Lecture 18. Attention.

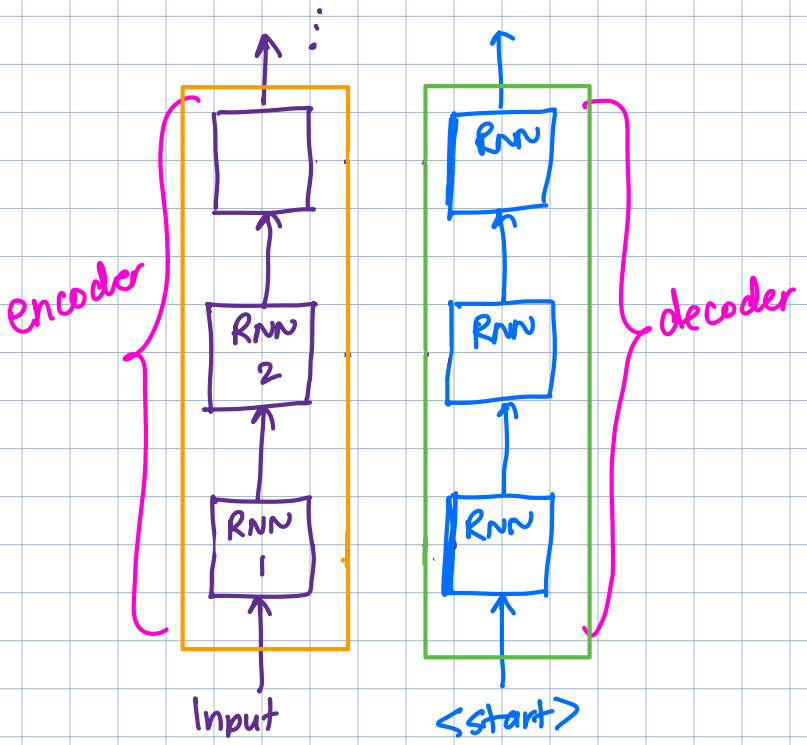
Logistics

- Projects
- Sign up to meet.

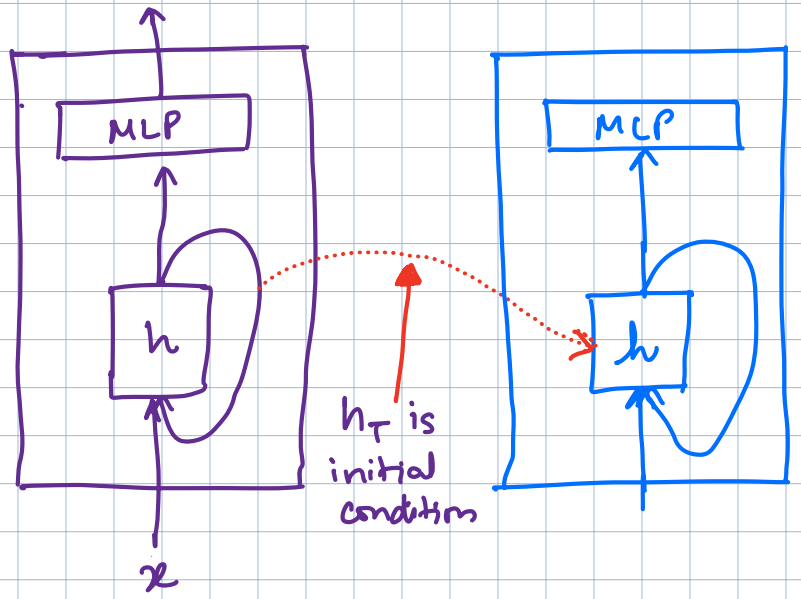
Sequence problems using RNNs.

Key example: translation.

RNN style network. : Encoder-Decoder Architecture.



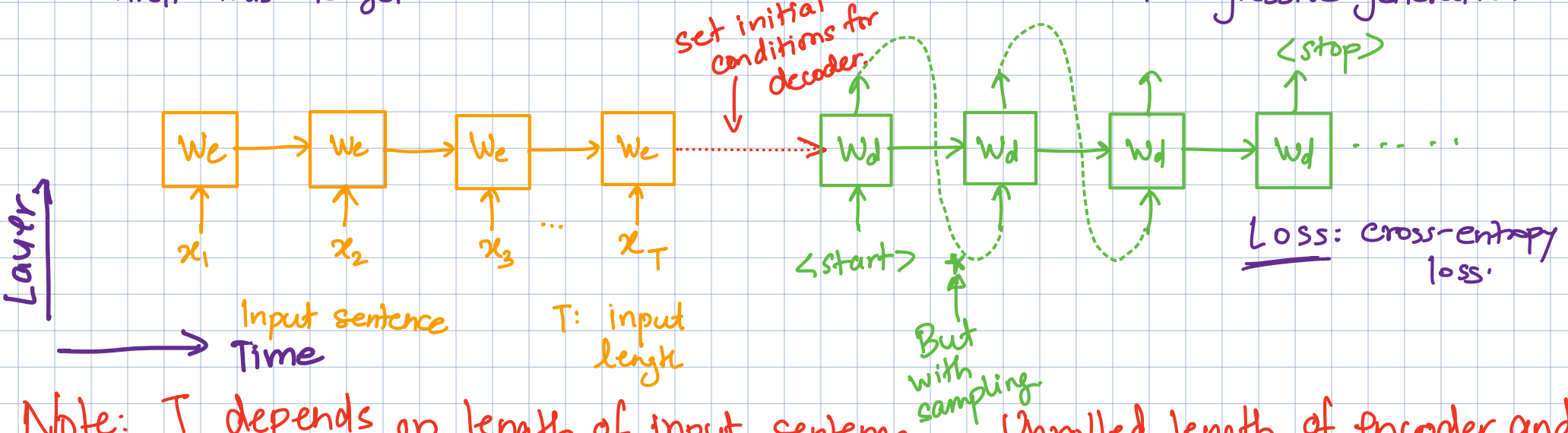
Each RNN block has inside approximately.



Residual connections not shown.

Unroll this to get:

Auto-regressive generation.

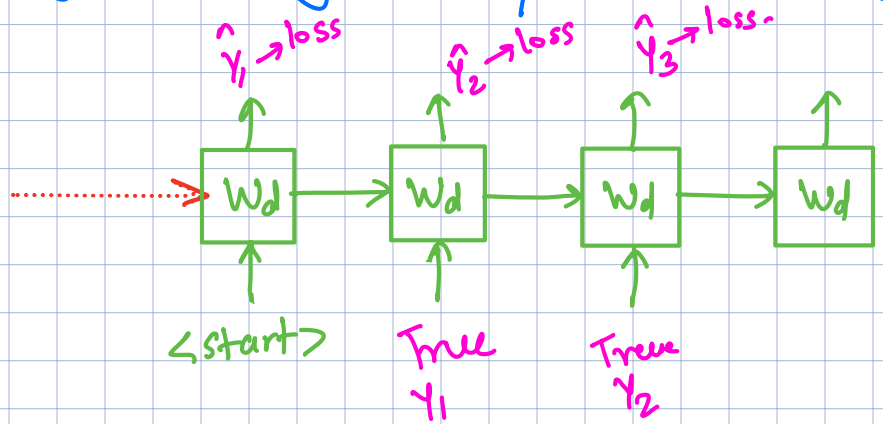


Note: T depends on length of input sentence. Unrolled length of encoder and decoder might be different.

Small bottleneck: problem. ∴

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

Teacher-forcing: Training technique for auto-regressive generation

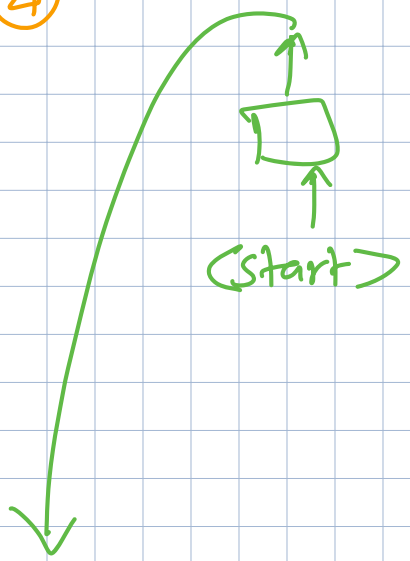


Sampling:

Have: Probabilities on output tokens

Give me the book.
① ② ③ ④

मला पुस्तक दे.
① ② ③



मला 50-%.

फुला 20-%.

- - -

Where did we see a bottleneck before? U-nets!

Solution for U-nets: Add connections back to the appropriate positions! Give the pixel appropriate "context".

Translation / Language-challenge: Words change position!

Challenge: Cannot set a topology of connections in advance!

→ Solution: Hash table.

→ ↳ need this to be differentiable.

Idea-1: Look for the exact match of query q among the stored keys, and then return value corresponding to exact match.

Problem: Never have exact match. Initialization doesn't work.

Idea 0: Return the closest match.

Problem: Bad gradients – like a piecewise constant.
Small changes to the query will not change closest match

Idea 1: look at the top l - matches.

l different values.

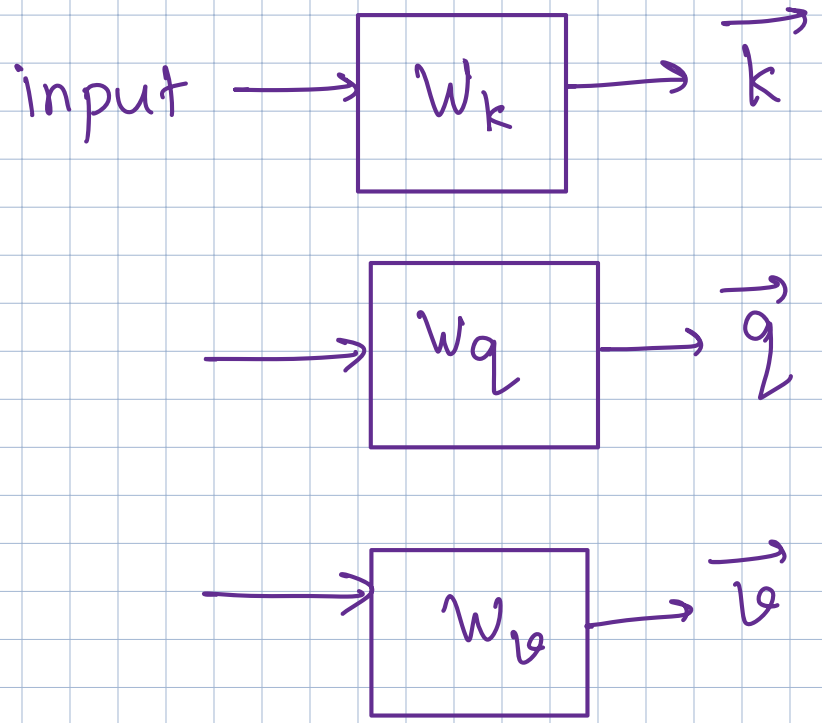
Try: $\frac{1}{l}\sum_{i=1}^{l} \vec{v_i}$

Next try: Weighted average. $\frac{1}{l}\sum_{i=1}^{l} \alpha_i \vec{v_i}$ (scalar)

$$\frac{1}{l}\sum_{i} \underbrace{sim(\vec{q}, \vec{k_i})}_{\text{inner product}} \vec{v_i}$$

→ softmax.

How do we get $\vec{q}, \vec{k}, \vec{v}$? Learn them!



Cross-attention:

Keys + values from encoder, Querys from decoder.

Self-attention.

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

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

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

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

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

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

✗ Doesn't work.

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.

Self-attention

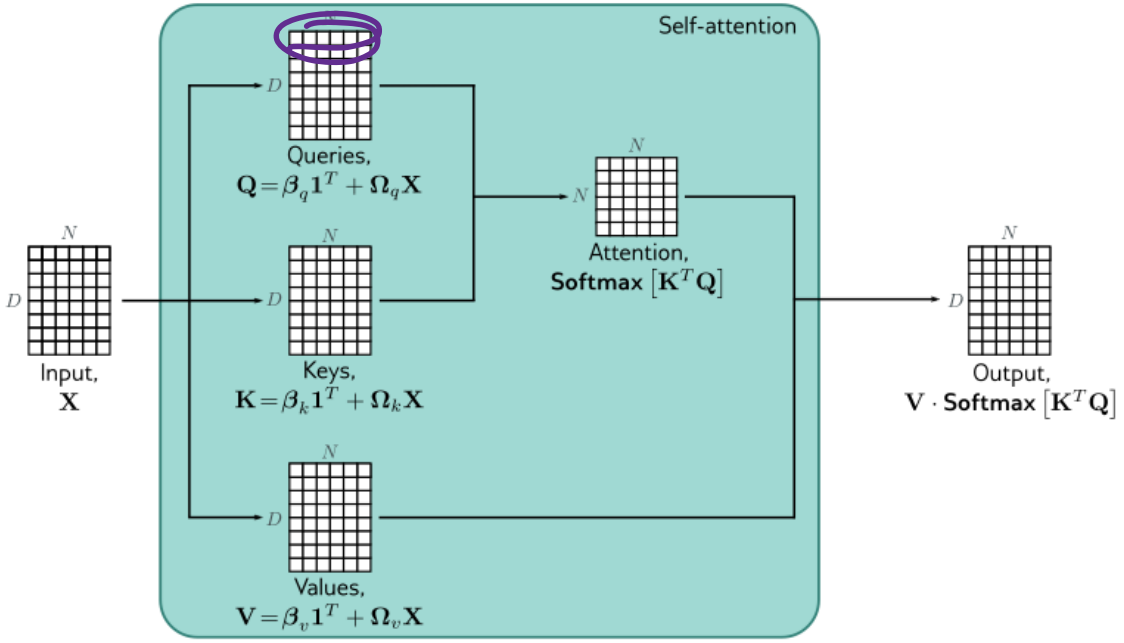


Figure 12.4 Self-attention in matrix form. Self-attention can be implemented efficiently if we store the N input vectors $\mathbf{x}_n$ in the columns of the $D\times N$ matrix $\mathbf{X}$. The input $\mathbf{X}$ is operated on separately by the query matrix $\mathbf{Q}$, key matrix $\mathbf{K}$, and 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.

Multi-head self attention.

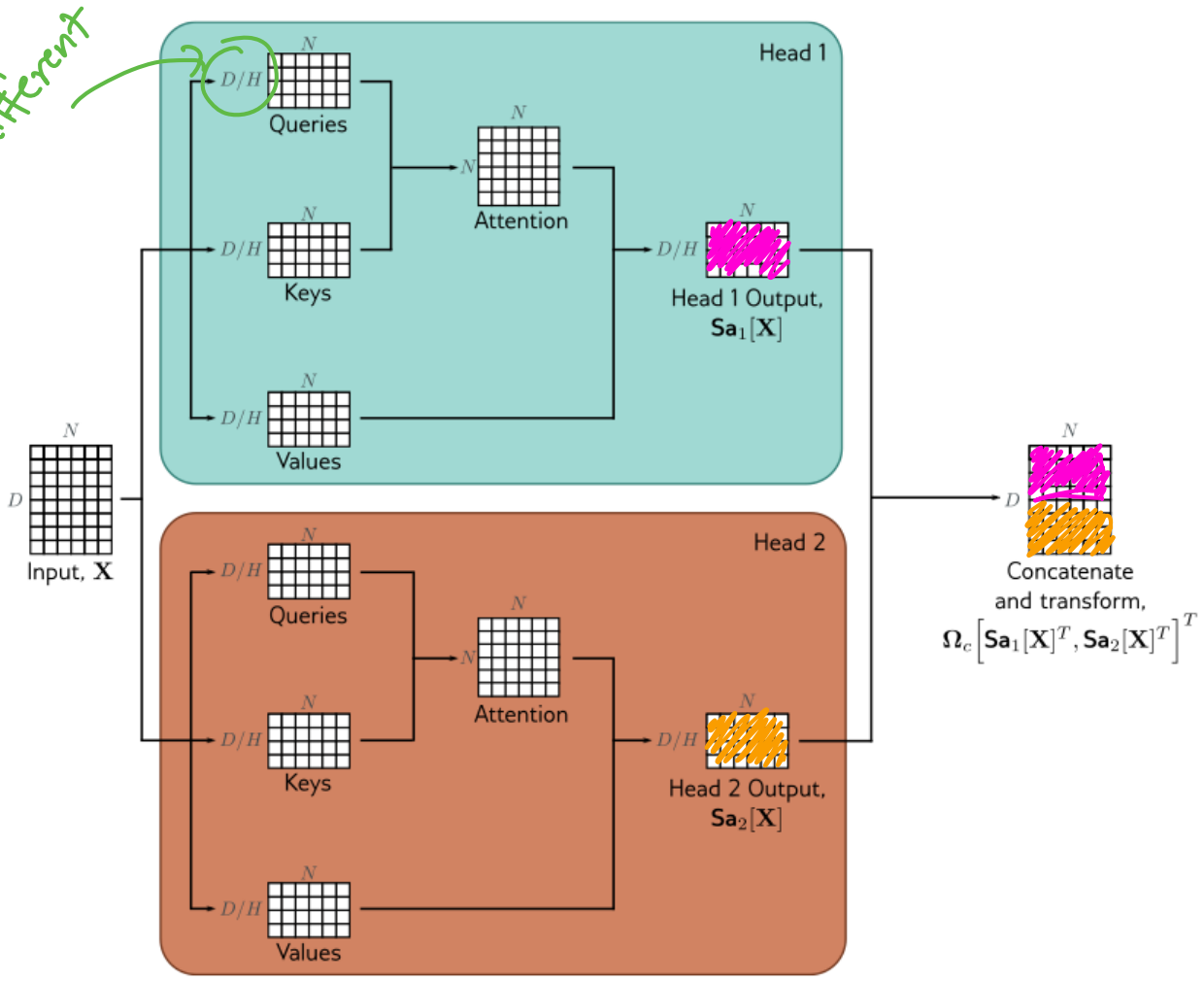


Figure 12.6 Multi-head self-attention. Self-attention occurs in parallel across multiple "heads." Each has its own queries, keys, and values. Here two heads are depicted, in the cyan and orange boxes, respectively. The outputs are vertically concatenated, and another linear transformation $\boldsymbol{\Omega}_c$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