GNN Toolkit (Python)
code overview and usage

August 9, 2025

Contents

1 Overview

This package builds graph snapshots from Pandas DataFrames and trains node–level models with PyTorch Geometric. It provides:

- a central run config (`GNNConfig`) with YAML/JSON I/O,
- utilities to convert `DataFrames` → `Data` snapshots → `DataLoader`s,
- an extensible model zoo (GCN, GraphSAGE, GAT, GATv2, Simple baseline),
- loss & metric registries (BCE/focal BCE/MSE/MAE/Huber, AUC/ACC/MAE/R^2),

- optional class imbalance handling and per–node weighting,
- a high-level `Trainer` with early stopping, schedulers, TensorBoard and checkpointing.

Requirements Python 3.10+, `torch`, `torch_geometric`, `pandas`, `numpy`, `scikit-learn` (for AUC), optional `pyyaml`.

2 Quick Start

Listing 1: Load frames, build loaders, train, evaluate

```python
from jp_da_imb.gnn.config import load_config
from jp_da_imb.gnn.data.preprocess import scale_targets
from jp_da_imb.gnn.data_loading import build_dataloaders
from jp_da_imb.gnn.trainer import Trainer
import networkx as nx

# 1) Load settings
cfg = load_config("configs/gnn_run.yaml")

# 2) Prepare node/edge frames dicts keyed by region and "src->dst"
node_frames = {"tokyo": df_tokyo, "kansai": df_kansai, ...} # indexed by timestamp
edge_frames = {"tokyo->kansai": df_tk, "kansai->tokyo": df_kt, ...} # indexed by timestamp

# 3) Optional: scale target(s) using training split stats; keep inverse() for reporting
scaled_nodes, scaler = scale_targets(node_frames, cfg)

# 4) Graph topology
g = nx.DiGraph()
g.add_nodes_from(sorted(node_frames))
g.add_edges_from([k.split("->") for k in edge_frames])

# 5) Dataloaders
train_dl, val_dl, test_dl = build_dataloaders(
    node_frames=scaled_nodes, edge_frames=edge_frames, graph=g, cfg=cfg
)

# 6) Train
trainer = Trainer(cfg, train_dl, val_dl, test_dl, log_dir=None, ckpt_dir="./ckpts")
trainer.fit()

# 7) Evaluate (+ optional tidy DataFrame)
stats, df = trainer.evaluate(split="test", return_df=True)
df_inv = scaler.inverse(df) # add 'pred_orig' / 'target_orig' cols (de-normalised)
print(stats)
```

3 Configuration (`config.py`)

3.1 `GNNConfig`

- **Task/model**: `task="node_clf"`, `model_name` in {`gcn`, `graphsage`, `gat`, `gatv2`, `simple`}, `num_layers`, `hidden_dim`, `heads`, `dropout`, `norm` in {`batch`, `layer`, `None`}.
- **Data**: `target_col`, `split_mode` in {`date`, `ratio`}, `cutoff_date`, `val_ratio`, `test_ratio`, `shuffle_in_split`.
- **Optimisation**: `batch_size`, `lr`, `epochs`, `patience`, `grad_clip`, `optimiser` in {`adam`, `adamw`, `sgd`} and `optimiser_kwargs`.

- **Scheduler**: `lr_scheduler` in {`step`, `plateau`, `cosine`, `onecycle`} plus `lr_scheduler_kwargs`; `print_lr_each_epoch`.
- **Loss/metric**: `loss_fn` in {`bce`, `focal_bce`, `mse`, `mae`, `huber`}; `class_weights` (`"auto"` / scalar / list); `node_pos_weights` (per node); `metric` in {`acc`, `auc`, `mae`, `r2`}.
- **Misc**: `in_dropout`, `edge_dropout`, `device`, `run_name`, `seed`.

I/O helpers
- `GNNConfig.load(x)` accepts an existing instance, a `dict`, or a YAML/JSON path.
- `to_dict()`, `to_yaml()`.
- Free function `load_config(x)` mirrors `GNNConfig.load`.

Example YAML

```yaml
task: node_clf
model_name: gatv2
num_layers: 2
hidden_dim: 64
heads: 4
dropout: 0.5
norm: batch
split_mode: date
cutoff_date: 2023-07-01
val_ratio: 0.10
test_ratio: 0.10
batch_size: 32
lr: 1e-3
epochs: 100
patience: 10
optimiser: adamw
lr_scheduler: plateau
lr_scheduler_kwargs: { factor: 0.5, patience: 3 }
loss_fn: focal_bce
class_weights: auto
metric: auc
device: cuda
run_name: demo_run
```

4 Data & Snapshots (`data_loading.py`)

4.1 `make_snapshots`

Converts aligned DataFrames into an ordered list of `torch_geometric.data.Data`:
- Node order is `sorted(graph.nodes)`; edge order follows the graph's edges.
- Each snapshot has: `x` [N, F_node], `edge_index` [2, E], `edge_attr` [E, F_edge] (optional), `y` [N, T], optional `node_weight` [N] or [N, T], and `snap_time` (int nanoseconds).
- Multi-target: set `cfg.target_cols` (else falls back to `cfg.target_col`).
- Optional per-node weights via `cfg.node_pos_weights`.

4.2 `split_snapshots`

- `split_mode="date"`: uses `cutoff_date` to split; post-cutoff half is val and half is test.

- `split_mode="ratio"`: uses `val_ratio`, `test_ratio`.

4.3 `build_dataloaders`

High-level wrapper: frames → snapshots → splits → 3 `DataLoader`s. Batch size and `shuffle_in_split` come from `GNNConfig`.

5 Preprocessing (`data/preprocess.py`)

5.1 `scale_targets`

Z-scores `target_col` per region using *training* subset only. Returns:
- scaled node frames (same keys);
- a `TargetScaler` that holds per-region `mu/sigma` and `inverse(df)` to add de-scaled columns (`pred_orig`, `target_orig`, or indexed for multi-T).

6 Model Registry & Zoo (`models/registry.py`, `models.py`)

6.1 Registry

`register("name")` decorates a builder; `build_model(name=..., **kw)` instantiates it.

6.2 Built-ins

- `gcn`: wrapper around `torch_geometric.nn.GCN` with `act="relu"`, `norm` {`batch`,`layer`,`None`}.
- `graphsage` (`sage` alias): adapter that calls `GraphSAGE` with `Data`.
- `gat`: wrapper for `GAT`.
- `gatv2`: flexible stack of `GATv2Conv` layers (keeps concatenated heads), optional `edge_attr`, per-layer norm & ReLU, final linear head.
- `simple`/`baseline`: two `GCNConv` layers + linear head.

7 Losses, Class Weights, Metrics

7.1 Loss registry (`loss/loss_functions.py`)

`build_loss(name=..., class_weights=..., node_reduction=..., **loss_kw)` returns a callable:
- `"bce"`: `binary_cross_entropy_with_logits`, optional global `pos_weight`.
- `"focal_bce"`: focal term $(1 - p\_t)^{\gamma}$ with optional `alpha`.
- `"mse"`, `"mae"`, `"huber"`.
- Per-node weights are broadcast and applied before `"mean"`/`"sum"` reduction.

7.2 Class imbalance (`loss/class_weights.py`)

`compute_class_weights(train_loader)` makes one pass over training targets and returns

$$\texttt{pos\_weight} = \frac{\#\text{neg} + \varepsilon}{\#\text{pos} + \varepsilon}$$

as a scalar (single target) or vector (per target).

7.3 Metrics (`loss/metrics.py`)

Registry with:
- `"auc"`: ROC AUC on sigmoid(logits).
- `"acc"`: threshold 0.5 on sigmoid(logits).
- `"mae"`: L1 between predictions and targets.
- `"r2"`: $1 - \mathrm{SS}_{\mathrm{res}}/\mathrm{SS}_{\mathrm{tot}}$ (regression).

8 Training Loop (`trainer.py`)

8.1 Trainer

Initialises device, infers `d_in`, `d_out`, optional `edge_dim` from a training batch, builds the model via the registry, constructs the loss callable (supports `class_weights="auto"`), selects optimiser (`adam/adamw/sgd`) and LR scheduler (`step/plateau/cosine/onecycle`).

Features
- `_train_epoch`: standard loop with optional gradient clipping.
- `_eval`: averages loss & metric over a loader.
- `fit`: main loop with scheduler stepping, TensorBoard logging, console prints, early stopping (`patience`) and `best.pt` checkpoint writing.
- `evaluate(split, return_df=False)`: computes mean loss/metric and optionally returns a tidy DataFrame with rows (`snap_time`, `node`, `pred[#]`, `target[#]`).
- `predict(...)`: (post-fit) forward-only helper analogous to `evaluate`.

9 Design Notes

- Timestamps across all frames are intersected to avoid leakage/misalignment.
- Node order is fixed (`sorted(graph.nodes)`) so tensors align across snapshots.
- Losses are callables to make weighting strategies pluggable without touching training code.
- Config is the single source of truth; YAML/JSON keeps runs reproducible.

10 API Index

Module	Functions / Classes
`config`	`GNNConfig`, `load_config`
`data_loading`	`make_snapshots`, `split_snapshots`, `build_dataloaders`
`data/preprocess`	`scale_targets`, `TargetScaler.inverse`
`models/registry`	`register`, `build_model`
`models`	`build_gcn`, `build_graphsage`, `build_gat`, `build_gatv2`, `build_simple`
`loss/loss_functions`	`register`, `build_loss` (`bce`, `focal_bce`, `mse`, `mae`, `huber`)
`loss/class_weights`	`compute_class_weights`
`loss/metrics`	`register`, `METRICS` (`auc`, `acc`, `mae`, `r2`)
`trainer`	`Trainer.fit`, `Trainer.evaluate`, `Trainer.predict`