Symbolic Reasoning with Differentiable Neural Computers

Alex Graves*, Greg Wayne*, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gomez, Edward Grefenstette, Tiago Ramalho, John Agapiou, Adrià Puigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, Demis Hassabis.

*Joint first authors

Recent breakthroughs demonstrate that neural networks are remarkably adept at sensory processing[1] and sequence[2,3] and reinforcement learning[4]. However, cognitive scientists and neuroscientists have argued that neural networks are limited in their ability to define variables and data structures[5–9], store data over long time scales without interference[10,11], and manipulate it to solve tasks. Conventional computers, on the other hand, can easily be programmed to store and process large data structures in memory, but cannot learn to recognise complex patterns. This work aims to combine the advantages of neural and computational processing by providing a neural network with read-write access to an external memory. We refer to the resulting architecture as a *Differentiable Neural Computer* (DNC). Memory access is sparse, minimising interference among memoranda and enabling long-term storage[12,13], and the entire system can be trained with gradient descent, allowing the network to learn how to operate and organise the memory in a goal-directed manner. We demonstrate DNC's ability to manipulate large data structures by applying it to a set of synthetic question-answering tasks involving graphs, such as finding shortest paths and inferring missing links. We then show that DNC can learn, based solely on behavioral reinforcement[14,15], to carry out complex symbolic instructions in a game environment[16]. Taken together, these results suggest

that DNC is a promising model for tasks requiring a combination of pattern recognition and symbol manipulation, such as question-answering and memory-based reinforcement learning.

Modern computers separate computation and memory. Computation is performed by a processor, which can use an addressable memory to bring operands in and out of play. This confers on the computer two important properties: it provides extensible storage to write new information as it arrives and the ability to treat the contents of memory locations as variables. Variables are critical to algorithm generality: to perform the same procedure on one datum or another, an algorithm merely has to change the address it looks up or the content of the address. By contrast to computers, the computational and memory resources of artificial neural networks are mixed together in the network weights and neuron activity. This is a major liability: as the memory demands of a task increase, these networks cannot allocate new storage dynamically, nor easily learn algorithms that act independently of the values realised by the task variables.

The *Differentiable Neural Computer* (DNC) is a neural network coupled to an external memory matrix (Figure 1). The behaviour of the controller network is independent of the memory size as long as the memory is not filled to capacity, which is why we view the memory as "external". If the memory can be thought of as DNC's RAM, then the network, referred to as the *controller*, is a CPU whose operations are learned. DNCs differ from recent neural memory frameworks[17,18] in that the memory can be selectively written to as well as read, allowing iterative modification of memory content. An earlier form of DNC, the *Neural Turing Machine*[19], had a similar structure but less flexible memory access methods (Methods).

While conventional computers use unique addresses to access memory contents, DNC uses differentiable attention mechanisms[2, 19–22] to define distributions over the rows, or *locations*, in the memory matrix. These distributions, which we call *weightings*, represent the degree to which each location is involved in a read or write operation, and are typically very sparse in a trained system. For example, the *read vector* $\mathbf{r}$ returned by weighting $\mathbf{w}$ over memory $\mathbf{M}$ is simply a weighted sum over the N memory locations: $\mathbf{r} = \sum_{i=1}^{N} \mathbf{M}[i, .]\mathbf{w}[i]$. The functional units that determine and apply the weightings are called *read* and *write heads*. Crucially, the heads are differentiable, allowing the complete system to learn by gradient descent.

The heads employ three distinct forms of attention. The first is content lookup[19, 20, 23–25] in which a *key* emitted by the controller is compared to the content of each location in memory according to a similarity measure (here: cosine similarity). The similarity scores determine a weighting that can be used by the read heads for associative recall[26] or by the write head to modify an existing vector in memory. Importantly, a key that only partially matches the content of a memory location can still be used to attend strongly to that location. This enables key-value retrieval where the value recovered by reading the memory location includes additional information not present in the key. Key-value retrieval provides a rich mechanism for navigating associative data structures in the external memory, as the content of one address can effectively encode references to other addresses. In our experiments, this proved essential to processing graph data.

A second attention mechanism records transitions between consecutively written locations

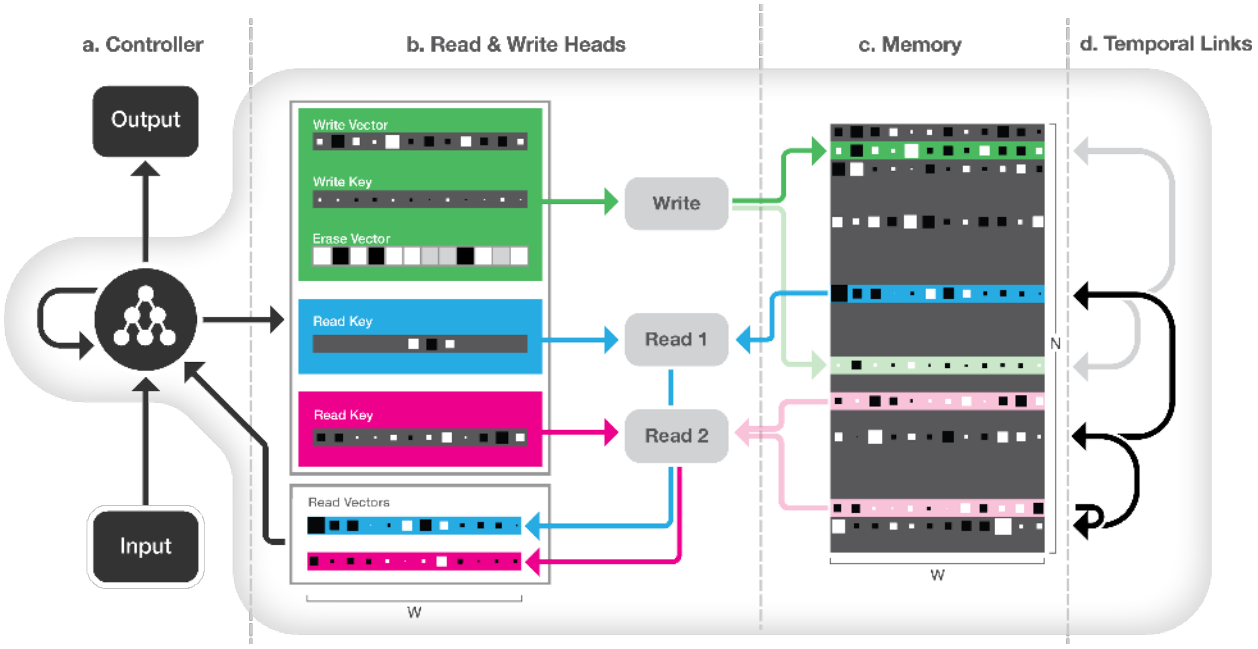


Figure 1: **DNC Architecture**. *a:* A recurrent *controller* network receives input from an external data source and produces output. *b & c:* The controller also outputs vectors that parameterise one *write head* (green) and multiple *read heads* (two in this case: blue and pink). The heads define *weightings* that selectively focus on the rows, or *locations*, in the *memory* matrix (stronger colour for higher weight). The *read vectors* returned by the read heads are passed to the controller at the next time step. *d:* A *temporal link matrix* records the order locations were written in; here, we represent the order locations were written to using directed arrows. The grey arrows indicates a write event that was split between two locations.

in an $N \times N$ temporal link matrix $\mathbf{L}$ (Figure 1d). $\mathbf{L}[i, j]$ is close to one if i was the next location written after j, and is close to zero otherwise. For any weighting $\mathbf{w}$, the operation $\mathbf{L}\mathbf{w}$ smoothly shifts the focus *forward* to the locations written after those emphasised in $\mathbf{w}$, while $\mathbf{L}^\top\mathbf{w}$ shifts the focus *backward*. This gives DNC the native ability to recover sequences in the order in which they were presented.

The third form of attention allocates memory for writing. The *usage* of each location is represented as a number between zero and one. Based on the usages, a weighting over unused locations is delivered to the write head. As well as automatically increasing with each write to a location, usage can be decreased after each read using the *free gates*. This allows the controller to reallocate memory that is no longer required (Supplementary Figure 3). As a consequence of its allocation mechanism, DNC can be trained to solve a task using one size of memory and later be upgraded to a larger memory without retraining and without any impact on performance (Supplementary Figure 1). This property would also make it possible to use an unbounded external memory by automatically increasing the number of locations every time the usage of all locations passes a certain threshold.

Although the design of DNC was motivated largely by computational considerations, we cannot resist drawing some connection between the attention mechanisms and the mammalian hippocampus' functional capabilities. DNC memory modification is fast and can be one-shot, resembling the associative long-term potentiation of hippocampal CA3 and CA1 synapses[27]. The hippocampal dentate gyrus, a region known to support neurogenesis[28], has been proposed to increase representational sparsity, thereby enhancing memory capacity[29]: usage-based memory allocation and sparse weightings may provide similar facilities in our model. Human "free recall" experiments demonstrate the increased probability of item recall in the same order as first presented, a hippocampus-dependent phenomenon accounted for by the temporal context model[30], bearing some similarity to the formation of temporal links (Methods).

Our first experiments investigated DNC's capacity to perform question answering, a paradigmatic example of symbolic reasoning. To compare DNC to other neural network architectures, we considered the bAbI dataset[31], which includes 20 types of synthetically generated questions designed as a benchmark test for textual reasoning. The dataset consists of short "story" snippets followed by questions with answers that can be inferred from the stories: for example, the story *"John is in the playground. John picked up the football. Where is the football? [playground]"* requires a system to combine two supporting facts, while *"Sheep are afraid of wolves. Gertrude is a sheep. Mice are afraid of cats. What is Gertrude afraid of? [wolves]"* tests its facility with basic deduction (and resilience to distractors). We found that a single DNC network, jointly trained on all 20 question types with 10,000 instances each, was able to achieve a mean test error rate of 3.8% with task failure (defined as $> 5\%$ error) on 2 types of questions, compared to 7.5% mean error and 6 failed tasks for the best previous jointly trained result[25]. We also found that DNC performed much better than either Long Short-Term Memory[32] (LSTM; at present the benchmark neural network for sequence processing) or the Neural Turing Machine[19] (see Supplement for details). In contrast to previous results on this data set, the inputs to our models were single word tokens without any preprocessing or handcrafted features (Methods).

Although bAbI is presented in natural language, each declarative sentence involves a limited vocabulary and is generated from a simple tuple containing an actor, an action, and a set of arguments. Such sentences could easily be rendered in graphical form: for example "John is in the playground" can be diagrammed as two named nodes, "Playground" and "John", connected by a named edge "Contains". In this sense, the propositional knowledge in many of the bAbI tasks is equivalent to a set of constraints on an underlying graph structure. The state-of-the-art performance of DNC on bAbI indicates that it inferred the underlying structure and was able to reason about it, but to further investigate DNC's capacity to reason about complex structures, we designed a set of reasoning problems on larger scale graphs.

Unlike bAbI, the edges in our graphs were presented explicitly, with each input vector specifying a *triple* consisting of two node labels and an edge label. Other than the input representation, the setup was identical to the bAbI tasks, with a sequence of inputs followed by a query and a required sequence of outputs. The labelling and connectivity of the graphs was random, as was the order in which the triples were presented; this ensured that the networks could not overfit on particular graphical structures or memorise specific labellings.

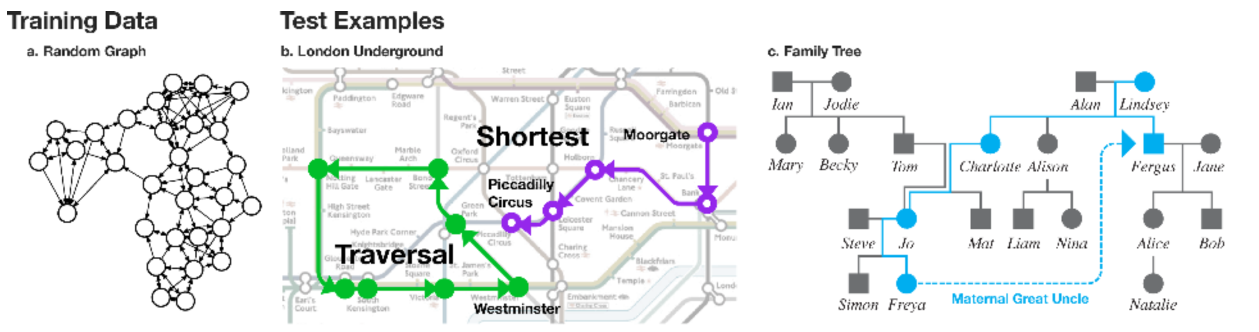
The three types of query we considered are illustrated in Figure 2. For *path traversal* (Figure 2b), the network was instructed to report the nodes that follow from a start node along a sequence of edge labels generated by a random walk. Because the edge labels were unique per node but not across the graph, the network had to remember which node it was on and what each edge connected to next.

For *shortest path* (Figure 2b), a random start and end node were given as the query, and the network was asked to return a sequence of triples corresponding to a minimum length path between them. Since we considered paths of up to length 5, this can be seen as a harder version of the *path finding* task in the bAbI dataset, which had a maximum length of 2.

For *inferred relations*, we predefined a set of sequences of up to five connected edge labels, which we defined as implicit *relations* connecting two nodes. A query consisted of a single triple specifying a start node and a relation label, and the required output was the final node in the relation sequence. For example, the relation labelled "paternal grandmother" would be specified by the edge sequence "father, mother", and when queried with "paternal grandmother of X" the net would have to follow the "father" link from "X" to some "Y", then follow the "mother" link from "Y" and return the result. Each relation was given a unique label that was never presented as part of the graph description; the edge sequence they corresponded to therefore had to be inferred by the network. Inferring missing edges is important in domains such as question answering and recommender systems, giving this synthetic task much real-world relevance.

After training, the networks were tested on two specific graphs: a symbolic map of the London Underground and a family tree, as shown in Figure 2. This was done both to ensure that they generalised beyond the random graphs used for training, and to give intuition into how the tasks relate to real-world data.

As a benchmark we again compared DNC with LSTM. In this case the best LSTM network we found in an extensive hyper-parameter search failed to complete the first level of its training



Underground Input:
(OxfordCircus, TottenhamCtRd, Central)
(TottenhamCtRd, OxfordCircus, Central)
(BakerSt, Marylebone, Circle)
(BakerSt, Marylebone, Bakerloo)
(BakerSt, OxfordCircus, Bakerloo)

...

(LeicesterSq, CharingCross, Northern)
(TottenhamCtRd, LeicesterSq, Northern)
(OxfordCircus, PicadillyCircus, Bakerloo)
(OxfordCircus, NottingHillGate, Central)
(OxfordCircus, Euston, Victoria)

- 84 edges in total

Traversal Question:
(OxfordCircus, _, Central), (_, _, Circle)
(_, _, Circle), (_, _, Circle),
(_, _, Bakerloo), (_, _, Victoria),
(_, _, Victoria), (_, _, Circle),
(_, _, Bakerloo), (_, _, Jubilee)

Answer:
(OxfordCircus, NottingHillGate, Central)
(NottingHillGate, Paddington, Circle)
...
(Embankment, Waterloo, Bakerloo)
(Waterloo, GreenPark, Jubilee)

Shortest Path Question:
(Moorgate, PicadillyCircus, _)

Answer:
(Moorgate, Bank, Northern)
(Bank, Holborn, Central)
(Holborn, LeicesterSq, Picadilly)
(LeicesterSq, PicadillyCircus, Picadilly)

Family Tree Input:
(Charlotte, Alan, Father)
(Simon, Steve, Father)
(Steve , Simon, Son1)
(Melanie, Alison, Mother)
(Lindsey, Fergus, Son1)

...

(Bob, Jane, Mother)
(Natalie, Alice, Mother)
(Mary, Ian, Father)
(Jane, Alice, Daughter1)
(Mat, Charlotte, Mother)

- 54 edges in total

Inference Question:
(Freya, _, MaternalGreatUncle)

Answer:
(Freya, Fergus, MaternalGreatUncle)

Figure 2: **Graph Tasks**. *a*. An example of a random graph used for training. *b*. Zone 1 of the London Underground map, used as a generalisation test for the *path traversal* and *shortest path* tasks. Random seven-step traversals, an example of which is shown in green, were tested yielding an average accuracy of 98.8%. All possible four-step shortest paths were tested giving an average accuracy of 55.3% for networks that had completed the training curriculum; an example is shown in purple. For brevity, only interchange stations were represented in the graph. *c*. The family tree that was used as a generalisation test for *inferred relations*; four-step relations such as the one shown in blue (from *Freya* to *Fergus*, her *maternal great uncle*) were tested, giving an average accuracy of 81.8%. Beneath the graphs the symbol sequences processed by the network during the test examples are shown. The *input* is an unordered list of *(FromNode, ToNode, edge)* triple vectors that describes the graph. For each task, the *question* is a sequence of triples with missing elements (denoted '_') and the *answer* is a sequence of completed triples.

curriculum of even the easiest task (*traversal*), reaching an average of only 37% accuracy after almost 2 million training examples, compared to DNC which reached an average of 98.8% accuracy on the *final* lesson of the same curriculum after around 1 million training examples.

DNC trained on *path traversal* was analysed for its memory usage (Figure 3). It made clear use of content lookup to navigate the London Underground (Figure 3d) and recorded the traversal query using the temporal links (Figure 3a&b). Visualisation of DNC trained on *shortest path* shows that each edge in the graph was written to a different location in memory. To answer the query, DNC appears to have found a heuristic by which it progressively explored the links radiating out from the start and end node until a connecting path was found (Supplementary Video 1).

The graph experiments demonstrate that DNC excels at structured data manipulation. These tasks were learned with an explicit teaching signal indicating the right answer to provide at every time step (supervised learning). In the next task, we tested the ability of DNC to learn from a less direct form of instruction, which only provides reward for successful sequences of output actions. We therefore investigated a form of reinforcement learning in which a sequence of instructions describing a goal is coupled to a reward function that evaluates whether the goal is satisfied. This resembles an animal training protocol with a complex task cue[33].

We built a puzzle game environment inspired by Winograd's SHRDLU[16], a classic AI demonstration of an environment with moveable objects and a hard-coded intelligent agent that interpreted user instructions. Here, we simplified the environment but taught the agent to interpret instructions purely from rewarded interactions. The Mini-SHRDLU environment contained a set

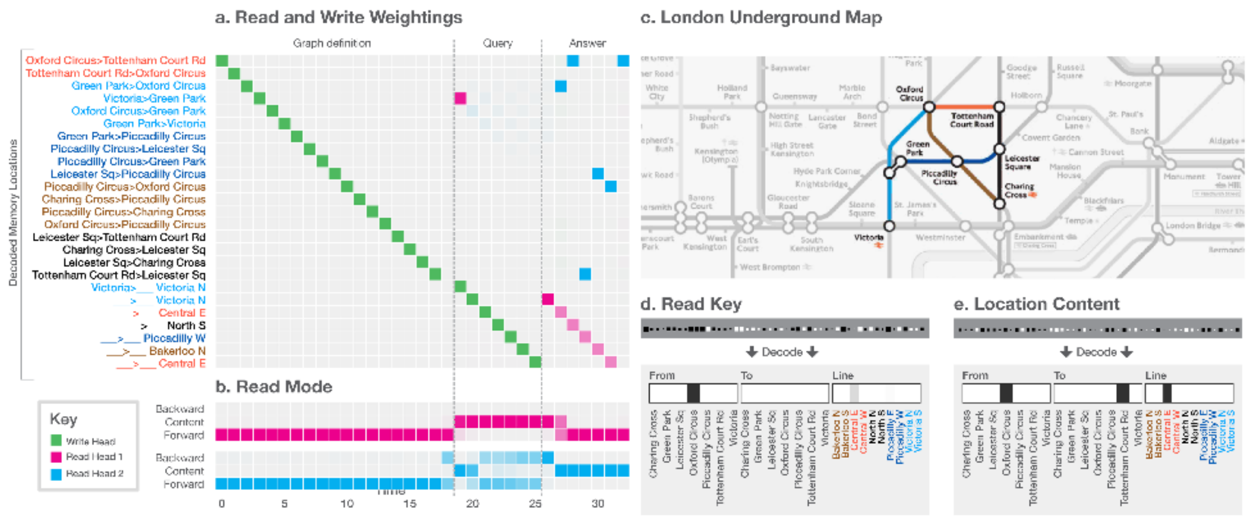


Figure 3: **Path Traversal on the London Underground.** *a*. During the *graph definition* phase each triple in the map is written to a separate memory location, as shown by the write weightings (green). During the *query* phase, the start station (Victoria) and lines to be traversed are recorded. The triple stored in each location can be recovered by a logistic regression decoder, as shown on the vertical axis. *b*. The *read mode* distribution during the *answer* phase reveals that read head 1 (pink) follows *forward* temporal links to retrieve the instructions in order, while read head 2 (blue) uses *content* lookup to find the stations along the path. *c*. The region of the map illustrated in the diagram. *d*. The final content key used by read head 2 is decoded as a triple with no destination. *e*. The memory location returned by the key contains the complete triple, allowing the network to infer the destination (Tottenham Court Rd.).

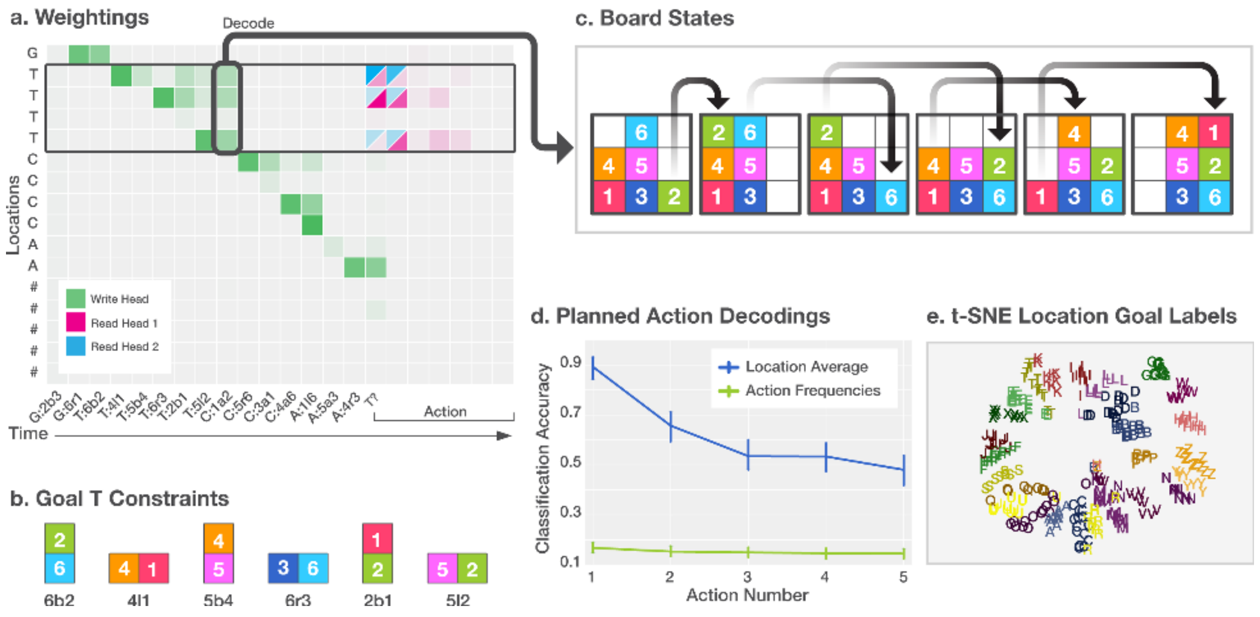


Figure 4: **Mini-SHRDLU Analysis**. *a.* In a short example episode, the network wrote goal-related information to sequences of memory locations. The chosen goal is “T”, and the read heads focused on the locations containing goal T. *b.* The constraints comprising goal T. *c.* The policy made an optimal sequence of moves to satisfy its constraints. *d.* On 800 random episodes, the first five actions the network took for the chosen goal were decoded from memory using logistic regression at the time-step after the goal was written (box in a. with arrow to c.). Decoding accuracy for the first action is 89%, compared to 17% using action frequencies alone, indicating that the network had determined a plan at the time of writing, many steps before execution. Error bars represent 5-95 percentile bootstrapped confidence intervals on validation data. *e.* On the same data, a colourised t-SNE dimensionality reduction of location contents indicates that the goal labels are distinguishable.

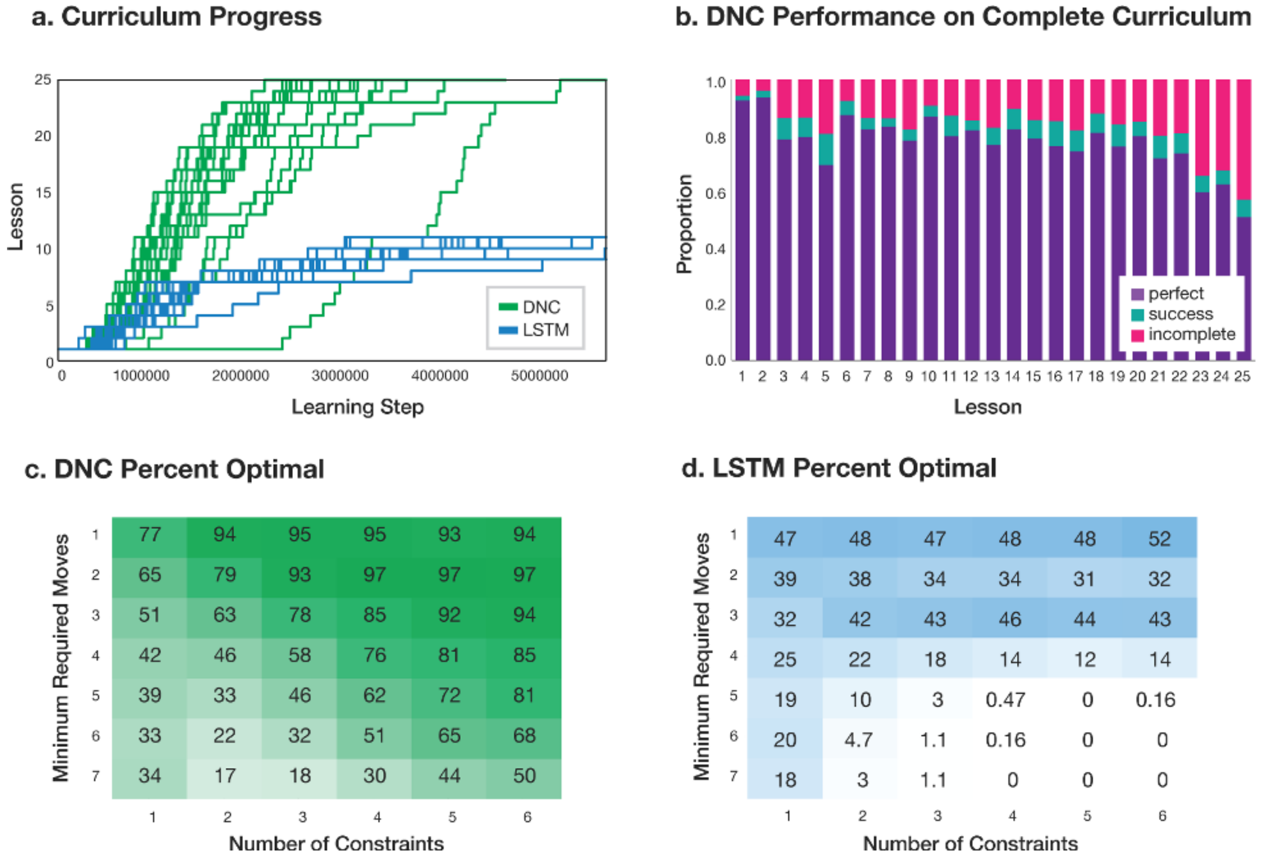


Figure 5: **Mini-SHRDLU Results**. *a.* 20 replicated training runs with different random number seeds for DNC and LSTM. DNC was able to progress through a learning curriculum, while the LSTM networks plateaued. *b.* A single DNC was able to solve a large percentage of problems optimally from each previous lesson (perfect), with few episodes solved in extra moves (success), and some failures to satisfy all constraints (incomplete). *c.* With 10 goals, the same network's performance at satisfying constraints as the minimum number of moves to a goal and the number of constraints in a goal are varied. Performance was highest when the number of constraints was large. *d.* The best LSTM results.

of numbered blocks on a grid board. A DNC, given a view of the board, could move the top block from a column and deposit it on top of a stack in another column. At every episode, we generated a start board configuration and several possible goals. Each goal, identified by a single-letter label, was composed of several individual constraints on adjacent block pairs and was transmitted one constraint per time step (goal A is "block 6 below 2; block 4 left of 1; etc") (Figure 4b-c). After all the goals were presented, a single goal label was chosen at random, and the agent was cued to satisfy that goal.

DNC was able to use its memory to store the instructions, iteratively writing goals to locations (Figure 4a), and then carry out the chosen goal (Figure 4c & Supplementary Video 2). We observed that, at the time a goal was written, but many steps before execution was required, the first action could be predictively decoded from memory, indicating that DNC had written its decision to memory before acting upon it; thus, remarkably, DNC learned to make a plan (4d). Learning followed a curriculum that gradually increases the number of blocks on the board and constraints in a goal as well as the number of goals, and the minimum number of actions needed to find a solution (Methods). Again, DNC performed significantly better than LSTM (Figure 5).

Taken together, the bAbI and graph tasks demonstrate that DNC is able to process and reason about symbolic data regardless of whether the underlying structure is implicit or explicit. Moreover, we have seen that the structure of the data source is directly reflected in the memory-access procedures learned by the controller. The Mini-SHRDLU problem shows that a systematic, structured use of memory also emerges when DNC learns by reinforcement to act in pursuit of a set of symbolic goals.

The theme connecting these tasks is the need to learn to represent and reason about the complex, *quasi-regular* structure embedded in data sequences. In each problem, domain regularities, such as the statistical structure of graphs and conventions for representing them, are invariant across all sequences shown; for any given sequence, DNC must, on the other hand, detect and capture novel variability as episodic variables in memory. This mixture of large-scale structure and more microscopic variability is generic to the problems that confront a cognitive agent[34,35]. For example, in the structure of scenes, stories, and action plans, broad regularities bind together novel variation in any exemplar. Rooms statistically have chairs in them, but the location of a particular chair in a particular room is a variable.

The rapidly writeable external memory of DNC enables the controller neural network to offload variables and exceptions to memory, preserving its capacity for learning and processing regular task structure. Our experiments here have focused on symbolic processing as it is easy to generate tasks synthetically with sufficiently complex structure. We expect that DNCs will bear further fruit as representational engines for one-shot learning[36], scene understanding[37], and cognitive mapping[38], capable of intuiting the structure of the world without neglecting its essential novelty and variability.

1. Krizhevsky, A., Sutskever, I. & Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 1097–1105 (2012).

2. Graves, A. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850* (2013).

3. Sutskever, I., Vinyals, O. & Le, Q. V. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, 3104–3112 (2014).

4. Mnih, V. *et al.* Human-level control through deep reinforcement learning. *Nature* **518**, 529–533 (2015).

5. Gallistel, C. R. & King, A. P. *Memory and the computational brain: Why cognitive science will transform neuroscience*, vol. 6 (John Wiley & Sons, 2011).

6. Marcus, G. F. *The algebraic mind: Integrating connectionism and cognitive science* (MIT press, 2001).

7. Kriete, T., Noelle, D. C., Cohen, J. D. & O'Reilly, R. C. Indirection and symbol-like processing in the prefrontal cortex and basal ganglia. *Proceedings of the National Academy of Sciences* **110**, 16390–16395 (2013).

8. Hinton, G. E. Learning distributed representations of concepts. In *Proceedings of the eighth annual conference of the cognitive science society*, vol. 1, 12 (Amherst, MA, 1986).

9. Bottou, L. From machine learning to machine reasoning. *Machine learning* **94**, 133–149 (2014).

10. Fusi, S., Drew, P. J. & Abbott, L. F. Cascade models of synaptically stored memories. *Neuron* **45**, 599–611 (2005).

11. Ganguli, S., Huh, D. & Sompolinsky, H. Memory traces in dynamical systems. *Proceedings of the National Academy of Sciences* **105**, 18970–18975 (2008).

12. Kanerva, P. *Sparse distributed memory* (MIT press, 1988).

13. Amari, S.-I. Characteristics of sparsely encoded associative memory. *Neural Networks* **2**, 451–457 (1989).

14. Skinner, B. F. *Verbal behavior* (BF Skinner Foundation, 2014).

15. Sutton, R. S. & Barto, A. G. *Reinforcement learning: An introduction*, vol. 1 (MIT press Cambridge, 1998).

16. Winograd, T. Procedures as a representation for data in a computer program for understanding natural language. Tech. Rep., DTIC Document (1971).

17. Weston, J., Chopra, S. & Bordes, A. Memory networks. *arXiv preprint arXiv:1410.3916* (2014).

18. Vinyals, O., Fortunato, M. & Jaitly, N. Pointer networks. *arXiv preprint arXiv:1506.03134* (2015).

19. Graves, A., Wayne, G. & Danihelka, I. Neural Turing Machines. *arXiv preprint arXiv:1410.5401* (2014).

20. Bahdanau, D., Cho, K. & Bengio, Y. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).

21. Gregor, K., Danihelka, I., Graves, A. & Wierstra, D. Draw: A recurrent neural network for image generation. *arXiv preprint arXiv:1502.04623* (2015).

22. Chan, W., Jaitly, N., Le, Q. V. & Vinyals, O. Listen, attend and spell. *arXiv preprint arXiv:1508.01211* (2015).

23. Hintzman, D. L. Minerva 2: A simulation model of human memory. *Behavior Research Methods, Instruments, & Computers* **16**, 96–101 (1984).

24. Kumar, A. *et al.* Ask me anything: Dynamic memory networks for natural language processing. *arXiv preprint arXiv:1506.07285* (2015).

25. Sukhbaatar, S., Weston, J., Fergus, R. *et al.* End-to-end memory networks. In *Advances in Neural Information Processing Systems*, 2431–2439 (2015).

26. Hopfield, J. J. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences* **79**, 2554–2558 (1982).

27. Magee, J. C. & Johnston, D. A synaptically controlled, associative signal for hebbian plasticity in hippocampal neurons. *Science* **275**, 209–213 (1997).

28. Johnston, S. T., Shtrahman, M., Parylak, S., Gonçalves, J. T. & Gage, F. H. Paradox of pattern separation and adult neurogenesis: A dual role for new neurons balancing memory resolution and robustness. *Neurobiology of learning and memory* (2015).

29. O'Reilly, R. C. & McClelland, J. L. Hippocampal conjunctive encoding, storage, and recall: avoiding a trade-off. *Hippocampus* **4**, 661–682 (1994).

30. Howard, M. W. & Kahana, M. J. A distributed representation of temporal context. *Journal of Mathematical Psychology* **46**, 269–299 (2002).

31. Weston, J., Bordes, A., Chopra, S. & Mikolov, T. Towards ai-complete question answering: A set of prerequisite toy tasks. *arXiv preprint arXiv:1502.05698* (2015).

32. Hochreiter, S. & Schmidhuber, J. Long short-term memory. *Neural computation* **9**, 1735–1780 (1997).

33. Epstein, R., Lanza, R. P. & Skinner, B. Symbolic communication between two pigeons (columba livia domestica). *Science* **207**, 543–545 (1980).

34. McClelland, J. L., McNaughton, B. L. & O'Reilly, R. C. Why there are complementary learning systems in the hippocampus and neocortex: insights from the successes and failures of connectionist models of learning and memory. *Psychological review* **102**, 419 (1995).

35. McClelland, J. L. & Goddard, N. H. Considerations arising from a complementary learning systems perspective on hippocampus and neocortex. *Hippocampus* **6**, 654–665 (1996).

36. Rezende, D. J., Mohamed, S., Danihelka, I., Gregor, K. & Wierstra, D. One-shot generalization in deep generative models. *arXiv preprint arXiv:1603.05106* (2016).

37. Oliva, A. & Torralba, A. The role of context in object recognition. *Trends in cognitive sciences* **11**, 520–527 (2007).

38. O'Keefe, J. & Nadel, L. Precis of O'Keefe & Nadel's The hippocampus as a cognitive map. *Behavioral and Brain Sciences* **2**, 487–494 (1979).

Acknowledgements

We thank D. Silver, M. Botvinick, and S. Legg for reviewing the paper; P. Dayan, D. Wierstra, G. Hinton, J. Dean, N. Kalchbrenner, J. Veness, I. Sutskever, V. Mnih, A. Mnih, D. Kumaran, N. de Freitas, L. Sifre, R. Pascanu, T. Lillicrap, J. Rae, A. Senior, M. Denil, T. Kocisky, A. Fidjeland, K. Gregor, A. Lerchner, C. Fernando, D. Rezende, C. Blundell, N. Heess for insightful discussions; the rest of the DeepMind team for support and encouragement.

Methods

1 Controller Network

At every time-step t the controller network receives an input vector $\mathbf{x}_t$ from the dataset or environment and emits an output vector $\mathbf{y}_t$ (Supplementary Figure 1a) that parameterises either a predictive distribution for a target vector $\mathbf{t}_t$ (supervised learning) or an action distribution (reinforcement learning). Additionally, the controller receives from the memory matrix $\mathbf{M} \in \mathbb{R}^{N\times W}$ a set of R *read vectors* $\mathbf{r}^1_{t-1}, \ldots, \mathbf{r}^R_{t-1}$ from the previous time-step and emits an interface vector $\mathbf{z}_t$ (Supplementary Figure 1b) that controls the interactions with the memory (Supplementary Figure 1c). The network inputs are concatenated to yield a single composite vector $\hat{\mathbf{x}}_t = [\mathbf{x}_t, \mathbf{r}^1_{t-1}, \ldots, \mathbf{r}^R_{t-1}]$. Any neural network can be used for the controller, but we have used the following variant of the deep Long Short-Term Memory architecture[1,2]

$$
\begin{aligned}
\mathbf{i}^l_t &= \sigma\left(W^l_{\mathbf{i}}[\hat{\mathbf{x}}_t, \mathbf{h}^l_{t-1}, \mathbf{h}^{l-1}_t] + \mathbf{b}^l_{\mathbf{i}}\right)\\
\mathbf{f}^l_t &= \sigma\left(W^l_{\mathbf{f}}[\hat{\mathbf{x}}_t, \mathbf{h}^l_{t-1}, \mathbf{h}^{l-1}_t] + \mathbf{b}^l_{\mathbf{f}}\right)\\
\mathbf{s}^l_t &= \mathbf{f}^l_t\mathbf{s}^l_{t-1} + \mathbf{i}^l_t \tanh\left(W^l_{\mathbf{s}}[\hat{\mathbf{x}}_t, \mathbf{h}^l_{t-1}, \mathbf{h}^{l-1}_t] + \mathbf{b}^l_{\mathbf{s}}\right)\\
\mathbf{o}^l_t &= \sigma\left(W^l_{\mathbf{o}}[\hat{\mathbf{x}}_t, \mathbf{h}^l_{t-1}, \mathbf{h}^{l-1}_t] + \mathbf{b}^l_{\mathbf{o}}\right)\\
\mathbf{h}^l_t &= \mathbf{o}^l_t \tanh(\mathbf{s}^l_t),
\end{aligned}
$$

where $\sigma(x) = 1/(1+\exp(-x))$ is the logistic sigmoid function, l is the layer index, $\mathbf{h}^l_t$, $\mathbf{i}^l_t$, $\mathbf{f}^l_t$, $\mathbf{s}^l_t$, and $\mathbf{o}^l_t$ are respectively the *hidden*, *input gate*, *forget gates*, *state* and *output gate* activation vectors of layer l at time t. $\mathbf{h}^0_t = \mathbf{0}$ for all t; $\mathbf{h}^l_0$ and $\mathbf{s}^l_0$ are $\mathbf{0}$ for all l. The W terms denote learnable weight matrices (e.g., $W^l_{\mathbf{i}}$ is the matrix of weights going *into* the layer l input gates) and the $\mathbf{b}$ terms are learnable biases. The output vector $\mathbf{y}_t$ and interface vector $\mathbf{z}_t$ are

$$\mathbf{y}_t = W_{\mathbf{y}}[\mathbf{h}_t^1, \ldots, \mathbf{h}_t^L, \mathbf{r}_t^1, \ldots, \mathbf{r}_t^R] \tag{1}$$

$$\mathbf{z}_t = W_{\mathbf{z}}[\mathbf{h}_t^1, \ldots, \mathbf{h}_t^L]. \tag{2}$$

Note that the read vectors $\mathbf{r}_t^r$ are passed to the output vector $\mathbf{y}_t^r$ to allow the DNC to condition its decisions on memory that has just been read; it would not be possible to pass them to the interface vector, and hence to the read and write heads, without creating a cycle in the computation graph.

The interface vector $\mathbf{z}_t$ is divided and processed into the following segments: the R *read keys* $\{\mathbf{k}_t^1, \ldots, \mathbf{k}_t^R \in \mathbb{R}^W\}$, the R *read strengths* $\{\beta_t^1, \ldots, \beta_t^R \in [1, \infty)\}$, the *write key* $\mathbf{k}_t^w$, the *write strength* β_t^w, the *erase vector* $\mathbf{e}_t \in [0,1]^W$, the *write vector* $\mathbf{v}_t \in \mathbb{R}^W$, the R *free gates* $\{f_t^1, \ldots, f_t^R \in [0,1]\}$, the *allocation gates* $g_t^a \in [0,1]$, the *write gates* $g_t^w \in [0,1]$, and the R *read modes* $\{\boldsymbol{\pi}_t^1, \ldots, \boldsymbol{\pi}_t^R \in \mathcal{S}_3\}$, where $\mathcal{S}_N$ is the $N-1$ dimensional unit simplex

$$\mathcal{S}_N = \{\boldsymbol{\alpha} \in \mathbb{R}^N : \boldsymbol{\alpha}_i \in [0,1], \sum_{i=1}^{N} \boldsymbol{\alpha}_i = 1\}. \tag{3}$$

$\boldsymbol{\pi}_t$ is constrained to $\mathcal{S}_3$ using the softmax function, the β_t^r are constrained to $[1, \infty)$ using the function $y = 1 + \log(1 + \exp(x))$, and all terms in the range $[0,1]$ are constrained using the logistic sigmoid function. The use and interpretation of these terms will be explored in the following sections.

2 Overview of Reading and Writing

Selecting locations for reading and writing depends on weightings, which are vectors of non-negative numbers whose elements sum to less than 1. The complete set of allowed weightings over

N locations is the non-negative orthant of $\mathbb{R}^N$ with the unit simplex as a boundary (known as the "corner of the cube"):

$$\Delta_N = \{\boldsymbol{\alpha} \in \mathbb{R}^N : \boldsymbol{\alpha}_i \in [0,1], \sum_{i=1}^{N} \boldsymbol{\alpha}_i \leq 1\}. \tag{4}$$

For the read operation, R *read weightings* $\{\mathbf{w}_t^1, \ldots, \mathbf{w}_t^R \in \Delta_N\}$ are used to compute weighted averages of the contents of the locations:

$$\mathbf{r}_t^r = \mathbf{M}_t^\top \mathbf{w}_t^r. \tag{5}$$

Each read head's *read vector* is appended to the controller input at the next time step. The write operation is mediated by a single *write weighting* $\mathbf{w}_t \in \Delta_N$, which is used to modify the memory content. Reading and writing are further detailed in Section 3.

3 Memory Access

The system employs a combination of *content-based addressing* and *dynamic memory allocation* to determine where to write in memory, and a combination of content-based addressing and *temporal memory linkage* to determine where to read. These mechanisms, all of which are parameterised by the *interface vector* emitted by the controller, $\mathbf{z}_t$, are described in detail below.

Content-based Addressing All content lookup operations on the memory $\mathbf{M} \in \mathbb{R}^{N \times W}$ use the following function

$$\mathcal{C}(\mathbf{M}, \mathbf{k}, \beta)[i] = \frac{\exp\big(\mathcal{D}(\mathbf{k}, \mathbf{M}[i])\beta\big)}{\sum_j \exp\big(\mathcal{D}(\mathbf{k}, \mathbf{M}[j])\beta\big)}, \tag{6}$$

where $\mathbf{k} \in \mathbb{R}^W$ is a lookup key, $\beta \in [1, \infty)$ is a *key strength* scalar, $\mathcal{C}(\mathbf{M}, \mathbf{k}, \beta)$ is a *weighting* over the memory locations, and $\mathcal{D}$ is the cosine similarity.

$$\mathcal{D}(\mathbf{u}, \mathbf{v}) = \frac{\mathbf{u} \cdot \mathbf{v}}{|\mathbf{u}||\mathbf{v}|}. \tag{7}$$

The weighting $\mathcal{C}(\mathbf{M}, \mathbf{k}, \beta) \in \mathcal{S}_N$ defines a normalised probability distribution over the memory locations. In later sections, however, we will encounter weightings in Δ_N that may sum to less than one, with the missing weight implicitly assigned to a *null* operation that does not access any of the locations. Content lookup operations are performed by both the read and write heads.

Dynamic Memory Allocation We approached memory management by developing a differentiable analogue of the *free list* memory allocation scheme[3] (Figure 1d), where a list of available memory locations is maintained by adding to and removing addresses from a linked list. Denote by $\mathbf{u}_t \in [0, 1]^N$ the *memory usage* vector at time t, and define $\mathbf{u}_0 = \mathbf{0}$. Before writing to memory, the controller emits a set of *free gates* f_t^r, one per read head, that determine whether the most recently read locations can be freed. Define $\boldsymbol{\psi}_t \in [0, 1]^N$ as

$$\boldsymbol{\psi}_t = \prod_{r=1}^{R} (\mathbf{1} - f_t^r \mathbf{w}_{t-1}^r). \tag{8}$$

$\boldsymbol{\psi}_t[i]$ is an intermediate variable, computed based on what locations were last attended to by the read heads and the values of the free gates. It represents the degree to which each location in memory is not freed. A location should be considered "used" whenever $\boldsymbol{\psi}_t[i] \approx 1$, and it was either just written to or was already used.

$$\begin{aligned} \mathbf{u}_t &= \big(\mathbf{u}_{t-1} + \mathbf{w}_{t-1} - \mathbf{u}_{t-1}\mathbf{w}_{t-1}\big)\boldsymbol{\psi}_t \\ &= \big(\mathbf{u}_{t-1} + (1 - \mathbf{u}_{t-1})\mathbf{w}_{t-1}\big)\boldsymbol{\psi}_t \end{aligned} \tag{9}$$

where we assume that products between vectors of the same size are taken element-wise. Every write to a location increases its usage, up to a maximum of 1, and usage can only be subsequently decreased (to a minimum of 0) using the free gates. The elements of $\mathbf{u}_t$ are bounded in the range $[0, 1]$. Once $\mathbf{u}_t$ has been determined, the *free list* $\boldsymbol{\phi}_t \in \mathbb{Z}^N$ is defined by sorting the indices of the memory locations in ascending order of usage: $\boldsymbol{\phi}_t[1]$ is therefore the index of the least used location. The *allocation weighting* $\mathbf{a}_t \in \Delta_N$, which is used to provide new locations for writing, is

$$\mathbf{a}_t[\boldsymbol{\phi}_t[j]] = (1 - \mathbf{u}_t[\boldsymbol{\phi}_t[j]]) \prod_{i=1}^{j-1} \mathbf{u}_t[\boldsymbol{\phi}_t[i]]. \tag{10}$$

If all usages are 1, $\mathbf{a}_t = \mathbf{0}$, and the controller can no longer allocate memory without first freeing used locations. The sort operation induces discontinuities at the points where the sort order changes, but we observed no adverse effect due to this because the sort is piecewise differentiable.

Memory Writing The controller can choose to write either to newly allocated locations or to locations addressed by content. First, a write content weighting $\mathbf{c}_t^w \in \mathcal{S}_N$ is constructed

$$\mathbf{c}_t^w = \mathcal{C}(\mathbf{M}_{t-1}, \mathbf{k}_t^w, \beta_t^w). \tag{11}$$

$\mathbf{c}_t^w$ is interpolated with the allocation weighting $\mathbf{a}_t$ defined in Eq. 10 to determine a *write weighting* $\mathbf{w}_t \in \Delta_N$

$$\mathbf{w}_t = g_t^w \big(g_t^a \mathbf{a}_t + \big(1 - g_t^a\big) \mathbf{c}_t^w \big), \tag{12}$$

where $g_t^a \in [0, 1]$ is the *allocation gate* governing interpolation, and $g_t^w \in [0, 1]$ is the *write gate*. If the write gate is 0, then nothing is written, regardless of the other write parameters: it can therefore be used to protect the memory from unnecessary modifications. The controller emits an *erase* *vector* $\mathbf{e}_t \in (0,1)^W$ and a *write vector* $\mathbf{v}_t \in \mathbb{R}^W$, which, along with the write weighting, update the memory to

$$\mathbf{M}_t = \mathbf{M}_{t-1} \circ (\mathbf{E} - \mathbf{w}_t \mathbf{e}_t^\top) + \mathbf{w}_t \mathbf{v}_t{}^\top, \tag{13}$$

where $\circ$ denotes pointwise multiplication and $\mathbf{E}$ is the $N \times W$ matrix of ones.

Temporal Memory Linkage The memory allocation system defined above stores no information about the order in which the memory locations are written to. However, there are many situations where retaining temporal information is useful: for example, when a sequence of instructions must be recorded and retrieved in order. We therefore use a *temporal link matrix* $\mathbf{L}_t \in [0,1]^{N\times N}$ to keep track of consecutively modified memory locations (Figure 1d).

$\mathbf{L}_t[i,j]$ represents the degree to which location i was the location written to after location j, and each row and column of $\mathbf{L}_t$ defines a weighting over locations. That is, $\mathbf{L}_t[i,.] \in \Delta_N$ and $\mathbf{L}_t[.,j] \in \Delta_N$ for all i,j,t. To define $\mathbf{L}_t$, we require a *precedence weighting* $\mathbf{p}_t \in \Delta_N$, where element $\mathbf{p}_t[i]$ represents the degree to which location i was the last one written to. $\mathbf{p}_t$ obeys the recurrence relation

$$\mathbf{p}_0 = \mathbf{0}, \tag{14}$$

$$\mathbf{p}_t = \big(1 - \sum_i \mathbf{w}_t[i]\big)\mathbf{p}_{t-1} + \mathbf{w}_t, \tag{15}$$

where $\mathbf{w}_t$ is the write weighting defined in Eq. 12. Every time a location is modified, the link matrix is updated to remove old links to and from that location. New links from the last written location are then added. We use the following recurrence relation to implement this logic:

$$\mathbf{L}_0 = \mathbf{0}, \tag{16}$$

$$\mathbf{L}_t[i,i] = 0 \ \ \forall i, \tag{17}$$

$$\mathbf{L}_t[i,j] = \big(1 - \mathbf{w}_t[i] - \mathbf{w}_t[j]\big)\mathbf{L}_{t-1}[i,j] + \mathbf{w}_t[i]\mathbf{p}_{t-1}[j]. \tag{18}$$

Note that self links are excluded (the diagonal of the link matrix is always 0). The rows and columns of $\mathbf{L}_t$ represent the weights of the temporal links going into and out from particular memory slots, respectively. Given $\mathbf{L}_t$, the *backward weighting* $\mathbf{b}_t^r \in \Delta_N$ and *forward weighting* $\mathbf{f}_t^r \in \Delta_N$ for read head r are defined as

$$\mathbf{f}_t^r = \mathbf{L}_t \mathbf{w}_{t-1}^r, \tag{19}$$

$$\mathbf{b}_t^r = \mathbf{L}_t^\top \mathbf{w}_{t-1}^r, \tag{20}$$

where $\mathbf{w}_{t-1}^r$ is the read weighting from the previous time-step.

The link matrix is $N \times N$ and therefore requires $O(N^2)$ resources in both memory and computation to calculate exactly. Although tolerable for the experiments in this paper, this cost rapidly becomes prohibitive as the number of locations is increased. Fortunately, the link matrix is typically very sparse and can be approximated with $O(N \log N)$ computation cost and $O(N)$ memory with no discernible loss in performance (Supplementary Figure 2).

For some fixed K, we first calculate sparse vectors $\hat{\mathbf{w}}_t$ and $\hat{\mathbf{p}}_{t-1}$ by sorting $\mathbf{w}_t$ and $\mathbf{p}_{t-1}$ and setting all but the K highest values to 0. This step has $O(N \log N)$ computational cost to account for the sort and $O(N)$ memory cost. We then compute the sparse outer product $\hat{\mathbf{w}}_t \hat{\mathbf{w}}_t^\top$, requiring $O(K^2)$ memory and computation. Assuming the sparse link matrix $\hat{\mathbf{L}}_{t-1}$ from the previous timestep has at most NK non-zero elements, $\hat{\mathbf{L}}_t$ can be updated with $O(NK)$ costs using

$$\hat{\mathbf{L}}_t[i,j] = \big(1 - \hat{\mathbf{w}}_t[i] - \hat{\mathbf{w}}_t[j]\big)\hat{\mathbf{L}}_{t-1}[i,j] + \hat{\mathbf{w}}_t[i]\hat{\mathbf{p}}_{t-1}[j], \tag{21}$$

then setting all elements of $\hat{\mathbf{L}}_t$ less than $1/K$ to zero. Since each row and column of $\hat{\mathbf{L}}_t$ sums to at most one, this operation guarantees that $\hat{\mathbf{L}}_t$ has at most K non-zero elements per row and column, and $\hat{\mathbf{L}}_t$ therefore has at most NK non-zero elements. Finally, the forward and backward weightings can be calculated with $O(NK)$ computation cost and $O(N)$ memory cost as follows:

$$\mathbf{f}_t^r = \hat{\mathbf{L}}_t \hat{\mathbf{w}}_{t-1}^r, \tag{22}$$

$$\mathbf{b}_t^r = \hat{\mathbf{L}}_t^\top \hat{\mathbf{w}}_{t-1}^r. \tag{23}$$

Since K is a constant independent of N (in practice $K = 5$ appears to be sufficient, regardless of memory size), the complete sparse update is $O(N \log N)$ in computation and $O(N)$ in memory.

Memory Reading Each read head r computes a content weighting $\mathbf{c}_t^r \in \Delta_N$ using a read key $\mathbf{k}_t^r \in \mathbb{R}^W$ from the controller

$$\mathbf{c}_t^r = \mathcal{C}(\mathbf{M}_t, \mathbf{k}_t^r, \beta_t^r). \tag{24}$$

Each read head also receives a *read mode* vector $\boldsymbol{\pi}_t^r \in \mathcal{S}_3$, which interpolates among the backward weighting $\mathbf{b}_t^r$, the forward weighting $\mathbf{f}_t^r$, and the content read weighting $\mathbf{c}_t^r$, thereby determining the read weighting $\mathbf{w}_t^r \in \mathcal{S}_3$

$$\mathbf{w}_t^r = \boldsymbol{\pi}_t^r[1]\mathbf{b}_t^r + \boldsymbol{\pi}_t^r[2]\mathbf{c}_t^r + \boldsymbol{\pi}_t^r[3]\mathbf{f}_t^r. \tag{25}$$

If $\boldsymbol{\pi}_t^r[2]$ dominates the read mode, the weighting reverts to content lookup using $\mathbf{k}_t^r$. If $\boldsymbol{\pi}_t^r[3]$ dominates, the read head iterates through memory locations in the order they were written, ignoring the read key. If $\boldsymbol{\pi}_t^r[1]$ dominates, the read head iterates in the reverse order. Finally, the read vector $\mathbf{r}_t^r$ is determined by application of the read weighting to memory, yielding equation 5. The read vectors themselves are concatenated and added to the inputs of the controller network.

A page with all equations is provided at the end of the supplement.

4 Comparison with Neural Turing Machine

The *Neural Turing Machine*[4] (NTM) was the predecessor to the DNC described in this work, differing principally in the access mechanism used to interface with the memory. In the NTM, content-based addressing was combined with *location-based addressing* to allow the network to iterate through memory locations in order of their indices (e.g. location n followed by $n + 1$ etc.). This allowed the network to store and retrieve temporal sequences in contiguous blocks of memory. However, there were several drawbacks. Firstly, NTM has no mechanism to ensure that blocks of allocated memory do not overlap and interfere — a basic problem of computer memory management. Interference is not an issue for the dynamic memory allocation used by DNC, which provides single free locations at a time, irrespective of index, and therefore does not require contiguous blocks. Secondly, NTM has no way of freeing locations that have already been written to, and hence no way of reusing memory when processing long sequences. This problem is addressed in DNC with the *free gates* used for deallocation. Thirdly, sequential information is only preserved as long as NTM continues to iterate through consecutive locations; as soon as the write head jumps to a different part of the memory (using content-based addressing) the order of writes before and after the jump cannot be recovered by the read head. The temporal link matrix in DNC, on the other hand, simply tracks the order in which writes were made.

5 bAbI Task Descriptions

The bAbI dataset[5] comprises a set of 20 synthetic question answering (QA) tasks designed to test different aspects of logical reasoning. As the bAbI data is programmatically generated, and the code is publicly available, multiple versions of the data can be used. To test our models, we used the `en-10k` subset of the data available for download from `http://www.thespermwhale.com/jaseweston/babi/tasks_1-20_v1-2.tar.gz`. For each of the 20 tasks, the data comes partitioned into a training set with 10,000 questions and a test set with 1000 questions[6]. The bAbI tasks are designed around *stories* that may contain more than one question. We treated each story as a separate sequence and presented it to the network in the form of word vectors, one word at a time. After removing all numbers, splitting the remaining text into words, and converting all words to lower case, there were 156 unique words in the lexicon and three punctuation symbols: ".", "?", and "-", the last of which we added to indicate points in the input sequence where outputs were required. Each word was therefore represented as a size 159 one-hot vector, and the network outputs were size 159 softmax distributions. The sentences were separated by full stop characters, and all questions were delimited by a question mark followed by as many dash characters as there were words in the answer to that question. For example, a story from the *Counting and Lists/Sets* task containing five questions was presented as the following input sequence of 108 word tokens:

mary journeyed to the kitchen . mary moved to the bedroom . john went back to the hallway . john picked up the milk there . what is john carrying ? - john travelled to the *garden . john journeyed to the bedroom . what is john carrying ? - mary travelled to the bathroom . john took the apple there . what is john carrying ? - - sandra travelled to the kitchen . john went to the hallway . what is john carrying ? - - john journeyed to the garden . mary went back to the garden . what is john carrying ? - -*

The corresponding answers required at the "-" symbols are as follows:

milk, milk, milk apple, milk apple, milk apple

The network was trained to minimise the cross-entropy of the softmax outputs with respect to the target words; the outputs during time-steps when no target was present were ignored. For each step where a target was present the most probable word in the network's output distribution was selected as its answer. The network was only considered to have correctly replied to a question if it got all the target words correct (for example it had to answer "milk" then "apple" to get the final question in the above story right). Following previous work, we evaluated our networks using the per-task *question error rate* (fraction of incorrectly answered questions).

For each task, we removed approximately 10% of the stories and added them to a validation set. All of the remaining stories were gathered together into a single training set from which a random story was drawn for each training sample. No distinction was drawn between the different tasks during training, and no explicit information was provided to the network to indicate which task the current story was drawn from. We performed a grid search over hyper-parameters for all three architectures, and kept the two settings that returned (1) the lowest average question error rate over the validation set and (2) the single network with lowest validation question error rate. We also used the validation error rate as the early stopping criterion during training (although in practice we did not observe a significant increase in validation error due to overfitting for any of the networks).

Note that using word tokens led to much longer sequences (sometimes more than 500 time-steps longer) than previous work on bAbI, where sentence embeddings were used as input[5,6]. The distinction is significant both in that it places greater stress on the long-range memory capacity of the models, and in that the word level approach is easier to generalise to natural language, which has far greater variability in sentence length and structure than the bAbI data.

For complete results and hyper-parameters on all the bAbI tasks for DNC, NTM and LSTM, see Supplementary Tables 1–3.

6 Graph Task Descriptions

The graph tasks were supervised learning problems, with each training example consisting of an input vector sexquence and corresponding target vector sequence. Each vector encoded a triple consisting of a source label, edge label, and destination label. All labels were represented as numbers between 0 and 999, with each digit represented as a 10-way one-hot encoding. There was also a special *blank* label represented by an all-zero encoding for the three digits, which was used to indicate an unspecified label. Each label required 30 elements, and each triple required 90. The sequences were divided into multiple phases: first a *graph description* phase, then a series of *query* and *answer* phases; in some cases the query and answer were separated by an additional *planning* phase with no input, where the network was given time to compute the answer. During the graph description phase the triples defining the input graph were presented in random order. Target vectors were only present during the answer phases. The input vectors had additional binary channels alongside the triples to indicate when transitions between the different phases occurred, and when a prediction was required of the network (this channel remained active throughout the answer phase). In total the input vectors were size 92 and the target vectors were size 90. The graph networks had 90 output units, corresponding to nine separate softmax distributions over the nine digits. The log-probability of correctly predicting an entire target triple was therefore the sum of the log-probabilities of correctly classifying each of the nine digits. Given input sequence $\mathbf{x}$, network output sequence $\mathbf{y}$, and target sequence $\mathbf{t}$, all of length T, this yields the following cross-entropy loss function:

$$\mathcal{L}(\mathbf{x}, \mathbf{t}) = -\sum_{t=1}^{T} A(t) \sum_{d=1}^{9} \log \Pr(\mathbf{t}_t^d | \mathbf{y}_t^d), \tag{26}$$

where $\mathbf{t}_t^d$ is the target at time t for digit d, $\mathbf{y}_t^d$ is the softmax distribution over digit d returned by the network at time t, and $A(t)$ is an indicator function whose value was one during answer phases (i.e., when predictions were required of the network) and zero otherwise.

The network's predictions were determined by taking the mode of the output distribution, and the network indicated it had completed an answer by outputting a specially reserved termination pattern. For all tasks apart from *shortest path*, performance was evaluated as the fraction of sequences where *all* target vectors were correctly predicted. The metric used for shortest path is described below.

7 Random Graph Generation

For all graph tasks, the graphs used to train the networks were generated by uniformly sampling a set of two-dimensional points from a unit square, where each point corresponded to a node in the graph. For each node, the K nearest neighbours in the square were used as the K outbound connections, where K was independently sampled from a uniform range for each node. The numerical labels for the nodes were chosen uniformly from the range $[0, 999]$. For the *path* problems, the edge labels were unique per outbound node but non-unique across the graph. For a graph with N nodes, N unique numbers in the range $[0, 999]$ were initially drawn. Then the outbound edge labels for each node were chosen at random from those N numbers. The edge labeling procedure for the *relation* problems is described below.

Path Traversal A path on the graph was defined based on a random walk from a random start node. At the query phase, the first input to the network was a partially specified triple (source label, edge label, blank) with the destination unspecified. The input triples for the rest of the query contained edge labels only, with source and destination unspecified. During the answer phase, no inputs were presented and the target output was the sequence of complete triples along the path. To succeed, the network had to infer the destination of each triple, and remember it as the implicit source for the next triple.

Shortest Path In the query phase, a single triple was presented (source, blank, destination), defining the start and end nodes. Each query was followed by a 10 time-step planning phase, allowing the network to perform computations and attempt to determine a shortest path. During the answer phase, the network predicted a sequence of triples corresponding to a path. Unlike the traversal task, it also received input triples during the answer phase, indicating the actions chosen on the previous time step. This makes the problem a "structured prediction" problem, which we explain further in the next section. As described in Section 8, the input triples were sometimes the network's own predictions from the previous time-step, and *during training* were sometimes provided by an optimal planner recalculating a shortest path to the end node. To ensure that the network always moved to a valid node, the output distribution was renormalised over the set of possible triples outgoing from the current node. The network was scored on the fraction of sequences for which the shortest possible path was as long as the path it predicted.

Inferred Relations We define a *relation* to be a concatenation of two or more edge labels that is given a distinct label itself. For this task, numbers from 0 to 9 indicated single edge labels, while numbers from 10 to 410 indicated *relation* labels. The relations were generated as unique sequences of single edges of length 2–5, with 100 distinct sequences for each length. The relation sequences were held constant throughout training, and the same relations were used for all networks trained on the task. During the query phase a partial triple was presented, consisting of a start node and relation label. This was followed by a ten time-step planning phase. The single target vector during the answer phase was the completed triple from the query: (start node, relation label, end node). To solve the problem the network had to recall the relation from its label and perform an implicit traversal during the planning phase to reach the destination. The relations were never passed as input the network, so it had to infer them from error signals alone.

8 Structured Prediction

The shortest path task can be considered a structured prediction problem[7,8], as the output decisions of the network determine the sequence of nodes traversed on the graph, and hence influence the network's future decisions and prediction costs. To bring nomenclature in line with the literature on this topic, we will here refer to the output distribution of the network as a *policy* $\pi(a|s)$ over the *actions* a available to the network in *state* s. The state incorporates both the node currently occupied by the network and the latent state of the network itself. The actions are the outgoing edges from the current node (recall that the output distribution is renormalised over the allowed triples after each move). Following policy π, the induced distribution over states at time-step t is denoted $\rho_t^\pi(s)$. We denote the optimal policy as $\pi^*(a|s)$ with corresponding state distribution $\rho_t^*(s)$. The conventional supervised loss function is

$$J^{\text{sup}}(\pi) = \sum_{t=1}^{T} E_{s_t \sim \rho_t^*(s)} l[\pi^*(\cdot|s_t), \pi(\cdot|s_t)], \tag{27}$$

where $l[\pi^*(\cdot|s_t), \pi(\cdot|s_t)] = -\sum_a \pi^*(a|s_t) \log \pi(a|s_t)$ is the cross-entropy loss. $\pi^*(a|s_t)$ is a delta-function on the action corresponding to the first step along one possible shortest path from the current node to the destination (there may be more than one). If the state distributions $\rho_t^*(s)$ and $\rho_t^\pi(s)$ are dissimilar, minimising the supervised loss function does not necessarily transfer to the true task objective

$$J^{\text{task}}(\pi) = \sum_{t=1}^{T} E_{s_t \sim \rho_t(s)} l[\pi^*(\cdot|s_t), \pi(\cdot|s_t)], \tag{28}$$

where the actions are sampled from the network policy and the states therefore from distribution $\rho_t(s)$. To counteract this problem, we follow a similar approach to the DAGGER algorithm[7], which constructs a mixture policy $\pi^\beta(a|s) = \beta\pi^*(a|s) + (1-\beta)\pi(a|s)$ with parameter β and induced state distribution $\rho_t^{\beta}(s)$. To simplify the implementation, we used a batch size of 1 and trained by taking a stochastic gradient of

$$J^{\beta}(\pi) = \sum_{t=1}^{T} E_{s_t \sim \rho_t^{\beta}(s)} l[\pi^*(\cdot|s_t), \pi(\cdot|s_t)], \tag{29}$$

where the actions are sampled from the network with probability $(1 - \beta)$ and from the optimal policy with probability β. To make transitions driven by the network output, all possible edges connected to a source node are assigned a probability, and the most likely edge is chosen.

9 Reinforcement Learning

Most reinforcement learning (RL) problems are *sequential decision problems*: the environment is in state s, and each action a issued by the agent causes a transition of the environment state based on the environment dynamics. We consider *episodic* problems where the agent acts in the environment for T steps before the environment is reset and a new episode begins. The agent thus acts to create a time series $s_1, a_1, s_2, a_2, s_3, a_3 \ldots, s_T, a_T$. A reward function defining the goal of the problem is given as a function of a state and action: $r(s_t, a_t)$. The goal of the agent is to maximise the total expected reward over an episode. The RL agent architecture presented here contains two DNC networks: a policy network that selects an action and a value network that estimates the expected future reward given the policy network and current state.

The policy specifies a parametric mapping from state observations to a probability distribution over actions: $a \sim \pi(a|s, \theta)$, where θ denotes the policy parameters. The policy's total expected reward over an episode is $J(\pi) = E[\sum_{t=1}^{T} r(s_t, a_t)|\pi]$. In the context of mini-SHRDLU, the policy DNC observes the environment states by receiving an observation sequence one step at a time $o_1, o_2, \ldots, o_t$ and conditions its action on the sequence seen so far: $\pi(a_t|o_1, \ldots, o_t, \theta)$. The value network DNC tries to predict the sum of future rewards for this policy given the current history of observations $V^\pi(o_1, \ldots, o_t, \phi)$, where ϕ comprises its parameters.

The learning algorithm for the value network is conceptually simpler than the policy's. The value network learns by supervised regression to predict the sum of future rewards. After a mini-batch of L episodes, the value network updates its parameters ϕ by gradient descent on the loss function $C(\phi) = \frac{1}{2L} \sum_{l=1}^{L} \sum_{t=1}^{T} \left|\left| \sum_{\tau=t}^{T} r(s_\tau^l, a_\tau^l) - V^\pi(o_1, \ldots, o_\tau, \phi) \right|\right|^2$.

The policy network's action distribution $\pi(a_t|o_1, \ldots, o_t, \theta)$ is a multinomial softmax over a discrete set of actions. We use a *policy gradient* method to optimise the policy parameters[9,10]. After each mini-batch of L episodes, the policy parameter gradient direction to ascend $J(\pi)$ is:

$$\nabla_\theta J(\pi) \approx \frac{1}{L} \sum_{l=1}^{L} \sum_{t=1}^{T} \nabla_\theta \log \pi(a_t^l|o_1^l, \ldots, o_t^l, \theta) \big(E[\sum_{\tau \geq t}^{T} r(s_\tau^l, a_\tau^l)|s_t^l, a_t^l] - E[\sum_{\tau \geq t}^{T} r(s_\tau^l, a_\tau^l)|s_t^l] \big) \tag{30}$$

The quantity $E[\sum_{\tau \geq t}^{T} r(s_\tau^l, a_\tau^l)|s_t^l, a_t^l] - E[\sum_{\tau \geq t}^{T} r(s_\tau^l, a_\tau^l)|s_t^l]$, known as the *advantage*, represents how much the value changes from taking action a_t^l in state s_t^l. Using the value network, we can approximate the advantage using the *temporal difference error*

$$\delta_t^l = r(s_t^l, a_t^l) + V^\pi(o_1^l, \ldots, o_{t+1}^l, \phi) - V^\pi(o_1^l, \ldots, o_t^l, \phi). \tag{31}$$

We use a slight modification of this expression for the advantage[11,12], which we derive more completely in Supplement, to reduce the bias in the value networks. The advantage is estimated using a geometric series of temporal difference errors $\sum_{\tau \geq t} \lambda^{\tau - t} \delta_\tau^l$, where λ is a parameter that controls a bias-variance tradeoff, explained further in Supplement. Finally, the policy gradient estimate is given by

$$\nabla_\theta J(\pi) \approx \frac{1}{L}\sum_{l=1}^{L}\sum_{t=1}^{T}\nabla_\theta \log \pi(a_t^l|o_1^l,\ldots,o_t^l,\theta)\sum_{\tau=t}^{T}\lambda^{\tau-t}\delta_\tau^l. \tag{32}$$

10 Mini-SHRDLU

The Mini-SHRDLU board consists of an $S \times S$ grid of squares, each square empty or filled with a numbered block. Blocks in a column are stacked from the bottom upwards. We report experiments with S = 3 and a maximum of 6 blocks on the board, numbered 1 through 6 uniquely. To generate a problem instance, we first randomly place the blocks on the board so that a block always rests on top of the highest block previously placed in its column. A sequence of G goals is generated, each goal composed of a number of constraints. An example of a single goal is “Goal A: Block 1 is below block 4; block 2 is to the right of block 5; block 3 is above block 4; etc.” Each goal represents a label for a set of constraints on the adjacency relations of the blocks. A goal can be ambiguous in that it doesn’t describe a unique board configuration. For example, “Goal B: Block 1 is left of block 2” allows any configuration of the unstated blocks. Each goal is chosen by constructing a tree search of all configurations of the board that are at minimum D moves away from the starting board, randomly selecting one of these configuration, then sampling a set of constraints on chosen board configuration. Redundant conditions “Block 1 is left of block 2; block 2 is right of block 1” are pruned as are constraints that are already fulfilled by the initial state of the board.

The goals are presented sequentially to the policy network during which time the policy cannot make any moves on the board. Each constraint in the goal is presented in one time-step using a place-coded vector: “(first block; adjacency relation; second block)”. For example,

$(100000; 1000; 010000)$ represents the constraint “Block 1 is above block 2”. In addition, each constraint is labeled with the goal of which it is a part: “(goal name; first block; adjacency relation; second block)”, where we have chosen to let the goals be 1 of 26 possible letters designated by one-hot encodings, i.e., $A = (1, 0, \ldots, 0)$, $Z = (0, 0, \ldots, 1)$, etc. The board is represented as a set of place-coded representations, one for each grid cell. Thus, $(000000; 100000; \ldots)$ designates that the bottom left-hand cell is empty, block 1 is in the bottom center cell, and so on. The network also sees a binary flag that represents a *go cue*. While the go cue is active, a goal is selected from the list of goals that have been shown to the network, and its label is also retransmitted to the network for one time-step, and the network can begin to move the blocks on the board. All told, the policy observes at each time step a vector with features “(goal name; first block; adjacency relation; second block; go cue; board state)”. Up to 10 goals with 6 constraints can be sent to the network before action begins.

Once the go cue arrives, it is possible for the policy network to move a block from one column to another or to pass at each turn. We parameterise these actions using another one-hot encoding so that for a 3×3 board, a move can be made from any column to any other; with the pass move, there are therefore 7 moves. The policy's outputs define the probability of selecting each one of these actions, and a move is sampled at each time-step, which changes the board configuration correspondingly. The policy has a fixed number of moves to make progress on the board until the episode ends. In this setting, we know the minimum number of moves to solve the problem, L. We found that the early stages of learning benefited from giving the policy a number of extra steps to satisfy the instructions, in total allowing $L + \Delta L$ moves with ΔL fixed at 6 in the reported experiments. This parameter did not need to be fine-tuned.

The reward function for the policy equalled the number of constraints in the chosen goal that are currently satisfied minus a small cost for making invalid moves such as picking a block up from a column without any blocks. There was also a penalty for achieving the goal configuration but undoing it. In addition, the policy received a direct error (not a part of the reward function) that promoted higher entropy output distributions.

11 Curriculum Learning

The problems we solve could not be learned in any reasonable amount of time without curriculum training. For each task, we defined a set of task parameters that govern the complexity of the problem. For example, for path traversal, these parameters were the number of nodes in the graph, number of outgoing edges from each node, and the number of steps in the traversal. We built a linear sequence of lessons in which the complexity of the task increases along at least one task parameter with each new lesson. Consistent with the observations of Zaremba and Sutskever[13], we have found in some tasks that performance on earlier lessons degraded if all training exemplars were drawn only from the current lesson's maximum difficulty. We followed their strategy to remedy this effect and drew 10% of exemplars uniformly from earlier lessons on the path traversal, inferred relations, and Mini-SHRDLU problems. (Shortest path lessons effectively include earlier lessons as proper subsets of the final lessons, so this is unnecessary.) After a predefined number of gradient updates, we ran a batch evaluation of the success of the network on the current problem distribution. During evaluation in each episode, on the graph problems, we deterministically sampled the most probable outputs of the network (modal sampling). For the graph problems besides shortest path, if 90% of the episodes were solved optimally, the lesson was completed. (Shortest path lesson completion required 80% of network paths to be no more than one step longer than the shortest path.) In the mini-SHRDLU problem, the lesson was marked complete if 85% of the relevant goal constraints were satisfied on average over the batch. The curricula for the tasks are presented in Supplementary Tables 6-2.

12 Network Analysis

In Figure 3a, the y-axis labels are the input triples provided at each time-step, written beside the location with strongest write magnitude. The locations were re-ordered spatially based on the writing order. Figures 3d and 3e are produced based on the output of a trainer classifier, called the "decoder". A logistic regression classifier was built from a data set of 40K data points in which write vectors were treated as classifier inputs with the input triples from the same time step taken to be the classifier targets, treating source, destination, and edges as independent outputs. The digits of each element were decoded independently, so that there were 9 1-of-10 classifiers used to decode each triple in total. The classifier was trained with an L2 regularisation of the classifier weights with coefficient 0.1. Output classes that were irrelevant to the episode were excluded from the diagram. 3d is produced by applying the classifier to the content lookup key, and 3e is produced by applying the classifier to the contents of the memory location with maximal read weighting.

In Figure 4d, classifiers are built from a data set of 800 Mini-SHRDLU episodes. On each episode, the locations that the read heads assign more than a threshold of 0.01 weighting to at the time of the query are considered relevant to the selected goal. These locations are noted, and their contents at the time when they were last written (determined by the same numerical threshold) are uniformly averaged into a single vector. These vectors therefore encapsulate an average of the locations containing the goal right after the goal has been written to memory but potentially many (up to about 60) time-steps before the first action occurs. The vectors are used as inputs to train the classifier to predict the first 5 actions that occur following the query; i.e., action 1 occurs at $t_{\text{query}} + 0$, action 2 occurs at $t_{\text{query}} + 1$, etc. The classifiers use logistic regression with an L2 regularisation coefficient of 1. The action frequencies baseline predicts each of the action choices based on its frequency at that time-step after the query. Classifier accuracy is determined by constructing 100 80%/20% random splits of the episodes into training and test data. The error bars represent 5-95 percentile accuracy on the test data partitions. In Figure 4e, two-dimensional t-SNE[14] dimensionality reduction is performed on those same averaged vectors. Each data point (only half of which are shown to reduce crowding) is marked with the relevant goal label.

13 Optimisation

For all experiments, results are reported based on statistics gathered from 20 randomly initialised networks sharing the same set of hyper-parameters. These hyper-parameters were derived from prior grid searches. Hyper-parameter settings for each problem are described in Supplementary Tables 5-4. In both the supervised and reinforcement settings we train a model using a single machine version of Downpour SGD[15]. Each CPU in the machine runs a *worker* process with its own copy of the model parameters. The workers generate episodes and compute gradients of the loss function with respect to the network weights using backpropagation through time[16]. The gradients are combined with a single central instance of the RMSProp[17] optimisation algorithm. Once a worker computes gradients for an episode, it acquires a mutual exclusion (mutex) lock guaranteeing that no other process is accessing the optimiser. The gradients are used to perform one optimisation step, modifying the central master copy of the parameters, and the new parameter values are copied back to the worker. The optimiser updates a single global copy of its state, which for RMSProp includes a moving average of gradient magnitudes. Finally the mutex is released, allowing a different worker to perform a gradient update. In the backpropagation-through-time backward pass, the gradient with respect to the LSTM controller activations were clipped element-wise to the range $[-10, 10]$.

1. Hochreiter, S. & Schmidhuber, J. Long short-term memory. *Neural computation* **9**, 1735–1780 (1997).

2. Graves, A., Mohamed, A.-r. & Hinton, G. Speech recognition with deep recurrent neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, 6645–6649 (IEEE, 2013).

3. Wilson, P. R., Johnstone, M. S., Neely, M. & Boles, D. Dynamic storage allocation: A survey and critical review. In *Memory Management*, 1–116 (Springer, 1995).

4. Graves, A., Wayne, G. & Danihelka, I. Neural Turing Machines. *arXiv preprint arXiv:1410.5401* (2014).

5. Weston, J., Bordes, A., Chopra, S. & Mikolov, T. Towards ai-complete question answering: A set of prerequisite toy tasks. *arXiv preprint arXiv:1502.05698* (2015).

6. Sukhbaatar, S., Weston, J., Fergus, R. *et al.* End-to-end memory networks. In *Advances in Neural Information Processing Systems*, 2431–2439 (2015).

7. Ross, S., Gordon, G. J. & Bagnell, J. A. A reduction of imitation learning and structured prediction to no-regret online learning. *arXiv preprint arXiv:1011.0686* (2010).

8. Daumé III, H., Langford, J. & Marcu, D. Search-based structured prediction. *Machine learning* **75**, 297–325 (2009).

9. Williams, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning* **8**, 229–256 (1992).

10. Sutton, R. S., McAllester, D. A., Singh, S. P., Mansour, Y. *et al.* Policy gradient methods for reinforcement learning with function approximation. In *NIPS*, vol. 99, 1057–1063 (1999).

11. Wawrzyński, P. Real-time reinforcement learning by sequential actor–critics and experience replay. *Neural Networks* **22**, 1484–1497 (2009).

12. Schulman, J., Moritz, P., Levine, S., Jordan, M. & Abbeel, P. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438* (2015).

13. Zaremba, W. & Sutskever, I. Learning to execute. *arXiv preprint arXiv:1410.4615* (2014).

14. Van der Maaten, L. & Hinton, G. Visualizing data using t-sne. *Journal of Machine Learning Research* **9**, 85 (2008).

15. Dean, J. *et al.* Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, 1223–1231 (2012).

16. Werbos, P. J. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE* **78**, 1550–1560 (1990).

17. Graves, A. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850* (2013).

Supplementary Information for *Symbolic Reasoning with Differentiable Neural Computers*

1 Reinforcement Learning

We utilise a variant of a policy gradient algorithm to learn to solve Mini-SHRDLU. The traditional formal framework for reinforcement learning is the Markov Decision Process (MDP), which is defined by a set of states $s \in S$, a set of actions $a \in A$, a reward function $r(s, a)$, and state transition probabilities $\Pr(s'|s, a)$, defining how an action influences movement between states.[1] The policy is a parametric distribution of actions conditioned on the state with parameters θ. We also define the time-indexed future state distribution of a policy $\rho_t(s|s_0, \theta)$. The expected value of a policy given an initial time t and state s_t is given as

$$V_t^{\theta}(s_t) = \sum_{\tau=t}^{T} E_{s_\tau \sim \rho_\tau(s|s_t,\theta)} E_{a_\tau \sim \pi(a|s_\tau,\theta)} [r(s_\tau, a_\tau)].$$

The performance difference between two policies can be given exactly as[1]

$$\begin{aligned}
V_t^{\theta'}(s_t) - V_t^{\theta}(s_t) &= \sum_{\tau=t}^{T} E_{s_\tau \sim \rho_\tau(s|s_t,\theta')} E_{a_\tau \sim \pi(a|s_\tau,\theta')} [r(s_\tau, a_\tau)] - V_t^{\theta}(s_t) \\
&= \sum_{\tau=t}^{T} E_{s_\tau \sim \rho_\tau(s|s_t,\theta')} E_{a_\tau \sim \pi(a|s_\tau,\theta')} [r(s_\tau, a_\tau) + V_\tau^{\theta}(s_\tau) - V_\tau^{\theta}(s_\tau)] - V_t^{\theta}(s_t) \\
&= \sum_{\tau=t}^{T} E_{s_\tau \sim \rho_\tau(s|s_t,\theta')} E_{a_\tau \sim \pi(a|s_\tau,\theta')} [r(s_\tau, a_\tau) + V_{\tau+1}^{\theta}(s_{\tau+1}) - V_\tau^{\theta}(s_\tau)].
\end{aligned}$$

[1]We use no discount factor since our tasks are finite length episodes.

We denote the advantage function $A^\theta_\tau(s_\tau, a_\tau) \equiv E_{s_{\tau+1}\sim \Pr(s|s_\tau,a_\tau)}[r(s_\tau, a_\tau) + V^\theta_{\tau+1}(s_{\tau+1}) - V^\theta_\tau(s_\tau)]$, yielding

$$V^{\theta'}_t(s_t) - V^\theta_t(s_t) = \sum_{\tau=t}^{T} E_{s_\tau\sim\rho_\tau(s|s_t,\theta')} E_{a_\tau\sim\pi(a|s_\tau,\theta')} A^\theta_\tau(s_\tau, a_\tau).$$

It is the case that $E_{a_\tau\sim\pi(a|s_\tau,\theta)} A^\theta_\tau(s_\tau, a_\tau) = 0$ (a policy has no advantage relative to itself). Therefore, if we consider the performance difference locally around θ, we have

$$\begin{aligned} V^{\theta+\Delta\theta}_t(s_t) - V^\theta_t(s_t) &= \Delta\theta \cdot \sum_{\tau=t}^{T} \Bigg\{ \nabla_\theta \Big[E_{s_\tau\sim\rho_\tau(s|s_t,\theta)} \Big] E_{a_\tau\sim\pi(a|s_\tau,\theta')} A^\theta_\tau(s_\tau, a_\tau) \\ &\qquad + E_{s_\tau\sim\rho_\tau(s|s_t,\theta)} \nabla_\theta \Big[E_{a_\tau\sim\pi(a|s_\tau,\theta)} \Big] A^\theta_\tau(s_\tau, a_\tau) \Bigg\} + o(|\Delta\theta|^2) \\ &= \Delta\theta \cdot \sum_{\tau=t}^{T} E_{s_\tau\sim\rho_\tau(s|s_t,\theta')} \nabla_\theta \Big[E_{a_\tau\sim\pi(a|s_\tau,\theta)} \Big] A^\theta_\tau(s_\tau, a_\tau) + o(|\Delta\theta|^2), \end{aligned}$$

where the first term disappears because the expected advantage is 0. The second line is commonly transformed using the identity $\nabla_x f(x) = f(x)\nabla_x \log f(x)$ to

$$\nabla_\theta V^\theta_t(s_t) = \sum_{\tau=t}^{T} E_{s_\tau\sim\rho_\tau(s|s_t,\theta')} E_{a_\tau\sim\pi(a|s_\tau,\theta)} \nabla_\theta \log \pi(a_\tau|s_\tau, \theta) A^\theta_\tau(s_\tau, a_\tau).$$

Now, shifting time to an origin at $t = 1$ and taking an expectation over initial states, we can write down the complete policy gradient.

$$\begin{aligned} \nabla_\theta J(\pi) &= E_{s_1\sim\rho_1(s)} \nabla_\theta V^\theta_1(s_1) \\ &= E_{s_1\sim\rho_1(s)} \sum_{t=1}^{T} E_{s_t\sim\rho_t(s|s_1,\theta')} E_{a_t\sim\pi(a|s_t,\theta)} \nabla_\theta \log \pi(a_t|s_t, \theta) A^\theta_t(s_t, a_t). \end{aligned} \tag{33}$$

Since Equation 33 is an expectation, it can be approximated with Monte Carlo samples:

$$\nabla_\theta J(\pi) \approx \frac{1}{L} \sum_{l=1}^{L} \sum_{t=1}^{T} \nabla_\theta \log \pi(a^l_t|s^l_t, \theta) A^\theta_t(s^l_t, a^l_t).$$

This equation does not specify how to estimate the advantage function. We first denote the *temporal difference error* as $\delta_t \equiv r(s_t, a_t) + V_{t+1}^{\theta}(s_{t+1}) - V_t^{\theta}(s_t)$. By the definition, we then have $A_t^{\theta}(s_t, a_t) = E_{a_t \sim \pi(a|s_t,\theta)} \delta_t$. The temporal difference error is an unbiased estimator of the advantage in the case that the value function V^{θ} is known exactly. In practice, this is not usually the case. It can be shown[2,3] that the parametric family of advantage estimators $A_t^{\theta}(s_t, a_t) \approx \sum_{\tau=t}^{T} E_{s_\tau \sim \rho_\tau(s|s_t,a_t,\theta)} \lambda^{\tau-t} \delta_\tau$ with $\lambda \in [0, 1]$ has the property that, even in the case that V^{θ} is approximated, the bias of the estimator vanishes as $\lambda \to 1$, while the variance of the estimator can increase. If V^{θ} is exact, all values of λ yield an unbiased estimator of the advantage. As the bias of our approximator is not directly known, we make a choice for λ based on experimentation. Approximating the advantage function gives the estimator

$$\nabla_{\theta} J(\pi) \approx \frac{1}{L} \sum_{l=1}^{L} \sum_{t=1}^{T} \nabla_{\theta} \log \pi(a_t^l | s_t^l, \theta) \sum_{\tau=t}^{T} \lambda^{\tau-t} \delta_\tau^l.$$

In Mini-SHRDLU, we have partially observed state because the instructions are presented one-by-one and must be memorised, and the number of steps until episode termination is not observable. Thus, we parameterise our policies using recurrent networks (DNC and LSTM) that take in the observations: $\pi(a_t|o_1, \ldots, o_t, \theta) = \pi(a_t|h_t^{\pi}, \theta)$, where $h_t^{\pi} = h^{\pi}(o_t, h_{t-1}^{\pi}, \theta)$ with h_0 part of the parameters to be learned. We represent the value functions as recurrent networks with parameters ϕ that are independent of the policy: $V_t^{\theta}(h_t^v, \phi)$ and $h_t^v = h^v(o_t, h_{t-1}^v, \phi)$. Once we make these replacements, our final policy gradient estimator is

$$\nabla_{\theta} J(\pi) \approx \frac{1}{L} \sum_{l=1}^{L} \sum_{t=1}^{T} \nabla_{\theta} \log \pi(a_t^l | h_t^{\pi,l}, \theta) \sum_{\tau=t}^{T} \lambda^{\tau-t} \delta_\tau^l. \tag{34}$$

We estimate the value function using regression based on the cost function

$$C(\phi) = \frac{1}{2L} \sum_{l=1}^{L} \sum_{t=1}^{T} || \sum_{\tau=t}^{T} r(s_\tau^l, a_\tau^l) - V_t^\theta(h_t^{v,l}, \phi)||^2. \quad (35)$$

2 bAbI Results

To compare with previous results we report the single best network (measured on the validation set) over 20 runs with identical hyper-parameters (Table 1). However, we also report the mean and standard deviation over all runs (Table 2). For all models (LSTM, NTM, DNC) we kept the hyper-parameter settings that (1) gave the lowest average validation error rate or (2) gave the single best validation error rate for a single model. For LSTM and NTM the same setting was best for both criteria, but for DNC two different settings were found (DNC1 for criterion 1, DNC2 for criterion 2).

Task	LSTM (Joint)	NTM (Joint)	DNC1 (Joint)	DNC2 (Joint)	MemN2N (Joint)[4]	MemN2N (Single Task)[4]	DMN (Single Task)[5]
1: 1 supporting fact	24.5	31.5	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
2: 2 supporting facts	53.2	54.5	1.3	0.4	1.0	**0.3**	1.8
3: 3 supporting facts	48.3	43.9	2.4	**1.8**	6.8	2.1	4.8
4: 2 argument relations	0.4	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
5: 3 argument relations	3.5	0.8	**0.5**	0.8	6.1	0.8	0.7
6: yes/no questions	11.5	17.1	**0.0**	**0.0**	0.1	0.1	**0.0**
7: counting	15.0	17.8	**0.2**	0.6	6.6	2.0	3.1
8: lists/sets	16.5	13.8	**0.1**	0.3	2.7	0.9	3.5
9: simple negation	10.5	16.4	**0.0**	0.2	**0.0**	0.3	**0.0**
10: indefinite knowledge	22.9	16.6	0.2	0.2	0.5	**0.0**	**0.0**
11: basic coreference	6.1	15.2	**0.0**	**0.0**	**0.0**	0.1	0.1
12: conjunction	3.8	8.9	0.1	**0.0**	0.1	**0.0**	**0.0**
13: compound coreference	0.5	7.4	**0.0**	0.1	**0.0**	**0.0**	0.2
14: time reasoning	55.3	24.2	0.3	0.4	**0.0**	0.1	**0.0**
15: basic deduction	44.7	47.0	**0.0**	**0.0**	0.2	**0.0**	**0.0**
16: basic induction	52.6	53.6	52.4	55.1	**0.2**	51.8	0.6
17: positional reasoning	39.2	25.5	24.1	**12.0**	41.8	18.6	40.4
18: size reasoning	4.8	2.2	4.0	**0.8**	8.0	5.3	4.7
19: path finding	89.5	4.3	**0.1**	3.9	75.7	2.3	65.5
20: agents motivations	1.3	1.5	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
Mean Error (%)	25.2	20.1	4.3	**3.8**	7.5	4.2	6.4
Failed Tasks (err. $> 5\%$)	15	16	**2**	**2**	6	3	**2**

Table 1: **bAbI Best Results**. Lowest error rate for each task shown in bold.

Task	LSTM	NTM	DNC1	DNC2
1: 1 supporting fact	28.4±1.5	40.6±6.7	**9.0 ± 12.6**	16.2±13.7
2: 2 supporting facts	56.0±1.5	56.3±1.5	**39.2 ± 20.5**	47.5±17.3
3: 3 supporting facts	51.3±1.4	47.8±1.7	**39.6 ± 16.4**	44.3±14.5
4: 2 argument relations	0.8±0.5	0.9±0.7	**0.4 ± 0.7**	**0.4 ± 0.3**
5: 3 argument relations	3.2±0.5	1.9±0.8	**1.5 ± 1.0**	1.9±0.6
6: yes/no questions	15.2±1.5	18.4±1.6	**6.9 ± 7.5**	11.1±7.1
7: counting	16.4±1.4	19.9±2.5	**9.8 ± 7.0**	15.4±7.1
8: lists/sets	17.7±1.2	18.5±4.9	**5.5 ± 5.9**	10.0±6.6
9: simple negation	15.4±1.5	17.9±2.0	**7.7 ± 8.3**	11.7±7.4
10: indefinite knowledge	28.7±1.7	25.7±7.3	**9.6 ± 11.4**	14.7±10.8
11: basic coreference	12.2±3.5	24.4±7.0	**3.3 ± 5.7**	7.2±8.1
12: conjunction	5.4±0.6	21.9±6.6	**5.0 ± 6.3**	10.1±8.1
13: compound coreference	7.2±2.3	8.2±0.8	**3.1 ± 3.6**	5.5±3.4
14: time reasoning	55.9±1.2	44.9±13.0	**11.0 ± 7.5**	15.0±7.4
15: basic deduction	47.0±1.7	46.5±1.6	**27.2 ± 20.1**	40.2±11.1
16: basic induction	**53.3 ± 1.3**	53.8±1.4	53.6±1.9	54.7±1.3
17: positional reasoning	34.8±4.1	**29.9 ± 5.2**	32.4±8.0	30.9±10.1
18: size reasoning	5.0±1.4	4.5±1.3	**4.2 ± 1.8**	4.3±2.1
19: path finding	90.9±1.1	86.5±19.4	**64.6 ± 37.4**	75.8±30.4
20: agents motivations	1.3±0.4	1.4±0.6	**0.0 ± 0.1**	**0.0 ± 0.0**
Mean Error (%)	27.3±0.8	28.5±2.9	**16.7 ± 7.6**	20.8±7.1
Failed Tasks (err. > 5%)	17.1±1.0	17.3±0.7	**11.2 ± 5.4**	14.0±5.0

Table 2: **bAbI Mean Results**. All means over 20 runs ± std. deviation. Lowest mean error rate for each task shown in bold.

	LSTM	NTM	DNC1	DNC2
LSTM Size	512	256	256	256
Batch Size	1	1	1	1
Learning Rate	1×10^{-4}	1×10^{-4}	1×10^{-4}	1×10^{-4}
Memory Dimensions	N/A	256×64	256×64	256×32
Read Heads	N/A	4	4	8
Async. Workers	16	16	16	16

Table 3: **Hyper-parameter settings for bAbI**

	Path Finding	Path Traversal	Inference Tasks
LSTM Size	2×256	3×256	3×256
Batch Size	1	2	32
Learning Rate	3×10^{-6}	1×10^{-5}	1×10^{-5}
Memory Dimensions	128×50	256×50	128×50
Read Heads	5	5	5
DAGGER β	0.8	N/A	N/A

Table 4: **Hyper-parameter settings for graph tasks**

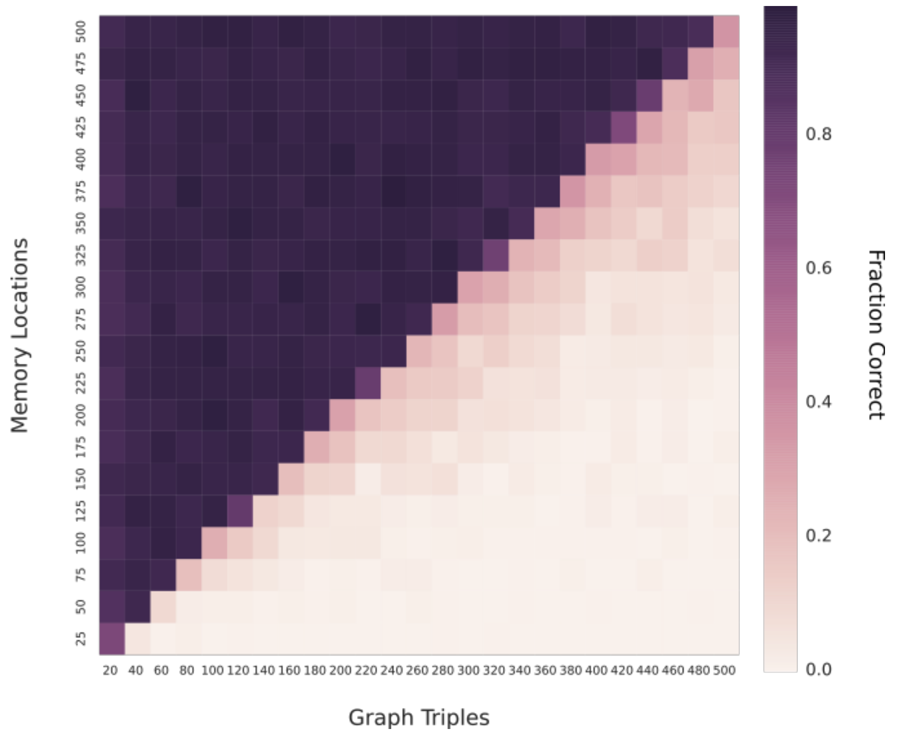


Figure 1: **Altering the Memory Size of a Trained Network**. A DNC trained on the path traversal task with 256 memory locations was tested while varying the number of memory locations and graph triples. The heatmap shows the fraction of length 1–10 traversals performed perfectly by the network, out of a batch of 100. There is a clear correspondence between the number of triples in the graph and the number of memory locations required to solve the task, reflecting our earlier analysis (Figure 3a) that DNC writes each triple to a separate location in memory. The network appears to exploit all available memory, regardless of how much memory it was trained with. This supports our claim that memory is independent of processing in a DCN, and points to large scale applications such as knowledge graph processing.

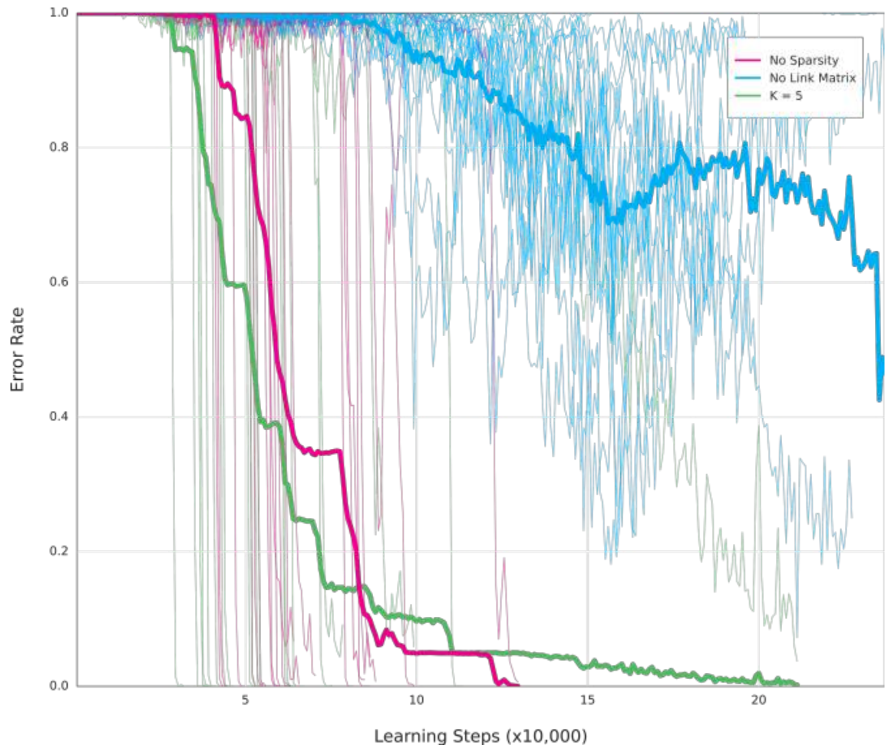


Figure 2: **Impact of Link Matrix Sparsity on Performance**. We trained DNC on a copy problem, where a length 1–100 sequence of size 6 random binary vectors was given as input, and an identical sequence was then required as output. A feedforward controller was used to ensure that the sequences could not be stored in the controller state. The faint lines show error curves for 20 randomly initialised runs with identical hyper-parameters with link matrix sparsity switched off (pink), sparsity used with $K = 5$ (green), and the link matrix disabled altogether (blue). The bold lines show the mean curve for each setting. The error rate is the fraction of sequences copied with no mistakes out of a batch of 100. There does not appear to be any systematic difference between no sparsity and $K = 5$. We observed similar behaviour for K from 2 and 20 (plots omitted for clarity). The task cannot easily be solved without the link matrix as the input sequence has to be recovered in the correct order. Note the abrupt drops in error for the networks with link matrices: these are the points when the system learns a copy algorithm that generalises to longer sequences.

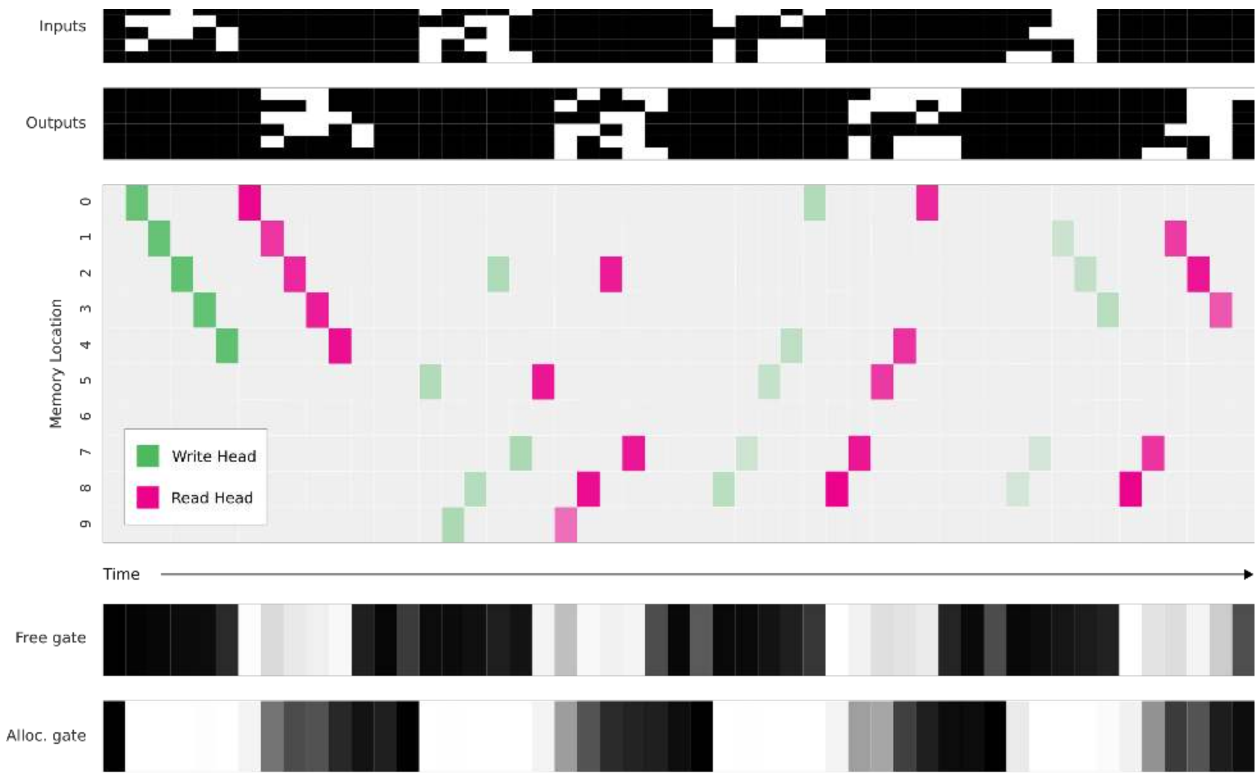


Figure 3: **Dynamic Memory Allocation**. We trained DNC on a copy problem, where a series of 10 random sequences was presented as input. After each input sequence was presented, it was recreated as output. Once the output was generated, that input sequence was not needed again and could be erased from memory. We used a DNC with a feedforward controller and a memory of 10 locations — insufficient to store all 50 input vectors with no overwriting. The goal was to test whether the memory allocation system would be used to free and re-use locations as needed. As shown by the read and write weightings, the same locations are repeatedly used. The *free gate* is active during the read phases, meaning that locations are deallocated immediately after they are read from. The *allocation gate* is active during the write phases, allowing the deallocated locations to be re-used.

	Fig 4 a DNC	Fig 4 a LSTM	Figure 5 DNC
LSTM Size	2×250	2×250	2×250
Batch Size	32	32	32
Learning Rate	3×10^{-5}	3×10^{-5}	3×10^{-5}
Memory Dimensions	32×100	N/A	32×100
Read Heads	3	N/A	2
λ	0.75	0.5	0.5
Entropy Cost Coeff.	0.5	0.5	0.5

Table 5: **Hyper-parameter settings for Mini-SHRDLU**

Lesson	Nodes	Out-degree	Path Length	Test	Final
1	(3, 10)	(2, 4)	(1, 1)	0.0±0.0	10.7±0.6
2	(3, 10)	(2, 4)	(1, 2)	0.0±0.0	19.7±1.3
3	(5, 10)	(2, 4)	(1, 3)	0.0±0.0	27.3±1.0
4	(5, 10)	(2, 4)	(1, 4)	0.0±0.0	38.2±2.2
5	(10, 15)	(2, 4)	(1, 4)	0.0±0.0	39.7±1.5
6	(10, 15)	(2, 4)	(1, 5)	0.0±0.0	47.5±2.2
7	(10, 20)	(2, 4)	(1, 5)	0.1±0.2	48.1±1.8
8	(10, 20)	(2, 4)	(1, 6)	13.6±20.8	59.4±5.2
9	(10, 30)	(2, 4)	(1, 6)	15.0±20.3	59.0±4.8
10	(10, 30)	(2, 4)	(1, 7)	72.8±9.3	72.3±3.5
11	(10, 30)	(2, 4)	(1, 8)	88.6±5.7	81.6±4.0
12	(10, 30)	(2, 4)	(1, 9)	91.8±3.9	90.7±3.2
13	(10, 40)	(2, 6)	(1, 10)	96.0±3.7	96.8±1.6
14	(10, 40)	(2, 6)	(1, 20)	98.8±1.6	99.0±1.1

Table 6: **Curriculum Results for Graph Traversal.** Parentheses represent ranges: (lower bound, upper bound). 'Test' is the accuracy (mean ± std. dev.) on the test set . Evaluation of lesson completion occurs after every group of 100 batches has been processed on the main worker thread. The completion threshold is met if 90% of modal samples (most likely output of network) are correct.

Lesson	Nodes	Out-degree	Relation Length	Queries	Test	Final
1	(3, 3)	(2, 2)	(2, 2)	3	0.0±0.0	2.3±2.3
2	(3, 5)	(2, 2)	(2, 2)	3	0.0±0.0	8.9±4.9
3	(4, 6)	(2, 4)	(2, 2)	3	0.0±0.0	14.3±5.7
4	(4, 6)	(2, 4)	(2, 2)	3	0.0±0.0	15.5±6.2
5	(6, 12)	(2, 4)	(2, 2)	3	0.0±0.0	22.7±3.6
6	(12, 18)	(2, 4)	(2, 2)	3	0.0±0.0	25.8±1.6
7	(4, 6)	(2, 4)	(2, 3)	3	0.1±0.2	31.2±4.0
8	(6, 12)	(2, 4)	(2, 3)	3	0.1±0.2	47.6±5.5
9	(12, 18)	(2, 4)	(2, 3)	3	0.0±0.0	55.1±2.5
10	(4, 6)	(2, 4)	(2, 4)	3	14.5±14.2	49.4±5.0
11	(4, 6)	(2, 4)	(2, 4)	3	11.1±13.1	50.2±5.8
12	(6, 12)	(2, 4)	(2, 4)	3	20.4±13.5	66.3±4.0
13	(12, 18)	(2, 4)	(2, 4)	3	43.6±19.5	78.9±2.9
14	(4, 6)	(2, 4)	(2, 5)	3	18.6±13.8	68.6±5.0
15	(6, 12)	(2, 4)	(2, 5)	3	26.4±19.1	78.4±3.8
16	(12, 18)	(2, 4)	(2, 5)	3	61.9±20.5	91.1±2.3
17	(20, 25)	(2, 4)	(2, 5)	3	81.8±13.5	95.5±2.2

Table 7: **Curriculum Results for Inferred Relations.** Parentheses represent ranges: (lower bound, upper bound). Evaluation of lesson completion occurs after every group of 100 batches has been processed on the main worker thread. The completion threshold is met if 90% of modal samples (most likely output of network) are correct.

Lesson	Nodes	Out-degree	Path Length	Test	Final
1	(5, 10)	(1, 2)	(2, 2)	8.2±6.1	11.2±4.5
2	(5, 20)	(1, 2)	(2, 2)	4.0±2.6	12.7±4.8
3	(10, 20)	(1, 2)	(2, 2)	4.0±3.7	13.2±5.1
4	(10, 20)	(1, 2)	(2, 3)	9.5±5.7	13.7±3.6
5	(10, 20)	(1, 3)	(2, 3)	14.4±5.7	22.8±7.1
6	(10, 20)	(2, 3)	(2, 3)	14.2±6.8	27.6±9.2
7	(10, 20)	(2, 3)	(2, 4)	28.3±9.0	34.0±6.7
8	(10, 20)	(2, 4)	(2, 4)	29.2±9.0	39.7±7.4
9	(10, 25)	(2, 4)	(2, 4)	34.4±11.4	41.1±9.4
10	(10, 25)	(2, 4)	(2, 5)	32.3±10.3	40.0±8.4
11	(10, 25)	(2, 4)	(2, 5)	33.8±9.7	44.0±11.4
12	(15, 25)	(2, 4)	(2, 5)	34.9±7.0	47.2±8.0
13	(15, 25)	(2, 5)	(2, 5)	40.9±6.6	54.3±5.6
14	(20, 25)	(2, 5)	(2, 5)	50.7±2.7	54.0±9.6
15	(20, 25)	(2, 6)	(2, 5)	55.3±6.4	64.0±3.5

Table 8: **Curriculum Results for Shortest Path Task.** Parentheses represent ranges: (lower bound, upper bound). Evaluation of lesson completion occurs after every group of 2000 batches (size 1) has been processed on main worker thread. A path is defined as "correct" if it is a shortest path. The completion threshold is met if 80% of modal samples (most likely output of network) are correct on a new group of 50 episodes.

Lesson	Number of goals	Number of blocks	Number of constraints	Search depth
1	(2, 2)	(3, 3)	(1, 1)	(1, 1)
2	(2, 3)	(3, 4)	(1, 2)	(1, 1)
3	(3, 3)	(4, 4)	(1, 2)	(1, 2)
4	(3, 4)	(4, 5)	(2, 2)	(1, 2)
5	(4, 4)	(5, 5)	(2, 2)	(2, 2)
6	(4, 5)	(5, 6)	(2, 3)	(2, 2)
7	(5, 5)	(6, 6)	(2, 3)	(2, 3)
8	(5, 6)	(6, 6)	(3, 3)	(2, 3)
9	(6, 6)	(6, 6)	(3, 3)	(3, 3)
10	(6, 7)	(6, 6)	(3, 4)	(3, 3)
11	(7, 7)	(6, 6)	(3, 4)	(3, 4)
12	(7, 8)	(6, 6)	(4, 4)	(3, 4)
13	(8, 8)	(6, 6)	(4, 4)	(4, 4)
14	(8, 9)	(6, 6)	(4, 5)	(4, 4)
15	(9, 9)	(6, 6)	(4, 5)	(4, 5)
16	(9, 10)	(6, 6)	(5, 5)	(4, 5)
17	(10, 10)	(6, 6)	(5, 5)	(5, 5)
18	(10, 10)	(6, 6)	(5, 6)	(5, 5)
19	(10, 10)	(6, 6)	(5, 6)	(5, 6)
20	(10, 10)	(6, 6)	(6, 6)	(5, 6)
21	(10, 10)	(6, 6)	(6, 6)	(6, 6)
22	(10, 10)	(6, 6)	(6, 6)	(6, 6)
23	(10, 10)	(6, 6)	(6, 6)	(6, 7)
24	(10, 10)	(6, 6)	(6, 6)	(6, 7)
25	(10, 10)	(6, 6)	(6, 6)	(7, 7)
26	(10, 10)	(6, 6)	(1, 6)	(1, 7)

Table 9: **Curriculum for Mini-SHRDLU.** Parentheses represent ranges: (lower bound, upper bound).Evaluation of lesson completion occurs after every group of 400 batches has been processed on the main worker thread. The completion threshold is met if 85% of constraints are satisfied at episode termination on average over 160 episodes (The last lesson has no termination).

3 Complete DNC Equations

Initial Conditions:

$$\mathbf{u}_0 = \mathbf{0}, \mathbf{p}_0 = \mathbf{0}, \mathbf{L}_0 = \mathbf{0}, \mathbf{L}_t[i,i] = 0 \ \ \forall i$$

Definitions:

$$\mathcal{C}(\mathbf{M}, \mathbf{k}, \beta)[i] = \frac{\exp\big(\mathcal{D}(\mathbf{k}, \mathbf{M}[i])\beta\big)}{\sum_j \exp\big(\mathcal{D}(\mathbf{k}, \mathbf{M}[j])\beta\big)}$$

$$\mathcal{D}(\mathbf{u}, \mathbf{v}) = \frac{\mathbf{u} \cdot \mathbf{v}}{|\mathbf{u}||\mathbf{v}|}$$

Update Equations:

$$\boldsymbol{\psi}_t = \prod_{r=1}^{R} (\mathbf{1} - f_t^r \mathbf{w}_{t-1}^r)$$

$$\mathbf{u}_t = \big(\mathbf{u}_{t-1} + (1 - \mathbf{u}_{t-1})\mathbf{w}_{t-1}\big)\boldsymbol{\psi}_t$$

$$\mathbf{a}_t[\boldsymbol{\phi}_t[j]] = (1 - \mathbf{u}_t[\boldsymbol{\phi}_t[j]]) \prod_{i=1}^{j-1} \mathbf{u}_t[\boldsymbol{\phi}_t[i]]$$

$$\mathbf{c}_t^w = \mathcal{C}(\mathbf{M}_{t-1}, \mathbf{k}_t^w, \beta_t^w)$$

$$\mathbf{w}_t = g_t^w \big(g_t^a \mathbf{a}_t + \big(1 - g_t^a\big)\mathbf{c}_t^w\big)$$

$$\mathbf{M}_t = \mathbf{M}_{t-1} \circ (\mathbf{E} - \mathbf{w}_t \mathbf{e}_t^\top) + \mathbf{w}_t \mathbf{v}_t{}^\top$$

$$\mathbf{p}_t = \big(1 - \sum_i \mathbf{w}_t[i]\big)\mathbf{p}_{t-1} + \mathbf{w}_t$$

$$\mathbf{L}_t[i,j] = \big(1 - \mathbf{w}_t[i] - \mathbf{w}_t[j]\big)\mathbf{L}_{t-1}[i,j] + \mathbf{w}_t[i]\mathbf{p}_{t-1}[j]$$

$$\mathbf{f}_t^r = \mathbf{L}_t \mathbf{w}_{t-1}^r$$

$$\mathbf{b}_t^r = \mathbf{L}_t^\top \mathbf{w}_{t-1}^r$$

$$\mathbf{c}_t^r = \mathcal{C}(\mathbf{M}_t, \mathbf{k}_t^r, \beta_t^r)$$

$$\mathbf{w}_t^r = \boldsymbol{\pi}_t^r[1]\mathbf{b}_t^r + \boldsymbol{\pi}_t^r[2]\mathbf{c}_t^r + \boldsymbol{\pi}_t^r[3]\mathbf{f}_t^r$$

$$\mathbf{r}_t^r = \mathbf{M}_t^\top \mathbf{w}_t^r$$

1. Kakade, S. & Langford, J. Approximately optimal approximate reinforcement learning. In *ICML*, vol. 2, 267–274 (2002).

2. Schulman, J., Moritz, P., Levine, S., Jordan, M. & Abbeel, P. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438* (2015).

3. Wawrzyński, P. Real-time reinforcement learning by sequential actor–critics and experience replay. *Neural Networks* **22**, 1484–1497 (2009).

4. Sukhbaatar, S., Weston, J., Fergus, R. *et al.* End-to-end memory networks. In *Advances in Neural Information Processing Systems*, 2431–2439 (2015).

5. Kumar, A. *et al.* Ask me anything: Dynamic memory networks for natural language processing. *arXiv preprint arXiv:1506.07285* (2015).