```
#!/usr/bin/env python3
# src/mantra/scripts/cnmf.py
"""
Run consensus NMF (cNMF) on a QC'd AnnData object to obtain
gene-program loadings W [G, K] and related artifacts.

This script:
  - loads a QC'd AnnData
  - optionally aligns genes to an EGGFM energy checkpoint (for Î\224E / NPZ compatibility)
  - builds a CNMFConfig from YAML + CLI overrides
  - runs cNMF
  - saves W_consensus, all per-run programs, cluster labels, run coverage, and manifest

Typical usage:

  python scripts/cnmf.py \
      --ad data/interim/k562_gwps_unperturbed_qc.h5ad \
      --out out/programs/k562_cnmf_hvg75 \
      --k 75 \
      --use-hvg-only \
      --n-restarts 20 \
      --seed 7 \
      --name k562_cnmf_hvg75 \
      --energy-ckpt out/models/eggfm/eggfm_energy_k562_hvg_hvg75.pt
"""

from __future__ import annotations

import argparse
from pathlib import Path
from typing import Any, Dict, Tuple

import numpy as np
import scanpy as sc
import torch
import yaml

from mantra.programs.cnmf import run_cnmf, save_cnmf_result
from mantra.programs.config import CNMFConfig


# ---------------------------------------------------------------------
# CLI / config glue
# ---------------------------------------------------------------------


def build_argparser() -> argparse.ArgumentParser:
    p = argparse.ArgumentParser(
        description=(
            "Run consensus NMF on a QC'd AnnData object to obtain "
            "gene-program loadings W [G,K] and related artifacts."
        )
    )

    p.add_argument(
        "--ad",
        type=str,
        required=True,
        help="Input QC AnnData (e.g. data/interim/k562_gwps_unperturbed_qc.h5ad)",
    )
    p.add_argument(
        "--out",
        type=str,
        required=True,
        help="Output directory for W arrays and manifest.",
    )
    p.add_argument(
        "--params",
        type=str,
        default=None,
        help="Optional YAML params file; if provided, uses the 'cnmf' section as defaults."
,
    )

    # core CNMF config (CLI overrides YAML)
    p.add_argument(
        "--k",
        type=int,
        default=None,
        help="Number of programs / components (rank K). Overrides cnmf.n_components.",
    )
    p.add_argument(
        "--use-hvg-only",
        action="store_true",
        help="Restrict to ad.var[hvg_key]==True if present. Overrides cnmf.use_hvg_only.",
    )
    p.add_argument(
        "--n-restarts",
        type=int,
        default=None,
        help="Number of independent NMF runs. Overrides cnmf.n_restarts.",
    )
    p.add_argument(
        "--max-iter",
        type=int,
        default=None,
        help="Max iterations for NMF per run. Overrides cnmf.max_iter.",
    )
    p.add_argument(
        "--tol",
        type=float,
        default=None,
        help="Convergence tolerance for NMF. Overrides cnmf.tol.",
    )
    p.add_argument(
        "--alpha",
        type=float,
        default=None,
        help="Overall regularization strength for NMF. Overrides cnmf.alpha.",
    )
    p.add_argument(
        "--l1-ratio",
        type=float,
        default=None,
        help="Mix between L1 (1.0) and L2 (0.0). Overrides cnmf.l1_ratio.",
    )
    p.add_argument(
        "--seed",
        type=int,
        default=None,
        help="Random seed for NMF + consensus. Overrides cnmf.random_state.",
    )
    p.add_argument(
        "--name",
        type=str,
        default="k562_cnmf",
        help="Short name used as prefix for output files.",
    )

    # optional alignment to EGGFM / NPZ gene space
    p.add_argument(
        "--energy-ckpt",
        type=str,
        default=None,
        help=(
            "Optional EGGFM checkpoint (.pt). If provided, subset AnnData to "
            "ckpt['var_names'] (or 'feature_names') so W rows align with Î\224E / energy pr
ior."
        ),
    )

    return p
def make_cfg(args: argparse.Namespace) -> CNMFConfig:
    """
    Merge YAML cnmf section (if present) with CLI overrides
    into a CNMFConfig dataclass.
    """
    base: Dict[str, Any] = {}
    if args.params is not None:
        base = yaml.safe_load(Path(args.params).read_text()).get("cnmf", {}) or {}

    default_cfg = CNMFConfig()

    def pick(field: str, cli_value, default):
        # CLI (if not None) overrides YAML; else YAML; else dataclass default
        if cli_value is not None:
            return cli_value
        if field in base:
            return base[field]
        return default

    # --- special-case the boolean flag use_hvg_only ---
    # YAML (or dataclass) defines the baseline; CLI can only turn it ON.
    use_hvg_only_base = base.get("use_hvg_only", default_cfg.use_hvg_only)
    use_hvg_only = use_hvg_only_base or bool(args.use_hvg_only)

    cfg = CNMFConfig(
        n_components=pick("n_components", args.k, default_cfg.n_components),
        n_restarts=pick("n_restarts", args.n_restarts, default_cfg.n_restarts),
        max_iter=pick("max_iter", args.max_iter, default_cfg.max_iter),
        tol=pick("tol", args.tol, default_cfg.tol),

        # use the special-cased value
        use_hvg_only=use_hvg_only,

        hvg_key=base.get("hvg_key", default_cfg.hvg_key),
        min_cells_per_gene=base.get(
            "min_cells_per_gene", default_cfg.min_cells_per_gene
        ),
        scale_cells=base.get("scale_cells", default_cfg.scale_cells),
        alpha=pick("alpha", args.alpha, default_cfg.alpha),
        l1_ratio=pick("l1_ratio", args.l1_ratio, default_cfg.l1_ratio),
        consensus_kmeans_n_init=base.get(
            "consensus_kmeans_n_init", default_cfg.consensus_kmeans_n_init
        ),
        filter_by_coverage=base.get(
            "filter_by_coverage", default_cfg.filter_by_coverage
        ),
        min_run_coverage=base.get(
            "min_run_coverage", default_cfg.min_run_coverage
        ),
        random_state=pick("random_state", args.seed, default_cfg.random_state),
    )
    return cfg

# ---------------------------------------------------------------------
# Optional alignment to energy checkpoint gene space
# ---------------------------------------------------------------------


def subset_to_energy_genes(
    ad: sc.AnnData,
    energy_ckpt_path: str,
) -> Tuple[sc.AnnData, np.ndarray]:
    """
    Subset AnnData to the genes (and order) used by the EGGFM energy checkpoint.

    This ensures:
      - W rows == #genes in Î\224E == len(ckpt.var_names)
      - ordering matches the Î\224E / energy prior space.
    """
    print(f"[CNMF] Aligning genes to energy checkpoint: {energy_ckpt_path}", flush=True)
    ckpt = torch.load(energy_ckpt_path, map_location="cpu")
    if "var_names" in ckpt:
        gene_names = np.array(ckpt["var_names"])
    elif "feature_names" in ckpt:
        gene_names = np.array(ckpt["feature_names"])
    else:
        raise KeyError(
            "Energy checkpoint missing 'var_names'/'feature_names'; "
            "cannot align genes."
        )

    var_names = np.array(ad.var_names.astype(str))
    gene_to_idx = {g: i for i, g in enumerate(var_names)}

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

    idx = np.array([gene_to_idx[g] for g in gene_names], dtype=int)
    ad_sub = ad[:, idx].copy()
    print(
        f"[CNMF] Subset AnnData from {ad.n_vars} â\206\222 {ad_sub.n_vars} genes "
        f"to match energy ckpt ordering.",
        flush=True,
    )
    return ad_sub, gene_names


# ----------------------------------------------------------------------
# Main
# ----------------------------------------------------------------------


def main() -> None:
    args = build_argparser().parse_args()

    ad_path = Path(args.ad)
    out_dir = Path(args.out)

    print(f"[CNMF] Loading AnnData from {ad_path} ...", flush=True)
    ad = sc.read_h5ad(ad_path)
    print(f"[CNMF] ad.n_obs={ad.n_obs}, ad.n_vars={ad.n_vars}", flush=True)

    # Optional alignment to EGGFM / NPZ gene space before running CNMF
    energy_genes: np.ndarray | None = None
    if args.energy_ckpt is not None:
        ad, energy_genes = subset_to_energy_genes(ad, args.energy_ckpt)
        print(
            f"[CNMF] Energy-aligned gene space: {energy_genes.shape[0]} genes",
            flush=True,
        )

    cfg = make_cfg(args)
    print(f"[CNMF] Config: {cfg}", flush=True)

    result = run_cnmf(ad, cfg)
    save_cnmf_result(out_dir, ad, result, prefix=args.name)

    if energy_genes is not None:
        genes_path = out_dir / f"{args.name}_genes_aligned.npy"
        np.save(genes_path, energy_genes)
        print(f"[CNMF] Saved aligned gene list to {genes_path}", flush=True)


if __name__ == "__main__":
    main()
```

```
#!/usr/bin/env python
# scripts/eval_grn.py

from __future__ import annotations

import argparse
from pathlib import Path
from types import SimpleNamespace
from typing import Dict

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

from mantra.grn.dataset import K562RegDeltaDataset
from mantra.grn.models import GRNGNN, TraitHead, compute_grn_losses
from mantra.grn.priors import build_energy_prior_from_ckpt


def load_grn_from_checkpoint(
    ckpt_path: Path,
    device: torch.device,
):
    """
    Reconstruct GRN model, optional TraitHead, adjacency A, W, x_ref,
    energy_prior, and loss_cfg from a saved GRN checkpoint.
    """
    ckpt = torch.load(ckpt_path, map_location=device)

    # ----- basic config -----
    model_cfg = ckpt["grn_model_cfg"]           # dict
    loss_cfg_dict = ckpt["grn_loss_cfg"]        # dict
    n_regulators = int(ckpt["n_regulators"])
    n_genes = int(ckpt["n_genes"])

    # ----- core GRN model -----
    model = GRNGNN(
        n_regulators=n_regulators,
        n_genes=n_genes,
        n_layers=model_cfg.get("n_layers", 3),
        gene_emb_dim=model_cfg.get("gene_emb_dim", 64),
        hidden_dim=model_cfg.get("hidden_dim", 128),
        dropout=model_cfg.get("dropout", 0.1),
        use_dose=model_cfg.get("use_dose", False),
    ).to(device)
    model.load_state_dict(ckpt["model_state_dict"])

    # ----- optional trait head -----
    trait_head = None
    n_traits = model_cfg.get("n_traits", 0)
    if n_traits > 0 and ckpt.get("trait_head_state_dict") is not None:
        W_np = ckpt["W"]
        K = W_np.shape[1]
        trait_head = TraitHead(
            n_programs=K,
            n_traits=n_traits,
            hidden_dim=model_cfg.get("trait_hidden_dim", 64),
        ).to(device)
        trait_head.load_state_dict(ckpt["trait_head_state_dict"])

    # ----- A, W, x_ref -----
    A_np = ckpt["A"].astype(np.float32)
    W_np = ckpt["W"].astype(np.float32)
    x_ref_np = ckpt["x_ref"].astype(np.float32)

    A = torch.from_numpy(A_np).to(device)
    W = torch.from_numpy(W_np).to(device)
    x_ref = torch.from_numpy(x_ref_np).to(device)
    # ----- energy prior -----
    energy_ckpt_path = ckpt["energy_ckpt_path"]
    energy_var_names = ckpt["energy_var_names"]
    energy_prior = build_energy_prior_from_ckpt(
        ckpt_path=energy_ckpt_path,
        gene_names=energy_var_names,
        device=device,
    )

    # ----- loss config -----
    # SimpleNamespace is enough; compute_grn_losses just reads attributes
    loss_cfg = SimpleNamespace(**loss_cfg_dict)

    return model, trait_head, A, W, x_ref, energy_prior, loss_cfg


def eval_split(
    name: str,
    npz_path: Path,
    model: GRNGNN,
    trait_head: TraitHead | None,
    A: torch.Tensor,
    x_ref: torch.Tensor,
    W: torch.Tensor,
    energy_prior: torch.nn.Module,
    loss_cfg,
    device: torch.device,
    batch_size: int = 256,
) -> Dict[str, float]:
    """
    Evaluate a trained GRN on a given NPZ split and return
    averaged metrics: loss, L_expr, L_geo, L_prog, L_trait.
    """
    ds = K562RegDeltaDataset(npz_path)
    loader = DataLoader(ds, batch_size=batch_size, shuffle=False)

    model.eval()
    if trait_head is not None:
        trait_head.eval()

    totals = {
        "loss": 0.0,
        "L_expr": 0.0,
        "L_geo": 0.0,
        "L_prog": 0.0,
        "L_trait": 0.0,
    }
    n_samples = 0

    with torch.no_grad():
        for batch in loader:
            batch = {k: v.to(device) for k, v in batch.items()}
            out = compute_grn_losses(
                model=model,
                A=A,
                batch=batch,
                x_ref=x_ref,
                energy_prior=energy_prior,
                W=W,
                loss_cfg=loss_cfg,
                trait_head=trait_head,
            )
            bsz = batch["reg_idx"].shape[0]
            n_samples += bsz

            totals["loss"] += out["loss"].item() * bsz
            totals["L_expr"] += out["L_expr"].item() * bsz
            totals["L_geo"] += out["L_geo"].item() * bsz
            totals["L_prog"] += out["L_prog"].item() * bsz
            totals["L_trait"] += out["L_trait"].item() * bsz

    metrics = {k: v / n_samples for k, v in totals.items()}
    print(f"[{name}] N={n_samples} | " +
          " ".join(f"{k}={metrics[k]:.4f}" for k in metrics))
    return metrics


def main() -> None:
    ap = argparse.ArgumentParser(
        description="Evaluate a trained GRN checkpoint on train/val NPZ splits."
    )
    ap.add_argument(
        "--ckpt",
        type=str,
        required=True,
        help="Path to GRN checkpoint (.pt), e.g. out/models/grn/hvg75/grn_k562_energy_prior
.pt",
    )
    ap.add_argument(
        "--train-npz",
        type=str,
        required=True,
        help="Train NPZ path, e.g. data/interim/grn_k562_gwps_hvg75_npz/train.npz",
    )
    ap.add_argument(
        "--val-npz",
        type=str,
        default=None,
        help="Optional val NPZ path, e.g. data/interim/grn_k562_gwps_hvg75_npz/val.npz",
    )
    ap.add_argument(
        "--batch-size",
        type=int,
        default=256,
        help="Batch size for evaluation",
    )

    args = ap.parse_args()

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    model, trait_head, A, W, x_ref, energy_prior, loss_cfg = load_grn_from_checkpoint(
        Path(args.ckpt),
        device=device,
    )

    # Train split metrics
    eval_split(
        "train",
        Path(args.train_npz),
        model,
        trait_head,
        A,
        x_ref,
        W,
        energy_prior,
        loss_cfg,
        device=device,
        batch_size=args.batch_size,
    )

    # Val split metrics (if provided)
    if args.val_npz is not None:
        eval_split(
            "val",
            Path(args.val_npz),
            model,
            trait_head,
            A,
            x_ref,
            W,
            energy_prior,
            loss_cfg,
            device=device,
            batch_size=args.batch_size,
        )


if __name__ == "__main__":
    main()
```

```python
#!/usr/bin/env python
# src/mantra/scripts/hvg_embed.py
"""
Compute manifold embeddings on a QC'd AnnData with HVGs and
store them in .obsm / .uns, driven by a YAML EmbeddingConfig.

This script:
  - loads a QC'd AnnData with HVGs annotated
  - reads embedding hyperparameters from configs/params.yml
  - computes PCA / Diffusion Map / UMAP / PHATE / etc. as configured
  - writes the updated AnnData (with .obsm embeddings) back to disk

Typical usage:

  python scripts/hvg_embed.py \
      --params configs/params.yml \
      --ad data/interim/k562_gwps_unperturbed_qc.h5ad \
      --out data/interim/k562_gwps_unperturbed_hvg_embeddings.h5ad

QC:
    python - << 'PY'
    import scanpy as sc

    ad = sc.read_h5ad("data/interim/k562_gwps_unperturbed_hvg_embeddings.h5ad")
    print("Embeddings X shape:", ad.X.shape)
    print("obsm keys:", list(ad.obsm.keys()))
    if "X_pca" in ad.obsm:
        print("X_pca shape:", ad.obsm["X_pca"].shape)
    PY


"""

from __future__ import annotations

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

import scanpy as sc
import yaml

from mantra.embeddings import EmbeddingConfig, compute_embeddings



def build_argparser() -> argparse.ArgumentParser:
    p = argparse.ArgumentParser(
        description=(
            "Compute manifold embeddings on a QCâ\200\231d AnnData with HVGs "
            "and store them in .obsm/.uns. Driven by YAML EmbeddingConfig."
        )
    )
    p.add_argument(
        "--params",
        type=str,
        required=True,
        help="YAML params file (must contain an 'embeddings' section)",
    )
    p.add_argument(
        "--ad",
        type=str,
        required=True,
        help="Input AnnData file (QCâ\200\231d, HVGs annotated).",
    )
    p.add_argument(
        "--out",
        type=str,
        default=None,
        help="Output AnnData file (default: overwrite --ad).",
    )
    return p


def main() -> None:
    args = build_argparser().parse_args()

    params: Dict[str, Any] = yaml.safe_load(Path(args.params).read_text())
    emb_cfg = EmbeddingConfig(**params.get("embeddings", {}))

    in_path = Path(args.ad)
    out_path = Path(args.out) if args.out is not None else in_path

    print(f"[load] Reading AnnData from {in_path} ...", flush=True)
    ad = sc.read_h5ad(in_path)

    # compute embeddings in-place
    ad = compute_embeddings(ad, emb_cfg)

    print(f"[save] Writing updated AnnData with embeddings to {out_path} ...", flush=True)
    ad.write(out_path)
    print("[done]", flush=True)


if __name__ == "__main__":
    main()
```

```
#!/usr/bin/env python
# src/mantra/scripts/inspect_h5ad.py
"""
Inspect an .h5ad file to understand its structure for QC ingestion
and downstream modeling.

This script:
  - prints AnnData shape and X type
  - summarizes obs (cell metadata) columns, dtypes, and value distributions
  - highlights likely-important obs columns (e.g. batch, cell_type, state)
  - summarizes var (gene metadata) columns and example gene names

Usage:

  python scripts/inspect_h5ad.py \
      --h5ad data/interim/k562_gwps_unperturbed_qc.h5ad
"""

import argparse

import scanpy as sc
from pandas.api.types import is_categorical_dtype, is_numeric_dtype


def inspect_h5ad(path: str) -> None:
    print(f"Loading {path}")
    ad = sc.read_h5ad(path, backed="r")

    print("\n=== AnnData overview ===")
    print(ad)
    print("shape (n_cells, n_genes):", ad.shape)
    print("X class:", type(ad.X))

    # ---------- OBS (cell metadata) ----------
    print("\n=== OBS (cell metadata) ===")
    print("obs columns:", list(ad.obs.columns))
    print("\nobs.head():")
    print(ad.obs.head())

    print("\n[obs summary by column]")
    for col in ad.obs.columns:
        s = ad.obs[col]
        nunique = s.nunique()
        print(f"\n---- {col} ----")
        print("dtype:", s.dtype)
        print("n_unique:", nunique)

        if nunique <= 20:
            # categorical-ish: show value counts
            print("value_counts():")
            print(s.value_counts().head(20))
        else:
            # high-cardinality: branch by dtype
            if is_numeric_dtype(s):
                print(
                    "min/mean/max:",
                    float(s.min()),
                    float(s.mean()),
                    float(s.max()),
                )
            elif is_categorical_dtype(s):
                cats = s.cat.categories
                print(f"categorical with {len(cats)} categories")
                print("categories (first 20):", list(cats[:20]))
            else:
                print("example values:", s.iloc[:10].tolist())

    # Highlight likely-important columns for QC / modeling if present
```

```python
    interesting_obs = [
        "timepoint",
        "day",
        "treatment",
        "condition",
        "sample",
        "batch",
        "clone",
        "clone_id",
        "lineage",
        "cell_type",
        "state",
    ]
    print("\n=== Selected interesting obs columns (if present) ===")
    for col in interesting_obs:
        if col in ad.obs:
            print(f"\n---- {col} ----")
            s = ad.obs[col]
            print("dtype:", s.dtype)
            print("n_unique:", s.nunique())
            print(s.value_counts().head(20))

    # ---------- VAR (gene metadata) ----------
    print("\n=== VAR (gene metadata) ===")
    print("var columns:", list(ad.var.columns))
    print("\nvar.head():")
    print(ad.var.head())
    print("\nvar_names (first 10):")
    print(ad.var_names[:10].tolist())


def main() -> None:
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "--h5ad",
        type=str,
        default="data/raw/stateFate_inVitro_normed_counts.h5ad",
        help="Path to .h5ad file to inspect",
    )
    args = parser.parse_args()
    inspect_h5ad(args.h5ad)


if __name__ == "__main__":
    main()
```

```
#!/usr/bin/env python3
# src/mantra/scripts/make_grn_adj.py
"""
Build a correlation-based geneâ\200\223gene adjacency matrix A [G, G]
in the EGGFM HVG space, aligned to the energy checkpoint var_names.

This script:
  - loads a QC'd unperturbed K562 AnnData (same used for EGGFM)
  - loads an EGGFM checkpoint to get HVG gene ordering
  - subsets AnnData to that gene set and ordering
  - computes geneâ\200\223gene Pearson correlations across cells
  - thresholds by |corr| >= corr_thresh
  - optionally keeps top-k neighbors per gene
  - row-normalizes to obtain a stochastic adjacency for message passing
  - saves A as a .npy file for use with GRNGNN (--adj flag)

Typical usage:

  python src/mantra/scripts/make_grn_adj.py \\
      --ad data/interim/k562_gwps_unperturbed_qc.h5ad \\
      --energy-ckpt out/models/eggfm/eggfm_energy_k562_hvg_hvg100.pt \\
      --out data/interim/A_k562_hvg100.npy \\
      --corr-thresh 0.2 \\
      --topk 10

"""

from __future__ import annotations

import argparse
from pathlib import Path

from mantra.grn.make_adj import make_grn_adj


def build_argparser() -> argparse.ArgumentParser:
    p = argparse.ArgumentParser(
        description=(
            "Build a correlation-based geneâ\200\223gene adjacency [G,G] "
            "aligned to an EGGFM HVG checkpoint."
        )
    )

    p.add_argument(
        "--ad",
        type=str,
        required=True,
        help="QC'd unperturbed AnnData (e.g. data/interim/k562_gwps_unperturbed_qc.h5ad)",
    )
    p.add_argument(
        "--energy-ckpt",
        type=str,
        required=True,
        help="EGGFM .pt checkpoint with 'var_names' or 'feature_names' for HVGs",
    )
    p.add_argument(
        "--out",
        type=str,
        required=True,
        help="Output .npy path for adjacency [G,G]",
    )
    p.add_argument(
        "--corr-thresh",
        type=float,
        default=0.2,
        help="Min absolute Pearson correlation to keep an edge (default: 0.2)",
    )
    p.add_argument(
        "--topk",
        type=int,
        default=10,
        help="Optional top-k neighbors per gene (per row); set <=0 to disable.",
    )

    return p


def main() -> None:
    ap = build_argparser()
    args = ap.parse_args()

    stats = make_grn_adj(
        ad_path=Path(args.ad),
        energy_ckpt_path=Path(args.energy_ckpt),
        out_path=Path(args.out),
        corr_thresh=args.corr_thresh,
        topk=args.topk if args.topk > 0 else None,
    )

    print(
        f"[adj][summary] G_eff={stats['G_eff']}, "
        f"corr_thresh={stats['corr_thresh']}, "
        f"topk={stats['topk']}, "
        f"n_edges={stats['n_edges']}, "
        f"out={stats['out_path']}",
        flush=True,
    )


if __name__ == "__main__":
    main()
```

```
#!/usr/bin/env python3
# src/mantra/scripts/make_grn_npz.py
"""
Streamingly aggregate K562 GWPS / Perturb-seq into train/val NPZs
in the EGGFM HVG space, using the energy checkpoint var_names.

This script:
  - opens the large raw K562 GWPS AnnData in backed mode
  - applies basic QC on cells (mito percent, UMI count)
  - identifies control vs. perturbed cells using an obs "reg" column
  - computes regulator-level Î\224E (and optionally Î\224P_obs via cNMF W)
  - performs a train/val split
  - saves compressed NPZ files with:
      reg_idx, deltaE, deltaP_obs, deltaY_obs, dose

Typical usage:

  python scripts/make_grn_npz.py \
      --ad-raw data/raw/k562_gwps.h5ad \
      --energy-ckpt out/models/eggfm/eggfm_energy_k562_hvg_hvg75.pt \
      --out-dir data/interim/grn_k562_gwps_hvg75_npz \
      --reg-col gene \
      --control-value non-targeting \
      --max-pct-mt 0.2 \
      --min-umi 2000 \
      --min-cells-per-group 10 \
      --val-frac 0.2 \
      --seed 7
"""

from __future__ import annotations

import argparse
from pathlib import Path

from mantra.grn.make_npz import make_grn_npz


def build_argparser() -> argparse.ArgumentParser:
    p = argparse.ArgumentParser(
        description=(
            "Streaming aggregation of K562 GWPS into train/val NPZs "
            "in the EGGFM HVG space, using the energy checkpoint var_names."
        )
    )

    p.add_argument(
        "--ad-raw",
        type=str,
        required=True,
        help="Big K562 GWPS AnnData (e.g. data/raw/k562_gwps.h5ad)",
    )
    p.add_argument(
        "--energy-ckpt",
        type=str,
        required=True,
        help="EGGFM .pt checkpoint with 'var_names' for HVGs",
    )
    p.add_argument(
        "--out-dir",
        type=str,
        required=True,
        help="Output dir for train.npz / val.npz",
    )

    # obs columns
    p.add_argument(
        "--reg-col",
        type=str,
        default="gene",
        help="obs column with perturbed target gene / regulator",
    )
    p.add_argument(
        "--dose-col",
        type=str,
        default="gem_group",
        help="obs column to treat as 'dose' (kept for API; ignored in K562)",
    )
    p.add_argument(
        "--control-value",
        type=str,
        default="non-targeting",
        help="Value in reg-col that denotes control / non-targeting cells",
    )

    # QC thresholds
    p.add_argument(
        "--max-pct-mt",
        type=float,
        default=0.2,
        help="Max allowed mitopercent (fraction, e.g. 0.2 for 20%)",
    )
    p.add_argument(
        "--min-umi",
        type=float,
        default=2000.0,
        help="Min UMI_count per cell",
    )

    p.add_argument(
        "--min-cells-per-group",
        type=int,
        default=10,
        help="Min #cells per regulator to keep a sample",
    )
    p.add_argument(
        "--val-frac",
        type=float,
        default=0.2,
        help="Fraction of regulator-level samples to use for validation",
    )
    p.add_argument(
        "--seed",
        type=int,
        default=7,
        help="Random seed for train/val split",
    )

    p.add_argument(
        "--cnmf-W",
        type=str,
        default=None,
        help="Optional W.npy [G,K] for Î\224P_obs; if missing, Î\224P_obs = Î\224E",
    )
    p.add_argument(
        "--traits-dim",
        type=int,
        default=3,
        help="Dimensionality of Î\224Y_obs stub (e.g. 3 for MCH, RDW, IRF)",
    )

    return p


def main() -> None:
    ap = build_argparser()
    args = ap.parse_args()

    stats = make_grn_npz(
        ad_raw_path=Path(args.ad_raw),
        energy_ckpt_path=Path(args.energy_ckpt),
        out_dir=Path(args.out_dir),
        reg_col=args.reg_col,
        dose_col=args.dose_col,
        control_value=args.control_value,
        max_pct_mt=args.max_pct_mt,
        min_umi=args.min_umi,
        min_cells_per_group=args.min_cells_per_group,
        val_frac=args.val_frac,
        seed=args.seed,
        cnmf_W_path=Path(args.cnmf_W) if args.cnmf_W is not None else None,
        traits_dim=args.traits_dim,
    )

    print(
        f"[summary] N={stats['N']} "
        f"(train={stats['N_train']}, val={stats['N_val']}), "
        f"G_eff={stats['G_eff']}, "
        f"train={stats['train_path']}, val={stats['val_path']}",
        flush=True,
    )


if __name__ == "__main__":
    main()
```

```
#!/usr/bin/env python3
# src/mantra/scripts/qc.py
"""
Run QC + EDA for single-cell RNA-seq AnnData objects.

This script is a thin wrapper around `mantra.qc.run_qc` and:
  - loads a raw / full AnnData (optionally containing both perturbed + control cells)
  - optionally restricts to non-targeting **control** cells only
  - filters genes and cells based on basic thresholds
  - computes HVGs and log-normalized expression
  - writes a QC'd AnnData to disk

Usage:

  # QC on all cells (perturbed + controls)
  python scripts/qc.py \
      --params configs/params.yml \
      --ad data/raw/k562_gwps.h5ad \
      --out data/interim/k562_gwps_qc.h5ad

  # controls-only QC (non-targeting / unperturbed cells)
  python scripts/qc.py \
      --params configs/params.yml \
      --ad data/raw/k562_gwps.h5ad \
      --out data/interim/k562_gwps_unperturbed_qc.h5ad \
      --controls-only
"""

from __future__ import annotations

import argparse
from pathlib import Path

from mantra.qc import run_qc


def build_argparser() -> argparse.ArgumentParser:
    ap = argparse.ArgumentParser(description="QC + EDA for cells.")
    ap.add_argument("--params", required=True, help="Path to configs/params.yml")
    ap.add_argument(
        "--out",
        required=True,
        help="Output QC AnnData .h5ad file (e.g. data/interim/k562_qc.h5ad)",
    )
    ap.add_argument("--ad", required=True, help="Path to input .h5ad")
    ap.add_argument(
        "--controls-only",
        action="store_true",
        dest="controls_only",
        help="If set, restrict to non-targeting control cells (gene == 'non-targeting').",
    )
    return ap


def main() -> None:
    args = build_argparser().parse_args()
    run_qc(
        params_path=Path(args.params),
        ad_path=Path(args.ad),
        out_path=Path(args.out),
        # `pet` in the underlying function really means "controls-only"
        pet=args.controls_only,
    )


if __name__ == "__main__":
    main()
```

```
#!/usr/bin/env python3
# src/mantra/scripts/strip_uns_log1p.py
"""
Remove the /uns/log1p group from an .h5ad file to fix
AnnData IORegistryError issues caused by incompatible log1p metadata.

This script:
  - optionally copies the input .h5ad before editing
  - opens the target file with h5py
  - deletes /uns/log1p if present
  - leaves other contents untouched

Usage:

  python scripts/strip_uns_log1p.py \
      --h5ad-in data/raw/some_dataset.h5ad \
      --h5ad-out data/raw/some_dataset_nolog1p.h5ad
"""

from __future__ import annotations

import argparse
import shutil
from pathlib import Path

import h5py  # type: ignore


def build_argparser() -> argparse.ArgumentParser:
    p = argparse.ArgumentParser(
        description="Remove /uns/log1p from an .h5ad file to fix AnnData IORegistryError."
    )
    p.add_argument(
        "--h5ad-in",
        required=True,
        help="Input .h5ad file (will be copied if --h5ad-out is set).",
    )
    p.add_argument(
        "--h5ad-out",
        default=None,
        help="Optional output .h5ad file. "
             "If omitted, edits are done in-place on --h5ad-in.",
    )
    return p


def main() -> None:
    args = build_argparser().parse_args()
    src = Path(args.h5ad_in)
    if not src.exists():
        raise SystemExit(f"Input file not found: {src}")

    # Decide where to write
    if args.h5ad_out is None:
        dst = src
        print(f"[info] editing in-place: {dst}")
    else:
        dst = Path(args.h5ad_out)
        dst.parent.mkdir(parents=True, exist_ok=True)
        print(f"[info] copying {src} -> {dst}")
        shutil.copy2(src, dst)

    # Open with h5py and delete /uns/log1p if present
    with h5py.File(dst, "r+") as f:
        if "uns" not in f:
            print("[info] no /uns group found; nothing to remove.")
            return
        uns_grp = f["uns"]
        if "log1p" in uns_grp:
            print("[info] found /uns/log1p; deleting...")
            del uns_grp["log1p"]
            print("[done] /uns/log1p removed.")
        else:
            print("[info] /uns/log1p not present; nothing to remove.")


if __name__ == "__main__":
    main()
```

```
#!/usr/bin/env python3
# src/mantra/eggfm/train_energy.py
"""
Train an EGGFM energy model on a QC'd K562 AnnData, using either:

  - HVG expression (ad.X in the HVG-restricted gene space), or
  - a precomputed embedding stored in ad.obsm (e.g. "X_pca", "X_diffmap", "X_umap")

as the feature space for the EnergyMLP.

This module:
  - loads QC'd AnnData
  - optionally subsamples cells
  - restricts to HVGs (and top max_hvg by dispersion if configured)
  - selects the feature representation based on `latent_space`:
        * latent_space == "hvg" â\206\222 use ad_prep.X
        * latent_space == "<key>" â\206\222 use ad_prep.obsm["<key>"]
  - runs denoising score-matching (DSM) training for EnergyMLP
  - saves an energy checkpoint containing:
      * model state_dict
      * feature / gene names
      * mean / std normalizers in the chosen feature space
      * feature-space tag ("hvg", "X_pca", etc.)

Typical CLI wrapper usage (via scripts/train_energy.py):

  python scripts/train_energy.py \
      --params configs/params.yml \
      --ad data/interim/k562_gwps_unperturbed_qc.h5ad \
      --out out/models/eggfm \
      --space hvg

or, to train directly on an embedding:

  python scripts/train_energy.py \
      --params configs/params.yml \
      --ad data/interim/k562_gwps_unperturbed_hvg_embeddings.h5ad \
      --out out/models/eggfm \
      --space X_pca
"""

from __future__ import annotations

import argparse
from pathlib import Path

from mantra.eggfm.run_energy import run_energy_training


def build_argparser() -> argparse.ArgumentParser:
    p = argparse.ArgumentParser(description="Train EGGFM energy model on K562")
    p.add_argument(
        "--params",
        type=str,
        required=True,
        help="YAML params file (must contain eggfm_model and eggfm_train)",
    )
    p.add_argument(
        "--ad",
        type=str,
        required=True,
        help="Preprocessed K562 AnnData (e.g. data/interim/k562_replogle_prep.h5ad)",
    )
    p.add_argument(
        "--out",
        type=str,
        required=True,
        help="Output directory for checkpoints (e.g. out/models/eggfm)",
    )
    p.add_argument(
        "--space",
        type=str,
        default="hvg",
        help="Representation to train on: 'hvg' or an .obsm key like 'X_pca'",
    )
    return p


def main() -> None:
    args = build_argparser().parse_args()
    run_energy_training(
        params_path=Path(args.params),
        ad_path=Path(args.ad),
        out_dir=Path(args.out),
        space=args.space,
    )


if __name__ == "__main__":
    main()
```

```python
#!/usr/bin/env python3
# src/mantra/eggfm/train_grn.py
"""
Train the GRN GNN block on K562 with a pre-trained EGGFM energy prior.

This script:
  - parses GRN model / train / loss configs from YAML
  - loads a QC'd K562 AnnData (for x_ref computation and gene alignment)
  - loads train/val NPZs with regulator-level Î\224E / Î\224P / Î\224Y
  - builds adjacency A and program loadings W (if provided)
  - constructs an energy prior from an EGGFM checkpoint
  - trains the GRNGNN (+ optional TraitHead)
  - saves a GRN checkpoint with model weights and prior metadata

Usage:

  python scripts/train_grn.py \
      --params configs/params.yml \
      --out out/models/grn_hvg75 \
      --ad data/interim/k562_gwps_unperturbed_qc.h5ad \
      --train-npz data/interim/grn_k562_gwps_hvg75_npz/train.npz \
      --val-npz data/interim/grn_k562_gwps_hvg75_npz/val.npz \
      --energy-ckpt out/models/eggfm/eggfm_energy_k562_hvg_hvg75.pt \
      --cnmf-W out/programs/k562_cnmf_hvg75/k562_cnmf_hvg75_W_consensus.npy
"""

from __future__ import annotations

import argparse
from pathlib import Path

from mantra.grn.run_grn import run_grn_training


def build_argparser() -> argparse.ArgumentParser:
    p = argparse.ArgumentParser(
        description="Train GRN GNN on K562 with pre-trained EGGFM energy prior"
    )

    p.add_argument(
        "--params",
        type=str,
        required=True,
        help="YAML params file (must contain grn_model, grn_train, grn_loss)",
    )
    p.add_argument(
        "--out",
        type=str,
        required=True,
        help="Output directory for GRN checkpoints / logs",
    )
    p.add_argument(
        "--ad",
        type=str,
        required=True,
        help="Preprocessed K562 AnnData (same gene order as Î\224E)",
    )
    p.add_argument(
        "--train-npz",
        type=str,
        required=True,
        help="NPZ with aggregated (reg_idx, deltaE, deltaP_obs, deltaY_obs, dose)",
    )
    p.add_argument(
        "--val-npz",
        type=str,
        default=None,
        help="Optional NPZ for validation set",
    )
    p.add_argument(
        "--adj",
        type=str,
        default=None,
        help="Optional .npy adjacency [G,G]. If not provided, uses identity.",
    )
    p.add_argument(
        "--cnmf-W",
        type=str,
        default=None,
        help="Optional .npy cNMF loadings W [G,K]. If missing, uses identity [G,G].",
    )
    p.add_argument(
        "--energy-ckpt",
        type=str,
        required=True,
        help="Path to pre-trained EGGFM energy checkpoint (.pt)",
    )
    return p


def main() -> None:
    args = build_argparser().parse_args()

    run_grn_training(
        params_path=Path(args.params),
        out_dir=Path(args.out),
        ad_path=Path(args.ad),
        train_npz_path=Path(args.train_npz),
        val_npz_path=Path(args.val_npz) if args.val_npz is not None else None,
        energy_ckpt_path=Path(args.energy_ckpt),
        adj_path=Path(args.adj) if args.adj is not None else None,
        cnmf_W_path=Path(args.cnmf_W) if args.cnmf_W is not None else None,
    )


if __name__ == "__main__":
    main()
```