practices in oil palm[4,5] and in other monoculture crops, such as coffee, cocoa and rice. The results also help to fill knowledge gaps that will support conservation efforts during the United Nations Decade on Ecosystem Restoration, which began in 2021 (see go.nature.com/3mbj9a8).

Of course, as for many such experiments, this is just the beginning. Trees and palms are long-lived, after all. Before a convincing argument can be made for the widespread adoption of the intervention – and before the approach should be considered for incorporation into certification standards, such as the Roundtable on Sustainable Palm Oil – several questions, including the following, need to be answered. What area and spatial distribution of tree islands would be needed to maximize the positive environmental effects while sustaining oil-palm yields? Should the establishment of such islands be recommended for all industrial palm estates? How can connectivity between habitats be optimized (for example, by using tree islands as 'stepping stones' for species, or by protecting vegetation along rivers and streams)?

What sorts of agroforestry practice might be recommended for the strikingly different contexts of large, homogeneous industrial oil-palm plantations compared with smallholder plots, which tend to be part of mosaic landscapes? Indonesia and Malaysia, which supply more than 80% of global oil-palm demands (see go.nature.com/42hhyjr), are dominated by sprawling industrial-scale estates[2]. However, research in Indonesia shows that, by 2030, the total area of smallholder oil-palm plots is expected to exceed 60% of the total national acreage[6].

Experiments in Brazil[7] indicate that, for smallholders, growing oil palm in a complex agroforestry system can increase oil productivity per palm and enable a more diversified income stream through the sale of other products (such as fruits, timber or cassava); such an approach might also avoid the need for fertilizer or pesticides. Zemp and colleagues suggest that smallholders adopting the use of tree islands would also benefit from improved ecosystem benefits (ecosystem services), lower susceptibility to disturbance and the diversification of risk. However, because it is unlikely that agroforestry practices will gain traction with large industrial producers, the tree-island option (either by retaining some vegetation when clearing the land or establishing tree islands after the oil-palm crop is planted) seems more adapted for adoption in industrial plantations than are agroforestry approaches.

What will happen as the trees continue to grow? On the basis of this study and previous measurements by members of the same team[8], one might expect an eventual reduction in productivity per palm in and around the tree islands as other species grow and compete for resources with oil palms (Fig. 1). How will this affect production on the landscape scale? What will happen after 25 years, when the palms need to be cut and replanted?

Finally, what are the economics of these different models of oil-palm management? That aspect of research is currently missing, because published papers have focused chiefly on demonstrating that agroforestry interventions do not affect productivity at the level of a plot (termed stand level).

The reality is that, despite its growing environmental footprint, oil-palm cultivation is not going away. With an estimated 2022 global market value of US$53 billion (see go.nature.com/41hj7xe), it is a key contributor to national economies and local livelihoods. Despite some progress in developing cultured or synthetic oil, the crop is simply too lucrative[9] for industrial developers and rural communities to pass it up. Agroforestry-based approaches are not a substitute for protecting remaining forests, yet this work by Zemp and colleagues, as well as other studies, demonstrates that the use of tree islands could go a long way to helping restore biodiversity and ecosystem health in oil-palm landscapes.

Robert Nasi is at the Center for International Forestry Research and World Agroforestry (CIFOR-ICRAF), Bogor 16115, Indonesia.
e-mail: r.nasi@cifor-icraf.org

1. Chiriacò, M. V., Bellotta, M., Jusić, J. & Perugini, L. *Environ. Res. Lett.* **17**, 063007 (2022).
2. Descals, A. *et al. Earth Syst. Sci. Data* **13**, 1211–1231 (2021).
3. Zemp, D. C. *et al. Nature* **618**, 316–321 (2023).
4. Khasanah, N. *et al. Front. Sustain. Food Syst.* **3**, 122 (2020).
5. Jezeer, R. & Pasiecznik, N. (eds) *Exploring Inclusive Palm Oil Production* (Tropenbos International, 2019); available at https://go.nature.com/3jvzkam
6. Schoneveld, G. C., Ekowati, D., Andrianto, A. & van der Haar, S. *Environ. Res. Lett.* **14**, 014006 (2019).
7. Miccolis, A., van Noordwijk, M. & Amaral, J. in *Tree Commodities and Resilient Green Economies in Africa* (eds Minang, P. A., Duguma, L. A. & van Noordwijk, M.) Ch. 27 (World Agroforestry, 2021); available at https://go.nature.com/3jfvuq9
8. Gérard, A. *et al. Agric. Ecosyst. Environ.* **240**, 253–260 (2017).
9. Qaim, M., Sibhatu, K. T., Siregar, H. & Grass, I. *Annu. Rev. Resour. Econ.* **12**, 321–344 (2020).

The author declares no competing interests.
This article was published online on 24 May 2023.

Computer science

AI learns to write sorting software on its own

Armando Solar-Lezama

Deep reinforcement learning has been used to improve computer code by treating the task as a game – with no special knowledge needed on the part of the player. The result has already worked its way into countless programs. **See p.257**

For decades, the computing industry relied on Moore's law: as transistors became ever smaller, the number that could be crammed onto a computer chip seemed to double every two years, enabling a similar leap in computing power. But Moore's law has a natural limit, so software optimization has become just as crucial as miniaturization. On page 257, Mankowitz *et al.*[1] reveal a key role for deep learning in this process, by showing that code generated by artificial intelligence (AI) can improve the efficiency with which the C++ programming language sorts items in a list. Although seemingly mundane, this task is needed in computer programs the world over, and the AI version is now baked into a widely used implementation of the C++ library. Perhaps even more remarkably, the AI system can improve the code without any previous knowledge of the problem itself.

To understand the implications of Mankowitz and colleagues' result, it is useful to first understand the way in which programs are translated into instructions that tell a machine how to flip its ones and zeroes. For a programming language such as C++, a program called a compiler takes source code and converts it into a set of 'assembly' instructions that are, in turn, translated into machine-level code.

The conventional approach[2] to improving the performance of a piece of source code was to apply a series of transformations that were guaranteed to preserve the behaviour of the program while changing performance characteristics such as speed and memory usage. Modern compilers can improve performance substantially by applying such optimizations, but the benefits are constrained by the limited set of transformation rules available to them, and by the difficulty of predicting whether a given change will improve performance.

In the late 1990s and early 2000s, there was a push to do better by searching through different sequences of transformations to find the optimal one for a specific machine. This process – known as autotuning – was particularly successful when it could be tailored to a particular type of problem[3,4]. For example, remarkable performance improvements were possible for mathematical operations such as fast Fourier transforms[5] and matrix multiplication[6], as well as for signal processing[7]. In many cases, these approaches surpassed the performance achievable by human developers[8], but building them required tremendous effort. For each new system, many years of research went into the careful design of the way in which computational patterns would be represented in the machine in a problem-specific manner, and the transformation rules that would allow the system to search through different patterns efficiently.

In the late 2000s and early 2010s, a technology emerged that could potentially eliminate the need for tailored transformation rules. The approach was called program synthesis, and it worked by directly searching for sequences of instructions that could be proved to be equivalent to the original code[9]. Such techniques could improve performance either by imposing constraints on the code (such as forcing it to minimize the number of instructions[10]) or by leveraging feedback about the code's performance[11]. The main limitation of this approach, however, was the complexity of searching for long sequences of instructions that achieved a correct result. This complexity meant that these techniques could be applied only to very small code fragments.

Program synthesis got a major boost, however, with the advent of deep learning. Early work showed that a neural network trained on a corpus of programs could guide the search for a program that satisfied a specification[12]. That research eventually led to systems that were able to complete challenging programming tasks when prompted with natural-language descriptions of problems. One such system is AlphaCode (ref. 13), which was developed by members of same team as Mankowitz and colleagues. A limitation of those approaches, however, was that they excelled mostly when solving problems that were similar to those in their training corpus; their applicability was restricted when it came to new programming challenges. Mankowitz and colleagues have now overcome this limitation.

The authors used deep reinforcement learning in an approach that views the program-synthesis problem as a game played by a single player: the program synthesizer, which the authors call AlphaDev. At each step in the game, the synthesizer must choose a move that corresponds to an instruction to add to the program. And with each move, the system responds by running the instruction on the processor, then checking whether the result is correct. If it is, the algorithm assigns a reward based on the program's performance.

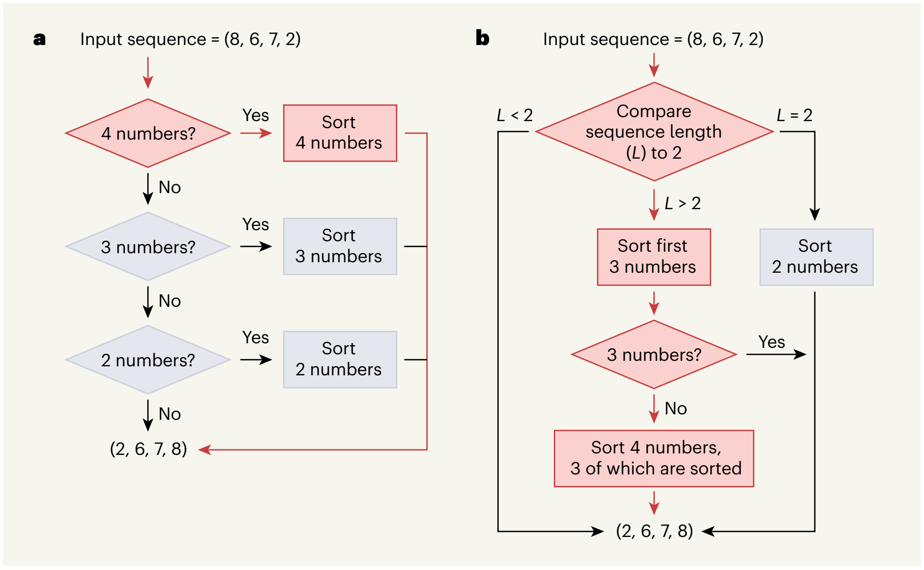

Figure 1 | An AI-generated approach to sorting. Mankowitz *et al.*[1] used deep reinforcement learning to improve the efficiency with which the C++ programming language sorts items in a list. The authors' algorithm used a reward-based system, without the need for any problem-specific training. In one set of experiments, the algorithm focuses on producing routines that sort short sequences of numbers, which are then used as building blocks to sort longer sequences. **a**, This is one approach to sorting a sequence containing up to four numbers, involving three separate routines for sorting two, three or four numbers. Red highlights the path taken to sort the example sequence (8, 6, 7, 2). **b**, Mankowitz and colleagues' algorithm generated a different procedure – incrementally sorting sequences of length four by first sorting the first three elements. This method for sorting is now part of the standard C++ library used worldwide. (Adapted from Fig. 4 of ref. 1.)

The power of this approach comes from the fact that the system can learn to generate efficient programs on the basis of a reward signal, without needing any guidance from training examples. Perhaps surprisingly, it leads to genuine innovation in the approach to tasks as simple and as fundamental as sorting a list of items (Fig. 1).

Previous work showed the promise of deep reinforcement learning for program synthesis[14], but this research generally used simplified problem-specific languages and, in many cases, relied on training data to help the learning process. By contrast, Mankowitz *et al.* devised an approach that needs no such data and that targets an assembly-level language. This is considerably more difficult – and more useful – than previous achievements, because the synthesis process encodes no knowledge about the problem, either in the choice of language or in the training algorithm itself. The authors demonstrate the generality of their approach by using an algorithm built for one type of problem to synthesize programs for two completely different problem types.

The key ingredient ensuring the success of Mankowitz and colleagues' approach is a neural architecture that can capture the current state of the computation and the current sequence of machine-level instructions, and can then make independent predictions about their probable correctness and performance.

The programs that can be optimized by their system are still relatively small. However, the generality of the approach, and its ability to operate without any previous knowledge of the problem, make it a crucial step towards high-performance programming with minimal intervention from experts.

Armando Solar-Lezama is in the Computer Science & Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139, USA.
e-mail: asolar@csail.mit.edu

1. Mankowitz, D. J. *et al. Nature* **618**, 257–263 (2023).
2. Bacon, D. F., Graham, S. L. & Sharp, O. J. *ACM Comput. Surv.* **26**, 345–420 (1994).
3. Balaprakash, P. *et al. Proc. IEEE* **106**, 2068–2083 (2018).
4. Vuduc, R. W. in *Encyclopedia of Parallel Computing* (ed. Padua, D.) 102–105 (Springer, 2011).
5. Frigo, M. *ACM SIGPLAN Not.* **34**, 169–180 (1999).
6. Whaley, R. C., Petitet, A. & Dongarra, J. J. *Parallel Comput.* **27**, 3–35 (2001).
7. Püschel, M. *et al. Proc. IEEE* **93**, 232–275 (2005).
8. Franchetti, F. & Püschel, M. *Proc. Int. Parallel Distributed Process. Symp.* (IEEE, 2003).
9. Alur, R. *et al.* in *Formal Methods in Computer-Aided Design 2013* 1–8 (IEEE, 2013).
10. Barthe, G., Crespo, J. M., Gulwani, S., Kunz, C. & Marron, M. *ACM SIGPLAN Not.* **48**, 123–134 (2013).
11. Schkufza, E., Sharma, R. & Aiken, A. in *Proc. 18th Int. Conf. Archit. Support for Program. Lang. Oper. Syst.* 305–316 (IEEE, 2013).
12. Devlin, J. *et al.* in *Proc. 34th Int. Conf. Mach. Learn.* Vol. 70 (eds Precup, D. & Teh, Y. W.) 990–998 (JMLR, 2017).
13. Li, Y. *et al. Science* **378**, 1092–1097 (2022).
14. Ellis, K. *et al.* in *Proc. 33rd Neural Inf. Process. Syst.* (eds Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F. & Fox, E. B.) 9169–9178 (Curran Associates, 2019).

The author declares competing interests. See go.nature.com/3qmfpha for details.