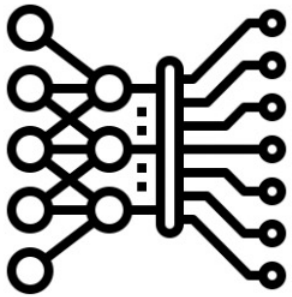
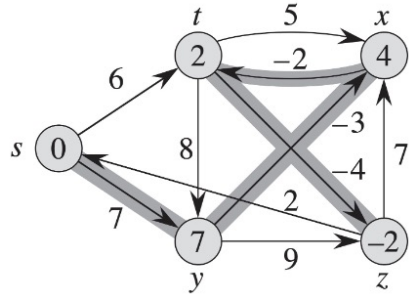
Neural Execution of Graph Algorithms

Petar Veličković, Rex Ying, Matilde Padovano, Raia Hadsell, Charles Blundell

International Conference on Learning Representations (ICLR) 2020

Problem-solving approaches





+ Operate on raw inputs
+ Generalize on noisy conditions
+ Models reusable across tasks
- Require big data
- Unreliable when extrapolating
- Lack of interpretability

+ Trivially strong generalization
+ Compositional (subroutines)
+ Guaranteed correctness
+ Interpretable operations
- Input must match spec
- Not robust to task variations

Is it possible to get the best of both worlds?

Slide by Veličković

Previous work

Previous work:

- Shortest path [Graves et al. ‘16; Xu et al., ‘19]
- Traveling salesman [Reed and De Freitas ‘15]
- Boolean satisfiability [Vinyals et al. ‘15; Bello et al., ‘16; …]
- Probabilistic inference [Yoon et al., ‘18]

Ground-truth solutions used to drive learning

Model has **complete freedom** mapping raw inputs to solutions

Neural graph algorithm execution

Key observation: Many algorithms share related **subroutines**

E.g. Bellman-Ford,BFS enumerate sets of edges adjacent to a node

Neural graph algorithm execution

- Learn several algorithms **simultaneously**
- Provide intermediate supervision signals

  Driven by how a known classical algorithm would process the input

Outline

1. Introduction
2. **Graph inputs**
3. GNN structure and implementation
4. Graph algorithms
5. Experiments

Graph inputs

Algorithm (GNN or classical algorithm):

- Processes a sequence of T graph-structured inputs
- Graph $G = (V, E)$ remains constant but meta-data varies

For $t \in [T]$:

- Each node $i \in V$ has features $\boldsymbol{x}_i^{(t)} \in \mathbb{R}^{N_x}$
- Each edge $(i, j) \in E$ has features $\boldsymbol{e}_{ij}^{(t)} \in \mathbb{R}^{N_e}$
- Algorithm produces node-level output $\boldsymbol{y}_i^{(t)} \in \mathbb{R}^{N_y}$
  - Parts of $\boldsymbol{y}_i^{(t)}$ may be used as next input $\boldsymbol{x}_i^{(t+1)}$

Outline

1. Introduction
2. Graph inputs
3. **GNN structure and implementation**
4. Graph algorithms
5. Experiments

GNN structure: *encode-process-decode*

For each algorithm A:

- **Encoder network $\boldsymbol{f_A}$**
  - **Input:** Previous latent features $\boldsymbol{h_i^{(t-1)}}$ (with $\boldsymbol{h_i^{(0)} = 0}$), input features $\boldsymbol{x_i^{(t)}}$
  - **Output:** Encoded inputs $\boldsymbol{z_i^{(t)} = f_A\left(h_i^{(t-1)}, x_i^{(t)}\right)}$
- **Processor network $\boldsymbol{P}$**
  - **Input:** Encoded inputs $\boldsymbol{Z^{(t)} = \left\{z_i^{(t)}\right\}_{i \in V}}$, edge features $\boldsymbol{E^{(t)} = \left\{e_{ij}^{(t)}\right\}_{e \in E}}$
  - **Output:** Latent node features $\boldsymbol{H^{(t)} = \left\{h_i^{(t)}\right\}_{i \in V} = P\left(Z^{(t)}, E^{(t)}\right)}$
- **Decoder network $\boldsymbol{g_A}$**
  - **Input:** Encoded inputs $\boldsymbol{z_i^{(t)}}$, latent features $\boldsymbol{h_i^{(t)}}$
  - **Output:** $\boldsymbol{y_i^{(t)} = g_A\left(z_i^{(t)}, h_i^{(t)}\right)}$

GNN structure: *encode-process-decode*

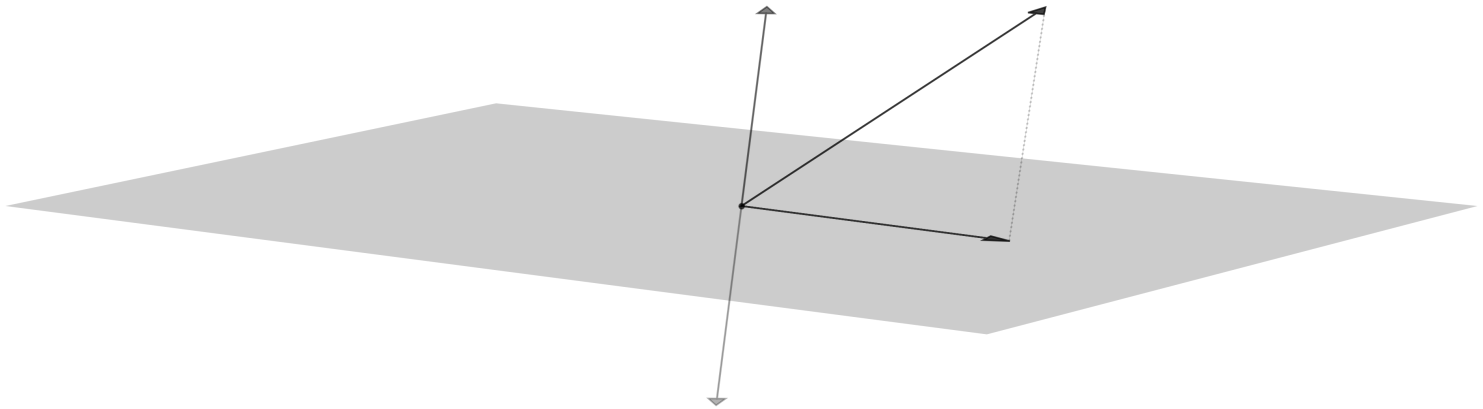
For each algorithm A:

- **Termination network $\boldsymbol{T_A}$**
  - Determines whether to terminate the algorithm
  - **Input:** Latent node features $\boldsymbol{H}^{(t)} = \left\{\boldsymbol{h}_i^{(t)}\right\}_{i \in V}$
  - **Output:** Probability of termination
- If not terminated:
  - Rerun *encode-process-decode*, potentially reusing parts of $\boldsymbol{y}_i^{(t)}$ for $\boldsymbol{x}_i^{(t+1)}$
- If not terminated after $|V|$ timesteps, terminate

Implementing *encode-process-decode*

Linear projections:

- Encoder network f_A
- Decoder network g_A
- Termination network T_A



Implementing *encode-process-decode*

- **Processor network P:** Graph neural network
- Evaluate two approaches:
  - Graph attention network (GAT, Veličković et al. '18)

$$\boldsymbol{h}_i^{(t)} = \mathrm{ReLU}\left(\sum_{(i,j)\in E} a\left(\boldsymbol{z}_i^{(t)}, \boldsymbol{z}_j^{(t)}, \boldsymbol{e}_{ij}^{(t)}\right) W \boldsymbol{z}_j^{(t)} \right)$$

Scalar coefficient from attention mechanism

Learnable projection matrix

Implementing *encode-process-decode*

- **Processor network P:** Graph neural network
- Evaluate two approaches:
  - Graph attention network (GAT, Veličković et al. '18)

$$\boldsymbol{h}_i^{(t)} = \mathrm{ReLU}\left(\sum_{(i,j)\in E} a\left(\boldsymbol{z}_i^{(t)}, \boldsymbol{z}_j^{(t)}, \boldsymbol{e}_{ij}^{(t)}\right) W \boldsymbol{z}_j^{(t)} \right)$$

  - Message-passing neural network (MPNN, Gilmer et al. '17)

$$\boldsymbol{h}_i^{(t)} = U\left(\boldsymbol{z}_i^{(t)}, \bigoplus_{(j,i)\in E} M\left(\boldsymbol{z}_i^{(t)}, \boldsymbol{z}_j^{(t)}, \boldsymbol{e}_{ij}^{(t)}\right)\right)$$

Neural networks producing vector messages
(this paper: linear projections)

Implementing *encode-process-decode*

- **Processor network $\boldsymbol{P}$:** Graph neural network
- Evaluate two approaches:
  - Graph attention network (GAT, Veličković et al. '18)

$$\boldsymbol{h}_i^{(t)} = \mathrm{ReLU}\left(\sum_{(i,j)\in E} a\left(\boldsymbol{z}_i^{(t)}, \boldsymbol{z}_j^{(t)}, \boldsymbol{e}_{ij}^{(t)}\right) W \boldsymbol{z}_j^{(t)}\right)$$

  - Message-passing neural network (MPNN, Gilmer et al. '17)

$$\boldsymbol{h}_i^{(t)} = U\left(\boldsymbol{z}_i^{(t)}, \bigoplus_{(j,i)\in E} M\left(\boldsymbol{z}_i^{(t)}, \boldsymbol{z}_j^{(t)}, \boldsymbol{e}_{ij}^{(t)}\right)\right)$$

Element-wise aggregator, e.g.:
Maximization, summation, averaging

Implementing *encode-process-decode*

- **Processor network P:** Graph neural network

- **Key idea:** P is algorithm-agnostic

  *Can be used to execute **multiple** algorithms*

Outline

1. Introduction
2. Graph inputs
3. GNN structure and implementation
4. **Graph algorithms**
5. Experiments

Breadth-first search

- Source node s
- Initial input $x_i^{(1)} = \begin{cases} 1 & \text{if } i = s \\ 0 & \text{if } i \neq s \end{cases}$
- Node is reachable from s if any of its neighbors are reachable:

$$x_i^{(t+1)} = \begin{cases} 1 & \text{if } x_i^{(t)} = 1 \\ 1 & \text{if } \exists j \text{ s.t. } (j,i) \in E \text{ and } x_j^{(t)} = 1 \\ 0 & \text{else} \end{cases}$$

- Algorithm output at round t: $y_i^{(t)} = x_i^{(t+1)}$

Bellman-Ford (shortest path)

- Source node s
- Initial input $x_i^{(1)} = \begin{cases} 0 & \text{if } i = s \\ \infty & \text{if } i \neq s \end{cases}$
- Node is reachable from s if any of its neighbors are reachable

  Update distance to node as minimal way to reach its neighbors

$$x_i^{(t+1)} = \min\left\{x_i^{(t+1)}, \min_{(j,i)\in E} x_j^{(t)} + e_{ji}^{(t)}\right\}$$

Bellman-Ford (shortest path)

- Source node s
- Initial input $x_i^{(1)} = \begin{cases} 0 & \text{if } i = s \\ \infty & \text{if } i \neq s \end{cases}$
- Also compute the predecessor node

$$p_i^{(t)} = \begin{cases} i & i = s \\ \underset{j:(j,i)\in E}{\operatorname{argmin}} x_j^{(t)} + e_{ji}^{(t)} & i \neq s \end{cases}$$

- Algorithm output at round t: $\boldsymbol{y}_i^{(t)} = \left(p_i^{(t)}, x_i^{(t+1)}\right)$

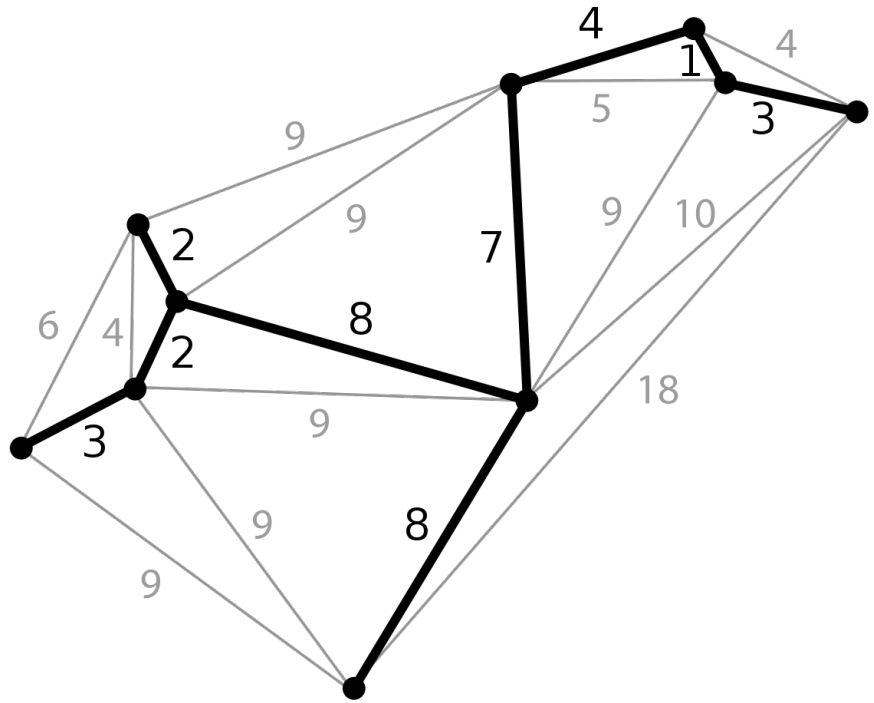
Learning multiple algorithms

Learn to execute both BFS and Bellman-Ford **simultaneously**

At each step t, concatenate relevant $x_i^{(t)}$ and $\boldsymbol{y}_i^{(t)}$ values

Minimum spanning tree

Also analyze Prim's algorithm for minimum spanning tree



Outline

1. Introduction
2. Graph inputs
3. GNN structure and implementation
4. Graph algorithms
5. **Experiments**

Experimental setup

Variety of different **graph structures**

Ladder graphs, 2D grids, trees, Erdős–Rényi, Barabási-Albert, …

Edge weights are drawn uniformly from $[0.2, 1]$

For each graph category:

- **Training:** 100 graphs with 20 nodes
- **Validation:** 5 graphs with 20 nodes
- **Testing:** 5 graphs with 20, 50, and 100 nodes

Experimental setup: Loss functions

Reachability: Binary cross-entropy

$$-\left(x_i^{(t)} \log \hat{x}_i^{(t)} + \left(1 - x_i^{(t)}\right) \log\left(1 - \hat{x}_i^{(t)}\right)\right)$$

Distance: Mean-squared error $\left\| x_i^{(t)} - \hat{x}_i^{(t)} \right\|^2$

Termination: Binary cross-entropy

Experimental setup: Loss functions

Predecessor of i $\left(p_i^{(t)}\right)$:

- For every neighbor j, use a NN to calculate a score
  - Input to NN is $\left(\boldsymbol{h}_i^{(t)}, \boldsymbol{h}_j^{(t)}, \boldsymbol{e}_{ij}^{(t)}\right)$
- Make prediction $\hat{p}_{ij}^{(t)}$ using a softmax of the scores
- **Categorical cross-entropy loss:**

  If $j = p_i^{(t)}$, equal to $-\log \hat{p}_{ij}^{(t)}$

Comparisons

(curriculum, Bengio et al., '09):

- BFS learnt in isolation to perfect validation accuracy
- Fine-tune on Bellman-Ford

(no-reach): Learn Bellman-Ford alone

- Doesn't simultaneously learn reachability

(no-algo):

- Don't supervise intermediate steps
- Learn predecessors directly from input $x_i^{(1)}$

Shortest-path predecessor prediction

Model	Predecessor (mean step accuracy / last-step accuracy) *20 nodes*	*50 nodes*	*100 nodes*
LSTM (Hochreiter & Schmidhuber, 1997)	47.20% / 47.04%	36.34% / 35.24%	27.59% / 27.31%
GAT* (Veličković et al., 2018)	64.77% / 60.37%	52.20% / 49.71%	47.23% / 44.90%
GAT-full* (Vaswani et al., 2017)	67.31% / 63.99%	50.54% / 48.51%	43.12% / 41.80%
MPNN-mean (Gilmer et al., 2017)	93.83% / 93.20%	58.60% / 58.02%	44.24% / 43.93%
MPNN-sum (Gilmer et al., 2017)	82.46% / 80.49%	54.78% / 52.06%	37.97% / 37.32%
MPNN-max (Gilmer et al., 2017)	**97.13% / 96.84%**	**94.71% / 93.88%**	**90.91% / 88.79%**
MPNN-max (*curriculum*)	95.88% / 95.54%	91.00% / 88.74%	84.18% / 83.16%
MPNN-max (*no-reach*)	82.40% / 78.29%	78.79% / 77.53%	81.04% / 81.06%
MPNN-max (*no-algo*)	78.97% / 95.56%	83.82% / 85.87%	79.77% / 78.84%

Improvement of max-aggregator increases with size

Shortest-path predecessor prediction

Model	Predecessor (mean step accuracy / last-step accuracy) *20 nodes*	*50 nodes*	*100 nodes*
LSTM (Hochreiter & Schmidhuber, 1997)	47.20% / 47.04%	36.34% / 35.24%	27.59% / 27.31%
GAT* (Veličković et al., 2018)	64.77% / 60.37%	52.20% / 49.71%	47.23% / 44.90%
GAT-full* (Vaswani et al., 2017)	67.31% / 63.99%	50.54% / 48.51%	43.12% / 41.80%
MPNN-mean (Gilmer et al., 2017)	93.83% / 93.20%	58.60% / 58.02%	44.24% / 43.93%
MPNN-sum (Gilmer et al., 2017)	82.46% / 80.49%	54.78% / 52.06%	37.97% / 37.32%
MPNN-max (Gilmer et al., 2017)	**97.13% / 96.84%**	**94.71% / 93.88%**	**90.91% / 88.79%**
MPNN-max (*curriculum*)	95.88% / 95.54%	91.00% / 88.74%	84.18% / 83.16%
MPNN-max (*no-reach*)	82.40% / 78.29%	78.79% / 77.53%	81.04% / 81.06%
MPNN-max (*no-algo*)	78.97% / 95.56%	83.82% / 85.87%	79.77% / 78.84%

- **(no-reach) results**: positive knowledge transfer
- **(no-algo) results**: benefit of supervising intermediate steps

Shortest-path predecessor prediction

	MPNN-max predecessor prediction					
Metric	*20 nodes*	*50 nodes*	*100 nodes*	*500 nodes*	*1000 nodes*	*1500 nodes*
Mean step accuracy	97.13%	94.71%	90.91%	83.08%	77.53%	74.90%
Last-step accuracy	96.84%	93.88%	88.79%	76.46%	72.74%	67.66%

MPNN-max **generalizes** to much **larger graphs**

Learning across graph structures

		Reachability		Predecessor	
	Graph type	*From Erdős-Rényi*	*From trees*	*From Erdős-Rényi*	*From trees*
Locally regular	Ladder	93.16% / 93.98%	99.93% / 99.67%	76.63% / 65.94%	94.99% / 92.55%
	2-D Grid	92.86% / 87.05%	99.85% / 99.32%	79.50% / 70.75%	94.06% / 91.39%
	Tree	82.72% / 82.07%	99.92% / 99.62%	70.16% / 63.26%	98.44% / 97.33%
More variable	Erdős-Rényi	100.0% / 100.0%	100.0% / 100.0%	96.17% / 93.94%	91.11% / 85.94%
	Barabási-Albert	100.0% / 100.0%	100.0% / 100.0%	94.91% / 92.90%	83.90% / 75.79%
	4-Community	100.0% / 100.0%	100.0% / 100.0%	90.01% / 86.38%	75.88% / 64.04%
	4-Caveman	100.0% / 100.0%	100.0% / 100.0%	91.55% / 90.04%	80.02% / 72.06%

- MPNN-max biased to structural regularities of input graph
- Still generalizes to other types of graphs

Overview

Introduced **neural graph algorithm execution**

- Train GNN to imitate **intermediate steps** of graph algorithms
- Learn **multiple algorithms** simultaneously

Applications to reachability, shortest paths, and MSTs

Experiments demonstrate benefits of:

- **Maximization**-based message passing NNs
- **Multi-task** learning and positive knowledge transfer