missing 'var_names'/'feature_names'; "
            "cannot infer HVG gene list."
        )
    G = hvg_genes.shape[0]
    print(f"[ckpt] n_HVG from checkpoint: G={G}", flush=True)

    # ---------- 2) Open raw AnnData in backed mode ----------
    print(f"[load] raw AnnData (backed): {ad_raw_path}", flush=True)
    ad_raw = sc.read_h5ad(str(ad_raw_path), backed="r")
    n_cells_raw = ad_raw.n_obs
    print(
        f"[info] raw AnnData: n_obs={ad_raw.n_obs}, n_vars={ad_raw.n_vars}",
        flush=True,
    )

    # Map HVG genes to raw var_names
    var_full = np.array(ad_raw.var_names)
    gene_to_idx: Dict[str, int] = {g: i for i, g in enumerate(var_full)}

    hvg_idx_full = []
    missing_genes = []
    for g in hvg_genes:
        idx = gene_to_idx.get(g)
        if idx is None:
            missing_genes.append(g)
        else:
            hvg_idx_full.append(idx)

    if missing_genes:
        print(
            f"[warn] {len(missing_genes)} HVG genes from checkpoint not in raw var_names. "
            f"Examples: {missing_genes[:10]}",
            flush=True,
        )
    hvg_idx_full_np = np.array(hvg_idx_full, dtype=int)
    G_eff = hvg_idx_full_np.shape[0]
    print(f"[info] using G={G_eff} genes after mapping into raw AnnData", flush=True)
    if G_eff == 0:
        raise RuntimeError("No HVG genes from checkpoint found in raw AnnData var_names!")

    # ---------- 3) Build cell-level QC mask from obs ----------
    obs = ad_raw.obs

    if "mitopercent" not in obs.columns:
        raise ValueError(
            "'mitopercent' not found in obs; "
            f"available columns: {list(obs.columns)}"
        )
    if "UMI_count" not in obs.columns:
        raise ValueError(
            "'UMI_count' not found in obs; "
            f"available columns: {list(obs.columns)}"
        )

    mitopercent = obs["mitopercent"].to_numpy()
    umi_count = obs["UMI_count"].to_numpy()

    mito_ok = mitopercent < float(max_pct_mt)
    umi_ok = umi_count > float(min_umi)
```

```
    qc_cells = mito_ok & umi_ok
    print(
        f"[qc] {qc_cells.sum()} / {n_cells_raw} cells pass "
        f"(mitopercent<{max_pct_mt}, UMI_count>{min_umi})",
        flush=True,
    )

    # ---------- 4) Control / perturbed, reg ----------
    if reg_col not in obs.columns:
        raise ValueError(
            f"reg-col '{reg_col}' not in obs; "
            f"available columns: {list(obs.columns)}"
        )
    if dose_col not in obs.columns:
        raise ValueError(
            f"dose-col '{dose_col}' not in obs; "
            f"available columns: {list(obs.columns)}"
        )

    reg_raw = obs[reg_col].to_numpy()
    reg = np.array(reg_raw)

    is_ctrl = (reg == control_value) & qc_cells
    is_pert = (reg != control_value) & qc_cells

    n_ctrl = int(is_ctrl.sum())
    n_pert = int(is_pert.sum())
    print(f"[split] control cells (reg={control_value!r}): {n_ctrl}", flush=True)
    print(f"[split] perturbed cells: {n_pert}", flush=True)
    if n_ctrl == 0:
        raise RuntimeError(
            f"No control cells found with {reg_col} == {control_value!r}"
        )

    # ---------- 5) Global control mean in HVG space (dose-free) ----------
    print("[ctrl] computing GLOBAL control mean in HVG space...", flush=True)

    ctrl_mask = is_ctrl
    n_ctrl_qc = int(ctrl_mask.sum())
    if n_ctrl_qc == 0:
        raise RuntimeError("No control cells after QC filtering!")

    ad_ctrl = ad_raw[ctrl_mask, :].to_memory()
    ad_ctrl_hvg = ad_ctrl[:, hvg_idx_full_np]

    X_ctrl = ad_ctrl_hvg.X
    if sparse.issparse(X_ctrl):
        X_ctrl = X_ctrl.toarray()
    X_ctrl = X_ctrl.astype(np.float32)

    global_ctrl_mean = X_ctrl.mean(axis=0, keepdims=True)  # [1, G_eff]
    print(
        f"[ctrl] global control mean over {n_ctrl_qc} cells; G={global_ctrl_mean.shape[1]}"
,
        flush=True,
    )

    # ---------- 6) Aggregate Î\224E per regulator (ignore dose for K562) ----------
    print("[agg] aggregating Î\224E per regulator (dose-free)...", flush=True)

    regs_pert = np.unique(reg[is_pert])
    print(f"[agg] {len(regs_pert)} unique perturbed regulators", flush=True)

    reg_to_idx = {r: i for i, r in enumerate(regs_pert)}

    deltaE_list = []
    reg_idx_list = []
    dose_list = []
```

```
    min_cells = int(min_cells_per_group)

    for r in regs_pert:
        mask_r = is_pert & (reg == r)
        n = int(mask_r.sum())
        if n < min_cells:
            continue

        ad_r = ad_raw[mask_r, :].to_memory()
        ad_r_hvg = ad_r[:, hvg_idx_full_np]

        X_r = ad_r_hvg.X
        if sparse.issparse(X_r):
            X_r = X_r.toarray()
        X_r = X_r.astype(np.float32)

        x_r_mean = X_r.mean(axis=0, keepdims=True)    # [1, G_eff]
        deltaE = x_r_mean - global_ctrl_mean          # [1, G_eff]

        deltaE_list.append(deltaE)
        reg_idx_list.append(reg_to_idx[r])
        dose_list.append(0.0)  # dummy dose; GRN has use_dose = False

        if len(deltaE_list) % 500 == 0:
            print(
                f"  [agg] processed {len(deltaE_list)} regulators so far...",
                flush=True,
            )

    if len(deltaE_list) == 0:
        raise RuntimeError("No regulators with enough cells after QC!")

    deltaE = np.vstack(deltaE_list).astype(np.float32)
    reg_idx_arr = np.array(reg_idx_list, dtype=np.int64)
    dose_arr = np.array(dose_list, dtype=np.float32)

    N, G_eff = deltaE.shape
    print(f"[agg] built {N} regulator-level samples; each Î\224E has G={G_eff} genes", flus
h=True)

    # ---------- 7) Î\224P_obs via W (optional) ----------
    if cnmf_W_path is not None:
        print(f"[prog] loading cNMF W: {cnmf_W_path}", flush=True)
        W = np.load(cnmf_W_path).astype(np.float32)  # [G_eff, K]
        if W.shape[0] != G_eff:
            raise ValueError(
                f"W has {W.shape[0]} genes but Î\224E has {G_eff}; check HVG alignment."
            )
        deltaP = deltaE @ W  # [N, K]
    else:
        print("[prog] no W provided; using Î\224P_obs = Î\224E", flush=True)
        deltaP = deltaE.copy()

    # ---------- 8) Stub Î\224Y_obs ----------
    T = int(traits_dim)
    deltaY = np.zeros((N, T), dtype=np.float32)

    # ---------- 9) Train/val split ----------
    rng = np.random.default_rng(seed)
    perm = rng.permutation(N)
    N_val = int(val_frac * N)
    val_idx = perm[:N_val]
    train_idx = perm[N_val:]

    def _save_npz(path: Path, idx: np.ndarray) -> None:
        path = Path(path)
        np.savez_compressed(
```

```
            path,
            reg_idx=reg_idx_arr[idx],
            deltaE=deltaE[idx],
            deltaP_obs=deltaP[idx],
            deltaY_obs=deltaY[idx],
            dose=dose_arr[idx],
        )
        print(f"[save] {path} (N={len(idx)})", flush=True)

    train_path = out_dir / "train.npz"
    val_path = out_dir / "val.npz"

    _save_npz(train_path, train_idx)
    _save_npz(val_path, val_idx)
    print("[done]", flush=True)

    return {
        "N": int(N),
        "N_train": int(train_idx.size),
        "N_val": int(val_idx.size),
        "G_eff": int(G_eff),
        "n_regulators_used": int(len(deltaE_list)),
        "train_path": train_path,
        "val_path": val_path,
    }
```

```python
# src/mantra/grn/models.py

from __future__ import annotations

from dataclasses import dataclass
from typing import Optional, Dict

import torch
from torch import nn, Tensor


# ---------------------------------------------------------------------
# 1. Conditioning encoder (regulator Â± dose)
# ---------------------------------------------------------------------

class ConditionEncoder(nn.Module):
    """
    Encodes regulator (always) and optionally dose -> conditioning vector c.

    Shapes:
        reg_idx: [B]  (long)
        dose:    [B] or [B, 1] (float, optional)
        output:  [B, hidden_dim]
    """
    def __init__(
        self,
        n_regulators: int,
        hidden_dim: int = 128,
        reg_dim: int = 64,
        dose_dim: int = 16,
        use_dose: bool = True,
    ) -> None:
        super().__init__()
        self.use_dose = use_dose

        self.reg_embed = nn.Embedding(n_regulators, reg_dim)

        if use_dose:
            self.dose_mlp = nn.Sequential(
                nn.Linear(1, dose_dim),
                nn.ReLU(),
                nn.Linear(dose_dim, dose_dim),
                nn.ReLU(),
            )
            in_dim = reg_dim + dose_dim
        else:
            self.dose_mlp = None
            in_dim = reg_dim

        self.out = nn.Sequential(
            nn.Linear(in_dim, hidden_dim),
            nn.LayerNorm(hidden_dim),
            nn.ReLU(),
        )

    def forward(
        self,
        reg_idx: Tensor,            # [B]
        dose: Optional[Tensor] = None,  # [B] or [B, 1] or None
    ) -> Tensor:
        reg_emb = self.reg_embed(reg_idx)  # [B, reg_dim]

        if self.use_dose:
            if dose is None:
                raise ValueError("dose tensor is required when use_dose=True")
            dose = dose.view(-1, 1)        # [B, 1]
            dose_emb = self.dose_mlp(dose) # [B, dose_dim]
            x = torch.cat([reg_emb, dose_emb], dim=-1)
```

```
        else:
            x = reg_emb

        cond = self.out(x)                  # [B, hidden_dim]
        return cond


# ------------------------------------------------------------------
# 2. FiLM-conditioned GNN layer
# ------------------------------------------------------------------

class GeneGNNLayer(nn.Module):
    """
    Single message-passing layer with FiLM conditioning on global cond vector.

    Inputs:
        h:    [B, G, d_in]  node features
        cond: [B, d_cond]   global condition (reg Â± dose)
        A:    [G, G]        (row- or sym-normalized adjacency)
    """
    def __init__(
        self,
        d_in: int,
        d_out: int,
        d_cond: int,
        dropout: float = 0.0,
    ) -> None:
        super().__init__()
        self.linear = nn.Linear(d_in, d_out)
        self.cond_to_film = nn.Linear(d_cond, 2 * d_out)
        self.norm = nn.LayerNorm(d_out)
        self.act = nn.ReLU()
        self.dropout = nn.Dropout(dropout)

    def forward(
        self,
        h: Tensor,          # [B, G, d_in]
        cond: Tensor,       # [B, d_cond]
        A: Tensor,          # [G, G]
    ) -> Tensor:
        # Message passing: (G,G) x (B,G,d_in) -> (B,G,d_in)
        agg = torch.einsum("ij,bjd->bid", A, h)  # [B, G, d_in]
        h_lin = self.linear(agg)                 # [B, G, d_out]

        # FiLM from global condition
        gamma_beta = self.cond_to_film(cond)     # [B, 2*d_out]
        gamma, beta = gamma_beta.chunk(2, dim=-1)  # [B, d_out] each
        gamma = gamma.unsqueeze(1)               # [B, 1, d_out]
        beta = beta.unsqueeze(1)                 # [B, 1, d_out]
        h_film = gamma * h_lin + beta
        h_norm = self.norm(h_film)
        out = self.act(h_norm)
        out = self.dropout(out)
        return out


# ------------------------------------------------------------------
# 3. GRN GNN model: (reg, dose?) -> Î\224E_pred
# ------------------------------------------------------------------

class GRNGNN(nn.Module):
    """
    f_theta: (reg_idx, dose?) -> Î\224E_pred per gene, conditioned on gene graph.

    Forward:
        reg_idx: [B]      (long)
        dose:    [B] or None
        A:       [G, G]  (normalized adjacency for genes)
```

```
        returns Î\224E_pred: [B, G]
    """
    def __init__(
        self,
        n_regulators: int,
        n_genes: int,
        n_layers: int = 3,
        gene_emb_dim: int = 64,
        hidden_dim: int = 128,
        dropout: float = 0.1,
        use_dose: bool = True,
    ) -> None:
        super().__init__()
        self.n_genes = n_genes
        self.use_dose = use_dose

        # Global condition encoder (reg Â± dose)
        self.cond_encoder = ConditionEncoder(
            n_regulators=n_regulators,
            hidden_dim=hidden_dim,
            reg_dim=hidden_dim // 2,
            dose_dim=hidden_dim // 4,
            use_dose=use_dose,
        )

        # Learnable per-gene initial embeddings h_g^(0)
        self.gene_emb = nn.Parameter(
            0.01 * torch.randn(n_genes, gene_emb_dim)
        )

        # Stack of FiLM-conditioned GNN layers
        layers = []
        d_in = gene_emb_dim
        for _ in range(n_layers):
            layers.append(
                GeneGNNLayer(
                    d_in=d_in,
                    d_out=hidden_dim,
                    d_cond=hidden_dim,
                    dropout=dropout,
                )
            )
            d_in = hidden_dim
        self.layers = nn.ModuleList(layers)

        # Per-gene readout â\206\222 scalar Î\224E_pred
        self.readout = nn.Linear(hidden_dim, 1)

    def forward(
        self,
        reg_idx: Tensor,            # [B]
        dose: Optional[Tensor],     # [B] or None
        A: Tensor,                  # [G, G]
    ) -> Tensor:
        cond = self.cond_encoder(
            reg_idx,
            dose if self.use_dose else None,
        )  # [B, hidden_dim]

        B = reg_idx.shape[0]
        # Broadcast gene embeddings across batch: [B, G, gene_emb_dim]
        h = self.gene_emb.unsqueeze(0).expand(B, self.n_genes, -1)

        # GNN layers
        for layer in self.layers:
            h = layer(h, cond, A)   # [B, G, hidden_dim]
```

```
        # Per-gene linear head â\206\222 Î\224E_pred
        delta_e = self.readout(h).squeeze(-1)    # [B, G]
        return delta_e


# ---------------------------------------------------------------------
# 4. Optional trait head: Î\224P -> Î\224y
# ---------------------------------------------------------------------

class TraitHead(nn.Module):
    """
    Simple MLP mapping program deltas Î\224P -> trait deltas Î\224y.

    Input:
        deltaP: [B, K]
    Output:
        deltaY: [B, T]
    """
    def __init__(
        self,
        n_programs: int,
        n_traits: int,
        hidden_dim: int = 64,
    ) -> None:
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(n_programs, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, n_traits),
        )

    def forward(self, deltaP: Tensor) -> Tensor:
        return self.net(deltaP)


# ---------------------------------------------------------------------
# 5. Loss configuration + loss computation
# ---------------------------------------------------------------------

@dataclass
class GRNLossConfig:
    """
    Lambda weights for each loss term.
    """
    lambda_geo: float = 0.0
    lambda_prog: float = 0.0
    lambda_trait: float = 0.0

def compute_grn_losses(
    model: GRNGNN,
    A: torch.Tensor,                              # [G, G]
    batch: dict[str, torch.Tensor],
    x_ref: torch.Tensor,                          # [G]
    energy_prior: nn.Module,                        # EnergyScorerPrior
    W: torch.Tensor,                              # [G, K]
    loss_cfg: GRNLossConfig,
    trait_head: Optional[nn.Module] = None,
) -> dict[str, torch.Tensor]:
    device = next(model.parameters()).device

    reg_idx = batch["reg_idx"].to(device)       # [B]
    deltaE_obs = batch["deltaE"].to(device)     # [B, G]

    dose = batch.get("dose", None)
    if dose is not None:
        dose = dose.to(device)

    A = A.to(device)
```

```
    x_ref = x_ref.to(device)
    W = W.to(device)

    # 1) Î\224E prediction
    deltaE_pred = model(reg_idx=reg_idx, dose=dose, A=A)  # [B, G]

    # 2) Expression loss
    L_expr = ((deltaE_pred - deltaE_obs) ** 2).mean()

    # 3) Geometric prior (frozen EGGFM)
    x_hat = x_ref.unsqueeze(0) + deltaE_pred   # [B, G]

    # energy at current prediction
    energy = energy_prior(x_hat)               # [B]

    # energy at reference control point (x_ref is already in ckpt)
    with torch.no_grad():
        energy_ref = energy_prior(x_ref.unsqueeze(0)).mean()

    # penalize energy ABOVE reference
    rel_energy = energy - energy_ref           # can be Â±
    rel_energy_pos = torch.relu(rel_energy)    # only push down high-energy states

    L_geo = float(loss_cfg.lambda_geo) * rel_energy_pos.mean()

    # 4) Program-level supervision
    deltaP_pred = deltaE_pred @ W              # [B, K]

    L_prog = torch.zeros((), device=device)
    if "deltaP_obs" in batch:
        deltaP_obs = batch["deltaP_obs"].to(device)
        L_prog = loss_cfg.lambda_prog * ((deltaP_pred - deltaP_obs) ** 2).mean()

    # 5) Trait head (optional)
    L_trait = torch.zeros((), device=device)
    if trait_head is not None and "deltaY_obs" in batch:
        deltaY_obs = batch["deltaY_obs"].to(device)
        deltaY_pred = trait_head(deltaP_pred)
        L_trait = loss_cfg.lambda_trait * ((deltaY_pred - deltaY_obs) ** 2).mean()

    L_total = L_expr + L_geo + L_prog + L_trait

    return {
        "loss": L_total,
        "L_expr": L_expr.detach(),
        "L_geo": L_geo.detach(),
        "L_prog": L_prog.detach(),
        "L_trait": L_trait.detach(),
        "deltaE_pred": deltaE_pred.detach(),
        "deltaP_pred": deltaP_pred.detach(),
    }
```

```python
# src/mantra/grn/priors.py

from __future__ import annotations

from pathlib import Path
from typing import Optional, Sequence

import torch
from torch import nn, Tensor

from mantra.eggfm.inference import EnergyScorer


class EnergyScorerPrior(nn.Module):
    """
    Wraps an EnergyScorer as a frozen prior: x_hat -> energy.

    GRN does not care whether the underlying energy lives in HVG
    space or an embedding; that logic is inside EnergyScorer.
    """

    def __init__(
        self,
        scorer: EnergyScorer,
        gene_names: Optional[Sequence[str]] = None,
    ) -> None:
        super().__init__()
        self.scorer = scorer
        # optional canonical gene order for GRN's feature space
        self.gene_names = list(gene_names) if gene_names is not None else None

    def forward(self, x_hat: Tensor) -> Tensor:
        # x_hat: [B, G_raw] in GRN's gene space
        return self.scorer.score(x_hat, gene_names=self.gene_names)


def build_energy_prior_from_ckpt(
    ckpt_path: str | Path,
    gene_names: Optional[Sequence[str]],
    device: Optional[torch.device] = None,
) -> EnergyScorerPrior:
    """
    Build an EnergyScorerPrior from a pre-trained EGGFM checkpoint.

    The checkpoint itself encodes:
      - HVG vs embedding space
      - normalization
      - (for embedding) projection matrix.
    """
    scorer = EnergyScorer.from_checkpoint(ckpt_path, device=device)
    prior = EnergyScorerPrior(scorer=scorer, gene_names=gene_names)
    prior.eval()
    return prior
```

```python
# src/mantra/grn/run_grn.py
from __future__ import annotations

from pathlib import Path
from typing import Dict, Any, Optional

import numpy as np
import torch
import yaml
import scanpy as sc
from scipy import sparse as sp_sparse
from torch.utils.data import DataLoader

from mantra.grn.config import GRNModelConfig, GRNTrainConfig, GRNLossConfig
from mantra.grn.dataset import K562RegDeltaDataset
from mantra.grn.models import GRNGNN, TraitHead
from mantra.grn.priors import build_energy_prior_from_ckpt
from mantra.grn.trainer import GRNTrainer


def run_grn_training(
    params_path: Path,
    out_dir: Path,
    ad_path: Path,
    train_npz_path: Path,
    val_npz_path: Optional[Path],
    energy_ckpt_path: Path,
    adj_path: Optional[Path] = None,
    cnmf_W_path: Optional[Path] = None,
) -> Path:
    """
    High-level entrypoint to train the GRN GNN with an EGGFM energy prior.

    Parameters
    ----------
    params_path : Path
        YAML file with grn_model, grn_train, grn_loss blocks.
    out_dir : Path
        Directory to write GRN checkpoint(s).
    ad_path : Path
        QCâ\200\231d AnnData used to compute x_ref (same gene space as Î\224E / energy).
    train_npz_path : Path
        NPZ with aggregated (reg_idx, deltaE, deltaP_obs, deltaY_obs, dose).
    val_npz_path : Optional[Path]
        Optional NPZ for validation set.
    energy_ckpt_path : Path
        Pre-trained EGGFM energy checkpoint (.pt).
    adj_path : Optional[Path]
        Optional adjacency matrix [G,G] as .npy. If None, uses identity.
    cnmf_W_path : Optional[Path]
        Optional cNMF loadings W [G,K]. If None, uses identity [G,G].

    Returns
    -------
    Path
        Path to the saved GRN checkpoint.
    """
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    out_dir.mkdir(parents=True, exist_ok=True)

    # ---- load params ----
    params: Dict[str, Any] = yaml.safe_load(params_path.read_text())
    grn_model_cfg = GRNModelConfig(**params.get("grn_model", {}))
    grn_train_cfg = GRNTrainConfig(**params.get("grn_train", {}))
    grn_loss_cfg = GRNLossConfig(**params.get("grn_loss", {}))

    # ---- load AnnData ----
```

```
    qc_ad = sc.read_h5ad(str(ad_path))

    # ---- datasets ----
    train_ds = K562RegDeltaDataset(train_npz_path)
    val_ds: Optional[K562RegDeltaDataset] = (
        K562RegDeltaDataset(val_npz_path) if val_npz_path is not None else None
    )

    G = train_ds.n_genes
    n_regulators = train_ds.n_regulators

    # ---- energy checkpoint (and enforce HVG space) ----
    ckpt = torch.load(energy_ckpt_path, map_location="cpu")
    space = ckpt.get("space", "hvg")
    if space != "hvg":
        raise ValueError(
            f"Energy checkpoint space={space!r}; GRN currently expects an "
            "EGGFM prior trained directly in HVG gene space (space='hvg'). "
            "Use an HVG-space checkpoint when building NPZs and training GRN."
        )

    hvg_names = np.array(ckpt["var_names"])
    if hvg_names.shape[0] != G:
        raise ValueError(
            f"Energy ckpt var_names has {hvg_names.shape[0]} genes, "
            f"but Î\224E has {G}. These must match."
        )

    # ---- adjacency ----
    if adj_path is not None:
        A_np = np.load(adj_path).astype(np.float32)
    else:
        A_np = np.eye(G, dtype=np.float32)
    A = torch.from_numpy(A_np).to(device)

    # ---- cNMF W ----
    if cnmf_W_path is not None:
        W_np = np.load(cnmf_W_path).astype(np.float32)  # [G,K]
    else:
        # identity: effectively disables program loss when lambda_prog > 0
        W_np = np.eye(G, dtype=np.float32)
    W = torch.from_numpy(W_np).to(device)

    # ---- reference state x_ref ----
    # We need x_ref in the SAME gene space as Î\224E and the energy prior (G genes).

    # 1) Load HVG names from the energy checkpoint
    ckpt = torch.load(energy_ckpt_path, map_location="cpu")
    hvg_names = np.array(ckpt["var_names"])
    if hvg_names.shape[0] != G:
        raise ValueError(
            f"Energy ckpt var_names has {hvg_names.shape[0]} genes, "
            f"but Î\224E has {G}. These must match."
        )

    # 2) Align qc_ad.var_names to this list
    var_names = np.array(qc_ad.var_names.astype(str))
    gene_to_idx = {g: i for i, g in enumerate(var_names)}

    missing = [g for g in hvg_names if g not in gene_to_idx]
    if missing:
        raise ValueError(
            "Could not align qc_ad genes to energy ckpt/NPZ space: "
            f"{len(missing)} genes missing. Examples: {missing[:10]}"
        )

    idx = np.array([gene_to_idx[g] for g in hvg_names], dtype=int)
    qc_ad_sub = qc_ad[:, idx].copy()
```

```
    print(
        f"[align] subset qc_ad from {qc_ad.n_vars} â\206\222 {qc_ad_sub.n_vars} genes "
        f"to match Î\224E / energy prior space.",
        flush=True,
    )

    # 3) Compute x_ref in this aligned space
    X = qc_ad_sub.X
    if sp_sparse.issparse(X):
        X = X.toarray()
    X = np.asarray(X, dtype=np.float32)

    x_ref_np = X.mean(axis=0)  # [G]
    if x_ref_np.shape[0] != G:
        raise ValueError(
            "Gene dimension mismatch after alignment: "
            f"x_ref has {x_ref_np.shape[0]} genes, but Î\224E has {G}."
        )
    x_ref = torch.from_numpy(x_ref_np).to(device)

    # ---- dataloaders ----
    train_loader = DataLoader(
        train_ds,
        batch_size=grn_train_cfg.batch_size,
        shuffle=True,
    )
    val_loader = None
    if val_ds is not None:
        val_loader = DataLoader(
            val_ds,
            batch_size=grn_train_cfg.batch_size,
            shuffle=False,
        )

    # ---- energy prior (pretrained EGGFM) ----
    energy_prior = build_energy_prior_from_ckpt(
        ckpt_path=str(energy_ckpt_path),
        gene_names=qc_ad.var_names,
        device=device,
    )

    # ---- GRN model ----
    model = GRNGNN(
        n_regulators=n_regulators,
        n_genes=G,
        n_layers=grn_model_cfg.n_layers,
        gene_emb_dim=grn_model_cfg.gene_emb_dim,
        hidden_dim=grn_model_cfg.hidden_dim,
        dropout=grn_model_cfg.dropout,
        use_dose=grn_model_cfg.use_dose,
    ).to(device)

    # ---- optional trait head ----
    trait_head: Optional[TraitHead] = None
    if grn_model_cfg.n_traits > 0:
        K = W_np.shape[1]
        trait_head = TraitHead(
            n_programs=K,
            n_traits=grn_model_cfg.n_traits,
            hidden_dim=grn_model_cfg.trait_hidden_dim,
        ).to(device)

    # ---- trainer ----
    trainer = GRNTrainer(
        grn_model=model,
        trait_head=trait_head,
        A=A,
        x_ref=x_ref,
```

```
        W=W,
        energy_prior=energy_prior,
        loss_cfg=grn_loss_cfg,
        train_cfg=grn_train_cfg,
        device=str(device),
    )

    trainer.fit(train_loader, val_loader)

    # ---- save best checkpoint ----
    ckpt_out = {
        "model_state_dict": (
            trainer.best_model_state
            if getattr(trainer, "best_model_state", None) is not None
            else model.state_dict()
        ),
        "trait_head_state_dict": (
            trainer.best_trait_state if trait_head is not None else None
        ),
        "grn_model_cfg": grn_model_cfg.__dict__,
        "grn_train_cfg": grn_train_cfg.__dict__,
        "grn_loss_cfg": grn_loss_cfg.__dict__,
        "n_regulators": n_regulators,
        "n_genes": G,
        "W": W_np,
        "A": A_np,
        "x_ref": x_ref_np,
        # --- prior metadata for later reload ---
        "prior_type": "eggfm_energy",
        "energy_ckpt_path": str(energy_ckpt_path.resolve()),
        "energy_var_names": hvg_names,  # np.array/list of gene IDs in this space
    }

    ckpt_path = out_dir / "grn_k562_energy_prior.pt"
    torch.save(ckpt_out, ckpt_path)
    print(f"Saved GRN checkpoint to {ckpt_path}", flush=True)

    return ckpt_path
```

```python
# src/mantra/grn/trainer.py

from __future__ import annotations

from dataclasses import dataclass
from typing import Dict, Optional

import torch
from torch import nn, optim
from torch.utils.data import DataLoader

from mantra.grn.models import GRNGNN, TraitHead, GRNLossConfig, compute_grn_losses
from mantra.grn.config import GRNTrainConfig

class GRNTrainer:
    """
    Trainer for the GRN GNN block with an arbitrary energy prior.

    Usage:
        trainer = GRNTrainer(
            grn_model=grn,
            trait_head=trait_head,
            A=A,
            x_ref=x_ref,
            W=W,
            energy_prior=hvg_prior or embed_prior,
            loss_cfg=loss_cfg,
            train_cfg=train_cfg,
            device="cuda",
        )
        trainer.fit(train_loader, val_loader)
    """
    def __init__(
        self,
        grn_model: GRNGNN,
        trait_head: Optional[TraitHead],
        A,
        x_ref,
        W,
        energy_prior: nn.Module,
        loss_cfg: GRNLossConfig,
        train_cfg: GRNTrainConfig,
        device: Optional[str] = None,
    ) -> None:
        self.device = device or ("cuda" if torch.cuda.is_available() else "cpu")

        self.grn_model = grn_model.to(self.device)
        self.trait_head = trait_head.to(self.device) if trait_head is not None else None

        self.A = A.to(self.device)
        self.x_ref = x_ref.to(self.device)
        self.W = W.to(self.device)

        self.energy_prior = energy_prior.to(self.device)
        self.loss_cfg = loss_cfg
        self.train_cfg = train_cfg

        params = list(self.grn_model.parameters())
        if self.trait_head is not None:
            params += list(self.trait_head.parameters())

        self.optimizer = optim.Adam(
            params,
            lr=float(train_cfg.lr),          # force-cast in case YAML gave a string
            weight_decay=train_cfg.weight_decay,
        )

        self.scheduler = None
```

```
        if getattr(train_cfg, "use_cosine_lr", False):
            self.scheduler = optim.lr_scheduler.CosineAnnealingLR(
                self.optimizer,
                T_max=train_cfg.max_epochs,
                eta_min=float(train_cfg.cosine_eta_min),
            )

    # ------------------- public API -------------------

    def fit(
        self,
        train_loader: DataLoader,
        val_loader: Optional[DataLoader] = None,
    ) -> None:
        cfg = self.train_cfg

        best_val_loss = float("inf")
        best_state = None
        epochs_without_improve = 0

        for epoch in range(cfg.max_epochs):
            train_stats = self.train_epoch(train_loader)

            # log current LR
            curr_lr = self.optimizer.param_groups[0]["lr"]
            msg = (
                f"[GRN] Epoch {epoch+1}/{cfg.max_epochs} "
                f"train_loss={train_stats['loss']:.4f} "
                f"expr={train_stats['L_expr']:.4f} "
                f"geo={train_stats['L_geo']:.4f} "
                f"prog={train_stats['L_prog']:.4f} "
                f"trait={train_stats['L_trait']:.4f} "
                f"lr={curr_lr:.2e}"
            )

            if val_loader is not None:
                val_stats = self.eval_epoch(val_loader)
                val_loss = val_stats["loss"]
                val_expr = val_stats["L_expr"]

                msg += f" | val_loss={val_loss:.4f} val_expr={val_expr:.4f}"

                # early stopping on expression only (as you have)
                improved = val_expr + cfg.early_stop_min_delta < best_val_loss
                if improved:
                    best_val_loss = val_expr
                    best_state = self._snapshot_state()
                    self.best_model_state = best_state["grn"]
                    self.best_trait_state = best_state.get("trait_head")
                    epochs_without_improve = 0
                else:
                    epochs_without_improve += 1
            else:
                train_expr = train_stats["L_expr"]
                improved = train_expr + cfg.early_stop_min_delta < best_val_loss
                if improved:
                    best_val_loss = train_expr
                    best_state = self._snapshot_state()
                    self.best_model_state = best_state["grn"]
                    self.best_trait_state = best_state.get("trait_head")
                epochs_without_improve = 0

            print(msg, flush=True)

            # NEW: step scheduler once per epoch
            if self.scheduler is not None:
                self.scheduler.step()
```

```
            if (
                val_loader is not None
                and cfg.early_stop_patience > 0
                and epochs_without_improve >= cfg.early_stop_patience
            ):
                print(
                    f"[GRN] Early stopping at epoch {epoch+1} "
                    f"(best_val_loss={best_val_loss:.4f})",
                    flush=True,
                )
                break

        if best_state is not None:
            self._load_state(best_state)

    def train_epoch(self, loader: DataLoader) -> Dict[str, float]:
        self.grn_model.train()
        if self.trait_head is not None:
            self.trait_head.train()

        total = 0
        sum_loss = sum_expr = sum_geo = sum_prog = sum_trait = 0.0

        for batch in loader:
            stats = self._forward_batch(batch, train_mode=True)

            bsz = batch["reg_idx"].shape[0]
            total += bsz
            sum_loss  += stats["loss"].item()    * bsz
            sum_expr  += stats["L_expr"].item()  * bsz
            sum_geo   += stats["L_geo"].item()   * bsz
            sum_prog  += stats["L_prog"].item()  * bsz
            sum_trait += stats["L_trait"].item() * bsz

        return {
            "loss":   sum_loss  / total,
            "L_expr": sum_expr  / total,
            "L_geo":  sum_geo   / total,
            "L_prog": sum_prog  / total,
            "L_trait":sum_trait / total,
        }

    @torch.no_grad()
    def eval_epoch(self, loader: DataLoader) -> Dict[str, float]:
        self.grn_model.eval()
        if self.trait_head is not None:
            self.trait_head.eval()

        total = 0
        sum_loss = sum_expr = sum_geo = sum_prog = sum_trait = 0.0

        for batch in loader:
            stats = self._forward_batch(batch, train_mode=False)

            bsz = batch["reg_idx"].shape[0]
            total += bsz
            sum_loss  += stats["loss"].item()    * bsz
            sum_expr  += stats["L_expr"].item()  * bsz
            sum_geo   += stats["L_geo"].item()   * bsz
            sum_prog  += stats["L_prog"].item()  * bsz
            sum_trait += stats["L_trait"].item() * bsz

        return {
            "loss":   sum_loss  / total,
            "L_expr": sum_expr  / total,
            "L_geo":  sum_geo   / total,
            "L_prog": sum_prog  / total,
            "L_trait":sum_trait / total,
```

```
        }

    # ------------------- internals -------------------

    def _forward_batch(self, batch: Dict[str, torch.Tensor], train_mode: bool) -> Dict[str,
 torch.Tensor]:
        batch = {k: v.to(self.device) for k, v in batch.items()}

        out = compute_grn_losses(
            model=self.grn_model,
            A=self.A,
            batch=batch,
            x_ref=self.x_ref,
            energy_prior=self.energy_prior,
            W=self.W,
            loss_cfg=self.loss_cfg,
            trait_head=self.trait_head,
        )
        loss = out["loss"]

        if train_mode:
            self.optimizer.zero_grad()
            loss.backward()
            if self.train_cfg.grad_clip > 0.0:
                torch.nn.utils.clip_grad_norm_(
                    list(self.grn_model.parameters())
                    + ([] if self.trait_head is None else list(self.trait_head.parameters()
)),
                    self.train_cfg.grad_clip,
                )
            self.optimizer.step()

        return out

    def _snapshot_state(self):
        state = {
            "grn": self.grn_model.state_dict(),
        }
        if self.trait_head is not None:
            state["trait_head"] = self.trait_head.state_dict()
        return state

    def _load_state(self, state):
        self.grn_model.load_state_dict(state["grn"])
        if self.trait_head is not None and "trait_head" in state:
            self.trait_head.load_state_dict(state["trait_head"])
```

```python
# src/mantra/grn/__init__.py
from __future__ import annotations

from .models import (
    ConditionEncoder,
    GeneGNNLayer,
    GRNGNN,
    TraitHead,
    compute_grn_losses,
)
from .trainer import GRNTrainer
from .config import GRNModelConfig, GRNTrainConfig, GRNLossConfig

__all__ = [
    "ConditionEncoder",
    "GeneGNNLayer",
    "GRNGNN",
    "TraitHead",
    "compute_grn_losses",
    "GRNTrainer",
    "GRNModelConfig",
    "GRNTrainConfig",
    "GRNLossConfig",
]
```