# Graph Neural Network (GNN) methodologies

**SkipGNN predicting molecular interactions with skip-graph networks (https://github.com/kexinhuang12345/SkipGNN)**

🡪 Capture the structure of interaction networks

### Convolution

* “Looking at a function’s surroundings to make better/accurate predictions of its outcome”

## Node embedding

* Aims to encode each node into an embedding space while preserving the network structure information
* Output of a node embedding algorithm can be used as an input to downstream a machine learning model (Real-world networks does not have a pre-defined structure (i.e, can be of various shapes and sizes)
* Encoding entity relationships: Capture the context of each data point instead of just focusing on its attributes

Empirically (i.e., by means of observation or experience rather than theory or pure logic) initializing node attributes using one-hot position encoding is also good for SkipGNN.

One-hot encoding

E.g. Using *np.eye(3, dtype=int)* to represent category variable ‘colour’ [red green blue] as binary vectors

*([[1, 0, 0],* - red

*[0, 1, 0],* - green

*[0. 0, 1]])* – blue

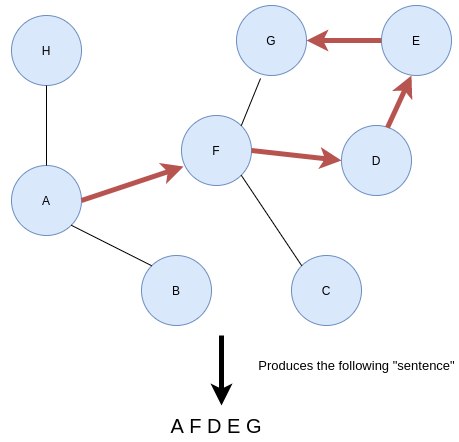
np.eye: 2-D array w/ ones on the diagonal and zeros elsewhere

Node2Vec (Node embedding algorithm)

*(Ref:* [*https://towardsdatascience.com/complete-guide-to-understanding-node2vec-algorithm-4e9a35e5d147*](https://towardsdatascience.com/complete-guide-to-understanding-node2vec-algorithm-4e9a35e5d147)*)*

*Source code: https://github.com/aditya-grover/node2vec*

* Resulting node embeddings are only affected by the network topology and not the node properties or attributes
* Need the whole graph to be available to learn the node embeddings
  + Cannot run separately on train and test data
  + When a new node is added to the graph, will need to re-run the node2vec algorithm on the whole graph to generate embedding for the new node
* Inspired by the skip-gram model
* Node2Vec = Skip-gram with Negative sampling (SGNS)
* Using random walks to generate a corpus of “sentences” from a given network



1. Start traversing the graph from a ‘start’ node (Node ‘A’ in this case)
2. Pick a neighboring node at **Random** and hop onto it
3. Repeat the process until a pre-defined **walk length** (defines how long the “sentences” will be)
4. For every node in the graph, node2vec algorithm generates a series of random walks with the particular node as the starting node

**Walks per node**: defines how many random walks should start from a particular node (i.e., how many “sentences” to generate starting from each node in the graph)

1. After “sentences” are generated using random walks, the algorithm inputs them into the skip-gram with negative sampling (SGNS) model and retrieve the hidden layer weights as node embeddings
2. Node2vec implements **second-order biased random walks (takes into account both the current and previous state)**

🡪 1st-order random walks: probability of returning to a previous node or any other node is equal

2nd-order: likelihood of backtracking the walk and immediately revisiting a node in the walk is controlled by the *return parameter p.*

(*higher p-value* = lower chance of revisiting a node. 🡪 Encourages moderate graph exploration

*Low p-value* = higher chances of backtracking in the walk 🡪 Keeps the random walk closer to the ‘start’ node)

Skip-Gram: Tries to predict the source context words (surrounding words) given a target word (centre word)

Word Vector (row of real-valued numbers): Represent words as multidimensional continuous floating point numbers where semantically similar words are mapped to proximate points in geometric space

Semantics of the word are embedded across the dimensions of the vector

(semantically similar words have similar vectors, words that are used in a similar context will be mapped to a proximate vector space)

Using *mathematical operators* on vectors (e.g. addition and subtraction) to create new word vectors:

E.g. *(king – man) + woman = queen*, where new word vector [*(king – man) + woman*] maps most closely to the word vector for [*queen*]

Input: One-hot encoded vector representing the input word

Output: One-hot encoded vector representing the context word

Goal: Learn weights of the hidden layer (i.e., word embedding)

Number of neurons in the hidden layer determines the embedding dimension (i.e, size of the vector representing each word in the vocabulary)

Influence of context ‘window size’ parameter:

*(Ref: Dependency-Based Word Embeddings by Levy & Goldberg)*

Larger 🡪 tends to capture more topic/domain information

Smaller 🡪 tends to capture more information about the word itself (e.g., what other words are functionally similar)

Negative sampling

1. Each training sample will update all of the weights in the neural network
2. Updating thousands of weights for every input context training pair is very expensive
   1. Negative sampling solves this performance issue by having each training sample modify only a small subset of the weights rather than all of them

**Node2Vec (params used for trimmed dataset)**

Github repo given in SkipGNN paper

Dimensions = 128

Walk-length = Used 20 walk length as suggested by SkipGNN paper) longer walk lengths improve the embedding quality (default = 80)

Num-walks = 10

Window-size (context size for optimization) = 10

Iter (number of epochs in SGD) = 1

Workers (number of parallel workers) = 8

P (return hyperparameter) = 1

Q (inout hyperparameter) = 1

Hadamart product perform better than average and weighted L1/L2 for link prediction

**Deepwalk** uses the exact same params as node2vec.

**Struc2vec** uses walk-length = 80 and num-walks = 20, remaining exact same params as node2vec

Divide each dataset into train, validation and test sets in a 7:1:2 ratio

Generate negative counterparts by sampling the complement set of positive samples

(No. of negative samples = no. of positive samples)

* Five independent runs with different random splits of the dataset

Ranges of hyper-parameters are set as follows (from paper):

Learning rate (lr): [1e-3 (0.001), 5e-4 (0.0005), 1e-4 (0.0001), 5e-5 (0.00005)]

Mini-batch size (batch\_size): [32, 64, 128, 256, 512]

Dropout rate (dropout): [0, 0.05, 0.1, 0.2]

Hidden size (hidden\_decode1): [16, 32, 64, 128]

Hidden size in the 1st layer (hidden1): 64

Hidden size in the 2nd layer (hidden2): 16

🡪 10 runs random search based on best average prediction performance on validation set

* My hyper-parameters dictionary:

epoch = [10, 15, 20, 25, 30]

Learning rate (lr): [5e-3 (0.005), 1e-3 (0.001), 5e-4 (0.0005), 1e-4 (0.0001), 5e-5 (0.00005)]

Mini-batch size (batch\_size): [16, 32, 64, 128, 256]

Hidden1, Hidden2: [[32, 16], [64, 32], [96, 48], [128, 64], [256, 128]]

Hidden size (hidden\_decode1): [32, 64, 128, 256, 512]

Dropout rate (dropout): [0, 0.05, 0.1, 0.2, 0.5]

Repeat construct\_datasets (to re-sample negative pairs), create\_fold

Randomly select values from hyper-params dictionary

## Feature importance

Hyper-parameters given in paper:

Epochs = 15, lr = 5e-4, batch\_size = 256, hidden1 = 64, hidden2 = 16, hidden\_decode1 = 512, dropout = 0.1

‘Baseline’ reference to random search from 30 iterations:

Epochs = 15, lr = 0.001, dropout = 0.1

To find: batch\_size, hidden1, hidden2, hidden\_decode1

Run training 10 times (on the same dataset), randomize a different hyper-parameter each time, and then compare the performance.

* For each value per hyper-parameter, run training twice (2 runs\*5 options/parameter)
* Tune batch\_size with hidden\_decode1 (hidden\_decode1 = 2 \* batch\_size)
* Tune hidden1 with hidden2 (hidden2 = hidden1 / 2 OR hidden1 / 4)

## Graph Embeddings

Graph2vec – modification to node2vec, graph2vec learns to embed a graph’s sub-graphs

Structural Deep Network Embedding (SDNE) – learn from two distinct metrics (First-order proximity and Second-order proximity)

## **Hierarchical Representation Learning for Networks (HARP)**

*(References:*

[*https://towardsdatascience.com/overview-of-deep-learning-on-graph-embeddings-4305c10ad4a4*](https://towardsdatascience.com/overview-of-deep-learning-on-graph-embeddings-4305c10ad4a4)

[*https://github.com/GTmac/HARP*](https://github.com/GTmac/HARP)*)*

* Improvement to embedding/walking based models
* Embedding/walking based models risked getting stuck in local optima since their objective functions are non-convex
* HARP avoid local optima by better weight initialization
* Use graph coarsening to aggregate related nodes into “supernodes”

Graph coarsening: Technique to reduce the size of a graph while maintaining essential properties

* Essentially a graph-preprocessing step that simplifies the graph to make for faster training
* After coarsening the graph, it then generates an embedding of the coarsest (most coarse) “supernode”, followed by an embedding of the entire graph (which itself is made of supernodes). This strategy is followed for each “supernode” in the entire graph.
* Can be used in conjunction with embedding algorithms like LINE, Node2Vec and Deepwalk

\*\* Nodes in input file indexed from 0 to N-1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Dataset description | ROC-AUC | PR-AUC | F1-Score |
| HuRI  (PPI interaction network from Human Reference Protein Interactome Mapping) | 5604 proteins  23,322 pos | 0.9121 | 0.9188 | 0.8137 |
| C.elegan (Roundworm) | 1598 proteins  2877 pos  1670 neg | 0.9944 | 0.9974 | 0.9737 |
| Drosophila  (Fruit flies) | 5503 proteins  19712 pos  14900 neg | 0.9971 | 0.9983 | 0.9856 |
| E.coli | 1254 proteins  5576 pos  4031 neg | 0.9607 | 0.9780 | 0.9161 |
| Human Protein Reference Database (HPRD) | 9463 proteins  36,591 pos  11,324 neg | 0.9913 | 0.9960 | 0.9801 |
| Human | 8516 proteins  31,761 pos  25,203 neg | 0.9904 | 0.9939 | 0.9669 |
| HPIDB | 2136 proteins  5193 interactions | 0.7606 | 0.7361 | 0.5839 |
| My dataset  (after hyper-parameter tuning, avg of 50 iterations) | 1126 proteins  (43 pathogens,  1083 humans)  4343 interactions | 0.8059 | 0.6951 | 0.7314 |