# Memorizing Transformers

Yuhuai Wu, Markus N. Rabe, DeLesley Hutchins, Christian Szegedy
Google Research

Presenter: Arman Cohan

# Problem

- How can we make use of longer context in transformers in an efficient way?

# Problem

- How can we make use of longer context in transformers in an efficient way?
  - Most techniques address smaller model sizes
  - At larger model sizes the Feed Forward layer is the bottleneck, not attention

# Efficient transformers

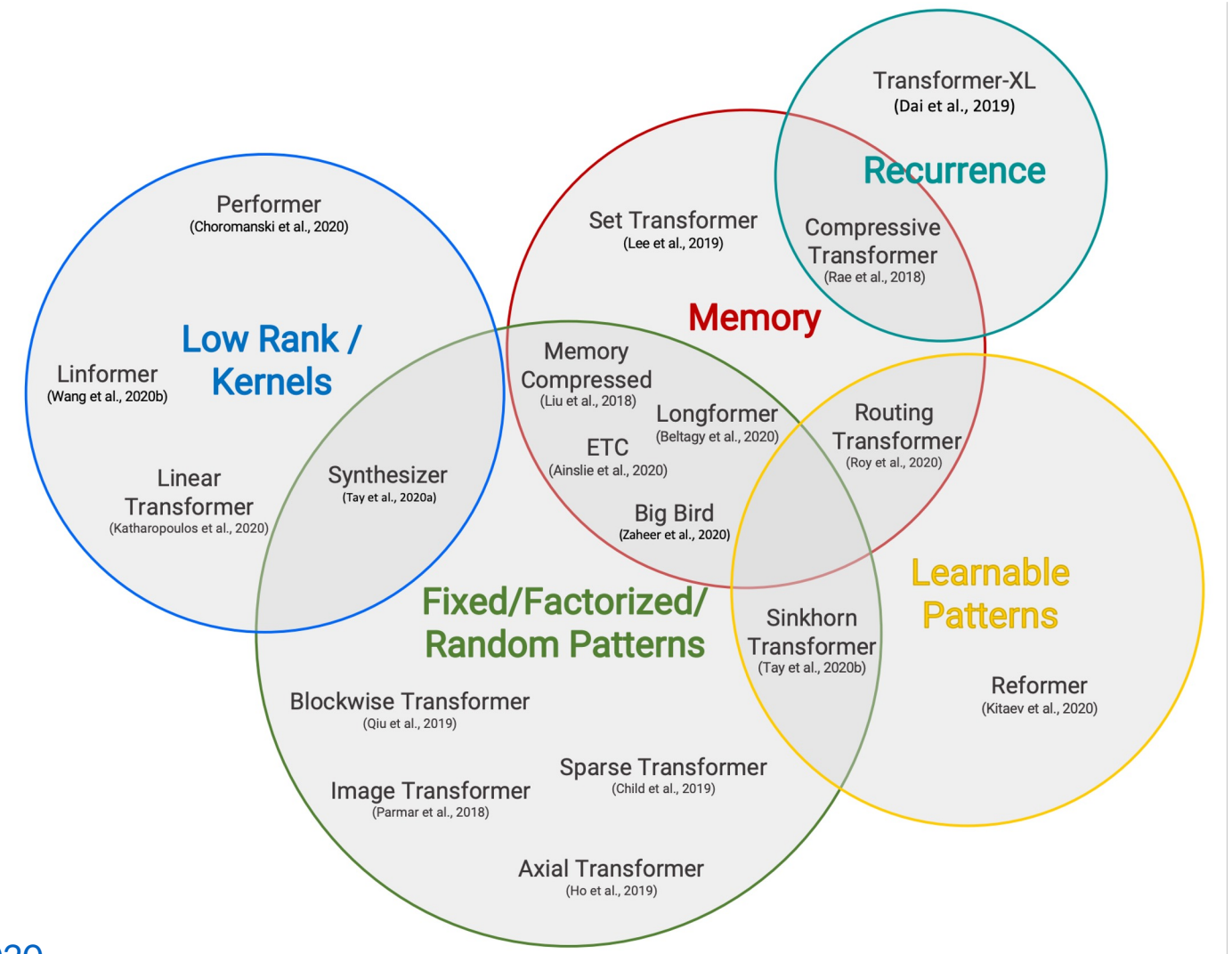
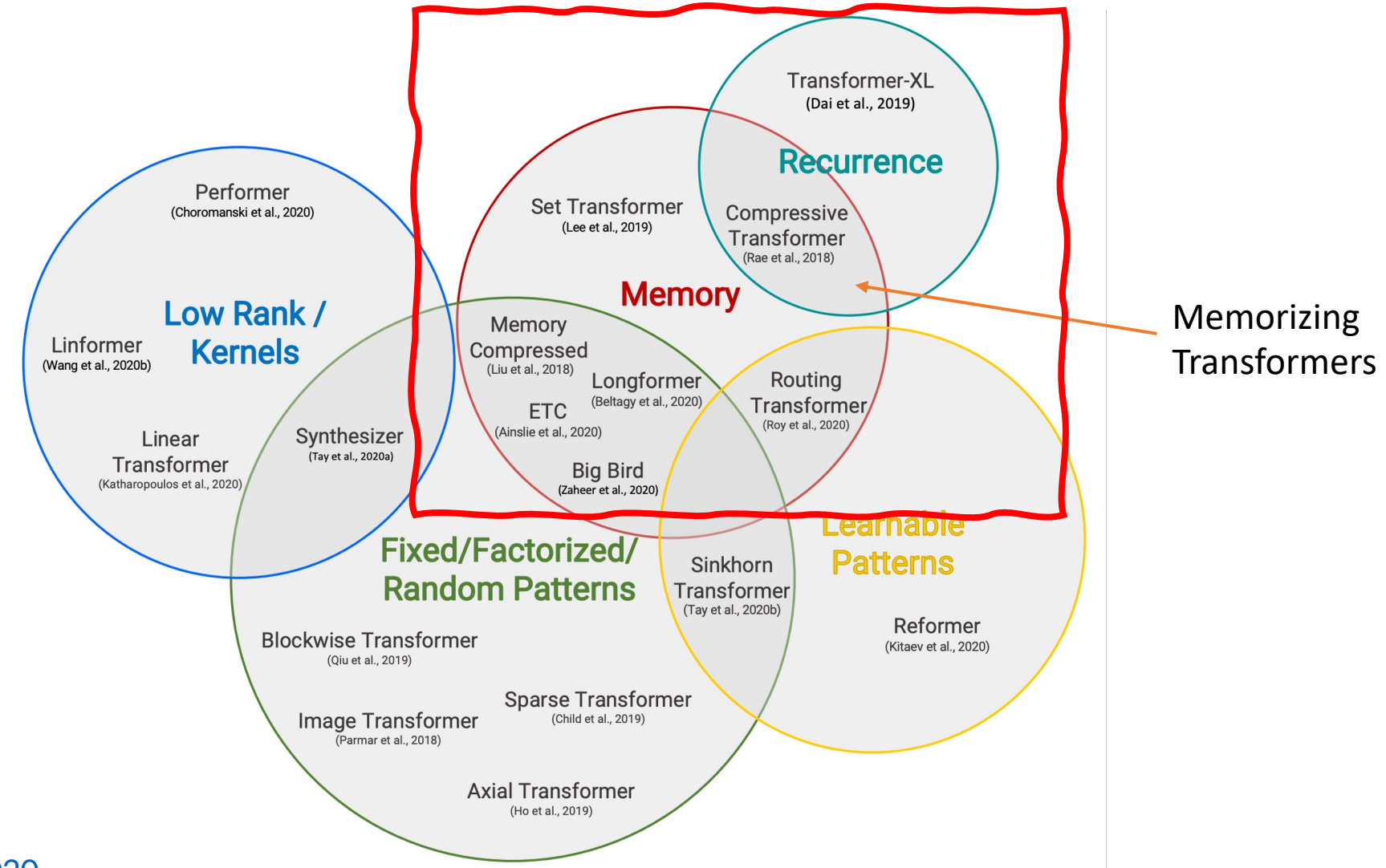



Fig reference: [Tay et al., 2020]

# Efficient transformers




Memorizing Transformers

Fig reference: [Tay et al., 2020]

# Prior work: Recurrence and compressing memory

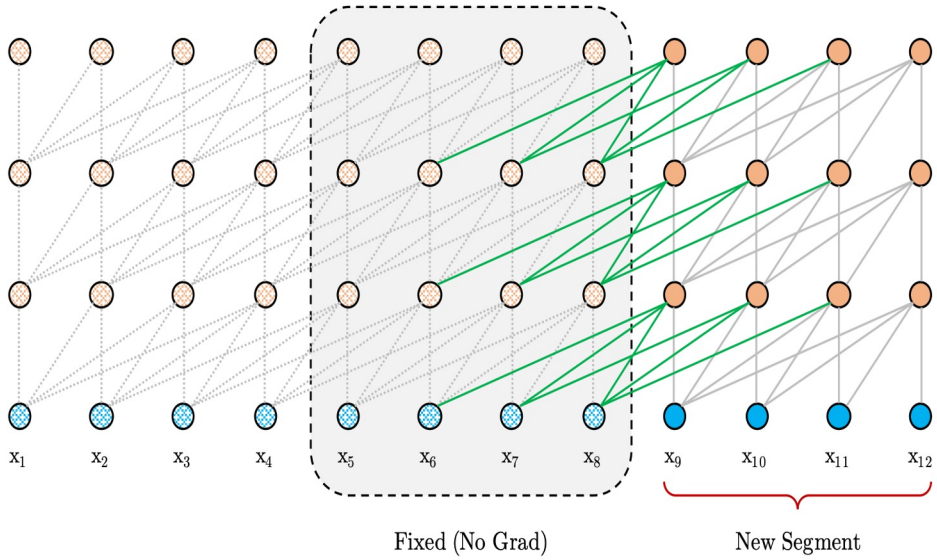
## Transformer-XL (Dai et al., 2019)

Segment-level recurrence

Representations from previous steps are cached and reused

No gradient updates on previous segments

Use of relative position embeddings in attention computation

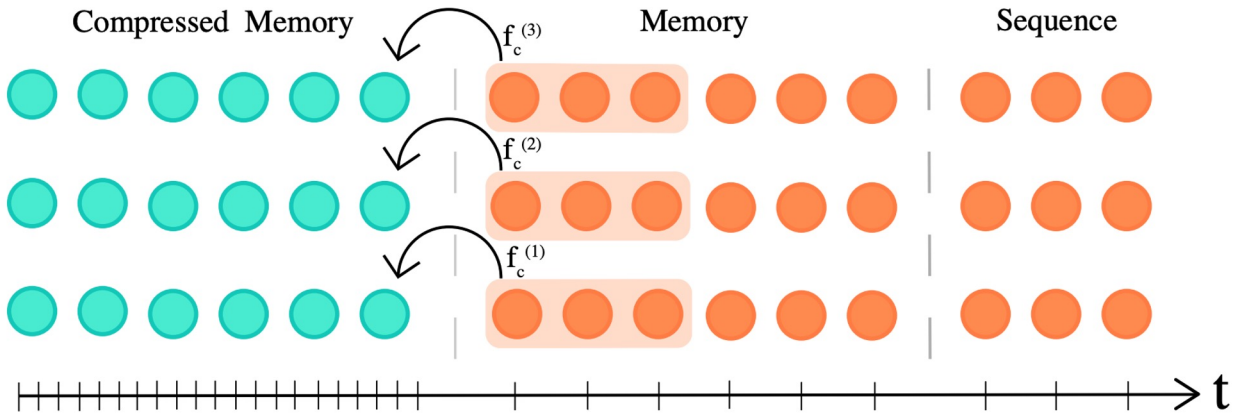


# Prior work: Recurrence and compressing memory

## Compressive Tranformer (Rae et al., 2019)

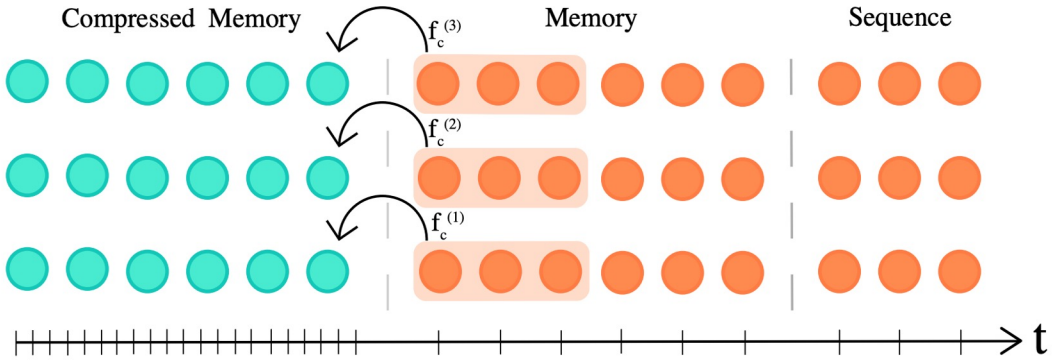
Similar to Transformer-XL

- Instead of discarding old past activations it compresses them
- Keeping two types of memory
  - A primary memory
  - An additional compressed memory



# Prior work: Recurrence and compressing memory

## Compressive Tranformer (Rae et al., 2019)




Approaches to compress the memory:

1. mean/max pooling
2. 1D convolutions
3. Dilated convolutions
4. Most used (e.g., sorted by usage of attention).

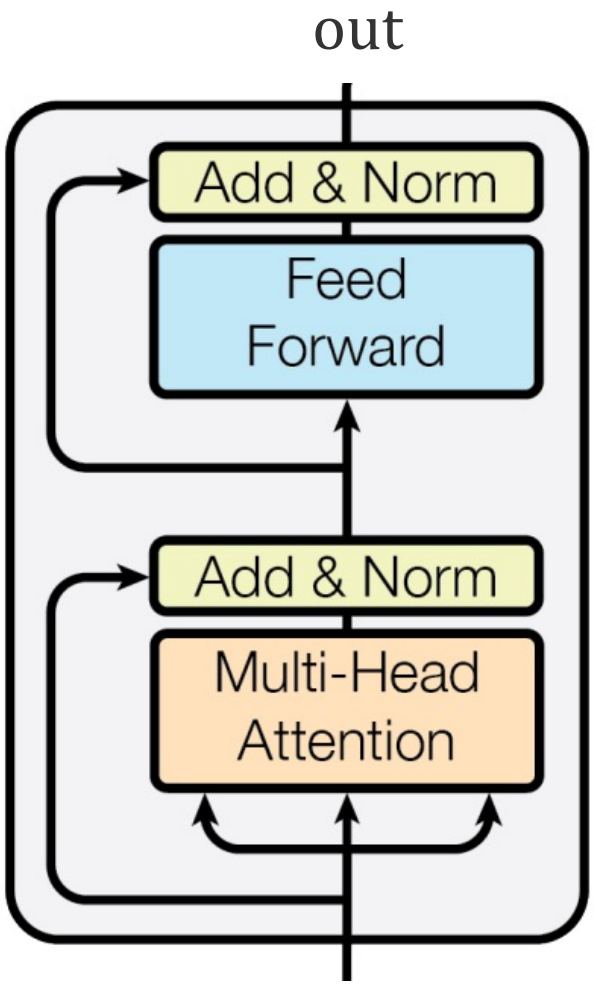
An additional auto-encoding loss

learns to reconstruct the original memory from its compressed version

# Memorizing Transformers - Overview

- Use approximate k-nearest-neighbor (kNN) lookup
- kNN lookup does not do averaging or summarization of tokens at long distances, but retrieves exact values even from the distant context.
- Gradients are not backpropagated into the external memory, which is critical to the scalability of the technique

# Recall Transformer architecture




$x$: input sequence

$$\text{out} = \text{LN}(c' + \text{FF}(c'))$$

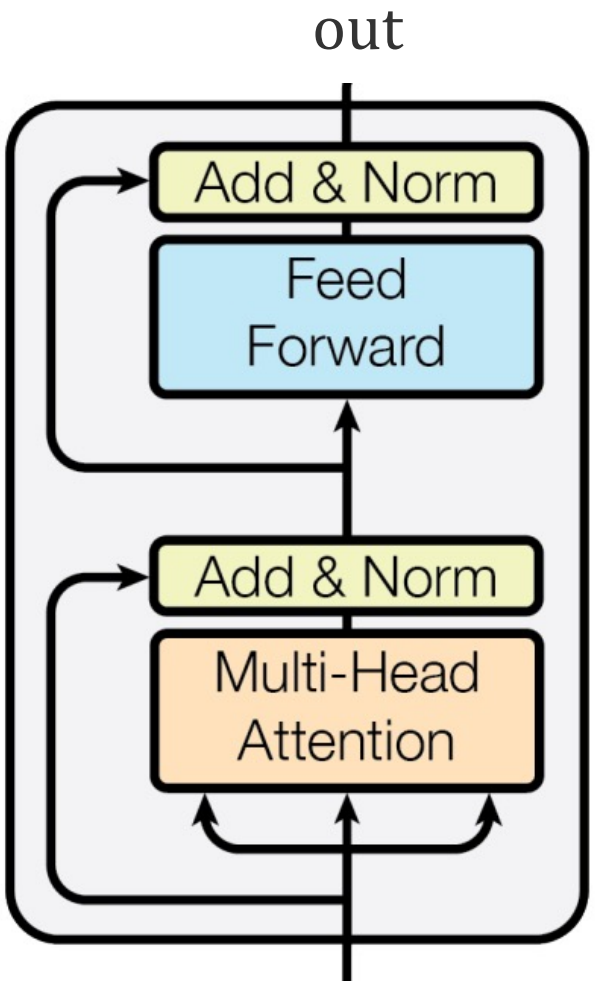
$$FF(c') = f(c'W_1 + b_1)W_2 + b_2$$

$$c' = LN(c + x)$$

$$c = \text{MultiHeadAttention}(\text{q}, \text{k}, \text{v})$$

$$q, k, v = \text{QKV\_Projection}(x)$$

# Recall Transformer architecture




$x$: input sequence

$$\text{out} = \text{LN}(c' + \text{FF}(c'))$$

$$FF(c') = f(c'W_1 + b_1)W_2 + b_2$$

$$c' = LN(c + x)$$

$$c = \text{MultiHeadAttention}(\text{q}, \text{k}, \text{v})$$

$$q, k, v = \text{QKV\_Projection}(x)$$

# Recall self attention

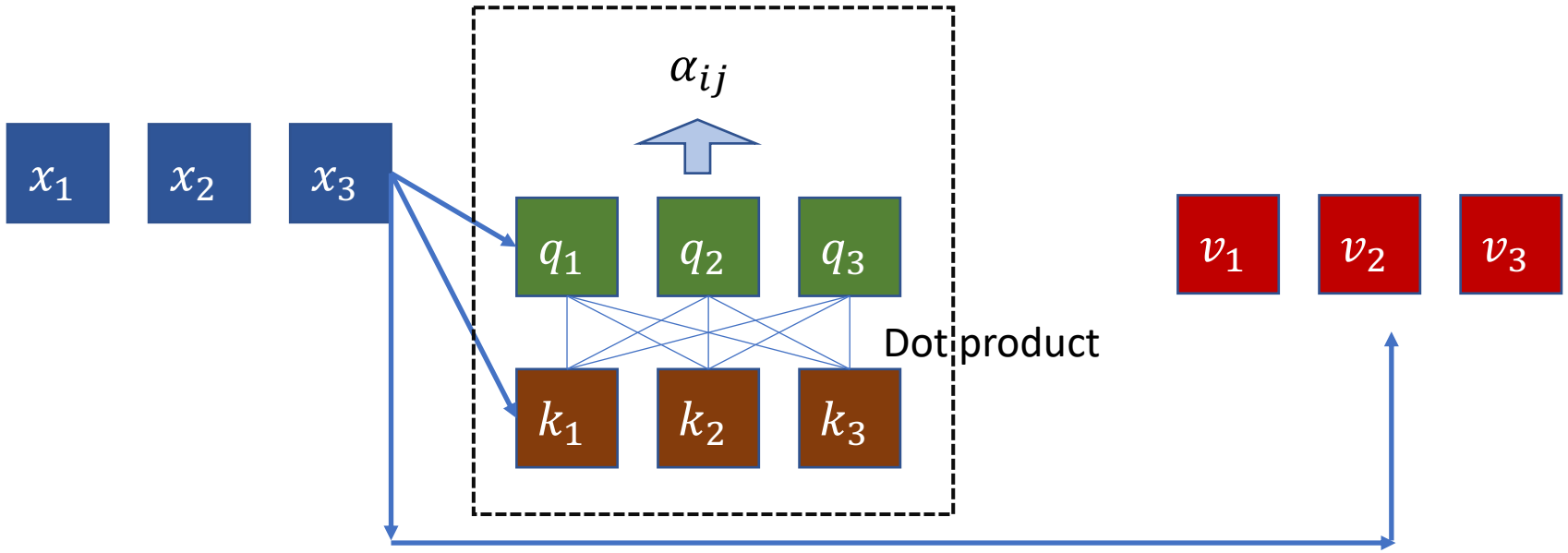
- Combine representations corresponding to each input location with a weighted sum

- The weights are computed by a dot product attention operation

# Recall self attention

- Instead of using one vector $x_i$ for each input location
  - Each input vector is projected into three vectors
    - Query vector $q_i = W_q x_i$
    - Key vector $k_i = W_k x_i$
    - Value vector $v_i = W_v x_i$

$$\alpha_{ij} = \text{softmax}_j(q_i . k_j)$$

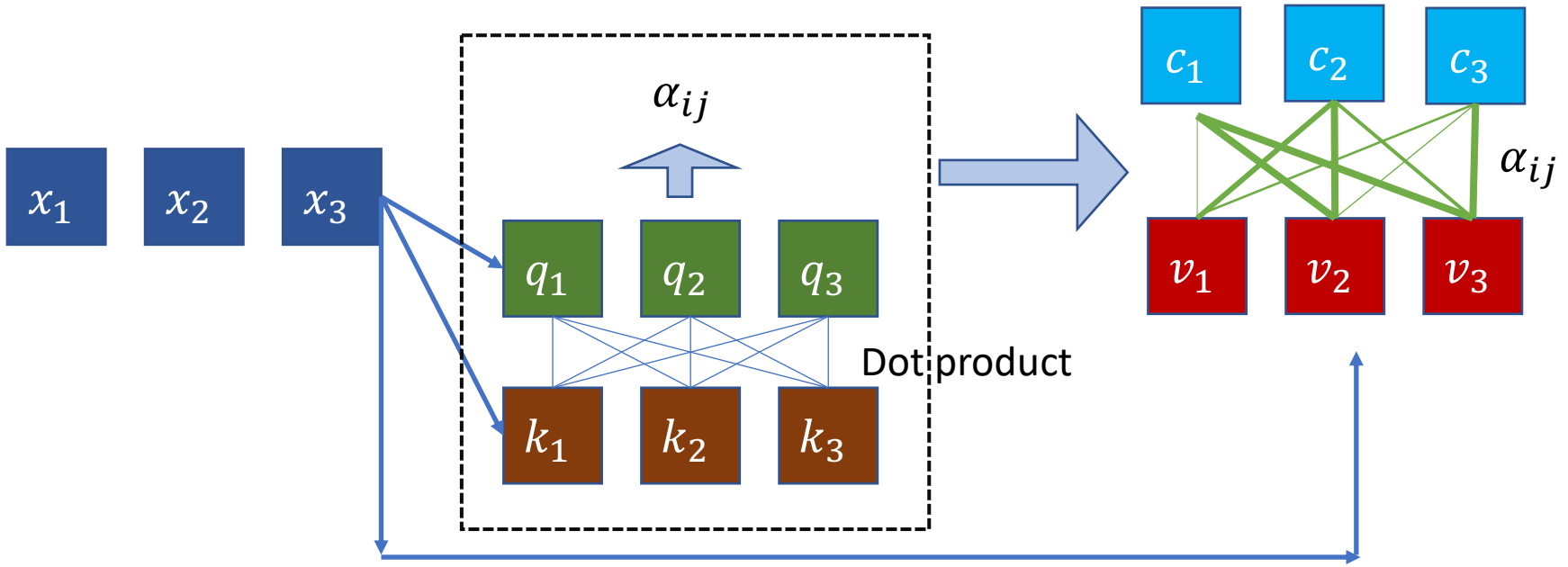


# Recall self attention

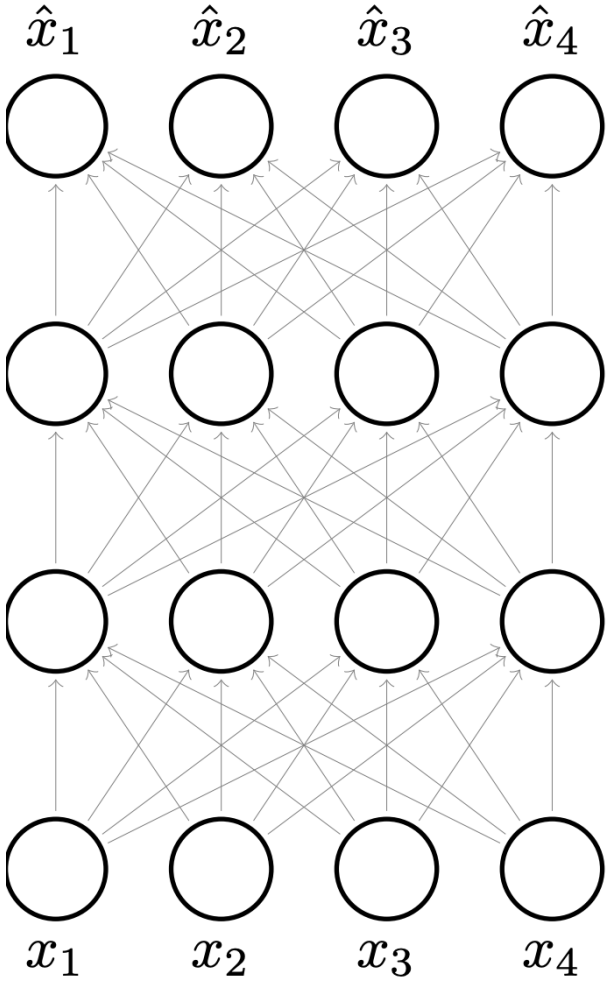
- Instead of using one vector $x_i$ for each input location
  - Each input vector is projected into three vectors
    - Query vector $q_i = W_q x_i$
    - Key vector $k_i = W_k x_i$
    - Value vector $v_i = W_v x_i$

$$\alpha_{ij} = \text{softmax}_j(q_i . k_j)$$

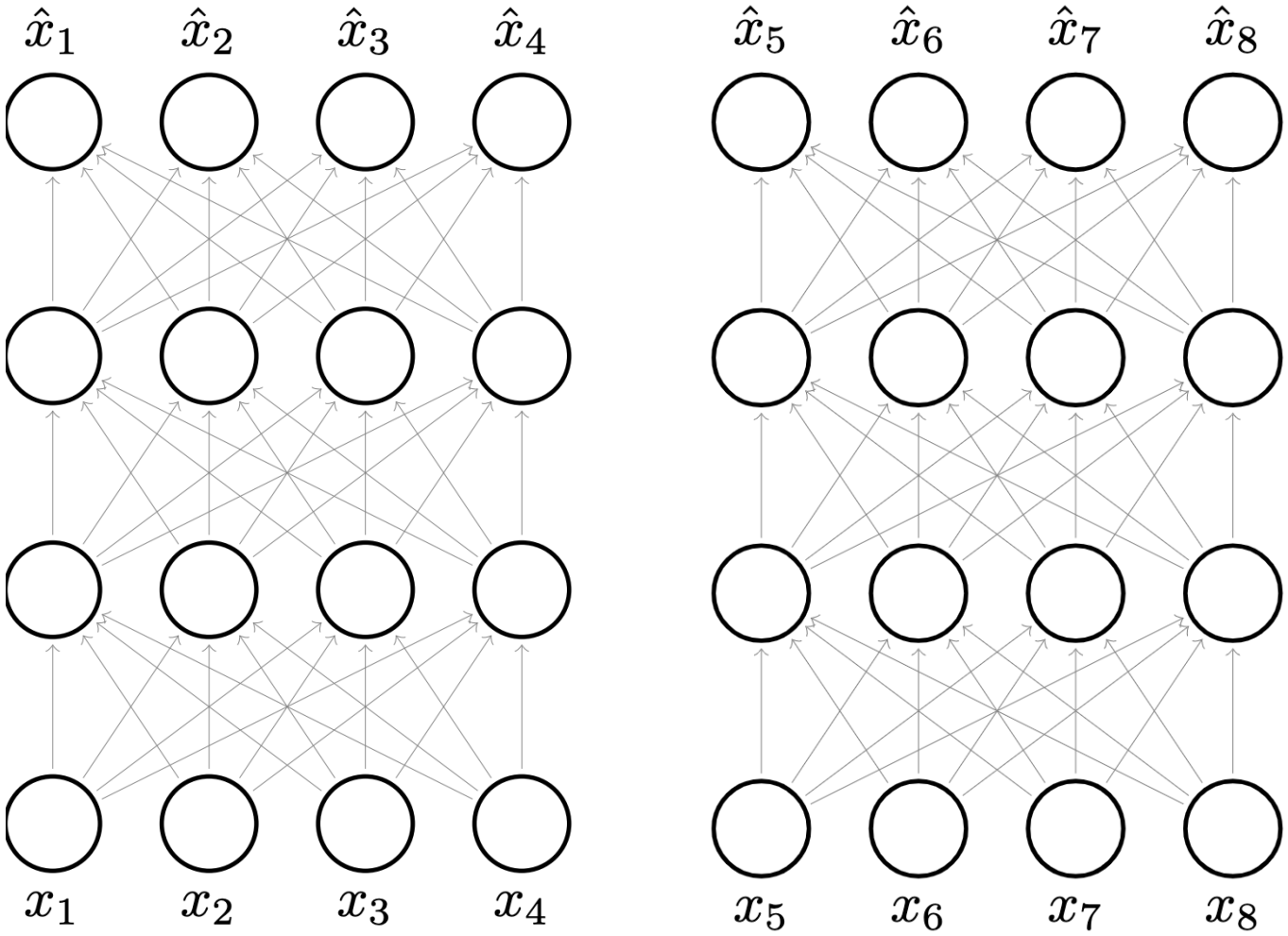
$$c_i = \sum_j \alpha_{ij} v_j$$



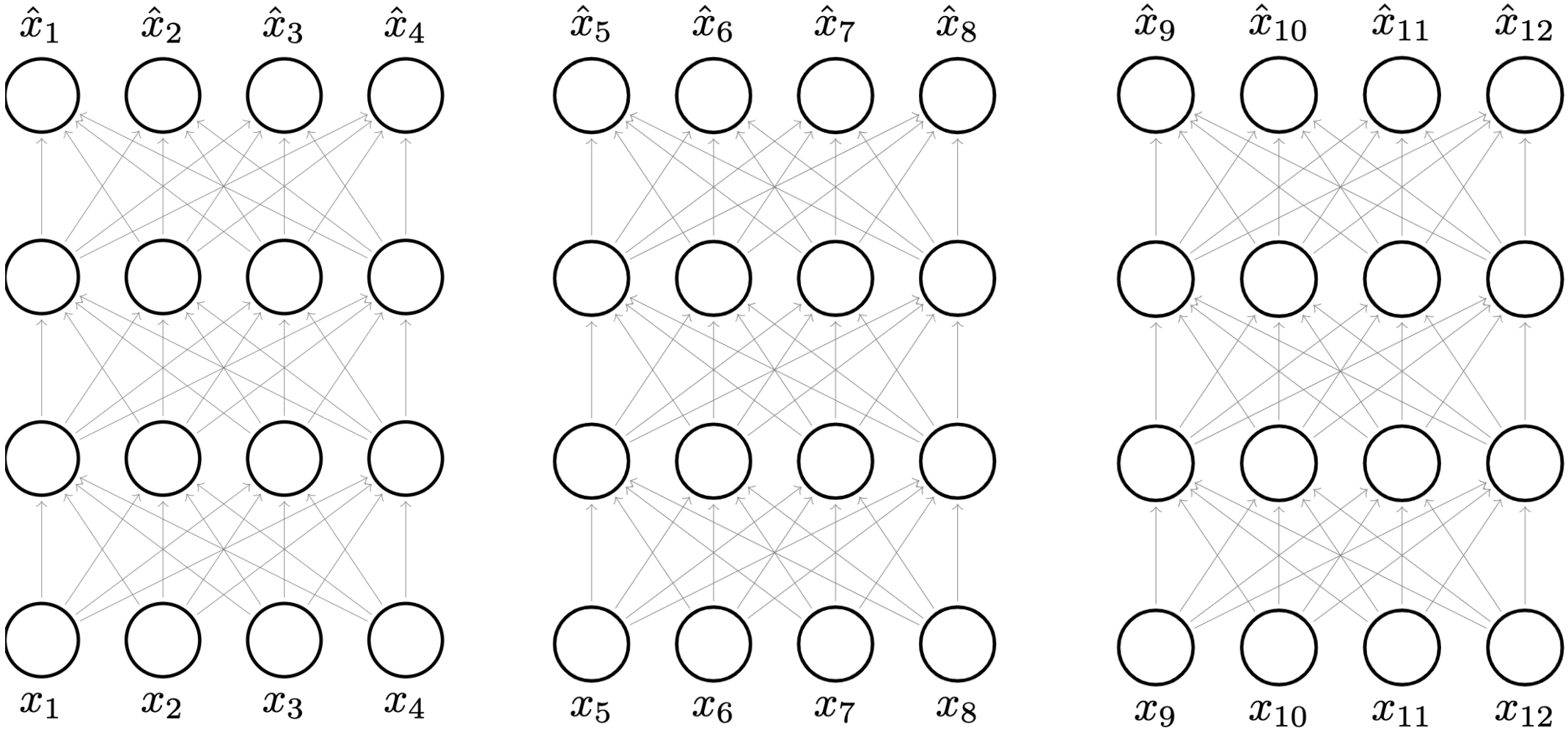
# Memorizing transformers



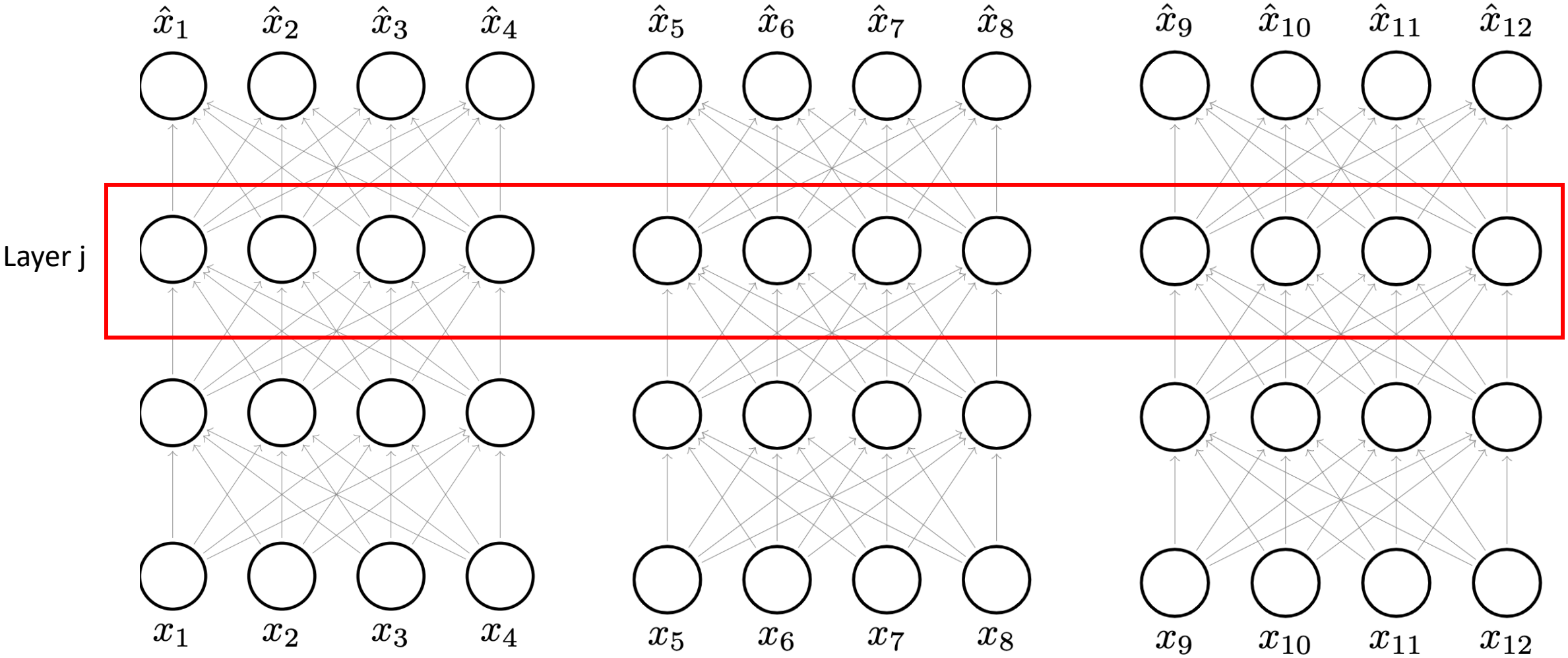
# Memorizing transformers



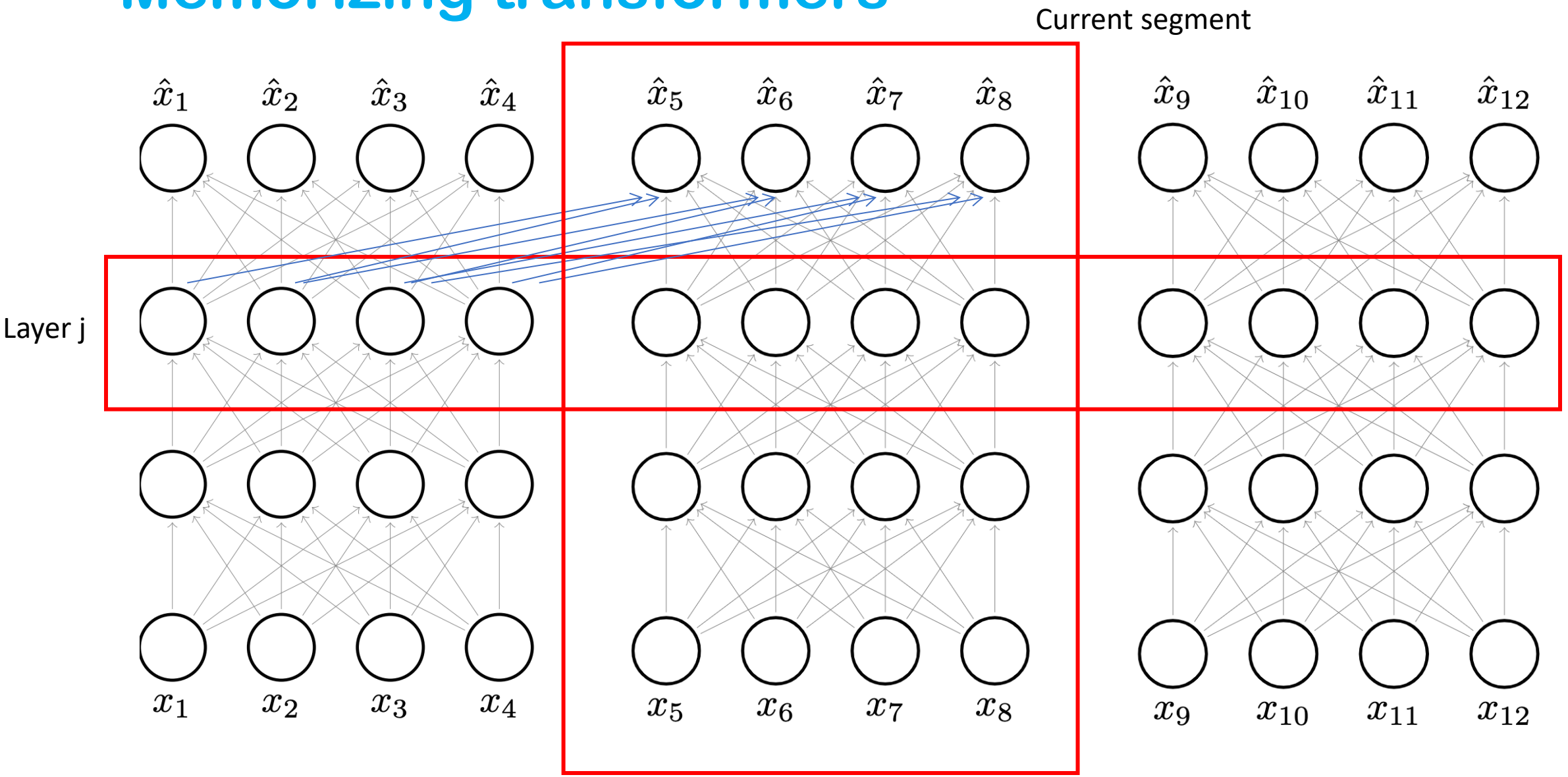
# Memorizing transformers



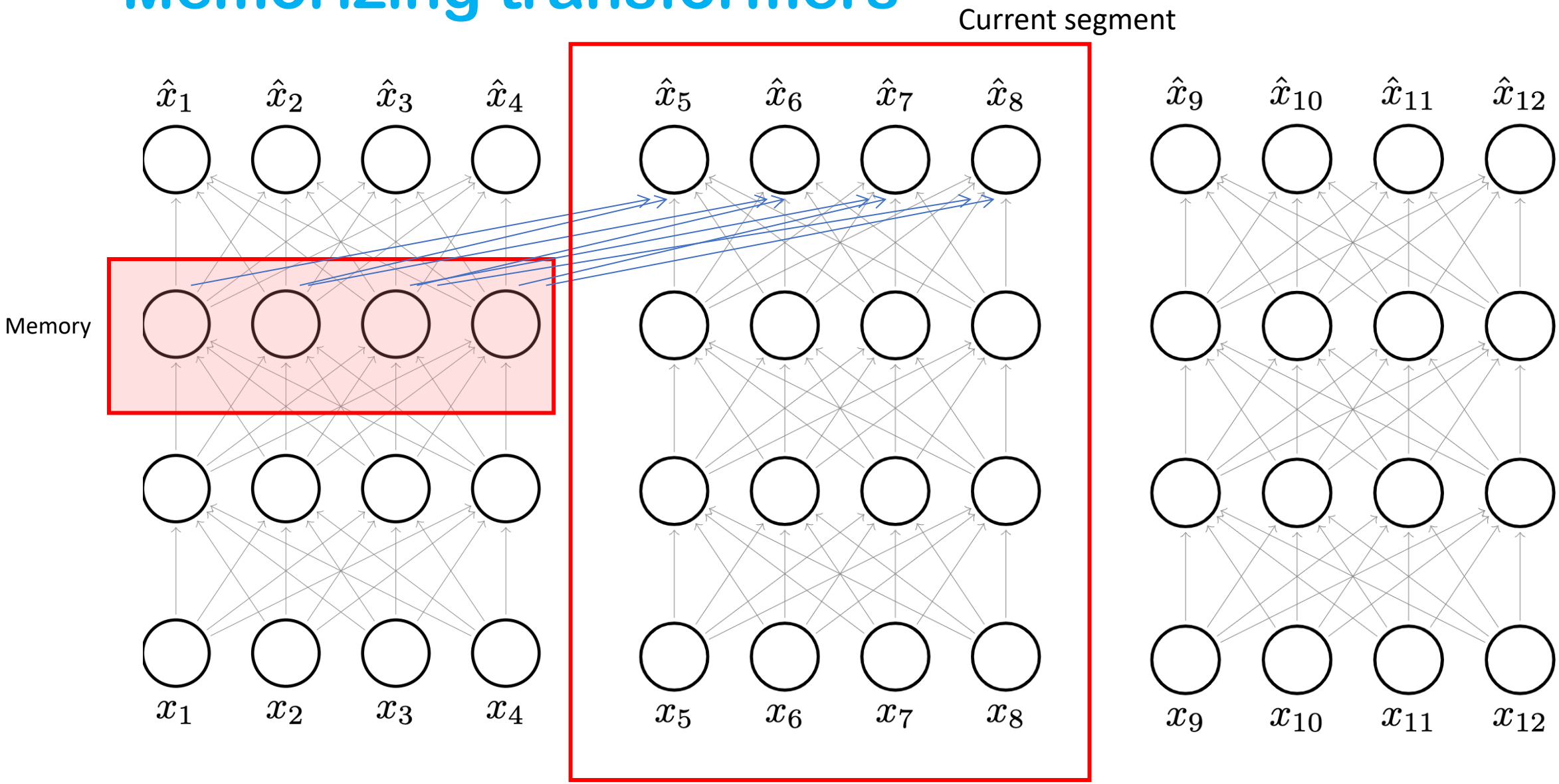
# Memorizing transformers



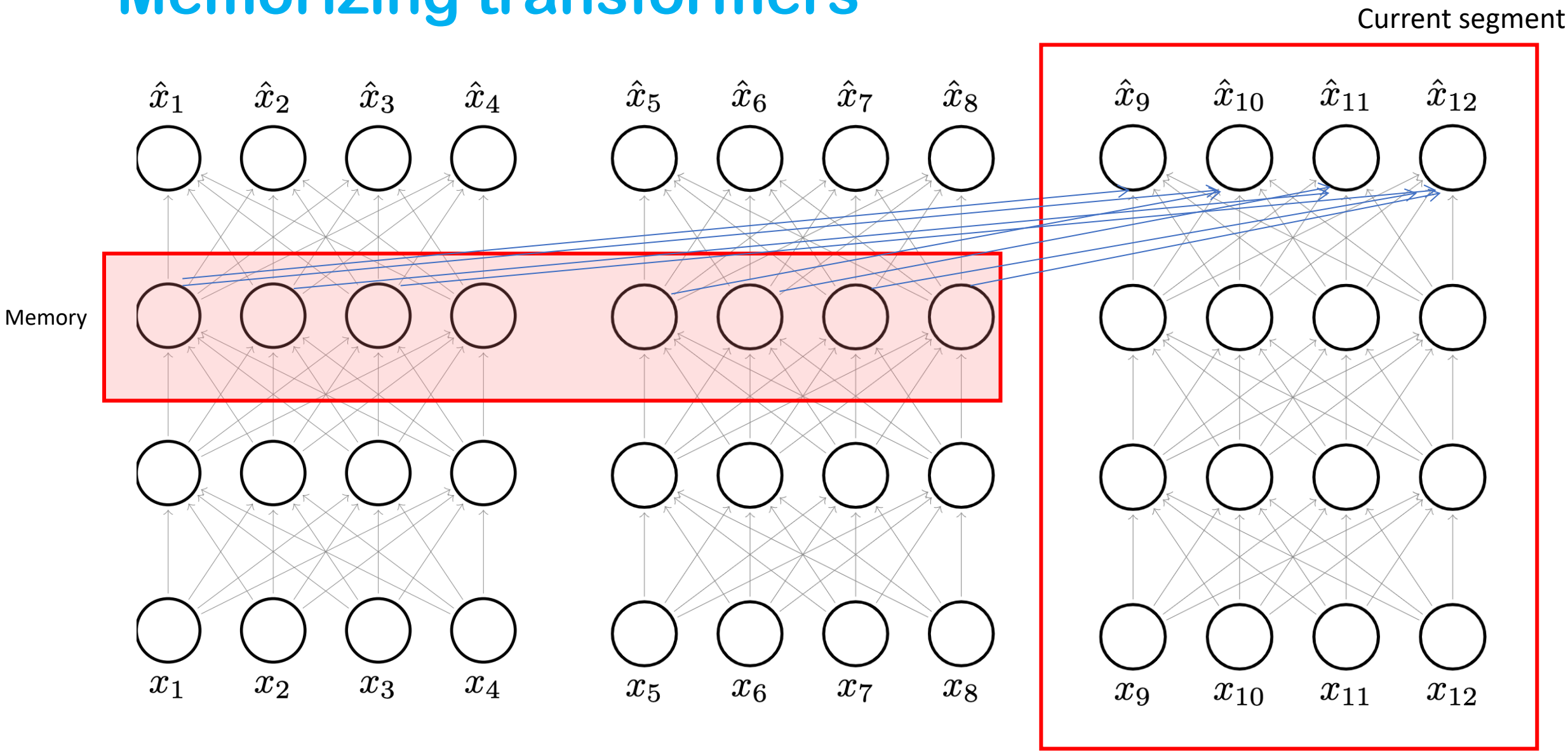
# Memorizing transformers



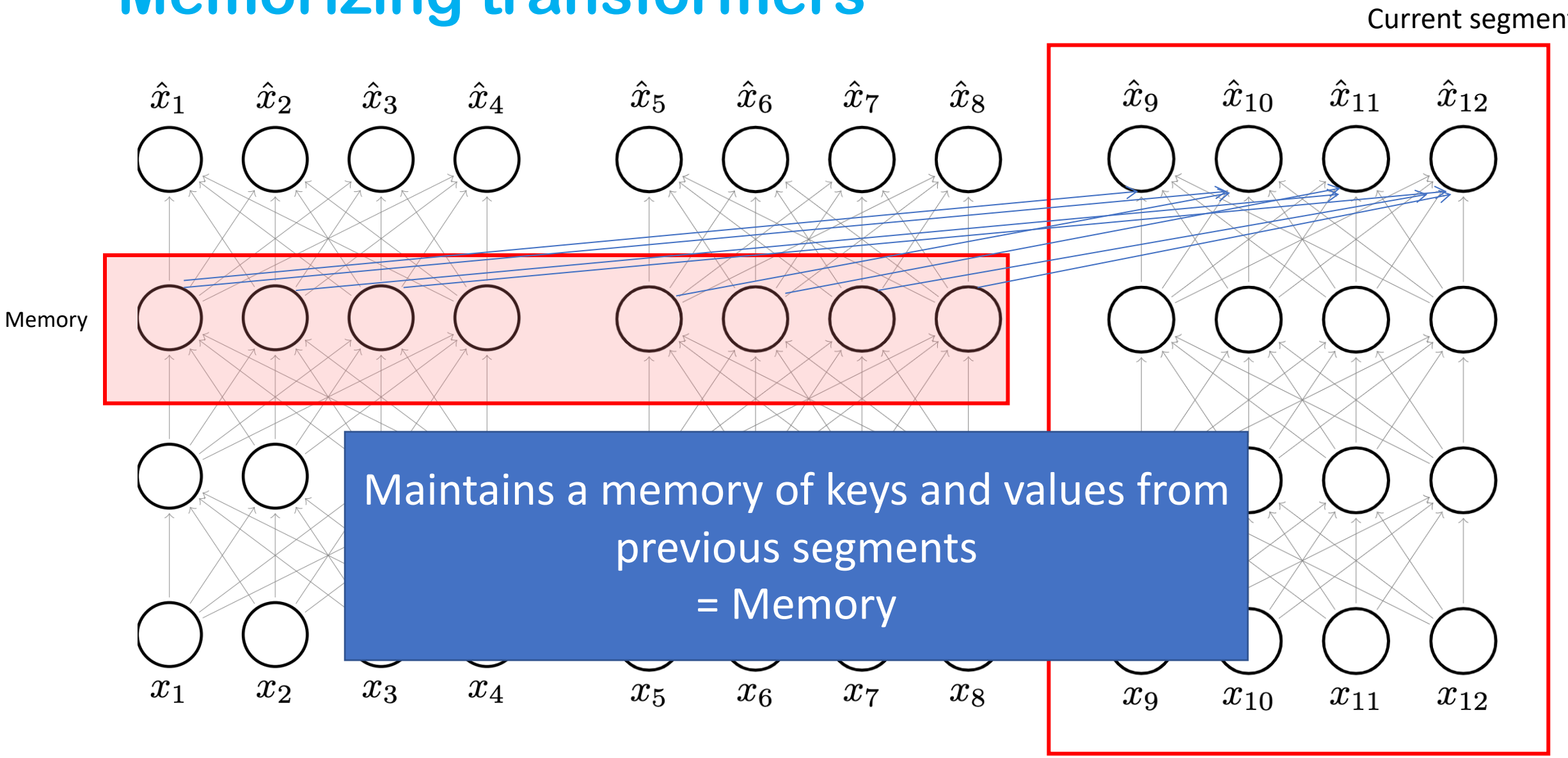
# Memorizing transformers



# Memorizing transformers



# Memorizing transformers



# Memorizing transformer

- If we have many segments, this becomes interactable
  - E.g., if we have 100 segments of 512
  - Then we would be attending to 51200 keys, values
  - Computationally expensive

# Memorizing transformer

- If we have many segments, this becomes interactable
  - E.g., if we have 100 segments of 512
  - Then we would be attending to 51200 keys, values
  - Computationally expensive

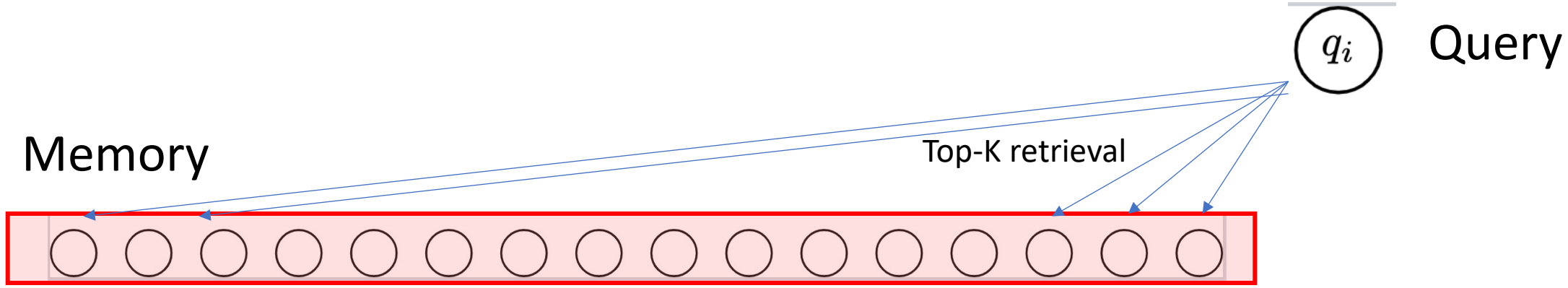
- Solution?
  - Not to attend to everything
  - Just attend to most relevant keys and values

# Memorizing transformer

- If we have many segments, this becomes interactable
  - E.g., if we have 100 segments of 512
  - Then we would be attending to 51200 keys, values
  - Computationally expensive

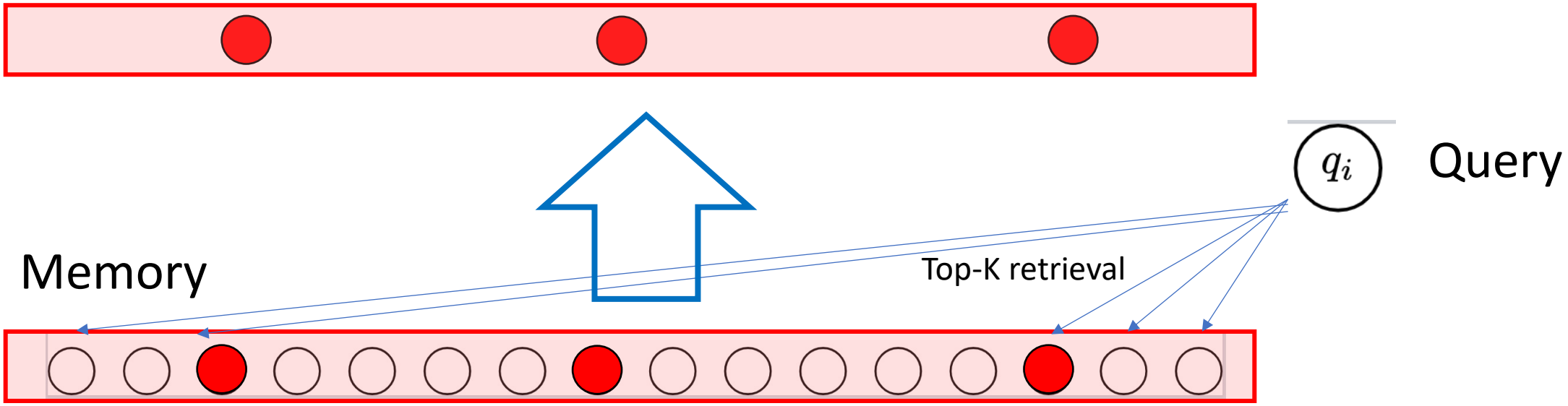
- Solution?
  - Not to attend to everything
  - Just attend to most relevant keys and values
  - How?
    - Retrieve the relevant ones!

# Memorizing transformer




Efficient approximate nearest neighbor algorithm to find top K similar keys

# Memorizing transformer

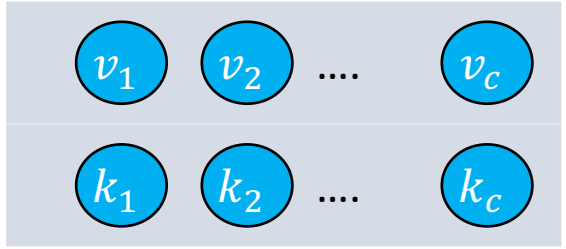



Efficient approximate nearest neighbor algorithm to find top K similar keys

# Memorizing transformers

Retrieved keys and values from memory

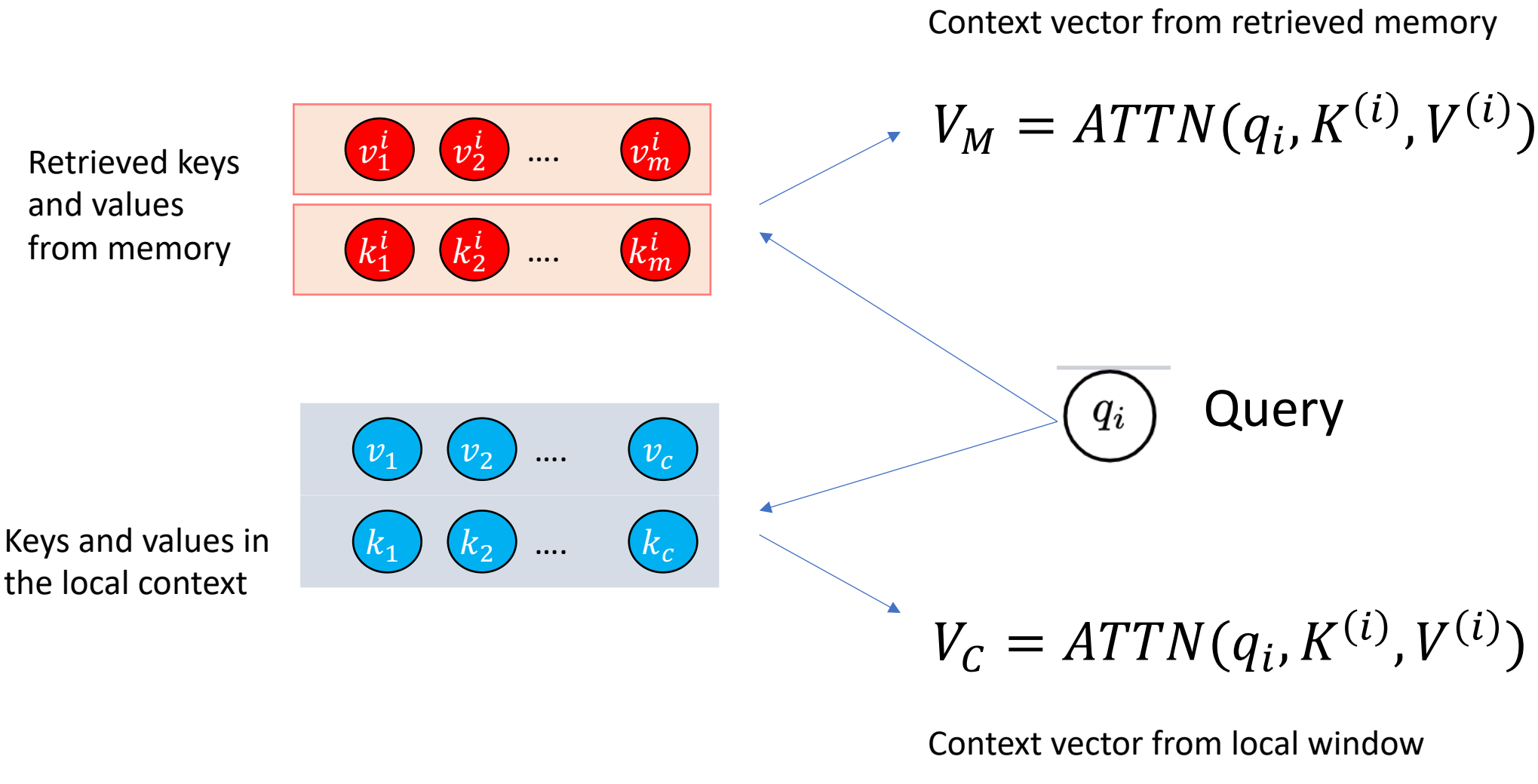

Keys and values in the local context



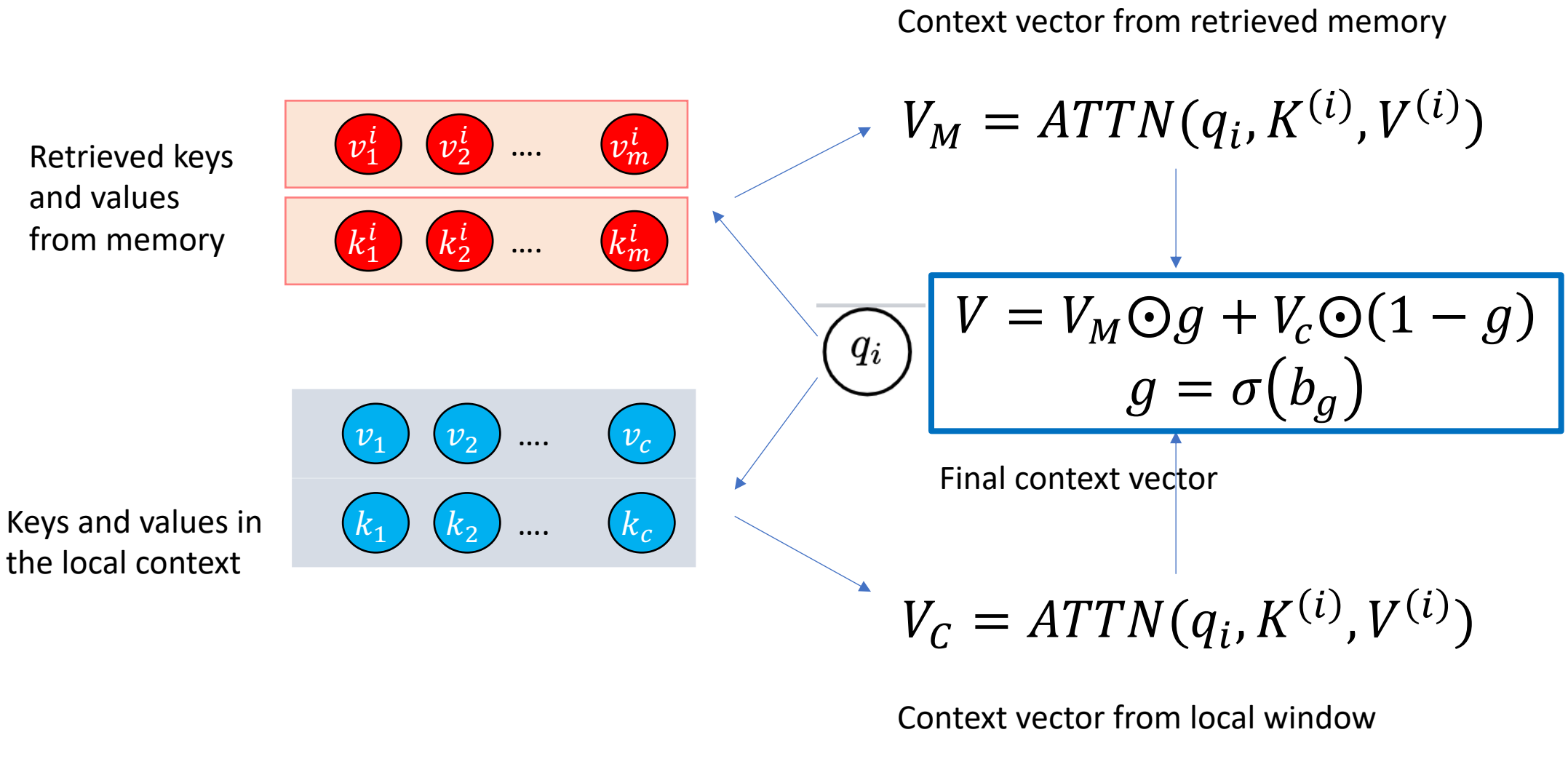


Query

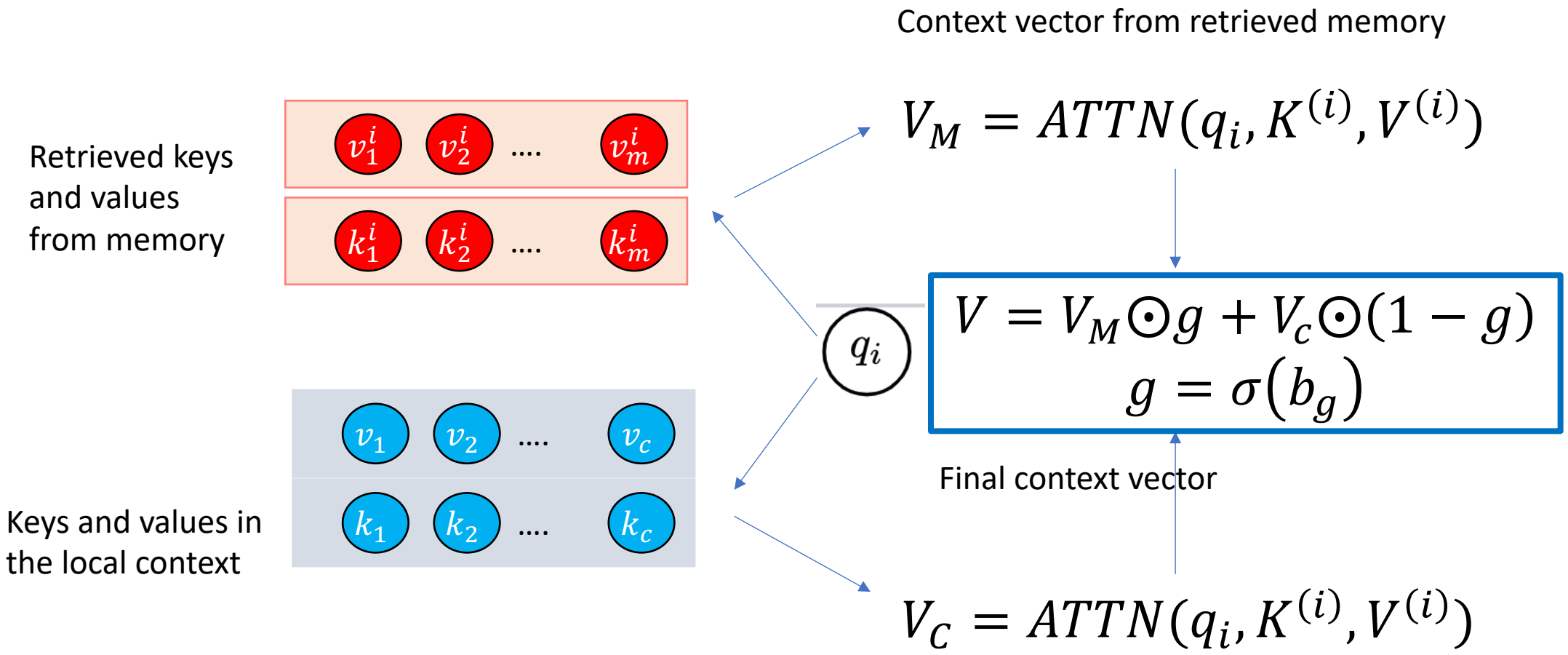
# Memorizing transformers - self attention



# Memorizing transformers – self attention



# Memorizing transformers – self attention



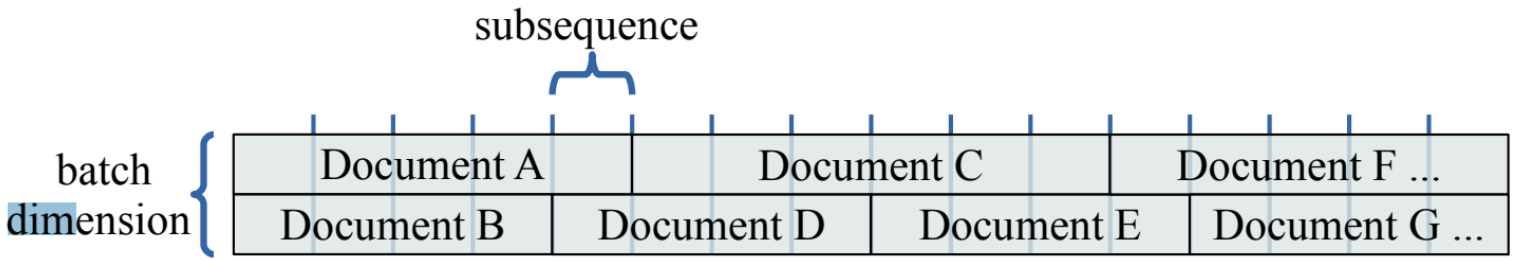

The bias $b_g$ is a learned per-head scalar parameter, which allows each head to choose between local and long-range attention

# Details

- Position information
  - Similar to T5, use relative position bias for local attention
  - For KNN-attention, they don’t use position information

# Details

- Batching




  - Separate memory for each slice in the batch
  - Clear the memory after each document

# Details

- Distributional shift
  - As memory becomes long, the older keys, and values become stale
  - Remember, no backpropagation into the memory
  - Their magnitude might also change
  - To alleviate this, they normalize keys and queries

# Details

- Approximate kNN instead of exact kNN
  - Faster and more efficient
  - Custom approximate kNN method for TPUs with recall of 90%

# Experiments

*Perplexity: The lower the better.*

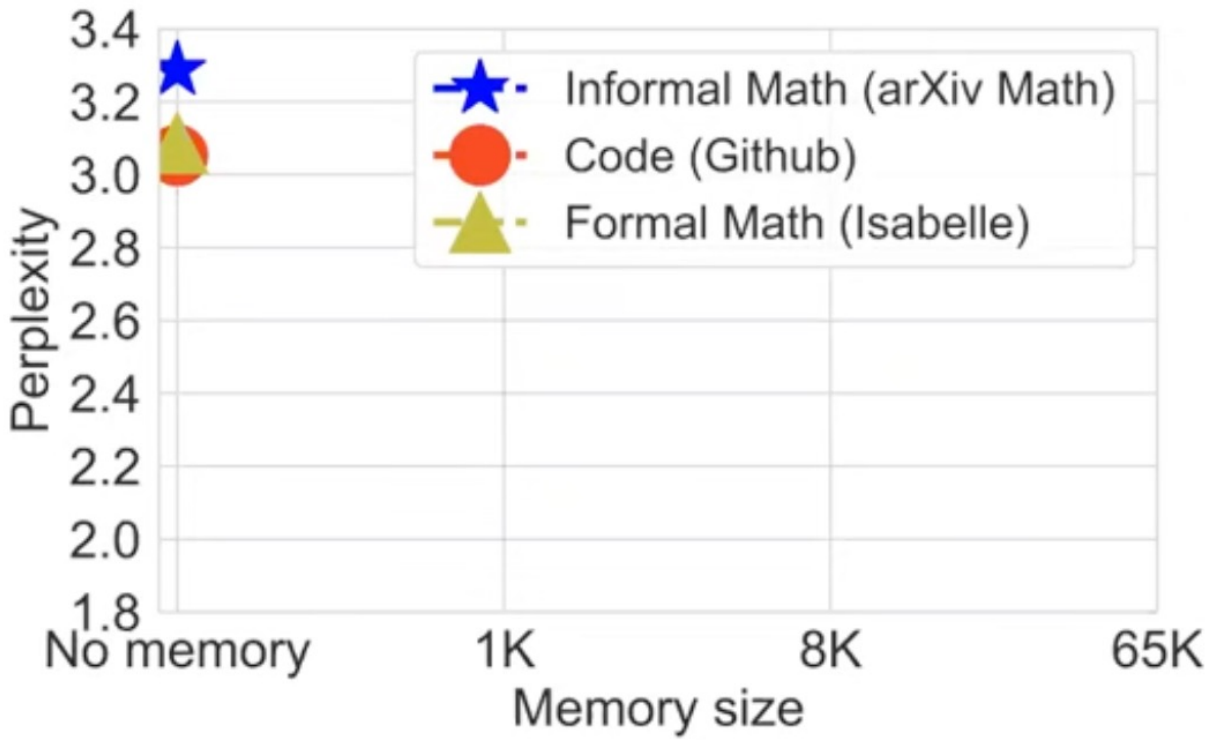



Fig from authors

# Experiments

*Perplexity: The lower the better.*

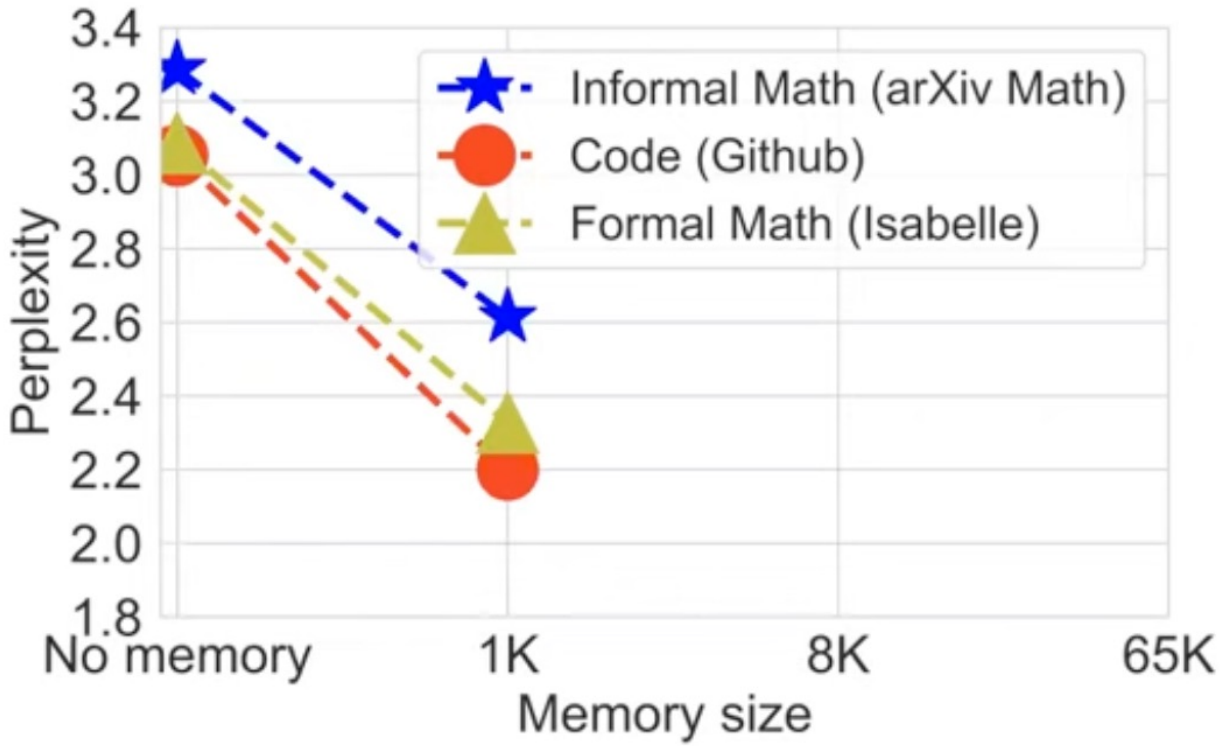



Fig from authors

# Experiments

*Perplexity: The lower the better.*

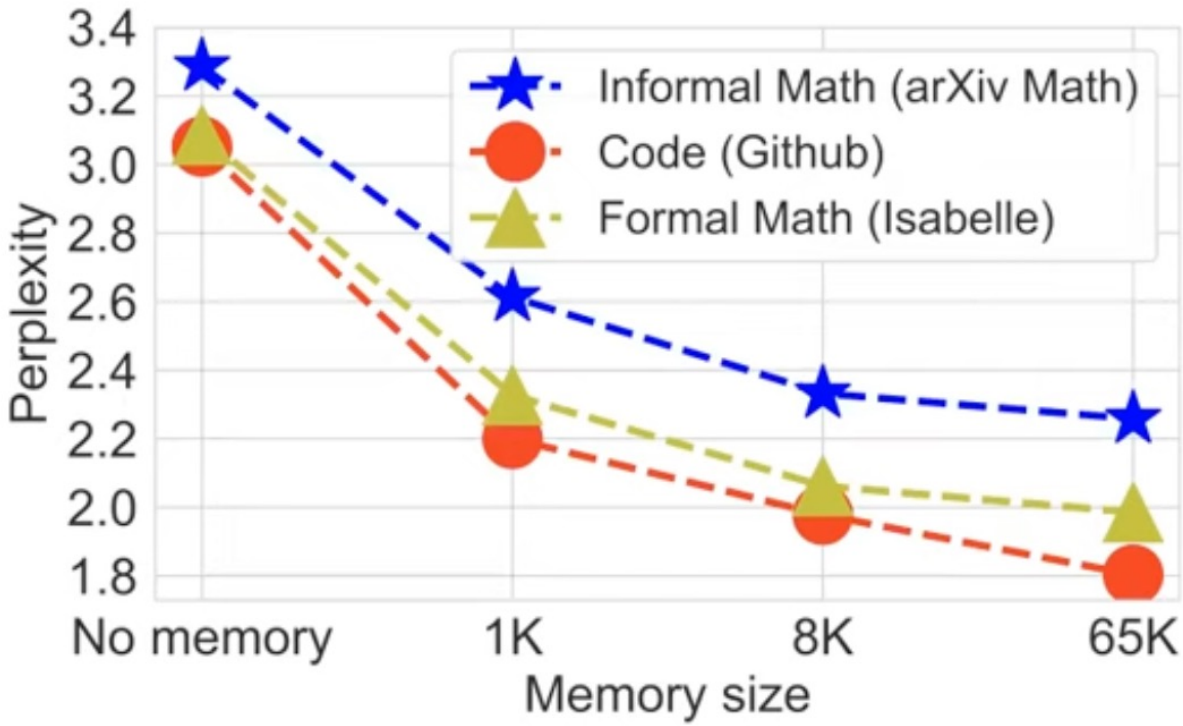



Fig from authors

# Results

- Also helps in language modeling tasks from general domain

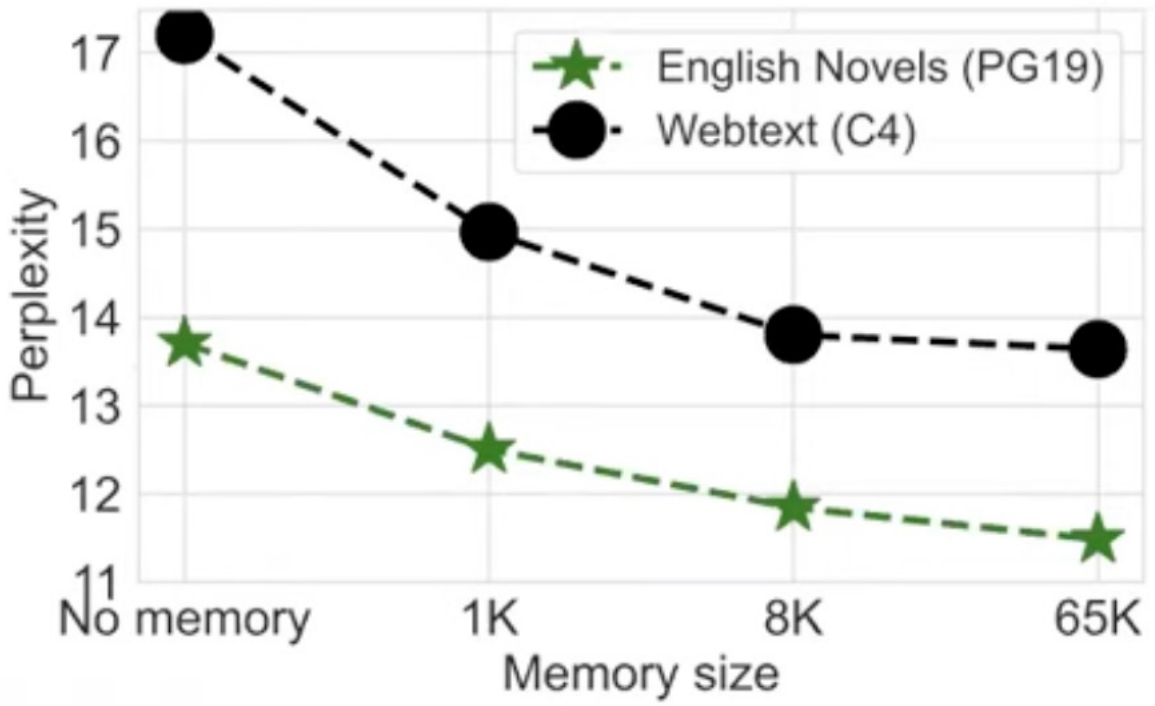
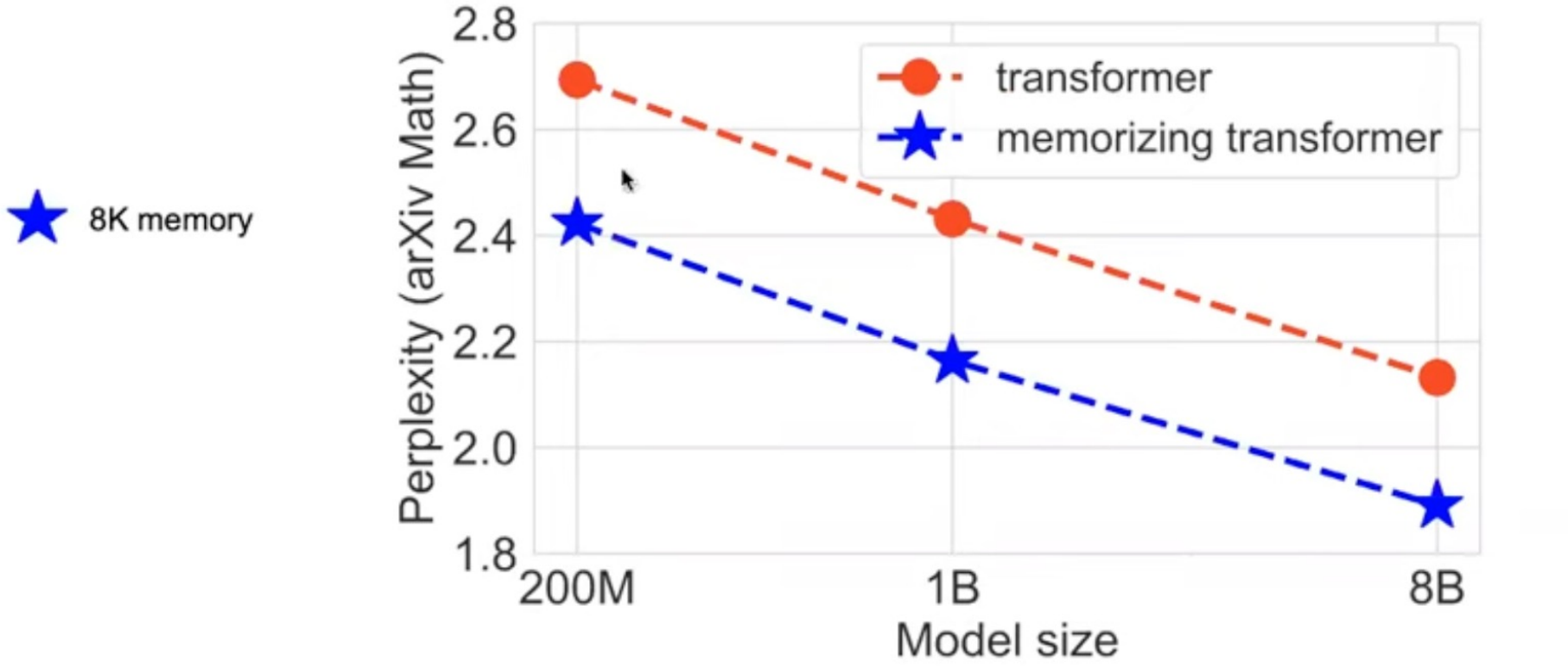



Fig from authors

# Scale

Adding an 8K memory results in similar performance to an 8X larger model

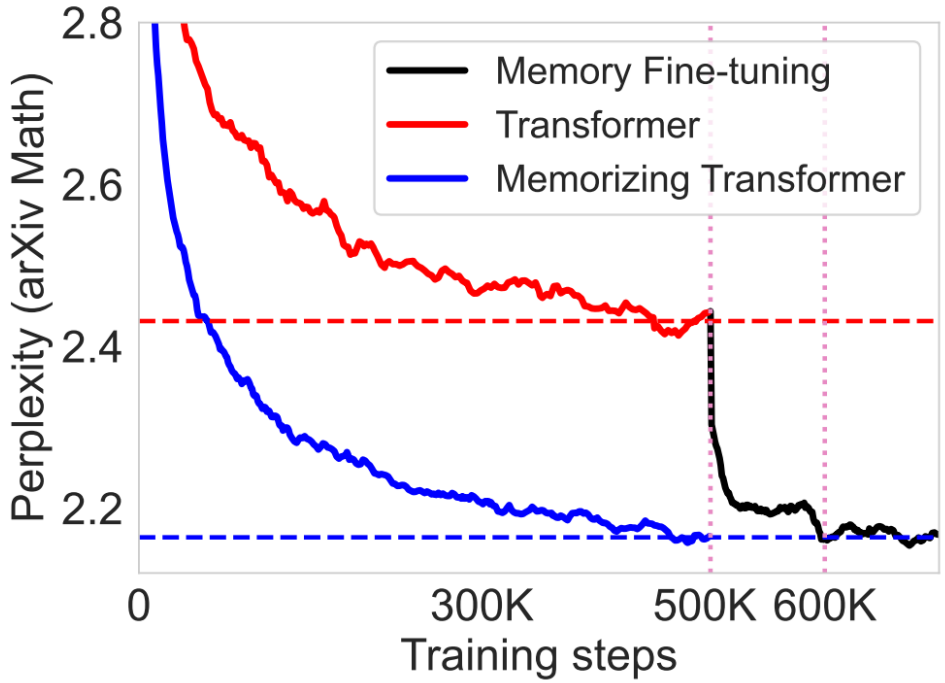


# Finetuning

- What if we don’t want to train with memory from scratch?

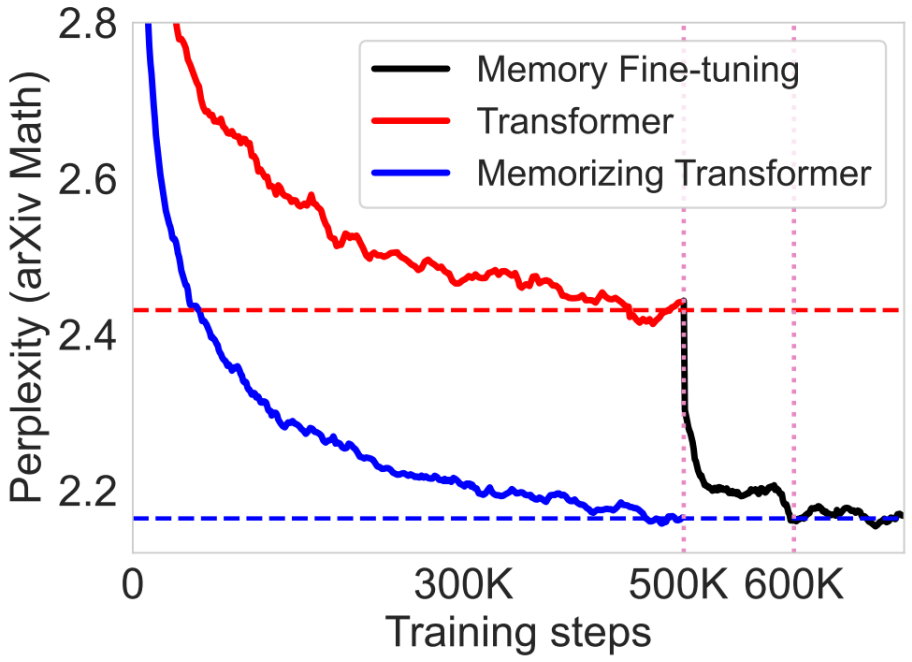
# Finetuning

- What if we don’t want to train with memory from scratch?



# Finetuning

- What if we don’t want to train with memory from scratch?
- Within 20K steps (4% of the pre-training time) the fine-tuned model has already closed 85% of the gap
- After 100k steps it has closed the gap entirely.



# Scaling the memory size

Results from a 256K memory is comparable to a 40X larger model

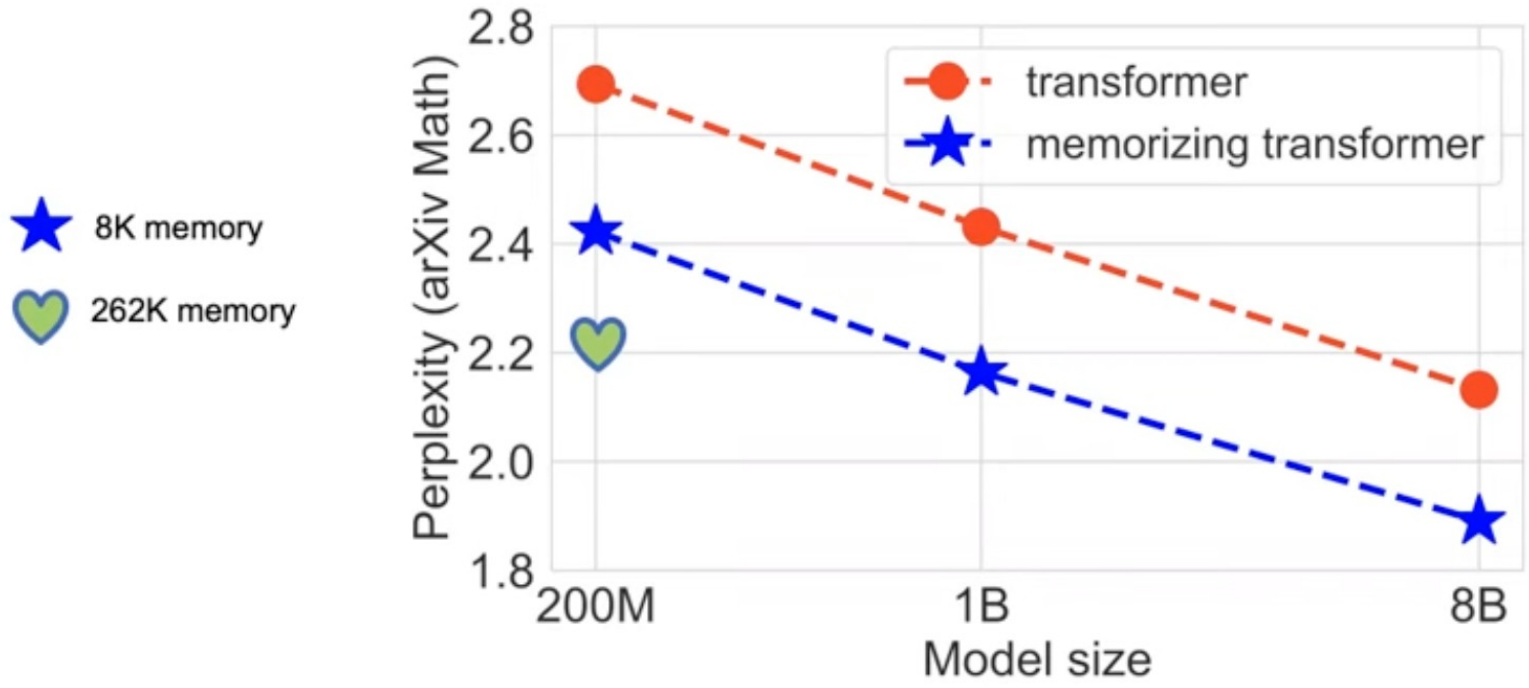



Fig from authors of memorizing transformers

# What does memorizing transformer retrieve?

Predicting lemma name

```
also have "... ≤ ES.expectation ?Y / 1"
  by (rule prob_space.markov_inequality)
```

# What does memorizing transformer retrieve?

Predicting lemma name

```
also have "... ≤ ES.expectation ?Y / 1"
  by (rule prob_space.markov_inequality)
```

Look up definitions -- 20K tokens apart.

```
lemma markov_inequality:
  assumes "⋀a. 0 ≤ X a" and "integrable M X" "0 < t"
  shows "prob {a ∈ space M. t ≤ X a} ≤ expectation X / t"
proof -
  --{* proof adapted from @{thm [source] edge_space.Markov_inequal
       @{term prob_space}s *}
  have "(∫⁺ x. ennreal (X x) ∂M) = (∫x. X x ∂M)"
    using assms by (intro nn_integral_eq_integral) auto
```

# What does memorizing transformer retrieve?

Predicting lemma name

Same structure

```
also have "... ≤ ES.expectation ?Y / 1"
  by (rule prob_space.markov_inequality)
```

Look up definitions -- 20K tokens apart.

```
lemma markov_inequality:
  assumes "⋀a. 0 ≤ X a" and "integrable M X" "0 < t"
  shows "prob {a ∈ space M. t ≤ X a} ≤ expectation X / t"
proof -
  --{* proof adapted from @{thm [source] edge_space.Markov_inequal
       @{term prob_space}s *}
  have "(∫⁺ x. ennreal (X x) ∂M) = (∫x. X x ∂M)"
    using assms by (intro nn_integral_eq_integral) auto
```

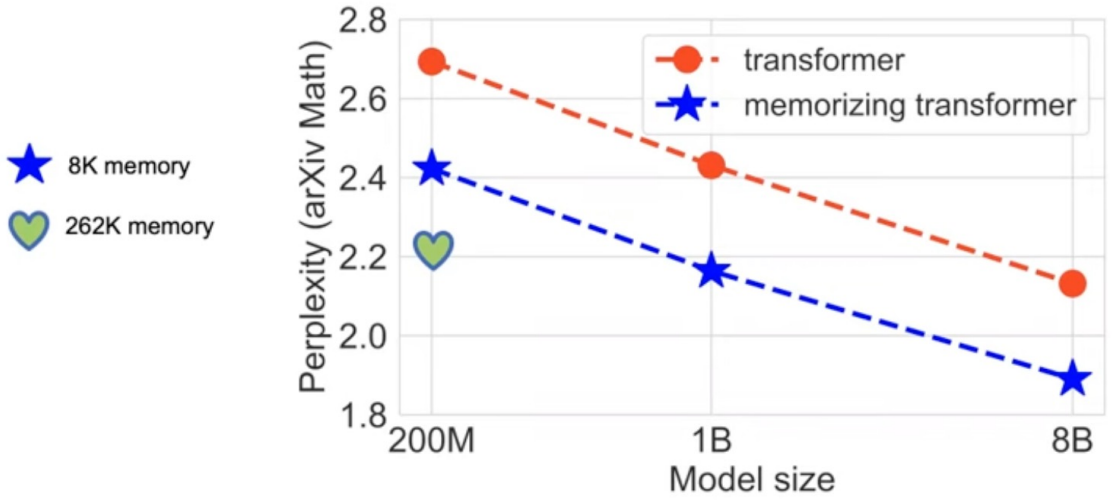
Example from authors

# Takeaways

- Memory helps utilizing longer dependencies
- Helps improving LM performance without scaling up the model size

# Discussion

- Exploring the limits of the memory size?
  - We saw that larger memory helps improving performance in a nontrivial way
  - What is the limit?
    - What if we could memorize the entire knowledge-base (e.g., Wikipedia)
    - Memorize the entire internet?



# Discussion

- How does memorizing transformers connect with other efficient transformers?
  - E.g., Sparse methods?

# Discussion

- How do these type of models can help with abilities like ICL or Few-shot learning?