



Evolutionary training and abstraction yields algorithmic generalization of neural computers

Daniel Tanneberg¹✉, Elmar Rueckert^{1,2} and Jan Peters^{1,3}

A key feature of intelligent behaviour is the ability to learn abstract strategies that scale and transfer to unfamiliar problems. An abstract strategy solves every sample from a problem class, no matter its representation or complexity—similar to algorithms in computer science. Neural networks are powerful models for processing sensory data, discovering hidden patterns and learning complex functions, but they struggle to learn such iterative, sequential or hierarchical algorithmic strategies. Extending neural networks with external memories has increased their capacities to learn such strategies, but they are still prone to data variations, struggle to learn scalable and transferable solutions, and require massive training data. We present the neural Harvard computer, a memory-augmented network-based architecture that employs abstraction by decoupling algorithmic operations from data manipulations, realized by splitting the information flow and separated modules. This abstraction mechanism and evolutionary training enable the learning of robust and scalable algorithmic solutions. On a diverse set of 11 algorithms with varying complexities, we show that the neural Harvard computer reliably learns algorithmic solutions with strong generalization and abstraction, achieves perfect generalization and scaling to arbitrary task configurations and complexities far beyond seen during training, and independence of the data representation and the task domain.

A crucial ability for intelligent behaviour is to transfer strategies from one problem to another, studied, for example, in the fields of lifelong and transfer learning^{1–4}. Learning and especially deep learning systems have been shown to learn a variety of complex specialized tasks^{5–10}, but extracting the underlying structure of the solution for effective transfer is an open research question³.

The key for effective transfer—and a main pillar of (human) intelligence—is the concept of structure and abstraction^{11–13}. To study the learning of such abstract strategies, the concept of algorithms (such as in computer science¹⁴) is an ideal example for such transferable, abstract and structured solution strategies.

An algorithm is a sequence of instructions, which often represent solutions to smaller subproblems. This sequence of instructions solves a given problem when executed, independent of the specific instantiation of the problem. For example, consider the task of sorting a set of objects. The algorithmic solution, specified as the sequence of instructions, is able to sort any number of arbitrary classes of objects in any order (for example, toys by colour, waste by type or numbers by value) by using the same sequence of instructions, as long as the features and comparison operators that define the order are specified.

Learning such structured, abstract strategies enables the effective transfer to new domains and representations as the abstract solution is independent of both. By contrast, transfer learning usually focuses on improving learning speed on a new task by leveraging knowledge from previously learned tasks, whereas algorithmic solutions do not need to (re)learn at all, only the data-specific operations need to adapt. In other words, the sequence of instructions does not need to be adapted, only the instructions, that is, the solutions to smaller subproblems. Moreover, such structured abstract strategies have built-in generalization capabilities to new task configurations and complexities and can be interpreted better than, for example, common blackbox models such as deep end-to-end networks.

The problem of learning algorithmic solutions

To study the learning of such abstract and structured strategies, we investigate the problem of learning algorithmic solutions, which we characterize by three requirements:

- R1—generalization and scaling to different and unseen task configurations and complexities
- R2—independence of the data representation
- R3—independence of the task domain

Picking up the sorting algorithm example again, R1 represents the generalization and scaling properties, which allow sorting of lists of arbitrary length and initial order, whereas R2 and R3 represent the abstract nature of the solution. This abstraction enables the algorithm, for example, to sort a list of binary numbers while being trained only on hexadecimal numbers (R2). Furthermore, the algorithm trained on numbers is able to sort lists of strings (R3). If R1–R3 are fulfilled, the algorithmic solution does not need to be retrained or adapted to solve unforeseen task instantiations—only the data-specific operations need to be adjusted.

Earlier research on solving algorithmic problems has been undertaken, for example, in grammar learning^{15–17}, and is becoming a more and more active field in recent years outside of it^{18–32}, with a typical focus on identifying algorithmic-generated patterns or solving algorithmic problems in an end-to-end set-up^{18–27}, and less on finding algorithmic solutions^{28–32} that consider the three discussed requirements R1–R3 for generalization, scaling and abstraction.

Although R1 is typically tackled in some (relaxed) form, as it represents the overall goal of generalization in machine learning, the abstraction abilities R2 and R3 are missing. Furthermore, most algorithms require a form of feedback, using computed intermediate results from one computational step in subsequent steps and a variable number of computational steps to solve a problem instance. It is therefore necessary to be able to cope with varying numbers

¹Intelligent Autonomous Systems, Technische Universität Darmstadt, Darmstadt, Germany. ²Institute for Robotics and Cognitive Systems, Universität zu Lübeck, Lübeck, Germany. ³Robot Learning Group, Max Planck Institute for Intelligent Systems, Tübingen, Germany. ✉e-mail: daniel@robot-learning.de

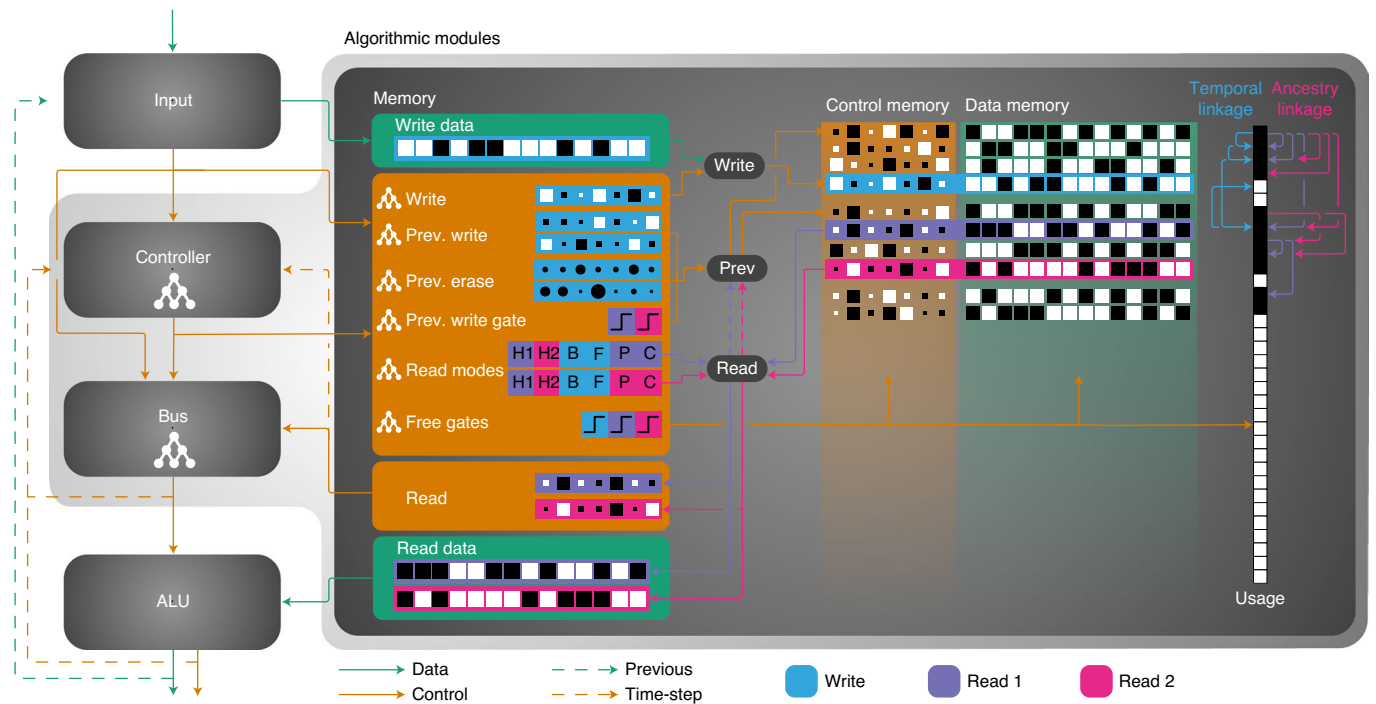


Fig. 1 | The NHC architecture. Information flow is divided into data (green) and control (orange) streams. The modules inside the light grey area—the controller, the memory and the bus—are learning the algorithmic solution on the control stream, whereas the data modules are either learned beforehand or hand-designed. The algorithmic solution operates solely on the control stream to steer the data access and manipulation, whereas the learning signal can be provided on any connection in the architecture (data or control) due to the evolution-based training. Inside the memory module the learnable interfaces that control the data access to the two memory matrices are shown. Sign and magnitude of vectors are shown as the colour and size of the boxes and circles.

of steps and determining when to stop, in contrast to using a fixed number of steps^{21,33}, and to be able to re-use intermediate results, that is, feeding back the models output as its input. These features make the learning problem even more challenging.

The neural Harvard computer

The proposed neural Harvard computer (NHC) is a modular architecture that is based on memory-augmented neural networks^{15–21,23,24,27–29,33–39} and inspired by modern computer architectures (see Fig. 1 for a sketch of the NHC). Memory-augmented networks add external memory to a neural network that allows separation of computation and memorization—in classical neural networks both are encoded into the synaptic weights.

The external memory can be realized differently, for example, as a memory matrix²⁰, tape²³ or stack¹⁹, and the so-called controller network can write and read information through a defined interface that controls the memory access, for example, moving the head one step to the right for a tape memory, or pop the top information in a stack memory. In the NHC, the external memory is realized as a matrix and interaction with the memory is performed via write and read heads, similar to the differentiable neural computer (DNC)²⁰. These heads interact with the memory by writing or reading information into or from the memory matrix, where each row corresponds to a memory location with a specified word size, that is, length of the information vector.

Information split. Learning algorithmic solutions requires the decoupling of algorithmic computations from data-dependent manipulations and domains. An abstraction level is therefore introduced by dividing the information flow into two streams: the data stream d and control stream c . The introduction of external memories to neural networks helps to separate computation and

memorization, the information flow split similarly helps to separate algorithmic computations and data-specific manipulations.

This information split induces two major features of the NHC: (1) the split into data modules that operate on d and algorithmic modules that operate on c ; and (2) the introduction of two coupled memories. The algorithmic modules operate on c , that is, $\text{AlgModule}(c) \rightarrow c$, whereas the data modules Input and arithmetic logic unit (ALU) are operating on d , that is, $\text{DataModule}(d, c) \rightarrow d, c$. They create an abstract interface and separation between algorithmic computation and data-specific processing. Although the Input module receives the external data and provides algorithm-specific control signals, the ALU receives data and control information to manipulate the data to create new data (hence the name arithmetic logic unit) that is fed back to Input to be available in the next computation step. These two modules are data specific and need to be adapted for a new data representation or domain.

The algorithmic modules. These modules consist of the Controller, Memory and Bus. These modules form the core of the NHC (see Fig. 1) and are responsible for encoding and learning the algorithmic solution based on c .

The Controller. The controller receives the control signals from the Input and the information read from the memory in the previous step; depending on the task to learn, it can also receive feedback from the Bus and ALU. It learns an internal representation of the algorithm state that is sent to the Memory and Bus modules.

The Memory. The memory module uses this representation in addition to the control signals from the Input to learn a set of interfaces for interacting with the memory matrices. These learned interfaces control the write and read heads and hence, what information is

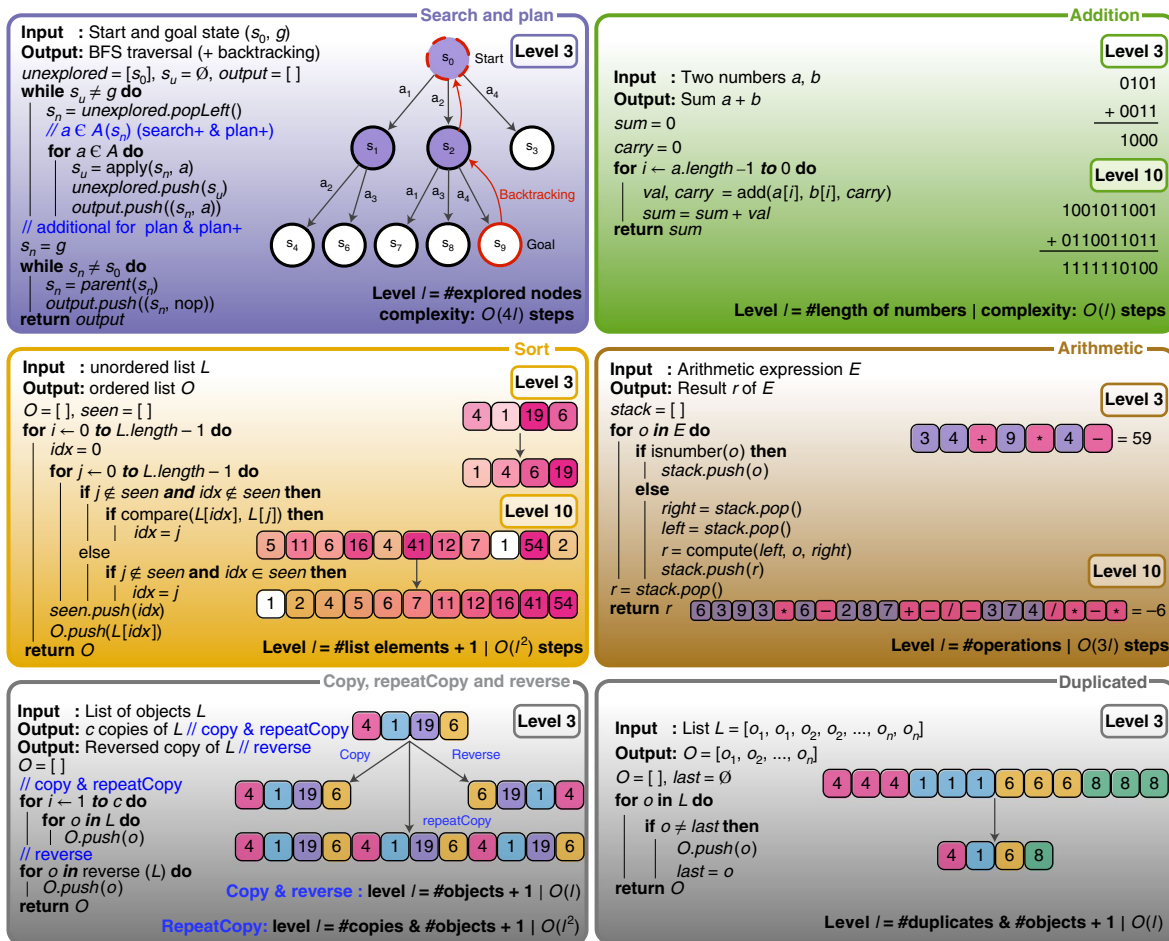


Fig. 2 | Overview of the learned algorithms. All considered algorithms to learn are shown with their pseudocode, how their curriculum level complexity is defined, how the step complexity scales with the level, and examples from indicated levels. Note that the step complexity indicates the runtime complexity and only considers the steps after the input data is shown, neither taking the complexity of the data manipulation into account nor the structure learning required while the input data is presented.

accessed. First, the locations read in the previous step are potentially updated (prev), then new information is written via the write heads (write) and, finally, the read heads read information (read) that is sent back to controller and the Bus. Write and read heads are using hard decisions, that is, each head interacts with one memory location.

The abstraction introduced by the information split also creates the necessity to store data and control information separately. The Memory module therefore uses two memory matrices ($M^c: N \times C$ and $M^d: N \times D$) to store the control and data information, respectively, with N locations, where C and D are control memory and data memory word sizes, respectively. The two memories are coupled such that the same locations are accessed at each step, which allows algorithmic control information to be stored alongside the data information. New information is written via the write heads to unused memory locations and locations can be freed by free gates to be reused. For reading information from the memory, there are several read modes to steer the read heads.

The HALT modes move the read head to the previously read location of the associated head. For example, with two read heads, each head can use two modes (H1 and H2) to move to the location previously read by the corresponding head.

To read data in the order in which it was written, the DNC introduced a temporal linkage mechanism that keeps track of the order of written locations. The NHC uses a simplified version of this temporal linkage. This temporal linkage provides two read modes: one

to move the read head forward to the location that was written next (F) and one to move the head backwards to the location that was written before (B).

Algorithms often require hierarchical data structures or dependencies. To provide such dependencies, the NHC employs an ancestry linkage mechanism. This mechanism keeps track of which location was read before for each written location and therefore provides two read modes: one to move the read head to the parent (P), the location that was read before, and one to move the read head to the child (C), the location that was written after.

The Bus. The Bus combines the representation learned by the Controller with the information read out from the Memory to produce the control signal that is sent to the ALU, indicating which operation to apply on the read data. By using the information read from memory, the Bus can incorporate this new information in the same computational step.

Learning algorithmic solutions

For evaluating the proposed NHC on the three algorithmic requirements R1–R3, a diverse set of algorithms was learned and the solutions were tested on their generalization, scaling and abstraction abilities.

The 11 learned algorithms solve search, plan, addition, sorting, evaluating arithmetic expressions and sequence-retrievals problems.

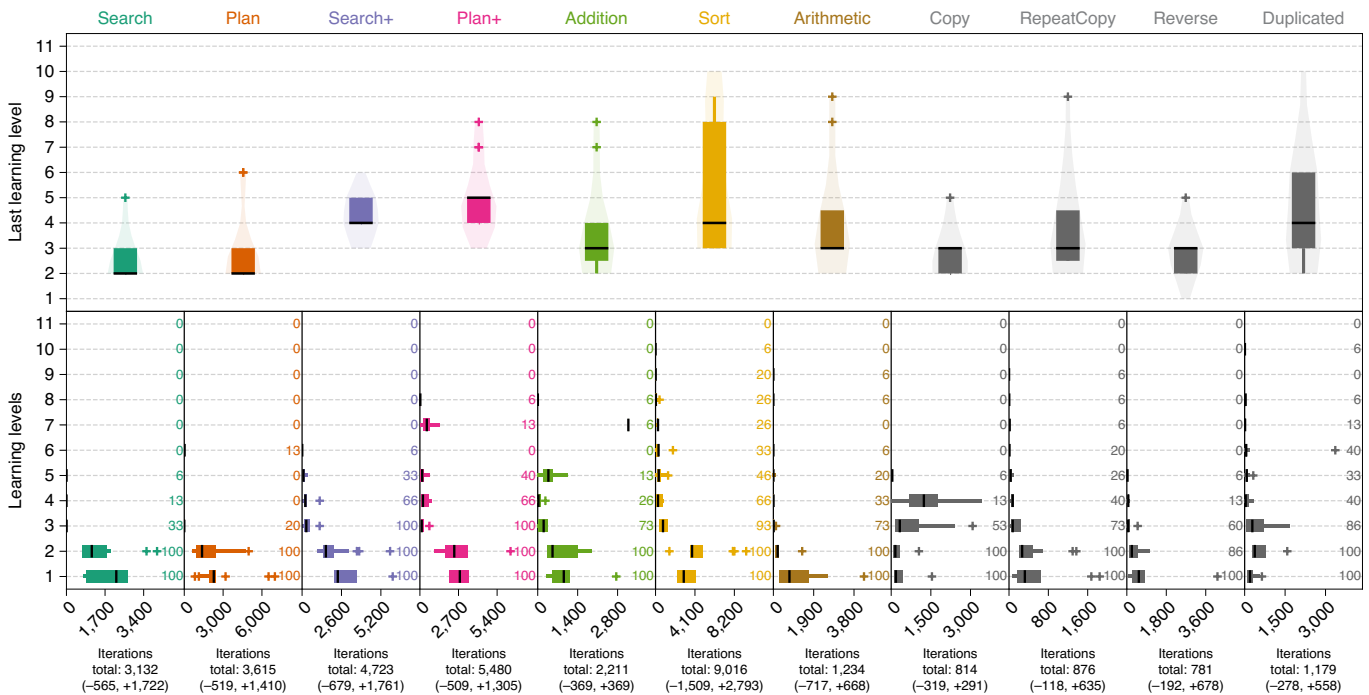


Fig. 3 | Learning overview of all 11 learned algorithms. Top: the last curriculum level that triggered learning (that is, where the last mistake occurred) is shown as the median (black horizontal line), the interquartile range (box) and outliers (plus symbol). The shaded area shows the probability density of the data. All algorithms are learned within the first few levels and the solution generalizes to higher levels. Bottom: the number of training iterations per curriculum level is shown. The coloured numbers indicate the percentage of runs that triggered learning at each level. Learning occurs in the first levels, mostly within the first two, and subsequent levels only need a small amount of iterations to adapt, if at all. The total iterations show median and distances to the interquartile range of the total number of learning iterations. Results are obtained over 15 runs for each algorithm.

In Fig. 2 all 11 algorithms are sketched with their pseudocode and examples (more details can be found in the Supplementary Information).

Learning is performed in a curriculum learning set-up⁴⁰, where the complexity of presented samples increases with each curriculum level. During learning, samples up to curriculum level 10 are considered, with an additional level 11 that samples from all previous levels. Generalization and scaling is tested on complexities up to level 1,000. The direct transfer is tested by transferring the learned solutions to novel problem representations.

Learning procedure overview. The algorithmic modules that encode the algorithmic solution are learned via natural evolution strategies (NES)⁴¹. In each iteration t , a population of P offspring (altered parameters θ_t^o) is generated, and the parameters are updated in the direction of the best-performing offspring. Parameters are updated based on their fitness, a measurement that scores how well the offspring perform. Such optimizers do not require differentiable models, giving more freedom to the model design; for example, using non-differentiable hard memory decisions²⁴ and instantiating the modules freely and flexibly.

An update at iteration $t+1$ of the parameters θ with learning rate α and search distribution variance σ_s^2 is performed as $\theta_{t+1} = \theta_t + \alpha \nabla_{\theta_t}$, with the sampled NES gradient given as

$$\nabla_{\theta_t} \mathbb{E}_{\epsilon \sim \mathcal{N}(0, I)} [f(\theta_t + \sigma_s \epsilon)] \approx \frac{1}{P \sigma_s} \sum_{o=1}^P f(\theta_t^o) \epsilon_i,$$

where \mathbb{E} is the expectation operator and I is the identity matrix. Hence parameters are updated based on a performance-weighted

sum of the offspring. Here the fitness function $f(\cdot)$ scores how many algorithmic steps were performed correctly; that is, the correct data was manipulated in the correct way at the correct step. These binary signals for each step are averaged over all steps and all samples in the minibatch to get a scalar fitness value. This results in a coarse feedback signal and harder learning problem in contrast to gradient-based training, where the error backpropagation gives localized feedback to each parameter.

For all algorithms, generalization and scaling (R1) was tested in two ways. First, testing for scaling to more complex configurations is integrated into our learning procedure and, second, the solutions were tested on complexities far beyond those seen during training.

A curriculum level is considered solved after a defined number of subsequent iterations with maximum fitness, that is, with perfect solutions where every bit in every step is correct. When a new level is unlocked, samples with higher complexity are presented and, hence, if the fitness stays at maximum, the acquired solution scaled to that new complexity. Learning is only performed in iterations that do not have maximum fitness.

In addition to this built-in generalization evaluation, the learned solutions were tested on complexities far beyond those seen during training, that is, corresponding to curriculum levels 100, 500 and 1,000, while being trained only up to level 10.

Learning results. The learning results for all 11 algorithms are presented in Fig. 3, Table 1 and Extended Data Fig. 1, where all results are obtained over 15 runs of each configuration, and Fig. 5 illustrates the learned algorithmic behaviour for four algorithms.

In Figure 3, we illustrate the curriculum levels at which learning was triggered. The top axis shows the last level at which training was triggered (the last level with an error), indicating that learning

Table 1 | Evaluation and comparison

	Search	Plan	Search+	Plan+	Addition	Sort	Arithmetic	Copy	RepeatCopy	Reverse	Duplicated			
NHC	Train	Iterations	3132 ⁺¹⁷²² ₋₅₆₅	3615 ⁺¹⁴¹⁰ ₋₅₁₉	4723 ⁺¹⁷⁶¹ ₋₆₇₉	5480 ⁺¹³⁰⁵ ₋₅₀₉	2211 ⁺³⁶⁹ ₋₃₆₉	9016 ⁺²⁷⁹³ ₋₅₀₉	1234 ⁺⁶⁶⁸ ₋₇₁₇	814 ⁺²⁹¹ ₋₃₁₉	876 ⁺⁶³⁵ ₋₁₈₈	781 ⁺⁶⁷⁸ ₋₁₉₂	1179 ⁺⁵⁵⁸ ₋₂₇₈	
		Last level	2 ⁺¹ ₋₀ 1 ⁺⁰ ₋₀	2 ⁺¹ ₋₀ 1 ⁺⁰ ₋₀	4 ⁺¹ ₋₀ 1 ⁺⁰ ₋₀	5 ⁺⁰ ₋₁ 1 ⁺⁰ ₋₀	3 ⁺⁰ ₋₅ 1 ⁺⁰ ₋₀	3 ^{+1.5} ₋₀ 1 ⁺⁰ ₋₀	3 ⁺⁰ ₋₁ 1 ⁺⁰ ₋₀	3 ^{+1.5} _{-0.5} 1 ⁺⁰ ₋₀	3 ⁺⁰ ₋₁ 1 ⁺⁰ ₋₀	3 ⁺⁰ ₋₁ 1 ⁺⁰ ₋₀	4 ⁺² ₋₁ 1 ⁺⁰ ₋₀	4 ⁺² ₋₁ 1 ⁺⁰ ₋₀
		Level 10	100% 100%	100% 100%	100% 100%	100% 100%	100% 100%	100% 100%	100% 100%	100% 100%	100% 100%	100% 100%	100% 100%	100% 100%
		Level 100	100% 100%	100% 100%	100% 100%	100% 100%	73.3% 86.1%	100% 100%	93.3% 93.3%	80% 80%	100% 100%	100% 100%	100% 100%	100% 100%
		Level 500	100% 100%	100% 100%	100% 100%	80% 80%	73.3% 87.7%	100% 100%	93.3% 93.3%	80% 80%	100% 100%	100% 100%	100% 100%	100% 100%
NHC-anc	Train	Iterations	4319 ⁺⁴⁴⁸ ₋₈₈₈	-	1586 ⁺¹⁴¹⁰ ₋₄₈₈	5677 ⁺⁸⁹² ₋₁₃₁₁	2467 ⁺⁴⁹⁴ ₋₁₀₄₁	1051 ⁺⁸⁴¹ ₋₁₅₉	3496 ⁺⁶⁴⁸ ₋₆₉₇	848 ⁺⁷¹⁷ ₋₅₅₈	2465 ⁺⁶⁰⁷ ₋₁₃₂	848 ⁺⁷¹⁷ ₋₅₅₈	2465 ⁺⁶⁰⁷ ₋₁₃₂	
		Last level	1 ⁺⁰ ₋₀ 0 ⁺⁰ ₋₀	-	3 ⁺¹ ₋₁ 1 ⁺⁰ ₋₀	2 ⁺⁰ ₋₁ 1 ⁺⁰ ₋₁	3 ^{+1.5} ₋₀ 1 ⁺⁰ ₋₀	2 ⁺⁰ ₋₀ 1 ⁺⁰ ₋₀	2 ⁺⁰ ₋₀ 1 ⁺⁰ ₋₁	2 ⁺⁰ ₋₀ 1 ⁺⁰ ₋₁	3 ⁺¹ ₋₁ 1 ⁺⁰ ₋₀	3 ⁺¹ ₋₁ 1 ⁺⁰ ₋₀	2 ⁺⁰ ₋₀ 1 ⁺⁰ ₋₀	
		Level 10	0% 0%	-	100% 100%	0% 0%	86.7% 86.7%	86.7% 86.7%	86.7% 86.7%	0% 1.9%	100% 100%	0% 2.86%	100% 100%	0% 2.86%
		Level 100	-	-	93.9% 93.3%	-	80% 86.3%	86.7% 86.7%	86.7% 86.7%	0% 0%	93.3% 93.3%	0% 0%	93.3% 93.3%	0% 0%
		Level 500	-	-	93.9% 93.3%	-	80% 82.3%	86.7% 86.7%	86.7% 86.7%	-	93.3% 93.3%	-	93.3% 93.3%	-
NHC-prev	Train	Iterations	3913 ⁺⁹³⁷ ₋₇₄₉	-	2722 ⁺⁷⁶⁵ ₋₈₆₈	3784 ⁺¹⁰⁰² ₋₅₆₄	2391 ⁺¹⁴³² ₋₁₂₃	362 ⁺⁶³ ₋₆₃	1118 ⁺²⁶⁷ ₋₆₀₀	322 ⁺⁷⁶² ₋₁₁₈	2398 ⁺²³⁵ ₋₂₃₅	322 ⁺⁷⁶² ₋₁₁₈	2398 ⁺²³⁵ ₋₂₃₅	
		Last level	1 ⁺⁰ ₋₀ 0 ⁺⁰ ₋₀	-	3 ⁺⁰ ₋₁ 1 ⁺⁰ ₋₀	1 ⁺⁰ ₋₀ 0 ⁺⁰ ₋₀	6 ^{+0.5} _{-0.5} 1 ⁺⁰ ₋₀	2 ^{+1.5} ₋₀ 1 ⁺⁰ ₋₀	3 ⁺⁰ ₋₁ 1 ⁺⁰ ₋₀	3 ⁺⁰ ₋₁ 1 ⁺⁰ ₋₀	3 ^{+0.5} ₋₁ 1 ⁺⁰ ₋₀	3 ^{+0.5} ₋₁ 1 ⁺⁰ ₋₀	2 ⁺⁰ _{-0.5} 1 ⁺⁰ _{-0.5}	
		Level 10	0% 0%	-	100% 100%	0% 0%	93.3% 93.3%	100% 100%	100% 100%	100% 100%	86.7% 86.7%	0% 0.7%	86.7% 86.7%	0% 0.7%
		Level 100	-	-	100% 100%	-	93.3% 93.3%	100% 100%	100% 100%	93.3% 93.3%	86.7% 86.7%	0% 0%	86.7% 86.7%	0% 0%
		Level 500	-	-	100% 100%	-	86.7% 93.2%	100% 100%	100% 100%	93.3% 93.3%	86.7% 86.7%	0% 0%	86.7% 86.7%	0% 0%
DNC+is+ha	Train	Iterations	3159 ⁺⁹³⁷ ₋₇₄₉	-	4315 ⁺¹⁵⁸⁴ ₋₃₆₅	5539 ⁺²⁶⁵³ ₋₉₀₅	4360 ⁺¹⁸⁴² ₋₈₄₅	4355 ⁺⁹⁴⁵ ₋₁₁₂₁	4163 ⁺⁴⁷⁵ ₋₅₉₉	2126 ⁺⁷⁶⁶ ₋₁₂₃	3756 ⁺²⁰²⁵ ₋₁₀₄₆	2126 ⁺⁷⁶⁶ ₋₁₂₃	3756 ⁺²⁰²⁵ ₋₁₀₄₆	
		Last level	1 ⁺⁰ ₋₀ 0 ⁺⁰ ₋₀	-	10 ⁺⁰ ₋₀ 1 ⁺⁰ ₋₀	1 ⁺⁰ ₋₀ 0 ⁺⁰ ₋₀	2 ⁺¹ ₋₀ 1 ⁺⁰ ₋₁	10 ⁺⁰ ₋₁ 1 ⁺⁰ ₋₀	10 ⁺⁰ ₋₀ 1 ⁺⁰ ₋₀	10 ⁺⁰ ₋₀ 1 ⁺⁰ ₋₀	10 ⁺⁰ ₋₁ 1 ⁺⁰ ₋₀	2 ⁺¹ ₋₀ 1 ⁺⁰ ₋₁	2 ⁺¹ ₋₀ 1 ⁺⁰ ₋₁	
		Level 10	0% 0%	-	20% 65.2%	0% 0%	0% 0%	0% 0%	100% 100%	20% 92%	100% 100%	0% 5.7%	100% 100%	0% 5.7%
		Level 100	-	-	0% 0%	-	0% 0%	0% 0%	0% 0%	13.3% 13.3%	0% 0%	0% 0%	0% 0%	0% 0%
		Level 500	-	-	-	-	-	-	-	13.3% 13.3%	-	-	-	-
DNC ²⁰	Train	Iterations	500,000	-	500,000	500,000	500,000	500,000	500,000	500,000	500,000	500,000	500,000	
		Last level	1 ⁺⁰ ₋₀ 0 ⁺⁰ ₋₀	-	6 ⁺⁰ ₋₀ 5 ⁺⁰ ₋₀	2 ⁺⁰ ₋₀ 1 ⁺⁰ ₋₀	2 ⁺⁰ ₋₀ 1 ⁺⁰ ₋₀	6 ⁺⁰ ₋₀ 5 ⁺⁰ ₋₀	6 ⁺⁰ ₋₀ 5 ⁺⁰ ₋₀	2 ⁺⁰ ₋₀ 1 ⁺⁰ ₋₀	6 ⁺⁰ ₋₀ 5 ⁺⁰ ₋₀	6 ⁺⁰ ₋₀ 5 ⁺⁰ ₋₀	5 ⁺⁰ ₋₀ 4 ⁺⁰ ₋₀	
		Level 10	0% 0%	-	0% 0%	0% 0%	0% 0%	0% 0%	0% 0%	0% 0%	13.3% 13.3%	0% 0%	13.3% 13.3%	0% 0%
		Level 100	-	-	-	-	-	-	-	0% 0%	-	-	0% 0%	-
		Level 500	-	-	-	-	-	-	-	-	-	-	-	-

Results over 15 runs for each algorithm and model are shown. Triplets such as 5⁺²₋₃ show median and distances to the interquartile range. Iterations refers to the number of learning iterations. The two last-level triplets show the last learning level (left) and the last solved level (right). The two percentages in level X indicate the amount of perfect runs (left), that is, runs that solved all presented samples, and the amount of solved samples over all runs (right). All NHC variants and the DNC+is+ha model use information split and data modules, and are trained with NES. The original DNC is trained in a supervised setting with backpropagation.

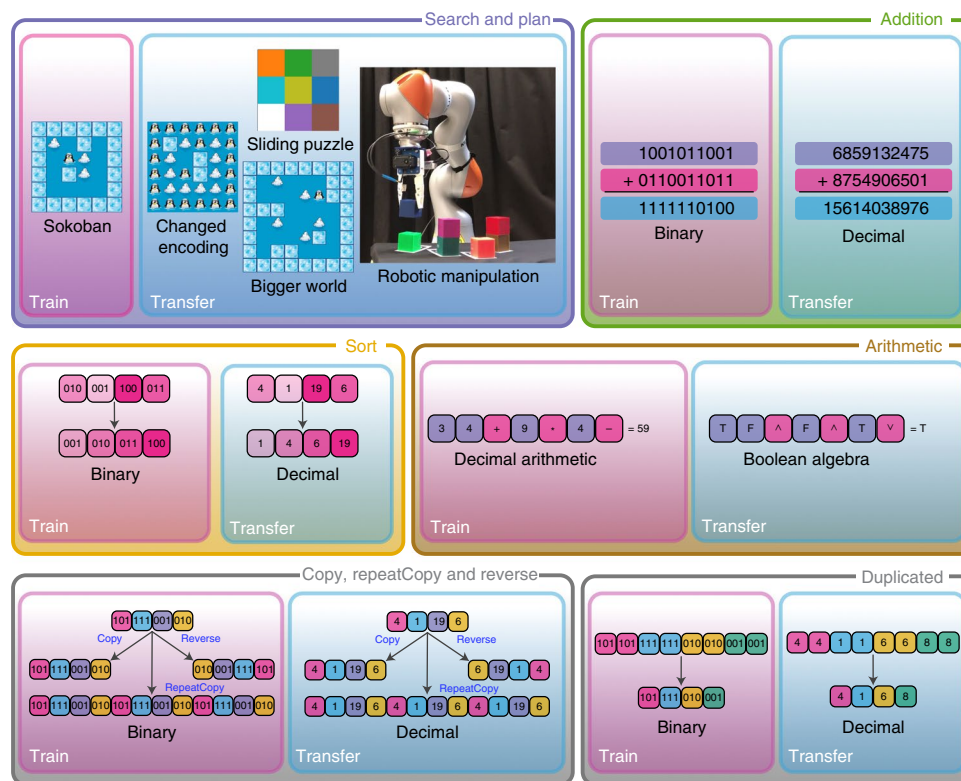


Fig. 4 | Overview of the transfers of the learned algorithms. To show the abstract nature of the learned algorithms, each learned algorithm was transferred and tested on at least one different data representation or domain. All transfers were successful, that is, the learned algorithm solved all samples in the new domain without triggering learning of the algorithmic modules, indicating the fulfilment of R2 and R3.

only occurs at the first levels and solutions generalize to subsequent levels, that is, to higher complexities, which can be observed for all 11 algorithms reliably over all runs. In the bottom axes we investigated how many learning iterations were observed at each level and in total. This highlights the fact that most training happens at the first levels and that subsequent levels only need a few iterations to adapt, if at all. The total number of learning iterations highlights the efficient training in terms of samples. This measure provides an indicator of the task complexity; for example, 9,016 iterations caused network updates for sorting, whereas copying is less challenging and only required 814 iterations of network updates.

More details on the learning, generalization and scaling evaluation for R1—and comparison methods—are shown in Table 1. The last-level entries show the last level to trigger learning alongside the last level that was solved successfully, highlighting that all runs for all algorithms were able to solve all 11 training levels while triggering learning only at the earlier levels. Next, Table 1 shows the results of testing the solutions on complexities far beyond those seen during training. Each run was presented 50 samples from the associated level (20 samples for sort levels 500 and 1,000 due to runtime scaling).

For the majority of algorithms, all runs generalized perfectly to complexities up to level 1,000. In the harder tasks such as sorting, some runs fail for perfect generalization (still performing well while the majority of runs show perfect generalization). Note that, a sample from level 1,000 in the sort task requires over one million perfect computational steps to be considered solved. The performances below 100% for some runs can be explained with the mechanisms of the previous write head. The model has to learn if the previously read location should be updated and with which information, without explicit feedback on these signals. An update mechanism that learned to slightly update the previous location thus works fine

on shorter sequences (such as those seen during training), but the small changes accumulate on longer sequences and may result in incorrect behaviour. A possible solution would be to add feedback to these signals during training if it can be provided.

Overall, the results summarized in Fig. 3 and Table 1 show that the solutions learned by the NHC fulfil algorithmic requirement R1 of generalization.

Comparison. We trained four additional models for comparison. First, the DNC²⁰ model as a state-of-the-art memory-augmented neural. This model is trained in a supervised setting with back-propagation, that is, with a much richer and localized learning signal. It was able to learn some of the baseline algorithms up to level 5, such as addition, copy and reverse, but failed at earlier levels at the remaining tasks, despite being trained for 500,000 iterations. Notably, the DNC struggled with those tasks that required the re-use of intermediate results or iterating over the data multiple times.

Second, we integrated the DNC into the NHC architecture by replacing the algorithmic modules of the NHC (controller, memory, bus) with the original DNC. This DNC+is+ha model uses the same data modules and is trained like the NHC with NES. It performs notably better than the DNC, indicating the help of the proposed abstraction mechanisms and evolutionary training. Nevertheless, it still is not able to generalize comparably to the NHC and struggles with the same algorithms as the DNC. More details on these comparisons and their learning are given in the Supplementary Information.

Next we removed the proposed ancestry linkage (NHC-anc) and the previous location update (NHC-prev) to evaluate their influence. To counter the removed update head, the NHC-prev model uses two write heads, enabling it to learn a similar update mechanism.

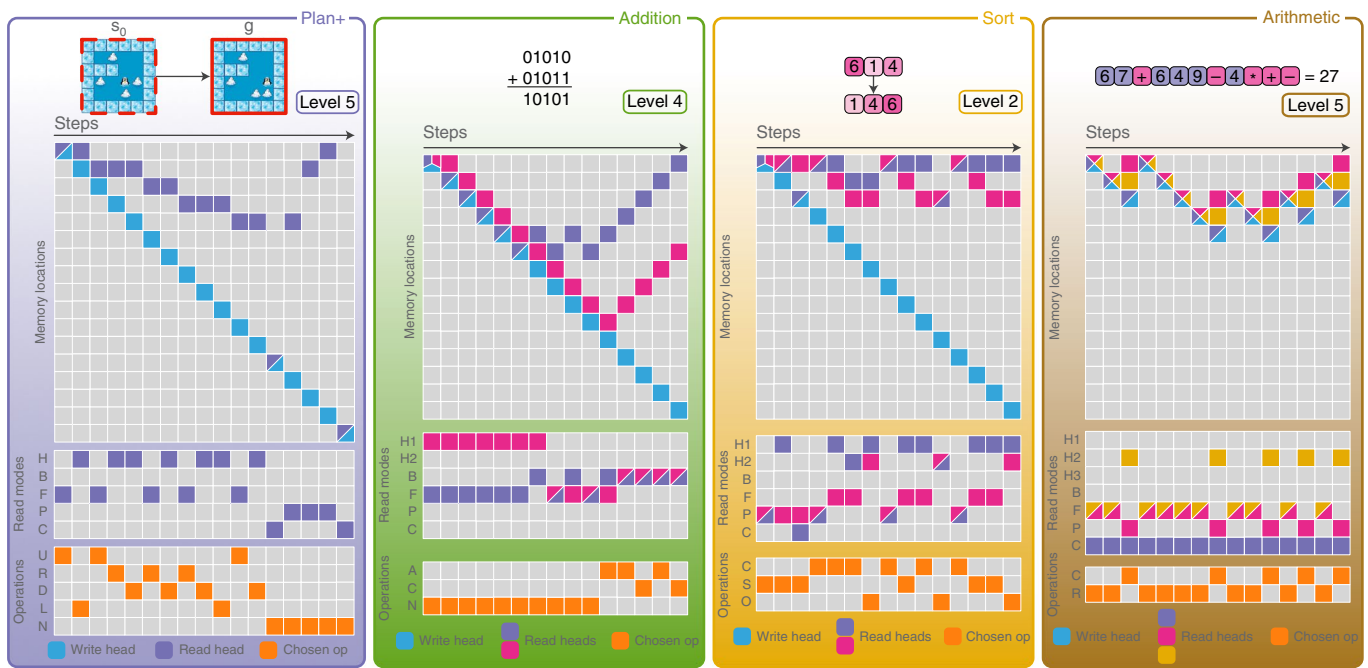


Fig. 5 | Learned algorithmic behaviour of the NHC. The learned behaviour is illustrated for four algorithms that solve the examples shown at the top. The written and read memory locations, the used read mode for each read head and the operation signal sent to the ALU are shown. The example from the (plan+) task shows that the algorithm first builds the search tree by applying all applicable operations in a state and then shifts reading to the next state until the goal is found, it then backtracks the solution. The example from the (addition) task shows that, first, the two numbers to be added are presented after each other and are just stored, the two numbers are then traversed from the low to the high end in parallel, adding the corresponding digits, including possible carry bits. The example from the (sort) task shows that, after reading the unsorted list, the algorithm iterates over the list, finding and outputting the smallest element in each iteration. Finally, the example from the (arithmetic) task shows that the free gates were activated and the model learned to re-use memory locations in order to emulate the behaviour of a stack. The read heads always keep track of the head of the stack and when an arithmetic operation should be applied, it pops the two top elements from the stack, which are then combined by the ALU according to the read operation.

Both models perform better than the DNC+is+ha and are able to learn the majority of algorithms and even achieve perfect generalization in some, strengthening the importance of the evolutionary training and highlighting the influence of the proposed mechanisms. The performance of the two ablation models depends on the algorithm to learn, that is, whether the algorithm requires the hierarchical knowledge provided by the ancestry linkage or the updating of previously read locations. Notably, both mechanisms are required to learn the search and plan algorithms.

These results suggest that the evolutionary training with the proposed abstraction mechanisms and the new memory module are key ingredients for reliably learning algorithmic solutions that generalize and scale, hence fulfilling R1.

Transfer of the learned algorithmic solutions. Next we evaluated the ability to generalize the learned solutions to new problem instantiations, testing the requirements R2 (independence of the data representation) and R3 (independence of the task domain). The algorithmic solutions were therefore tested on unseen data representations and task domains. For these transfers, the learned algorithmic modules were used with adapted data modules for the new set-ups.

All transfers are illustrated in Fig. 4, showing the training set-up and the successful transfers. The transferred solutions solved all 11 curriculum levels in the new set-up without triggering learning once, that is, no single error occurred.

For search and plan, we investigated whether the strategy learned in sokoban could be transferred to larger environments, to a different data representation, to a sliding puzzle problem and to a robot manipulation task. The solutions were learned in 6×6

environments, and could perfectly solve 8×8 environments and a changed encoding of the environment; for example, the penguin represents a wall instead of the agent (see Fig. 4). In the 3×3 sliding puzzles, the white space represents empty space onto which adjacent tiles can be moved. In the robotic set-up, the task is to rearrange the four stacks of boxes from one configuration into another.

Furthermore, sort and the baseline algorithms (copy, repeat-Copy, reverse, duplicated) were trained on binary numbers and were successfully transferred to decimal numbers.

The arithmetic algorithm was trained on decimal arithmetic and was transferred to a boolean algebra. As the atomic operations $+$, $-$, $*$, $/$ are part of the data input sequence, the solution is independent from the number of atomic operations, shown by having only two atomic operations AND & OR in the boolean algebra set-up.

Limitations and assumptions. In our transfer experiments, we assumed the same number of operations available for the ALU and adapted data modules. The number of operations needs to be the same as these, together with the control signals from the Input, form the abstraction interface between data and algorithmic modules. This can be relaxed either by including the domain-specific operations into the data sequence, as shown with the arithmetic transfer, or by extending the interface between Bus and ALU. The learned algorithmic solution is represented by the Controller, Memory and Bus, which encode the abstract strategies fulfilling R1–R3, building on the data modules implementing the abstract interface. As the data modules are domain and representation dependent, they need to be relearned or handcrafted for new set-ups. Typically learning these modules is less complex than learning a new algorithm as they solve smaller subproblems (and often can be hardcoded), and is a

benefit of the modular architecture with its abstraction mechanism and the evolutionary training.

Conclusion

A major challenge for intelligent artificial agents is to learn strategies that scale to higher complexities and that can be transferred to new problem instantiations. We presented a modular architecture for representing and learning such algorithmic solutions that fulfil the three introduced algorithmic requirements: generalization and scaling to arbitrary task configurations and complexities (R1), as well as independence from both the data representation (R2) and the task domain (R3). Algorithmic solutions fulfilling R1–R3 represent strategies that generalize, scale and can be transferred to novel problem instantiations, providing a promising building block for intelligent behaviour.

On a diverse set of 11 algorithms with varying complexities, the proposed NHC was able to reliably learn such algorithmic solutions. These solutions were successfully tested on complexities far beyond seen during training, involving up to over one million recurrent computational steps without a single bit error, and were transferred to novel data representations and task domains. Experimental results highlight the importance of the employed abstraction mechanisms, supporting the ablation study results of past work⁴², providing a potential building block for intelligent agents to be incorporated in other models.

Discussion. The modular structure and the information flow of the NHC enable the learning and transfer of algorithmic solutions, and the incorporation of prior knowledge. Using NES for learning removes constraints on the modules, allowing for arbitrary instantiations and combinations, and the beneficial use of non-differentiable memories²⁴. As the complexity and structure of the algorithmic modules need to be specified, it is an interesting road for future work to learn these in addition, utilizing recent ideas^{24,38}. To speed up computation, parallel models like the neural GPU²² may be incorporated into the NHC architecture.

The presented work showed how algorithmic solutions with R1–R3 can be represented and learned. Based on this foundation, a challenging and interesting research question is how such algorithms can be learned with less feedback. The usage of NES allows to provide different kinds of feedback on any connection in the architecture, and on different time-scales. This opens the opportunity to discover new and unexpected strategies, novel algorithms, and may be achieved by incorporating intrinsic motivation^{43,44} to explore the space of hidden algorithmic solutions in the model.

Methods

In this section a detailed description of the NHC architecture and its modules is given, the learning procedure is described, the task-specific data module instantiations are discussed and details about the comparison methods are given.

All modules are described with their formal functionality (that is, the input signals they receive and the output signals they produce) in the form of $\text{Module}(\text{inputs}) \rightarrow \text{output}$. The information flow is split into d and c , respectively. The superscript indices i, c, b, a, m mark signals coming from the modules Input, Controller, Bus, ALU and Memory, respectively, and index t indicates the computational step. Subscript indices i, c, b, a, m also relate to the respective module to mark the learned parameters of those. In addition to this high-level description, details on how the output signals are generated are given for each module.

The algorithmic modules. The algorithmic modules consist of the Controller, Memory and Bus modules and form the core of the NHC architecture. These modules are learning the algorithmic solution on the control stream and are responsible for the data management in the memory and steer the data manipulation performed by the ALU module. They share similarities with the original DNC²⁰, such as the temporal linkage and usage vectors, but with major changes, for example: hard decisions for the heads and read modes, two coupled memories, simplified and additional attention mechanisms, and more, described in detail in the following sections.

Controller. The controller module receives input from the Input and the signals read from Memory from the previous step. Furthermore, feedback signals from the Bus and ALU from the previous step can be activated, if desired. It produces one output signal going to the Memory and Bus modules, formally given by

$$\text{Ctr}(c_t^{i-c}, c_{t-1}^b, c_{t-1}^a, c_{t-1}^m) \rightarrow c_t^c.$$

Here we use a single layer of size L_C to learn $c_t^c \in (-1, 1)^{L_C}$ at t , given by

$$c_t^c = \tanh(W_c x_c + b_c),$$

where $x_c = [c_t^{i-c}, c_{t-1}^b, c_{t-1}^a, c_{t-1}^m]$. Depending on the task to learn, the feedback signals c_{t-1}^b and c_{t-1}^a can be activated, and more complex instantiations can be used for the controller, such as more layers or recurrent networks.

Memory. The memory module receives signals from the Input and the Controller and is responsible for storing and retrieving information from the two memories. It therefore produces two output signals: a data and a control signal, given by

$$\text{Mem}(d_t^i, c_t^{i-m}, c_t^c) \rightarrow (d_t^m, c_t^m).$$

The memory module has two coupled control and data memories, M^c and M^d , which are matrices of size $N \times C$ and $N \times D$. Multiple write and read heads can be used, where the number of write and read heads is set task-dependently to h_w and h_r , respectively.

Learnable interfaces. Only the concatenated control signals are used as input for all learned layers, that is, $x_m = [c_t^{i-m}, c_t^c]$, and the weight matrices W and biases b are the parameters that are learned.

The write vectors $v_t^i \in \mathbb{R}^C$ at t are the control signals that are stored in M^c via the write heads and are given by $v_t = W_v x_m + b_v$, with v_t split into $\{v_t^i \mid \forall i : h_w\}$ for each write head.

The previous write vectors $\hat{v}_t^i \in \mathbb{R}^C$ at t are the control signals that are used to update M^c and are given by

$$\hat{v}_t = W_{\hat{v}} x_m + b_{\hat{v}},$$

with \hat{v}_t split into $\{\hat{v}_t^i \mid \forall j : h_r\}$ for each read head.

The previous erase vectors $\hat{e}_t^i \in (0, 1)^C$ at t are the control signals used to erase values in M^c and are given by $\hat{e}_t = \sigma(W_e x_m + b_e)$, where $\sigma(\cdot)$ is the logistic sigmoid function and \hat{e}_t is split into $\{\hat{e}_t^i \mid \forall j : h_r\}$ for each read head.

The previous write gate $\hat{g}_t^i \in \{0, 1\}^{h_w}$ at t determines if the memory M^c is updated with \hat{v}_t^i and \hat{e}_t^i , given by

$$\hat{g}_t = H(W_g x_m + b_g),$$

where $H(\cdot)$ is the heavyside step function.

The read modes $m_t^i \in \{0, 1\}^{h_r + 4h_w}$ at t are the control signals that determine which attention mechanism is used to read from the memory, given by $m_t = W_m x_m + b_m$, with m_t split into $\{\text{onehot}(m_t^i) \mid \forall j : h_r\}$ and $\text{onehot}(x) = \{x_k = 1 \text{ if } x_k = \max(x), x_k = 0 \text{ else}\}$.

The free gates $f_t^w \in \{0, 1\}^{h_w}$ and $f_t^r \in \{0, 1\}^{h_r}$ at t determine if locations written to and read from can be freed after interaction, and are given by

$$f_t^w = H(W_{f^w} x_m + b_{f^w}) \quad \text{and} \quad f_t^r = H(W_{f^r} x_m + b_{f^r}).$$

These are all learned parameters of the memory module that define the interfaces to manipulate the memory.

Writing and reading. Given the learned interface described before and the write w_t^i and read r_t^i head locations, information is stored and retrieved from memory as follows.

Writing v_t^i to location w_t^i in M^c at t is performed via

$$M_t^c = M_{t-1}^c \circ (E - w_t^i 1^\top) + w_t^i v_t^{i\top},$$

where $\circ(\cdot)$ denotes element-wise multiplication and E is a matrix of ones of the same size as M^c .

Writing d_t^i to location w_t^i in M^d at t is performed via

$$M_t^d = M_{t-1}^d \circ (E - w_t^i 1^\top) + w_t^i d_t^{i\top},$$

where E is a matrix of ones of the same size as M^d . Note that the same write location w_t^i is used to couple the control and data memories.

Updating the previously read location r_{t-1}^j in M^c is performed via

$$M_t^c = M_{t-1}^c \circ (E - \hat{g}_t^j r_{t-1}^j \hat{e}_t^{j\top}) + \hat{g}_t^j r_{t-1}^j \hat{v}_t^{j\top},$$

where E is a matrix of ones of the same size as M^c . If the previous write gate $\hat{g}_t^j = 0$ no update is performed, and with $\hat{g}_t^j = 1$ the previously read location r_{t-1}^j is erased with \hat{e}_t^j and \hat{v}_t^j is written to it.

Reading from memory is performed via the read locations \mathbf{r}_t^j used on both memories to obtain the data and control output of the memory module via

$$\mathbf{d}_t^{m,j} = M_t^{d,T} \mathbf{r}_t^j \quad \text{and} \quad \mathbf{c}_t^{m,j} = M_t^{c,T} \mathbf{r}_t^j,$$

and are concatenated for the final memory module output $\mathbf{d}_t^m = [\mathbf{d}_t^{m,1}; \dots; \mathbf{d}_t^{m,h_i}]$ and $\mathbf{c}_t^m = [\mathbf{c}_t^{m,1}; \dots; \mathbf{c}_t^{m,h_i}]$.

We next demonstrate how to obtain the head locations in detail.

Head locations. The write and read heads locations ($\mathbf{w}_t^i \in \{0, 1\}^N$ and $\mathbf{r}_t^j \in \{0, 1\}^N$, respectively) are hard decisions, that is, onehot-encoded vectors, where exactly one location is written to or read from, respectively. A simplified dynamic memory allocation scheme from the DNC is used to determine \mathbf{w}_t^i (that is, the memory locations for writing to). It is based on a free list memory allocation scheme, where a linked list is used to maintain the available memory locations. Here, a usage vector $\mathbf{u}_t \in \{0, 1\}^N$ indicates which memory locations are currently used (with $\mathbf{u}_0 = \mathbf{0}$), which are updated in each step with \mathbf{w}_t^i and \mathbf{r}_t^j locations via

$$\mathbf{u}_t = \mathbf{u}_{t-1} + (1 - f_t^{w,i}) \mathbf{w}_t^i \quad \text{and} \quad \mathbf{u}_t = \mathbf{u}_{t-1} (1 - f_t^{r,j} \mathbf{r}_t^j),$$

with the free gates $f_t^{w,i}$ and $f_t^{r,j}$ determining if the write location is marked as used and whether the read location can be freed, respectively. Due to this dynamic allocation scheme, the model is independent from the size of the memory, that is, it can be trained and later used with different sized memories. To obtain the write location \mathbf{w}_t^i , the memory locations are ordered by \mathbf{u}_t , and \mathbf{w}_t^i is set to the first entry in this list, that is, the first unused location is used to write to.

Read head locations \mathbf{r}_t^j are determined by the active read mode given by $\mathbf{m}_t^j \in \{0, 1\}^{h_r + 4h_w}$, that is, only one mode can be active. There are three main attentions implemented for reading from memory: HALT, temporal linkage and ancestry linkage. The total number of available read modes is $h_r + 4h_w$ as HALT is dependent on the number of read heads and both linkages can be used in two directions for each write head.

The HALT attentions are used to read the previously read locations again. When multiple read heads are used, each head can read its own last location or the locations from the other read heads; for example, with three read heads, each head has three HALT attentions (H1, H2 and H3).

The temporal linkage attention is used to read locations in the order they were written, either in forwards or backwards direction. This mechanism enables the architecture to retrieve sequences—or parts of sequences—in the order they were presented or in the reversed order. Here we use a simplified version of the mechanism from the DNC. As our architecture uses hard decisions for the heads locations, the linkages can be stored more efficiently in N -dimensional vectors, in contrast to $N \times N$ matrices in the DNC. Each temporal linkage vector $\mathbf{L}^{t,i}$ stores the order of write locations for one write head, updated at t via

$$\mathbf{L}_t^{t,i} = \mathbf{L}_{t-1}^{t,i} \circ (1 - \mathbf{w}_{t-1}^i) + \tilde{\mathbf{w}}_t^i \mathbf{w}_{t-1}^i,$$

where $\tilde{\mathbf{w}}_t^i = \arg \max(\mathbf{w}_t^i)$. The temporal linkage mechanism can be used in two directions. Either move the read head in the order of which the locations were written, or in reversed order, resulting in two read modes: backwards (B) and forwards (F), per write head for each read head, given by

$$\begin{aligned} \text{B: } \mathbf{r}_t^j &= I(\mathbf{L}_t^{t,i}, \tilde{\mathbf{r}}_{t-1}^j) \quad \text{and} \\ \text{F: } \mathbf{r}_t^j &= \text{onehot}(\mathbf{L}_t^{t,i} \circ \mathbf{r}_{t-1}^j), \end{aligned}$$

where $\tilde{\mathbf{r}}_{t-1}^j$ is the previously read location, $\tilde{\mathbf{r}}_{t-1}^j = \arg \max(\mathbf{r}_{t-1}^j)$ and $I(\mathbf{x}, \mathbf{y}) = \{x_k = 1 \text{ if } x_k = y_k, x_k = 0 \text{ else}\}$. When a location is freed through the free gates, the location is removed from the linkage such that it remains a linked list.

The ancestry linkage also uses N -dimensional vectors to store relations between memory locations. Although the temporal linkage stores information about the order of which locations were written to, the ancestry linkage stores information about which memory locations were read before a location was written, thus capturing a form of usage or hierarchical relation instead of temporal relation. Each ancestry linkage vector $\mathbf{L}^{a,i,j}$ stores which location \mathbf{r}_{t-1}^j was read before location \mathbf{w}_t^i was written, and is updated at t via

$$\mathbf{L}_t^{a,i,j} = \mathbf{L}_{t-1}^{a,i,j} \circ (1 - \mathbf{w}_t^i) + \tilde{\mathbf{r}}_{t-1}^j \mathbf{w}_t^i,$$

where $\tilde{\mathbf{r}}_{t-1}^j$ is the previously read location and $\tilde{\mathbf{r}}_{t-1}^j = \arg \max(\mathbf{r}_{t-1}^j)$. The ancestry linkage mechanism can also be used in two directions: to either move the read head to parent (P) location or the child (C) location. This results in two modes per write head for each read head, given by

$$\begin{aligned} \text{P: } \mathbf{r}_t^j &= \text{onehot}(\mathbf{L}_t^{a,i,j} \circ \mathbf{r}_{t-1}^j) \quad \text{and} \\ \text{C: } \mathbf{r}_t^j &= \text{onehot}(I(\mathbf{L}_t^{a,i,j}, \tilde{\mathbf{r}}_{t-1}^j) \circ \mathbf{h}_t), \end{aligned}$$

where \mathbf{h}_t is a N -dimensional vector storing for each location the t when it was written. A location can be read multiple times, thus it can have multiple children. But as we need a single location to read, the C mode returns the location that was written to the latest when \mathbf{r}_{t-1}^j was read, that is, the newest child. This is

implemented with the history vector \mathbf{h}_t . When a location is freed through the free gates, the location is removed from the linkage and its children are attached to its parent.

Bus. The Bus module is responsible to generate the control signal that indicates how the ALU module should manipulate the data stream, that is, which action or operation to perform. It therefore receives the control signal from the Controller and the Input as well as the output from the memory signal, given by

$$\text{Bus}(\mathbf{c}_t^{i-b}, \mathbf{c}_t^i, \mathbf{c}_t^m) \rightarrow \mathbf{c}_t^b.$$

Here we use a single layer of size L_b to learn $\mathbf{c}_t^b \in \{0, 1\}^{L_b}$ at t , given by

$$\mathbf{c}_t^b = \text{onehot}(W_b \mathbf{x}_b + \mathbf{b}_b),$$

with $\mathbf{x}_b = [\mathbf{c}_t^{i-b}; \mathbf{c}_t^i; \mathbf{c}_t^m]$.

Learning procedure. Learning the algorithmic modules—and hence the algorithmic solution—is performed using NES⁴¹, which is a blackbox optimizer that does not require differentiable models, giving more freedom to the model design; for example, the hard attention mechanisms are not differentiable and the data modules can be instantiated arbitrarily. Recent research showed that NES and related approaches such as Random Search⁴⁵ or NEAT⁴⁶ are powerful alternatives to gradient-based optimization in reinforcement learning. They are easier to implement and scale, perform better with sparse rewards and credit assignment over long time-scales, have fewer hyperparameters⁴⁷ and were used to train memory-augmented networks^{24,38,39}.

Natural evolution strategies updates a search distribution of the parameters to be learned by following the natural gradient towards regions of higher fitness using a P of offspring o (altered parameters) for exploration. The performance of o is measured with one scalar value summarized over all samples N in the minibatch and over all computational steps T_{\max} of each sample, with sparse binary signals for each step, albeit framing a challenging learning problem given the sequence. Let θ be the parameters to be learned (the weight matrices and biases in the three algorithmic modules $\theta = [W_c; b_c; W_v; b_v; W_{\bar{v}}; b_{\bar{v}}; W_{\bar{g}}; b_{\bar{g}}; W_m; b_m; W_{f^w}; b_{f^w}; W_{f^r}; b_{f^r}; W_b; b_b]$); using an isotropic multivariate Gaussian search distribution with fixed variance σ_s^2 , the stochastic natural gradient at t is given by

$$\nabla_{\theta} \mathbb{E}_{e \sim N(0, I)} [u(\theta_t + \sigma_s e)] \approx \frac{1}{P \sigma_s} \sum_{o=1}^P u(\theta_t^o) \epsilon_i,$$

where $u(\cdot)$ is the rank-transformed fitness $f(\cdot)$ ⁴¹. With α , the parameters are updated at t by

$$\theta_{t+1} = \theta_t + \frac{\alpha}{P \sigma_s} \sum_{o=1}^P u(\theta_t^o) \epsilon_i.$$

For all experiments the fitness function is defined for S samples as $f(\theta_t^o) = 1/S \sum_{s=1}^S f_s(\theta_t^o)$ with

$$f_s(\theta_t^o) = \frac{1}{T_{\max}} \sum_{k=1}^{T_s} \delta(\mathbf{d}_k^m - \bar{\mathbf{d}}_k^m) + \delta(\mathbf{c}_k^b - \bar{\mathbf{c}}_k^b) m(\mathbf{c}_k^b)$$

to evaluate the offspring parameters θ_t^o on one sample, s . Here, $\delta(x) = \{1 \text{ if } x = 0, 0 \text{ else}\}$ gives sparse binary reward if the two signals are equal or not, where \mathbf{d}_k^m is the data output from the memory, \mathbf{c}_k^b the control output from the Bus, and $\bar{\mathbf{d}}_k^m$ and $\bar{\mathbf{c}}_k^b$ the true values, respectively. Reward is thus given for choosing the correct data and operation for the ALU in each step. Note that there is no feedback on memory access, only on the output, that is, where, when and how to write and read has to be learned without explicit feedback. The stepwise signals are summed up until the first mistake occurs (T_s) or until the maximum length of the sample T_{\max} , and is normalized with $1/T_{\max}$, that is, $f(\theta_t^o)$ measures the fraction of subsequently correct algorithmic steps. To encourage strong operation choices, the operation reward is multiplied with the margin penalty

$$m(\mathbf{c}_k^b) = \text{clip}\left(\frac{\bar{c}^1/\bar{c}^2 - 1}{m_{\max}}, 0, 1\right),$$

where \bar{c}^1, \bar{c}^2 are first and second largest values of \mathbf{c}_k^b , that is, the chosen operation and the runner up, and m_{\max} is a chosen percentage indicating how much bigger the chosen action should be. Note that this penalty is only considered if the operation is already correct.

For robustness and learning efficiency, weight decay for regularization⁴⁸ and automatic restarts of runs stuck in local optima are used⁴¹. This restarting can be seen as another level of evolution, where some lineages die out. Another way of dealing with early converged or stuck lineages is to add intrinsic motivation signals such as novelty, which help to get attracted by another local optima, as in NSRA-ES⁴⁹. In the experiments however, we found that within our setting,

restarting—or having an additional survival of the fittest on the lineages—was more effective in terms of training time.

The algorithmic solutions are learned in a curriculum learning set-up⁴⁰, with sampling from old lessons to prevent unlearning and to foster generalization. Furthermore, we created bad memories, a learning from mistakes strategy similar to the idea of AdaBoost⁵⁰, which samples previously failed samples to encourage focusing on the hard cases. This can also be seen as a form of experience replay⁵¹, but only using the initial input data to the model, not the full generated sequences. Bad memories were initially developed for training the data-dependent modules to ensure their robustness and 100% accuracy, which is crucial to learn algorithmic solutions. If the individual modules do not have 100% accuracy, no stable algorithmic solution can be learned even if the algorithmic modules are doing the correct computations. For example, if one module has an accuracy of 99%, the 1% error prevents learning an algorithmic solution that always works. This problem is even reinforced as the proposed model is an output–input architecture that works over multiple computation steps using its own output as the new input, meaning the overall accuracy drops to 36.6% for 100 computation steps. Using the bad memories strategy, and thus focusing on the mistakes, therefore considerably helps with achieving robust results when learning the modules, enabling the learning of algorithmic solutions.

Experimental set-up. In all experiments, the hyperparameters were set to: batch size $S=32$, $P=20$, $\alpha=0.01$, search distribution exploration $\sigma=0.1$, weight decay $\lambda=0.9995$, action margin $m_{\max}=0.1$, max iterations=20,000, restart iterations=2,000. In each batch, 33% of the samples were drawn from previous levels and another 33% were drawn from the bad memories buffer, which stores the last 200 mistakes. A curriculum level is considered solved when 750 subsequent iteration are perfectly solved, that is, no single mistake in any sample, any step, any bit, that is 24,000 perfectly solved samples. In levels were training was triggered, the required subsequent perfect iterations are doubled, that is, 48,000 perfectly solved samples. Whenever an iteration achieves maximum fitness, no learning is triggered, that is, no parameter update is performed.

The modules instantiations. The preceding sections described the design and functionality of the algorithmic modules in general. Here, the used instantiations and parameters for the experiments are presented, as well as the data modules and their task-dependent instantiations.

Algorithmic modules. In all experiments, $L_c=6$ and $C=4$. All tasks use $h_w=1$ and $h_s=1$ for the four search and plan, and the four copy tasks, whereas $h_s=2$ for addition and sort, and $h_s=3$ for the arithmetic task; D and L_b are set by the task, as each task has a different data representation (D) and a different amount of available operations for the ALU (L_b). In all tasks the ALU to Controller feedback (c_{t-1}^i) was activated, except for the four copy tasks as the ALU has no functionality there. The free gates were activated for the four copy tasks and the arithmetic task. In total, depending on the algorithm to learn, this results in 300–650 trainable parameters in the algorithmic modules.

Input. The first data-dependent module is the Input module. That is the interface to receive data and provide control signals. It receives the data input from the outside \mathbf{d}_t^{in} as well as the data output from the ALU from the previous computational step $\mathbf{d}_{t-1}^{\text{out}}$, formally given as

$$\text{In}(\mathbf{d}_t^{\text{in}}, \mathbf{d}_{t-1}^{\text{out}}) \longrightarrow (\mathbf{d}_t^i, \mathbf{c}_t^{i-c}, \mathbf{c}_t^{i-m}, \mathbf{c}_t^{i-b}).$$

The main functionality is to generate task related control signals, data preprocessing if applicable and determining to stop. The control signals \mathbf{c}_t^{i-c} , \mathbf{c}_t^{i-m} , \mathbf{c}_t^{i-b} may be unique to provide different signals to the Controller, Memory and Bus, but can also share the same information. The data \mathbf{d}_t^i is forwarded to the memory module with or without preprocessing, depending on the task.

ALU. The ALU module is responsible for data manipulation. It receives the read data from the memory and the operation to apply on these from the Bus to produce the next data output alongside control signals via

$$\text{ALU}(\mathbf{d}_t^m, \mathbf{c}_t^b) \longrightarrow (\mathbf{d}_t^{\text{out}}, \mathbf{c}_t^a).$$

This module implements elemental operations for each task such that the algorithmic solution can be learned by applying the correct operation on the correct data in the correct step.

Both data modules can be instantiated arbitrarily due to the NES approach for learning the algorithmic solution. They can also be trained from data beforehand or be hardcoded if possible. In the experiments, we tested both variations and details for each algorithm are given in the Supplementary Information.

Data availability

Data is generated online during training and the generating methods are provided in the source code.

Code availability

The source code of the NHC is available via Code Ocean at <https://doi.org/10.24433/CO.6921369.v1> (ref. 52).

Received: 15 June 2020; Accepted: 10 October 2020;
Published online: 16 November 2020

References

- Taylor, M. E. & Stone, P. Transfer learning for reinforcement learning domains: a survey. *J. Mach. Learn. Res.* **10**, 1633–1685 (2009).
- Silver, D. L., Yang, Q. & Li, L. Lifelong machine learning systems: beyond learning algorithms. In *2013 AAAI Spring Symposium: Lifelong Machine Learning* Vol. 13, 49–55 (AAAI, 2013).
- Weiss, K., Khoshgoftaar, T. M. & Wang, D. A survey of transfer learning. *J. Big Data* **3**, 9 (2016).
- Parisi, G. I., Kemker, R., Part, J. L., Kanan, C. & Wermter, S. Continual lifelong learning with neural networks: a review. *Neural Networks* **113**, 54–71 (2019).
- Schmidhuber, J. Deep learning in neural networks: an overview. *Neural Networks* **61**, 85–117 (2015).
- Mnih, V. et al. Human-level control through deep reinforcement learning. *Nature* **518**, 529–533 (2015).
- Silver, D. et al. Mastering the game of Go with deep neural networks and tree search. *Nature* **529**, 484–489 (2016).
- Fawaz, H. I., Forestier, G., Weber, J., Idoumghar, L. & Muller, P.-A. Deep learning for time series classification: a review. *Data Min. Knowl. Discov.* **33**, 917–963 (2019).
- Botvinick, M. et al. Reinforcement learning, fast and slow. *Trends Cogn. Sci.* **23**, 408–422 (2019).
- Liu, L. et al. Deep learning for generic object detection: a survey. *Int. J. Compu. Vis.* **128**, 261–318 (2020).
- Tenenbaum, J. B., Kemp, C., Griffiths, T. L. & Goodman, N. D. How to grow a mind: statistics, structure, and abstraction. *Science* **331**, 1279–1285 (2011).
- Lake, B. M., Ullman, T. D., Tenenbaum, J. B. & Gershman, S. J. Building machines that learn and think like people. *Behav. Brain Sci.* **40**, e253 (2017).
- Konidaris, G. On the necessity of abstraction. *Curr. Opin. Behav. Sci.* **29**, 1–7 (2019).
- Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. *Introduction to Algorithms* (MIT Press, 2009).
- Das, S., Giles, C. L. & Sun, G.-Z. Learning context-free grammars: capabilities and limitations of a recurrent neural network with an external stack memory. In *Proc. 14th Annual Conference of the Cognitive Science Society* 791–795 (The Cognitive Science Society, 1992).
- Mozer, M. C. & Das, S. A connectionist symbol manipulator that discovers the structure of context-free languages. In *Advances in Neural Information Processing Systems* 863–870 (1993).
- Zeng, Z., Goodman, R. M. & Smyth, P. Discrete recurrent neural networks for grammatical inference. *IEEE Trans. Neural Netw. Learn. Syst.* **5**, 320–330 (1994).
- Graves, A., Wayne, G. & Danihelka, I. Neural turing machines. Preprint at <https://arxiv.org/abs/1410.5401> (2014).
- Joulin, A. & Mikolov, T. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Advances in Neural Information Processing Systems* 190–198 (2015).
- Graves, A. et al. Hybrid computing using a neural network with dynamic external memory. *Nature* **538**, 471–476 (2016).
- Neelakantan, A., Le, Q. V. & Sutskever, I. Neural programmer: inducing latent programs with gradient descent. In *International Conference on Learning Representations* (2016).
- Kaiser, E. & Sutskever, I. Neural GPUs learn algorithms. *International Conference on Learning Representations* (2016).
- Zaremba, W., Mikolov, T., Joulin, A. & Fergus, R. Learning simple algorithms from examples. In *International Conference on Machine Learning*, 421–429 (2016).
- Greve, R. B., Jacobsen, E. J. & Risi, S. Evolving neural turing machines for reward-based learning. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, 117–124 (ACM, 2016).
- Trask, A. et al. Neural arithmetic logic units. In *Advances in Neural Information Processing Systems* 8035–8044 (2018).
- Madsen, A. & Johansen, A. R. Neural arithmetic units. In *International Conference on Learning Representations* (2020).
- Le, H., Tran, T. & Venkatesh, S. Neural stored-program memory. In *International Conference on Learning Representations* (2020).
- Reed, S. & De Freitas, N. Neural programmer-interpreters. *International Conference on Learning Representations* (2016).
- Kurach, K., Andrychowicz, M. & Sutskever, I. Neural random-access machines. *International Conference on Learning Representations* (2016).

30. Cai, J., Shin, R. & Song, D. Making neural programming architectures generalize via recursion. *International Conference on Learning Representations* (2017).
31. Dong, H. et al. Neural logic machines. In *International Conference on Learning Representations* (2019).
32. Velickovic, P., Ying, R., Padovano, M., Hadsell, R. & Blundell, C. Neural execution of graph algorithms. In *International Conference on Learning Representations* (2020).
33. Sukhbaatar, S., Weston, J., Fergus, R. et al. End-to-end memory networks. In *Advances in Neural Information Processing Systems*, 2440–2448 (2015).
34. Weston, J., Chopra, S. & Bordes, A. Memory networks. In *International Conference on Learning Representations* (2015).
35. Grefenstette, E., Hermann, K. M., Suleyman, M. & Blunsom, P. Learning to transduce with unbounded memory. In *Advances in Neural Information Processing Systems* 1828–1836 (2015).
36. Kumar, A. et al. Ask me anything: dynamic memory networks for natural language processing. In *International Conference on Machine Learning* 1378–1387 (2016).
37. Wayne, G. et al. Unsupervised predictive memory in a goal-directed agent. Preprint at <https://arxiv.org/abs/1803.10760> (2018).
38. Merrild, J., Rasmussen, M. A. & Risi, S. HyperNTM: evolving scalable neural turing machines through HyperNEAT. In *International Conference on the Applications of Evolutionary Computation* 750–766 (Springer, 2018).
39. Khadka, S., Chung, J. J. & Tumer, K. Neuroevolution of a modular memory-augmented neural network for deep memory problems. *Evol. Comput.* **27**, 639–664 (2019).
40. Bengio, Y., Louradour, J., Collobert, R. & Weston, J. Curriculum learning. In *International Conference on Machine Learning* 41–48 (ACM, 2009).
41. Wierstra, D. et al. Natural evolution strategies. *J. Mach. Learn. Res.* **15**, 949–980 (2014).
42. Tanneberg, D., Rueckert, E. & Peters, J. Learning algorithmic solutions to symbolic planning tasks with a neural computer architecture. Preprint at <https://arxiv.org/abs/1911.00926> (2019).
43. Oudeyer, P.-Y. & Kaplan, F. What is intrinsic motivation? A typology of computational approaches. *Front. Neurobotics* **1**, 6 (2009).
44. Baldassarre, G. & Mirrolli, M. Intrinsically motivated learning systems: an overview. In *Intrinsically Motivated Learning in Natural and Artificial Systems* 1–14 (Springer, 2013).
45. Mania, H., Guy, A. & Recht, B. Simple random search of static linear policies is competitive for reinforcement learning. In *Advances in Neural Information Processing Systems* 1803–1812 (2018).
46. Stanley, K. O. & Miikkulainen, R. Evolving neural networks through augmenting topologies. *Evol. Comput.* **10**, 99–127 (2002).
47. Salimans, T., Ho, J., Chen, X., Sidor, S. & Sutskever, I. Evolution strategies as a scalable alternative to reinforcement learning. Preprint at <https://arxiv.org/abs/1703.03864> (2017).
48. Krogh, A. & Hertz, J. A. A simple weight decay can improve generalization. In *Advances in Neural Information Processing Systems* 950–957 (1992).
49. Conti, E. et al. Improving exploration in evolution strategies for deep reinforcement learning via a population of novelty-seeking agents. In *Advances in Neural Information Processing Systems* 5027–5038 (2018).
50. Freund, Y. & Schapire, R. E. A decision-theoretic generalization of online learning and an application to boosting. *Journal Comput. Syst. Sci.* **55**, 119–139 (1997).
51. Lin, L.-J. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Mach. Learn.* **8**, 293–321 (1992).
52. Tanneberg, D. *The Neural Harvard Computer* (Code Ocean, accessed 25 September 2020); <https://doi.org/10.24433/CO.6921369.v1>.

Acknowledgements

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement nos. 713010 (GOAL-Robots) and 640554 (SKILLS4ROBOTS), and from the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under no. 430054590. This research was supported by NVIDIA. We want to thank K. O'Regan for inspiring discussions on defining algorithmic solutions.

Author contributions

D.T. conceived the project, designed and implemented the model, conducted the experiments and analysis, created the graphics. D.T., E.R. and J.P. wrote the manuscript.

Competing interests

The authors declare no competing interests.

Additional information

Extended data is available for this paper at <https://doi.org/10.1038/s42256-020-00255-1>.

Supplementary information is available for this paper at <https://doi.org/10.1038/s42256-020-00255-1>.

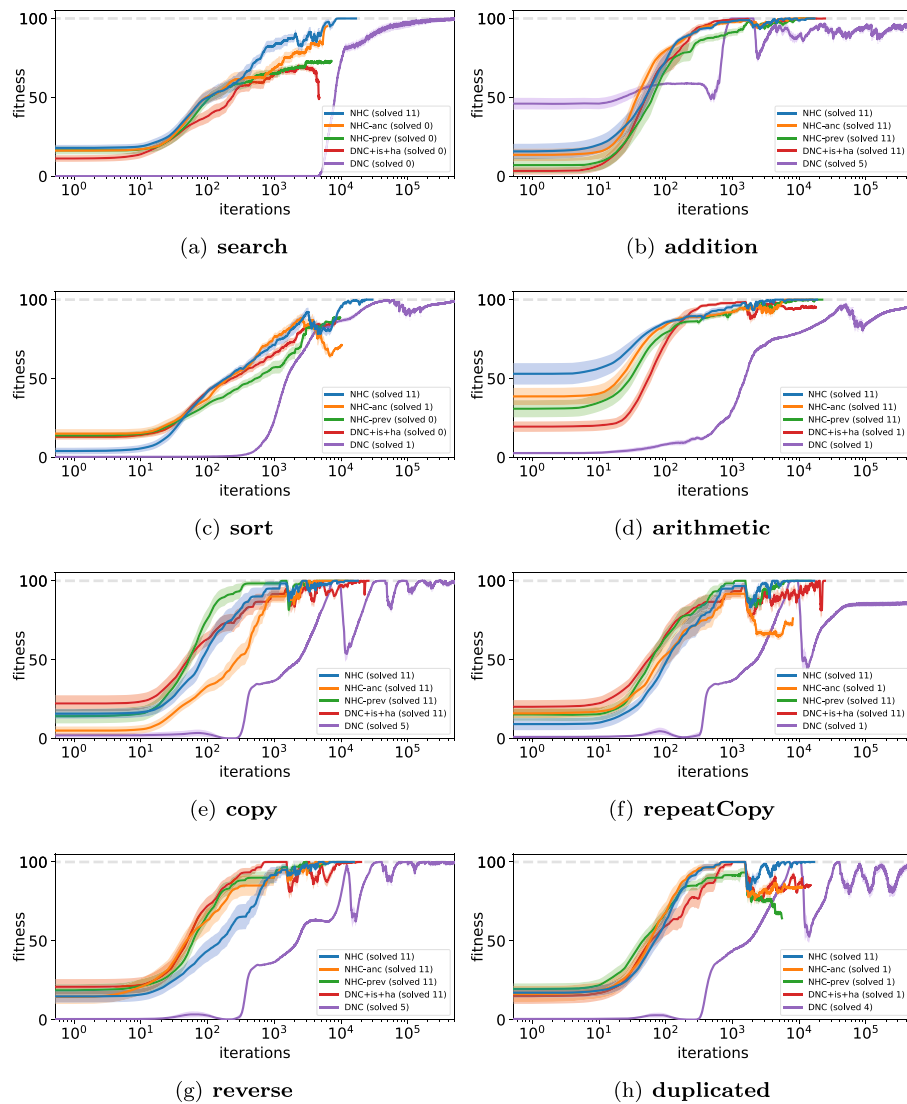
Correspondence and requests for materials should be addressed to D.T.

Peer review information Nature Machine Intelligence thanks Greg Wayne and the other, anonymous, reviewer(s) for their contribution to the peer review of this work.

Reprints and permissions information is available at www.nature.com/reprints.

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

© The Author(s), under exclusive licence to Springer Nature Limited 2020



Extended Data Fig. 1 | Learning curves comparison. Shown are the mean and the standard error of the fitness during learning over 15 runs. Note the log-scale of the x-axis. Solved X in the legend indicates the median solved level. The full NHC is the only model that successfully learns all algorithms reliably. More details on these evaluations are given in Table 1.