

```
# src/mantra/eggfm/config.py
```

```
from __future__ import annotations
```

```
from mantra.config import EnergyModelConfig, EnergyTrainConfig, EnergyModelBundle
```

```
__all__ = [  
    "EnergyModelConfig",  
    "EnergyTrainConfig",  
    "EnergyModelBundle",  
]
```

```
# AnnDataPyTorch.py
```

```
import numpy as np
import torch
from torch.utils.data import Dataset
from scipy import sparse
import scanpy as sc # only for type hints; not strictly necessary

class AnnDataExpressionDataset(Dataset):
    """
    Wraps an AnnData object's X matrix (after prep()) as a PyTorch dataset.
    Uses HVG, log-normalized expression directly.
    """

    def __init__(self, X, float_dtype=np.float32):
        if sparse.issparse(X):
            X = X.toarray()
        X = np.asarray(X, dtype=float_dtype)

        mean = X.mean(axis=0, keepdims=True)
        std = X.std(axis=0, keepdims=True)

        # prevent divide-by-zero or tiny variance explosions
        std = np.clip(std, 1e-2, None)

        # store for later (without the extra batch dim)
        self.mean = mean.astype(float_dtype).squeeze(0) # shape [D]
        self.std = std.astype(float_dtype).squeeze(0) # shape [D]

        self.X = (X - mean) / std

    def __len__(self) -> int:
        return self.X.shape[0]

    def __getitem__(self, idx: int) -> torch.Tensor:
        return torch.from_numpy(self.X[idx])
```

```
# src/mantra/eggfm/inference.py

from __future__ import annotations

from pathlib import Path
from typing import Any, Dict, List, Optional, Sequence, Union

import numpy as np
import torch
from torch import nn, Tensor

from mantra.eggfm.models import EnergyMLP

class EnergyScorer:
    """
    Wraps a trained EnergyMLP + normalization (+ optional projection)
    so we can compute energies in a consistent way.

    Supports:
    - HVG / gene space:  $x \rightarrow \text{normalize} \rightarrow E(x)$ 
    - Embedding space:  $x \rightarrow \text{project (PCA)} \rightarrow \text{normalize} \rightarrow E_z(z)$ 
    """

    def __init__(
        self,
        energy_model: nn.Module,
        mean: Optional[Tensor],
        std: Optional[Tensor],
        var_names: Optional[Sequence[str]] = None,
        proj_matrix: Optional[Tensor] = None,  # [G, d] for embedding case
        space: str = "hvg",
        device: Optional[torch.device] = None,
    ) -> None:
        self.device = device or torch.device(
            "cuda" if torch.cuda.is_available() else "cpu"
        )

        self.energy_model = energy_model.to(self.device)
        self.energy_model.eval()
        for p in self.energy_model.parameters():
            p.requires_grad_(False)

        self.space = space

        # mean/std are always in the *model feature space*: [D_model]
        self.mean = None if mean is None else mean.to(self.device).view(1, -1)
        self.std = None if std is None else std.to(self.device).view(1, -1)

        # gene feature metadata (for HVG space alignment; optional)
        self.var_names: Optional[List[str]] = None
        self._name_to_idx: Optional[Dict[str, int]] = None
        if var_names is not None:
            self.var_names = [str(v) for v in var_names]
            self._name_to_idx = {name: i for i, name in enumerate(self.var_names)}

        # optional projection (for embedding case): [G_raw, D_model]
        self.proj_matrix: Optional[Tensor] = None
        if proj_matrix is not None:
            proj_matrix = proj_matrix.to(self.device)
            self.proj_matrix = proj_matrix

    # -----
    # Construction helper
    # -----

    @classmethod
    def from_checkpoint(
```

```

cls,
ckpt_path: Union[str, Path],
device: Optional[torch.device] = None,
) -> "EnergyScorer":
    """
    Load an EnergyScorer from a .pt checkpoint.

    Expects ckpt to contain something like:

        {
            "state_dict": ...,
            "model_cfg": {"hidden_dims": [...]},
            "n_genes": int,          # D_model
            "space": "hvg" or "embedding",
            "var_names": [...],     # optional, for HVG space alignment
            "mean": ...,
            "std": ...,
            # optional for embedding:
            "proj_matrix": np.ndarray [G_raw, D_model],
        }
    """
    ckpt_path = Path(ckpt_path)
    ckpt = torch.load(ckpt_path, map_location=device or "cpu")

    n_genes = ckpt.get("n_genes")
    model_cfg = ckpt.get("model_cfg", {})
    space = ckpt.get("space", "hvg")

    # reconstruct EnergyMLP in model feature space
    energy_model = EnergyMLP(
        n_genes=n_genes,
        **model_cfg,
    )
    energy_model.load_state_dict(ckpt["state_dict"])

    def _to_tensor_or_none(key: str) -> Optional[Tensor]:
        if key not in ckpt or ckpt[key] is None:
            return None
        arr = ckpt[key]
        if isinstance(arr, Tensor):
            return arr
        return torch.as_tensor(arr, dtype=torch.float32)

    mean = _to_tensor_or_none("mean")
    std = _to_tensor_or_none("std")
    var_names = ckpt.get("var_names", None)

    proj_matrix = _to_tensor_or_none("proj_matrix") # for embedding space

    return cls(
        energy_model=energy_model,
        mean=mean,
        std=std,
        var_names=var_names,
        proj_matrix=proj_matrix,
        space=space,
        device=device,
    )

# -----
# Internal helpers
# -----

def _ensure_tensor(self, x: Union[Tensor, np.ndarray]) -> Tensor:
    if isinstance(x, Tensor):
        return x.to(self.device, dtype=torch.float32)
    else:
        return torch.as_tensor(x, dtype=torch.float32, device=self.device)

```

```

def _reorder_by_genes(
    self,
    x: Tensor,
    gene_names: Optional[Sequence[str]],
    # [B, G_in]
) -> Tensor:
    """
    If var_names and gene_names are provided, reorder x to match training order.
    Otherwise, assume x is already aligned.
    """
    if self.var_names is None or gene_names is None:
        return x

    if len(self.var_names) != x.shape[1]:
        raise ValueError(
            f"EnergyScorer: mismatch between model gene dim ({len(self.var_names)}) "
            f"and input x.shape[1] ({x.shape[1]})."
        )

    input_name_to_idx = {str(name): i for i, name in enumerate(gene_names)}

    try:
        indices = [input_name_to_idx[name] for name in self.var_names]
    except KeyError as e:
        missing = str(e.args[0])
        raise KeyError(
            f"EnergyScorer: gene {missing!r} from training var_names "
            f"not found in provided gene_names."
        )

    idx = torch.as_tensor(indices, dtype=torch.long, device=x.device)
    return x[:, idx]

def _apply_normalization(self, z: Tensor) -> Tensor:
    if self.mean is None or self.std is None:
        return z
    return (z - self.mean) / self.std

# -----
# Public API
# -----

@torch.no_grad()
def score(
    self,
    x_raw: Union[Tensor, np.ndarray],
    gene_names: Optional[Sequence[str]] = None,
) -> Tensor:
    """
    Compute energy for a batch of states x_raw in *gene space*:

    - If space == "hvg": x_raw is already in model feature space (after alignment).
    - If space == "embedding": x_raw is in gene space; we project with proj_matrix.

    Returns energies: [B].
    """
    x = self._ensure_tensor(x_raw) # [B, G_in]
    x = self._reorder_by_genes(x, gene_names) # optionally align to var_names

    # If we have a projection matrix, go to embedding space
    if self.proj_matrix is not None:
        z = x @ self.proj_matrix # [B, D_model]
    else:
        z = x # [B, D_model]

    z_norm = self._apply_normalization(z) # [B, D_model]
    energy = self.energy_model(z_norm) # [B] or [B,1]
    if energy.ndim == 2:

```

```
        energy = energy.squeeze(-1)
    return energy

@torch.no_grad()
def score_delta(
    self,
    x_ref: Union[Tensor, np.ndarray],
    deltaE_pred: Union[Tensor, np.ndarray],
    gene_names: Optional[Sequence[str]] = None,
) -> Tensor:
    """
    Convenience: score energy of  $\hat{x} = x_{ref} + \hat{I} \backslash \Delta E_{pred}$ .

    x_ref: [G] or [1,G]
    deltaE_pred: [B,G]
    """
    x_ref_t = self._ensure_tensor(x_ref)
    if x_ref_t.ndim == 1:
        x_ref_t = x_ref_t.unsqueeze(0) # [1,G]
    delta_t = self._ensure_tensor(deltaE_pred) # [B,G]

    if x_ref_t.shape[1] != delta_t.shape[1]:
        raise ValueError(
            f"x_ref dim {x_ref_t.shape[1]} != deltaE_pred dim {delta_t.shape[1]}"
        )

    x_hat = x_ref_t + delta_t
    return self.score(x_hat, gene_names=gene_names)
```

```
# EnergyMLP.py
```

```
from typing import Sequence, Optional
import torch
from torch import nn
```

```
class EnergyMLP(nn.Module):
```

```
    """
```

```
     $E(x) = \langle E_{\theta}(x), x \rangle$  where  $E_{\theta}$  is an MLP with nonlinearities.
```

```
     $x$  is HVG, log-normalized expression (optionally mean-centered).
```

```
We also expose a latent representation  $z(x)$  from the last hidden layer,
which can be used as a geometry for manifold learning.
```

```
    """
```

```
def __init__(
    self,
    n_genes: int,
    hidden_dims: Sequence[int] = (512, 512, 512, 512),
    activation: Optional[nn.Module] = None,
):
    super().__init__()
    if activation is None:
        activation = nn.Softplus()

    layers = []
    in_dim = n_genes
    for h in hidden_dims:
        layers.append(nn.Linear(in_dim, h))
        layers.append(activation)
        in_dim = h

    # encoder: maps  $x \in \mathbb{R}^{hidden\_dims[-1]}$ 
    self.hidden = nn.Sequential(*layers)

    # head: maps  $z \in \mathbb{R}^{D}$  ( $D = n\_genes$ )
    self.vector_head = nn.Linear(in_dim, n_genes)

    # store for convenience
    self.n_genes = n_genes
    self.latent_dim = in_dim
```

```
def forward(self, x: torch.Tensor) -> torch.Tensor:
```

```
    """
```

```
    Standard forward used in training:
```

```
     $x$ : (B, D)
```

```
    returns: energy (B,)
```

```
    """
```

```
    if x.dim() == 1:
```

```
        x = x.unsqueeze(0)
```

```
    z = self.hidden(x) # (B, latent_dim)
```

```
    v = self.vector_head(z) # (B, D)
```

```
    energy = (v * x).sum(dim=-1) #  $\langle v(x), x \rangle$ 
```

```
    return energy
```

```
@torch.no_grad()
```

```
def encode(self, x: torch.Tensor) -> torch.Tensor:
```

```
    """
```

```
    Return latent representation  $z(x)$  from the last hidden layer.
```

```
     $x$ : (B, D)
```

```
    returns:  $z$  (B, latent_dim)
```

```
    """
```

```
    if x.dim() == 1:
```

```
        x = x.unsqueeze(0)
```

```
    z = self.hidden(x)
```

```
    return z
```

```
def score(self, x: torch.Tensor) -> torch.Tensor:
```

```
    """
```

```
    score(x)  $\hat{=}$   $\frac{1}{N} \sum_x \log p(x) = -\frac{1}{N} \sum_x E(x)$ 
```

```
    """
```

```
    x = x.clone().detach().requires_grad_(True)
```

```
    energy = self.forward(x) # (B,)
```

```
    energy_sum = energy.sum()
```

```
    (grad,) = torch.autograd.grad(
```

```
        energy_sum,
```

```
        x,
```

```
        create_graph=False,
```

```
        retain_graph=False,
```

```
        only_inputs=True,
```

```
    )
```

```
    score = -grad
```

```
    return score
```



```

# src/mantra/eggfm/run_energy.py
from __future__ import annotations

from pathlib import Path
from typing import Dict, Any

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

from mantra.config import EnergyModelConfig, EnergyTrainConfig
from mantra.eggfm.trainer import train_energy_model
from utils import subset_anndata # or from mantra.utils import subset_anndata

def run_energy_training(
    params_path: Path,
    ad_path: Path,
    out_dir: Path,
    space: str = "hvg",
) -> Path:
    """
    High-level entrypoint: load QCâ\200\231d AnnData, subset HVGs, train EGGFM, save checkpoint.

    Returns the checkpoint Path.
    """
    out_dir.mkdir(parents=True, exist_ok=True)

    params: Dict[str, Any] = yaml.safe_load(params_path.read_text())
    model_cfg = EnergyModelConfig(**params.get("eggfm_model", {}))
    train_cfg = EnergyTrainConfig(**params.get("eggfm_train", {}))

    # 1) Load prepped K562 AnnData
    ad = sc.read_h5ad(str(ad_path))

    # 2) optional subsample for this experiment
    train_n_cells = params["eggfm_train"].get("n_cells_sample", None)
    if train_n_cells is not None:
        ad_prep = subset_anndata(ad, train_n_cells, random_state=params.get("seed", 0))
    else:
        ad_prep = ad

    # 3) restrict to HVGs if present
    if "highly_variable" in ad_prep.var:
        ad_prep = ad_prep[:, ad_prep.var["highly_variable"]].copy()
        print(f"Using HVGs only: n_vars = {ad_prep.n_vars}")

        # further clamp HVGs to top N by dispersions_norm
        max_hvg = params["eggfm_train"].get("max_hvg", None)
        if max_hvg is not None and ad_prep.n_vars > max_hvg:
            if "dispersions_norm" in ad_prep.var:
                disp = ad_prep.var["dispersions_norm"].to_numpy()
                order = np.argsort(disp)[::-1] # descending
            else:
                # fallback: arbitrary but deterministic
                order = np.arange(ad_prep.n_vars)

            keep_idx = order[:max_hvg]
            ad_prep = ad_prep[:, keep_idx].copy()
            print(
                f"Subsetting HVGs from {len(order)} â\206\222 {ad_prep.n_vars} "
                f"(top {max_hvg} by dispersions_norm)"
            )
        else:
            print("No 'highly_variable' flag in ad.var; using all genes as-is.")

```

```
# 4) Train energy model
bundle = train_energy_model(
    ad_prep=ad_prep,
    model_cfg=model_cfg,
    train_cfg=train_cfg,
    latent_space=space,
)

energy_model = bundle.model
mean = bundle.mean
std = bundle.std
var_names = bundle.feature_names

# 5) Save checkpoint
ckpt = {
    "state_dict": energy_model.state_dict(),
    "model_cfg": {
        "hidden_dims": list(model_cfg.hidden_dims),
    },
    "n_genes": energy_model.n_genes,
    "var_names": var_names,
    "mean": mean,
    "std": std,
    "space": bundle.space,
}

space_tag = str(bundle.space).replace("X_", "") # e.g. "hvg", "pca"
n_genes = int(energy_model.n_genes)

ckpt_name = f"eggfm_energy_k562_{space_tag}_hvg{n_genes}.pt"
ckpt_path = out_dir / ckpt_name

torch.save(ckpt, ckpt_path)
print(f"Saved EGGFM energy checkpoint to {ckpt_path}", flush=True)

return ckpt_path
```

```
# src/mantra/eggfm/trainer.py
```

```
from __future__ import annotations
```

```
from typing import Optional, Dict
```

```
import numpy as np
```

```
import torch
```

```
import scanpy as sc
```

```
from torch import optim
```

```
from torch.utils.data import DataLoader
```

```
from mantra.eggfm.models import EnergyMLP
```

```
from mantra.eggfm.dataset import AnnDataExpressionDataset
```

```
from mantra.config import EnergyModelConfig, EnergyTrainConfig, EnergyModelBundle
```

```
class EnergyTrainer:
```

```
    """
```

```
    Denoising score-matching trainer for EnergyMLP.
```

```
    Given:
```

- model
- standardized dataset (AnnDataExpressionDataset)
- EnergyTrainConfig

```
    It runs the DSM loop and returns the best-trained model.
```

```
    """
```

```
def __init__(
```

```
    self,
```

```
    model: EnergyMLP,
```

```
    dataset: AnnDataExpressionDataset,
```

```
    train_cfg: EnergyTrainConfig,
```

```
) -> None:
```

```
    self.model = model
```

```
    self.dataset = dataset
```

```
    self.train_cfg = train_cfg
```

```
    device_str = train_cfg.device or ("cuda" if torch.cuda.is_available() else "cpu")
```

```
    self.device = torch.device(device_str)
```

```
    self.model.to(self.device)
```

```
    self.loader = DataLoader(
```

```
        dataset,
```

```
        batch_size=train_cfg.batch_size,
```

```
        shuffle=True,
```

```
        drop_last=True,
```

```
    )
```

```
    self.optimizer = optim.Adam(
```

```
        self.model.parameters(),
```

```
        lr=float(train_cfg.lr),
```

```
        weight_decay=float(train_cfg.weight_decay),
```

```
    )
```

```
    self.best_model_state: Optional[Dict[str, torch.Tensor]] = None
```

```
    self.best_trait_state: Optional[Dict[str, torch.Tensor]] = None
```

```
    self.best_loss: float = float("inf")
```

```
    self.best_state_dict: Optional[dict] = None
```

```
def train(self) -> EnergyMLP:
```

```
    sigma = float(self.train_cfg.sigma)
```

```
    grad_clip = float(self.train_cfg.grad_clip)
```

```
    early_stop_patience = int(self.train_cfg.early_stop_patience)
```

```
    early_stop_min_delta = float(self.train_cfg.early_stop_min_delta)
```

```
    num_epochs = int(self.train_cfg.num_epochs)
```

```

self.model.train()
epochs_without_improve = 0

n_total = len(self.dataset)

for epoch in range(num_epochs):
    running_loss = 0.0

    for xb in self.loader:
        xb = xb.to(self.device)  # (B, D), already standardized

        # Sample Gaussian noise
        eps = torch.randn_like(xb)
        y = xb + sigma * eps
        y.requires_grad_(True)

        # Energy and score
        energy = self.model(y)  # (B,)
        energy_sum = energy.sum()  # scalar

        (grad_y,) = torch.autograd.grad(
            energy_sum,
            y,
            create_graph=True,
            retain_graph=True,
            only_inputs=True,
        )
        s_theta = -grad_y

        # DSM target:  $-(y - x) / \sigma^2$ 
        target = -(y - xb) / (sigma**2)

        # MSE over batch and dimensions
        loss = ((s_theta - target) ** 2).sum(dim=1).mean()

        self.optimizer.zero_grad()
        loss.backward()
        if grad_clip > 0.0:
            torch.nn.utils.clip_grad_norm_(
                self.model.parameters(),
                grad_clip,
            )
        self.optimizer.step()

        running_loss += loss.item() * xb.size(0)
    epoch_loss = running_loss / n_total
    print(
        f"[Energy DSM] Epoch {epoch+1}/{num_epochs}  loss={epoch_loss:.6e}",
        flush=True,
    )

    improved = epoch_loss + early_stop_min_delta < self.best_loss
    if improved:
        self.best_loss = epoch_loss
        self.best_state_dict = self.model.state_dict()
        epochs_without_improve = 0
    else:
        epochs_without_improve += 1

    if early_stop_patience > 0 and epochs_without_improve >= early_stop_patience:
        print(
            f"[Energy DSM] Early stopping at epoch {epoch+1} "
            f"(best_loss={self.best_loss:.6e})",
            flush=True,
        )
        break

if self.best_state_dict is not None:

```

```

        self.model.load_state_dict(self.best_state_dict)

        return self.model

# -----
# High-level convenience wrapper: AnnData -> EnergyModelBundle
# -----

def train_energy_model(
    ad_prep: sc.AnnData,
    model_cfg: EnergyModelConfig,
    train_cfg: EnergyTrainConfig,
    latent_space: str = "hvg",
) -> EnergyModelBundle:
    """
    Convenience wrapper used by scripts:
    AnnData -> AnnDataExpressionDataset -> EnergyMLP -> EnergyTrainer

    `latent_space` controls which representation we train on:
    - "hvg": use ad_prep.X (HVG log-normalized expression)
    - any other string s: use ad_prep.obsm[s] (e.g. "X_pca", "X_phate")
    """
    # ----- dataset: HVG or embedding -----
    if latent_space == "hvg":
        X = ad_prep.X
    else:
        if latent_space not in ad_prep.obsm:
            raise KeyError(
                f"Requested latent_space={latent_space!r}, "
                f"but it is not in ad_prep.obsm. "
                f"Available keys: {list(ad_prep.obsm.keys())}"
            )
        X = ad_prep.obsm[latent_space]
    print(f"[EGGFM trainer] Latent_space: {latent_space}", latent_space)

    dataset = AnnDataExpressionDataset(X)
    n_genes = dataset.X.shape[1]

    # record normalization (always in the *model feature space*)
    mean = dataset.mean # [D]
    std = dataset.std # [D]

    # For HVG we keep gene names; for embeddings we can still stash HVG var_names.
    feature_names = np.array(ad_prep.var_names)

    # ----- model -----
    hidden_dims = tuple(model_cfg.hidden_dims)
    model = EnergyMLP(
        n_genes=n_genes,
        hidden_dims=hidden_dims,
    )

    # ----- trainer -----
    trainer = EnergyTrainer(
        model=model,
        dataset=dataset,
        train_cfg=train_cfg,
    )
    best_model = trainer.train()

    return EnergyModelBundle(
        model=best_model,
        mean=mean,
        std=std,
        feature_names=feature_names,
        space=latent_space,
    )

```

```
# src/mantra/eggfm/.__init__.py
```

```
from .models import EnergyMLP
from .dataset import AnnDataExpressionDataset
from .trainer import EnergyTrainer, train_energy_model
from .inference import EnergyScorer
```

```
__all__ = [
    "EnergyMLP",
    "AnnDataExpressionDataset",
    "EnergyTrainer",
    "train_energy_model",
    "EnergyScorer",
]
```