ICCM

Hey Pentti, We Did It Again!:
Differentiable vector-symbolic types that prove polynomial termination

Eilene Tomkins-Flanagan*, Connor Hanley*, Mary Kelly*
* Department of Cognitive Science, Carleton University

Carleton University

Outline

Demise of the Power Law

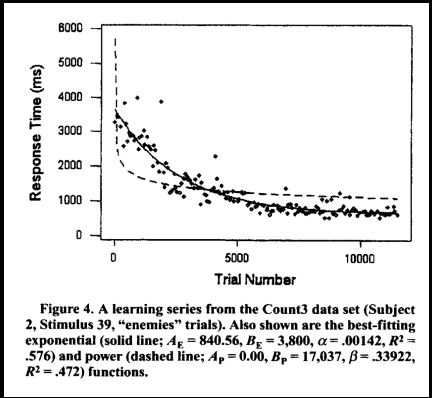
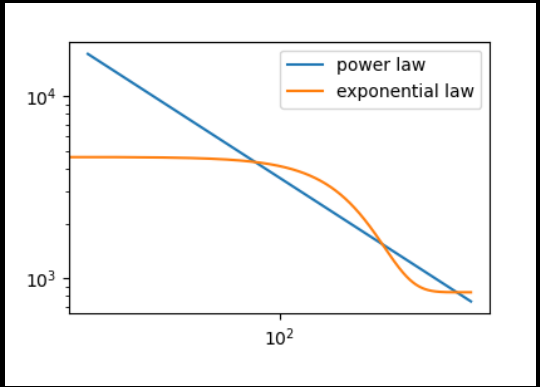
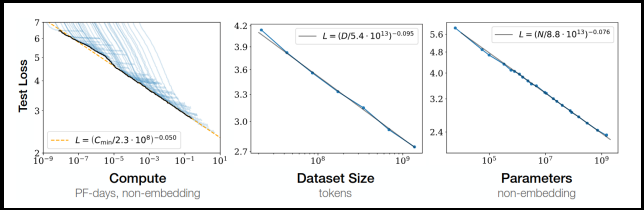


Figure 4. A learning series from the Count3 data set (Subject 2, Stimulus 39, "enemies" trials). Also shown are the best-fitting exponential (solid line; $A_E = 840.56$, $B_E = 3{,}800$, $\alpha = .00142$, $R^2 = .576$) and power (dashed line; $A_P = 0.00$, $B_P = 17{,}037$, $\beta = .33922$, $R^2 = .472$) functions.



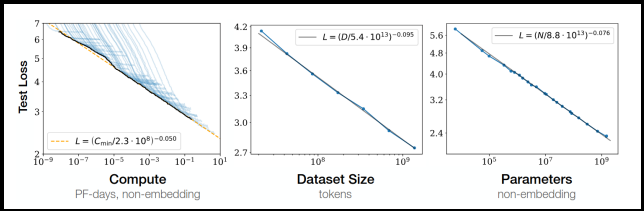
Reproduced from Heathcote et al. (2000)

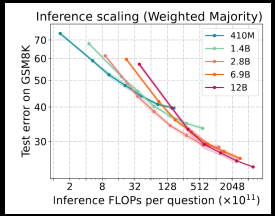
Are Transformers So Great After All?



Reproduced from Kaplan et al. (2020)

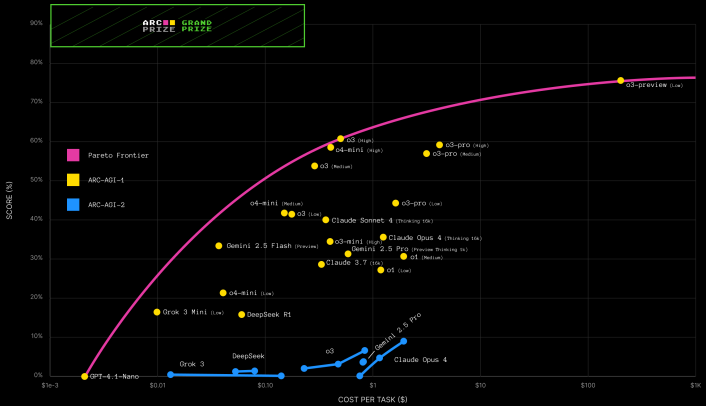
Are Transformers So Great After All?





Reproduced from Kaplan et al. (2020) and Wu et al. (2025)

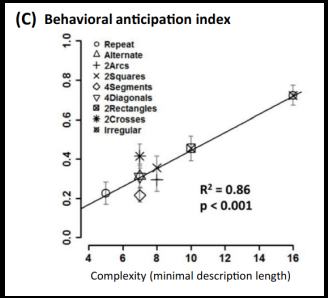
The Wages of GPS



Reproduced from ARC Prize Foundation (2025)

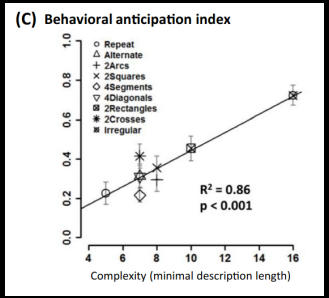
There is an alternative to the Empiricist idea that all learning consists of a kind of statistical inference, realized by adjusting parameters; it's the Rationalist idea that some learning is a kind of theory construction, effected by framing hypotheses and evaluating them against evidence. We seem to remember having been through this argument before. (Fodor & Pylyshyn, 1988)

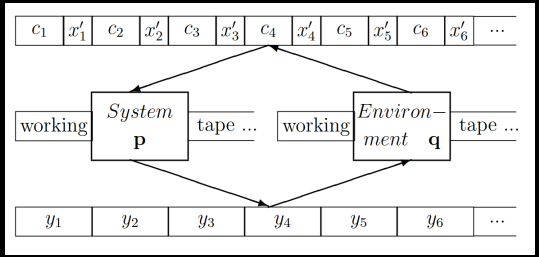
Is Rationalist Learning Possible?



Reproduced from Dehaene et al. (2022)

Is Rationalist Learning Possible?





Reproduced from Dehaene et al. (2022) and Hutter (2000)

Outline

Goals

1. Mental representations need to encode programs

Goals

1. Mental representations need to encode programs
2. We need constraints on program search!

Cutting Edge Technology: Lisp 1.5

- Representations need to have propositional contents *and* be distributed (Fodor & Pylyshyn, 1988; Jackendoff, 2002, ch. 3.5)

Cutting Edge Technology: Lisp 1.5

- Representations need to have propositional contents *and* be distributed (Fodor & Pylyshyn, 1988; Jackendoff, 2002, ch. 3.5)
  - Vector-symbolic architectures (VSAs; Gayler, 2003) satisfy this requirement

Cutting Edge Technology: Lisp 1.5

- Representations need to have propositional contents *and* be distributed (Fodor & Pylyshyn, 1988; Jackendoff, 2002, ch. 3.5)
  - Vector-symbolic architectures (VSAs; Gayler, 2003) satisfy this requirement
- Can we encode arbitrary programs in a VSA?

Cutting Edge Technology: Lisp 1.5

- Representations need to have propositional contents *and* be distributed (Fodor & Pylyshyn, 1988; Jackendoff, 2002, ch. 3.5)
  - Vector-symbolic architectures (VSAs; Gayler, 2003) satisfy this requirement
- Can we encode arbitrary programs in a VSA?
  - Yes! (Tomkins-Flanagan & Kelly, 2024)

Cutting Edge Technology: Lisp 1.5

- Representations need to have propositional contents *and* be distributed (Fodor & Pylyshyn, 1988; Jackendoff, 2002, ch. 3.5)
  - Vector-symbolic architectures (VSAs; Gayler, 2003) satisfy this requirement
- Can we encode arbitrary programs in a VSA?
  - Yes! (Tomkins-Flanagan & Kelly, 2024)
  - TL;DR: if we can get arbitrarily long strings and free substitution, we can compute anything a Turing machine can (Lawvere, 1969)

Cutting Edge Technology: Lisp 1.5

- Representations need to have propositional contents *and* be distributed (Fodor & Pylyshyn, 1988; Jackendoff, 2002, ch. 3.5)
  - Vector-symbolic architectures (VSAs; Gayler, 2003) satisfy this requirement
- Can we encode arbitrary programs in a VSA?
  - Yes! (Tomkins-Flanagan & Kelly, 2024)
  - TL;DR: if we can get arbitrarily long strings and free substitution, we can compute anything a Turing machine can (Lawvere, 1969)
  - To good approximation, the λ-calculus is a subset of Lisp 1.5 (McCarthy, 1960)

To Demonstrate an Arbitrary VSA Can Do This

Definition (Vector-Symbolic Architectures)

A vector-symbolic architecture (VSA) is an algebra

1. that is closed under the product, $\otimes : V \times V \to V$,

To Demonstrate an Arbitrary VSA Can Do This

Definition (Vector-Symbolic Architectures)

A vector-symbolic architecture (VSA) is an algebra

1. that is closed under the product, $\otimes : V \times V \to V$,
2. whose product has an "approximate inverse" of the product, $v \mathbin{\overline{\otimes}} (v \otimes w) \approx w$,

To Demonstrate an Arbitrary VSA Can Do This

Definition (Vector-Symbolic Architectures)

A vector-symbolic architecture (VSA) is an algebra

1. that is closed under the product, $\otimes : V \times V \to V$,
2. whose product has an "approximate inverse" of the product, $v \overline{\otimes} (v \otimes w) \approx w$,
3. for which there is a dogma for selecting atomic "symbols" from the space,

To Demonstrate an Arbitrary VSA Can Do This

Definition (Vector-Symbolic Architectures)

A vector-symbolic architecture (VSA) is an algebra

1. that is closed under the product, $\otimes : V \times V \to V$,
2. whose product has an "approximate inverse" of the product, $v \mathbin{\overline{\otimes}} (v \otimes w) \approx w$,
3. for which there is a dogma for selecting atomic "symbols" from the space,
4. that is paired with a memory system $\mathcal{M}$, and

To Demonstrate an Arbitrary VSA Can Do This

Definition (Vector-Symbolic Architectures)

A vector-symbolic architecture (VSA) is an algebra

1. that is closed under the product, $\otimes : V \times V \to V$,
2. whose product has an "approximate inverse" of the product, $v \mathbin{\overline{\otimes}} (v \otimes w) \approx w$,
3. for which there is a dogma for selecting atomic "symbols" from the space,
4. that is paired with a memory system $\mathcal{M}$, and
5. possesses a measure of the correlation $\mathrm{sim}(u, v) \in [-1, 1]$.

To Demonstrate an Arbitrary VSA Can Do This

Definition (Vector-Symbolic Architectures)

A vector-symbolic architecture (VSA) is an algebra

1. that is closed under the product, $\otimes : V \times V \to V$,
2. whose product has an "approximate inverse" of the product, $v \mathbin{\overline{\otimes}} (v \otimes w) \approx w$,
3. for which there is a dogma for selecting atomic "symbols" from the space,
4. that is paired with a memory system $\mathcal{M}$, and
5. possesses a measure of the correlation $\mathrm{sim}(u, v) \in [-1, 1]$.

Inessential to our VSA definition, but used frequently (and here), are permutations of symbols $\mathbf{P}(v)$

So We Encoded Lisp 1.5

Lists are defined recursively, in terms of the empty list nil.

So We Encoded Lisp 1.5

Lists are defined recursively, in terms of the empty list nil.
For value v and another list l, a list is encoded as:

$$(\mathsf{left} \otimes v) + (\mathsf{right} \otimes l) + \varphi$$

So We Encoded Lisp 1.5

Lists are defined recursively, in terms of the empty list nil.
For value v and another list l, a list is encoded as:

$$(\mathsf{left} \otimes v) + (\mathsf{right} \otimes l) + \varphi$$

This allows for

So We Encoded Lisp 1.5

Lists are defined recursively, in terms of the empty list nil.
For value v and another list l, a list is encoded as:

$$(\mathsf{left} \otimes v) + (\mathsf{right} \otimes l) + \varphi$$

This allows for (1) retrieving the value v,

So We Encoded Lisp 1.5

Lists are defined recursively, in terms of the empty list nil.
For value v and another list l, a list is encoded as:

$$(\mathsf{left} \otimes v) + (\mathsf{right} \otimes l) + \varphi$$

This allows for (1) retrieving the value v, (2) retrieving a pointer to the tail of the list l, and

So We Encoded Lisp 1.5

Lists are defined recursively, in terms of the empty list nil.
For value v and another list l, a list is encoded as:

$$(\text{left} \otimes v) + (\text{right} \otimes l) + \varphi$$

This allows for (1) retrieving the value v, (2) retrieving a pointer to the tail of the list l, and (3) testing whether or not some vector u encodes a list $\text{sim}(u, \varphi) > \theta$

So We Encoded Lisp 1.5

Lists are defined recursively, in terms of the empty list nil.
For value v and another list l, a list is encoded as:

$$(\text{left} \otimes v) + (\text{right} \otimes l) + \varphi$$

This allows for (1) retrieving the value v, (2) retrieving a pointer to the tail of the list l, and (3) testing whether or not some vector u encodes a list $\text{sim}(u, \varphi) > \theta$
In the interpreter, this behaviour is captured by the function:

$$\text{cons}(v, l) := \nu((\text{left} \otimes v) + (\text{right} \otimes l) + \varphi))$$

Where ν denotes normalization.

So We Encoded Lisp 1.5

Lists are defined recursively, in terms of the empty list nil.
For value v and another list l, a list is encoded as:

$$(\mathsf{left} \otimes v) + (\mathsf{right} \otimes l) + \varphi$$

This allows for (1) retrieving the value v, (2) retrieving a pointer to the tail of the list l, and (3) testing whether or not some vector u encodes a list $\mathsf{sim}(u, \varphi) > \theta$
In the interpreter, this behaviour is captured by the function:

$$\mathsf{cons}(v, l) := \nu((\mathsf{left} \otimes v) + (\mathsf{right} \otimes l) + \varphi))$$

Where ν denotes normalization.
We denote $\mathsf{cons}(x, \mathsf{cons}(y, \ldots))$ as $(x\ y\ \ldots)$, omitting the final nil, as is convention.

And Represented Lambda Expressions as Lists

In Lisp, anonymous functions are defined as lists

And Represented Lambda Expressions as Lists

In Lisp, anonymous functions are defined as lists

$$(\texttt{LAMBDA}\ (x\ \ldots)\ e)$$

And Represented Lambda Expressions as Lists

In Lisp, anonymous functions are defined as lists

$$(\text{LAMBDA}\ (x\ \ldots)\ e)$$

Function evaluation is defined using a recursive substitution rule:

And Represented Lambda Expressions as Lists

In Lisp, anonymous functions are defined as lists

$$(\texttt{LAMBDA}\ (x\ \ldots)\ e)$$

Function evaluation is defined using a recursive substitution rule:

$$((\texttt{LAMBDA}\ ()\ e)) \mapsto e \tag{1}$$

$$((\texttt{LAMBDA}\ (x\ y\ \ldots)\ e)\ a) \mapsto (\texttt{LAMBDA}\ (y\ \ldots)\ e[x := a]) \tag{2}$$

$$((\texttt{LAMBDA}\ (x\ y\ \ldots)\ e)\ a\ b\ \ldots) \mapsto (\ldots(((\texttt{LAMBDA}\ (x\ y\ \ldots)\ e)\ a)\ b)\ \ldots) \tag{3}$$

What Kinds of Constraints Do We Need?

1. We need to restrict the class of programs our language can express to tractably do search (recall Hutter, 2005)

What Kinds of Constraints Do We Need?

1. We need to restrict the class of programs our language can express to tractably do search (recall Hutter, 2005)
    - That class should remain as generally expressive as possible

What Kinds of Constraints Do We Need?

1. We need to restrict the class of programs our language can express to tractably do search (recall Hutter, 2005)
   - That class should remain as generally expressive as possible
2. Our expression should allow us to further constrain the search for terms

What Kinds of Constraints Do We Need?

1. We need to restrict the class of programs our language can express to tractably do search (recall Hutter, 2005)
   - That class should remain as generally expressive as possible
2. Our expression should allow us to further constrain the search for terms

In typed languages, types constrain terms!

What Kinds of Constraints Do We Need?

1. We need to restrict the class of programs our language can express to tractably do search (recall Hutter, 2005)
   - That class should remain as generally expressive as possible
2. Our expression should allow us to further constrain the search for terms

In typed languages, types constrain terms!
Let's choose a typed language that *all* and *only* tractable programs

What Kinds of Constraints Do We Need?

1. We need to restrict the class of programs our language can express to tractably do search (recall Hutter, 2005)
   - That class should remain as generally expressive as possible
2. Our expression should allow us to further constrain the search for terms

In typed languages, types constrain terms!
Let's choose a typed language that *all* and *only* tractable programs
A nice language for this is LLFPL (Schimanski, 2009, ch. 7)

What Kinds of Constraints Do We Need?

1. We need to restrict the class of programs our language can express to tractably do search (recall Hutter, 2005)
   - That class should remain as generally expressive as possible
2. Our expression should allow us to further constrain the search for terms

In typed languages, types constrain terms!
Let's choose a typed language that *all* and *only* tractable programs
A nice language for this is LLFPL (Schimanski, 2009, ch. 7) And call our encoding Doug because I'm a sap

Outline

Terms

Terms of Doug may be either constants or composite expressions

Terms

Terms of Doug may be either constants or composite expressions
They are represented using traditional slot-value encodings; some
examples:

Terms

Terms of Doug may be either constants or composite expressions They are represented using traditional slot-value encodings; some examples:

$$\begin{aligned}
\mathsf{tt}(n) &:= \mathbf{TT} + (\mathbf{level} \otimes n) \\
\mathsf{cons}(n, t) &:= \mathbf{Cons} + (\mathbf{level} \otimes n) + (\mathbf{type} \otimes t) \\
\mathsf{nil}(n, t) &:= \mathbf{Nil} + (\mathbf{level} \otimes n) + (\mathbf{type} \otimes t) \\
\mathsf{dollar}(n) &:= \mathbf{Dollar} + (\mathbf{level} \otimes n) \\
\mathsf{app}(t, s) &:= \mathbf{App} + (\mathbf{rator} \otimes t) + (\mathbf{rand} \otimes s) \\
\mathsf{lambda}(x, \tau, t) &:= \mathbf{Lambda} + (\mathbf{var} \otimes x) + (\mathbf{type} \otimes \tau) \\
\mathsf{box}(n, x, s, t) &:= \mathbf{Box} + (\mathbf{let} \otimes x) + (\mathbf{this} \otimes s) + (\mathbf{that} \otimes t) + (\mathbf{level} \otimes n)
\end{aligned}$$

where n is some natural number encoding.

Terms

Terms of Doug may be either constants or composite expressions
They are represented using traditional slot-value encodings; some examples:

$$\begin{aligned}
\mathsf{tt}(n) &:= \mathbf{TT} + (\mathbf{level} \otimes n) \\
\mathsf{cons}(n, t) &:= \mathbf{Cons} + (\mathbf{level} \otimes n) + (\mathbf{type} \otimes t) \\
\mathsf{nil}(n, t) &:= \mathbf{Nil} + (\mathbf{level} \otimes n) + (\mathbf{type} \otimes t) \\
\mathsf{dollar}(n) &:= \mathbf{Dollar} + (\mathbf{level} \otimes n) \\
\mathsf{app}(t, s) &:= \mathbf{App} + (\mathbf{rator} \otimes t) + (\mathbf{rand} \otimes s) \\
\mathsf{lambda}(x, \tau, t) &:= \mathbf{Lambda} + (\mathbf{var} \otimes x) + (\mathbf{type} \otimes \tau) \\
\mathsf{box}(n, x, s, t) &:= \mathbf{Box} + (\mathbf{let} \otimes x) + (\mathbf{this} \otimes s) + (\mathbf{that} \otimes t) + (\mathbf{level} \otimes n)
\end{aligned}$$

where n is some natural number encoding.
Doug (encoding LLFPL) uses the special *dollar* and *box* terms with *levels* to constrain recursion to polynomial runtime

Types

- We want types to be learnable, so you should be able to smoothly interpolate between types that are similar in a meaningful way

Types

- We want types to be learnable, so you should be able to smoothly interpolate between types that are similar in a meaningful way
- We say types are *differentiable* because

Types

- We want types to be learnable, so you should be able to smoothly interpolate between types that are similar in a meaningful way
- We say types are *differentiable* because (1) they live on a continuous embedding space,

Types

- We want types to be learnable, so you should be able to smoothly interpolate between types that are similar in a meaningful way
- We say types are *differentiable* because (1) they live on a continuous embedding space, (2) types are analytically decodable from points on that space,

Types

- We want types to be learnable, so you should be able to smoothly interpolate between types that are similar in a meaningful way
- We say types are *differentiable* because (1) they live on a continuous embedding space, (2) types are analytically decodable from points on that space, (3) a memory system can resolve points near a type to the nearest types, and

Types

- We want types to be learnable, so you should be able to smoothly interpolate between types that are similar in a meaningful way
- We say types are *differentiable* because (1) they live on a continuous embedding space, (2) types are analytically decodable from points on that space, (3) a memory system can resolve points near a type to the nearest types, and (4) spatially nearby types are similar structures comprised of similar elements

Types

- We want types to be learnable, so you should be able to smoothly interpolate between types that are similar in a meaningful way
- We say types are *differentiable* because (1) they live on a continuous embedding space, (2) types are analytically decodable from points on that space, (3) a memory system can resolve points near a type to the nearest types, and (4) spatially nearby types are similar structures comprised of similar elements
- To achieve these properties, we use an encoding based on Holographic Declarative Memory (HDM; Kelly et al., 2020)

Type Encoding

- Let's consider, for example, the function type $\sigma \multimap \tau$, it includes a domain and codomain type, the minimum of their two levels, and a label denoting it as a function

Type Encoding

- Let's consider, for example, the function type $\sigma \multimap \tau$, it includes a domain and codomain type, the minimum of their two levels, and a label denoting it as a function
- We represent the type as a recursive set of key-value pairs

$$\{\text{kind} : \textbf{Map},\ \ \text{type} : \{\text{dom} : d,\ \ \text{codom} : c\},\ \ \text{level} : n\}$$

Type Encoding

- Let's consider, for example, the function type $\sigma \multimap \tau$, it includes a domain and codomain type, the minimum of their two levels, and a label denoting it as a function
- We represent the type as a recursive set of key-value pairs

$$\{\text{kind} : \textbf{Map},\ \ \text{type} : \{\text{dom} : d,\ \ \text{codom} : c\},\ \ \text{level} : n\}$$

- To encode a type T represented as a set, we take the powerset $\mathcal{P}(T)$

Type Encoding

- Let's consider, for example, the function type $\sigma \multimap \tau$, it includes a domain and codomain type, the minimum of their two levels, and a label denoting it as a function
- We represent the type as a recursive set of key-value pairs

$$\{\text{kind} : \mathbf{Map},\ \ \text{type} : \{\text{dom} : d,\ \ \text{codom} : c\},\ \ \text{level} : n\}$$

- To encode a type T represented as a set, we take the powerset $\mathcal{P}(T)$
- Then, we generate the encoding recursively, where $\mathbf{P}$ is a permutation,

$$\mathbf{c}(T) := \sum_{c \in \mathcal{P}(T)} \bigotimes_{\text{slot}:v \in c} \mathbf{P}_{\text{slot}}(v)$$

Type Encoding

- Let's consider, for example, the function type $\sigma \multimap \tau$, it includes a domain and codomain type, the minimum of their two levels, and a label denoting it as a function
- We represent the type as a recursive set of key-value pairs

$$\{\text{kind} : \mathbf{Map},\ \ \text{type} : \{\text{dom} : d,\ \ \text{codom} : c\},\ \ \text{level} : n\}$$

- To encode a type T represented as a set, we take the powerset $\mathcal{P}(T)$
- Then, we generate the encoding recursively, where $\mathbf{P}$ is a permutation,

$$\mathbf{c}(T) := \sum_{c \in \mathcal{P}(T)} \bigotimes_{\text{slot}:v \in c} \mathbf{P}_{\text{slot}}(v)$$

- As a result, types will be similar if their set encodings share any subsets (or different subsets with similar encodings)

Outline

Future Work

- Identifying types for which a divide-and-conquer strategy is provably viable in constrained search

Future Work

- Identifying types for which a divide-and-conquer strategy is provably viable in constrained search
- Proving good constraints can be efficiently learned with sufficient information

Future Work

- Identifying types for which a divide-and-conquer strategy is provably viable in constrained search
- Proving good constraints can be efficiently learned with sufficient information
- Designing neural networks that efficiently learn constraints over the type space

Outline

References I

- ARC Prize Foundation. (2025). Arcpize.org [Accessed: 2025-07-27].
- Dehaene, S., Al Roumi, F., Lakretz, Y., Planton, S., & Sablé-Meyer, M. (2022).Symbols and mental programs: A hypothesis about human singularity. *Trends in Cognitive Sciences*, *26*(9), 751–766. https://doi.org/10.1016/j.tics.2022.06.010
- Fodor, J. A., & Pylyshyn, Z. W. (1988).Connectionism and cognitive architecture: A critical analysis. *Cognition*, *28*(1), 3–71. https://doi.org/10.1016/0010-0277(88)90031-5
- Gayler, R. (2003).Vector symbolic architectures answer jackendoff's challenges for cognitive neuroscience. *Proceedings of the ICCS 2003*.

References II

- Heathcote, A., Brown, S., & Mewhort, D. J. K. (2000).The power law repealed: The case for an exponential law of practice. *Psychonomic Bulletin & Review*, *7*(2), 185–207. https://doi.org/10.3758/BF03212979
- Hutter, M. (2000). Towards a universal theory of artificial intelligence based on algorithmic probability and sequential decision theory.
- Hutter, M. (2005). *Universal artificial intelligence: Sequential decisions based on algorithmic probability*. Springer.
- Jackendoff, R. (2002). *Foundations of language : Brain, meaning, grammar, evolution*. Oxford University Press.
- Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., & Amodei, D. (2020). Scaling laws for neural language models. https://arxiv.org/abs/2001.08361

References III

- Kelly, M. A., Arora, N., West, R. L., & Reitter, D. (2020). Holographic declarative memory: Distributional semantics as the architecture of memory. *Cognitive Science*, *44*(11), e12904. https://doi.org/10.1111/cogs.12904
- Lawvere, F. W. (1969). Diagonal arguments and cartesian closed categories. *Category Theory, Homology Theory and their Applications II*, 134–145.
- McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, *3*(4), 184–195. https://doi.org/10.1145/367177.367199
- Schimanski, S. (2009). *Polynomial time calculi* [Doctoral dissertation, Ludwig Maximilian University of Munich].

References IV

- Tomkins-Flanagan, E., & Kelly, M. A. (2024).Hey Pentti, We Did It!: A Fully Vector-Symbolic Lisp (C. Sibert, Ed.). *Proceedings of the 22nd International Conference on Cognitive Modeling*, 65–72. https://github.com/eilene-ftf/holis
- Wu, Y., Sun, Z., Li, S., Welleck, S., & Yang, Y. (2025). Inference scaling laws: An empirical analysis of compute-optimal inference for problem-solving with language models. https://arxiv.org/abs/2408.00724

https://carleton.ca/animus


NSERC
CRSNG

Viewers Like You

Thank You

https://github.com/eilene-ftf/doug