
Algorithms, Compute, Data: Why Was Compute Easier to Scale?

Odysseus Kaloidas
UC Berkeley

Abstract

Why was expanding computation easier to organize than inventing a better algorithm? Once a procedure worked, its next, larger run could often be planned before its replacement could be discovered. Compute, the processing resources used to train and run models, could pay for more attempts. Better algorithms could make each attempt more useful, while informative data could reveal where to look. Thinking of discovery as search helps explain these contributions without reducing invention to brute force. Relative predictability offers one explanation for compute's prominence. As computation increasingly helps discover better algorithms, expanding a procedure can itself become a way to invent its replacement.

1 The Search Behind Discovery

Why did expanding compute become such a prominent route to AI progress? Once a method produced useful results, running it more extensively could be easier to organize than inventing its replacement. The difficulty of search helps explain that choice. A program must run faster; a candidate explanation must fit the observations; a configuration must meet its performance target. Each achievement can require navigating an immense space of alternatives.

This is the sense in which discovery is about search. The sought object need not be a pre-existing data point: it may be a new program, hypothesis, or representation. Generating it and establishing its significance are parts of the problem. Newell and Simon's classical account connects problem solving to search guided by the structure of a problem space [8].

A larger run can often be described before a better algorithm can: more attempts, more training examples, more time. That does not make the larger run the better choice. It makes it a choice whose cost and likely return may be easier to estimate. The distinction between the most powerful improvement and the most predictable investment is central to compute's appeal.

2 Algorithms, Compute, and Informative Data

An algorithm changes how possibilities are represented, generated, and examined. It can prioritize promising regions, eliminate impossible candidates, or make each evaluation cheaper. Compute supplies the capacity to execute those operations. Data supplies observations from which the procedure can learn useful distinctions, including feedback from earlier attempts.

Training spends compute on data to produce a model; the model may then guide the next search. Suppose earlier attempts reveal a pattern shared by promising candidates. Using that pattern can avoid searching whole regions that no longer look useful. In this way, data can improve the procedure, and the improved procedure can make the same compute budget go further.

Bergstra and Bengio show how even the choice between random and grid search changes the return on a fixed budget for choosing training settings, known as hyperparameters [2]. Jones, Schonlau, and Welch develop an optimization method that uses observations of expensive evaluations to decide

where to sample next [6]. The latter makes data’s contribution especially clear: an observation matters because it changes the next decision, not simply because the dataset has grown.

Sutton’s Bitter Lesson gives the historical argument for methods, especially search and learning, that can exploit expanding computation [10]. Such methods create a practical alternative to waiting for a more efficient conceptual shortcut.

A conceptual advance can dwarf a hardware increase. Newell and Simon illustrate how changing a problem’s representation can replace exhaustive enumeration with a short argument [8]. Hernandez and Brown measure substantial algorithmic efficiency gains at fixed neural-network performance [3]. Compute’s prominence therefore cannot establish algorithmic stagnation.

3 A Simple Search Trade-off

Suppose a procedure proposes candidates, and a reliable check tells us whether each is acceptable. To isolate the tradeoff, hold the procedure and its data fixed and assume attempts are independent. If each attempt costs $c > 0$ and succeeds with probability p , a compute budget C buys n attempts:

$$n = \left\lfloor \frac{C}{c} \right\rfloor, \quad P_{\text{success}} = 1 - (1 - p)^n.$$

One attempt fails with probability $1 - p$. All n fail with probability $(1 - p)^n$, so subtracting from one gives the chance of at least one success. More compute raises n . A better algorithm can lower the cost c or raise the success rate p . Informative data can also raise p by steering proposals toward better candidates.

At success probability 10^{-4} per attempt, 10,000 attempts give approximately a 63.2 percent chance of at least one success. Raising that probability tenfold gives approximately the same chance in 1,000 attempts. A better method or better evidence can therefore replace considerable search, provided the improvement does not impose offsetting costs.

There is no fixed exchange rate between these resources. In adaptive search, each result can change the next proposal; repeated attempts may also make correlated mistakes. Both features violate the simple model. More fundamentally, a procedure that never proposes an acceptable candidate cannot succeed through repetition. The budget here covers searching and checking after preparation; collecting data and training the model cost more.

4 Why Compute Became the Predictable Lever

A team can often specify a larger run before it can specify a better invention. Once a pipeline produces useful results, additional processors specialized for numerical computation and more execution time can extend it without requiring a conceptual discovery for every increment. Power, capital, and networking remain real constraints. Relative predictability, rather than effortless expansion, is the argument.

Language-model scaling laws strengthened this route by documenting regular relationships between prediction error, measured by a training loss, and resources within studied regimes [7]. They supplied evidence that further investment could pay. A loss curve is not itself a probability of scientific discovery; it supports the narrower claim that a productive learning process admitted systematic expansion.

Data offered a second expandable resource where large digital corpora were already available. It was less uniformly expandable because additional examples could be repetitive, irrelevant, or expensive to obtain. Hoffmann and colleagues show that model size and the number of text units used in training, called tokens, require joint allocation under a compute budget [4]. Compute-first describes an investment tendency, not a technical rule that data can always follow later.

Hooker’s Hardware Lottery adds a qualification: the methods that become practical are partly selected by available hardware and software [5]. The supposedly easy resource is easiest for procedures already suited to using it.

The distinction is also becoming recursive. AlphaEvolve combines language-model proposals, evaluation, and evolutionary search to discover improved programs [9]. DeepMind’s May 2026

account reports applications to cache replacement and other infrastructure optimizations [1]. These are developer-reported outcomes, but they illustrate the relationship: computation can help discover algorithms that subsequently make computation more effective.

5 The Limits of Repetition

Search also needs evidence against which to judge its proposals. An unmeasured phenomenon may require an experiment; an unreliable evaluator can reward the wrong candidate. These limits explain why the relative ease of compute expansion varies sharply across discovery settings.

The larger run was attractive because the smaller one had already given us a reason to expect something useful. That is a practical response to the difficulty of invention, not a ranking of compute above algorithms or data. A better representation may remove the need for much of the search; a decisive observation may change what is worth searching for. And when computation helps discover a better algorithm, expanding a procedure becomes one way of inventing its replacement.

References

- [1] AlphaEvolve team. AlphaEvolve: How our Gemini-powered coding agent is scaling impact across fields. Google DeepMind, May 7, 2026. URL <https://deepmind.google/blog/alphaevolve-impact/>. Developer research and application report.
- [2] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13:281–305, 2012. URL <https://jmlr.org/papers/v13/bergstra12a.html>.
- [3] Danny Hernandez and Tom B. Brown. Measuring the Algorithmic Efficiency of Neural Networks. arXiv:2005.04305, 2020. URL <https://arxiv.org/abs/2005.04305>.
- [4] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, et al. Training compute-optimal large language models, 2022. URL <https://arxiv.org/abs/2203.15556>. arXiv:2203.15556.
- [5] Sara Hooker. The hardware lottery, 2020. URL <https://arxiv.org/abs/2009.06489>. arXiv:2009.06489. Historical position essay.
- [6] Donald R. Jones, Matthias Schonlau, and William J. Welch. Efficient global optimization of expensive black-box functions. *Journal of Global Optimization*, 13:455–492, 1998. doi: 10.1023/A:1008306431147.
- [7] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling Laws for Neural Language Models. arXiv:2001.08361, 2020. URL <https://arxiv.org/abs/2001.08361>.
- [8] Allen Newell and Herbert A. Simon. Computer science as empirical inquiry: Symbols and search. *Communications of the ACM*, 19(3):113–126, 1976. doi: 10.1145/360018.360022. URL https://web.cs.wpi.edu/~nth/cs534/papers_worth_reading/Computer_Science_as_Empirical_Inquiry.htm.
- [9] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, et al. AlphaEvolve: A coding agent for scientific and algorithmic discovery, 2025. URL <https://arxiv.org/abs/2506.13131>. arXiv:2506.13131.
- [10] Richard S. Sutton. The bitter lesson. Author essay, March 13, 2019. URL <http://www.incompleteideas.net/IncIdeas/BitterLesson.html>.