DrBC Report
Learning to Identify High BC Nodes

數據所碩一 楊子萱

March 25, 2023

Abstract

根據"Learning to Identify High Betweenness Centrality Nodes from Scratch: A Novel Graph Neural Network Approach"去實作DrBC演算法，其中包含DrBC的基礎介紹，以及Encoder與Decoder實作的詳細辦法，使用pairwise ranking loss的方式計算node pair間的Loss值，最終再透過論文中Evaluation Metrics的Top-N% accuracy以及Kendall tau distance去對模型做出評分，共使用了30個5000個nodes的synthetic graphs去測試模型的預測正確性，而模型最高的預測率在top-1% accuracy可到達93.87%的準確率。

1 Introduction

Betweenness centrality (BC)是一個在於network探索時常見的評量標準，主要用來測量到底有多少節點間的最短路徑會透過此節點，當多數節點間的最短路徑都會通過此節點時，我們就會推估此節點具有高的Betweenness centrality，然而傳統的BC演算法無法應該變動的network topology，因此在於現今的social network與p2p network將會無法使用，或是需要耗費大量的運算資源再次重新訓練BC值。

本篇論文提出了DrBC，是一個GNN-based的ranking model主要用於辨識高BC score的節點，使用了Encoder-Decoder的方法，目的在於將節點的information映射到ranking score vector，在此我們取節點間的相對排名，並非直接的去估算出正確的BC值，在Encoder階段會去抓取每一個節點的重要資訊崁入一個embedding vector中，Decoder時再將embedding vector轉換成BC ranking scores。在於model參數的設計上，使用end-to-end manner，因此可以訓練在不同大小的small-scale人工合成的training data上，並且在將訓練好的模型應用於真實世界的大型網路中。

2 Proposed method: DrBC

在這個章節中，會詳細的解說DrBC實作的各個組件，其中包含在於可調整的參數設計、degree、adjacent matrix計算上的發現、Encoder中Neighborhood Aggregation與GRU cell、Decoder中的two-layer MLP、loss function的計算、backward後權重的更新，其中會特別針對Encoder中的Neighborhood Aggregation如何去平行計算多個節點的Neighborhood weight多做解釋。

2.1 Parameters

在於模型的初始值設定上，給予了幾個可以設定的參數，去測試在不同參數下模型的Loss下降情形，以及預測的正確性，期望能夠找出最佳的模型組合。

1. learning rate：可調控gradient的更新速度，當值越大時，可能會造成更新速度過快反而無法收斂，反之。則可能需要充足的epoch才能夠收斂到最佳解。

2. embedding size：決定Encoder每一個node的embedding vector size，也會影響W0-U3各個權重矩陣的size。

3. node init：用於決定node initial feature X_v 。

4. layer num：能夠調控在於Encoder階段，會將Neighborhood Aggregation以及GRU cell運算幾次。

5. batch：設計為可以組合多張power-law graph去訓練model。

在此篇報告中的參數設定主要都是參考原始論文中的建議去做測試，Embedding size設為128，node init的feature數量設為3，組成[d_v, 1, 1]的初始向量。 layer num因為在論文中提到多次訓練的結果後，最佳的層數為5，因此也遵照論文的設定設為5，最後batch主要設為1，因為在於多次測試後發現，同樣數量的node，例如一次輸入500個node與100個node分為5個power-law graph去組合，組合多個會導致訓練上較難以收斂，以及訓練速度大幅減低且正確性下降的問題，因此此篇報告還是以一張較大的power-law graph去訓練為主。

另外，在於原始DrBC的權重設定上，也將其撰寫在於__init__ 在於每一次呼叫class時，就會初始化設定這些參數，以下介紹這些權重的初始化方式以及形狀。

- W0：normal distribution，shape=[node init,embedding size]
- W1、W2、W3：normal distribution，shape=[embedding size,embedding size]
- U1、U2、U3：normal distribution，shape=[embedding size,embedding size]
- W4：normal distribution，shape=[embedding size,1]
- W5：normal distribution，shape=[1, 1]
- activation function：Relu

後續會根據上面使用者能夠輸入的參數，以及DrBC原始設定參數上，在Encoder Decoder的章節中，逐步透過公式及運算方法做出解釋。

2.2 Encoder & Decoder

$$z_v = ENC(A, X, \theta_{ENC}) \tag{1}$$

A為相鄰矩陣，X為node feature，θ_{ENC}為Encoder中會使用到的權重矩陣 {W0、W1、W2、W3、 U1、U2、U3}，首先會先介紹兩個重要的組件Neighborhood Aggregation與GRU Cell，以及Decoder，後續再完整說明Algorithm1在加上Decoder後的整個實作運行流程。

Neighborhood Aggregation.

$$h_{N_{(v)}}^{(l)} = \sum_{j \in N_{(v)}} \frac{1}{\sqrt{d_v + 1} \times \sqrt{d_j + 1}} h_j^{(l-1)} \tag{2}$$

d_v d_j分別代表 v 與 j 點的degree，$h_j^{(l-1)}$表示j點在於$l-1$層的embedding vector，上述公式顯示將節點 v的所有鄰居節點 j經過權重的轉換再全部加總去表示為這一層節點 v 的aggregate neighbors。因此在模型設計時，寫了一個adjcent_with_weight( )，用於計算graph的degree與adjacent matrix，並且計算每一條邊的權重，將權重值傳回adjacent matrix，形成adjcent matrix with weight，因此當$h_j^{(l-1)}$乘以adjcent matrix with weight即可形成$h_{N_{(v)}}^{(l)}$。

1. 計算graph的degree，因為是undirect graph，而torch_geometric中的degree僅計算單邊，因此計算兩邊的degree再做加總，degree總和為edge數量的兩倍。
2. 計算graph的adjacent matrix，原因與上述相同，因此再計算左上與右下後，將其組合成完整的adjacent matrix，有邊的為1，沒有邊的為0。
3. 計算edge weight。每一條邊的edge weight為v點的degree+1開根號與j點的degree+1開根號相乘後的倒數，因此先算出每一個點的degree+1開根號後，取出每一條邊(v,j)各自的weight再相乘,形成該條邊的weight。
4. 最後再將計算完的edge weight依照每一條邊回傳至adjacent matrix，形成adjcent matrix with weight。

GRU Cell

$$u_1 = sigmoid(W_1 h_{N_{(v)}}^{(l)} + U_1 h_v^{(l-1)}) \tag{3}$$

$$r_1 = sigmoid(W_2 h_{N_{(v)}}^{(l)} + U_2 h_v^{(l-1)}) \tag{4}$$

$$f_1 = tanh(W_3 h_{N_{(v)}}^{(l)} + U_3(r_1 \odot h_v^{(l-1)})) \tag{5}$$

$$h_v^{(l} = u_1 \odot f_1 + (1 - u_1) \odot h_v^{(l-1)}) \tag{6}$$

此篇論文中的GRU cell為原始GRU的變形，原始的GRU中X的輸入為一個時間段的資料，且會再次傳入上一個時間段的hidden state，而本篇文章中的x輸入

為current layer的node embedding vector $h_v^{(l-1)}$，與傳入上一層的 Aggregate neighborhood的information $h_{N(v)}^{(l)}$。因此在本實作中，直接地針對公式做實現，其中為了使得權重能夠正確的在於計算後被backward( )，因此在於矩陣乘法以及element-wise的乘法時，皆使用torch的套件做運算。

Decoder.

$$y_v = DEC(z_v; \theta_{DEC}) = W_5 ReLU(W_4 z_v) \tag{7}$$

Decoder是一個簡單2層的MLP，z_v為Encoder的output，θ_{DEC}為Decoder中會使用到的權重矩陣{W4、W5}，主要將Encoder後每一個節點的embedding vector轉換成BC ranking score y_v，最終去計算Loss值，再更新權重。

Algorithm 1 DrBC encoder function

Input: Network $G = (V, E)$; input features $\{X_v \in \mathbb{R}^c, \forall v \in V\}$; depth L; weight matrices $W_0, W_1, U_1, W_2, U_2, W_3, U_3$.
Output: Vector representations $z_v, \forall v \in V$.

Initialize $h_v^{(0)} = X_v$;
$h_v^{(1)} = ReLU(W_0 h_v^{(0)})$, $h_v^{(1)} = h_v^{(1)} / \|h_v^{(1)}\|_2, \forall v \in V$;
for $l = 2$ to L **do**
  for $v \in V$ **do**
    $h_{N(v)}^{(l)} = \sum_{j \in N(v)} \frac{1}{\sqrt{d_v+1} \cdot \sqrt{d_j+1}} h_j^{(l-1)}$;
    $h_v^{(l)} = GRUCell(h_v^{(l-1)}, h_{N(v)}^{(l)})$;
  end for
  $h_v^{(l)} = h_v^{(l)} / \|h_v^{(l)}\|_2, \forall v \in V$;
end for
$z_v = max(h_v^{(1)}, h_v^{(2)}, \ldots, h_v^{(L)}), \forall v \in V$;

Figure 1: Algorithm1 DrBC encoder function.

簡述完上述在於實作中的重要組件後，可以看到Figure1 Algorithm1的演算法，逐步講解Encoder & Decoder的流程。

1. 將edge index與num nodes傳入上面介紹過的adjcent_with_weight( )，回傳得到每個節點的degree與adjcent matrix with weight做後續的使用，相鄰矩陣的形狀為[node cnt, node cnt]。

2. 初始化X_v，根據上面得到的degree去初始化每一個v的特徵值為$[d_v, 1, 1]$，因為是將全部的點做平行運算，因此X的形狀為[node cnt, 3]，$X = h^{(0)}$。

3. 將$h^{(0)}$與W_0矩陣相乘形成[node cnt, embed dim]的矩陣，再經過一層的激勵函數ReLU，形成$h^{(1)}$，因為期望算出來的值能夠較大，因此在這邊就不執行normalize。

4. 在第二到第L層的Layer中，重複進行Neighborhood Aggregation與GRU Cell。

   (a) Neighborhood Aggregation：adjcent matrix with weight$\times h^{(l-1)} = h_N^{(l)}$，形狀為[node cnt, embed dim]。

(b) GRUcell：在GRU運算中按照上述公式所實作，得到l層的embedding vector，因為GRU中的權重矩陣形狀皆為[embed dim, embed dim]，因此最終輸出皆為[node cnt, embed dim]。

(c) 透過torch.minimum去比較原始layer與新的layer中，每一個點值較大的做回傳，此處也因為期望獲得較大的值，而改良模型未做normalize。

5. 將最終的embedding vector z_v傳入Decoder中，與W_4矩陣相乘經過一次的ReLU，再與W_5矩陣相乘形成[node cnt, 1]的矩陣，回傳預測結果y。

2.3 Train & Predict

Algorithm 2 Training algorithm for DrBC

Input: Encoder parameters $\Theta_{ENC} = (W_0, W_1, U_1, W_2, U_2, W_3, U_3)$, Decoder parameters $\Theta_{DEC} = (W_4, W_5)$
Output: Trained Model M
1: **for** each episode **do**
2: Draw network G from distribution D (like the power-law model)
3: Calculate each node's exact BC value $b_v, \forall v \in V$
4: Get each node's embedding $z_v, \forall v \in V$ with Algorithm 1
5: Compute BC ranking score y_v for each node v with Eq. (13)
6: Sample source nodes and target nodes, and form a batch of node pairs
7: Update $\Theta = (\Theta_{ENC}, \Theta_{DEC})$ with Adam by minimizing Eq. (15)
8: **end for**

Figure 2: Algorithm2 Training algorithm for DrBC.

上一小節講述了Encoder & Decoder的詳細實作方法，而在本小節將透過論文中的Algorithm2，講解細部的實作方式。

Train.

1. 在train( )中可傳入兩個參數，分別是episode表示做幾個epoch，num nodes表示期望生成節點數為多少的power-law graph，其中的batch為模型初始設定，表示同時生成幾張power-law graph去做訓練。

2. 將生成的graph透過networkx的套件計算出正確的BC值，再針對BC值取log去減少BC值間的差異，因為在loss function中也有針對預測值取log。

3. 去計算graph的edge index與number of nodes傳入Encoder_Decoder( )中回傳預測值。

4. 將預測值與BC的ground truth傳入loss( )中。

   (a) S為建立隨機node pair i的index，E為建立隨機node pair j的index，參照論文的建議隨機取得node數量的5倍node pair。

   (b) 從預測值中根據index(S,E)去取出相對應的值並相減，對ground truth也做相同的操作，得到兩者的pairwise loss。

$$C_{i,j} = -g(b_{ij}) * log\sigma(y_{ij}) - (1 - g(b_{ij})) * log(1 - \sigma(y_{ij})) \tag{8}$$

$$Loss = \sum_{i,j \in V} C_{i,j} \tag{9}$$

(c) 根據上列的公式計算出Loss值並回傳，其中的g(x)為sigmoid function，σ為ReLU function 。

5. 將得到的Loss值傳入optimal( )

   (a) 將Loss值傳入backward( )，得到各個權重矩陣的gradient。

   (b) 更新權重矩陣。將原始的權重矩陣減去learning rate*gradient。

   (c) 由於經過計算後的權重矩陣會脫離原始的計算圖中，W.is_leaf = False，因此需要再加上W.retain_grad使得新的權重矩陣也能夠計算對應的gradient。

Predict.

1. 傳入torch.Data所形成的grpah，以及其BC score，取得其edge index再傳入Encoder_Decoder( )中。

2. 根據回傳的預測值與BC score一同傳入loss( )，回傳本次預測的預測值與loss值。

3 Experiment

3.1 Experiment setup

在本次的測試中，將我實做的DrBC與Networkit現有的KADABRA去相比較，KADABRA主要用於識別前N%BC節點，雖然在於論文中有提到為了運算效能的問題而損失了正確性，但因為時間的緣故，因此還是選用KADABRA作為主要比較工具。

其中將測試不同training node size的訓練下，用於30個scale 5000的synthetic graphs下預測準確性為何，測試的size為200,300,500,800,1000，並且計算不同scale底下訓練所需的時間為何，本次測試用於20-core server with 512GB memory，再加上NVIDIA Ampere A100的GPU。

3.2 Evaluation Metric

在於實驗的階段使用了兩種Evaluation Metric，分別是Top-N% accuracy與Kendall tau distance。

$$Top - N\% = \frac{|returnTop - N\%nodes \bigcap trueTop - N\%nodes|}{\lceil |V| \times N\% \rceil} \tag{10}$$

$$K(\tau_1, \tau_1) = \frac{2(\alpha - \beta)}{n * (n-1)} \tag{11}$$

其中Top-N%主要是針對預測中BC分數最高的N%與真實BC分數最高的N%去做比對得出正確性，是現在Network analysis中最常用的一種評量方式，本次的報告中分別使用了Top-1%、Top-5%、Top-10%去做測試，但缺點是會較集中的去觀察前幾%的正確性，但未對整體的正確性做出評分，因此在這邊也引入了Kendall tau distance，會針對整體的序列去做檢測，將一致的pair減去不一致的pair乘以2再除以節點成節點數-1，值域的範圍會在[-1, 1]之間，當值越大則表示一致性越高，預測的準確性越高。

Hyper-parameter	learning rate	embedding size	node init	layer num	batch	Epoch
Value	0.01	128	3	5	1	100

Table 1: Hyper-parameter configuration for DrBC.

3.3 Results on Synthetic Networks

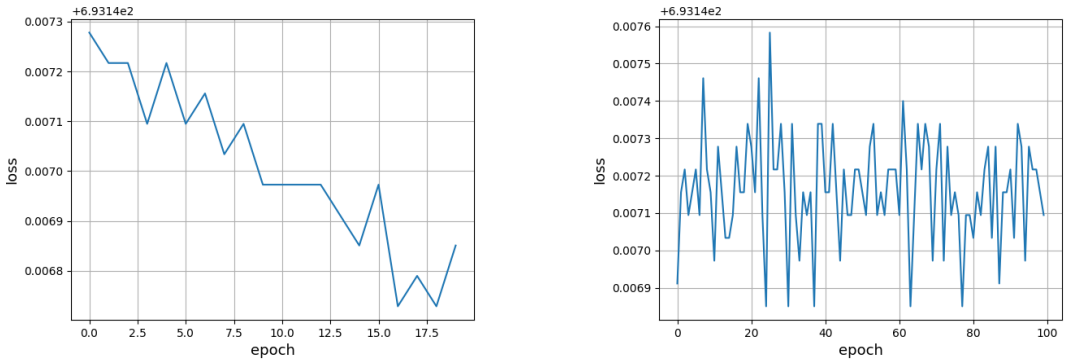

Figure 3: training loss on 200 nodes

Accuracy	Top-1%	Top-5%	Top-10%	Kendall tau	Training times
200	0.9387	0.8839	0.8592	0.4121	41.98
300	0.9387	0.8825	0.8561	0.4215	90.08
500	0.792	0.5963	0.3096	-0.2176	251.85
800	0.77	0.6623	0.6306	-0.1165	699.70
1000	0.9387	0.8816	0.8538	0.4242	1116.69
KADABRA	0.962	0.9375	0.9181	0.3191	0.1169

Table 2: DrBC's generalization results on different scales.

本次實驗了在於不同的training scale下模型對於30個5000nodes的Synthetic Networks的預測準確率為何，結果已呈現在table2中，由於本身的模型穩定度不夠，可以透過Figure3看出loss值的相當浮動，無法保證在設定的epoch下一定能夠收斂，因此結果取自多次預測後預測效果最好的作為此模型能夠達成的效果為呈現目的，因為若是取預測效果不佳的結果，將無法看出預測結果在top-N% (1,5,10)間的差異。

透過table2看出，因為模型測試的scale差異不大，因此在於scale為200與1000時的結果幾乎沒有甚麼差異，而準確性的部分Top-1%都是最好的，顯示模型能夠準確地去預測Top-1%的節點，但是若是模型訓練結果較差時，就會導致後面的節點間差異不大，近乎沒有差異，因此無法正確地去排序後面的節點，使得Top-5%&Top-10%容易有迅速下降的趨勢。

在於Kendall tau distance的部分也可以看出，若是當top-N% (1,5,10) accuracy值皆高時，表示此模型的訓練效果較好，因此Kendall tau分數也會較高，表示能夠將整張圖的節點正確排序，但是當Top-1%分數相當高，而後續Top-5%&Top-10%分數卻下降快速時，也可以看出Kendall tau分數容易呈現負數，顯示排序不一致的節點

數量較多，也表示雖然對於排名前面的預測效果很好，卻對於整體的預測效果較差。

最後，也在測試中加入Networkit的papckage model中的KadabraBetweenness去與本次實作的DrBC做比較，從結果中可以輕易地看出，的確Kadabra在於模型穩定上好許多，且運算效能也是非常的好，但由於Kadabra主要是用於識別top N%的節點，因此在於Kendall tau分數時就能夠看到明顯的下降，反之DrBC model能夠獲得較高的分數。

4 Conclusion

在本次的實作中，實現了論文中提出的DrBc model，透過Encoder&Decoder的方式建構一個GNN model，主要包含一個neighborhood-aggregation encoder與一個multi-layer perceptron decoder，且能夠透過少量節點的訓練資料，推展至大型的真實世界網路，因此更加的能夠去因應變動的社群網絡系統。

而在實作中也發現了，在於模型實作前應該多加考量模型的架構，這樣才不容易在小細節上出問題，也能夠增加模型的穩定性，而非需要來回的多加修改，未來可以針對現有的模型去做延伸，尋找相關有趣的議題作深入研究。

5 Difficulties Encountered

1. 本次的實作任務中在於模型權重更新的設計上，全部都是透過手刻程式去撰寫，因此發現許多權重更新容易出現的問題。
   (a) 一開始不熟悉gradient儲存於變數上的使用方式，與在於W計算後is_leaf=False的問題，因此在於更新多次的權重運算時遇到問題，後續是靠著在於每一個W後面加retain_grad去解決。
   (b) pytorch變數的串接是透過公式將整個的計算圖串接起來，而一開始公式刻錯，沒有將某一變數算入，導致backward後的gradient出現None的情形，後續才發現若是有正確的運行頂多為Nan，而不會為None。
   (c) 因為權重矩陣並非由nn.Module的組成方式去組成parameter，因此沒有辦法導入現有的optimizer，因此為最簡單的權重更新方式，而導致不容易收斂，可能於之後要多加留意初始設計時的架構。
2. 模型設計時，一開始class的可調控參數上將number of node寫死了，後續才發現train與test 在於訓練規模上會不一致，應該要根據graph的大小在自動計算，因此花了一些時間做更動，感覺要先確定init的架構再做實作，而不是根據當下的部分做增減。
3. 在於Neighborhood Aggregation時，思考了許久要如何將全部的節點透過矩陣做平行運算，使得計算效能能夠提升，因為在於論文中是寫說單一節點的計算方式，在於參考source code後確實提供了許多的靈感，透過節點的相鄰矩陣能夠做出多種的變化。

4. 因為在於每一個episode中，皆會重新生成一個power-law的grpah，所以不同的graph生成會對模型的影響很大，造成模型不穩定性很高，可能會需要再更多的episode使得模型收斂。

5. 一開始在於測試時的loss下降速度非常的慢，因此對於模型做了多種的嘗試，例如在於ground truth BC上加入Log，或者是想辦法去加大計算時的predict值，但依舊效果不佳，卻又因為每次生成power-law的grpah不同，無法去確認到底是透過何種方式能夠確實去降低loss值，或是程式小細節尚有何種錯誤。

6. 因為每次重新生成一個power-law的grpah使得模型穩定度較差，因此嘗試著加入batch size，期望能夠一次生成多張的graph在將其組合後形成一張大張的grpah去做訓練，但反而使得模型的準確率下降，且訓練時間變成超級長，設想的可能因素是多張小的power-law grpah組織而成使得grpah & graph之間各自獨立，沒有連接的通道，導致資訊會遇到沒法傳遞的問題而準確性下降，且因為多張組織後變成不是一張power-law的grpah因此難以收斂，使得訓練時間變成很長。

References

DrBC github source code: https://github.com/FFrankyy/DrBC/blob/b8a84db280e941d27484e338cbfc736774f6a5a6/BetLearn.py.
GCN:Message Passing Networks https://pytorch-geometric.readthedocs.io/en/latest/tutorial/create_gnn.html
GRUCELL https://pytorch.org/docs/stable/generated/torch.nn.GRUCell.html
KadabraBetweenness： https://networkit.github.io/dev-docs/cpp_api/classNetworKit_1_1KadabraBetweenness.html
pytorch-geometric： https://pytorch-geometric.readthedocs.io/en/latest/modules/data.html
networkit.graph： https://networkit.github.io/dev-docs/python_api/graph.html
torch_geometric.utils： https://pytorch-geometric.readthedocs.io/en/latest/modules/utils.html
scipy.stats.kendalltau： https://docs.scipy.org/doc/scipy-0.15.1/reference/generated/scipy.stats.kendalltau.html
loss.backward(), optimizer.step()： https://blog.csdn.net/PanYHHH/article/details/107361827