# A report on
# Deep Neural Networks for Learning Graph Representation

by

**Julina Maharjan**

**Department of Computer Science**
**Kent State University**

**Original Authors:**
Shaosheng Cao, Wei Lu, Qiongkai Xu

*Submitted for the*
*Course CS-79995*

**Supervisor:**
Dr. Ruoming Jin
`rjin1@kent.edu`

May, 2021

# Contents

# Chapter 1

# Introduction

## 1.1 Objective

DNGR is the model that generates low dimensional vector representation for each vertex by capturing the graph structural information. It includes two steps:

1. Random Surfing model to capture the graph structure
   - instead of the sampling-based method for generating linear sequences
2. PMI matrix factorization as a solution for an objective function of SGNS
   - instead of SVD, uses the Stacked denoising autoencoders for extracting complex non-linear features

## 1.2 Approach

Many methods existed to learn effective vector representation for linear sequence in NLP
e.g. Random walk
Can we leverage this idea to learn vertex representations for graph structure?

- The graph structure can be converted into a large collection of linear structures
- And Skip-Gram model can be utilized to learn low-dimensional representations for vertices from such linear structures

### 1.2.1 Random Surfing Model

Why?

- yields a probabilistic co-occurrence for weighted graphs
- very similar to the one obtained by sampling linear sequence from graphs (w/o sampling)

### 1.2.2 SGNS: Skip Gram Negative Sampling

Why¿

- the objective function of SGNS has a intrinsic relation with factorizing PPMI matrix of the words and contexts
- SVD only reduces to linear dimension reduction
- Levy and Goldberg showed that PPMI matrix itself is an explicit representation matrix of the graph
- SVD didn't outperform the representations from the PPMI matrix
- thus, the stacked autoencoders are used to learn low level features with high level abstraction

# Chapter 2

# Literature Review

## 2.1 Algorithms

### 2.1.1 Deep Walk

**Algorithm 1:** DeepWalk

```
for $n = 1, 2, \ldots, N$ do
    Pick $w_1^n$ according to a probability distribution $P(w_1)$;
    Generate a vertex sequence $(w_1^n, \cdots, w_L^n)$ of length $L$ by a
      random walk on network $G$;
    for $j = 1, 2, \ldots, L - T$ do
        for $r = 1, \ldots, T$ do
            Add vertex-context pair $(w_j^n, w_{j+r}^n)$ to multiset $\mathcal{D}$;
            Add vertex-context pair $(w_{j+r}^n, w_j^n)$ to multiset $\mathcal{D}$;
Run SGNS on $\mathcal{D}$ with $b$ negative samples.
```

### 2.1.2 Random Walk

Let $G = (V, E)$ be an undirected graph. Consider the random process that starts from some vertex $v \in V(G)$, and repeatedly moves to a neighbor of the current vertex chosen uniformly at random.
For $t \geq 0$, and for $u \in V(G)$, let $p_t(u)$ denote the probability that you are at vertex $u$ at time $t.p_t \in R^n$. Clearly, $\sum_{u \in V(G)} p_t(u) = 1$

$$p_{t+1}(v) = \sum_{(u,v) \in E(G)} p_t(u) \cdot \frac{1}{d(u)}$$

We can write this using matrix notation $W = W_G$ :

$$[W_G]_{i,j} = \begin{cases} \frac{1}{d(j)} & if(i,j) \in E(G) \\ 0 & otherwise \end{cases}$$
$$W_G = AD^{-1}$$

**Limitation:**

- only transform unweighted graph structural information into linear sequence
- constraints on hyper parameters walk length, $\eta$ and total walk, $\gamma$
- based on sampled linear sequences

**Solution:**

- Random Surfing: model that generates POM from a weighted graph which avoids the expensive sampling process

### 2.1.3 Skip Gram Negative Sampling

**Setting and Notation:** The skip-gram model assumes a corpus of words $w \in V_W$ and their contexts $c \in V_C$, where $V_W$ and $V_C$ are the word and context vocabularies.
The words come from an unannotated textual corpus of words $w_1, w_2, \ldots, w_n$ (typically $n$ is in the billions) and the contexts for word $w_i$ are the words surrounding it in an L-sized window $w_{i-L}, \ldots, w_{i-1}, w_{i+1}, \ldots, w_{i+L}$

We denote the collection of observed words and context pairs as $D$.
We use $\#(w,c)$ to denote the number of times the pair $(w,c)$ appears in $D$. Similarly, $\#(w) = \sum_{c' \in V_C} \#\left(w, c'\right)$ and $\#(w) = \sum_{w' \in V_w} \#\left(w', c\right)$ are the number of times $w$ and $c$ occurred in $D$, respectively.

**SGNS Objective Function**

$$\ell = \sum_{w \in V_W} \sum_{c \in V_C} \#(w,c) \left(\log \sigma(\vec{w} \cdot \vec{c}) + k \cdot E_{c_N \sim P_D} \left[\log \sigma\left(-\vec{w} \cdot \vec{c}_N\right)\right]\right)$$

Simplifying above equation leads to following: (Paper: Neural Word Embedding as Implicit Matrix Factorization - Levy and Goldberg 20014)

$$\vec{w} \cdot \vec{c} = \log\left(\frac{\#(w,c) \cdot |D|}{\#(w) \cdot \#(c)} \cdot \frac{1}{k}\right) = \log\left(\frac{\#(w,c) \cdot |D|}{\#(w) \cdot \#(c)}\right) - \log k$$

Interestingly, the expression $\log\left(\frac{\#(w,c)\cdot|D|}{\#(w)\cdot\#(c)}\right)$ is the well-known pointwise mutual information (PMI) of $(w,c)$, which we discuss in depth below. Finally, we can describe the matrix $M$ that SGNS is factorizing:

$$M_{ij}^{\text{SGNS}} = W_i \cdot C_j = \vec{w}_i \cdot \vec{c}_j = PMI\left(w_i, c_j\right) - \log k$$

### 2.1.4 Auto Encoder

Auto Encoder performs two actions

- – an encoding step, followed by decoding step. In the encoding step, a function $f_{\theta_1}(\cdot)$ applied to the vector in the input space and send it to new feature space. An activation function is typically involved in this process to model the non-linearities between the two vector spaces – the space of input vectors and th space of latent vector representations.
- At the decoding step a reconstruction function $g_{\theta_2}(\cdot)$ is used to reconstruct th original input vectors back from the latent representation space.

#### Auto Encoder Objective Function

Let us assume that $f_{\theta_1}(x) = \sigma\left(W_1 x + b_1\right)$ and $g_{\theta_2}(y) = \sigma\left(W_2 y + b_2\right)$, where $\sigma(\cdot)$ is the activation function, $\theta_1 = \{W_1, b_1\}$ are the weights (parameters) involved in the encoder, and $\theta_2 = \{W_2, b_2\}$ are the weights (parameters) involved in the decoder. Here $W_1$ and $W_2$ are linear maps (matrices) transforming the vectors from the input space, and $b_1$ and $b_2$ are the bias vectors. Our goal is to minimize the following reconstruction loss function by finding $\theta_1$ and $\theta_2$ :

$$\sum L\left(x^{(i)}, g_{\theta_2}\left(f_{\theta_1}\left(x^{(i)}\right)\right)\right)$$

where $L$ is sample-wise loss, and $x^(i)$ is the $i$-th instance.

### 2.1.5 Stacked Auto Encoder

- Multi-layer deep neural network consisting of multiple layers of such autoencoders.
- use the layer-wise training approach to extract essential regularities capturing different levels of abstractions from the data layer by layer, with higher layers conveying higher-level abstractions from the data.

### 2.1.6 Stacked denoising Auto Encoder

- partially corrupts the input data before taking the training step
- each input sample $x$ is corrupted randomly by assigning some of the entries in the vector to 0 with a certain probability.
- **Loss function:**
  Similar to standard autoencoders, we again reconstruct data from the latent representations.

  $$\min_{\theta_1,\theta_2} \sum_{i=1}^{n} L\left(x^{(i)}, g_{\theta_2}\left(f_{\theta_1}\left(\tilde{x}^{(i)}\right)\right)\right)$$

  where $\tilde{x}^{(i)}$ is corrupted input data of $x^{(i)}$, and $L$ is the standard squared loss.

# Chapter 3

# Methodology

## 3.1 DNGR Model

### 3.1.1 3 main steps

1. Random surfing model to capture graph structural information and generate a probabilistic co-occurrence matrix

2. Calculation of PPMI matrix based on our probabilistic co-occurrence matrix

3. A Stacked Denoising autoencoder to learn low-dimensional vertex representations

### 3.1.2 Random Surfing and Context Weighting

Why Random Surfing ?

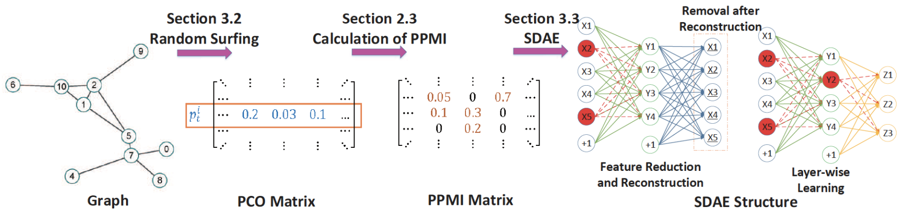


Figure 3.1: Main Components: Random Surfing, Calculation of PPMI matrix and feature reduction by SDAE

- Sampling sequences have finite length, thus makes difficult to capture the accurate contextual info for vertices that appears at the edges of the sequence
- Also, it is very difficult to find hyper parameters ; walk length $\eta$ and total walks $\gamma$

### 3.1.3 Random Surfing Model

Let current vertex be $i$-th vertex and $A$ be a transition matrix.
Then, $p_k$ is a row vector, whose $j$-th entry is the probability of reaching the $j$-th vertex after $k$ steps

$$p_k = \alpha \cdot p_{k-1}A + (1-\alpha)p_0$$

### 3.1.4 Stacked Denoising Autoencoder

- Matrix factorization of PPMI matrix to low dimensional word/vertex embedding
- SVD: Limitation in Linear Factorization
- Need non-linear factorization

Why stacked and denoising autoencoder ?

- to learn high-level abstractions at each layer
- denoising to corrupt the input data corrupt each input sample x randomly by assigning some of the entries in the vector o 0 with a certain probability

**PPMI Algorithm**

```
$Input : PPMImatrixX, NumberofSDAElayers\Gamma$
$1.InitializeSDAE$
$Setnumberofnodesn_jinlayerj, X^{(1)} = X$
$2.Greedylayer - wisedtraining$
$Forj = 2to\Gamma$
   $2.1ConstructonehiddenlayerSDAEwithinputofX^{(j)}$
   $2.2Learnhiddenlayerrepresentationh^{(j)}$
   $2.3X^{(j)} = h^{(j)} \left(X^{(j)} \in R^{n \times n_j}\right)$
$Output : MatrixofverticesrepresentationsR$
```

- all the $X_i$'s are input data
- $Y_i$'s are learned representation in first layer
- $Z_i$'s are learned representation in second layer

The objection function for SDAE:

$$\min_{\theta_1,\theta_2} \sum_{i=1}^{n} L\left(x^{(i)}, g_{\theta_2}\left(f_{\theta_1}\left(\tilde{x}^{(i)}\right)\right)\right)$$

where $x^{(i)}$ is corrupted input data of $x^{(i)}$, and $L$ is the standard squared loss.

# Chapter 4

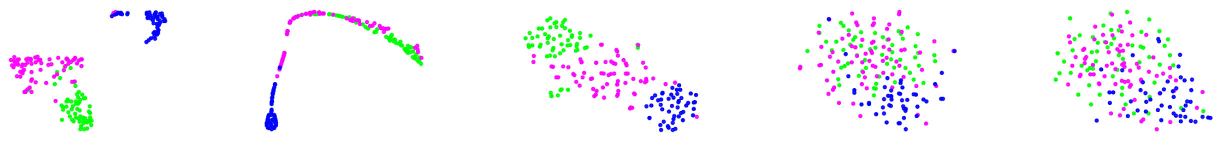
# Result

## 4.1 Experiments

### 4.1.1 Visualization of Wine

- Comparison on the performance of DNGR with SVD, DeepWalk and SGNS on Wine

- Vertex representation were fed to t-SNE; mapped to 2D.
- DNGR ($\alpha = 0.98$) showed better clustering among all.



(a) DNGR ($\alpha$=0.98) (b) DNGR($\alpha$=1) (c) SVD ($\alpha$=0.98) (d) DeepWalk (e) SGNS

Figure 4.1: Comparison result across different models for Wine Classification

### 4.1.2 Experiment Set up

**GitHub Link:** https://github.com/ShelsonCao/DNGR
**Usage:**
python3 DNGR.py −− graphType undirected −− input wine.edgelist −− output representation

```
def random_surfing(adj_matrix, max_step, alpha):
    # Random Surfing
    nm_nodes = len(adj_matrix)
    adj_matrix = scale_sim_mat(adj_matrix)
    P0 = np.eye(nm_nodes, dtype='float32')
    M = np.zeros((nm_nodes, nm_nodes), dtype='float32')
    P = np.eye(nm_nodes, dtype='float32')
    for i in range(0, max_step):
        P = alpha * np.dot(P, adj_matrix) + (1 - alpha) * P0
        M = M + P

    return M

def PPMI_matrix(M):
    M = scale_sim_mat(M)
    nm_nodes = len(M)

    col_s = np.sum(M, axis=0).reshape(1, nm_nodes)
    row_s = np.sum(M, axis=1).reshape(nm_nodes, 1)
    D = np.sum(col_s)
    rowcol_s = np.dot(row_s, col_s)
    np.seterr(divide='ignore')
    PPMI = np.log(np.divide(D * M, rowcol_s))
    PPMI[np.isnan(PPMI)] = 0.0
    PPMI[np.isinf(PPMI)] = 0.0
    PPMI[np.isneginf(PPMI)] = 0.0
    PPMI[PPMI < 0] = 0.0

    return PPMI


def model(data, hidden_layers, hidden_neurons, output_file, validation_split=0.9
    train_n = int(validation_split * len(data))
    batch_size = 50
```

```
train_data = data[:train_n, :]
val_data = data[train_n:, :]

input_sh = Input(shape=(data.shape[1],))
encoded = noise.GaussianNoise(0.2)(input_sh)

for i in range(hidden_layers):
    encoded = Dense(hidden_neurons[i], activation='relu')(encoded)
    encoded = noise.GaussianNoise(0.2)(encoded)

# input 128 64 32 64  128 output
# //[128, 64, 32]
decoded = Dense(hidden_neurons[-2], activation='relu')(encoded)
for j in range(hidden_layers - 3, -1, -1):
    decoded = Dense(hidden_neurons[j], activation='relu')(decoded)
decoded = Dense(data.shape[1], activation='sigmoid')(decoded)

autoencoder = Model(input_sh, decoded)
autoencoder.compile(optimizer='adadelta', loss='mse')

checkpointer = ModelCheckpoint(filepath='data/bestmodel' + output_file + ".h
earlystopper = EarlyStopping(monitor='val_loss', patience=15, verbose=1)

train_generator = DataGenerator(batch_size, train_data, train_data)
val_generator = DataGenerator(batch_size, val_data, val_data)

autoencoder.fit_generator(train_generator,
                          steps_per_epoch=len(train_data),
                          epochs=100,
                          verbose=1,
                          validation_data=val_generator,
                          validation_steps=len(val_data),
                          max_queue_size=batch_size,
                          callbacks=[checkpointer, earlystopper])
enco = Model(input_sh, encoded)
enco.compile(optimizer='adadelta', loss='mse')
reprsn = enco.predict(data)
return reprsn
```

**Console Output:**

```
*** Generating Adjacency matrix (EDGELIST) ***
Num GPUs Available: 0
** Training DNGR (Deep Neural Network For Learning Graph Representation) Model ***
** Data: wine.edgeList
Calling ... DNGR Model
———————————————————–
** Parameters DNGR Model ***
Filename: wine.edgelist
Graph Type: undirected
(Random Surfing) Ksteps: 10
(PPMI) alpha: 0.98
(AutoEncoder) Hidden Layers: 3
(AutoEncoder) Hidden Neurons: [128, 64, 32]
———————————————————–
```

```
Epoch 00100: val-loss improved from 0.23136 to 0.23124, saving model to
data/bestmodelrepresentation.hdf5
Preprocessing the data using PCA...
Computing pairwise distances...
Computing P-values for point 0 of 178 ...
Mean value of sigma: 0.008671895811543645
Iteration 10 : error is 11.369875219491831
Iteration 20 : error is 9.80103401468988
Iteration 30 : error is 10.893799513029181
Iteration 40 : error is 11.059345662952541
Iteration 50 : error is 11.061018372618026
Iteration 60 : error is 10.94345349413246
Iteration 70 : error is 10.635025757175395
Iteration 80 : error is 10.87103052174422
.
.
.
Iteration 990 : error is 0.28491236291143884 Iteration 1000 : error is 0.2849123629114369
```

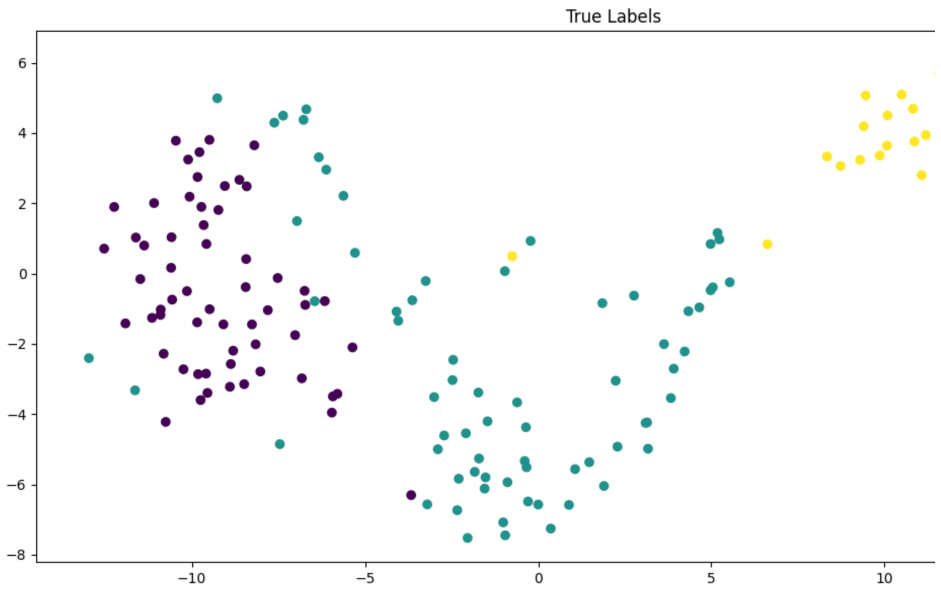


Figure 4.2: Wine classified for the given wine input

# Chapter 5

# Conclusion

- A Deep Graph Representation Model that encodes vertex representation in a low dimension
- Uses Random surfing and SDAE as the main components
- Experiments showed it outperformed many baseline algorithms

# References

[1] Wei Lu Shaosheng Cao and Qiongkai Xu. Deep neural networks for learning graph representations. *Xidian University, Singapore University of Technology and Design and Australian National University Nation UCT Australia -CRL.*

[2] Omer Levy and Yoav Goldberg. Neuralword embedding as implicit matrix factorization. *Bar-Ilan University.*

[3] Bullinara and Levy. Positive pointwise mutual information. 2007.

[4] Mikolov et al. Skip gram negative sampling. 2013.

[5] Al-Rfou Perozzi and Skiena. Deepwalk. 2014.

[6] Gutmann Michael and Hyv¨arinen Aapo. Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. *Journal of Machine Learning Research*, 135(35):13096–13106, 2013.