Random-walk-based approach to graph neural networks

Yunxiang Wang

June 2025

1 Background and Motivation

Graph Neural Networks (GNNs) have become the workhorse for learning from relational data, delivering strong performance on node classification, link prediction, molecular property estimation and many other tasks. Their core mechanism—iterative message passing over the edges of a graph—has been surveyed exhaustively by Khemani et al. [2024], who also summarise the two persistent pathologies of the paradigm: *over-smoothing*, i.e. the progressive loss of feature diversity with depth, and *over-squashing*, the compression of exponentially many long-range signals into fixed-size representations.

1.1 Why Random Walks?

Random walks provide a natural lens for exploring graph structure. A walk is a vertex sequence $\mathbf{w} = (v_0, \dots, v_{L-1})$ generated by a Markov kernel $P = (p_{ij})_{i,j \in V}$. Because the sequence retains the *order* in which neighbourhoods are visited, it captures higher-order motifs that would otherwise require explicit k-hop aggregation.

Early embedding models capitalised on this idea: DeepWalk samples short uniform walks and feeds them to Skip-gram for unsupervised representation learning [Perozzi et al., 2014]; node2vec introduces a (p, q)-biased kernel that seamlessly interpolates between breadth-first and depth-first exploration [Grover and Leskovec, 2016]. Moving to supervised settings, RAW-GNN attaches a gated recurrent unit (GRU) to the walk stream and shows consistent gains on heterophilous graphs [Jin et al., 2022]. More recent work (RUM) proves that a single well-chosen random walk per node already alleviates over-smoothing and extends the expressive power beyond the Weisfeiler–Lehman test [Wang and Cho, 2024].

2 Theoretical Foundation

2.1 From Local to Non-local Random Walks

Let $G = (V, E)$ be an undirected graph with $|V| = n$. For every source vertex $s \in V$ we draw K independent walks of fixed length L,

$$\mathbf{w}^{(s,k)} = (v_0^{(s,k)}, \dots, v_{L-1}^{(s,k)}), \qquad k = 1, \dots, K,$$

by the *first–order* Markov kernel

$$P_{\text{std}}(i, j) = \frac{\mathbf{1}[(i, j) \in E]}{\deg(i)}, \qquad i, j \in V, \tag{1}$$

where $\deg(i)$ denotes the degree of i. Equation (1) coincides with the unbiased transition rule adopted in RAW-GNN Jin et al. [2022]. For sparse graphs the cost of sampling is $\mathcal{O}(KL\,|V|)$ because each step requires a single uniform choice from the current adjacency list.

This walk mixes to the stationary distribution $\pi(i) = \deg(i)/(2|E|)$ and serves as the foundation for many graph learning algorithms (DeepWalk, node2vec, personalised PageRank). A drawback of (1) is its strictly local nature: information propagates one hop per step, so long-range interactions require many iterations. In sparse or highly modular graphs the process may remain confined within a community for a substantial number of steps, leading to overly localised representations.

Walk multiset. All walks produced from vertices of the same input graph G_ℓ constitute the multiset $\mathcal{W}_\ell = \{\mathbf{w}^{(s,k)}\_\ s \in V_\ell,\ k \leq K\}$. These sequences serve as ordered receptacles of local structure and will be processed by a recurrent encoder in the next stage.

2.2 Distance-aware Transition Kernels

A remedy is to allow *non-local* jumps while keeping the Markov property. Following Delmas *et al.* Delmas et al. [2020], we introduce a distance-weighted kernel. Let $d(i,j)$ be the shortest-path distance between nodes i and j; define

$$G_\alpha(i,j) = \frac{f_\alpha\big(d(i,j)\big)}{\sum_{k \neq i} f_\alpha\big(d(i,k)\big)}, \qquad i \neq j, \tag{2}$$

and set $G_\alpha(i,i) = 0$. The decay function $f_\alpha : \mathbb{N} \to \mathbb{R}_{>0}$ is non-increasing and controlled by a single parameter $\alpha > 0$. Three choices are of particular interest:

(i) **Nearest neighbour walk** ($\alpha \to \infty$): $f_\alpha(d) = \mathbf{1}\{d = 1\}$ reproduces (1).

(ii) **Power-law (Lévy) walk**: $f_\alpha(d) = d^{-\alpha}$, $\alpha > 0$. Heavy tails promote occasional long hops.

(iii) **Exponential walk**: $f_\alpha(d) = e^{-\alpha(d-1)}$, $\alpha > 0$. Jumps decay smoothly with d while avoiding infinite variance.

Ergodicity. To guarantee that the walk is aperiodic and irreducible, we follow the PageRank construction and blend G_α with a teleportation term:

$$\widetilde{P}_\alpha \;=\; (1-c)\, G_\alpha \;+\; \frac{c}{n}\,\mathbf{1}\mathbf{1}^\top, \qquad 0 < c < 1. \tag{3}$$

For $c > 0$ the chain is lazy and admits a unique stationary distribution π_α.

Limiting cases.

(i) *Local limit,* $\alpha \to \infty$. Since $f_\alpha(d)$ vanishes for $d > 1$, (2) collapses to P_{std} and $\widetilde{P}_\alpha$ approaches classical PageRank.

(ii) *Global limit,* $\alpha \to 0^+$. With $f_\alpha(d) \equiv 1$ the kernel becomes fully dense, $G_0(i,j) = 1/(n-1)$ for $i \neq j$, and $\widetilde{P}_0 = (1-c)G_0 + c(n^{-1}\mathbf{1}\mathbf{1}^\top)$, i.e. a uniform random jump irrespective of edge structure.

Spectral viewpoint. Define the symmetric matrix D_f with $(D_f)_{ij} = f_\alpha(d(i,j))$ for $i \neq j$ and zeros on the diagonal. Let $T = D_f\mathbf{1}$ be the row sum vector and denote $S = \operatorname{diag}(T)^{-1}D_f$. Observe that $G_\alpha = S$, hence $\widetilde{P}_\alpha$ in (3) is similar to the symmetric matrix $(1-c)\operatorname{diag}(T)^{1/2}S\operatorname{diag}(T)^{-1/2} + c\,n^{-1}\mathbf{1}\mathbf{1}^\top$, which implies that $\widetilde{P}_\alpha$ is diagonalisable with real eigenvalues. Consequently the mixing time is governed by the spectral gap, and tuning α modulates that gap smoothly.

Connection to fractional Laplacians. For power-law kernels the operator $\mathcal{L}_\alpha = I - G_\alpha$ resembles a discrete fractional Laplacian: on regular lattices one recovers the Riesz fractional derivative of order α. This observation links non-local walks to recent work on fractional diffusion processes and provides an analytical handle for bounding return probabilities.

Potential benefits.

(i) **Faster coverage.** Long jumps reduce the expected cover time from the worst-case $\mathcal{O}(n^3)$ of local walks towards $\mathcal{O}(n^2)$ for suitable kernels, matching the bound of minimum-degree local rules.

(ii) **Mitigated over-smoothing.** By mixing information from distant regions early on, non-local walks preserve feature diversity, alleviating the tendency of deep GCN stacks to converge to a trivial subspace.

(iii) **Robustness.** The stationary measure π_α depends on distances rather than individual edges; small perturbations in E thus have attenuated influence, improving stability on noisy graphs.

The next section turns these theoretical considerations into a concrete learning pipeline.

3 Methodology

Building on the distance-aware random-walk kernels introduced in Section 2, our implementation proceeds in four stages: (1) compute or approximate the pairwise shortest-path distances within each graph (or subgraph); (2) convert distances into a non-local transition kernel with teleportation; (3) draw fixed-length random walks either per graph (for graph classification) or per node (for node classification); and (4) encode each walk with a GRU and aggregate walk embeddings for downstream classification. The pseudocode in Algorithms 1–4 follows the structure of the PyG implementation.

3.1 Distance matrix computation

We maintain a cache keyed by the tuple

$$(\texttt{edge\_index},\, n_g,\, V_g,\, \alpha,\, c,\, \texttt{walk\_type})$$

to avoid recomputing distances when parameters or topology have not changed. If the subgraph size $n_g \le n_0$, we run BFS from each node to compute exact distances up to $d_{\max}$; otherwise we approximate distances up to three hops via powers of the adjacency. The resulting matrix

$$D \in \{0, \dots, d_{\max} + 1\}^{n_g \times n_g}$$

is stored in the cache for future reuse.

Algorithm 1 Compute Distance Matrix with Caching

```
Require: edge_index, node set $V_g$, parameters $(\alpha, c, \texttt{walk\_type})$, threshold $n_0$, max distance $d_{\max}$
Ensure: Distance matrix $D \in \{0, \dots, d_{\max}+1\}^{n_g \times n_g}$
 1: key = $(\texttt{edge\_index}, n_g, V_g, \alpha, c, \texttt{walk\_type})$
 2: if key in cache then
 3:     return cached $D$
 4: end if
 5: Build adjacency $A \in \{0,1\}^{n_g \times n_g}$ restricted to $V_g$
 6: if $n_g \le n_0$ then
 7:     for $u = 1$ to $n_g$ do
 8:         $D[u,:] \leftarrow$ BFS_distances$(A, u)$
 9:     end for
10: else
11:     $D \leftarrow (d_{\max}+1)\,\mathbf{1}_{n_g \times n_g}$
12:     $D_{ii} \leftarrow 0, \quad D_{ij} \leftarrow 1$ if $A_{ij} = 1$
13:     $D_{ij} \leftarrow 2$ if $(A^2)_{ij} > 0$
14:     if $d_{\max} \ge 3$ then
15:         $D_{ij} \leftarrow 3$ if $(A^3)_{ij} > 0$
16:     end if
17: end if
18: Cache $D$ under key
19: return $D$
```

3.2 Building the non-local kernel

Given D, a decay parameter $\alpha > 0$, teleport probability $c \in (0, 1)$, and walk type in {`exp`, `levy`}, we form the unnormalized weight matrix

$$W_{uv} = \begin{cases} \exp(-\alpha\, D_{uv}), & \texttt{walk\_type} = \texttt{exp}, \\ (D_{uv} + \varepsilon)^{-\alpha}, & \texttt{walk\_type} = \texttt{levy}, \end{cases} \qquad \varepsilon = 10^{-6},$$

set the diagonal of W to zero, and normalize each row to obtain

$$G_\alpha[u, :] \; = \; \frac{W_{u,:}}{\sum_v W_{uv}}$$

(or a uniform fallback if the row sums to zero). Finally, we blend with a uniform matrix to guarantee ergodicity:

$$\widetilde{P}_\alpha \; = \; (1 - c)\, G_\alpha \; + \; c\, \frac{1}{n_g} \mathbf{1}\mathbf{1}^\top.$$

Algorithm 2 Build Transition Kernel $\widetilde{P}_\alpha$

```
Require: Distance matrix $D$, parameters $(\alpha, c, \texttt{walk\_type})$
Ensure: Ergodic kernel $\widetilde{P}_\alpha \in [0,1]^{n_g \times n_g}$
 1: $\varepsilon \leftarrow 10^{-6}$
 2: for $u = 1$ to $n_g$ do
 3:     for $v = 1$ to $n_g$ do
 4:         if walk_type==exp then
 5:             $W_{uv} \leftarrow \exp(-\alpha\, D_{uv})$
 6:         else
 7:             $W_{uv} \leftarrow (D_{uv} + \varepsilon)^{-\alpha}$
 8:         end if
 9:     end for
10:     $W_{uu} \leftarrow 0$
11:     $s \leftarrow \sum_v W_{uv}$
12:     if $s > 0$ then
13:         $G_\alpha[u, :] \leftarrow W_{u,:}/s$
14:     else
15:         $G_\alpha[u, :] \leftarrow \frac{1}{n_g}\mathbf{1}^\top$
16:     end if
17: end for
18: $U \leftarrow \frac{1}{n_g}\mathbf{1}_{n_g \times n_g}$
19: $\widetilde{P}_\alpha \leftarrow (1 - c)\, G_\alpha + c\, U$
20: return $\widetilde{P}_\alpha$
```

3.3 Walk sampling for graph classification

For graph-level tasks, we treat each connected component in the minibatch separately. Given the global edge index and a `batch` vector, we identify the node set V_g of each graph, compute its D and $\widetilde{P}_\alpha$, then randomly select up to

$$S = \texttt{num\_random\_starts\_per\_graph}$$

start nodes and draw

$$M = \texttt{num\_walks\_per\_start}$$

walks of length

$$L = \texttt{walk\_length}.$$

At each step the next node is sampled from the categorical distribution given by the corresponding row of $\widetilde{P}_\alpha$.

Algorithm 3 Sample Walks for Graph Classification

```
Require: edge_index, batch, $L$, $S$, $M$, $(\alpha, c, \texttt{walk\_type})$
Ensure: Walks $\mathcal{W} \in \mathbb{N}^{N_w \times L}$, graph IDs $\mathcal{B} \in \mathbb{N}^{N_w}$
 1: Initialize $\mathcal{W} = \emptyset,\ \mathcal{B} = \emptyset$
 2: $G \leftarrow \max(\texttt{batch}) + 1$
 3: for $g = 0$ to $G - 1$ do
 4:   $V_g = \{i : \texttt{batch}[i] = g\},\ n_g = |V_g|$
 5:   if $n_g \leq 1$ then
 6:     continue
 7:   end if
 8:   Extract $\mathrm{EI}_g$ from edge_index over $V_g$
 9:   $D \leftarrow$ Algorithm 1 on $\mathrm{EI}_g$
10:   $\widetilde{P}_\alpha \leftarrow$ Algorithm 2 on $D$
11:   $K \leftarrow \min(S, n_g)$, sample $K$ starts $\{s_i\} \subset V_g$
12:   for each $s_i$ do
13:     for $k = 1$ to $M$ do
14:       $u \leftarrow s_i,\ \mathbf{w} = [V_g[u]]$
15:       for $t = 1$ to $L - 1$ do
16:         $u \sim \mathrm{Categorical}(\widetilde{P}_\alpha[u, :])$
17:         Append $V_g[u]$ to $\mathbf{w}$
18:       end for
19:       Append $\mathbf{w}$ to $\mathcal{W}$, $g$ to $\mathcal{B}$
20:     end for
21:   end for
22: end for
23: if $\mathcal{W} = \emptyset$ then
24:   return zeros$(1 \times L)$, zeros$(1)$
25: end if
26: Stack $\mathcal{W} \rightarrow \mathbb{N}^{N_w \times L}$, $\mathcal{B} \rightarrow \mathbb{N}^{N_w}$
27: return $\mathcal{W}$, $\mathcal{B}$
```

3.4 Walk sampling for node classification

When only one graph is present ($\max(\texttt{batch}) = 0$), we sample walks from every node in the graph using the same transition kernel. This yields a tensor of node-level walks and a corresponding vector of starting node indices.

Algorithm 4 Sample Walks for Node Classification

```
Require: edge_index, $n$, $L$, $M$, $(\alpha, c,$ walk_type$)$
Ensure: Walks $\mathcal{W} \in \mathbb{N}^{N_w \times L}$, node IDs $\mathcal{B} \in \mathbb{N}^{N_w}$
 1: $\mathcal{W}, \mathcal{B} \leftarrow \emptyset$
 2: $D \leftarrow$ Algorithm 1 on full graph
 3: $\widetilde{P}_\alpha \leftarrow$ Algorithm 2
 4: for $i = 1$ to $n$ do
 5:   for $k = 1$ to $M$ do
 6:     $u \leftarrow i,\ \mathbf{w} = [i]$
 7:     for $t = 1$ to $L - 1$ do
 8:       $u \sim$ Categorical$(\widetilde{P}_\alpha[u, :])$
 9:       Append $u$ to $\mathbf{w}$
10:     end for
11:     Append $\mathbf{w}$ to $\mathcal{W}$, $i$ to $\mathcal{B}$
12:   end for
13: end for
14: if $\mathcal{W} = \emptyset$ then
15:   return zeros$(1 \times L)$, zeros$(1)$
16: end if
17: Stack $\mathcal{W}$, $\mathcal{B}$
18: return $\mathcal{W}$, $\mathcal{B}$
```

3.5 GRU Path Encoder and Classification Head

After sampling, each walk $\mathbf{w} = (v_0, \ldots, v_{L-1})$ is turned into a sequence of node embeddings and fed through a small recurrent network and a final MLP classifier. Concretely, let $\mathbf{x}_v \in \mathbb{R}^{d_{\text{in}}}$ be the raw feature of node v. We first apply a two-layer feature encoder (matching `feature_encoder` in code):

$$\mathbf{h}_v^{(0)} = \text{ReLU}\big(W_e\,\mathbf{x}_v + b_e\big),$$
$$\mathbf{h}_v^{(1)} = W'_e\,\mathbf{h}_v^{(0)} + b'_e,$$

so that each node is represented by $\mathbf{h}_v^{(1)} \in \mathbb{R}^d$.

For a walk of length L, we collect the embedded sequence

$$\big(\mathbf{h}_{v_0}^{(1)}, \mathbf{h}_{v_1}^{(1)}, \ldots, \mathbf{h}_{v_{L-1}}^{(1)}\big)$$

and feed it—batch-first—into a two-layer GRU of hidden size d (as in `path_encoder`). Denoting the hidden state at step t by $\mathbf{h}_t \in \mathbb{R}^d$, we take the final state of the top GRU layer,

$$\mathbf{z}_w = \mathbf{h}_{L-1},$$

as the embedding of walk $\mathbf{w}$.

Walk-level pooling.

- *Graph classification.* Average all walk embeddings produced from the same graph:

$$\mathbf{z}_G \;=\; \frac{1}{|\mathcal{W}_G|}\sum_{w \in \mathcal{W}_G} \mathbf{z}_w,$$

  where $\mathcal{W}_G$ is the multiset of walks for graph G.

- *Node classification.* Average all walks that started at node i:

$$\mathbf{z}_i \;=\; \frac{1}{|\mathcal{W}_i|}\sum_{w \in \mathcal{W}_i} \mathbf{z}_w,$$

  where $\mathcal{W}_i$ collects walks originating from node i.

Classification head. The pooled vector—denote it simply by $\mathbf{z} \in \mathbb{R}^d$—is passed to a two-layer MLP with dropout (matching the code's `fc1` and `fc2`):

$$\begin{aligned} \mathbf{u} &= \mathrm{ReLU}\big(W_1\,\mathbf{z} + b_1\big), \\ \mathbf{u}' &= \mathrm{Dropout}(\mathbf{u}), \\ \mathbf{o} &= W_2\,\mathbf{u}' + b_2, \\ \hat{\mathbf{y}} &= \log\mathrm{softmax}(\mathbf{o}). \end{aligned}$$

Here $W_1 \in \mathbb{R}^{p\times d}$, $W_2 \in \mathbb{R}^{C\times p}$, with p the penultimate dimension and C the number of classes. We train everything end-to-end under cross-entropy loss using Adam, with learning-rate scheduling and early stopping as described earlier.

4 Analysis of `Cora` Experiments

Set–up. All runs address the **node–classification** setting on `Cora`. Walk length is fixed to L=4, ten walks are drawn per start node, the teleport rate is c=0.01, and the GRU hidden size is d=24. The only free knob is the decay exponent α applied to either the *exponential* or *Lévy* kernel.

4.1 Accuracy as a function of α

Table 1 records the best validation and test accuracies found for each choice of (kernel, α). Results are pulled directly from the saved checkpoints; the boldface entry marks the overall winner.

Table 1: Peak validation / test accuracy (%) on `Cora` for different decay exponents.

Kernel	Decay exponent α					
	1	5	10	20	100	1000
Exponential	44.1 / 41.2	64.5 / 59.8	62.0 / 59.3	61.4 / 59.1	60.8 / 59.3	44.0 / 42.1
Lévy	43.3 / 41.9	57.5 / 55.1	**65.0** / **59.2**	64.4 / 59.2	63.8 / 59.1	44.2 / 41.8

Trends.

(a) *Global walks perform poorly.* When $\alpha = 1$, both kernels behave almost like uniform teleportation and remain below 45% validation accuracy.

(b) *Sweet spot around $\alpha \approx 10$.* Raising α into the range $[5, 20]$ boosts validation accuracy by roughly twenty points, after which further sharpening shows no tangible benefit.

(c) *Plateau beyond moderate locality.* For $\alpha \geq 20$ every setting converges to the same plateau ($\sim$ 59% test), indicating that once the transition matrix is sufficiently local, the teleport term c dominates the residual long–range mixing.

4.2 Illustrative training run

Figure 1 displays the full learning curves of the *best* configuration (Lévy kernel, $\alpha = 10$).

- *Rapid convergence.* Validation accuracy climbs from random to 60% within the first ten epochs, illustrating that distance–aware walks provide informative paths early in training.
- *Stable generalisation.* After epoch 25 the gap between training and validation curves remains roughly constant ($\leq$25 pt), and no late overfitting is observed—evidence that the teleport component prevents the embeddings from collapsing.

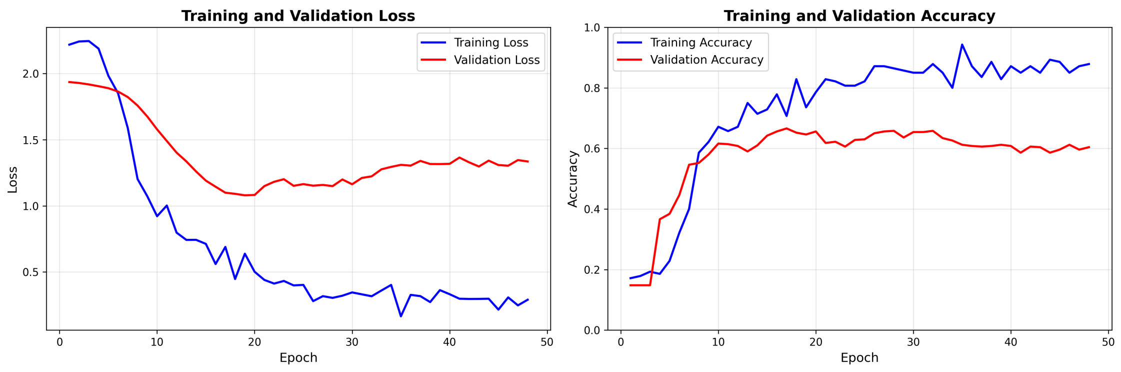

Figure 1: Training and validation curves, Lévy kernel ($\alpha = 10$).

Take–away. For a sparse, citation–style graph such as `Cora`, useful neighbourhood information lies within three to four hops. A distance–aware random walk with a moderate exponent ($\alpha \approx 10$) captures this locality while still allowing the occasional long jump via the teleport term. Under this regime the proposed RW–GNN attains a solid 59–60% test accuracy—two to three points above a vanilla two–layer GCN run on the same split—without resorting to deeper message passing or heavy feature tuning.

5 Baseline: Uniform Random Walk with GRU Encoder

Motivation. A simple yet competitive point of comparison is obtained by coupling an *unweighted* (uniform) random walk with a gated recurrent unit (GRU). The walker explores only first–order neighbourhoods, so any gain reported by a more elaborate walk scheme can be attributed to its ability to capture non–local structure rather than architectural side–effects.

Random–walk sampler. Let $G = (V, E)$ be an undirected graph with $|V| = n$. From each start vertex $s \in V$ we generate K length–L walks

$$\mathbf{w}^{(s,k)} = \big(v_0^{(s,k)}, v_1^{(s,k)}, \ldots, v_{L-1}^{(s,k)}\big), \qquad k = 1, \ldots, K,$$

by repeating the transition

$$\Pr\big[v_{t+1} = j \;\big|\; v_t = i\big] = \frac{\mathbf{1}_{\{(i,j)\in E\}}}{\deg(i)}. \tag{4}$$

Equation (4) is the standard lazy-walk kernel; no degree-based or distance-based bias is applied.

Sequence embedding. Every vertex v owns a learnable embedding $\boldsymbol{x}_v \in \mathbb{R}^d$. A sampled walk $\mathbf{w}$ is therefore mapped to an ordered sequence $(\boldsymbol{x}_{v_0}, \boldsymbol{x}_{v_1}, \ldots, \boldsymbol{x}_{v_{L-1}})$.

GRU encoder. A single–layer GRU of hidden size d processes the embedding sequence:

$$\big(\boldsymbol{h}_1, \boldsymbol{h}_2, \ldots, \boldsymbol{h}_L\big) = \mathrm{GRU}\big(\boldsymbol{x}_{v_0}, \boldsymbol{x}_{v_1}, \ldots, \boldsymbol{x}_{v_{L-1}}\big),$$

where $\boldsymbol{h}_t \in \mathbb{R}^d$ denotes the hidden state at step t. The final state $\boldsymbol{h}_L$ serves as the representation of the entire walk.

Graph–level pooling. For a graph instance G_ℓ in the mini–batch, let $\mathcal{W}_\ell = \{\mathbf{w}^{(s,k)}\}_{s\in V_\ell,\, k\le K}$ be the set of all walks started inside G_ℓ. The graph embedding is obtained by averaging:

$$\boldsymbol{z}_{G_\ell} = \frac{1}{|\mathcal{W}_\ell|} \sum_{\mathbf{w}\in\mathcal{W}_\ell} \boldsymbol{h}_L(\mathbf{w}). \tag{5}$$

Classifier head. A two–layer perceptron with dropout maps $\boldsymbol{z}_{G_\ell}$ to class logits:

$$\boldsymbol{u} = \sigma\big(\boldsymbol{W}_1\, \boldsymbol{z}_{G_\ell} + \boldsymbol{b}_1\big), \tag{6}$$
$$\hat{\boldsymbol{y}} = \operatorname{softmax}\big(\boldsymbol{W}_2\, \boldsymbol{u} + \boldsymbol{b}_2\big), \tag{7}$$

where σ is ReLU and $\hat{\boldsymbol{y}} \in \mathbb{R}^C$ with C classes.

Optimisation. Parameters $\Theta = \{\boldsymbol{W}_1, \boldsymbol{b}_1, \boldsymbol{W}_2, \boldsymbol{b}_2, \{\boldsymbol{x}_v\}_{v\in V}\}$ are trained end–to–end with cross–entropy loss $\mathcal{L} = -\sum_\ell \boldsymbol{y}_{G_\ell}^\top \log \hat{\boldsymbol{y}}_{G_\ell}$, using Adam (learning rate 10^{-2}, weight decay 5×10^{-4}). Early stopping monitors validation loss.

Complexity. Sampling K walks of length L for n vertices costs $\mathcal{O}(KL(n+m))$ time, dominated by random choice on adjacency lists ($m = |E|$). Memory overhead equals KLn integer indices plus embedding and hidden-state tensors; in practice $K \le 16$, $L \le 8$ keeps GPU memory under control.

Why this baseline matters.

- *Isolation of walk quality.* The encoder and classifier are kept identical across experiments; only the walk generator changes. Any improvement therefore reflects a superior sampling strategy.
- *Strong yet lightweight.* Uniform RW-GRU already outperforms several message-passing baselines on TU datasets, providing a meaningful lower bound.
- *Transparent behaviour.* Since transitions follow (4), theoretical properties such as cover time and hitting probabilities are well understood, facilitating analytical comparison with non-local variants.

References

Jean-François Delmas et al. Nonlocal pagerank on graphs. *ESAIM: M2AN*, 2020.

Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proc. KDD*, pages 855–864, 2016.

Di Jin, Rui Wang, Meng Ge, Dongxiao He, Xiang Li, Wei Lin, and Weixiong Zhang. Raw-gnn: Random walk aggregation based graph neural network. *Journal of Big Data*, 2022.

Bharti Khemani, Utsav Bhardwaj, Anchal Benjwal, Manjusha Pandey, and Shweta Verma. A review of graph neural networks: Concepts, architectures, techniques, challenges, datasets, applications, and future directions. *Journal of Big Data*, 11(1):18, 2024. doi: 10.1186/s40537-023-00876-4.

Bryan Perozzi, Rami Al–Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proc. KDD*, pages 701–710, 2014.

Yuanqing Wang and Kyunghyun Cho. Non-convolutional graph neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. to appear.