Persistent Message Passing

Heiko Strathmann, Mohammadamin Barekatain, Charles Blundell and Petar Veličković
DeepMind
{strathmann,barekatain,cblundell,petarv}@google.com

Abstract

Graph neural networks (GNNs) are a powerful inductive bias for modelling algorithmic reasoning procedures and data structures. Their prowess was mainly demonstrated on tasks featuring Markovian dynamics, where querying any associated data structure depends only on its *latest* state. For many tasks of interest, however, it may be highly beneficial to support efficient data structure queries dependent on previous states. This requires tracking the data structure's evolution through time, placing significant pressure on the GNN's latent representations. We introduce Persistent Message Passing (PMP), a mechanism which endows GNNs with capability of querying past state by explicitly *persisting* it: rather than overwriting node representations, it creates new nodes whenever required. PMP generalises out-of-distribution to more than $2\times$ larger test inputs on dynamic temporal range queries, significantly outperforming GNNs which overwrite states.

1 Introduction

Graph neural networks (GNNs) are one of the the most impactful approaches to processing data over irregular domains (e.g., quantum chemistry (Klicpera et al., 2020), drug discovery (Stokes et al., 2020), social network analysis (Pal et al., 2020) and physics simulations (Pfaff et al., 2020)). *Combinatorial optimisation* tasks (Nair et al., 2020) have been used to test the limits of GNNs particularly when *extrapolation* is required. Extrapolation, in general, is now known to occur in GNNs only under stringent conditions on their architecture and input featurisation (Xu et al., 2020).

Algorithmic reasoning (Cappart et al., 2021, Section 3.3.) has been the source of much of the knowledge of how to build extrapolating GNNs. This emerging area of research seeks to emulate each iteration of classical algorithms (Cormen et al., 2009) directly within neural networks. Through the lens of algorithmic alignment (Xu et al., 2019), GNNs can be constructed that closely mimic *iterative computation* (Veličković et al., 2019; Tang et al., 2020), *linearithmic sequence processing* (Freivalds et al., 2019), and *pointer-based data structures* (Veličković et al., 2020). Also such approaches are capable of *strongly* generalising (Yan et al., 2020) and *data-efficient* planning (Deac et al., 2020).

We study GNN-based reasoners acting on inputs that *dynamically change* over time. This setup was previously studied briefly (Veličković et al., 2020) and only featured *Markovian* querying: only using the *latest* version of the data. In the non-Markovian case, queries require knowledge of *previous* versions of the data. For example, during search, a particular path of a search tree may be expanded but later backtracked. Efficient queries over histories of states serve as a basis for agents solving POMDPs. In economics, historical data are revised and these revisions themselves are of interest.

Most GNNs are designed with Markovian querying in mind: latent representations are *overwritten* in every step, and the last representation is used to answer queries. This overloads GNNs' latents, as *all* past snapshots of the data must be represented within them. Further, Hinton (2021) recently makes a strong case for *replicated* embeddings. We propose **Persistent Message Passing** (PMP), replacing overwriting with *persisting*: when updating a node's state, a *copy* of that node is preserved for later use. To avoid excessive memory usage, PMP has a mechanism to select which nodes to persist.

PMP effectively provides GNNs with an *episodic memory* (Pritzel et al., 2017) of their previous computations, just as in social networks where dynamically changing graph data may necessitate explicit memory modules (Rossi et al., 2020). PMP aligns with the broad class of *persistent data structures* (Driscoll et al., 1989), which further *expands* the space of general-purpose algorithms that can be neurally executed. We show on *dynamic range querying* that our method provides significant benefits to overwriting-based GNNs, both in- and out-of-distribution.

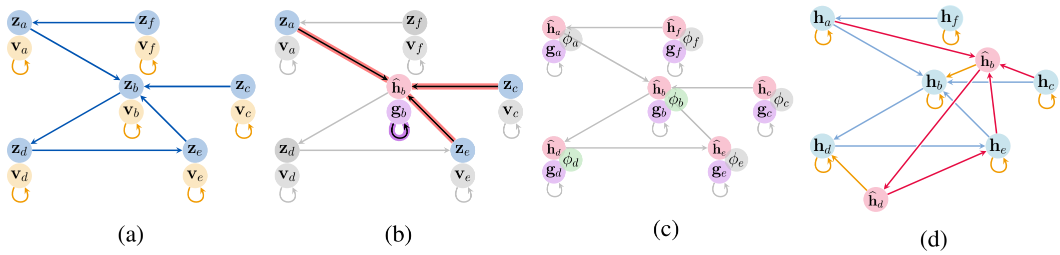

Figure 1: **Overview of PMP**. **(a)** Derived representations for operation features $\boldsymbol{z}$ (blue) and relevance features $\boldsymbol{v}$ (yellow). Initially, relevance nodes point to themselves and the operation connectivity is provided. **(b)** Node b receives messages from its neighbours, leading to next-step candidate latents $\widehat{\boldsymbol{h}}_b$, and relevance latents $\boldsymbol{g}_b$. **(c)** Persistency mask prediction selects nodes b and d. **(d)** Selected nodes are copied (red) while original nodes (blue) remain untouched. Connectivity of new nodes mimics that of their predecessors, and relevance connectivity (yellow) points to persisted "old" nodes.

2 Problem setup and PMP architecture

Consider a sequential supervised learning setting: We are given a set of n entities, along with connectivity information between them. At each time step, we perform *operations*, which both may change the entities' states and emit an output that we aim to predict. Each operation may further be associated with a specific *snapshot* of the entities, allowing operations over past states.

More formally, we are given an adjacency matrix, $\boldsymbol{\Pi} \in \{0,1\}^{n\times n}$, and a sequence of input pairs $(\mathcal{E}^{(1)}, s^{(1)}), (\mathcal{E}^{(2)}, s^{(2)}), \ldots$ where $\mathcal{E}^{(t)} = (\boldsymbol{e}_1^{(t)}, \ldots, \boldsymbol{e}_n^{(t)})$ are feature vectors (operation), and $s^{(t)} \in \{1, \ldots, t-1\}$ are integers (snapshot indices). The task is to predict operation outputs $\boldsymbol{y}^{(t)}$ given $(\mathcal{E}^{(t)}, s^{(t)})$ where we have *persistency*: $(\mathcal{E}^{(t)}, s^{(t)}) = (\mathcal{E}^{(t')}, s^{(t')}) \Rightarrow (\boldsymbol{y}^{(t)} = \boldsymbol{y}^{(t')})$ for all $t' < t$.

Predicting targets $\boldsymbol{y}^{(t)}$ requires the model to maintain a *persistent* set of internal states to represent the history of operations for all entities throughout their lifetime. A naïve solution is copying *all* entities after operations, but this quickly becomes intractable due to $\mathcal{O}(tn)$ storage costs—a selective approach is more desirable. In the following, we present our proposed model, termed *Persistent Message Passing (PMP)*, that combines these desiderata in an efficient way. See fig. 1 for an overview.

Overview At each step, t, our PMP model maintains a set of $N^{(t)}$ *hidden states* $\mathcal{H}^{(t)} = \{\boldsymbol{h}_j^{(t)} \in \mathbb{R}^d\}_{j=1}^{N^{(t)}}$. Initially, $N^{(0)} = n$ in one-to-one correspondence with the n entities, and $\boldsymbol{h}_j^{(0)} = \boldsymbol{0}$. During operations, instead of overwriting hidden states, PMP first *selects* a subset of states to persist, and then adds an updated version of these to $\mathcal{H}^{(t)}$—preserving information about previous states in $\mathcal{H}^{(t+1)}$.

To predict outputs, PMP needs to track (i) updated connectivity and (ii) which hidden states are relevant at time-step $s^{(t)}$. To achieve this, PMP maintains two $N^{(t)} \times N^{(t)}$ adjacency matrices, $\boldsymbol{\Pi}^{(t)}$ for *connectivity* and $\boldsymbol{\Lambda}^{(t)}$ for *relevance*. We initialise $\boldsymbol{\Pi}^{(0)} = \boldsymbol{\Pi}$ from the task input, and replicate its structure when adding copies of persisted nodes. By that, we focus on persistency in isolation—even though our framework can easily be extended to inferred $\boldsymbol{\Pi}^{(t)}$, e.g. as in Veličković et al. (2020). $\boldsymbol{\Lambda}^{(t)}$ is updated so that new nodes link to their old versions. Note that this form needs $\mathcal{O}(t)$ steps to reach past nodes and we leave improvements to $\mathcal{O}(\log t)$ (e.g. Pugh (1989)) for future work.

PMP closely follows the *encode-process-decode* paradigm (Hamrick et al., 2018), with additional *masking* to select nodes to persist (as done in Yan et al. (2020); Veličković et al. (2019)).

Input encoding We encode the current hidden states with both time-stamps and the current operation, using *encoder* networks $f_{\text{relevance}}$ and $f_{\text{operation}}$:

$$\boldsymbol{v}_j^{(t)} = f_{\text{relevance}}\left(\texttt{time\_stamp}(j), t, \boldsymbol{h}_j^{(t-1)}\right) \qquad \boldsymbol{z}_j^{(t)} = f_{\text{operation}}\left(\texttt{expand}(\mathcal{E}^{(t)}, s^{(t)}), \boldsymbol{h}_j^{(t-1)}\right) \tag{1}$$

where, $\texttt{time\_stamp}(i)$ is the time-step when $\boldsymbol{h}_i^{(t-1)}$ was added to the set of hidden states $\mathcal{H}^{(t)}$, and $\texttt{expand}(\mathcal{E}^{(t)}, s^{(t)})$ maps the n operation features to the $N^{(t-1)}$ hidden states, c.f. section 3.

Message passing The derived representations, $\mathbf{Z}^{(t)} = (\boldsymbol{z}_1^{(t)}, \ldots, \boldsymbol{z}_{N^{(t)}}^{(t)})$ and $\mathbf{V}^{(t)} = (\boldsymbol{v}_1^{(t)}, \ldots, \boldsymbol{v}_{N^{(t)}}^{(t)})$ are fed into a *processor network*, P, using $\mathbf{\Pi}^{(t-1)}, \mathbf{\Lambda}^{(t-1)}$ as relational information:

$$\widehat{\boldsymbol{H}}^{(t)} = P\left(\mathbf{Z}^{(t)}, \mathbf{\Pi}^{(t-1)}\right) \qquad \boldsymbol{G}^{(t)} = P\left(\mathbf{V}^{(t)}, \mathbf{\Lambda}^{(t-1)}\right) \tag{2}$$

yielding candidate next-step latent features $\widehat{\boldsymbol{H}}^{(t)} = (\widehat{\boldsymbol{h}}_1^{(t)}, \ldots, \widehat{\boldsymbol{h}}_{N^{(t)}}^{(t)})$ and relevance latent features $\boldsymbol{G}^{(t)} = (\boldsymbol{g}_1^{(t)}, \boldsymbol{g}_2^{(t)}, \ldots, \boldsymbol{g}_{N^{(t)}}^{(t)})$. $\widehat{\boldsymbol{H}}^{(t)}$ are candidates, as $\widehat{\boldsymbol{h}}_j^{(t)}$ will be retained only if state j is persisted.

Relevance mechanism In order to decide which candidate next-step latent features are used for computing the operation response, we compute a per-state *relevance* mask using the relevance latents and a *masking* network $\psi_{\text{relevance}}$ as $\mu_j^{(t)} = \psi_{\text{relevance}}(\boldsymbol{g}_j^{(t)}) \in \{0, 1\}$. This mask is used to select relevant hidden states from $\widehat{\boldsymbol{H}}^{(t)}$ in downstream computation of response and persistency.

Persistency mechanism Many efficient data structures only modify a small (e.g. $\mathcal{O}(\log n)$) subset of the entities at once (Cormen et al., 2009). We explicitly incorporate this inductive bias into PMP: In order to decide which of the candidate next-step latent features $\widehat{\boldsymbol{h}}_i^{(t)}$ are appended to the set of hidden states $\mathcal{H}^{(t)}$, we predict a per-state *persistency* mask (on relevant states only), $\phi_j^{(t)} = \psi_{\text{persistency}}(\widehat{\boldsymbol{h}}_j^{(t)} \cdot \mu_j^{(t)}) \in \{0, 1\}$. Using this mask, we generate a set of *next-step latent features* by appending masked candidates to the previous set of hidden features as $\mathcal{H}^{(t)} = \texttt{concat}(\mathcal{H}^{(t-1)}, \{\widehat{\boldsymbol{h}}_j^{(t)}\}_{j:\phi_j^{(t)}=1})$.

Updating adjacency We subsequently generate next-step adjacency matrices. For $\mathbf{\Pi}^{(t)}$, we append copies of masked rows $\{j : \phi_i^{(t)} = 1\}$ to the end, and additionally update connections to others replaced nodes within those rows. For $\mathbf{\Lambda}^{(t)}$, we simply let each updated node point to its persisted predecessor. These updated hidden states and adjacency matrices are used in the next time-step.

Readout Before moving to the next time-step, we compute a *response* to the input operation from the $\widehat{\boldsymbol{H}}^{(t)}$ using a *decoder* network g on *relevant* nodes ($\mu_j^{(t)} = 1$), $\boldsymbol{y}^{(t)} = g\left(\bigoplus_{j:\mu_i^{(t)}=1} \boldsymbol{z}_j^{(t)}, \bigoplus_{j:\mu_i^{(t)}=1} \widehat{\boldsymbol{h}}_j^{(t)}\right)$, where $\bigoplus$ is a *readout* aggregator (we use maximisation).

PMP components In our implementation, encoder, decoder, masking and query networks are all linear transformations. Echoing the results of prior work on algorithmic modelling with GNNs (Veličković et al., 2019), we recovered strongest performance when using *message passing neural networks* (Gilmer et al., 2017) for P in eq. (2): nodes aggregate messages from neighbouring nodes and combine those via an aggregation mechanism (we use maximisation), see appendix A for details.

Optimisation Besides the operation response loss for $\boldsymbol{y}^{(t)}$, PMP optimises the cross-entropy of relevance and persistency masks $\mu_j^{(t)}, \phi_j^{(t)}$ against the ground-truth. It is also possible to provide additional supervision for e.g. predicting per-node responses from the $\widehat{\boldsymbol{H}}^{(t)}$, as done in the experiments.

3 Task: Range minimum query with persistent segment trees

We focus on the highly versatile *range minimum query* (RMQ) task on an array A of K integers. At each step t, we perform one of two operations: (a) *setting* an individual array element to a new value, $A_k^{(t+1)} \leftarrow x$, or (b) *querying* for the *minimum* value over a particular contiguous range, at a previous point in time, $\min_{a \leq k \leq b} A_k^{(t')}$, for $t' \leq t$.

RMQ-style tasks are effectively solved with *segment trees* (Bentley, 1977). Segment trees are almost-complete, full binary trees consisting of $n = 2K - 1$ nodes, such that leaf nodes correspond to array elements, and any intermediate node maintains the minimum of its two children. Updating a leaf nodes' value only requires updating the $\mathcal{O}(\log n)$ nodes along the path to the root. Querying requires minimising over the *canonical* covering of the query range, which contains $\mathcal{O}(\log n)$ nodes. Segment

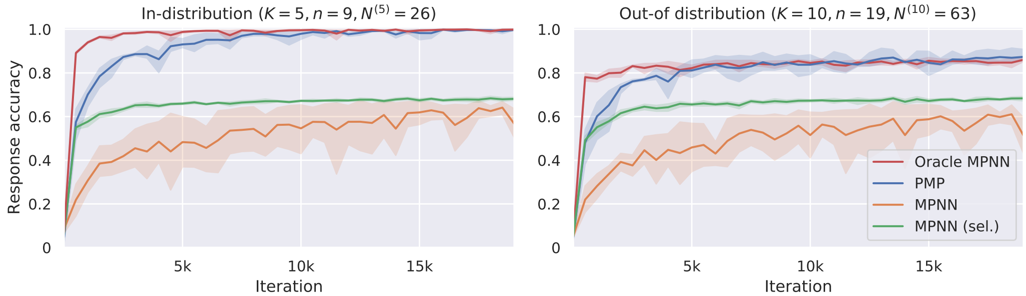

Figure 2: Query response accuracies (all bits correct) for the persistent RMQ task on in-distribution test data (**left**) and on $2\times$ larger test inputs (**right**). PMP gradually catches up to the Oracle MPNN, significantly outperforming the two overwriting baseline MPNNs. Errors bars are over 5 seeds.

trees can be made *persistent* (PST) to for querying of past states: upon update, we copy instead of modifying nodes, resulting in additional $\mathcal{O}(\log n)$ storage per update.

We choose to use PMPs to imitate computations of persistent segment trees in our experiments, not only because of their high importance across computational geometry and combinatorial heuristics, but also because their "backbone" tree connectivity remains identical across steps. This allows us to focus on the utility of PMP's persistency mechanism (with hard-coded connectivity updates).

To model segment trees with PMP, we let our n entities correspond to the PST nodes. Specifically, PMP is provided with operation representations for (i) updating a leaf node (indicator feature) to a new integer value (global feature), or (ii) querying the minimum value in a given range (indicator feature), at a past state. Our PMP model is supervised in a *teacher-forced* setup: it is guided to either: (a) select appropriate nodes to persist, and make their latents predictive of updated minimum values, in case of updates; or (b) select the relevant canonical cover of the relevant snapshot of the PST, and produce the appropriate answers over them, in the case of queries. We represent all integers in binary form (c.f. Yan et al. (2020)) and consider a query answer correct if *all* bits are correctly predicted.

Dataset & training We generate a dataset consisting of 10,000 PST rollouts over arrays of size $K = 5$, resulting in $N^{(0)} = n = 9$ initial nodes. For each rollout, we perform 5 updates with random position and new value, resulting in $N^{(5)} \approx 26$ nodes on average, followed by 5 random queries. We compute features representations of queries (indicators) and updates (indicators & binary encodings) dynamically during rollouts, including their mapping onto the model's growing collection of hidden states via `expand` in eq. (1). For in-distribution evaluation, we generate another 200 rollouts from the same distribution. For out-of-distribution evaluation, we in addition use arrays of size $K = 10$, and 10 updates, resulting in $n = 19$ and $N^{(10)} \approx 63$ nodes. Further details can be found in appendix A.

Results In fig. 2, we provide the test query accuracy (fraction of all bits correctly predicted) of our PMP model for answering RMQ, in-distribution and over a dataset of more than *twice* as many nodes and operations, *i.e.,* out-of-distribution. At test time, teacher forcing is disabled, and a model must be properly rolled out over the entire *lifetime* of the data structure, including both persistency and relevance mechanisms. We find that PMP is close to perfect in predicting masks for both mechanisms (results not shown)—this is crucial as persisting wrong nodes, or selecting wrong relevant nodes would severely corrupt the model's memory. We compare PMP against message passing networks (MPNNs) (Gilmer et al., 2017) which *overwrite* rather than persist (either entirely, or selectively by using a node-level mask), and an *oracle* MPNN, which predicts from the *given correct* snapshot of the segment tree and hence doesn't need to memorise any evolution of the nodes.

Our test results are highly indicative of the strong performance of PMP: not only are they consistently more accurate at answering temporal range queries than their overwriting counterparts, but, given enough training time, they are capable of *matching* the oracle GNN even over long rollouts. This demonstrates the robustness of the proposed neural persistency mechanism, and for temporal querying, they appear to effectively relieve the pressure on a GNN's internal representations.

Interesting directions for future work are improving scalability of the number of updates, dealing with dynamic connectivity structure, and unsupervised learning of relevance and persistency masks.

References

Jon Louis Bentley. Solutions to klee's rectangle problems. *Unpublished manuscript*, pp. 282–300, 1977.

Quentin Cappart, Didier Chételat, Elias Khalil, Andrea Lodi, Christopher Morris, and Petar Veličković. Combinatorial optimization and reasoning with graph neural networks. *arXiv preprint arXiv:2102.09544*, 2021.

Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

Andreea Deac, Petar Veličković, Ognjen Milinković, Pierre-Luc Bacon, Jian Tang, and Mladen Nikolić. Xlvin: executed latent value iteration nets. *arXiv preprint arXiv:2010.13146*, 2020.

James R Driscoll, Neil Sarnak, Daniel D Sleator, and Robert E Tarjan. Making data structures persistent. *Journal of computer and system sciences*, 38(1):86–124, 1989.

Kārlis Freivalds, Emīls Ozoliņš, and Agris Šostaks. Neural shuffle-exchange networks–sequence processing in o (n log n) time. *arXiv preprint arXiv:1907.07897*, 2019.

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.

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.

Geoffrey Hinton. How to represent part-whole hierarchies in a neural network. *arXiv preprint arXiv:2102.12627*, 2021.

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

Johannes Klicpera, Janek Groß, and Stephan Günnemann. Directional message passing for molecular graphs. *arXiv preprint arXiv:2003.03123*, 2020.

Vinod Nair, Sergey Bartunov, Felix Gimeno, Ingrid von Glehn, Pawel Lichocki, Ivan Lobov, Brendan O'Donoghue, Nicolas Sonnerat, Christian Tjandraatmadja, Pengming Wang, et al. Solving mixed integer programs using neural networks. *arXiv preprint arXiv:2012.13349*, 2020.

Aditya Pal, Chantat Eksombatchai, Yitong Zhou, Bo Zhao, Charles Rosenberg, and Jure Leskovec. Pinnersage: Multi-modal user embedding framework for recommendations at pinterest. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 2311–2320, 2020.

Tobias Pfaff, Meire Fortunato, Alvaro Sanchez-Gonzalez, and Peter W Battaglia. Learning mesh-based simulation with graph networks. *arXiv preprint arXiv:2010.03409*, 2020.

Alexander Pritzel, Benigno Uria, Sriram Srinivasan, Adria Puigdomenech Badia, Oriol Vinyals, Demis Hassabis, Daan Wierstra, and Charles Blundell. Neural episodic control. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 2827–2836. JMLR. org, 2017.

William Pugh. Skip lists: A probabilistic alternative to balanced trees. In *Workshop on Algorithms and Data Structures*, pp. 437–449. Springer, 1989.

Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael Bronstein. Temporal graph networks for deep learning on dynamic graphs. *arXiv preprint arXiv:2006.10637*, 2020.

Jonathan M Stokes, Kevin Yang, Kyle Swanson, Wengong Jin, Andres Cubillos-Ruiz, Nina M Donghia, Craig R MacNair, Shawn French, Lindsey A Carfrae, Zohar Bloom-Ackermann, et al. A deep learning approach to antibiotic discovery. *Cell*, 180(4):688–702, 2020.

Hao Tang, Zhiao Huang, Jiayuan Gu, Bao-Liang Lu, and Hao Su. Towards scale-invariant graph-related problem solving by iterative homogeneous graph neural networks. *arXiv preprint arXiv:2010.13547*, 2020.

Petar Veličković, Rex Ying, Matilde Padovano, Raia Hadsell, and Charles Blundell. Neural execution of graph algorithms. *arXiv preprint arXiv:1910.10593*, 2019.

Petar Veličković, Lars Buesing, Matthew C Overlan, Razvan Pascanu, Oriol Vinyals, and Charles Blundell. Pointer graph networks. *arXiv preprint arXiv:2006.06380*, 2020.

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.

Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. How neural networks extrapolate: From feedforward to graph neural networks. *arXiv preprint arXiv:2009.11848*, 2020.

Yujun Yan, Kevin Swersky, Danai Koutra, Parthasarathy Ranganathan, and Milad Heshemi. Neural execution engines: Learning to execute subroutines. *arXiv preprint arXiv:2006.08084*, 2020.

Appendices

A Training details

We generate a dataset consisting of 10,000 PST rollouts over arrays of size $K = 5$. The array values are randomly initialized between $[1, 15]$ (represented using a 4 bit binary encoding), by first sampling lower bound from the initial interval, and then sampling array elements uniformly between that and the chosen upper bound. We choose this distribution in order to ensure non-trivial query responses for minimum queries.

We perform 5 PST updates where we uniformly sample update index k and new value x from the same distribution as the existing values. This leads to $12, 16, 19, 23, 26$ nodes on average, respectively after each update. After the updates, we perform 5 PST queries, with uniformly sampled query bounds $[a, b]$ and a random past time-step $t' < t$. We use a shared representation for both queries and updates, setting query features to $\mathbf{0}$ when computing updates and vice versa. In the update features, we add an indicator for the node corresponding to the updated PST leaf node, indicators for all leaf nodes, and a global binary encoding of new array value x. For query features, we add an indicator for the leaf nodes which correspond to the query interval bounds $[a, b]$, and indicators whether nodes are left or right children or root nodes. Using a 4-bit binary encoding of array values, this leads to a 10-dimensional representation of operations.

It is straight-forward to dynamically compute these features (of n entities) for the growing set of $N^{(t)}$ hidden states via `expand` in eq. (1): tagging leaf nodes, node type (left/right/root), etc can be inferred from the maintained connectivity of added hidden states within the PMP model.

For in-distribution evaluation, we generate another 200 rollouts from the same distribution. For out-of-distribution evaluation, we in addition use arrays of size 10, and 10 updates resulting in $n = 19$ and on average $23, 28, 32, 37, 41, 45, 50, 54, 58, 63$ nodes respectively after each update. This is followed by 5 queries.

We train PMP for $2 \cdot 10^4$ iterations using a batch size of 16 (rollouts) using Adam (Kingma & Ba, 2014) with a learning rate of 10^{-3}, re-running the model with 5 different seeds. Hidden states have $d = 64$ dimensions. We use 10 message passing steps in the processor network, which we found to lead to slightly better performance of PMP compared to fewer steps. We hypothesise this is due to the linked list approach in implemented in the relevance connectivity matrix $\mathbf{\Lambda^{(t)}}$—using a data structure that allows search in logarithmic time (such as skip lists, (Pugh, 1989)) are likely to allow lower the number of message passing steps.

Message passing network PMP builds on message passing neural networks (Gilmer et al., 2017), used as processor network P in eq. (2). The explicit computation of candidate next-step latent features $\widehat{\boldsymbol{h}}_j^{(t)}$ and relevance latent features $\boldsymbol{g}_j^{(t)}$ is:

$$\widehat{\boldsymbol{h}}_j^{(t)} = U\left(\boldsymbol{z}_j^{(t)}, \max_{\mathbf{\Pi}_{jj'}^{(t-1)}=1} M\left(\boldsymbol{z}_{j'}^{(t)}, \boldsymbol{z}_j^{(t)}\right)\right) \qquad \boldsymbol{g}_j^{(t)} = U\left(\boldsymbol{v}_j^{(t)}, \max_{\mathbf{\Lambda}_{jj'}^{(t-1)}=1} M\left(\boldsymbol{v}_{j'}^{(t)}, \boldsymbol{v}_j^{(t)}\right)\right)$$

where M and U are linear layers producing *vector messages*, followed by ReLU.