RW_GNN Code: Functional Overview and Workflow

Yunxaing Wang

June 22, 2025

1 Overall Workflow and Structure

Algorithm 1 Quick start with RW_GNN

```
python rwgnn_dis_s6.py --dataset Cora \
    --alpha 5.0 --c 0.01 \
    --num-walks-per-node 10 \
    --epochs 50 --walk-bluetype levy
```

1. **Argument Parsing**: Use `argparse` to read dataset name, learning rate, dropout, weight decay, random-walk hyperparameters (walks per node, random starts per graph), early stopping patience, etc.
2. **Environment Setup**: Fix NumPy and PyTorch seeds, select CPU/GPU device, create checkpoint directory.
3. **Data Loading**:
   - If `--dataset Cora`, instantiate `LocalCoraDataset`: read raw `ind.cora.*` files, merge and re-order features/labels, build `edge_index`, split into 140/500/1000 train/val/test masks with no overlap or empty sets.
   - Otherwise, use `torch_geometric.datasets.TUDataset`, and apply one-hot degree encoding for graphs without node attributes.
4. **Model Construction**: Instantiate `RW_GNN`, which encapsulates `ExplicitRandomWalkEncoder`, passing all hyperparameters; set up Adam optimizer and a ReduceLROnPlateau scheduler.
5. **Training and Evaluation**:
   - **Cora**: Single-graph training over epochs, record train/val/test loss and accuracy, save best model, trigger early stopping, and emit overfitting warnings when validation loss rises.
   - **TUDataset**: 10-fold cross-validation, 90% train / 10% validation per fold, save best model each fold, report average test accuracy across folds.
6. **Visualization**: Call `plot_training_curves` to generate and save a PNG of loss and accuracy curves side by side.

2 Key Classes and Functions

2.1 Class `ExplicitRandomWalkEncoder`

`__init__`(input_dim, hidden_dim, walk_length, num_walks_per_start, device, num_random_starts_per_graph
Initialize the encoder: set up a two-layer feature encoder with ReLU and Dropout, a two-layer GRU with dropout, a distance cache, and all hyperparameters.

`clear_cache()` Clear the cached distance matrices when parameters like `alpha` or `walk_type` change.

`compute_graph_distances`**(edge_index, num_nodes, graph_nodes)** Compute the shortest-path distance matrix and distance-based transition probabilities for a subgraph, with caching.

- `edge_index`: edge list of the subgraph $[2 \times E]$.
- `graph_nodes`: global node indices of the subgraph $[N]$.
- Returns $(\text{walk\_probs}, \text{distance\_matrix})$.

`_compute_simple_distance_matrix`**(adj, num_nodes, max_distance)** Approximate distances via 1/2/3-hop adjacency powers.

`_compute_shortest_path_distances_scipy`**(adj, num_nodes, max_distance)** Use SciPy Dijkstra for exact distances up to a cutoff; fallback to approximation on failure.

`_compute_shortest_path_distances_networkx`**(adj, num_nodes, max_distance)** Use NetworkX all-pairs for small graphs, sampling-based approximation for large graphs.

`_compute_approximate_distances_networkx`**(G, num_nodes, max_distance)** Approximate large-graph distances by sampling single-source shortest paths and filling gaps.

`_compute_walk_probabilities`**(distance_matrix, alpha)** Build a normalized transition matrix from distances, mix with uniform jumps via damping factor `c`.

`sample_random_walks_for_graph_classification`**(edge_index, batch, num_nodes)** For each graph in the batch, sample multiple random walks per start node, returning walk sequences and their graph IDs.

`forward`**(x, edge_index, batch)** Execute sampling, feature encoding, GRU encoding; return $\{\text{path\_encodings, walk\_batc}$
for downstream aggregation.

2.2 Class `RW_GNN`

`__init__`**(input_dim, max_step, hidden_graphs, size_hidden_graphs, hidden_dim, penultimate_dim, normaliz**
Initialize the GNN: wrap the random-walk encoder and build a two-layer downstream network with ReLU, BatchNorm, and Dropout.

`init_weights`**()** Uniformly initialize hidden-graph parameters (reserved for future extensions).

`forward`**(data)** Dispatch to node or graph classification based on `data.batch`.

`forward_node_classification`**(data)** Node-level forward: compute per-node walk encodings, apply fc1 $\to$ bn $\to$ dropout $\to$ fc2 $\to$ dropout2, output log-softmax.

`_node_specific_walk_encoding`**(x, edge_index, num_nodes)** For each node, sample multiple biased or simple walks, encode via GRU, and average to obtain a node embedding.

`_generate_biased_walk`**(start_node, walk_probs, walk_length)** Generate one random walk using the provided transition probabilities.

`_generate_single_walk`**(start_node, adj_list, walk_length)** Generate one pure random walk by uniform neighbor sampling.

`_simple_neighborhood_encoding`**(x, edge_index, num_nodes)** Fallback one/two-hop feature aggregation when walk-based encoding fails.

`forward_graph_classification`**(data)** Graph-level forward: pool path encodings by graph ID, apply downstream network, output log-softmax.

2.3 Auxiliary Functions

`plot_training_curves`**(train_losses, val_losses, train_accs, val_accs, save_path)** Plot and save side-by-side training vs. validation loss and accuracy curves.

`train`**(model, loader, optimizer, device)** Perform one epoch of training, return average loss and accuracy.

`test`**(model, loader, device)** Perform one epoch of evaluation (no gradient), return average loss and accuracy.

2.4 Class `LocalCoraDataset`

`raw_file_names` / `processed_file_names` Specify raw and processed file lists (PyG interface).

`download()` No-op to disable auto-download.

`process()`
1. Read `.x, .y, .allx, .ally, .graph, .test.index`.
2. Merge and reorder features/labels, build an undirected `edge_index`.
3. Split into 140/500/1000 train/val/test masks, replenishing randomly if needed; enforce no overlap or empty sets.
4. Save the resulting `Data` object to disk.

`_read_pickle_file` / `_read_test_index` Load raw data from Pickle files or test index text.