The Deep Equilibrium Algorithmic Reasoner

Dobrik Georgiev
University of Cambridge
dgg30@cam.ac.uk

Pietro Liò
University of Cambridge
pl219@cam.ac.uk

Davide Buffelli
MediaTek Research
davide.buffelli@mtkresearch.com

Abstract

Recent work on neural algorithmic reasoning has demonstrated that graph neural networks (GNNs) could learn to execute classical algorithms. Doing so, however, has always used a recurrent architecture, where each iteration of the GNN aligns with an algorithm's iteration. Since an algorithm's solution is often an equilibrium, we conjecture and empirically validate that one can train a network to solve algorithmic problems by directly finding the equilibrium. Note that this does not require matching each GNN iteration with a step of the algorithm.

1. Introduction

Neural Algorithmic Reasoning (a.k.a. NAR; 21) models are a class of neural networks (usually graph neural networks, a.k.a. GNNs) that learn to imitate classical algorithms [14]. One of the key reasons that this performance is achievable is *alignment* [25]: GNNs that align better to the target algorithm achieve better generalisation. This alignment game has led to a sequence of exciting research – from aligning the architecture with iterative algorithms [20] to proving that "graph neural networks are dynamic programmers" [8].[1]

Notably, the aforementioned papers focus on aligning the computation of the GNN with an algorithm or a specific algorithm class (e.g. dynamic programming), but ignore the properties *at the time of algorithm termination*. For a number of algorithms (e.g. sorting, shortest-path and dynamic programming) in the CLRS-30 algorithmic benchmark [23] once the optimal solution is found, further algorithm iterations will not change the algorithm's output prediction values. For example, in shortest-paths algorithms [e.g. ones found in 7] making additional iterations would not alter the optimality of the shortest paths' distances found. We will call such state an *equilibrium* – additional applications of a function (an algorithm's iteration) to the state leave it unchanged. There are other examples of algorithms in the benchmark that have equilibriums at their solved state: Kruskal's [16] minimum spanning tree algorithm[2], sorting algorithms, etc.

In this paper, we show that:

1. We investigate a novel approach to learning algorithms by identifying the equilibrium point of the GNN equation.
2. By aligning the NAR models to the equilibrium property discussed in the previous paragraph we can improve the model accuracy.
3. By removing the requirement that one step of the GNN $\leftrightarrow$ one step of the algorithm and by finding equilibrium points it is possible to reduce the required number of GNN iterations.

The rest of the paper is organised as follows: Sec. 2 provides a short background on NAR and deep equilibrium models (DEQs; the approach we used), Sec. 3 outlines our model in more detail, presenting theoretically why we can express NAR with DEQ, Sec. 4 gives information on our experimental setup and presents the results.

2. Background

2.1. Algorithmic Reasoning

Let $A : \mathbb{I}_A \to \mathbb{O}_A$ be an algorithm, acting on some input $\boldsymbol{x} \in \mathbb{I}_A$, producing an output $A(\boldsymbol{x}) \in \mathbb{O}_A$ and let $\mathbb{I}_A/\mathbb{O}_A$ be the set of possible inputs/outputs A can read/return. In algorithmic reasoning, we aim to learn a function $\mathcal{A} : \mathbb{I}_A \to \mathbb{O}_A$, such that $\mathcal{A} \approx A$. Importantly, we will not be learning simply an input-output mapping, but we will aim to align to the algorithm A's trajectory. The alignment is often achieved through direct supervision[3] on the intermediate states of the algorithm. To capture the execution of A on an input x we

[1] We refer the reader to Veličković [24] for a more comprehensive overview of NAR, the "alignment game" and examples of why learning to execute algorithms is useful in real-world practice.

[2] Once the tree is built, iterating over the rest of the edges, if any remain, will not change the solution.

[3] Recent research [4, 18] has shown that alternative, causality-inspired, ways of alignment also exist.

can represent it as

$$\bar{\boldsymbol{h}}_0 = \textsc{Preprocess}(\boldsymbol{x}) \tag{1}$$

$$\bar{\boldsymbol{h}}_\tau = \underbrace{A_t(A_t(\dots A_t}_{\tau \text{ times}}(\bar{\boldsymbol{h}}_0)\dots)) \tag{2}$$

$$A(\boldsymbol{x}) = \textsc{Postprocess}(\bar{\boldsymbol{h}}_\tau) \tag{3}$$

where PREPROCESS and POSTPROCESS are some simple pre- and post-processing (e.g.initialising auxiliary variables or returning the correct variable), $\bar{\boldsymbol{h}}_\tau \in \mathbb{H}_A$ is A's internal (**h**idden) state and A_t is a subroutine (or a set of) that is executed at each step. It is therefore no surprise that the encode-process-decode architecture [13] is the de-facto choice when it comes to NAR. Thus, the architecture can be neatly represented as a composition of three learnable components: $\mathcal{A} = g_\mathcal{A} \circ P \circ f_\mathcal{A}$, where $g_\mathcal{A} : \mathbb{I}_A \to \mathbb{R}^d$ and $f_\mathcal{A} : \mathbb{R}^d \to \mathbb{O}_A$ are the encoder and decoder function respectively (usually linear projections) and $P : \mathbb{R}^d \to \mathbb{R}^d$ is a processor that mimics the rollout (Eq. (2)) of A. The processor is often modelled as a message-passing GNN with the message function containing nonlinearities (e.g. ReLU).

CLRS-30 The *CLRS-30* benchmark [23] includes 30 iconic algorithms from the *Introduction to Algorithms* textbook [6]. Each data instance for an algorithm A is a graph annotated with features from different algorithm stages (*input*, *output*, and *hint*), each associated with a location (*node*, *edge*, and *graph*). Hints contain time series data representing the algorithm rollout and include a temporal dimension often used to infer the number of steps τ. Features in CLRS-30 have various datatypes with associated losses for training. The test split, designed for assessing out-of-distribution (OOD) generalization, comprises graphs four times larger than the training set. For more details, see Veličković et al. [23].

2.2. Deep Equilibrium Models

Deep equilibrium models [DEQs 2] are a class of implicit neural networks [11]. The functions modelled with DEQs are of the form:

$$\boldsymbol{z}^* = f_\theta(\boldsymbol{z}^*, \boldsymbol{x}) \tag{4}$$

where $\boldsymbol{x}$ is input, f_θ is a function parametrised by θ (e.g. a neural network) and $\boldsymbol{z}^*$ is the output. $\boldsymbol{z}^*$ is an equilibrium point to the eventual output value of an infinite depth network where each layer's weights are shared, i.e. $f_\theta^{[i]} = f_\theta$. By re-expressing (4) as $g_\theta = f_\theta(\boldsymbol{z}^*, \boldsymbol{x}) - \boldsymbol{z}^*$ DEQs allow us to find the fixed point $\boldsymbol{z}^*$ via any black-box root-finding method [e.g. 1, 5], without the actual need of unrolling the equation until convergence. DEQs also integrate with back-propagation – gradient $\partial\mathcal{L}/\partial\theta$ could be calculated using the Implicit Function Theorem (cf. 2) and no intermediate state has to be stored, giving us constant memory cost of gradient computation regardless of the number of iterations until convergence.

Another nice property DEQs possess is that they are independent of the choice of f_θ. Ultimately, this is what allows us to integrate them with ease with algorithmic reasoner architectures. Other implicit GNN models [e.g 12] do not provide the flexibility to incorporate edge features or use a `max` aggregator.

3. The Deep Equilibrium Algorithmic Reasoner

3.1. Motivation

The previous section has hinted at the existence of an alignment between equilibrium models and algorithms. This was also observed empirically by Mirjanic et al. [17], who showed the presence of attractor states in the latent space trajectories of the algorithm: graphs of similar executions tend to cluster together and do not diverge. This further motivates the usage of DEQs for NAR since finding the equilibrium of f_θ reliably and efficiently requires the function to be stable.

3.2. Implementation

We implement our processor as a PGN architecture [22] with a gating mechanism as in Ibarz et al. [14]:

$$\mathbf{z}_i^{(t)} = \mathbf{u}_i \| \mathbf{h}_i^{(t-1)} \tag{5}$$

$$\mathbf{m}_i^{(t)} = \max_{j \in \mathcal{N}_i} P_m\left(\mathbf{z}_i^{(t)}, \mathbf{z}_j^{(t)}, \mathbf{e}_{ij}\right) \tag{6}$$

$$\hat{\mathbf{h}}_i^{(t)} = P_r\left(\mathbf{z}_i^{(t)}, \mathbf{m}_i^{(t)}\right) \tag{7}$$

$$\mathbf{g}_i^{(t)} = P_g(\mathbf{z}_i^{(t)}, \mathbf{m}_i^{(t)}) \tag{8}$$

$$\mathbf{h}_i^{(t)} = \mathbf{g}_i^{(t)} \odot \hat{\mathbf{h}}_i^{(t)} + (1 - \mathbf{g}_i^{(t)}) \odot \hat{\mathbf{h}}_i^{(t-1)} \tag{9}$$

where $\mathbf{u}_i$ and e_{ij} are the node and edge input features[4], $\boldsymbol{h}_i^{(t)}$ is the latent state of node i at timestep t (with $\boldsymbol{h}_i^{(0)} = \mathbf{0}$), $\|$ and $\odot$ denotes concatenation and elementwise multiplication. P_m, P_g and P_r are the processor's message, gating and readout functions. P_m is parametrised as an MLP, the rest are linear projection layers. Eqs (6)-(9) can also be viewed as a processor function $P(\mathbf{H}^{(t-1)}, \mathbf{U}, \mathbf{E})$ taking node ($\mathbf{U}$), edge ($\mathbf{E}$) and previous latent ($\mathbf{H}^{(t-1)}$) features. Noting that $\boldsymbol{u}_i$ and $\boldsymbol{e}_{ij}$ do not change between iterations for all i and j, the fixed point of the rollout of Eqs (6)-(9) can be expressed as:

$$\mathbf{H}^{(*)} = P_{\mathbf{UE}}(\mathbf{H}^{(*)}) \tag{10}$$

where $P_{\mathbf{UE}}(\mathbf{H}^{(*)}) = P(\mathbf{H}^{(*)}, \mathbf{U}, \mathbf{E})$. We want to emphasise that for a given problem instance $\mathbf{U}$ and $\mathbf{E}$ do

[4] Obtained by linearly encoding algorithm's inputs

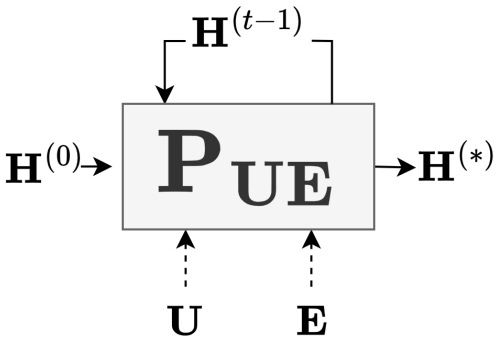

Figure 1. The deep equilibrium algorithmic reasoner. Dashed arrows represent constant variables.

not change during the search for equilibrium. The above Eq. (10) matches the signature of Eq. (4), is solved via root-finding (as if it is f_θ of a DEQ) and is what we will call the *deep equilibrium algorithmic reasoner* (DEAR) in our experiments. The rest of NAR architecture, i.e. encoders and decoders are implemented as in Ibarz et al. [14].

A note on underreaching In the DEQ implementation we used[5] *one step of the solver calls the GNN processor exactly once*. It is in theory possible that the number of solver iterations needed to find equilibrium is less than the diameter of the graph, resulting in underreaching: the information from one node (e.g. starting node in a shortest-path algorithm) cannot reach a target node. In our experiments, however, the solver either needed more iterations than the ground-truth or, if it needed less, was on data instances that are represented as full-graphs (sorting in CLRS).

A note on oversmoothing A similar argument can be made, but in the opposite direction – what happens if the solver has to use more calls/iterations and does this lead to oversmoothing? First, we did not empirically observe such an issue – if all $\boldsymbol{h}_i^{(*)}$ are equivalent, we would be decoding the same outputs across all nodes/edges and accuracy/loss will suffer. Second, our GNN model makes use of gating (cf. Eq. (9)). Although less intricate than works targeting oversmoothing, according to a recent survey on GNN [19] gating is one of the strategies to avoid it.

A note on hints As we already mentioned, in Eqs (6)-(9) the node and edge features are the same across timesteps and this is due to the fact we do not make any use of hints (i.e. predicting intermediate algorithm state). First, although it may seem counterintuitive, it has been shown

[5] `torchdeq` [10], MIT License

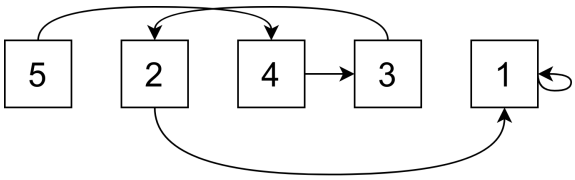

Figure 2. An example input list and predecessors indicating order. Source: Ibarz et al. [14]

that a NAR model can successfully generalise, even when trained to only predict the correct output [4]. Second, the fact that the solver uses the GNN exactly once per call *does not imply that one step of the solver corresponds to one iteration of the algorithm*. Adding hint-based loss, be it directly supervising on the outputs or performing contrastive learning, based on hint values as in Bevilacqua et al. [4], is ambiguous as there is no one-to-one correspondence of solver iteration and step of the algorithm and is left for future work.

4. Evaluation

4.1. Setup

For each algorithm we generated 10^5/100/100-sized training/validation/test datasets. Training samples sizes vary between 8 and 16 elements (uniformly randomly chosen), validation and test samples are of size 16 and 64 respectively. For requiring graphs as input (non-sorting tasks in our experiments) we generate Erdős–Rényi graphs [9] with edge probabilities p uniformly randomly sampled from the interval $[0.1, 0.9]$, with increments of 0.1. We obtained the ground truth execution trajectories and targets using the CLRS-30 implementation.[6]

In our experiments the models have a latent dimensionality of 128, the batch size is 32, learning rate is 3×10^{-4} and we use the Adam optimizer [15]. We train our algorithmic reasoners for 100 epochs, choosing the model with the lowest validation loss for testing. Each task is independently learned, minimising the output loss (losses depend on the algorithm, cf. CLRS-30). We did not include hint trajectories in the loss computation, since as already mentioned in §3.2 adding supervision on hints is ambiguous.

The performance metric we measure is the out-of-distribution accuracy. The definition of accuracy varies between algorithms and is based on the specification of the algorithm itself (see Ibarz et al. [14] and CLRS-30 for all possible accuracy metrics). For the four algorithms we evaluate on, the accuracy metric is going to be the node/edge pointer accuracy:

- Bellman-Ford – predecessor node pointer accuracy – for

[6] `https://github.com/google-deepmind/clrs/`, Apache-2.0 License

Table 1. Performance metrics for different algorithms and models. Note that for DEAR models the type of sorting algorithm should not matter as the output target is always the same.

	Algorighm			
Model	Bellman-Ford	Floyd-Warshall	SCC	(Insertion) Sort
NAR[7] (Triplet-MPNN)	$93.26 \pm 0.04\%$	$40.80 \pm 2.90\%$	$57.63 \pm 0.68\%$	$77.29 \pm 7.42\%$
NAR[8]	$97.03 \pm 0.10\%$	$52.58 \pm 1.08\%$	$40.82 \pm 0.75\%$	$63.20 \pm 11.32\%$
NAR[8] (w/ jac. reg)	$97.02 \pm 0.08\%$	$53.34 \pm 1.51\%$	$39.95 \pm 1.12\%$	$52.44 \pm 10.07\%$
DEAR	$97.50 \pm 0.29\%$	$52.79 \pm 0.32\%$	$42.04 \pm 0.93\%$	$85.29 \pm 6.13\%$
DEAR (rel. tol. &lower ϵ)	$96.78 \pm 0.43\%$	$52.39 \pm 1.34\%$	$43.38 \pm 0.39\%$	$86.93 \pm 3.87\%$

each node u predict the immediate predecessor on the shortest-path from the starting node to u

- Floyd-Warshall – predecessor *edge* pointer accuracy – similar to the above. For each edge $(i, j) \in \mathcal{E}$ predict the node k, such that $k = \arg\min_{k \in \mathcal{V}} dist[i][k] + dist[k][j]$
- Strongly connected components – node pointer accuracy – for each node predict a pointer to the immediate predecessor in the same component. If more than one such predecessors exist, chose the one with the lowest index.
- Insertion sort – node pointer accuracy – the sorting array iq represented by the vector of pointers π. π_i points the the array element before i in the sorted sequence. E.g. in the example (Fig. 2) if we are given $a = [5, 2, 4, 3, 1]$ as input $pi_0 = 2$ because $a[0] = 5$ and $a[2] = 4$.

The main baseline we compare against is a NAR architecture with the same processor trained in the no-hint regime as in Bevilacqua et al. [4], however, we also provide a comparison with a more expressive processor. Unless specified, DEARs employ the Anderson root-finding method with default parameters from the `torchdeq` library and include Jacobian regularization [3]. Standard deviations are based on 3 seeds.

4.2. Results

We currently provide results for 4 algorithms: Bellman-Ford, Floyd-Warshall, Strongly Connected Components (SCC) and Sorting (insertion sort, in particular) [6]. Results are presented in Table 1. For completeness, although it uses a more expressive processor, we also copy the results for the corresponding no-hint implementation from [4].

[7] As reported in Bevilacqua et al. [4]. Note that the processor in their experiments is Triplet-MPNN [8].

[8] Run with our framework (processor is PGN; Eqs (6)-(9)). We account the improved shortest-path results to our framework's requirement that a pointer must be a graph edge which is a strong inductive bias for sparse graphs.

Table 2. Test inference times (seconds/sample; A100 GPU) for NAR and DEAR (rel.tol & lower ϵ) . As runtimes for DEAR depend on model weights, we report standard deviation across three seeds.

	Model	
Algorithm	**NAR**[8]	**DEAR** (rel.tol & lower ϵ)
Bellman-Ford ↑	0.028	0.054 ± 0.004
Floyd-Warshall ↓	0.211	0.146 ± 0.009
SCC ↓	0.999	0.055 ± 0.003
Insertion Sort ↓	1.458	0.050 ± 0.002

Our model performed on par with the baseline no-hint model for shortest-path finding algorithms. It further outperformed the corresponding baseline on SCC, even though it fell short of the Triplet-MPNN architecture. We further ran an experiment adding the Jacobian regularisation term used for DEARs and confirmed that results are not only due to the added regularisation.

What struck us the most was the fact that on sorting we outperformed both the baseline and the Triplet-MPNN architecture by a substantial margin. Noticing that `torchdeq` uses absolute tolerance of $\epsilon = 10^{-}3$ when deciding when to stop the root-finding method we decided to test the resilience of the method to the stopping criteria. The last result in Table 1 uses the same setup but stops when the *relative* error drops below $\epsilon = 0.1$. The results did not change significantly, however, the processing speed improved dramatically, outperformed NAR processing speeds on 3/4 algorithms (Table 2) and achieving more than 10-fold improvement on sorting, while retaining accuracy. Since for sorting datapoints are represented as full graphs, we conjecture that the solver learns to perform some form of parallel sorting. Proving this however, requires substantial amount of work and is left for the future.

References

[1] Donald G. Anderson. Iterative procedures for nonlinear integral equations. *Journal of the ACM (JACM)*, 1965. 2

[2] Shaojie Bai, J. Z. Kolter, and V. Koltun. Deep equilibrium models. *Neural Information Processing Systems*, 2019. 2

[3] Shaojie Bai, V. Koltun, and J. Z. Kolter. Stabilizing equilibrium models by jacobian regularization. *International Conference on Machine Learning*, 2021. 4

[4] Beatrice Bevilacqua, Kyriacos Nikiforou, Borja Ibarz, Ioana Bica, Michela Paganini, Charles Blundell, Jovana Mitrovic, and Petar Velickovic. Neural algorithmic reasoning with causal regularisation. *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, 202:2272–2288, 2023. 1, 3, 4

[5] C. G. Broyden. A class of methods for solving nonlinear simultaneous equations. *Mathematics of Computation*, 19: 577–593, 1965. 2

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. 2, 4

[7] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms, 3rd edition*, pages 1111–1117. MIT press, 2009. 1

[8] Andrew J Dudzik and Petar Veličković. Graph neural networks are dynamic programmers. *Advances in Neural Information Processing Systems*, 35:20635–20647, 2022. 1, 4

[9] Paul Erdős, Alfréd Rényi, et al. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5(1):17–60, 1960. 3

[10] Zhengyang Geng and J. Zico Kolter. Torchdeq: A library for deep equilibrium models. https://github.com/locuslab/torchdeq, 2023. 3

[11] L. Ghaoui, Fangda Gu, Bertrand Travacca, Armin Askari, and Alicia Y. Tsai. Implicit deep learning. *SIAM Journal on Mathematics of Data Science*, 2019. 2

[12] Fangda Gu, Heng Chang, Wenwu Zhu, S. Sojoudi, and L. Ghaoui. Implicit graph neural networks. *Neural Information Processing Systems*, 2020. 2

[13] 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. 2

[14] Borja Ibarz, Vitaly Kurin, George Papamakarios, Kyriacos Nikiforou, Mehdi Bennani, Róbert Csordás, Andrew Joseph Dudzik, Matko Bosnjak, Alex Vitvitskyi, Yulia Rubanova, Andreea Deac, Beatrice Bevilacqua, Yaroslav Ganin, Charles Blundell, and Petar Velickovic. A generalist neural algorithmic learner. In *Learning on Graphs Conference, LoG 2022, 9-12 December 2022, Virtual Event*, page 2. PMLR, 2022. 1, 2, 3

[15] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. 3

[16] Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956. 1

[17] Vladimir V Mirjanic, Razvan Pascanu, and Petar Veličković. Latent space representations of neural algorithmic reasoners. In *The Second Learning on Graphs Conference*, 2023. 2

[18] Gleb Rodionov and Liudmila Prokhorenkova. Neural algorithmic reasoning without intermediate supervision. *arXiv preprint arXiv:2306.13411*, 2023. 1

[19] T. Konstantin Rusch, Michael M. Bronstein, and Siddhartha Mishra. A survey on oversmoothing in graph neural networks. *arXiv preprint arXiv: 2303.10993*, 2023. 3

[20] Hao Tang, Zhiao Huang, Jiayuan Gu, Bao-Liang Lu, and Hao Su. Towards scale-invariant graph-related problem solving by iterative homogeneous gnns. *Advances in Neural Information Processing Systems*, 33, 2020. 1

[21] Petar Velickovic and Charles Blundell. Neural algorithmic reasoning. *Patterns*, 2(7):100273, 2021. 1

[22] Petar Veličković, Lars Buesing, Matthew Overlan, Razvan Pascanu, Oriol Vinyals, and Charles Blundell. Pointer graph networks. *Advances in Neural Information Processing Systems*, 33:2232–2244, 2020. 2

[23] Petar Veličković, Adrià Puigdomènech Badia, David Budden, Razvan Pascanu, Andrea Banino, Misha Dashevskiy, Raia Hadsell, and Charles Blundell. The clrs algorithmic reasoning benchmark. In *International Conference on Machine Learning*, pages 22084–22102. PMLR, 2022. 1, 2

[24] Petar Veličković. Neural algorithmic reasoning. *The Gradient*, 2023. 1

[25] Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S. Du, Ken ichi Kawarabayashi, and Stefanie Jegelka. What can neural networks reason about? In *International Conference on Learning Representations*, 2020. 1