```python
import sys
import numpy.core.numeric as _numeric
sys.modules['numpy._core.numeric'] = _numeric  # map missing path to the real one

import pickle
with open("csi_data.pkl", "rb") as f:
    obj = pickle.load(f)
#with open("factorvae_csi300_alpha158.pkl", "rb") as f:
 #   obj = pickle.load(f)
```

```python
import pandas as pd
import numpy as np
import io

def summarize_df(df: pd.DataFrame, name: str = "df", sample_rows: int = 5):
    print(f"=== Summary for `{name}` ===")
    # Shape
    print(f"Rows: {len(df):,} | Columns: {df.shape[1]:,}\n")

    # Columns
    print("• Columns:")
    print(list(df.columns))
    print()

    # dtypes & non-null counts
    print("• dtypes & non-null counts:")
    buf = io.StringIO()
    df.info(buf=buf, memory_usage="deep")
    print(buf.getvalue())
    print()

    # Missing values
    print("• Missing values by column:")
    na = df.isna().sum()
    na_pct = (na / len(df) * 100).round(2)
    na_tbl = (
        pd.DataFrame({"n_missing": na, "pct_missing": na_pct})
        .sort_values(["pct_missing", "n_missing"], ascending=False)
    )
    display(na_tbl)

    # Cardinality
    print("\n• Cardinality (unique values) per column:")
    card = df.nunique(dropna=True).sort_values(ascending=False)
    display(card.to_frame("n_unique"))
```

```
    # Descriptive stats (numeric)
    print("\n• Descriptive stats (numeric columns):")
    if df.select_dtypes(include=np.number).shape[1] > 0:
        display(df.describe().T)
    else:
        print("(no numeric columns)")

    # Descriptive stats (non-numeric)
    print("\n• Descriptive stats (non-numeric columns):")
    obj_cols = df.select_dtypes(exclude=np.number).columns
    if len(obj_cols) > 0:
        display(df[obj_cols].describe(include="all").T)
    else:
        print("(no non-numeric columns)")

    # Top categories for object/categorical
    if len(obj_cols) > 0:
        print("\n• Top categories (up to 5) for object/categorical columns:")
        for c in obj_cols:
            print(f"  - {c}")
            display(df[c].value_counts(dropna=False).head(5).to_frame("count"))

    # Sample rows
    print(f"\n• Head ({sample_rows}) and Tail ({sample_rows})")
    display(df.head(sample_rows))
    display(df.tail(sample_rows))

# ---- Use it on your object ----
summarize_df(obj, name="obj")
```

```
=== Summary for `obj` ===
Rows: 870,621 | Columns: 159

• Columns:
['KMID', 'KLEN', 'KMID2', 'KUP', 'KUP2', 'KLOW', 'KLOW2', 'KSFT', 'KSFT2', 'OPEN0', 'H

• dtypes & non-null counts:
<class 'pandas.core.frame.DataFrame'>
MultiIndex: 870621 entries, (Timestamp('2008-01-02 00:00:00'), 'SH600000') to (Timesta
Columns: 159 entries, KMID to Ref($close, -2)/Ref($close, -1) - 1
dtypes: float32(158), float64(1)
memory usage: 534.8 MB


• Missing values by column:

• Cardinality (unique values) per column:

• Descriptive stats (numeric columns):
```

	n_missing	pct_missing
KMID	0	0.0
KLEN	0	0.0
KMID2	0	0.0
KUP	0	0.0
KUP2	0	0.0
...	...	...
VSUMD10	0	0.0
VSUMD20	0	0.0
VSUMD30	0	0.0
VSUMD60	0	0.0
Ref($close, -2)/Ref($close, -1) - 1	0	0.0

159 rows × 2 columns

	n_unique
CORD10	856472
CORD20	854065
CORR5	853774

CORR10	852549
CORD30	852339
...	...
IMXD5	9
CNTP5	9
IMIN5	5
IMAX5	5
VWAP0	1

159 rows × 1 columns

	count	mean	std	min	25%	50%	75%	max
KMID	870621.0	0.050305	1.236695	-3.000000	-0.631845	0.000000	0.703944	3.00
KLEN	870621.0	0.289531	1.143781	-1.731185	-0.573360	0.000000	0.878632	3.00
KMID2	870621.0	0.000716	0.784240	-2.751917	-0.673028	0.000000	0.664717	3.00
KUP	870621.0	0.308372	1.129894	-3.000000	-0.571138	0.000000	0.897638	3.00
KUP2	870621.0	0.141023	0.917857	-3.000000	-0.604351	0.000000	0.746810	3.00
...	...	...	...	...	...	...	...	...
VSUMD10	870621.0	-0.050589	1.190094	-3.000000	-0.711650	-0.000984	0.632216	3.00
VSUMD20	870621.0	-0.055146	1.212688	-3.000000	-0.709434	-0.002235	0.632267	3.00
VSUMD30	870621.0	-0.054862	1.220845	-3.000000	-0.708033	-0.002740	0.632148	3.00
VSUMD60	870621.0	-0.074790	1.242467	-3.000000	-0.720635	-0.004804	0.617355	3.00
Ref($close, -2)/Ref($close, -1) - 1	870621.0	0.006053	0.998771	-1.718467	-0.858951	0.006070	0.871049	1.73

159 rows × 8 columns

		KMID	KLEN	KMID2	KUP	KUP2	KLOW	KLOW2	KSFT	KSFT2	O
datetime	instrument										
2008-01-02	SH600000	0.571643	1.800068	0.253804	3.000000	0.947448	2.712642	0.656584	0.216772	0.095455	-(
	SH600004	3.000000	1.715257	1.435924	-0.632512	-0.887051	-1.055251	-1.075074	2.626281	1.184890	-2
	SH600006	0.698338	0.598130	0.470045	0.197643	-0.192792	2.230872	1.214465	1.157374	0.772598	-(

	SH600007	3.000000	3.000000	1.189876	0.174635	-0.652805	0.505547	-0.554218	3.000000	1.045781	-3
	SH600008	2.819362	3.000000	0.929474	3.000000	0.409912	-0.508966	-0.888653	1.268276	0.414676	-2

5 rows × 159 columns

		KMID	KLEN	KMID2	KUP	KUP2	KLOW	KLOW2	KSFT	KSFT2	(
datetime	instrument										
2020-09-23	SZ300413	0.428904	0.112254	0.364781	1.316138	1.076447	0.355149	0.166595	0.022342	0.018848	-
	SZ300433	0.215259	0.525347	0.149562	1.477475	0.807947	2.020149	1.136760	0.258749	0.178299	-
	SZ300498	0.132981	-1.006315	0.287629	0.359085	2.108934	-0.893320	-0.712529	-0.277052	-0.594305	-
	SZ300601	3.000000	3.000000	1.264779	0.295952	-0.759123	0.738728	-0.674729	3.000000	1.108919	-
	SZ300628	0.090940	-0.520556	0.117773	1.237139	2.081331	-0.363078	-0.147186	-0.449953	-0.577913	-

5 rows × 159 columns

```
obj.head(-50)
```

		KMID	KLEN	KMID2	KUP	KUP2	KLOW	KLOW2	KSFT	KSFT2
datetime	instrument									
2008-01-02	SH600000	0.571643	1.800068	0.253804	3.000000	0.947448	2.712642	0.656584	0.216772	0.095455
	SH600004	3.000000	1.715257	1.435924	-0.632512	-0.887051	-1.055251	-1.075074	2.626281	1.184890
	SH600006	0.698338	0.598130	0.470045	0.197643	-0.192792	2.230872	1.214465	1.157374	0.772598
	SH600007	3.000000	3.000000	1.189876	0.174635	-0.652805	0.505547	-0.554218	3.000000	1.045781
	SH600008	2.819362	3.000000	0.929474	3.000000	0.409912	-0.508966	-0.888653	1.268276	0.414676
...	...	...	...	...	...	...	...	...	...	...
2020-09-23	SZ002236	0.725078	-0.689579	1.091397	-0.268482	0.111917	-1.055251	-1.075074	0.396331	0.591643
	SZ002241	-0.274159	0.496149	-0.192983	0.291404	-0.080882	3.000000	1.966044	0.535500	0.373838
	SZ002252	0.834937	-0.182653	0.845348	-0.299331	-0.305924	0.118062	0.154588	0.836335	0.839783
	SZ002271	-0.312192	-0.670053	-0.461269	-0.152796	0.277335	0.125720	0.731110	-0.205317	-0.30085[illegible]
	SZ002304	0.434145	0.066713	0.378592	-0.228322	-0.342788	1.981101	1.665740	0.996324	0.861674

870571 rows × 159 columns

```
all_stocks = obj.index.get_level_values("instrument").unique().tolist()

print(f"Total unique stocks: {len(all_stocks)}")
print(all_stocks[:20])
```

```
Total unique stocks: 682
['SH600000', 'SH600004', 'SH600006', 'SH600007', 'SH600008', 'SH600009', 'SH600010', '
```

```
# -*- coding: utf-8 -*-

# -- Sheet --

# Core
import os, math, random, warnings
from dataclasses import dataclass, field
from typing import Tuple, List
from pathlib import Path

# Numerics
import numpy as np
import pandas as pd
```

```python
# Torch
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader, Sampler

# Stats
from scipy.stats import spearmanr

# Misc
from tqdm.auto import tqdm

# For clean logs
warnings.filterwarnings("ignore")

print("Python:", os.sys.version)
print("Torch:", torch.__version__)
print("CUDA available:", torch.cuda.is_available())


def set_seed(seed: int = 43):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

set_seed(43)


torch.backends.cudnn.deterministic = False
torch.backends.cudnn.benchmark = True
torch.set_float32_matmul_precision("high")
if torch.cuda.is_available():
    torch.backends.cuda.matmul.allow_tf32 = True
    torch.backends.cudnn.allow_tf32 = True


# ==== EDIT THESE IF NEEDED ====
DATASET_PATH = Path("csi_data.pkl")
SEQ_LEN      = 1
NUM_LATENT   = 158
NUM_FACTORS  = 1
HIDDEN_SIZE  = 1
```

```
NUM_PORTFOLIO = 1

# Train/val/test windows from your description
TRAIN_START = "2010-01-01"
TRAIN_END   = "2017-12-31"

VAL_START   = "2018-01-03"
VAL_END     = "2018-12-29"

TEST_START  = "2019-01-02"
TEST_END    = "2020-09-20"

LR           = 1e-3
EPOCHS       = 1

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

def rankic(df: pd.DataFrame, label_col="LABEL0", pred_col="Pred") -> pd.DataFrame:
    """Compute per-day RankIC and overall mean/IR."""
    dates = df.index.get_level_values(0).unique()
    daily = []
    for d in dates:
        dd = df.loc[d]
        ric, _ = spearmanr(dd[label_col].rank(), dd[pred_col].rank())
        daily.append({"datetime": d, "RankIC": ric})
    daily_df = pd.DataFrame(daily).set_index("datetime").sort_index()
    mean = daily_df["RankIC"].mean()
    std  = daily_df["RankIC"].std()
    ir   = (mean / std) if std and std == std else np.nan
    summary = pd.DataFrame({"RankIC":[mean], "RankIC_IR":[ir]})
    return daily_df, summary
import bisect

def np_ffill(arr: np.ndarray) -> np.ndarray:
    mask = np.isnan(arr)
    idx = np.where(~mask, np.arange(mask.shape[0]), 0)
    np.maximum.accumulate(idx, axis=0, out=idx)
    return arr[idx]

class TSDataSampler:
    """
    Time-series sampler for a MultiIndex (datetime, instrument) DataFrame.
    Returns windows of length step_len per instrument for a given date.
    """

    def __init__(self, data: pd.DataFrame, start, end, step_len: int, fillna_type: st
```

```
r = "none", dtype=None, flt_data=None):
        assert data.index.names == ["datetime", "instrument"]
        self.step_len = step_len
        self.fillna_type = fillna_type
        self.data = data.sort_index()
        self.data_arr = data.to_numpy(dtype=dtype)
        self.data_arr = np.append(self.data_arr, np.full((1, self.data_arr.shape[1]),
 np.nan, dtype=self.data_arr.dtype), axis=0)
        self.nan_idx = -1  # last row is NaNs
        self.idx_df, self.idx_map = self.build_index(self.data)
        self.data_index = self.data.index

        if flt_data is not None:
            flt = flt_data.reindex(self.data_index).fillna(False).astype(bool)
            self.idx_map = self._flt_idx_map(flt, self.idx_map)
            self.data_index = self.data_index[flt]

        self.idx_map = self._idx_map2arr(self.idx_map)
        self.start_idx, self.end_idx = self.data_index.slice_locs(start=pd.Time-
stamp(start), end=pd.Timestamp(end))
        self.idx_arr = np.array(self.idx_df.values, dtype=np.float64)

    @staticmethod
    def build_index(data: pd.DataFrame) -> tuple:
        idx_df = pd.Series(range(data.shape[0]), index=data.index, dtype=object).un-
stack()
        idx_df = idx_df.sort_index().sort_index(axis=1)
        idx_map = {}
        for _, row in idx_df.iterrows():
            for j, real_idx in enumerate(row):
                if not np.isnan(real_idx):
                    idx_map[real_idx] = (idx_df.index.get_loc(row.name), j)
        return idx_df, idx_map

    @staticmethod
    def _idx_map2arr(idx_map):
        dtype = np.int32
        NO = (np.iinfo(dtype).max, np.iinfo(dtype).max)
        max_idx = max(idx_map.keys())
        arr_map = [idx_map.get(i, NO) for i in range(max_idx + 1)]
        return np.array(arr_map, dtype=dtype)

    @staticmethod
    def _flt_idx_map(flt_data, idx_map):
        idx = 0
        new_idx_map = {}
        for i, exist in enumerate(flt_data):
```

```
            if exist:
                new_idx_map[idx] = idx_map[i]
                idx += 1
        return new_idx_map

    def get_index(self):
        return self.data_index[self.start_idx:self.end_idx]

    def _rowcol_from_idx(self, idx) -> tuple:
        if isinstance(idx, (int, np.integer)):
            real_idx = self.start_idx + idx
            if self.start_idx <= real_idx < self.end_idx:
                i, j = self.idx_map[real_idx]
            else:
                raise KeyError(f"{real_idx} out of bounds")
        elif isinstance(idx, tuple):
            date, inst = idx
            date = pd.Timestamp(date)
            i = bisect.bisect_right(self.idx_df.index, date) - 1
            j = bisect.bisect_left(self.idx_df.columns, inst)
        else:
            raise NotImplementedError
        return i, j

    def _window_indices(self, row: int, col: int) -> np.ndarray:
        indices = self.idx_arr[max(row - self.step_len + 1, 0): row + 1, col]
        if len(indices) < self.step_len:
            indices = np.concatenate([np.full((self.step_len - len(indices),), np.-
nan), indices])
        if self.fillna_type == "ffill":
            indices = np_ffill(indices)
        elif self.fillna_type == "ffill+bfill":
            indices = np_ffill(np_ffill(indices)[::-1])[::-1]
        else:
            assert self.fillna_type == "none"
        return indices

    def __getitem__(self, idx):
        if isinstance(idx, (list, np.ndarray)):
            windows = [self._window_indices(*self._rowcol_from_idx(i)) for i in idx]
            indices = np.concatenate(windows)
        else:
            indices = self._window_indices(*self._rowcol_from_idx(idx))

        indices = np.nan_to_num(indices.astype(np.float64), nan=self.nan_idx).astype(
int)
        data = self.data_arr[indices]
```

```
        actual_idx = self.data_index[indices]
        if isinstance(idx, (list, np.ndarray)):
            data = data.reshape(-1, self.step_len, *data.shape[1:])
        return data, actual_idx

    def __len__(self):
        return self.end_idx - self.start_idx


class TSDatasetH(Dataset):
    def __init__(self, data, step_len=1, **kwargs):
        self.step_len = step_len
        self.data = data
        self.sampler = TSDataSampler(data=data, step_len=step_len, **kwargs)

    def __getitem__(self, idx):
        return self.sampler[idx]

    def __len__(self):
        return len(self.sampler)


class DateGroupedBatchSampler(Sampler):
    """Yield one batch per date (all instruments that date)."""
    def __init__(self, data_source: TSDatasetH, shuffle: bool = False):
        self.data_source = data_source
        self.shuffle = shuffle
        self.grouped_indices = self._group_indices()

    def _group_indices(self):
        start_idx = self.data_source.sampler.start_idx
        end_idx = self.data_source.sampler.end_idx
        data_index = self.data_source.sampler.data_index[start_idx:end_idx]
        ser = pd.Series(range(len(data_index)), index=data_index.get_level_values("da
tetime"))
        grouped = ser.groupby(level="datetime").apply(list).values
        return list(grouped)

    def __iter__(self):
        if self.shuffle:
            np.random.shuffle(self.grouped_indices)
        for group in self.grouped_indices:
            yield group

    def __len__(self):
        return len(self.grouped_indices)
```

```
def custom_collate_fn(batch):
    data, indices = zip(*batch)
    data = torch.utils.data.dataloader.default_collate(data)
    indices = [list(ix) for ix in indices]
    return data, indices


def init_data_loader(df, step_len, shuffle, start, end, select_feature=None):
    if select_feature is not None:
        df = df[select_feature]
    ds = TSDatasetH(df, step_len=step_len, start=start, end=end, fillna_type="ffill+bfill")
    sampler = DateGroupedBatchSampler(ds, shuffle=shuffle)
    dl = DataLoader(ds, batch_sampler=sampler, collate_fn=custom_collate_fn, pin_memory=True)
    return dl

import pandas as pd
dataset = pd.read_pickle("csi_data.pkl").copy()

#dataset = pd.read_pickle("/data/workspace_files/sp500_data_20250106_20250126.pkl").copy()

# Keep only the 158 features + 1 label (already your format).
# Rename last column to LABEL0 (the repo convention)
dataset.rename(columns={dataset.columns[-1]: "LABEL0"}, inplace=True)

# Basic checks
print(dataset.shape, "rows x cols")
print("Index names:", dataset.index.names)
print("Date range:", dataset.index.get_level_values(0).min(), "→", dataset.index.get_level_values(0).max())
print("Unique dates:", dataset.index.get_level_values(0).nunique())
print("Unique instruments:", dataset.index.get_level_values(1).nunique())

# Peek
display(dataset.head(3))

%pip install -q --upgrade "numpy>=2.0,<3.0" "pandas>=2.2"
import numpy as np, pandas as pd
print("NumPy:", np.__version__, "Pandas:", pd.__version__)

class FeatureExtractor(nn.Module):
    def __init__(self, num_latent: int, hidden_size: int, num_layers: int = 1):
        super().__init__()
        self.normalize = nn.LayerNorm(num_latent)
```

```
        self.linear = nn.Linear(num_latent, num_latent)
        self.act = nn.LeakyReLU()
        self.gru = nn.GRU(num_latent, hidden_size, num_layers, batch_first=True)

    def forward(self, x):
        # x: (N, T, F)
        x = self.normalize(x)
        x = self.act(self.linear(x))
        h, _ = self.gru(x)
        return h[:, -1, :]  # (N, H)


class FactorEncoder(nn.Module):
    def __init__(self, num_factors: int, num_portfolio: int, hidden_size: int):
        super().__init__()
        self.linear = nn.Linear(hidden_size, num_portfolio)
        self.softmax = nn.Softmax(dim=0)  # cross-sectional softmax (N dimension)
        self.mu = nn.Linear(num_portfolio, num_factors)
        self.sigma = nn.Linear(num_portfolio, num_factors)
        self.softplus = nn.Softplus()

    def forward(self, stock_latent, returns):
        # stock_latent: (N,H), returns: (N,1)
        w = self.softmax(self.linear(stock_latent))  # (N,M)
        if returns.dim() == 1:
            returns = returns.unsqueeze(1)
        port_ret = torch.mm(w.T, returns)  # (M,1)
        m = self.mu(port_ret.squeeze(1))
        s = self.softplus(self.sigma(port_ret.squeeze(1)))
        return m, s


class AlphaLayer(nn.Module):
    def __init__(self, hidden_size: int):
        super().__init__()
        self.fc = nn.Linear(hidden_size, hidden_size)
        self.act = nn.LeakyReLU()
        self.mu = nn.Linear(hidden_size, 1)
        self.sig = nn.Linear(hidden_size, 1)
        self.softplus = nn.Softplus()

    def forward(self, stock_latent):
        h = self.act(self.fc(stock_latent))
        mu = self.mu(h)
        sig = self.softplus(self.sig(h))
        return mu, sig
```

```
class BetaLayer(nn.Module):
    def __init__(self, hidden_size: int, num_factors: int):
        super().__init__()
        self.fc = nn.Linear(hidden_size, num_factors)

    def forward(self, stock_latent):
        return self.fc(stock_latent)  # (N,K)


class FactorDecoder(nn.Module):
    def __init__(self, alpha_layer: AlphaLayer, beta_layer: BetaLayer):
        super().__init__()
        self.alpha = alpha_layer
        self.beta = beta_layer

    def forward(self, stock_latent, factor_mu, factor_sigma, return_mu_sigma=False, s
ample=True):
        # alpha/beta
        a_mu, a_sig = self.alpha(stock_latent)     # (N,1), (N,1)
        beta = self.beta(stock_latent)             # (N,K)

        # factors
        f_mu = factor_mu.view(-1, 1)               # (K,1)
        f_sig = factor_sigma.view(-1, 1).clamp_min(1e-6)

        # predictive mean/var
        mu = a_mu + torch.matmul(beta, f_mu)       # (N,1)
        sig = torch.sqrt(a_sig**2 + torch.matmul(beta**2, f_sig**2) + 1e-6)

        if return_mu_sigma:
            return mu, sig
        if sample:
            eps = torch.randn_like(sig)
            return mu + eps * sig
        return mu  # deterministic


class AttentionLayer(nn.Module):
    def __init__(self, hidden_size: int):
        super().__init__()
        self.query = nn.Parameter(torch.randn(hidden_size))
        self.key = nn.Linear(hidden_size, hidden_size)
        self.val = nn.Linear(hidden_size, hidden_size)
        self.drop = nn.Dropout(0.1)

    def forward(self, stock_latent):
```

```
        K = self.key(stock_latent)   # (N,H)
        V = self.val(stock_latent)   # (N,H)
        att = torch.matmul(self.query, K.T)  # (N,)
        att = att / math.sqrt(K.shape[1] + 1e-6)
        att = self.drop(att)
        att = F.relu(att)
        att = F.softmax(att, dim=0)
        if torch.isnan(att).any() or torch.isinf(att).any():
            return torch.zeros_like(V[0])
        ctx = torch.matmul(att, V)   # (H,)
        return ctx


class FactorPredictor(nn.Module):
    def __init__(self, hidden_size: int, num_factors: int):
        super().__init__()
        self.attn = nn.ModuleList([AttentionLayer(hidden_size) for _ in range(num_-
factors)])
        self.fc = nn.Linear(hidden_size, hidden_size)
        self.act = nn.LeakyReLU()
        self.mu = nn.Linear(hidden_size, 1)
        self.sig = nn.Linear(hidden_size, 1)
        self.softplus = nn.Softplus()

    def forward(self, stock_latent):
        heads = [l(stock_latent) for l in self.attn]   # list of (H,)
        h = torch.stack(heads, dim=0)                  # (K,H)
        h = self.act(self.fc(h))
        mu = self.mu(h).view(-1)                       # (K,)
        sig = self.softplus(self.sig(h)).view(-1)      # (K,)
        return mu, sig


class FactorVAE(nn.Module):
    def __init__(self, feature_extractor, factor_encoder, factor_decoder, factor_pre-
dictor):
        super().__init__()
        self.feat = feature_extractor
        self.enc  = factor_encoder
        self.dec  = factor_decoder
        self.pred = factor_predictor

    @staticmethod
    def kl_divergence(mu1, sig1, mu2, sig2):
        # KL(q||p) for diagonal Gaussians; sum over dims
        sig2 = torch.clamp(sig2, min=1e-6)
        sig1 = torch.clamp(sig1, min=1e-6)
```

```
        return (torch.log(sig2/sig1) + (sig1**2 + (mu1 - mu2)**2)/(2*sig2**2) -
 0.5).sum()

    def forward(self, x, returns):
        # x: (N,T,F), returns: (N,1)
        z_stock = self.feat(x)
        post_mu, post_sig = self.enc(z_stock, returns)       # q(z|x,y)
        recon = self.dec(z_stock, post_mu, post_sig)          # sample for training
        prior_mu, prior_sig = self.pred(z_stock)              # p(z|x)

        rec_loss = F.mse_loss(recon, returns)
        kl = self.kl_divergence(post_mu, post_sig, prior_mu, prior_sig)
        loss = rec_loss + kl
        return loss, recon, post_mu, post_sig, prior_mu, prior_sig

    @torch.no_grad()
    def predict_mean_sigma(self, x):
        z_stock = self.feat(x)
        prior_mu, prior_sig = self.pred(z_stock)
        mu, sigma = self.dec(z_stock, prior_mu, prior_sig, return_mu_sigma=True, sam-
ple=False)
        return mu, sigma  # (N,1), (N,1)
```

```
train_loader = init_data_loader(
    dataset, step_len=SEQ_LEN, shuffle=True,
    start=TRAIN_START, end=TRAIN_END, select_feature=None
)

val_loader = init_data_loader(
    dataset, step_len=SEQ_LEN, shuffle=False,
    start=VAL_START, end=VAL_END, select_feature=None
)

test_loader = init_data_loader(
    dataset, step_len=SEQ_LEN, shuffle=False,
    start=TEST_START, end=TEST_END, select_feature=None
)

len(train_loader), len(val_loader), len(test_loader)

feature_extractor = FeatureExtractor(NUM_LATENT, HIDDEN_SIZE)
factor_encoder    = FactorEncoder(NUM_FACTORS, NUM_PORTFOLIO, HIDDEN_SIZE)
alpha_layer       = AlphaLayer(HIDDEN_SIZE)
```

```
beta_layer        = BetaLayer(HIDDEN_SIZE, NUM_FACTORS)
factor_decoder    = FactorDecoder(alpha_layer, beta_layer)
factor_predictor  = FactorPredictor(HIDDEN_SIZE, NUM_FACTORS)

model = FactorVAE(feature_extractor, factor_encoder, factor_decoder, factor_predic-
tor).to(DEVICE)

optimizer = torch.optim.Adam(model.parameters(), lr=LR)
T_max = len(train_loader) * EPOCHS if len(train_loader) > 0 else EPOCHS
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=T_max)

sum(p.numel() for p in model.parameters())  # parameter count

import copy

class EarlyStopping:
    """
    Stop training when the monitored metric stops improving.
    - mode='min' for losses; patience = epochs to wait after last improvement
    - min_delta = required improvement amount
    - restore_best=True will keep best weights in memory and restore on stop
    """
    def __init__(self, patience=6, min_delta=1e-4, mode='min', restore_best=True):
        self.patience = patience
        self.min_delta = min_delta
        self.mode = mode
        self.restore_best = restore_best

        self.best = None
        self.num_bad = 0
        self.best_state = None

    def _is_better(self, current, best):
        if best is None:
            return True
        if self.mode == 'min':
            return (best - current) > self.min_delta
        else:
            return (current - best) > self.min_delta

    def step(self, metric, model):
        """
        Returns True if training should stop.
        """
        if self.best is None or self._is_better(metric, self.best):
            self.best = metric
            self.num_bad = 0
```

```
            if self.restore_best:
                # keep best weights in memory
                self.best_state = copy.deepcopy(model.state_dict())
            return False

        self.num_bad += 1
        return self.num_bad > self.patience

def train_one_epoch(model, loader, optimizer, scheduler=None, device=DEVICE, grad_-
clip=1.0):
    model.train()
    total = 0.0
    with tqdm(total=len(loader), desc="Train") as pbar:
        for batch, _ in loader:
            x = batch[:, :, :-1].to(device).float()  # (N,T,158)
            y = batch[:, :, -1].to(device).float()
            y = y[:, -1].unsqueeze(1)                # (N,1)

            optimizer.zero_grad()
            loss, *_ = model(x, y)
            loss.backward()

            # gradient clipping helps on tiny/noisy sets
            if grad_clip is not None:
                torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=grad_-
clip)

            optimizer.step()
            if scheduler is not None:
                # keep per-batch cosine schedule if you’re using CosineAnnealingLR
                scheduler.step()

            total += loss.item()
            pbar.set_postfix(loss=loss.item())
            pbar.update(1)
    return total / max(1, len(loader))


@torch.no_grad()
def validate(model, loader, device=DEVICE):
    model.eval()
    total = 0.0
    with tqdm(total=len(loader), desc="Valid") as pbar:
        for batch, _ in loader:
            x = batch[:, :, :-1].to(device).float()
            y = batch[:, :, -1].to(device).float()
            y = y[:, -1].unsqueeze(1)
```

```
            loss, *_ = model(x, y)
            total += loss.item()
            pbar.update(1)
    return total / max(1, len(loader))

best_val = float("inf")
best_path = "factorvae_sp500_best.pt"

# patience=6 means: stop if no val improvement for 6 consecutive epochs
early = EarlyStopping(patience=6, min_delta=1e-4, mode='min', restore_best=True)

for epoch in range(1, EPOCHS + 1):
    tr = train_one_epoch(model, train_loader, optimizer, scheduler, DEVICE, grad_clip
=1.0)
    va = validate(model, val_loader, DEVICE)
    print(f"Epoch {epoch:03d} | train {tr:.6f} | val {va:.6f}")

    # Track best & save
    if va < best_val - 1e-4:
        best_val = va
        torch.save(model.state_dict(), best_path)
        print(f"  ↳ saved new best → {best_path}")

    # Early stopping check
    if early.step(va, model):
        print(f"Early stopping triggered at epoch {epoch}.")
        if early.restore_best and early.best_state is not None:
            model.load_state_dict(early.best_state)
            torch.save(model.state_dict(), best_path)
            print(f"  ↳ restored best weights and re-saved to {best_path}")
        break

# ---- Predict on test set (deterministic mean) & compute RankIC ----
# Uses sampler's full index to avoid nested-idx headaches.

# (Re)load best checkpoint if present
if os.path.exists("factorvae_sp500_best.pt"):
    model.load_state_dict(torch.load("factorvae_sp500_best.pt", map_location=DEVICE))

model.eval()
preds = []           # list of np arrays, one per date-batch
with torch.no_grad():
    for batch, _ in test_loader:     # ignore `_` indices
        x = batch[:, :, :-1].to(DEVICE).float()
        mu, sigma = model.predict_mean_sigma(x)      # (N,1)
        preds.append(mu.squeeze(1).cpu().numpy())    # shape (N,)
```

```
# 1) concatenate predictions
pred_arr = np.concatenate(preds, axis=0)           # to-
tal length = sum over all dates

# 2) get the *ordered* full MultiIndex for the test window
mi_full = list(test_loader.dataset.sampler.get_index())  # list of (datetime, instru-
ment) tuples

# 3) align index to preds by slicing sequentially
mi_seq = []
cursor = 0
for arr in preds:
    n = int(arr.shape[0])
    mi_seq.extend(mi_full[cursor: cursor + n])
    cursor += n

# 4) sanity check and build DataFrame
assert len(mi_seq) == pred_arr.shape[0], f"Index ({len(mi_seq)}) vs preds ({pred_ar-
r.shape[0]}) mismatch"
mi = pd.MultiIndex.from_tuples(mi_seq, names=["datetime","instrument"])
pred_df = pd.DataFrame({"Pred": pred_arr}, index=mi).sort_index()

# 5) join with labels and compute RankIC
labels = dataset[["LABEL0"]].copy()
eval_df = labels.join(pred_df, how="inner")

daily_rankic, summary = rankic(eval_df, label_col="LABEL0", pred_col="Pred")
display(daily_rankic)
display(summary)

# After you have pred_df with index (datetime, instrument)
labels_aligned = (
    dataset[["LABEL0"]]
    .groupby(level="instrument")
    .shift(-2)                        # align label to prediction date
    .dropna()
)

eval_df = labels_aligned.join(pred_df, how="inner")
daily_rankic, summary = rankic(eval_df, label_col="LABEL0", pred_col="Pred")
#display(daily_rankic)
display(summary)

# ===== Linear dynamic baseline (fast) =====
import math, os, numpy as np, pandas as pd, torch
import torch.nn as nn, torch.nn.functional as F
```

```
from scipy.stats import spearmanr
from tqdm.auto import tqdm

# --- tiny helpers ---
def inverse_normalize_labels(y: torch.Tensor) -> torch.Tensor:
    # y: (N,1) --- per-date batch
    N = y.shape[0]
    # ranks in [1..N]
    ranks = torch.argsort(torch.argsort(y.squeeze(1))) + 1
    u = (ranks.float() - 0.5) / float(N)
    z = math.sqrt(2.0) * torch.erfinv(2.0*u - 1.0)
    return z.view(-1,1)

@torch.no_grad()
def batch_spearman(y_true: torch.Tensor, y_pred: torch.Tensor) -> float:
    # y_true,y_pred: (N,1) for one date
    a = y_true.squeeze(1).cpu().numpy()
    b = y_pred.squeeze(1).cpu().numpy()
    if np.std(b) == 0:
        return 0.0
    r, _ = spearmanr(a, b)
    return 0.0 if (r != r) else float(r)

class EarlyStopping:
    def __init__(self, patience=6, mode='max', min_delta=1e-4):
        self.patience, self.mode, self.min_delta = patience, mode, min_delta
        self.best, self.bad = None, 0
        self.best_state = None
    def step(self, metric, model):
        if self.best is None:
            self.best = metric
            self.best_state = {k: v.detach().cpu().clone() for k,v in model.state_-
dict().items()}
            return False
        improve = (metric > self.best + self.min_delta) if self.mode=='max' else (met
ric < self.best - self.min_delta)
        if improve:
            self.best = metric
            self.bad = 0
            self.best_state = {k: v.detach().cpu().clone() for k,v in model.state_-
dict().items()}
        else:
            self.bad += 1
        return self.bad > self.patience

# --- model: linear on last time step features ---
class LinearDFM(nn.Module):
```

```
    def __init__(self, num_features):
        super().__init__()
        self.fc = nn.Linear(num_features, 1)
    def forward(self, x):          # x: (N,T,F)
        last = x[:, -1, :]         # (N,F)
        return self.fc(last)       # (N,1)

# ===== train / validate / test =====
def train_linear(model, optimizer, loaders, device, epochs=50, weight_decay=1e-5):
    train_loader, val_loader = loaders
    early = EarlyStopping(patience=8, mode='max')
    best_path = "baseline_linear_best.pt"
    for ep in range(1, epochs+1):
        # ---- train
        model.train()
        tr_loss = 0.0
        for batch, _ in tqdm(train_loader, desc=f"Linear Train {ep}", leave=False):
            x = batch[:, :, :-1].to(device).float()
            y = batch[:, :, -1].to(device).float()
            y = inverse_normalize_labels(y[:, -1].unsqueeze(1))
            pred = model(x)
            loss = F.mse_loss(pred, y)
            optimizer.zero_grad(set_to_none=True)
            loss.backward()
            optimizer.step()
            tr_loss += loss.item()
        tr_loss /= max(1, len(train_loader))

        # ---- validate rankIC
        model.eval()
        ric_sum, n_batches = 0.0, 0
        with torch.no_grad():
            for batch, _ in tqdm(val_loader, desc=f"Linear Valid {ep}", leave=False):
                x = batch[:, :, :-1].to(device).float()
                y = batch[:, :, -1].to(device).float()
                y = inverse_normalize_labels(y[:, -1].unsqueeze(1))
                pred = model(x)
                ric_sum += batch_spearman(y, pred)
                n_batches += 1
        val_rankic = ric_sum / max(1, n_batches)

        print(f"Epoch {ep:03d} | train_loss {tr_loss:.5f} | val_RankIC {val_ran-
kic:+.4f}")

        if early.step(val_rankic, model):
            print(f"Early stop at epoch {ep}. Restoring best...")
            model.load_state_dict(early.best_state)
```

```
            torch.save(model.state_dict(), best_path)
            break
        torch.save(model.state_dict(), best_path)  # always keep last/best
    # restore best if not already
    if early.best_state is not None:
        model.load_state_dict(early.best_state)

@torch.no_grad()
def eval_test_rankic(model, test_loader, device):
    model.eval()
    ric_list = []
    for batch, _ in tqdm(test_loader, desc="Linear Test", leave=False):
        x = batch[:, :, :-1].to(device).float()
        y = batch[:, :, -1].to(device).float()
        y = inverse_normalize_labels(y[:, -1].unsqueeze(1))
        pred = model(x)
        ric_list.append(batch_spearman(y, pred))
    ric = float(np.mean(ric_list))
    ir  = float(ric / (np.std(ric_list)+1e-12))
    print(f"[Linear] Test RankIC={ric:+.4f} | IR={ir:+.4f}")
    return ric, ir

# ===== run it (assumes train_loader/val_loader/test_loader exist) =====
lin = LinearDFM(NUM_LATENT).to(DEVICE)
opt = torch.optim.Adam(lin.parameters(), lr=1e-3, weight_decay=1e-5)
train_linear(lin, opt, (train_loader, val_loader), DEVICE, epochs=EPOCHS)
eval_test_rankic(lin, test_loader, DEVICE)
```

```
Python: 3.11.10 (main, Aug  6 2025, 09:13:17) [GCC 11.4.0]
Torch: 2.7.0+cu126
CUDA available: True
(870621, 159) rows x cols
Index names: ['datetime', 'instrument']
Date range: 2008-01-02 00:00:00 → 2020-09-23 00:00:00
Unique dates: 3046
Unique instruments: 682
ERROR: pip's dependency resolver does not currently take into account all the packages
tensorflow 2.19.0 requires numpy<2.2.0,>=1.26.0, but you have numpy 2.3.2 which is inc
gensim 4.3.3 requires numpy<2.0,>=1.18.5, but you have numpy 2.3.2 which is incompatib
scipy 1.13.1 requires numpy<2.3,>=1.22.4, but you have numpy 2.3.2 which is incompatib
Note: you may need to restart the kernel to use updated packages.
NumPy: 1.26.4 Pandas: 2.2.3
Epoch 001 | train 1.398899 | val 1.066486
  ↳ saved new best → factorvae_sp500_best.pt
Epoch 001 | train_loss 1.01157 | val_RankIC +0.0260
[Linear] Test RankIC=+0.0299 | IR=+0.1928
```

		KMID	KLEN	KMID2	KUP	KUP2	KLOW	KLOW2	KSFT	KSFT2	OPEN
datetime	instrument										
2008-01-02	SH600000	0.571643	1.800068	0.253804	3.000000	0.947448	2.712642	0.656584	0.216772	0.095455	-0.56
	SH600004	3.000000	1.715257	1.435924	-0.632512	-0.887051	-1.055251	-1.075074	2.626281	1.184890	-2.98
	SH600006	0.698338	0.598130	0.470045	0.197643	-0.192792	2.230872	1.214465	1.157374	0.772598	-0.68

3 rows × 159 columns

	RankIC
datetime	
2019-01-02	-0.018183
2019-01-03	0.085388
2019-01-04	0.054325
2019-01-07	-0.052135
2019-01-08	0.123606
...	...
2020-09-14	-0.155044
2020-09-15	0.053940
2020-09-16	0.156411
2020-09-17	-0.116219
2020-09-18	-0.254085

416 rows × 1 columns

	RankIC	RankIC_IR
0	0.004136	0.042744

	RankIC	RankIC_IR
0	-0.001964	-0.021021

```
(0.02990900206086925, 0.19282449845561112)
```

```python
def inverse_normalize_labels(y: torch.Tensor) -> torch.Tensor:
    # y: (N,1) --- per-date batch
    N = y.shape[0]
    # ranks in [1..N]
    ranks = torch.argsort(torch.argsort(y.squeeze(1))) + 1
    u = (ranks.float() - 0.5) / float(N)
    z = math.sqrt(2.0) * torch.erfinv(2.0*u - 1.0)
    return z.view(-1,1)

@torch.no_grad()
def batch_spearman(y_true: torch.Tensor, y_pred: torch.Tensor) -> float:
    # y_true,y_pred: (N,1) for one date
    a = y_true.squeeze(1).cpu().numpy()
    b = y_pred.squeeze(1).cpu().numpy()
    if np.std(b) == 0:
        return 0.0
    r, _ = spearmanr(a, b)
    return 0.0 if (r != r) else float(r)


@torch.no_grad()
def eval_test_rankic(model, test_loader, device):
    model.eval()
    ric_list = []
    for batch, _ in tqdm(test_loader, desc="Linear Test", leave=False):
        x = batch[:, :, :-1].to(device).float()
        y = batch[:, :, -1].to(device).float()
        y = inverse_normalize_labels(y[:, -1].unsqueeze(1))
        pred = model(x)
        ric_list.append(batch_spearman(y, pred))
    ric = float(np.mean(ric_list))
    ir  = float(ric / (np.std(ric_list)+1e-12))
    print(f"[Linear] Test RankIC={ric:+.4f} | IR={ir:+.4f}")
    return ric, ir

eval_test_rankic(lin, test_loader, DEVICE)
```

```
[Linear] Test RankIC=+0.0299 | IR=+0.1928

(0.029909002060869 25, 0.19282449845561112)
```