Yale

Transformers and GNNs

CPSC483: Deep Learning on Graph-Structured Data

Rex Ying

Readings

- Readings are updated on the website (syllabus page)
- **Lecture 7 readings**:
  - Graph Attention Networks
  - Multi-hop Attention Graph Neural Networks
- **Lecture 8 readings:**
  - Attention is All You Need
  - Graph Structure of Neural Networks

Outline of Today's Lecture

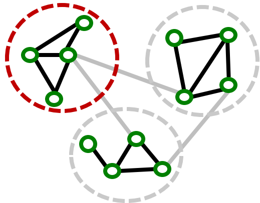
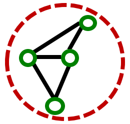
1. **Self-Attention and Transformers**

2. Transformers Applications

3. Graph Transformers and Sparse Transformers

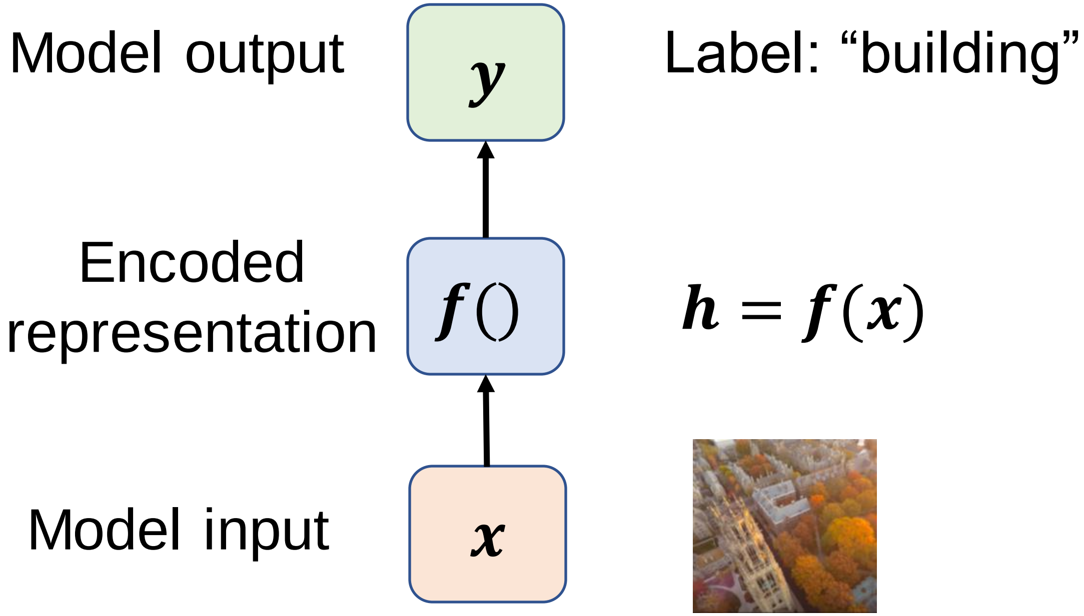
Sequence Learning

- Inputs from different domains can be seen as the general **sequence** of **tokens**

Domain	Sequence	Token	Structure
NLP	**Sentence:** [SOS, “graph”, “neural”, “networks”, “are”, “powerful”, EOS]	**Word:** “graph” **Phrase:** [“graph”, “neural”, “networks”]	Sequential correlations
CV	**Image:** 	**Pixel** **Patch:** 	Spatial correlations
Graph	**Graph:** 	**Node:**  **Subgraph:** 	Adjacency

Standard Supervised Learning Setting

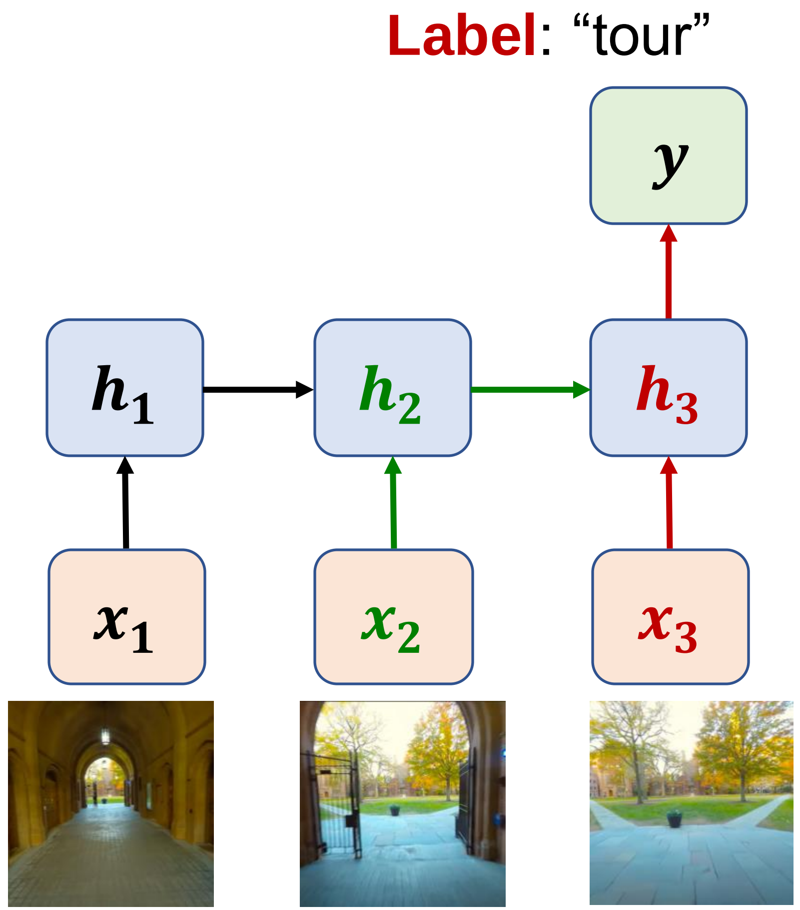
- **One (token) to One (token)**
  - Input is a single token (e.g., an image), and the output is its attribute (e.g., label) or another token.
  - $\boldsymbol{h} = \boldsymbol{f}(\boldsymbol{x})$, $\boldsymbol{f}()$ is the model to learn.



Sequence Learning — Application

- **Many (tokens) to One**
  - Input is a sequence of tokens (e.g. a video with frames), and the output is its attribute (e.g. label) or another token.
  - $\boldsymbol{h_1} = \boldsymbol{f}(\boldsymbol{x_1})$
  - To generate $\boldsymbol{h_2}$, we would like to incorporate both $\boldsymbol{x_2}$ and the preceding frame $\boldsymbol{x_1}$ $\boldsymbol{h_2} = \boldsymbol{f}(\boldsymbol{x_2}, \boldsymbol{h_1})$, and $\boldsymbol{f}()$ is still the same model (shared parameters)
  - $\boldsymbol{h_i} = \boldsymbol{f}(\boldsymbol{x_i}, \boldsymbol{h_{i-1}})$

Current token **Previous token**



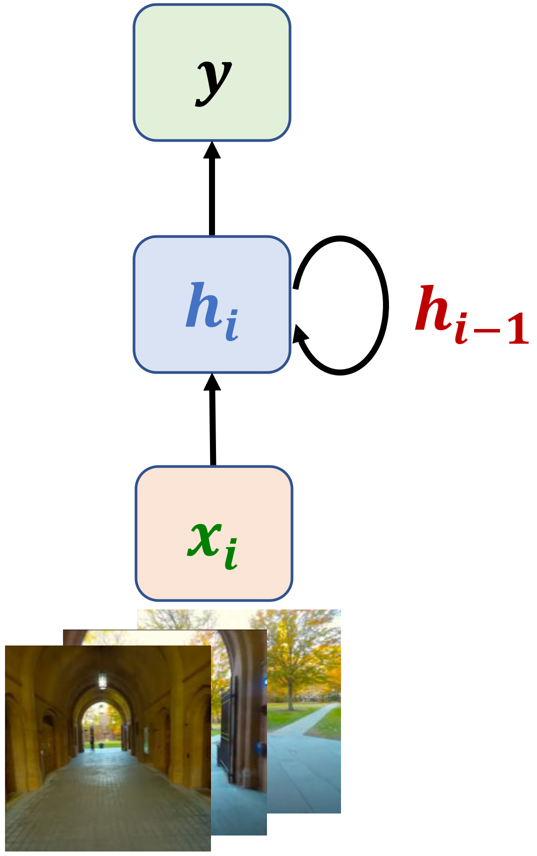
Sequence Learning — Application

- We can process a sequence of tokens $\boldsymbol{X} = [\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n]$ by applying a recurrence formula at every time step
- **Recurrent neural networks**

$$\boldsymbol{h}_i = \boldsymbol{f}_W(\boldsymbol{x}_i, \boldsymbol{h}_{i-1})$$

new state | current input | old state

- For example, $\boldsymbol{h}_i = \boldsymbol{\sigma}(\boldsymbol{W}_x \boldsymbol{x}_i + \boldsymbol{W}_h \boldsymbol{h}_{i-1} + \boldsymbol{b}_h)$, and $\boldsymbol{y}_i = \boldsymbol{\sigma}(\boldsymbol{W}_y \boldsymbol{h}_i + \boldsymbol{b}_y)$



A folded diagram of RNNs

Sequence Learning — Application

- **Many (tokens) to Many**
  - The sequence is first encoded into a hidden representation, then gradually decoded by the decoder.

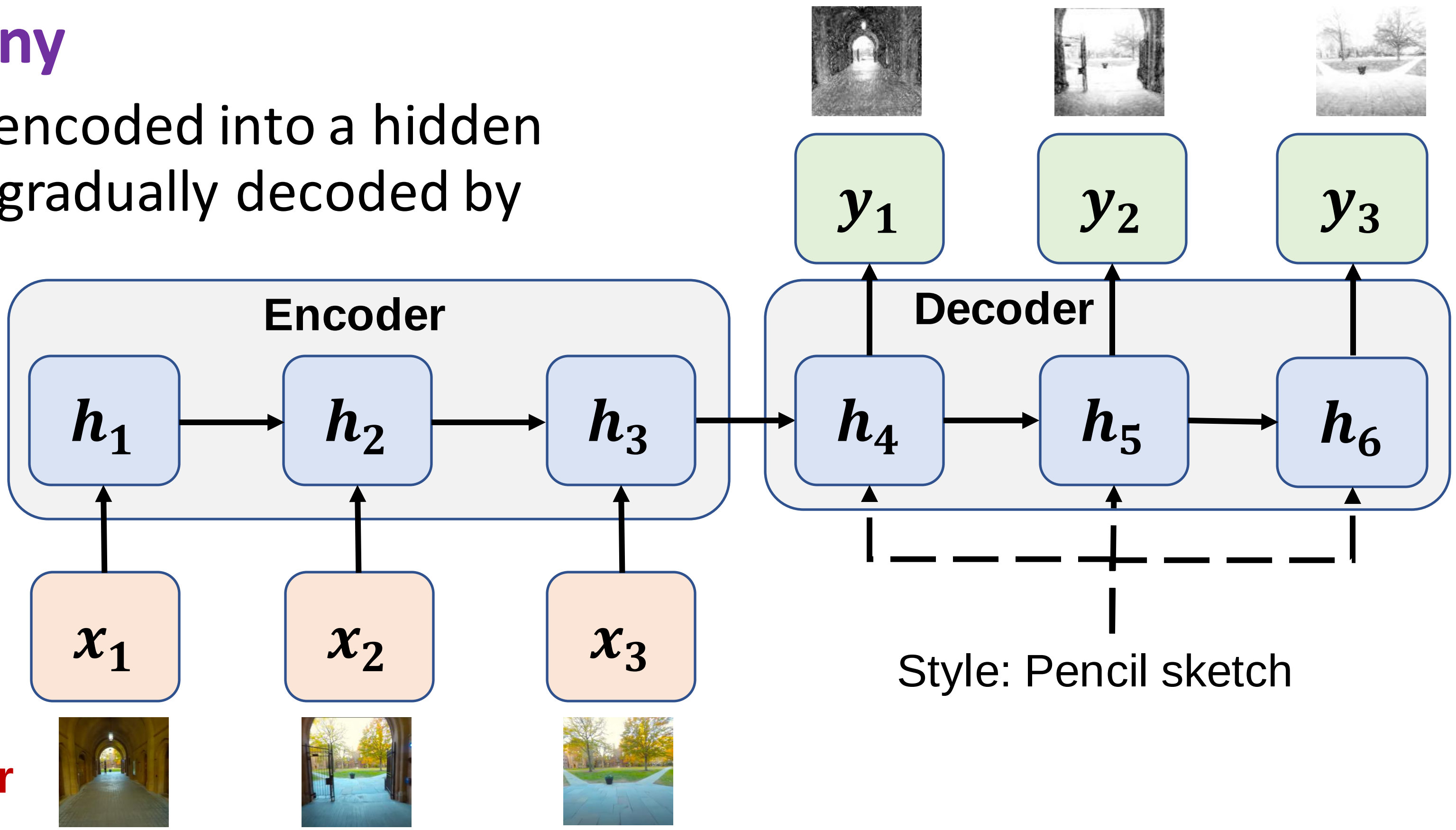
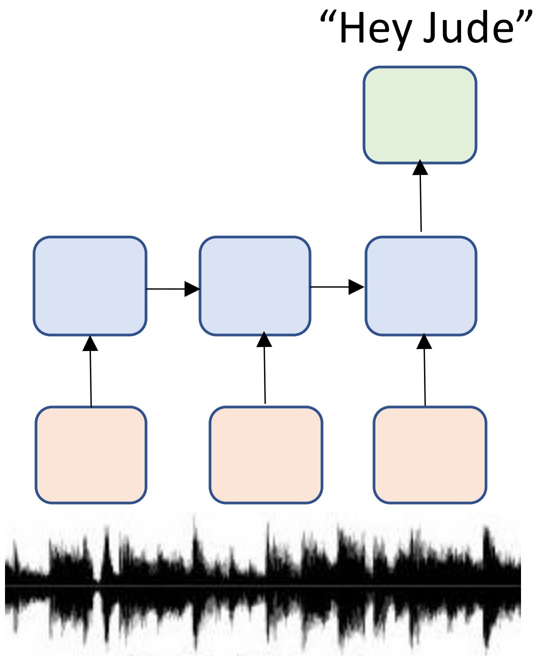


Diagram of video **style transfer**

Sequence Learning — Application

- **One (token) to One (token)**
- **Many to One**
  - Protein to property
  - Sentence to sentiment
  - Song to name
- **One to Many**
  - Image to caption
- **Many to many**
  - English to Chinese
  - Time series: history to future
  - Graph autoregression



Speech classification
(many-to-one)

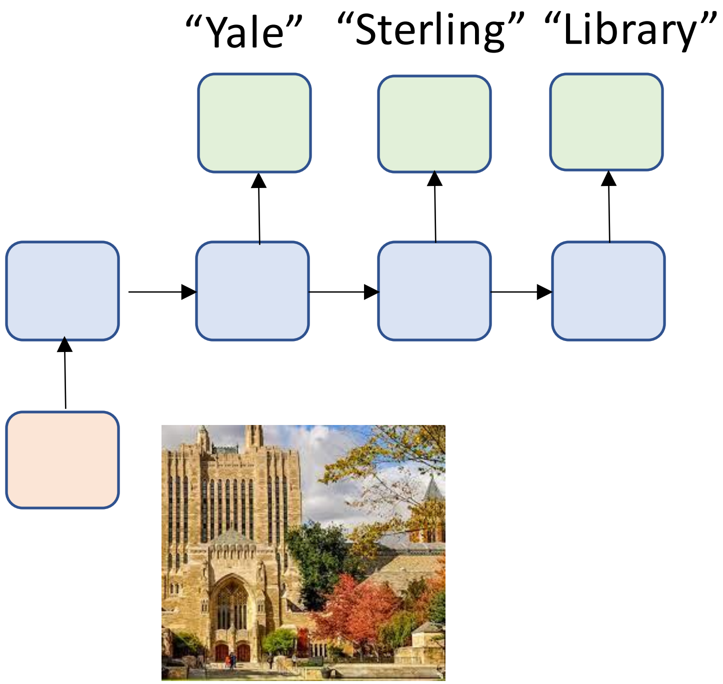
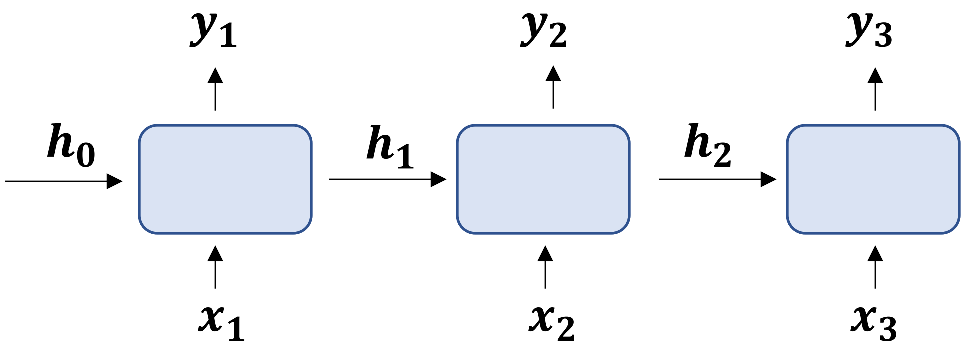


Image Captioning
(one-to-many)

Sequence Learning

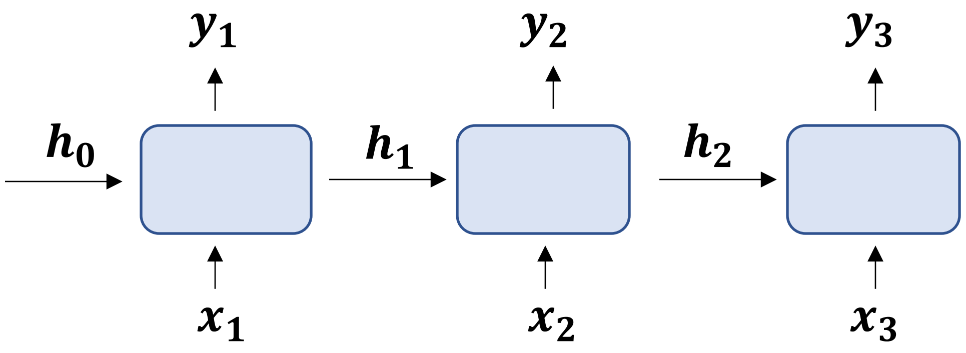


$$h_i = f_W(x_i, h_{i-1}),\ y_i = f_Y(h_i)$$

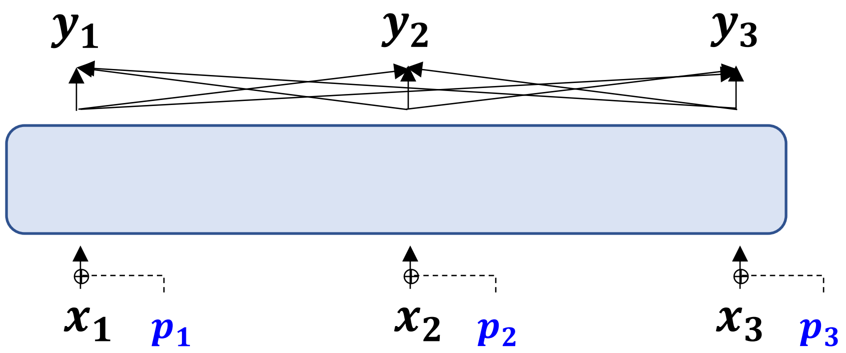
Problems of RNNs

- Sequential computation prevents parallelization
- Capacity of handling long sequences
- Mainly focusing on modeling recurrence
    - does not capture other correlations (hierarchical, long-range, polysemy….) well

Sequence Learning



$$h_i = f_W\,(x_i\,, h_{i-1}),\ y_i = f_Y\,(h_i)$$



$$y_i = \text{self-att}([x_i + p_i]_i)$$

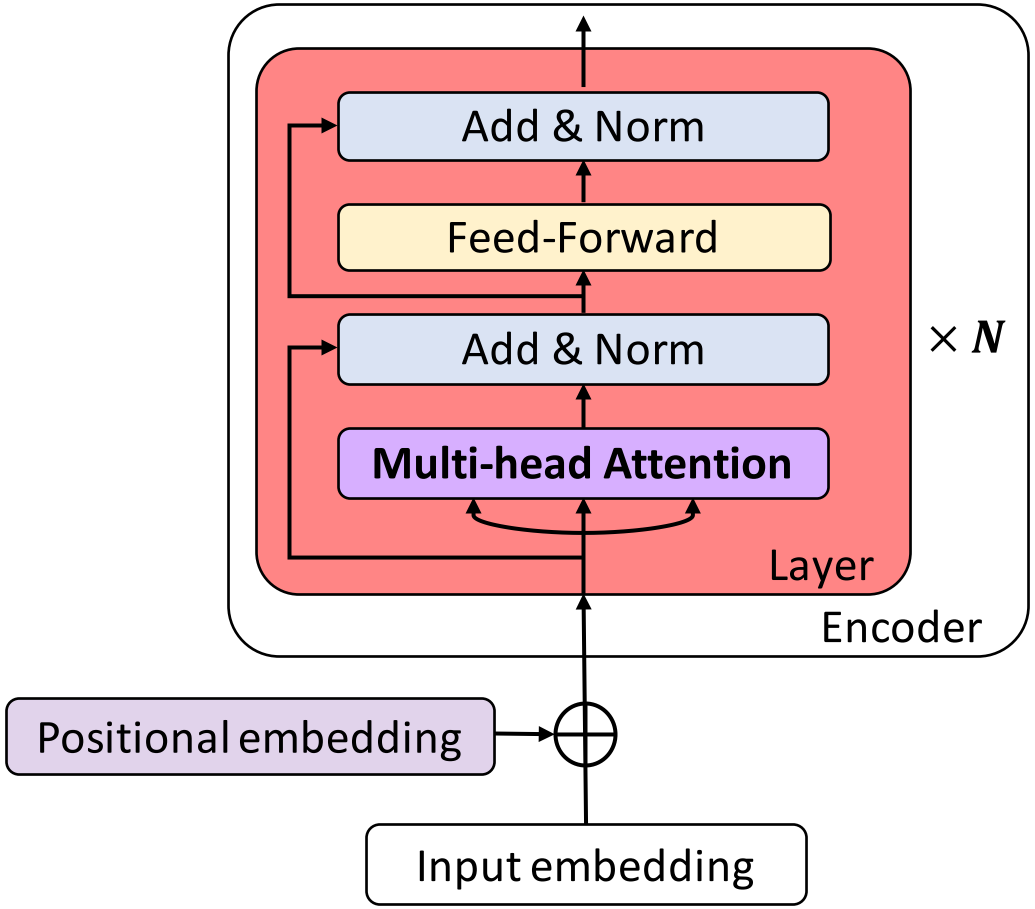
Problems of RNNs

1. parallelization
2. long sequences
3. only recurrence

Solutions by Transformers

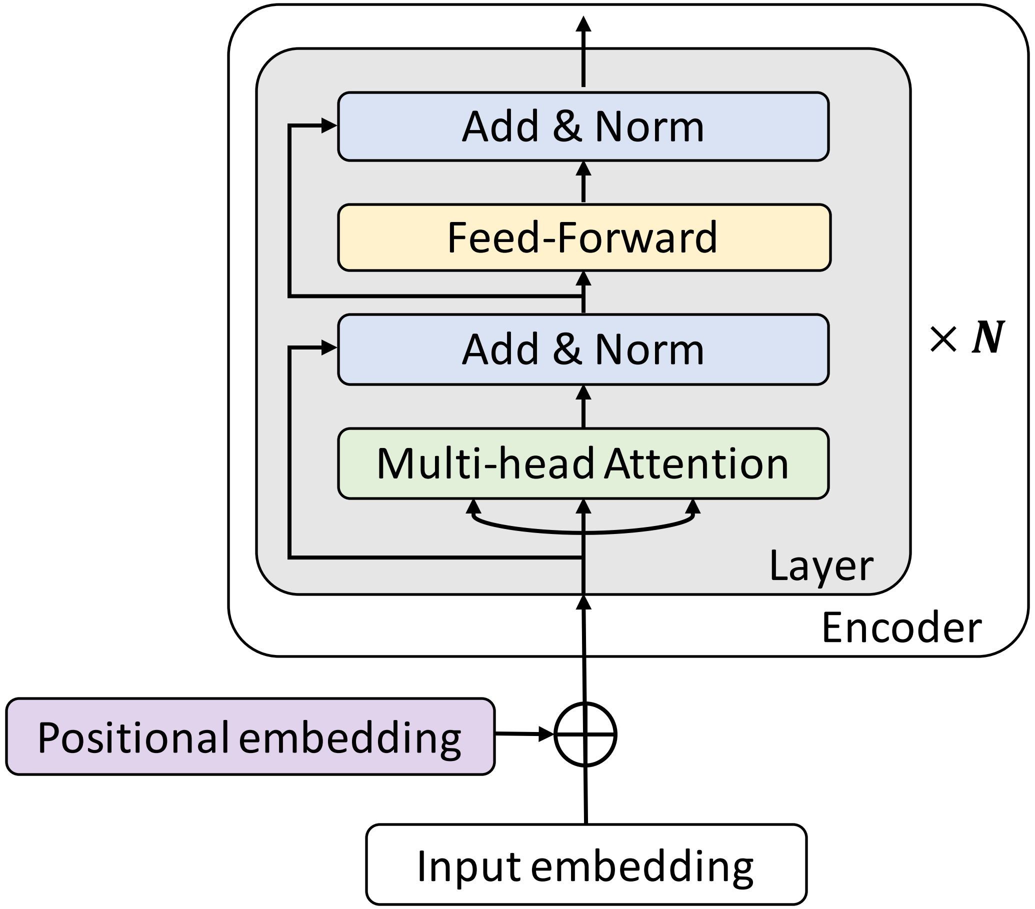
1. **Parallel input**: Input All tokens at the same time
2. **Self-Attention**: Enable attention in long-range
3. **Positional Embeddings** p_i: Model all possible correlations

Transformers — Overview



- **Original paper**: Attention is all you need [Vaswani et al., 2019].
- **Key component**: multi-head self-attention
- **Transformer layer**: Multi-Head Self-Attention, layer normalization, skip connection, position-wise FFN
- **Model usage**: Pretraining then fine-tune, with Encoder only/ Encoder-Encoder/ Decoder only (discussing later)

Transformers — Overview



- **Design choices** of transformers:
  (for those interested in transformer architectures)

Absolute/relative position, equivariant embedding (for graph)

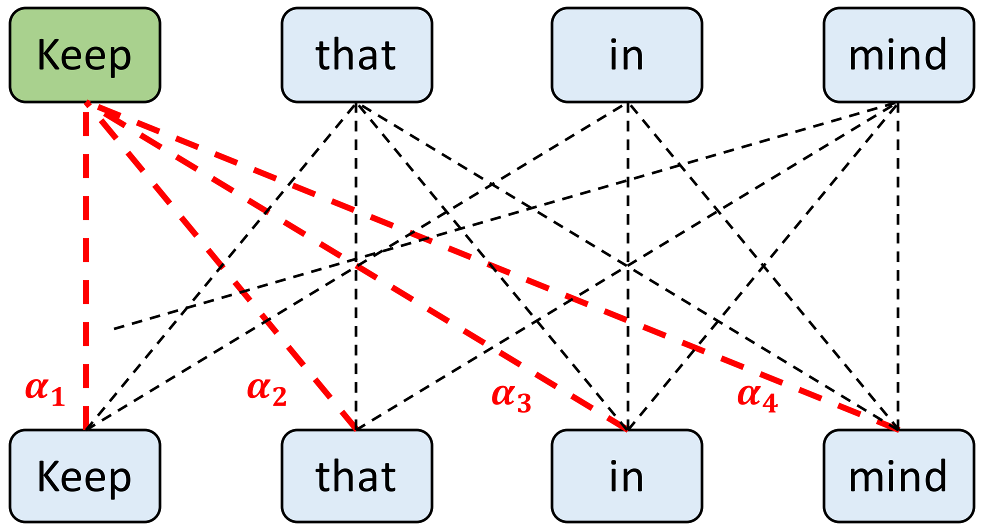
Sparse attention, low-rank attention, attention with prior, memory compression, query prototyping…

Placement, substitutes, normalization-free

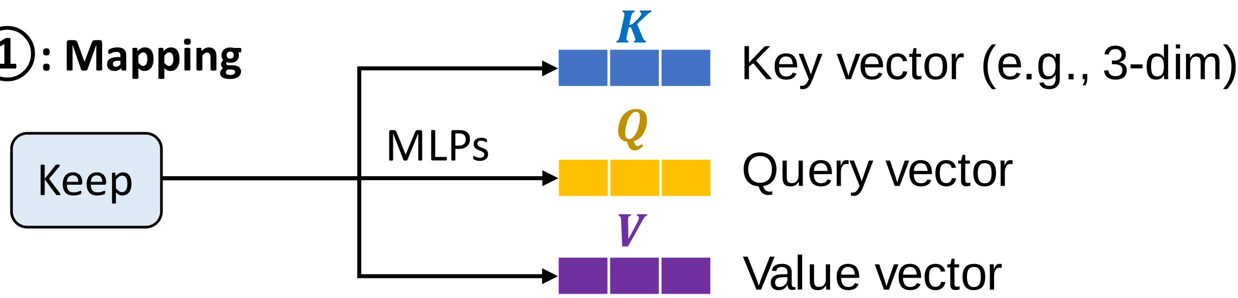
Cross-block connections, recurrence/hierarchy, other architecture

Transformers — Self-Attention (1/5)

Example:



Step ①: Mapping

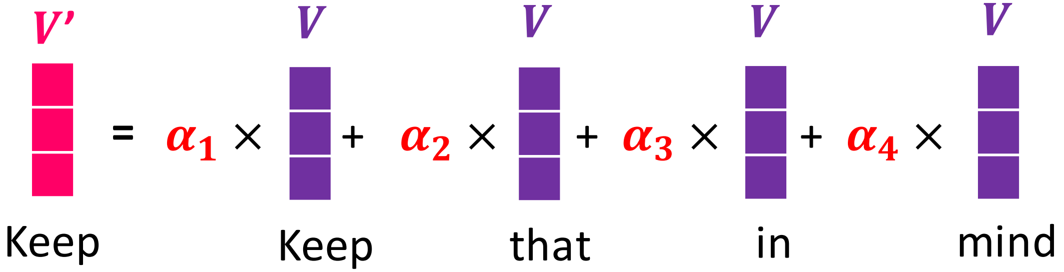


Step ②: Attention

$\alpha_1, \alpha_2, \alpha_3, \alpha_4$ = **Softmax** (Q × K K K K)

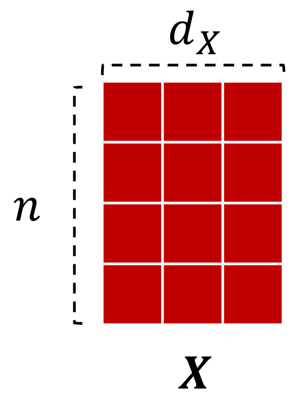
Keep · Keep that in mind

Step ③: Update

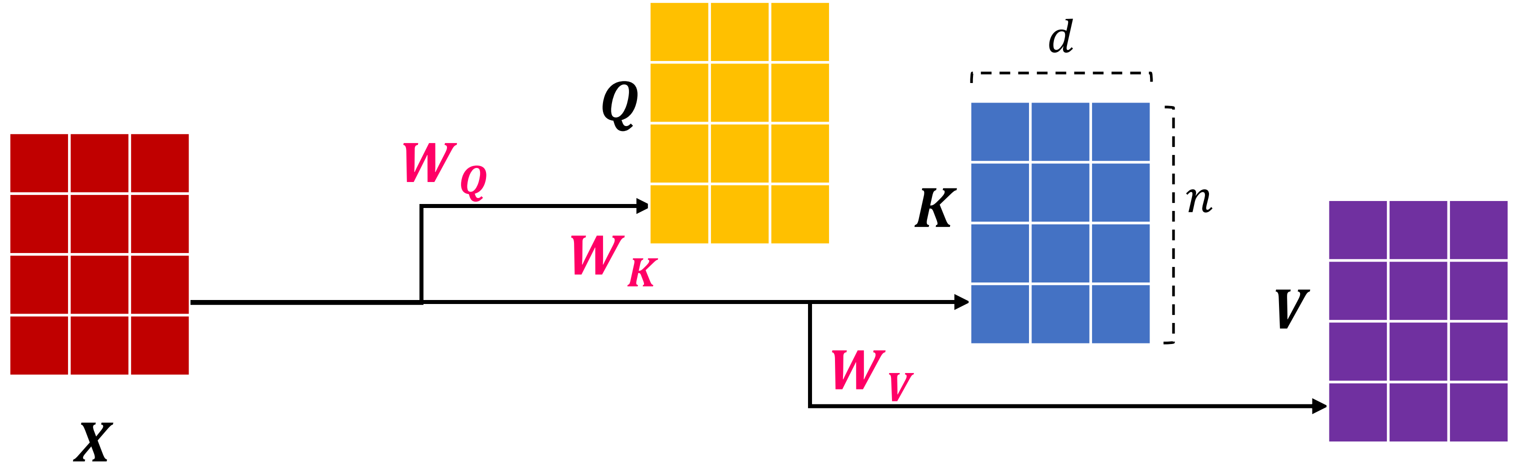


Transformers — Self-Attention (2/5)

- Formally, given an input sequence $\boldsymbol{X} = [\boldsymbol{x_1}, \boldsymbol{x_2}, \dots, \boldsymbol{x_n}] \in \mathbb{R}^{n \times d_X}$

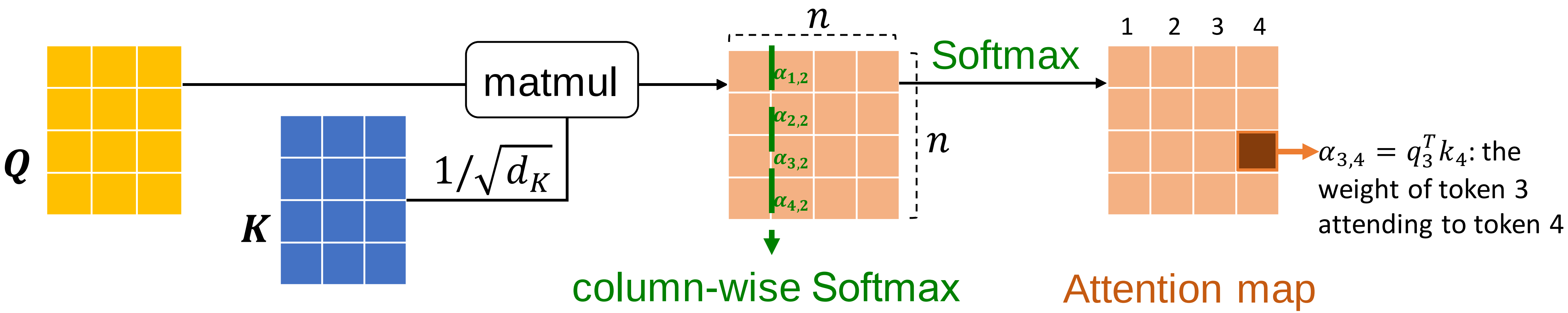


- Step ①: Query $\boldsymbol{Q} = \boldsymbol{XW_Q}$, Key $\boldsymbol{K} = \boldsymbol{XW_K}$, Value $\boldsymbol{V} = \boldsymbol{XW_V}$
  - $W_K \in \mathbb{R}^{d_X \times d_K}$, and thus $\boldsymbol{K} \in \mathbb{R}^{n \times d_K}$
  - We require $d_K = d_Q$, for simplicity, we set $d_K = d_Q = d_V \coloneqq d$



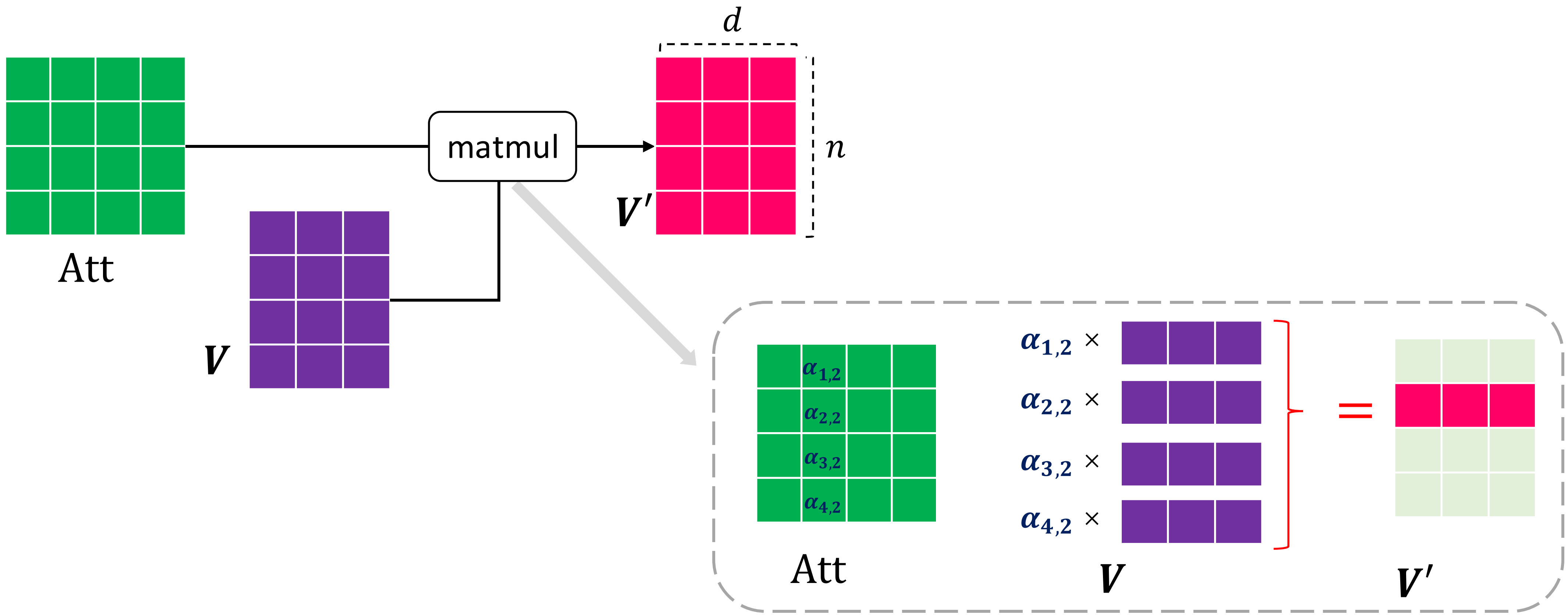
Transformers — Self-Attention (3/5)

- Step ② : Attention map $\text{Att} = \text{Softmax}\left(\frac{\boldsymbol{QK}^T}{\sqrt{d}}\right) \in \mathbb{R}^{n \times n}$ (Softmax is col-wise)
  - The matrix multiplication $\boldsymbol{QK}^T$ performs dot-product for every possible pair of queries and keys, resulting in an attention map.
  - **Normalization factor** $1/\sqrt{d_K}$: performing dot-product over two vectors with variance σ^2 results in a scalar having d_K-times higher variance,
    - $q \sim N(0, \sigma^2), k \sim N(0, \sigma^2) \rightarrow \text{Var}\left(\sum_{i=1}^{d_K} q[i]k[i]\right) = \sigma^4 d_K$



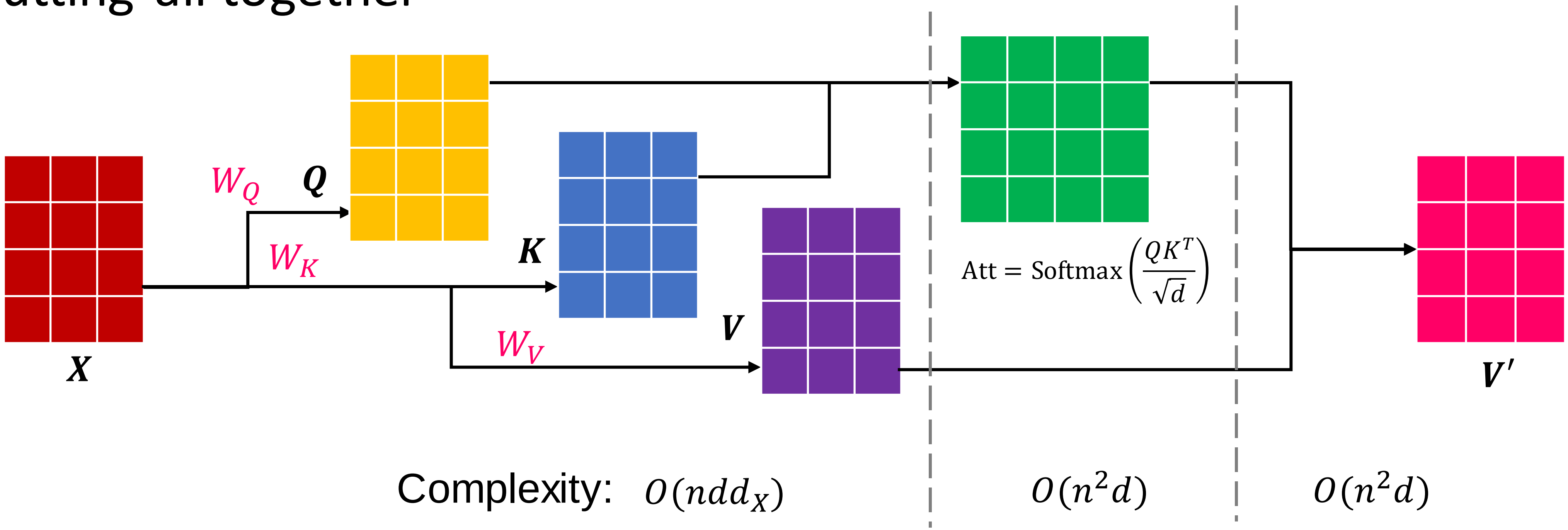
Transformers — Self-Attention (4/5)

- Step ③: Updated value $\boldsymbol{V}' = \text{Att}\,\boldsymbol{V} \in \mathbb{R}^{n \times d}$ Matrix product



Transformers — Self-Attention (5/5)

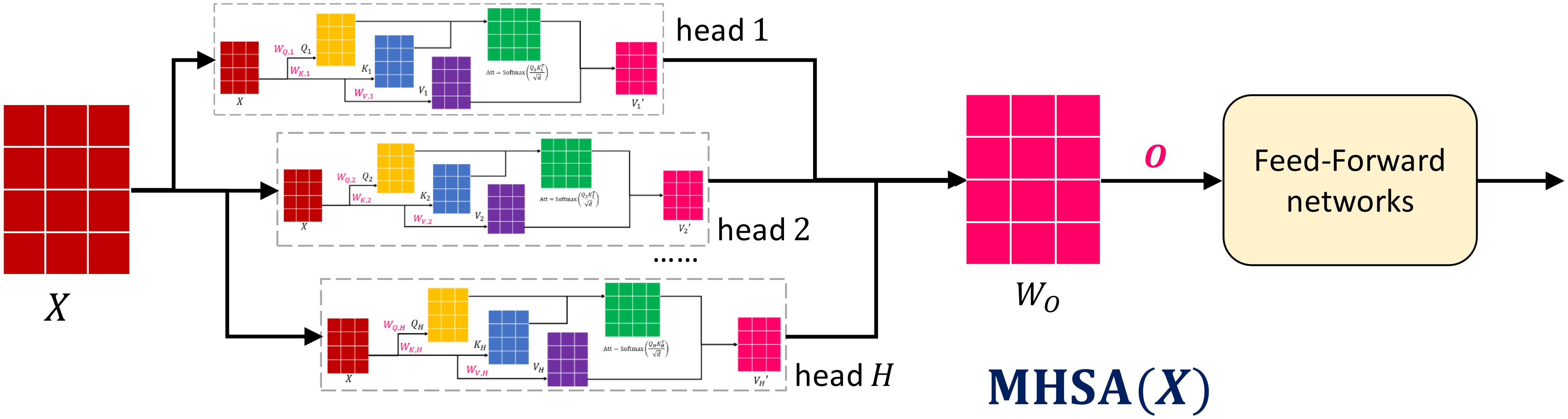
- Putting all together



The computation complexity is quadratic to number of tokens

Transformers — Multi-Head Self-Attention

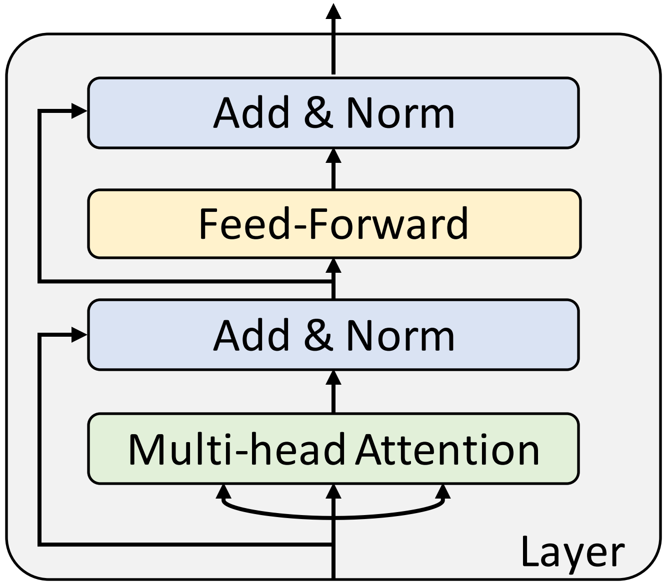
- There are usually **multiple aspects** that a token can attend to.
- We extend the attention to multiple heads, with multiple (Q, K, V) triplets on the same features.
  - The output of multi-head self-attention $O = \text{Concat}([V_1', V_2', \dots, V_H'])W_O$
  - Learnable parameters in each attention layer: $W_{Q,i}, W_{K,i}, W_{V,i} \in R^{d\times d}$ for head i, $W_O \in R^{Hd\times d_o}$



Transformers — Layer (1)

- **MHSA**: multi-head self-attention
- Transformer layer: $X \rightarrow \mathrm{LayerNorm}(X + \mathrm{MHSA}(X))$

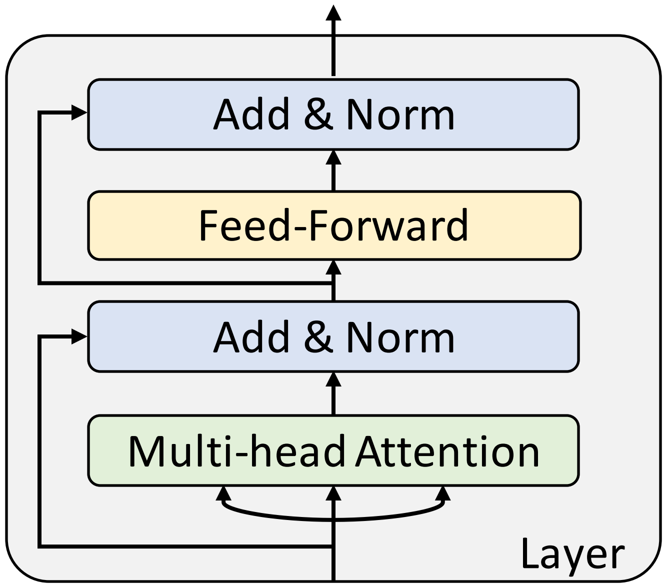
- **Residual connections** are added to
  - Enable smooth gradient flow in deep transformers
  - Keep the information of the original sequence.



Transformers — Layer (2)

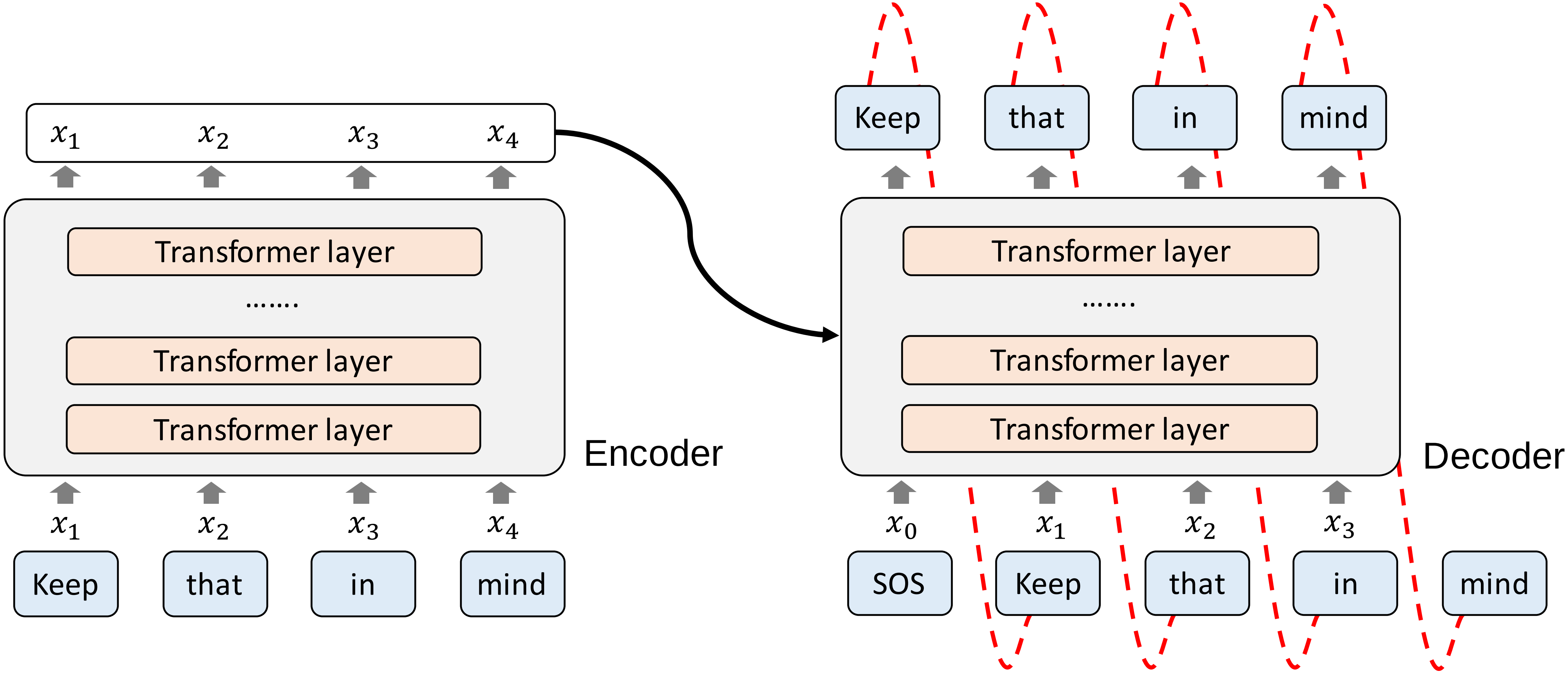
- Transformer layer: $X \rightarrow \text{LayerNorm}(X + \text{MHSA}(X)) \rightarrow \text{LayerNorm}(X + \text{FFN}(X))$

- **Layer Normalization** is used to enable faster training with small regularization and keep features in similar magnitudes.
  - BatchNorm isn't applied because batch size is usually small in Transformers due to GPU memory constraints. Besides, BatchNorm has been shown to lead to worse performance in NLP.



- **MLPs** are added for "post-processing", and allow transformations on each sequence token.

Transformers —Encoder / Decoder



Transformers —Positional Encoding (1)

Positional encoding is adopted to indicate the position of the token in the sequence.

Input $\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x_1} & \boldsymbol{x_2} & \boldsymbol{x_3} & \boldsymbol{x_4} & \boldsymbol{x_5} \end{bmatrix}$

0	**1**	**2**	**3**	**4**

- Indexing according to its position.
- **Problem**: if input length is 501, then the largest index is 500, while the neural network weights hover around 0.

0	**0.2**	**0.4**	**0.6**	**0.8**

- Normalizing indices to $[\mathbf{0}, \mathbf{1}]$.
- **Problem**: Cannot handle variable input length.

Length 5 : 0.2 means 2-nd token
Length 10: 0.2 means 4-th token

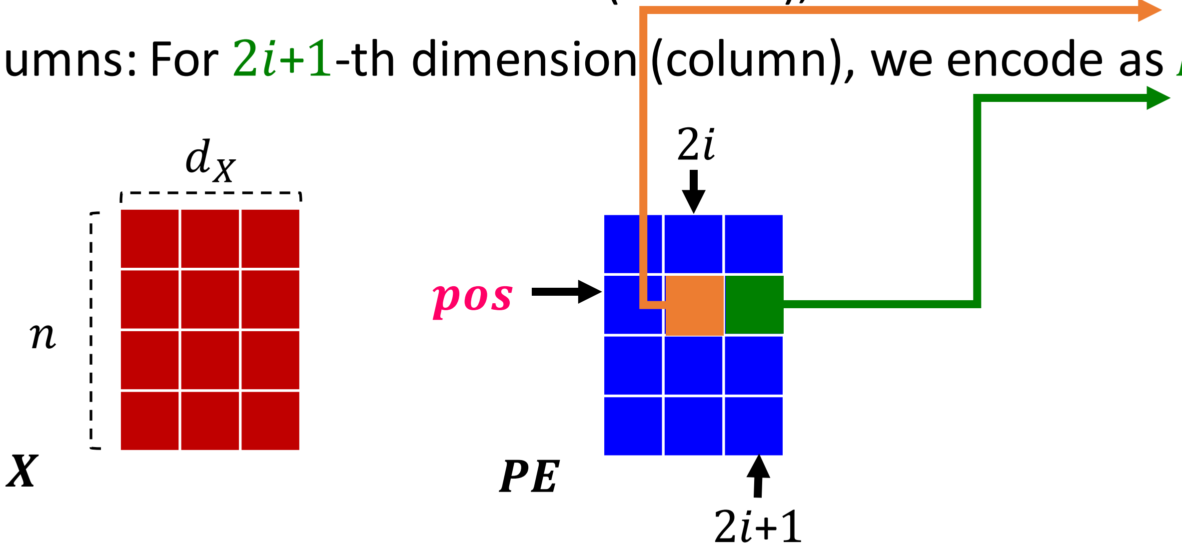
0	**1**	**2**	**3**	**4**
0	**0**	**0**	**0**	**1**
0	**0**	**1**	**1**	**0**
0	**1**	**0**	**1**	**0**

- Binary encoding
- **Problem**: Cannot represent relative positions (cannot do interpolations).

We need discretization of something continuous!

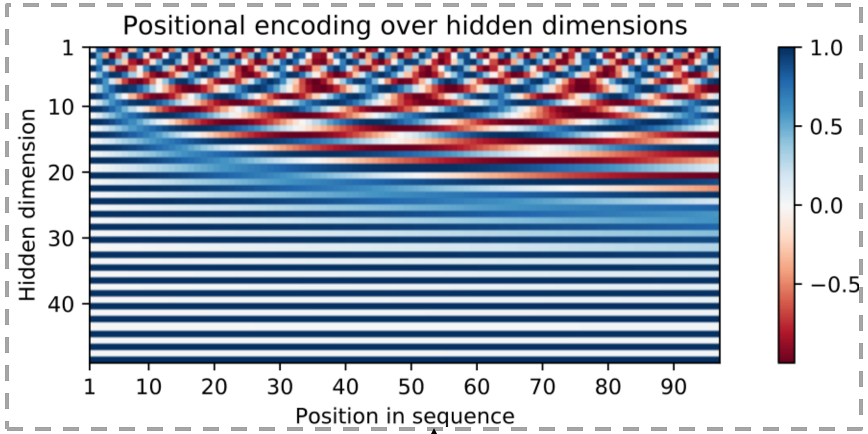
Transformers —Positional Encoding (2)

- **Learnable** 1-D/2-D embedding (commonly adopted in NLP/ CV):
  - We hope the relative positional relationship can be learnt through **back-propagation**.
- **Cosine encoding** (commonly adopted in NLP):
  - Recall the **input shape is of** $\mathbb{R}^{n\times d_X}$, we want the **PE has the same shape** $\mathbb{R}^{n\times d_X}$
  - For the $\boldsymbol{pos}$-th token ($\boldsymbol{pos}$-th row),
    - Even columns: For $2i$-th dimension (column), we encode as $PE_{(\boldsymbol{pos},\boldsymbol{2i})} = \sin\left(\boldsymbol{pos}/10000^{\boldsymbol{2i}/d_X}\right)$
    - Odd columns: For $2i+1$-th dimension (column), we encode as $PE_{(\boldsymbol{pos},\boldsymbol{2i+1})} = \cos\left(\boldsymbol{pos}/10000^{\boldsymbol{2i}/d_X}\right)$

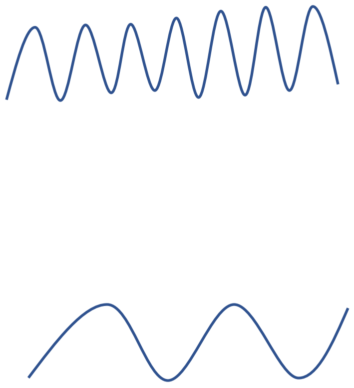


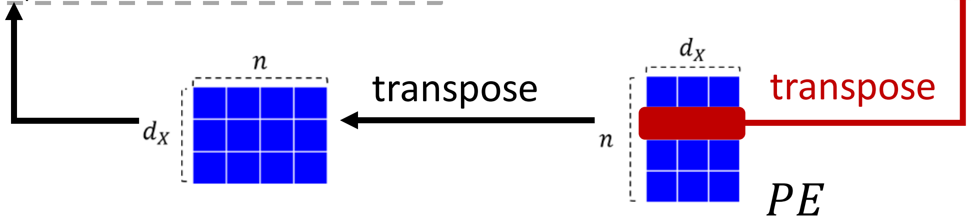
Transformers —Positional Encoding (3)

- **Cosine encoding**
  - $PE_{(pos,2i)} = \sin\left(pos/10000^{2i/d_X}\right)$, $PE_{(pos,2i+1)} = \cos\left(pos/10000^{2i/d_X}\right)$.
  - $\omega_i = 1/10000^{2i/d_X}$.
  - Relative distance: $PE_{(pos+k)}$ can be easily represented as a linear function of $PE_{(pos)}$ (show it).



$$PE_{(pos)} = \begin{bmatrix} \sin(\omega_1 \cdot pos) \\ \cos(\omega_1 \cdot pos) \\ \sin(\omega_2 \cdot pos) \\ \cos(\omega_2 \cdot pos) \\ \vdots \\ \sin(\omega_{d/2} \cdot pos) \\ \cos(\omega_{d/2} \cdot pos) \end{bmatrix}_{d\times 1}$$





Summary: Transformer Architecture

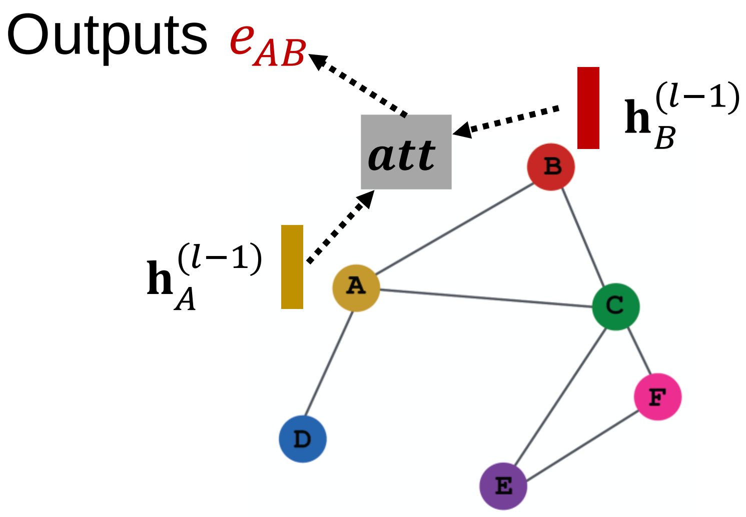
- Multi-Head Self-Attention ($\mathbf{MHSA}(\boldsymbol{X})$)
  - For head $\boldsymbol{i}$
    - $\boldsymbol{Q_i} = \boldsymbol{X}\boldsymbol{W}_{\boldsymbol{Q_i}}, \boldsymbol{K_i} = \boldsymbol{X}\boldsymbol{W}_{\boldsymbol{K_i}}, \boldsymbol{V_i} = \boldsymbol{X}\boldsymbol{W}_{\boldsymbol{V_i}}$
    - $\text{Att}_i = \text{Softmax}\left(\frac{\boldsymbol{Q_i}\boldsymbol{K_i}^T}{\sqrt{d}}\right) \in \mathbb{R}^{n\times n}$
    - $\boldsymbol{V_i}' = \text{Att}_i \boldsymbol{V_i} \in \mathbb{R}^{n\times d}$
  - Concatenating all heads: $\boldsymbol{O} = \text{Concat}([\boldsymbol{V}'_1, \boldsymbol{V}'_2, \ldots, \boldsymbol{V}'_H])\boldsymbol{W_O}$
- $\boldsymbol{X} = \mathbf{LayerNorm}(\boldsymbol{X} + \mathbf{MHSA}(\boldsymbol{X}))$
- $\boldsymbol{X} = \mathbf{LayerNorm}(\boldsymbol{X} + \mathbf{FFN}(\boldsymbol{X}))$

Recall: Graph Attention Mechanism (1)

- Let att be an **attention mechanism**
  - Attention coefficient e_{vu} is computed by a based on the messages of v, u:

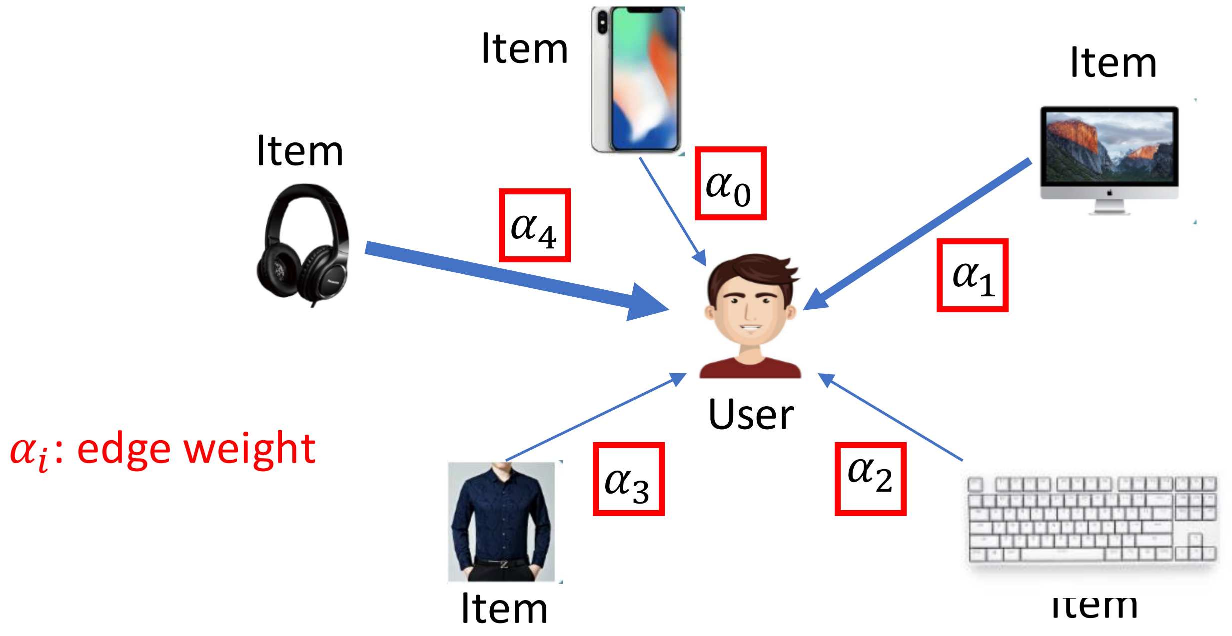
$$e_{vu} = att(\mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)}, \mathbf{W}^{(l)}\mathbf{h}_v^{(l-1)})$$

  - **e_{vu} indicates the importance of u's message to node v**



Recall: Graph Attention Mechanism (2)

- Learnable weighting function can provide a good **interpretability**
  - Different edge weights indicates difference importance of the neighbor nodes
  - Take recommender system as an example



- User aggregates the information from items with different weights
  - High attention weights indicate that user prefers these corresponding items

Recall: Graph Attention Network (GAT)

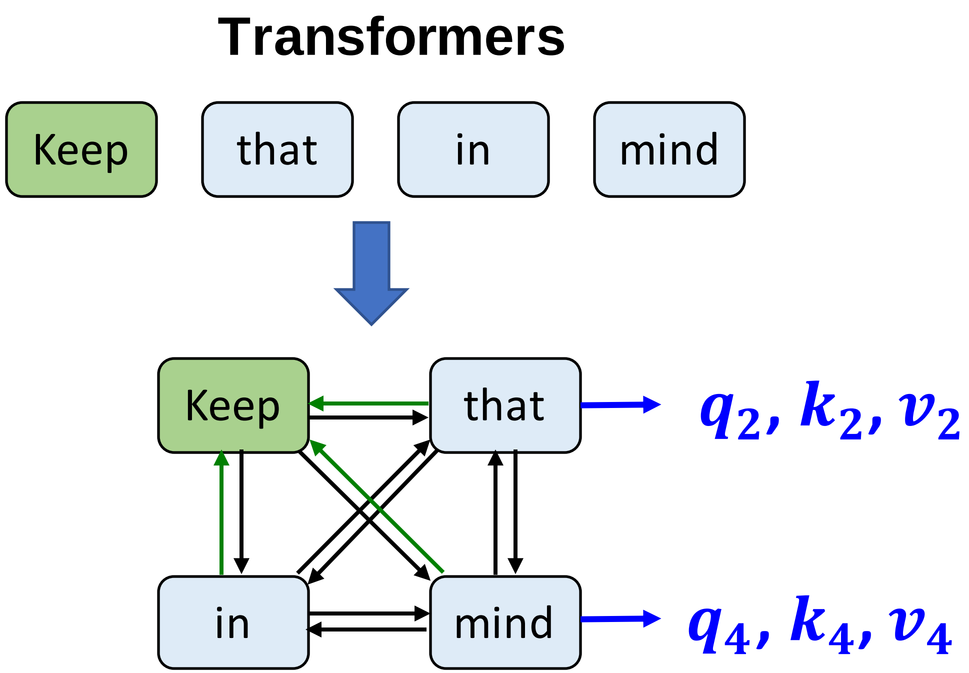
- A GAT layer (single head):

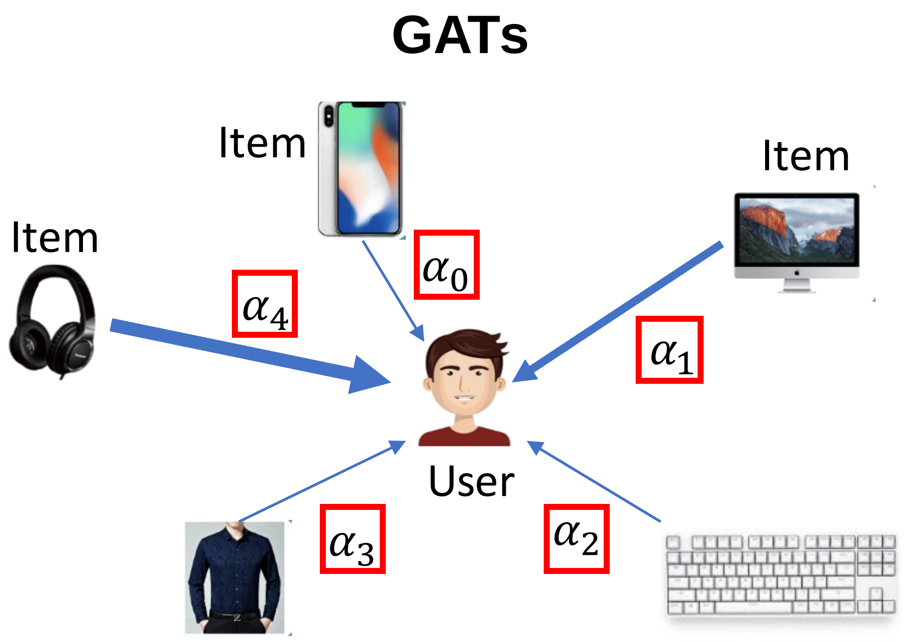
Learnable single-head or multi-head attention mechanism

  - **Attention** computing: calculate the importance of neighbors
  $$\alpha_{vu} = att\left(\mathbf{h}_v^{(l-1)}, \mathbf{h}_u^{(l-1)}\right)$$
  - **Message** computing: transform information of neighbor node to a message
  $$\mathbf{m}_u^{(l)} = \alpha_{vu}\mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)}, u \in N_v$$
  - **Aggregate** message: aggregate messages from neighbor nodes
  $$\mathbf{h}_v^{(l)} = \sigma\left(\sum_{u \in N_v} \mathbf{m}_u^{(l)}\right)$$

Looks similar to self-attention?

Transformers — in the Language of Graphs (1)





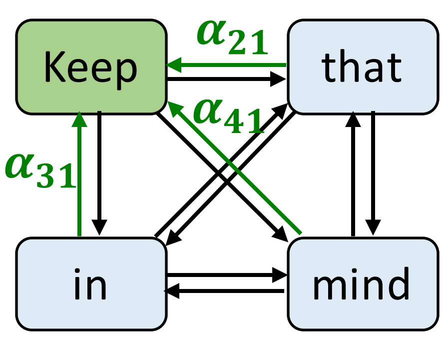
Step ① Mapping: Each node feature $\boldsymbol{x_i}$ is projected to $\boldsymbol{q_i, k_i, v_i}$.

Attention computation: calculate the importance of neighbors

$$\alpha_{vu} = att\left(\mathbf{h}_v^{(l-1)}, \mathbf{h}_u^{(l-1)}\right)$$

Transformers — in the Language of Graphs (2)

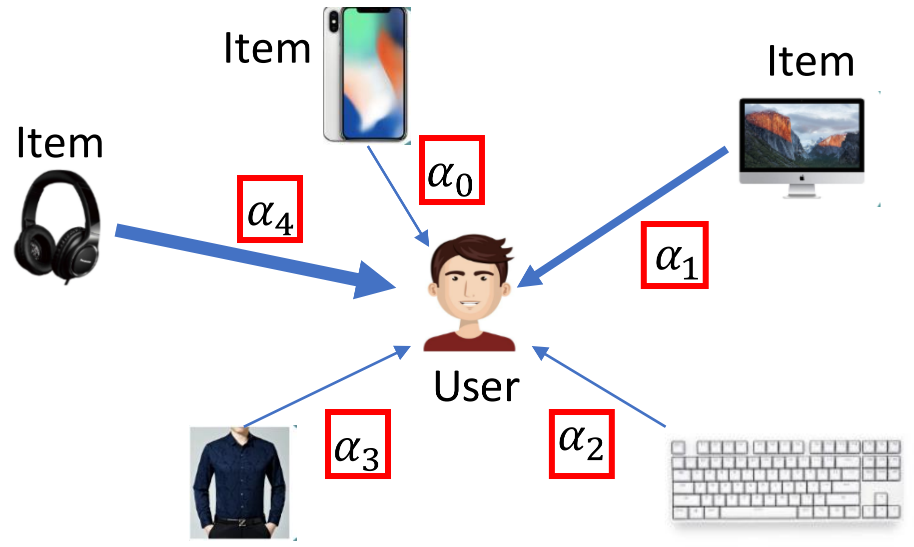
Transformers



Step ② Attention: Calculate the edge weights using $\boldsymbol{q_i}, \boldsymbol{k_j}$ of the two endpoints node i and j as $e_{ij} = \boldsymbol{q}_{\boldsymbol{i}}^T \boldsymbol{k}_{\boldsymbol{j}} / \sqrt{d}$, then normalizing it by the neighbors of node i

$$\boldsymbol{\alpha_{ij}} = \text{softmax}_i(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k \in N_i} \exp(e_{ik})}$$

GATs

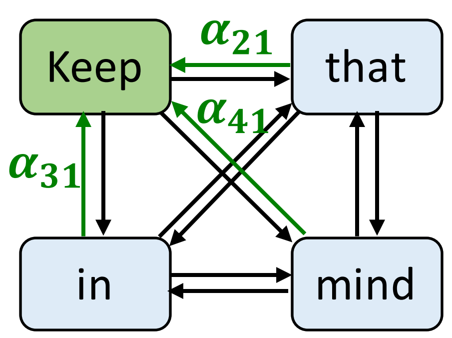


Message computing: transform information of neighbor node to a message

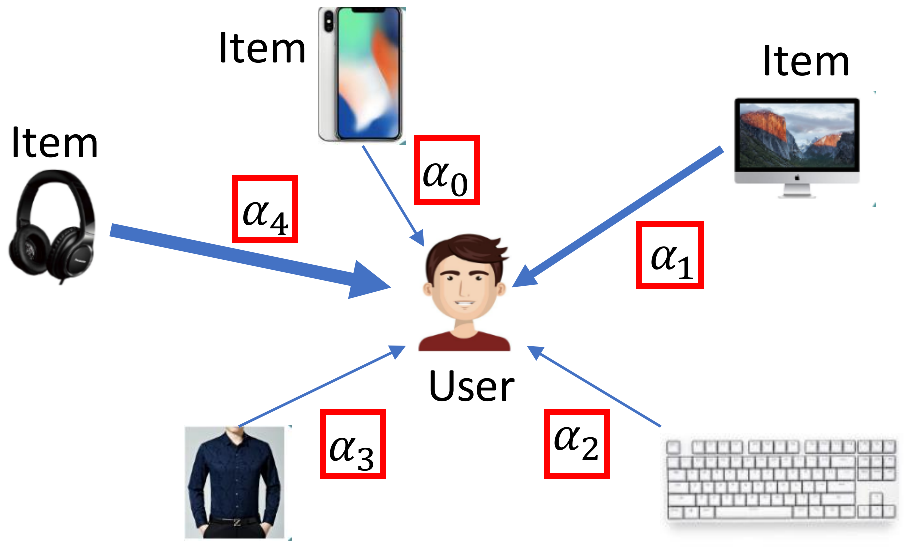
$$\mathbf{m}_u^{(l)} = \alpha_{vu} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)}, u \in N_v$$

Transformers — in the Language of Graphs (3)

Transformers



GATs



Step ③ Update: Update each node feature according to its neighbors as

$$\boldsymbol{x_i}' = \sum_{k \in N_i} \boldsymbol{\alpha_{ij}} \boldsymbol{x_j}$$

Aggregate message: aggregate messages from neighbor nodes

$$\mathbf{h}_v^{(l)} = \sigma\left(\sum_{u \in N_v} \mathbf{m}_u^{(l)}\right)$$

Transformers — in the Language of Graphs (4)

Summary: Comparison of **Self-attention (SA)** and **Graph Attention Networks (GAT)**

- Step ① Mapping
  - **SA**: different weights for q, k, v. $q = w_q x, k = w_k x, v = w_v x$.
  - **GAT**: shared weights for q, k, v. $q = wx, k = wx, v = wx$.
- Step ② Attention: **SA** uses dot-product attention, while (the original) **GAT** uses concatenation with MLP
  - Dot-product: $e_{ij} = q_i^T k_j / \sqrt{d}$
  - Concat: $e_{ij} = \mathrm{act}\left(W \quad [q_i || k_j]\right)$, where c is a weight vector and act is the activation function like $\mathrm{LeakyReLU}$

Transformers — in the Language of Graphs (5)

- The above computations do not require the assumption of **the complete graph**.
  - We assume full connectivity, mostly because we do not want to miss any potential token correlations.
- Self-attention can be easily adapted to graph-structured input data where the token correlations are given by the **adjacency matrix**, by replacing the **complete graph** with the **input graph**.
  - $\text{Self–Att}(X) = \text{Softmax}\left(\frac{(\mathbf{W}_k \boldsymbol{X})\,(\mathbf{W}_q \boldsymbol{X})^T}{\sqrt{d}} \odot A_G \odot \mathbf{W}_E E\right) V.$
  - A_G is the adjacency matrix of the graph and E is the edge weights of the graph if any.
- The complexity is no longer $O(n^2 d)$ but related to the edge number $O(E)$

Outline of Today's Lecture

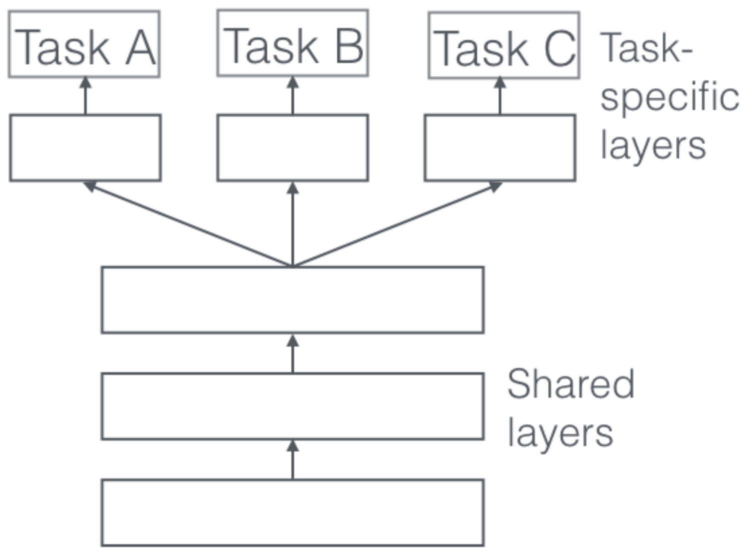
1. Self-Attention and Transformers

2. **Transformers Applications**

3. Graph Transformers and Sparse Transformers

Why is Transformer a Popular Choice

- Resolves various challenges of RNN-based architectures
- Attention makes the architecture **expressive and flexible** for different application scenarios
- It is very amenable to **self-supervised objectives**
  - We can leverage the vast number of **unsupervised examples** to learn a general model
  - Can be fine-tuned for **many downstream tasks**
  - Can out-perform models that are only trained for a specific downstream tasks

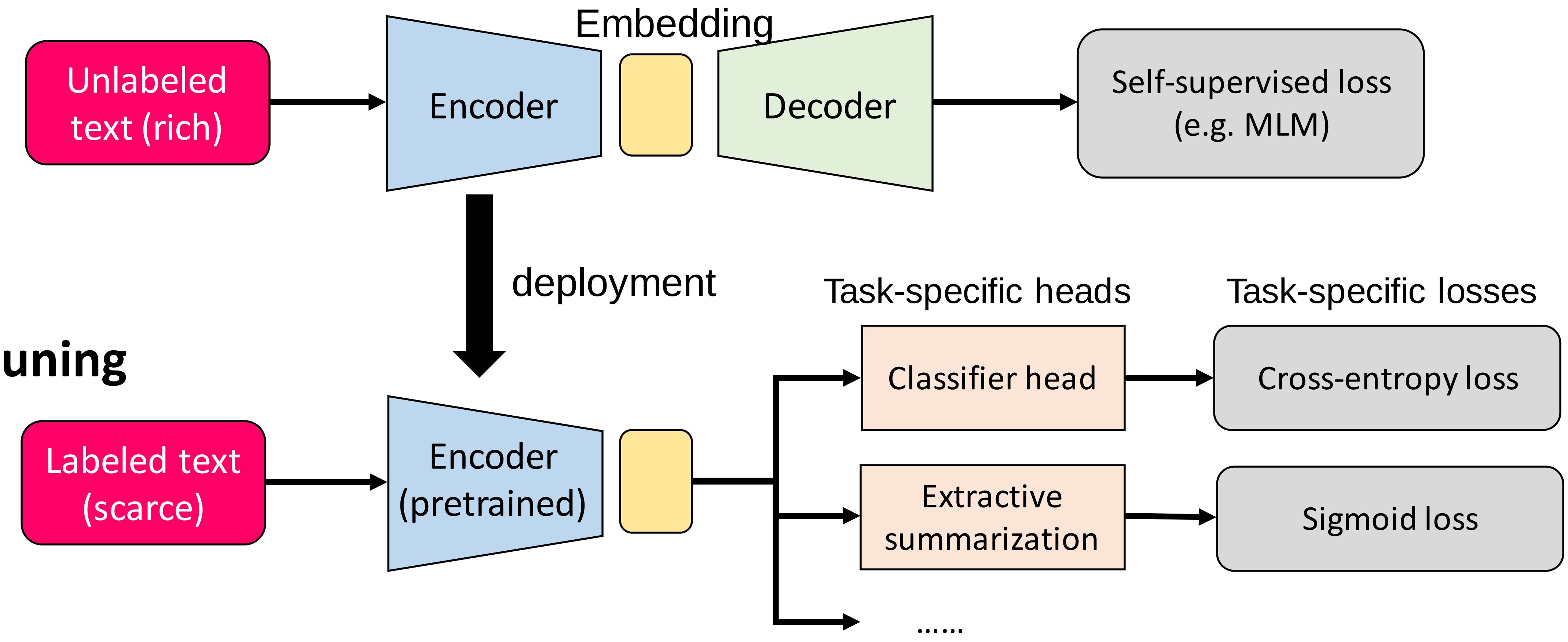


Label Scarcity

- ML models are hungry to data, especially labeled data for supervised task.
- The fast development of computer vision is largely contributed to **ImageNet**. It contains 14 million images **hand-annotated** by a team of researchers.
- This is often not possible for many domains. Most of time, it's easy to collect rich unlabeled data, but hard to obtain labeled data.
- Lack of annotated training example in NLP? **Pre-training** general-purpose language model on unlabeled large corpora (billions of characters) in **unsupervised** or **self-supervised** setting, then **fine-tuning** on smaller-scale tasks.

Pre-training and Fine-tuning

- **Pre-training**



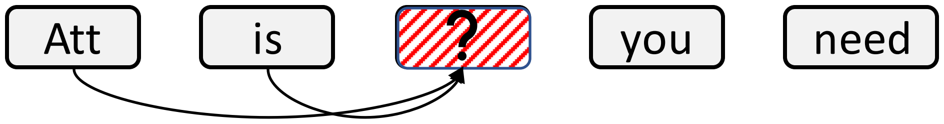
- **Fine-tuning**

Transformers in NLP — BERT/RoBERTa (1)

BERT —**Bidirectional** Encoder Representations from Transformers [Devlin et al., 2018]

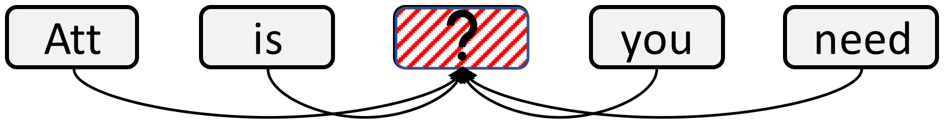
- **Pre-training task** (unsupervised): **Masked Language Model (MLM)**
  - First randomly masking $m\%$ tokens in the input sequence.
    - In BERT, 15% tokens are masked at random (replaced with the special `[MASK]` token)
  - Predicting masked tokens using remaining tokens.
  - Two modes: **Unidirectional** and **Bidirectional.**

Unidirectional [Radford et al., 2018]



- Maximize Likelihood of “all” given “Att” and “is”

Bidirectional



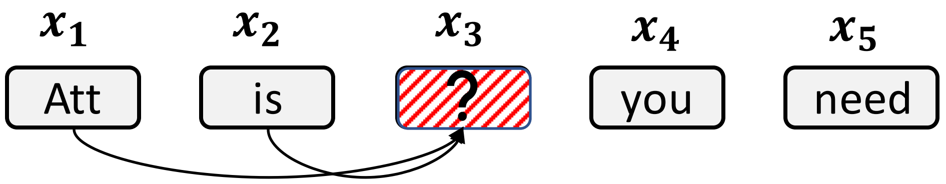
- Maximize Likelihood of “all”, given “Att” , “is”, “you”, “need”.

Transformers in NLP — BERT/RoBERTa (2)

BERT —**Bidirectional** Encoder Representations from Transformers [Devlin et al., 2018]

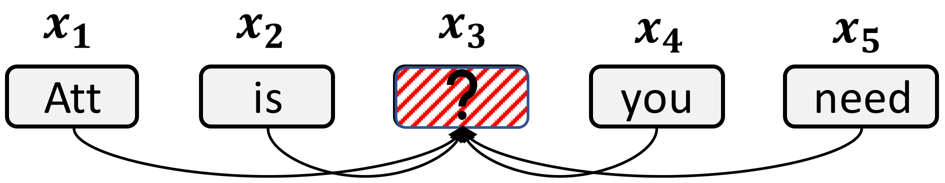
- **Masked Language Model (MLM) loss function**

Unidirectional



- Maximize Likelihood of "all" given "Att" and "is"
- $\max_\theta L(\boldsymbol{X}) = \max_\theta \sum_{i \in \boldsymbol{M}} \log P(\boldsymbol{x_i} | \boldsymbol{x_{i-k}}, \dots, \boldsymbol{x_{i-1}}; \theta)$, where $\boldsymbol{M}$ is the set of tokens masked.
- In implementation,
  - Masking the upper triangle of attention matrix Att
  - $\text{Loss} = \sum_{i \in \boldsymbol{M}} \text{CrossEntropy}(\text{Logit}_i, \text{Label}_i)$

Bidirectional



- Maximize Likelihood of "all", given "Att" , "is", "you", "need".
- $\max_\theta L(\boldsymbol{X}) = \max_\theta \sum_{i \in \boldsymbol{M}} \log P(\boldsymbol{x_i} | \{\boldsymbol{x_j}, i \neq j\}; \theta)$
- In implementation,
  - Keep full attention matrix Att
  - $\text{Loss} = \sum_{i \in \boldsymbol{M}} \text{CrossEntropy}(\text{Logit}_i, \text{Label}_i)$

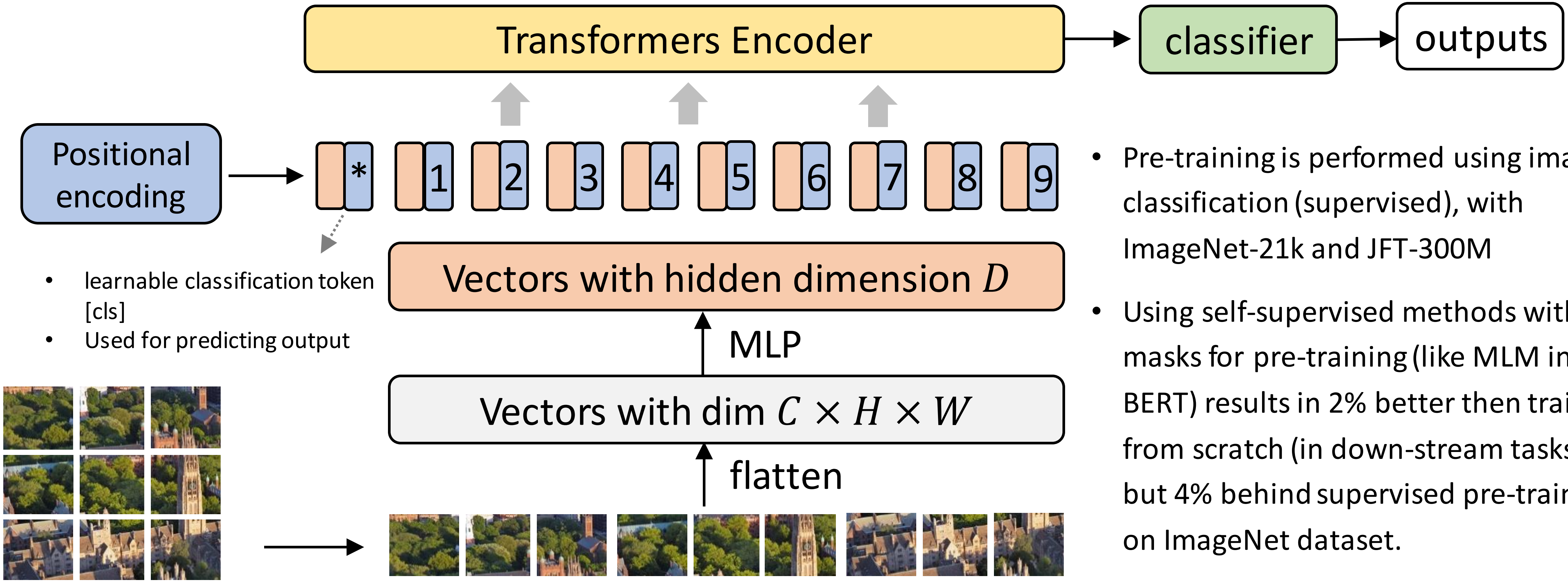
Transformers in NLP — BERT/RoBERTa (3)

RoBERTa —**Robustly** Optimized BERT [Liu et al., 2019]

- **Pretraining data**: BooksCorpus (800 M words) [Zhu et al., 2015], English Wiki (2500 M words), CC-News, OpenWebText [Gokaslan and Cohen,2019], Stories [Trinh and Le, 2018]
  - Partition the corpus into “sentences” with fixed length of 512 tokens.
- **Hyperparameters** in use (also commonly adopted in most NLP Transformers):
  - **12-Layer** Encoder + **12-Layer** Decoder
    (Pretrained Encoder is used more frequently in down-stream tasks)
  - Hidden dimension **768** = 12 (num of Heads) × 64 (dim of Head)
  - Learning rate: Warmup then linear decay
    - Warmup: Gradually increasing the learning rate to a specific value in the first few epochs
    - Linear decay: Decreasing the learning rate by the same amount (decrement) every epoch.

Transformers in CV — ViT [Dosovitskiy et al., ICLR 2021]

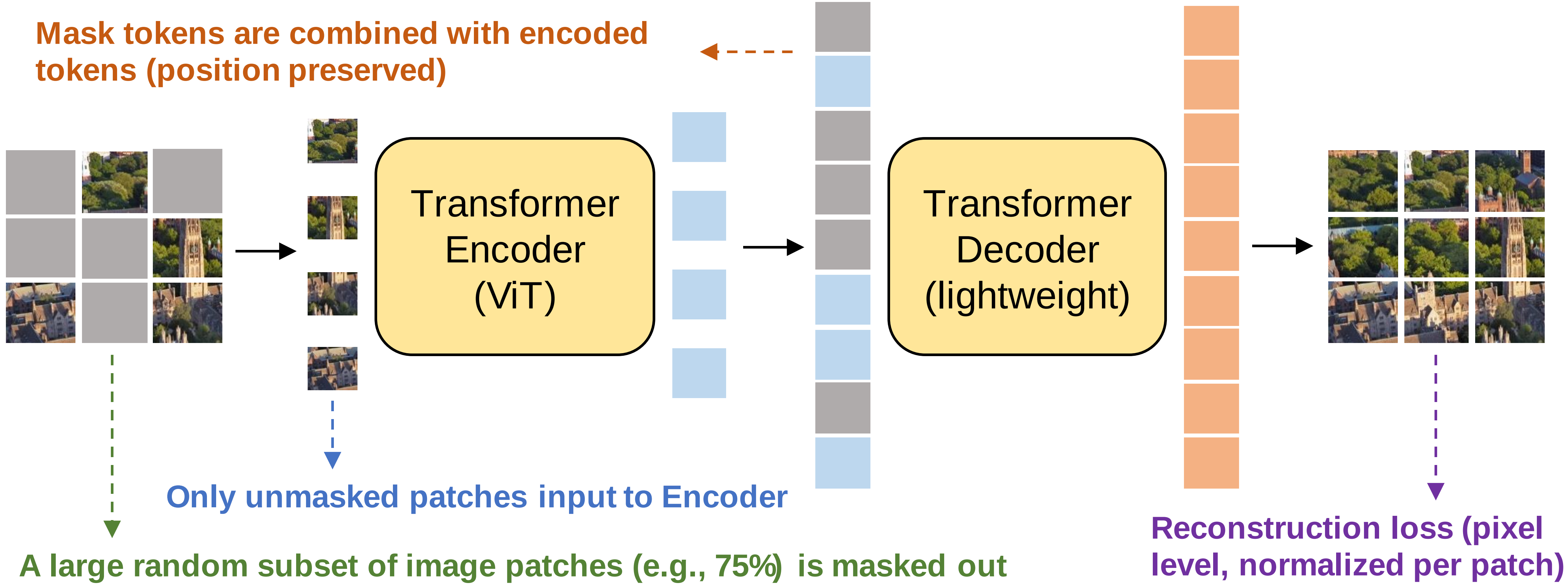
- An image patch is treated as a word in this context, and an image is partitioned to 16×16 tokens.



- Pre-training is performed using image classification (supervised), with ImageNet-21k and JFT-300M
- Using self-supervised methods with masks for pre-training (like MLM in BERT) results in 2% better then training from scratch (in down-stream tasks), but 4% behind supervised pre-training, on ImageNet dataset.

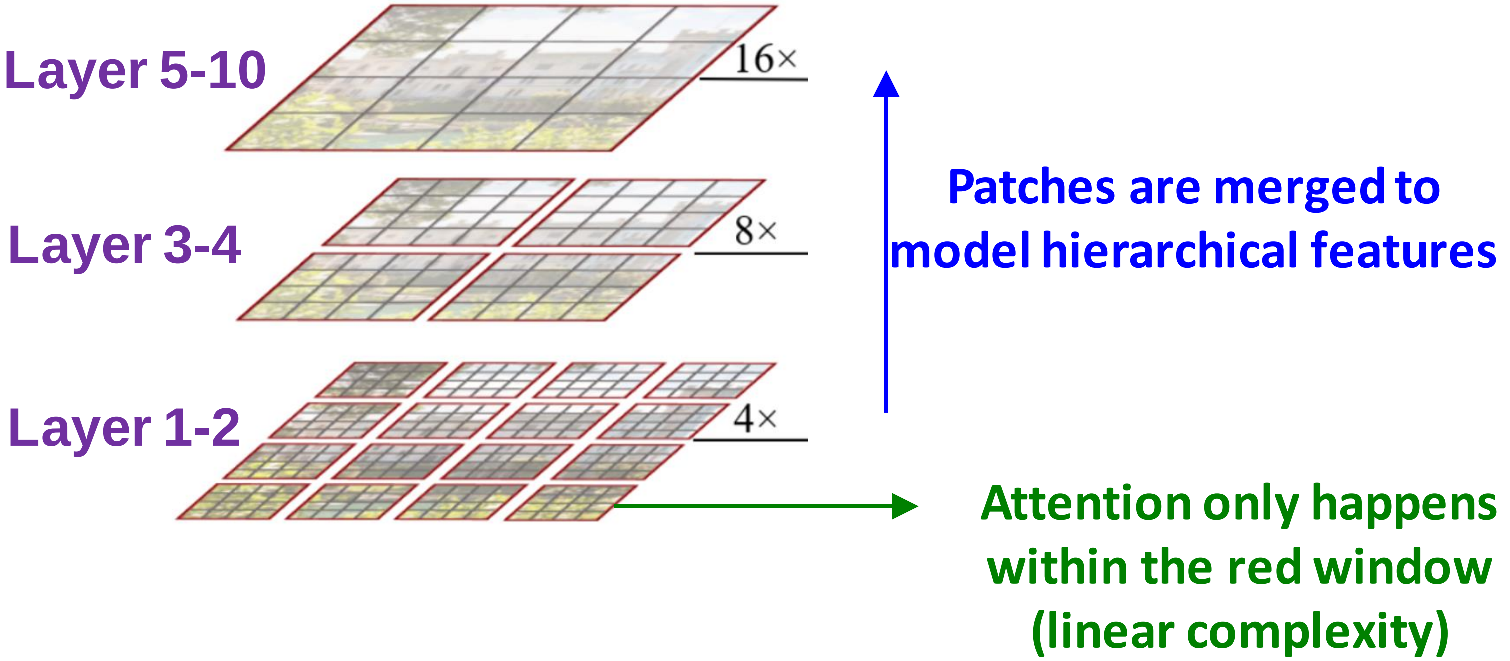
Transformers in CV — MAE [He et al., 2021]

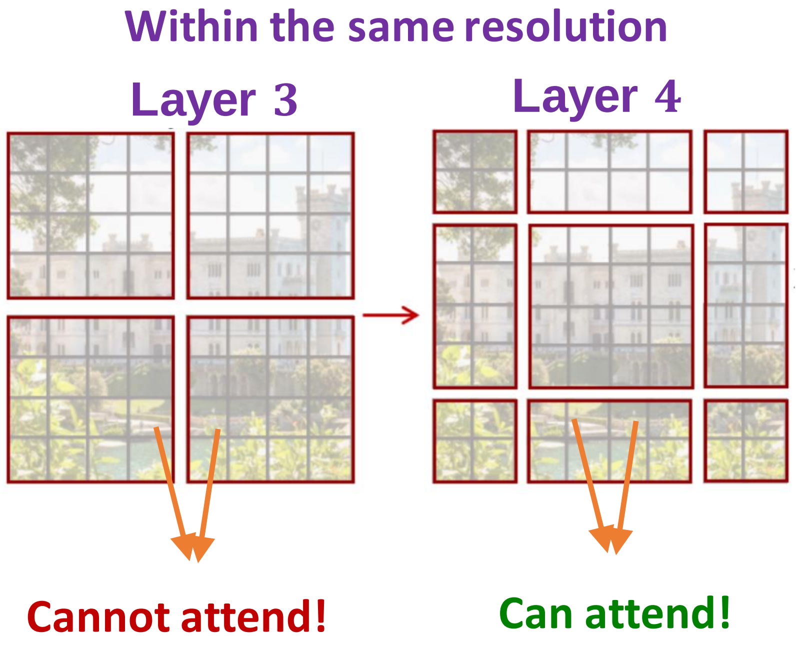
- Can we use self-supervised pretraining for vision Transformers?
  - **Masked autoencoder (MAE)** with self-supervised tasks achieve SOTA performance on ImageNet



Transformers in CV — Swin [Liu et al., CVPR 2021]

- **Limitation of ViT**: using patch (64×64×3) as the unit token can be limited due to high resolution and variance of pixel values.
- **Solution**: Swin calculates fine-grained attention with hierarchical attention and shifted window.





Transformer Application Summary

- Transformer architectures are dominant when it comes to **self-supervised learning and pre-training**
  - The paradigm of pre-training and then fine-tuning / multi-task learning on downstream tasks has been very successful in many machine learning areas
  - **NLP** transformer (e.g. BERT)
  - **Vision** transformer (e.g. ViT)

Outline of Today's Lecture

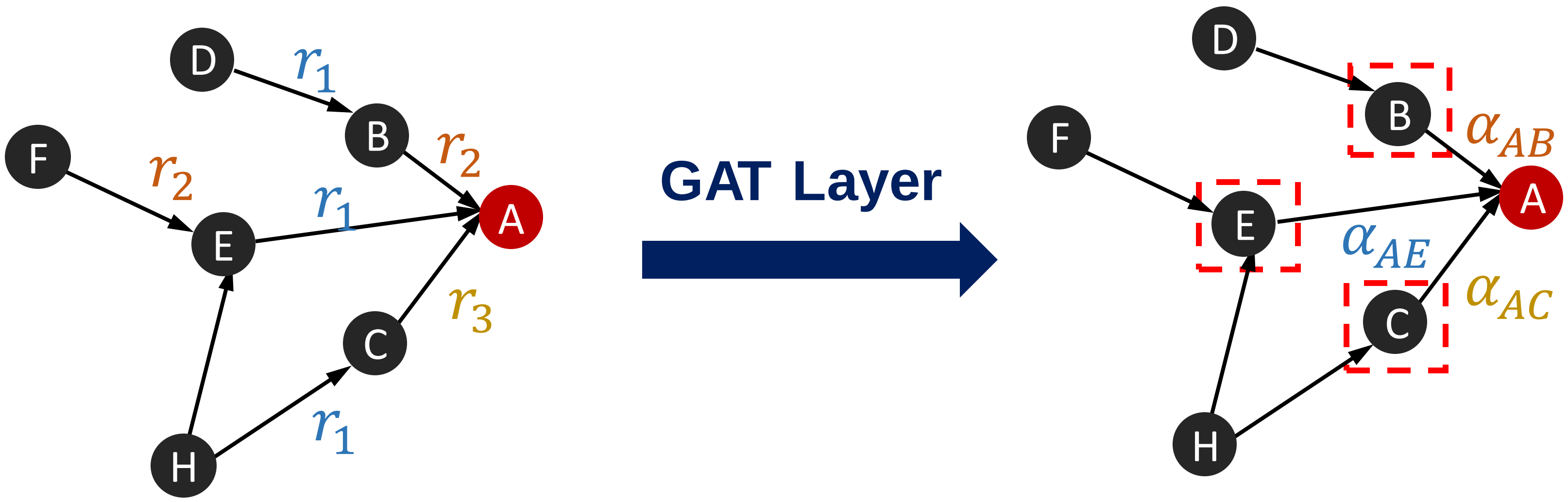
1. Self-Attention and Transformers

2. Transformers Applications

3. $\underbrace{\text{Graph Transformers}}_{\text{Transformers help Graph}}$ and $\underbrace{\text{Sparse Transformers}}_{\text{Graphs help Transformers}}$

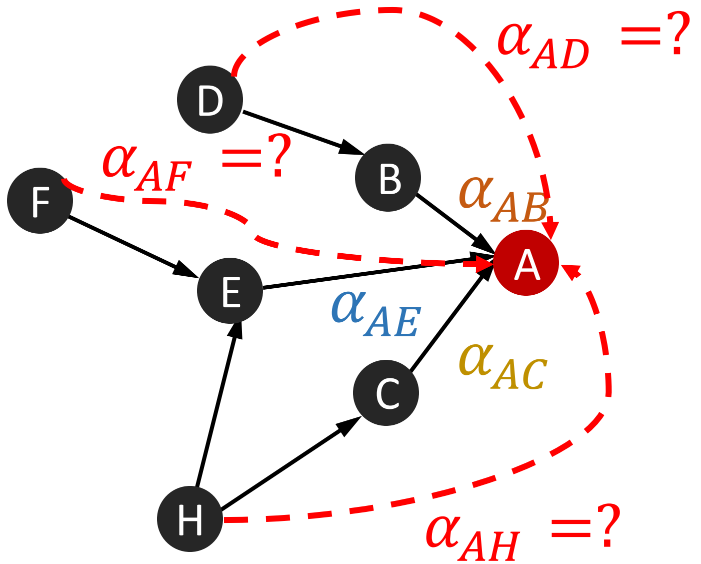
Recall: Limitation of Single Hop Attention (1)

- A single GAT layer can only explore the relationship between node and its **one-hop** neighbors
  - Target node only attends to its immediate neighbors



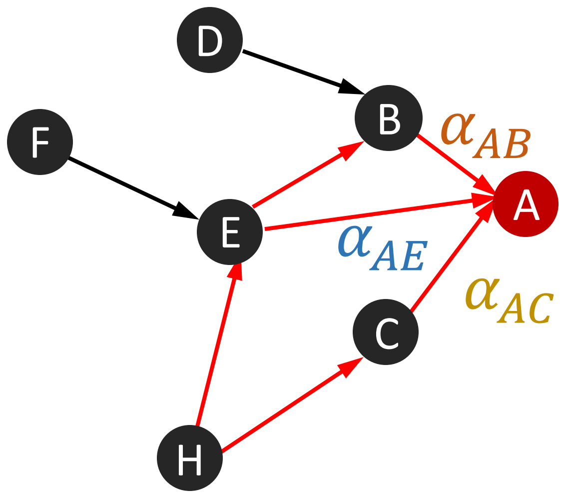
Recall: Limitation of Single Hop Attention (2)

- A single hop attention falls short in exploring **broader graph structure** and **multi-hop** neighbors
  - Stacking multiple GAT layers causes **over-smoothing** and **over-fitting**



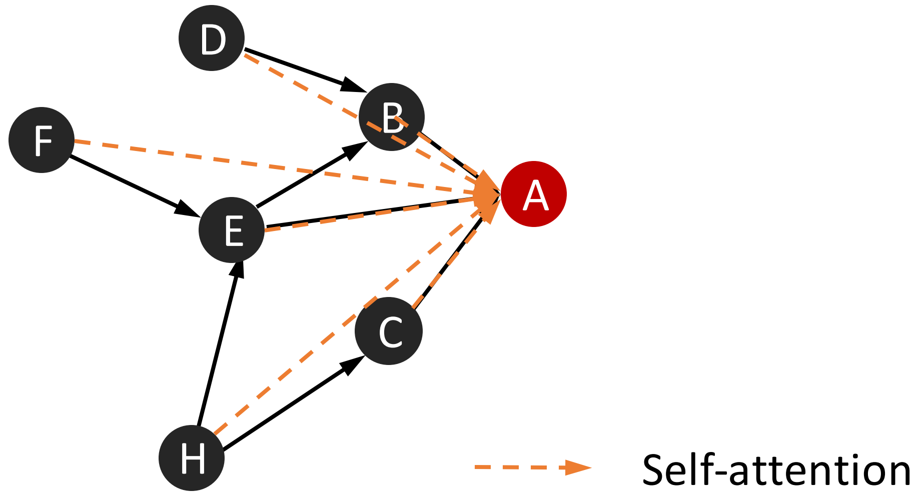
Multi-hop Attention v.s. Transformers (1)

Multi-hop Attention



- Enabling attention to **multi-hop** neighbors, resulting in direct/indirect attention between every pair of nodes

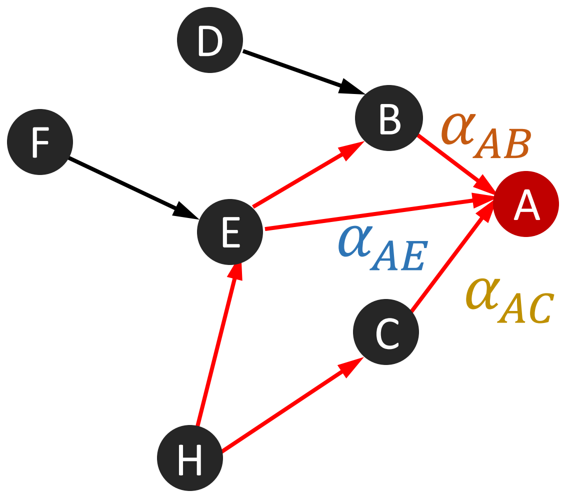
Transformers



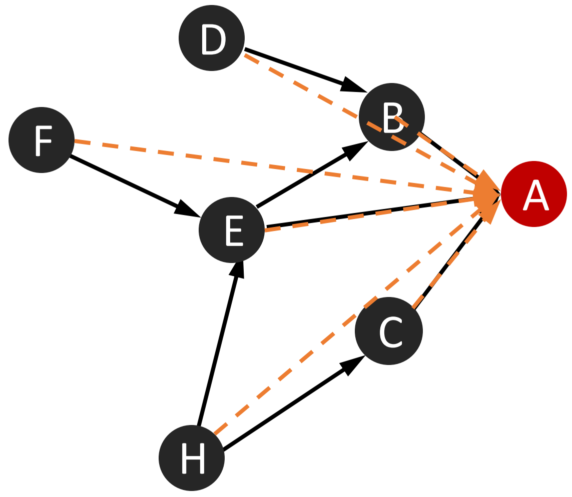
- Enabling attention of target node to **all remaining nodes** in the graph, resulting larger receptive field

Multi-hop Attention v.s. Transformers (2)

Multi-hop Attention



Transformers



- Using diffusion to reduce the complexity to $\boldsymbol{O(E)}$
- Full-attention with $\boldsymbol{O(n^2)}$ complexity

Graph Transformer as Matrix Operation

- No formal definitions of **Graph Transformers**
  - Self-attention can be defined among **1-hop neighbors**, **multi-hop neighbors**, or **all nodes** in the graph.
- E.g., **Graph Transformers** with **1-hop Attention**
  - In the matrix form
    - $\text{Self-Att}(X) = \text{softmax}\left(\frac{(\mathbf{W}_k X)(\mathbf{W}_q X)^T}{\sqrt{d}} \odot \boldsymbol{A_G} \odot \mathbf{W}_E \boldsymbol{E}\right) V.$

      $\odot$: Hadamard product (element-wise product)
    - $\boldsymbol{A_G}$ is the adjacency matrix of the graph and $\boldsymbol{E}$ is the edge weights of the graph if any.
  - The complexity is no longer $O(n^2 d)$ but related to the edge number $O(E)$
- But this is not enough, any new challenges for graph Transformers?

Graph Transformers Challenges (1)

- **Challenge 1:** How to describe the position of a node in a **graph**?
  - **Permutation equivariance** should be preserved (this is why naïve one-hot PE does not work).
  - Weisfeiler-Lehman-PE [Zhang et al., Graph-BERT, arXiv 2020]; Laplacian PE [Dwivedi, arXiv 2020]
  - Random features (RF) or deterministic distance encoding (DE) [Li & Leskovec, 2021]
  - Besides graph Transformers, such PE are also applied in other GNNs to increase **expressiveness** (recall 1-WL constraints). Will discuss in future lectures.

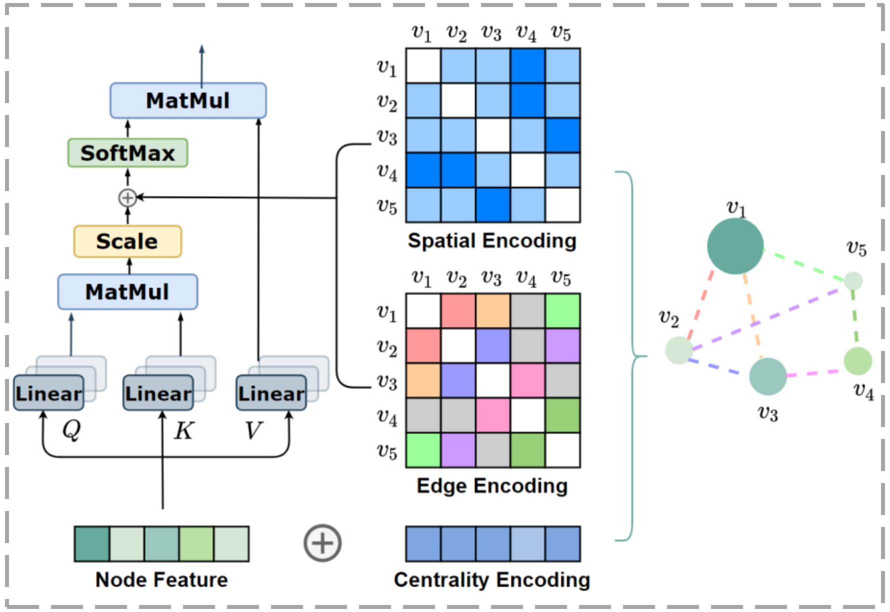
Graph Transformers Challenges (2)

- **Challenge 2**: How to incorporate graph features (topology, edge weights…)
  - Naïve approach: restricting a node only attending to its neighbors
  - Graphormer [NeurIPS 2021]:
    - Attending to all nodes, besides its neighbors, $O(n^2)$ complexity

$$e_{ij} = \frac{(\mathbf{h}_i \mathbf{W}_q)(\mathbf{h}_j \mathbf{W}_k)^T}{\sqrt{d}} + b_{\boldsymbol{\phi}(\boldsymbol{v}_i, \boldsymbol{v}_j)} + \boldsymbol{c}_{\boldsymbol{ij}}$$

Spatial Encoding:
Shortest path between $\boldsymbol{v}_i, \boldsymbol{v}_j$

Edge Encoding:
Average all edge features along the shortest path between $\boldsymbol{v}_i, \boldsymbol{v}_j$ $c_{ij} = \frac{1}{N} \sum_{e \in SP(i,j)} x_e w_e$



Outline of Today's Lecture

1. Self-Attention and Transformers

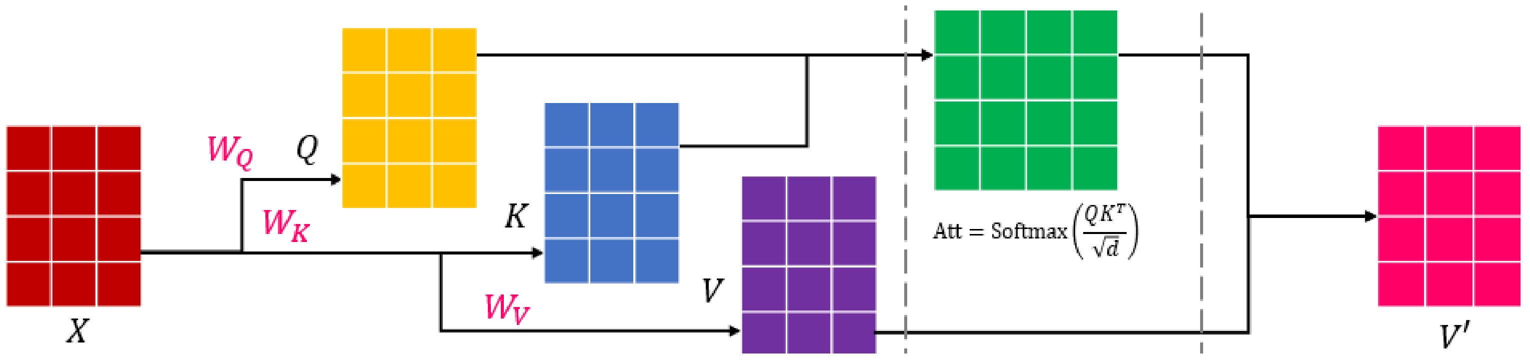
2. Transformers Applications

3. $\underbrace{\text{Graph Transformers}}_{\text{Transformers help Graph}}$ and $\underbrace{\textbf{Sparse Transformers}}_{\text{Graphs help Transformers}}$

Sparse Transformers

Conventional Transformers cannot scale to **long sequences** due to $\boldsymbol{O(n^2)}$ complexity from the full-attention

- The $\boldsymbol{QK}^T$ matrix multiplication, $\mathrm{Softmax}()$, $\mathrm{Att}V$ value updates all consume $\boldsymbol{n^2}$ time and memory.
- Recall that Roberta and ViT are only designed for 512 (words) and 256 (patches), respectively.

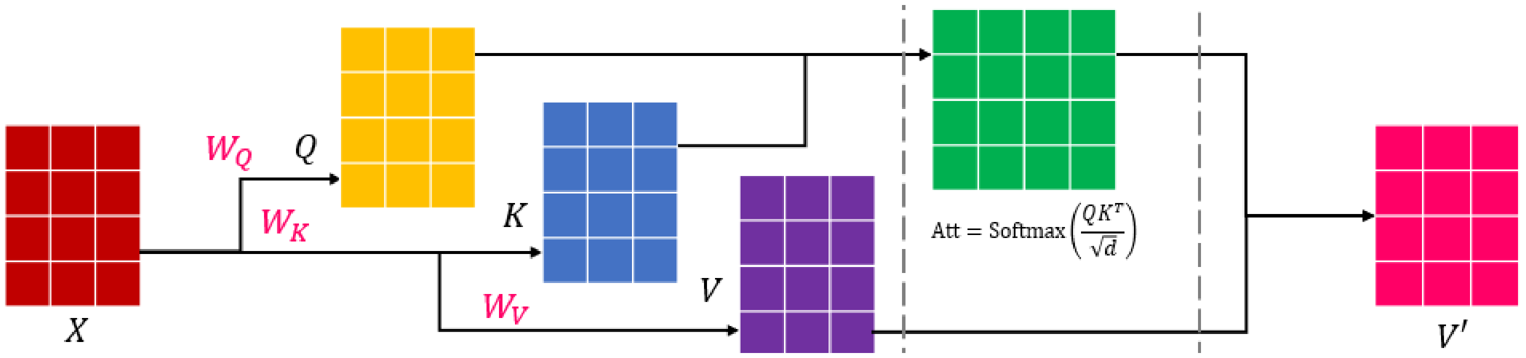


Complexity: $O(ndd_X)$ $O(n^2d)$ $O(n^2d)$

Sparse Transformers

Conventional Transformers cannot scale to **long sequences** due to $\boldsymbol{O(n^2)}$ complexity from the full-attention

- This is not enough for long sequences like a paragraph with **thousands** of words or an image with **64×64×3=4096** pixels.
- Because of quadratic dependency, increasing sequence length from 512 to 4096 results in around 64× more memory usage and running time, which usually cannot fit into a standard modern GPU (e.g NVIDIA A6000, 48 GB).

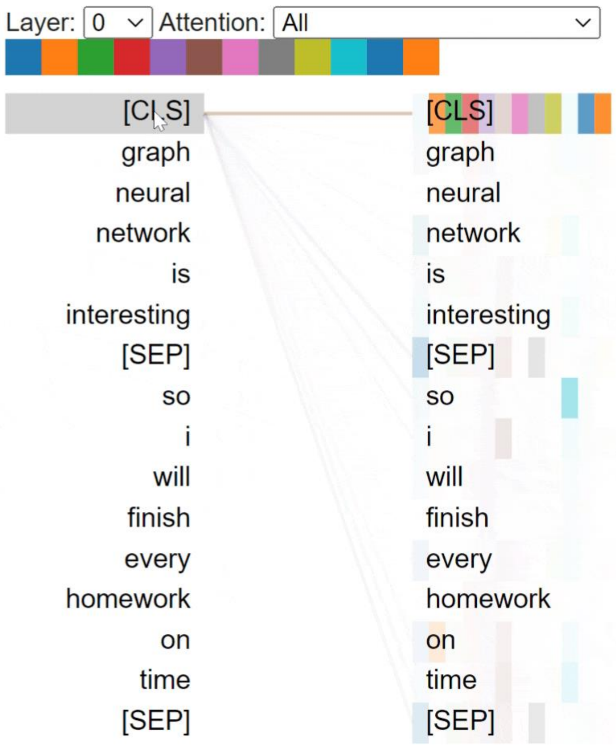


Sparse Transformers

- **Observation 1**: Although every attention is calculated, most of them are close to 0, the resulting attention maps are usually **sparse**.
- **Observation 2**: non-zero attention mostly appear between the node and its local neighbors. (**local** attention).
- **Observation 3**: some key words like "so" almost attend to every token in the sentence. (**global** attention)

Can we simplify self-attention (full-attention) using graph?

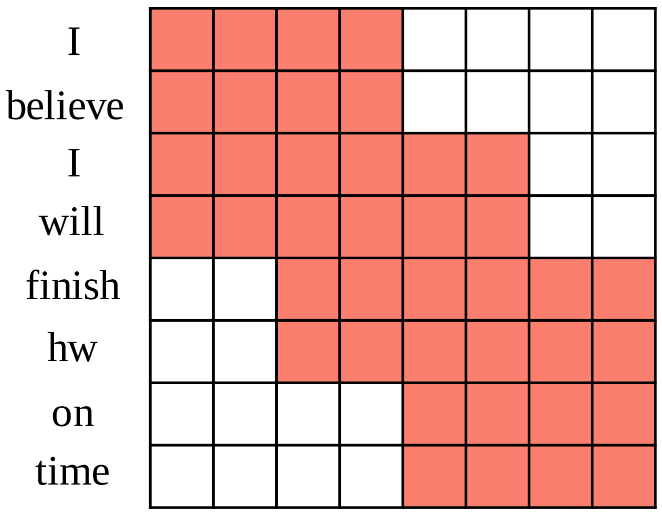
A sentence encoded by pretrained BERT



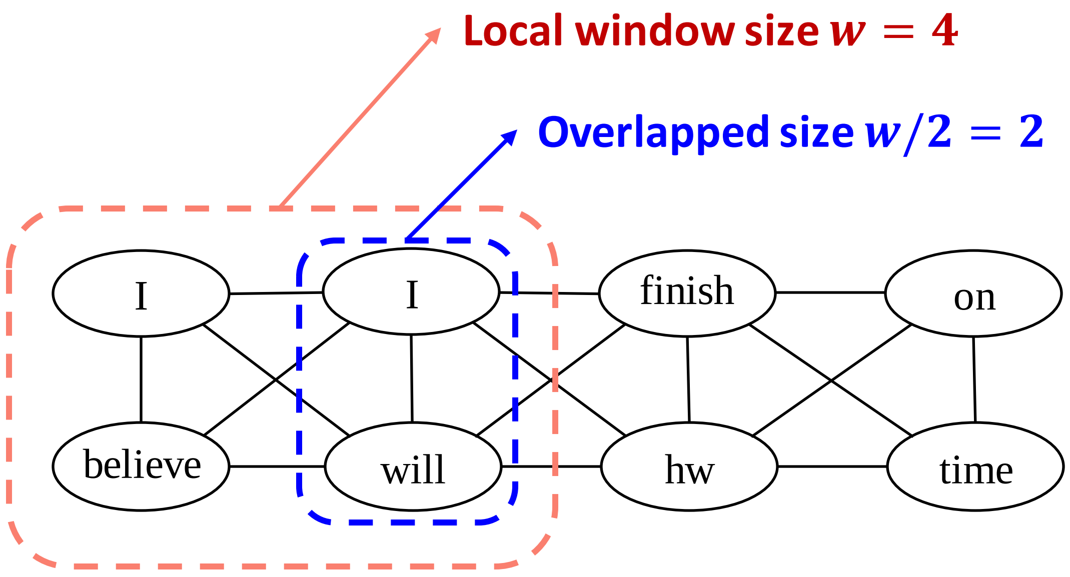
Try yourself!
https://github.com/jessevig/bertviz#-quick-tour

Sparse Transformers — Longformer [Beltagy et al., 2020]

- Applying overlapped local window attention to approximate the full-attention, only calculating attentions shaded in red



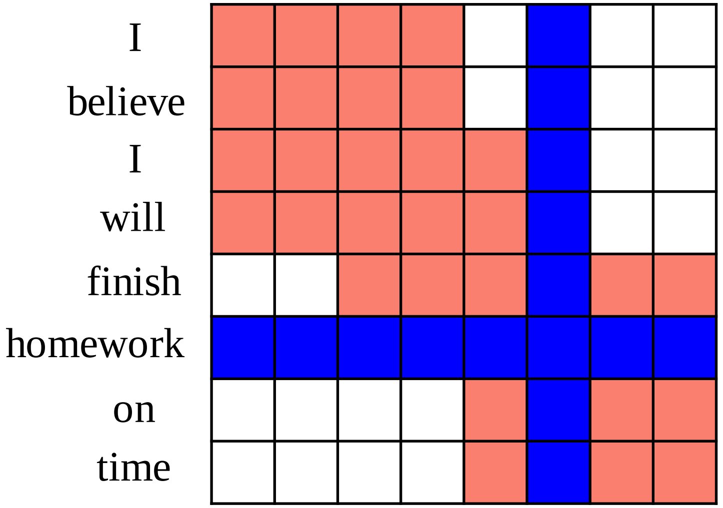
Masked Attention Pattern



Associated graph structure

Sparse Transformers — Longformer

- Longformer is based on the assumption that adjacent words have stronger correlations

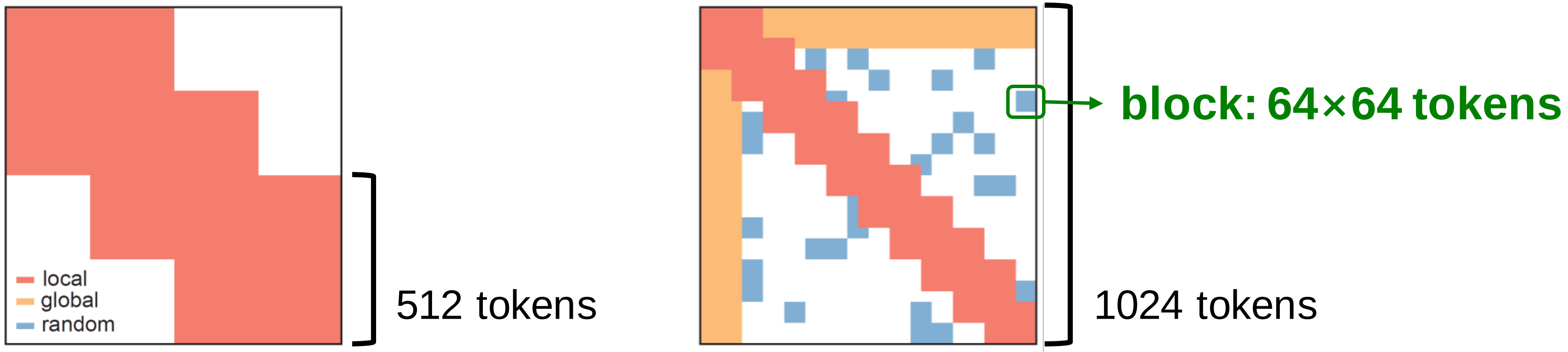


Masked Attention Pattern

- **Local window** is overlapped in half to enable cross-window attention (to ensure the graph is connected so that every pair of tokens can attend by stacking layers).
- **Global attention** is further introduced for specific down-stream task
- Complexity is now $O(nw)$ compared to $O(n^2)$.
- Longformer can handle long sequences like 4096 tokens, by specifying local window size to be 512

Sparse Transformers — BigBird [Zaheer et al., 2020]

- BigBird model further introduces **Random Attention** to better approximate the full-attention.
- The smallest unit in BigBird is called a **block** (64 adjacent tokens)
- “Blockifying” is used to accelerate the sparse attention computation



(a) Longformer

(b) BigBird

Sparse Transformers — Reformerer [Kitaev et al., 2020]

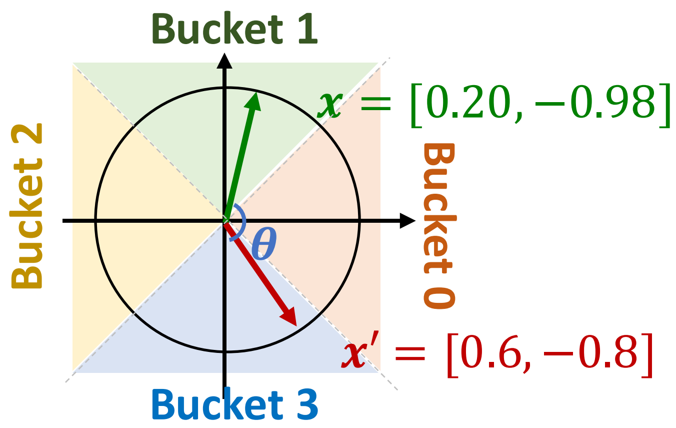
- Reformer is based on **locality sensitive hashing (LSH)** – a nearest-neighbor algorithm
  - **LSH** Definition: A hashing is called **locality-sensitive** if nearby vectors get the same hash with high probability and distant ones do no.
  - Main idea: First **grouping** tokens into different buckets (using LSH), then calculating **attentions within the same bucket**
  - The complexity is reduced to $O(b)$, b is the bucket size.

Sparse Transformers — Reformer [Kitaev et al., 2020]

- Details of **locality sensitive hashing (LSH)**
  - To get b hashes (buckets), introducing a fixed random matrix $R \in \mathbb{R}^{d \times \frac{b}{2}}$, the hash of input token $\boldsymbol{x} \in \mathbb{R}^d$ is computed as

$$h(x) = \operatorname{argmax}([\boldsymbol{x}R; -\boldsymbol{x}R])$$

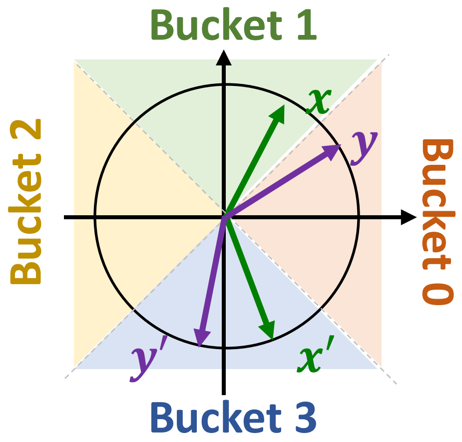
  - Setting $d = 2$, $\boldsymbol{x}$ becomes a 2-d vector
  - Setting $b = 4$, we get 4 buckets (0, 1, 2, 3)
  - For $\boldsymbol{x}' = R\boldsymbol{x}$, if we project $\boldsymbol{x}$ and $\boldsymbol{x}'$ to the unit circle, the function of $\boldsymbol{R} \in \mathbb{R}^{2\times 2}$ is rotation, i.e., $\boldsymbol{x} \xrightarrow{\text{rotated by } \theta} \boldsymbol{x}'$



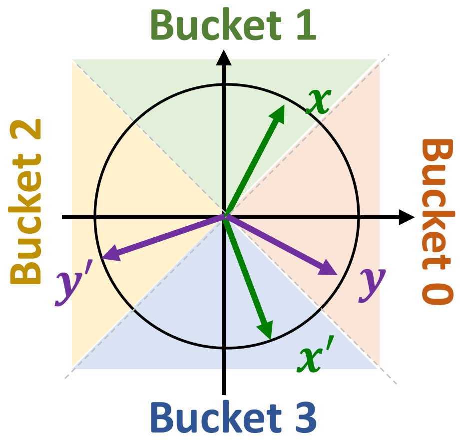
$$h(\boldsymbol{x}) = \operatorname{argmax}([0.6, -0.8, -0.6, \mathbf{0.8}]) = 3$$

Sparse Transformers — Reformerer [Kitaev et al., 2020]

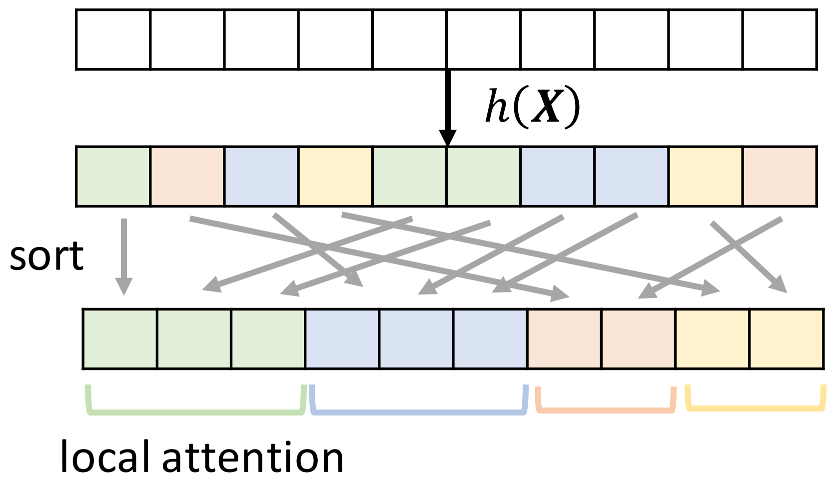
- If two tokens $\boldsymbol{x}$, $\boldsymbol{y}$ are close to each other (regarding e.g., angular distance), they have larger probabilities to share the same bucket.



$h(\boldsymbol{x}) = h(\boldsymbol{y}) = \mathbf{3}$



$h(\boldsymbol{x}) = \mathbf{3},\ h(\boldsymbol{y}) = \mathbf{2}$



- Attention in Reformer: only calculating attention within the same buckets

Summary

- Transformers
  - Transformers use **query (Q), key (K), value (V)** to compute attention values
  - All-pair correlation
  - Quadratic complexity
- Transformer applications
  - BERT (pre-trained **language** model)
  - ViT (**image** pre-training)
- (Multi-hop) graph attention is closely related to transformers!
  - Transformer can **inspire GNN architectures**
  - Graph learning techniques can **make transformers more efficient**