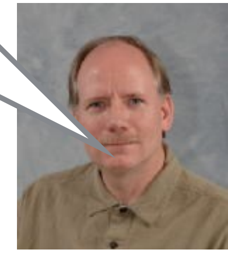
Attention & Transformers for Text Classification

(Vaswani et. al. 2017)
(Slides/Figures by Jay Alammar, original papers)

Attention

Sentence Representation

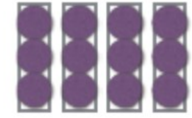
You can't cram the meaning of a whole %&!$# sentence into a single $&!#* vector!



But what if we could use multiple vectors, based on the length of the sentence.

this is an example →

this is an example →



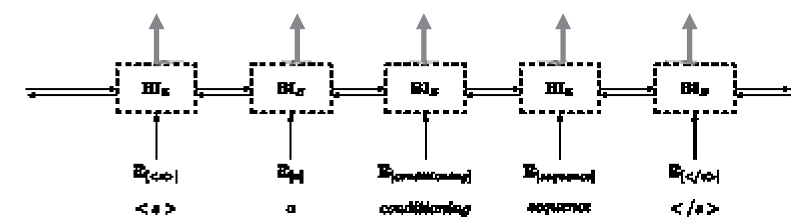
Intuition

- Encoding a single vector is too restrictive
  Instead of the encoder producing a single vector for the sentence,
  it will produce one vector **for each word**.
- But, we still need 1 vector. Multiple vectors → Single vector
  Sum/Avg operators are good. But give equal importance to each input
- We dynamically decide which input is more/less important for a task. Create a weighted sum to reflect this variation
- Attention:
  - query (q): decides how much importance to give each input
    attention weights (α_i): importance of each input i (normalized to 1)
    unnormalized attention weights ($\bar{\alpha}_i$): intermediate step to compute α_i
    attended summary: weighted average of input with α as weights

Encoder for One Vector/Word

$$\mathbf{c_{1:n}} = \textsc{Enc}(\mathbf{x_{1:n}}) = \text{biRNN}^{\star}(\mathbf{x_{1:n}})$$

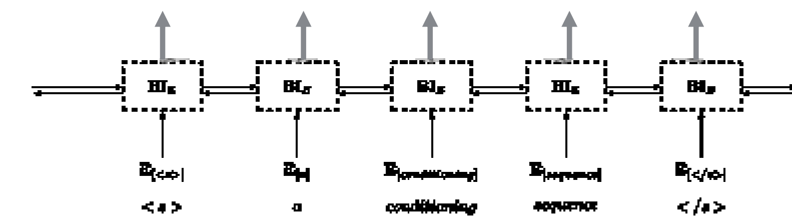
Encoder



Encoder for One Vector/Word

$$\mathbf{c_{1:n}} = \text{Enc}(\mathbf{x_{1:n}}) = \text{biRNN}^{\star}(\mathbf{x_{1:n}})$$

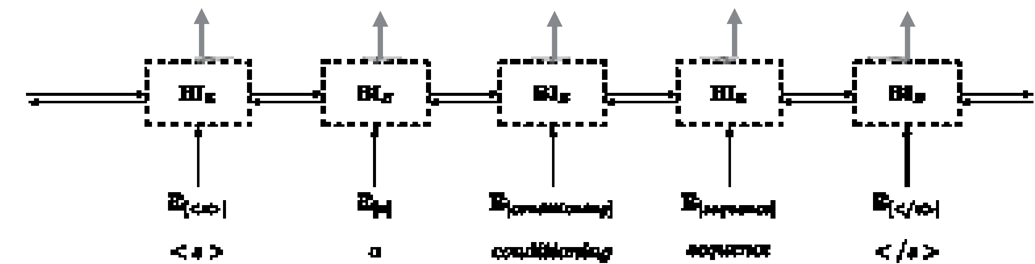
but how do we feed this sequence to the MLP?



Encoder

Encoder for One Vector/Word

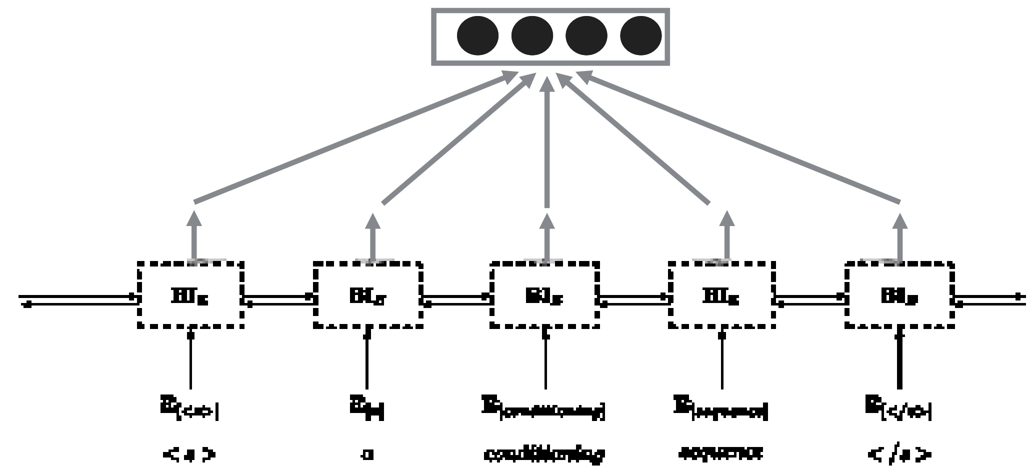
we can combine the different outputs
into a single vector (attended summary)



Encoder

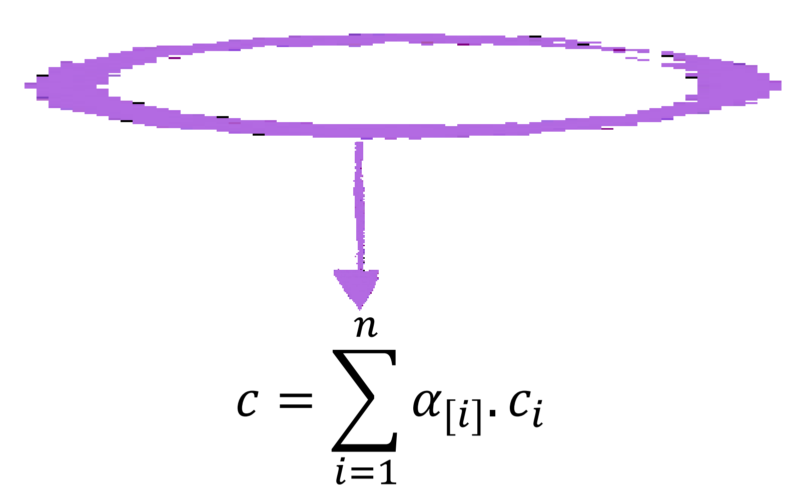
Multiple Vectors → Single Vector

we can combine the different outputs
into a single vector (attended summary)



Encoder

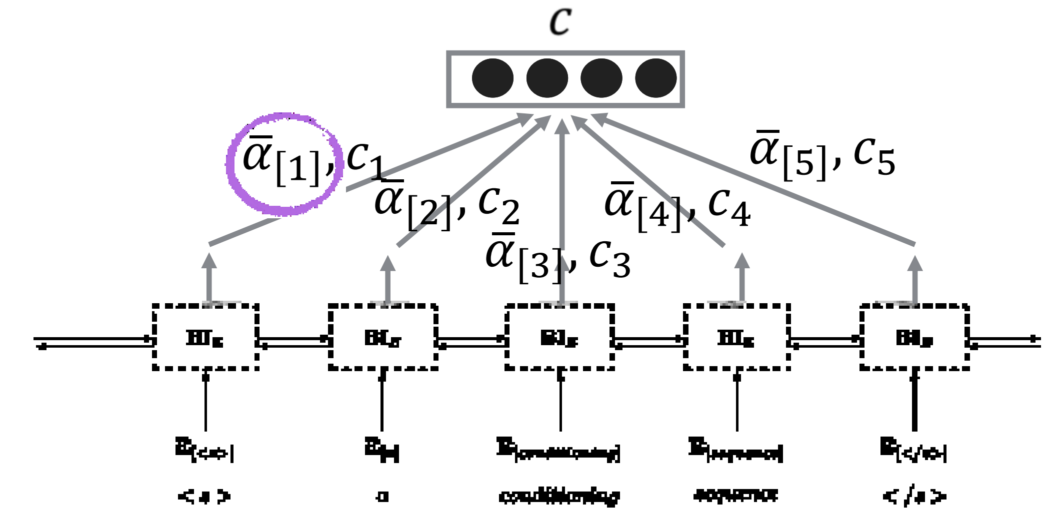
Attention



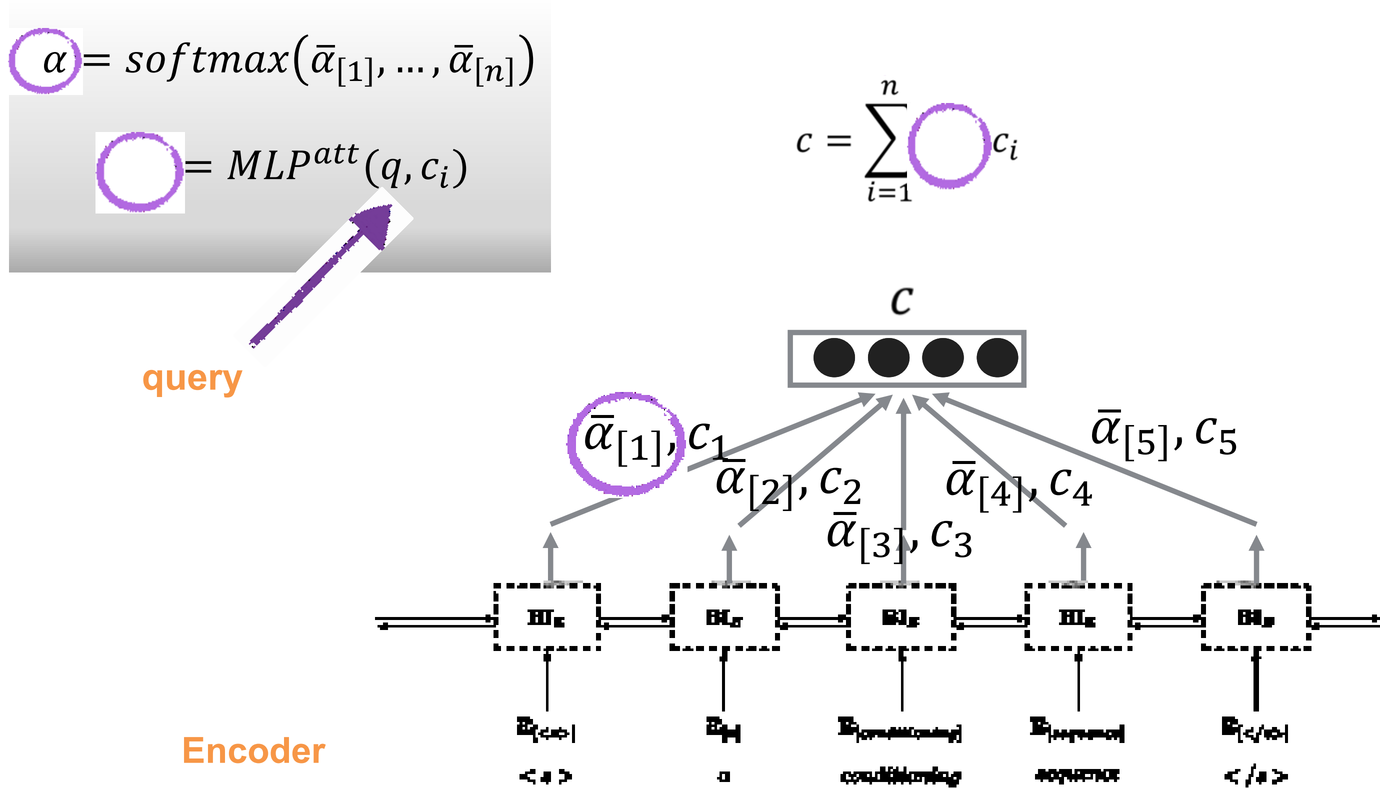
Attention

$\bigcirc = softmax(\bar{\alpha}_{[1]}, \ldots, \bar{\alpha}_{[n]})$

$$c = \sum_{i=1}^{n} \bigcirc c_i$$



Attention



Attention

$$y = softmax(MLP^{out}(c))$$

$$c = \sum_{i=1}^{n} \alpha_{[i]} . c_i$$

$$c_{1:n} = biLSTM_{enc}(x_{1:n})$$

$$\alpha = softmax\left(\bar{\alpha}_{[1]}, \dots, \bar{\alpha}_{[n]}\right)$$

$$\bar{\alpha}_{[i]} = MLP^{att}(q, c_i)$$

Two things missing. What are they?

Attention and/vs Interpretation

Dialogue Act

(A) Ground truth : Statement-opinion
Predict : Statement-opinion

And if you try to do anything , uh , like , uh , not identify yourself to the government , they know who you are .

(B) Ground truth : Statement-non-opinion
Predict : Statement-non-opinion

I , uh , ride bicycles , uh , fifteen , twenty miles , I don't know , maybe three times , maybe four times a week .

Key Term

(C) Ground truth : ios, facebook
5-best predict : ios, facebook-graph-api, facebook, objective-c, iphone

I have an iOS application that already using some methods of Facebook Graph API, but I need to implement sending private message to friend by Facebook from my application.
As I know, there is no way to sending private messages by Graph API, but it maybe possible by help Facebook Chat API.
I already read documentation but it do not help me.
If anybody has some kind of example or tutorial, how to implement Facebook Chat API in iOS application, how sending requests or something, it will be very helpfull. Thanks.

(D) Ground truth : python, numpy, matrix
5-best predict : python, numpy, arrays, matrix, indexing

I have a huge matrix that I saved with savetxt with numpy library. Now I want to read a single cell from that matrix, e.g.,

```
cell = getCell(i, j); print cell
return the value 10 for example.
```

I tried this:

```
x = np.loadtxt("fname.m", dtype="int", usecols=([i]))
cell = x[j]
```

but it is really slow because I loop over many index. Is there a way to do that without reading useless lines ?

Published in INTERSPEECH 2016

Neural Attention Models for Sequence Classification: Analysis and Application to Key Term Extraction and Dialogue Act Detection

S. Shen, Hung-yi Lee

Attention Functions

v: attended vec, q: query vec
MLP^{att}(q;v)=

- Additive Attention: $\mathbf{u}g(\mathbf{W^1 v} + \mathbf{W^2 q})$

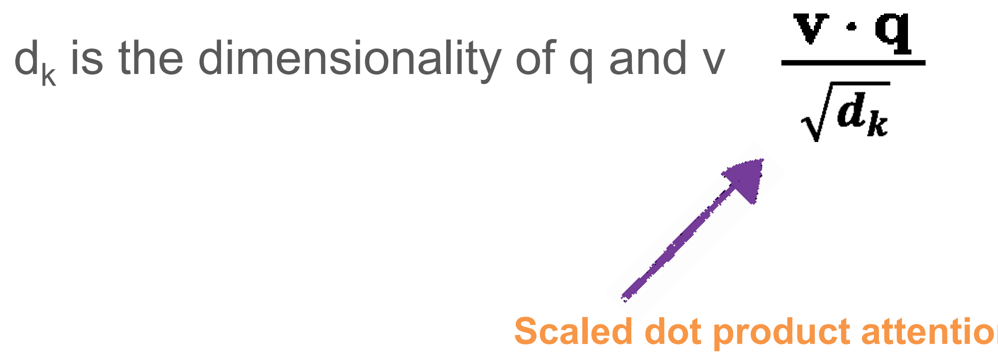
- Dot Product: $\mathbf{v} \cdot \mathbf{q}$

- Multiplicative Attention: $\mathbf{v}^\top \mathbf{W} \mathbf{q}$

Additive vs Multiplicative

While the two are similar in theoretical complexity, dot-product attention is much faster and more space-efficient in practice, since it can be implemented using highly optimized matrix multiplication code.

While for small values of d_k the two mechanisms perform similarly, additive attention outperforms dot product attention without scaling for larger values of d_k [3]. We suspect that for large values of d_k, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients [4]. To counteract this effect, we scale the dot products by $\frac{1}{\sqrt{d_k}}$.



Paper's Justification:

To illustrate why the dot products get large, assume that the components of q and k are independent random variables with mean 0 and variance
→ Then their dot product, $q \cdot k$ has mean 0 and variance d_k

Multi-head Key-Value Self Attention

Key-Value Attention

- Split an input vector x_i into two vectors $x_i=[k_i;v_i]$
  k: key vector
  v: value vector

- Use key vector for computing attention

  $MLP^{att}(q;x_i)=\frac{k_i.q}{\sqrt{d}}$ //scaled multiplicative

- Use value vector for computing attended summary

$$c = \sum_{i=1}^{n} \alpha_{[i]}.v_i$$

Key-Value Attention (alternative)

- an input vector x_i into two vectors
  k: key vector: $k_i = W^K x_i$
  v: value vector: $v_i = W^V x_i$

- Use key vector for computing attention

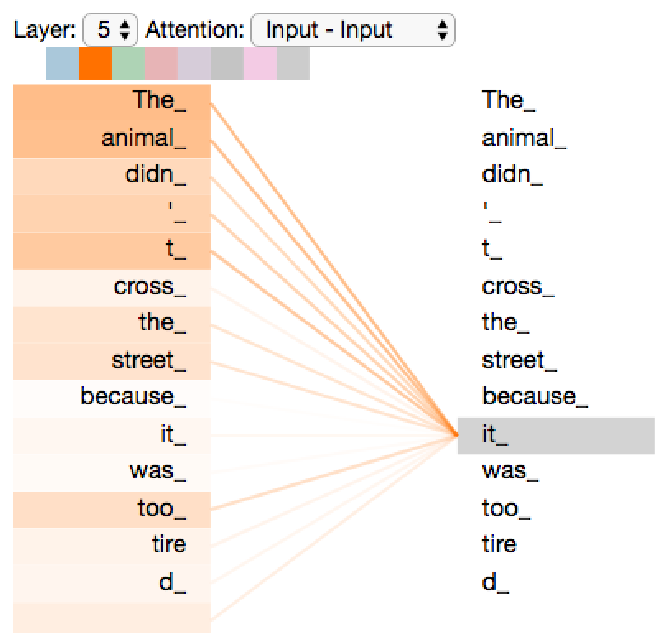
  $MLP^{att}(q; x_i) = \frac{k_i . q}{\sqrt{d}}$ //scaled multiplicative

- Use value vector for computing attended summary

$$c = \sum_{i=1}^{n} \alpha_{[i]} . v_i$$

Self-attention (single-head, high-level)

”The animal didn't cross the street because it was too tired”

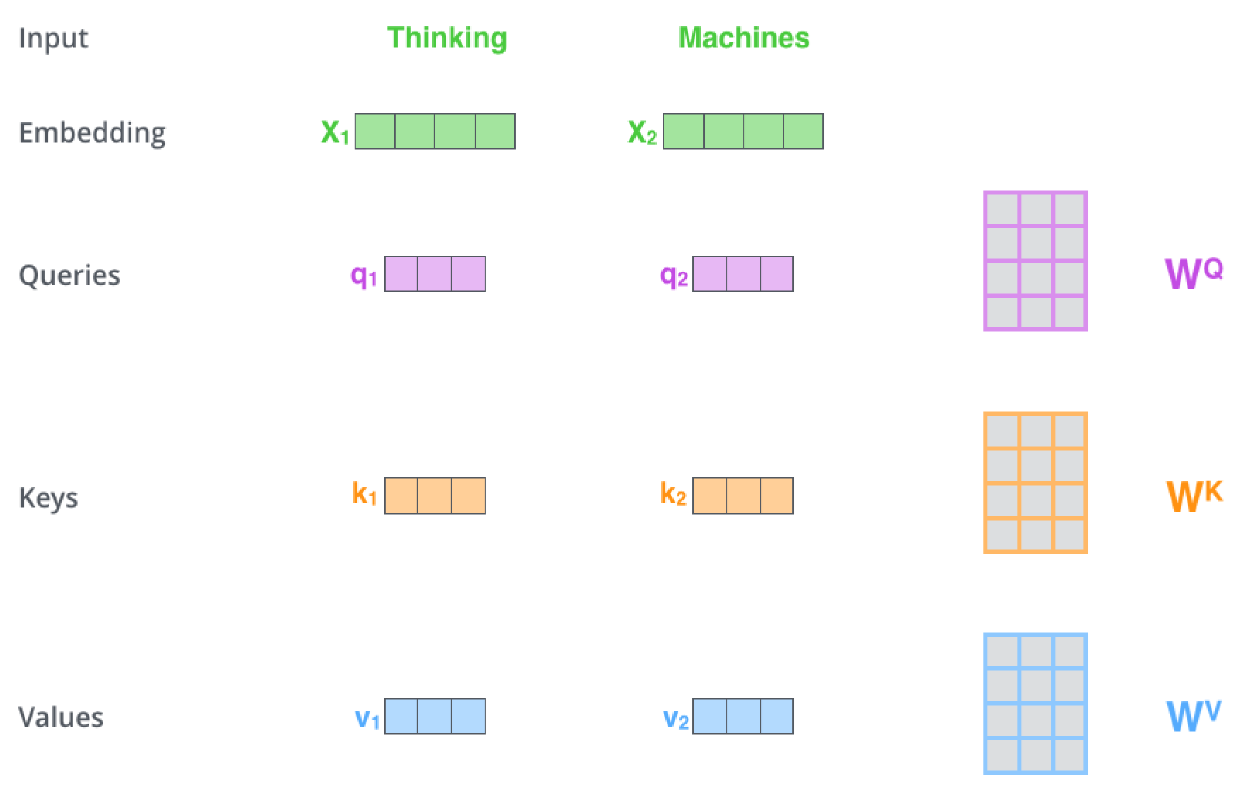


There is no external query q.
The input is also the query.
Many approaches:
https://ruder.io/deep-learning-nlp-best-practices/

Key-Value Single-Head Self Attention

- Project an input vector x_i into ( ) vectors
  k: key vector: $k_i = W^K x_i$
  v: value vector: $v_i = W^v x_i$
  q: ( ) vector: $q_i = W^Q x_i$
- Use key and query vectors for computing attention of ith word at word j
  $MLP^{att}(x_j; x_i) = \frac{k_i . q_j}{\sqrt{d}}$ //scaled multiplicative
- Use value vector for computing attended summary ( ) $= \sum_{i=1}^{n} \alpha_{[i]} . v_i$

Key-Value Single-Head Self Attention



Creation of query, key and value vectors by multiplying by weight matrices

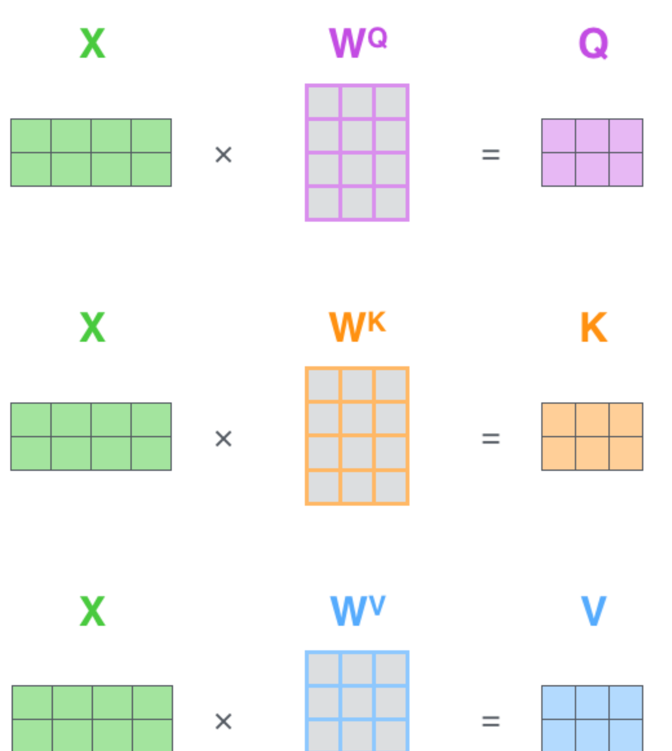
Separation of Value and Key

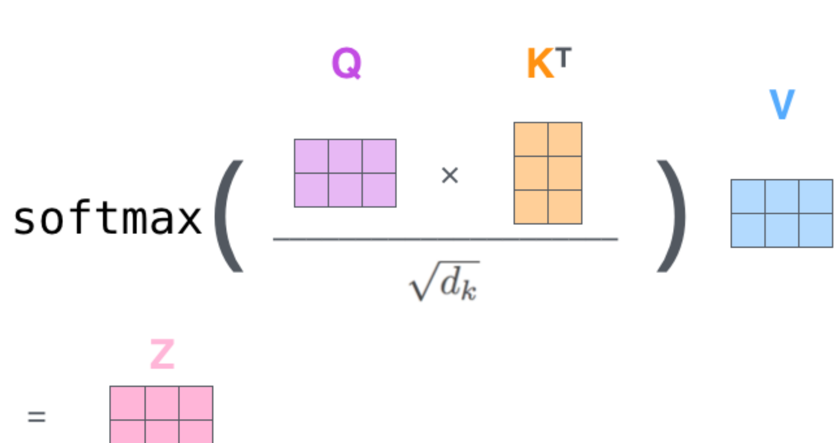
Matrix multiplications are quite efficient and can be done in aggregated manner

Key-Value Single-Head Self Attention

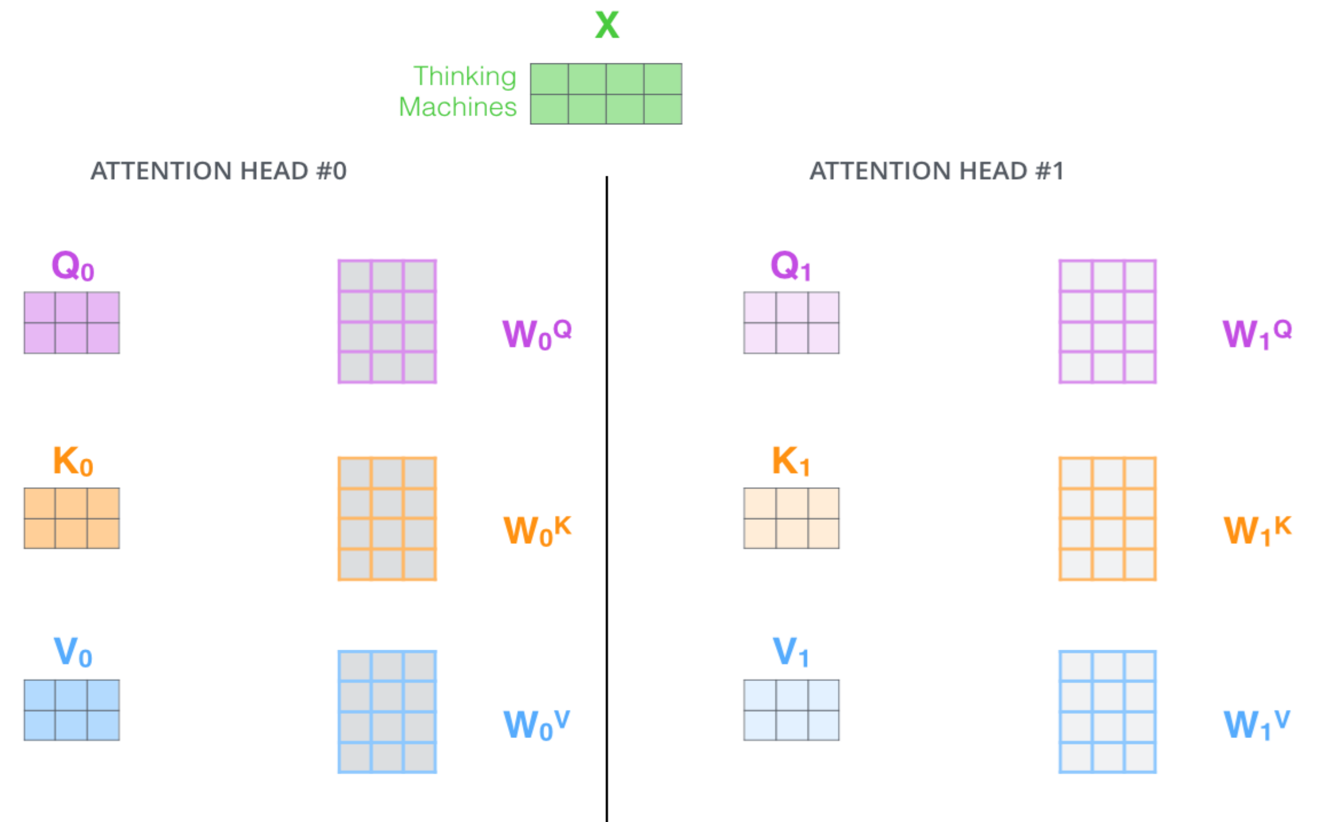
	Thinking	Machines
Input	Thinking	Machines
Embedding	x_1	x_2
Queries	q_1	q_2
Keys	k_1	k_2
Values	v_1	v_2
Score	$q_1 \bullet k_1 = 112$	$q_1 \bullet k_2 = 96$
Divide by 8 ($\sqrt{d_k}$)	14	12
Softmax	0.88	0.12
Softmax X Value	v_1	v_2
Sum	z_1	z_2

Key-Value Single-Head Self Attention

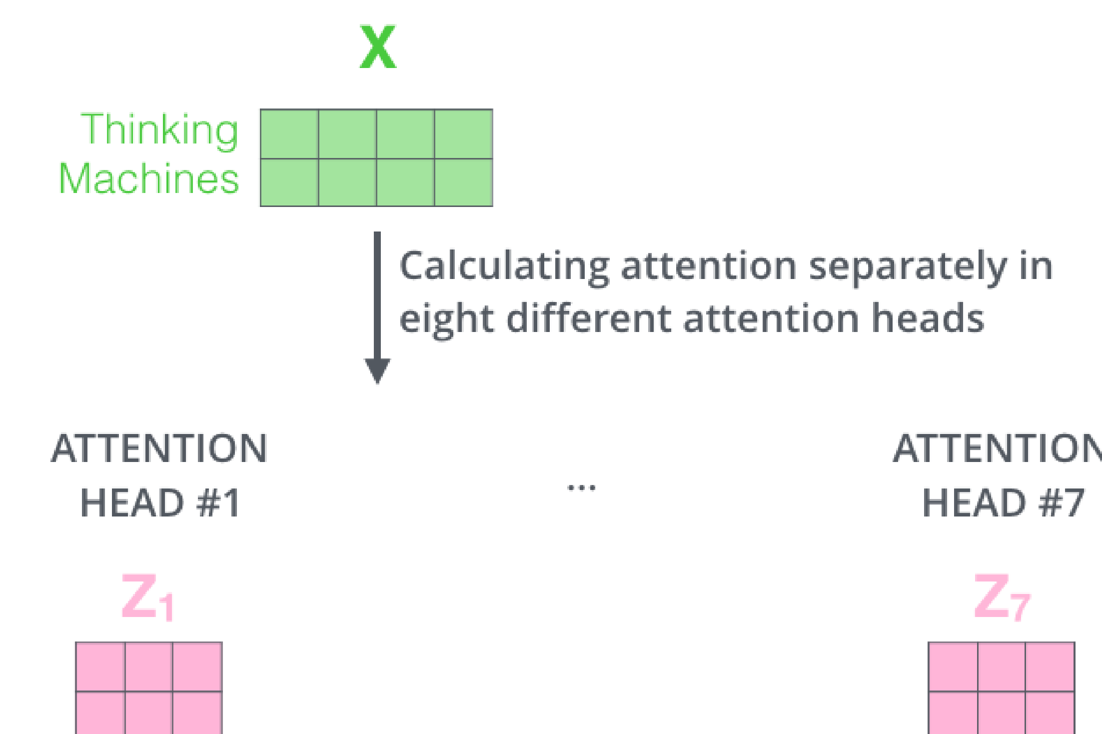




Key-Value Multi-Head Self Attention

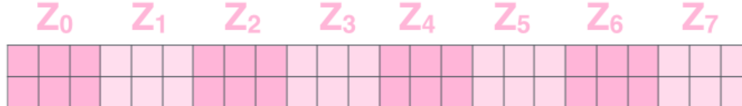


Multi-Head Attention



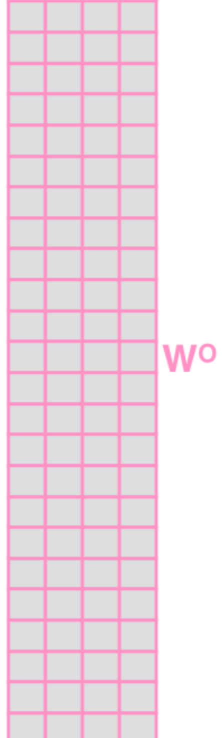
Multi-Head Attended Vector → Output

1) Concatenate all the attention heads



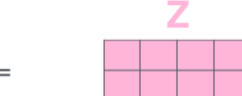
2) Multiply with a weight matrix W^O that was trained jointly with the model

X

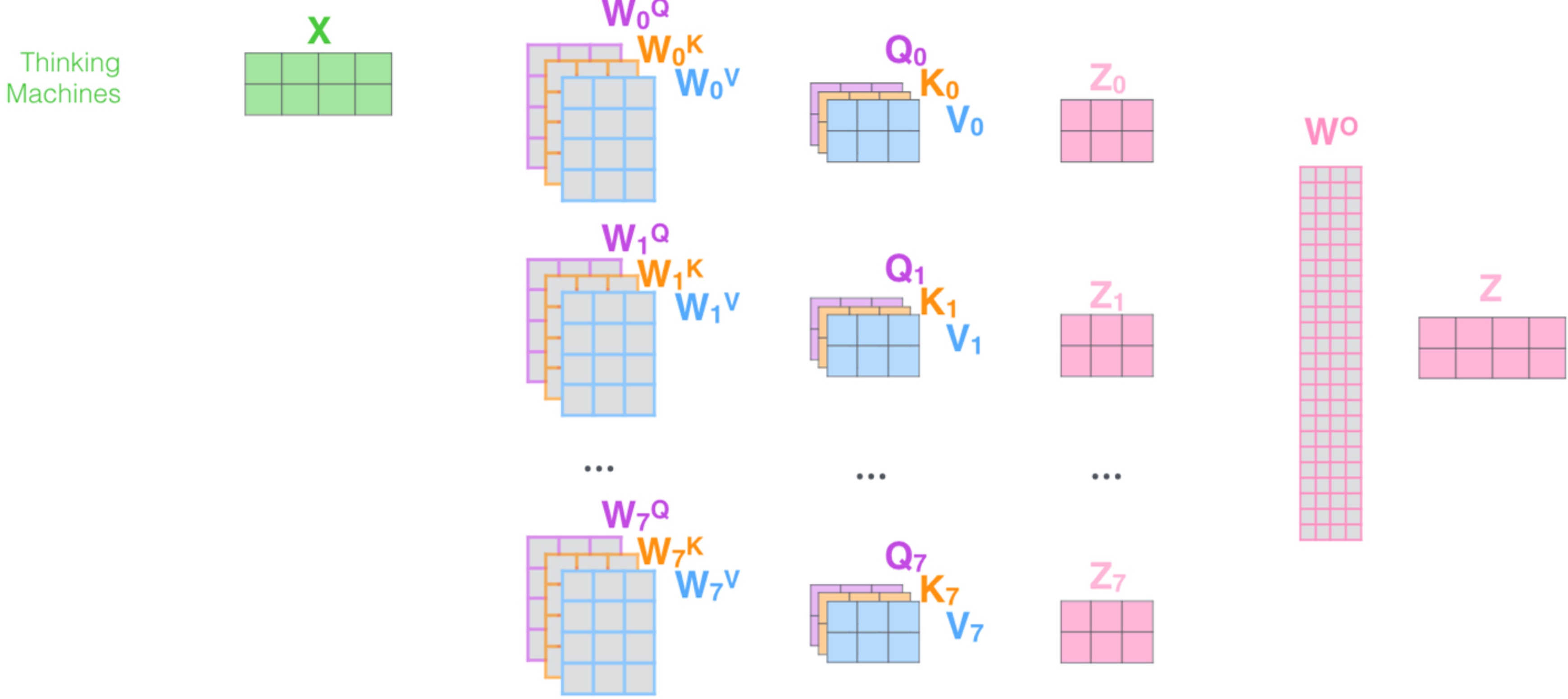


3) The result would be the Z matrix that captures information from all the attention heads. We can send this forward to the FFNN

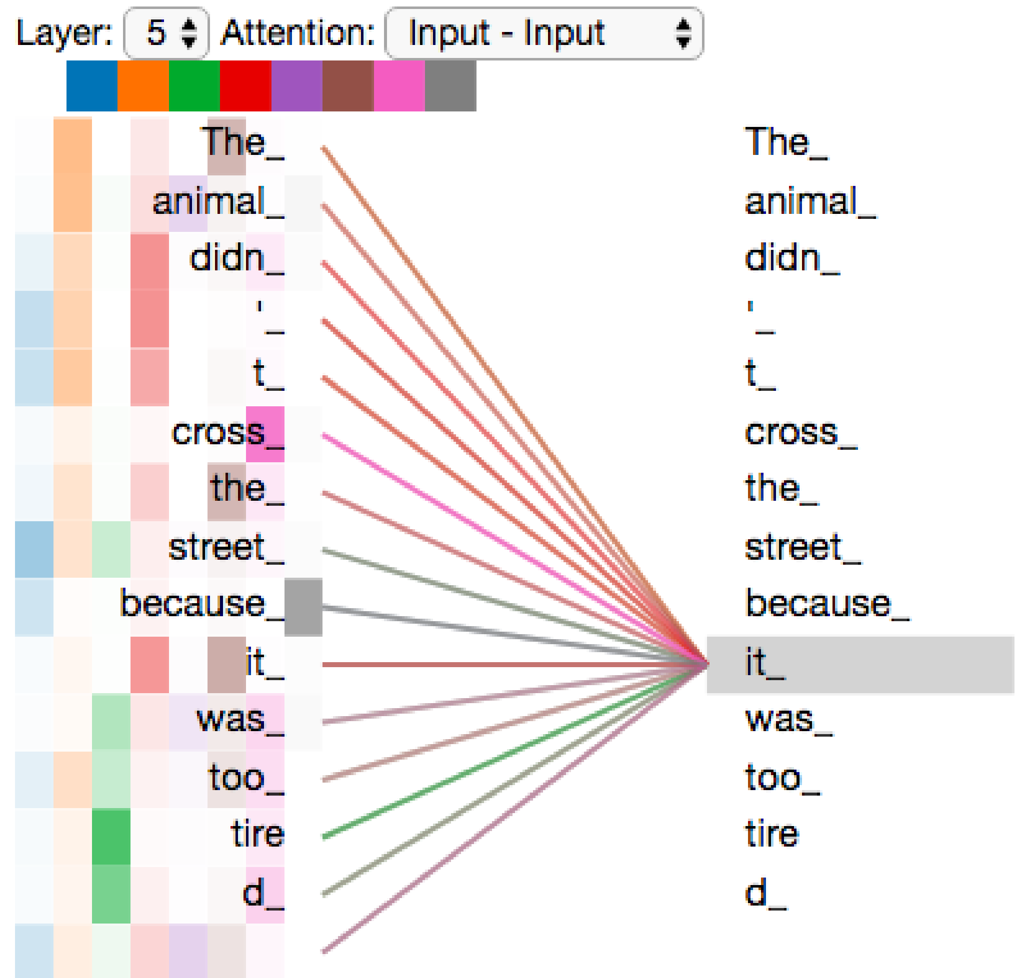
=



Key-Value Multi-Head Self Attention (summary)



Self attention visualisation (Interpretable?!)

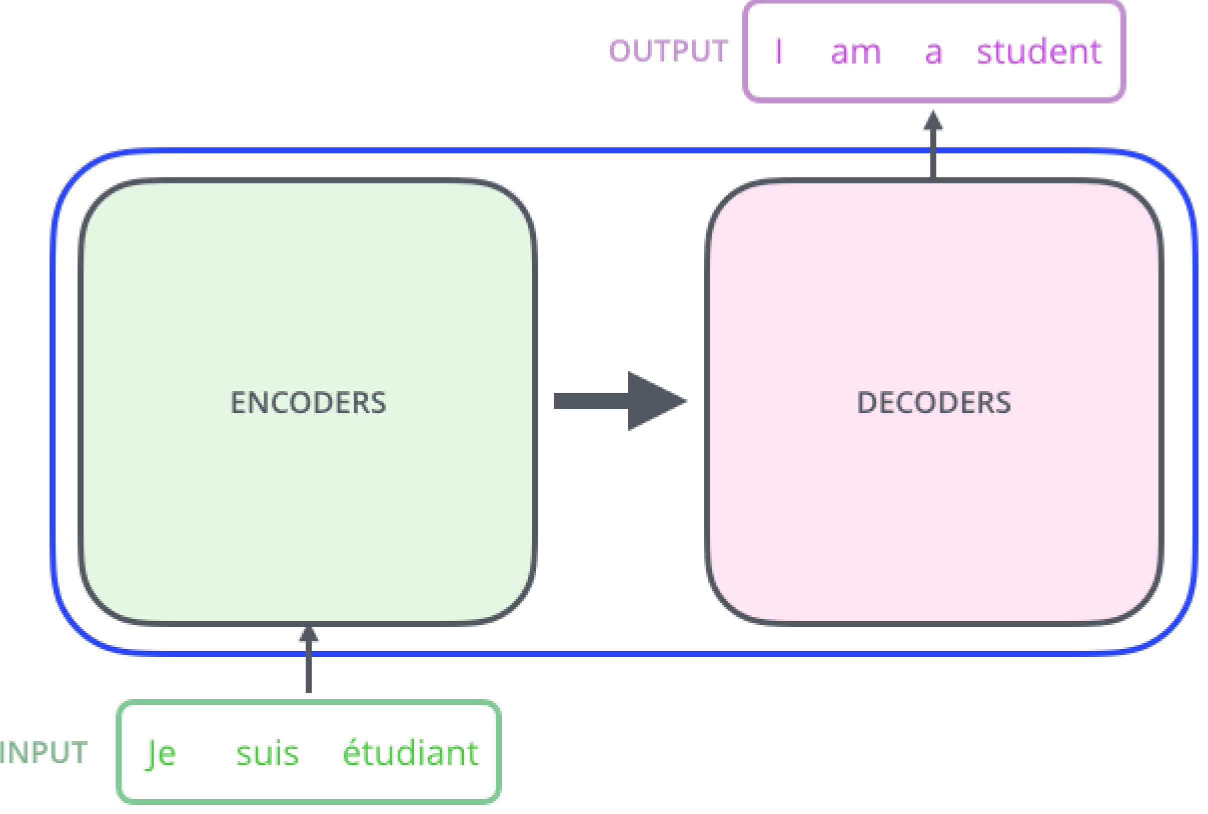


Transformer Architecture
(for text classification)

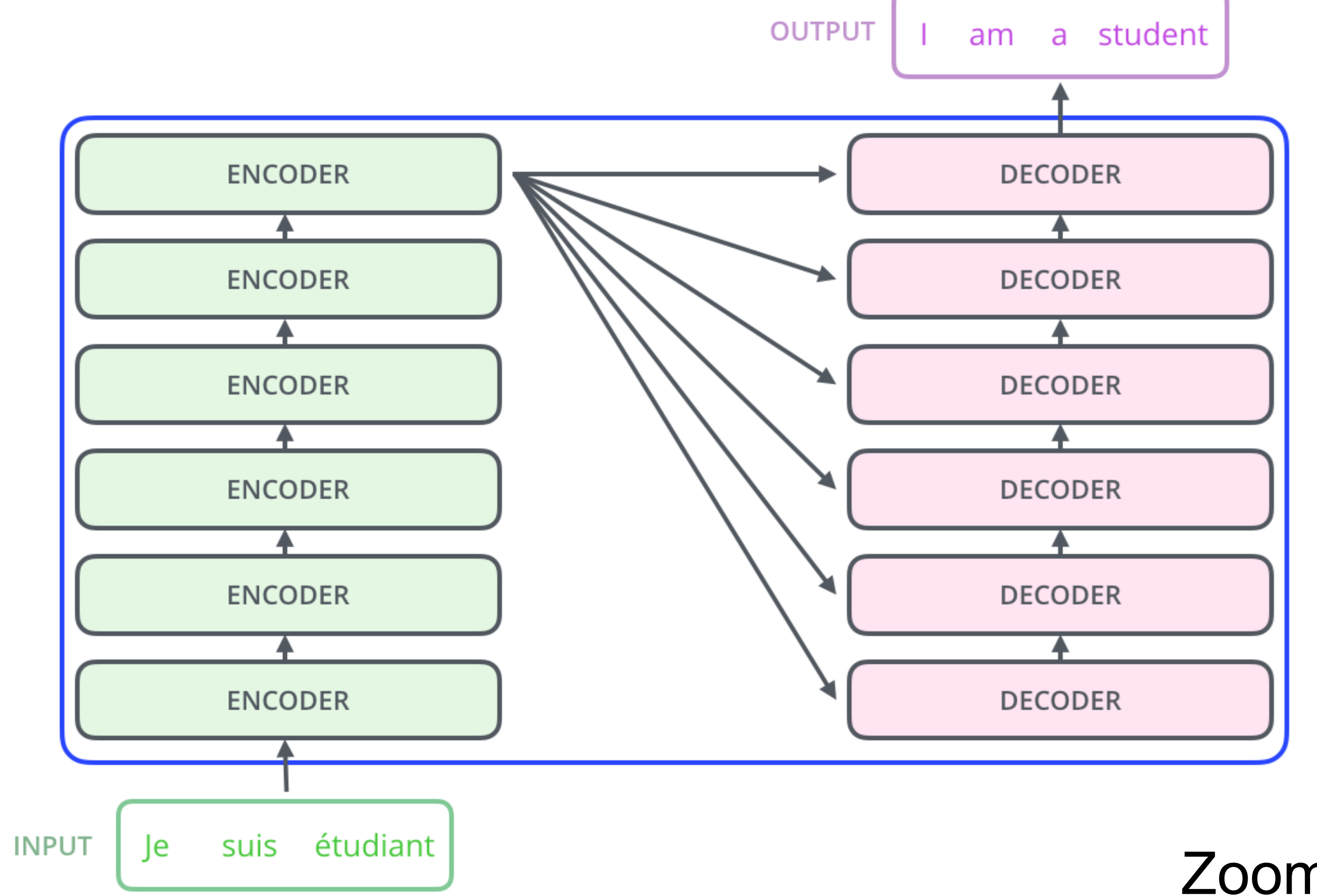
Motivation

- Recurrence is powerful
  - Issues with learnability: vanishing gradients
  - Issues with remembering long sentences
  - Issues with scalability: backpropagation time high due to sequentiality in sentence length
  - Issues with scalability: can’t be parallelized even at test time – O(sentence length)

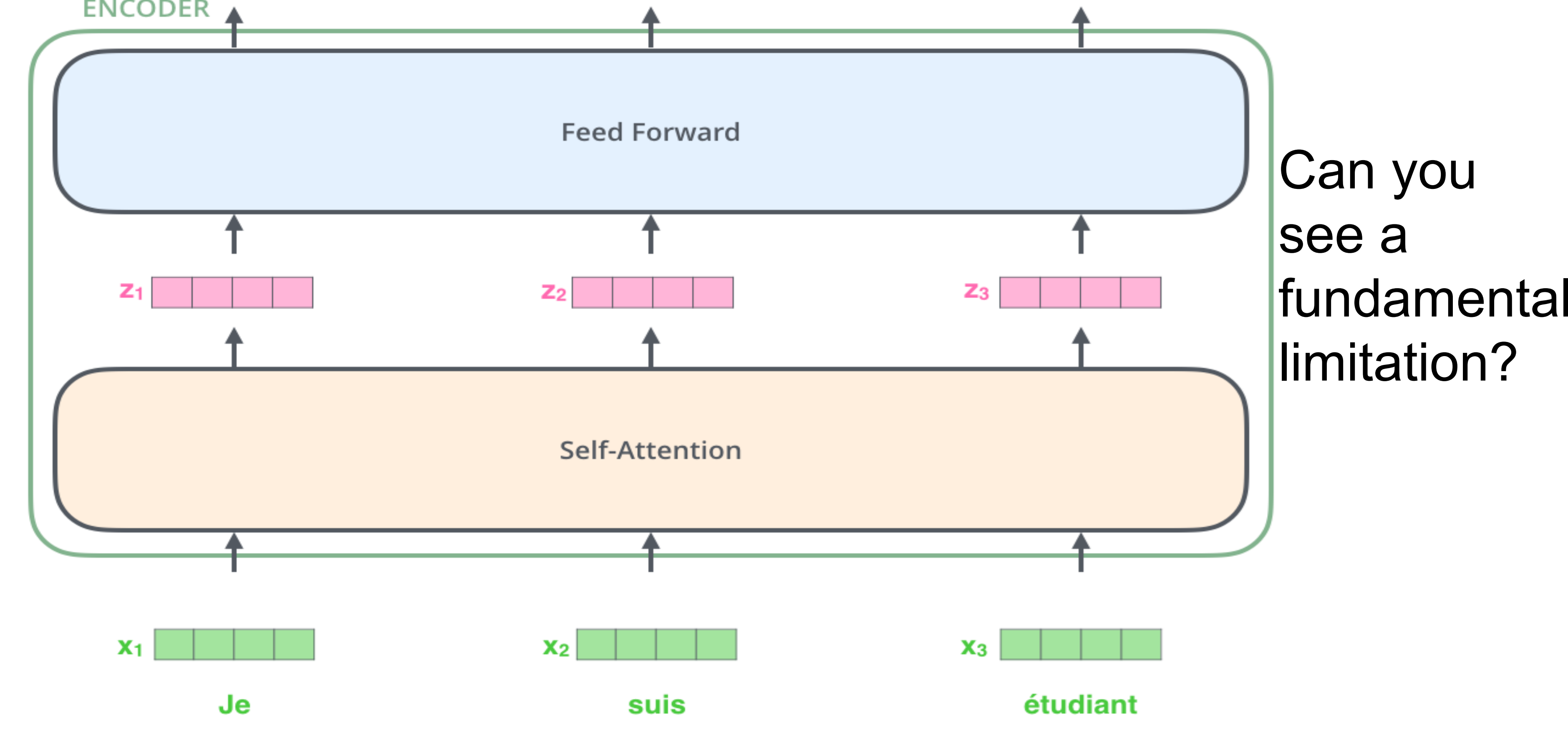
- Remove recurrence: only use attention
  “Attention is All You Need”



We focus only on encoder… (decoder is for sequence generation – will study later)



Zooming in...

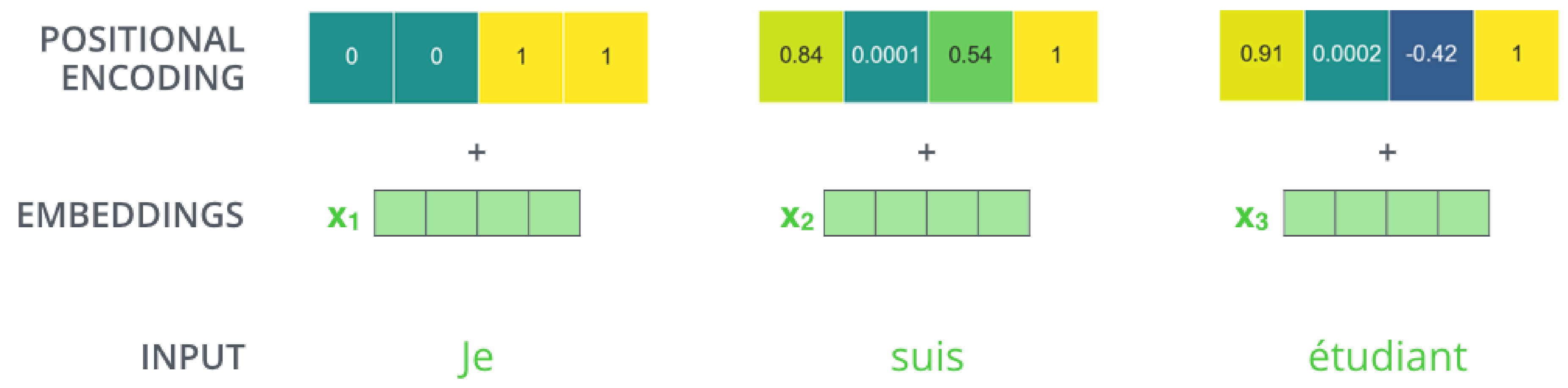


Can you see a fundamental limitation?

Encoders have same architecture but different weights…

Zooming in further...

A note on Positional embeddings



Positional embeddings can be extended to any sentence length but if any test input is longer than all training inputs then we will face issues.

Solution: use a functional form (as in Transformer paper – sinuisoidal encoding)

ENCODER #1

Add & Normalize

Feed Forward

Feed Forward

Add & Normalize

Self-Attention

POSITIONAL
ENCODING

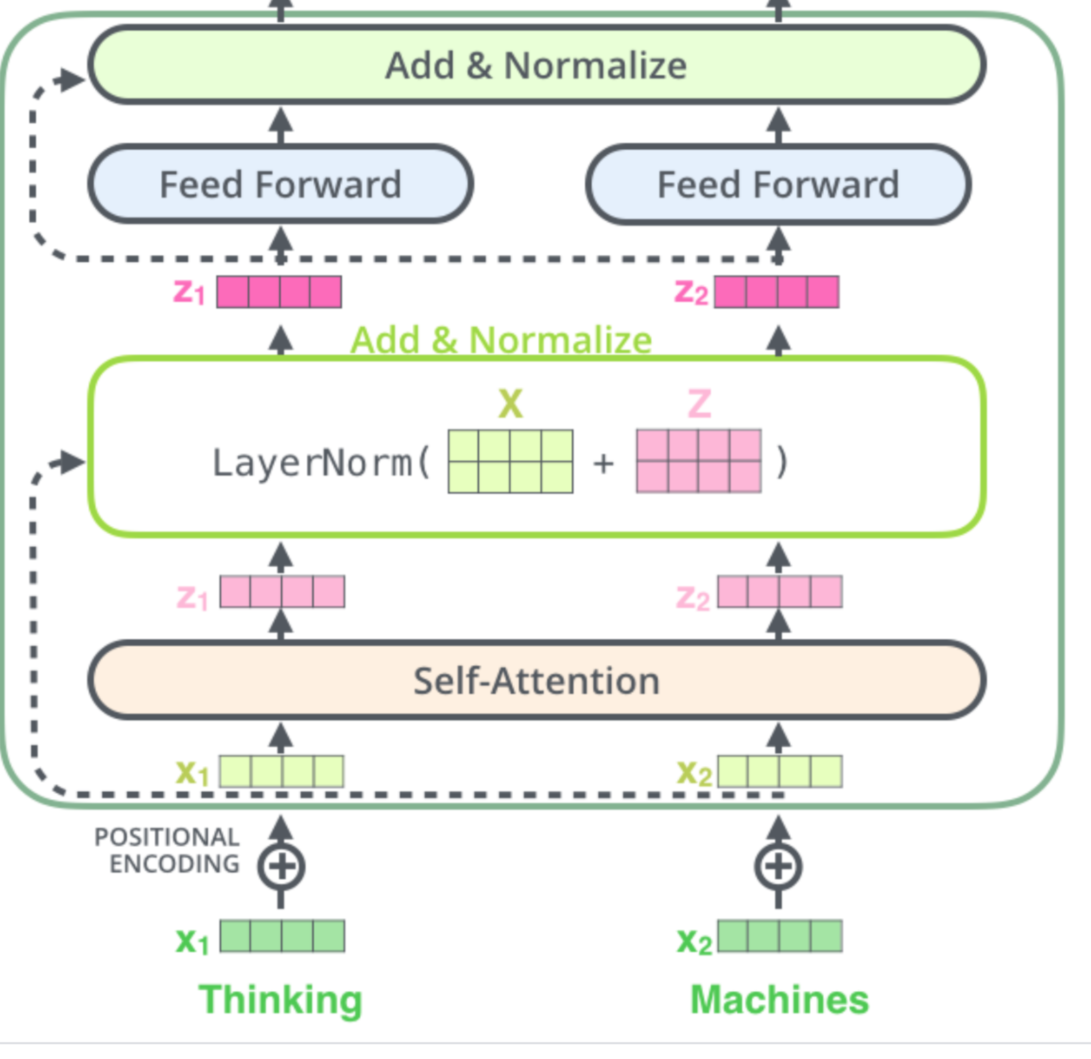
x_1

x_2

Thinking

Machines

Adding residual connections...

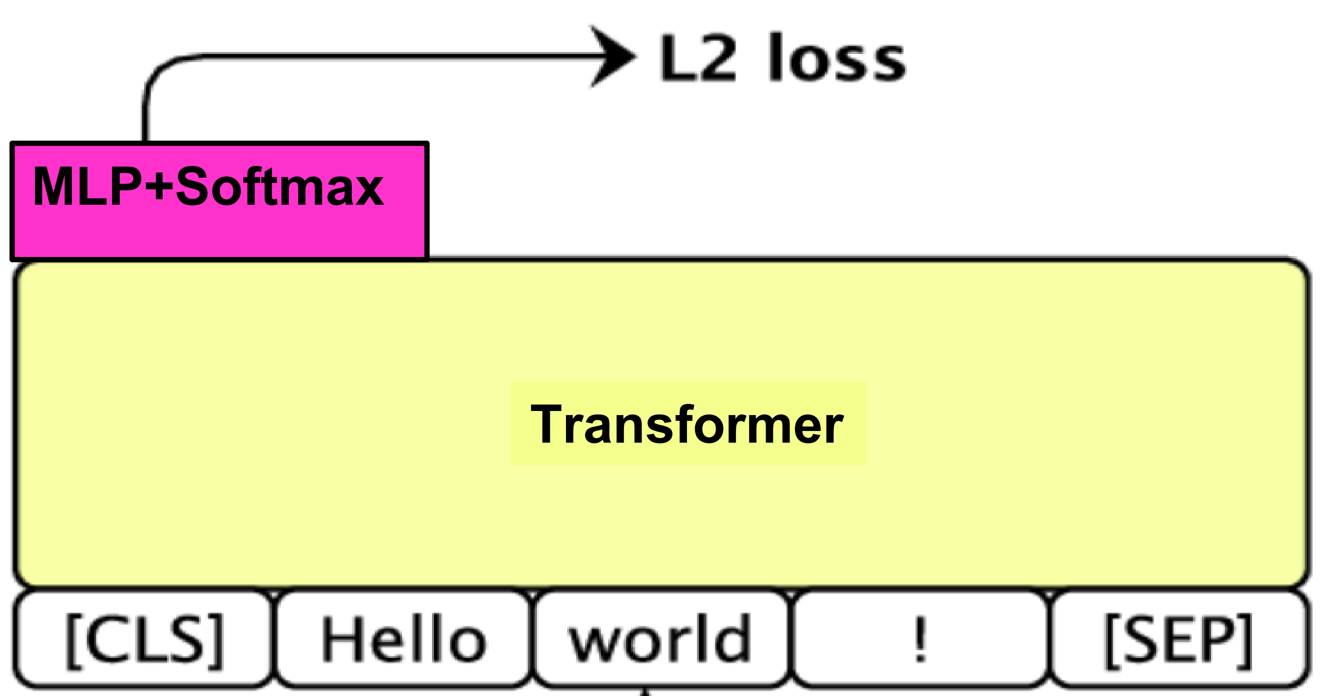


The residual connections help the network train, by allowing gradients to flow through the networks directly.

The layer normalizations stabilize the network -- substantially reducing the training time necessary.

The pointwise feedforward layer is used to project the attention outputs potentially giving it a richer representation.

Use of [CLS] for Text Classification



Pros

- Current state-of-the-art in machine translation and text simplification.
- Enables deep architectures
- Intuition of model well explained
- Easier learning of long-range dependencies
- Can be efficiently parallelized
- Gradients don’t suffer from vanishing gradients

Cons

Huge number of parameters so-

- Very data hungry
- Takes a long time to train
- No study of memory utilisation

Other issues

- Keeping sentence length limited
- How to ensure multi-head attention has diverse perspectives.

Reformer & Longformer
The Efficient Transformers

Kitaev et. al. (January 2020, ICLR)
Beltagy et. al. (April 2020, Arxiv)

Concerns about the transformer

“Transformer models are also used on increasingly long sequences. Up to 11 thousand tokens of text in a single example were processed in (Liu et al., 2018) … These large-scale long-sequence models yield great results but **strain resources to the point where some argue that this trend is breaking NLP research**”

“Many large Transformer models **can only realistically be trained in large industrial research laboratories** and such models trained with model parallelism cannot even be fine-tuned on a single GPU as their **memory requirements demand a multi-accelerator hardware setup**"

Memory requirement estimate (per layer)

Largest transformer layer ever: 0.5B parameters = 2GB

Activations for 64K tokens for embedding size 1K and batch size 8

= 64K * 1K * 8 = 2GB

Training data used in BERT = 17GB

Why can’t we fit everything in one GPU? 32GB GPUs are common today.

Caveats follow ->>>>>

Caveats

1. There are N layers in a transformer, whose activations need to be stored for backpropagation
2. We have been ignoring the feed-forward networks uptil now, whose depth even exceeds the attention mechanism so contributes to significant fraction of memory use.
3. Dot product attention is $O(L^2)$ in space complexity where L is length of text input.

Solutions

1. Reversible layers, first introduced in Gomez et al. (2017), enable storing only a single copy of activations in the whole model, so the N factor disappears.
2. Splitting activations inside feed-forward layers and processing them in chunks saves memory inside feed-forward layers.
3. Approximate attention computation based on locality-sensitive hashing replaces the $O(L^2)$ factor in attention layers with $O(L \log L)$ and so allows operating on long sequences.

Locality Sensitive Hashing

Hypothesis: Attending on all vectors is approximately same as attending to the 32/64 closest vectors to query in key projection space.

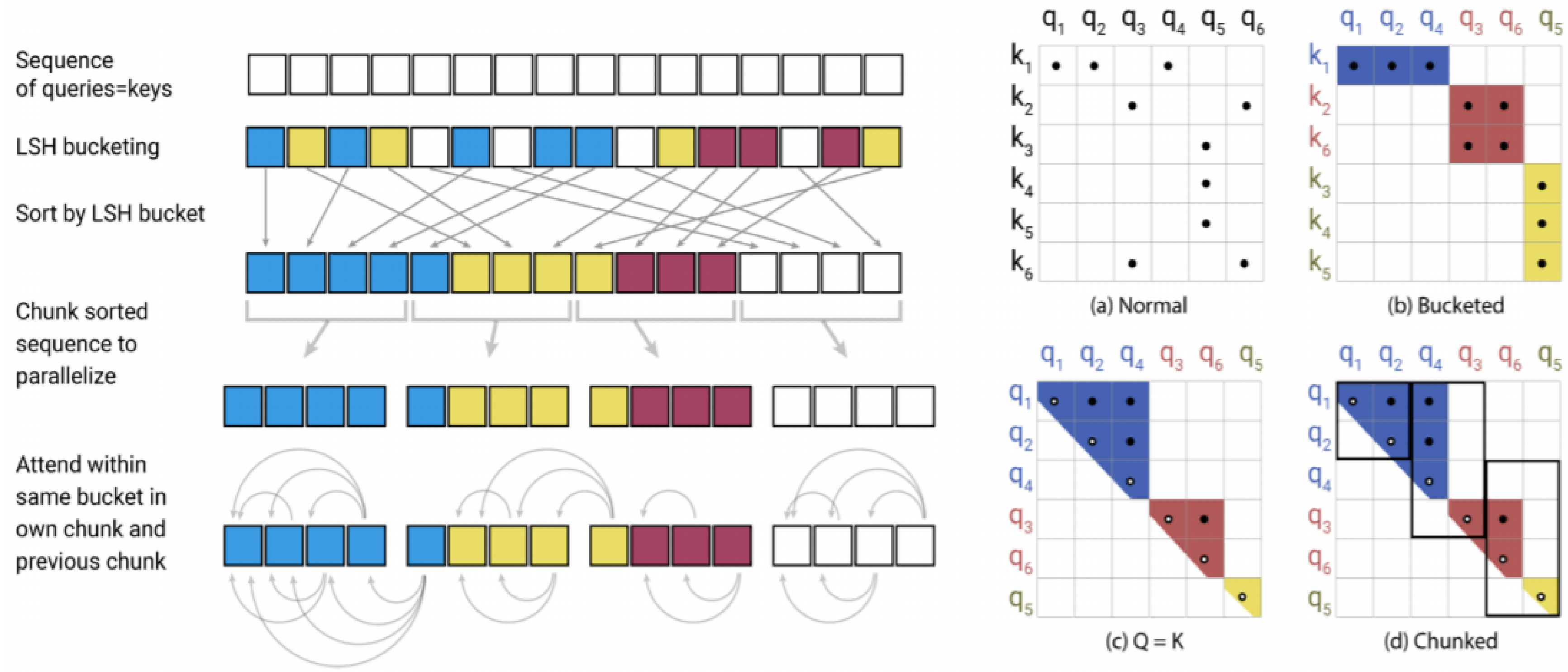
To find such vectors easily we require:

- Key and Query to be in same space
- Locality sensitive hashing i.e. if distance between key and query is less then distance between their hash values is less.

Locality sensitive hashing scheme taken from Andoni et al., 2015

For simplicity, a bucketing scheme chosen: attend on everything in your bucket

Locality sensitive hashing



Solutions

1. Reversible layers, first introduced in Gomez et al. (2017), enable storing only a single copy of activations in the whole model, so the N factor disappears.
2. Splitting activations inside feed-forward layers and processing them in chunks saves memory inside feed-forward layers.
3. Approximate attention computation based on locality-sensitive hashing replaces the $O(L^2)$ factor in attention layers with $O(L \log L)$ and so allows operating on long sequences.

RevNets

Reversible residual layers were introduced in Gomez et. al. 2017

Idea: Activations of previous layer can be recovered from activations of subsequent layers, using model parameters.

Normal residual layer: y = x + F(x)

Reversible laver:

$$y_1 = x_1 + F(x_2) \qquad y_2 = x_2 + G(y_1)$$

$$x_2 = y_2 - G(y_1) \qquad x_1 = y_1 - F(x_2)$$

So, for transformer:

$$Y_1 = X_1 + \text{Attention}(X_2) \qquad Y_2 = X_2 + \text{FeedForward}(Y_1)$$

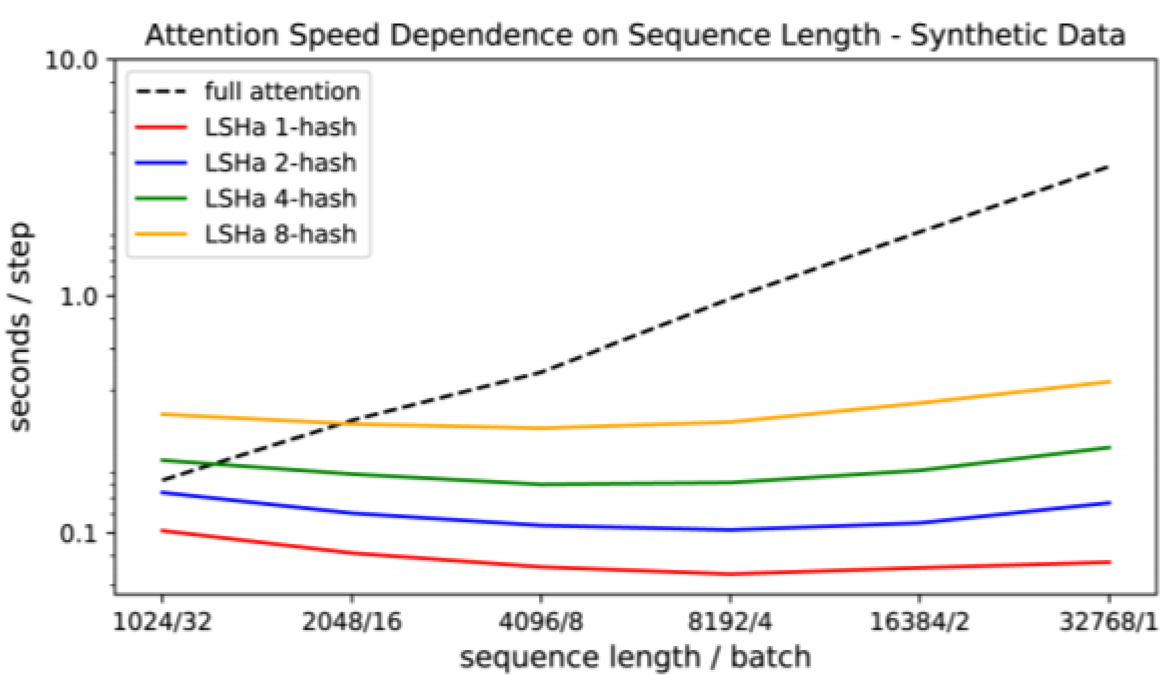
Chunking

$$Y_2 = \left[Y_2^{(1)}; \ldots; Y_2^{(c)}\right] = \left[X_2^{(1)} + \text{FeedForward}(Y_1^{(1)}); \ldots; X_2^{(c)} + \text{FeedForward}(Y_1^{(c)})\right]$$

Operations done a chunk at a time:

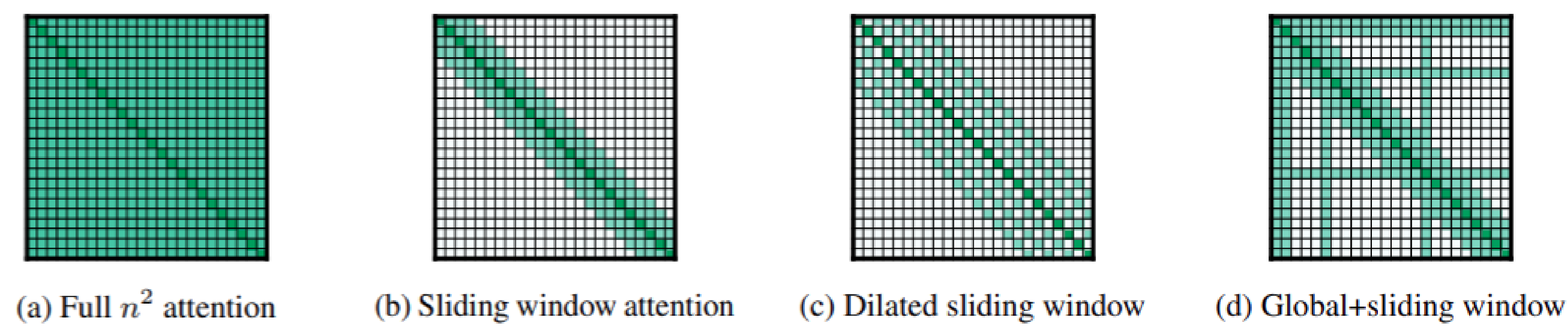
- Forward pass of Feed-forward network
- Reversing the activations during backpropagation
- For large vocabularies, chunk the log probabilities

Experiments



LSH Attention on Imagenet64
full attention
2 hashes
4 hashes
8 hashes
16 hashes
bpd
4.3
4.2
4.0
3.9
3.7
3.6
steps
20K
40K
60K
80K
100K
120K
140K

Longformer



(a) Full n^2 attention
(b) Sliding window attention
(c) Dilated sliding window
(d) Global+sliding window

Figure 2: Comparing the full self-attention pattern and the configuration of attention patterns in our Longformer.