Attention is all you need
Is LLM?

CSE 240D

Spring 2026

Hanyang Xu

How do we model language: Tokenization

Byte Pair Encoding (BPE)

Let vocabulary be the set of all individual characters

= {A, B, C, D,…,a, b, c, d….}

Repeat:

- choose the two symbols that are most frequently adjacent in training corpus (say 'A', 'B'),
- adds a new merged symbol 'AB' to the vocabulary
- replace every adjacent 'A' 'B' in corpus with 'AB'.

Until *k* merges have been done.

Byte Pair Encoding: Efficient Estimation of Word Representations in Vector Space – ACL 2016

How do we model language: Tokenization

BPE token learner

Original (very fascinating🙄) corpus:

`set␣new␣new␣renew␣reset␣renew`

Put space token at start of words

corpus		vocabulary
2	␣ n e w	␣, e, n, r, s, t, w
2	␣ r e n e w	
1	s e t	
1	␣ r e s e t	

Credit to Dan Jurafsky

How do we model language: Tokenization

BPE token learner

corpus

```
2   ␣ n e w
2   ␣ r e n e w
1   s e t
1   ␣ r e s e t
```

vocabulary

```
␣, e, n, r, s, t, w
```

Merge n e to ne (count 4 = 2 new + 2 renew)

corpus

```
2   ␣ ne w
2   ␣ r e ne w
1   s e t
1   ␣ r e s e t
```

vocabulary

```
␣, e, n, r, s, t, w, ne
```

Credit to Dan Jurafsky

How do we model language: Tokenization

BPE token learner

corpus		vocabulary
2	_ ne w	_, e, n, r, s, t, w, ne
2	_ r e ne w	
1	s e t	
1	_ r e s e t	

Merge ne w to new (count 4)

corpus		vocabulary
2	_ new	_, e, n, r, s, t, w, ne, new
2	_ r e new	
1	s e t	
1	_ r e s e t	

Credit to Dan Jurafsky

How do we model language: Tokenization

BPE token learner

Original (very fascinating🙄) corpus:

`set␣new␣new␣renew␣reset␣renew`

Put space token at start of words

corpus		vocabulary
2	␣ n e w	␣, e, n, r, s, t, w
2	␣ r e n e w	
1	s e t	
1	␣ r e s e t	

Credit to Julia Hockenmaier

How do we model language: Embeddings

Main idea:

Train a binary classifier to predict which words *c* appear in the context of (i.e. near) a target word *t*.

The parameters of that classifier provide a dense vector representation (embedding) of the target word *t*.

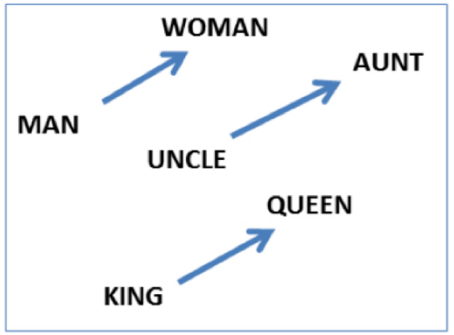
word2vec: Efficient Estimation of Word Representations in Vector Space – ICLR 2013

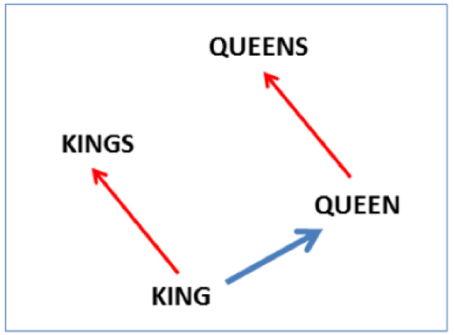
Credit to Julia Hockenmaier

How do we model language: Embeddings

Analogy: Embeddings capture relational meaning!

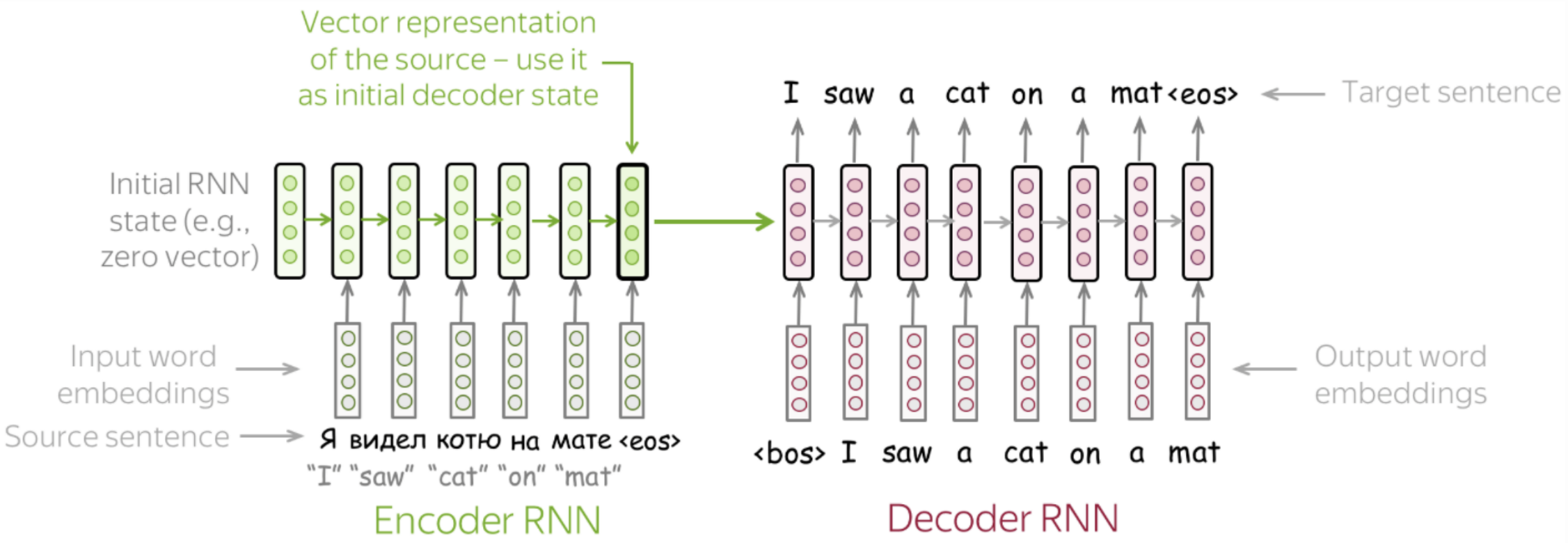
vector(‘*king*’) - vector(‘*man*’) + vector(‘*woman*’) =
vector(‘queen’)
vector(‘*Paris*’) - vector(‘*France*’) + vector(‘*Italy*’) =
vector(‘*Rome*’)





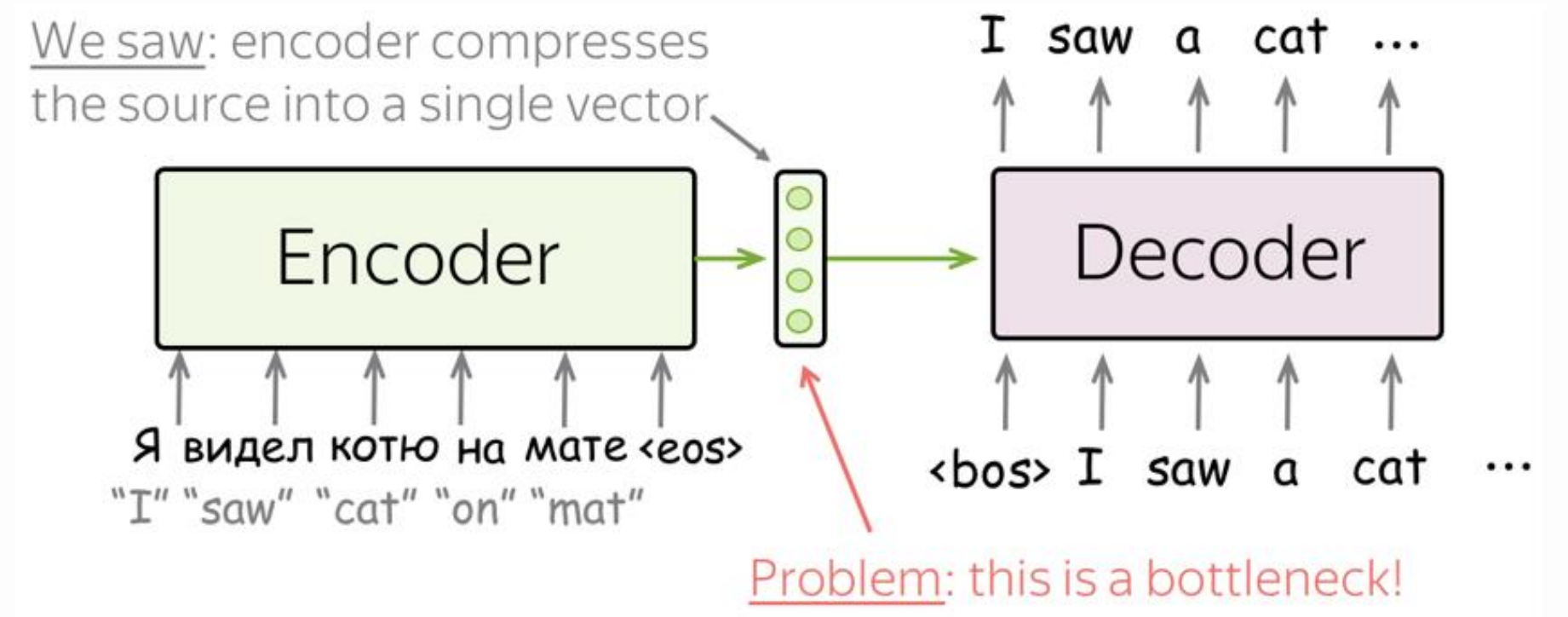
Credit to Julia Hockenmaier

Models before Transformers: Recurrent Neural Networks



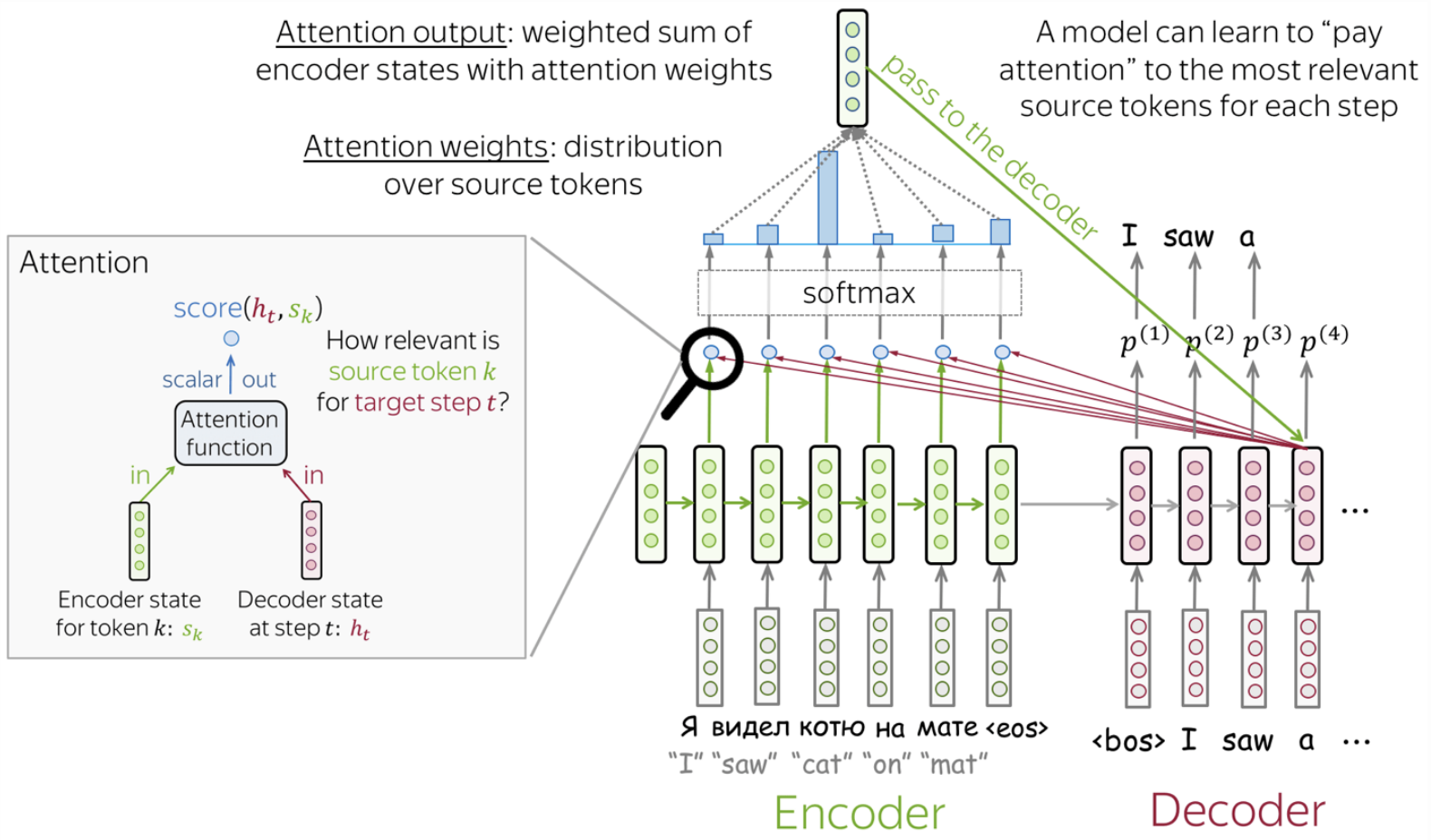
Credit to Lena Voita

Models before Transformers: Recurrent Neural Networks



Credit to Lena Voita

Attention in RNNs



Neural Machine Translation by Jointly Learning to Align and Translate – ICLR'2015

Credit to Lena Voita

How do we model language: Models before Transformers

Attention output

$$c^{(t)} = a_1^{(t)} s_1 + a_2^{(t)} s_2 + \cdots + a_m^{(t)} s_m = \sum_{k=1}^{m} a_k^{(t)} s_k$$

"source context for decoder step t"

(weighted sum)

Attention weights

$$a_k^{(t)} = \frac{\exp(\text{score}(h_t, s_k))}{\sum_{i=1}^{m} \exp(\text{score}(h_t, s_i))}, \text{k} = 1..\text{m}$$

"attention weight for source token k at decoder step t"

(softmax)

Attention scores

$$\text{score}(h_t, s_k), \text{k} = 1..\text{m}$$

"How relevant is source token k for target step t?"

Attention input

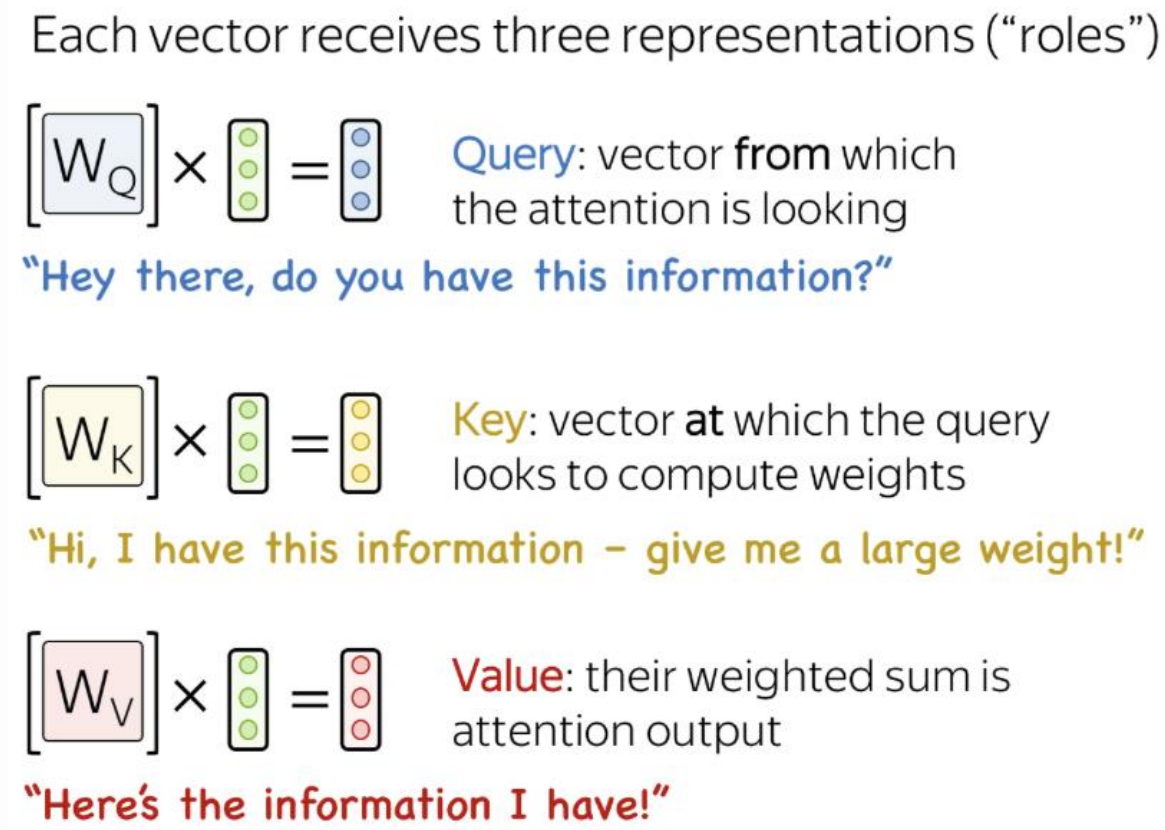
$s_1, s_2, \ldots, s_m$ all encoder states

h_t one decoder state

Neural Machine Translation by Jointly Learning to Align and Translate – ICLR'2015

Credit to Lena Voita

Attention Step1: Query/Key/Value Projection



Sequence Length = ***n***, *Model/Embedding Dimension* = ***d***

Query Q: The vector representation used to *probe* for relevant information.
For a single token: (1×d) @ (d×d) → (1×d) vector
For a sequence: (n×d) @ (d×d) → (n×d) vector

Key K: The vector representation used to *be matched against* queries to determine relevance.
For a single token: (1×d) @ (d×d) → (1×d) vector
For a sequence: (n×d) @ (d×d) → (n×d) vector

Value V: The vector representation used as the *content payload* combined into the attention output.
For a single token: (1×d) @ (d×d) → (1×d) vector
For a sequence: (n×d) @ (d×d) → (n×d) vector

Credit to Lena Voita

Attention Overheads: Compute/Memory Load

Sequence Length = n, Model/Embedding Dimension = d, number of attention head = h

Input X = (n,d)
W_q, W_k, W_v, W_o = (d, d)

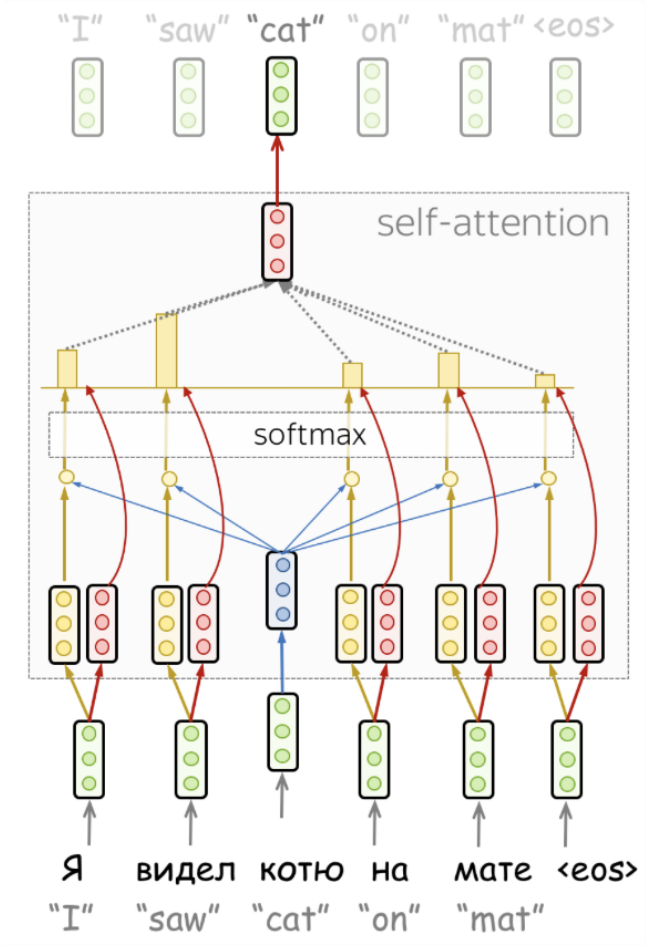
Steps	Operation	Compute Complexity	Memory Complexity
1. QKV Projection	X @ (W_q)	3 n * d^2	3nd
2. Attention Score	Q @ (K^T)	n^2 * d	n^2 * h
3. Softmax+Scale	softmax(scores / √d)	n^2 * h	n^2 * h
4. Weight sum	score @ V	n^2 * d	n * d
5. Output Projection	output @ W_O	n * d^2	n * d

Attention scales quadratically with sequence length.

Self-Attention Step 2-4:

$$Attention(q, k, v) = \overbrace{softmax\left(\frac{qk^T}{\sqrt{d_k}}\right)}^{\text{Attention weights}} v$$

from (q) · to (k) · vector dimensionality of K, V



Step2 - Q @ (K^T) :

Q of token "cat" **dot product with** all other tokens's Key

Meaning: Cosine Similarity between query embedding of cat and key embeddings of all other tokens, scaled by Magnitude

For a single token: (1×d) @ (dxn) → (1×n) vector

For a sequence: (n×d) @ (dxn) → (n×n) vector

Step3 - Scale by dimension:

Both angle(cosine similarity) AND magnitude carries correlation meaning, but dot product also scales with dimension, so we offset that

Credit to Lena Voita

Attention Overheads: Compute/Memory Load

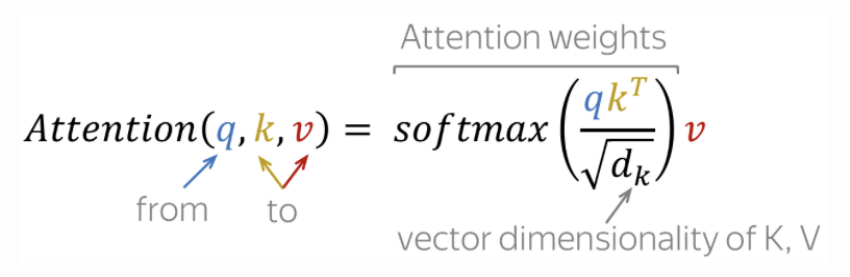
Sequence Length = n, Model/Embedding Dimension = d, number of attention head = h

Input X = (n,d)
W_q, W_k, W_v, W_o = (d, d)

Steps	Operation	Compute Complexity	Memory Complexity
1. QKV Projection	X @ (W_q)	3 n * d^2	3nd
2. Attention Score	Q @ (K^T)	n^2 * d	n^2 * h
3. Softmax+Scale	softmax(scores / √d)	n^2 * h	n^2 * h
4. Weight sum	score @ V	n^2 * d	n * d
5. Output Projection	output @ W_O	n * d^2	n * d

Attention scales quadratically with sequence length.

Self-Attention Step 4-5:



$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Step4 - Softmax:

1. Covert raw score to probability distribution summing to 1: weight sum
2. Sharpen the difference between

For a sequence 3n^2 complexity:

Step 1: exp(z_ij) for each entry → n^2 exponentials
Step 2: sum across each row → n^2 adds
Step 3: divide each entry by row sum → n^2 divisions

Now you have ***Attention Score*** of query of "cat" on the key of all other tokens

Step5 – Weighted Sum: *Attention Score @ V*:

the token **absorbs context: i.e. each token embedding now carries the weight of other tokens that matters to it**

For a sequence: (n×n) @ (nxd) → (n×d) vector

Residual connections and layer normalization

Feed-forward network: after taking information from other tokens, take a moment to think and process this information

Feed-forward network: after taking information from other tokens, take a moment to think and process this information

↑

Encoder self-attention: tokens look at each other

queries, keys, values are computed from encoder states

↑

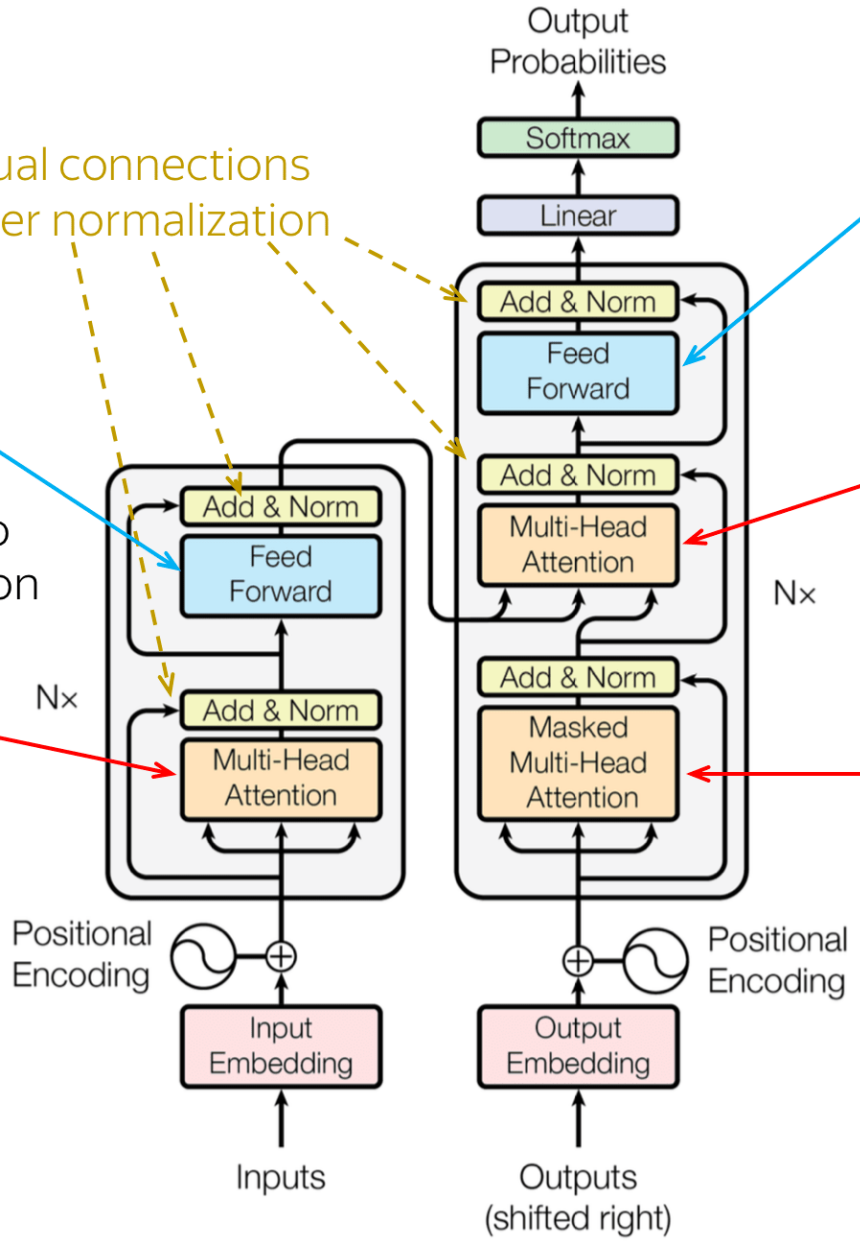
Decoder-encoder attention: target token looks at the source

queries – from decoder states; keys and values from encoder states

↑

Decoder self-attention (masked): tokens look at the previous tokens

queries, keys, values are computed from decoder states



Credit to Lena Voita

Attention Overheads: Compute/Memory Load

Sequence Length = n, Model/Embedding Dimension = d, number of attention head = h

Input X = (n,d)
W_q, W_k, W_v, W_o = (d, d)

Steps	Operation	Compute Complexity	Memory Complexity
1. QKV Projection	X @ (W_q)	3 n * d^2	3nd
2. Attention Score	Q @ (K^T)	n^2 * d	n^2 * h
3. Softmax+Scale	softmax(scores / √d)	n^2 * h	n^2 * h
4. Weight sum	score @ V	n^2 * d	n * d
5. Output Projection	output @ W_O	n * d^2	n * d

Attention scales quadratically with sequence length.

Attention Overheads: KV Cache and Prefill vs Decode

Prefill: process all prompt tokens in parallel in a single forward pass, computing K and V for every prompt position and storing them in the KV cache.

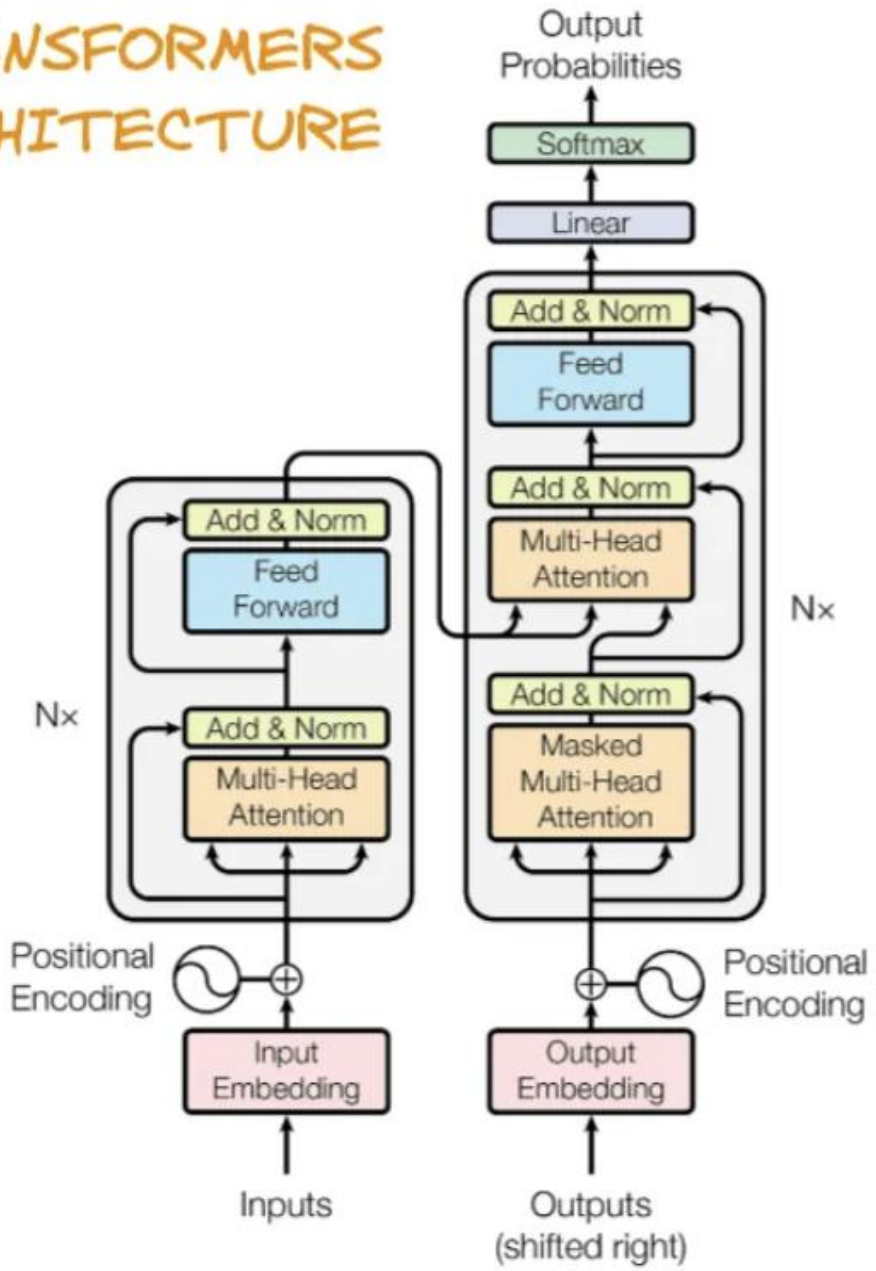
Steps	Operation	Compute Complexity	Memory Complexity
1. QKV Projection	X @ (W_q)	3 n * d^2	3nd (activation)
2. Attention Score	Q @ (K^T)	n^2 * d	n^2 * h (activation)
3. Softmax+Scale	softmax(scores / √d)	n^2 * h	n^2 * h (activation)
4. Weight sum	score @ V	n^2 * d	n * d (activation)
5. Output Projection	output @ W_O	n * d^2	n * d (activation)

Decode: sequential per-token generation, reusing cached K/V from all prior tokens.

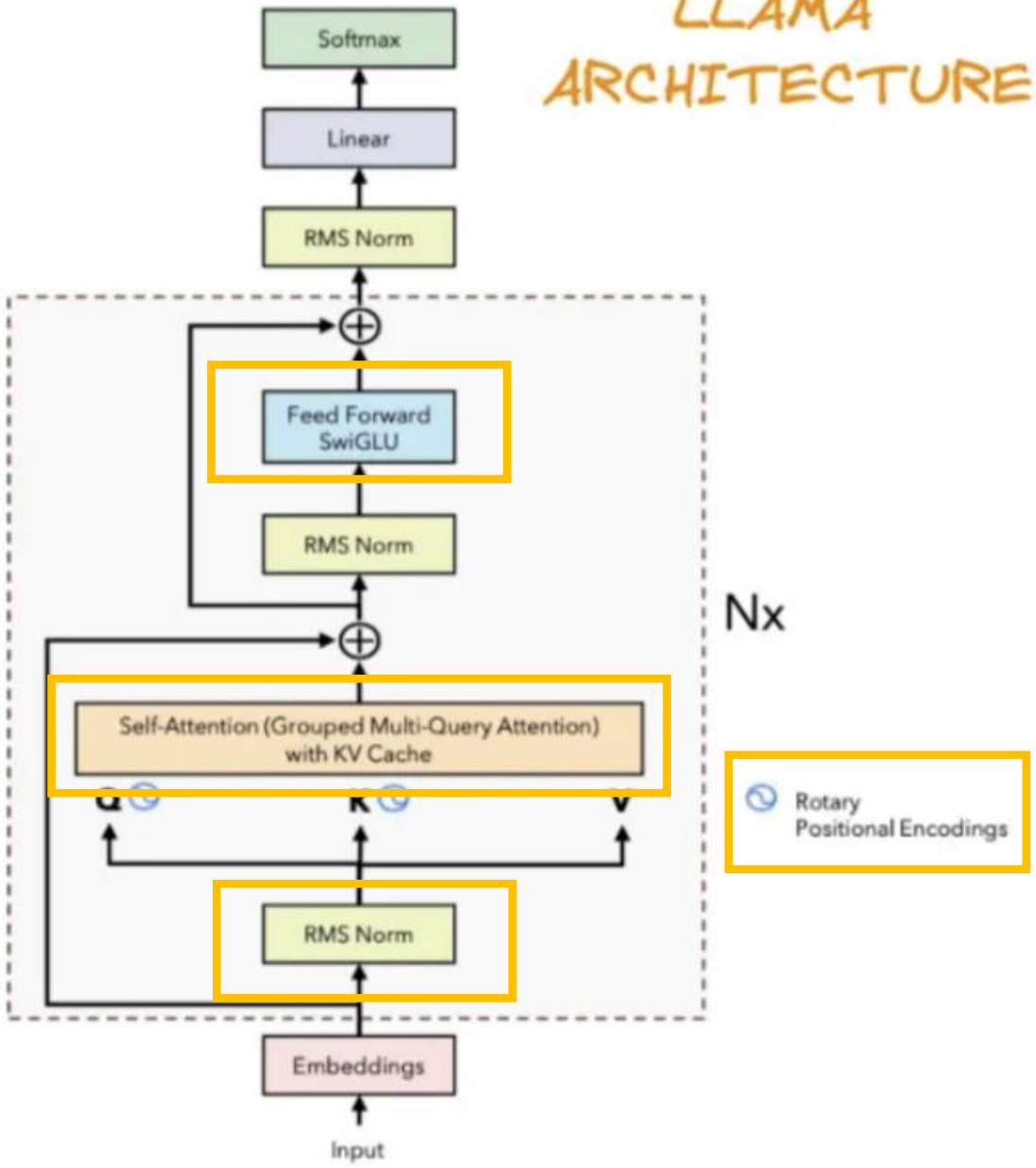
Steps	Operation	Compute Complexity	Memory Complexity
1. QKV Projection	X(1xd) @ (W_q)	3 d^2	3d^2 (weight)
2. Attention Score	Q(1xd) @ (K^T_full)	n_past * d	n_past * d (KV Cache)
3. Softmax+Scale	softmax(scores / √d)	n_past * h	n_past * h (activation)
4. Weight sum	score(1xd) @ V_full	n_past * d	n_past * d (KV Cache)
5. Output Projection	output @ W_O	d^2	d^2 (weight)\

Bottleneck Shifts From Compute(Prefill) to Memory(Decode)!

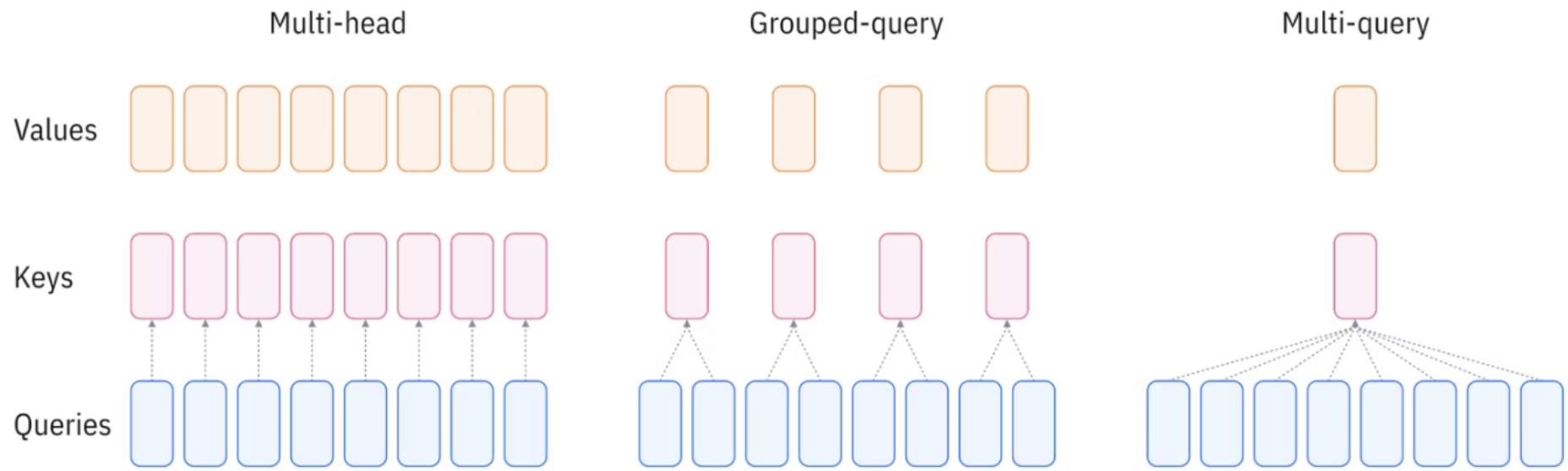
TRANSFORMERS ARCHITECTURE



LLAMA ARCHITECTURE



Grouped Query Attention



The problem: Decode is memory-bandwidth-bound, and the KV cache grows linearly with sequence length.

The insight: Across attention heads, query diversity matters (different heads ask genuinely different questions), but key/value diversity is largely redundant (many heads end up offering similar matchable features and payloads).

The solution: Share K and V across groups of query heads (g query heads per K/V pair) — shrinking the KV cache by factor g, reducing HBM bandwidth per decode step by g, with negligible quality loss.