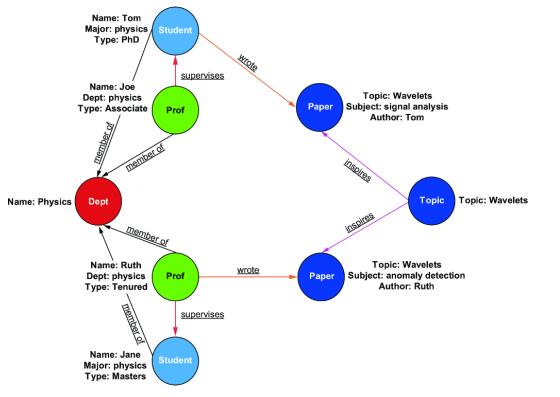
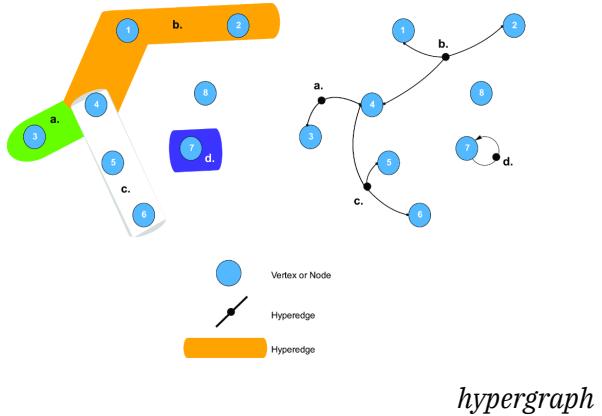


Graph Machine Learning



Introduction

Machine Learning as Optimization: A Simple Summary

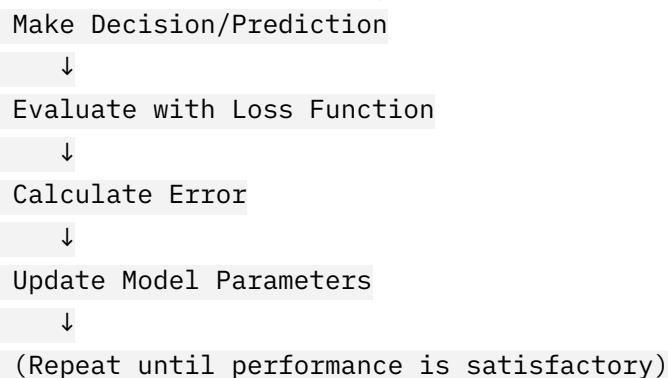
Machine learning is fundamentally an **optimization problem** where we search for the best possible model to perform a specific task.

The Core Process

Goal: Find a mathematical model that achieves optimal performance on a given task

Key Components:

1. **Performance Metric** (Loss Function/Cost Function)
2. Quantifies how well the model is performing
3. Lower loss = better performance
4. **Data-Driven Learning**
5. Algorithm receives data (often large amounts)
6. Uses this data to make iterative improvements
7. **The Learning Cycle** (Training):



The Essence: At each iteration, the model makes predictions, measures how wrong it is, and adjusts its internal parameters to do better next time. Through repeated cycles, the model progressively improves its performance.

This iterative process of **learning from mistakes** is what we call **training** - it's how machines get "smarter" at their tasks over time.

🎓 Supervised Learning in Graph ML

$\langle x, y \rangle$

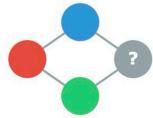
x = Input (graph, node features, network structure)
 y = Known output (labels we want to predict)



Goal: Learn from labeled examples, then predict on new data

Node Classification

Discrete Labels

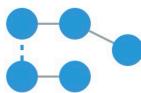


Predict: Node Category

- Social Network: User interests
- Biology: Protein function
- Citation: Paper topic

Link Prediction

Edge Exists?



Predict: Will edge exist?

- Social: Friend suggestions
- E-commerce: Product recommendations
- Drug: Protein interactions

Graph Classification

Discrete Labels



Predict: Graph Property

- Chemistry: Molecule toxicity
- Social: Community type
- Neuroscience: Brain state

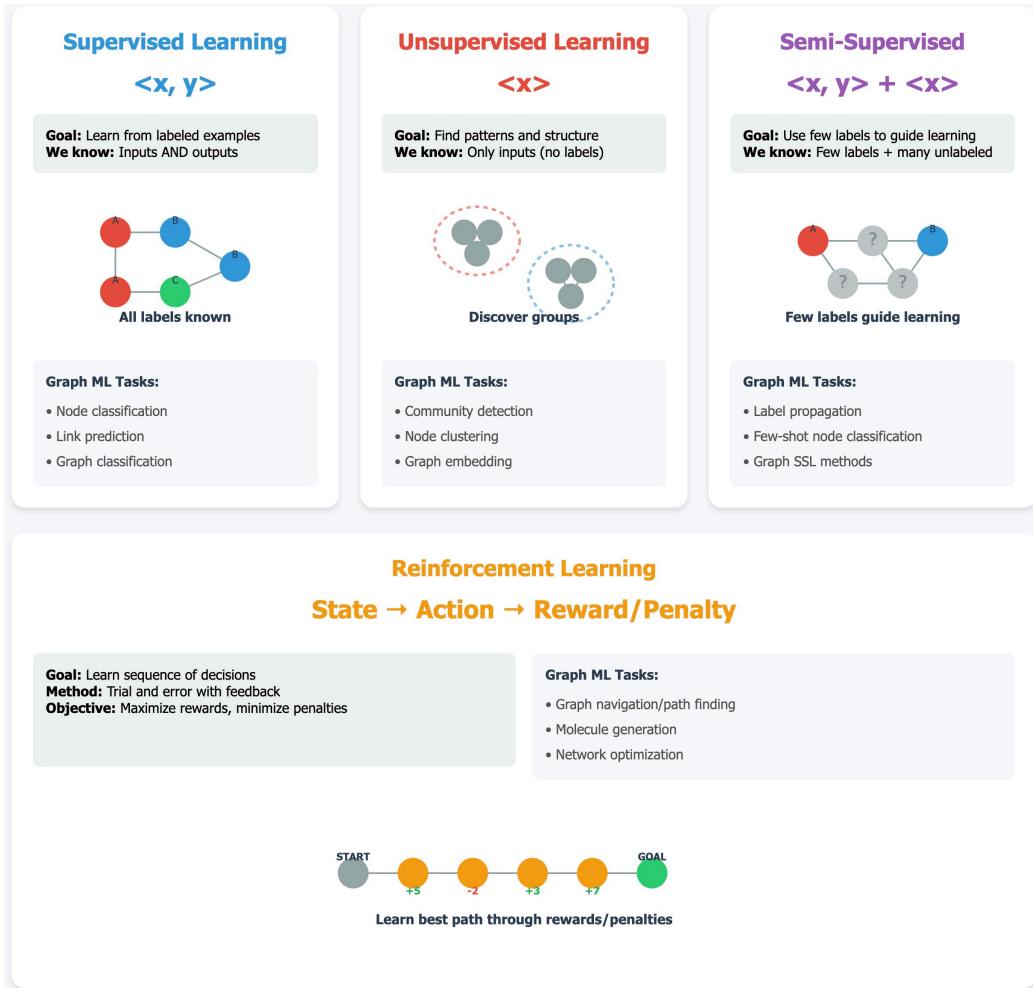
Key Differences from Traditional ML

Traditional ML:

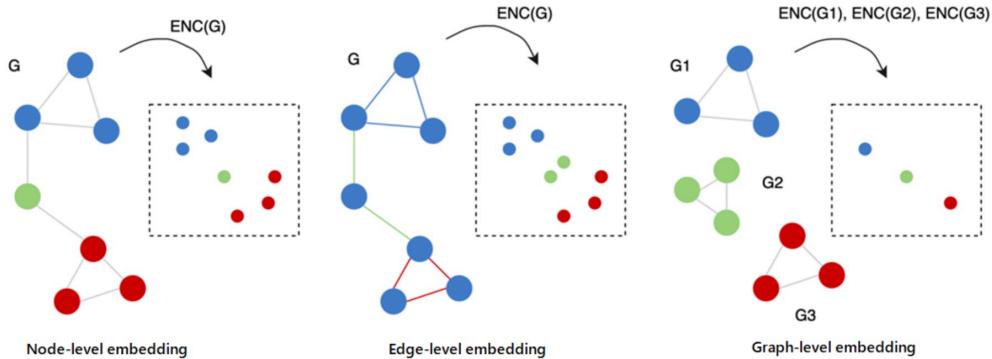
- Independent samples
- Fixed features
- No relationships

Graph ML:

- Connected samples
- Relational features
- Network structure matters



Graph Learning Tasks



Graph Machine Learning levels

Node Representation Learning

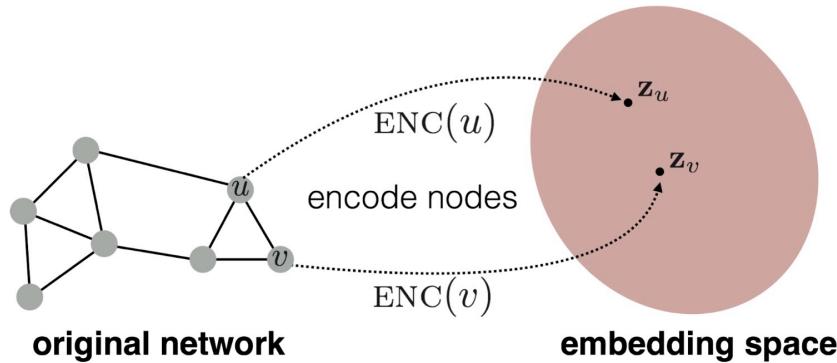


Figure 3.1: Illustration of the node embedding problem. Our goal is to learn an encoder (ENC), which maps nodes to a low-dimensional embedding space. These embeddings are optimized so that distances in the embedding space reflect the relative positions of the nodes in the original graph.

$$\text{ENC} : \mathcal{V} \rightarrow \mathbb{R}^d,$$

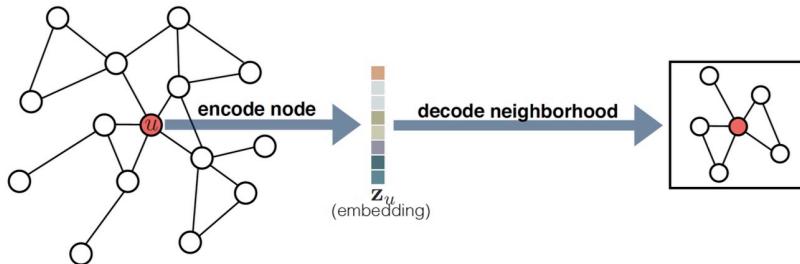


Figure 3.2: Overview of the encoder-decoder approach. The encoder maps the node u to a low-dimensional embedding \mathbf{z}_u . The decoder then uses \mathbf{z}_u to reconstruct u 's local neighborhood information.

$$\text{DEC} : \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}^+.$$

$$\text{DEC}(\text{ENC}(u), \text{ENC}(v)) = \text{DEC}(\mathbf{z}_u, \mathbf{z}_v) \approx \mathbf{S}[u, v].$$

To achieve the reconstruction this objective, the standard practice is to minimize an empirical reconstruction loss over a set of training node pairs D:

$$\mathcal{L} = \sum_{(u,v) \in \mathcal{D}} \ell(\text{DEC}(\mathbf{z}_u, \mathbf{z}_v), \mathbf{S}[u, v]),$$

METHOD	DECODER	SIMILARITY MEASURE	LOSS FUNCTION
Lap. Eigenmaps	$\ \mathbf{z}_u - \mathbf{z}_v\ _2^2$	general	$\text{DEC}(\mathbf{z}_u, \mathbf{z}_v) \cdot \mathbf{S}[u, v]$
Graph Fact.	$\mathbf{z}_u^\top \mathbf{z}_v$	$\mathbf{A}[u, v]$	$\ \text{DEC}(\mathbf{z}_u, \mathbf{z}_v) - \mathbf{S}[u, v]\ _2^2$
GraRep	$\mathbf{z}_u^\top \mathbf{z}_v$	$\mathbf{A}[u, v], \dots, \mathbf{A}^k[u, v]$	$\ \text{DEC}(\mathbf{z}_u, \mathbf{z}_v) - \mathbf{S}[u, v]\ _2^2$
HOPE	$\mathbf{z}_u^\top \mathbf{z}_v$	general	$\ \text{DEC}(\mathbf{z}_u, \mathbf{z}_v) - \mathbf{S}[u, v]\ _2^2$
DeepWalk	$\frac{e^{\mathbf{z}_u^\top \mathbf{z}_v}}{\sum_{k \in \mathcal{V}} e^{\mathbf{z}_u^\top \mathbf{z}_k}}$	$p_{\mathcal{G}}(v u)$	$-\mathbf{S}[u, v] \log(\text{DEC}(\mathbf{z}_u, \mathbf{z}_v))$
node2vec	$\frac{e^{\mathbf{z}_u^\top \mathbf{z}_v}}{\sum_{k \in \mathcal{V}} e^{\mathbf{z}_u^\top \mathbf{z}_k}}$	$p_{\mathcal{G}}(v u)$ (biased)	$-\mathbf{S}[u, v] \log(\text{DEC}(\mathbf{z}_u, \mathbf{z}_v))$

Shallow embedding approaches face three major drawbacks:

1. No Parameter Sharing:

Each node has its own independent embedding vector, leading to poor statistical efficiency and high computational cost. The number of parameters grows linearly with the number of nodes $O(|V|)$, making the method unscalable for large graphs.

2. No Use of Node Features:

These methods ignore rich node attribute information that could improve the quality and generalizability of learned representations.

3. Transductive Limitation:

Shallow embeddings can only represent nodes seen during training. They cannot generalize to unseen nodes without retraining, preventing their use in inductive settings.

DeepWalk Architecture

DeepWalk is a two-step approach for learning node representations:

1. **Random Walk Generation:** Generate random walks from the graph (similar to sequences of words)
2. **SkipGram Training:** Use SkipGram to learn node representations

Word2Vec and SkipGram

SkipGram Model: Predicts context words given a target word.

Mathematical Formulation: For a vocabulary of N words: w_1, w_2, \dots, w_N

The goal is to maximize the probability of context words given the target word:

$$\text{P}(w_{i+j}|w_i) = \frac{\exp(v_{w_{i+j}}^T v_{w_i})}{\sum_{w=1}^N \exp(v_w^T v_{w_i})}$$

where v_w is the vector representation of word w .

Objective Function:

$$\frac{1}{N} \sum_{n=1}^N \sum_{-c \leq j \leq c, j \neq 0} \log p(W_{n+j} | W_n)$$

where c is the context window size.

Node2Vec

Extension of DeepWalk with biased random walks:

- **p parameter:** Controls return to previous node
- **q parameter:** Controls exploration vs exploitation

Random Walk Strategy:

- **BFS-like:** Explores local neighborhood (homophily)

- **DFS-like:** Explores distant nodes (structural roles)

node2vec: Scalable Feature Learning for Networks

A Comprehensive Tutorial with Examples

Table of Contents

1. [Introduction and Motivation](#)
 2. [Background: From Word2Vec to node2vec](#)
 3. [Random Walks on Graphs](#)
 4. [The Key Innovation: Biased Random Walks](#)
 5. [The node2vec Algorithm](#)
 6. [Mathematical Formulation](#)
 7. [Complete Worked Example](#)
 8. [Implementation Guide](#)
 9. [Comparison with Other Methods](#)
 10. [Applications and Use Cases](#)
-

Introduction and Motivation

What is node2vec?

node2vec is an algorithmic framework for learning continuous feature representations (embeddings) for nodes in networks. It was introduced by Grover & Leskovec in 2016 and extends the earlier DeepWalk algorithm.

The Big Idea

Analogy: If words that appear in similar contexts should have similar embeddings (Word2Vec), then **nodes that appear in similar "graph contexts" should have similar embeddings.**

Key Innovation

node2vec introduces **flexible, biased random walks** that can smoothly interpolate between:

- **Breadth-First Search (BFS):** Local neighborhood exploration
- **Depth-First Search (DFS):** Exploring farther nodes

This flexibility allows node2vec to capture both:

1. **Homophily:** Nodes in the same community (nearby nodes)
 2. **Structural equivalence:** Nodes with similar roles (e.g., both are hubs)
-

Background: From Word2Vec to node2vec

Word2Vec: The Inspiration

Skip-gram model learns word embeddings by:

1. Taking a sentence: "the quick brown fox jumps"
2. Creating pairs: (quick, the), (quick, brown), (brown, quick), (brown, fox), ...
3. Learning embeddings where: $P(\text{context_word} | \text{center_word})$ is high

Mathematical objective: ```` maximize: $\sum \log P(w_{\text{context}} | w_{\text{center}})$

where: $P(w_c | w_{\text{center}}) = \exp(z_c^T \cdot z_{\text{center}}) / \sum_w \exp(z_w^T \cdot z_{\text{center}})$ ````

Adapting to Graphs

The mapping:

- Sentences \rightarrow **Random walks** on the graph

- Words → **Nodes**
- Context window → **Nodes appearing together in walks**

Example: Walk: $1 \rightarrow 3 \rightarrow 5 \rightarrow 7 \rightarrow 9$

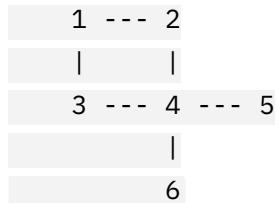
- Creates pairs: $(1,3), (1,5), (3,1), (3,5), (3,7), (5,3), (5,7), (5,9), \dots$
-

Random Walks on Graphs

What is a Random Walk?

A **random walk** is a path through the graph where at each step, we randomly choose one of the current node's neighbors.

Example Graph:



Random walk starting from node 1:

Step 0: At node 1
 Step 1: Randomly pick neighbor → go to 2 or 3
 Step 2: From there, randomly pick a neighbor
 ...
 Continue for k steps

Example walk: $1 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 4 \rightarrow 6$

Uniform Random Walk (DeepWalk)

In **DeepWalk**, the probability of moving to a neighbor is uniform:

$$P(\text{next} = v \mid \text{current} = u) = \begin{cases} 1/\text{degree}(u) & \text{if } (u,v) \text{ is an edge} \\ 0 & \text{otherwise} \end{cases}$$

Problem: Uniform walks may not capture all graph properties equally well.

Biased Random Walks

The Problem with Uniform Walks

Consider these two graph properties:

1. **Homophily** (Community): Nodes 1, 2, 3 form a tight cluster
2. **Structural Role**: Node 1 and Node 10 are both "hub" nodes

Uniform random walks might not capture both equally well!

The node2vec Solution: Bias the Walk!

node2vec introduces **two parameters** to control the walk:

1. **p** (return parameter): Controls likelihood of revisiting a node
2. **q** (in-out parameter): Controls exploration vs. exploitation

The Biased Random Walk Mechanism

Setup: Currently at node **v**, came from node **t**, deciding where to go next.

Neighbors of v:

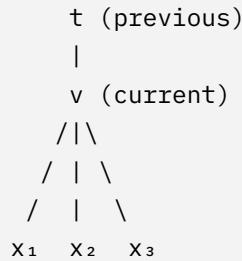
- **t**: The previous node (where we came from)
- **x₁**: Neighbor at distance 1 from t (same distance as v)
- **x₂**: Neighbor at distance 2 from t (farther than v)

Transition probabilities (unnormalized):

```
a(t, x) = 1/p    if x = t      (returning to previous node)
           1        if d(t,x) = 1  (staying at same distance)
           1/q      if d(t,x) = 2  (moving away)
```

where $d(t, x)$ is the shortest path distance between t and x .

Visual Example



where:

- x_1 is also neighbor of t (distance 1 from t)
- x_2 is not neighbor of t (distance 2 from t)
- $x_3 = t$ (distance 0, returning)

Transition probabilities:

$P(x_3 | v, \text{ came from } t) \propto 1/p \quad (\text{return to } t)$

$P(x_1 | v, \text{ came from } t) \propto 1 \quad (\text{explore at same level})$

$P(x_2 | v, \text{ came from } t) \propto 1/q \quad (\text{go farther})$

Understanding Parameters p and q

Parameter p (Return Parameter)

Controls: Likelihood of immediately returning to the previous node

$p < 1$: High probability of returning \rightarrow more "backtracking"

$p = 1$: No bias

$p > 1$: Low probability of returning \rightarrow more "forward moving"

Use case:

- **Low p :** Keeps walk local, good for community detection
- **High p :** Encourages exploration, prevents getting stuck

Parameter q (In-Out Parameter)

Controls: Balance between BFS and DFS

```
q < 1: DFS-like behavior (explore far)
    → Good for capturing structural equivalence
    → Nodes with similar "roles" get similar embeddings

q = 1: No bias (similar to unbiased walk)

q > 1: BFS-like behavior (stay local)
    → Good for capturing homophily
    → Nodes in same community get similar embeddings
```

Example Settings

Setting 1: Community Detection (capture homophily)

p = 1

q = 2

→ BFS-like: Explores local neighborhood thoroughly

Setting 2: Structural Equivalence (capture roles)

p = 1

q = 0.5

→ DFS-like: Explores farther, finds nodes with similar structural patterns

Setting 3: Balanced

p = 1

q = 1

→ Unbiased (like DeepWalk)

The node2vec Algorithm

High-Level Algorithm

Algorithm: node2vec

Input:

- Graph $G = (V, E)$
- Dimensions d
- Walk length l
- Walks per node r
- Context window size k
- Parameters p, q

Output:

- Embeddings $Z \in \mathbb{R}^{|V| \times d}$

Steps:

1. Precompute transition probabilities for biased walks
2. Generate walks:
 - For each node $u \in V$:
 - For $i = 1$ to r :
Generate biased random walk of length l starting from u
3. Learn embeddings using Skip-gram:
Optimize embeddings to predict context nodes from center nodes
4. Return learned embeddings Z

Step-by-Step Detailed Algorithm

Step 1: Precompute Transition Probabilities

For computational efficiency, we precompute all transition probabilities:

```
def precompute_transition_probs(G, p, q):  
    """  
    For each edge (t, v) in the graph, compute  
    probabilities for the next step from v.  
    """  
    transition_probs = {}  
  
    for edge in G.edges():  
        t, v = edge # came from t, currently at v  
        probs = {}
```

```

        for neighbor in G.neighbors(v):
            if neighbor == t:
                # Returning to previous node
                probs[neighbor] = 1/p
            elif G.has_edge(neighbor, t):
                # Distance 1 from t
                probs[neighbor] = 1
            else:
                # Distance 2 from t
                probs[neighbor] = 1/q

        # Normalize
        norm = sum(probs.values())
        for neighbor in probs:
            probs[neighbor] /= norm

    transition_probs[(t, v)] = probs

return transition_probs

```

Step 2: Generate Biased Random Walks

```

def generate_walk(G, start_node, walk_length, transition_probs):
    """
    Generate a single biased random walk.
    """
    walk = [start_node]

    for i in range(walk_length - 1):
        current = walk[-1]
        neighbors = list(G.neighbors(current))

        if len(neighbors) == 0:
            break

        if len(walk) == 1:
            # First step: uniform random choice
            next_node = random.choice(neighbors)
        else:
            # Use biased probabilities
            previous = walk[-2]

```

```

        probs = transition_probs[(previous, current)]
        next_node = random.choices(neighbors,
                                    weights=[probs[n] for n in
                                              neighbors])[0]

        walk.append(next_node)

    return walk

```

Step 3: Train Skip-gram Model

```

def train_skipgram(walks, d, window_size, num_epochs):
    """
    Learn embeddings using skip-gram objective.
    Uses negative sampling for efficiency.
    """

    # Initialize embeddings
    vocab_size = max(max(walk) for walk in walks) + 1
    Z = np.random.randn(vocab_size, d) * 0.01

    for epoch in range(num_epochs):
        for walk in walks:
            # Generate training pairs
            for i, center_node in enumerate(walk):
                # Context window
                start = max(0, i - window_size)
                end = min(len(walk), i + window_size + 1)

                for j in range(start, end):
                    if i != j:
                        context_node = walk[j]

                        # Positive pair: (center_node,
                        context_node)
                        # Negative samples: (center_node,
                        random_node)

                        # Update embeddings using gradient descent
                        # [Implementation details omitted for
                        brevity]

```

```
return Z
```

Mathematical Formulation

Objective Function

node2vec maximizes the log-probability of observing network neighborhoods:

```
maximize:  $\sum_{u \in V} \log P(N_S(u) | z_u)$ 
```

Where:

- $N_S(u)$ is the network neighborhood of node u generated by strategy S
- z_u is the embedding of node u

Skip-gram Formulation

Using the skip-gram assumption (independence):

```
 $P(N_S(u) | z_u) = \prod_{v \in N_S(u)} P(v | z_u)$ 
```

Prediction Probability

```
 $P(v | z_u) = \exp(z_u^T \cdot z_v) / \sum_{w \in V} \exp(z_u^T \cdot z_w)$ 
```

This is expensive to compute ($O(|V|)$ for each pair)!

Negative Sampling (Optimization Trick)

To make training efficient, node2vec uses **negative sampling**:

```
 $\log P(v | z_u) \approx \log \sigma(z_u^T \cdot z_v) + \sum_{i=1}^k E_{v_i \sim P_n} [\log \sigma(-z_u^T \cdot z_{v_i})]$ 
```

Where:

- σ is the sigmoid function
- P_n is the negative sampling distribution (usually uniform)
- k is the number of negative samples (typically 5-20)

Final Loss Function:

$$L = \sum_{(u,v) \in D} [-\log \sigma(z_u^T \cdot z_v) - \gamma \sum_{v_n \sim P_n} \log \sigma(-z_u^T \cdot z_{v_n})]$$

Where D is the set of (center, context) pairs from random walks.

Complete Worked Example

Let's work through node2vec on a small graph!

The Graph

```
1 --- 2 --- 3
|       |       |
4 --- 5 --- 6
```

Adjacency:

- 1:[2, 4]
- 2:[1, 3, 5]
- 3:[2, 6]
- 4:[1, 5]
- 5:[2, 4, 6]
- 6:[3, 5]

Parameters

```
p = 1      (no return bias)
q = 0.5    (DFS-like: prefer exploring far)
l = 5      (walk length)
r = 3      (walks per node)
d = 3      (embedding dimension)
k = 2      (context window)
```

Step 1: Compute Transition Probabilities

Example: Currently at node 5, came from node 2

Neighbors of 5: {2, 4, 6}

Distances from node 2 (where we came from):

- Node 2: distance 0 (it's where we came from) → probability $\propto 1/p = 1$
- Node 4: distance 2 (not neighbor of 2) → probability $\propto 1/q = 2$
- Node 6: distance 2 (not neighbor of 2) → probability $\propto 1/q = 2$

Unnormalized weights: {2: 1, 4: 2, 6: 2}

Normalized probabilities:

$$P(2 \mid \text{at 5, from 2}) = 1/5 = 0.20$$

$$P(4 \mid \text{at 5, from 2}) = 2/5 = 0.40$$

$$P(6 \mid \text{at 5, from 2}) = 2/5 = 0.40$$

Interpretation: With $q = 0.5$ (DFS-like), we're more likely to explore farther (nodes 4, 6) than return to 2.

Step 2: Generate Random Walks

Walk 1 from node 1: ```` Start: 1 Step 1: 1 → 2 (random choice from neighbors)
Step 2: 2 → 5 (using transition probs based on coming from 1) Step 3: 5 → 6
(using transition probs based on coming from 2) Step 4: 6 → 3 (using transition
probs based on coming from 5)

Walk: [1, 2, 5, 6, 3] ````

Walk 2 from node 1:

Walk: [1, 4, 5, 2, 3]

Walk 3 from node 1:

Walk: [1, 2, 3, 6, 5]

Similar walks from all other nodes (2, 3, 4, 5, 6)...Total walks: 6 nodes \times 3 walks/node = **18 walks****Step 3: Extract Training Pairs**

From walk [1, 2, 5, 6, 3] with window size k=2:

Center = 1 (position 0):

- Context: {2, 5}
- Pairs: (1,2), (1,5)

Center = 2 (position 1):

- Context: {1, 5, 6}
- Pairs: (2,1), (2,5), (2,6)

Center = 5 (position 2):

- Context: {1, 2, 6, 3}
- Pairs: (5,1), (5,2), (5,6), (5,3)

Center = 6 (position 3):

- Context: {2, 5, 3}
- Pairs: (6,2), (6,5), (6,3)

Center = 3 (position 4):

- Context: {5, 6}
- Pairs: (3,5), (3,6)

Step 4: Train Skip-gram

Initialize embeddings randomly:

```
z_1 = [0.12, -0.34, 0.56]  
z_2 = [-0.23, 0.45, 0.12]  
z_3 = [0.34, 0.22, -0.11]  
z_4 = [-0.45, 0.13, 0.28]  
z_5 = [0.21, -0.12, 0.33]  
z_6 = [0.33, 0.24, -0.15]
```

Training pair example: (5, 6) (node 5 predicts node 6)

Forward pass: score = $z_5^T \cdot z_6 = 0.21 \times 0.33 + (-0.12) \times 0.24 + 0.33 \times (-0.15)$
 $= 0.0693 + (-0.0288) + (-0.0495) = -0.009$

$P(6 | 5) = \exp(-0.009) / \sum_w \exp(z_5^T \cdot z_w)$

Negative sampling: Sample negative nodes (say nodes 1, 4)

Loss:

```
loss = -log σ(z_5^T · z_6) - log σ(-z_5^T · z_1) - log σ(-z_5^T · z_4)
```

Gradient descent: Update embeddings to minimize loss

After training (many iterations over all pairs):

```
Learned embeddings (example):  
z_1 = [0.67, 0.45, -0.12]  
z_2 = [0.71, 0.52, -0.08] ← similar to z_1 (neighbors)  
z_3 = [0.69, 0.48, -0.15] ← similar to z_2, z_6  
z_4 = [0.64, 0.43, -0.18] ← similar to z_1 (neighbors)  
z_5 = [0.70, 0.49, -0.10] ← central node, similar to all  
z_6 = [0.68, 0.47, -0.14] ← similar to z_3, z_5
```

Observation: Nodes in the same cluster have similar embeddings!

Implementation Guide

Pseudocode

```
import numpy as np
import random
from collections import defaultdict

class Node2Vec:
    def __init__(self, G, dimensions=128, walk_length=80,
                 num_walks=10, p=1, q=1, window_size=10):
        self.G = G
        self.d = dimensions
        self.walk_length = walk_length
        self.num_walks = num_walks
        self.p = p
        self.q = q
        self.window_size = window_size

    def fit(self):
        # Step 1: Precompute probabilities
        self.transition_probs = self.precompute_transition_probs()

        # Step 2: Generate walks
        walks = self.generate_walks()

        # Step 3: Train skip-gram
        self.embeddings = self.train_skipgram(walks)

        return self.embeddings

    def precompute_transition_probs(self):
        # Implementation as shown earlier
        pass

    def generate_walks(self):
        walks = []
        nodes = list(self.G.nodes())

        for _ in range(self.num_walks):
            random.shuffle(nodes)
            for node in nodes:
```

```

        walk = self.generate_walk(node)
        walks.append(walk)

    return walks

def generate_walk(self, start_node):
    # Implementation as shown earlier
    pass

def train_skipgram(self, walks):
    # Can use existing libraries like gensim.Word2Vec
    from gensim.models import Word2Vec

    # Convert walks to strings for gensim
    walks_str = [[str(node) for node in walk] for walk in
walks]

    model = Word2Vec(walks_str,
                      vector_size=self.d,
                      window=self.window_size,
                      min_count=0,
                      sg=1, # skip-gram
                      workers=4,
                      epochs=5)

    # Extract embeddings
    embeddings = {}
    for node in self.G.nodes():
        embeddings[node] = model.wv[str(node)]

    return embeddings

```

Using Existing Implementation

The easiest way is to use existing libraries:

```

# Install: pip install node2vec

from node2vec import Node2Vec
import networkx as nx

```

```

# Create graph
G = nx.karate_club_graph()

# Generate walks and learn embeddings
node2vec = Node2Vec(G,
                     dimensions=64,           # embedding dimension
                     walk_length=30,          # length of each walk
                     num_walks=200,           # walks per node
                     p=1,                     # return parameter
                     q=1,                     # in-out parameter
                     workers=4)

# Train
model = node2vec.fit(window=10,           # context window
                      min_count=1,
                      batch_words=4)

# Get embeddings
embeddings = model.wv

```

Comparison with Other Methods

node2vec vs DeepWalk

ASPECT	DEEPWALK	NODE2VEC
Random walks	Uniform	Biased (p, q parameters)
Flexibility	Less flexible	More flexible
Captures	Mixed properties	Can target specific properties
Speed	Faster	Slightly slower (precomputation)

node2vec vs Graph Factorization

ASPECT	GRAPH FACTORIZATION	NODE2VEC
Approach	Matrix factorization	Random walks

Similarity	Deterministic	Stochastic
Scalability	$O(V ^2)$	$O(E)$
Flexibility	Limited	High (via p, q)

When to Use node2vec?

Use node2vec when:

- You need flexible embeddings
- Graph is large (millions of nodes)
- You want to tune homophily vs structural equivalence
- You need state-of-the-art performance

Consider alternatives when:

- Graph is very small (< 100 nodes) → matrix methods may be simpler
- You need interpretable embeddings → spectral methods
- You have rich node features → GNNs

Applications and Use Cases

1. Link Prediction

Task: Predict missing edges

Setup:

- Train node2vec on partial graph
- For candidate edge (u,v), compute similarity: $z_u^T \cdot z_v$
- High similarity → likely edge

Hyperparameters: p=1, q=1 (balanced)

2. Node Classification

Task: Classify nodes into categories

Setup:

- Learn embeddings with node2vec
- Use embeddings as features in classifier (SVM, Logistic Regression)

Hyperparameters: $p=1, q=0.5$ (DFS-like, captures roles)

3. Community Detection

Task: Find tightly connected groups

Setup:

- Learn embeddings with BFS-like walks
- Cluster embeddings (K-means)

Hyperparameters: $p=1, q=2$ (BFS-like, captures homophily)

4. Recommendation Systems

Task: Recommend items to users

Setup:

- Build user-item graph
- Learn embeddings
- Recommend items with high similarity to user's embedding

Real-World Examples

Bioinformatics: Protein-protein interaction networks

- $p=1, q=0.5$ (find proteins with similar functions)

Social Networks: Friend recommendation

- $p=1, q=2$ (find people in same community)

Knowledge Graphs: Entity linking

- p=0.5, q=2 (explore local neighborhood)
-

Advantages and Limitations

Advantages

1. **Scalable:** Linear in number of edges $O(|E|)$
2. **Flexible:** Parameters p, q control embedding properties
3. **General:** Works on any graph (directed, weighted, etc.)
4. **Proven:** Strong empirical performance
5. **Unsupervised:** No labels needed

Limitations

1. **Transductive:** Can't embed new nodes without retraining
2. **No features:** Ignores node attributes
3. **Hyperparameters:** Need to tune p, q
4. **Memory:** Stores many random walks
5. **Static:** Doesn't handle dynamic graphs well

Solutions to Limitations

For new nodes:

- Retrain (expensive)
- Or use inductive methods (GraphSAGE)

For node features:

- Concatenate node2vec embeddings with features
- Or use GNNs

For dynamic graphs:

- Incremental training

- Or use temporal GNN methods
-

Tips and Best Practices

Hyperparameter Tuning

Walk length (l):

- Short walks (10-40): Faster, local structure
- Long walks (80-100): Slower, global structure
- **Typical:** 80

Walks per node (r):

- Few walks (10): Faster, less stable
- Many walks (100+): Slower, more stable
- **Typical:** 10-20

Dimensions (d):

- Low (32-64): Faster, less expressive
- High (128-256): Slower, more expressive
- **Typical:** 128

Parameters p and q:

- Start with $p=1, q=1$ (unbiased)
- For communities: increase q ($q=2$)
- For roles: decrease q ($q=0.5$)
- Grid search over $\{0.5, 1, 2, 4\}$

Computational Tips

1. **Precompute transition probabilities** once
2. **Parallelize** walk generation
3. **Use existing Word2Vec implementations** (gensim)

4. **Sample fewer walks** for initial experiments
 5. **Cache** random walks for multiple experiments
-

Summary

The node2vec Algorithm in Three Steps

1. **Generate biased random walks** using parameters p and q
2. **Extract (center, context) pairs** from walks
3. **Learn embeddings** using skip-gram with negative sampling

Key Takeaways

- node2vec = **Flexible random walks** + **Skip-gram**
- Parameter **p** controls return probability
- Parameter **q** controls BFS vs DFS behavior
- Captures both **homophily** and **structural equivalence**
- **Scalable** to millions of nodes
- **State-of-the-art** performance on many tasks

When You've Mastered node2vec

You understand:

- ✓ How random walks capture graph structure
 - ✓ How p and q parameters control exploration
 - ✓ How skip-gram learns from walks
 - ✓ How to tune hyperparameters for your task
 - ✓ When to use node2vec vs alternatives
-

References

Original Paper:

- Grover, A., & Leskovec, J. (2016). node2vec: Scalable Feature Learning for Networks. KDD.

Related Methods:

- Perozzi, B., et al. (2014). DeepWalk: Online Learning of Social Representations. KDD.
- Mikolov, T., et al. (2013). Efficient Estimation of Word Representations. ICLR.

Implementations:

- Official: <https://github.com/aditya-grover/node2vec>
- Python library: <https://github.com/eliorc/node2vec>

GRL Book: Hamilton, W. L. (2020). Graph Representation Learning. Morgan & Claypool.

Exercises

Exercise 1: Parameter Effects

Try different (p, q) combinations on the Karate Club graph. How do embeddings change?

Exercise 2: Implementation

Implement the biased random walk from scratch.

Exercise 3: Application

Use node2vec for link prediction on a dataset of your choice.

Exercise 4: Comparison

Compare node2vec with DeepWalk and graph factorization on the same task.

[illustration here](#)

The node embedding approaches we discussed used a shallow embedding approach to generate representations of nodes, where we simply optimized a unique embedding vector for each node.

The key idea is that we want to generate representations of nodes that actually depend on the structure of the graph, as well as any feature information we might have.

$$f(\mathbf{P}\mathbf{A}\mathbf{P}^\top) = f(\mathbf{A})$$

$$f(\mathbf{P}\mathbf{A}\mathbf{P}^\top) = \mathbf{P}f(\mathbf{A})$$

The shallow encoders are an example of permutation equivariant functions.)

Graph Neural Networks (GNNs)

Regardless of the motivation, the defining feature of a GNN is that it uses a form of neural message passing in which vector messages are exchanged between nodes and updated using neural networks [Gilmer et al., 2017].

We will describe how we can take an input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, along with a set of node features $\mathbf{X} \in \mathbb{R}^{d \times |\mathcal{V}|}$, and use this information to generate node embeddings $\mathbf{z}_u, \forall u \in \mathcal{V}$.

Message Passing Framework

Core Idea: Update node representations by aggregating information from neighbors.

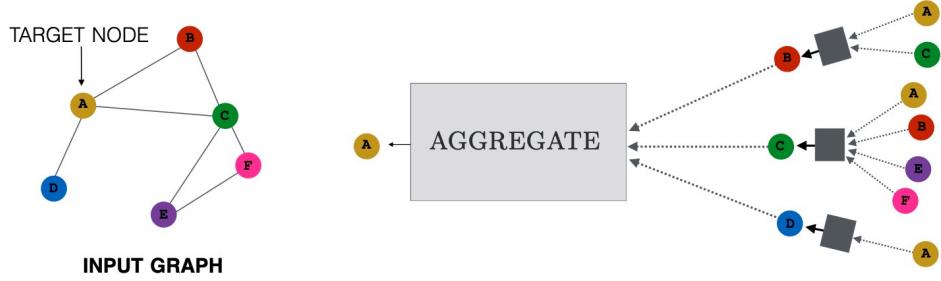


Figure 5.1: Overview of how a single node aggregates messages from its local neighborhood. The model aggregates messages from A's local graph neighbors (i.e., B, C, and D), and in turn, the messages coming from these neighbors are based on information aggregated from their respective neighborhoods, and so on. This visualization shows a two-layer version of a message-passing model. Notice that the computation graph of the GNN forms a tree structure by unfolding the neighborhood around the target node.

$$\begin{aligned}
 \mathbf{h}_u^{(k+1)} &= \text{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \text{AGGREGATE}^{(k)}\left(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\}\right)\right) \\
 &= \text{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \mathbf{m}_{\mathcal{N}(u)}^{(k)}\right),
 \end{aligned}$$

where **UPDATE** and **AGGREGATE** are arbitrary differentiable functions (i.e., neural networks), and $\mathbf{m}_{\mathcal{N}(u)}$ is the “message” that is aggregated from u 's graph neighborhood $\mathcal{N}(u)$. We use superscripts to distinguish the embeddings and functions at different iterations of message passing.Footnote: For example, $\mathbf{h}_u^{(k)}$ and $\text{UPDATE}^{(k)}$ refer to the embedding and update function at the k -th iteration.

Mathematical Formulation:

$$h_v^{(l+1)} = \sigma(W^{(l)} \cdot \text{AGGREGATE}^{(l)}(\{h_u^{(l)} : u \in \mathcal{N}(v)\}))$$

where:

- $h_v^{(l)}$ is the representation of node v at layer l
- $\mathcal{N}(v)$ is the neighborhood of node v
- **AGGREGATE** is an aggregation function
- σ is an activation function

Step 1: $\mathbf{h}^{(1)} = \mathbf{A}[1]$
 Step 2: $\mathbf{h}^{(2)} = \mathbf{A}[1] \oplus \mathbf{A}[2]$
 Step 3: $\mathbf{h}^{(3)} = \mathbf{A}[1] \oplus \mathbf{A}[2] \oplus \mathbf{A}[3]$
 Step 4: $\mathbf{h}^{(4)} = \mathbf{A}[1] \oplus \mathbf{A}[2] \oplus \mathbf{A}[3] \oplus \mathbf{A}[4]$
 \vdots
 Step n : $\mathbf{h}^{(n)} = \mathbf{A}[1] \oplus \mathbf{A}[2] \oplus \cdots \oplus \mathbf{A}[n]$
 Final: $\mathbf{z}_G = \text{MLP}(\mathbf{h}^{(|\mathcal{V}|)})$
 $= \text{MLP}(\mathbf{A}[1] \oplus \mathbf{A}[2] \oplus \cdots \oplus \mathbf{A}[|\mathcal{V}|])$

Node Features (not really but similar)

$$\mathbf{A}[1], \mathbf{A}[2], \dots, \mathbf{A}[|\mathcal{V}|]$$

Aggregate them

Apply an operation like summation:

$$\mathbf{h}_G = \mathbf{A}[1] \oplus \mathbf{A}[2] \oplus \cdots \oplus \mathbf{A}[|\mathcal{V}|]$$

$$If \oplus = \sum, then: \mathbf{h}_G = \sum_{i=1}^{|\mathcal{V}|} \mathbf{A}[I]$$

Feed the aggregated representation into an MLP

$$\mathbf{z}_G = \text{MLP}(\mathbf{h}_G)$$

Example

Suppose your graph has 3 nodes with features:

$$\mathbf{A}[1] = [1, 0], \quad \mathbf{A}[2] = [0, 1], \quad \mathbf{A}[3] = [1, 1]$$

Aggregation by summation:

$$\mathbf{h}_G = [1, 0] + [0, 1] + [1, 1] = [2, 2]$$

$$\mathbf{z}_G = \text{MLP}([2, 2])$$

$$\mathbf{z}_G = \sigma(\mathbf{W}_2 \text{ReLU}(\mathbf{W}_1[2, 2]^\top + \mathbf{b}_1) + \mathbf{b}_2)$$

If each node feature vector $\mathbf{A}[i]$ has dimension d , and your graph has $|\mathcal{V}|$ nodes, then concatenating them gives:

$$\mathbf{h}_G = \mathbf{A}[1] \oplus \mathbf{A}[2] \oplus \cdots \oplus \mathbf{A}[|\mathcal{V}|] \Rightarrow \mathbf{h}_G \in \mathbb{R}^{d \times |\mathcal{V}|}$$

–or as a single long vector of size $d|\mathcal{V}|$.

So if each $\mathbf{A}[i] = [a_{1i}, a_{2i}, a_{3i}]$ (3 features per node), and you have 4 nodes, then:

$$\mathbf{h}_G = [a_{11}, a_{21}, a_{31}, a_{12}, a_{22}, a_{32}, a_{13}, a_{23}, a_{33}, a_{14}, a_{24}, a_{34}]$$

That is permutation variant!

```
A = torch.tensor([[1,2,3],
                 [4,5,6],
                 [7,8,9],
                 [10,11,12]])
h_G = torch.cat([A[0], A[1], A[2], A[3]]) # [1,2,3,4,5,6,7,8,9,10,11,12]
z_G = mlp(h_G)
```

Shallow encoders are equivariant functions:

Adjacency (path 1–2–3):

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \quad \mathbf{X} = \begin{bmatrix} 2 \\ -1 \\ 3 \end{bmatrix}$$

Shallow encoder:

$$f(\mathbf{A}, \mathbf{X}) = \mathbf{AX}.$$

Permutation (relabel to order 2,3,1):

$$\mathbf{P} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

Original output

$$\mathbf{AX} = \begin{bmatrix} -1 \\ 5 \\ -1 \end{bmatrix}$$

Permuted inputs

$$\mathbf{X}' = \mathbf{PX} = \begin{bmatrix} -1 \\ 3 \\ 2 \end{bmatrix}, \quad \mathbf{A}' = \mathbf{PAP}^\top = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

Output after permutation

$$\mathbf{A}'\mathbf{X}' = \begin{bmatrix} 5 \\ -1 \\ -1 \end{bmatrix}$$

Compare with permuted original output

$$\mathbf{P}(\mathbf{AX}) = \mathbf{P} \begin{bmatrix} -1 \\ 5 \\ -1 \end{bmatrix}$$

$$\begin{bmatrix} 5 \\ -1 \\ -1 \end{bmatrix}$$

They match, so

$f(\mathbf{PAP}^\top, \mathbf{PX}) = \mathbf{P} f(\mathbf{A}, \mathbf{X})$, which demonstrates permutation equivariance for the shallow encoder $f(\mathbf{A}, \mathbf{X}) = \mathbf{AX}$.

Message Passing

Neural message passing in GNNs: nodes exchange vector “messages” with neighbors and update their states using neural networks.

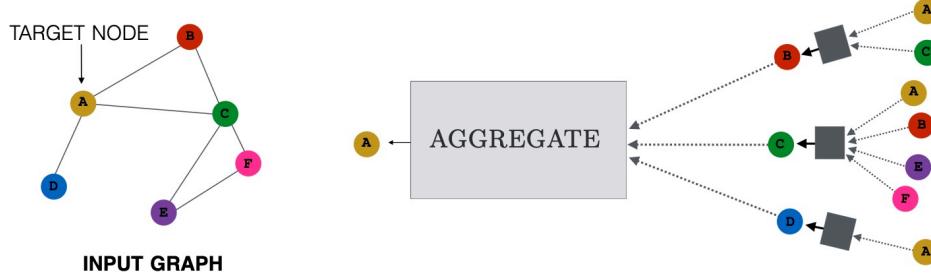


Figure 5.1: Overview of how a single node aggregates messages from its local neighborhood. The model aggregates messages from A’s local graph neighbors (i.e., B, C, and D), and in turn, the messages coming from these neighbors are based on information aggregated from their respective neighborhoods, and so on. This visualization shows a two-layer version of a message-passing model. Notice that the computation graph of the GNN forms a tree structure by unfolding the neighborhood around the target node.

At each iteration k :

$$\mathbf{h}_u^{(k+1)} = \text{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \text{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k)} \mid v \in \mathcal{N}(u)\})\right)$$

- $\mathbf{h}_u^{(k)}$ – current embedding (feature) of node u
- $\mathcal{N}(u)$ – neighbors of node u
- AGGREGATE – combines messages from neighbors (permutation-invariant)
- UPDATE – updates node u's representation using its previous embedding and the aggregated message

At the end (after K iterations):

$$\mathbf{z}_u = \mathbf{h}_u^{(K)} \forall u$$

Example

1 – 2 – 3

Adjacency list:

- 1's neighbors: {2}
- 2's neighbors: {1, 3}
- 3's neighbors: {2}

Step 1: Initialize features

At iteration k = 0:

$$\mathbf{h}_1^{(0)} = 1, \quad \mathbf{h}_2^{(0)} = 2, \quad \mathbf{h}_3^{(0)} = 3$$

Step 2: Define AGGREGATE and UPDATE

For simplicity: $\text{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k)}\}) = \sum_{v \in \mathcal{N}(u)} \mathbf{h}_v^{(k)}$

$$\text{UPDATE}^{(k)}(x, y) = x + y$$

Step 3: Compute iteration k=1

Node 1:

$$\text{AGGREGATE} = \mathbf{h}_2^{(0)} = 2$$

$$\text{UPDATE} : \mathbf{h}_1^{(1)} = 1 + 2 = 3$$

Node 2:

$$\text{AGGREGATE} = \mathbf{h}_1^{(0)} + \mathbf{h}_3^{(0)} = 1 + 3 = 4$$

$$\text{UPDATE} : \mathbf{h}_2^{(1)} = 2 + 4 = 6$$

Node 3:

$$\text{AGGREGATE} = \mathbf{h}_2^{(0)} = 2$$

$$\text{UPDATE} : \mathbf{h}_3^{(1)} = 3 + 2 = 5$$

Step 4: Output after one iteration

$$\mathbf{h}_1^{(1)} = 3, \quad \mathbf{h}_2^{(1)} = 6, \quad \mathbf{h}_3^{(1)} = 5$$

Step 5: Next iteration k=2

Node 1:

$$\text{AGGREGATE} = \mathbf{h}_2^{(1)} = 6$$

$$\text{UPDATE} : \mathbf{h}_1^{(2)} = 3 + 6 = 9$$

Node 2:

$$\text{AGGREGATE} = \mathbf{h}_1^{(1)} + \mathbf{h}_3^{(1)} = 3 + 5 = 8$$

$$\text{UPDATE} : \mathbf{h}_2^{(2)} = 6 + 8 = 14$$

Node 3:

$$\text{AGGREGATE} = \mathbf{h}_2^{(1)} = 6$$

$$\text{UPDATE} : \mathbf{h}_3^{(2)} = 5 + 6 = 11$$

—

Step 6: Final embeddings (after K=2)

$$\mathbf{z}_1 = 9, \quad \mathbf{z}_2 = 14, \quad \mathbf{z}_3 = 11$$

Term	Meaning	In our example
$\mathbf{h}_u^{(0)}$	Node's initial features	1, 2, 3
AGGREGATE	Gathers neighbor info	Sum of neighbor values
UPDATE	Combines own info + neighbor info	Simple addition
Permutation equivariance	Node reordering yields permuted results	True here (aggregation uses a set)

- After the first iteration ($k = 1$), each node's embedding contains information from its 1-hop neighborhood – i.e., its immediate neighbors.
- After the second iteration ($k = 2$), each node's embedding includes information from its 2-hop neighborhood (neighbors of neighbors).
- In general, after k iterations, every node embedding encodes

information from its k-hop neighborhood.

The information captured by GNNs comes in two main forms:

1. Structural information – the topology of the graph, such as node degrees or local motifs (e.g., molecular substructures like benzene rings).
2. Feature-based information – aggregated attributes or features of neighboring nodes, similar to how CNNs gather information from spatially nearby pixels.

References

- Hamilton, W. L. (2020). Graph representation learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 14(3), 1-159.
- Stamile, Claudio, et al. *Graph Machine Learning : Take Graph Data to the Next Level by Applying Machine Learning Techniques and Algorithms*, Packt Publishing,
- Keita Broadwater and Namid Stillman. *Graph Neural Networks in Action*. Manning Publications, 2025. ISBN: 9781617299056.