Neural Execution of Graph Algorithms

Petar Veličković[1], **Rex Ying**[1,2], **Matilde Padovano**[1,3], **Raia Hadsell**[1] **and Charles Blundell**[1]
[1]DeepMind [2]Stanford University [3]University of Cambridge

Abstract

Graph Neural Networks (GNNs) are a powerful representational tool for solving problems on graph-structured inputs. In almost all cases so far, however, they have been applied to directly recovering a final solution from raw inputs, without explicit guidance on how to *structure* their problem-solving. Here, instead, we focus on learning in the space of *algorithms*: we train several state-of-the-art GNN architectures to imitate individual steps of classical graph algorithms. As graph algorithms usually rely on making discrete decisions within neighbourhoods, we hypothesise that maximisation-based message passing neural networks are best-suited for such objectives, and validate this claim empirically. We also demonstrate how learning in the space of algorithms can yield new opportunities for *positive transfer* between tasks—showing how learning a shortest-path algorithm can be substantially improved when simultaneously learning a reachability algorithm.

1 Introduction

A multitude of important real-world tasks can be formulated as tasks over graph-structured inputs, such as navigation, web search, protein folding, and game-playing. Theoretical computer science has successfully discovered effective and highly influential algorithms for many of these tasks. But many problems are still considered intractable from this perspective.

Machine learning approaches have been applied to many of these classic tasks, from tasks with known polynomial time algorithms such as *shortest paths* [8, 28] and *sorting* [19], to intractable tasks such as *travelling salesman* [4, 16, 25] and *boolean satisfiability* [21, 22]. Recently, this work often relies on advancements in graph representation learning [2, 5, 9] with graph neural networks (GNNs) [7, 15, 18, 24]. In almost all cases so far, ground-truth *solutions* are used to drive learning, giving the model complete freedom to find a mapping from raw inputs to such solution.

Many classical algorithms share related *subroutines*: for example, shortest path computation (via the Bellman-Ford [3] algorithm) and breadth-first search both must enumerate sets of edges adjacent to a particular node. Inspired by previous work on the more general tasks of program synthesis and learning to execute [13, 17, 19, 20, 32], we show that by learning several algorithms simultaneously and providing a supervision signal, our neural network is able to demonstrate positive knowledge transfer among learning different algorithms. The supervision signal is driven by how a known classical algorithm would process such inputs (including any relevant *intermediate outputs*), providing explicit (and *reusable*) guidance on how to tackle graph-structured problems. We call this approach **neural graph algorithm execution**.

Given that the majority of popular algorithms requires making *discrete decisions* over neighbourhoods (e.g. *"which edge should be taken?"*), we suggest that a highly suitable architecture for this task is a message-passing neural network (MPNN) [7] with a *maximisation* aggregator—a claim we verify by demonstrating clear performance benefits to simultaneously learning breadth-first search for reachability with the Bellman-Ford algorithm for shortest paths.

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

2 Problem setup

Graph algorithms We consider graphs $G = (V, E)$ with node features $\vec{x}_i^{(t)} \in \mathbb{R}^{N_x} (i \in V)$ and edge features $\vec{e}_{ij}^{(t)} \in \mathbb{R}^{N_e} ((i, j) \in E)$ given as input at step $t \in \mathbb{N}$ of the algorithm. At each step, the algorithm produces node-level outputs $\vec{y}_i^{(t)} \in \mathbb{R}^{N_y}$. Some of these outputs may then be reused as inputs on the next step; i.e., $\vec{x}_i^{(t+1)}$ may contain some elements of $\vec{y}_i^{(t)}$. Overall, the algorithm runs for T steps (where T may vary across input graphs).

Learning to execute graph algorithms To define a generic architecture for learning to execute algorithms, we leverage the *encode-process-decode* paradigm [10]. At each step, the algorithm learner computes K-dimensional *latent node features* $\vec{h}_i^{(t)} \in \mathbb{R}^K$ (specially, initialised as $\vec{h}_i^{(0)} = \vec{0}$). First, an algorithm-specific *encoder network* f_A is applied to the current input features and previous latent features to produce *encoded inputs*, $\vec{z}_i^{(t)}$, as such:

$$\vec{z}_i^{(t)} = f_A(\vec{x}_i^{(t)}, \vec{h}_i^{(t-1)}) \tag{1}$$

The encoded inputs are then processed using the *processor network* P, computing latent node features for this step (denoting by $\mathbf{H}^{(t)}$ the set of all $\vec{h}_i^{(t)}$, $\mathbf{Z}^{(t)}$ the set of all $\vec{z}_i^{(t)}$, and $\mathbf{E}^{(t)}$ the set of all $\vec{e}_{ij}^{(t)}$)):

$$\mathbf{H}^{(t)} = P(\mathbf{Z}^{(t)}, \mathbf{E}^{(t)}) \tag{2}$$

We may then compute node-level outputs, using an algorithm-specific *decoder-network*, g_A:

$$\vec{y}_i^{(t)} = g_A(\vec{z}_i^{(t)}, \vec{h}_i^{(t)}) \tag{3}$$

Specially, the processor network also needs to make a decision on whether to terminate the algorithm. This is performed by an algorithm-specific *termination network*, T_A, which provides the probability of termination $\tau^{(t)}$—after applying the logistic sigmoid activation σ—as follows:

$$\tau^{(t)} = \sigma(T_A(\mathbf{H}^{(t)})) \tag{4}$$

If the algorithm hasn't terminated (e.g. if $\tau^{(t)} > 0.5$) the computation of Equations 1–4 is repeated—with parts of $\vec{y}_i^{(t)}$ potentially reused within $\vec{x}_i^{(t+1)}$.

In our experiments, f_A, g_A and T_A are all linear projections, with T_A receiving the average node embedding $\frac{1}{|V|} \sum_{i \in V} \vec{h}_i^{(t)}$ as input. As we would like the processor network to be mindful of the structural properties of the input, we employ a graph neural network (GNN) layer capable of exploiting edge features as P. Specifically, we compare graph attention networks (GATs) [24] (Equation 5, left) against message-passing neural networks (MPNNs) [7] (Equation 5, right):

$$\vec{h}_i^{(t)} = \mathrm{ReLU}\left(\sum_{(j,i) \in E} a\left(\vec{z}_i^{(t)}, \vec{z}_j^{(t)}, \vec{e}_{ij}^{(t)}\right) \mathbf{W} \vec{z}_j^{(t)}\right) \qquad \vec{h}_i^{(t)} = U\left(\vec{z}_i^{(t)}, \bigoplus_{(j,i) \in E} M\left(\vec{z}_i^{(t)}, \vec{z}_j^{(t)}, \vec{e}_{ij}^{(t)}\right)\right) \tag{5}$$

where $\mathbf{W}$ is a learnable projection matrix, a is an attention mechanism producing *scalar coefficients*, while M and U are neural networks producing *vector messages*. $\bigoplus$ is an elementwise aggregation operator, such as maximisation, summation or averaging. We use linear projections for M and U.

3 Evaluation

Graph generation To provide our learner with a wide variety of input graph structure types, we follow prior work [30, 31] and generate graphs from seven categories: *Erdős-Rényi* [6], *Barabási-Albert* [1], *grid*, *4-community*, *4-caveman* [26], *trees* and *ladder* graphs. We additionally add a self-edge to every node in the graphs, and attach a real-valued weight to every edge (drawn uniformly from the range $[0.2, 1]$). These weight values serve as the sole edge features, $e_{ij}^{(t)}$, for all steps t.

We aim to study the algorithm execution task from a *"programmer"* perspective: human experts may manually inspect only relatively small graphs, and any algorithms derived from this should apply to

arbitrarily large graphs. As such, we generate training and validation graphs of only 20 nodes—100 of each category for training, and 5 of each category for validation. For the test dataset, we once again generate 5 graphs of each category: testing generalisation at 20, 50 and 100 nodes.

Algorithms under study We consider two classical algorithms here: *breadth-first search* for reachability, and the *Bellman-Ford* algorithm [3] for shortest paths. The former maintains a single-bit value in each node, determining whether said node is reachable from a source node, and the latter maintains a scalar value in each node, representing its distance from the source node.

In both cases, the algorithm is initialised by randomly selecting the *source* node, s. As the initial input to the algorithms, $x_i^{(1)}$, we have:

$$\text{BFS} : x_i^{(1)} = \begin{cases} 1 & i = s \\ 0 & i \neq s \end{cases} \qquad \text{Bellman-Ford} : x_i^{(1)} = \begin{cases} 0 & i = s \\ +\infty & i \neq s \end{cases} \tag{6}$$

This information is then propagated according to the chosen algorithm: a node becomes reachable from s if any of its neighbours are reachable from s, and we may update the distance to a given node as the minimal way to reach any of its neighbours, then taking the connecting edge:

$$\text{BFS} : x_i^{(t+1)} = \begin{cases} 1 & x_i^{(t)} = 1 \\ 1 & \exists j.(j,i) \in E \wedge x_j^{(t)} = 1 \\ 0 & \text{otherwise} \end{cases} \qquad \text{B-F} : x_i^{(t+1)} = \min\left(\vec{x}_i^{(t)}, \min_{(j,i)\in E} x_j^{(t)} + e_{ji}^{(t)}\right) \tag{7}$$

For breadth-first search, no additional information is being computed, hence $y_i^{(t)} = x_i^{(t+1)}$. Additionally, at each step the Bellman-Ford algorithm may compute, for each node, the *"predecessor"* node, $p_i^{(t)}$ in the shortest path (indicating which edge should be taken to reach this node). This information is ultimately used to reconstruct shortest paths, and hence represents the crucial output:

$$\text{Bellman-Ford} : p_i^{(t)} = \begin{cases} i & i = s \\ \operatorname*{argmin}_{j;(j,i)\in E} x_j^{(t)} + e_{ji}^{(t)} & i \neq s \end{cases} \tag{8}$$

Hence, for Bellman-Ford, $\vec{y}_i^{(t)} = p_i^{(t)} \| x_i^{(t+1)}$, where $\|$ is concatenation. To provide a numerically stable value for $+\infty$, we set all such entries to the length of the longest shortest path in the graph + 1.

We learn to execute these two algorithms *simultaneously*—at each step, concatenating the relevant $\vec{x}_i^{(t)}$ and $\vec{y}_i^{(t)}$ values for them. As both of the algorithms considered here (and most others) rely on discrete decisions over neighbourhoods, learning to execute them should be naturally suited for the MPNN with the max-aggregator—a claim which we directly verify in the remainder of this section.

Neural network architectures To assess the comparative benefits of different architectures for the neural algorithm execution task, we consider many candidate networks executing the computation of Equations 1–5, especially the processor network P: For the MPNN update rule, we consider maximisation, mean and summation aggregators; For the GAT update rule, we consider the originally proposed attention mechanism of [24], as well as the Transformer attention mechanism [23]. We consider also attending over the *full graph*—adding a second attention head, only acting on the *non-edges* of the graph (and hence not accepting any edge features). The two heads' features are then concatenated. Analogously to our expectation that the best-performing MPNN rule will perform maximisation, we attempt to force the attentional coefficients of GAT to be as *sharp* as possible—applying either an *entropy penalty* to them (as in [29]) or the *Gumbel softmax* trick [12].

We perform an additional sanity check to ensure that a GNN-like architecture is necessary in this case. Prior work [28] has already demonstrated the unsuitability of MLPs for reasoning tasks like these, and they will not support variable amounts of nodes. Here, instead, we consider an LSTM [11] architecture to which the graph is fed in serialised form of an edge list (in a setup similar to [8]).

In all cases, the neural networks compute a latent dimension of $K = 32$ features, and are optimised using the Adam SGD optimiser [14] on the binary cross-entropy for the reachability predictions,

Table 1: Accuracy of predicting reachability at different test-set sizes, trained on graphs of 20 nodes. GAT* correspond to the best GAT setup as per Section 3 (GAT-full using the full graph).

	Reachability (mean step accuracy / last-step accuracy)		
Model	*20 nodes*	*50 nodes*	*100 nodes*
LSTM [11]	81.97% / 82.29%	88.35% / 91.49%	68.19% / 63.37%
GAT* [24]	93.28% / 99.86%	93.97% / **100.0%**	92.34% / **99.97%**
GAT-full* [23]	78.40% / 77.86%	85.76% / 91.83%	88.98% / 91.51%
MPNN-mean [7]	**100.0% / 100.0%**	61.05% / 57.89%	27.17% / 21.40%
MPNN-sum [7]	99.66% / **100.0%**	94.25% / **100.0%**	94.72% / 98.63%
MPNN-max [7]	**100.0% / 100.0%**	**100.0% / 100.0%**	**99.92%** / 99.80%

Table 2: Accuracy of predicting the shortest-path predecessor node at different test-set sizes. MPNN-max (*no-reach*) corresponds to training without the reachability task. MPNN-max (*no-algo*) corresponds to the classical setup of directly training on the predecessor, without predicting any intermediate outputs or distances.

	Predecessor (mean step accuracy / last-step accuracy)		
Model	*20 nodes*	*50 nodes*	*100 nodes*
LSTM [11]	47.20% / 47.04%	36.34% / 35.24%	27.59% / 27.31%
GAT* [24]	64.77% / 60.37%	52.20% / 49.71%	47.23% / 44.90%
GAT-full* [23]	67.31% / 63.99%	50.54% / 48.51%	43.12% / 41.80%
MPNN-mean [7]	93.83% / 93.20%	58.60% / 58.02%	44.24% / 43.93%
MPNN-sum [7]	82.46% / 80.49%	54.78% / 52.06%	37.97% / 37.32%
MPNN-max [7]	**97.13% / 96.84%**	**94.71% / 93.88%**	**90.91% / 88.79%**
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%

mean squared error for the distance predictions, categorical cross-entropy for the predecessor node predictions, and binary cross-entropy for predicting termination (all applied simultaneously). We use an initial learning rate of 0.0005, and perform early stopping on the validation accuracy for the predecessor node (with a patience of 10 epochs). If the termination network T_A does not terminate the neural network computation within $|V|$ steps, it is assumed terminated at that point.

Results and discussion In order to evaluate how faithfully the neural algorithm executor replicates the two algorithms, we propose reporting the accuracy of predicting the reachability (for breadth-first search; Table 1), as well as predicting the predecessor node (for Bellman-Ford; Table 2). We report this metric *averaged across all steps* t (to give a sense of how well the algorithm is imitated across time), as well as the *last-step performance* (which corresponds to the final solution).

The results confirm our hypotheses: the MPNN-max model exhibits superior generalisation performance on both reachability and shortest-path predecessor node prediction. Even when allowing for hardening the attention of GAT-like models, the more general computational model of MPNN is capable of outperforming them. The MPNN-sum model may also learn various thresholding functions, as demonstrated by [27]—however, aggregating messages in this way can lead to outputs of exploding magnitude, rendering the network hard to control for larger graphs.

The performance gap on predicting the predecessor widens significantly as the test graph size increases. Furthermore, we note clear signs of *positive knowledge transfer* occurring between the reachability and shortest-path task: when the shortest-path task is learned in isolation, the predictive power of MPNN-max drops significantly (while still outperforming many other approaches). We believe that this should serve as strong motivation for further work in the area, attempting to learn more algorithms simultaneously and exploiting the similarities between their respective subroutines whenever appropriate.

References

[1] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Reviews of modern physics*, 74(1):47, 2002.

[2] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

[3] Richard Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958.

[4] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

[5] Michael M Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017.

[6] Paul Erdős and Alfréd Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5(1):17–60, 1960.

[7] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. *arXiv preprint arXiv:1704.01212*, 2017.

[8] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471, 2016.

[9] William L Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. *arXiv preprint arXiv:1709.05584*, 2017.

[10] Jessica B Hamrick, Kelsey R Allen, Victor Bapst, Tina Zhu, Kevin R McKee, Joshua B Tenenbaum, and Peter W Battaglia. Relational inductive bias for physical construction in humans and machines. *arXiv preprint arXiv:1806.01203*, 2018.

[11] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[12] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*, 2016.

[13] Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015.

[14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[15] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[16] Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems! *arXiv preprint arXiv:1803.08475*, 2018.

[17] Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random-access machines. *arXiv preprint arXiv:1511.06392*, 2015.

[18] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

[19] Scott Reed and Nando De Freitas. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*, 2015.

[20] Adam Santoro, Ryan Faulkner, David Raposo, Jack Rae, Mike Chrzanowski, Theophane Weber, Daan Wierstra, Oriol Vinyals, Razvan Pascanu, and Timothy Lillicrap. Relational recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 7299–7310, 2018.

[21] Daniel Selsam and Nikolaj Bjørner. Guiding high-performance sat solvers with unsat-core predictions. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 336–353. Springer, 2019.

[22] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L Dill. Learning a sat solver from single-bit supervision. *arXiv preprint arXiv:1802.03685*, 2018.

[23] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[24] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 1(2), 2017.

[25] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems*, pages 2692–2700, 2015.

[26] Duncan J Watts. Networks, dynamics, and the small-world phenomenon. *American Journal of sociology*, 105(2):493–527, 1999.

[27] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[28] Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. What can neural networks reason about? *arXiv preprint arXiv:1905.13211*, 2019.

[29] Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. *arXiv preprint arXiv:1806.08804*, 2018.

[30] Jiaxuan You, Rex Ying, and Jure Leskovec. Position-aware graph neural networks. *arXiv preprint arXiv:1906.04817*, 2019.

[31] Jiaxuan You, Rex Ying, Xiang Ren, William L Hamilton, and Jure Leskovec. Graphrnn: A deep generative model for graphs. *arXiv preprint arXiv:1802.08773*, 2018.

[32] Wojciech Zaremba and Ilya Sutskever. Learning to execute. *arXiv preprint arXiv:1410.4615*, 2014.