Pointer Graph Networks

Petar Velickovic, Lars Buesing, Matthew C. Overlan, Razvan Pascanu, Oriol Vinyals, Charles Blundell

Presenter: Arshdeep Sekhon
https://qdata.github.io/deep2Read

Motivation

- (1) Solving classical graph algorithms known to be hard for GNNs
- (2) Graph specification for GNNs:
  - prespecified: we can use fully connected but doesnt work for large p
  - hand designed: can be error or bias prone
- Solution: Learning data driven graph:
  - scalability 2^p graphs
  - error term because of message or wrong graph

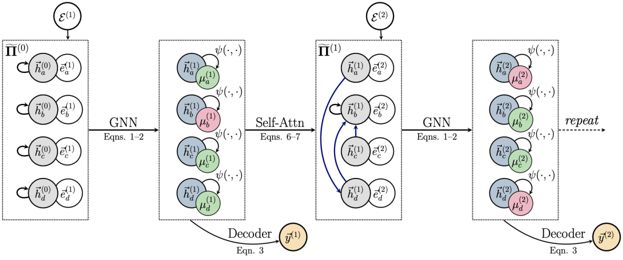
PGNs vs Related work

- (1) Solving classical graph algorithms known to be hard for GNNs
  - more difficult algorithms compared to previous work
  - out of distribution generalization
- (2) Graph specification for GNNs:
  - supervision from known graphs
  - scalable
  - node masking to encourage sparsity instead of ℓ_1 regularization
  - Use both $A_{partially\ known} + A_{learnt}$

PGN Key Contributions

- latent graph inference
- for classical graph algorithms
- out of distribution generalization

PGN Architecture



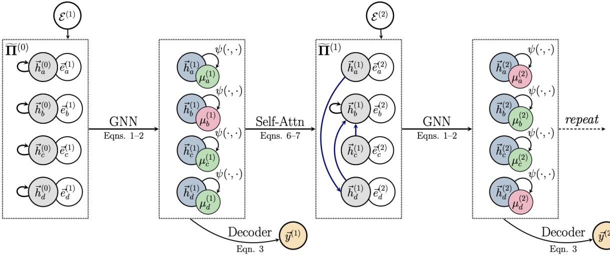
- Assume an underlying set of n entities.
- Given are sequences of inputs $\mathcal{E}^{(1)}, \mathcal{E}^{(2)}, \ldots$ where $\mathcal{E}^{(t)} = (\vec{e}_1^{(t)}, \vec{e}_2^{(t)}, \ldots, \vec{e}_n^{(t)})$ for $t \geq 1$ is defined by a list of feature vectors $\vec{e}_i^{(t)} \in \mathbb{R}^m$ for every entity $i \in \{1, \ldots, n\}$.
- Sequential Prediction Task: predicting target outputs $\vec{y}^{(t)} \in \mathbb{R}^l$ based on operation sequence $\mathcal{E}^{(1)}, \ldots, \mathcal{E}^{(t)}$ up to t.

Example Task: Dynamic Graph Connectivity

Dynamic Graph Connectivity: Are two vertices connected?

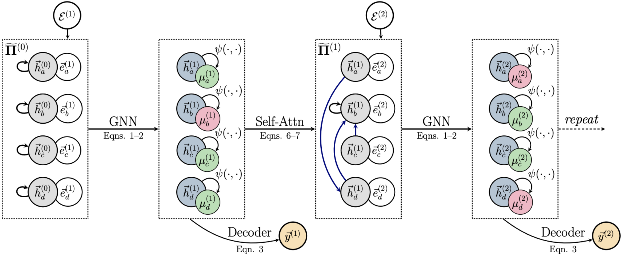
- inputs/ operations $\vec{e}_i^{(t)}$
- outputs $\vec{y}^{(t)}$: binary indicators of whether pairs of vertices are connected.

PGN Architecture



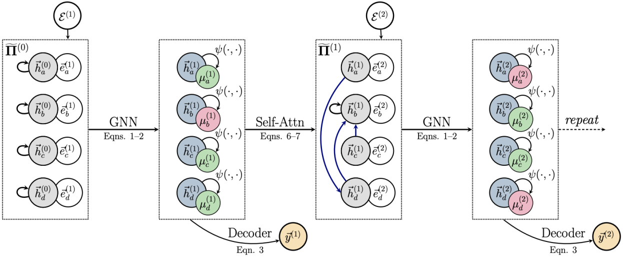
- Sequential prediction task: history of operations for all entities
- defined on unordered set of entities: permutation invariant!

PGN Architecture: Three Parts



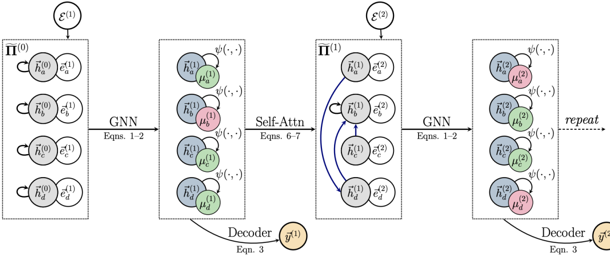
- Encoder
- Processor
- Decoder

PGN Architecture: Initialization



- PGN computes *latent features* $\vec{h}_i^{(t)} \in \mathbb{R}^k$ for each entity i. Initially, $\vec{h}_i^{(0)} = \vec{0}$.
- dynamic *pointers*: pointer adjacency matrix $\Pi^{(t)} \in \mathbb{R}^{n \times n}$.
- Pointers correspond to undirected edges between two entities: indicating that one of them points to the other. $\Pi^{(t)}$ is a binary symmetric matrix, Initially, each node points to itself: $\Pi^{(0)} = I_n$.

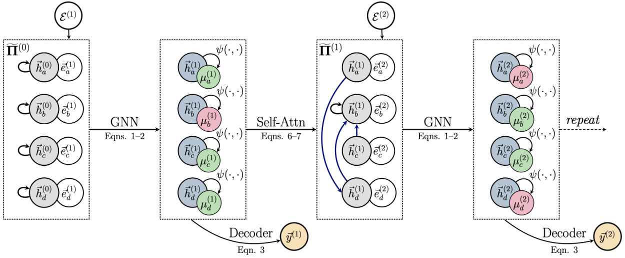
PGN Architecture: Encoder



- Encoder f:

$$\vec{z}_i^{(t)} = f\left(\vec{e}_i^{(t)}, \vec{h}_i^{(t-1)}\right) \tag{1}$$

PGN Architecture: Processor

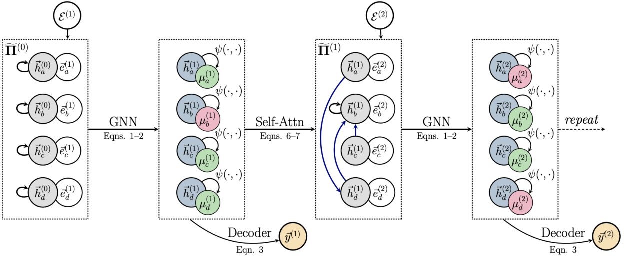


- *processor network*, P, which takes into account the current pointer adjacency matrix as relational information:

$$\mathsf{H}^{(t)} = P\left(\mathsf{Z}^{(t)}, \Pi^{(t-1)}\right) \tag{2}$$

yielding next-step latent features, $\mathsf{H}^{(t)} = (\vec{h}_1^{(t)}, \vec{h}_2^{(t)}, \ldots, \vec{h}_n^{(t)})$;

PGN Architecture: Decoder



- These latents can be used to answer set-level queries using a *decoder* network g:

$$\vec{y}^{(t)} = g\left(\bigoplus_i \vec{z}_i^{(t)}, \bigoplus_i \vec{h}_i^{(t)}\right) \tag{3}$$

where $\bigoplus$ is any permutation-invariant *readout* aggregator, such as summation or maximisation.

PGN Architecture: Learning Pointers

- Many efficient data structures only modify a small[1] subset of the entities at once.
- masking their pointer modifications through a sparse *mask* $\mu_i^{(t)} \in \{0, 1\}$ for each node that is generated by a *masking* network ψ:

$$\mathbb{P}\left(\mu_i^{(t)} = 1\right) = \psi\left(\vec{z}_i^{(t)}, \vec{h}_i^{(t)}\right) \tag{4}$$

- ψ is the logistic sigmoid function,
- threshold the output of ψ as follows:

$$\mu_i^{(t)} = \mathbb{I}_{\psi\left(\vec{z}_i^{(t)}, \vec{h}_i^{(t)}\right) > 0.5} \tag{5}$$

- The PGN now re-estimates the pointer adjacency matrix $\Pi^{(t)}$ using $\vec{h}_i^{(t)}$.

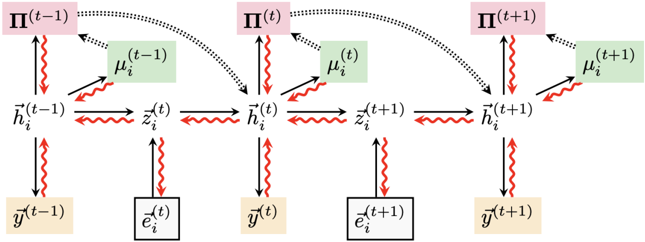
$$\vec{q}_i^{(t)} = \mathbf{W}_q \vec{h}_i^{(t)} \qquad \vec{k}_i^{(t)} = \mathbf{W}_k \vec{h}_i^{(t)} \qquad \alpha_{ij}^{(t)} = \mathrm{softmax}_j\left(\left\langle \vec{q}_i^{(t)}, \vec{k}_j^{(t)} \right\rangle\right) \tag{6}$$

[1] Typically on the order of $O(\log n)$ elements.

PGN Architecture: Learning Pointers

- $$\widetilde{\Pi}_{ij}^{(t)} = \mu_i^{(t)} \widetilde{\Pi}_{ij}^{(t-1)} + \left(1 - \mu_i^{(t)}\right) \mathbb{I}_{j=\mathrm{argmax}_k\left(\alpha_{ik}^{(t)}\right)} \qquad \Pi_{ij}^{(t)} = \widetilde{\Pi}_{ij}^{(t)} \vee \widetilde{\Pi}_{ji}^{(t)} \tag{7}$$
- direct supervision with respect to *ground-truth* pointers, $\hat{\Pi}^{(t)}$, of a target data structure.
- Applying $\mu_i^{(t)}$ effectively *masks out* parts of the computation graph for Equation 6, yielding a *graph attention network*-style update.

PGN Optimization



- query loss from y^t
- cross entropy from attention $\alpha_{ij}^{(t)}$ vs ground truth pointers
- cross entropy from attention $\mu_i^{(t)}$ vs ground truth masks

Dynamic Graph Connectivity

- Task: Is there a path between (u, v) in given Graph?
- subroutine in
  - Minimum Spanning Trees
  - Maximum Flows
- For example, in Kruskal's algorithm

```
algorithm Kruskal(G) is
    F:= ∅
    for each v ∈ G.V do
        MAKE-SET(v)
    for each (u, v) in G.E ordered by weight(u, v), increasing do
        if FIND-SET(u) ≠ FIND-SET(v) then
           F:= F ∪ {(u, v)}
           UNION(FIND-SET(u), FIND-SET(v))
    return F
```

Dynamic Graph Connectivity

- undirected and unweighted graphs of n nodes
- $G^t = (V, E^t)$
- $E^0 = \phi$
- $E^t = E^{t-1}\{u, v\}$, is the symmetric difference operator
- $\hat{y}^t$: Is there a path between (u, v) in given Graph G^t?

Incremental Graph Connectivity

- edges can only be added to the graph
- as edges cant removed, combining disconnected components is simply set union
- maintain disjoint sets
- *find*(u, v) if connected, check if the nodes (u, v) are in the same set

Incremental Graph Connectivity: Disjoint Set Union Data structure

INIT(u)

$\hat{\pi}_u = u$
$r_u \sim \mathcal{U}(0,1)$

FIND(u)

if $\hat{\pi}_u \neq u$
    $\hat{\pi}_u = \text{FIND}(\hat{\pi}_u)$
return $\hat{\pi}_u$

UNION(u, v)

$x = \text{FIND}(u)$
$y = \text{FIND}(v)$
if $x \neq y$
    if $r_x < r_y$
        $\hat{\pi}_x = y$
    else $\hat{\pi}_y = x$

QUERY-UNION(u, v)

if FIND(u) = FIND(v)
    return 0 // $\hat{y}^{(t)} = 0$
else UNION(u, v)
return 1 // $\hat{y}^{(t)} = 1$

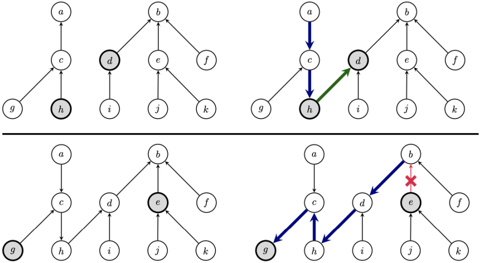
- DSU represents sets as rooted trees—each node, u, has a parent pointer, π_u
- the set identifier will be its root node, ρ_u, which by convention points to itself ($\pi_{\rho_u} = \rho_u$).
- find(u) reduces to recursively calling find(pi[u]) until the root is reached.
- path compression is applied: upon calling find(u), all nodes on the path from u to u will point to u.

Incremental Graph Connectivity: Disjoint Set Union Data structure

- at each step t, call query-union(u, v),
- operation descriptions $e_i^t = r_i || I_{i=u \downarrow i=v}$, containing the nodes' priorities, and a binary feature indicating which nodes are u and v.
- The corresponding output $\hat{y}^t$ indicates the return value of the $query - union(u, v)$.
- supervision for the PGN's (asymmetric) pointers: $\pi^i (i.e., \Pi_{ij}^{(t)} = 1$ iff $\pi^i = j$ else $\Pi_{ij}^{(t)} = 0$
- Ground-truth mask values, $\hat{\mu}_i^{(t)}$ are set to 0 for only the paths from u and v to their respective roots—no other node's state is changed

Fully dynamic tree connectivity with link/cut trees

```
QUERY-TOGGLE(u, v)
if r_u < r_v
    SWAP(u, v)
EVERT(u)
if FIND-ROOT(v) ≠ u
    LINK(u, v)
    return 0 // ŷ^(t) = 0
else CUT(v)
return 1 // ŷ^(t) = 1
```



- The operations supported by LCTs are: find-root(u) retrieves the root of node u;
- link(u, v) links nodes u and v, with the precondition that u is the root of its own tree; cut(v) removes the edge from v to its parent;
- evert(u) re-roots u's tree, such that u becomes the new root.

Fully dynamic tree connectivity with link/cut trees

- , here we will compress updates and queries into one operation, *querytoggle*(u, v), which our models will attempt to support.
- This operation first calls evert(u), then checks if u and v are connected: if they are not, adding the edge between them wouldn't introduce cycles (and u is now the root of its tree), so link(u, v) is performed.
- Otherwise, cut(v) is performed—it is guaranteed to succeed, as v is not going to be the root node.

Fully dynamic tree connectivity with link/cut trees

- encode each query-toggle(u, v) as $e(t)i = r_i||I_{i=u\downarrow i=v}$.
- we supervise the asymmetric pointers: π^i and ground-truth mask values, $\hat{\mu}_i^t$, are set to 0 only if π^i is modified in the operation at time t.

Experimental Evaluation: Data Generation

- out of distribution generalization: train on $n = 20$, $ops = 30$
- test on $n = 50, ops = 75$ and $n = 100, ops = 150$
- $n_{train} = 75$, $n_{test} = 35$, $n_{valid} = 35$
- sample u, v randomly at each time t
- Generate ground truth $\hat{y}^t, \hat{\mu}_i^t, \hat{\Pi}^t$ by running $query - union(u, v)$ and $query - toggle(u, v)$
- Training $k = 32$ features for each layer
- measure F_1 score on valid for same setting as training to select validation performance

Baselines

- FC-GNN : only query loss is used
- DeepSets: $\hat{\Pi}^t = I$
- PGN-NM (without masks): $\hat{\mu}_i^t = 0$
- Ablations
  - Oracle-Ptrs: $\hat{\Pi}^t$ are the ground truth pointers
  - PGN-Ptrs: Learn a PGN on the training set. Apply on all sets. Retrain on only query answering on the learnt pointers.

Results

Table 1: F_1 scores on the dynamic graph connectivity tasks for all models considered, on five random seeds. All models are trained on $n = 20$, ops $= 30$ and tested on larger test sets.

Model	**Disjoint-set union** $n=20$ ops $=30$	$n=50$ ops $=75$	$n=100$ ops $=150$	**Link/cut tree** $n=20$ ops $=30$	$n=50$ ops $=75$	$n=100$ ops $=150$
GNN	$0.892_{\pm.007}$	$0.851_{\pm.048}$	$0.733_{\pm.114}$	$0.558_{\pm.044}$	$0.510_{\pm.079}$	$0.401_{\pm.123}$
Deep Sets	$0.870_{\pm.029}$	$0.720_{\pm.132}$	$0.547_{\pm.217}$	$0.515_{\pm.080}$	$0.488_{\pm.074}$	$0.441_{\pm.068}$
PGN-NM	$\mathbf{0.910}_{\pm.011}$	$0.628_{\pm.071}$	$0.499_{\pm.096}$	$0.524_{\pm.063}$	$0.367_{\pm.018}$	$0.353_{\pm.029}$
PGN	$0.895_{\pm.006}$	$\mathbf{0.887}_{\pm.008}$	$\mathbf{0.866}_{\pm.011}$	$\mathbf{0.651}_{\pm.017}$	$\mathbf{0.624}_{\pm.016}$	$\mathbf{0.616}_{\pm.009}$
PGN-Ptrs	$0.902_{\pm.010}$	$0.902_{\pm.008}$	$0.889_{\pm.007}$	$0.630_{\pm.022}$	$0.603_{\pm.036}$	$0.546_{\pm.110}$
Oracle-Ptrs	$0.944_{\pm.006}$	$0.964_{\pm.007}$	$0.968_{\pm.013}$	$0.776_{\pm.011}$	$0.744_{\pm.026}$	$0.636_{\pm.065}$

Table 2: Pointer and mask accuracies of the PGN model w.r.t. ground-truth pointers.

Accuracy of	**Disjoint-set union** $n=20$ ops $=30$	$n=50$ ops $=75$	$n=100$ ops $=150$	**Link/cut tree** $n=20$ ops $=30$	$n=50$ ops $=75$	$n=100$ ops $=150$
Pointers (NM)	$\mathbf{80.3}_{\pm2.2\%}$	$32.9_{\pm2.7\%}$	$20.3_{\pm3.7\%}$	$\mathbf{61.3}_{\pm5.1\%}$	$17.8_{\pm3.3\%}$	$8.4_{\pm2.1\%}$
Pointers	$76.9_{\pm3.3\%}$	$\mathbf{64.7}_{\pm6.6\%}$	$\mathbf{55.0}_{\pm4.8\%}$	$60.0_{\pm1.3\%}$	$\mathbf{54.7}_{\pm1.9\%}$	$\mathbf{53.2}_{\pm2.2\%}$
Masks	$95.0_{\pm0.9\%}$	$96.4_{\pm0.6\%}$	$97.3_{\pm0.4\%}$	$82.8_{\pm0.9\%}$	$86.8_{\pm1.1\%}$	$91.1_{\pm1.0\%}$

Results: Rollout analysis of PGN Pointers

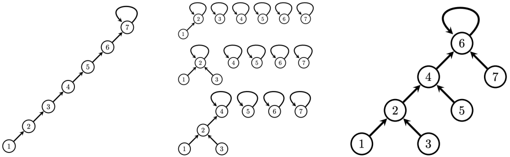


Figure 4: Visualisation of a PGN rollout on the DSU setup, for a pathological ground-truth case of repeated `union(i, i+1)` (**Left**). The first few pointers in $\mathbf{\Pi}^{(t)}$ are visualised (**Middle**) as well as the final state (**Right**)—the PGN produced a valid DSU at all times, but $2\times$ shallower than ground-truth.

- bad graph but good performance
- During rollout, the PGN models a correct DSU at all times, but halving its depth—easing GNN usage and GPU parallelisability.
- explains the reduced performance gap of PGNs to Oracle-Ptrs on LCT; as LCTs cannot apply path-compression-like tricks, the ground-truth LCT pointer graphs are expected to be of substantially larger diameters as test set size increases.

Conclusion

- Goal: learning a latent graph for answering classical algorithmic queries
- How: supervision from classical data structures, inductive biases from theoretical CS
- Results: out of distribution generalization, interpretable and parallelizable data structures for challenging graph connectivity tasks