Parallelising Differentiable Algorithms Removes the Scalar Bottleneck: A Case Study

Euan Ong [1] **Ferenc Huszár** [* 1] **Pietro Liò** [* 1] **Petar Veličković** [* 1 2]

Abstract

While differentiable algorithms are a popular way to imbue neural networks with an algorithmic inductive bias, it's been hypothesised that their performance is limited by the 'scalar bottleneck': the requirement that rich latent states be projected to scalars in order to be used as algorithmic inputs. This motivated the development of neural algorithmic processors (NAPs), neural networks that imitate algorithms in high-dimensional space, as a way to avoid this bottleneck while still retaining an algorithmic inductive bias. So far, however, there has been little work exploring whether this bottleneck exists in practice, and if so, the extent to which NAPs successfully mitigate it. Here, we present a case study of the scalar bottleneck on a new synthetic dataset inspired by recent work in neural algorithmics. While we found that the differentiable algorithm we tested did indeed suffer from a 'scalar bottleneck', we also found that this bottleneck was not alleviated by frozen NAPs, but rather by simply using an unfrozen, algorithmically-aligned neural network. Based on these results, we hypothesise that the problem might be better thought of as an 'ensembling bottleneck', caused by the inability to execute multiple instances of the same algorithm in parallel. We thus develop the parallel differentiable algorithmic black-box (pDAB), which preserves the efficiency and correctness guarantees of its scalar counterpart, while avoiding the scalar bottleneck.

1. Introduction

Algorithms are used to automatically solve *complex real-world problems* (e.g. finding the shortest path between two cities), provided we can model them mathematically (e.g. as a graph with edge weights in $\mathbb{R}$). In machine learning, many tasks we wish to solve (e.g. finding the shortest path between two cities given weather and traffic conditions) are *algorithmic in nature* but *hard to model mathematically*. As such, there is much interest in building neural networks with an **algorithmic inductive bias** (i.e. incentivised to learn computations that 'look like' those of some algorithm A) through the use of **algorithmic modules**: differentiable functions, imitating the behaviour of a particular algorithm, usable as components within a larger neural network.

One popular way to build an algorithmic module is to take an implementation of algorithm A and simply make it differentiable, thereby constructing a **scalar differentiable algorithmic black-box (sDAB)** [1]–[4] that can be used as a module in a larger network. Veličković and Blundell [5], however, claim that the performance of sDABs is impaired by what they call the **scalar bottleneck**: the requirement that we project rich latent states down to single scalar inputs.

As such, they propose the alternative approach of *neural algorithmic reasoning* [5]: training a neural network $\hat{A}$ to be a **neural algorithmic processor (NAP)** imitating A in high-dimensional space, freezing it, and using it as a module in a larger network. Although NAPs require costly pre-training and lack the correctness guarantees of sDABs (especially out-of-distribution [6]), Veličković and Blundell [5] claim that NAPs should alleviate the scalar bottleneck as they operate over a high-dimensional latent space.

So far, however, there has been little work exploring whether this scalar bottleneck exists in practice, and if so, whether NAPs successfully remove it. As such, we perform a case study of the scalar bottleneck phenomenon in sDABs and NAPs, in the context of the synthetic **WARCRAFT-SHORTEST-PATH-TREE** dataset (requiring models to find the shortest-path tree from an image of a $k \times k$ Warcraft terrain map). Our contributions are as follows:

1. Contrary to Veličković and Blundell [5], we find that **sDABs and frozen NAPs suffer from the scalar bottleneck**, but not unfrozen NAPs or ANNs (Section 3.1).
2. We therefore hypothesise that **this 'scalar bottleneck' is better thought of as an *ensembling* bottleneck**, caused by the inability to learn to execute multiple in-

---

[*] Equal advising [1]University of Cambridge [2]Google DeepMind. Correspondence to: Euan Ong <euan@ong.ac>.

Published at the 2nd Differentiable Almost Everything Workshop at the 41st International Conference on Machine Learning, Vienna, Austria. July 2024. Copyright 2024 by the author(s).

stances of the same algorithm in parallel (Section 3.2).

3. In line with our hypothesis, we find that **a parallelised version of our sDAB outperforms all other models tested** both in and out of distribution, giving us the efficiency and correctness guarantees of sDABs without their performance bottleneck (Section 4).

2. Our benchmark environment

For our case study, we must choose both a problem with an appropriate algorithmic prior, and the NAPs and sDABs we wish to compare. To study the scalar bottleneck in a controlled environment representative of the literature, we design the synthetic **WARCRAFT-SHORTEST-PATH-TREE** benchmark based on a popular environment for testing sDABs [1]. This benchmark evaluates NAPs and sDABs by using them as 'algorithmic modules' within a larger neural network, which we train to find the shortest-path tree from a *Warcraft II* terrain map (Figure 1).

Architectural details. This network uses the first five layers of ResNet-18 [8] to extract a $k \times k$ grid of latent features, runs the algorithmic module over a grid graph constructed from these features, and applies the pointer decoder from Veličković, Badia, Budden, *et al.* [9] to extract the predecessor node for each cell. For more details, see Appendix A.

Choice of algorithmic module. Note that, to avoid the confounding effects of different continuous relaxation methods, our chosen problem's underlying algorithm should not require any continuous relaxations to be used as an sDAB. Indeed, WARCRAFT-SHORTEST-PATH-TREE admits solutions using the **Bellman-Ford** algorithm, through which we can differentiate directly. As such, we compare GNN-based NAPs trained to imitate Bellman-Ford (as per the CLRS benchmark [9]), and 'natively differentiable' sDABs implementing Bellman-Ford (following [4]).

Problem variants. We explore two different variants of this problem: the simpler **optimal** variant, where we train our model to predict a distribution equally weighted over all optimal shortest-path predecessors, and the more complex **tie-breaking** variant, where we train our model to break ties between optimal predecessors in a deterministic way (see Appendix B). Note that, as neither our sDAB nor our NAP break ties in this way, the tiebreaking environment lets us explore the case where the underlying algorithm of our problem differs slightly from our algorithmic prior.

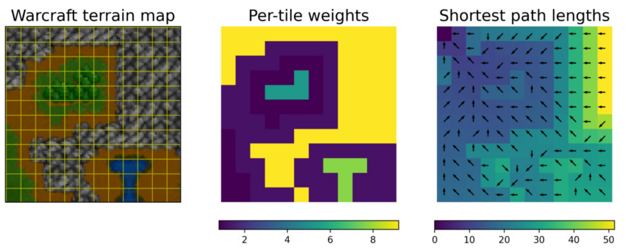


Figure 1: An example *Warcraft II* terrain map [7], the cost to traverse each tile in its underlying grid, and the length of the shortest path from the top left tile to each tile in the grid.

Metrics. We assessed each model on either **exact tree-accuracy** (i.e. the % of grids with all predecessors correctly predicted) or **optimal tree-accuracy** (i.e. the % of grids for which all predicted predecessor distributions in that grid have an optimal pointer as their mode) as appropriate. We report model performance through bootstrapped 95% confidence intervals (CIs) for mean (exact / optimal) tree-accuracy, and compare models through bootstrapped 95% CIs for probability-of-improvement (PoI).

Hyperparameters, training and evaluation. For each algorithmic module tested, across each problem variant, we performed 5 training runs with different seeds. For each run, we trained on maps of size 12×12, and periodically evaluated on maps of size 18×18. We reported the results of the highest-performing checkpoint of each run on the test sets of Vlastelica, Paulus, Musil, *et al.* [1], assessing in-distribution (12×12) and out-of-distribution (18×18) performance. For full details, see Appendix C.

3. Comparing modules: both NAPs and sDABs suffer from the 'scalar bottleneck'

We now compare algorithmic modules in the WARCRAFT-SHORTEST-PATH-TREE environment, evaluating the performance of non-algorithmic baselines, NAPs and sDABs, and exploring the effect of both *unfreezing* our NAPs, and *randomly initialising their weights*.

3.1. Results

Algorithmic modules beat non-algorithmic baselines. To check the correctness of our environment, we first compare the performance of sDABs and NAPs against non-algorithmic baselines. We evaluate our algorithmic modules against both ResNet-18 [8] and a baseline where we only use the feature extractor. To verify that our NAPs are properly trained, we also evaluate them against frozen, randomly-initialised GNNs. As per Figure 2, we see that *both sDABs and NAPs outperform all three of our baselines*. Specifically, we observe from Figure 3 that both sDABs and NAPs outperform ResNet-18 (left), that NAPs (i.e. frozen, pre-trained GNNs) outperform frozen, randomly-initialised GNNs (right), and that ablating the executor from WARCRAFT-NET does impair its performance (middle).

Frozen NAPs do not outperform sDABs. But, although both NAPs and sDABs outperform algorithmic baselines, *NAPs do not substantially outperform sDABs* on either the optimal or tiebreaking environments (Figure 4).

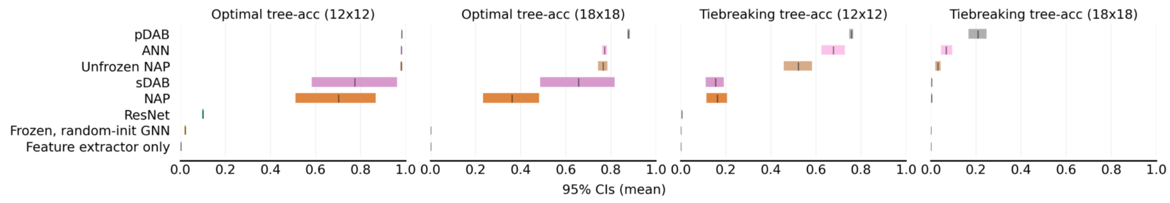


Figure 2: 95% CIs for mean tree accuracy, in-distribution (12x12) and out of distribution (18x18), in both environments.

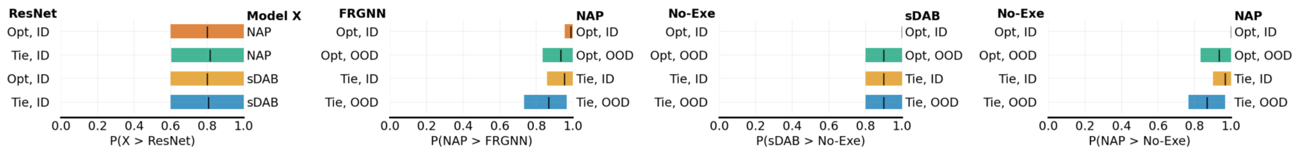


Figure 3: 95% CIs for PoI in tree-accuracy for sDABs and NAPs over various non-algorithmic baselines, in-distribution (ID) and out-of-distribution (OOD), in the optimal (Opt) and tie-breaking (Tie) environments.

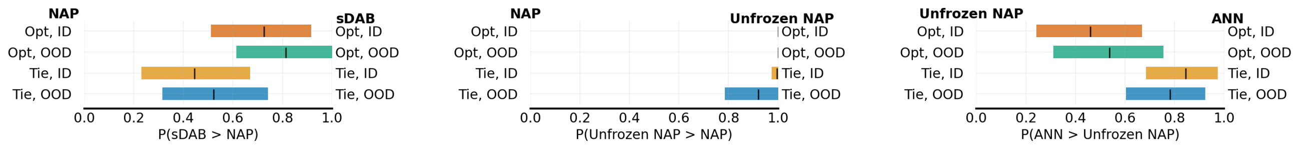


Figure 4: 95% CIs for PoI in tree-accuracy for sDABs over NAPs, for unfrozen NAPs over NAPs, and for ANNs over unfrozen NAPs, in-distribution (ID) and out-of-distribution (OOD), in the optimal (Opt) and tie-breaking (Tie) environments.

Unfreezing NAPs improves their performance – but they're still no better than a randomly-initialised GNN. As NAPs are simply frozen GNNs pre-trained on abstract algorithmic tasks, we explore the effect of *unfreezing* them during training. We also contrast their performance with unfrozen, randomly-initialised GNNs: as GNNs have been shown to align with Bellman-Ford [10], we consider these to be **algorithmically-aligned neural networks (ANNs)**. We observe from Figure 4 that, contrary to established wisdom [11] (but in line with the more recent observations of Georgiev, Numeroso, Bacciu, *et al.* [12]), *unfreezing NAPs substantially improves their performance* across all environments, with the largest performance improvements observed in the (more algorithmically-aligned) optimal environment. But we also observe that *our ANNs match or beat the performance of unfrozen NAPs*: while ANNs perform comparably to NAPs in the optimal environment, they substantially outperform NAPs in the tiebreaking environment.

Conclusions: NAPs and sDABs suffer from the 'scalar bottleneck', but ANNs do not. Contrary to Veličković and Blundell [5], we found that *NAPs do not outperform sDABs in either environment* – and seem to suffer from the same instability issues as sDABs. Moreover, we find that *both unfrozen NAPs and ANNs have better stability and performance than either NAPs or sDABs* (even out-of-distribution). And, on the more complex tiebreaking task, it appears that *introducing a parameter-based algorithmic prior is actively harmful to performance*. So, in this environment, it is very likely that NAPs trained as per Veličković, Badia, Budden, *et al.* [9] do not alleviate the scalar bottleneck of sDABs, and that this bottleneck can instead be overcome by ANNs.

3.2. Understanding our results: developing hypotheses on the nature of the scalar bottleneck

Now, these results leave us with a compelling question: *why are sDABs outperformed by ANNs, but not by NAPs?* We list two hypotheses, which we explore in the next section:

The ensembling-bottleneck hypothesis. ANNs outperform sDABs and frozen NAPs because they can learn to perform *many versions of the algorithm in parallel*, over *simple (possibly scalar) representations*. (Indeed, there is some evidence to suggest that neural networks solving complex problems 'in the wild' learn multiple independent modules performing the same algorithm in parallel [13].)

The expressivity-bottleneck hypothesis. ANNs outperform sDABs and frozen NAPs because they can learn *different variants of the algorithm*, over *complex representations*, that map more closely to the exact problem at hand.

4. Testing the ensembling-bottleneck hypothesis: the power of parallel DABs

One way to test the ensembling-bottleneck hypothesis (i.e. that *increasing algorithmic parallelism without increasing network expressivity* can improve model performance) is to simply modify our sDAB to execute an ensemble of algorithms in parallel, and to compare the performance of the

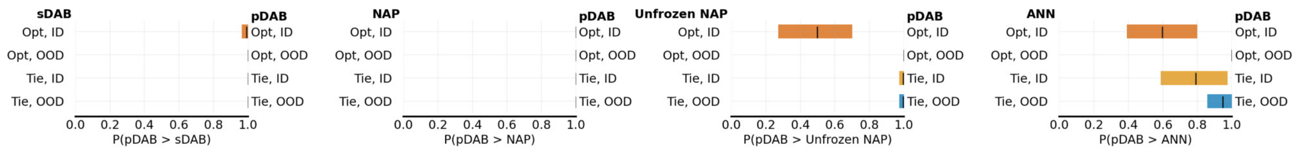


Figure 5: 95% CIs for probabilities of improvement in tree-accuracy for pDABs over all other modules tested, both in-distribution (ID) and out-of-distribution (OOD), in the optimal (Opt) and tie-breaking (Tie) environments.

resulting **parallel DAB (pDAB)** to that of other algorithmic modules in the optimal environment. As such, we built a pDAB for Bellman-Ford as illustrated in Algorithm 1, and tested it under all conditions explored in Section 3.

pDABs dominate all other algorithmic modules across all environments. Observe from Figure 2 that *pDABs dominate all other algorithmic modules in the optimal environment*. We see that pDABs very substantially outperform both sDABs (top left) and NAPs (top right), in and out of distribution. And, while pDABs perform on par with unfrozen NAPs (and marginally better than ANNs) in-distribution, they very substantially outperform both out-of-distribution. Moreover, we observe that *this result holds even in the tie-breaking environment*, where Bellman-Ford alone should not be sufficient to solve the problem. This is surprising, as no positional information is passed to the pDAB.

The high dimensionality of pDABs can be used in unanticipated ways to handle variant algorithms. To understand why pDABs do so well in the tiebreaking environment, we visualise the per-tile weight and initial distance matrices they receive as input (Figure 6). While some dimensions of the pDAB are used to predict the true shortest-path lengths for each tile, others appear to be used to generate vertical and horizontal gradients. As these artefacts only appear in pDABs trained on the tiebreaking problem, we hypothesise that our models have learned to use the extra dimensions of the pDAB in an unexpected way, to generate *robust positional encodings* for tiebreaking.

Algorithm 1 A d-dimensional Bellman-Ford pDAB, for use as a module in WARCRAFT-NET (Algorithm 2).

```
Require: The following learnable linear layers:
        $enc_w : \mathcal{I} \to \mathbb{R}^d$    $dec_h : \mathbb{R}^d \to \mathcal{O}_n$
        $enc_d : \mathcal{I} \to \mathbb{R}^d$    $dec_e : \mathcal{I} \to \mathcal{O}_e$
 1: def PDAB($G(\{\mathbf{h}_i\}, \{\mathbf{e}_{ij}\}) : G[\mathcal{I}, \mathcal{I}]) : G[\mathcal{O}_n, \mathcal{O}_e]$
 2:     # Get stack of weights from edge features
 3:     $\mathbf{w}_{ij} : \mathbb{R}^d \leftarrow enc_w(\mathbf{e}_{ij})$
 4:     # Get stack of initial distances from node features
 5:     $\mathbf{d}_i^{(0)} : \mathbb{R}^d \leftarrow 100 \cdot \sigma(enc_d(\mathbf{h}_i))$
 6:     # Perform BF on stack of d grid graphs
 7:     for $t \leftarrow 1, ..., (k \times k)$ do
 8:         $\mathbf{d}_j^{(t)} = \min(\mathbf{d}_j^{(t-1)}, \min_{i \to j}(\mathbf{d}_i^{(t-1)} + \mathbf{w}_{ij}))$
 9:     # Map final distances to latent space
10:     return $G(\{dec_h(\mathbf{d}_i^{(k \times k)})\}, \{dec_e(\mathbf{e}_{ij})\})$
```

pDABs are much more efficient to train than ANNs. Finally, we observe that, not only do pDABs not require pre-training, but they are much more efficient to train than ANNs. Indeed, training pDABs incurred an average time per epoch of 11.4 ± 1.7 seconds (across 15 runs); while slightly more compute-intensive than sDABs (9.2 ± 1.4 seconds), they are *over four times faster* than both NAPs (42.5 ± 1.6 seconds) and ANNs (51.4 ± 0.7 seconds).

Conclusions: our results support the ensembling-bottleneck hypothesis. So, as pDABs dominate all other models in the optimal environment, we have strong evidence supporting the ensembling-bottleneck hypothesis in this environment – specifically, that *increasing the dimensionality of sDABs* is enough to make them match the performance of ANNs in-distribution. But we also observed that pDABs are much more efficient than ANNs, generalise much better than ANNs out-of-distribution, and even outperform ANNs (which should, in principle, be more flexible than pDABs) on problems whose underlying algorithm deviates slightly from our algorithmic prior. As such, while we leave a systematic exploration of pDABs across a range of architectures and domains to future work, we are potentially close to a long-standing goal of neural algorithmics [14]: developing a way to *deterministically distill an algorithm into a robust, high-dimensional processor network* that preserves both the efficiency and correctness guarantees of sDABs while avoiding their performance bottleneck.

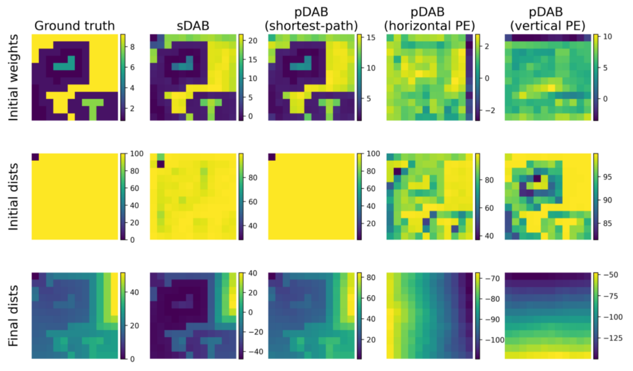


Figure 6: The ground-truth initial weight, initial distance and final distance matrices for the terrain map from Figure 1, alongside the inputs that we learn to pass to our sDAB, and representative examples of the three main classes of inputs that we learn to pass to individual Bellman-Ford instances within our pDAB. (See Appendix D for more details.)

References

[1] M. Vlastelica, A. Paulus, V. Musil, G. Martius, and M. Rol´inek, "Differentiation of Blackbox Combinatorial Solvers," Dec. 2019. [Online]. Available: http://arxiv.org/abs/1912.02175.

[2] G. Farquhar, T. Rocktäschel, M. Igl, and S. Whiteson, "TreeQN and ATreeC: Differentiable Tree-Structured Models for Deep Reinforcement Learning," *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, Oct. 2017. DOI: 10.48550/arxiv.1710.11417. [Online]. Available: https://arxiv.org/abs/1710.11417v2.

[3] P. W. Wang, P. L. Donti, B. Wilder, and Z. Kolter, "SATNet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver," *36th International Conference on Machine Learning, ICML 2019*, vol. 2019-June, pp. 11 373–11 386, May 2019. [Online]. Available: https://arxiv.org/abs/1905.12149v1.

[4] F. Petersen, C. Borgelt, H. Kuehne, and O. Deussen, "Learning with Algorithmic Supervision via Continuous Relaxations," *Advances in Neural Information Processing Systems*, vol. 20, no. NeurIPS, pp. 16 520–16 531, 2021, ISSN: 10495258.

[5] P. Veličković and C. Blundell, "Neural Algorithmic Reasoning," *Patterns*, vol. 2, no. 7, pp. 1–7, May 2021, ISSN: 26663899. DOI: 10.1016/j.patter.2021.100273. [Online]. Available: http://arxiv.org/abs/2105.02761%20http://dx.doi.org/10.1016/j.patter.2021.100273.

[6] K. Xu, M. Zhang, J. Li, S. S. Du, K.-i. Kawarabayashi, and S. Jegelka, "How Neural Networks Extrapolate: From Feedforward to Graph Neural Networks," Sep. 2020. [Online]. Available: https://arxiv.org/abs/2009.11848v5.

[7] J. Guyomarch, *Warcraft II Open-Source Map Editor*, 2017.

[8] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2016-December, pp. 770–778, Dec. 2015, ISSN: 10636919. DOI: 10.1109/CVPR.2016.90. [Online]. Available: https://arxiv.org/abs/1512.03385v1.

[9] P. Veličković, A. P. Badia, D. Budden, *et al.*, "The CLRS Algorithmic Reasoning Benchmark," May 2022. DOI: 10.48550/arxiv.2205.15659. [Online]. Available: https://arxiv.org/abs/2205.15659v2.

[10] A. Dudzik and P. Veličković, "Graph Neural Networks are Dynamic Programmers," pp. 1–9, 2022. [Online]. Available: http://arxiv.org/abs/2203.15544.

[11] A. Deac, P. Veličković, O. Milinkov´ic, P. L. Bacon, J. Tang, and M. Nikol´ic, "Neural Algorithmic Reasoners are Implicit Planners," *Advances in Neural Information Processing Systems*, vol. 19, pp. 15 529–15 542, Oct. 2021, ISSN: 10495258. DOI: 10.48550/arxiv.2110.05442. [Online]. Available: https://arxiv.org/abs/2110.05442v1.

[12] D. Georgiev, D. Numeroso, D. Bacciu, and P. Liò, "Neural Algorithmic Reasoning for Combinatorial Optimisation," 2023. [Online]. Available: http://arxiv.org/abs/2306.06064.

[13] K. Wang, A. Variengien, A. Conmy, B. Shlegeris, and J. Steinhardt, "Interpretability in the Wild: a Circuit for Indirect Object Identification in GPT-2 small," Nov. 2022. DOI: 10.48550/arxiv.2211.00593. [Online]. Available: https://arxiv.org/abs/2211.00593v1.

[14] Q. Cappart, D. Chételat, E. B. Khalil, A. Lodi, C. Morris, and P. Veličković, "Combinatorial Optimization and Reasoning with Graph Neural Networks," pp. 4348–4355, 2021. DOI: 10.24963/ijcai.2021/595.

[15] K. Xu, J. Li, M. Zhang, S. S. Du, K.-i. Kawarabayashi, and S. Jegelka, "What Can Neural Networks Reason About?," May 2019. [Online]. Available: https://arxiv.org/abs/1905.13211v4.

[16] T. N. Kipf and M. Welling, "Semi-Supervised Classification with Graph Convolutional Networks," in *International Conference on Learning Representations*, 2017. [Online]. Available: https://openreview.net/forum?id=SJU4ayYgl.

[17] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph Attention Networks," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=rJXMpikCZ.

[18] W. L. Hamilton, *Graph Representation Learning*. Morgan and Claypool, 2021, vol. 14, pp. 1–159.

[19] D. Numeroso, D. Bacciu, and P. Veličković, "Dual Algorithmic Reasoning," Feb. 2023. [Online]. Available: https://arxiv.org/abs/2302.04496v1.

[20] M. Vlastelica, M. Rol´inek, and G. Martius, "Neuro-algorithmic Policies enable Fast Combinatorial Generalization," Feb. 2021. [Online]. Available: https://arxiv.org/abs/2102.07456v1.

[21] S. S. Sahoo, A. Paulus, M. Vlastelica, V. Musil, V. Kuleshov, and G. Martius, "Backpropagation through Combinatorial Algorithms: Identity with Projection Works," May 2022. [Online]. Available: https://arxiv.org/abs/2205.15213v3.

[22] A. Paulus, M. Rol´inek, V. Musil, B. Amos, and G. Martius, "CombOptNet: Fit the Right NP-Hard Problem by Learning Integer Programming Constraints," May 2021. DOI: 10.48550/arxiv.2105.02343. [Online]. Available: https://arxiv.org/abs/2105.02343v2.

[23] Q. Berthet, M. Blondel, O. Teboul, M. Cuturi, J. P. Vert, and F. Bach, "Learning with Differentiable Perturbed Optimizers," *Advances in Neural Information Processing Systems*, vol. 2020-December, Feb. 2020, ISSN: 10495258. [Online]. Available: https://arxiv.org/abs/2002.08676v2.

[24] E. Ong, "Probing the foundations of neural algorithmic reasoning," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-990, Dec. 2023. DOI: 10.48456/tr-990. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-990.pdf.

[25] J. Lin, D. Campos, N. Craswell, B. Mitra, and E. Yilmaz, "Significant Improvements over the State of the Art? A Case Study of the MS MARCO Document Ranking Leaderboard," *SIGIR 2021 - Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 2283–2287, Feb. 2021. DOI: 10.1145/3404835.3463034. [Online]. Available: https://arxiv.org/abs/2102.12887v1.

[26] N. Reimers and I. Gurevych, "Reporting Score Distributions Makes a Difference: Performance Study of LSTM-networks for Sequence Tagging," *EMNLP 2017 - Conference on Empirical Methods in Natural Language Processing, Proceedings*, pp. 338–348, Jul. 2017. DOI: 10.18653/v1/d17-1035. [Online]. Available: https://arxiv.org/abs/1707.09861v1.

[27] R. Agarwal, M. Schwarzer, P. S. Castro, A. Courville, and M. G. Bellemare, "Deep Reinforcement Learning at the Edge of the Statistical Precipice," *Advances in Neural Information Processing Systems*, vol. 35, pp. 29 304–29 320, Aug. 2021, ISSN: 10495258. [Online]. Available: https://arxiv.org/abs/2108.13264v4.

[28] B. Efron, "Bootstrap Methods: Another Look at the Jackknife," *https://doi.org/10.1214/aos/1176344552*, vol. 7, no. 1, pp. 1–26, Jan. 1979, ISSN: 0090-5364. DOI: 10.1214/AOS/1176344552. [Online]. Available: https://projecteuclid.org/journals/annals-of-statistics/volume-7/issue-1/Bootstrap-Methods-Another-Look-at-the-Jackknife/10.1214/aos/1176344552.full%20https://projecteuclid.org/journals/annals-of-statistics/volume-7/issue-1/Bootstrap-Methods-Another-Look-at-the-Jackknife/10.1214/aos/1176344552.short.

[29] H. B. Mann and D. R. Whitney, "On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other," *https://doi.org/10.1214/aoms/1177730491*, vol. 18, no. 1, pp. 50–60, Mar. 1947, ISSN: 0003-4851. DOI: 10.1214/AOMS/1177730491. [Online]. Available: https://projecteuclid.org/journals/annals-of-mathematical-statistics/volume-18/issue-1/On-a-Test-of-Whether-one-of-Two-Random-Variables/10.1214/aoms/1177730491.full%20https://projecteuclid.org/journals/annals-of-mathematical-statistics/volume-18/issue-1/On-a-Test-of-Whether-one-of-Two-Random-Variables/10.1214/aoms/1177730491.short.

Acknowledgments

We thank Zhe Wang and Simon Osindero for reviewing the paper prior to submission, and all our anonymous reviewers for their careful feedback.

Glossary

ANN (algorithmically-aligned neural network) A neural network whose modules correspond to the subroutines of some target algorithm [15]. For this case study exploring Bellman-Ford, our ANN of choice is a graph neural network [10]. 1, 3, 4

GNN (graph neural network) neural networks acting over graphs $G = (V, E)$ (whose nodes u have one-hop neighbourhoods $\mathcal{N}_u = \{v \in V \mid (v, u) \in E\}$ and features $\mathbf{x}_u$), of the form

$$\mathbf{h}_u = \phi\left(\mathbf{x}_u, \bigoplus_{v\in\mathcal{N}_u} \psi(\mathbf{x}_u, \mathbf{x}_v)\right)$$

for ψ a learnable *message function*, ϕ a learnable *readout function* and $\oplus$ a permutation-invariant *aggregation function*. Note that this 'template' can be instantiated in many ways, with different choices of ϕ, ψ and $\oplus$ yielding popular architectures such as GCNs [16] and GATs [17]; for more background on GNNs, see e.g. [18]. 2, 3

NAP (neural algorithmic processor) A neural network $\hat{A}$ trained to imitate the action of an algorithm A in a high-dimensional latent space (e.g. [11], [19]). 1–4, 8–11

pDAB (parallel differentiable algorithmic black-box) Our modified sDAB that executes an ensemble of instances of the same algorithm in parallel. 4, 13

PoI (probability of improvement) For architectures A and B, the probability that a randomly-sampled training run for A will outperform a randomly-sampled training run for B. 2–4, 12

sDAB (scalar differentiable algorithmic black-box) A differentiable implementation of an algorithm A that can be used as a module within a larger neural network, so called because latent states must be projected to scalars before being passed through the module. These can be obtained either by using an algorithm that is differentiable almost everywhere (e.g. [4]), approximating the gradient of a non-differentiable algorithm (e.g. [1], [20]–[22]), or constructing a continuous relaxation of a discontinuous algorithm (e.g. [3], [4]). 1–4, 7–11

A. The WARCRAFT-SHORTEST-PATH benchmark

In this appendix, we describe the design of the WARCRAFT-SHORTEST-PATH benchmark. This benchmark, inspired by the work of Vlastelica, Paulus, Musil, *et al.* [1], allows for the comparison of algorithmic modules in the context of *finding the shortest-path tree for a Warcraft terrain map*; as such, we outline both the adaptations we made to the problem setting of Vlastelica, Paulus, Musil, *et al.* [1], and the architecture we used to compare algorithmic modules on this problem.

A.1. Adaptations made to prior work

As we're trying to explore the null hypothesis that NAPs encounter the scalar bottleneck but sDABs don't, in order to minimise the risk of false positives, we typically choose design decisions that are favourable towards the NAP.

The original problem setting. Recall that we based the design of this problem on WARCRAFT-SHORTEST-PATH, a popular problem for benchmarking sDABs [1], [4], [23]. This problem involved training a network (with an algorithmic inductive bias) to take as input an image of a $k \times k$ Warcraft terrain map, and return as output a $k \times k$ matrix indicating the cells of the terrain map involved in the *optimal shortest path* from the top left to the bottom right corner.

Exploring the problem in the context of Bellman-Ford. Note that, to solve this problem, we can use algorithmic modules with an algorithmic inductive bias (AIB) towards any single-source shortest path algorithm (e.g. Dijkstra [1] or Bellman-Ford [4]). As we wish to benchmark step-level NAPs against natively-differentiable sDABs, we choose to explore algorithmic modules with an AIB towards Bellman-Ford, a simple, natively-differentiable algorithm that aligns well with graph-based NAPs.

From shortest-path to shortest-path-tree. Observe, however, that to actually solve this problem, we need algorithmic modules that not only compute the minimum distance from the source to every other node, but also walk the resulting predecessor tree in $O(V)$ time in order to recover the actual shortest path from the top left to the bottom right cell. Now, while it is easy to add this postprocessing step to an sDAB, adding another $O(V)$ steps to a Bellman-Ford NAP could substantially impact performance. As such, to minimise the trajectory length of our NAP, and to align more closely with the version of Bellman-Ford used to train NAPs in the literature [9], we avoid this postprocessing overhead by instead supervising on the *shortest path tree* of per-node predecessors rooted in the top-left grid cell.

Removing the inductive bias of algorithmic supervision. Now, in its original form, WARCRAFT-SHORTEST-PATH is a problem of *algorithmic supervision* [4]: given an sDAB mapping a grid with weights to an indicator matrix representing the shortest path across it, we precompose this sDAB with a feature extractor (which should learn to predict a cost for each type of tile) and supervise directly on the shortest-path output of our sDAB. We note, however, that in most real-world problems with AIBs (which are ultimately where we want to apply NAR), we can't directly supervise on algorithmic outputs – instead, we must typically learn to *postprocess* (or project out relevant information from) the output of our algorithmic module. As such, we adapt our architecture by both pre-composing and post-composing our algorithmic module with learnable layers, and ensuring the outputs of our algorithmic modules require some mild post-processing in order to extract the final outputs.

A.2. The WARCRAFT-NET architecture

So, given our adapted problem of WARCRAFT-SHORTEST-PATH-TREE, we now outline the architecture we built to solve it. We define this architecture as a function WARCRAFT-NET : $Grid[(8k, 8k), 3] \rightarrow Grid[(8k, 8k), Categorical[8]]$, mapping $8k \times 8k \times 3$ images of Warcraft terrain maps to grids of categorical variables indicating the predecessor cell for each cell in the grid.

We present a pseudocode implementation of the WARCRAFT-NET architecture in Algorithm 2.

A high-level summary of Warcraft-Net. In a manner akin to other NAR architectures [11], our WARCRAFT-NET has four main components: *extracting a $k \times k$ grid of features* from the original image, *generating a latent graph* from these features, *applying an algorithmic module* to this latent graph, and *projecting out predecessor pointers* from this latent graph.

Extracting the grid of features. As per Vlastelica, Paulus, Musil, *et al.* [1], we extract features from our input image using the first five layers of *ResNet-18*. But while Vlastelica, Paulus, Musil, *et al.* [1] use this feature extractor to directly predict a $k \times k$ grid of cell costs which they pass to their sDAB, as we wish to use our architecture with either sDABs or NAPs, we instead return a $k \times k$ grid $\mathbf{f} : Grid[(k, k), \mathcal{I}]$ of latent per-cell features.

Generating the grid graph. Given such a grid $\mathbf{f}$, in order to compute its shortest-path tree, we must first generate its

Algorithm 2 A pseudocode implementation of WARCRAFT-NET for a $k \times k$ terrain map.

```
Require: The following learnable functions:
    $enc : \mathbb{R}^3 \to \mathcal{I} := Linear$
    $f_1, f_2 : \mathcal{O}_n \to \mathcal{P} := Linear$
    $f_e : \mathcal{O}_e \to \mathcal{P} := Linear$
    (alongside the first five layers of ResNet-18)

 1: def RESNET-FEATURE-EXTRACTOR($\mathbf{img} : Grid[(8k, 8k), 3]$) : $Grid[(8k, 8k), \mathcal{I}]$
 2:     $\mathbf{h} : Grid[(2k, 2k), 3] \leftarrow$ ResNet-18.(conv1 ▷ bn1 ▷ ReLU ▷ MaxPool ▷ layer1)($\mathbf{img}$)
 3:     $\mathbf{h}' : Grid[(2k, 2k), \mathcal{I}] \leftarrow \text{map}(enc, \mathbf{h})$
 4:     return $\text{maxPool}(\mathbf{h}', (2k, 2k) \to (k, k))$

 5: def BUILD-GRAPH($\mathbf{h} : Grid[(k, k), \mathcal{I}]$) : $G[\mathcal{I}, \mathcal{I}]$
 6:     $g \leftarrow \text{newGraph}(\text{nodes} = \{(y, x) \mid y, x \in [1..k]\})$
 7:     for $(y, x) \in [1..k] \times [1..k]$ do
 8:         $g.\text{nodes}[(y, x)] \leftarrow \mathbf{h}_{yx}$
 9:         $g.\text{addEdges}(\{(y', x') \xrightarrow{\mathbf{h}_{yx}} (y, x) \mid (y', x') \in \text{getAdjacent8Cells}(y, x)\})$
10:     return $g$

11: def PREDICT-POINTERS($G(\{\mathbf{h}_i\}, \{\mathbf{e}_{ij}\}) : G[\mathcal{O}_n, \mathcal{O}_e]$) : $Grid[(k, k), Categorical[8]]$
12:     ▷ For each node, compute a weighted distribution over its in-edges.
13:     $\pi_{i \to j} : \mathbb{R} \leftarrow f_m(\max[f_1(\mathbf{h}_j), f_2(\mathbf{h}_i) + f_e(\mathbf{e}_{ij})])$

14:     ▷ Convert these per-node weighted distributions to a grid of categoricals
15:       representing the direction $\{N, NE, E, SE, S, ...\}$ of the pointed-to in-edge
16:     $preds_{yx} \leftarrow \text{softmax}([\pi_{(y', x') \to (y, x)} \mid (y', x') \in \text{getAdjacent8Cells}(i, j)])$
17:     return $preds$

18: def ALGORITHMIC-MODULE($g : G[\mathcal{I}, \mathcal{I}]$) : $G[\mathcal{O}_n, \mathcal{O}_e]$
19:     ▷ A NAP or sDAB, wrapped in appropriate encoders and decoders.

20: def WARCRAFT-NET($\mathbf{img} : Grid[(8k, 8k), 3]$) : $Grid[(k, k), Categorical[8]]$
21:     ▷ Extract a $k \times k$ grid of features from the terrain map.
22:     $\mathbf{h} \leftarrow$ RESNET-FEATURE-EXTRACTOR($\mathbf{obs}$)

23:     ▷ Generate latent grid graph from grid
24:     $g^{(in)} \leftarrow$ BUILD-GRAPH($\mathbf{h}_0$)

25:     ▷ Execute algorithmic module on grid graph
26:     $g^{(out)} \leftarrow$ ALGORITHMIC-MODULE($g^{(in)}$)

27:     ▷ For each node, identify its optimal predecessor(s) by attending to its in-edges.
28:     return PREDICT-POINTERS($g^{(out)}$)
```

underlying *latent grid graph*. This graph has a node (i, j) for every cell $\mathbf{f}_{ij}$ in the grid, and each node (i, j) has an out-edge to each neighbouring cell (i', j') (including those diagonally adjacent to (i, j)). Now, to decide how to populate nodes and edges with features, based on the type signature of the Bellman-Ford algorithm we want our nodes to carry information about whether or not their corresponding cell is the source node, and our edges to carry information about their cost of traversal. As such, we populate nodes (i, j) with node features $\mathbf{f}_{ij}$, and following Vlastelica, Paulus, Musil, *et al.* [1], who synthesise their grid graph such that the weight of an edge $(i, j) \to (i', j')$ is the cost of the terrain type of cell (i', j'), we populate edges $(i, j) \to (i', j')$ with edge features $\mathbf{f}_{i'j'}$.

Executing the algorithmic module. Now, once we have a latent grid graph $g : G[\mathcal{I}, \mathcal{I}]$, we can apply a wrapped algorithmic module in order to execute Bellman-Ford over it. For this benchmark, we provide two algorithmic modules for comparison:

- **NAP.** A frozen, graph-based encode-process-decode NAP (pre-trained as per Veličković, Badia, Budden, *et al.* [9], and with adaptations as per Ong [24]), wrapped with linear encoders and decoders (Algorithm 3).
- **sDAB.** A scalar implementation of Bellman-Ford, wrapped with skip-connected linear encoders and decoders (Algorithm 4). Note that, when extracting initial distances from node features, we apply a scaled sigmoid to introduce an inductive bias towards either predicting $d_i^{(0)} = 0$ or $d_i^{(0)} = \infty$ (where $\infty \approx 100$ to avoid issues with numerical instability).

Predicting predecessor pointers. Finally, once we have our output graph $g : G[\mathcal{O}_n, \mathcal{O}_e]$, for each node in our graph, we attend to its in-edges (using node pointer decoding as in [9]) to obtain a distribution over its 8 possible predecessors. We output these per-node predecessor distributions as a grid of per-cell categorical variables indicating the cardinal direction (e.g. North, North-East, East etc) of the predecessor cell.

Algorithm 3 A pseudocode implementation of a Bellman-Ford NAP (i.e. a wrapper around an EPD-NAP pre-trained as in Ong [24]), for use as an ALGORITHMIC-MODULE in WARCRAFT-NET (Algorithm 2).

```
Require: The following learnable functions:
    $enc_n : \mathcal{I} \to \mathcal{V}_n^{(in)} := Linear$        $dec_h : \mathcal{V}_n^{(out)} \to \mathcal{O}_n := Linear$
    $enc_e : \mathcal{I} \to \mathcal{V}_e^{(in)} := Linear$        $dec_e : \mathcal{V}_e^{(out)} \to \mathcal{O}_e := Linear$

1: def NAP($G(\{\mathbf{h}_i\}, \{\mathbf{e}_{ij}\}) : G[\mathcal{I}, \mathcal{I}]) : G[\mathcal{O}_n, \mathcal{O}_e]$
2:     ▷ Map into algorithmic latent space
3:     $\mathbf{h}_i^{(in)}, \mathbf{e}_{ij}^{(in)} \leftarrow enc_h(\mathbf{h}_i), enc_e(\mathbf{e}_{ij})$

4:     ▷ Execute EPD-trained NAP
5:     $G(\{\mathbf{h}_i^{(out)}\}, \{\mathbf{e}_{ij}^{(out)}\}) \leftarrow$ EPD-NAP($G(\{\mathbf{h}_i^{(in)}, \mathbf{e}_{ij}^{(in)}\})$)

6:     ▷ Map out of algorithmic latent space
7:     $\mathbf{h}_i^{(ret)}, \mathbf{e}_{ij}^{(ret)} \leftarrow dec_h(\mathbf{h}_i^{(in)}, \mathbf{h}_i^{(out)}), dec_e(\mathbf{e}_{ij}^{(in)}, \mathbf{e}_{ij}^{(out)})$

8:     ▷ Return graph
9:     return $G(\{\mathbf{h}_i^{(ret)}\}, \{\mathbf{e}_{ij}^{(ret)}\})$
```

B. Details of the *optimal* and *tiebreaking* dataset variants

For comparability, we base our dataset on that of Vlastelica, Paulus, Musil, *et al.* [1], consisting of 10,000 training, 1,000 validation and 1,000 test images of randomly-generated terrain maps from the *Warcraft II* tileset [7].

Now, we observe that these terrain maps do not always have unique shortest paths (let alone shortest path trees). Indeed, while Vlastelica, Paulus, Musil, *et al.* [1] simply ignore this issue,[1] we handle it by adapting our training and evaluation metrics accordingly.

[1]Note that, during training, Vlastelica, Paulus, Musil, *et al.* [1] select a particular shortest path to supervise on for each map, and during evaluation, they score models based on whether or not their predicted shortest path is optimal. This can potentially lead to performance

Algorithm 4 A pseudocode implementation of a Bellman-Ford sDAB, for use as an ALGORITHMIC-MODULE in WARCRAFT-NET (Algorithm 2).

```
Require: The following learnable functions:
    $enc_w : \mathcal{I} \to \mathbb{R} := Linear$        $dec_h : \mathbb{R} \to \mathcal{O}_n := Linear$
    $enc_d : \mathcal{I} \to \mathbb{R} := Linear$        $dec_e : \mathcal{I} \to \mathcal{O}_e := Linear$

1:  def sDAB($G(\{\mathbf{h}_i\}, \{\mathbf{e}_{ij}\}) : G[\mathcal{I}, \mathcal{I}]) : G[\mathcal{O}_n, \mathcal{O}_e]$
2:      ▷ Extract weights from edge features
3:      $w_{ij} : \mathbb{R} \leftarrow enc_w(\mathbf{e}_{ij})$

4:      ▷ Extract initial distances from node features
5:      $d_i^{(0)} : \mathbb{R} \leftarrow 100 \cdot \sigma(enc_d(\mathbf{h}_i))$

6:      ▷ Perform Bellman-Ford relaxations on grid graph
7:      for $t \leftarrow 1, ..., (k \times k)$ do
8:          $d_j^{(t)} = \min(d_j^{(t-1)}, \min_{i \to j}(d_i^{(t-1)} + w_{ij}))$

9:      ▷ Map final distances to latent space and return latent graph
10:     return $G(\{dec_h(d_i^{(k \times k)})\}, \{dec_e(\mathbf{e}_{ij})\})$
```

Indeed, we explore two different ways of dealing with non-unique predecessors:

Optimal variant. For each grid cell, we train via cross-entropy loss to predict a distribution over its predecessors, with uniform weight over all optimal predecessors, and zero weight everywhere else. We evaluate our models based on both **optimal accuracy** (i.e. the percentage of predicted predecessor distributions which have an optimal pointer as their mode), and **optimal tree-accuracy** (i.e. the percentage of grids for which all predicted predecessor distributions in that grid have an optimal pointer as their mode).

Tie-breaking variant. For each grid cell, we train via cross-entropy loss to predict the optimal predecessor, breaking ties by priority (with the highest priority predecessor being the eastern cell, and priority decreasing clockwise). We evaluate our models based on **exact accuracy** (i.e. the percentage of correctly-predicted predecessors), and **exact tree-accuracy** (i.e. the percentage of trees with correctly-predicted predecessors).

As neither our sDAB nor our NAP are designed to break ties in the manner described above[2], while the optimal problem variant is easily solvable with only information about per-node distances, to solve the tiebreaking variant, our NAPs and sDABs must need to learn to distinguish between different nodes with the same shortest path length. As such, comparing performance across these two environments lets us explore the case where the underlying algorithm of our problem differs slightly from our algorithmic prior.

C. Experimental details

Executors. Recall that the problem of WARCRAFT-SHORTEST-PATH-TREE has an algorithmic prior towards Bellman-Ford. As such, in the following experiments, we compare the relative performance of WARCRAFT-NET (Appendix A) when equipped with various executors with an algorithmic inductive bias towards Bellman-Ford – specifically, the Bellman-Ford sDAB (as described in Appendix A), and a Bellman-Ford NAP (as trained in Ong [24]). The NAP we used achieved a pointer accuracy of 0.9941 in-distribution (on graphs of size 16), and a pointer accuracy of 0.9561 out-of-distribution (on graphs of size 64).

Hyperparameters and training. For each model, across each of our two environments, we performed 5 training runs

issues if the shortest path on which we supervise is selected in a deterministic way: in particular, the model may try to learn the algorithm for choosing the exact shortest path in specific cases, at the cost of decreasing its overall performance at choosing an optimal path.

[2]Indeed, our sDAB only outputs per-node shortest path lengths, and while the NAP we use (i.e. the Bellman-Ford NAP from Section **??**) was trained with supervision on both per-node shortest path lengths and predecessor pointers, as all edges had weights uniformly randomly sampled from $[0, 1]$, it never learned to perform tie-breaking.

with different seeds. For each run, we adopted the hyperparameters and training regime of Vlastelica, Paulus, Musil, *et al.* [1], training for 50 epochs with batch size 70 and learning rate 5×10^{-4}, and evaluating on a validation set of size 1,000 after every epoch to choose the highest-performing checkpoint for evaluation. We then evaluated the highest-performing checkpoint of each run on the test sets of [1], assessing both in-distribution (tree size 12×12) and out-of-distribution (tree size 18×18) performance in terms of either tree-accuracy or optimal tree-accuracy as appropriate.[3]

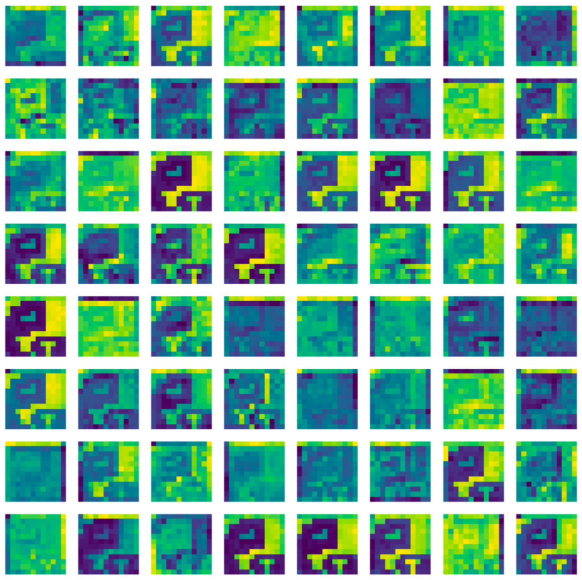
Note that, while in principle we should run our executors for $|V| = 144$ steps, in order for Bellman-Ford to converge, we need only apply our executors for n steps, where n is the maximum number of edges in any shortest path from the root node. As such, due to compute limitations, we only apply our executors for 45 steps, the maximum number of edges in any shortest path from the root node across all our training and test data.

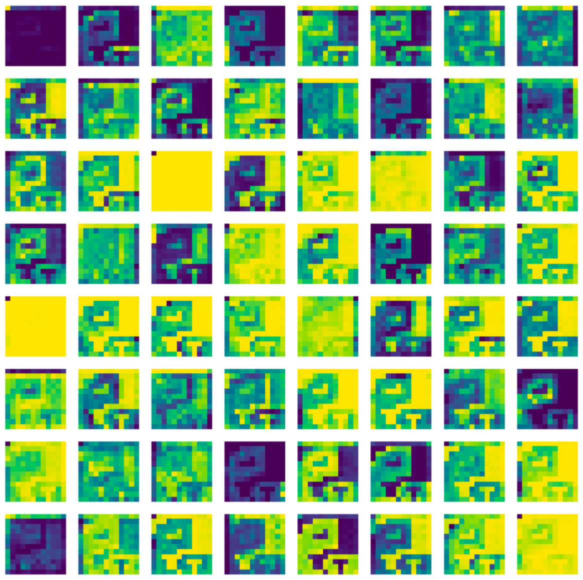
Performance evaluation. As only reporting point estimates has historically led the field to erroneously conclude which methods are state-of-the-art [25], [26], we follow the recommendations of the *RLiable* framework [27] and instead report *interval estimates* of performance. Specifically, for each set of runs, we report model performance through **bootstraped 95% confidence intervals** [28] for mean tree-accuracy (or optimal tree-accuracy). And, in order to compare models X and Y, we estimate 95% confidence intervals for the **probability of improvement** of X over Y (in terms of per-run maximum average reward) via bootstrap resampling [27], by sampling from the empirical distributions of X and Y, and computing the U-statistic from the *Mann-Whitney U test* [29] over these samples. Observe that, as we collect $n = 15$ runs, we have $\binom{15+15-1}{15} = 7.8 \times 10^7$ possible bootstrap resamples, so we have sufficient data for bootstrap resampling to be meaningful.

[3]Note that, due to limitations of the ResNet-18 architecture, as per [1], we are unable to evaluate its out-of-distribution performance.

D. An illustration of weight matrices from a learned pDAB-WARCRAFT-NET

For the Warcraft terrain map analysed in Section 4, we present all 64 sets of weight and initial distance matrices that our feature extractor learns to pass to each of the 64 instances of Bellman-Ford in our pDAB, alongside their corresponding final distance matrices, in Figure 7.


(a) Initial weight matrices (w_{ij})


(b) Initial distance matrices ($d_{ij}^{(0)}$)

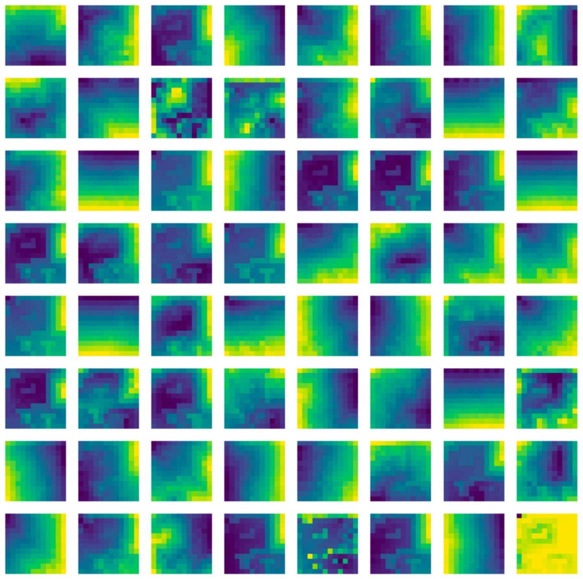

(c) Final distance matrices ($d_{ij}^{(k\times k)}$)

Figure 7: The inputs and outputs to each of the 64 Bellman-Ford instances within our pDAB, when run on the Warcraft terrain map analysed in Section 4, presented as (individually-scaled) heatmaps. Each cell (i, j) corresponds to the learned input / output for the $8i + j$th Bellman-Ford instance in our pDAB.