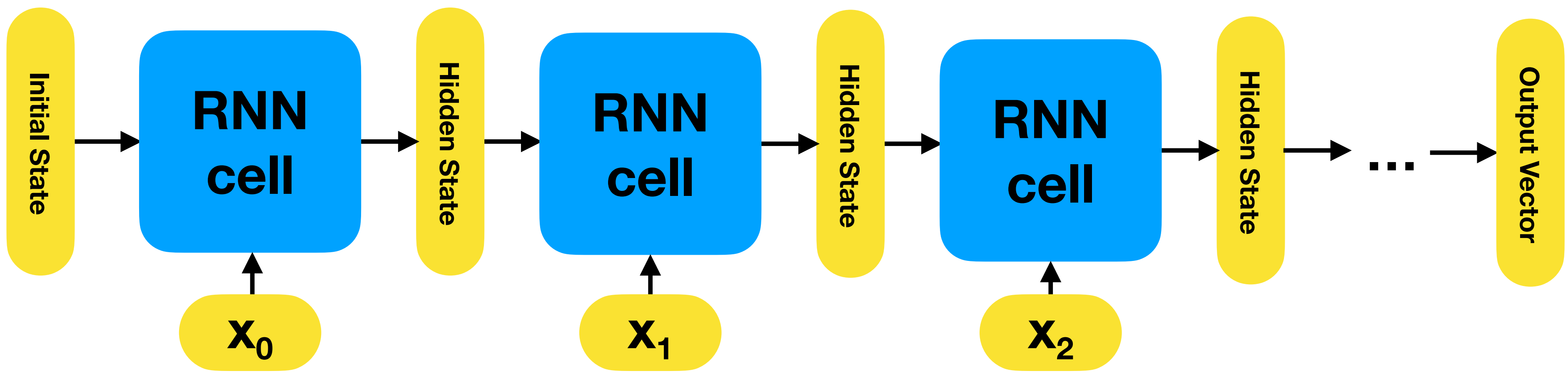
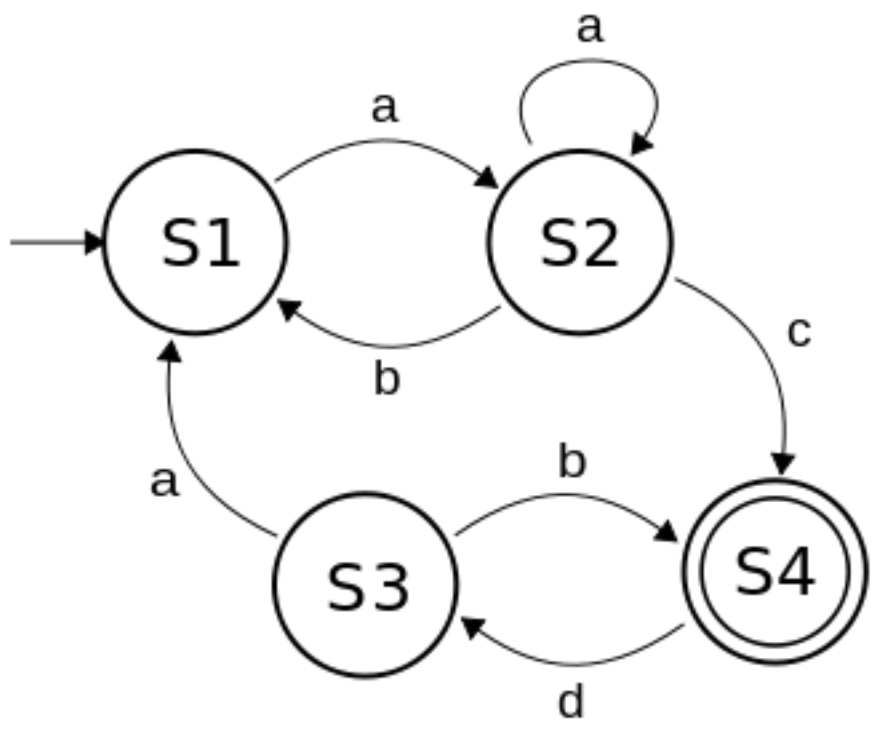
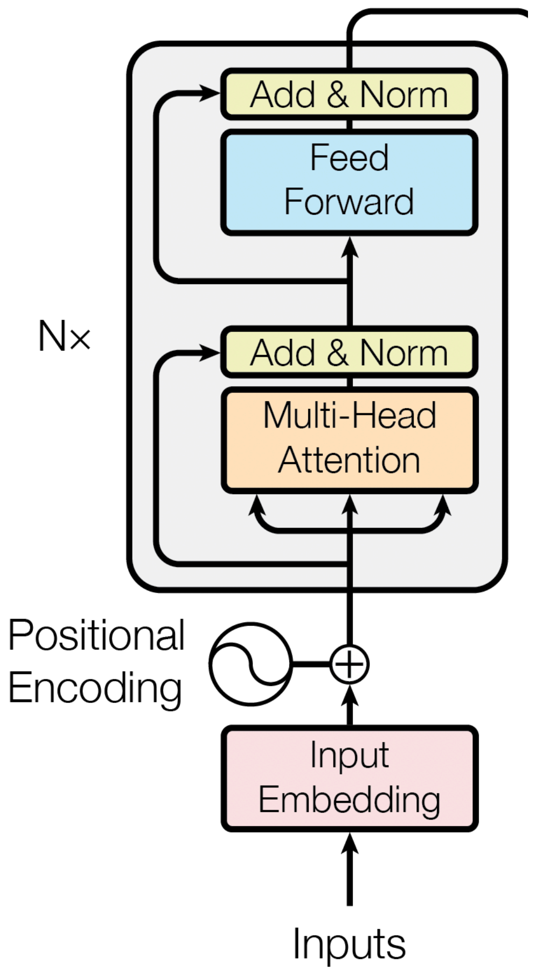
Thinking Like Transformers



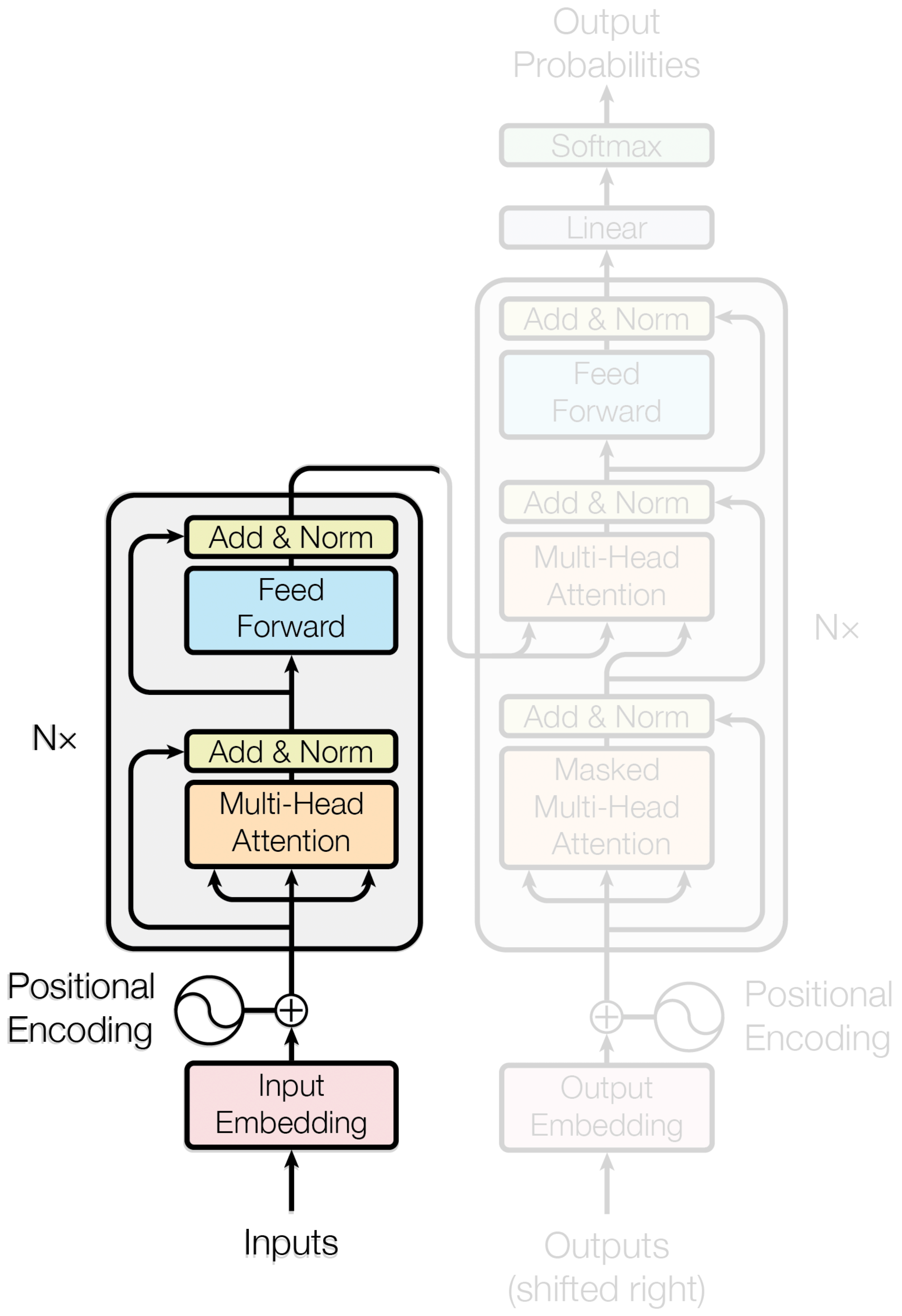


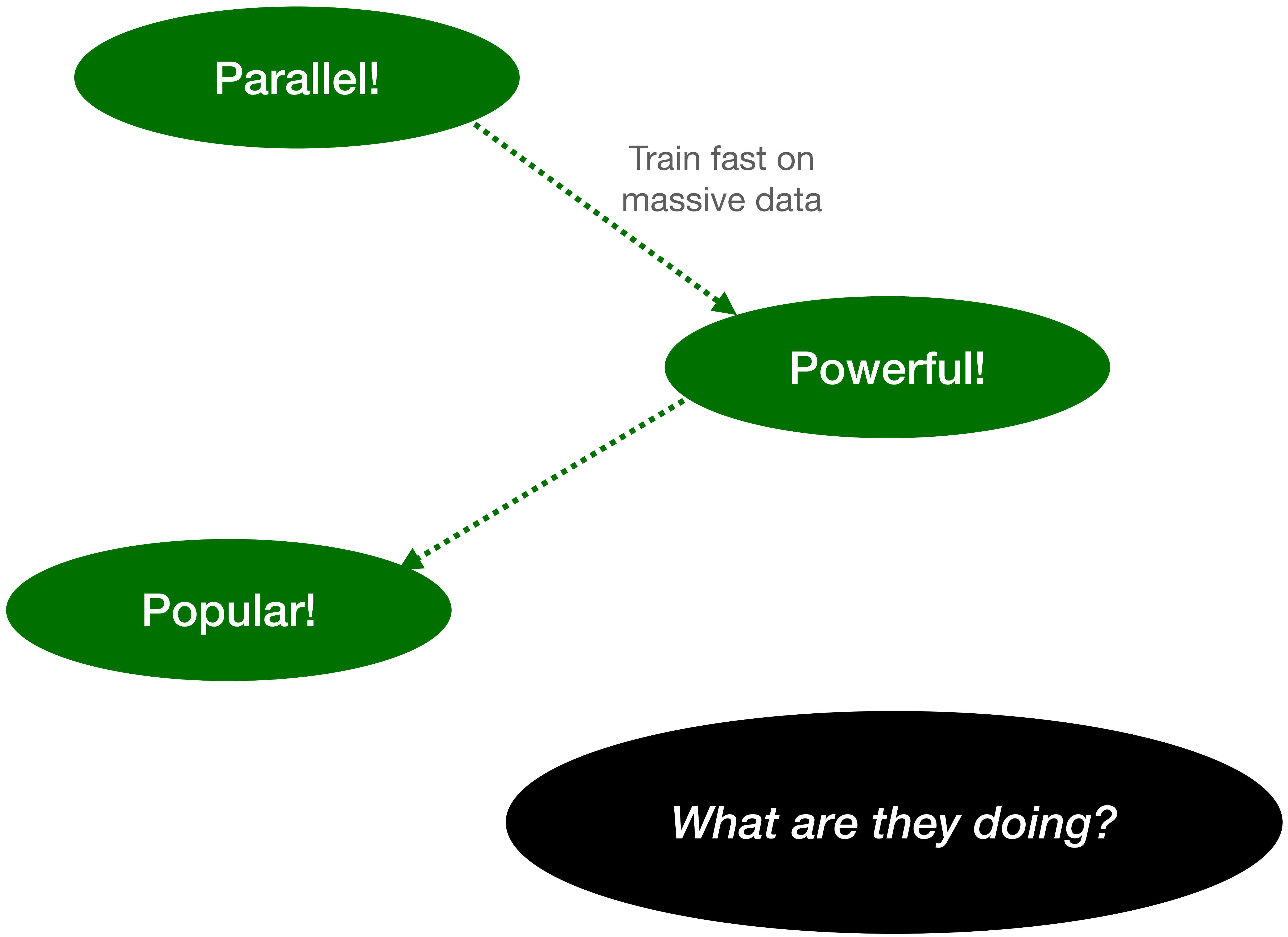


?

Gail Weiss, Yoav Goldberg, Eran Yahav

Motivation: Transformer Encoders

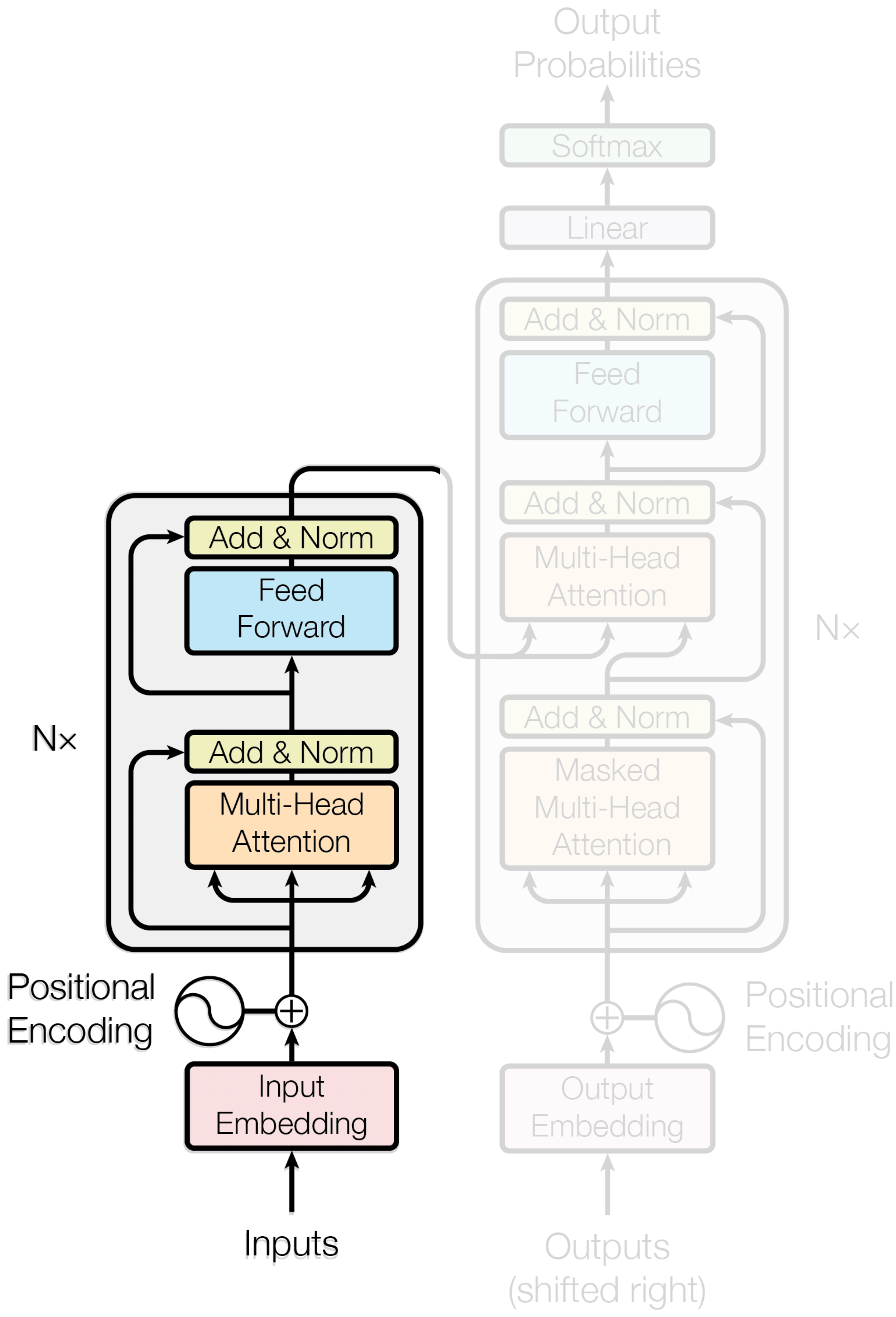




Attention Is All You Need

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin

Motivation: Transformer Encoders



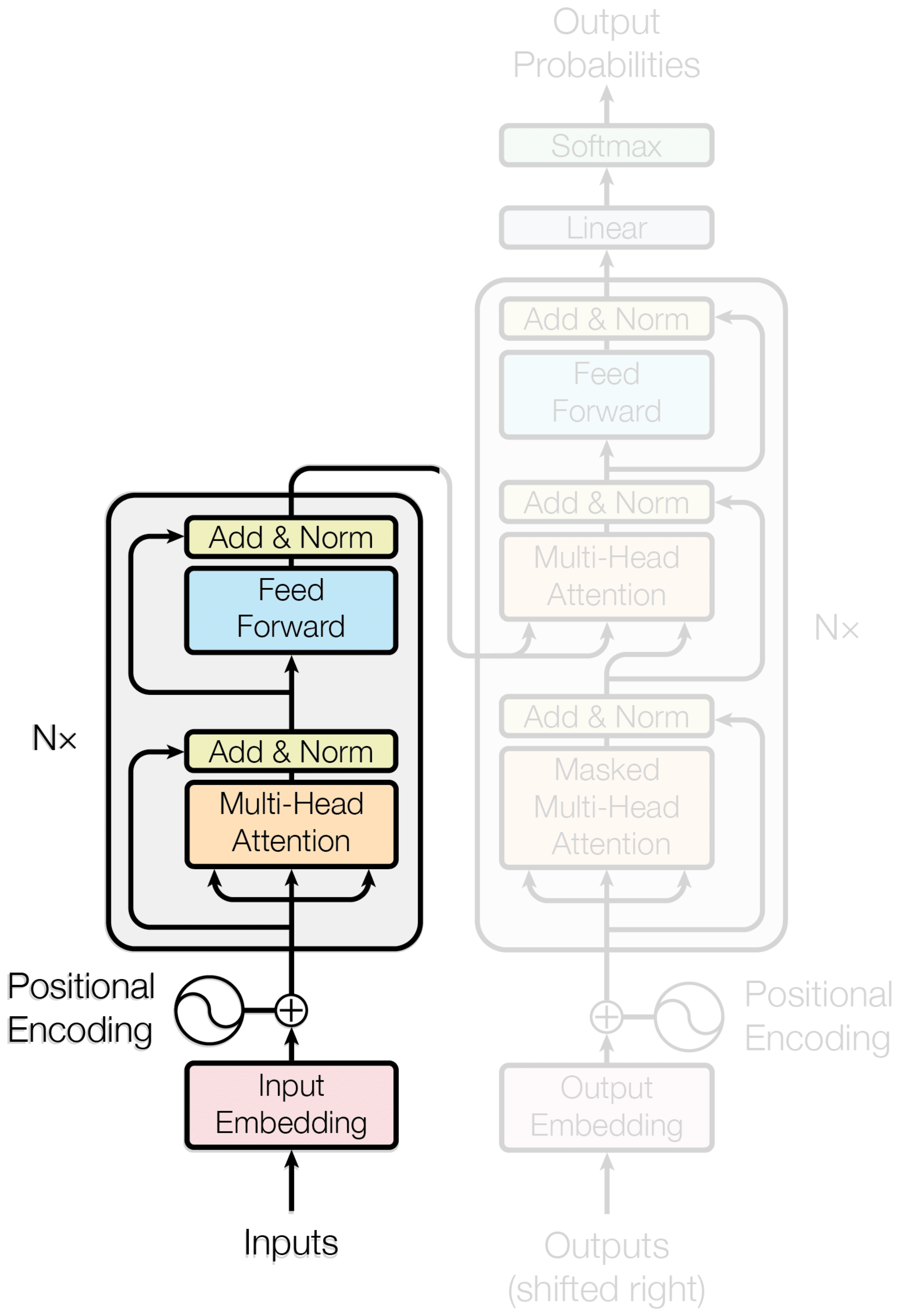
What are they doing?

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

Attention Is All You Need

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin

Motivation: Transformer Encoders



What are they doing?

We're figuring out all kinds of things…

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

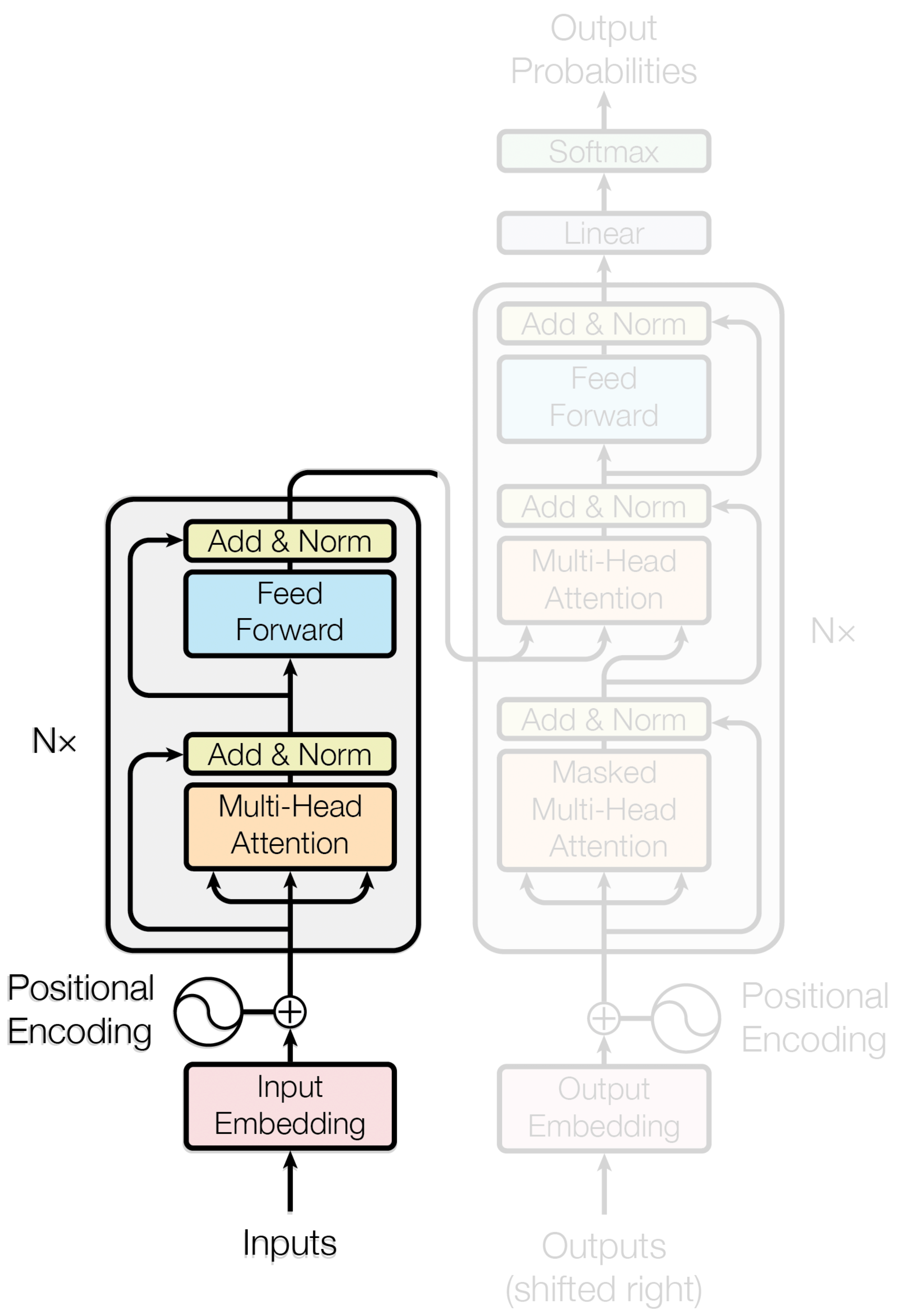
Chulhee Yun, Srinadh Bhojanapalli, Ankit Singh Rawat, Sashank J. Reddi, Sanjiv Kumar

Attention Is All You Need

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin

…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

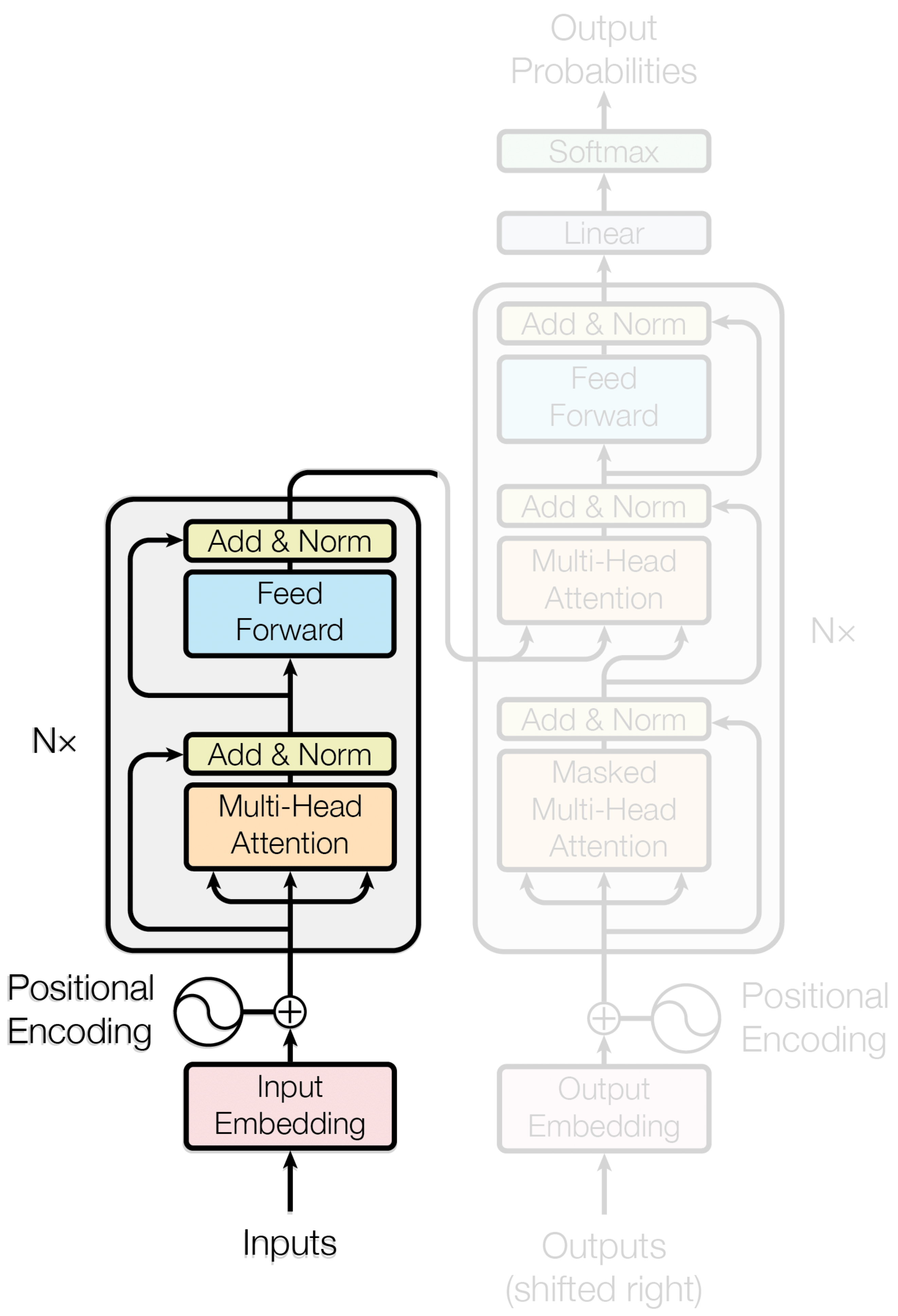
Michael Hahn

Attention Is All You Need

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin

…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

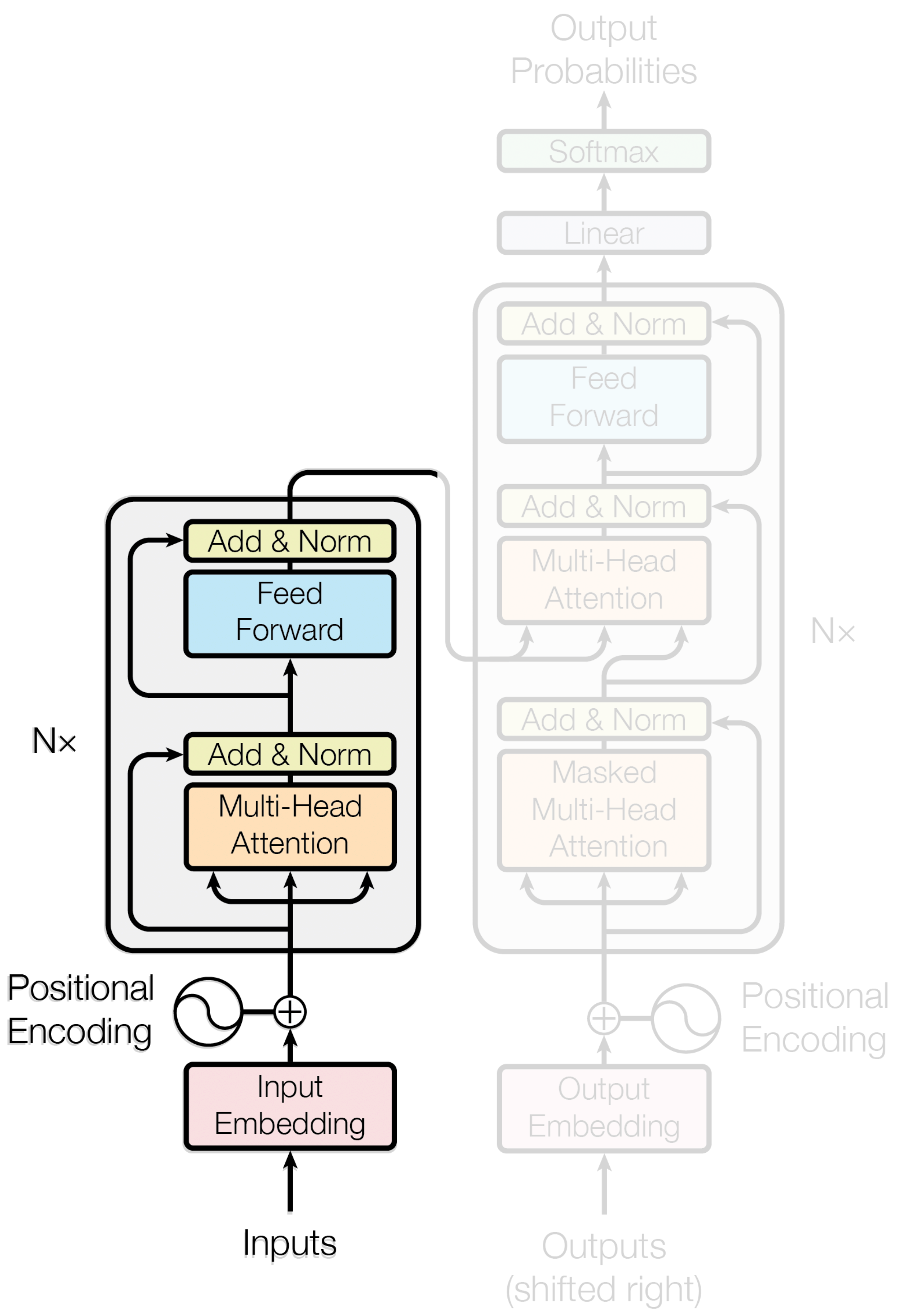
Satwik Bhattamishra, Kabir Ahuja, Navin Goyal

Attention Is All You Need

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin

...but it would be nice to have a model!

Motivation: Transformer Encoders



Attention Is All You Need

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin

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 Bhattamishra, Kabir Ahuja, Navin Goyal

Attention is Turing-Complete

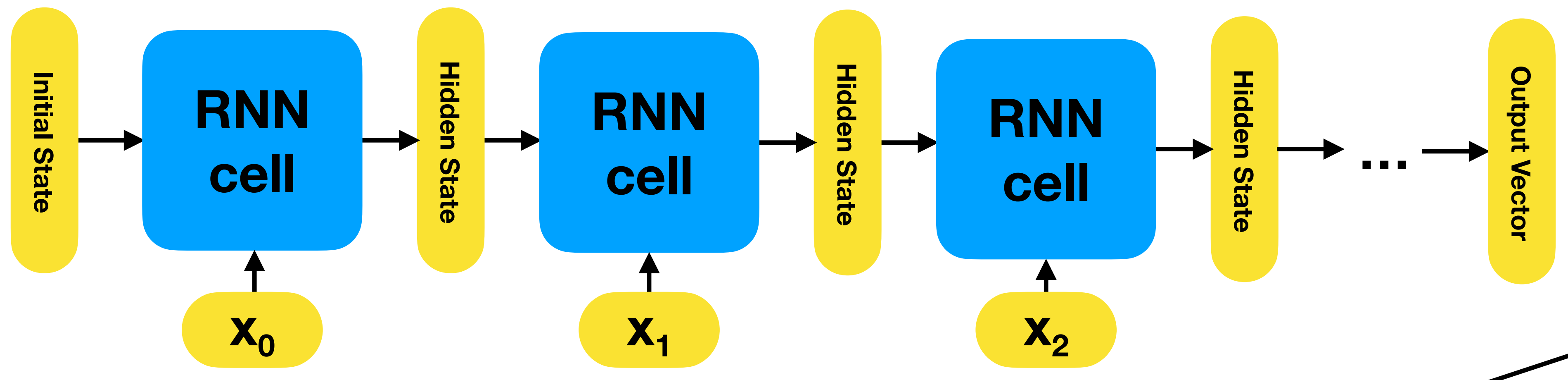
Jorge Pérez, Pablo Barceló, Javier Marinkovic; 22(75):1−35, 2021.

Statistically Meaningful Approximation: a Case Study on Approximating Turing Machines with Transformers

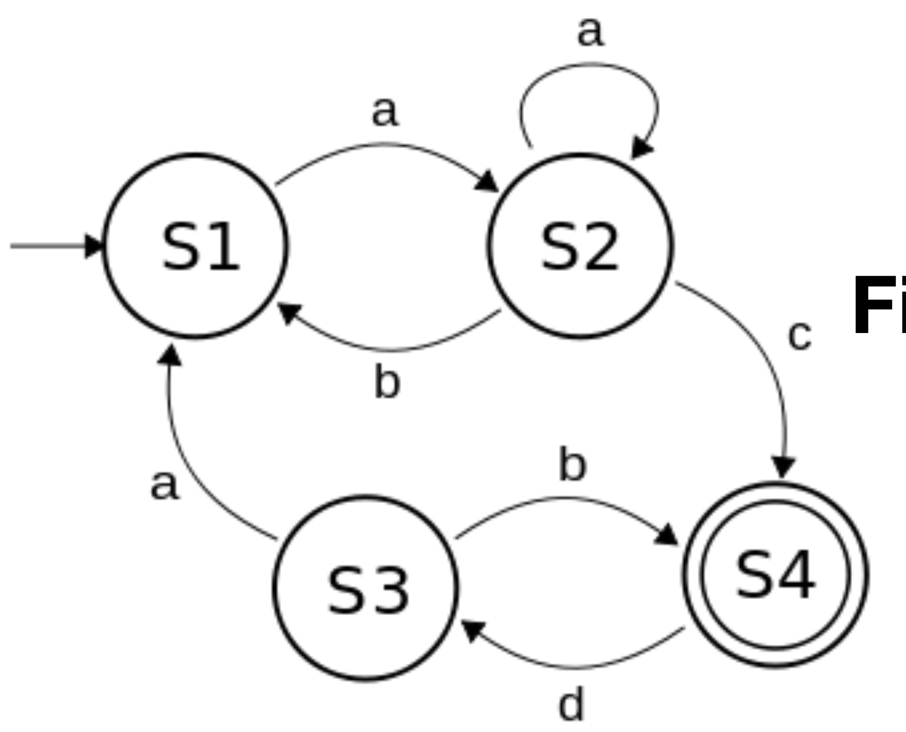
Colin Wei, Yining Chen, Tengyu Ma

…but it would be nice to have a model!

Motivation: What RNNs have



Computational Model(s)!



Deterministic Finite Automata (DFAs)

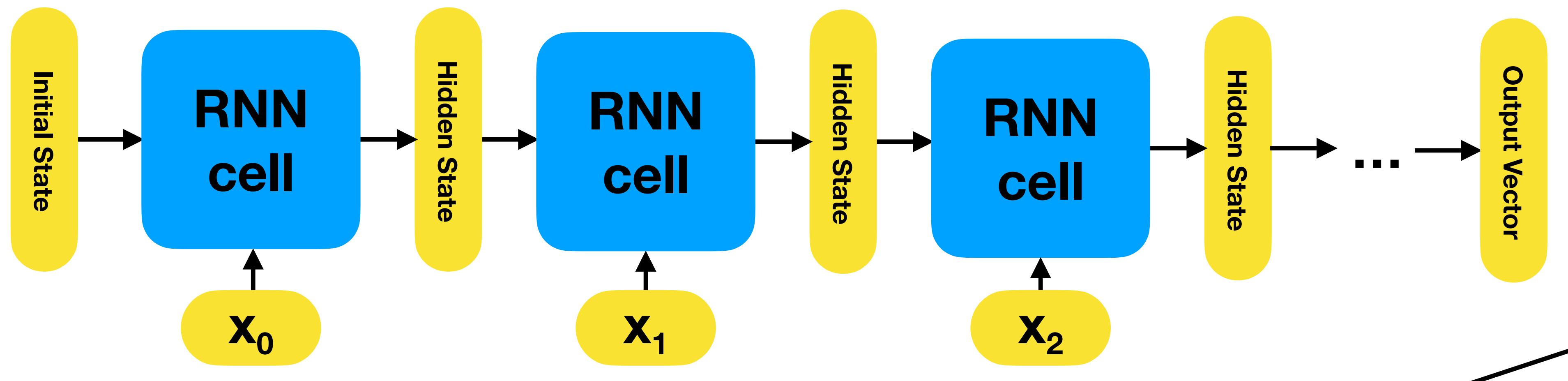
Extraction!

Spectral extraction:
RNNs to WFAs

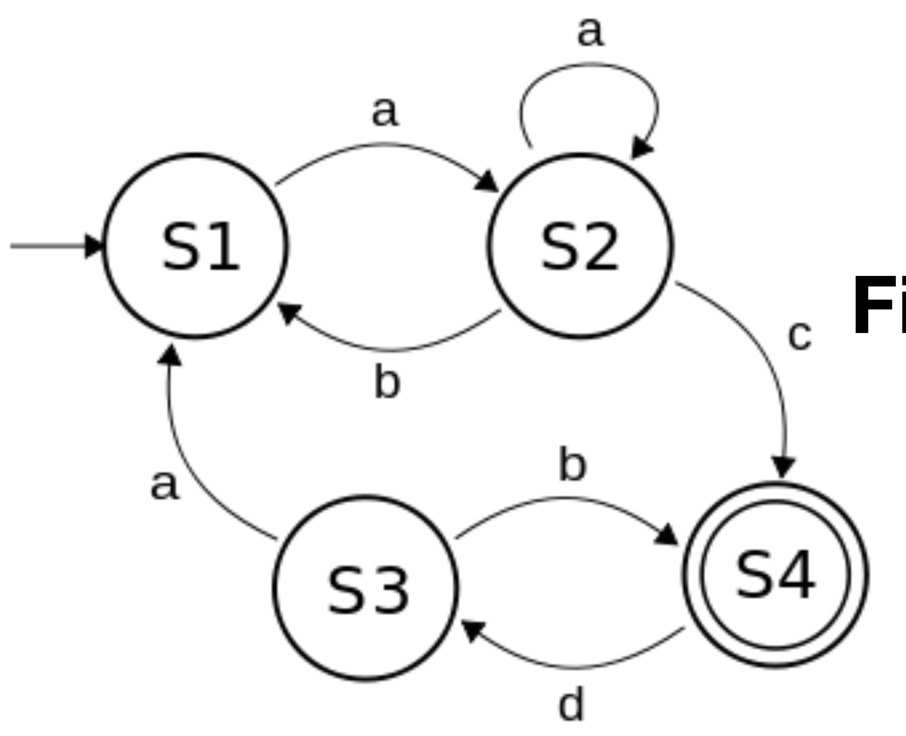
DFA extraction:
Clustering

DFA and WDFA extraction:
L-star variants

Motivation: What RNNs have



Computational Model(s)!



Deterministic Finite Automata (DFAs)

Extraction!

Spectral extraction:
RNNs to WFAs

DFA extraction:
Clustering

DFA and WDFA extraction:
L-star variants

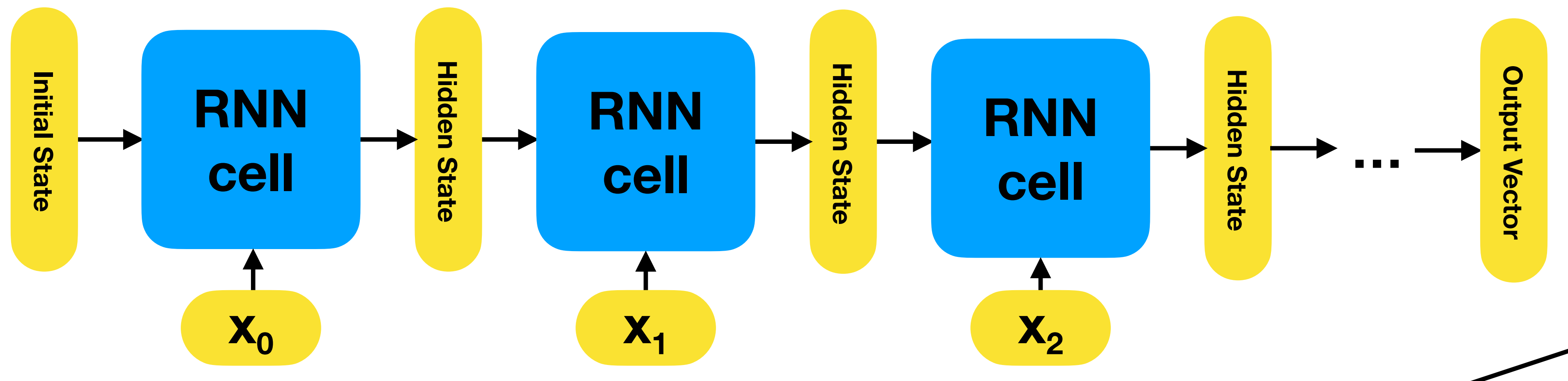
Analysis of Expressive Power!

2-RNNs are WFAs

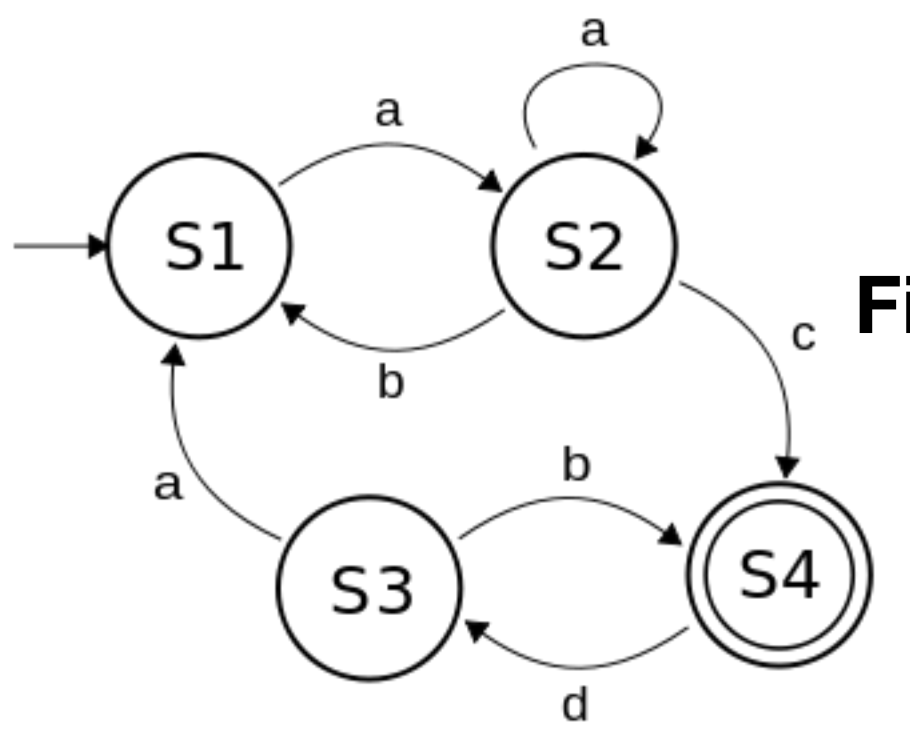
LSTMs are counter machines

GRUs are DFAs

Motivation: What RNNs have



Computational Model(s)!



Deterministic Finite Automata (DFAs)

Extraction!

Spectral extraction:
RNNs to WFAs

DFA extraction:
Clustering

DFA and WDFA extraction:
L-star variants

Analysis of Expressive Power!

2-RNNs are WFAs

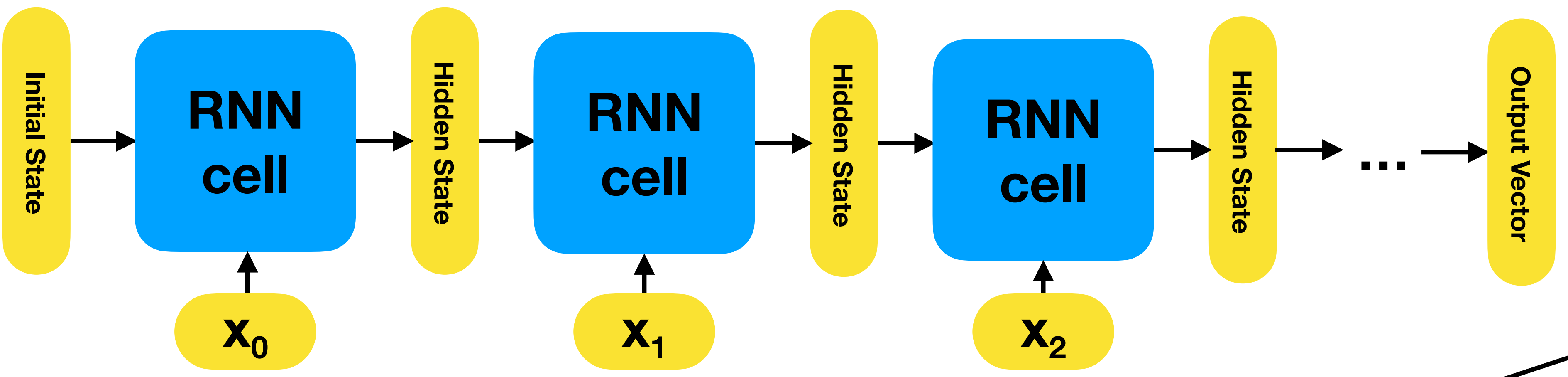
LSTMs are counter machines

GRUs are DFAs

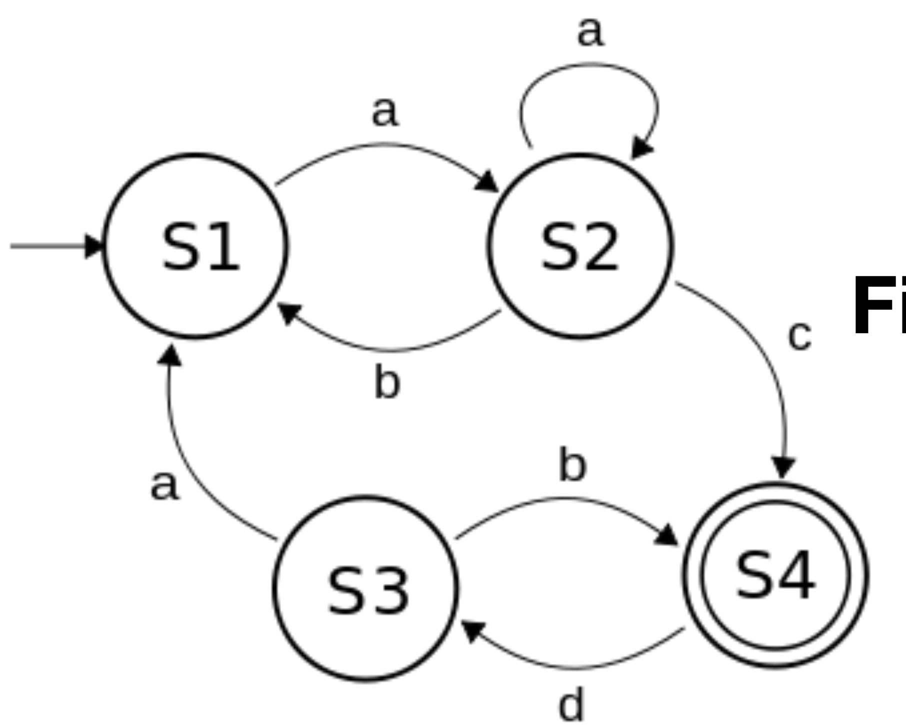
Inspiration from existing theory!

Stack-RNNs

Motivation: What RNNs have



Computational Model(s)!



Deterministic Finite Automata (DFAs)

Extraction!

Spectral extraction: RNNs to WFAs

DFA extraction: Clustering

DFA and WDFA extraction: L-star variants

Analysis of Expressive Power!

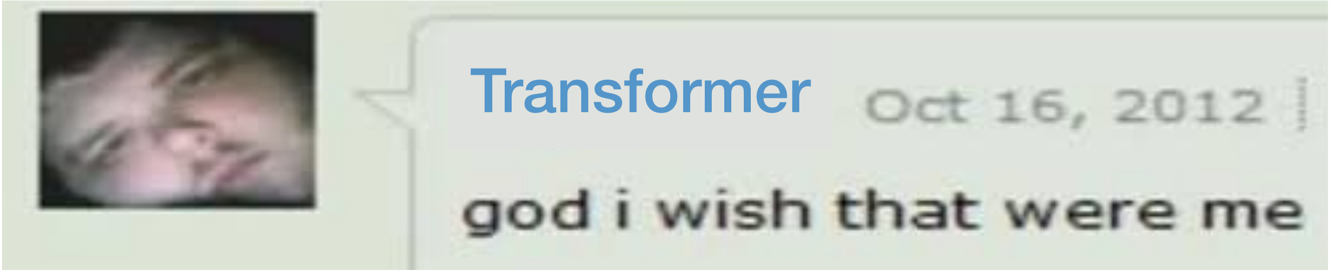
2-RNNs are WFAs

LSTMs are counter machines

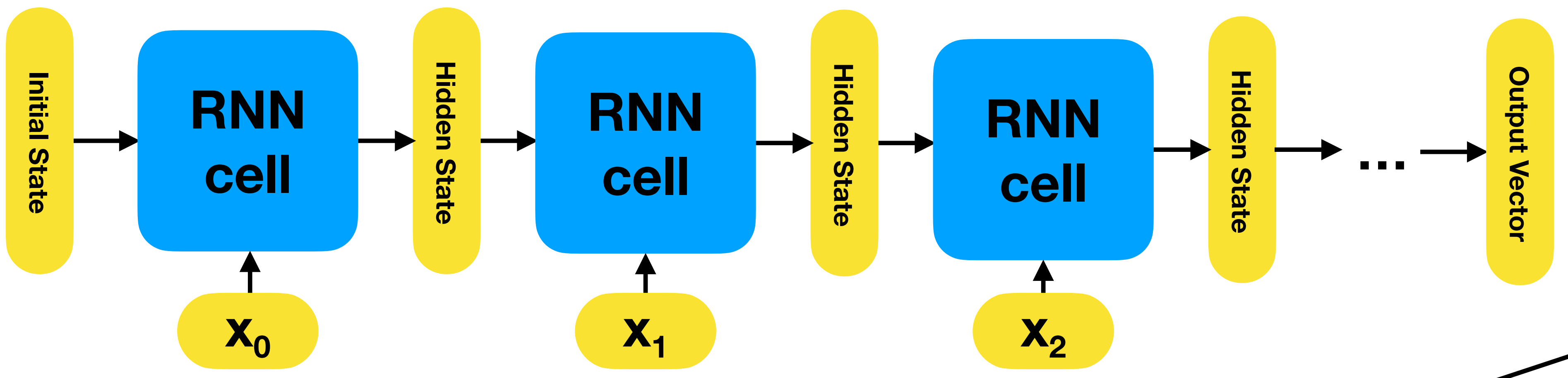
GRUs are DFAs

Inspiration from existing theory!

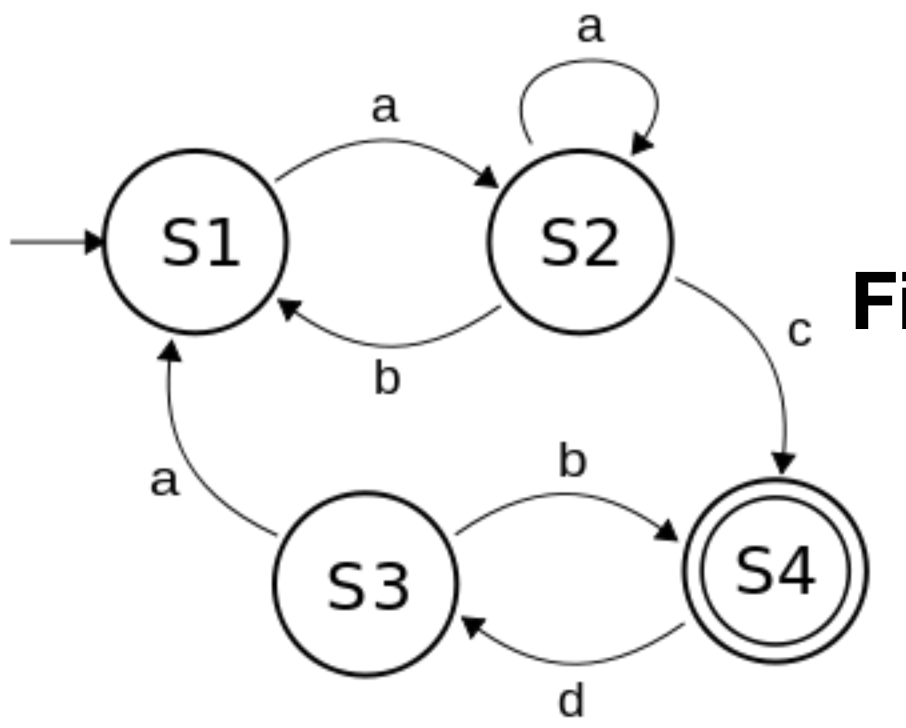
Stack-RNNs



(References for the Interested)



Computational Model(s)!



Deterministic Finite Automata (DFAs)

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

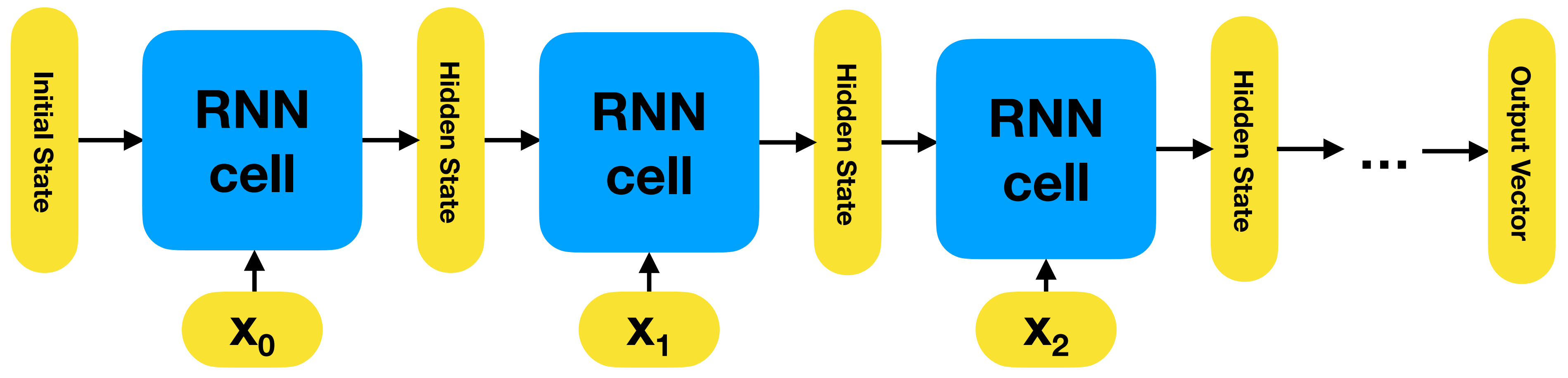
A Formal Hierarchy of RNN Architectures

Inspiration from existing theory!

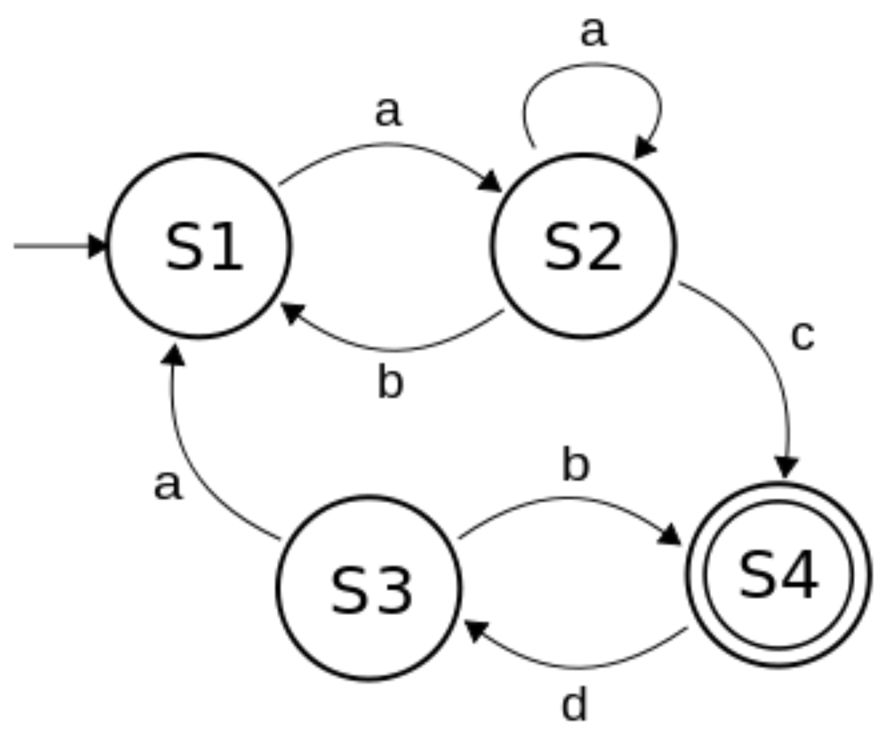
Inferring Algorithmic Patterns with Stack-Augmented Recurrent Nets

Learning to Transduce with Unbounded Memory

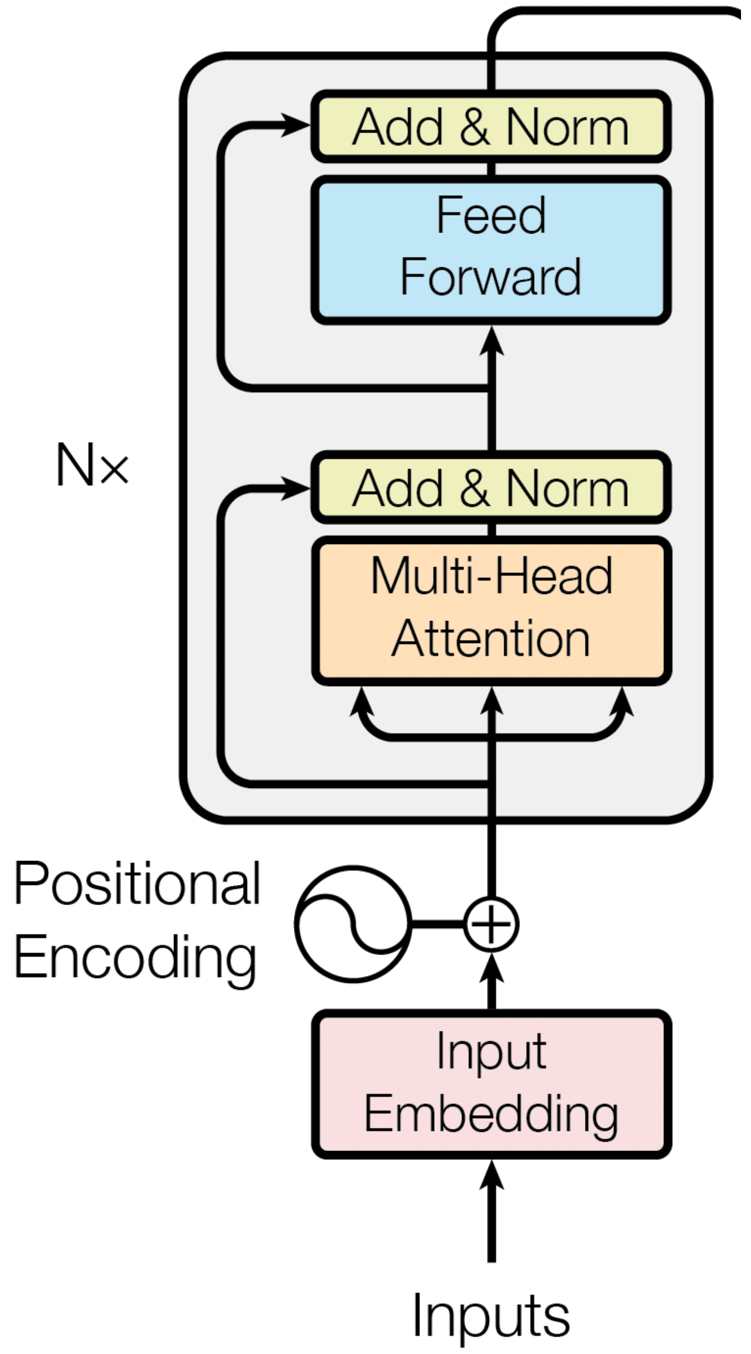
But what are Transformer-Encoders?



Computational Model(s)!



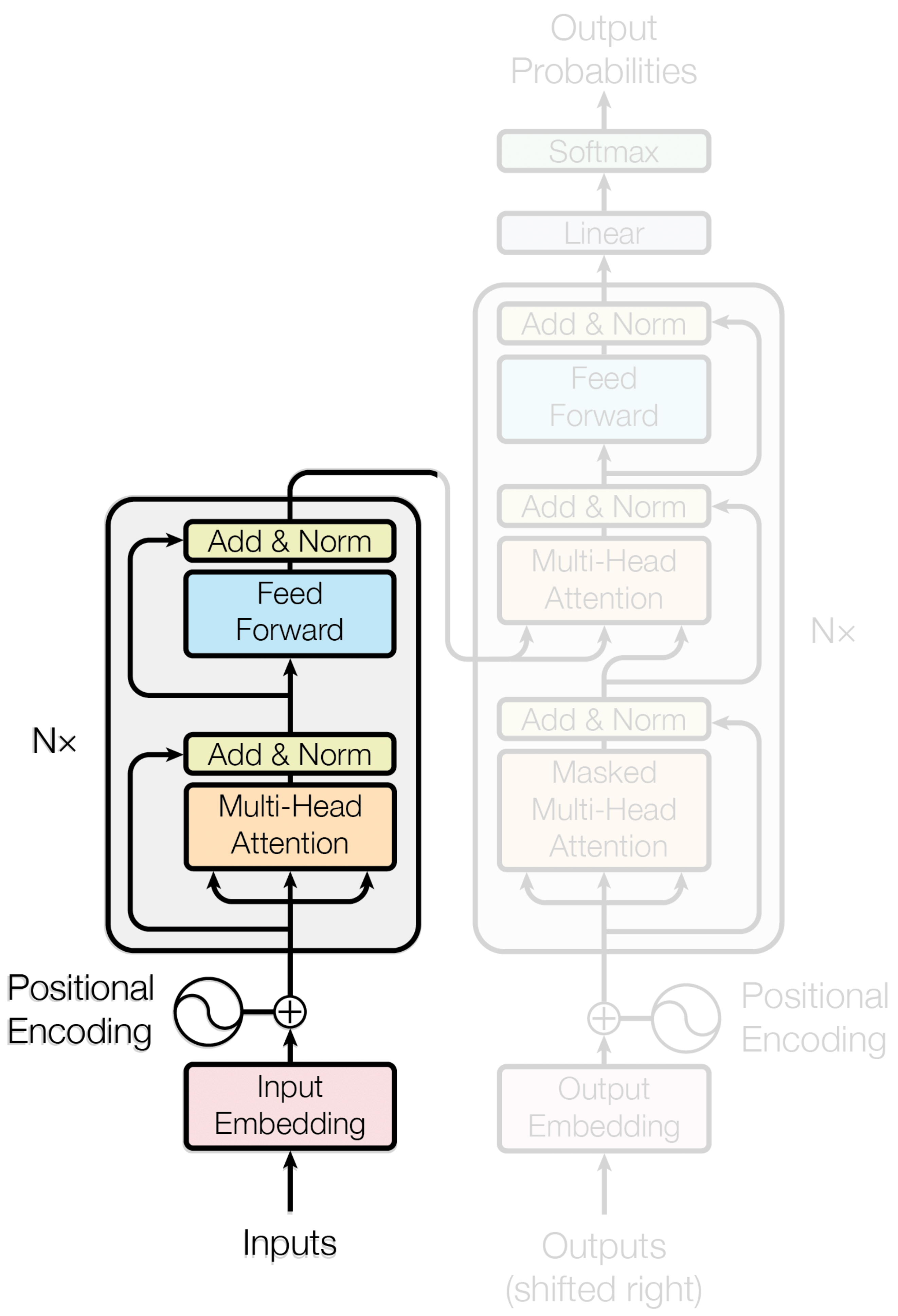
Transformer-Encoder



?

Any ideas?

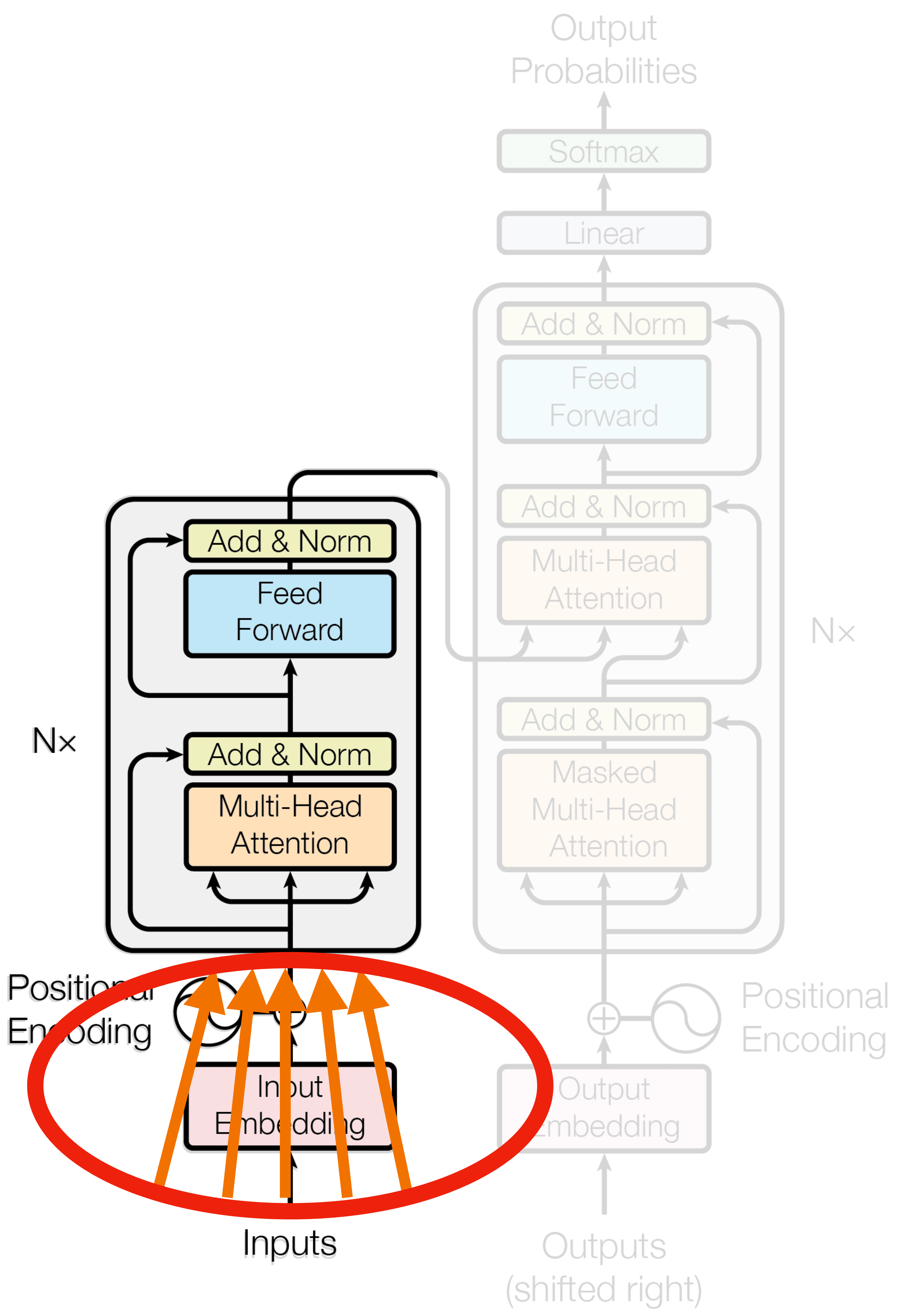
Transformer Encoders



Attention Is All You Need

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin

Transformer Encoders

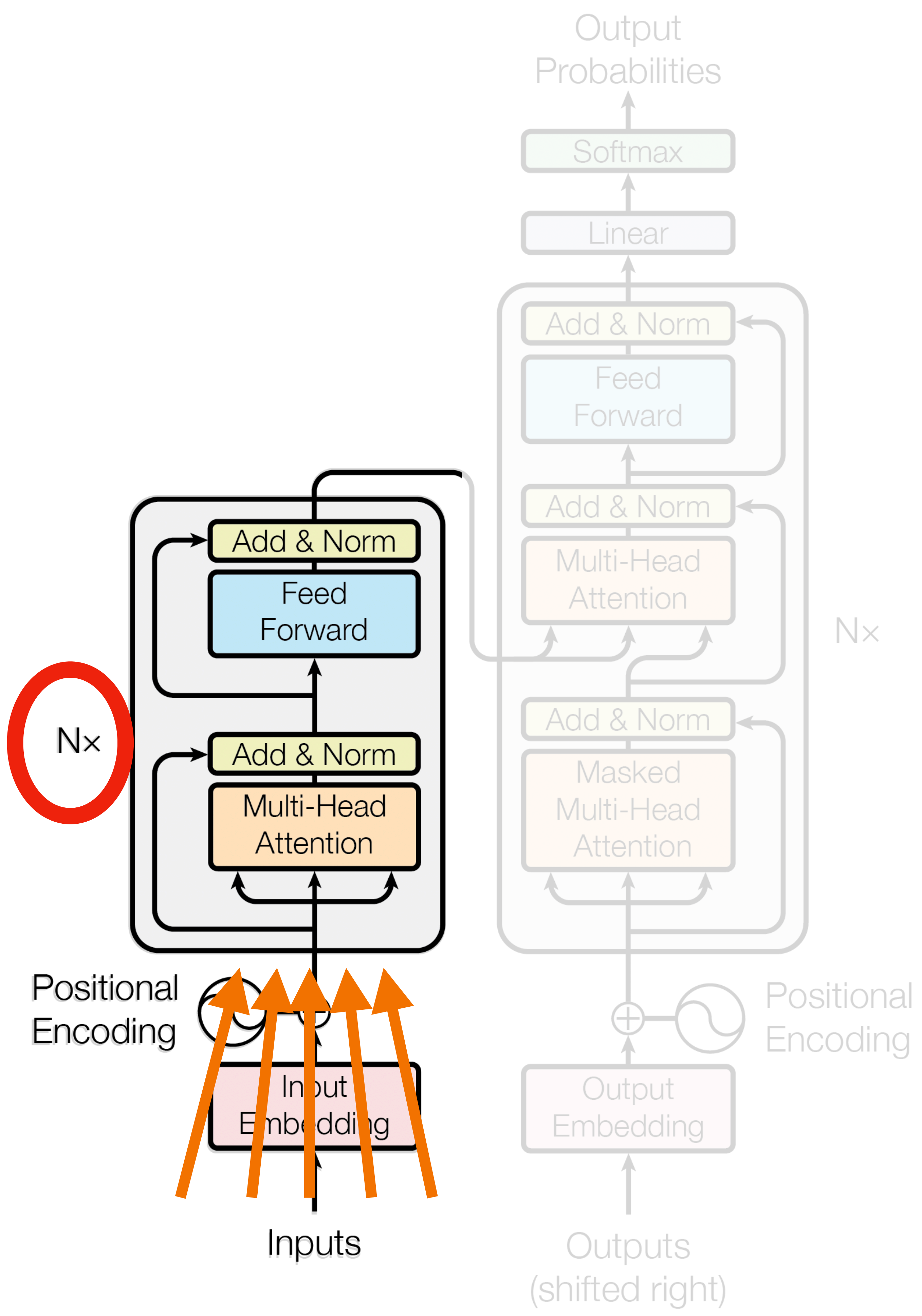


- Receive their entire input 'at once', processing all tokens in parallel

Attention Is All You Need

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin

Transformer Encoders

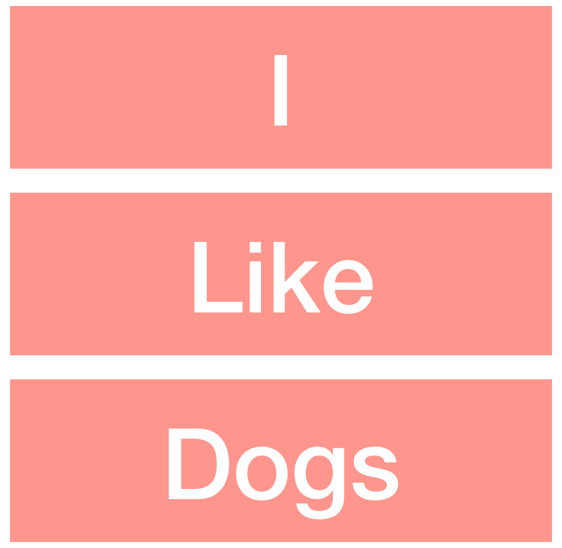


- Receive their entire input 'at once', processing all tokens in parallel
- Have a fixed number of layers, where the output of one is the input of the next

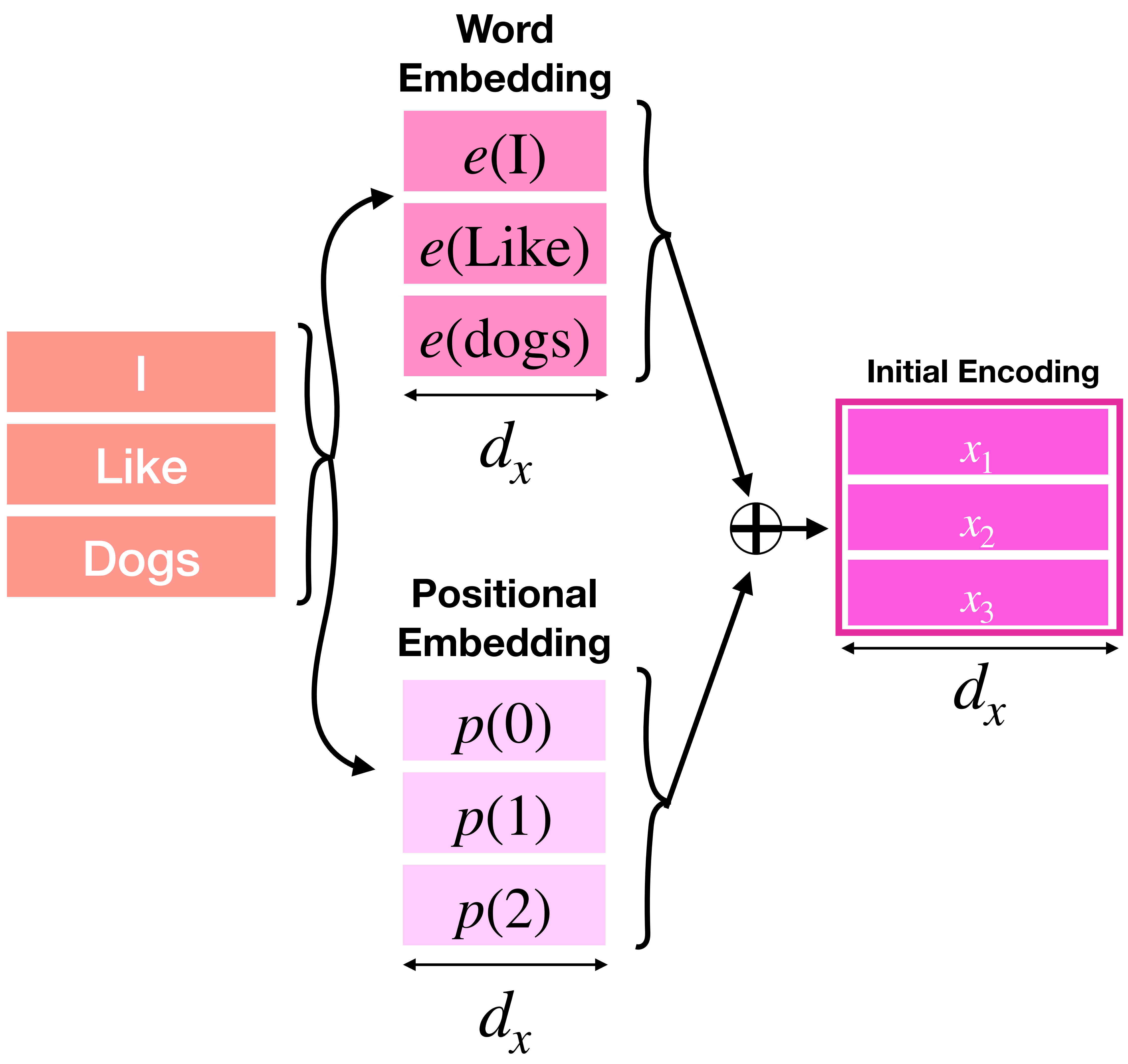
Attention Is All You Need

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin

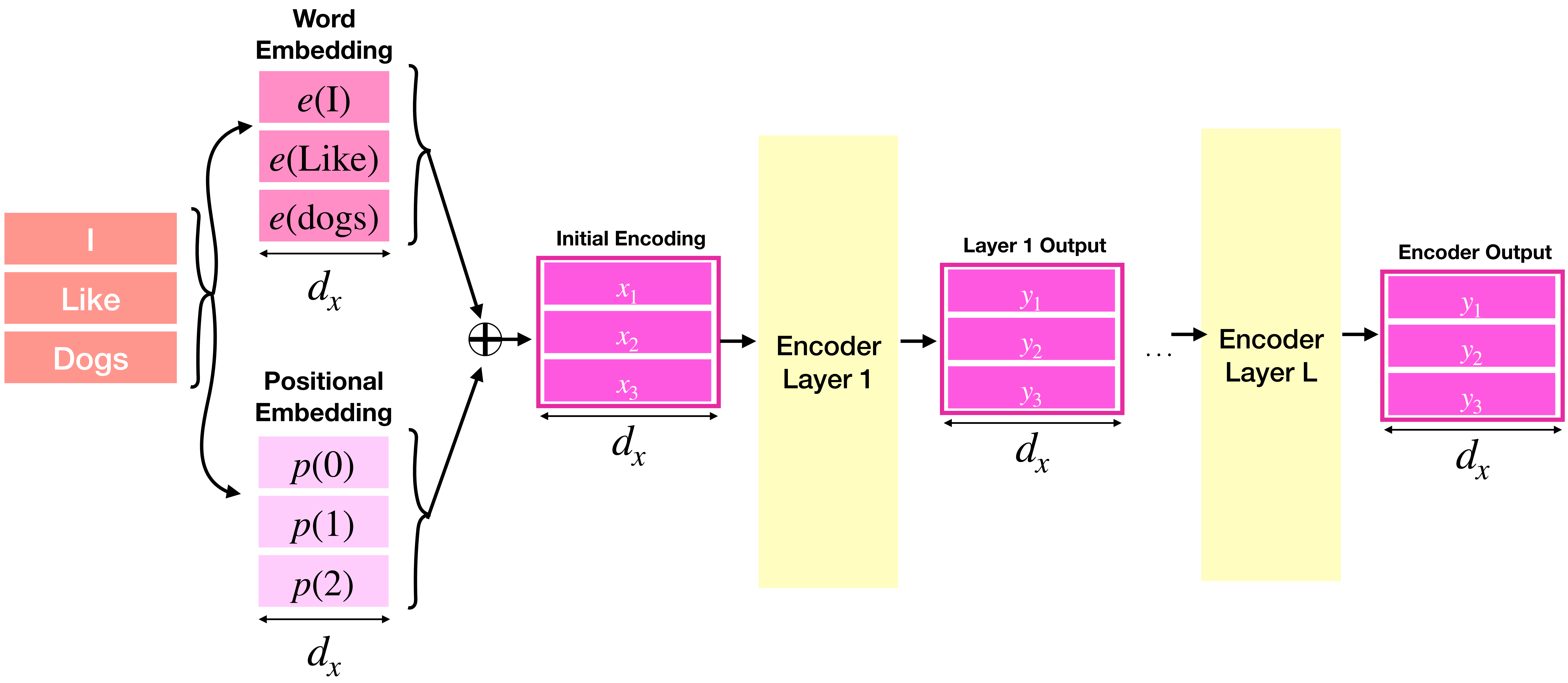
The Transformer-Encoder



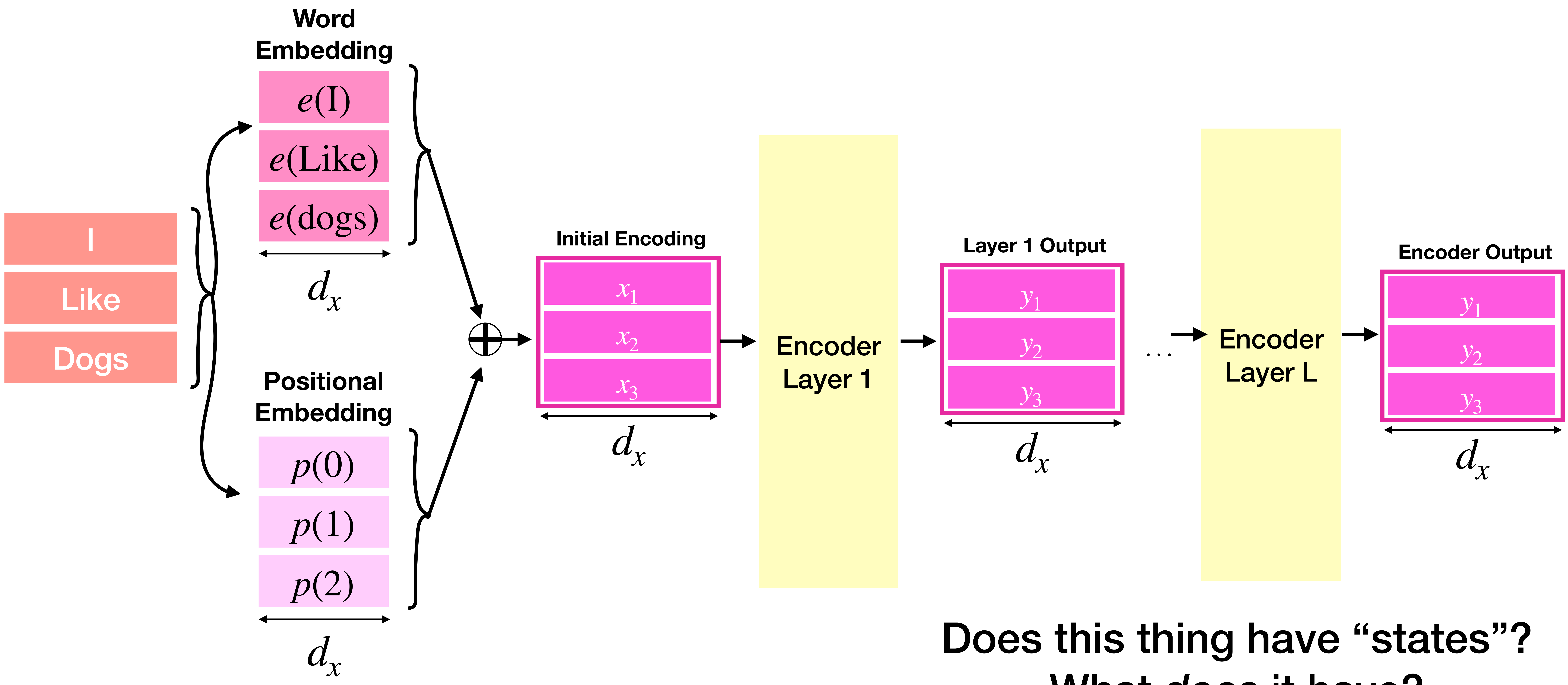
The Transformer-Encoder



The Transformer-Encoder



The Transformer-Encoder



Does this thing have "states"?
What *does* it have?

The Transformer-Encoder

No states… but there is a sense of information being propagated

The Transformer-Encoder

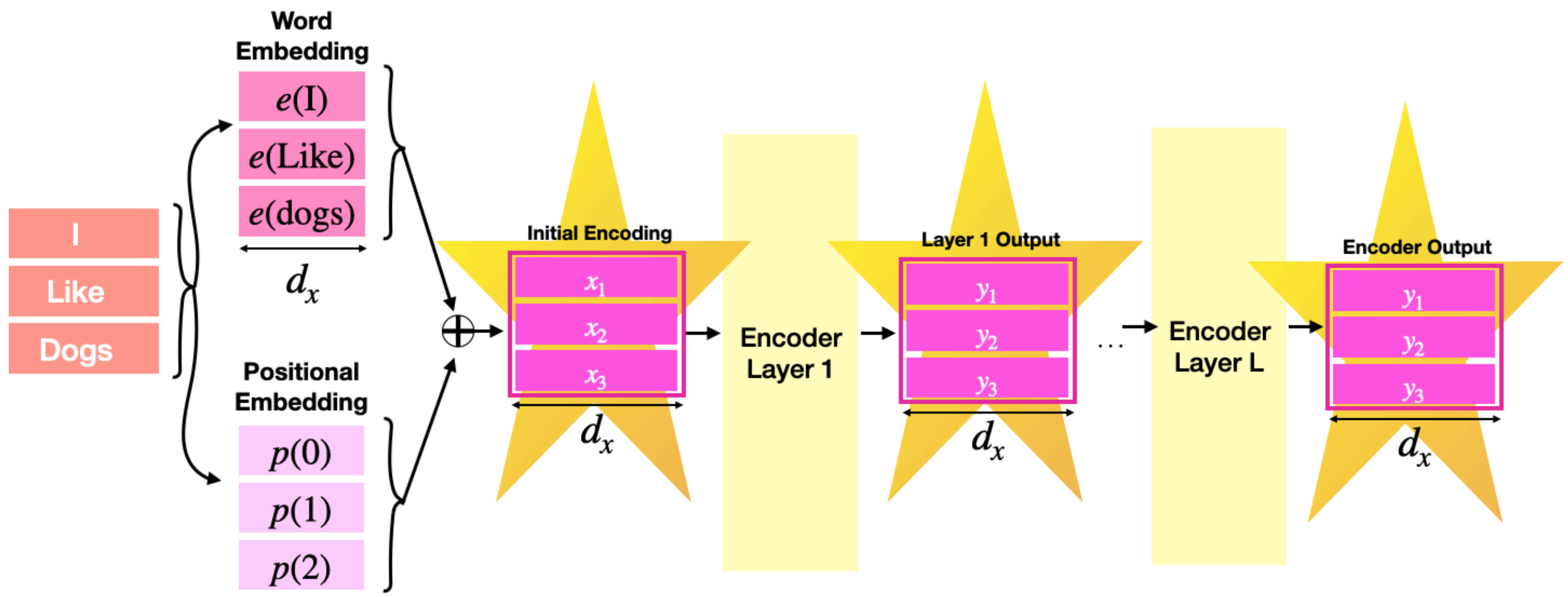
Layer input/outputs are "variables" of a transformer "program"

The layers themselves are "operations"

So… a programming language?

RASP (Restricted Access Sequence Processing)

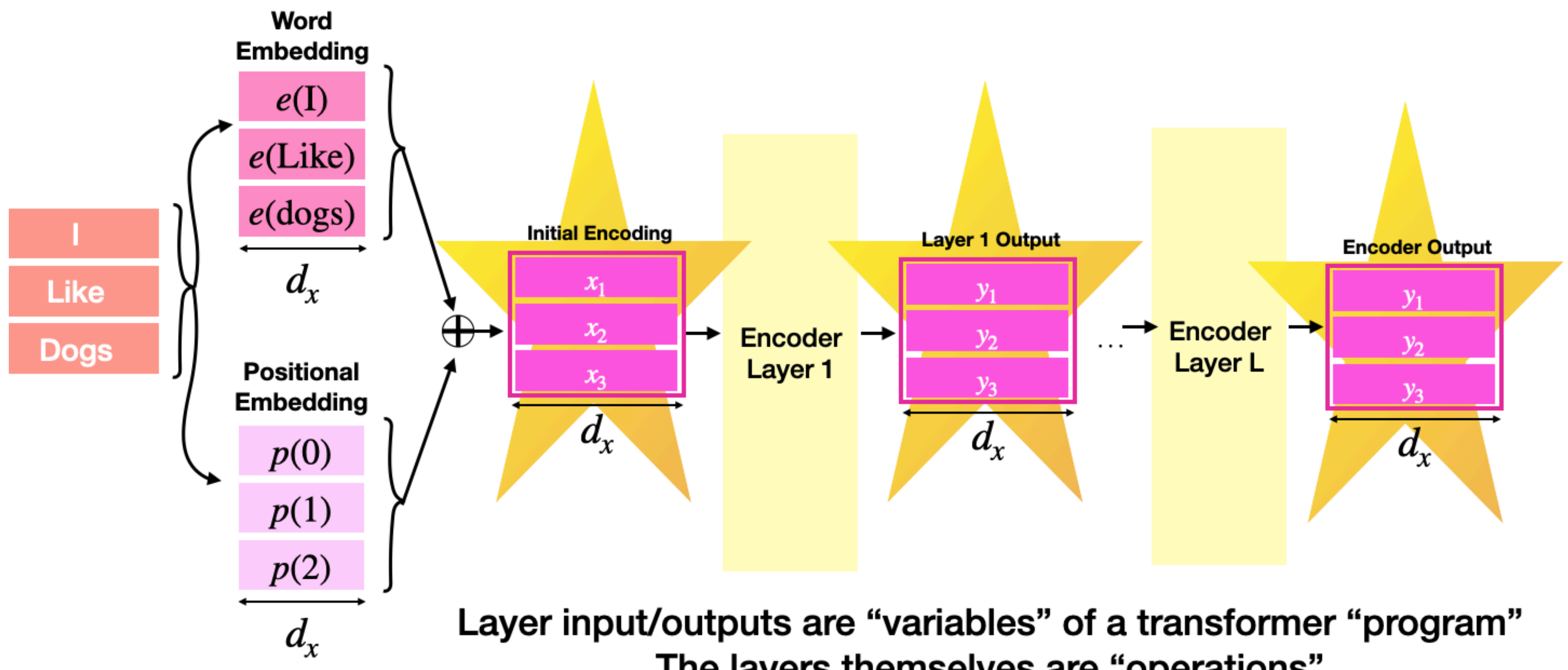
- A transformer-encoder is a sequence to sequence function (“sequence operator”, or, “s-op”)
- Its layers apply operations to the sequence
- RASP describes the input sequences and what the layers can do with them



Layer input/outputs are “variables” of a transformer “program”
The layers themselves are “operations”

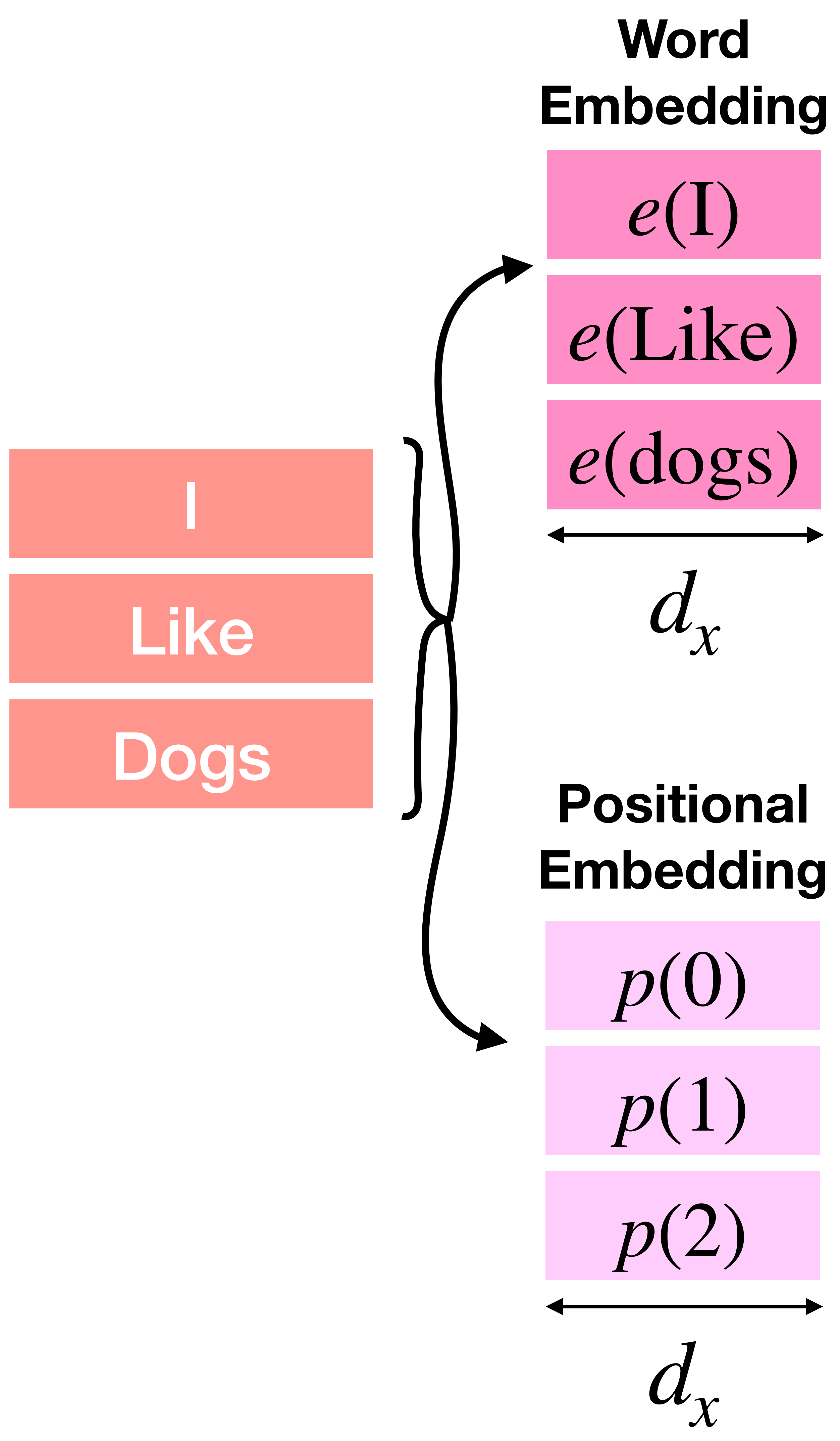
RASP (Restricted Access Sequence Processing)

- 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



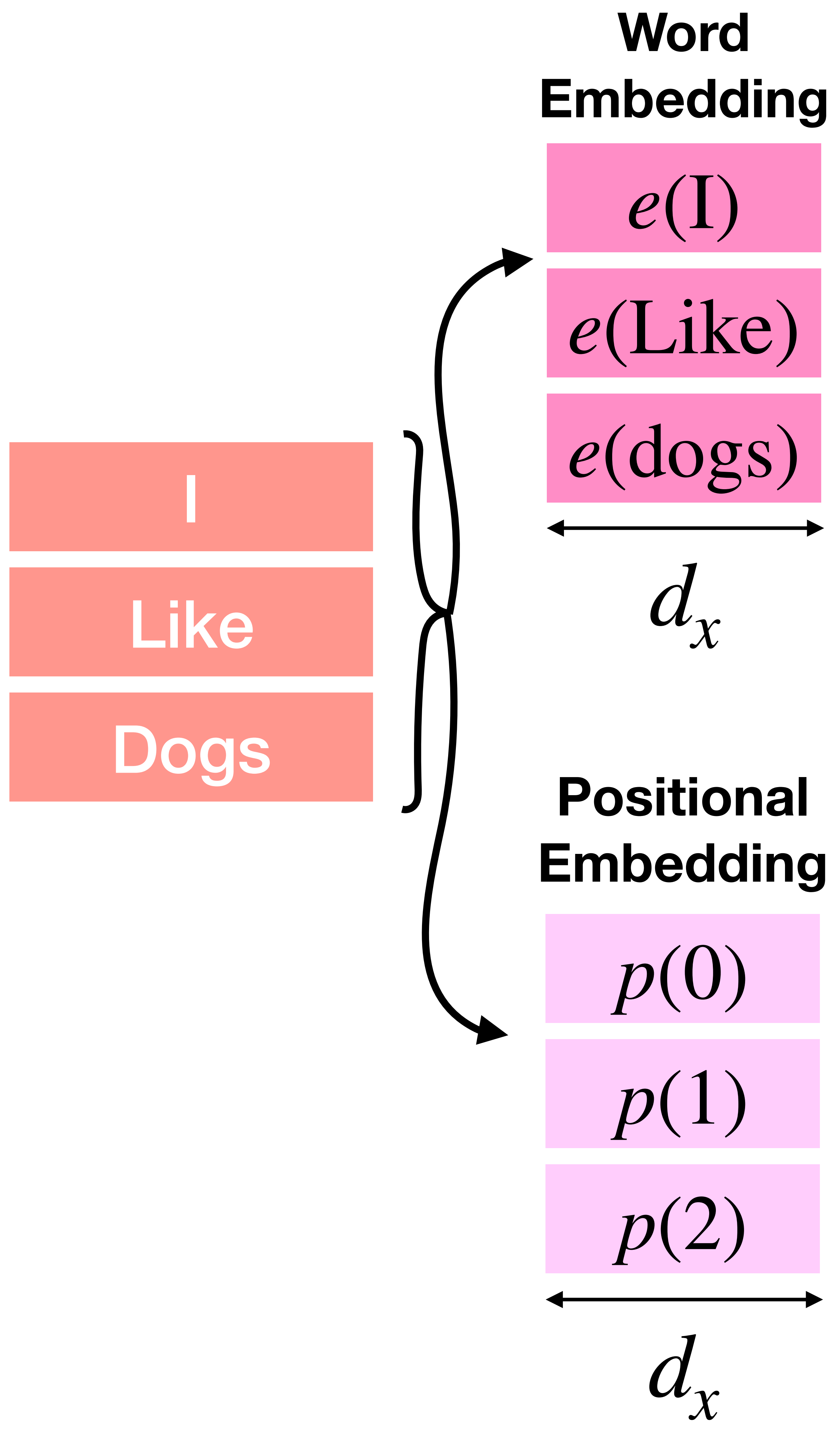
Layer input/outputs are “variables” of a transformer “program”
The layers themselves are “operations”

RASP base s-ops



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

RASP base s-ops



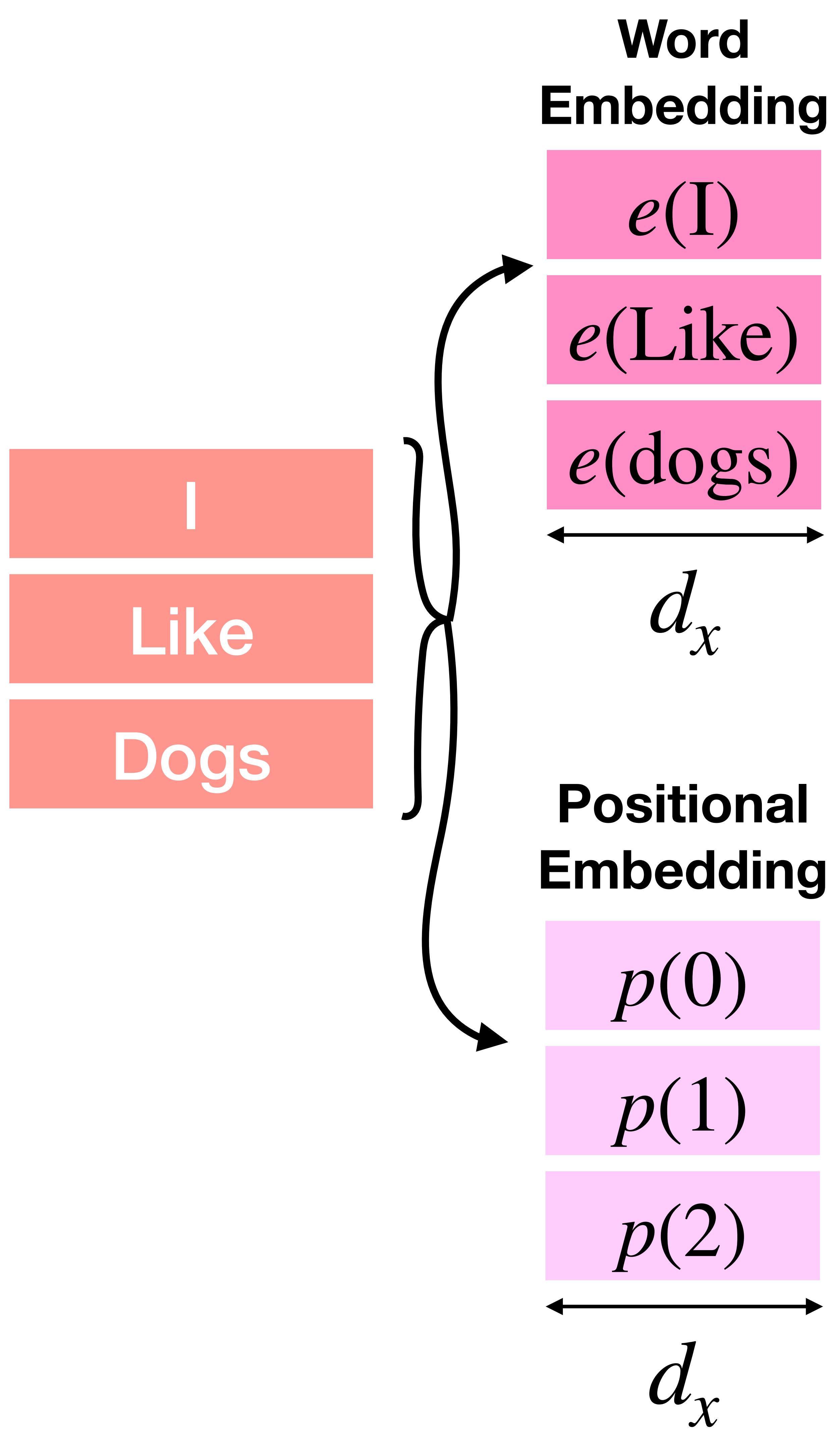
The information before a transformer has done anything ("0 layer transformer")

tokens and *indices* are RASP built-ins:

```
>> tokens;
     s-op: tokens

>> indices;
     s-op: indices
```

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
        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)
```

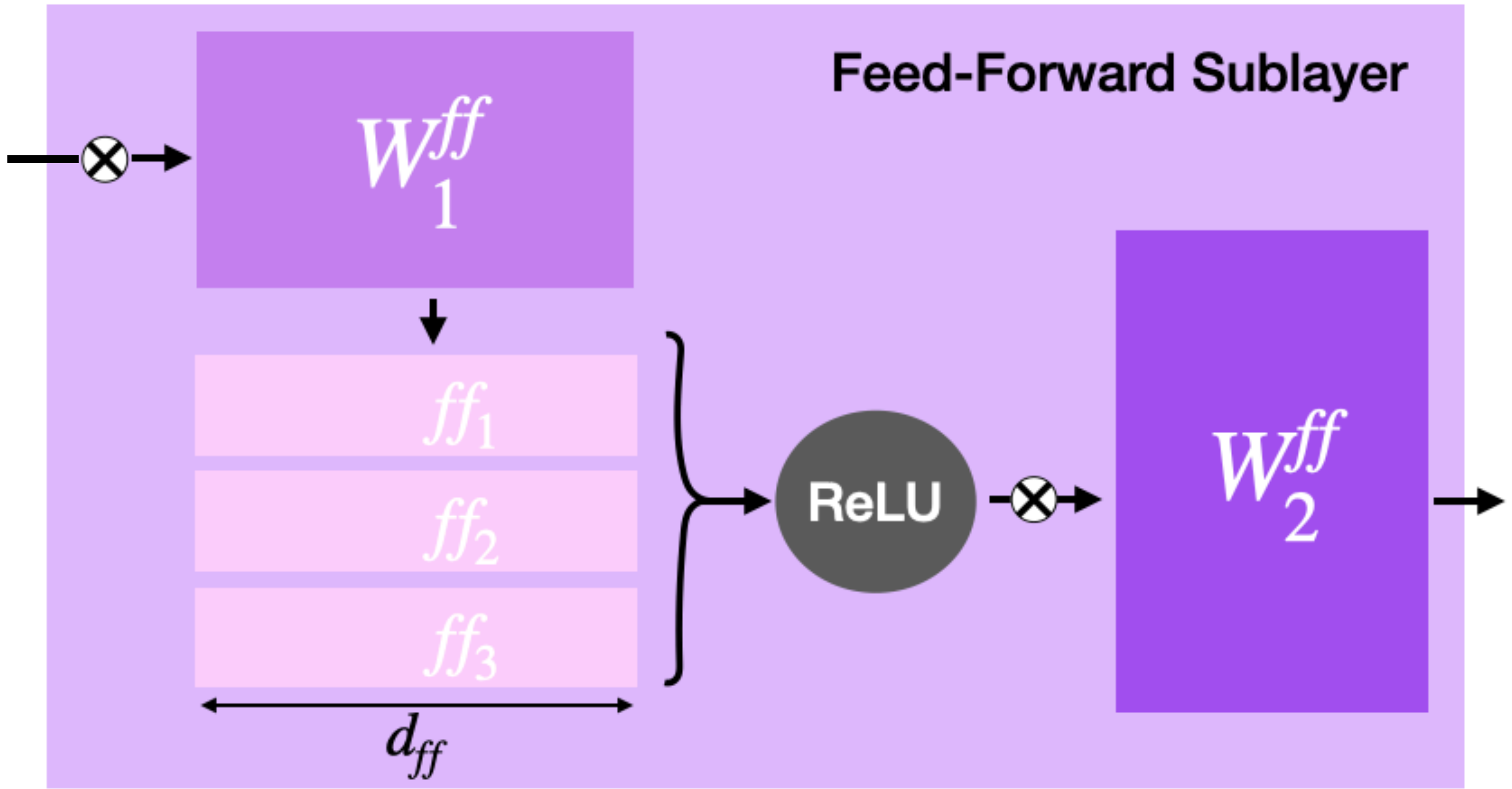
To know what operations RASP may have, we must inspect the transformer-encoder layers!

Transformer-Encoder Layer

Feed-Forward Sublayer

Feed-Forward Sublayer

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] (bools)
```

So far

```
>> 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?

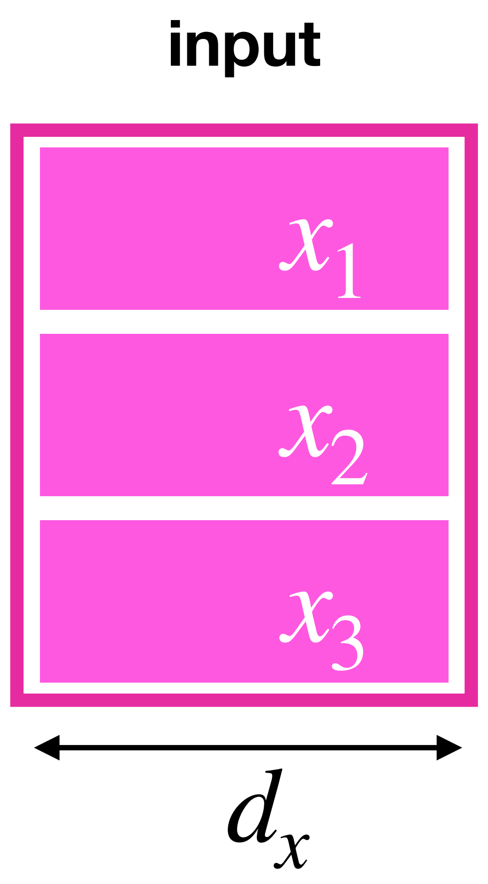
Transformer-Encoder Layer

Attention Sublayer

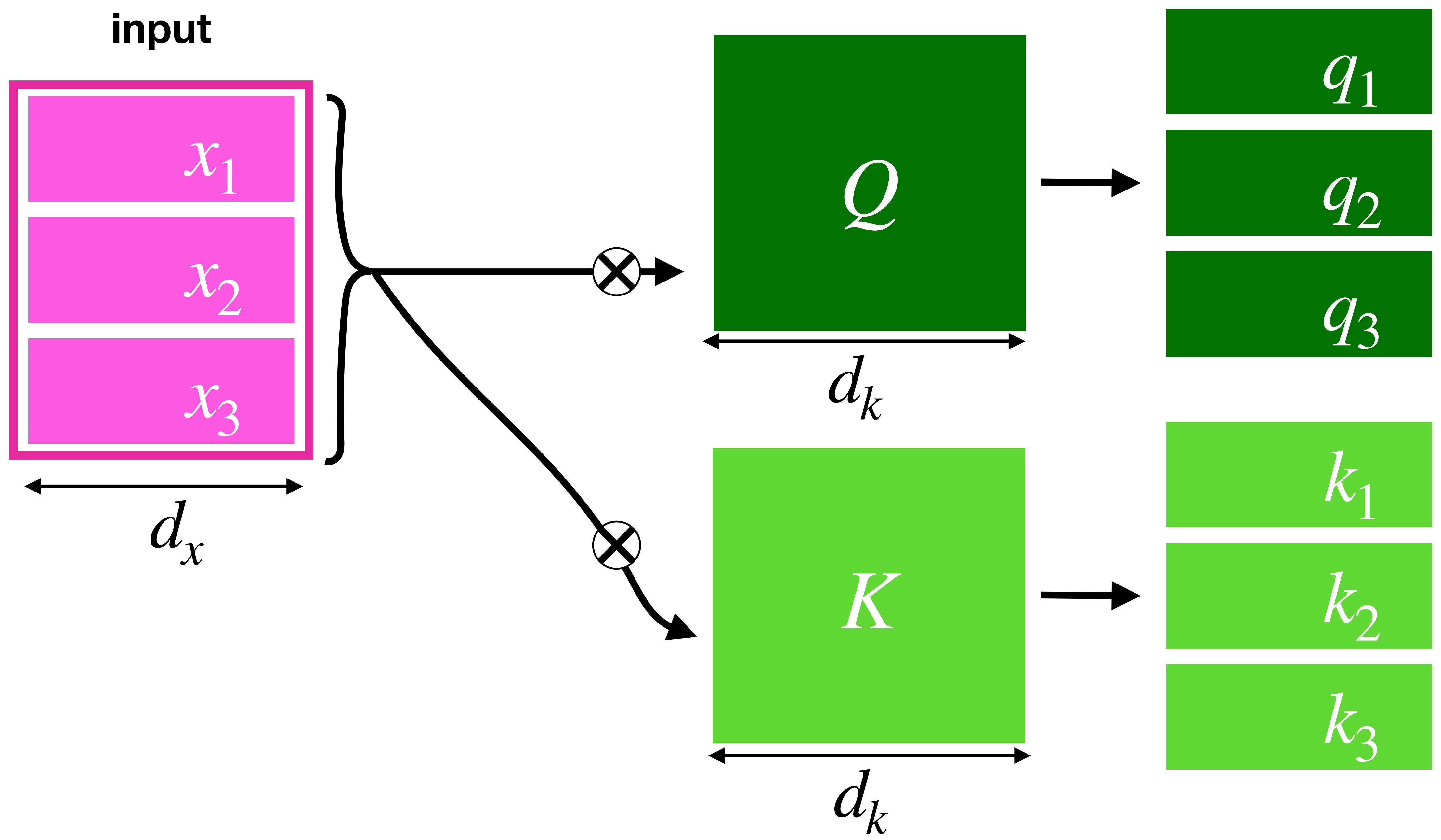
Background - Multi Head Attention

Starting from single-head attention…

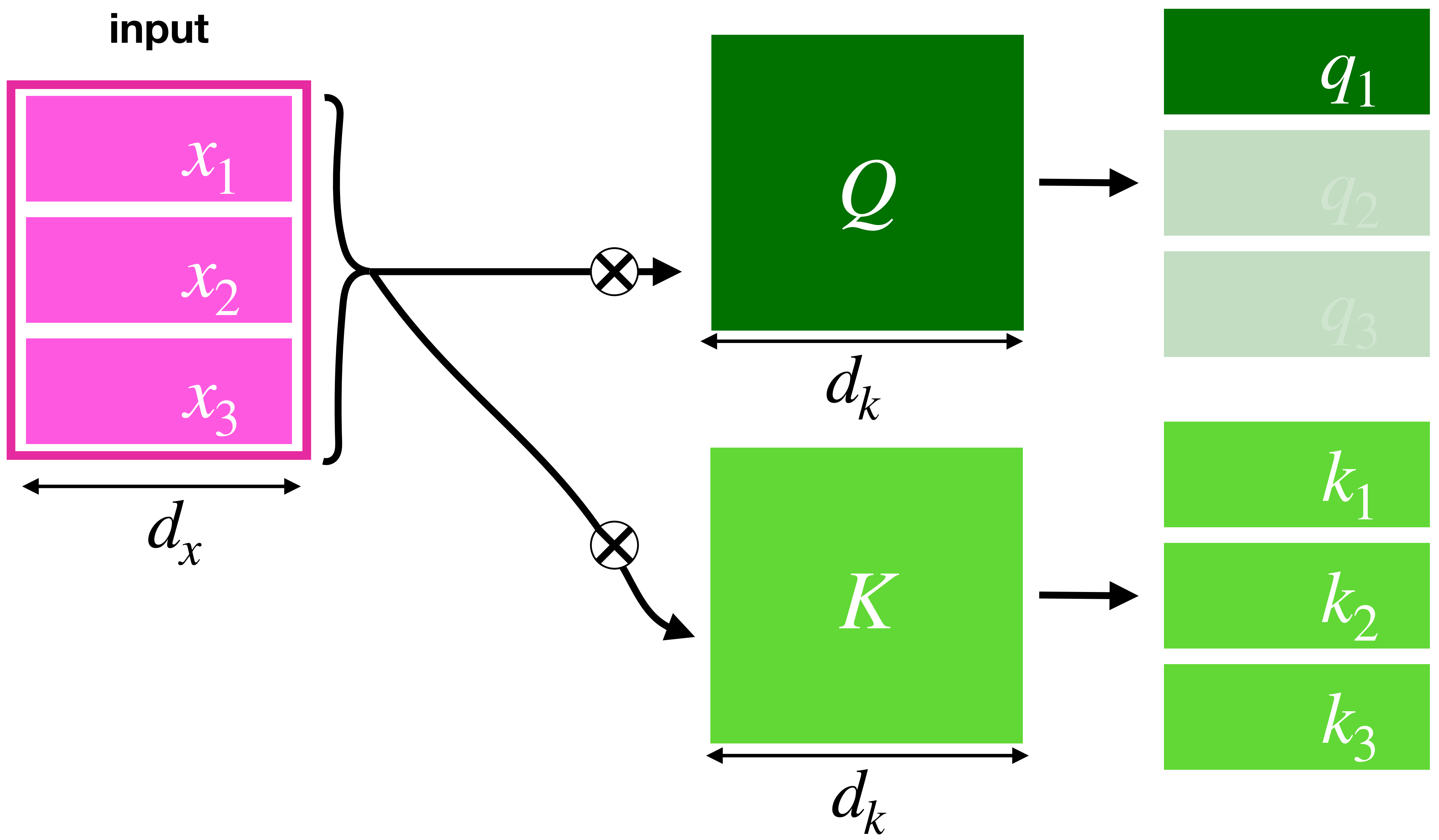
Background - Self Attention (Single Head)



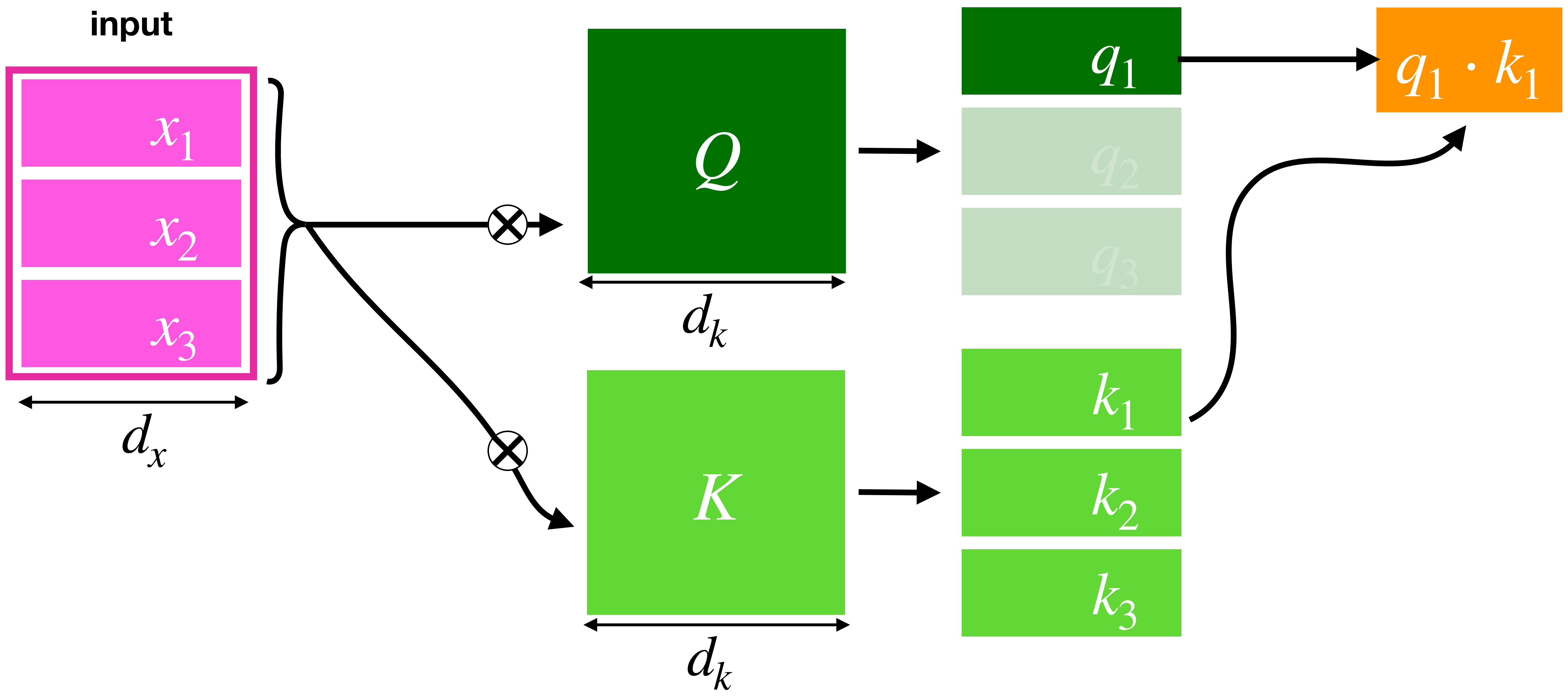
Background - Self Attention (Single Head)



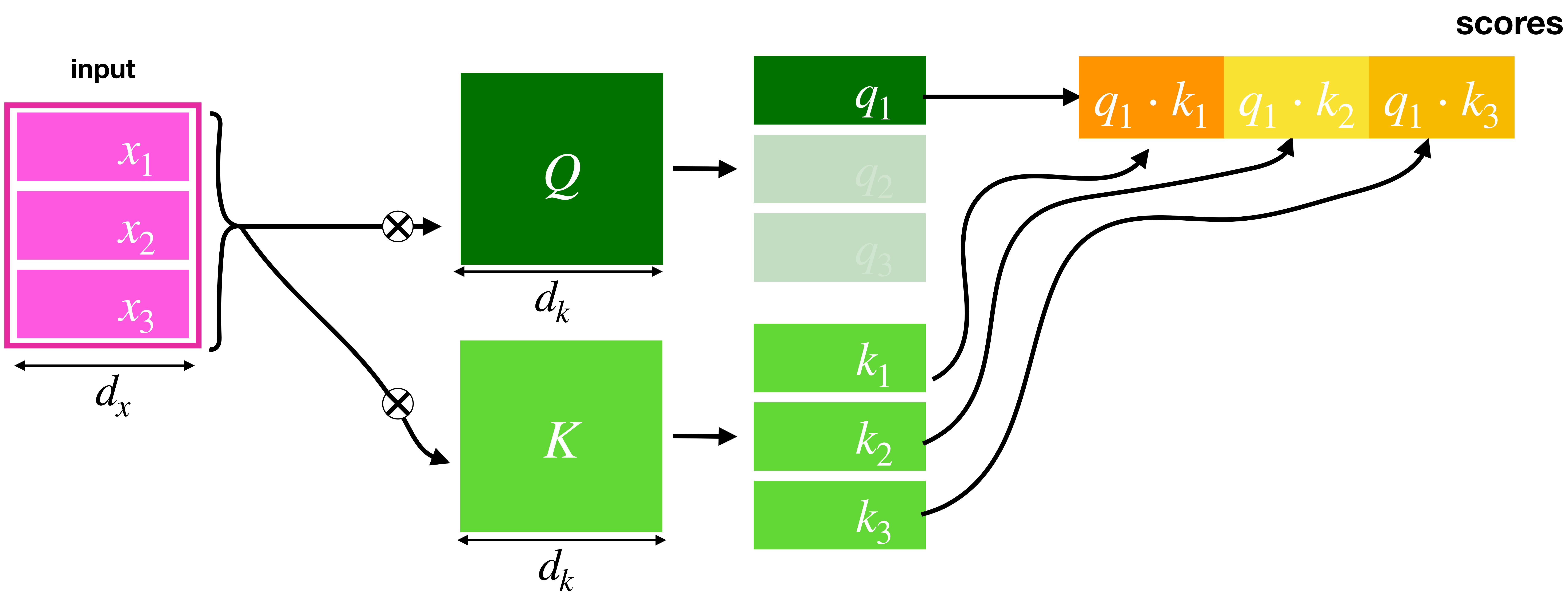
Background - Self Attention (Single Head)



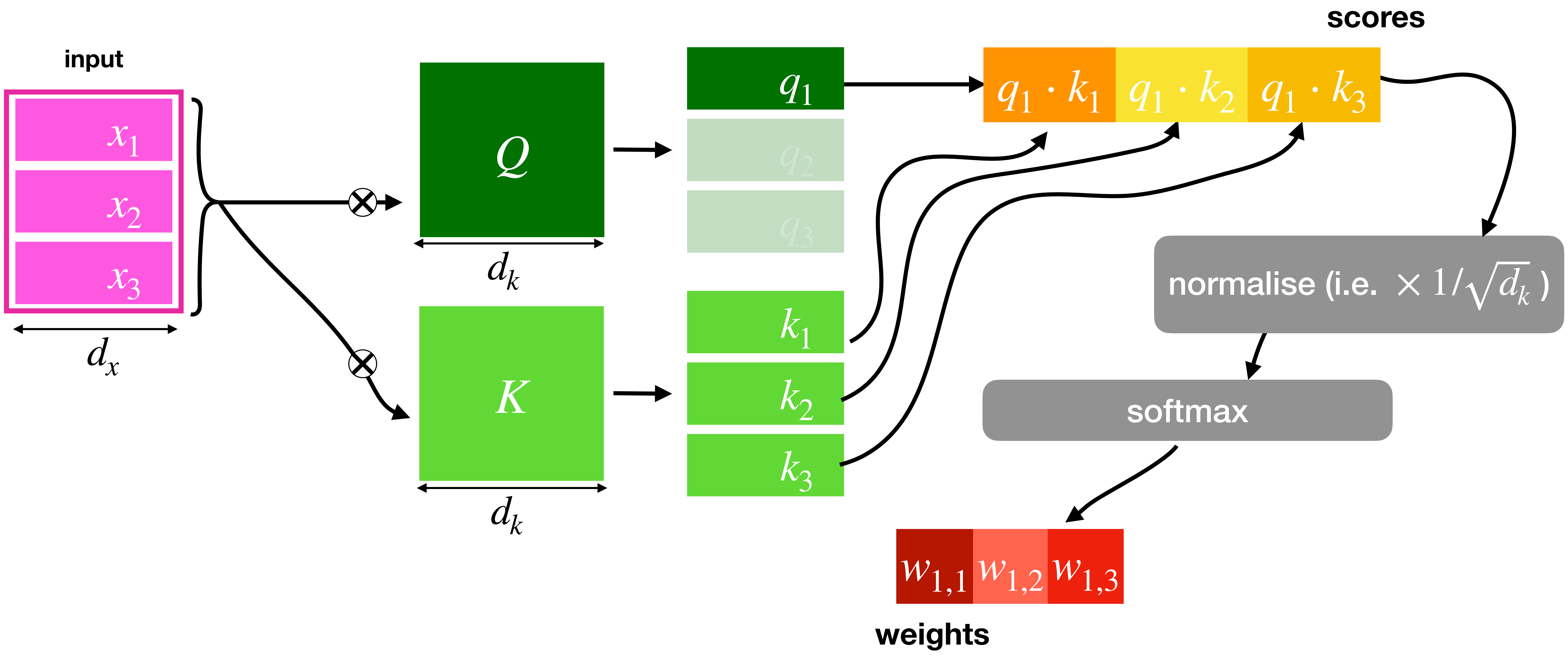
Background - Self Attention (Single Head)



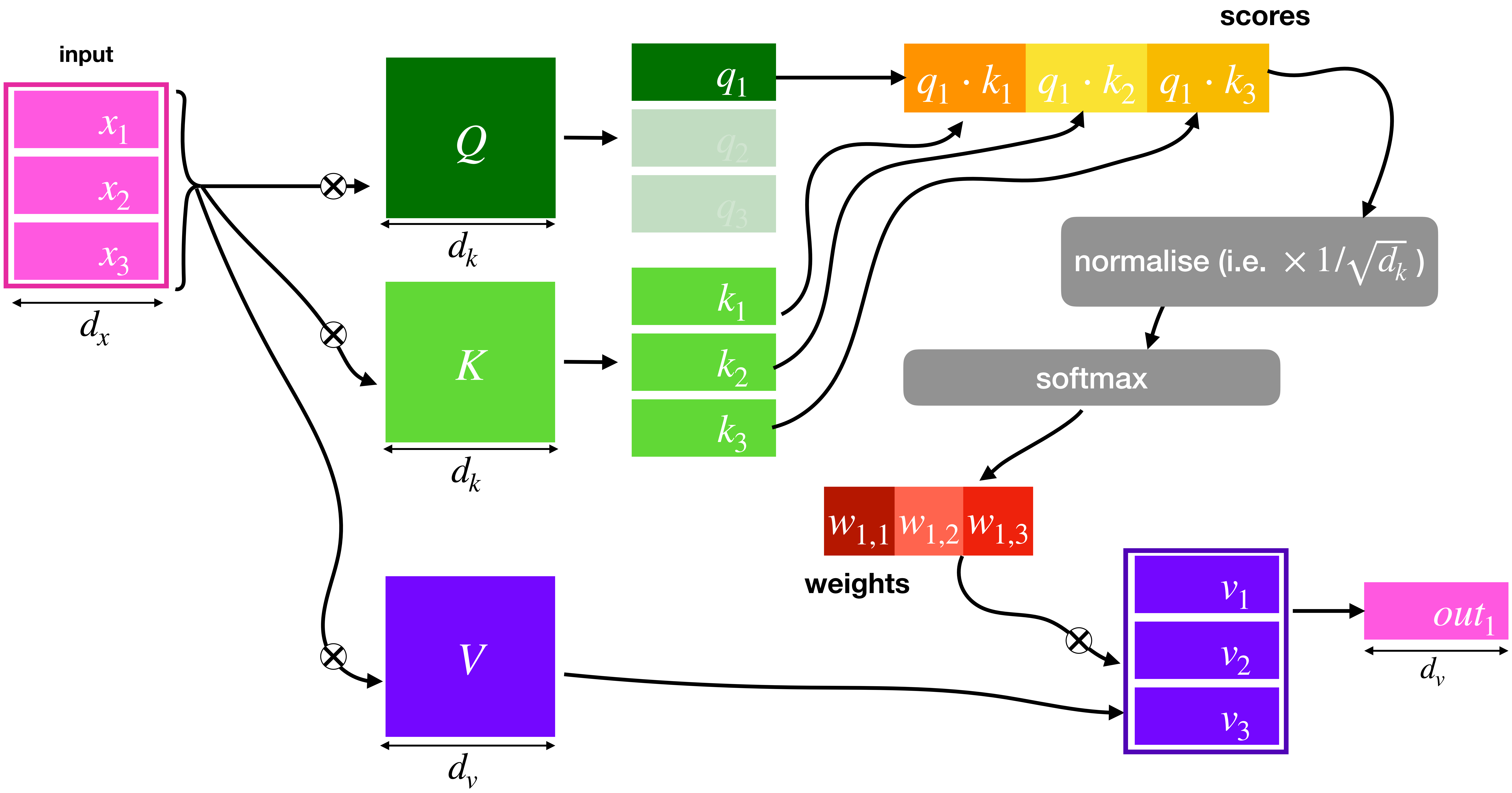
Background - Self Attention (Single Head)



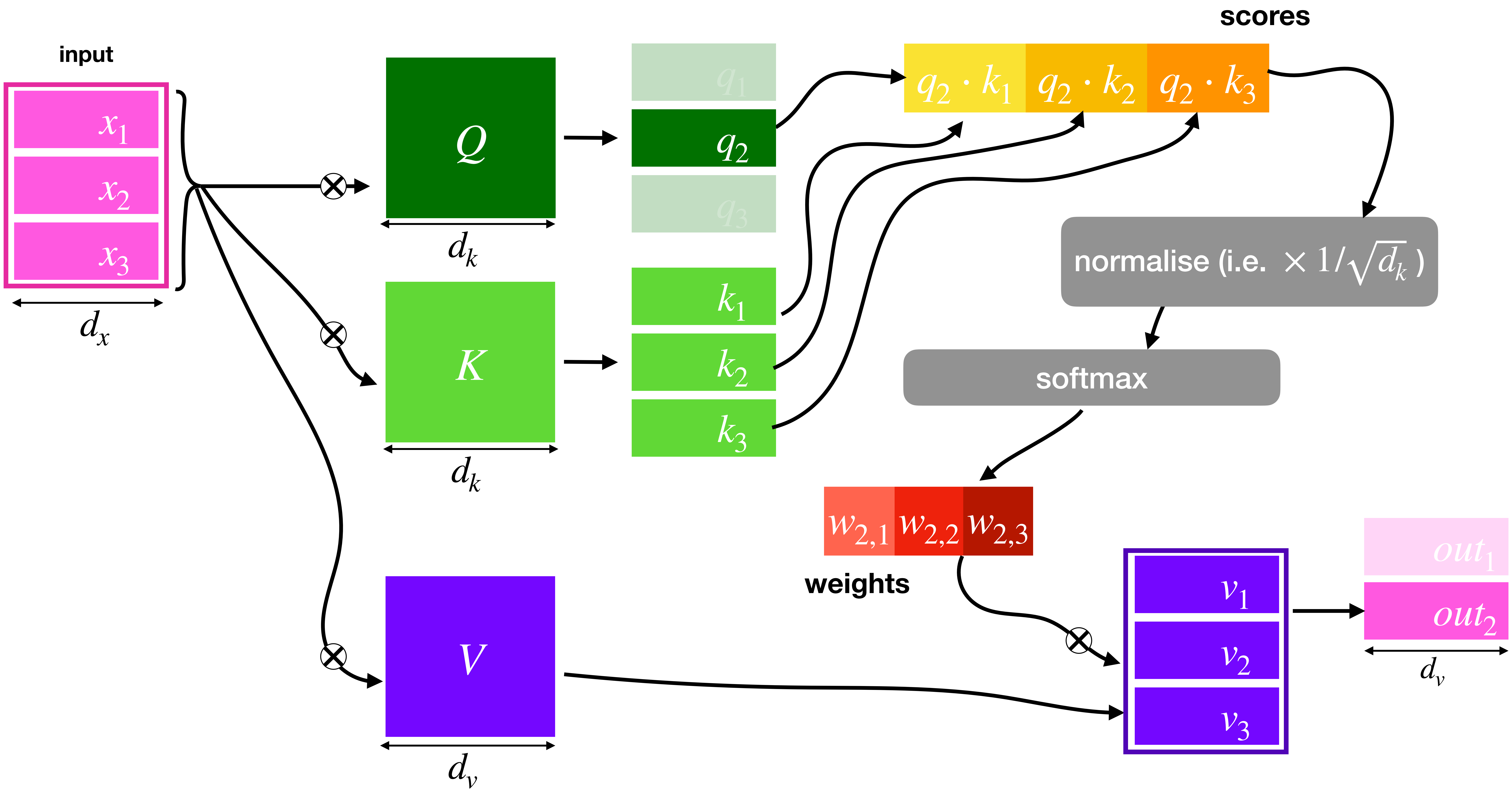
Background - Self Attention (Single Head)



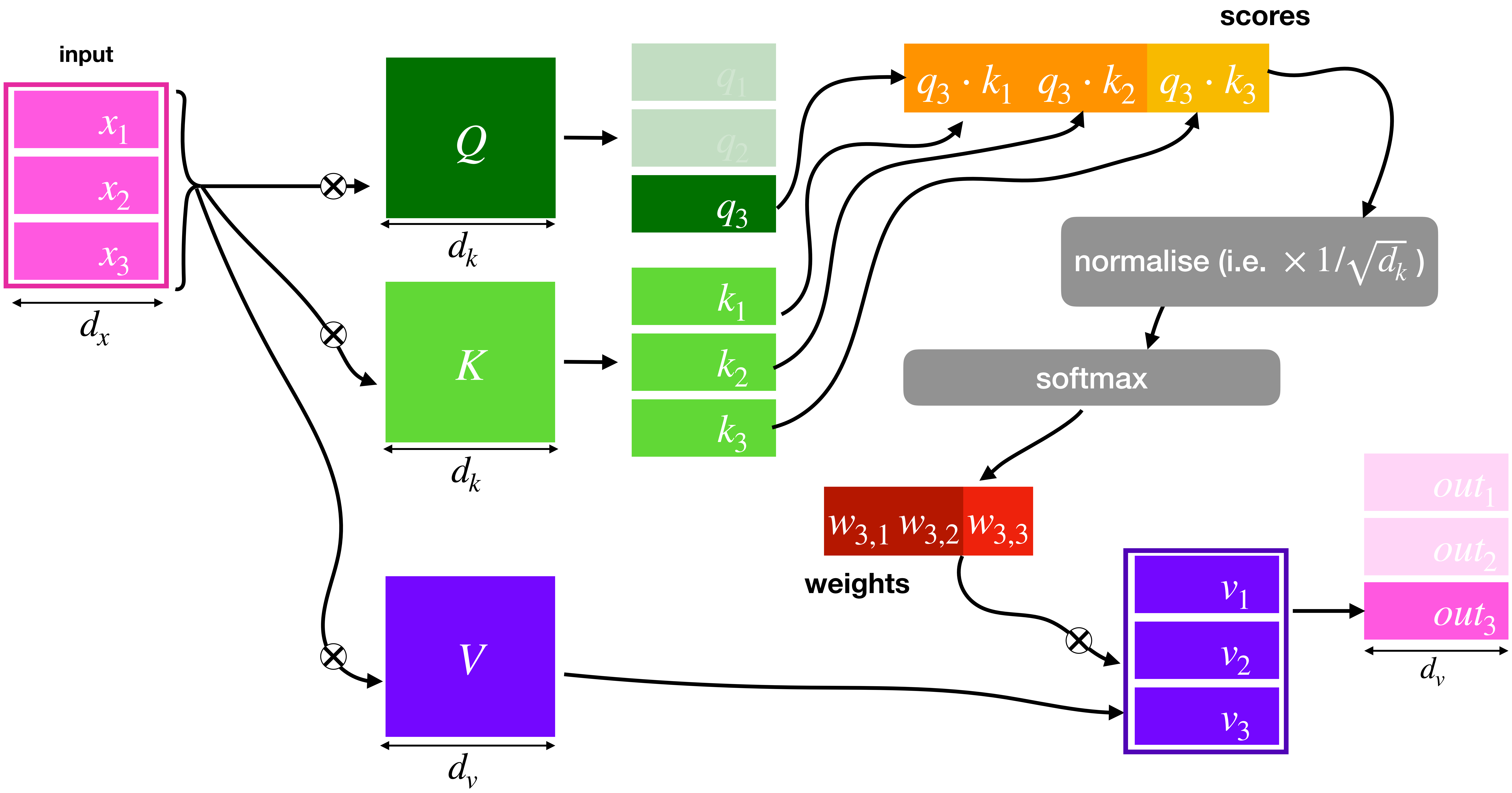
Background - Self Attention (Single Head)



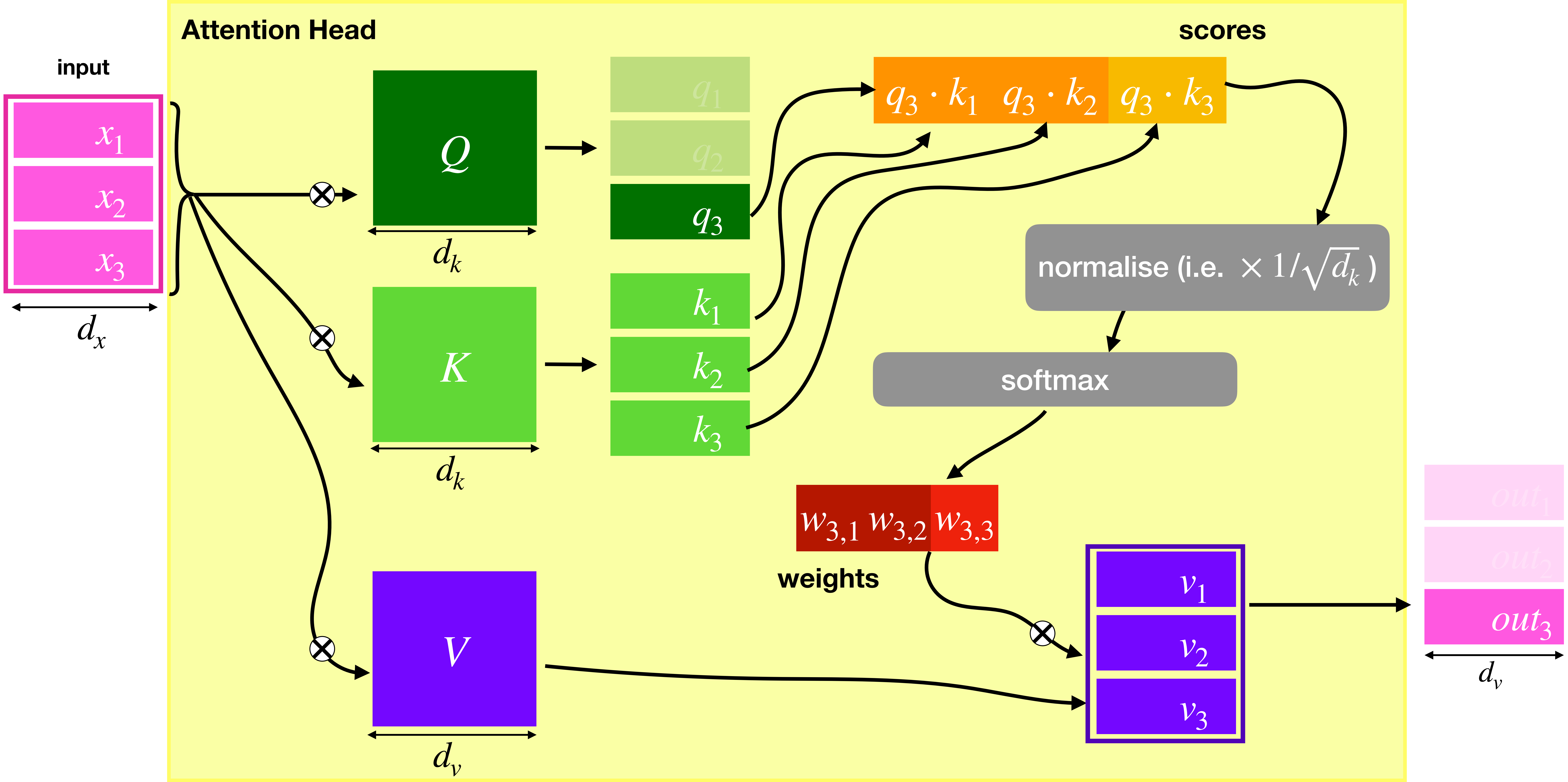
Background - Self Attention (Single Head)



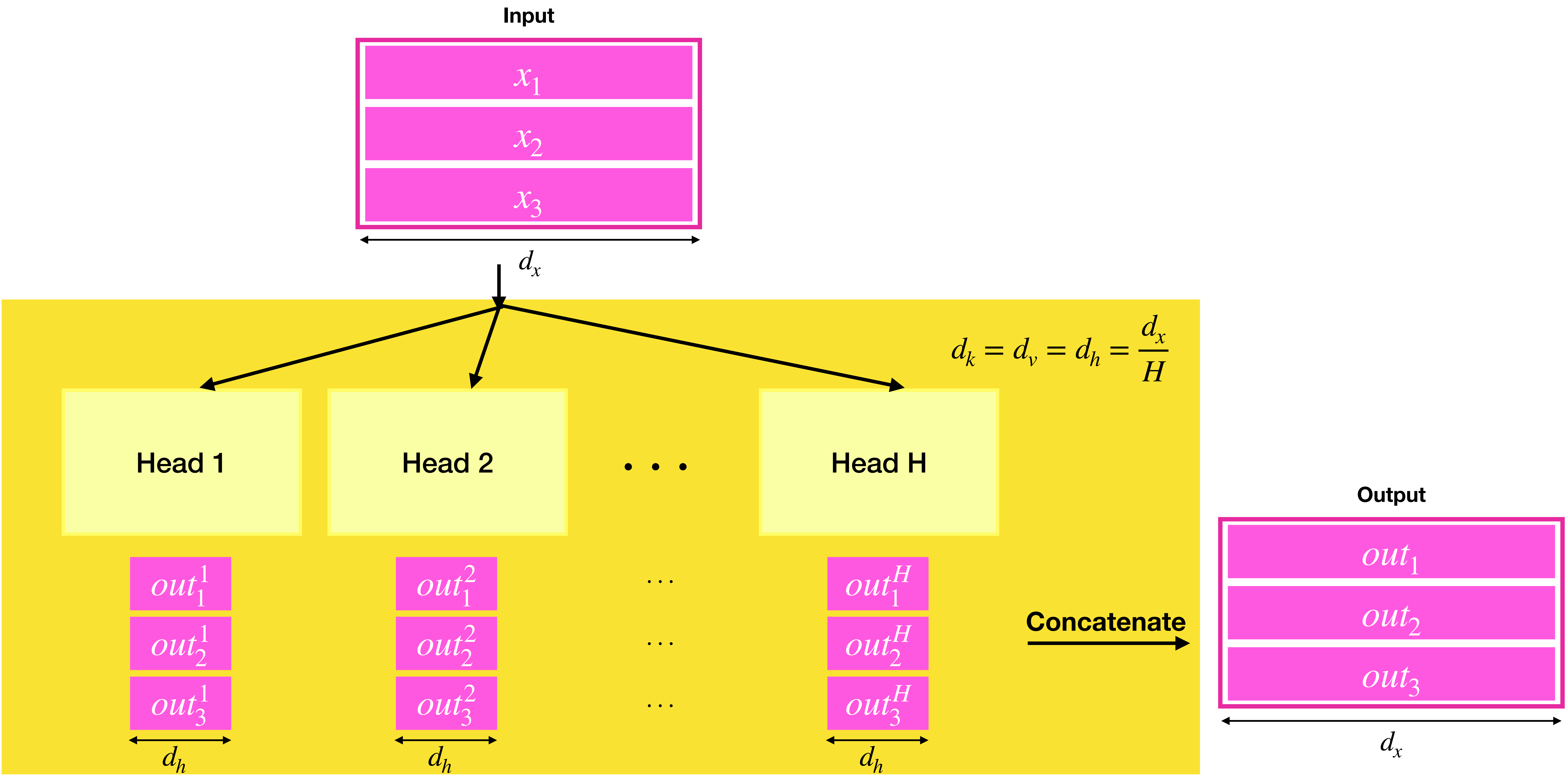
Background - Self Attention (Single Head)



Background - Self Attention (Single Head)



Background - Multi-Headed Self Attention

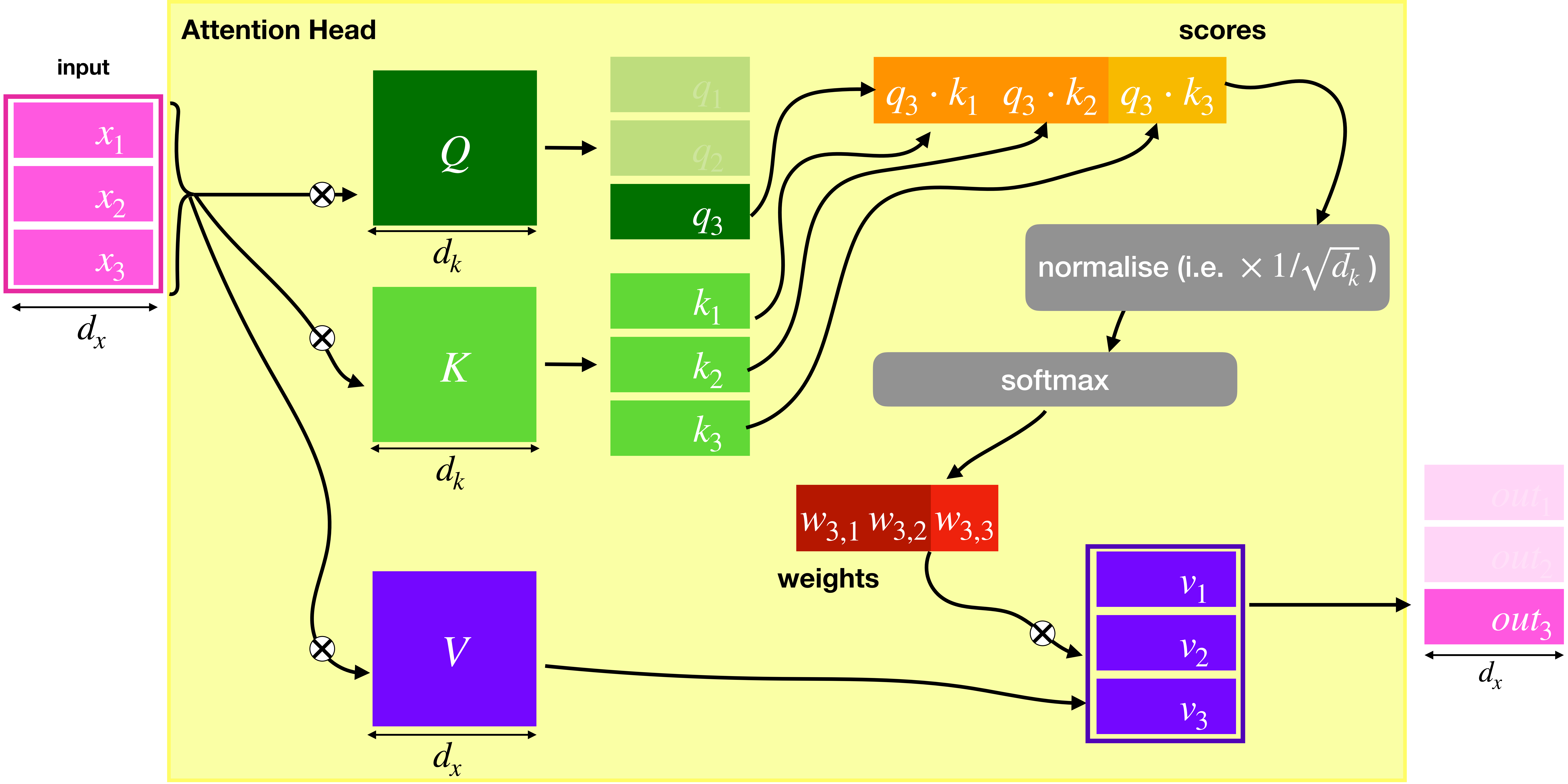


The multi-headed attention lets one layer do multiple operations

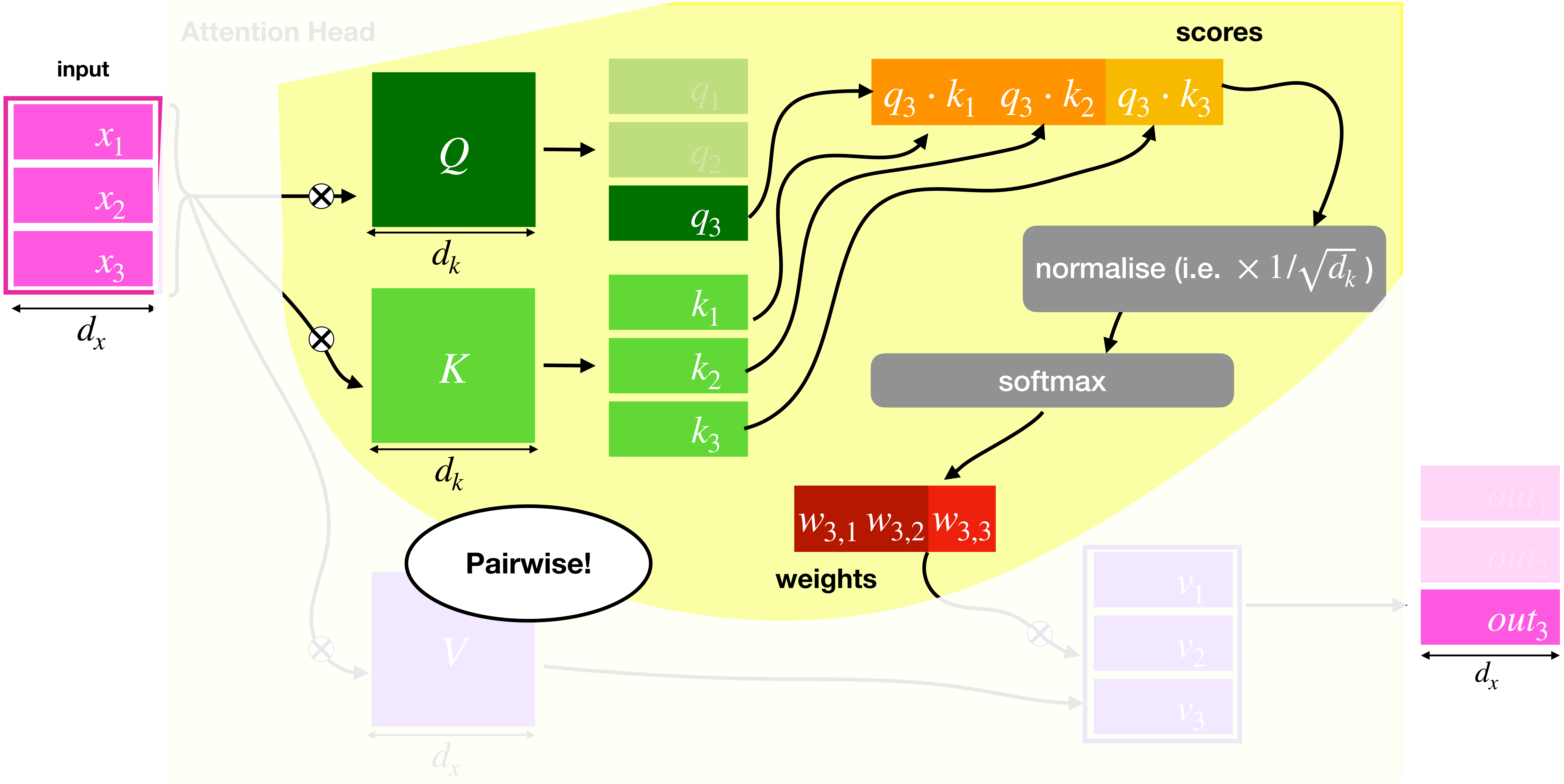
It does not in itself add new power

So, how do we present *one* head?

Self Attention (Single Head)



Single Head: Scoring ↔ Selecting

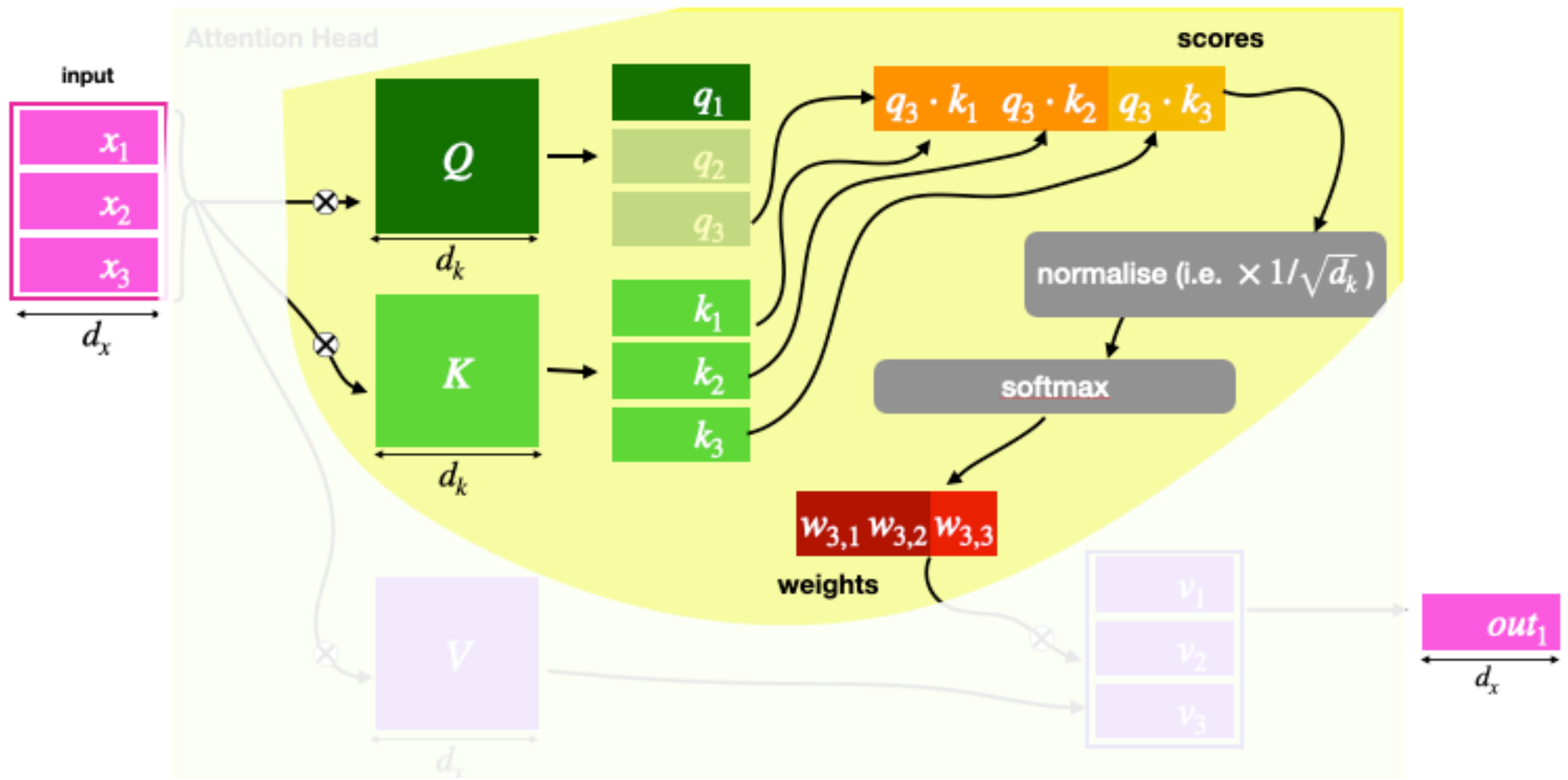


Single Head: Scoring ↔ Selecting

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

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

	2	0	0
0	F	**T**	**T**
1	F	F	F
2	**T**	F	F

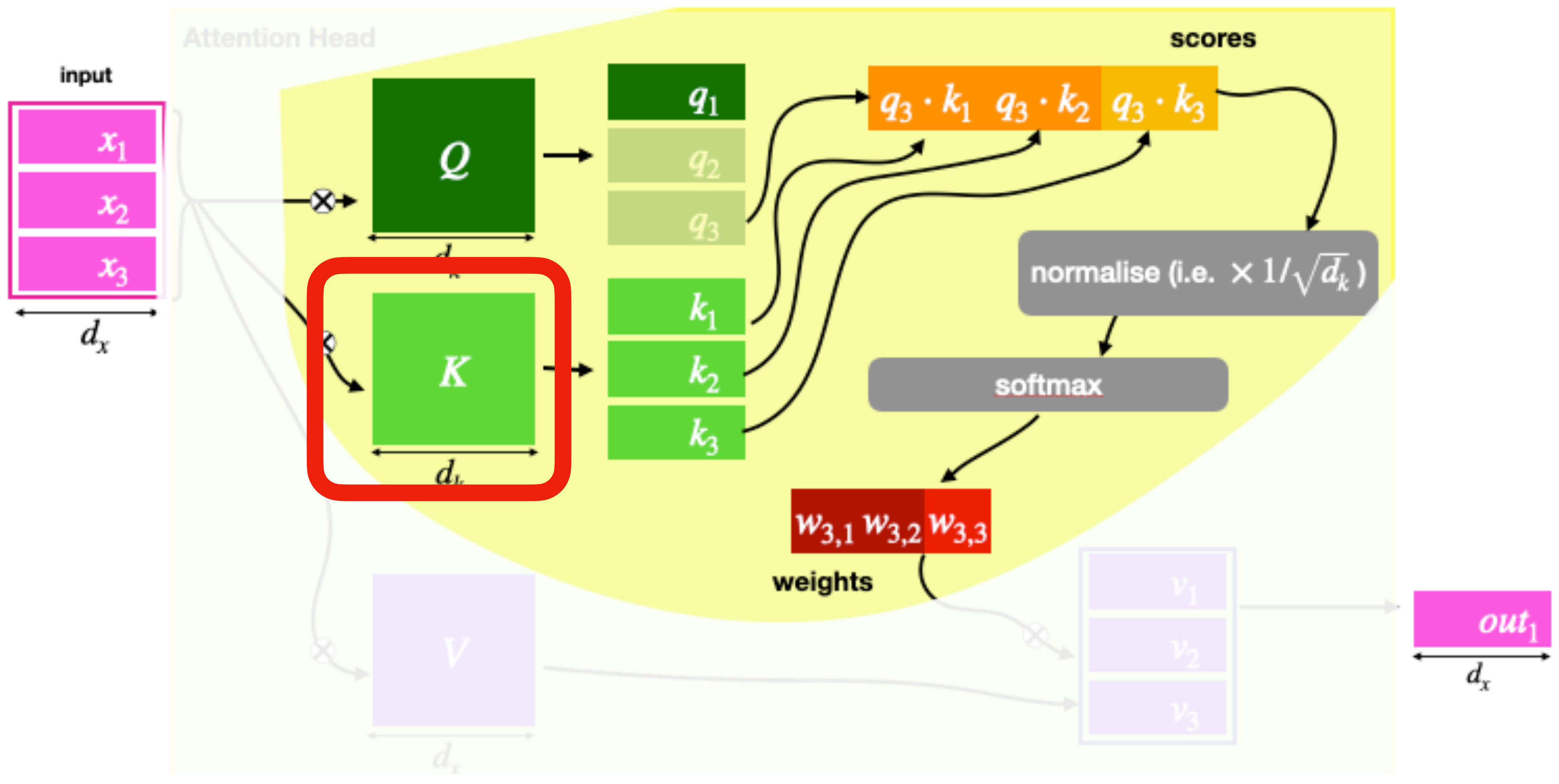


Single Head: Scoring ↔ Selecting

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

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

	2	0	0
0	F	T	T
1	F	F	F
2	T	F	F

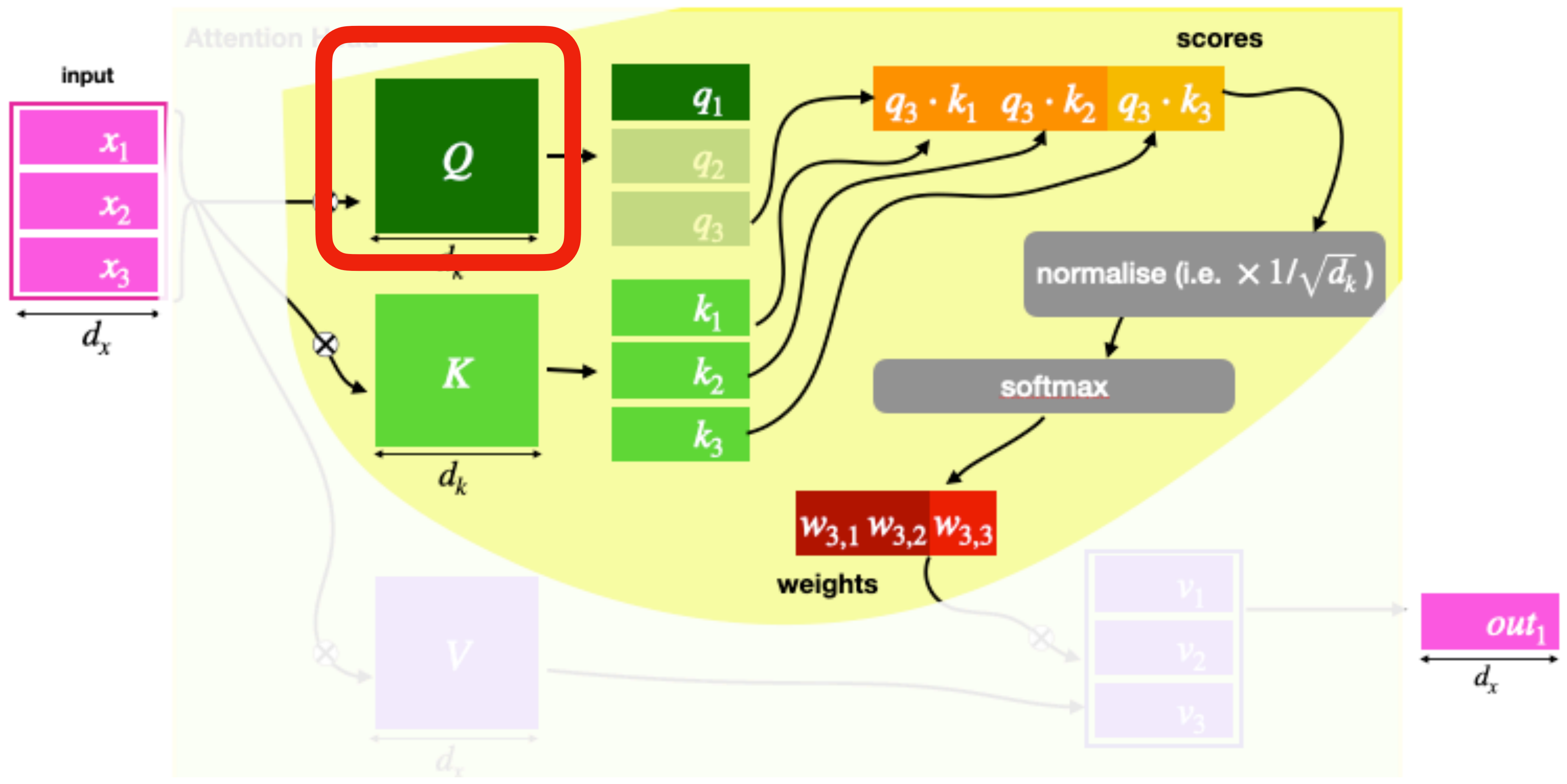


Single Head: Scoring ↔ Selecting

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

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

	2	0	0
0	F	T	T
1	F	F	F
2	T	F	F

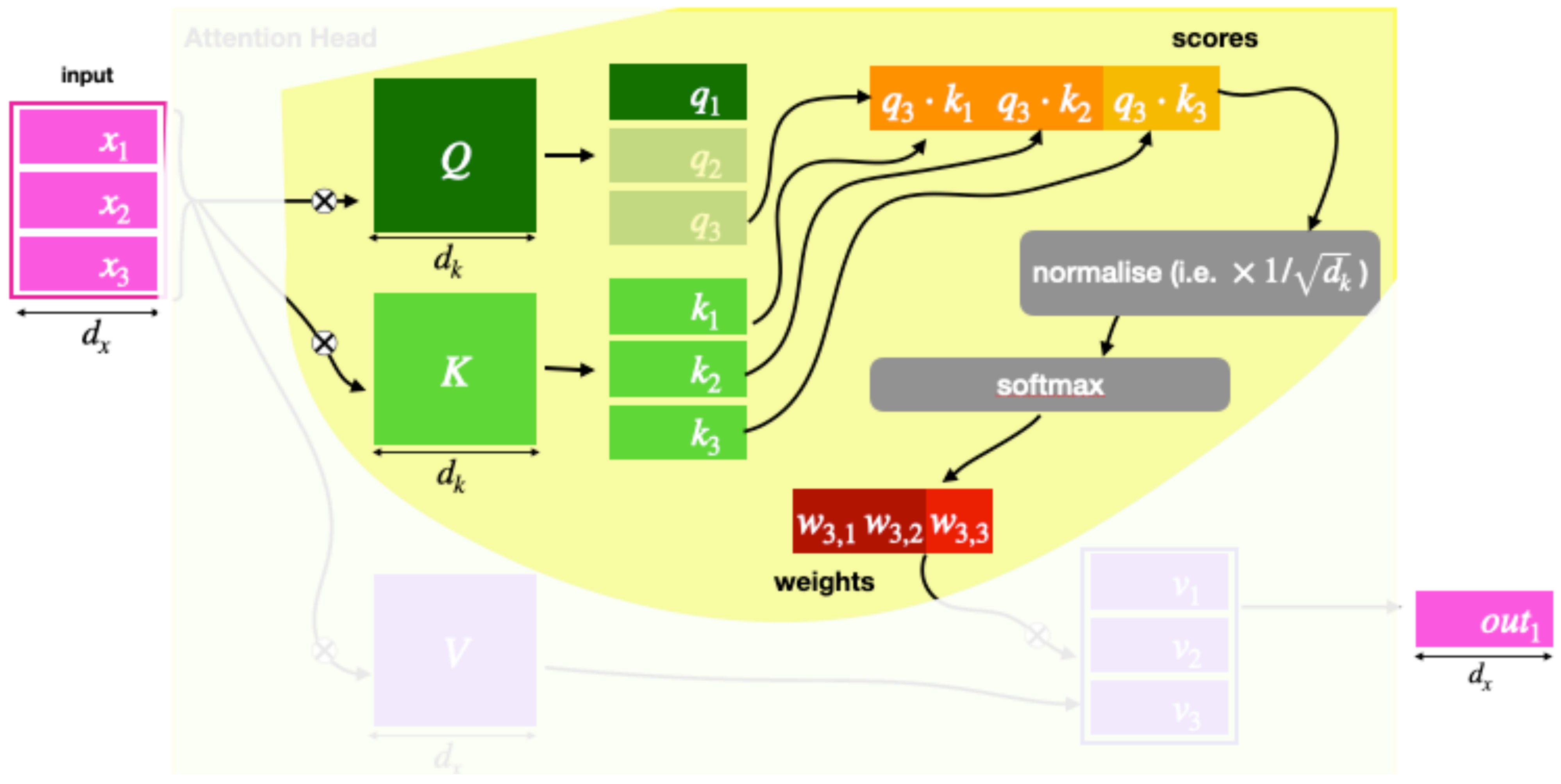


Single Head: Scoring ↔ Selecting

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

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

	2	0	0
0	F	T	T
1	F	F	F
2	T	F	F

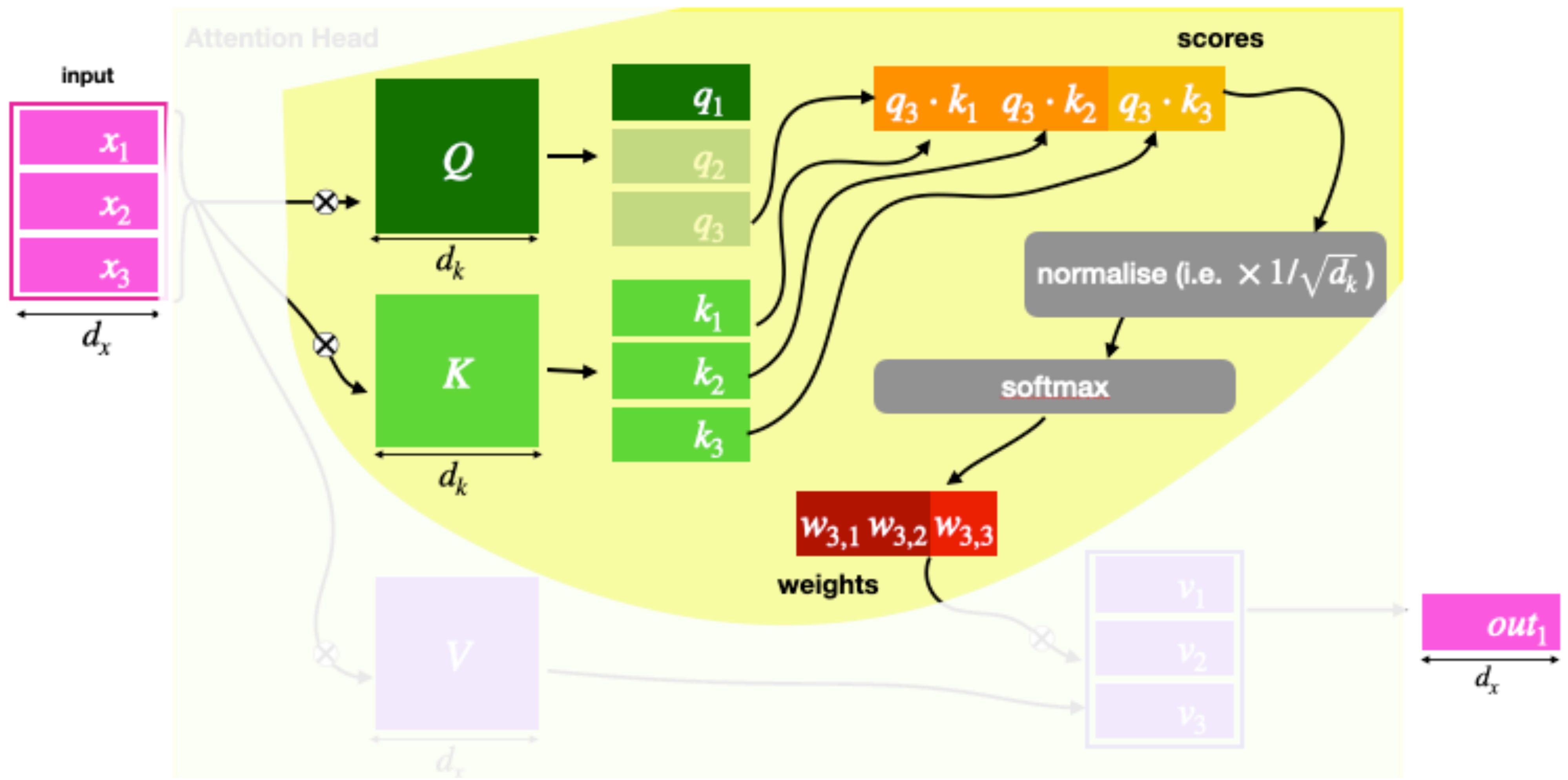


Single Head: Scoring ↔ Selecting

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

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

	2	0	0
0	F	T	T
1	F	F	F
2	T	F	F

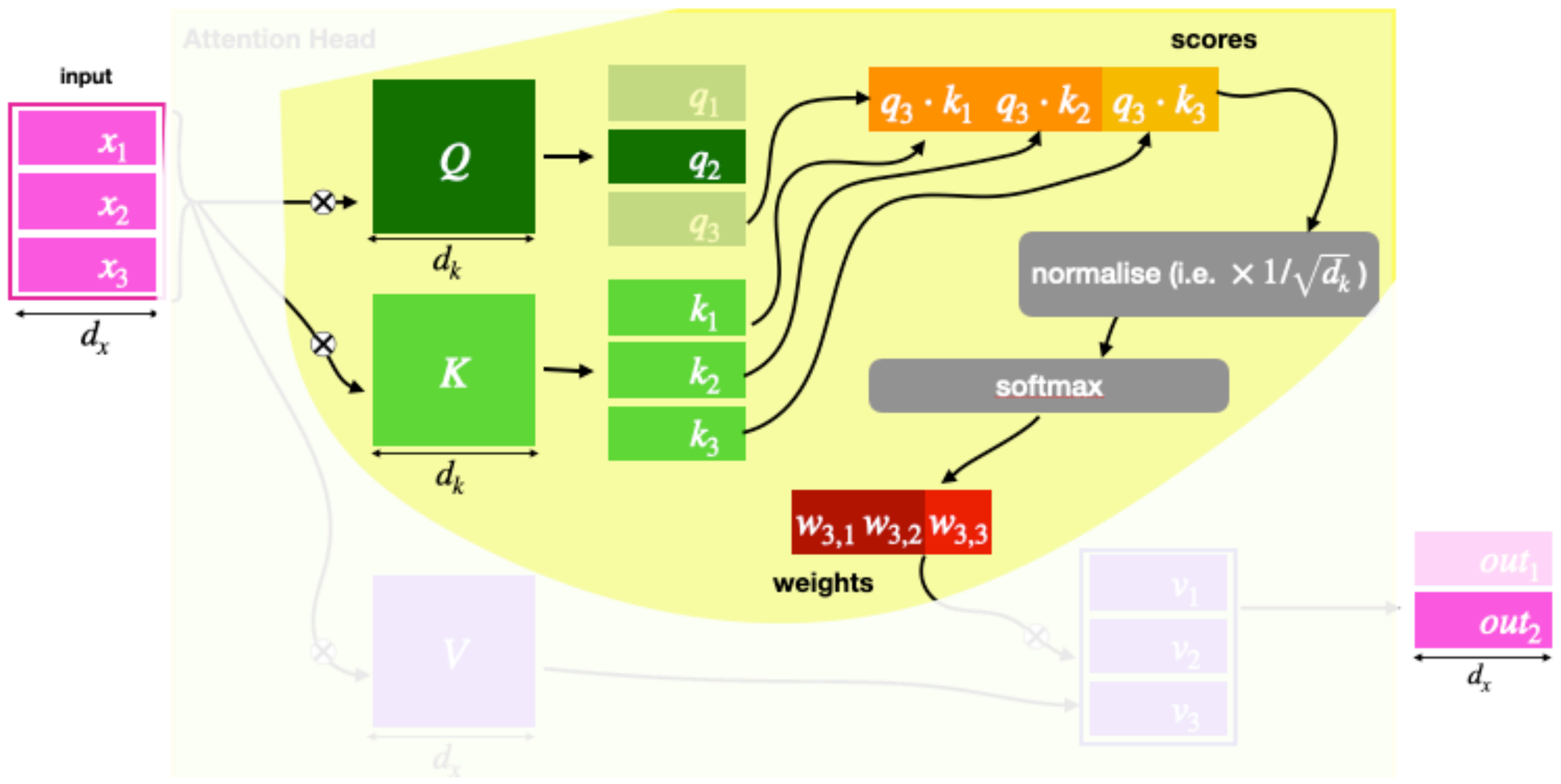


Single Head: Scoring ↔ Selecting

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

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

	2	0	0
0	F	T	T
1	F	F	F
2	T	F	F

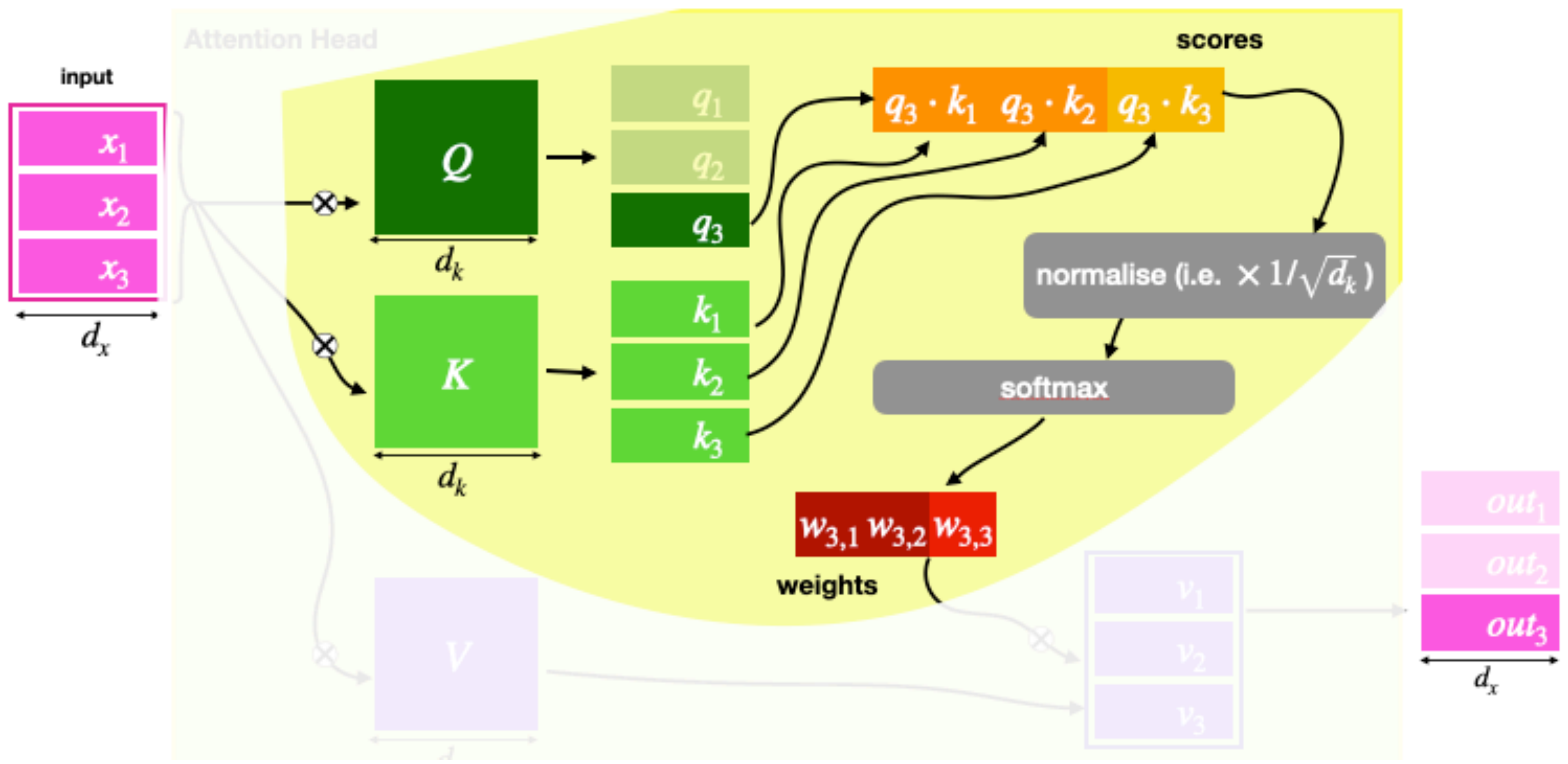


Single Head: Scoring ↔ Selecting

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

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

	2	0	0
0	F	T	T
1	F	F	F
2	T	F	F

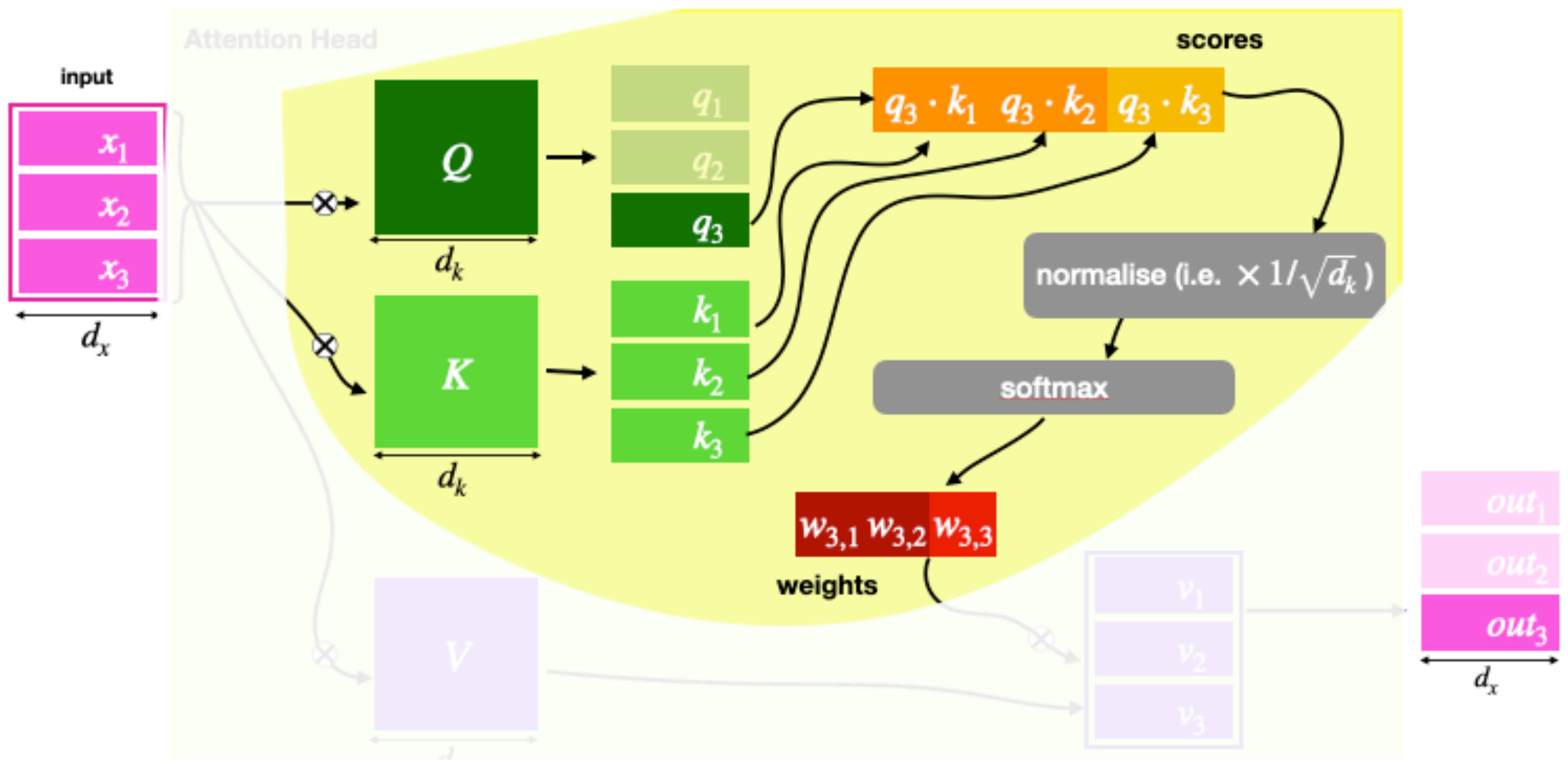


Single Head: Scoring ↔ Selecting

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

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

	2	0	0
0	F	**T**	**T**
1	F	F	F
2	**T**	F	F

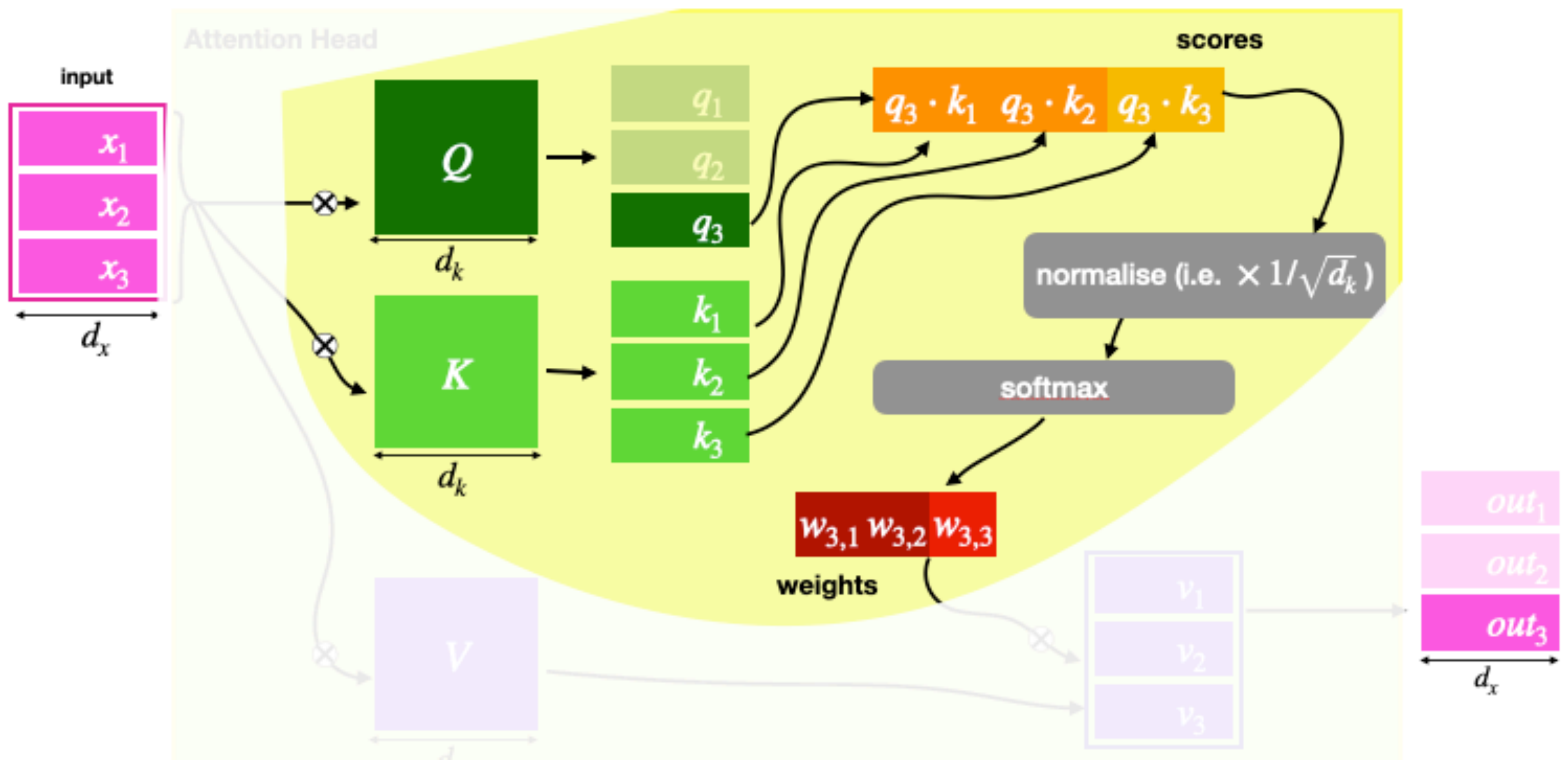


Single Head: Scoring ↔ Selecting

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

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

	2	0	0
0	F	T	T
1	F	F	F
2	T	F	F



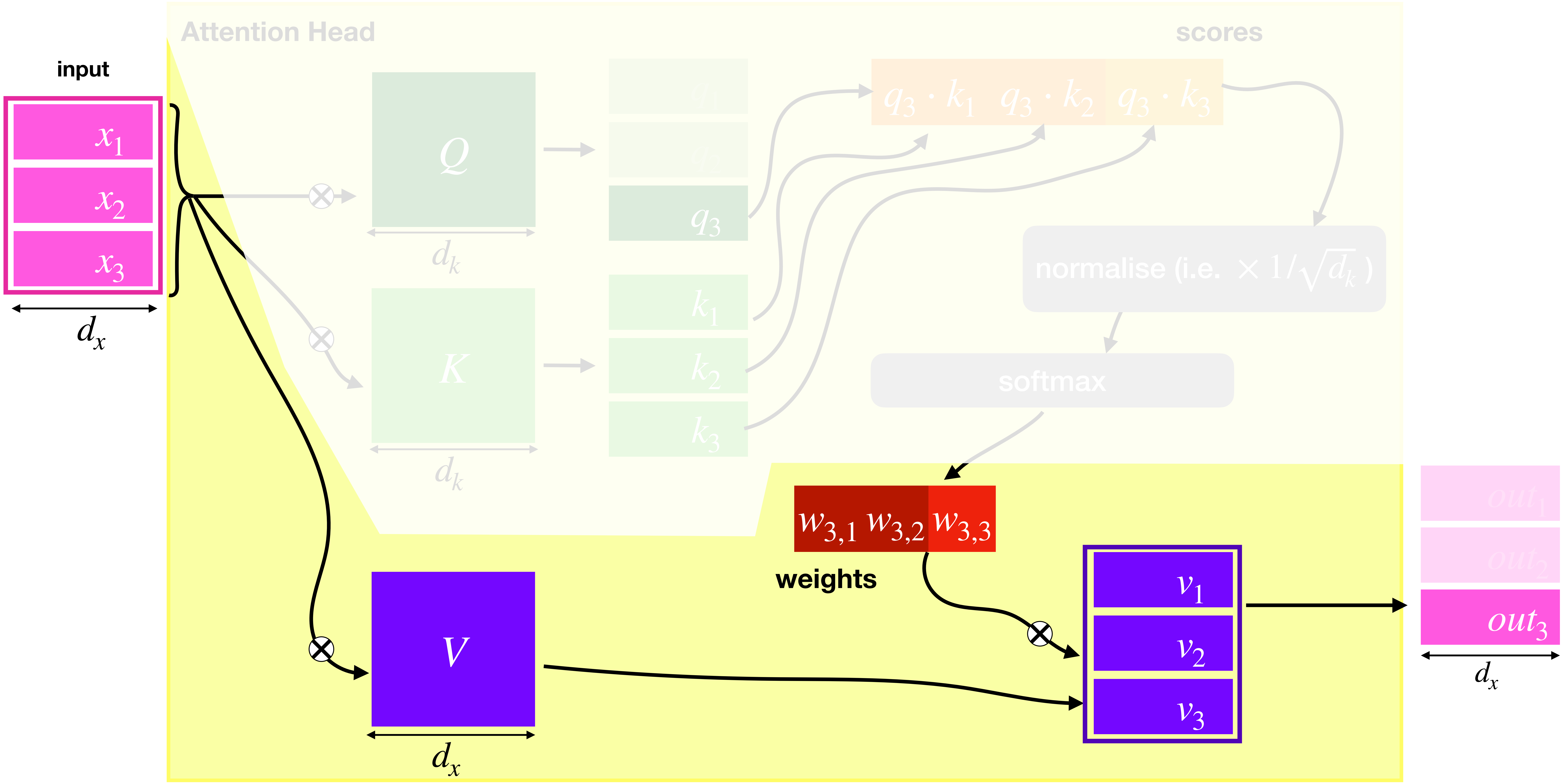
Another example:

sel2 = select([2,0,0],[0,1,2],>=)

	2	0	0
0	T	T	T
1	T	F	F
2	T	F	F

Is this "reasonable"?
What does it mean with respect to Q and K?

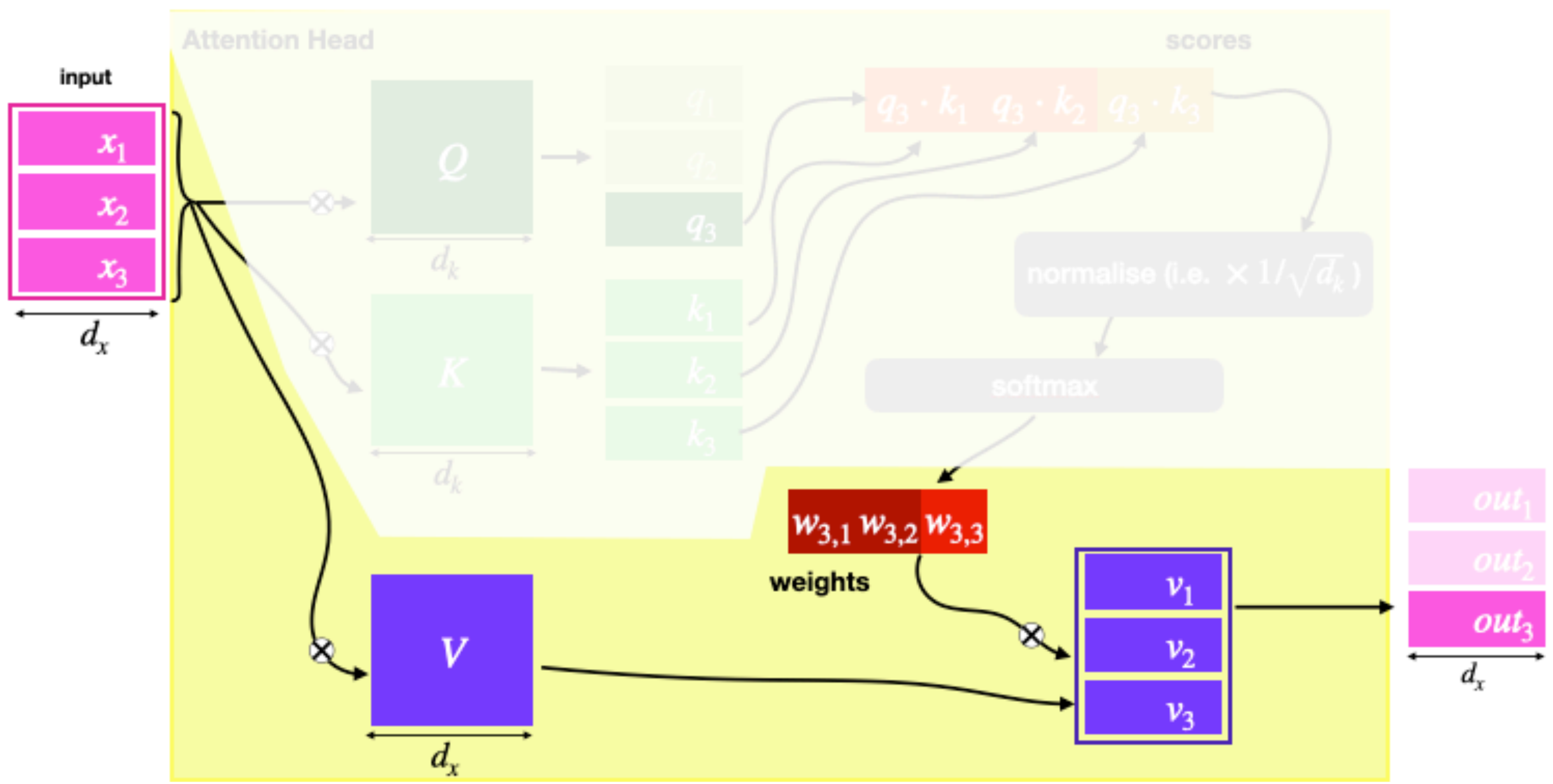
Single Head: Weighted Average ↔ Aggregation



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	

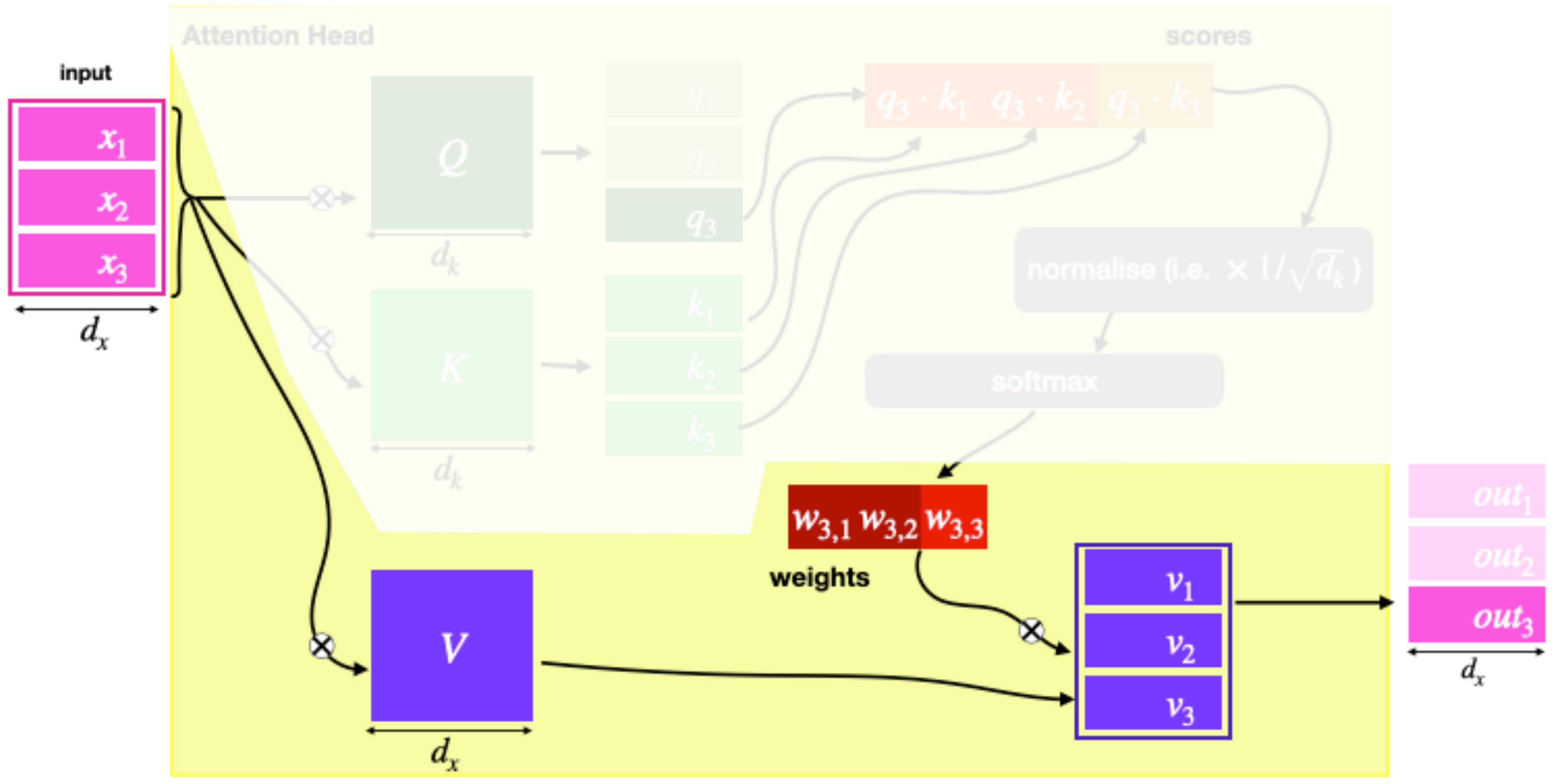


Single Head: Weighted Average ↔ Aggregation

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

			1 2 4		
F	T	T	1 2 4	=>	3
F	F	F	1 2 4	=>	0
T	F	F	1 2 4	=>	1

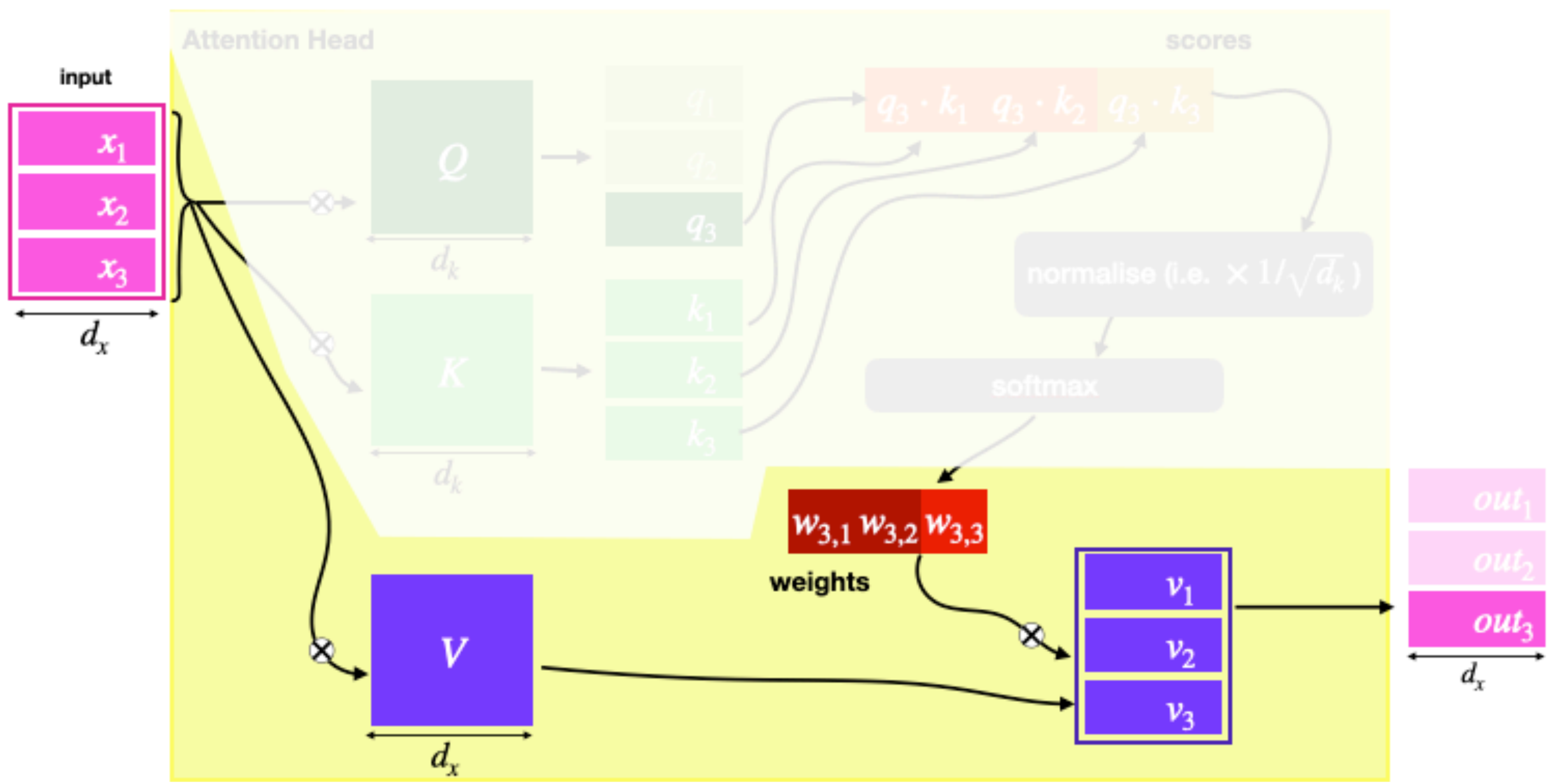
=> [3,0,1]



Single Head: Weighted Average ↔ Aggregation

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

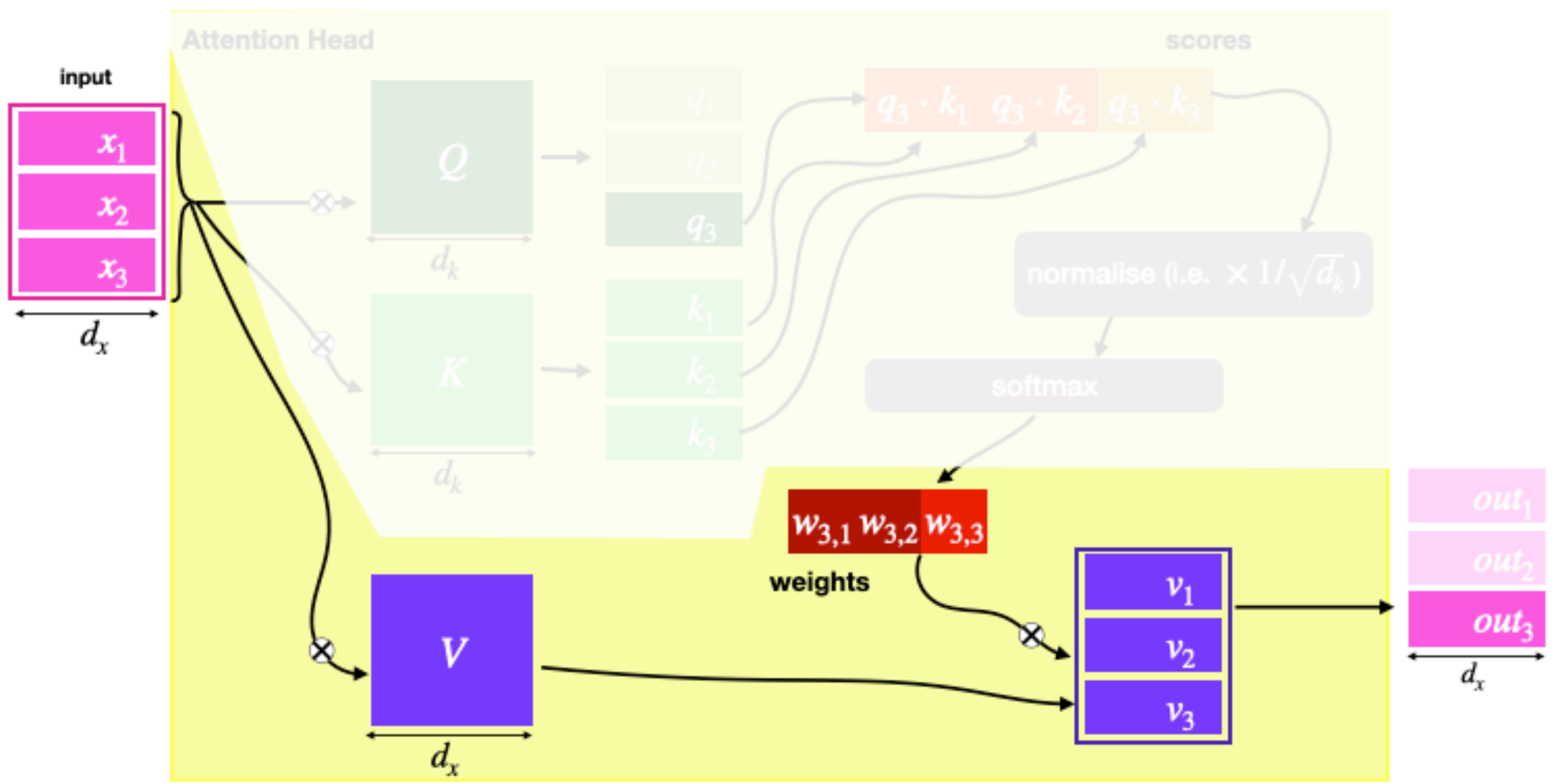
				1 2 4		
F	T	T	1 2 4	=>	3	
F	F	F	1 2 4	=>	0	=> [3,0,1]
T	F	F	1 2 4	=>	1	



Single Head: Weighted Average ↔ Aggregation

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

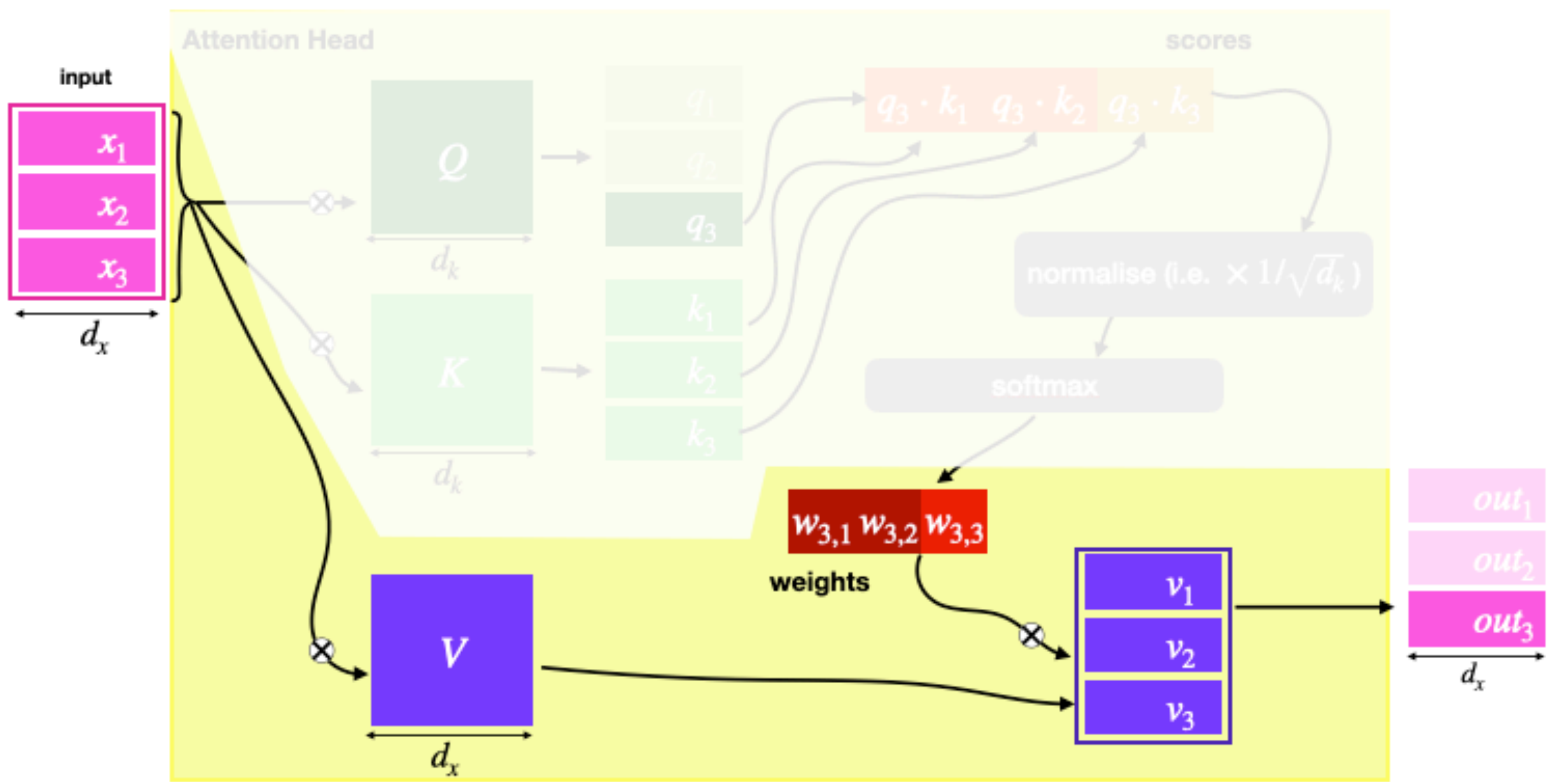
	1 2 4				
F T T	1 2 4	=>	3		
F F F	1 2 4	=>	0	=>	[3,0,1]
T F F	1 2 4	=>	1		



Single Head: Weighted Average ↔ Aggregation

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

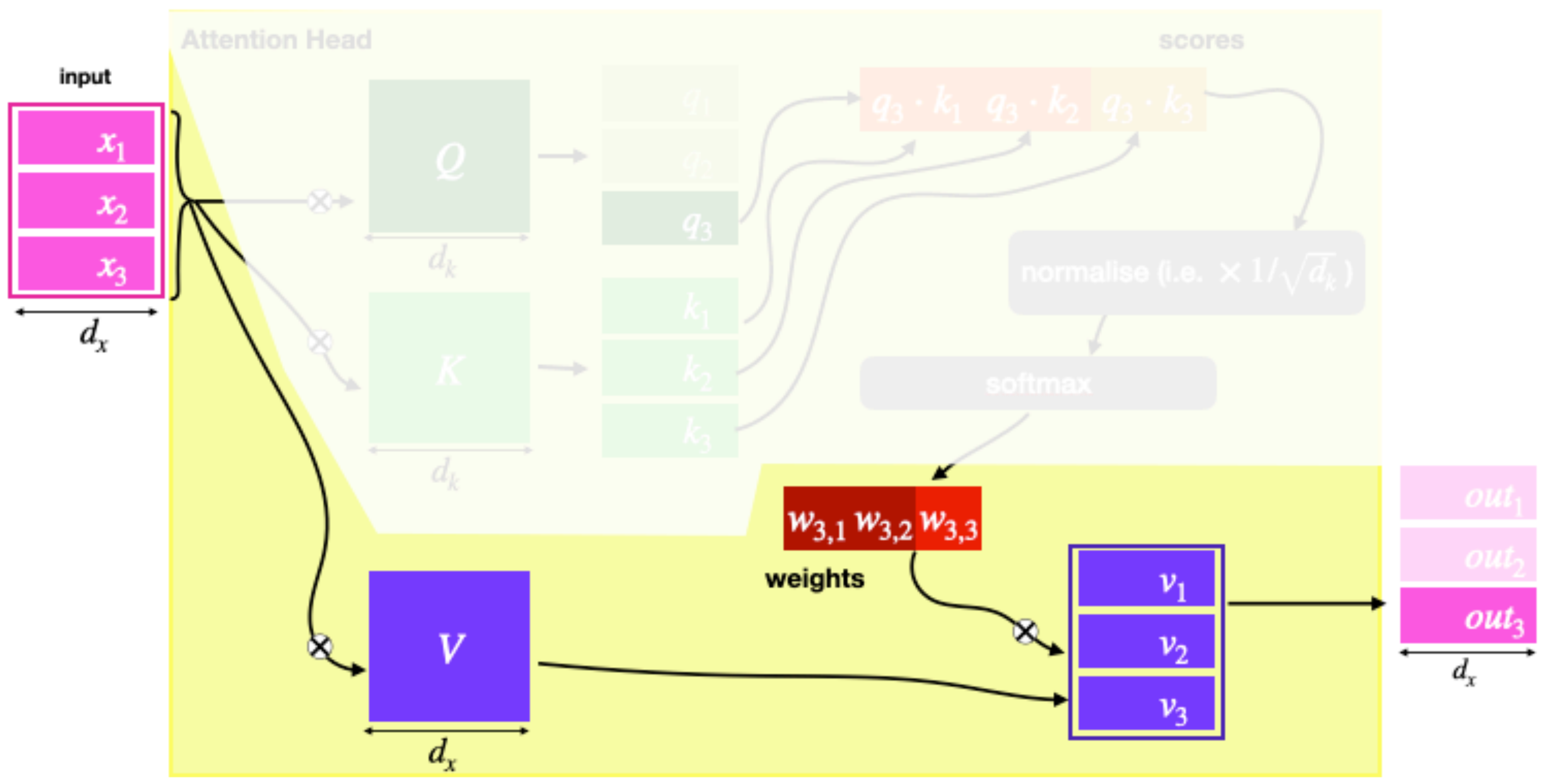
			1 2 4				
F	T	T	1 2 4	=>	3		
F	F	F	1 2 4	=>	0	=>	[3,0,1]
T	F	F	1 2 4	=>	1		



Single Head: Weighted Average ↔ Aggregation

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
```



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

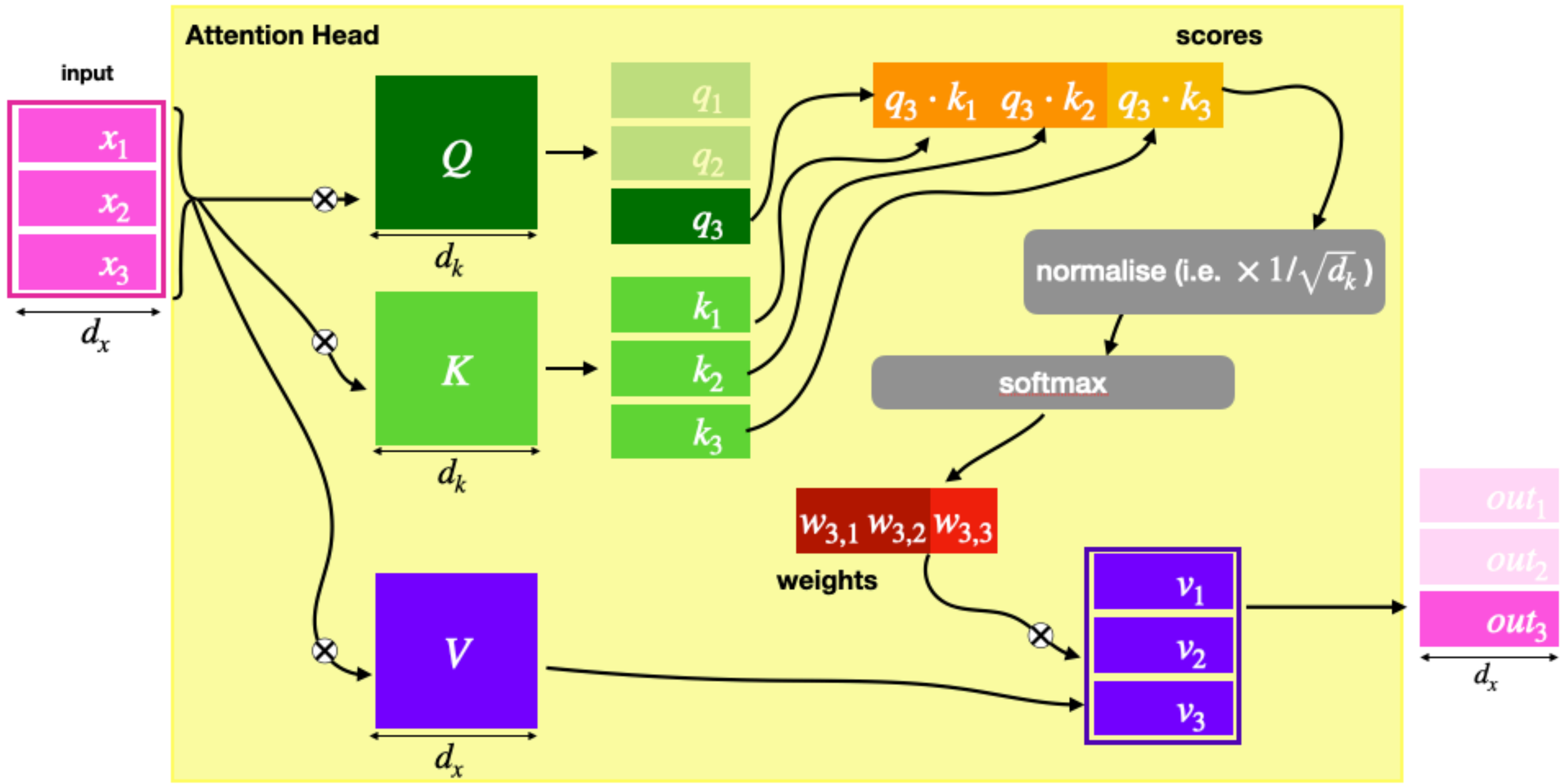
flip = select([2,1,0],[0,1,2],==)

```
     2  1  0
0    F  F  T
1    F  T  F
2    T  F  F
```

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

```
           A B C
F  F  T    A B C   =>  C
F  T  F    A B C   =>  B  =>  [C,B,A]
T  F  F    A B C   =>  A
```

Single Head: Select/Aggregate in RASP



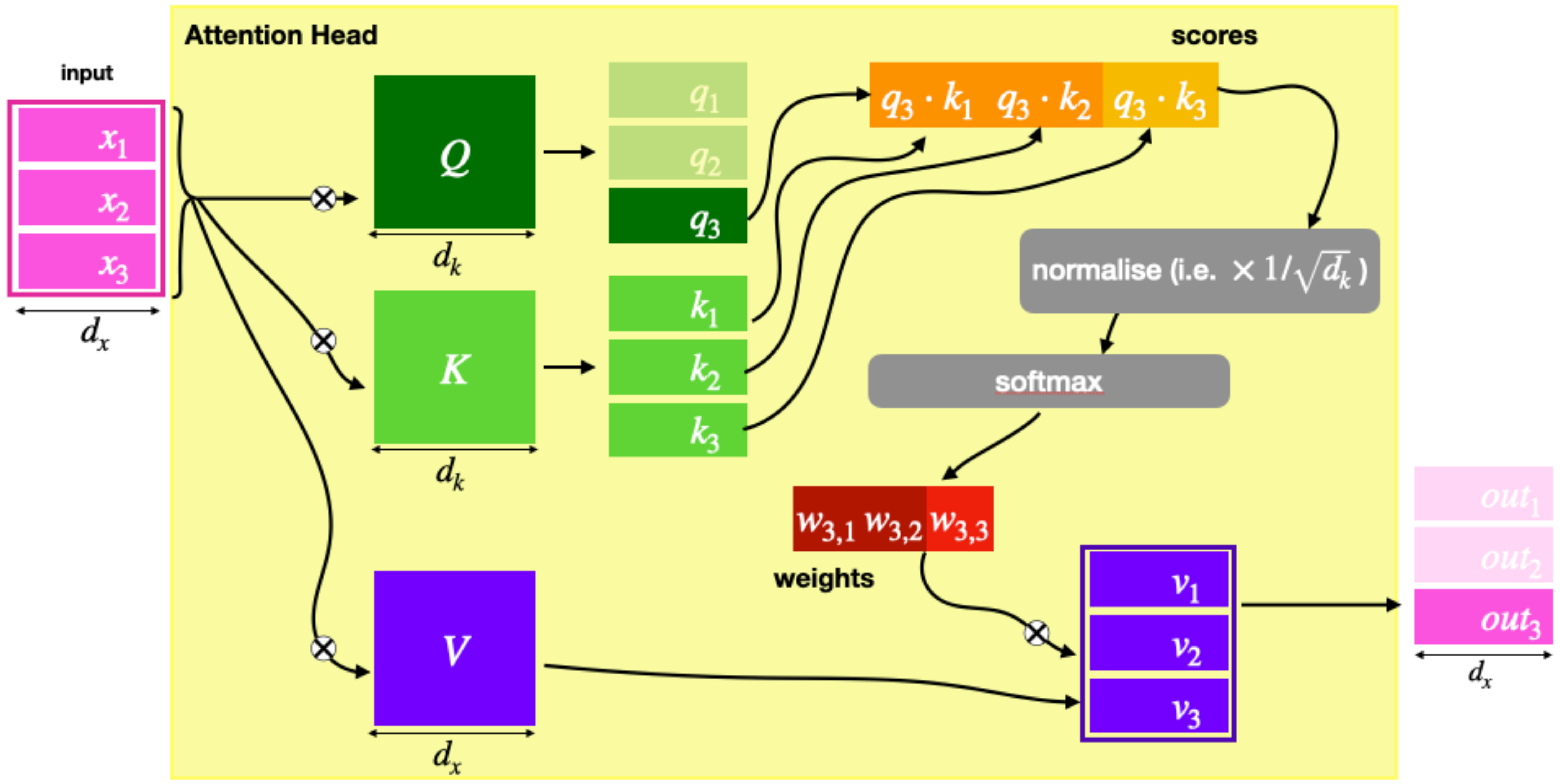
Example from before: reverse in RASP

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

The select decisions are pairwise!!

What would happen if they weren't?

Single Head: Select/Aggregate in RASP



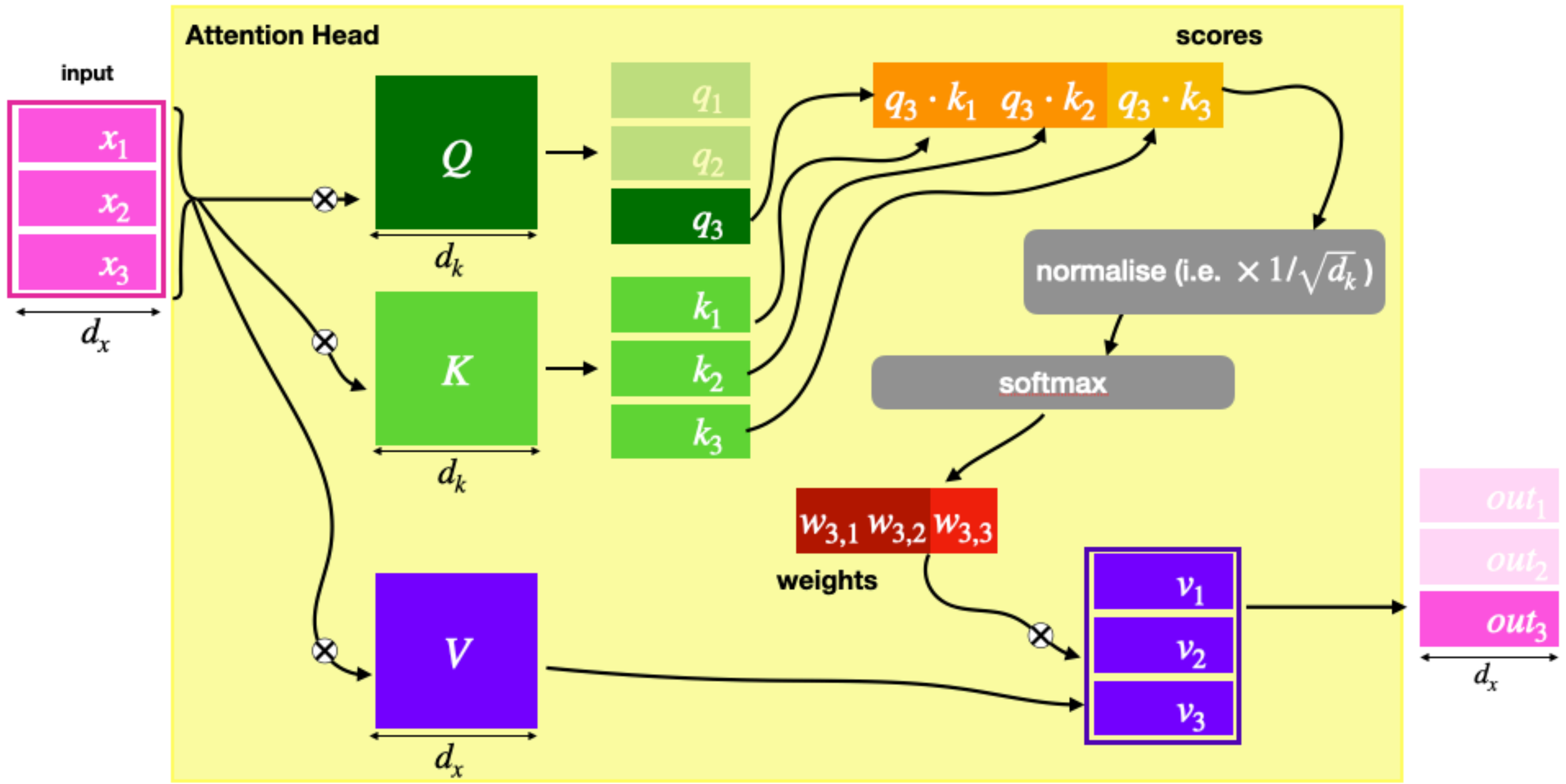
Example from before: reverse in RASP

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

The select decisions are pairwise!!

What would happen if they weren't?

Single Head: Select/Aggregate in RASP



Example from before: reverse in RASP

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

See anything suspicious in the example?

The select decisions are pairwise!!

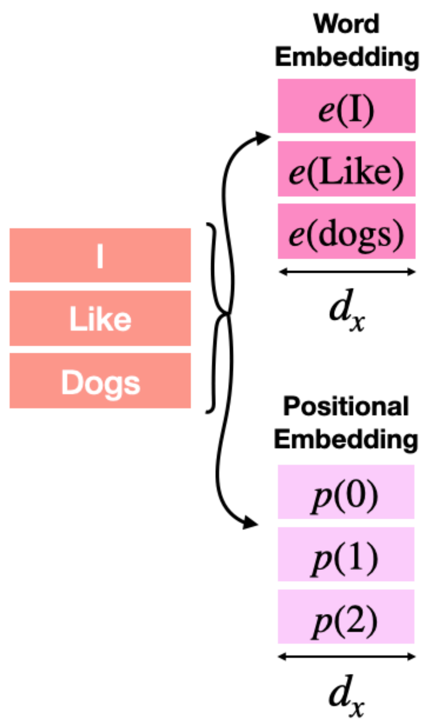
What would happen if they weren’t?

Okay, that’s our parts!

Recap: The main transformer components are:

Recap: The main transformer components are:

The Initial Sequences

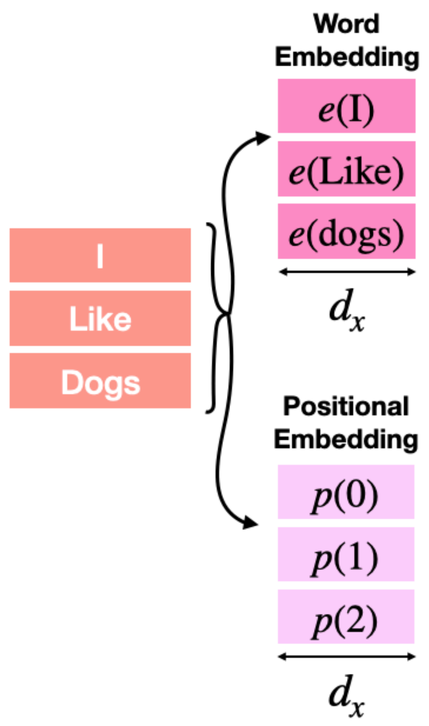


"indices"
and "tokens"

(…and "length"!)

Recap: The main transformer components are:

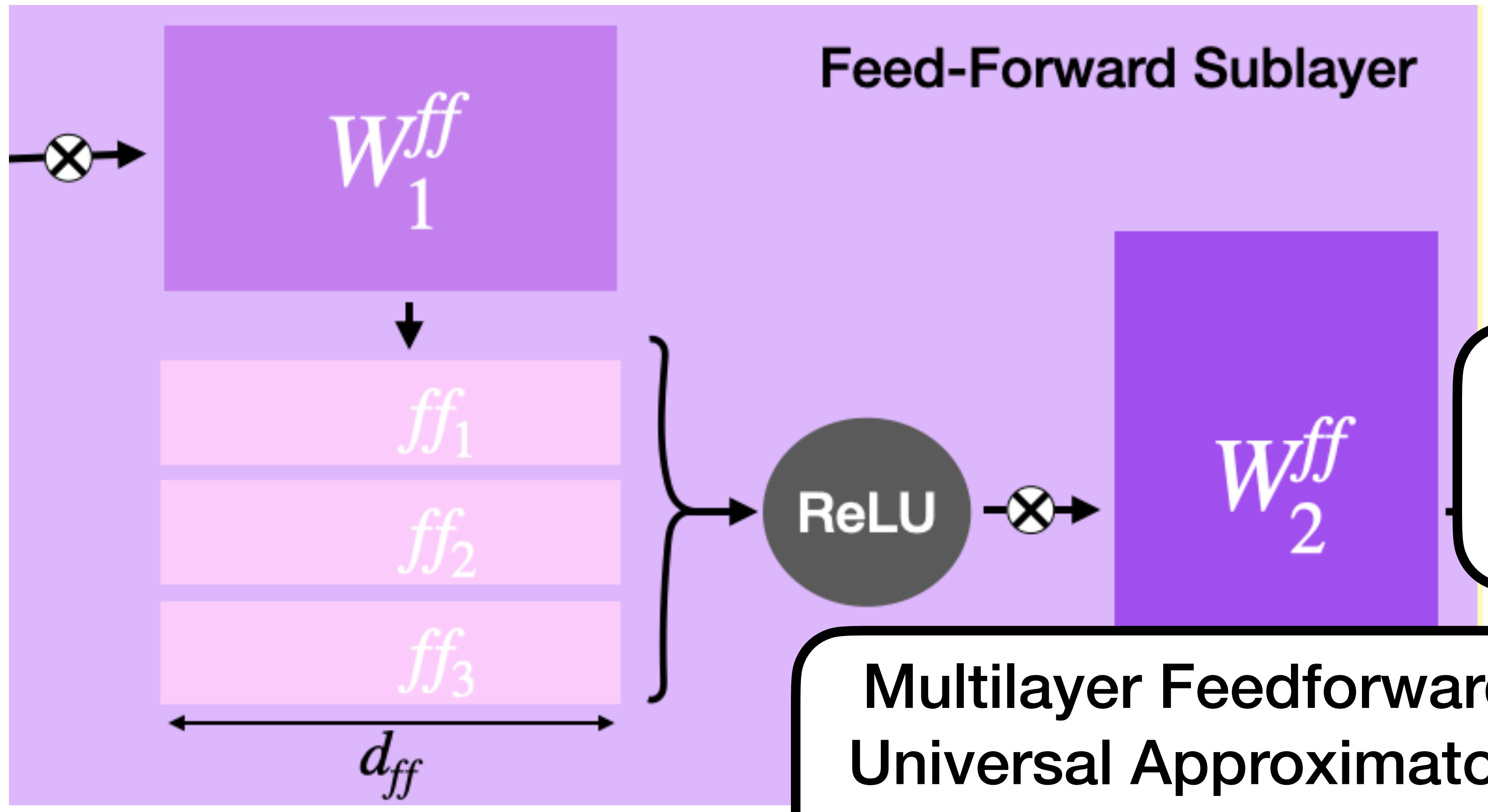
The Initial Sequences



"indices" and "tokens"

(…and "length"!)

Feed-Forward Sublayers

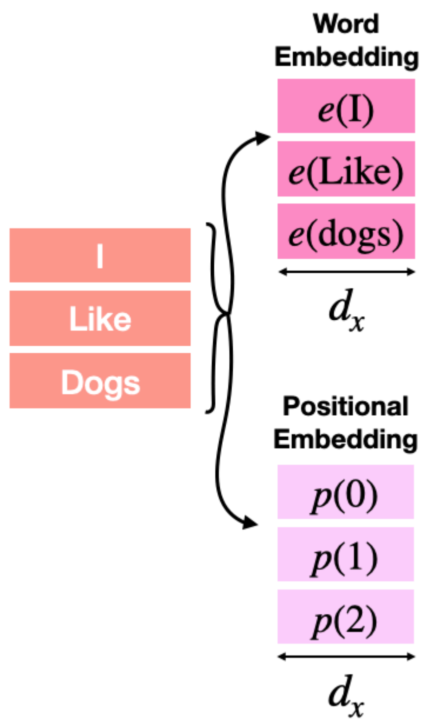


Powerful elementwise operations

Multilayer Feedforward Networks are Universal Approximators (Hornik et al, 1989)

Recap: The main transformer components are:

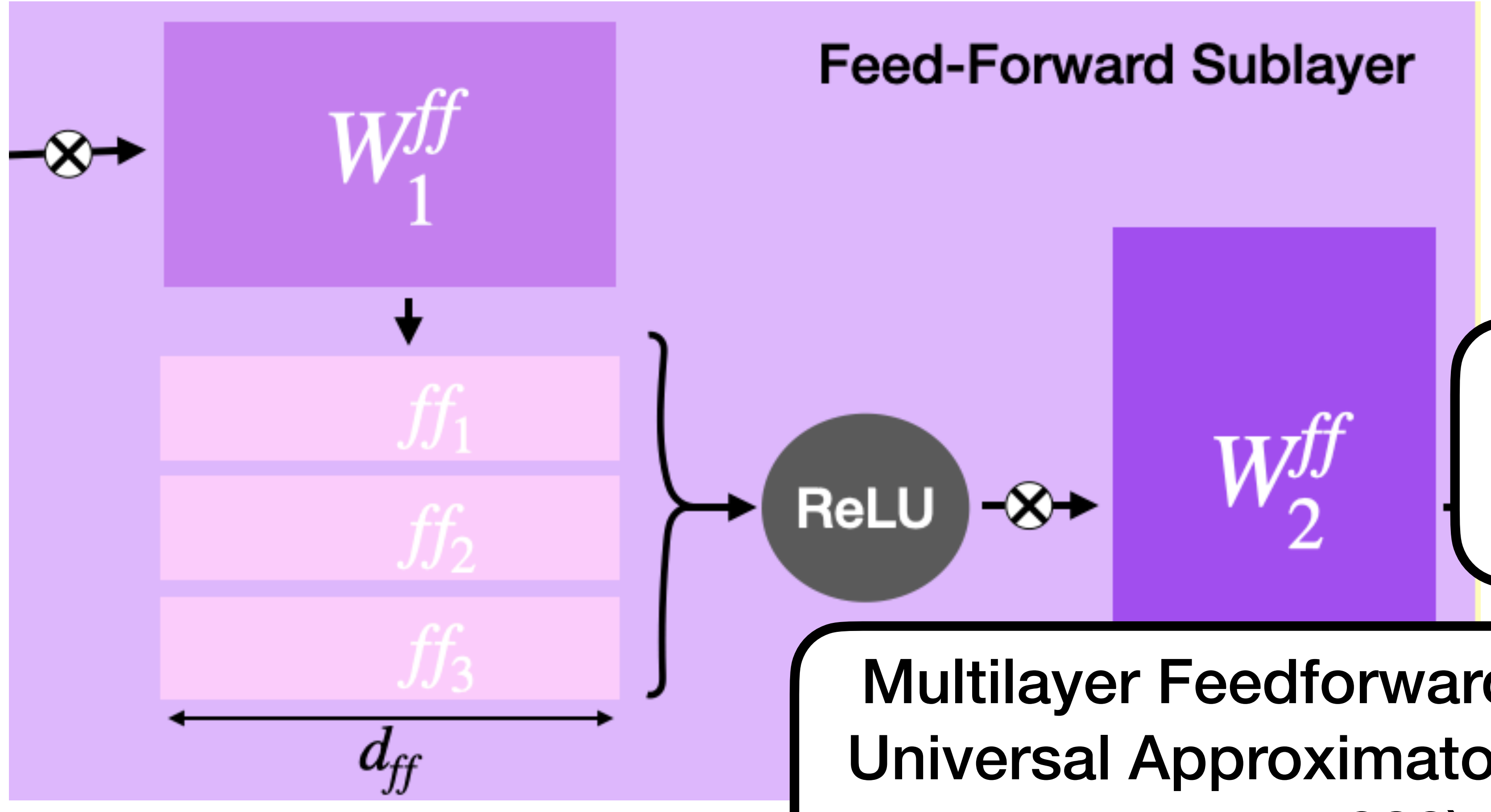
The Initial Sequences



"indices" and "tokens"

(...and "length"!)

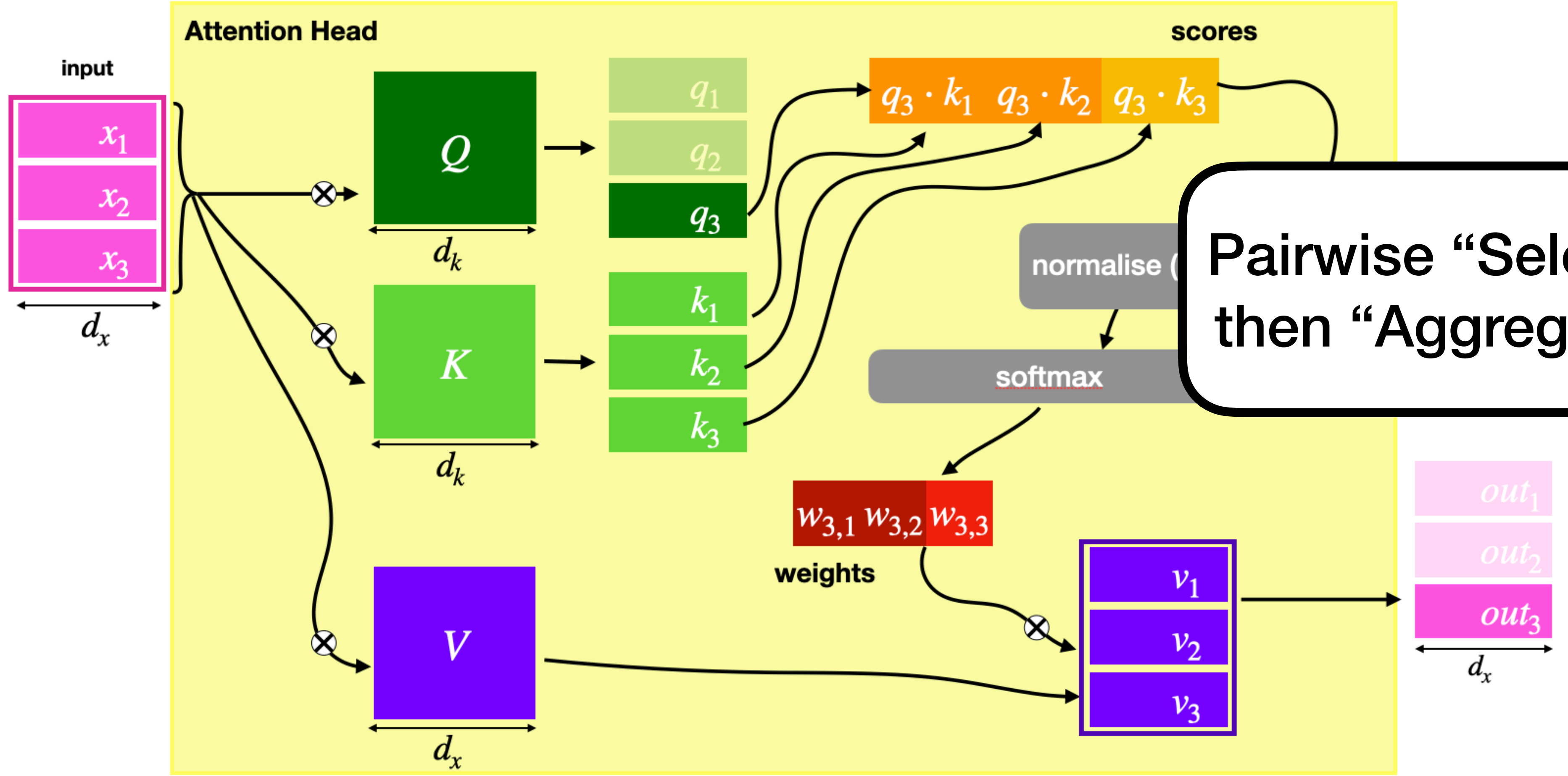
Feed-Forward Sublayers



Powerful elementwise operations

Multilayer Feedforward Networks are Universal Approximators (Hornik et al, 1989)

Attention Heads



Pairwise "Select", then "Aggregate"

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"

But also..

Recap: The main transformer components are:

The Initial Sequences

Feed-Forward Sublayers

Attention Heads

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

"indices" and "tokens"

(…and "length"!)

Feed-Forward Sublayers

Powerful elementwise operations

Multilayer Feedforward Networks are Universal Approximators (Hornik et al, 1989)

Skip connection

⊕

Deep Residual Learning for Image Recognition

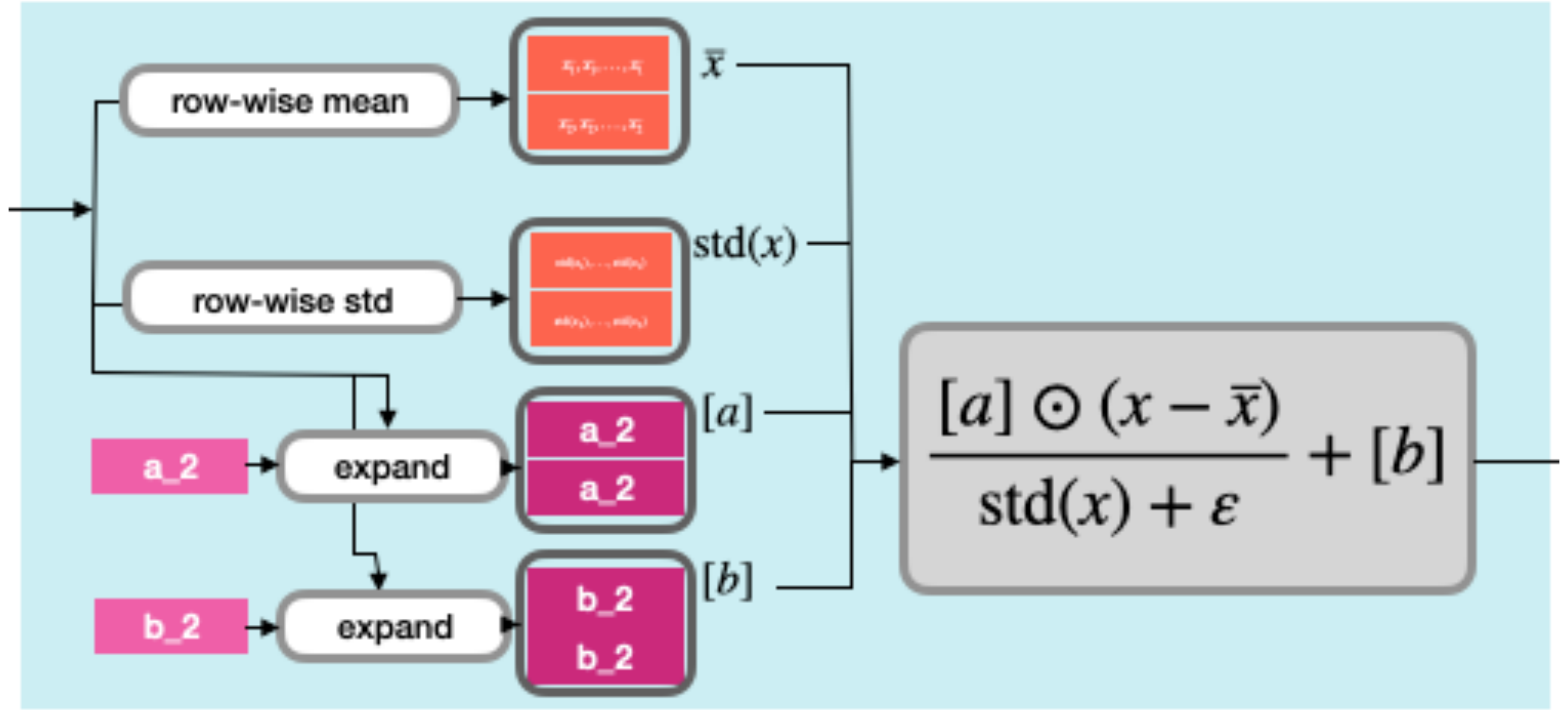
Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun

Encourages idea that information can be retained through many layers…

Attention Heads

Pairwise "Select", then "Aggregate"

Layernorms

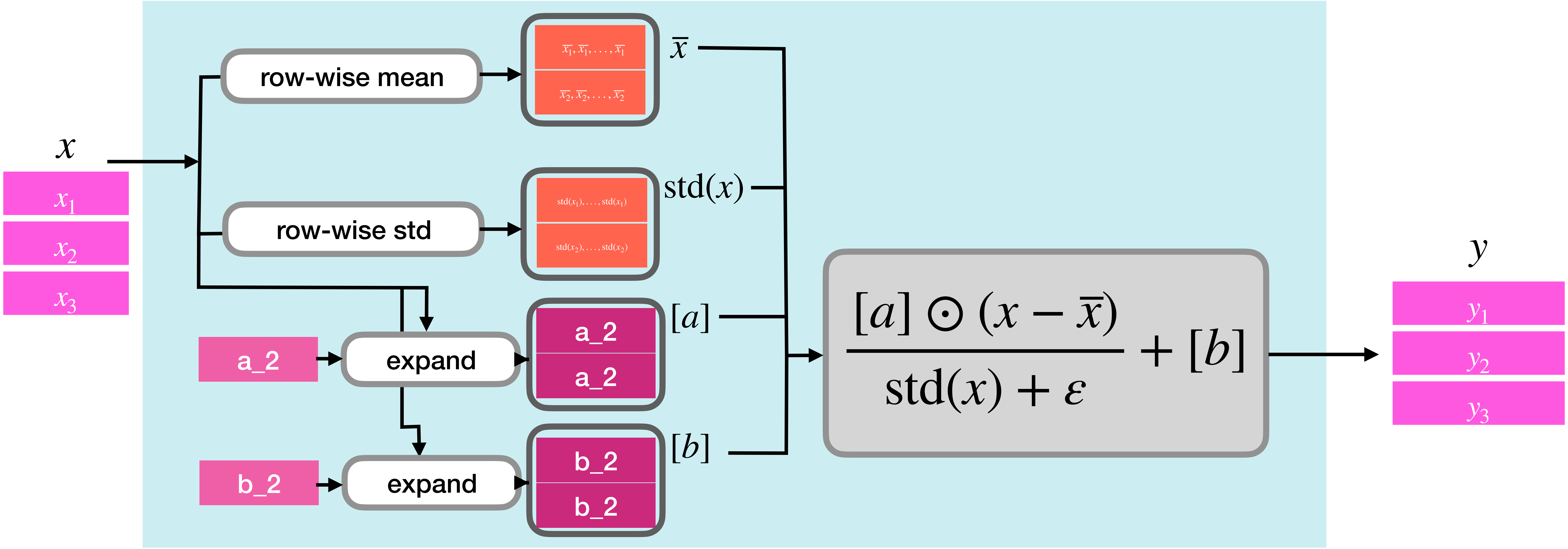


$$\frac{[a] \odot (x - \bar{x})}{\mathrm{std}(x) + \varepsilon} + [b]$$

LayerNorm

Parameters

- a_2 (vector)
- b_2 (vector)
- ε (small constant)



So the 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)

Skip connection

⊕

Deep Residual Learning for Image Recognition

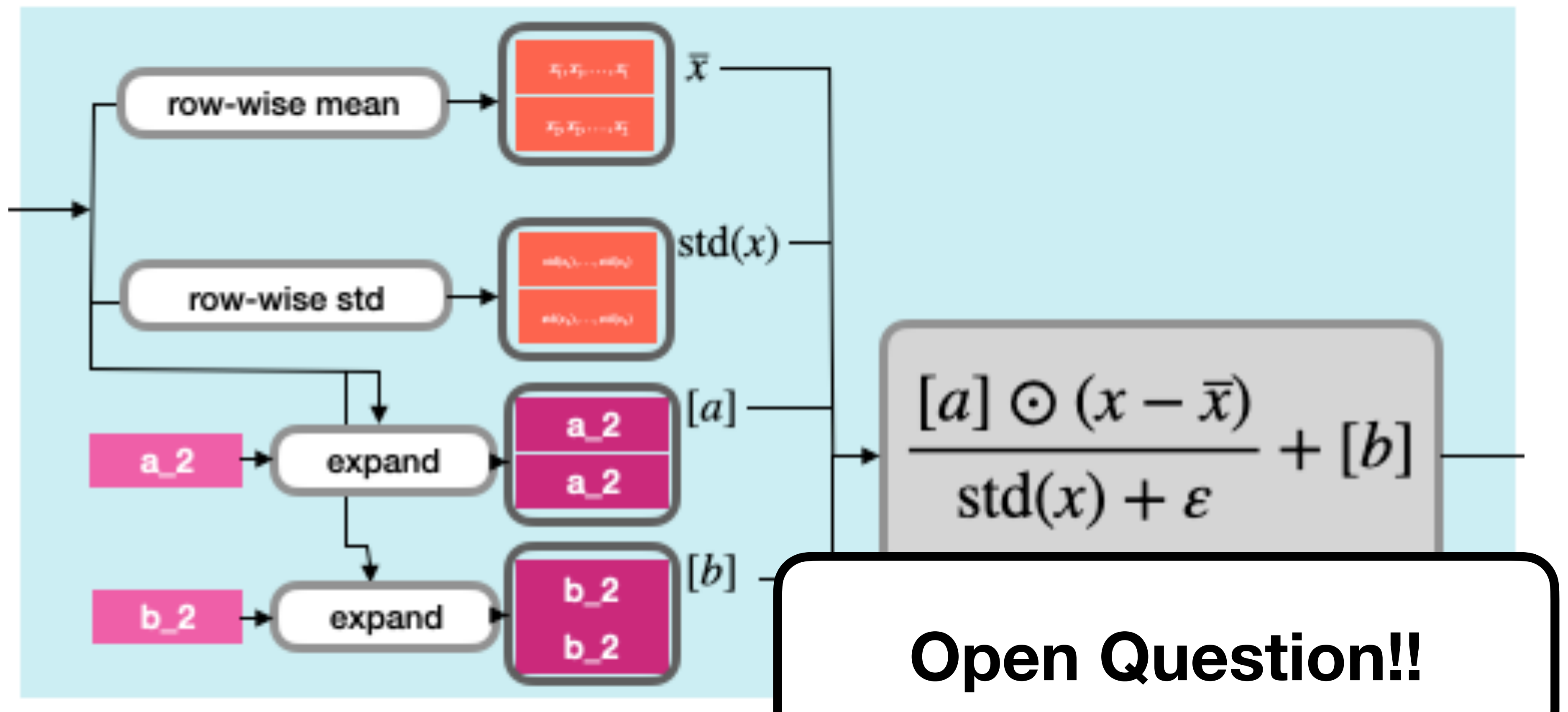
Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun

Encourages idea that information can be retained through many layers…

Attention Heads

Pairwise "Select", then "Aggregate"

Layernorms



$$\frac{[a] \odot (x - \bar{x})}{\mathrm{std}(x) + \varepsilon} + [b]$$

Open Question!!

RASP (Restricted Access Sequence Processing)

Initial Sequences

```
>> 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)

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

```
>> 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

Selectors, and aggregate

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

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

Small Example

Computing *length*:

Small Example

Computing *length*:

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

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
```

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

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
```

```
>> 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)
```

```
>> 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

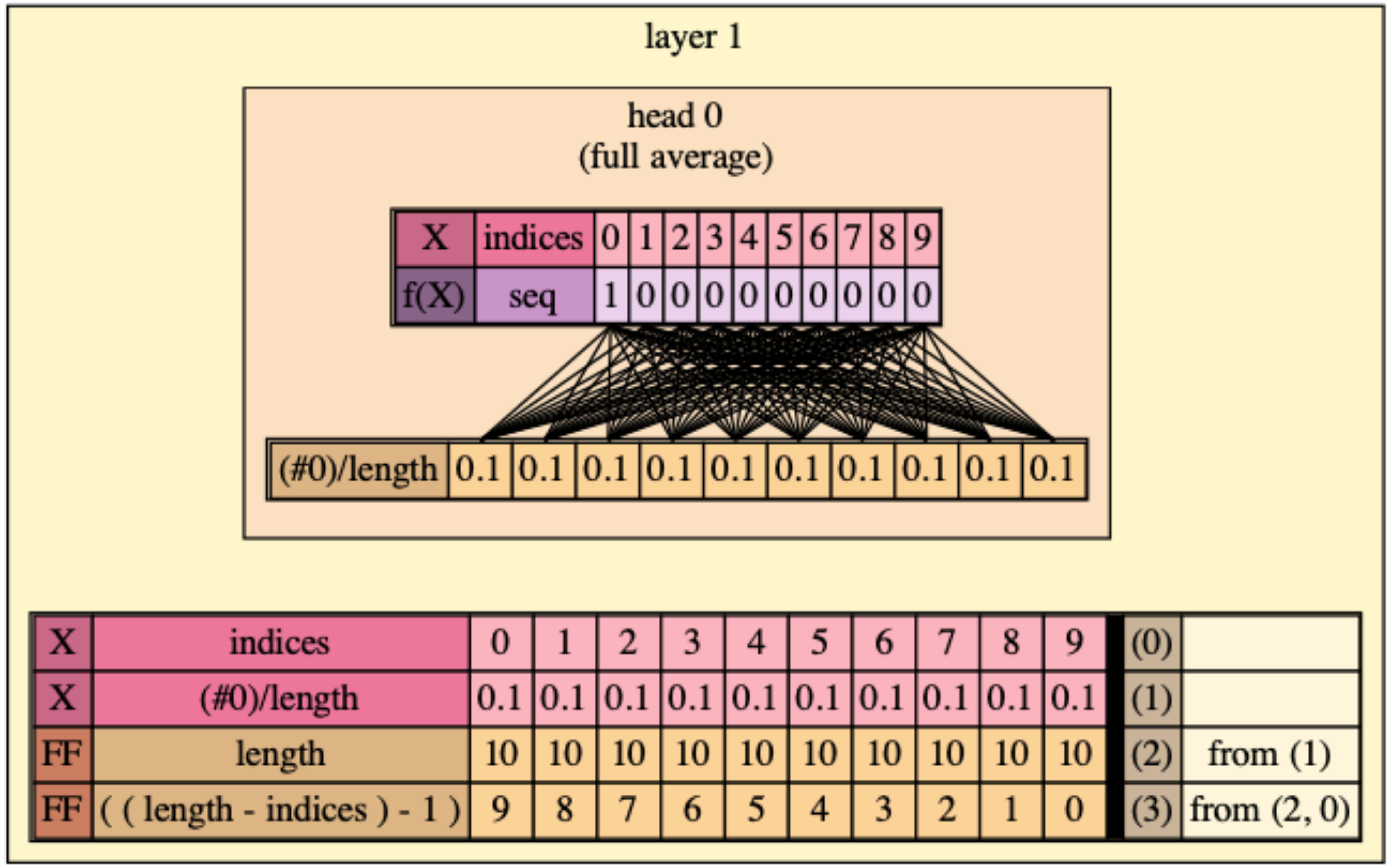
- select-aggregate pairs:
  must be at least one layer after all of their dependencies
  can have multiple pairs in one layer (multi-headed attention)

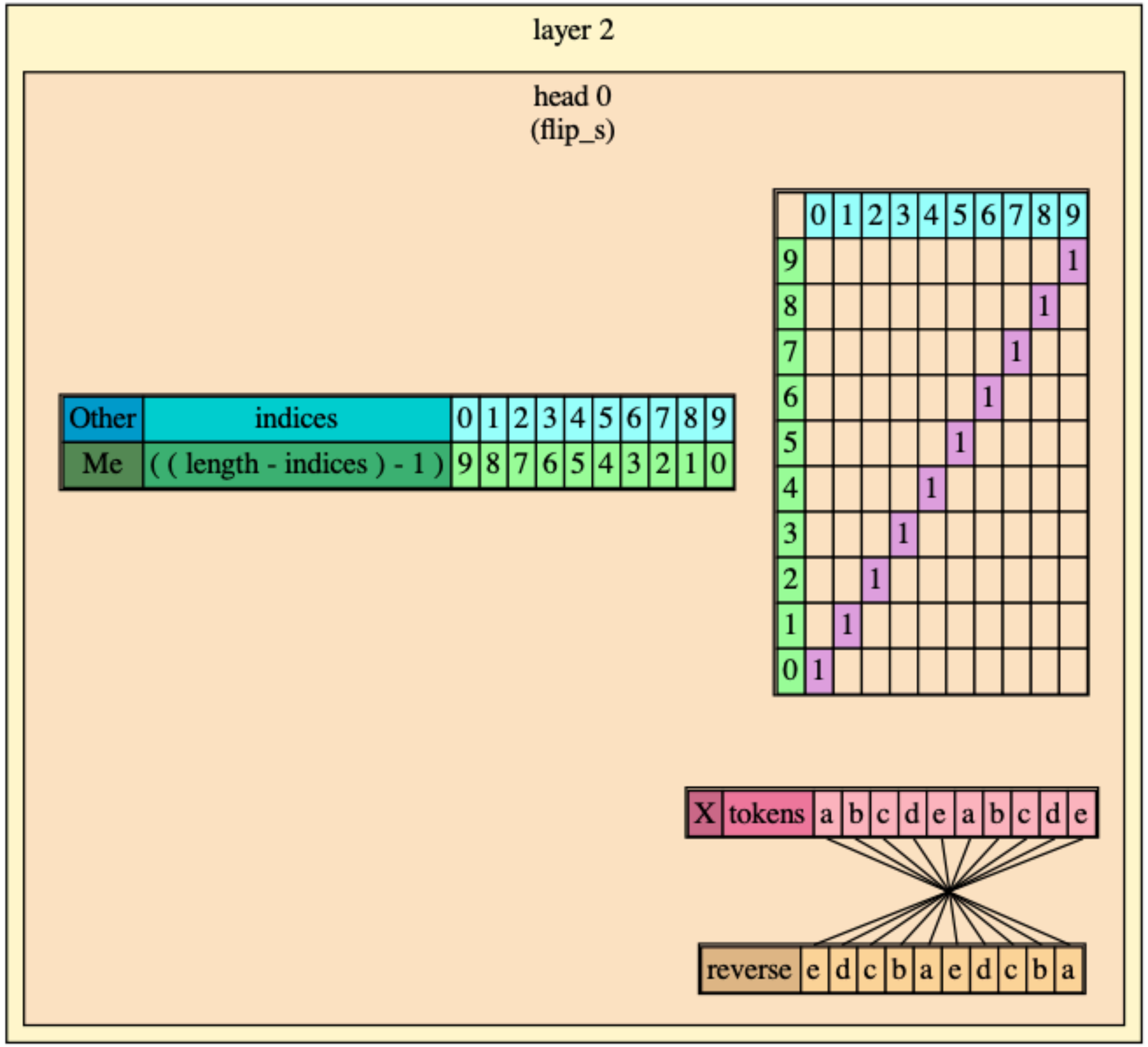
- local (feed-forward) operations:
  don’t add layers (attached to earliest layer after dependencies are finished)

RASP analysis?

Can draw head/layer analysis, eg:

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

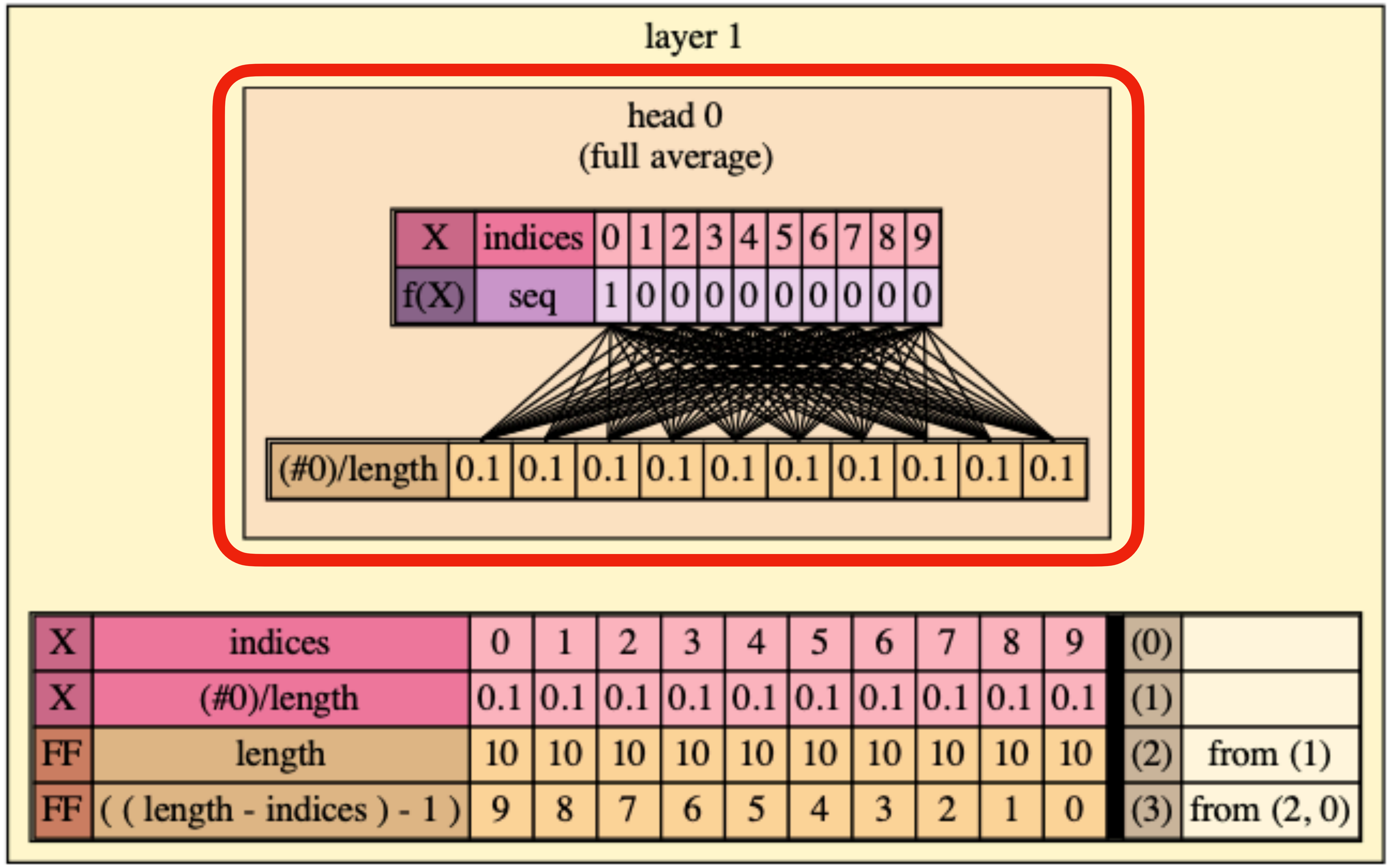


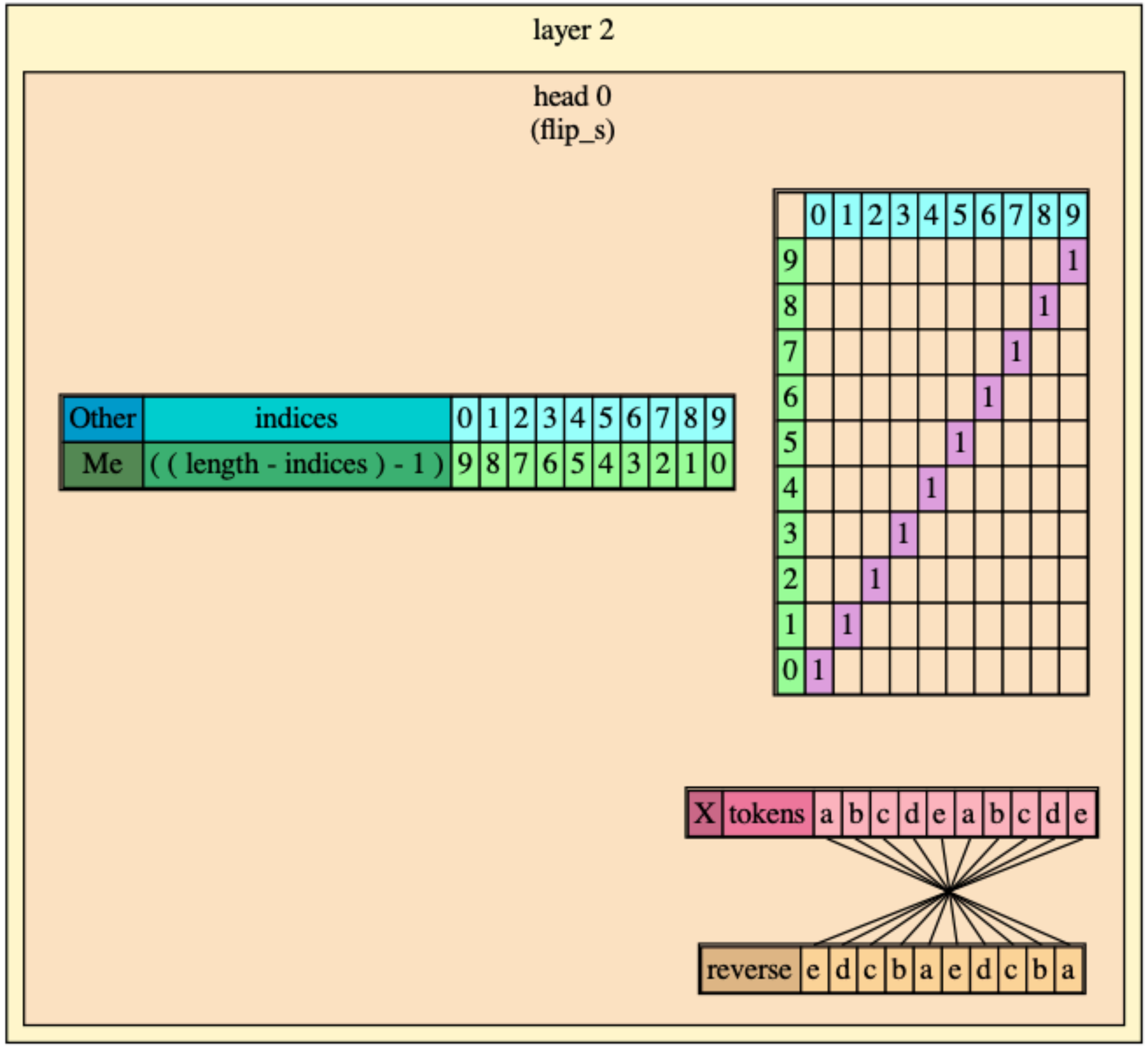


RASP analysis?

Can draw head/layer analysis, eg:

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

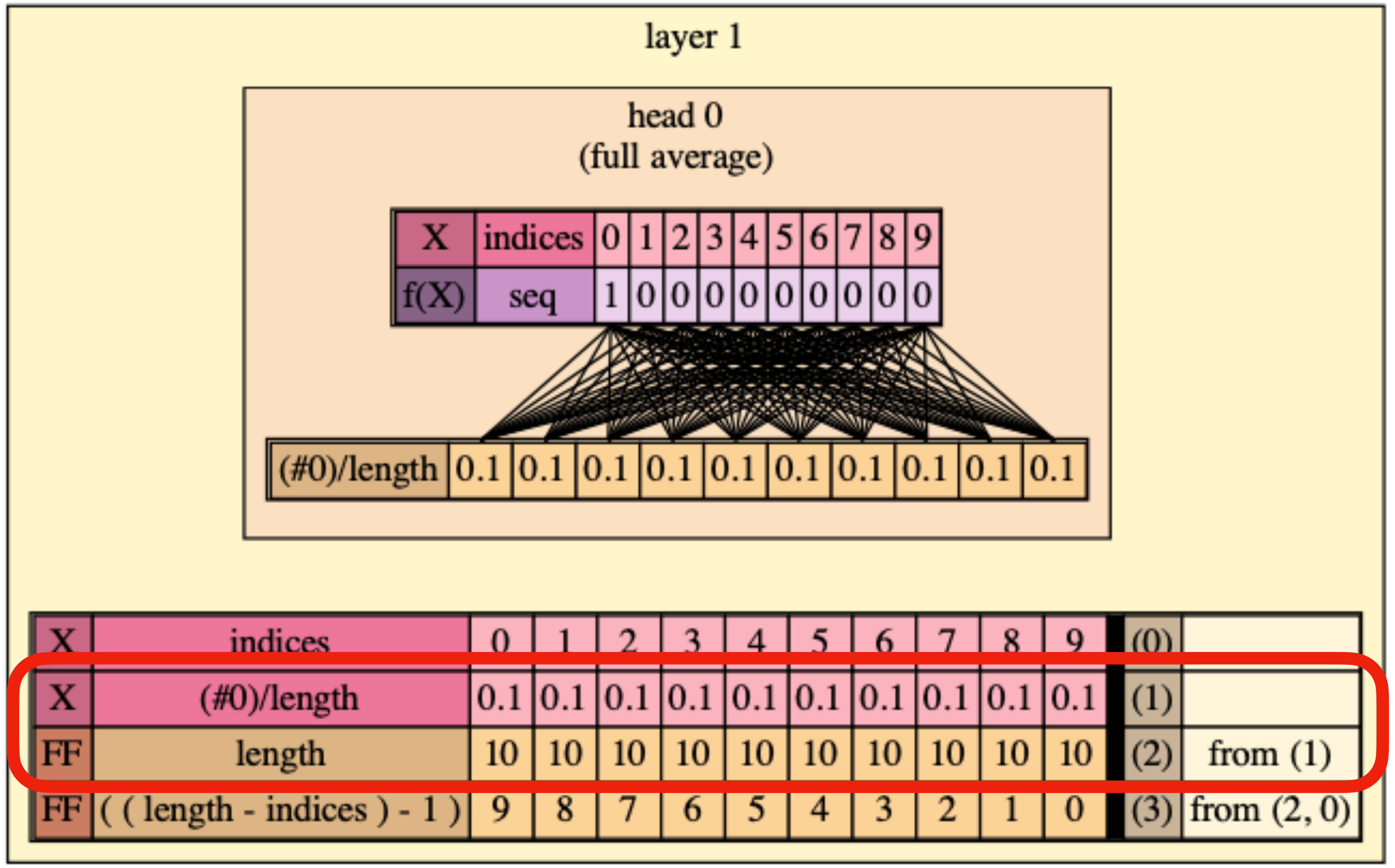


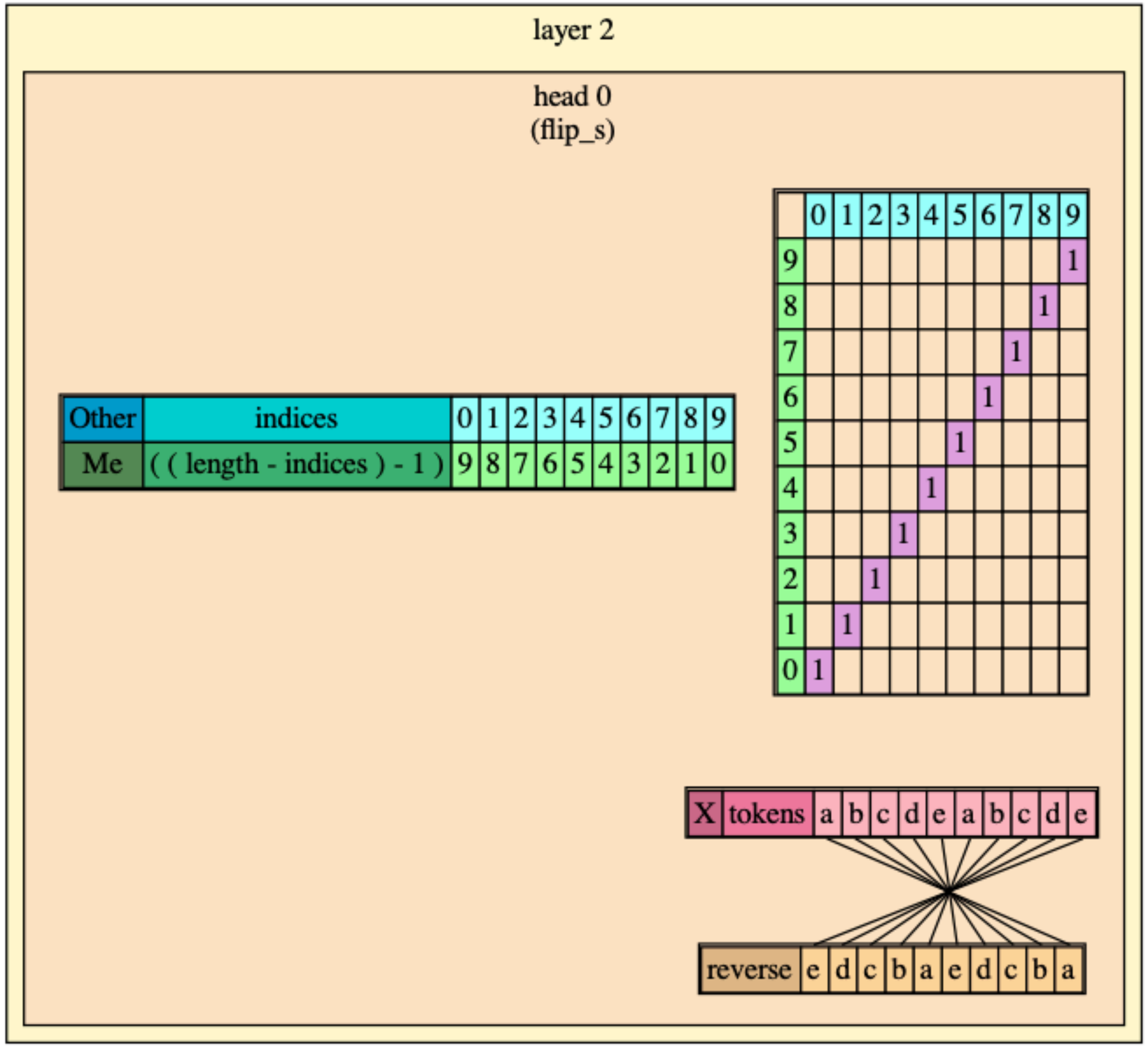


RASP analysis?

Can draw head/layer analysis, eg:

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

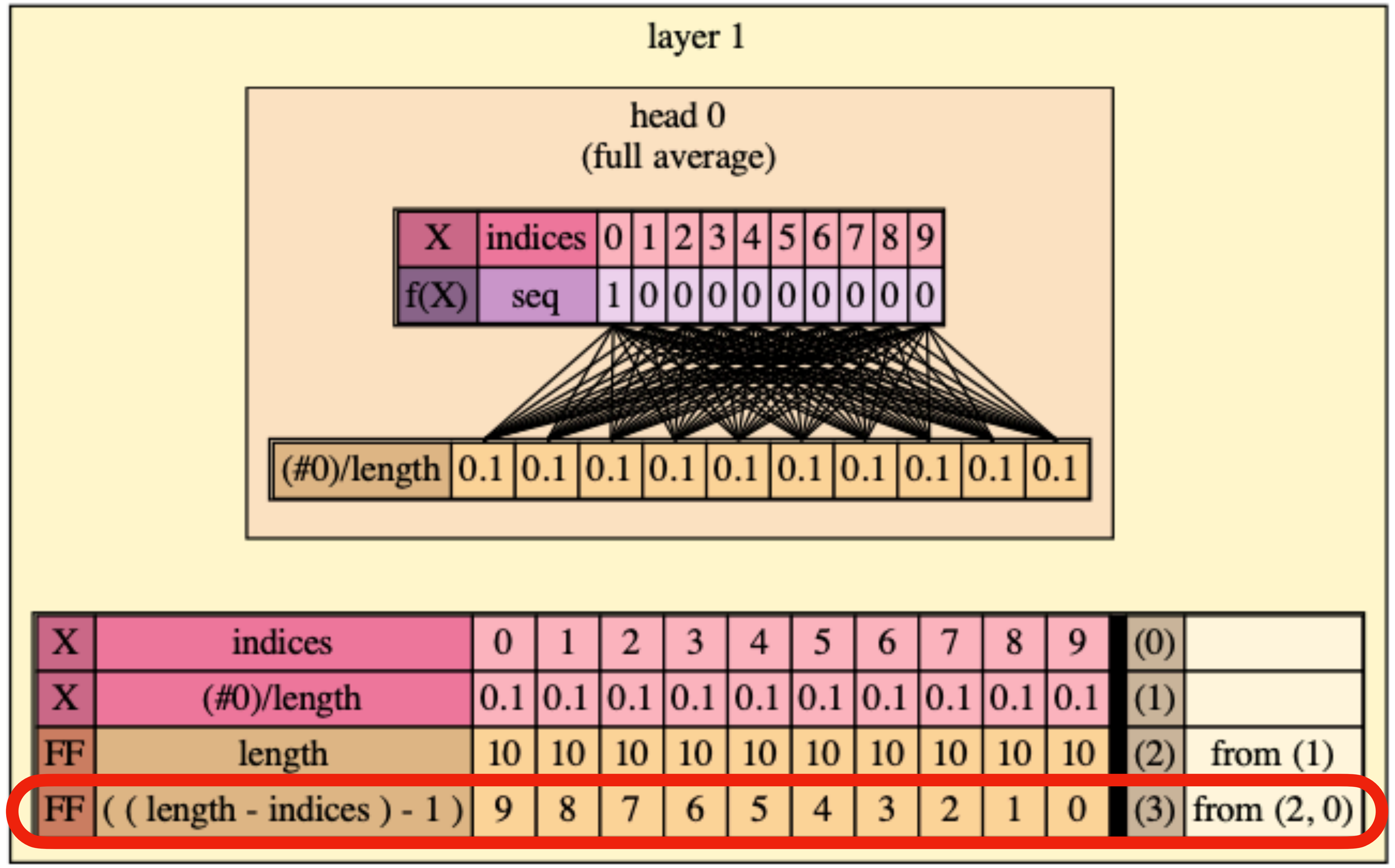


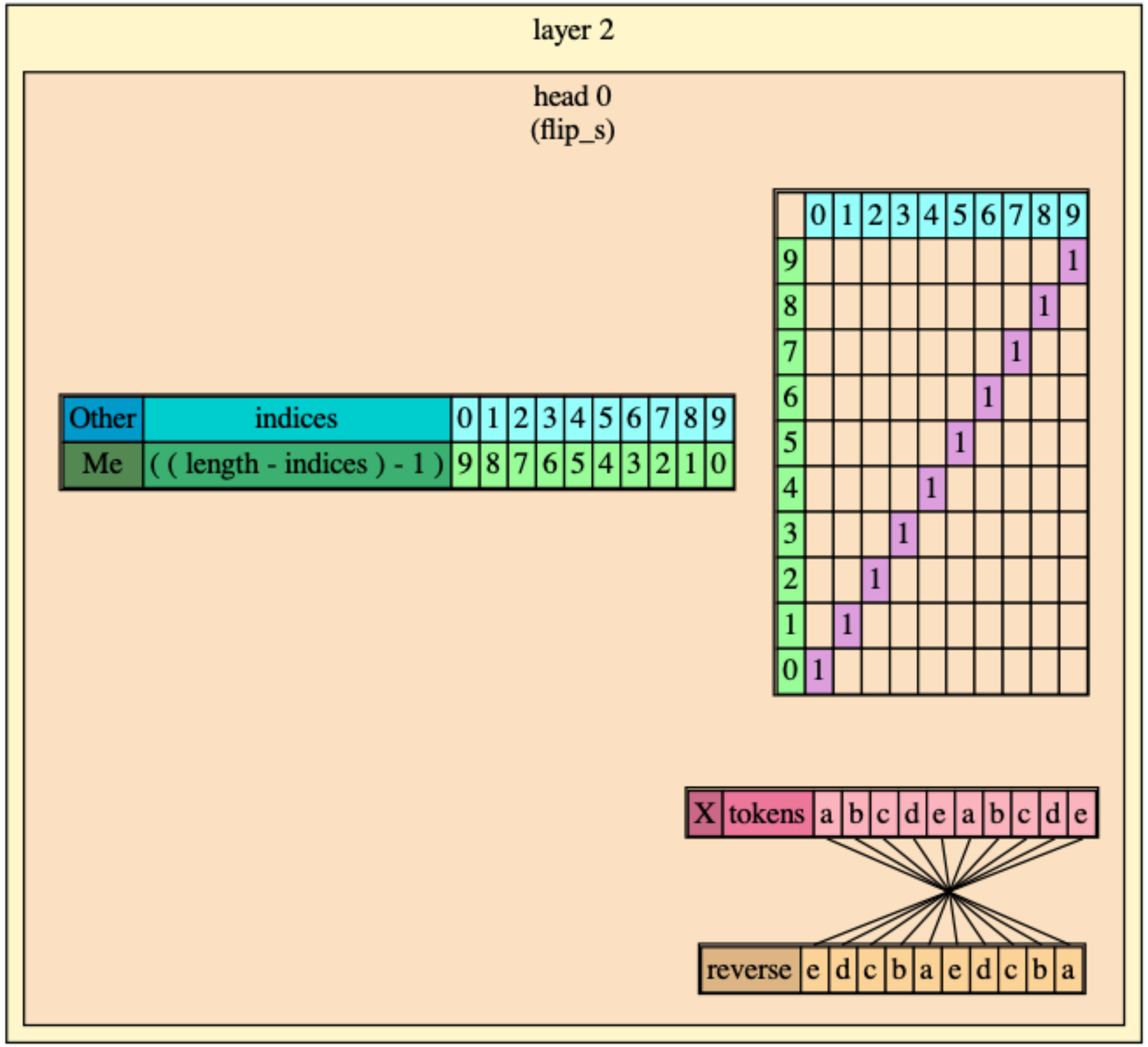


RASP analysis?

Can draw head/layer analysis, eg:

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

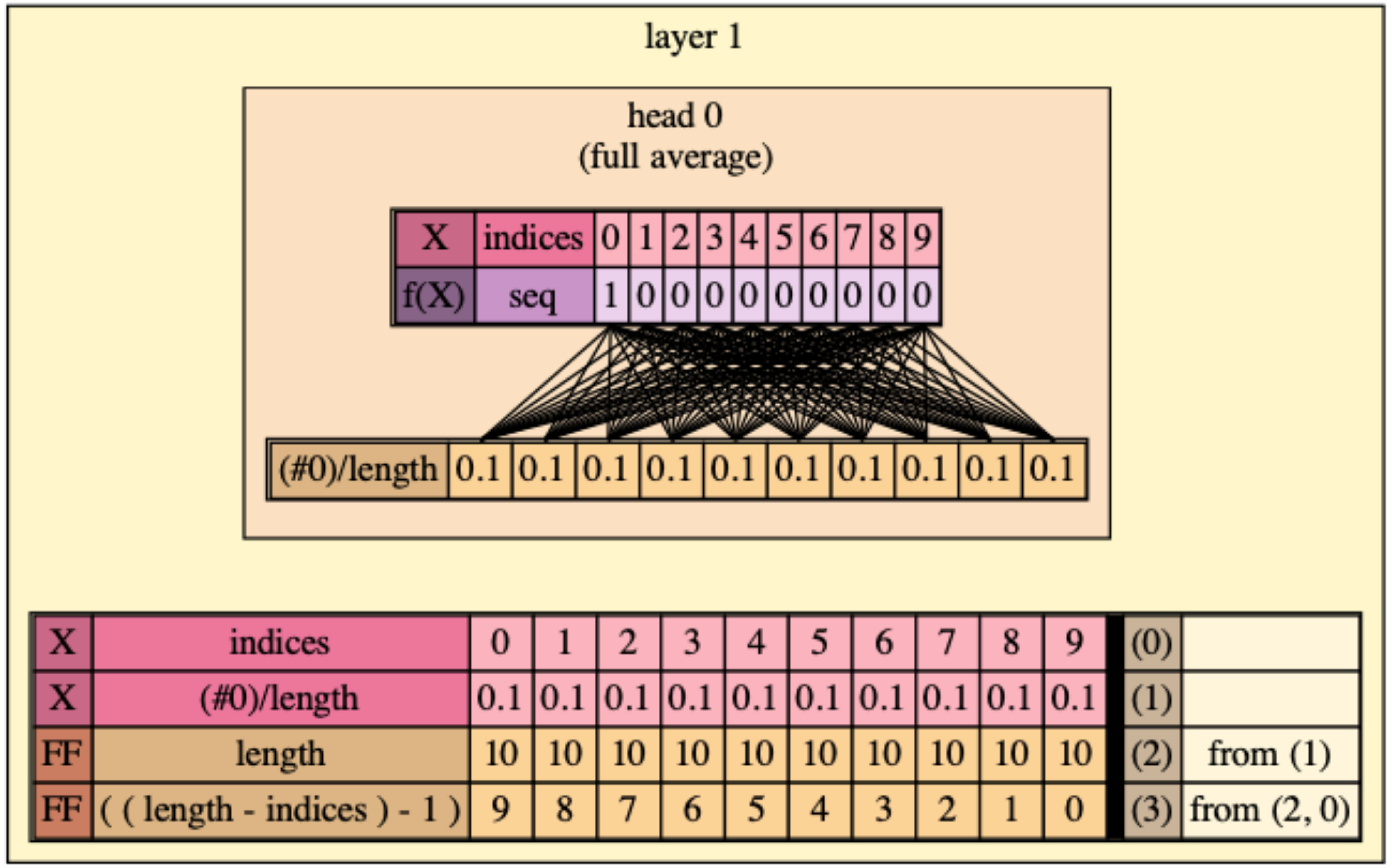


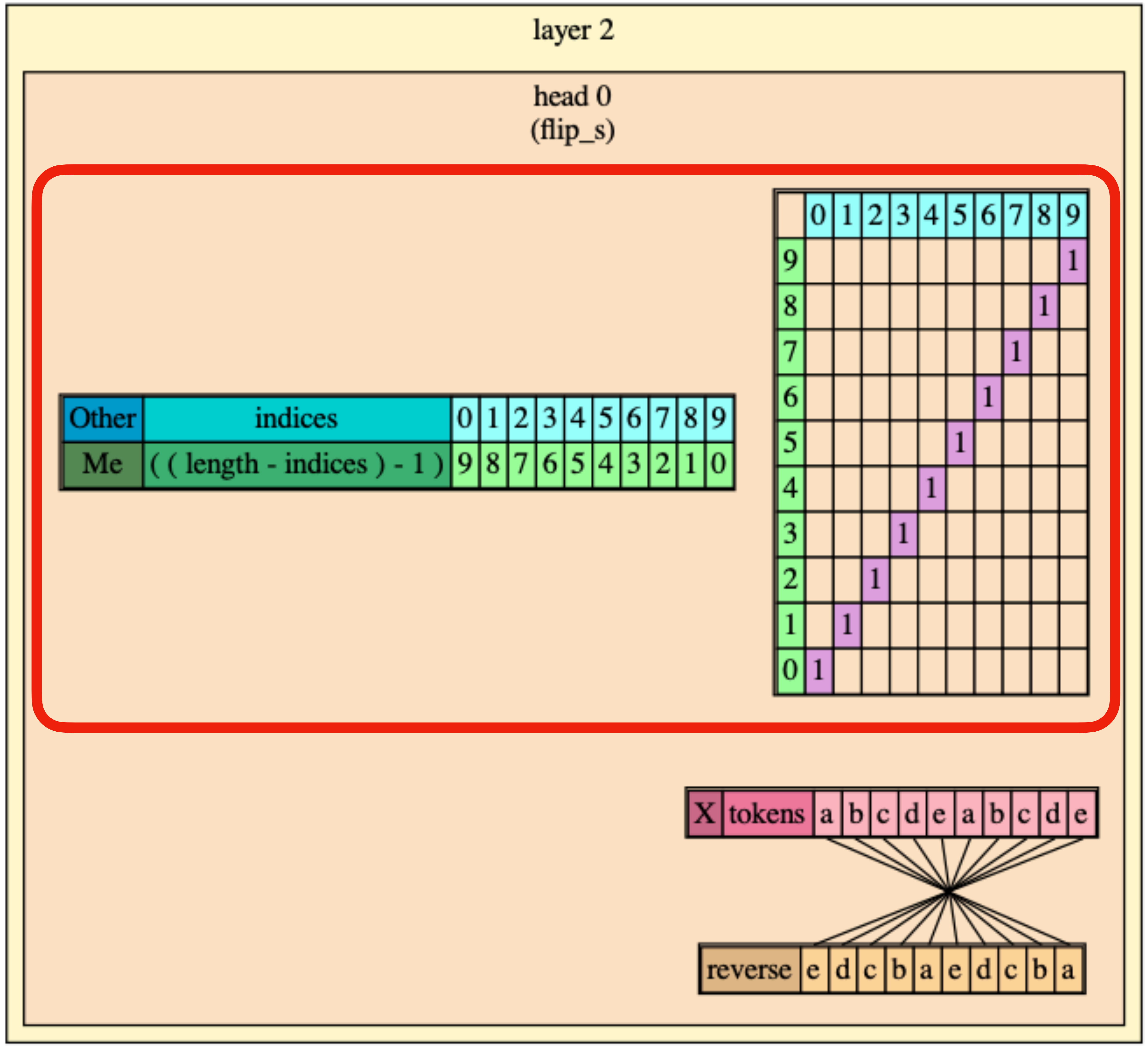


RASP analysis?

Can draw head/layer analysis, eg:

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

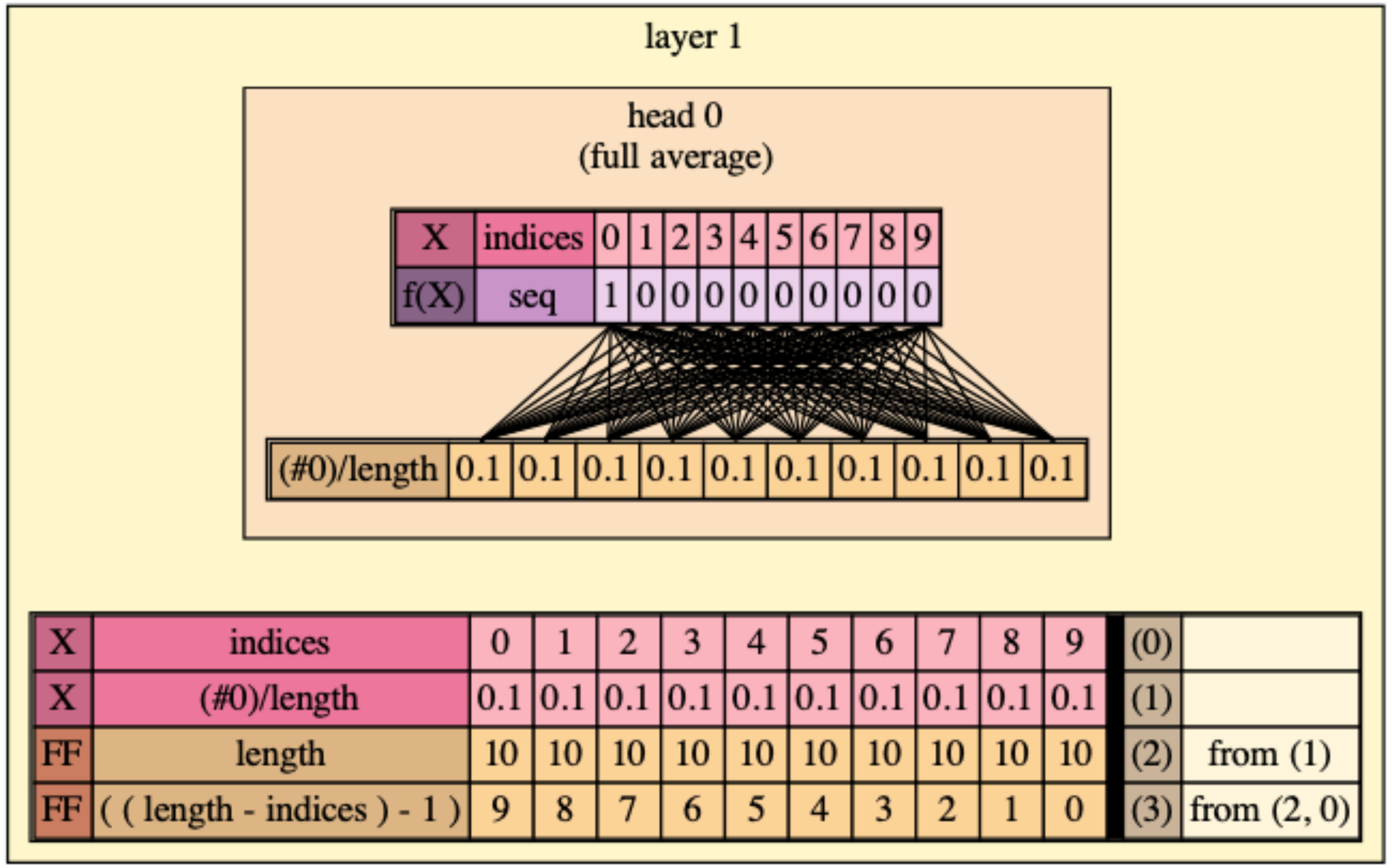


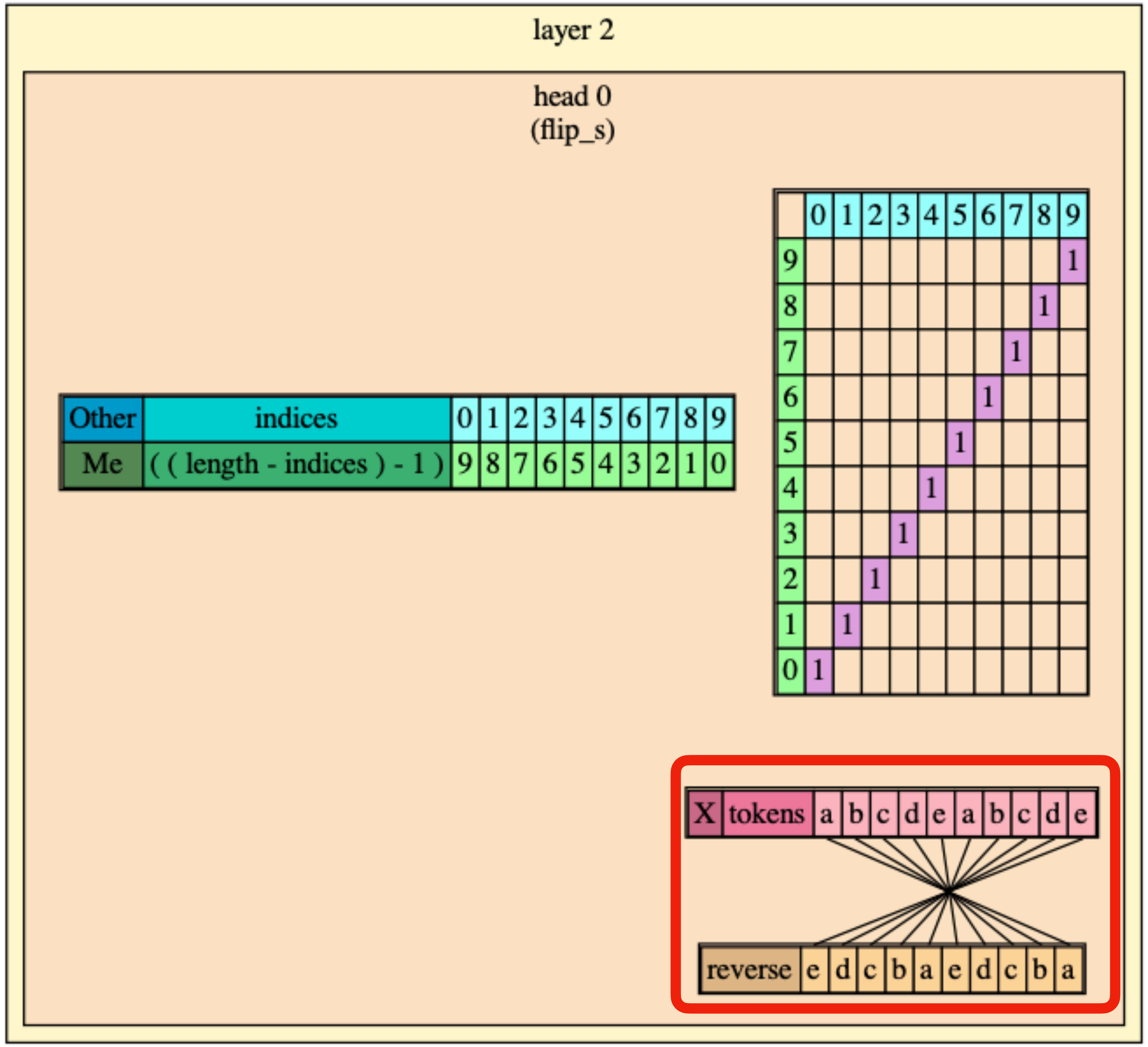


RASP analysis?

Can draw head/layer analysis, eg:

```
>> 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:
                                 h e l l o
                             h |         1
                             e |       1
                             l |     1
                             l |   1
                             o | 1
>> reverse=aggregate(flip_s,tokens);
     s-op: reverse
         Example: reverse("hello") = [o, l, l, e, h]
```

Connection to Reality?

Example 1: *reverse*

```
>> flip_s = select(length-indices-1,indices,==);
     selector: flip_s
         Example:
                                  h e l l o
                              h |         1
                              e |       1
                              l |     1
                              l |   1
                              o | 1
>> reverse=aggregate(flip_s,tokens);
     s-op: reverse
         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)

Connection to Reality?

Example 1: *reverse*

```
>> flip_s = select(length-indices-1,indices,==);
     selector: flip_s
         Example:
                                  h e l l o
                              h |         1
                              e |       1
                              l |     1
                              l |   1
                              o | 1
>> reverse=aggregate(flip_s,tokens);
     s-op: reverse
         Example: reverse("hello") = [o, l, l, e, h]
```

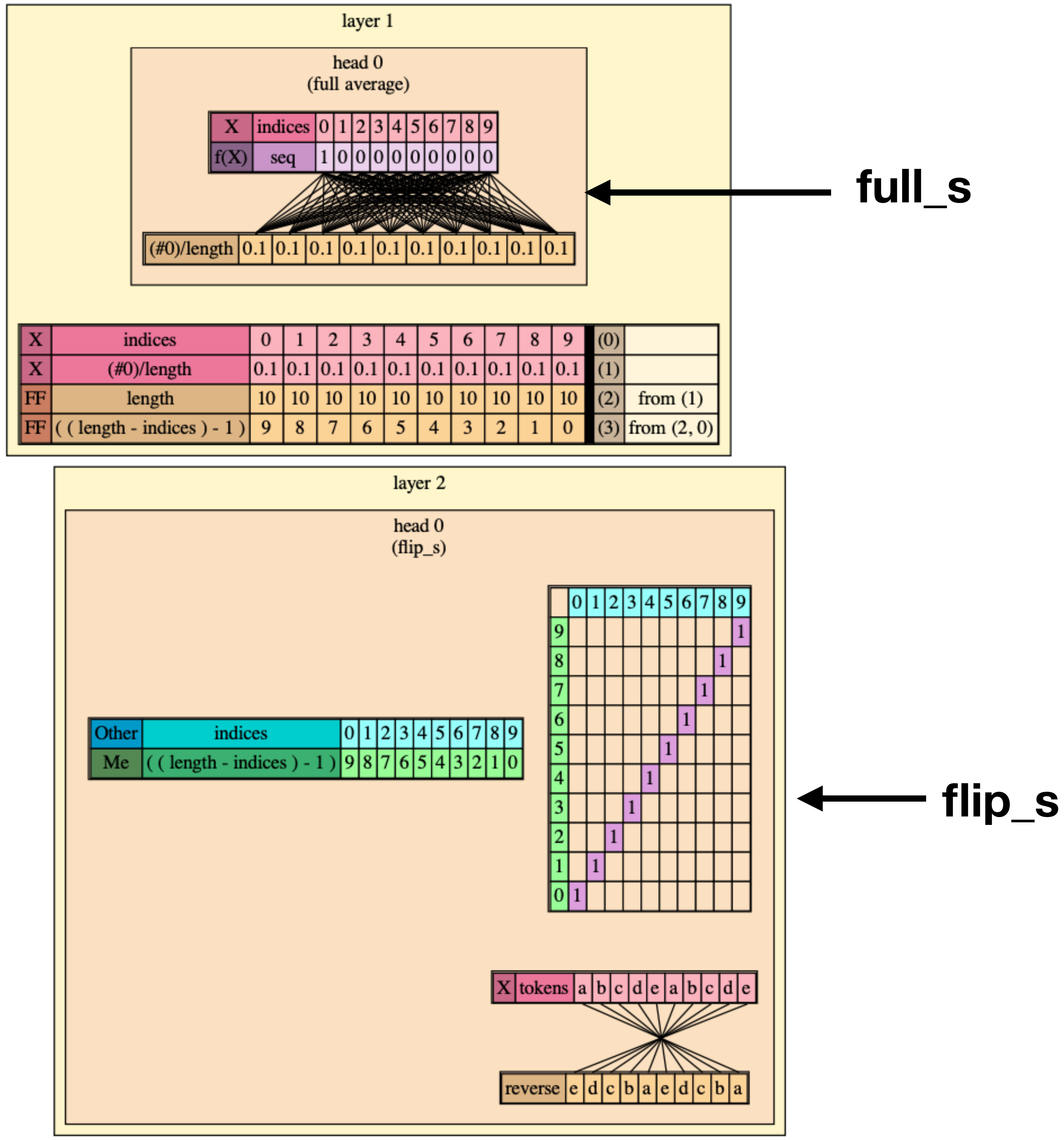
RASP analysis:

- 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?

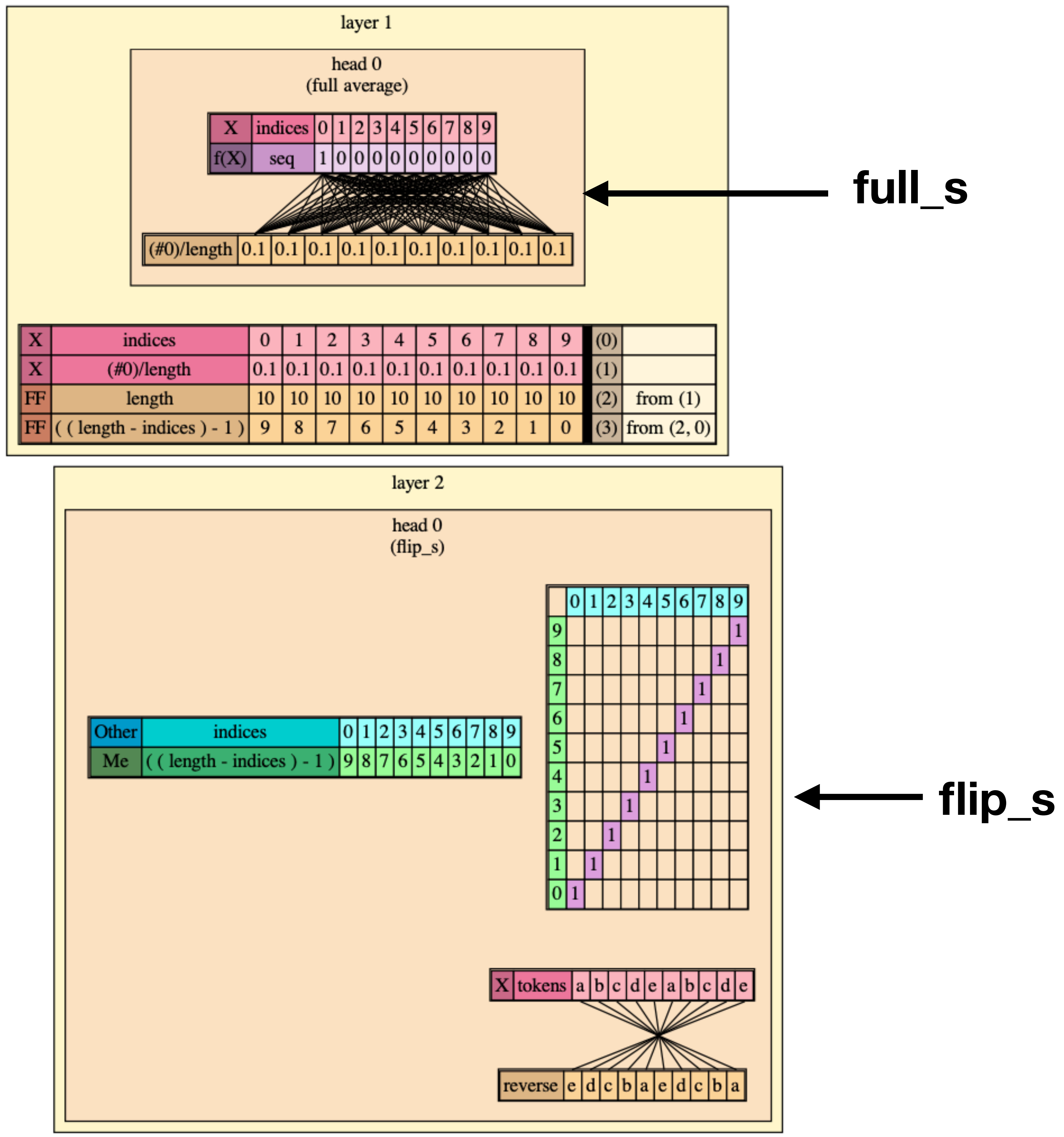
```
>> draw(reverse,"abcdeabcde")
```



RASP expects 2 layers for arbitrary-length reverse

Connection to Reality?

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



RASP expects 2 layers for arbitrary-length reverse

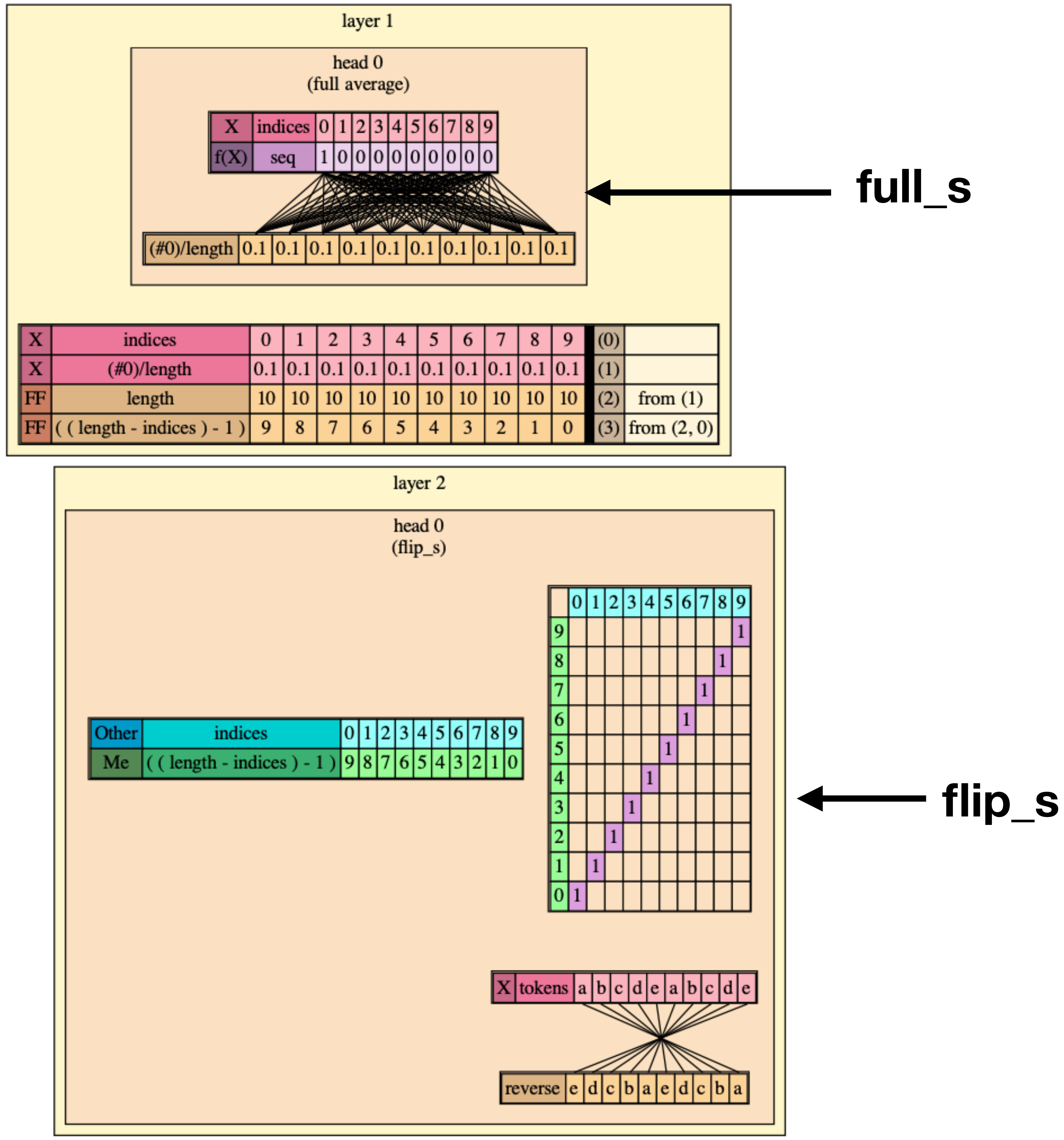
Test:
Training small transformers on lengths 0-100:

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

Even with compensation for
number of heads and parameters!

Connection to Reality?

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



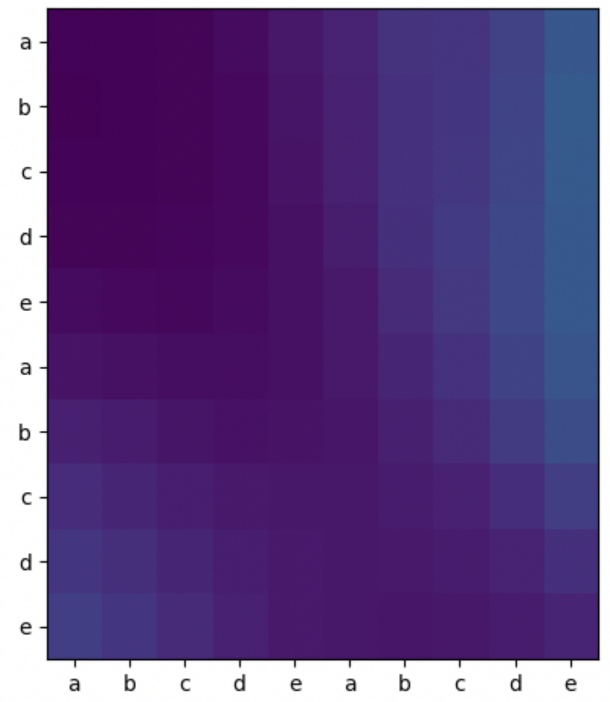
RASP expects 2 layers for arbitrary-length reverse

Test:
Training small transformers on lengths 0-100:

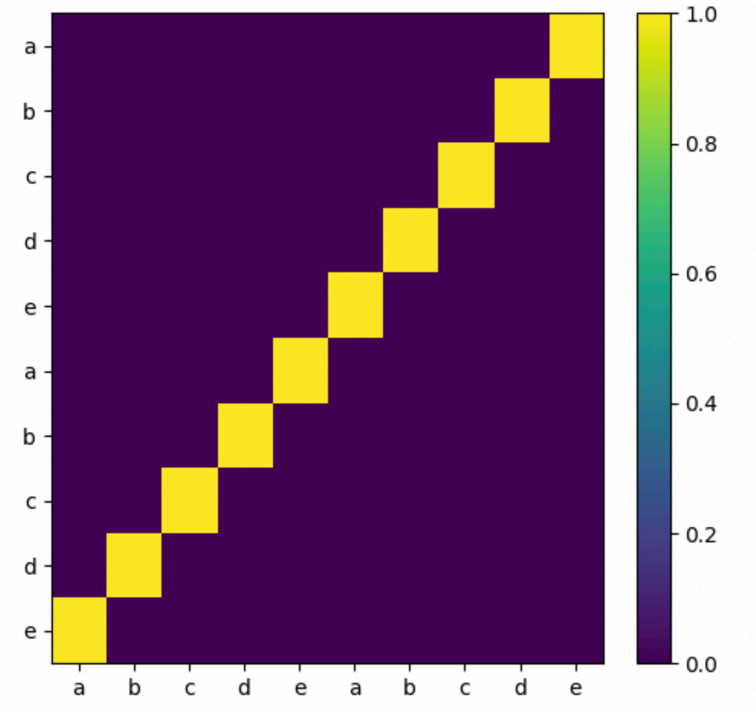
2 layers: **99.6**% accuracy after 20 epochs
1 layer: **39.6**% accuracy after 50 epochs

Bonus: the 2 layer transformer's attention patterns:

Layer 1 (*full_s*)



Layer 2 (*flip_s*)



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]

Connection to Reality?

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

Reminder: computing length

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			

Eg:

[§,h,e,l,l,o] $\mapsto$ [0,1,1,2,2,1]

[§,a,b,c,c,c] $\mapsto$ [0,1,1,3,3,3]

[§,a,b,a] $\mapsto$ [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]

Reminder: computing length

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
```

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)

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]

Reminder: computing length

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	

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)

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]

Reminder: computing length

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			

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?

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]

Reminder: computing length

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	

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)

```
>> set example "§hello"
>> same_or_0 = select(tokens,tokens,==) or select(indices,0,==);
     selector: same_or_0
         Example:
                             § h e l l o
                         § | 1
                         h | 1 1
                         e | 1   1
                         l | 1     1 1
                         l | 1     1 1
                         o | 1         1
```

Connection to Reality?

Example 2: *histogram* (assuming BOS)

```
>> set example "§hello"
>> same_or_0 = select(tokens,tokens,==) or select(indices,0,==);
     selector: same_or_0
         Example:
                                  § h e l l o
                              § | 1
                              h | 1 1
                              e | 1   1
                              l | 1     1 1
                              l | 1     1 1
                              o | 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)
```

Connection to Reality?

Example 2: *histogram* (assuming BOS)

```
>> set example "§hello"
>> same_or_0 = select(tokens,tokens,==) or select(indices,0,==);
   selector: same_or_0
       Example:
                            § h e l l o
                        § | 1
                        h | 1 1
                        e | 1   1
                        l | 1     1 1
                        l | 1     1 1
                        o | 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)

```
>> 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)

```
>> 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?

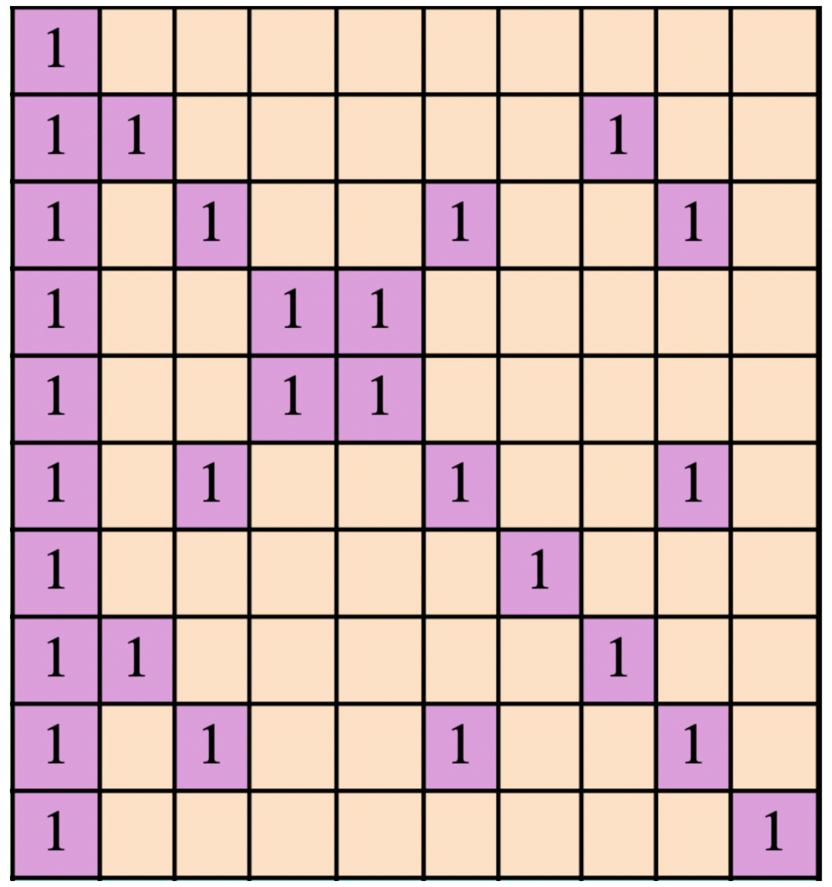
Example 2: *histogram* (assuming BOS)

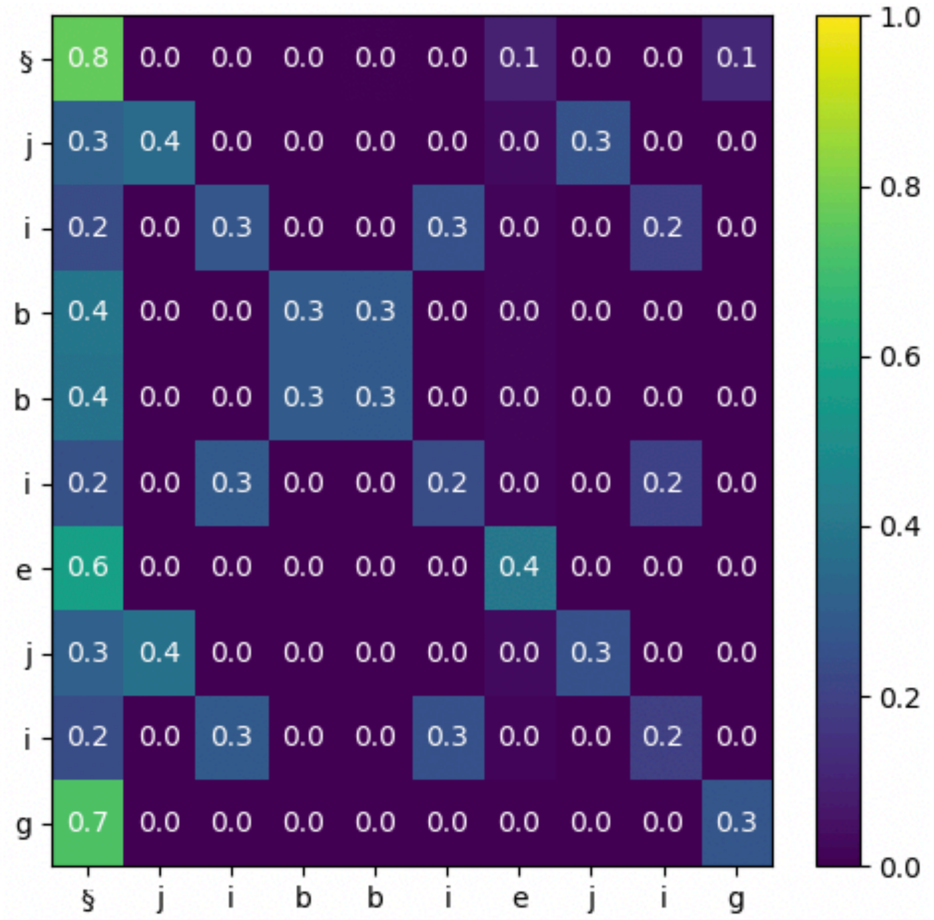
```
>> examples off
>> same_or_0 = select(tokens,tokens,==) or select(indices,0,==);
     selector: same_or_0
>> frac_with_0 = aggregate(same_or_0,indicator(indices==0));
     s-op: frac_with_0
>> histogram_assuming_bos = round(1/frac_with_0)-1;
     s-op: histogram_assuming_bos
>> histogram_assuming_bos("§hello");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

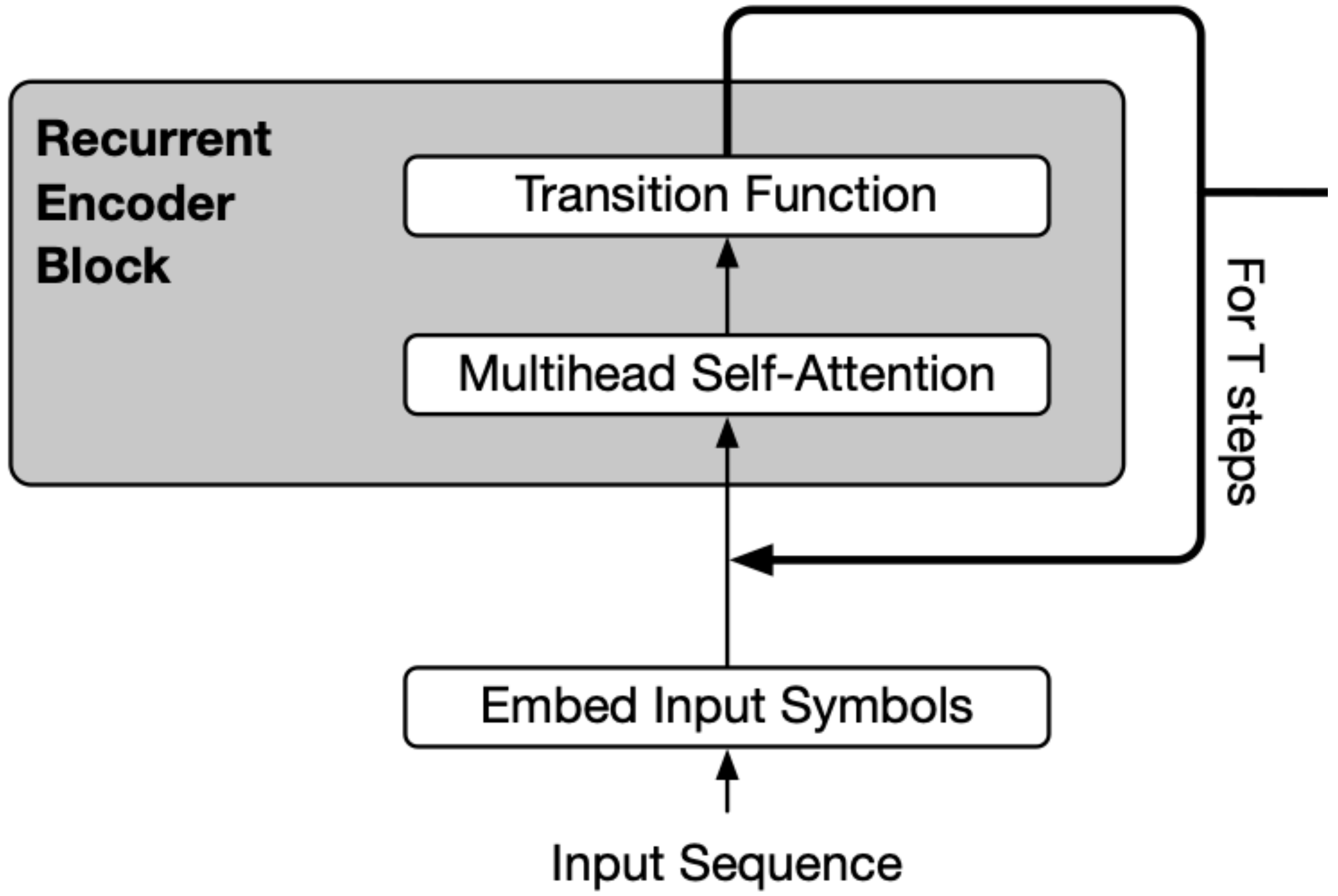
Selector pattern vs trained transformer's attention for same input sequence:





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

Insight

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

Model	PPL ↓
sfffssfsfsfssffffsfsffsffffff	22.80
sffssfsssssssssssssfsfsssfsffsssfsssfs	21.02
ssssssffsffffssfffffsssfsfsssssssss	20.98
fffffffffsffssffsffssssfsfsssf	20.75
fssfsssffffffssfsssfsfffssssfsfss	20.43
sffsffffffsfsfssfsssfsfsfssfssfs	20.28
sffssffsfffsfsfssssffffffssssff	20.02
fsffsfssffffsfsfffsfffssfffsss	19.93
sffsffssffsfsffsssfsssssfsssfffsss	19.85
ssfffffffssfffssfssffsfsfsffsf	19.82
sfsfsfffsfffssfsfffsffssfsfsfss	19.77
sfsffsssffsffsssfssfffffssssfsssf	19.55
sffsfssfffsffsfssssfsfsffffsfsss	19.49
sffffsffssssfsssfssfffsssfssssfsfs	19.47
fsssffssssssfsfsfsffsffffssfsfssss	19.25
sfsfsfsfsfsfsfsfsfsfsfsfsfsfsfsf	**19.13**
fssssssfsfsfsfffsfsssfssffssssfsff	18.86
sfsfsfsfsfsfsfsfsfsfsfsfsfsfsfsf	**18.83**
ssfsfsssfsssssffsfsfsssfssfsfsssssssf	18.62
sfsfsfsfsfsfsfsfsfsfsfsfsfsfsfsf	**18.54**
sfsfsfsfsfsfsfsfsfsfsfsfsfsfsfsf	**18.49**
sssfsffsfssfsssffsffffffssfsfff	18.34
sssfsfsffsssfsfffffsfsffffsssff	18.31
sfsfsfsfsfsfsfsfsfsfsfsfsfsfsfsf	**18.25**
ssssssfsssffffsfsfffffffffffsf	18.12

s self-attention

f feed-forward

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:

```
>> 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:

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

This can be used to implement sort:

```
>> 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:

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

This can be used to implement sort:

```
>> 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?

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

End

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

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 *and*, *or*, and *not* between selectors?

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