Thinking Like Transformers

Gail Weiss, Yoav Goldberg, Eran Yahav
Motivation: Transformer Encoders

- Powerful!
- Parallel!
- Popular!
- What are they doing?

Train fast on massive data
Motivation: Transformer Encoders

What are they doing?

We’re figuring out all kinds of things...

Attention Is All You Need
Motivation: Transformer Encoders

We’re figuring out all kinds of things...

Are Transformers universal approximators of sequence-to-sequence functions?

Chulhee Yun, Srinadh Bhojanapalli, Ankit Singh Rawat, Sashank J. Reddi, Sanjiv Kumar

Attention Is All You Need

...but it would be nice to have a model!
Motivation: Transformer Encoders

We’re figuring out all kinds of things...

Are Transformers universal approximators of sequence-to-sequence functions?

Chulhee Yun, Srinadh Bhojanapalli, Ankit Singh Rawat, Sashank J. Reddi, Sanjiv Kumar

Theoretical Limitations of Self-Attention in Neural Sequence Models

Michael Hahn

…but it would be nice to have a model!
Motivation: Transformer Encoders

What are they doing?

We’re figuring out all kinds of things...

Are Transformers universal approximators of sequence-to-sequence functions?
Chuihee Yun, Srinadh Bhojanapalli, Ankit Singh Rawat, Sashank J. Reddi, Sanjiv Kumar

Theoretical Limitations of Self-Attention in Neural Sequence Models
Michael Hahn

On the Ability and Limitations of Transformers to Recognize Formal Languages
Satwik Bhattamishra, Kabir Ahuja, Navin Goyal

...but it would be nice to have a model!
Motivation: Transformer Encoders

What are they doing?

We’re figuring out all kinds of things...

Are Transformers universal approximators of sequence-to-sequence functions?
Chulhee Yun, Srinadh Bhojanapalli, Ankit Singh Rawat, Sashank J. Reddi, Sanjiv Kumar

Theoretical Limitations of Self-Attention in Neural Sequence Models
Michael Hahn

On the Ability and Limitations of Transformers to Recognize Formal Languages
Satwik Bhattacharya, Kabir Ahuja, Navin Goyal

Attention is Turing-Complete

Statistically Meaningful Approximation: A Case Study on Approximating Turing Machines with Transformers
Colin Wei, Yining Chen, Tengyu Ma

...but it would be nice to have a model!
Motivation: What RNNs have

Spectral extraction: RNNs to WFA

DFA extraction:
  Clustering

DFA and WDFA extraction:
  L-star variants

Computational Model(s)!

Deterministic Finite Automata (DFAs)
Motivation: What RNNs have

Spectral extraction: RNNs to WFAs
DFA extraction: Clustering
DFA and WDFA extraction: L-star variants

Analysis of Expressive Power!

2-RNNs are WFAs
LSTMs are counter machines
GRUs are DFAs
Motivation: What RNNs have

- **RNN cell**
- **Hidden State**
- **Output Vector**

Initial State: $x_0$

Extraction! Analysis of Expressive Power! Inspiration from existing theory!

- **Extraction!**
  - Spectral extraction: RNNs to WFAs
  - DFA extraction: Clustering
  - DFA and WDFA extraction: L-star variants

- **Analysis of Expressive Power!**
  - 2-RNNs are WFAs
  - LSTMds are counter machines
  - GRUs are DFAs

- **Inspiration from existing theory!**
  - Stack-RNNs
  - Deterministic Finite Automata (DFAs)
Motivation: What RNNs have

**Extraction!**
Spectral extraction: RNNs to WFAs

DFA extraction: Clustering

DFA and WDFA extraction: L-star variants

**Analysis of Expressive Power!**
2-RNNs are WFAs

LSTMs are counter machines

GRUs are DFAs

**Inspiration from existing theory!**
Stack-RNNs

Deterministic Finite Automata (DFAs)
(References for the Interested)

Extraction!
- Explaining Black Boxes on Sequential Data using Weighted Automata
- Extraction of Rules from Discrete-Time Recurrent Neural Networks
- Extracting Automata from Recurrent Neural Networks Using Queries and Counterexamples

Analysis of Expressive Power!
- Connecting Weighted Automata and Recurrent Neural Networks through Spectral Learning
- On the Practical Computational Power of Finite Precision RNNs for Language Recognition
- Sequential Neural Networks as Automata

Inspiration from existing theory!
- Inferring Algorithmic Patterns with Stack-Augmented Recurrent Nets
- Learning to Transduce with Unbounded Memory

Computational Model(s)!
- Deterministic Finite Automata (DFAs)
- Connecting Weighted Automata and Recurrent Neural Networks through Spectral Learning
- On the Practical Computational Power of Finite Precision RNNs for Language Recognition
- Sequential Neural Networks as Automata
- A Formal Hierarchy of RNN Architectures
But what are Transformer-Encoders?

Transformer-Encoder

Computational Model(s)!

? Any ideas?
Transformer Encoders

Attention Is All You Need
Transformer Encoders

- Receive their entire input ‘at once’, processing all tokens in parallel
Transformer Encoders

- Receive their entire input ‘at once’, processing all tokens in parallel
- Have a fixed number of layers, where the output of one is the input of the next
The Transformer-Encoder

I
Like
Dogs
The Transformer-Encoder

**Word Embedding**
- $e(I)$
- $e$(Like)
- $e$(dogs)

**Positional Embedding**
- $p(0)$
- $p(1)$
- $p(2)$

**Initial Encoding**
- $x_1$
- $x_2$
- $x_3$
The Transformer-Encoder

Encoder
Layer 1

Encoder Output

Layer 1 Output

Encoder Layer L

Initial Encoding

Layer 1 Output

Encoder Output

Word Embedding

Positional Embedding

Like

Dogs

I

$e(I)$

$e(\text{Like})$

$e(\text{dogs})$

$p(0)$

$p(1)$

$p(2)$

$x_1$

$x_2$

$x_3$

$y_1$

$y_2$

$y_3$

$y_1$

$y_2$

$y_3$

Like

Dogs
The Transformer-Encoder

Does this thing have “states”? What *does* it have?
The Transformer-Encoder

Initial Encoding

Layer 1 Output

Encoder Output

No states… but there is a sense of information being propagated
The Transformer-Encoder

Layer input/outputs are “variables” of a transformer “program”

The layers themselves are “operations”
So... a programming language?
RASP (Restricted Access Sequence Processing)

- A transformer-encoder is a sequence to sequence function ("sequence operator", or, "s-op")
- Its layers apply operations to the sequence
- RASP describes the input sequences and what the layers can do with them

Layer input/outputs are "variables" of a transformer "program"  
The layers themselves are "operations"
RASP (Restricted Access Sequence Processing)

- A transformer-encoder is a sequence to sequence function (“sequence operator”, or, “s-op”)
- Its layers apply operations to the sequence
- RASP describes the **input sequences** and what the layers can do with them
RASP base s-ops

The information before a transformer has done anything ("0 layer transformer")
RASP base s-ops

The information before a transformer has done anything ("0 layer transformer")

tokens and indices are RASP built-ins:

>> tokens;
   s-op: tokens

>> indices;
   s-op: indices
The RASP REPL gives you examples (until you ask it not to)

I Like Dogs

Positional Embedding
\[ p(0) \]
\[ p(1) \]
\[ p(2) \]

Word Embedding
\[ e(I) \]
\[ e(\text{Like}) \]
\[ e(\text{dogs}) \]

d_x

d_x

d_x

d_x

d_x

d_x

The information before a transformer has done anything ("0 layer transformer")

tokens and indices are RASP built-ins:

>> tokens;
   s-op: tokens
   Example: tokens("hello") = [h, e, l, l, o] (strings)

>> indices;
   s-op: indices
   Example: indices("hello") = [0, 1, 2, 3, 4] (ints)

The RASP REPL gives you examples (until you ask it not to)
Okay, now what?

```
>> tokens;
  s-op: tokens
  Example: tokens("hello") = [h, e, l, l, o] (strings)

>> indices;
  s-op: indices
  Example: indices("hello") = [0, 1, 2, 3, 4] (ints)
```

To know what operations RASP may have, we must inspect the transformer-encoder layers!
Transformer-Encoder Layer

- **Input**:\( x_1, x_2, x_3 \)
- **Multi-Head Attention**:\( o_1, o_2, o_3 \)
- **Linear Transformation**:\( A \)
- **Residual (“Skip”) Connection**:\( A \)
- **Layer Norm 1**:\( o'_1, o'_2, o'_3 \)
- **Feed-Forward Sublayer**:\( W^{ff}_1, W^{ff}_2 \)
- **ReLU**:\( o''_1, o''_2, o''_3 \)
- **Residual (“Skip”) Connection**:\( o''_1, o''_2, o''_3 \)
- **Layer Norm 2**:\( out_1, out_2, out_3 \)

There's a lot in here…
Feed-Forward Sublayer

Input $x_1, x_2, x_3$ of dimension $d_x$ go through a Multi-Head Attention layer, which outputs $o_1, o_2, o_3$. These outputs are then linearly transformed and go through a residual ("Skip") connection. The output of this transformation is then passed through Layer Norm 1. The output of Layer Norm 1 is then passed through a Feed-Forward Sublayer, which consists of two layers: $W^{ff}_1$ and $W^{ff}_2$. The output of the Feed-Forward Sublayer is then passed through a Residual ("Skip") connection and Layer Norm 2, resulting in the final output $out_1, out_2, out_3$. The dimension of the input and output is $d_x$. The ReLU activation function is applied between $W^{ff}_1$ and $W^{ff}_2$.
Feed-Forward Sublayer

Multilayer Feedforward Networks are Universal Approximators (Hornik et al, 1989)
Feed-Forward gives us (Many) Elementwise Operations

Multilayer Feedforward Networks are Universal Approximators

Kurt Hornik
Technische Universität Wien

Maxwell Stinchcombe and Halbert White
University of California, San Diego

(Received 16 September 1988; revised and accepted 9 March 1989)

Abstract—This paper rigorously establishes that standard multilayer feedforward networks with as few as one hidden layer using arbitrary squashing functions are capable of approximating any Borel measurable function from one finite dimensional space to another to any desired degree of accuracy, provided sufficiently many hidden units are available. In this sense, multilayer feedforward networks are a class of universal approximators.

>> indices+1;
   s-op: out
   Example: out("hello") = [1, 2, 3, 4, 5] (ints)

>> tokens=="e" or tokens=="o";
   s-op: out
   Example: out("hello") = [F, T, F, F, T] (booleans)
So far

```python
>> tokens;
  s-op: tokens
  Example: tokens("hello") = [h, e, l, l, o] (strings)

>> indices;
  s-op: indices
  Example: indices("hello") = [0, 1, 2, 3, 4] (ints)

>> indices+1;
  s-op: out
  Example: out("hello") = [1, 2, 3, 4, 5] (ints)

>> tokens=="e" or tokens=="o";
  s-op: out
  Example: out("hello") = [F, T, F, F, T] (bools)

Are we all-powerful
(well, transformer-powerful) yet?
Transformer-Encoder Layer

Multi-Head Attention

Linear Transformation

Residual ("Skip") Connection

Layer Norm 1

Output

Feed-Forward Sublayer

ReLU

Layer Norm 2

Input

$x_1$

$x_2$

$x_3$

$d_x$

$W^{ff}_1$

$ff_1$

$ff_2$

$ff_3$

$d_{ff}$

$W^{ff}_2$

$ReLU$

$d_x$

$W^{ff}_2$

$ff_1$

$ff_2$

$ff_3$

$d_{ff}$

Residual ("Skip") Connection

Output

$out_1$

$out_2$

$out_3$

$d_x$
Attention Sublayer

- Multi-Head Attention
- Feed-Forward Sublayer
- Linear Transformation
- Residual ("Skip") Connection
- Layer Norm 1
- Layer Norm 2
- Output

- $x_1, x_2, x_3$
- $d_x$
- $d_{ff}$

- $o_1, o_2, o_3$
- $d_x$

- $W_{ff1, f1, f2, f3}$
- ReLU

- $out_1, out_2, out_3$
- $d_x$
Background - Multi Head Attention

Starting from single-head attention...
Background - Self Attention (Single Head)
Background - Self Attention (Single Head)
Background - Self Attention (Single Head)
Background - Self Attention (Single Head)
Background - Self Attention (Single Head)
Background - Self Attention (Single Head)

Given an input sequence \( x_1, x_2, x_3 \) of length \( d_x \), we compute the queries \( Q \) and keys \( K \) matrices.

The queries \( Q \) and keys \( K \) are then multiplied element-wise:

\[
Q \cdot K = \begin{bmatrix} q_1 \cdot k_1 \\ q_2 \cdot k_2 \\ q_3 \cdot k_3 \end{bmatrix}
\]

These scores are then normalized by dividing by \( \sqrt{d_k} \):

\[
\text{scores} = \frac{Q \cdot K}{\sqrt{d_k}}
\]

Finally, the normalized scores are used to compute weights \( w_{1,1}, w_{1,2}, w_{1,3} \) using the softmax function:

\[
\text{weights} = \text{softmax}(\text{scores})
\]
Background - Self Attention (Single Head)
Background - Self Attention (Single Head)
Background - Self Attention (Single Head)

A diagram illustrating the self-attention process with a single head. The input sequence is represented as $x_1, x_2, x_3$ with dimension $d_x$. The query, key, and value matrices are denoted as $Q$, $K$, and $V$ respectively, each with dimension $d_k$ and $d_v$.

The query, key, and value matrices are multiplied element-wise, resulting in three matrices $q_1, q_2, q_3$ and $k_1, k_2, k_3$. These matrices are then multiplied with the weights $w_3,1, w_3,2, w_3,3$.

The weights are then used to calculate a score for each element, which is then normalized (i.e., $\times 1/\sqrt{d_k}$) and subjected to a softmax function to produce attention weights.

The output is then calculated as a weighted sum of the value matrices, resulting in $out_1, out_2, out_3$.
Attention Head

- Input: \(x_1, x_2, x_3\)
- \(Q\): Query matrix
- \(K\): Key matrix
- \(V\): Value matrix
- \(d_x\): Dimension of input
- \(d_k\): Dimension of key
- \(d_v\): Dimension of value

**Scores**

- \(q_3 \cdot k_1\), \(q_3 \cdot k_2\), \(q_3 \cdot k_3\)

**Normalise** (i.e. \(\times 1/\sqrt{d_k}\))

**Softmax**

**Weights**

- \(w_{3,1}, w_{3,2}, w_{3,3}\)

**Output**

- \(out_1, out_2, out_3\)
Background - Multi-Headed Self Attention

- **Input**: \( x_1, x_2, x_3 \)

- **Output**: \( \text{concatenate} \)

- **Heads**: Head 1, Head 2, ..., Head H

- **Dimensions**:
  - \( d_k = d_v = d_h = \frac{d_x}{H} \)
  - \( d_x \)
  - \( d_h \)

- **Outputs**:
  - \( \text{out}_1^1, \text{out}_1^2, ..., \text{out}_1^H \)
  - \( \text{out}_2^1, \text{out}_2^2, ..., \text{out}_2^H \)
  - \( \text{out}_3^1, \text{out}_3^2, ..., \text{out}_3^H \)
The multi-headed attention lets one layer do multiple operations

It does not in itself add new power
So, how do we present one head?
Self Attention (Single Head)
Single Head: Scoring ↔ Selecting

Attention Head

input

$\begin{align*}
Q & \quad k_1 \\
K & \quad k_2 \\
V & \quad k_3
\end{align*}$

scores

$\begin{align*}
q_3 \cdot k_1 & \quad q_3 \cdot k_2 & \quad q_3 \cdot k_3 \\
\text{normalise (i.e. } \times 1/\sqrt{d_k})
\end{align*}$

softmax

weights

$\begin{align*}
w_{3,1} & \quad w_{3,2} & \quad w_{3,3}
\end{align*}$

Pairwise!

$\begin{align*}
q_1 & \quad q_2 & \quad q_3
\end{align*}$

$\begin{align*}
\text{softmax}
\end{align*}$

$\begin{align*}
\text{normalise (i.e. } \times 1/\sqrt{d_k})
\end{align*}$

$\begin{align*}
\text{softmax}
\end{align*}$

$\begin{align*}
\text{normalise (i.e. } \times 1/\sqrt{d_k})
\end{align*}$

$\begin{align*}
\text{softmax}
\end{align*}$

$\begin{align*}
\text{softmax}
\end{align*}$

$\begin{align*}
\text{softmax}
\end{align*}$

$\begin{align*}
\text{softmax}
\end{align*}$

$\begin{align*}
\text{softmax}
\end{align*}$

$\begin{align*}
\text{softmax}
\end{align*}$

$\begin{align*}
\text{softmax}
\end{align*}$
Single Head: Scoring ↔ Selecting

Decision: RASP abstracts to binary choose/don’t choose decisions

\[ \text{sel} = \text{select}([2,0,0],[0,1,2],==) \]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>
Single Head: Scoring ↔ Selecting

Decision: RASP abstracts to binary choose/don’t choose decisions

$$\text{sel} = \text{select}(\begin{bmatrix} 2, 0, 0 \end{bmatrix}, [0, 1, 2], ==)$$

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
Single Head: Scoring ↔ Selecting

Decision: RASP abstracts to binary choose/don’t choose decisions

\[ \text{sel} = \text{select}([2,0,0],[0,1,2],==) \]
Single Head: Scoring $\leftrightarrow$ Selecting

Decision: RASP abstracts to binary choose/don’t choose decisions

$$\text{sel} = \text{select}([2,0,0],[0,1,2],[0])$$

| 2 | 0 | 0 |
| 0 | F | T | T |
| 1 | F | F | F |
| 2 | T | F | F |
Single Head: Scoring ↔ Selecting

Decision: RASP abstracts to binary choose/don’t choose decisions

\[ \text{sel} = \text{select}(\mathbf{[2,0,0],[0,1,2]},==) \]

<table>
<thead>
<tr>
<th></th>
<th>F</th>
<th>T</th>
<th>T</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
Single Head: Scoring ↔ Selecting

Decision: RASP abstracts to binary choose/don’t choose decisions

```
\text{sel} = \text{select}(\begin{bmatrix} 2, 0, 0 \end{bmatrix}, \begin{bmatrix} 0, 1, 2 \end{bmatrix}, \text{==})
```

```
\begin{array}{ccc}
2 & 0 & 0 \\
0 & F & T & T \\
1 & F & F & F \\
2 & T & F & F
\end{array}
```
Single Head: Scoring ↔ Selecting

Decision: RASP abstracts to binary choose/don’t choose decisions

\[ \text{sel} = \text{select}([2,0,0],[0,1,2],==) \]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>F</th>
<th>T</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
Single Head: Scoring ↔ Selecting

Decision: RASP abstracts to binary choose/don’t choose decisions

\[
\text{sel} = \text{select}([2,0,0],[0,1,2],==)
\]

\[
\begin{array}{ccc}
2 & 0 & 0 \\
0 & F & T & T \\
1 & F & F & F \\
2 & T & F & F \\
\end{array}
\]
Single Head: Scoring ↔ Selecting

Decision: RASP abstracts to binary choose/don’t choose decisions

\[ \text{sel} = \text{select}([2,0,0],[0,1,2],==) \]

<table>
<thead>
<tr>
<th>2</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

Another example:

\[ \text{sel2} = \text{select}([2,0,0],[0,1,2],==) \]

<table>
<thead>
<tr>
<th>2</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

Is this “reasonable”? What does it mean with respect to Q and K?
Single Head: Weighted Average ↔ Aggregation

Attention Head

scores

$q_3 \cdot k_1 \quad q_3 \cdot k_2 \quad q_3 \cdot k_3$

normalise (i.e. $\times 1/\sqrt{d_k}$)

softmax

weights

$v_1 \quad v_2 \quad v_3$

out1

out2

out3

$w_{3,1} \quad w_{3,2} \quad w_{3,3}$

input

$x_1 \quad x_2 \quad x_3$

$d_x$
Single Head: Weighted Average ↔ Aggregation

new = aggregate(sel, [1, 2, 4])

1 2 4
F T T 1 2 4 => 3
F F F 1 2 4 => 0 => [3, 0, 1]
T F F 1 2 4 => 1
Single Head: Weighted Average $\leftrightarrow$ Aggregation

$$\text{new} = \text{aggregate}(\text{sel}, [1, 2, 4])$$

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>1</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
</table>
| F | T | T | 1 | 2 | 4 | => 3
| F | F | F | 1 | 2 | 4 | => 0 => [3, 0, 1]
| T | F | F | 1 | 2 | 4 | => 1
Single Head: Weighted Average ↔ Aggregation

\[ \text{new} = \text{aggregate(} \text{sel, [1, 2, 4]} \text{)} \]

\[
\begin{array}{cccc}
\text{F} & \text{T} & \text{T} & 1 2 4 \\
\text{F} & \text{F} & \text{F} & 1 2 4 \\
\text{T} & \text{F} & \text{F} & 1 2 4 \\
\end{array}
\Rightarrow
\begin{array}{c}
3 \\
0 \\
1 \\
\end{array}
\Rightarrow [3, 0, 1]

Single Head: Weighted Average ↔ Aggregation

new = aggregate(sel, [1, 2, 4])

1 2 4
F T T 1 2 4 => 3
F F F 1 2 4 => 0 => [3, 0, 1]
T F F 1 2 4 => 1
Single Head: Weighted Average ↔ Aggregation

\[ \text{new} = \text{aggregate}(\text{sel}, [1, 2, 4]) \]

\[
\begin{array}{cccc}
F & T & T & 1 2 4 \\
F & F & F & 1 2 4 \\
T & F & F & 1 2 4 \\
\end{array}
\implies \begin{array}{c}
3 \\
0 \\
1 \\
\end{array} \\
\implies [3, 0, 1]
Single Head: Weighted Average ↔ Aggregation

new = aggregate(sel, [1,2,4])

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>=&gt; 3</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>=&gt;  0 =&gt; [3,0,1]</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>=&gt;  1</td>
</tr>
</tbody>
</table>

Symbolic language + no averaging when only one position selected allows (for example):

flip = select([2,1,0],[0,1,2],==)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

reverse = aggregate(flip, [A,B,C])

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>=&gt; C</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>=&gt; B =&gt; [C,B,A]</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>=&gt; A</td>
</tr>
</tbody>
</table>
Single Head: Select/Aggregate in RASP

The select decisions are pairwise!!
What would happen if they weren't?

Example from before: reverse in RASP

```r
>> flip = select(length-indices-1, indices, ==);
selector: flip
Example:
  hello
  h  e  l  o
  1  1  1  1
```
The select decisions are pairwise!!
What would happen if they weren’t?

Example from before: reverse in RASP

>> flip = select(length-indices-1, indices, ==);
selector: flip
Example:

| h | 1 |
| e | 1 |
| l | 1 |
| l | 1 |
| o | 1 |

>> reverse = aggregate(flip, tokens);
S-op: reverse
Example: reverse("hello") = [o, l, l, e, h] (strings)
Single Head: Select/Aggregate in RASP

Example from before: reverse in RASP

```plaintext
>> flip = select(length-indices-1, indices, ==);
>> reverse = aggregate(flip, tokens);
```

S-op: reverse
Example: reverse("hello") = [o, l, l, e, h] (strings)

See anything suspicious in the example?

The select decisions are pairwise!!
What would happen if they weren’t?
Okay, that’s our parts!
Recap: The main transformer components are:
Recap: The main transformer components are:

The Initial Sequences

Word Embedding
- \( e(0) \)
- \( e(like) \)
- \( e(dogs) \)

Positional Embedding
- \( p(0) \)
- \( p(1) \)
- \( p(2) \)

\( d_i \)

“indices” and “tokens”

(...and “length”!)
Recap: The main transformer components are:

The Initial Sequences

Feed-Forward Sublayers

“indices” and “tokens”

(...and “length”!)

Multilayer Feedforward Networks are Universal Approximators (Hornik et al, 1989)
Recap: The main transformer components are:

The Initial Sequences

- Word Embedding
- Positional Embedding

“indices” and “tokens”

(...and “length”!)

Feed-Forward Sublayers

- Multilayer Feedforward Networks are Universal Approximators (Hornik et al, 1989)
- Powerful elementwise operations

Attention Heads

- Pairwise “Select”, then “Aggregate”

Input

- $x_1$
- $x_2$
- $x_3$
- $x_4$

Output

- $q_1$
- $q_2$
- $q_3$
- $q_4$

Attention Head

- $K$
- $V$

Scores

- Weights
- Normalise
- Softmax

Feed-Forward Sublayer

- $W_{ff}$

$W_{ff}$ Values

- $W_{ff1}$
- $W_{ff2}$
- $W_{ff3}$

Feed-Forward Sublayer

- ReLU
Recap: The main transformer components are:

The Initial Sequences
- Word Embedding
- Positional Embedding
- “indices” and “tokens”
- (…and “length”!)

Feed-Forward Sublayers
- Feed-Forward Sublayer
- Powerful elementwise operations
- Multilayer Feedforward Networks are Universal Approximators (Hornik et al, 1989)

Attention Heads
- Attention Head
- Pairwise “Select”, then “Aggregate”
Recap: The main transformer components are:

The Initial Sequences
- "indices" and "tokens"
- (...and "length"!)

Feed-Forward Sublayers
- Powerful elementwise operations
- Multilayer Feedforward Networks are Universal Approximators (Hornik et al, 1989)

Attention Heads
- Pairwise "Select", then "Aggregate"

Skip connection

Deep Residual Learning for Image Recognition
Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun

Encourages idea that information can be retained through many layers...
Recap: The main transformer components are:

The Initial Sequences
- Word Embedding
- Positional Embedding
- "indices" and "tokens" (and "length")

Feed-Forward Sublayers
- Powerful elementwise operations
- Multilayer Feedforward Networks are Universal Approximators (Hornik et al., 1989)

Attention Heads
- Pairwise "Select", then "Aggregate"

Skip connection
- Deep Residual Learning for Image Recognition
- Encourages idea that information can be retained through many layers...

Layernorms
LayerNorm

Parameters
- $a_2$ (vector)
- $b_2$ (vector)
- $\epsilon$ (small constant)

$$\bar{x} = \frac{[a] \odot (x - \bar{x})}{\text{std}(x) + \epsilon} + [b]$$
So the components are:

The Initial Sequences

- Word Embedding
- Positional Embedding
  - $p(0)$
  - $p(1)$
  - $p(2)$

```
“indices” and “tokens”
```

(...and “length”!)

Attention Heads

- Pairwise “Select”, then “Aggregate”

Feed-Forward Sublayers

- Powerful elementwise operations
- Multilayer Feedforward Networks are Universal Approximators (Hornik et al, 1989)

Skip connection

Deep Residual Learning for Image Recognition
Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun

```
Encourages idea that information can be retained through many layers…
```

LayerNorms

```
[a] ⊗ (x - \bar{x}) + [b]
```

Open Question!!
RASP (Restricted Access Sequence Processing)

**Initial Sequences**

```plaintext
>> tokens;
  s-op: tokens
  Example: tokens("hello") = [h, e, l, l, o] (strings)

>> indices;
  s-op: indices
  Example: indices("hello") = [0, 1, 2, 3, 4] (ints)
```

Sequence values: integer, float, string, boolean (uniform per sequence)

**Selectors, and aggregate**

```plaintext
>> flip = select(length-indices-1,indices,==);
  selector: flip
  Example:
      h e l l o
    __ | 1
    h | 1
    e | 1
    l | 1
    o | 1

>> reverse = aggregate(flip,tokens);
  s-op: reverse
  Example: reverse("hello") = [o, l, l, e, h] (strings)
```

Selectors can be composed with `and`, `not`, and `or`. Aggregate can take default values for rows where the selector is empty.

**Most “normal” operators present, and applied elementwise to sequences**

```plaintext
>> indices+1;
  s-op: out
  Example: out("hello") = [1, 2, 3, 4, 5] (ints)

>> tokens=="e" or tokens=="o";
  s-op: out
  Example: out("hello") = [F, T, F, F, T] (bools)
```

“Element” primitives also present, e.g. “e” and 1 above. These are implicitly converted to constant-value sequences.
Small Example

Computing \textit{length}: 
Small Example

Computing *length*:

```plaintext
>> indicator(indices==0);
  s-op: out
Example: out("hello") = [1, 0, 0, 0, 0] (ints)
```
Small Example

Computing length:

```plaintext
>> full_s = select(1,1,==);
  selector: full_s
  Example:

    h e l l o
    h | 1 1 1 1 1
    e | 1 1 1 1 1
    l | 1 1 1 1 1
    l | 1 1 1 1 1
    o | 1 1 1 1 1
```

```plaintext
>> indicator(indices==0);
  s-op: out
  Example: out("hello") = [1, 0, 0, 0, 0] (ints)
```
Small Example

Computing length:

```python
>> full_s = select([1,1,==]);
    selector: full_s
Example:

   h e l l o
h | 1 1 1 1 1
  e | 1 1 1 1 1
  l | 1 1 1 1 1
  l | 1 1 1 1 1
  o | 1 1 1 1 1
```

```python
>> indicator(indices==0);
  s-op: out
Example: out("hello") = [1, 0, 0, 0, 0] (ints)
```

(On an example of length 4:)

```python
frac_0=aggregate(full_s, [1,0,0,0])
    1 0 0 0
  T T T T 1 0 0 0 => 0.25
  T T T T 1 0 0 0 => 0.25 => [0.25,0.25,0.25,0.25]
  T T T T 1 0 0 0 => 0.25
  T T T T 1 0 0 0 => 0.25
```
Small Example

Computing *length*:

```
>> full_s = select(1,1,==);
   selector: full_s
   Example:
     h e l l o
     h | 1 1 1 1 1
     e | 1 1 1 1 1
     l | 1 1 1 1 1
     l | 1 1 1 1 1
     o | 1 1 1 1 1

>> frac_0=aggregate(full_s, indicator(indices==0));
   s-op: frac_0
   Example: frac_0("hello") = [0.2]*5 (floats)

>> indicator(indices==0);
   s-op: out
   Example: out("hello") = [1, 0, 0, 0, 0] (ints)
```

(On an example of length 4:)

```
frac_0=aggregate(full_s, [1,0,0,0])

1 0 0 0
T T T T 1 0 0 0 => 0.25
T T T T 1 0 0 0 => 0.25 => [0.25,0.25,0.25,0.25]
T T T T 1 0 0 0 => 0.25
T T T T 1 0 0 0 => 0.25
```
Small Example

Computing length:

```
>> full_s = select(1,1,==);
  selector: full_s
  Example:
    h e l l o
    h | 1 1 1 1 1
    e | 1 1 1 1 1
    l | 1 1 1 1 1
    l | 1 1 1 1 1
    o | 1 1 1 1 1

>> frac_0=aggregate(full_s,indicator(indices==0));
  s-op: frac_0
  Example: frac_0("hello") = [0.2]*5 (floats)

>> round(1/frac_0);
  s-op: out
  Example: out("hello") = [5]*5 (ints)
```

```
>> indicator(indices==0);
  s-op: out
  Example: out("hello") = [1, 0, 0, 0, 0] (ints)
```

(On an example of length 4:)

```
frac_0=aggregate(full_s, [1,0,0,0])
  1 0 0 0
  T T T T 1 0 0 0 => 0.25
  T T T T 1 0 0 0 => 0.25 => [0.25,0.25,0.25,0.25]
  T T T T 1 0 0 0 => 0.25
  T T T T 1 0 0 0 => 0.25
```
RASP analysis?

- *indices* and *tokens*:
  require zero layers

- *select-aggregate* pairs:
  must be at least one layer after all of their dependencies
  *can have multiple pairs in one layer (multi-headed attention)*

- local (feed-forward) operations:
  don’t add layers (attached to earliest layer after dependencies are finished)
RASP analysis?

Can draw head/layer analysis, eg:

```markdown
>> draw(reverse,"abcdeabcde")
```
RASP analysis?

Can draw head/layer analysis, eg:

```python
>>> draw(reverse,"abcdeabcde")
```
RASP analysis?

Can draw head/layer analysis, eg:

```python
>> draw(reverse,"abcdeabcde")
```
RASP analysis?

Can draw head/layer analysis, eg:

```python
>> draw(reverse,"abcdeabcde")
```
RASP analysis?

Can draw head/layer analysis, eg:

```python
>> draw(reverse,"abcdeabcde")
```
RASP analysis?

Can draw head/layer analysis, eg:

```javascript
>> draw(reverse,"abcdeabcde")
```
Connection to Reality?

Are our RASP programs predicting the right number of layers?

Are our RASP programs predicting relevant selector patterns?
Connection to Reality?

Example 1: reverse

```
>> flip_s = select(length-indices-1,indices,==);
selector: flip_s
Example:

Example: reverse("hello") = [o, l, l, e, h]
```
Example 1: reverse

RASP analysis:

- First, \textit{length} is computed
  \hfill (1 layer, uniform attention)
- Then, \textit{length} is used to create \textit{flip_s}
  \hfill (necessarily in next layer, ‘flipped’ attention)
Connection to Reality?

Example 1: reverse

```
>> flip_s = select(length-indices-1,indices, ==);
selector: flip_s
Example:

<table>
<thead>
<tr>
<th>h</th>
<th>l</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>1</td>
</tr>
<tr>
<td>l</td>
<td>1</td>
</tr>
<tr>
<td>o</td>
<td>1</td>
</tr>
</tbody>
</table>

>> reverse=aggregate(flip_s,tokens);

Example: reverse("hello") = [o, l, l, e, h]
```

RASP analysis:

- First, length is computed
  (1 layer, uniform attention)
- Then, length is used to create flip_s
  (necessarily in next layer, ‘flipped’ attention)

→ hypothesis: reverse requires 2 layers?
Connection to Reality?

```python
>> draw(reverse,"abcdeabcde")
```

RASP expects 2 layers for arbitrary-length reverse
Connection to Reality?

RASP expects 2 layers for arbitrary-length reverse

Test:
Training small transformers on lengths 0-100:

2 layers: 99.6% accuracy after 20 epochs
1 layer: 39.6% accuracy after 50 epochs

Even with compensation for number of heads and parameters!
Connection to Reality?

[>> draw(reverse,"abcdeabcde")]

RASP expects 2 layers for arbitrary-length reverse

Test:
Training small transformers on lengths 0-100:

2 layers: 99.6% accuracy after 20 epochs
1 layer: 39.6% accuracy after 50 epochs

Bonus: the 2 layer transformer's attention patterns:
Connection to Reality?

Example 2: histogram (assuming BOS)

Eg:

[$\$h,e,l,l,o] \rightarrow [0,1,1,2,2,1]

[$\$a,b,c,c,c] \rightarrow [0,1,1,3,3,3]

[$\$a,b,a] \rightarrow [0,2,1,2]
Connection to Reality?

Example 2: histogram (assuming BOS)

Eg:

§,h,e,l,l,o → [0,1,1,2,2,1]
§,a,b,c,c,c → [0,1,1,3,3,3]
§,a,b,a → [0,2,1,2]
Example 2: *histogram* (assuming BOS)

Reminder: computing length

$$\text{frac}_0 = \text{aggregate}([1,0,0,0])$$

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>[§,h,e,l,l,o]</td>
<td>[0,1,1,2,2,1]</td>
</tr>
<tr>
<td>[§,a,b,c,c,c]</td>
<td>[0,1,1,3,3,3]</td>
</tr>
<tr>
<td>[§,a,b,a]</td>
<td>[0,2,1,2]</td>
</tr>
</tbody>
</table>

Trick was: send 1 from exactly one position, and then use weighted average to compute inverse of number of selected positions (for length, this was all positions).
Connection to Reality?

Example 2: histogram  (assuming BOS)

Reminder: computing length

\[
\text{frac}_0 = \text{aggregate}(\text{full}_s, [1,0,0,0])
\]

Eg:

\[
\begin{align*}
\text{[§,h,e,l,l,o]} & \mapsto [0,1,1,2,2,1] \\
\text{[§,a,b,c,c,c]} & \mapsto [0,1,1,3,3,3] \\
\text{[§,a,b,a]} & \mapsto [0,2,1,2]
\end{align*}
\]

Trick was: send 1 from exactly one position, and then use weighted average to compute inverse of number of selected positions (for length, this was all positions)

Can we use a similar trick for histograms?
Connection to Reality?

Example 2: histogram  (assuming BOS)

Reminder: computing length

\[ \text{frac}_0 = \text{aggregate}(\text{full}_s, [1,0,0,0]) \]

Can we use a similar trick for histograms?

Eg:

\[
\begin{align*}
[\$,h,e,l,l,o] & \rightarrow [0,1,1,2,2,1] \\
[\$,a,b,c,c,c] & \rightarrow [0,1,1,3,3,3] \\
[\$,a,b,a] & \rightarrow [0,2,1,2]
\end{align*}
\]
Connection to Reality?

**Example 2: histogram**  (assuming BOS)

Reminder: computing length

\[
\text{frac}_0 = \text{aggregate}(\text{full}_s, [1,0,0,0])
\]

<table>
<thead>
<tr>
<th>Position</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0</td>
<td>1.000</td>
</tr>
<tr>
<td>T T T T T</td>
<td>1.000 =&gt; 0.25</td>
</tr>
<tr>
<td>T T T T T</td>
<td>1.000 =&gt; 0.25 =&gt; [0.25,0.25,0.25,0.25]</td>
</tr>
<tr>
<td>T T T T T</td>
<td>1.000 =&gt; 0.25</td>
</tr>
<tr>
<td>T T T T T</td>
<td>1.000 =&gt; 0.25</td>
</tr>
</tbody>
</table>

Trick was: send 1 from exactly one position, and then use weighted average to compute inverse of number of selected positions (for length, this was all positions)

Can we use a similar trick for histograms?

**Q:** Specifically, what's the challenge for histograms?

**A:** Need a known position to send 1 from!
Connection to Reality?

Example 2: *histogram*  (assuming BOS)

>> set example "$hello"
Connection to Reality?

Example 2: *histogram* (assuming BOS)

```python
>> set example "$hello"
>> same_or_0 = select(tokens, tokens, ==) or select(indices, 0, ==);  
```

```
selector: same_or_0
Example:

```

| $ | 1 |
| h | 1 |
| e | 1 |
| l | 1 |
| o | 1 |

```

```
Connection to Reality?

Example 2: histogram (assuming BOS)

```python
>> set example "$hello"

Example:

<table>
<thead>
<tr>
<th>$</th>
<th>h</th>
<th>e</th>
<th>l</th>
<th>l</th>
<th>o</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

>> same_or_0 = select(tokens, tokens, ==) or select(indices, 0, ==);

Example:

$ \text{selector: same\_or\_0}$

Example: $\text{frac\_with\_0}("$hello") = [1, 0.5, 0.5, 0.333, 0.333, 0.5]$ (floats)

>> frac_with_0 = aggregate(same_or_0, indicator(indices==0));
```

\text{Example: frac\_with\_0}$
Connection to Reality?

Example 2: histogram (assuming BOS)

```python
>> set example "hello"
>> same_or_0 = select(tokens, tokens, ==) or select(indices, 0, ==);

selector: same_or_0
Example:

+---+---+---+---+---+
| $ | h | e | l | o |
+---+---+---+---+---+
| 1 | 1 | 1 | 1 | 1 |

>> frac_with_0 = aggregate(same_or_0, indicator(indices==0));

s-op: frac_with_0
Example: frac_with_0("hello") = [1, 0.5, 0.5, 0.333, 0.333, 0.5] (floats)

>> histogram_assuming_bos = round(1/frac_with_0)-1;

s-op: histogram_assuming_bos
Example: histogram_assuming_bos("hello") = [0, 1, 1, 2, 2, 1] (ints)
**Connection to Reality?**

**Example 2**: *histogram* (assuming BOS)

```plaintext
>> examples off
>> same_or_0 = select(tokens, tokens, ==) or select(indices, 0, ==);
    selector: same_or_0
>> frac_with_0 = aggregate(same_or_0, indicator(indices==0));
    s-op: frac_with_0
>> histogram_assuming_bos = round(1/frac_with_0)-1;
    s-op: histogram_assuming_bos
>> histogram_assuming_bos("$hello");
    = [0, 1, 1, 2, 2, 1] (ints)
```
Connection to Reality?

Example 2: *histogram* (assuming BOS)

```python
>> examples off
>> same_or_0 = select(tokens, tokens, ==) or select(indices, 0, ==);
    selector: same_or_0
>> frac_with_0 = aggregate(same_or_0, indicator(indices==0));
    s-op: frac_with_0
>> histogram_assuming_bos = round(1/frac_with_0)-1;
    s-op: histogram_assuming_bos
>> histogram_assuming_bos("$hello");
    = [0, 1, 1, 2, 2, 1] (ints)
```

**RASP analysis:**

- Just one attention head
- It focuses on:
  1. All positions with same token, and:
  2. Position 0 (regardless of content)
Connection to Reality?

Example 2: *histogram* (assuming BOS)

```plaintext
>> examples off
>> same_or_0 = select(tokens,tokens,==) or select(indices,0,==);
    selector: same_or_0
>> frac_with_0 = aggregate(same_or_0,indicator(indices==0));
    s-op: frac_with_0
>> histogram_assuming_bos = round(1/frac_with_0)-1;
    s-op: histogram_assuming_bos
>> histogram_assuming_bos("$hello"):
    = [0, 1, 1, 2, 2, 1] (ints)
```

RASP analysis:
- Just one attention head
- It focuses on:
  1. All positions with same token, and:
  2. Position 0 (regardless of content)

Selector pattern vs trained transformer’s attention for same input sequence:
Insight

1. Further motivates the *Universal Transformer*

Recurrent blocks are like allowing loops in RASP!

**Universal Transformers**

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, Łukasz Kaiser
2. Explains results of the Sandwich Transformer

Improving Transformer Models by Reordering their Sublayers

Ofir Press, Noah A. Smith, Omer Levy

If re-ordering and switching attention and feed-forward layers of a transformer (while adjusting to keep same number of parameters):

1. Better to have attention earlier, and feed-forward later
2. Only attention not enough
Insight

3. Transformers can “use” at least $n \log(n)$ of the $n^2$ computational cost they have:

“selector_width” is a RASP operation that takes an arbitrary selector and computes its width, for example:

```plaintext
>> selector_width(select(tokens, tokens, ==));

s-op: out
Example: out("hello") = [1, 1, 2, 2, 1] (ints)
```
Insight

3. Transformers can “use” at least $n \log(n)$ of the $n^2$ computational cost they have:

“selector_width” is a RASP operation that takes an arbitrary selector and computes its width, for example:

```plaintext
>> selector_width(select(tokens,tokens,==));

   s-op: out
Example: out("hello") = [1, 1, 2, 2, 1] (ints)
```

This can be used to implement sort:

```plaintext
>> selector examples off
>> earlier_token = select(tokens,tokens,<) or (select(tokens,tokens,==) and select(indices,indices,<));

   selector: earlier_token

>> num_prev = selector_width(earlier_token);

   s-op: num_prev
Example: num_prev("hello") = [1, 0, 2, 3, 4] (ints)

>> sorted = aggregate(select(num_prev,indices,==),tokens);

   s-op: sorted
Example: sorted("hello") = [e, h, l, l, o] (strings)
```

which we know requires at least $n \log(n)$ operations
(if making no assumptions on input data)
Insight

3. Transformers can “use” at least $n \log(n)$ of the $n^2$ computational cost they have:

“selector_width” is a RASP operation that takes an arbitrary selector and computes its width, for example:

```python
>> selector_width(select(tokens, tokens, ==));
  s-op: out
  Example: out("hello") = [1, 1, 2, 2, 1] (ints)
```

This can be used to implement sort:

```python
>> selector examples off
>> earlier_token = select(tokens, tokens, <) or (select(tokens, tokens, ==) and select(indices, indices, <));
  selector: earlier_token
>> num_prev = selector_width(earlier_token);
  s-op: num_prev
    Example: num_prev("hello") = [1, 0, 2, 3, 4] (ints)
>> sorted = aggregate(select(num_prev, indices, ==), tokens);
  s-op: sorted
    Example: sorted("hello") = [e, h, l, l, o] (strings)
```

which we know requires at least $n \log(n)$ operations
(if making no assumptions on input data)

Open Question: is there something that “uses” all $n^2$ of the attention head cost?
End

“Thinking Like Transformers” - ICML 2021
(Available on Arxiv)

Try it out!
🌟 github.com/tech-srl/RASP 🌟
Optional Talking Points

• Bhattamishra et al (2020) note that, unlike LSTMs, transformers struggle with some regular languages. Why might that be? (What would a general method for encoding a DFA in a transformer be?)

• Hahn (2019) proves that transformers with hard attention cannot compute Parity with hard attention. RASP can compute parity. What is the difference?

• How should we convert a RASP program to ‘real’ transformers? How big does our head-dimension need to be for “select(indices,indices,<-)”? How do we implement \textit{and}, \textit{or}, and \textit{not} between selectors?

• Do our selectors cover all the possible attention patterns? What is missing?