Lecture 18. Attention.

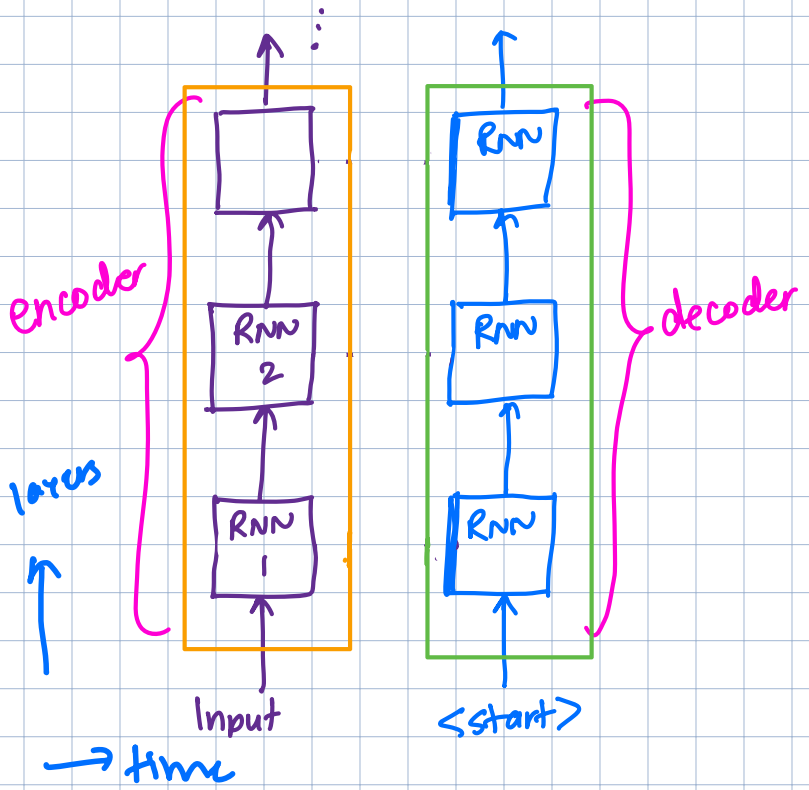
Logistics

- Projects
- Sign up to meet.

Sequence problems using RNNs.

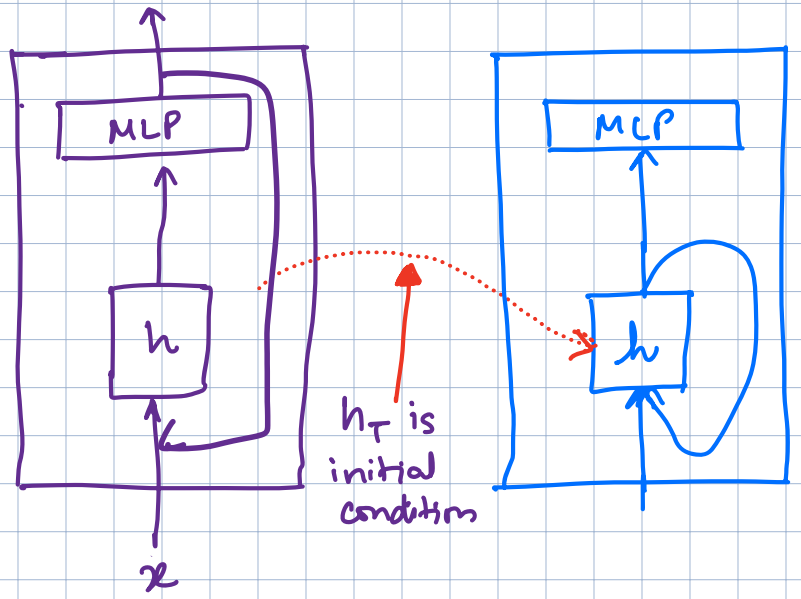
Key example: translation.

RNN style network. : Encoder-Decoder Architecture.



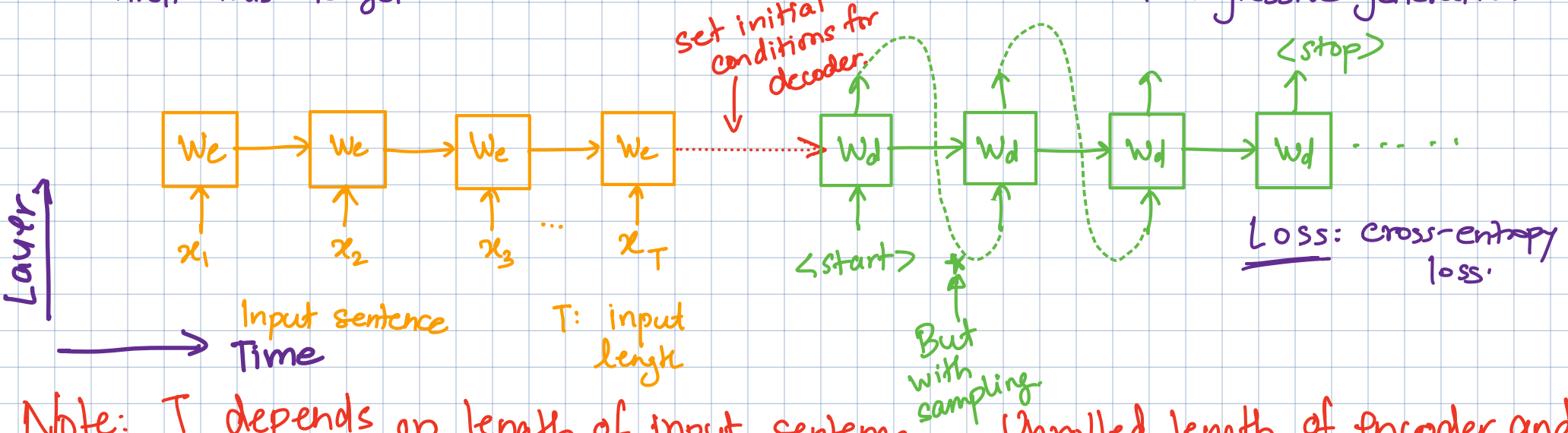
Residual connections not shown.

Each RNN block has inside approximately.



Unroll this to get:

Auto-regressive generation.

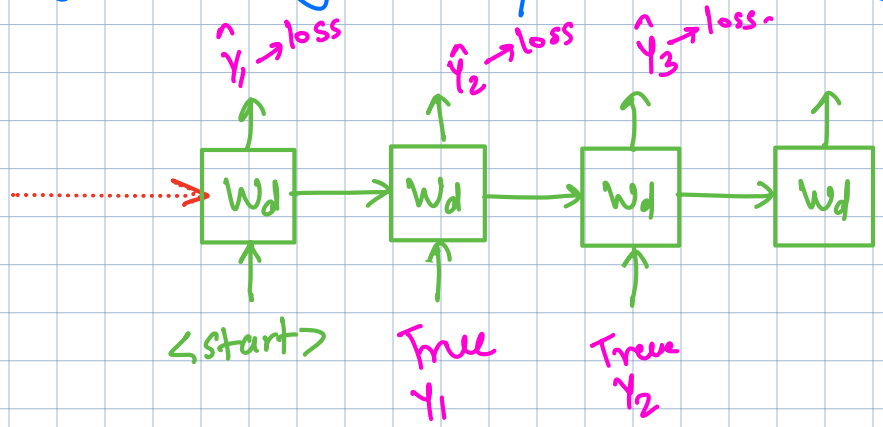


Note: T depends on length of input sentence. Unrolled length of encoder and decoder might be different.

Small bottleneck: problem. ∴

Need to capture all of the input sentence in this bottleneck.

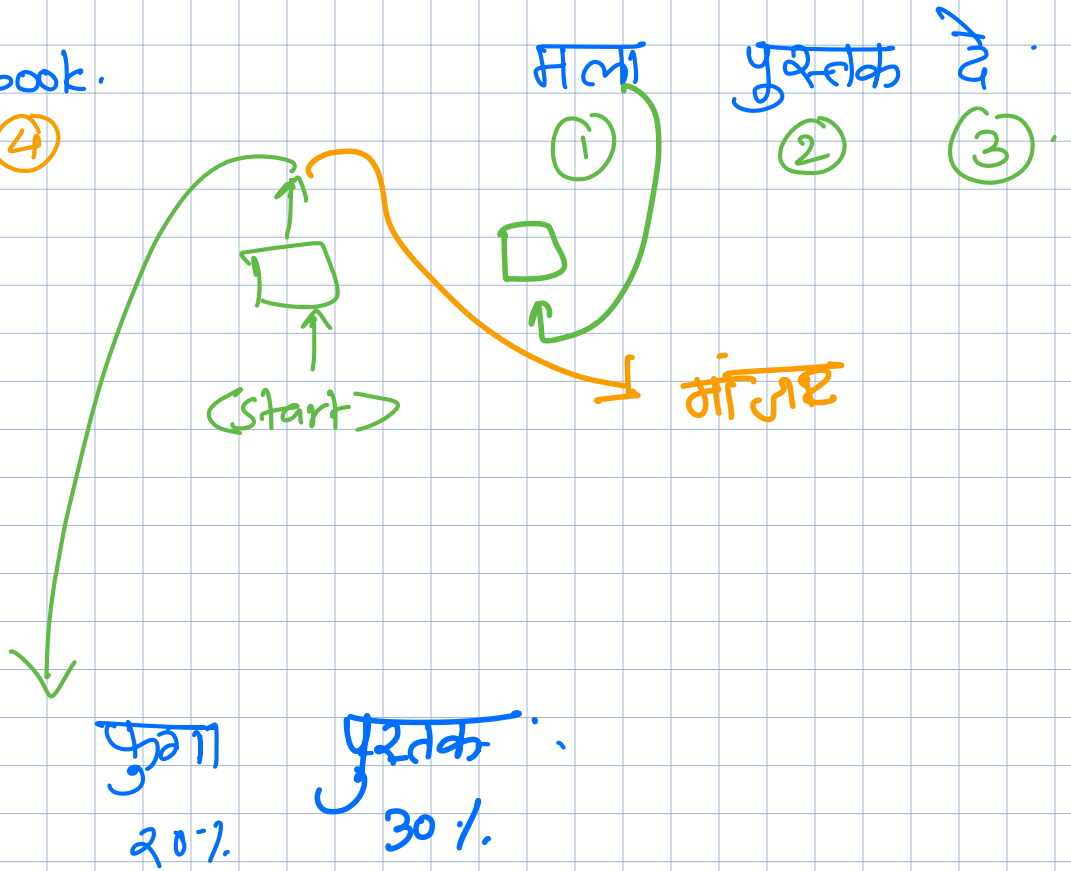
Teacher-forcing: Training technique for auto-regressive generation



Sampling:

Have: Probabilities on output tokens

Give me the book.
① ② ③ ④



Where did we see a bottleneck before?

→ U nets → Image segmentation

→ Want a layer to see appropriate content.

→ How can we bring this trick here?

→ In translation, the topology of expected connections can't be set in advance.

→ So we don't know where to look.

→ Can we use a hash table?

→ Need a learnable, differentiable hash table.

Idea: Query: from decoder

→ look for matching keys that are stored.

→ Fails: initialization is hard.

can't pass useful gradients.

Idea 2: Return the closest match.

→ Bad gradients.

→ Small change doesn't affect the loss function

Idea 3: Look at multiple closest matches.

→ How do I combine different values?

Average: $\sum_{i=1}^{l} v_i$

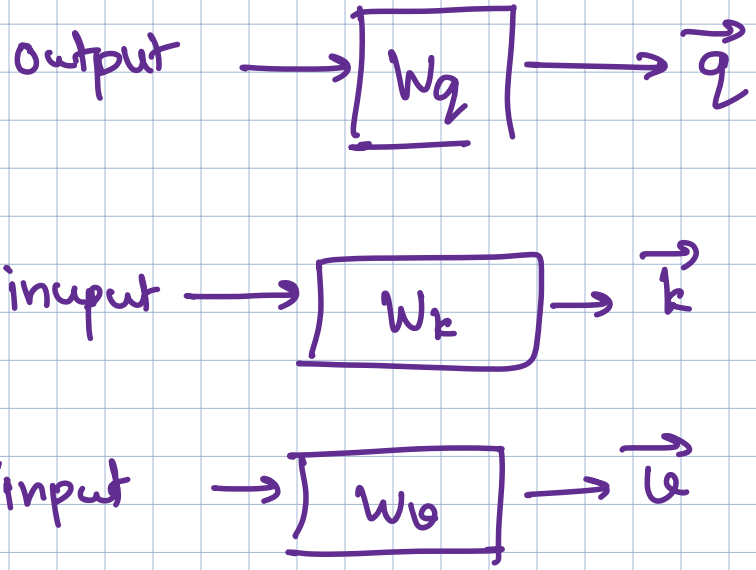
Idea 4: Weighted average.

$$\sum_{i} \text{sim}(\vec{q}, \vec{k}) \cdot \vec{v}$$

→ $\text{sim}(\vec{q}, \vec{k}) = \langle \vec{q}, \vec{k} \rangle = s_i$ "Score"

→ Use softmax to convert score → probability $\frac{e^{s_i}}{\sum e^{s_i}}$

How to get $\vec{q}, \vec{k}, \vec{v}$.



Cross-attention:

Keys + values from encoder, Querys from decoder.

Self-attention.

Keys + values + queries from decoder. (or encoder).

$$\vec{q}_n = \beta_q + W_q \vec{x}_n$$

$$\vec{k}_m = \beta_k + W_k \vec{x}_m$$

$$\vec{v}_l = \beta_v + W_v \cdot \vec{x}_l$$

$\beta_q, \beta_k, \beta_v, W_q, W_k, W_v$ all learnable.

Cross attention:

query: output/decoder.

key: encoder/input

$$a[\vec{x}_m, \vec{x}_n] = \frac{\exp[\langle \vec{k}_m, \vec{q}_n \rangle]}{\sum_{j=1}^{N} \exp[\langle \vec{k}_j, \vec{q}_n \rangle]}$$

key ↗ query ↑

Higher similarity ⇒ higher attention.

Scaled self-attention

$$a[\vec{x}_m, \vec{x}_n] = \frac{\exp\left[\frac{\langle \vec{k}_m, \vec{q}_n \rangle}{\sqrt{D}}\right]}{\sum_{j=1}^{N} \exp\left[\frac{\langle \vec{k}_j, \vec{q}_n \rangle}{\sqrt{D}}\right]} \qquad W_q, W_k \in \mathbb{R}^{D \times N}$$

- Prevents entries to softmax from getting too large.

Output: $\sum a[\vec{x}_m, \vec{x}_n] \cdot v_m$

Self-attention

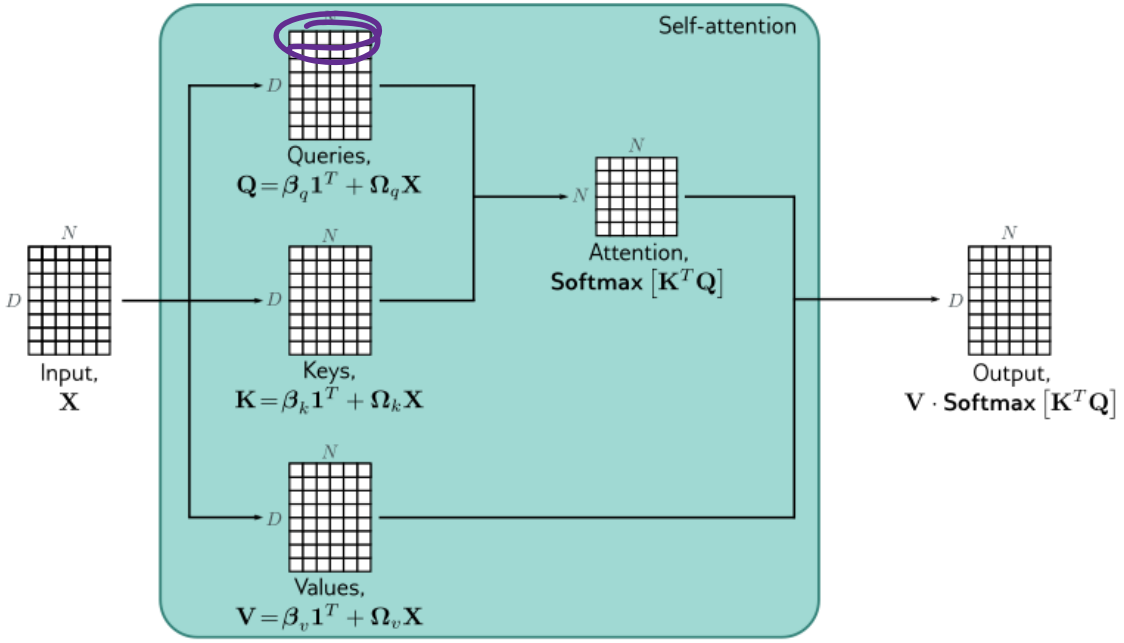


Figure 12.4 Self-attention in matrix form. Self-attention can be implemented efficiently if we store the N input vectors $\mathbf{x}_n$ in the columns of the $D\times N$ matrix $\mathbf{X}$. The input $\mathbf{X}$ is operated on separately by the query matrix $\mathbf{Q}$, key matrix $\mathbf{K}$, and value matrix $\mathbf{V}$. The dot products are then computed using matrix multiplication, and a softmax operation is applied independently to each column of the resulting matrix to calculate the attentions. Finally, the values are post-multiplied by the attentions to create an output of the same size as the input.

Attention as routing

Fig 12.1

Multi-head self attention.

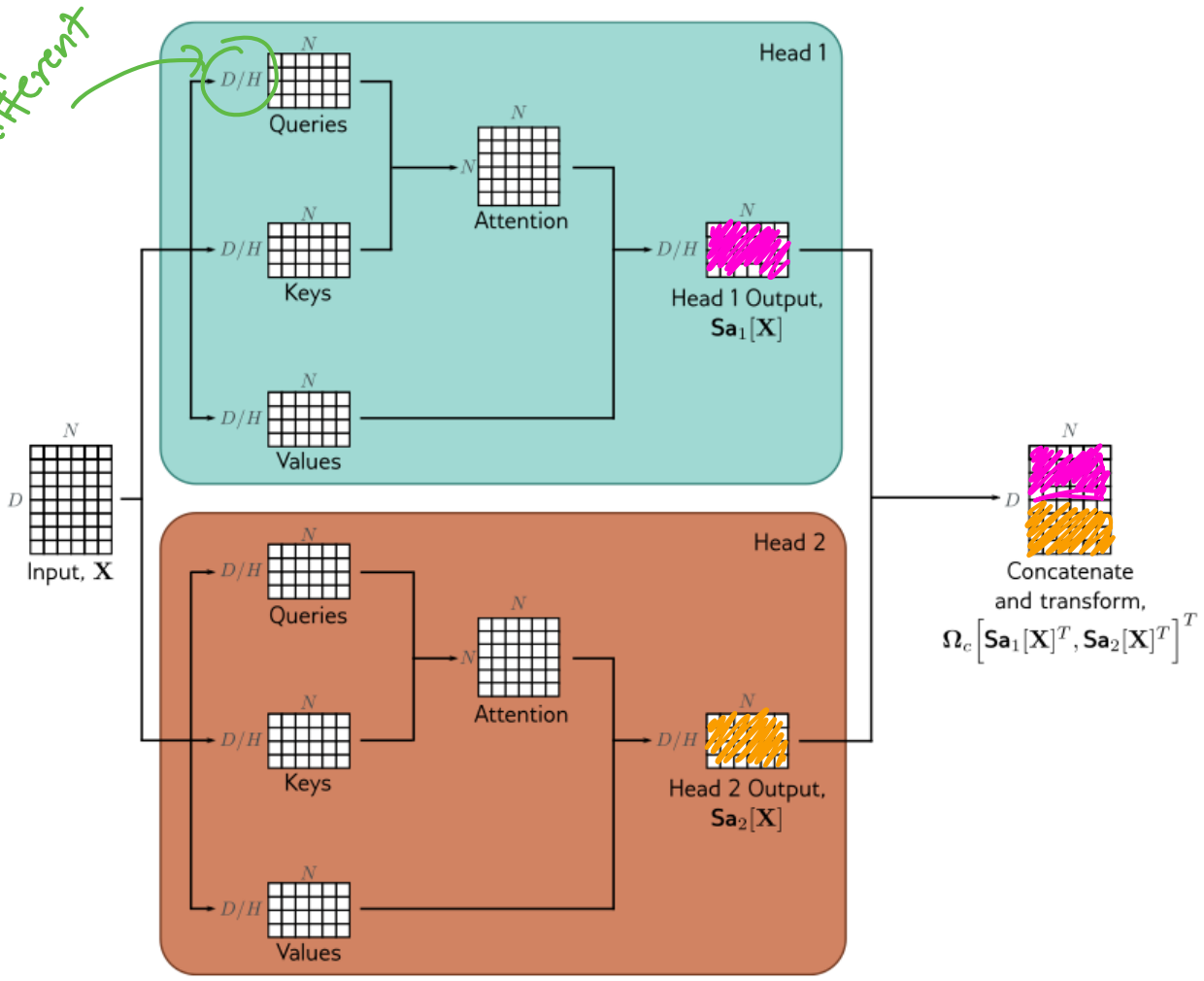


Figure 12.6 Multi-head self-attention. Self-attention occurs in parallel across multiple "heads." Each has its own queries, keys, and values. Here two heads are depicted, in the cyan and orange boxes, respectively. The outputs are vertically concatenated, and another linear transformation $\boldsymbol{\Omega}_c$ is used to recombine them.

Multi-query attention: Use the same keys and values throughout

Change the query for each head.

Saves on memory.

Muti-query group attention: Have groups of keys and values instead.