NLP&AI 연구실 세미나 (07/07, Thu)

The Graph Neural Network Model

손수현 & 김진성

Introducing the GNN

* Previous approach for graph data
  - 그래프의 노드의 저차원 임베딩을 학습

  - 즉, 노드 표현 생성 시, **얕은 임베딩**(shallow embedding) 사용
    → 각 **노드 별 unique 임베딩**을 최적화

* Limitations (Ch.3 참고)
  - Parameters sharing 불가
    → Regularization 효과 보장 X

  - 노드 간 관계 정보의 배제
    → 1대1 mapping 으로 인한 한계

  - 학습 과정에 포함되지 않은 노드에 대해서는 임베딩을 구하지 못함.

Introducing the GNN

* GNN?
  - Graph Neural Network (GNN)
  - 보다 **더 복잡한 인코더 모델**을 설계하는 것에 집중.
  - **그래프 데이터에 대한 DNN**을 정의하기 위한 framework 의 필요성 존재.

* Key Idea of GNN
  1) 우리가 가지는 feature information 으로 노드에 대한 표현을 생성하되,
  2) **실제 그래프의 구조**에 의존(반영)해야함.

Introducing the GNN

* Challenges of GNN
  - **그래프 구조 데이터**를 위한 **복잡한 인코더 모델** 개발에의 **어려움** 존재.

  - 기존의 딥러닝 toolbox로는 그래프 데이터를 처리하는 것이 용이하지 않음.
    . CNN : grid 구조 입력(e.g., 이미지)에 대해서만 well-defined.
    . RNN : sequence 데이터(e.g., 텍스트)에 대해서만 well-defined.

    → **general graphs 에 대한 DNN**을 well-define 하기 위해, **새로운 종류**의 **딥러닝 아키텍쳐**를 정의함.

Introducing the GNN

* General architecture over graphs
  - DNN의 입력으로 **인접 행렬(adjacency matrix)** 사용.

  - 인접 행렬 A = [[~, ~, ~, ⋯, ~], → A[1]
    [~, ~, ~, ⋯, ~], → A[2]
    [~, ~, ~, ⋯, ~], → A[3]
    ⋯
    [~, ~, ~, ⋯, ~]] → A[|V|]

    Vector concat.

  - 전체 그래프의 임베딩 z_G 생성 ?
    : 인접 행렬을 flatten → 이 결과를 MLP에 feed 하여 산출

$$\mathbf{z}_{\mathcal{G}} = \mathbf{MLP}(\mathbf{A}[1] \oplus \mathbf{A}[2] \oplus ... \oplus \mathbf{A}[|\mathcal{V}|]),$$

* **문제점?** 인접 행렬에 사용된 arbitrary ordering 에 의존적임.
  → **not permutation invariant** (= 순열이 가변적임.)

The New Deep Learning Architecture

* Important features for the new architecture
  - permutation invariance and equivariance : 순열의 불변성 및 등변성

    → 불변하지는 못하더라도, 최소한 등변 해야함.

  - 즉, 인접 행렬 A 를 입력으로 받는 어떤 함수 f 는 아래 두 가지 조건 중 한 가지는 만족 해야함.
    → 둘 중 하나 이상의 특성을 만족하는 모델을 구성해야함.

$$
\begin{aligned}
f(\mathbf{PAP}^\top) &= f(\mathbf{A}) \qquad (\text{Permutation Invariance}) \\
f(\mathbf{PAP}^\top) &= \mathbf{P}f(\mathbf{A}) \qquad (\text{Permutation Equivariance})
\end{aligned}
$$

  - Invariance vs Equivariance
    . Invariance : 함수 f 가 인접 행렬의 행/열에 대한 임의적 정렬에 의존적이 않음.
    . Equivariance : 인접 행렬이 치환(permute)될 때, 함수 f의 output이 일관적으로(in a consistent way) 치환됨.

    ex) Ch.3에 등장했던 shallow encoders 가 permutation equivariant 함수의 예시.

The New Deep Learning Architecture

* Important features for the new architecture over general graphs

$$f(\mathbf{PAP}^\top) = f(\mathbf{A}) \quad \text{(Permutation Invariance)}$$
$$f(\mathbf{PAP}^\top) = \mathbf{P}f(\mathbf{A}) \quad \text{(Permutation Equivariance)}$$

- Permutation matrix(치환 행렬) P
  . 순서가 부여된 임의의 행렬을 **의도된 다른 순서로** 뒤섞는(행/열 교환을 수행하는) 행렬
  . 일반적으로 단위 행렬 I에서 각 행을 서로 교환한 조합으로 얻음.

$$I \quad P_{12} \quad P_{13} \quad P_{23}$$
$$\begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix}\begin{bmatrix}0&1&0\\1&0&0\\0&0&1\end{bmatrix}\begin{bmatrix}0&0&1\\0&1&0\\1&0&0\end{bmatrix}\begin{bmatrix}1&0&0\\0&0&1\\0&1&0\end{bmatrix}$$

ex) A = [[a b c]
[d e f]
[g h I]] 에 대해,

P_13A = [[g h I]
[d e f]
[a b c]]

AP_13 = [[c b a]
[f e d]
[i h g]]

The New Deep Learning Architecture

*** 본 챕터의 관심사**

- How to take **입력 그래프 G = (V, E)** along with a **set of node features X,**
  → 결국, 어떻게 **X** 로부터 **노드 임베딩** z_u 를 생성하는지.

- GNN framework 가 subgraphs & entire graphs 를 생성하기 위해 어떻게 사용되는지.

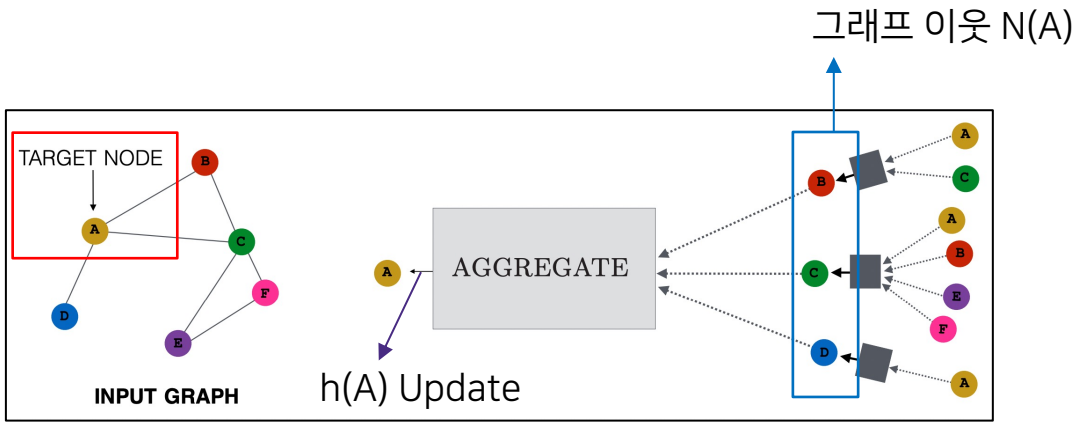
→ 본 챕터에서는 **message passing** 에 대해서 설명하고,
GNN 모델을 실제로 어떻게 훈련 및 최적화 하는지는 Ch.6 에서 설명.

Neural Message Passing (1): Overview

* Neural message passing?
  - GNN 구조의 큰 특징 중 하나.
  - 벡터 메시지들이 **노드들 간 교환**되고, **NN 을 통해 update** 되는 것.

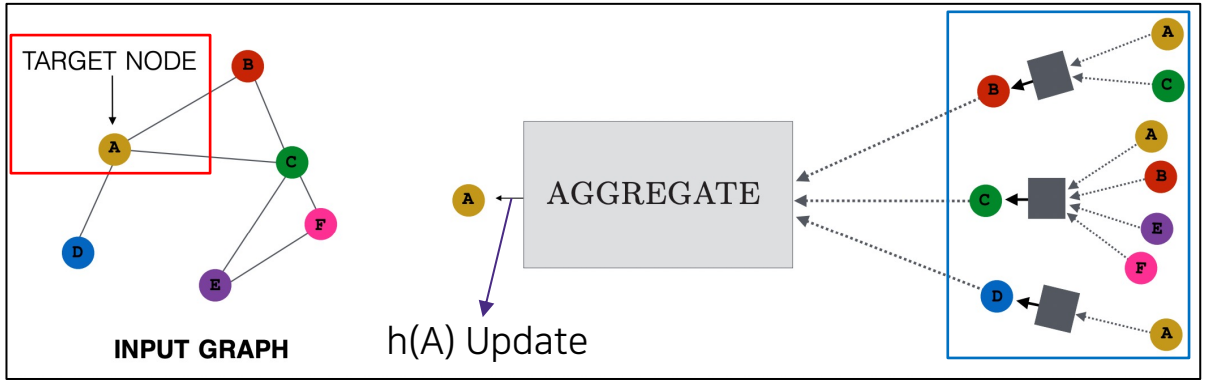
  - GNN의 각 message passing iteration 동안, 각 노드 **u** 에 상응하는 hidden embedding h_{u^k} 는 **u** 의 그래프 이웃 **N(u) 에 의해 합쳐진 정보에 의해 update** 됨.

    cf) message passing의 각 iteration == GNN의 각 layer 라고 이해.



Neural Message Passing (1): Overview

* Update a node u from its local neighborhood N(u)



- 모델이 노드 A의 local graph neighbors (B, C, D)로부터 메시지들을 aggregate.
- 이 메시지들은 다시, A의 이웃들(B, C, D)의 각 이웃들인 (A,C), (A,B,E,F), A의 정보에 기반하며, 이러한 과정의 연속.

→ 결국, GNN의 연산 그래프는, 타겟 노드 주위의 이웃들을 펼쳐놓은 트리 구조를 형성함.

Neural Message Passing (1): Overview

* Neural message passing?
  - 위 과정을 수식으로 나타내면 아래와 같음.

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

  - m_N(u)^k : 노드 u의 이웃인 N(u)로부터 모인 메시지
  - k : GNN의 k번째 iteration
  - AGGREGATE() : 매 iteration 마다, N(u)의 임베딩 셋을 입력으로 받아서, 메시지 m_N(u)^k를 생성함.
  - UPDATE() : 메시지 m_N(u)^k 을 노드 u의 이전 임베딩 h_u^(k-1)와 결합하여,
    업데이트된 임베딩 h_u^k를 생성.

Neural Message Passing (1): Overview

* Neural message passing?
  - k=0 일 때의 초기 임베딩은 모든 노드들에 대한 input feature 로 세팅.
    (i.e., $h_{u^0} = x_u$)
    → 즉, shallow embedding 과 달리,
    GNN framework은 노드 자질 x_u 를 모델 입력으로 받음.

  - K번의 GNN message passing iteration 후에, **마지막 layer 의 output인 각 노드 u의 임베딩 z_u** 를 얻음.

$$\mathbf{z}_u = \mathbf{h}_u^{(K)}, \forall u \in \mathcal{V}.$$

cf) 노드 자질 x_u 가 not available 한 경우에만,
  1) node statistics 사용(Ch.2.1 참고)
  2) 각 노드를 one-hot feature 로 identity features 로 만듦(각 노드별 unique 임베딩).
    → unseen nodes 에 대한 generalization 능력 저하.

Neural Message Passing (2): Motivation

* Intuition (GNN message passing framework)
  - 각 passing iteration (GNN layer)이 반복됨에 따라,
    그래프의 모든 노드 u는 그 local neighborhood N(u)로부터 정보를 aggregate 함.

    → 각 노드 임베딩은 그래프 내의 **더 먼 reaches** 로부터 **점점 더 많은 정보**를 획득하게 됨.

  - 즉, iteration k = 1일 때, 1-hop 거리에 있는 N(u) 의 정보를 모으고,
    k = 2 일 때, 2-hop 거리에 있는 이웃의 정보를 모으고 …(반복)…
    k일 때, k-hop 거리 이웃의 정보까지 획득.

Neural Message Passing (2): Motivation

* What actually encode the node embeddings
  - message passing iteration 이 진행됨에 따라, 각 노드 u의 임베딩 h(u) 는 두 가지 유형의 정보를 얻게 됨.

    1) 구조적 정보 (structural)
      . k번째 iteration 에서 노드 u 의 임베딩 $\boldsymbol{h}_{\boldsymbol{u}(\boldsymbol{k})}$ 는 u 의 k-hop 이웃들의
      모든 노드의 **차수(degrees) 정보**를 인코딩함.
      → e.g., molecule 그래프 분석 시, 차수 정보로 molecule 유형 구분에 사용됨.

    2) 자질 기반 정보 (feature-based)
      . $\boldsymbol{h}_{\boldsymbol{u}(\boldsymbol{k})}$ 는 노드 u 의 **k-hop 이웃 노드들**의 **모든 자질 정보**들을 인코딩함.
      → 마치 CNN이 spatially-defined patches in an image 로부터 자질 정보를 모으듯이,
      GNN은 local graph neighborhoods 로부터 정보를 모음.

Neural Message Passing (3): The Basic GNN

*** 추상적인 GNN framework 의 구체화**

- 그래서 이제, UPDATE, AGGREGATE 함수에 대한 구체적 설계 필요.
- 2005년에 최초 제안된 original GNN 모델에 대한 단순화한 가장 기본적인 GNN 모델에서 시작.

$$\mathbf{h}_u^{(k)} = \sigma\left(\mathbf{W}_{\text{self}}^{(k)}\mathbf{h}_u^{(k-1)} + \mathbf{W}_{\text{neigh}}^{(k)}\sum_{v\in\mathcal{N}(u)}\mathbf{h}_v^{(k-1)} + \mathbf{b}^{(k)}\right)$$

- W_{self}, W_{neigh} : trainable parameter matrix
- $\boldsymbol{\sigma}$: non-linear layer (e.g., tanh, ReLU)
- **b** : bias term

1) neighbors 로부터의 메시지들을 sum
2) 노드의 이전 임베딩과 neighborhood 정보를 결합(더해줌)
3) 비선형 layer에 적용

cf) basic MLP 와 유사 → single non-linearity 에 의존.

Neural Message Passing (3): The Basic GNN

* The basic GNN through UPDATE, AGGREGATE functions
  - 추상적 concept

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

$$\mathbf{m}_{\mathcal{N}(u)} = \text{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\})$$

  - 구체화

$$\mathbf{h}_u^{(k)} = \sigma\left(\mathbf{W}_{\text{self}}^{(k)}\mathbf{h}_u^{(k-1)} + \mathbf{W}_{\text{neigh}}^{(k)}\sum_{v \in \mathcal{N}(u)}\mathbf{h}_v^{(k-1)} + \mathbf{b}^{(k)}\right)$$

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{v \in \mathcal{N}(u)} \mathbf{h}_v,$$
$$\text{UPDATE}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) = \sigma\left(\mathbf{W}_{\text{self}}\mathbf{h}_u + \mathbf{W}_{\text{neigh}}\mathbf{m}_{\mathcal{N}(u)}\right),$$

Neural Message Passing (3): The Basic GNN

* Node vs Graph-level equations
  - 앞서 표현한 basic GNN 식은 **node-level** 의 message-passing 조작에 대한 정의임.

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{v \in \mathcal{N}(u)} \mathbf{h}_v,$$

$$\text{UPDATE}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) = \sigma\left(\mathbf{W}_{\text{self}}\mathbf{h}_u + \mathbf{W}_{\text{neigh}}\mathbf{m}_{\mathcal{N}(u)}\right)$$

$$\mathbf{m}_{\mathcal{N}(u)} = \text{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\})$$

→ 그렇다면, 이를 **graph-level** 의 정의로 변환한다면?

$$\mathbf{H}^{(t)} = \sigma\left(\mathbf{A}\mathbf{H}^{(k-1)}\mathbf{W}_{\text{neigh}}^{(k)} + \mathbf{H}^{(k-1)}\mathbf{W}_{\text{self}}^{(k)}\right)$$

- $\boldsymbol{H}^{(k)}$: t-th GNN layer 에서의 node representation matrix
  → 각 **행이 하나의 노드**에 대응.
- A : 그래프 인접 행렬
  → 이웃 노드들의 정보에 대응한다고 이해하면 됨.

Neural Message Passing (4): Message Passing with Self-loops

* Adding self-loops
  - neural message passing 기법의 단순화를 위한 일반적인 방법.
    . to add self-loops to the input graph
    . to omit the explicit update step

$$\mathbf{h}_u^{(k)} = \text{AGGREGATE}(\{\mathbf{h}_v^{(k-1)}, \forall v \in \mathcal{N}(u) \cup \{u\}\}),$$

  - AGGREGATE 함수가 N(u)와 {u}의 합집합 set을 받음.
    : 노드 u의 이웃 N(u)과 더불어 노드 그 자신도 포함.

* 장단점
  - 명시적인 update 함수를 더 이상 정의할 필요가 없으며,
    → aggregation 기법을 통해 implicitly 정의된다고 할 수 있음.
  - 이러한 방식으로 단순화된 message passing은 overfitting을 완화함.

  - 하지만, GNN의 expressivity를 심각하게 제한하기도 함.
    → 이웃 노드들로부터 오는 정보를 노드 자신의 정보로부터 식별할 수가 없기 때문.

Neural Message Passing (4): Message Passing with Self-loops

* self-loops 추가하는 것의 의미
  - Wself와 Wneigh 행렬 간의 parameters sharing과 동일함.

$$\mathbf{h}_u^{(k)} = \text{AGGREGATE}(\{\mathbf{h}_v^{(k-1)}, \forall v \in \mathcal{N}(u) \cup \{u\}\}),$$

  - 이를 graph-level 업데이트로 표현하면,

$$\mathbf{H}^{(t)} = \sigma\left(\mathbf{A}\mathbf{H}^{(k-1)}\mathbf{W}_{\text{neigh}}^{(k)} + \mathbf{H}^{(k-1)}\mathbf{W}_{\text{self}}^{(k)}\right)$$

$$\mathbf{H}^{(t)} = \sigma\left((\mathbf{A}+\mathbf{I})\mathbf{H}^{(t-1)}\mathbf{W}^{(t)}\right)$$

Generalized Neighborhood Aggregation

* Generalized Neighborhood Aggregation
  - 앞서 소개한 아래의 basic GNN 모델은 강력한 성능을 내는 것은 맞지만,
    단순한 MLP와 마찬가지로 여러 가지 방법을 통해 개선/일반화 가능함.

$$\mathbf{h}_u^{(k)} = \sigma\left(\mathbf{W}_{\text{self}}^{(k)}\mathbf{h}_u^{(k-1)} + \mathbf{W}_{\text{neigh}}^{(k)} \sum_{v \in \mathcal{N}(u)} \mathbf{h}_v^{(k-1)} + \mathbf{b}^{(k)}\right)$$

  - 특히, AGGREGATE operator와 UPDATE operator를 어떻게 개선/일반화 하는가에 초점.
    → 먼저, aggregation에 대해 다룬 후, update를 다룸.

Generalized Neighborhood Aggregation (1): Neighborhood Normalization

* Neighborhood aggregation operation
  - 가장 단순한 구조는, the neighbor embeddings의 sum만 취하는 것.
    → 이러한 구조의 문제점은, 노드의 차수(node degrees)에 매우 민감하고, 불안정하다는 것.
    ex) 노드 u가 다른 어떤 노드 u' 보다 100배는 많은 이웃을 가진다면,
    이렇게 큰 규모의 차이는 numerical instabilities, 최적화의 어려움을 초래 가능.

  - 위 문제에 대한 해결책 : 노드의 차수에 기반한 aggregate operation을 normalize 하면 됨.
    . 가장 쉬운 방법 : 이웃 임베딩들의 sum 대신에 average를 취하는 방법.

$$\mathbf{m}_{\mathcal{N}(u)} = \frac{\sum_{v \in \mathcal{N}(u)} \mathbf{h}_v}{|\mathcal{N}(u)|},$$

    . Symmetric normalization : spectral graph 이론에 영감을 받은,
    symmetric-normalized aggregation을 적용함.

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{v \in \mathcal{N}(u)} \frac{\mathbf{h}_v}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}}.$$

Generalized Neighborhood Aggregation (1): Neighborhood Normalization

* Graph convolutional networks (GCNs)
  - GNN 모델 중 가장 유명한 베이스라인 모델
  - 앞서 언급한 symmetric-normalized aggregation과 self-loop update approach를 둘 다 사용함.
  - 따라서, message passing 함수를 다음과 같이 정의 가능함.

$$\mathbf{h}_u^{(k)} = \sigma\left(\mathbf{W}^{(k)} \sum_{v \in \mathcal{N}(u) \cup \{u\}} \frac{\mathbf{h}_v}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}}\right).$$

Generalized Neighborhood Aggregation (2): Set Aggregators

* Normalization 외의 AGGREGATE 개선 방안?
  - neighborhood aggregation operation은 근본적으로 set 함수임.
    : 즉, a set of neighbor embeddings $\{h_v\}$를 취하고, 이 set을 single vector $m_{N(u)}$로 맵핑(축소)함.
  - set이라는 것이 왜 중요한가?
    . Set 구조의 경우, 노드의 이웃들의 자연스러운 순서(natural ordering)라는 것이 없음.
    → 따라서, 우리가 정의하는 aggregation 함수는 반드시 permutation invariant 함.

* Set pooling
  - permutation invariant NN 이론에 근거한 aggregation 함수를 정의하는 접근법 중 하나.
  ex) universal set function approximator

$$\mathbf{m}_{\mathcal{N}(u)} = \mathbf{MLP}_\theta\left(\sum_{v \in N(u)} \mathbf{MLP}_\phi(\mathbf{h}_v)\right)$$

Generalized Neighborhood Aggregation (2): Set Aggregators

* Set poolingType equation here.
  - 성능의 소폭 상승을 보여주지만,
  - 사용된 MLP 레이어의 깊이에 의존적이어서, overfitting의 위험성이 증가함.
    → 따라서, 주로 해당 방법 사용 시 하나의 hidden layer만 가진 MLPs를 사용하는 것이 일반적.

* Janossy pooling
  - 위 Set pooling 기법은 기존의 permutation invariant한 basic neighborhood aggregation에 추가적인 MLP 레이어를 적용한 구조
  - 이와 반대로, permutation sensitive function을 사용하고, many possible permutations에 대한 결과값들의 평균을 구함.
  - 즉, 주어진 임베딩 셋을 특정한 순서 $(h_{v1}, \ldots, h_{vN(u)})$로 나열한 후, 아래와 같은 식을 적용함.

$$\mathbf{m}_{\mathcal{N}(u)} = \mathbf{MLP}_\theta \left(\frac{1}{|\Pi|} \sum_{\pi \in \Pi} \rho_\phi \left(\mathbf{h}_{v_1}, \mathbf{h}_{v_2}, \ldots, \mathbf{h}_{v_{|\mathcal{N}(u)|}} \right)_{\pi_i} \right),$$

  - π : 순열의 셋 (a set of permutations)
  - ρ_ϕ : permutation-sensitive 함수 (e.g., NN that operates on sequences, 주로 LSTM)

Generalized Neighborhood Aggregation (2): Set Aggregators

* Janossy pooling

$$\mathbf{m}_{\mathcal{N}(u)} = \mathbf{MLP}_\theta \left(\frac{1}{|\Pi|} \sum_{\pi \in \Pi} \rho_\phi \left(\mathbf{h}_{v_1}, \mathbf{h}_{v_2}, ..., \mathbf{h}_{v_{|\mathcal{N}(u)|}} \right)_{\pi_i} \right),$$

- π가 가능한 모든 permutations의 set이라면, set pooling의 식과 같이 set 데이터를 위한 universal 함수이겠으나, 모든 가능한 순열에 대해 더하는 것은 일반적으로 intractable.
  → 따라서, 다음 두 가지 방법을 통해 접근함.

  1) To sample a random subset : 가능한 순열들 간, 뽑힌 무작위의 subset에 대해서만 sum.
  2) 이웃 노드들 간에 canonical 정렬 사용 : e.g., 차수에 따른 내림차순 정렬 등.

Generalized Neighborhood Aggregation (3): Neighborhood Attention

* To apply attention → aggregation layer
  - 각 이웃 노드 마다 attention weight(importance)를 assign.
  ex) GAT (Graph Attention Network) 모델
  . 이웃 노드들의 가중합을 하기 위한 각 노드 마다의 attention weights를 정의하여 부여

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{v \in \mathcal{N}(u)} \alpha_{u,v} \mathbf{h}_v,$$

. $\alpha_{u,v}$: 이웃 노드 $v \in N(u)$에 대한 attention weight

$$\alpha_{u,v} = \frac{\exp\left(\mathbf{a}^\top [\mathbf{W}\mathbf{h}_u \oplus \mathbf{W}\mathbf{h}_v]\right)}{\sum_{v' \in \mathcal{N}(u)} \exp\left(\mathbf{a}^\top [\mathbf{W}\mathbf{h}_u \oplus \mathbf{W}\mathbf{h}_{v'}]\right)},$$

. a : trainable attention vector
. W : trainable matrix

→ Work well with graph data.

Generalized Update Methods

* The basic update methods
  - update operation : 노드의 현재 임베딩을 그 이웃들의 메시지와 linear combination 하는 과정

* Generalized UPDATE 기법의 필요성
  - Over-smoothing 문제
    : GNN message passing의 여러 iteration 과정을 거치고 나면, 각 노드 간의 표현들이 서로 매우 유사해지는 경향이 있음.
    → 특히, basic GNN 모델과 self-loop update를 사용한 모델에서 흔히 발생.

  - Over-smoothing은 더 깊은 GNN 모델을 만들 수 없게 만들기 때문에 문제가 됨.
    : over-smoothed embeddings를 생성.

Generalized Update Methods (1): Concatenation and Skip-connections

* Over-smoothing의 원인
  - over-smoothing 문제는 GNN 구조의 message passing 과정에서 핵심적인 이슈임.
  - $N(u)$로부터 모인 정보가, 새로 업데이트된 노드 표현 $h_u^{(k+1)}$을 dominate 할 때 발생.
    → 즉, $m_{N(u)}$에 $h_u^{(k+1)}$ 이 너무 강하게 의존하여, 이전 노드 표현인 h_u^k 를 소실하는 경우.

* Vector concatenation or skip-connections
  - update 단계 동안에 이전 노드 임베딩 정보를 보존하는 방법.
  cf) skip connection? deep architectures에서 하나의 layer의 output을 몇 개의 layer를 건너뛰고, 다음 layer의 input에 추가하는 것
  - 가장 단순한 Vector concatenation update : 노드 레벨 정보를 보존하기 위해 concatenation 사용.

$$\text{UPDATE}_{\text{concat}}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) = [\text{UPDATE}_{\text{base}}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) \oplus \mathbf{h}_u],$$

  : 기존 update 함수의 output을, 다시 한번 노드의 이전 레이어 표현과 concat. 해줌.
  → intuition : "이웃 노드들로부터 온 정보($m_{N(u)}$)를 현재 각 노드의 표현 h_u 로부터 구별하겠다."

Generalized Update Methods (1): Concatenation and Skip-connections

* linear interpolation
  - concatenation을 사용한 방법 외에도 linear interpolation 기법을 사용하여 skip-connections 활용 가능.

$$\text{UPDATE}_{\text{interpolate}}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) = \boldsymbol{\alpha}_1 \circ \text{UPDATE}_{\text{base}}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) + \boldsymbol{\alpha}_2 \odot \mathbf{h}_u,$$

  - α_1, α_2 : gating vectors (α_2 = 1- α_1)
  - o : multiplication
  - final updated representation : 이전 representation h_u 와 이웃 정보 $m_{N(u)}$ 로 update 된 representation 간의 linear interpolation

  - parameter α_1의 경우, 모델과 함께 jointly 다양한 방법으로 학습 가능함.
    e.g., 별도의 single-layer GNN의 아웃풋을 α_1 으로 사용. (현재 hidden-layer 표현을 입력 자질로 넣어서)
  - 혹은 더 간단하게는, 직접적으로 각 message passing 레이어를 통해 학습 시키거나,
    현재 노드 표현에서 MLP를 사용해서 학습 시킬 수도 있음.

Generalized Update Methods (2): Gated Updates

* GNN message passing 알고리즘을 보는 또 다른 관점
  - "aggregation 함수는 neighbors로부터 observation을 받고 있는 것이며, 이는 각 노드의 hidden state의 update에 사용된다."고 보면,
  - observations에 기반하여 RNN 구조의 hidden state를 update 하는데 쓰인 기법을 가져다 쓸 수 있음. e.g., 가장 초기 GNN 아키텍쳐 중 하나는 update 함수를 다음과 같이 정의함.

$$\mathbf{h}_u^{(k)} = \mathrm{GRU}(\mathbf{h}_u^{(k-1)}, \mathbf{m}_{\mathcal{N}(u)}^{(k)}),$$

  - GRU : gated recurrent unit cell의 update 방정식.

* GNN <-> RNN
  - 위와 같이, 일반적으로 RNNs을 위해 정의된 대부분의 update 함수들은 GNN의 맥락에서도 사용 가능함.
  - RNN의 update 함수의 hidden state 논항 h^t 를 노드의 hidden state h_u 로 대치하고, observation vector x^t 를 local neighborhood로부터 모은 message $m_{N_{(u)}}$ 로 대치하여 사용.

  - 이러한 gated update는 깊은 GNN 아키텍쳐(10 레이어 이상)를 효과적으로 촉진하며, Over-smoothing을 방지 가능.

Generalized Update Methods (3): Jumping Knowledge Connections

* Use of the final layer node embeddings
  - 많은 GNN의 approches가 우리가 GNN의 최종 계층의 output을 사용한다고 가정함.
  - 즉, 우리가 다운스트림 태스크에 사용하는 노드 표현 z_u가, 다음과 같이
    GNN의 최종 계층 노드 임베딩 h_u와 같다는 것. → 앞서 언급한 over-smoothing과 같은 문제 발생 가능.

$$\mathbf{z}_u = \mathbf{h}_u^{(K)}, \forall u \in \mathcal{V}.$$

  - 이를 보완하기 위해, skip-connection, gated updates 등의 방법이 나왔음.

* Leveraging the representations at each layer of message passing
  - 또 다른 최종 노드 표현의 품질 향상 방법 : 최종 계층 output만 사용하는게 아니라,
    거기에다 each layer of message passing에서의 표현들까지 활용하자!

$$\mathbf{z}_u = f_{\text{JK}}(\mathbf{h}_u^{(0)} \oplus \mathbf{h}_u^{(1)} \oplus \ldots \oplus \mathbf{h}_u^{(K)}),$$

  - f_{JK} : 임의의 differentiable 함수
    → 많은 경우 identity 함수로 정의. (각 GNN 레이어로부터의 노드 임베딩을 단순히 concat.만 하겠다는 의미.)
    → 또한, max-pooling 이나, LSTM attentions layers를 사용하기도 함.

Edge Features and Multi-relational GNNs (1): Relational Graph Neural Networks

* So far…
  - 지금까지는 simple graphs를 다룬다고 가정했음.
  - multi-relational한 그래프나, knowledge graphs와 같은 더 복잡한 데이터는 그러면 어떻게?

* Relational Graph Neural Networks
  - Relational Graph Convolutional Network (RGCN)
    . 복수의 관계 유형을 다루기 위해, aggregation function을 augment
    . 관계 유형 마다, 별도의 transformation 행렬을 specify 해놓음.

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{\tau \in \mathcal{R}} \sum_{v \in \mathcal{N}_\tau(u)} \frac{\mathbf{W}_\tau \mathbf{h}_v}{f_n(\mathcal{N}(u), \mathcal{N}(v))},$$

. f_n : normalization function
→ normalization이 가미된 basic GNN과 유사하지만, 다른 edge 유형에 대해서는 별도로 정보를 모은다는 점에서 다름. (노드 u와 노드 v에 대해서)

Edge Features and Multi-relational GNNs (1): Relational Graph Neural Networks

* Relational Graph Neural Networks (2): parameter sharing
  - RGCN의 단점은, relation type 당 하나의 trainable matrix를 가지므로,
    파라미터 수가 급격하게 증가한다는 것.
    → 이는 overfitting, 느린 학습을 초래 가능.
  - 이를 해결하기 위해, basis matrices를 통한 파라미터 공유가 제안됨.

$$\mathbf{W}_\tau = \sum_{i=1}^{b} \alpha_{i,\tau} \mathbf{B}_i.$$

  - 모든 relation matrices는 b개의 basis matrices $B_1, \ldots, Bb$로 정의하고,

Edge Features and Multi-relational GNNs (2): Attention and Feature Concatenation

* For mor general edge features
  - 각 관계마다 별도의 aggregation parameter를 정의하는 relational GNN approach는, multi-relational graphs와 edge features가 discrete 하는 경우에 쓰임.
  - 그렇다면 edge features가 더 일반적인 형태인 경우에는?
    → 이 정보들(features)을 message passing 동안 neighbor embeddings와 concat. (혹은 attention 사용)

$$\mathbf{m}_{\mathcal{N}(u)} = \text{AGGREGATE}_{\text{base}}(\{\mathbf{h}_v \oplus \mathbf{e}_{(u,\tau,v)}, \forall v \in \mathcal{N}(u)\}),$$

  - $e_{(u,\tau,v)}$: edge (u, τ, v)에 대한 임의의 vector-valued feature

Graph Pooling (1)

* Make predictions at the graph-level
  - neural message passing approach는 a set of node embeddings를 산출함.
    → 그럼 우리가 graph-level에서 predictions 하고 싶다면?
  - 즉, 지금까지 우리 목표는 node representations z_u를 학습하는 것이었다면,
    entire graph G의 embedding z_G를 학습하는 것을 **graph pooling** 이라고 함.
    → 전체 그래프의 임베딩 학습을 위해 node embeddings를 pool together 하기 때문.

* Set pooling approaches (1)
  - aggregate operator와 유사하게, graph pooling은 sets data에 대한 학습 문제라고 할 수 있음.
  - 즉, a set of node embeddings $\{z1, ..., z_{|V|}\}$를 z_G로 맵핑하는 pooling function f_p가 필요함.
  - set pooling 기법을 이용한 두 가지 approach.
    1) node embeddings에 대해 sum 혹은 mean을 취해서 생성.

$$\mathbf{z}_{\mathcal{G}} = \frac{\sum_{v \in \mathcal{V}} \mathbf{z}_u}{f_n(|\mathcal{V}|)},$$

. f_n은 identity function과 같은 normalizing 함수.

Graph Pooling

* Set pooling approaches (2)
  2) LSTMs와 attention을 결합하는 방법

$$\mathbf{q}_t = \mathrm{LSTM}(\mathbf{o}_{t-1}, \mathbf{q}_{t-1}), \rightarrow \text{ㄱ})$$
$$e_{v,t} = f_a(\mathbf{z}_v, \mathbf{q}_t), \forall v \in \mathcal{V}, \rightarrow \text{ㄴ})$$
$$a_{v,t} = \frac{\exp(e_{v,i})}{\sum_{u \in \mathcal{V}} \exp(e_{u,t})}, \forall v \in \mathcal{V}, \rightarrow \text{ㄷ})$$
$$\mathbf{o}_t = \sum_{v \in \mathcal{V}} a_{v,t} \mathbf{z}_v. \rightarrow \text{ㄹ})$$

$$\mathbf{z}_{\mathcal{G}} = \mathbf{o}_1 \oplus \mathbf{o}_2 \oplus \ldots \oplus \mathbf{o}_T.$$

- T번의 iteration 후에, t개의 모든 o_t 값을 더하여, 전체 그래프의 임베딩을 계산함.

- t = 1, ⋯,T 번의 step을 iteration 하면서, 위 식과 같은 일련의 attention-based aggregation을 수행함.
- q_t : 각 iteration t에서의 attention을 위한 Query vector
  → ㄴ)에서 각 노드 임베딩 z_v에 대한 attention score를 구할 때, attention 함수 f_a (e.g., dot product)에 함께 쓰임.
- ㄷ) : attention score normalization
- ㄹ) : attention weight $a_{v,t}$와 node embeddings의 가중합
- ㄱ) : ㄹ)에서 산출된 값을 LSTM을 이용하여 ㄱ)의 다음 query vector q_t를 update 하는 데에 사용.

Generalized Message Passing

* Generalize to leverage edge and graph-level information (1)
  - 지금까지는 node-level에서 다루는 GNN message passing에 대한 것.
  - message passing 각 단계에서 edge와 graph-level의 정보를 이용하여, 일반화 가능
  - message passing 동안, 그래프의 각 edge에 대한 hidden embeddings $h_{(u,v)}$와 전체 그래프에 대한 임베딩 h_g를 생성함.
    → 이는 모델이 edge와 graph-level features를 통합하기 용이하게 함.
    → basic GNN 모델과 비교하여, logical expressiveness가 뛰어나다는 연구 결과 있음.

$$\mathbf{h}_{(u,v)}^{(k)} = \text{UPDATE}_{\text{edge}}(\mathbf{h}_{(u,v)}^{(k-1)}, \mathbf{h}_u^{(k-1)}, \mathbf{h}_v^{(k-1)}, \mathbf{h}_{\mathcal{G}}^{(k-1)}) \quad \rightarrow 1)$$

$$\mathbf{m}_{\mathcal{N}(u)} = \text{AGGREGATE}_{\text{node}}(\{\mathbf{h}_{(u,v)}^{(k)} \forall v \in \mathcal{N}(u)\})$$

$$\mathbf{h}_u^{(k)} = \text{UPDATE}_{\text{node}}(\mathbf{h}_u^{(k-1)}, \mathbf{m}_{\mathcal{N}(u)}, \mathbf{h}_{\mathcal{G}}^{(k-1)}) \quad \rightarrow 2)$$

$$\mathbf{h}_{\mathcal{G}}^{(k)} = \text{UPDATE}_{\text{graph}}(\mathbf{h}_{\mathcal{G}}^{(k-1)}, \{\mathbf{h}_u^{(k)}, \forall u \in \mathcal{V}\}, \{\mathbf{h}_{(u,v)}^{(k)} \forall (u,v) \in \mathcal{E}\}). \rightarrow 3)$$

1) 먼저, edge embeddings를 부속 노드들(incident nodes)의 임베딩에 기반하여 update함.
2) 모든 incident edges의 edge embeddings를 aggregate 하여 node embeddings를 update함.
3) entire graph embedding은 node and edge representations 둘 다의 update에 사용되며, 그 자신은 각 iteration의 마지막에 모든 node and edge embeddings를 aggregate 하여 update 됨.

- GCN
- GraphSAGE
- GAT (Graph Attention Network)
- GTN (Graph Transformer Networks)
- RGCN (Relational GCN)
- GGCN (Gated GCN)
- SGConv (Simplifying GCN)

Thank you