

A Building Blocks

A.1 Residual Stream Storage Interface

Our masked pre-norm transformer architecture always normalizes values when reading them from the residual stream. This means that it's not always the case that what's added to the residual stream by one layer is accessible as-is in future layers, which can be problematic if there is a need to "erase" that value. We discuss how values are stored and, if needed, erased from the stream.

For any general scalar z , storing z in the residual stream results in $\text{sgn}(z)$ being retrieved when masked pre-norm is applied to this cell. This will be useful when we want to collapse multiple values or perform equality or threshold checks. As a special case, when $z \in \{-1, 0, 1\}$, the retrieved value after masked pre-norm is precisely z . Thus scalars in $\{-1, 0, 1\}$ can be stored and retrieved without any information loss.

In order to retrieve a value z with masked pre-norm (rather than just its sign), we can instead represent z as a 4-dimensional vector $\psi(z) = \langle z, 1, -z, -1 \rangle$. Then, pre-norm masked to only this vector will return $\phi(z) = \psi(z) / \sqrt{z^2 + 1}$. Scalars z stored as $\psi(z)$ or $\phi(z)$ in the residual stream can be trivially retrieved as $\phi(z)$ by masked pre-norm:

Lemma 2. *There exists a masked pre-norm ν such that, if $\phi(z)$ or $\psi(z)$ is stored in \mathbf{h} , $\nu(\mathbf{h}) = \phi(z)$.*

Furthermore, a single masked pre-norm can even be used to retrieve multiple scalars stored in the residual stream. Since $\phi(z)$ is a unit-norm vector, this is a consequence of the following lemma:

Lemma 3. *There exists a masked pre-norm ν such that, if \mathbf{h} stores unit-norm vectors ϕ_1, \dots, ϕ_k , then $\nu(\mathbf{h}) = \langle \phi_1, \dots, \phi_k \rangle$.*

Proof. We apply the mask to focus on the positions where ϕ_1, \dots, ϕ_k are stored. Then, the masked pre-norm outputs

$$\frac{1}{\sqrt{2k}} \langle \phi_1, \dots, \phi_k \rangle.$$

We can hardcode the scalar multiplier of layer-norm to remove the scalar factor, or, equivalently, bake it into the next linear transformation. Either way, we are able to retrieve the concatenation of ϕ_1, \dots, ϕ_k as input to the layer. \square

The following establishes that we can compute numerical values z with attention heads and make them accessible as $\phi(z)$ in later layers:

Lemma 4. *Let z be a scalar computable by an attention head from residual stream \mathbf{h} . There exist two layers producing residual streams \mathbf{h}' , \mathbf{h}'' such that*

1. $\phi(z)$ can be read via masked pre-norm from \mathbf{h}' or \mathbf{h}'' .
2. $\phi(z)$ is stored in \mathbf{h}'' at (formerly blank) indices I .

Proof. The first layer computes z and stores $\psi(z)$ at blank indices I in the residual stream, producing \mathbf{h}' . Thus, the second layer can read $\phi(z)$ with masked pre-norm via Lemma 2 and can also recompute z from \mathbf{h} , which is a subspace of \mathbf{h}' . At this point, it outputs $-\psi(z) + \phi(z)$ at indices I , which leads to \mathbf{h}'' storing $\phi(z)$ at I . \square

A.2 Clearing Stored Values

In the repeated layers of a universal transformer, we will need to overwrite the values stored at particular indices in the residual stream. That is, if $[\mathbf{h}]_I = \mathbf{x}$, it will be useful to produce \mathbf{h}' such that $[\mathbf{h}']_I = \mathbf{y}$ instead. The following lemmas will help implement constructions of this form.

Lemma 5. *If a unit-norm vector ϕ is stored in \mathbf{h} at I , there exists a feedforward sub-layer that removes ϕ , i.e., produces \mathbf{h}' such that $[\mathbf{h}']_i = \vec{0}$.*

Proof. The layer reads ϕ via masked pre-norm and writes $-\phi$ to \mathbf{h} at I , setting $[\mathbf{h}]_I = \phi - \phi = \vec{0}$. \square

Combining Lemma 5 with a parallel layer that stores some new value at I , we see that we can effectively *overwrite* values at I rather than just deleting them.

It is also possible to remove information that is not a unit-norm vector, although the construction is less direct.

Lemma 6. *Let δ be the output of a transformer layer on \mathbf{h} , targeted to indices I at which \mathbf{h} is blank. Then there exists another transformer layer that resets the residual stream $\mathbf{h}' = \mathbf{h} + \delta$ to \mathbf{h} .*

Proof. The second layer is a copy of the initial layer that considers the subvector \mathbf{h} of \mathbf{h}' as its input and where all signs are flipped. Thus, it outputs $-\delta$, which guarantees that the final residual stream is $\mathbf{h}'' = \mathbf{h} + \delta - \delta = \mathbf{h}$. \square

A.3 Computing Position Offsets

It will be useful to show how a transformer can compute the position index of the previous token.

Lemma 7. *Assume a transformer stores $\mathbb{1}[i = 0]$ and $\mathbb{1}[i < k]$ in the residual stream. Then, with 1 layer, it is possible to add $\phi(i - k)$ in the residual stream at indices $i \geq k$.*

Proof. We construct two attention heads. The first is uniform with value $\mathbb{1}[j = 0]$, and thus computes $1/i$. The second is uniform with value $\mathbb{1}[j \geq k]$, and thus computes $(i - k)/i$. We then use a feedforward layer to compute $\phi((i - k)/i, 1/i) = \phi(i - k)$ and store it in the residual stream. \square

The precondition that we can identify the initial token (cf. Merrill and Sabharwal, 2024) is easy to meet with any natural representation of position, including $1/i$ or $\phi(i)$, as we can simply compare the position representation against some constant.

We assume that the positional encodings used by the model allow detecting the initial token (Merrill and Sabharwal, 2024). One way to enable this would simply be to add a beginning-of-sequence token, although most position embeddings should also enable it directly.

A.4 Equality Checks

We show how to perform an equality check between two scalars and store the output as a boolean.

Lemma 8. *Given two scalars x, y computable by attention heads or stored in the residual stream, we can use a single transformer layer to write $\mathbb{1}[x = y]$ in the residual stream. Furthermore, a second layer can be used to clear all intermediate values.*

Proof. After computing x, y in a self-attention layer, we write $x - y$ to a temporary cell in the residual stream. The feedforward sublayer reads $\sigma_1 = \text{sgn}(x - y)$, computes $z = 1 - \text{ReLU}(\sigma_1) - \text{ReLU}(-\sigma_1)$, and writes z to the residual stream.

The next transformer layer then recomputes $y - x$ and adds it to the intermediate memory cell, which sets it back to 0. Thus, the output is correct and intermediate memory is cleared. \square

B Division Construction Correctness

The proof of Lemma 1 presents the full construction to implement division in a transformer. For space, we omitted a full proof of correctness for the construction, which we now present.

Proof of Correctness. In the first layer, suppose first that i is a multiple of m . In this case, there exists a position $j^* \leq i$ such that $i = mj^*$, which means the query $q_i = \phi(i/m) = \phi(j^*)$ exactly matches the key k_{j^*} . The head will thus return $v_{j^*} = \phi(j^*) = \phi(i/m)$, representing precisely the quotient i/m . Further, the equality check will pass, making $e_i = 1$. The layer thus behaves as intended when i is a multiple of m . On the other hand, when i is *not* a multiple of m , no such j^* exists. The head will instead attend to some j for which $i \neq mj$ and therefore $\phi(i/m) \neq \phi(j)$, making the subsequent equality check fail and setting $e_i = 0$, as intended.

For the second layer, let h_i^2 denote the head's output at position i . By construction, $q_i \cdot k_j = e_j - [\phi(j)]_0$ where $[\phi(j)]_0 = j/\sqrt{2j^2 + 2}$ is the first coordinate of $\phi(j)$. Note that $[\phi(j)]_0 \in [0, 1]$

for positions $j \leq i$ and that it is monotonically increasing in j . It follows that the dot product is maximized at the largest $j \leq i$ such that $e_j = 1$, i.e., at the largest $j \leq i$ that is a multiple of m . This j has the property that $\lfloor i/m \rfloor = j/m$. Thus, the head at this layer attends solely to this j . Recall that the value v_j at this position is $h_j^1 = \phi(j/m)$, which thus equals $\phi(\lfloor i/m \rfloor)$. The head's output, h_i^2 , is therefore $\phi(\lfloor i/m \rfloor)$, as intended, which is stored in the residual stream.

The correctness of the third and fourth layer is easy to verify.

In the fifth layer, position i attends with query $q_i = \langle \phi(\lfloor i/m \rfloor), 1 \rangle$, key $k_j = \langle \phi(\lfloor j/m \rfloor), b_j^1 \rangle$, and value $v_j = 1 - b_j^2$; each b_j^k for $k \in \{1, 2\}$ can be retrieved from the residual stream. The query-key product achieves its upper bound of 2 exactly when two conditions hold: $\lfloor i/m \rfloor = \lfloor j/m \rfloor$ and $b_j^1 = 1$. Thus, the head attends from i to all $j \leq i$ that store the same quotient as i and also have $b_j^1 = 1$. To make this clearer, let's write i as $i = b'm + c'$ for some $c' < m$. The query-key dot product is then maximized precisely at the c' positions j in $\{b'm + 1, b'm + 2, \dots, b'm + c'\}$, for all of which $\lfloor j/m \rfloor = \lfloor i/m \rfloor = b'$; note that $b'm$ is *not* included in this list as $b_j^1 = 0$ when $j = b'm$. Of these positions, only $b'm + 1$ has the property that the quotient there is *not* the same as the quotient two position earlier, as captured by the value $v_j = 1 - b_j^2$. Thus, the value v_j is 1 among these positions only at $j = b'm + 1$, and 0 elsewhere.

Assuming m does not divide i , $c' > 0$ and the head attends uniformly at c' positions, returning $1/c'$ as the head output. By construction, $c' = i - b'm = i \bmod m$. The layer adds the vector $\psi(1, 1/c')$ defined as $\langle 1, 1/c', -1, -1/c' \rangle$ to the residual stream at position i . This, when read in the next layer using masked pre-norm, will yield $\phi(1, 1/c') = \phi(c') = \phi(i \bmod m)$.

On the other hand, if m does divide i (which can be checked with a separate, parallel head), we write $\psi(0)$ to the residual stream, which, when read by the next layer, will yield $\phi(0) = \phi(i \bmod m)$.

The sixth layer attends with query $q_i = \phi(a_i)$, key $k_j = \phi(j)$, and value $v_j = \langle \phi(\lfloor j/m \rfloor), \phi(j \bmod m) \rangle$, where both components of the value have been previously computed and stored in the residual stream in layers two and five. Since $a_i \leq i$, the query matches the key at exactly one position $j = a_i$, and the head retrieves $\langle \phi(\lfloor a_i/m \rfloor), \phi(a_i \bmod m) \rangle$, which is precisely $\langle \phi(b_i), \phi(c_i) \rangle$. The layer can thus store $\phi(b_i)$ and $\phi(c_i)$ to the residual stream at position i .

That the seventh "cleanup" layer operates as desired is easy to see from the construction. \square

C Regular Language Recognition Proof

Theorem 1 (Regular Language Recognition). *Let L be a regular language over Σ and $\$ \notin \Sigma$. Then there exists a $(0, 8, 9)$ -universal transformer with causal masking that, on any string $w\$$, recognizes whether $w \in L$ when unrolled to $\lceil \log_2 |w| \rceil$ depth.*

Proof. Regular language recognition can be framed as multiplying a sequence of elements in the automaton's transition monoid (Myhill, 1957; Thérien, 1981). It thus suffices to show how n elements in a finite monoid can be multiplied with $\Theta(\log n)$ depth. We show how a log-depth universal transformer can implement the standard binary tree construction (Barrington and Thérien, 1988; Liu et al., 2023; Merrill et al., 2024) where each level multiplies two items, meaning the overall depth is $\Theta(\log |w|)$. We will represent a tree over the input tokens within the transformer. Each level of the tree will take 8 transformer layers. We define a notion of active tokens: at level 0, all tokens are active, and, at level ℓ , tokens at $t \cdot 2^\ell - 1$ for any t will remain active, and all other tokens will be marked as inactive. As an invariant, active token $i = t \cdot 2^\ell - 1$ in level ℓ will store a unit-norm vector δ_i^ℓ that represents the cumulative product of tokens from $i - 2^\ell + 1$ to i .

We now proceed by induction over ℓ , defining the behavior of non- $\$$ tokens at layers that make up level ℓ . The current group element δ_i^ℓ stored at active token i is, by inductive assumption, the cumulative product from $i - 2^\ell + 1$ to i . Let α_i^ℓ denote that token i is active. By Lemma 7 we use a layer to store $i - 1$ at token i . The next layer attends with query $\phi(i - 1)$, key $\phi(j)$, and value δ_j^ℓ to retrieve δ_{i-1}^ℓ , the group element stored at the previous token. Finally, another layer attends with query $\vec{1}$, key $\langle \phi(j)_1, \alpha_i^\ell \rangle$, and value δ_{j-1}^ℓ to retrieve the group element δ_{j*}^ℓ stored at the previous active token, which represents the cumulative product from $i - 2 \cdot 2^\ell + 1$ to $i - 2^\ell$. Next, we will use two layers to update $\delta_i^\ell \leftarrow \delta_i^{\ell+1}$ and $\delta_j^\ell \leftarrow \vec{0}$, which is achieved as follows. First, we assert

917 there exists a single feedforward layer that uses a table lookup to compute $\delta_{j*}^\ell, \delta_i^\ell \mapsto d$ such that
 918 $d/\|d\| = \delta_{j*}^\ell \cdot \delta_i^\ell = \delta_i^{\ell+1}$. Next, we invoke Lemma 6 to construct a layer that adds d to an empty cell
 919 of the residual stream and then another layer that deletes it. This second layer can now read both
 920 $\delta_i^\ell, \delta_{j*}^\ell$ and $\delta_i^{\ell+1}$ (from d) as input, and we modify it to add $\delta_i^{\ell+1} - \delta_i^\ell$ to δ_i^ℓ , changing its value to
 921 $\delta_i^{\ell+1}$. Similarly, we modify it to add $-\delta_{j*}^\ell$ to δ_{j*}^ℓ to set it to 0. A feedforward network then subtracts
 922 δ_i^ℓ from the residual stream and adds $\delta_i^\ell \cdot \delta_{j*}^\ell$. This requires at most 4 layers.

923 To determine activeness in layer $\ell + 1$, each token i attends to its left to compute c_i/i , where c_i is the
 924 prefix count of active tokens, inclusive of the current token. We then compute $\phi(c_i/i, 1/i) = \phi(c_i)$
 925 and store c_i temporarily in the residual stream. At this point, we use Lemma 1 to construct 7 layers
 926 that compute $c_i \bmod 2$ with no storage overhead. The current token is marked as active in layer $\ell + 1$
 927 iff $c_i = 0 \bmod 2$, which is equivalent to checking whether $i = t \cdot 2^\ell - 1$ for some t . In addition to
 928 updating the new activeness $\alpha_i^{\ell+1}$, we also persist store the previous activeness α_i^ℓ in a separate cell
 929 of the residual stream and clear c_i . This requires at most 8 layers.

930 Finally, we describe how to aggregate the cumulative product at the $\$$ token, which happens in parallel
 931 to the behavior at other tokens. Let $\delta_\$^\ell$ be a monoid element stored at $\$$ that is initialized to the identity
 932 and will be updated at each layer. Using the previously stored value $i - 1$, we can use a layer to
 933 compute and store α_{i-1}^ℓ and $\alpha_{i-1}^{\ell+1}$ at each i . A head then attends with query $\vec{1}$, key $\langle \phi(j)_1, 10 \cdot \alpha_{i-1}^\ell \rangle$,
 934 and value $\langle (1 - \alpha_{j-1}^{\ell+1}) \cdot \delta_{j-1}^{\ell+1} \rangle$. This retrieves a value from the previous active token j at level
 935 ℓ that is δ_j^ℓ if j will become inactive at $\ell + 1$ and $\vec{0}$ otherwise. If δ_j^ℓ is retrieved, a feedforward
 936 network subtracts $\delta_\$^\ell$ from the residual stream and adds $\delta_j^\ell \cdot \delta_\$^\ell$. This guarantees that whenever a tree
 937 is deactivated, its cumulative product is incorporated into $\delta_\$^\ell$. Thus, after $\ell = \lceil \log_2 |w| \rceil + 1$ levels,
 938 $\delta_\$^\ell$ will be the transition monoid element for w . We can use one additional layer to check whether
 939 this monoid element maps the initial state to an accepting state using a finite lookup table. Overall,
 940 this can be expressed with 8 layers repeated $\lceil \log_2 |w| \rceil$ times and 9 final layers (to implement the
 941 additional step beyond $\lceil \log n \rceil$). \square

942 D Graph Connectivity Proof

943 **Theorem 2** (Graph Connectivity). *There exists a $(17, 2, 1)$ -universal transformer T with both*
 944 *causal and unmasked heads that, when unrolled $\lceil \log_2 n \rceil$ times, solves connectivity on (directed or*
 945 *undirected) graphs over n vertices: given the $n \times n$ adjacency matrix of a graph G , n^3 padding*
 946 *tokens, and $s, t \in \{1, \dots, n\}$ in unary, T checks whether G has a path from vertex s to vertex t .*

947 *Proof.* We will prove this for directed graphs, as an undirected edge between two vertices can be
 948 equivalently represented as two directed edges between those vertices. Let G be a directed graph over
 949 n vertices. Let $A \in \{0, 1\}^{n \times n}$ be G 's adjacency matrix: for $i, j \in \{1, \dots, n\}$, $A_{i,j}$ is 1 if G has an
 950 edge from i to j , and 0 otherwise.

951 The idea is to use the first n^2 tokens of the transformer to construct binary predicates $B_\ell(i, j)$ for
 952 $\ell \in \{0, 1, \dots, \lceil \log n \rceil\}$ capturing whether G has a path of length at most 2^ℓ from i to j . To this
 953 end, the transformer will use the n^3 padding tokens to also construct intermediate ternary predicates
 954 $C_\ell(i, k, j)$ for $\ell \in \{1, \dots, \lceil \log n \rceil\}$ capturing whether G has paths of length at most $2^{\ell-1}$ from i to
 955 k and from k to j . These two series of predicates are computed from each other iteratively, as in
 956 standard algorithms for graph connectivity:

$$B_0(i, j) \iff A(i, j) \vee i = j \quad (4)$$

$$C_{\ell+1}(i, k, j) \iff B_\ell(i, k) \wedge B_\ell(k, j) \quad (5)$$

$$B_{\ell+1}(i, j) \iff \exists k \text{ s.t. } C_{\ell+1}(i, k, j) \quad (6)$$

957 We first argue that $B_{\lceil \log n \rceil}(i, j) = 1$ if and only if G has a path from i to j . Clearly, there is such
 958 a path if and only if there is a “simple path” of length at most n from i to j . To this end, we argue
 959 by induction over ℓ that $B_\ell(i, j) = 1$ if and only if G has a path of length at most 2^ℓ from i to j . For
 960 the base case of $\ell = 0$, by construction, $B_0(i, j) = 1$ if and only if either $i = j$ (which we treat as a
 961 path of length 0) or $A_{i,j} = 1$ (i.e., there is a direct edge from i to j). Thus, $B_\ell(i, j) = 1$ if and only
 962 if there is a path of length at most $2^0 = 1$ from i to j . Now suppose the claim holds for $B_\ell(i, j)$. By

construction, $C_{\ell+1}(i, k, j) = 1$ if and only if $B_\ell(i, k) = B_\ell(k, j) = 1$, which by induction means there are paths of length at most 2^ℓ from i to k and from k to j , which in turn implies that there is a path of length at most $2 \cdot 2^\ell = 2^{\ell+1}$ from i to j (through k). Furthermore, note conversely that if there is a path of length at most $2^{\ell+1}$ from i to j , then there must exist a “mid-point” k in this path such that there are paths of length at most 2^ℓ from i to k and from k to j , i.e., $C_{\ell+1}(i, k, j) = 1$ for some k . This is precisely what the definition of $B_{\ell+1}(i, j)$ captures: it is 1 if and only if there exists a k such that $C_{\ell+1}(i, k, j) = 1$, which, as argued above, holds if and only if there is a path of length at most $2^{\ell+1}$ from i to j . This completes the inductive step.

The crucial part is to construct a transformer that correctly operationalizes the computation of predicates B_ℓ and C_ℓ . The input to the transformer is the adjacency matrix A represented using n^2 tokens from $\{0, 1\}$, followed by n^3 padding tokens \square , and finally the source and target nodes $s, t \in \{1, \dots, n\}$ represented in unary notation using special tokens a and b :

$$A_{1,1} \dots A_{1,n} A_{2,1} \dots A_{2,n} \dots A_{n,1} \dots A_{n,n} \underbrace{\square \dots \square}_{n^3} \underbrace{a \dots a}_s \underbrace{b \dots b}_t$$

Let $N = n^2 + n^3 + s + t$, the length of the input to the transformer. The first n^2 token positions will be used to compute predicates B_ℓ , while the next n^3 token positions will be used for predicates C_ℓ .

Initial Layers. The transformer starts off by using layer 1 to store $1/N, n, n^2, s$, and t in the residual stream at every position, as follows. The layer uses one head with uniform attention and with value 1 only at the first token (recall that the position embedding is assumed to separate 1 from other positions). This head computes $1/N$ and the layer adds $\psi(1/N)$ to the residual stream. Note that the input tokens in the first set of n^2 positions, namely 0 and 1, are distinct from tokens in the rest of the input. The layer, at every position, uses a second head with uniform attention, and with value 1 at tokens in $\{0, 1\}$ and value 0 at all other tokens. This head computes n^2/N . The layer now adds $\psi(n^2/N, 1/N)$, where $\psi(a, b)$ is defined as the (unnormalized) vector $\langle a, b, -a, -b \rangle$. When these coordinates are later read from the residual stream via masked pre-norm, they will get normalized and one would obtain $\phi(n^2/N, 1/N) = \phi(n^2)$. Thus, future layers will have access to $\phi(n^2)$ through the residual stream. The layer similarly uses three additional heads to compute $n^3/N, s/N$, and t/N . From the latter two values, it computes $\psi(s/N, 1/N)$ and $\psi(t/N, 1/N)$ and adds them to the residual stream; as discussed above, these can be read in future layers as $\phi(s/N, 1/N) = \phi(s)$ and $\phi(t/N, 1/N) = \phi(t)$. Finally, the layer computes $\psi(n^3/N, n^2/N)$ and adds it to the residual stream. Again, this will be available to future layers as $\phi(n^3/N, n^2/N) = \phi(n)$.

The transformer uses the next 15 layers to compute and store in the residual stream the semantic “coordinates” of each of the first $n^2 + n^3$ token position as follows. For each of the first n^2 positions $p = in + j$ with $1 \leq p \leq n^2$, it uses Lemma 1 (7 layers) with a_i set to p and m set n in order to add $\phi(i)$ and $\phi(j)$ to the residual stream at position p . In parallel, for each of the next n^3 positions $p = n^2 + (in^2 + kn + j)$ with $n^2 + 1 \leq p \leq n^2 + n^3$, it uses Lemma 1 with a_i set to p and m set n in order to add $\phi((i+1)n + k)$ and $\phi(j)$ to the residual stream. It then uses the lemma again (7 more layers), this time with a_i set to $(i+1)n + k$ and m again set to n , to add $\phi(i+1)$ and $\phi(k)$ to the residual stream. Lastly, it uses Lemma 7 applied to $\phi(i+1)$ to add $\phi(i)$ to the residual stream.

Layer 17 of the transformer computes the predicate $B_0(i, j)$ at the first n^2 token positions as follows. At position $p = in + j$, it uses Lemma 8 to compute $\mathbb{I}(\phi(A(i, j)) = \phi(1))$ and $\mathbb{I}(\phi(i) = \phi(j))$; note that $\phi(A(i, j))$, $\phi(i)$, and $\phi(j)$ are available in the residual stream at position p . It then uses a feedforward layer to output 1 if both of these are 1, and output 0 otherwise. This is precisely the intended value of $B_0(i, j)$. The sublayer then adds $B_0(i, j)$ to the residual stream. The layer also adds to the residual stream the value 1, which will be used to initialize the boolean that controls layer alternation in the repeated layers as discussed next.

Repeating Layers. The next set of layers alternates between computing the C_ℓ and the B_ℓ predicates for $\ell \in \{1, \dots, \lceil \log n \rceil\}$. To implement this, each position i at layer updates in the residual stream the value of a single boolean r computed as follows. r is initially set to 1 at layer 8. Each repeating layer retrieves r from the residual stream and adds $1 - r$ to the same coordinate in the residual stream. The net effect is that the value of r alternates between 1 and 0 at the repeating layers. The transformer uses this to alternate between the computation of the C_ℓ and the B_ℓ predicates.

For $\ell \in \{1, \dots, \lceil \log n \rceil\}$, layer $(2\ell - 1) + 8$ of the transformer computes the predicate $C_\ell(i, k, j)$ at the set of n^3 (padding) positions $p = n^2 + in^2 + kn + j$, as follows. It uses two heads, one with query $\langle \phi(i), \phi(k) \rangle$ and the other with query $\langle \phi(k), \phi(j) \rangle$. The keys in the first n^2 positions $q = i'n + j'$ are set to $\langle \phi(i'), \phi(j') \rangle$, and the values are set to $B_{\ell-1}(i', j')$. The two heads thus attend solely to positions with coordinates (i, k) and (k, j) , respectively, and retrieve boolean values $B_{\ell-1}(i, k)$ and $B_{\ell-1}(k, j)$, respectively, stored there in the previous layer. The layer then uses Lemma 8 to compute $\mathbb{I}(B_{\ell-1}(i, k) = 1) \text{ and } \mathbb{I}(B_{\ell-1}(k, j) = 1)$, and uses a feedforward layer to output 1 if both of these checks pass, and output 0 otherwise. This is precisely the intended value of $C_\ell(i, k, j)$. If $\ell > 1$, the layer replaces the value $C_{\ell-1}(i, k, j)$ stored previously in the residual stream with the new boolean value $C_\ell(i, k, j)$ by adding $C_\ell(i, k, j) - C_{\ell-1}(i, k, j)$ to the same coordinates of the residual stream. If $\ell = 1$, it simply adds $C_\ell(i, k, j)$ to the residual stream.

For $\ell \in \{1, \dots, \lceil \log n \rceil\}$, layer $2\ell + 8$ computes the predicate $B_\ell(i, j)$ at the first n^2 positions $p = in + j$, as follows. It uses a head with query $\langle \phi(i), \phi(j) \rangle$. The keys in the second set of n^3 positions $q = n^2 + i'n^2 + k'n + j'$ are set to $\langle \phi(i'), \phi(j') \rangle$ (recall that $\phi(i')$ and $\phi(j')$ are available in the residual stream at q) and the corresponding values are set to the boolean $C_\ell(i', k', j')$, stored previously in the residual stream. The head thus attends uniformly to the n padding positions that have coordinates (i, k', j) for various choices of k' . It computes the average of their values, which equals $h = \frac{1}{n} \sum_{k'=1}^n C_\ell(i, k', j)$ as well as $1/(2n)$ using an additional head. We observe that $h \geq 1/n$ if there exists a k' such that $C_\ell(i, k', j) = 1$, and $h = 0$ otherwise. These conditions correspond precisely to $B_\ell(i, j)$ being 1 and 0, respectively. We compute $h - 1/(2n)$ and store it in the residual stream. Similar to the proof of Lemma 8, the feedforward layer reads $\sigma = \text{sgn}(h - 1/(2n))$, computes $z = (1 + \text{ReLU}(\sigma))/2$, and writes z to the residual stream. The value z is precisely the desired $B_\ell(i, j)$ as σ is 1 when $h \geq 1/n$ and 0 when $h = 0$. As in Lemma 8, the intermediate value $h - 1/(2n)$ written to the residual stream can be recomputed and reset in the next layer. As before, the transformer replaces the value $B_{\ell-1}(i, j)$ stored previously in the residual stream with the newly computed value $B_\ell(i, j)$ by adding $B_\ell(i, j) - B_{\ell-1}(i, j)$ to the stream at the same coordinates.

Final Layers. Finally, in layer $2\lceil \log n \rceil + 18$, the final token uses a head that attends with query $\langle \phi(s), \phi(t) \rangle$ corresponding to the source and target nodes s and t mentioned in the input; recall that $\phi(s)$ and $\phi(t)$ are available in the residual stream. The keys in the first n^2 positions $p = in + j$ are, as before, set to $\langle \phi(i), \phi(j) \rangle$, and the values are set to $B_{\lceil \log n \rceil}(i, j)$ retrieved from the residual stream. The head thus attends solely to the position with coordinates (s, t) , and retrieves and outputs the value $B_{\lceil \log n \rceil}(s, t)$. This value, as argued earlier, is 1 if and only if G has a path from s to t . \square

E Proofs for Width Scaling and Chain of Thought Claims

Theorem 3 (Width Scaling). *Let T be a fixed-depth transformer whose width (model dimension or padding tokens; Pfau et al., 2024) grows as a polynomial in n and whose weights on input length n (to accommodate growing width) are computable in L . Then T can be simulated in L -uniform TC^0 .*

Proof. By assumption, we can construct an L -uniform TC^0 circuit family in which the transformer weights for sequence length n are hardcoded as constants. Next, we can apply standard arguments (Merrill et al., 2022; Merrill and Sabharwal, 2023a,b) to show that the self-attention and feedforward sublayers can both be simulated by constant-depth threshold circuits, and the size remains polynomial (though a larger polynomial). Thus, any function computable by a constant-depth, polynomial-width transformer is in L -uniform TC^0 . \square

Theorem 4 (CoT Scaling). *Transformers with $O(\log n)$ chain-of-thought steps can only recognize languages in L -uniform TC^0 .*

Proof. The high-level idea is that a polynomial-size circuit can enumerate all possible $O(\log n)$ -length chains of thought. Then, in parallel for each chain of thought, we construct a threshold circuit that simulates a transformer (Merrill and Sabharwal, 2023a) on the input concatenated with the chain of thought, outputting the transformer's next token. We then select the chain of thought in which all simulated outputs match the correct next token and output its final answer. The overall circuit has constant depth, polynomial size, and can be shown to be L -uniform. Thus, any function computable by a transformer with $O(\log n)$ chain of thought is in TC^0 . \square

1064 F Experimental Details

1065 **Curriculum Training.** In early experiments, we found that learning from long A_5 sequences
1066 directly was infeasible for our transformer models. We hypothesize this was because, unless earlier
1067 tokens are predicted correctly, later tokens contribute significant noise to the gradient. In order to
1068 make the learning problem feasible, we follow a curriculum training process, first training on A_5
1069 sequences of length 2, then length 4, and continuing up to some fixed maximum power 2^i . We can
1070 then measure the maximum $n^* \leq 2^i$ such that the model achieves strong validation accuracy, as
1071 mentioned in Section 7.

1072 **Depth Experiments.** All depth experiments used a fixed width of 512. For historical reasons, we
1073 have slightly different numbers of runs for different experimental conditions, and some of the runs use
1074 different batch sizes (64 and 128). We originally ran a single sweep of depths and widths with 5 runs
1075 for depths 6, 12, 18, and 24, each using a batch size of 64 and maximum depth of $2^i = 128$. Seeking
1076 to clarify the trend between these original data points, we launched 3 additional runs at depths 9,
1077 15, 18, and 21 using a batch size of 128, which anecdotally sped up training without harming final
1078 performance. We also observed that the original depth 24 runs were at the ceiling $n^* = 128$, so we
1079 launched 3 additional depth-24 runs with a batch size of 128 and $2^i = 512$ (we also used this larger
1080 sequence length for all other runs in the second set). In total, this made:

- 1081 • 5 runs at depths 6, 12, and 8;
- 1082 • 3 runs at depths 9, 15, 18, and 21;
- 1083 • 8 runs at depth 24.

1084 **Width Experiments.** All width experiments used a fixed depth of 6. We launched 5 runs at widths
1085 128, 258, 512, 1024 with the same hyperparameters, each using a batch size of 64 and $2^i = 128$.

1086 **Compute.** Each training run was launched on a single GPU. We estimate that, together, these
1087 experiments took about 1000 GPU hours.

1088 **License.** The codebase of [Merrill et al. \(2024\)](#), which we used for data generation, has MIT license.