CMPUT 654: Mathematical Foundations of Modern AI Systems Fall 2026

Lecture 14: Neural networks as computational models

October 20, 2026

Lecturer: Csaba Szepesvári *Note prepared by: Csaba Szepesvári*

Reading guide. Lecture 13 asked what a learned representation preserves for later tasks. This lecture asks whether neural networks can represent symbolic computation. With repeated application and an external memory, the answer is yes. The Church–Turing thesis reduces the question to simulating a Turing machine, and a Turing machine separates into a finite controller and an unbounded tape. A neural network therefore needs to implement only a finite transition table. Bit encodings turn the table into Boolean functions, and Boolean circuits compile exactly into ReLU networks.

The remaining topics study how computational resources and the location of a map affect how the map is represented. A repeated tent-map construction shows that depth can make an exact representation exponentially smaller. Associative and iterated lookup move the map from the network parameters into the input and ask how a sequence model can apply and compose the supplied map.

The material presented in class runs through Section 14.3. The starred Boolean-depth section is supplementary. Section 14.4 and all later sections are extra reading; they separate program correctness from learning, record arithmetic assumptions, and give a conditional lower bound. The main arguments require only elementary finite mathematics, algebra, and induction.

The exact constructions establish representability and resource counts. They do not show that training will find the required computation. The extra reading in Section 14.4 returns to the learning question. Lecture 15 replaces exact equality by approximation error and studies the resulting tradeoff.

14.1 From symbolic computation to Boolean circuits

14.1.1 What symbolic computation meant

The physical symbol system hypothesis proposed that general intelligent action requires and can be produced by a system that stores symbol structures and transforms them by explicit rules. Here a symbol structure is a well-formed expression in a formal language. To specify the syntax of a formal language, one starts with primitive symbols and formation rules that determine which finite strings are well formed. In first-order logic, the nonlogical vocabulary contains constant, function, and relation symbols; variables, logical connectives, and quantifiers are also part of the syntax. A memory can store one well-formed sentence or a collection of sentences. Explicit inference or rewrite rules transform stored expressions into other well-formed expressions. Classical symbolic AI used formal expressions and explicit transformation rules to represent knowledge and carry out deductions.

The symbol grounding problem asks how the formal symbols in such a system become connected to objects, properties, and situations outside the formal system. Syntactic rules determine how one expression produces another, but syntax alone does not determine that a particular constant refers to a particular person or object. An external user can supply an interpretation. The grounding

problem asks how the system itself can acquire and maintain the connections expressed by such an interpretation.

Machine learning eliminates the requirement that a designer enumerate a complete symbolic knowledge base and every rule for manipulating the stored expressions. A learning algorithm instead receives data from outside the formal system and uses the observations and feedback to shape the system's behavior. For a dialogue system, the data can include utterances, responses, and feedback. For a robot, the data can include sensor observations, actions, and their consequences. The relation to the external world can therefore enter through data and interaction rather than only through a hand-written interpretation.

Sequence models add the ability to learn transformations whose inputs and outputs are strings. Well-formed expressions of a formal language are strings, so a learned sequence model can carry out syntactic symbol manipulation as part of the model's behavior. Machine learning therefore bypasses the requirement that all knowledge and manipulation rules be written down in advance. Learning string transformations does not guarantee exact, systematic, or reliable symbol manipulation.

Neural networks are not excluded from symbolic computation by their mathematical form. Recurrent neural networks, and neural networks combined with external memory, were known to simulate universal computation before the development of modern sequence models. The remaining question is whether data, objectives, and optimization produce the required computation reliably. Current systems can perform broadly and still fail on closely related inputs, so the learning question remains open. Current systems also use scratchpads, retrieval, tools, and other external memory. The division used in the Turing-machine construction below therefore reappears: a learned finite component operates together with a memory environment. Lecture 16 studies the resources supplied by the memory environment.

14.1.2 Turing machines reduce universality to finite control

The Church–Turing thesis identifies effectively calculable procedures with the functions computable by a Turing machine. The thesis is not a theorem about an independently formalized notion of every possible procedure; the thesis states the intended scope of the formal model. A deterministic Turing machine has a finite state set Q, a finite tape alphabet Γ, a head that reads and writes one tape cell, and an unbounded tape. One step applies a finite transition function

$$\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\mathtt{L}, \mathtt{R}\}. \tag{14.1}$$

The output specifies the next state, the symbol written at the current cell, and the head movement. The tape and head are the memory environment; the transition function is the finite controller.

Suppose a neural network computes an exact bit encoding of δ. Reapply the same network after every tape update. Induction over time shows that the network-controlled system has exactly the same state, tape, and head position as the Turing machine after every step. A universal Turing machine has one fixed transition function and receives the simulated machine description and input on the tape. Consequently, universal computation requires the neural component to represent only one fixed finite table, provided that the environment supplies an unbounded read–write tape and permits repeated application. A fixed feedforward network without an external tape and repeated execution does not by itself perform an arbitrarily long computation.

14.1.3 Finite transition tables are Boolean functions

Encode the state, scanned symbol, next state, written symbol, and head direction by bits. After extending the transition rule arbitrarily on unused bit strings, each output bit is a Boolean function

$$f : \{0,1\}^r \longrightarrow \{0,1\}.$$

There are 2^{2^r} such functions: a truth table contains 2^r rows, and each row receives one of two output bits. A transition rule with s output bits therefore has $(2^{2^r})^s = 2^{s2^r}$ possible truth tables.

Circuits represent a many-input Boolean function by composing gates with smaller fan-in. Unary gates are insufficient: the four unary Boolean functions are the two constants, the identity, and negation, and a circuit made only of unary gates cannot combine two input coordinates. Exercise 2 asks for the proof. There are sixteen binary Boolean functions, including constants, projections and their negations, AND, OR, NAND, NOR, XOR, and the remaining binary gates. The goal is to find a small collection of gates from which every finite truth table can be built.

Boolean circuits also record two resources that the later compilation should preserve: size measures the number of gates, while depth measures the number of sequential stages. A circuit-to-ReLU translation can therefore establish representability and compare resources.

Write

$$\mathrm{ReLU}(t) = \max\{0, t\}.$$

On bit inputs $x, y \in \{0,1\}$, the formulas

$$\mathrm{NOT}(x) = 1 - x, \qquad \mathrm{AND}(x,y) = \mathrm{ReLU}(x + y - 1) \tag{14.2}$$

are exact. The truth table for AND follows because $x + y - 1$ is positive only when $x = y = 1$. De Morgan's law then gives

$$\mathrm{OR}(x,y) = \mathrm{NOT}\big(\mathrm{AND}(\mathrm{NOT}(x), \mathrm{NOT}(y))\big). \tag{14.3}$$

The De Morgan construction makes circuit composition explicit: once a network implements a collection of gates, connecting the gate outputs builds a larger computation. A more compact formula for OR is recorded in an endnote.[1]

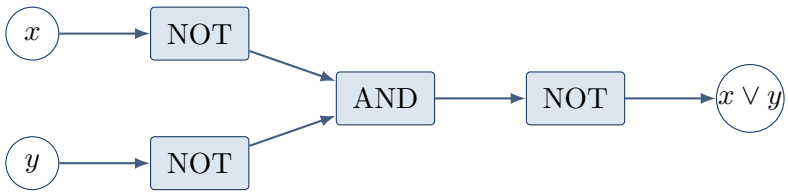


Figure 1: The ReLU realization of OR obtained by composing NOT and AND. The diagram is also a small circuit-compilation example.

Definition 14.1.1 (Boolean circuit)**.** A Boolean circuit is a finite directed acyclic graph. Source nodes contain input bits. Every non-source node applies one of NOT, AND, and OR to the values at its incoming edges. The *size* is the number of gates, and the *depth* is the maximum number of gates on an input-to-output path.

The literature allows many choices of primitive gates. A set of Boolean gates is *functionally complete*, or *universal*, if circuits using those gates compute every Boolean function on every finite

number of inputs. Among subsets of $\{\mathrm{AND}, \mathrm{OR}, \mathrm{NOT}\}$, a set is universal exactly when it contains NOT and at least one of AND and OR. Sufficiency follows from De Morgan's laws. For necessity, circuits without NOT compute only monotone functions, while circuits made only from NOT cannot combine two input bits. No one gate in the displayed three-gate set is individually necessary: NAND alone is universal because

$$\mathrm{NOT}(x) = \mathrm{NAND}(x, x), \qquad \mathrm{AND}(x, y) = \mathrm{NOT}(\mathrm{NAND}(x, y)).$$

The analogous statement holds for NOR.

Functional completeness is what makes circuits a general representation tool, rather than a collection of examples. Once a finite input and output are encoded by bits, the desired computation is a Boolean function. The next theorem constructs a circuit for every such function.

Theorem 14.1.2 (Every Boolean function has a circuit)**.** *For every integer $n \geq 1$ and every function $f : \{0,1\}^n \to \{0,1\}$, there is an $\{\mathrm{AND}, \mathrm{OR}, \mathrm{NOT}\}$ circuit that computes f.*

Proof. For each $a \in \{0,1\}$, $x \in \{0,1\}$, define the literal

$$\ell_a(x) = \begin{cases} x, & a = 1, \\ \mathrm{NOT}(x), & a = 0, \end{cases}$$

and for $a, x \in \{0,1\}^n$, let

$$m_a(x) = \bigwedge_{i=1}^{n} \ell_{a_i}(x_i).$$

The conjunction $m_a(x)$ equals one exactly when $x = a$. Therefore

$$f(x) = \bigvee_{\substack{a \in \{0,1\}^n \\ f(a)=1}} m_a(x). \tag{14.4}$$

If the index set in Eq. (14.4) is empty, use $x_1 \wedge \mathrm{NOT}(x_1)$ for the constant-zero function. Binary trees of gates implement the displayed finite conjunctions and disjunction. □

The explicit construction uses at most $O(n2^n)$ gates. The theorem proves universality, not economy. Circuit complexity asks whether a particular function has a much smaller circuit.

Theorem 14.1.3 (Exact simulation of Boolean circuits)**.** *Let C be a Boolean circuit with bounded fan-in, size S, and depth D. There is a ReLU network that agrees with C on every Boolean input, uses $O(S)$ ReLU units, and has depth $O(D)$.*

Proof. Order the circuit gates by circuit layer. Replace every gate by the constant-size ReLU gadget in Eqs. (14.2) and (14.3). When an edge must cross an added network layer, transmit its bit using $\mathrm{ReLU}(z) = z$ for $z \in \{0,1\}$.

The correctness proof is induction over circuit layers. The input layer agrees by construction. Suppose every wire entering circuit layer d carries the same bit in the circuit and in the ReLU network. Each replacement gadget computes the corresponding gate exactly on the incoming bits, so every wire leaving layer d also agrees. The induction proves equality at the output. Every circuit gate contributes only a constant number of ReLU units and network layers, which gives the stated resource bounds. □

Combining Theorems 14.1.2 and 14.1.3 shows that ReLU networks represent every Boolean function exactly. More strongly, when a Boolean function has a circuit of size S and depth D, the compiled ReLU network preserves size and depth up to constant factors.

Corollary 14.1.4 (Neural finite control with an external tape)**.** *For every deterministic Turing machine with finite state set and finite tape alphabet, a finite ReLU network can implement its transition function exactly on valid bit encodings. Repeatedly applying that network while the environment updates the tape and head simulates the Turing machine step for step. In particular, one fixed finite ReLU controller coupled to an unbounded tape can simulate a universal Turing machine.*

Proof. Each output bit of the finite transition table in Eq. (14.1) is a Boolean function of the encoded current state and scanned symbol. Apply Theorems 14.1.2 and 14.1.3 to each output bit, sharing subcircuits if desired. The resulting network produces the encoded next state, written symbol, and head direction exactly. Induction over the number of steps proves equality with the Turing-machine execution. Choosing a universal Turing machine makes the finite controller independent of the simulated program. The simulated program is supplied on the tape. □

The corollary settles an existence question under an explicit environment. The corollary does not bound the resources needed for a useful algorithm or show that a training procedure finds the controller. The rest of the lecture studies representation size and computations whose data arrive in the input.

14.2 Why depth can be exponentially more economical

Does depth reduce the number of ReLU units needed to represent a function? We will prove that it can. For every L, we will construct a function that a depth-$O(L)$ ReLU network represents with $O(L)$ units, while every exact one-hidden-layer representation of the same function requires at least $2^L - 1$ units. Allowing depth therefore reduces the required size from exponential in L to linear in L for this family.

We first name the network classes used in the comparison. For integers $L, m \geq 1$, let $\mathcal{F}_{L\text{-NN}}(m)$ be the class of functions $f : \mathbb{R} \to \mathbb{R}$ represented by a ReLU network with L hidden layers and at most m hidden ReLU units in total. Each hidden layer applies an affine map and coordinatewise ReLU, and the output layer is affine. Write

$$\mathcal{F}_{L\text{-NN}} = \bigcup_{m \geq 1} \mathcal{F}_{L\text{-NN}}(m).$$

For scalar functions, network composition gives

$$f \in \mathcal{F}_{L\text{-NN}}(m), \quad g \in \mathcal{F}_{K\text{-NN}}(n) \quad \Longrightarrow \quad f \circ g \in \mathcal{F}_{(L+K)\text{-NN}}(m+n). \tag{14.5}$$

The class notation separates the depth of a representation from the particular parameter values used to obtain the representation.

The proof needs a property of a function that grows exponentially when small networks are composed but grows only linearly with the width of one hidden layer. For scalar piecewise-affine functions, the number of affine pieces has exactly these two features. Each scalar ReLU in one hidden layer introduces at most one new breakpoint, so m hidden units produce at most $m + 1$ affine pieces. Repeated composition can split every existing affine piece at once.

The tent map makes both sides of the comparison explicit. The map has two affine pieces, and a constant-size ReLU module computes the tent map exactly. Every piece maps its half of the domain onto all of $[0,1]$, so one more composition splits every current piece into two. Stacking L copies of the module gives a deep network with $O(L)$ units and 2^L affine pieces. A one-hidden-layer network must create the corresponding $2^L - 1$ breakpoints one at a time. The rest of the section proves these two statements.

The tent map is the continuous piecewise-affine function

$$\tau(x) = \min\{2x, 2-2x\} = \begin{cases} 2x, & 0 \le x \le \frac{1}{2}, \\ 2-2x, & \frac{1}{2} \le x \le 1 \end{cases} \qquad (x \in [0,1]). \tag{14.6}$$

The graph joins $(0,0)$, $(1/2,1)$, and $(1,0)$. The slope of τ starts at 2 and changes by -4 at $1/2$. Because $\mathrm{ReLU}(x-1/2)$ introduces exactly the slope change at $1/2$,

$$\tau(x) = 2x - 4\,\mathrm{ReLU}\left(x - \frac{1}{2}\right), \qquad x \in [0,1].$$

Extending the tent map by zero outside $[0,1]$ gives the global ReLU formula

$$\tau(x) = 2\,\mathrm{ReLU}(x) - 4\,\mathrm{ReLU}\left(x - \frac{1}{2}\right) + 2\,\mathrm{ReLU}(x-1), \qquad x \in \mathbb{R}. \tag{14.7}$$

Thus one hidden ReLU layer followed by an affine readout represents the tent map exactly: $\tau \in \mathcal{F}_{1\text{-NN}}(3)$. A network with one hidden ReLU layer and an affine output layer is also called a two-layer ReLU network when both affine maps are counted as layers.

Values of the zero extension outside $[0,1]$ do not affect iterations starting in $[0,1]$ because $\tau([0,1]) = [0,1]$. The extension makes τ a standard network function $\mathbb{R} \to \mathbb{R}$ represented without a direct linear skip from input to output, so copies of the same module can be composed using Eq. (14.5). The graph is also called a *triangle function* in the neural-network depth literature. A periodic extension would be a triangle wave; no periodic extension is used here.

The tent map and its iterates are classical objects in one-dimensional dynamical systems.[2] The neural-network construction uses one further observation: a small ReLU module computes the classical map exactly. Reusing that module by composition produces the L-fold map $\tau^{\circ L}$.

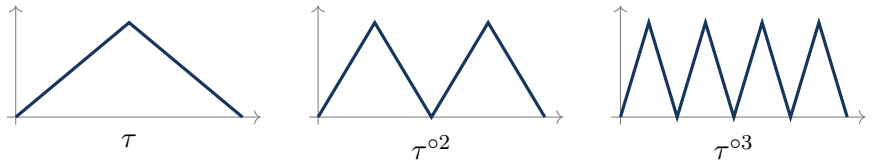


Figure 2: Each composition folds every existing affine piece into two pieces. The visible counts are 2, 4, and 8.

The standard *cobweb diagram* for a dynamical system makes the composition rule visible in a different way. Start at x_0 on the horizontal axis, move vertically to the graph of τ, move horizontally to the line $y = x$, and repeat. The successive diagonal coordinates are $x_1 = \tau(x_0), x_2 = \tau(x_1), \ldots$.

Proposition 14.2.1 (Repeated folding)**.** *For every integer $L \ge 1$, the function $\tau^{\circ L}$ has exactly 2^L maximal affine pieces on $[0,1]$. More precisely, $\tau^{\circ L}$ is affine on every dyadic interval*

$$I_{L,j} = \left[\frac{j}{2^L}, \frac{j+1}{2^L}\right], \qquad j = 0, \ldots, 2^L - 1,$$

and $\tau^{\circ L}$ maps each interval onto $[0,1]$.

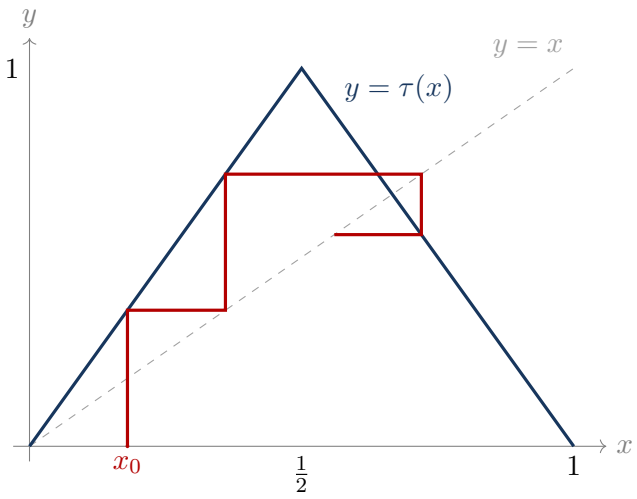


Figure 3: A cobweb diagram for the tent map. Alternating vertical moves to the graph and horizontal moves to the diagonal displays repeated application of the same map.

Proof. The proposition holds for $L = 1$ because the two halves of $[0, 1]$ are mapped affinely onto $[0, 1]$. Suppose the proposition holds at depth L. On each interval $I_{L,j}$, the function $\tau^{\circ L}$ is affine and covers $[0, 1]$. Consequently, there is one point in the interior of $I_{L,j}$ at which $\tau^{\circ L}(x) = 1/2$. Composing with τ folds the graph at the preimage of $1/2$ and splits $I_{L,j}$ into two affine pieces, each of which maps onto $[0, 1]$. The split points are the dyadic points at resolution $2^{-(L+1)}$. Induction gives 2^{L+1} pieces at the next depth. □

Equation (14.7) uses three ReLU units. Applying the same three-unit module L times therefore realizes $\tau^{\circ L}$ with depth L, width three, and $O(L)$ units. The next proposition gives the number of units needed by a single hidden layer to represent the same function.

Proposition 14.2.2 (Breakpoint count for one hidden layer)**.** *Let*

$$f(x) = a_0 + a_1 x + \sum_{r=1}^{m} c_r \,\mathrm{ReLU}(w_r x + b_r). \tag{14.8}$$

The function f has at most $m + 1$ affine pieces on the real line. Hence an exact representation of $\tau^{\circ L}$ in the form (14.8) requires $m \geq 2^L - 1$.

Proof. If $w_r \neq 0$, the rth ReLU term has one possible breakpoint, $-b_r/w_r$. A term with $w_r = 0$ has no breakpoint. After ordering the at most m distinct breakpoints, every summand and therefore their sum is affine on each of the at most $m + 1$ intervening intervals. Proposition 14.2.1 gives 2^L maximal affine pieces for $\tau^{\circ L}$, so $m + 1 \geq 2^L$ is necessary. □

Theorem 14.2.3 (Folding depth separation)**.** *For every integer $L \geq 1$,*

$$\tau^{\circ L} \in \mathcal{F}_{L\text{-}NN}(3L), \qquad \tau^{\circ L} \notin \mathcal{F}_{1\text{-}NN}(2^L - 2).$$

The lower bound remains valid for the larger shallow class that also allows an affine skip $a_0 + a_1 x$ in the output.

Proof. The three-unit representation in Eq. (14.7) and the composition rule (14.5) give the first membership. Every function in $\mathcal{F}_{1\text{-NN}}(m)$ has the form (14.8) with $a_1 = 0$. Proposition 14.2.2 shows that equality with $\tau^{\circ L}$ requires $m \geq 2^L - 1$, even when an affine skip is allowed. □

The theorem is an exact representation statement for one function family. Composition reuses the same small folding operation; a single hidden layer must list all breakpoints separately.

Optional: depth separation on Boolean inputs

The tent-map argument raises a natural question. Does the binary branching of the dyadic intervals conceal a depth separation for Boolean functions? The answer is partly yes. The 2^L intervals form a binary tree whose leaves have addresses in $\{0,1\}^L$. The tent-map construction does not, however, use the adjacency structure of the L-dimensional hypercube. More importantly, the scalar lower bound can disappear when the interval address is presented as bits rather than as one high-precision real number.

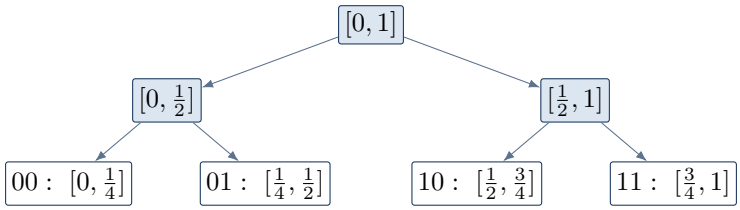


Figure 4: The dyadic intervals form a binary tree. At depth L, an L-bit address identifies one of 2^L leaves. The tree structure alone does not make the leaves into the Boolean hypercube graph.

The next proposition makes the role of the input encoding precise.

Proposition 14.2.4 (One table under two encodings)**.** *Let $N = 2^L$, $x_j = j/N$, and $y_j = \tau^{\circ L}(x_j)$ for $j = 0, \ldots, N-1$. Then $y_j = j \bmod 2$. If a one-hidden-layer scalar ReLU network of the form (14.8) satisfies $f(x_j) = y_j$ at all N points, then $m \geq N/2 - 1$. In contrast, if j is presented by its L-bit binary expansion $b_1 \cdots b_L$, with b_L the least significant bit, then the same table is computed by the coordinate projection $b \mapsto b_L$.*

Proof. The identity

$$\tau^{\circ L}(x) = 2\,\operatorname{dist}\big(2^{L-1}x, \mathbb{Z}\big), \qquad x \in [0,1],$$

gives the alternating values on the dyadic grid. For the lower bound, take second differences across consecutive grid points. The target sequence has $y_{j+1} - 2y_j + y_{j-1} \neq 0$ for every $j = 1, \ldots, N-2$. The affine part of (14.8) has zero second difference, and the sampled values of one ReLU hinge have nonzero second differences at at most two adjacent indices. Consequently, $2m \geq N - 2$, so at least $N/2 - 1$ hinges are necessary. The coordinate-projection claim follows from $j \bmod 2 = b_L$. □

Thus the scalar tent-map example stores L bits of an interval address in the precision of one real input. The corresponding table is easy under the usual bit-vector encoding. Boolean circuit complexity avoids the encoding dependence by studying a family $F_n : \{0,1\}^n \to \{0,1\}$ as the dimension n grows and by fixing the permitted gate set.

Theorem 14.2.5 (Two Boolean depth separations)**.** *Let* $\mathrm{PARITY}_n(x) = x_1 \oplus \cdots \oplus x_n$.

(a) Bounded-fan-in AND, OR, *and* NOT *gates compute* PARITY_n *with size $O(n)$ and depth $O(\log n)$, but every DNF or CNF for parity has at least 2^{n-1} terms or clauses.*

(b) For every fixed $d \geq 2$, there is an explicit Boolean function family computed by polynomial-size depth-d AC^0 circuits but requiring $\exp(n^{c_d})$ size at depth $d-1$, for a constant $c_d > 0$.

Here AC^0 permits unbounded-fan-in AND *and* OR *gates and* NOT *gates at the inputs.*

For part (a), use a balanced tree of constant-size XOR circuits. A DNF term that accepts a parity-one input must mention every variable: otherwise, flipping an omitted bit preserves the term and changes the parity. Each term therefore accepts only one of the 2^{n-1} positive inputs. The CNF claim is dual. Part (b) is the classical AC^0 depth hierarchy based on Sipser functions and switching lemmas; see Håstad [1986] and the average-case refinement of Håstad et al. [2017].

The gate set matters. Parity, which is difficult for DNF and CNF, is shallow for ReLU networks. If $s = \sum_{i=1}^{n} x_i$, then

$$g_n(s) = s + 2\sum_{k=1}^{n-1}(-1)^k \operatorname{ReLU}(s-k) \tag{14.9}$$

equals $s \bmod 2$ at every integer $s \in \{0, \ldots, n\}$. Hence one hidden layer with $n-1$ ReLU units computes parity on $\{0,1\}^n$. The exponential DNF/CNF lower bound therefore does not transfer to ReLU networks.

Threshold and ReLU gates are much stronger than AND and OR. Known threshold-circuit results separate depth two from depth three under additional restrictions such as polynomially bounded weights [Hajnal et al., 1993]. With arbitrary real weights, even strong lower bounds for depth-two and depth-three threshold circuits remain difficult [Kane and Williams, 2016]. The following recent theorem obtains an all-depth result for ReLU networks by using exact real-valued outputs.

Theorem 14.2.6 (Boolean inputs and exact real outputs; Krishnan and Mossel, 2026)**.** *Let $p_1 < \cdots < p_n$ be the first n primes and define*

$$P_n(x) = \sqrt{\prod_{i:x_i=1} p_i}, \qquad x \in \{0,1\}^n.$$

A constant-width ReLU network of depth $n+1$ computes P_n exactly. If a depth-d, width-w ReLU network with arbitrary real weights computes P_n exactly, then $w^d = \Omega(2^n)$.

The output values in Theorem 14.2.6 are linearly independent over $\mathbb{Q}$. The dimension of their $\mathbb{Q}$-linear span supplies the lower bound. Exact real output is also essential: polynomially many leading bits of $P_n(x)$ are computable by polynomial-size constant-depth threshold circuits, the class TC^0. An analogous all-depth separation for bounded-precision ReLU networks with Boolean outputs would imply major new lower bounds against TC^0 and remains open. The tent map, Boolean circuits, and Theorem 14.2.6 therefore establish different depth advantages. Input encoding, gate set, output type, and numerical precision are part of every precise comparison.

14.3 Input-specified maps and their composition

For fixed input and output lengths, a sequence model still computes an ordinary map between finite-dimensional spaces after the symbols have been encoded. The Boolean construction already shows that a neural network can perform any fixed-size Boolean computation. Merely calling the inputs and outputs sequences does not produce a new representational result.

Allowing either length to vary without a fixed bound changes the representational question. One length-uniform program must operate across lengths, rather than using an unrelated network for

each pair of lengths. The examples below vary the number of input records while returning one fixed-size answer.

What do transformers add in these examples? Attention supplies a length-uniform selection rule that is applied in parallel at every position. The input records can also specify the map that the program applies. The map is then part of the input rather than part of the network parameters. A prompt may contain a table, a set of bindings, a family relation, or a collection of redirects, followed by a query about the supplied records. The same selection and update rules can be reused after the names change or more records are added.

Evaluating a function or relation on supplied symbols is a basic symbolic operation: substitute the symbols into the argument positions and return a function value or a truth value. When a finite function or relation is represented by records, associative lookup implements function or relation evaluation. The examples below therefore model a basic step of symbolic manipulation as well as a memory operation.

The input-specified map can also be composed. Associative lookup applies the map once. Iterated lookup uses the returned value as the key for another application of the same supplied map. Each application depends on the result of the preceding application, so the number of lookup rounds records the sequential dependence. The pointer-doubling construction in the extra reading shows how parallel updates can reduce the required number of rounds.

14.3.1 One parallel lookup round

We first need a unit for counting sequential computation. A transformer layer performs many retrievals at once, but a retrieval whose query uses the output of an earlier retrieval must occur in a later layer. We therefore regard one layer as one parallel lookup round.

We use a RASP-style abstraction of that round. At round ℓ, position r stores a state $h_\ell(r)$. The layer forms a query from $h_\ell(r)$, selects one source position by comparing that query with the source keys, retrieves a value from the selected source, and combines the retrieved value with $h_\ell(r)$ by a positionwise map. Every position performs query formation, selection, retrieval, and update simultaneously. Full attention permits any source position. Causal attention permits only source positions at or before the destination. The abstraction retains the operations needed below and suppresses the particular vector encoding and softmax approximation.

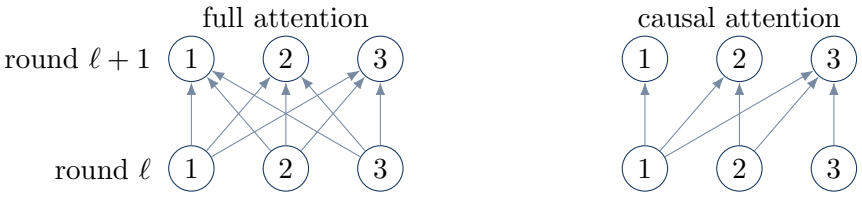


Figure 5: Every dependency allowed by full and causal attention on three positions. A hard lookup selects one of the permitted sources for each destination; the diagrams display the maximum set of possible arrows.

Lemma 14.3.1 (Prefix dependence)**.** *In a program composed of causal selectors and positionwise maps, the state at position r after any number of rounds depends only on the input prefix $(x_1, \ldots, x_r)$.*

Proof. Use induction over rounds. The round-zero state at position r uses only x_r and fixed positional information. A causal selector at position r reads round-ℓ states only at positions $j \le r$,

which depend by the induction hypothesis only on prefixes contained in $(x_1, \ldots, x_r)$; a positionwise map adds no cross-position dependence. □

The lemma converts the causal mask into a reusable limitation on every later state. In particular, a proposed parallel update is invalid if some position must read a map record located later in the sequence. The causal analysis of pointer doubling will use exactly this test.

14.3.2 Associative lookup

Associative lookup asks whether one fixed operation can use facts supplied in the current sequence. The task is the basic operation behind dictionary access: a query supplies a key and the sequence supplies the key–value pairs. In a language context, the pairs may encode an attribute attached to a name or a value assigned to a variable. Changing the names or adding more pairs should not require new network parameters.

Let $\mathcal{D} = \{k_1, \ldots, k_M\}$ be a set of distinct keys, let $\mathcal{V}$ be an arbitrary value set, and let $f : \mathcal{D} \to \mathcal{V}$. Store one pair at each of the first M sequence positions and place a query last:

$$X(f, x) = \big((k_1, f(k_1)), \ldots, (k_M, f(k_M)), (\texttt{QUERY}, x)\big), \qquad x \in \mathcal{D}. \tag{14.10}$$

Associative lookup returns $f(x)$. The keys are symbols, not position indices, and the values need not be keys.

One hard lookup round suffices. The final position uses x as its query, selects the unique earlier position whose key equals x, and retrieves the value stored there. Equality matching and value retrieval do not depend on M, so the same program handles every sequence length.

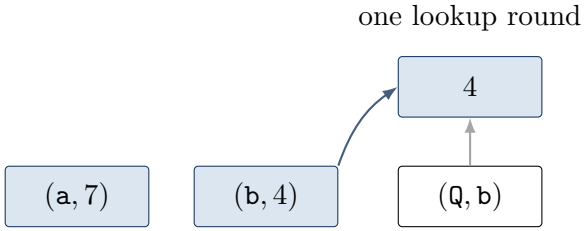


Figure 6: Associative lookup with one key–value pair stored at each position. The query matches the key `b` and retrieves the paired value 4; the gray arrow denotes the residual query state.

Proposition 14.3.2 (Exact associative lookup)**.** *Under the distinct-key and present-query assumptions, one hard attention round computes $X(f, x) \mapsto f(x)$ for every finite key set $\mathcal{D}$ and every map $f : \mathcal{D} \to \mathcal{V}$. The construction is causal when the query follows all stored pairs.*

Proof. Distinctness and the assumption $x \in \mathcal{D}$ give a unique index r with $k_r = x$. Equality matching therefore selects position r and retrieves $f(k_r) = f(x)$. The selected position precedes the query position, so the lookup is causal. Neither the equality predicate nor the retrieval rule depends on M. □

The one-pair-per-position encoding isolates the lookup operation from record assembly. An ordinary token sequence may instead place the key and value at consecutive positions,

$$(k_1, f(k_1), \ldots, k_M, f(k_M), \texttt{QUERY}, x).$$

One preparation round lets every value position retrieve its predecessor key and combine the retrieved key with its residual value. The resulting key–value pairs support associative lookup in the next round. Exercise 12 asks you to compose two tables supplied in the same input, so that the first retrieved value becomes the key for a second lookup.

14.3.3 Iterated lookup

A single lookup retrieves one supplied fact. Many symbolic computations use the retrieved value to determine the next operation. For example, after retrieving the mother of a person, the retrieved name becomes the key in the query for the maternal grandmother. The second query is not known until the first lookup has returned the selected value. Iterated lookup isolates the compositional dependence.

Iteration becomes defined when the values are also keys. Specialize the same formalism to a self-map $p : \mathcal{D} \to \mathcal{D}$ and ask for

$$p^K(x) = \underbrace{p \circ \cdots \circ p}_{K \text{ copies}}(x), \qquad p^0(x) = x.$$

The map is supplied by the pairs in $X(p, x)$; the network parameters specify the lookup procedure, not the particular map.

The example in Fig. 7 uses exactly three lookups, so the input needs only the three records on the queried path: Dana's mother, Cleo's mother, and Bea's mother. Ada is the final returned value and need not appear as a key. The general self-map formulation is needed only when every returned value may become the key for another lookup. The same formal computation models following links in a linked list, redirects in a table, or ancestors in a rooted forest.

A direct procedure keeps the current key at the query position:

$$z_0 = x, \qquad z_{t+1} = p(z_t), \qquad t = 0, \ldots, K-1. \tag{14.11}$$

Round $t+1$ selects the pair keyed by z_t and retrieves its value. Therefore $z_t = p^t(x)$ by induction, and K lookup rounds return $p^K(x)$. The construction is causal for every self-map p when all map pairs precede the query.

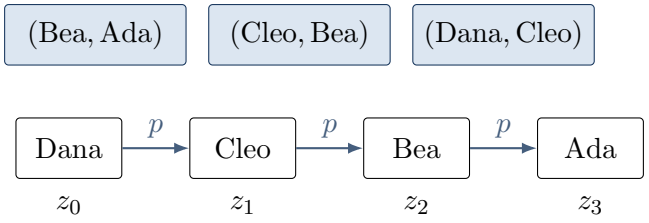


Figure 7: A map described by the input and the successive query states. Each arrow is one lookup round. With $p(u)$ equal to the mother of u, the states name Dana, her mother, her maternal grandmother, and her maternal great-grandmother.

14.3.4 Parallel pointer doubling

Transformers update all sequence positions in parallel. The direct procedure above uses only the query position to advance along the map, so it takes K sequential layers for K map applications

and leaves the parallel work at the map-record positions unused. We now ask whether simultaneous computation at those positions can return the same iterate in fewer sequential rounds.

Pointer doubling gives an affirmative answer under full attention. The map positions preprocess the input map in parallel and store progressively longer jumps. The construction reduces sequential depth by performing more parallel work; it does not reduce the total number of table updates. The preprocessing is especially useful when the same input map will answer many ancestor, link, or redirect queries. At the position keyed by $u \in \mathcal{D}$, store

$$q_0(u) = p(u), \qquad q_{\ell+1}(u) = q_\ell(q_\ell(u)). \tag{14.12}$$

To perform one update, position u uses its current value $q_\ell(u)$ as a query key and retrieves $q_\ell(q_\ell(u))$ from the matching map position. Every position performs the update simultaneously.

Proposition 14.3.3 (Doubling invariant)**.** *For every self-map $p : \mathcal{D} \to \mathcal{D}$, every $u \in \mathcal{D}$, and every $\ell \geq 0$,*

$$q_\ell(u) = p^{2^\ell}(u).$$

Thus L parallel doubling rounds transform the one-step table q_0 into the 2^L-step table q_L.

Proof. The base case is $q_0(u) = p(u) = p^{2^0}(u)$. If the invariant holds at round ℓ, then

$$q_{\ell+1}(u) = q_\ell(q_\ell(u)) = p^{2^\ell}\big(p^{2^\ell}(u)\big) = p^{2^{\ell+1}}(u).$$

Induction proves the invariant. □

The invariant contains two separate facts. First, all table entries can be updated in parallel. Second, the resulting entries answer later queries: when $K = 2^L$, the query for $p^K(x)$ retrieves $q_L(x)$. For general K, retain the fields $q_0, \ldots, q_{\lfloor \log_2 K \rfloor}$ and compose the fields selected by the binary expansion of K. For example,

$$K = 13 = 8 + 4 + 1, \qquad x \xrightarrow{q_3} p^8(x) \xrightarrow{q_2} p^{12}(x) \xrightarrow{q_0} p^{13}(x).$$

The preprocessing and query phases each use $O(\log K)$ rounds.

key u	a	b	c	d	e
$q_0(u) = p(u)$	b	c	d	e	c
$q_1(u) = p^2(u)$	c	d	e	c	d
$q_2(u) = p^4(u)$	e	c	d	e	c

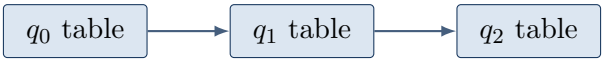


Figure 8: Pointer doubling for the map $a \mapsto b \mapsto c \mapsto d \mapsto e \mapsto c$. Each row is a stream distributed across all five map positions. One round updates every column, so two rounds produce the four-step map $q_2 = p^4$.

14.3.5 What changes under causal attention

Decoder-only transformers normally use a causal mask, so a full-attention speedup is relevant only after checking that every retrieval respects the sequence order. The sequential query construction

in Eq. (14.11) remains causal for an arbitrary self-map: every map pair precedes the final query. Parallel doubling imposes an additional requirement because a map position must retrieve from another map position. Fix an order of the pairs and write $\mathrm{pos}(u)$ for the position storing the pair keyed by u. The first doubling update is causal when

$$\mathrm{pos}(p(u)) \le \mathrm{pos}(u) \qquad \text{for every } u \in \mathcal{D}. \tag{14.13}$$

The same inequality then holds for every iterate of p, so all later doubling rounds are causal as well.

Condition (14.13) is a restriction, not a relabeling that works for every self-map. A family forest can be ordered with each mother before her children. A directed cycle of length greater than one cannot be ordered so that every edge points backward. In the special case $\mathcal{D} = [M]$ with the pair keyed by i stored at position i, the condition is $p(i) \le i$; the graph points $(i, p(i))$ then lie on or below the diagonal $y = x$. The picture resembles a discrete cobweb diagram, but the inequality is needed only for causal parallel preprocessing, not for defining or computing the iterates themselves.

14.4 Extra reading: program correctness, representation, and learning

The associative-lookup, iterated-lookup, and pointer-doubling constructions prove the correctness of abstract lookup programs. An empirical claim about a trained transformer requires two additional steps: the architecture must implement the abstract operations with stated resources, and training must find suitable parameters from finite data. Keeping the three statements separate identifies what a construction, a compiler theorem, or an experiment contributes:

1. **Program correctness.** A symbolic sequence program computes the desired map for every valid input.

2. **Model representation.** A specified transformer architecture implements or approximates the program with stated depth, width, head count, and numerical assumptions.

3. **Learning.** A specified training procedure, sample, and optimization process produce parameters that execute the program, including at lengths not used in training.

The RASP language supports the first aspect by expressing streams, selectors, and aggregations. Compiler results and direct constructions support the second aspect for suitable transformer models. RASP-L restricts the program language to operations intended to be length-uniform. The RASP-L generalization claim is a conjecture supported by experiments: tasks with short RASP-L programs tend to be learned with better length generalization. Program existence alone does not imply that gradient-based training finds the program.

RASP-L studies whether the existence of a short symbolic program predicts length generalization across tasks. MicroPy examines a complementary question: whether one learned transition mechanism can be applied many more times at test time than occurred in training executions. A 59.5-million-parameter transformer was trained to execute programs whose training traces contained at most 128 lines. Across 252 reported test instances, including bit operations, arithmetic, and SAT, the model achieved 100 percent token accuracy; the longest successful execution contained 7,552 trace lines, about 59 times the training cap. The experiment demonstrates learned repeated computation in the trained MicroPy system. The relevant extrapolation variable is execution-trace length; the reported maximum test context was 2,197 tokens, and the paper does not give a matching training-context range from which to infer context-length extrapolation.

14.5 Extra reading: arithmetic assumptions

Counts of layers, heads, and coordinates do not by themselves specify the computational resources of an exact construction. Equality tests may require score margins, and a real-valued coordinate with unrestricted precision can encode unbounded information.

The Boolean construction in Section 14.1 uses small integer weights and has exact semantics on bit inputs. The attention construction uses exact equality predicates and hard selection. A finite-temperature softmax can approximate a hard selector when the selected score has a positive margin, but the required scale depends on that margin and the desired error.

Resource statements must therefore identify the arithmetic model. Fixed finite precision bounds the number of distinguishable values in each internal coordinate. Growing precision introduces another resource. Unrestricted real parameters can encode infinitely many bits and can hide advice about the answers. Size and depth counts are meaningful when the construction also controls weight magnitudes, margins, or bit complexity.

14.6 Extra reading: a matching conditional lower bound

The pointer-doubling construction proves that logarithmic depth is sufficient. A lower bound asks whether a substantially shallower parameter-efficient masked transformer could solve the same kind of repeated-lookup problem by a different method. There is a conditional lower bound for a standard repeated lookup problem. For a sequence $X = (X_1, \ldots, X_N)$, define the previous-link map

$$\mathrm{find}_X(i) = \max\big(\{0\} \cup \{j \in [N] : j \le i,\ X_{j-1} = X_i\}\big),$$

where position 0 contains a distinguished null symbol. The k-hop task iterates the previous-link map k times and returns the symbol reached by the chain.

Assume the one-versus-two-cycle conjecture for massively parallel computation. For any constants $\xi \in (0, 1/2]$ and $\varepsilon \in (0, 1)$ and any even $k = \Theta(N^{\xi})$, every masked transformer that computes the k-hop task, has embedding dimension m, has H heads per layer, and satisfies $mH = O(k^{1-\varepsilon})$ must have depth

$$L = \Omega(\log k).$$

Sanford et al. also give an $O(\log k)$ masked-transformer construction for the same task. The upper and lower bounds give a matching logarithmic dependence on k for parameter-efficient masked transformers, conditional on the stated parallel-computation conjecture.

14.7 Take-home messages

- A Turing machine separates finite control from unbounded memory. Exact representation of its finite transition table by a ReLU network, together with repeated application and an external tape, is enough for universal computation. The tape and repeated execution are required computational resources.
- Universal Boolean gates represent every Boolean function. ReLU networks simulate a circuit of size S and depth D exactly with $O(S)$ size and $O(D)$ depth. The proof compiles gates and uses induction over circuit layers.
- Repeated composition can be exponentially more economical than one hidden layer: $\tau^{\circ L} \in \mathcal{F}_{L\text{-NN}}(3L)$, whereas $\tau^{\circ L} \notin \mathcal{F}_{1\text{-NN}}(2^L - 2)$. The zero extension of the tent map supplies a standard network map on $\mathbb{R}$; only its values on $[0, 1]$ affect the iterates.

- One length-uniform attention program can apply a map supplied as key–value pairs in its input. If the stored values are also keys, K successive rounds compose that supplied map and compute $p^K(x)$.
- Under full attention, pointer doubling constructs the 2^L-step map in L parallel rounds. Causal parallel doubling additionally requires the map records to be ordered so that every retrieved record is no later than the requesting record.
- Program correctness, transformer representation, and learning the program from data are distinct aspects. Exact constructions settle the first two under stated models; RASP-L and MicroPy provide empirical evidence about the third.

Endnotes

1. For $x, y \in \{0,1\}$, $\mathrm{OR}(x,y) = x + y - \mathrm{ReLU}(x+y-1)$. The De Morgan construction in Eq. (14.3) exposes circuit composition more directly.

2. **The tent map in dynamical systems.** The full-height tent map is an exactly solvable model used to study symbolic dynamics, periodic orbits, invariant measures, mixing, sensitivity to initial conditions, and entropy. Recording whether each iterate lies to the left or right of $1/2$ gives binary symbolic dynamics, apart from the usual ambiguity at points whose orbit reaches $1/2$. The 2^L affine pieces of $\tau^{\circ L}$ imply topological entropy $\log 2$. The orbit $2/9 \mapsto 4/9 \mapsto 8/9 \mapsto 2/9$ has least period three, so the Li–Yorke theorem applies [Li and Yorke, 1975]. The commonly pictured logistic map $f(x) = 4x(1-x)$ is not merely similar: the logistic map is topologically conjugate to the tent map through $h(x) = \sin^2(\pi x/2)$, since $f \circ h = h \circ \tau$. See Alligood et al. [1997] for the surrounding dynamical-systems theory.

Bibliographical notes

The Church–Turing thesis is associated with the equivalent formalisms introduced by Church [1936] and Turing [1937]. The lecture uses the ordinary computability thesis, not the stronger claim that every physically realizable process admits an efficient digital simulation.

Newell and Simon [1976] stated the physical symbol system hypothesis as a necessary and sufficient account of general intelligent action. Harnad [1990] formulated the symbol grounding problem and discussed learned nonsymbolic representations as a possible component of grounding. Together, the physical symbol system hypothesis and the grounding problem separate symbolic representation from the question of how a system acquires its representations.

Siegelmann and Sontag [1995] proved that a fixed finite recurrent neural network with rational weights can simulate Turing computation under their synchronous recurrent model. The elementary construction in Corollary 14.1.4 instead places the unbounded tape explicitly in the environment and uses the network only for finite control. Both results show why representability and learnability must be analyzed separately.

The circuit-counting exercise uses the description-counting method of Shannon [1949]: count the functions represented by circuits below a given size and compare that count with the number of all Boolean functions. The final lower bound is a pigeonhole argument and is nonconstructive. The monotone and affine invariants in another exercise are elementary cases of the composition-closed classes in Post [1941].

The recurrent-automaton exercise isolates the finite-state part of the simulation question. The fixed-precision exercise gives an elementary finite-state limitation adjacent to the arithmetic assumptions in Pérez et al. [2021]; their transformer completeness theorem instead permits rational activations of arbitrary precision.

The tent-map construction and the use of rapidly growing oscillation to separate deep and shallow networks follow the depth-separation program of Telgarsky [2016]. The elementary one-dimensional breakpoint argument in Proposition 14.2.2 is sufficient for the exact separation used here. Telgarsky uses the term triangle function for the same single-peak shape. The dynamical-systems interpretation of the tent map is developed by Alligood et al. [1997]; the period-three theorem is due to Li and Yorke [1975].

Weiss et al. [2021] introduced RASP as a computational model for describing transformer operations through sequence-valued maps, selection, and aggregation. The language makes dependency and resource accounting visible. Zhou et al. [2024] introduced RASP-L and formulated the empirical conjecture connecting short length-uniform programs with length generalization.

The associative-memory exercise uses the continuous Hopfield energy and update studied by Ramsauer et al. [2021], whose update coincides with softmax attention when the stored keys are also the retrieved values. The two-round successor-retrieval exercise formalizes the induction-head mechanism studied empirically by Olsson et al. [2022]; the exercise proves only the stated abstract lookup construction.

Pointer jumping is a classical parallel algorithm. The masked-transformer k-hop construction and the conditional lower bound in Section 14.6 are due to Sanford et al. [2024]. Their paper also gives unconditional tradeoffs for several restricted recurrent, kernel, and sparse-attention architectures. The conditional result quoted here directly matches logarithmic masked-transformer depth under the theorem's parameter constraint.

The MicroPy experiment is reported by Xu et al. [2026]. Its execution traces externalize intermediate program state, allowing the same learned transition mechanism to be applied many more times at test time than during training.

Glossary

Physical symbol system hypothesis The hypothesis that a physical system capable of storing and manipulating well-formed symbolic expressions by explicit rules has the necessary and sufficient means for general intelligent action.

Symbol grounding problem The problem of explaining how formal symbols become connected to objects, properties, and situations outside the formal system, rather than receiving the symbol-to-world connections only from an external interpretation.

Turing-machine transition function The finite map from the current control state and scanned tape symbol to the next state, written symbol, and head movement. The unbounded tape is separate from the finite transition controller.

Boolean circuit A directed acyclic graph of Boolean gates. Size counts gates; depth counts the longest input-to-output gate path.

Functionally complete gate set A collection of Boolean gates from which circuits can compute every Boolean function on every finite number of inputs.

Exact representation Equality between the target function and the network output on the stated input domain, rather than approximation within a nonzero error tolerance.

Affine piece A maximal interval on which a scalar function has the form $x \mapsto ax + b$.

$\mathcal{F}_{L\text{-}\mathbf{NN}}(m)$ The class of scalar functions represented by a ReLU network with L hidden layers and at most m hidden ReLU units in total.

Stream A sequence-valued variable $s : [N] \to \mathcal{D}$ with one value per position.

Causal selector A content- or position-dependent rule that lets position i select only positions $j \le i$.

Associative lookup Retrieval of the value paired with a specified key.

Iterated lookup Repeated associative lookup for a self-map $p : \mathcal{D} \to \mathcal{D}$, where each returned value supplies the key for the next lookup.

Pointer doubling The parallel update $q_{\ell+1}(u) = q_\ell(q_\ell(u))$, which doubles the number of original map applications represented after each stage.

Length-uniform program One sequence program whose operations do not change with the input length.

14.8 Exercises

General instructions. Justify every answer, including yes/no answers, even when the question does not explicitly ask for a justification.

Exercise profiles. Filled mountains show difficulty on a four-mountain scale: one is a direct application, two uses a familiar method, three combines ideas, and four is a substantial challenge. Filled cogs show technical load on a three-cog scale: one is light, two is sustained, and three is dense. Filled hourglasses show estimated time: one is under 20 minutes, two is 20–45 minutes, three is 45–90 minutes, and four is more than 90 minutes. Pale pictograms show the unused part of each scale. The work row uses the canonical order proof, calculation, construction, computation, and interpretation; pale icons mark work types not used. ◎ marks the intended learning outcome. Time estimates cover the non-optional parts. Ratings are not prerequisites or grading weights. A superscript asterisk marks an exercise based on supplementary lecture material; it is not a difficulty rating and does not make the exercise optional when assigned.

Symbols and grounding

Exercise 1: Symbol renaming and external grounding.

◎ Separate invariance under symbol renaming from the external relation that assigns meanings to symbols.

This exercise gives a precise model of formal symbol manipulation. The model isolates what equality, copying, and lookup preserve, and the final parts state exactly what additional information a grounding relation supplies.

Let $\mathcal{S}$ be a finite set of symbols. A *symbolic input* x is an ordered finite collection of symbols together with zero or more finite tables of ordered pairs from $\mathcal{S} \times \mathcal{S}$. A deterministic *symbolic program* $\mathcal{A}$ has symbol registers and Boolean registers, contains no hard-coded element of $\mathcal{S}$, and may perform only the following operations:

(i) copy a symbol from an input location or symbol register;

(ii) test whether two symbol registers contain the same symbol;

(iii) use a symbol as a key in an input table with distinct keys and copy the corresponding value; and

(iv) branch according to a Boolean register.

Assume that $\mathcal{A}$ terminates and returns one symbol in $\mathcal{S}$ on every valid input.

Let $\pi : \mathcal{S} \to \mathcal{S}$ be a bijection. Define πx by replacing every symbol s occurring in x, including every table key and value, by $\pi(s)$, while preserving positions and table order. For an output symbol s, the expression $\pi(s)$ has its ordinary meaning: apply the map π to s. The program $\mathcal{A}$ itself is unchanged.

(a) Prove that copying, equality testing, and table lookup commute with the renaming π. State the corresponding relation for each operation.

(b) Prove by induction over the executed operations that

$$\mathcal{A}(\pi x) = \pi\big(\mathcal{A}(x)\big) \qquad \text{for every valid symbolic input } x.$$

(c) Let $\mathcal{S} = \{s_0, s_1\}$ and let $\mathcal{O} = \{o_0, o_1\}$ be a set of external objects. A *grounding* is a bijection $g : \mathcal{S} \to \mathcal{O}$. Consider

$$g_0(s_i) = o_i, \qquad g_1(s_i) = o_{1-i}, \qquad i \in \{0, 1\}.$$

Fix one symbolic input x, and suppose $\mathcal{A}$ receives neither g_0 nor g_1. Prove that $\mathcal{A}(x)$ cannot name the target object o_0 under both groundings; that is, at least one of

$$g_0(\mathcal{A}(x)) = o_0, \qquad g_1(\mathcal{A}(x)) = o_0$$

must fail.

(d) Explain how parts (b) and (c) differ. Your explanation must identify what the renaming result proves, why the impossibility in part (c) requires the grounding to be absent from the input, and what kind of additional input could break the ambiguity.

Boolean circuits and recurrent computation



Exercise 2: Prove that unary gates cannot combine input bits.

Identify the structural limitation of circuits whose gates have at most one input.

Let $r \geq 2$. Consider a Boolean circuit with input bits $x_1, \ldots, x_r$ in which every gate has fan-in at most one. A zero-input gate may output a constant, and a one-input gate may apply any function $g : \{0, 1\} \to \{0, 1\}$. Fan-out is unrestricted.

(a) Prove by induction over a topological ordering of the circuit that the value at every gate is either constant or a function of one input coordinate x_i.

(b) Deduce that no such circuit computes $\mathrm{AND}(x_1, x_2)$ or $\mathrm{XOR}(x_1, x_2)$ on all four Boolean inputs.

(c) List all four functions $g : \{0, 1\} \to \{0, 1\}$. Explain why allowing every unary Boolean gate still does not give a functionally complete gate set for functions of two or more input bits.

Exercise 3: Invariants that prevent functional completeness.

Use closure under composition to prove that two natural Boolean gate families are not functionally complete.

Functional completeness can fail because every available gate preserves a property that some target Boolean function violates. This exercise develops the monotone and affine invariants, two basic examples of that method.

For $x, y \in \{0,1\}^r$, write $x \leq y$ when $x_i \leq y_i$ for every i. A Boolean function f is *monotone* when $x \leq y$ implies $f(x) \leq f(y)$. A Boolean function is *affine over* $\mathbb{F}_2$ when it has the form

$$f(x) = c \oplus a_1 x_1 \oplus \cdots \oplus a_r x_r, \qquad c, a_1, \ldots, a_r \in \{0,1\},$$

where $\oplus$ denotes addition modulo two.

(a) Prove that every circuit whose gates are monotone computes a monotone function. Deduce that $\{\mathrm{AND}, \mathrm{OR}, 0, 1\}$ is not functionally complete.

(b) Prove that composing affine Boolean functions produces another affine Boolean function. Deduce that $\{\mathrm{XOR}, \mathrm{NOT}, 0, 1\}$ is not functionally complete by proving that AND is not affine.

(c) State the general obstruction illustrated by parts (a)–(b). Explain why finding a composition-closed property shared by all available gates and violated by one Boolean function is sufficient to disprove functional completeness.

Exercise 4: Parity requires exponentially many DNF terms.[*]

Prove that changing the allowed representation can change parity from a linear-size computation to an exponential-size formula.

The parity function has a linear-size XOR circuit. This exercise proves that the same function has no comparably small disjunctive normal form, so the representation language is part of every size comparison.

Fix an integer $r \geq 1$, and let

$$\mathrm{PARITY}_r(x) = x_1 \oplus \cdots \oplus x_r, \qquad x \in \{0,1\}^r.$$

A *term* is a conjunction of literals, where a literal is x_i or $\mathrm{NOT}(x_i)$. A *DNF* is a disjunction of terms. Terms containing both x_i and $\mathrm{NOT}(x_i)$ may be removed because they are never satisfied.

(a) Prove that exactly 2^{r-1} inputs have parity one.

(b) Let a satisfiable term occur in a DNF that computes PARITY_r. Prove that the term must contain a literal for every input coordinate. Deduce that the term is satisfied by at most one input.

(c) Prove that every DNF for PARITY_r has at least 2^{r-1} terms. Construct a DNF with exactly 2^{r-1} terms, and therefore prove that the minimum number of terms is exactly 2^{r-1}.

(d) Explain why the lower bound concerns the DNF representation and does not contradict the existence of a linear-size XOR circuit.

Exercise 5: Most Boolean functions require large circuits.

Use a counting argument to distinguish universal representation from efficient representation.

The Boolean universality theorem constructs a circuit for every Boolean function, but the theorem does not give a small circuit for every function. This exercise supplies the counting plan, so the task is to carry out its consequences rather than invent a circuit encoding.

Fix r input bits. A circuit has s numbered gates in topological order. Each gate chooses one of the sixteen functions $\{0,1\}^2 \to \{0,1\}$, and each of its two ordered incoming wires may come from an input or an earlier gate. The circuit output may be any input or gate value. Let $N_{r,s}$ be the number of Boolean functions representable by circuits with at most s gates.

(a) Prove the description-count bound

$$N_{r,s} \le (s+1)(r+s)\big(16(r+s)^2\big)^s.$$

The bound may count several descriptions of the same circuit or function. Explain why such overcounting does not invalidate the inequality.

(b) Define

$$s_r = \left\lfloor \frac{2^r}{16r} \right\rfloor.$$

Show that, for all sufficiently large r,

$$\log_2 N_{r,s_r} \le 2r + 3rs_r \le 2r + \frac{3}{16}2^r < 2^r.$$

You may use $r + s_r \le 2^r$ and $4 + 2r \le 3r$ once r is sufficiently large.

(c) There are 2^{2^r} Boolean functions on r input bits. Deduce that some Boolean function requires more than s_r gates. Show further that the fraction of Boolean functions representable with at most s_r gates tends to zero as $r \to \infty$.

(d) State what the argument establishes about circuit size and what it does not provide about an explicit hard Boolean function.

Hints For part (a), count descriptions gate by gate, using $r + s$ as an upper bound on every predecessor choice, and then choose the output. For part (b), take base-two logarithms before substituting s_r. Part (c) is a pigeonhole argument.

Exercise 6: One ReLU unit cannot compute OR.

Prove that one affine map followed by one ReLU cannot represent OR on the Boolean domain.

Let $g(x,y) = \mathrm{ReLU}(ax + by + c)$, where $a, b, c \in \mathbb{R}$. Prove that no choice of a, b, c makes $g(x,y)$ equal to $\mathrm{OR}(x,y)$ for all $(x,y) \in \{0,1\}^2$.

Exercise 7: Construct a recurrent ReLU simulation of a finite automaton.

Construct one fixed neural transition cell and distinguish its parameter count from the number of times it is executed.

The Turing-machine model separates finite control from external memory. This exercise constructs the finite-control part directly and shows how recurrence reuses one transition network for every input position.

Let Q and Σ be finite sets, and let $\delta : Q \times \Sigma \to Q$ be the transition function of a deterministic finite automaton. Encode $q \in Q$ and $a \in \Sigma$ by the one-hot vectors $e_q \in \{0,1\}^{|Q|}$ and $e_a \in \{0,1\}^{|\Sigma|}$.

(a) Construct a one-hidden-layer ReLU network

$$F : \mathbb{R}^{|Q|+|\Sigma|} \to \mathbb{R}^{|Q|}$$

with at most $|Q||\Sigma|$ hidden units and an affine output layer such that

$$F(e_q, e_a) = e_{\delta(q,a)} \qquad (q \in Q,\ a \in \Sigma).$$

Give all hidden-unit and output formulas.

(b) For an input string $a_1 \cdots a_T$, define

$$h_0 = e_{q_0}, \qquad h_t = F(h_{t-1}, e_{a_t}), \qquad t \in [T].$$

Prove that h_t is the one-hot encoding of the automaton state after reading $a_1 \cdots a_t$.

(c) Compare the recurrent implementation with a feedforward network obtained by unrolling T copies of F. State separately how the number of distinct parameters, the number of executed cell evaluations, and the unrolled depth depend on T.

Hints On bit inputs, one ReLU can indicate that two specified coordinates are both one. If a second hint is needed, introduce one hidden unit for every pair $(q, a) \in Q \times \Sigma$, then collect the pairs that lead to the same next state.

Exercise 8: Finite precision produces an eventually periodic state.[*]

Identify the finite-state limitation created by fixed memory and fixed precision.

Turing-completeness claims depend on the available memory or numerical precision. This exercise proves a limitation that follows from fixed finite precision without using any architecture-specific lower bound.

Consider a deterministic recurrent system with d state coordinates, each stored with exactly b bits. The system reads the unary input $a^t = a \cdots a$ one symbol at a time and produces one output bit after each update. The state-update rule and output rule are fixed for all input lengths.

(a) Prove that the recurrent system has at most 2^{bd} internal states. Deduce that its state sequence on the infinite unary stream $a, a, a, \ldots$ is eventually periodic, with both the preperiod and period bounded by 2^{bd}.

(b) Deduce that the output sequence is eventually periodic.

(c) Let

$$\mathcal{L}_{\text{pow2}} = \{a^{2^k} : k \geq 0\}.$$

Prove that the indicator sequence $y_t = \mathbb{I}(a^t \in \mathcal{L}_{\text{pow2}})$ is not eventually periodic. Conclude that no recurrent system satisfying the stated fixed-precision assumptions recognizes $\mathcal{L}_{\text{pow2}}$ correctly at every input length.

(d) Explain which step of the proof fails when the system has an external unbounded tape or a state coordinate whose available precision can grow with the execution length.

Depth and symbolic dynamics

Exercise 9: Construct the exact shallow form of an iterated tent map.

Use slope changes to construct the smallest one-hidden-layer representation allowed by the affine-skip form.

Define the tent map on $[0, 1]$ by

$$\tau(x) = \min\{2x, 2 - 2x\}.$$

Fix an integer $L \geq 1$. You may use the lecture result that $\tau^{\circ L}$ is affine on the 2^L dyadic subintervals of $[0,1]$ and maps each subinterval onto $[0,1]$.

(a) Show that the slope of $\tau^{\circ L}$ on the interval $[j/2^L, (j+1)/2^L]$ is $(-1)^j 2^L$. Use the slope changes at the interior dyadic points to derive

$$\tau^{\circ L}(x) = 2^L x + 2^{L+1} \sum_{j=1}^{2^L-1} (-1)^j \operatorname{ReLU}\left(x - \frac{j}{2^L}\right), \qquad x \in [0,1].$$

(b) Consider the one-hidden-layer form

$$f(x) = a_0 + a_1 x + \sum_{r=1}^{m} c_r \operatorname{ReLU}(w_r x + b_r).$$

Use part (a) and the lecture's breakpoint lower bound to prove that the smallest m for which this form can equal $\tau^{\circ L}$ on $[0,1]$ is exactly $2^L - 1$.

(c) Compare the shallow representation from part (b) with the membership $\tau^{\circ L} \in \mathcal{F}_{L\text{-NN}}(3L)$. State separately how depth and the total number of hidden units scale with L in the two constructions.

Exercise 10: A robust separation from alternating tent-map labels.

Extend the exact folding separation to a quantitative classification-error lower bound.

The lecture proves that an exact shallow representation of an iterated tent map needs exponentially many hinges. This exercise shows that a shallow network also needs exponentially many pieces to obtain small error on a specified finite classification problem.

Fix $L \geq 1$, put $N = 2^L$, and define

$$x_j = \frac{j + 1/4}{N}, \qquad y_j = \mathbb{I}\big(\tau^{\circ L}(x_j) \geq 1/2\big), \qquad j = 0, \ldots, N-1.$$

For a scalar function f, define

$$\widehat{y}_j(f) = \mathbb{I}(f(x_j) \geq 1/2), \qquad \operatorname{err}_L(f) = \frac{1}{N} \sum_{j=0}^{N-1} \mathbb{I}\big(\widehat{y}_j(f) \neq y_j\big).$$

(a) Use the slopes and endpoint values of $\tau^{\circ L}$ to prove

$$\tau^{\circ L}(x_j) = \begin{cases} 1/4, & j \text{ even}, \\ 3/4, & j \text{ odd}, \end{cases} \qquad y_j = j \bmod 2.$$

(b) Let $\widehat{y}_0, \ldots, \widehat{y}_{N-1}$ be any binary sequence with R maximal constant runs. Prove that its number of disagreements with the alternating sequence $0, 1, 0, 1, \ldots$ is at least $(N - R)/2$.

(c) Suppose f is continuous and affine on at most K intervals. Prove that $\widehat{y}_0(f), \ldots, \widehat{y}_{N-1}(f)$ has at most $2K$ constant runs. Deduce

$$\operatorname{err}_L(f) \geq \frac{1}{2} - \frac{K}{2^L}.$$

(d) A one-hidden-layer scalar ReLU network with m hidden units is affine on at most $m+1$ intervals. Prove that

$$\mathrm{err}_L(f) < \frac{1}{4} \quad \Longrightarrow \quad m+1 > 2^{L-2}.$$

Compare this necessary shallow width with the $O(L)$-unit deep representation of $\tau^{\circ L}$.

HINTS For part (b), count the errors inside each constant run separately. For part (c), the superlevel set of an affine function on one interval is empty, the full interval, or an interval touching one endpoint.

Exercise 11: Symbolic histories of the tent map.

Connect the affine pieces of an iterated tent map with uniformly distributed binary symbolic histories.

The term *symbolic dynamics* records which branch of a dynamical system an orbit visits. This exercise shows that the 2^L affine pieces of the tent-map iterate correspond exactly to L-bit branch histories.

Let X be uniformly distributed on $[0,1]$ under the ambient probability measure $\mathbb{P}$. Define

$$B_t = \mathbb{I}\big(\tau^{\circ t}(X) \geq 1/2\big), \qquad t = 0, 1, \ldots.$$

Events involving dyadic endpoints may be assigned to either adjacent branch; these endpoints have probability zero.

(a) Prove directly from the two inverse branches of τ that

$$\mathbb{P}(\tau(X) \leq y) = y, \qquad 0 \leq y \leq 1.$$

Thus $\tau(X)$ is uniformly distributed on $[0,1]$.

(b) For $b = (b_0, \ldots, b_{L-1}) \in \{0,1\}^L$, define

$$C_b = \big\{x \in [0,1] : \mathbb{I}(\tau^{\circ t}(x) \geq 1/2) = b_t \text{ for } t = 0, \ldots, L-1\big\}.$$

Prove by induction on L that, up to endpoints, C_b is one interval of length 2^{-L}, and $\tau^{\circ L}$ maps that interval affinely onto $[0,1]$.

(c) Deduce that

$$\mathbb{P}(B_0 = b_0, \ldots, B_{L-1} = b_{L-1}) = 2^{-L} \qquad (b \in \{0,1\}^L).$$

Conclude that $B_0, \ldots, B_{L-1}$ are independent Bernoulli random variables with success probability $1/2$.

(d) Use part (a) inductively to identify the law of $\tau^{\circ t}(X)$ for every $t \geq 0$. Explain how parts (b)–(c) relate the 2^L affine pieces to symbolic histories rather than merely counting pieces.

Lookup, attention, and associative memory

Exercise 12: Compose two input-specified lookup tables.

Use the output of one associative lookup as the query for a second table.

Let

$$f : \{\mathtt{a}, \mathtt{b}\} \to \{1, 2\}, \qquad g : \{1, 2\} \to \{\mathtt{red}, \mathtt{blue}\}$$

be specified by

$$f(\texttt{a}) = 2, \quad f(\texttt{b}) = 1, \qquad g(1) = \texttt{red}, \quad g(2) = \texttt{blue}.$$

The input sequence contains the tagged records

$$((\texttt{F}, \texttt{a}, 2), (\texttt{F}, \texttt{b}, 1), (\texttt{G}, 1, \texttt{red}), (\texttt{G}, 2, \texttt{blue}), (\texttt{QUERY}, \texttt{b})).$$

In one lookup round, the final position may form a query from its current state, equality-select one earlier tagged record, retrieve that record's value, and update its state. All lookups must be causal.

(a) List the query, selected record, retrieved value, and updated final state in each of two rounds that return $g(f(\texttt{b}))$. Compute the final answer.

(b) Let $\mathcal{D}, \mathcal{V}, \mathcal{W}$ be finite sets, let $f : \mathcal{D} \to \mathcal{V}$ and $g : \mathcal{V} \to \mathcal{W}$, and suppose the input contains tagged records for both maps followed by a query $x \in \mathcal{D}$. Assume the keys in each table are distinct. Give one two-round lookup program and prove that its output is $g(f(x))$ for every choice of the two tables and every allowed set size.

(c) Explain why the final query position cannot in general return $g(f(x))$ in one round under the stated one-record-per-round abstraction. Identify one change to the input representation that would make one-round retrieval possible.

Exercise 13: Construct a two-round induction lookup.

Show how one attention round constructs predecessor–current records and a second round uses those records for content-based retrieval.

An induction head retrieves the token that followed an earlier occurrence of the current token. The construction below expresses that mechanism as two causal lookup rounds and makes the dependency between the rounds explicit.

Let $x_1, \ldots, x_n \in \mathcal{S}$, where $n \geq 3$. Assume that there is a unique index $j \in [n-2]$ such that $x_j = x_n$, and that no other index in $[n-1]$ contains x_n. Initially, position i stores x_i and its position index. A round may use a fixed positional query or an equality query, retrieve one causal source state, and combine the retrieved value with the destination's residual state.

(a) Give a first causal round after which every position $i \geq 2$ stores the record (x_{i-1}, x_i). State the source selected at position i and the residual information retained there.

(b) Give a second causal round at position n that uses x_n as a key and returns x_{j+1}. Identify the unique selected position.

(c) Prove that the two-round program returns the token following the unique earlier occurrence of x_n for every sequence satisfying the assumptions.

(d) Explain why the required key–value record is absent from the round-zero states. Under the stated one-source-per-round abstraction, identify the two dependent retrievals that prevent the same construction from being performed in one round.

Exercise 14: Attention as a modern Hopfield update.*

Derive the energy update underlying softmax attention and quantify its approach to hard associative lookup.

Earlier exercises analyze how score ranges control softmax concentration and retrieval error. The new result here is the energy interpretation: when keys are also values, one softmax-attention update is one gradient step for a modern Hopfield energy.

Let $M \geq 2$, let $k_1, \ldots, k_M \in \mathbb{R}^d$, let $\beta > 0$, and define

$$\alpha_i(q) = \frac{e^{\beta k_i^\top q}}{\sum_{j=1}^M e^{\beta k_j^\top q}}, \qquad E_\beta(q) = \frac{1}{2}\|q\|_2^2 - \frac{1}{\beta}\log\left(\sum_{i=1}^M e^{\beta k_i^\top q}\right).$$

(a) Differentiate E_β and prove

$$\nabla E_\beta(q) = q - \sum_{i=1}^M \alpha_i(q) k_i.$$

(b) Show that one gradient step of length one gives

$$q^+ = q - \nabla E_\beta(q) = \sum_{i=1}^M \alpha_i(q) k_i.$$

Identify the queries, keys, and values in the corresponding one-query attention operation. Explain why a general attention output $\sum_i \alpha_i(q) v_i$ with arbitrary v_i need not be a gradient step in the query space $\mathbb{R}^d$.

(c) Suppose $i^\star$ uniquely maximizes $k_i^\top q$, define

$$\Delta = k_{i^\star}^\top q - \max_{i \neq i^\star} k_i^\top q > 0,$$

and assume $\|k_i\|_2 \leq B$ for every i. You may use

$$1 - \alpha_{i^\star}(q) \leq (M-1)e^{-\beta\Delta}.$$

Prove

$$\|q^+ - k_{i^\star}\|_2 \leq 2B(M-1)e^{-\beta\Delta}.$$

For $0 < \varepsilon < 2B(M-1)$, give a sufficient lower bound on β for this error to be at most ε.

(d) Explain which limit produces hard associative lookup and identify the roles of the score margin Δ, the number of memories M, and the norm bound B in the quantitative approximation.



Exercise 15: When does a reusable jump table reduce the total cost?*

Count the preprocessing and query operations that the asymptotic depth comparison suppresses.

Let $p : \mathcal{D} \to \mathcal{D}$ be a self-map on a set of size M. Suppose full-attention preprocessing has constructed

$$q_\ell(u) = p^{2^\ell}(u), \qquad \ell = 0, \ldots, J,$$

at every map-record position. Constructing each of $q_1, \ldots, q_J$ costs M table updates. Count one application of a stored table entry to one key as one table lookup, and count the construction of one new $q_\ell(u)$ value as one table update.

(a) Let $K = \sum_{\ell=0}^J b_\ell 2^\ell$, where $b_\ell \in \{0, 1\}$. Prove that applying the maps q_ℓ for which $b_\ell = 1$ returns $p^K(x)$ in any order. Deduce that the query uses $\sum_{\ell=0}^J b_\ell$ lookups.

(b) Consider the exhaustive batch containing one query for every $K \in \{0, \ldots, 2^{J+1} - 1\}$. Compute exactly the total number of lookups used by direct iteration and by the stored jump tables. Your answer should include the MJ preprocessing updates in the jump-table total.

(c) Suppose instead that the same input map must answer Q queries, each with iteration count $K_{\max} = 2^{J+1} - 1$ but possibly with a different initial state. Compute the total jump-table cost and the total direct-iteration cost. State an inequality on $M, Q, K_{\max}$ under which the jump tables use fewer total operations.

(d) Compare the exhaustive batch in part (b) with the maximum-length batch in part (c). Explain why a sequential-depth advantage does not by itself imply a reduction in the total number of table operations.



Exercise 16: Characterize causally orderable self-maps.[*]

◎ Characterize exactly when every pointer update can retrieve from an earlier record in one causal ordering.

The pointer-doubling construction requires an ordering in which every map record points backward or to itself. This exercise identifies exactly which finite self-maps admit such an ordering.

Let $p : \mathcal{D} \to \mathcal{D}$ be a self-map on a finite set. Its functional directed graph has one edge $u \to p(u)$ from every vertex. A *causal ordering* is a bijection $\mathrm{pos} : \mathcal{D} \to [|\mathcal{D}|]$ satisfying

$$\mathrm{pos}(p(u)) \le \mathrm{pos}(u) \qquad (u \in \mathcal{D}).$$

(a) Prove that if the functional graph contains a directed cycle of length greater than one, then no causal ordering exists.

(b) Suppose every directed cycle is a self-loop. Prove that every orbit eventually reaches a unique fixed point in its connected component. Define the depth of a vertex to be the number of map applications needed to reach that fixed point. Construct a causal ordering and prove that it satisfies the required inequality.

(c) Conclude that a causal ordering exists if and only if every directed cycle is a self-loop.

(d) Prove that a causal ordering also satisfies

$$\mathrm{pos}(p^k(u)) \le \mathrm{pos}(u) \qquad (u \in \mathcal{D},\ k \ge 0).$$

Explain why this inequality makes every lookup in causal pointer-doubling preprocessing available at or before the requesting record.

References

Kathleen T. Alligood, Tim D. Sauer, and James A. Yorke. *Chaos: An Introduction to Dynamical Systems*. Springer, 1997. https://doi.org/10.1007/978-3-642-59281-2.

Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936. https://doi.org/10.2307/2371045.

András Hajnal, Wolfgang Maass, Pavel Pudlák, Mário Szegedy, and György Turán. Threshold circuits of bounded depth. *Journal of Computer and System Sciences*, 46(2):129–154, 1993. https://doi.org/10.1016/0022-0000(93)90001-D.

Johan Håstad. Almost optimal lower bounds for small depth circuits. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, pages 6–20, 1986. https://doi.org/10.1145/12130.12132.

Johan Håstad, Benjamin Rossman, Rocco A. Servedio, and Li-Yang Tan. An average-case depth hierarchy theorem for Boolean circuits. *Journal of the ACM*, 64(5), article 35, 2017. https://doi.org/10.1145/3095799.

Stevan Harnad. The symbol grounding problem. *Physica D: Nonlinear Phenomena*, 42(1–3):335–346, 1990. https://doi.org/10.1016/0167-2789(90)90087-6.

Daniel M. Kane and Ryan Williams. Super-linear gate and super-quadratic wire lower bounds for depth-two and depth-three threshold circuits. In *Proceedings of the 48th Annual ACM Symposium on Theory of Computing*, pages 633–643, 2016. https://doi.org/10.1145/2897518.2897636.

Neil Krishnan and Elchanan Mossel. Depth lower bounds for ReLU networks with binary inputs. arXiv:2606.18540, 2026. https://arxiv.org/abs/2606.18540.

Tien-Yien Li and James A. Yorke. Period Three Implies Chaos. *American Mathematical Monthly*, 82(10):985–992, 1975. https://doi.org/10.1080/00029890.1975.11994008.

Allen Newell and Herbert A. Simon. Computer science as empirical inquiry: Symbols and search. *Communications of the ACM*, 19(3):113–126, 1976. https://doi.org/10.1145/360018.360022.

Catherine Olsson et al. In-context learning and induction heads. arXiv:2209.11895, 2022. https://arxiv.org/abs/2209.11895.

Jorge Pérez, Pablo Barceló, and Javier Marinkovic. Attention is Turing-complete. *Journal of Machine Learning Research*, 22(75):1–35, 2021. https://www.jmlr.org/papers/v22/20-302.html.

Emil L. Post. *The Two-Valued Iterative Systems of Mathematical Logic*. Annals of Mathematics Studies 5. Princeton University Press, 1941.

Hubert Ramsauer et al. Hopfield networks is all you need. In *International Conference on Learning Representations*, 2021. https://openreview.net/forum?id=tL89RnzIiCd.

Clayton Sanford, Daniel Hsu, and Matus Telgarsky. Transformers, parallel computation, and logarithmic depth. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 43276–43327, 2024. https://proceedings.mlr.press/v235/sanford24a.html.

Hava T. Siegelmann and Eduardo D. Sontag. On the computational power of neural nets. *Journal of Computer and System Sciences*, 50(1):132–150, 1995. https://doi.org/10.1006/jcss.1995.1013.

Claude E. Shannon. The synthesis of two-terminal switching circuits. *Bell System Technical Journal*, 28(1):59–98, 1949. https://doi.org/10.1002/j.1538-7305.1949.tb03624.x.

Matus Telgarsky. Benefits of depth in neural networks. In *Proceedings of the 29th Conference on Learning Theory*, volume 49 of *Proceedings of Machine Learning Research*, pages 1517–1539, 2016. https://proceedings.mlr.press/v49/telgarsky16.html.

Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937. https://doi.org/10.1112/plms/s2-42.1.230.

Gail Weiss, Yoav Goldberg, and Eran Yahav. Thinking like transformers. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 11080–11090, 2021. https://proceedings.mlr.press/v139/weiss21a.html.

Ruize Xu, Chenxiao Yang, Yanhong Li, and David McAllester. Training transformers as a universal computer. arXiv:2604.25166, 2026. https://arxiv.org/abs/2604.25166.

Hattie Zhou, Arwen Bradley, Etai Littwin, Noam Razin, Omid Saremi, Josh Susskind, Samy Bengio, and Preetum Nakkiran. What algorithms can transformers learn? A study in length generalization. In *The Twelfth International Conference on Learning Representations*, 2024. https://proceedings.iclr.cc/paper_files/paper/2024/hash/45ed1a72597594c097152ef9cc187762-Abstract-Conference.html.