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

from mantra.config import GRNModelConfig, GRNTrainConfig, GRNLossConfig

__all__ = [
    "GRNModelConfig",
    "GRNTrainConfig",
    "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_fn (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_fn: 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_fn = energy_fn.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_fn.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_fn(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
# 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.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.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]

        out = self.act(gamma * h_lin + beta)
        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
```

```python
# ----------------------------------------------------------------------
# 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 = energy_prior(x_hat)                  # [B]
    L_geo = loss_cfg.lambda_geo * energy.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/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.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=train_cfg.lr,
            weight_decay=train_cfg.weight_decay,
        )

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

```
    def fit(
        self,
        train_loader: DataLoader,
        val_loader: Optional[DataLoader] = None,
    ) -> None:
        """
        Simple training loop with optional early stopping on val loss.
        """
        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)
            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}"
            )

            if val_loader is not None:
                val_stats = self.eval_epoch(val_loader)
                val_loss = val_stats["loss"]
                msg += f" | val_loss={val_loss:.4f}"
                improved = val_loss + cfg.early_stop_min_delta < best_val_loss
                if improved:
                    best_val_loss = val_loss
                    best_state = self._snapshot_state()
                    epochs_without_improve = 0
                else:
                    epochs_without_improve += 1
            else:
                # no val set â\206\222 just track best train loss
                improved = train_stats["loss"] + cfg.early_stop_min_delta < best_val_loss
                if improved:
                    best_val_loss = train_stats["loss"]
                    best_state = self._snapshot_state()
                epochs_without_improve = 0  # no early stopping

            print(msg, flush=True)

            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",
]
```