Transformers

CPSC483: Deep Learning on Graph-Structured Data

Rex Ying

Questions

- How to summarize what it means for an ML system to be **trusted**?

- Name one of the four major characteristics of a **trustworthy** ML system according to the book's opinion.

  Explain what does it mean and why it matters

Questions

- Have you noticed any news, articles, policies, events that have implications in trustworthy deep learning in recent years?

Readings

- Readings are updated on the website (syllabus page)
- **Readings:**
  - Attention is All You Need
  - Generative pre-training
  - GNN Survey

- This lecture is not explicitly tested
  - But in future lectures we will assume knowledge of this when developing trustworthy components on top of Transformers

Outline of Today's Lecture

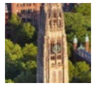
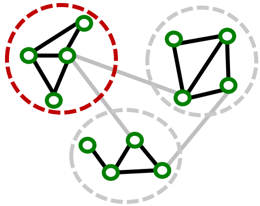
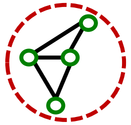
1. **Self-Attention and Transformers**

2. **Transformers Applications**

3. **Graph Neural Networks**

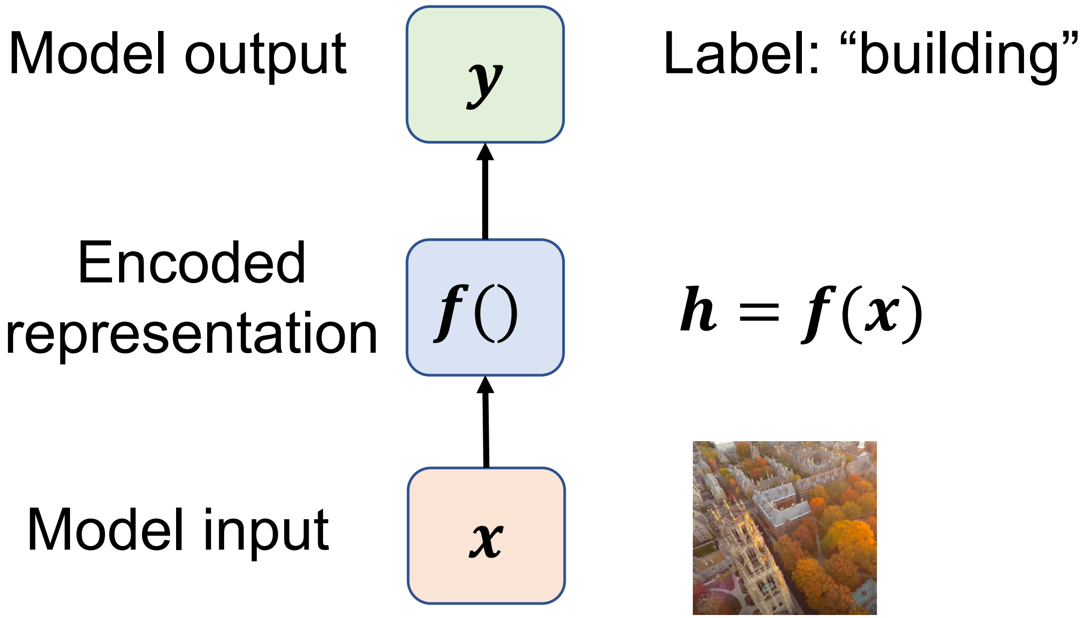
Sequence Learning

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

Domain	Sequence	Token	Structure
NLP	**Sentence:** [SOS, “Deep”, “learning”, “can”, “empower”, “sciences”, EOS]	**Word:** “learning” **Phrase:** [“Deep”, “learning”]	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.

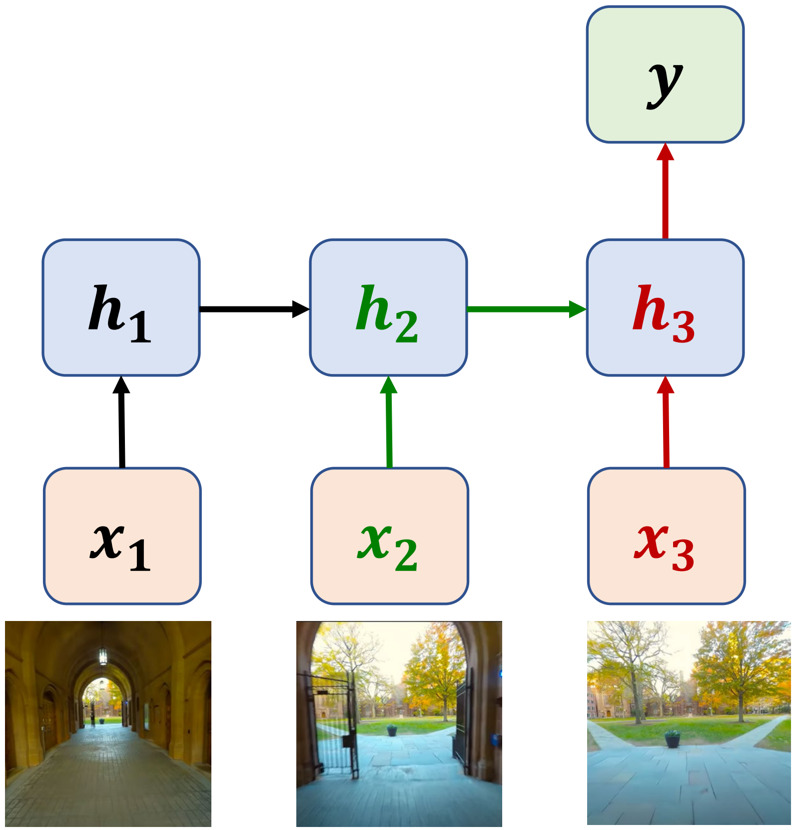


Sequence Learning — Application

- **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 $\boldsymbol{x_1}$ and $\boldsymbol{h_2} = \boldsymbol{f}(\boldsymbol{x_2}, \boldsymbol{h_1})$. Here $\boldsymbol{f}()$ is shared across all timesteps
  - $\boldsymbol{h_i} = \boldsymbol{f}(\boldsymbol{x_i}, \boldsymbol{h_{i-1}})$

Current token **Previous token**

Label: "tour"



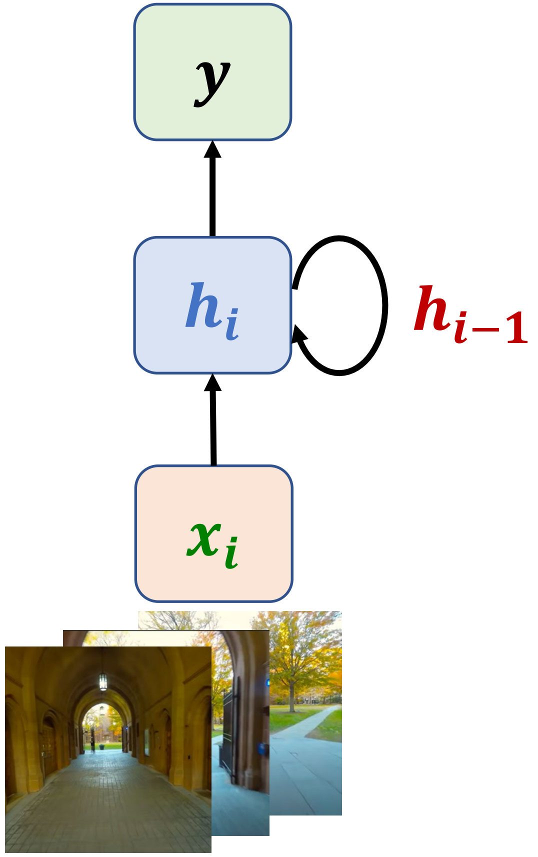
Sequence Learning — Application

- 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.

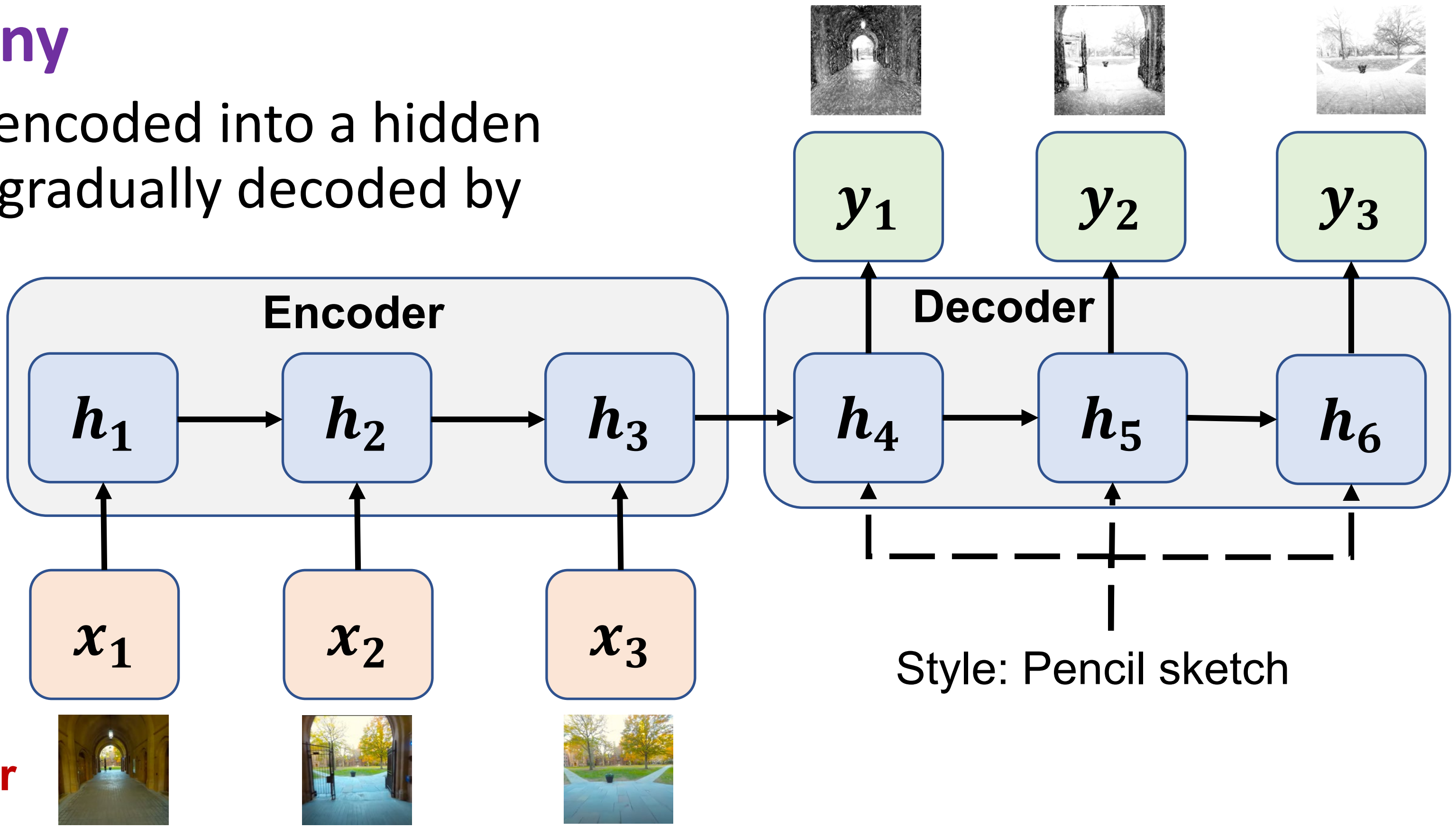
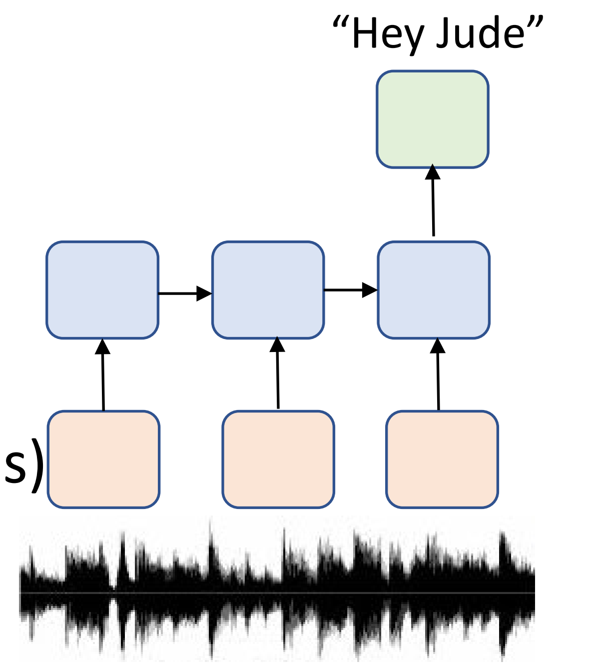


Diagram of video **style transfer**

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 (multiple words)
- **Many to many**
  - Translation: English to French
  - Time series: history to future
  - Graph autoregression



Speech classification
(many-to-one)

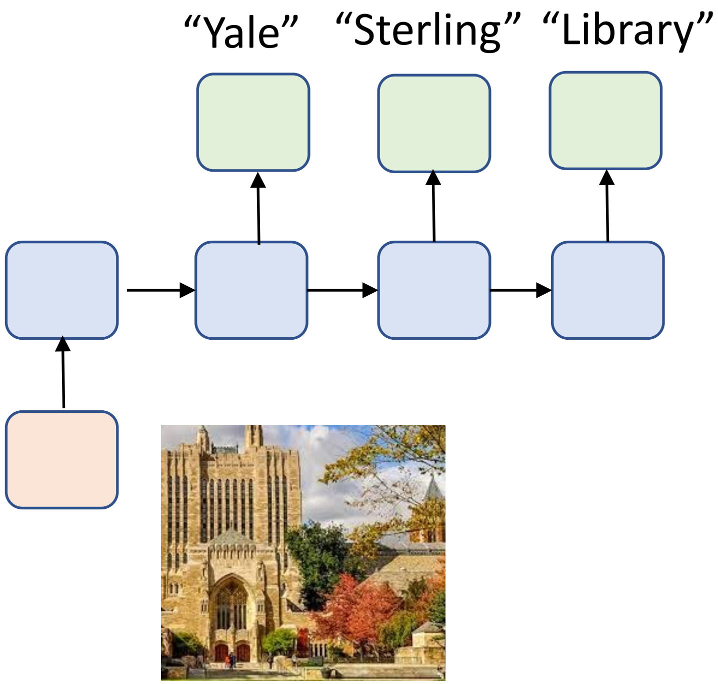
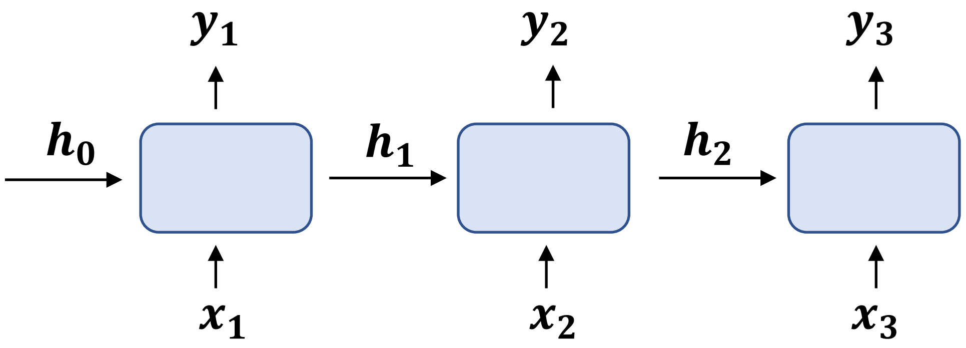


Image Captioning
(one-to-many)

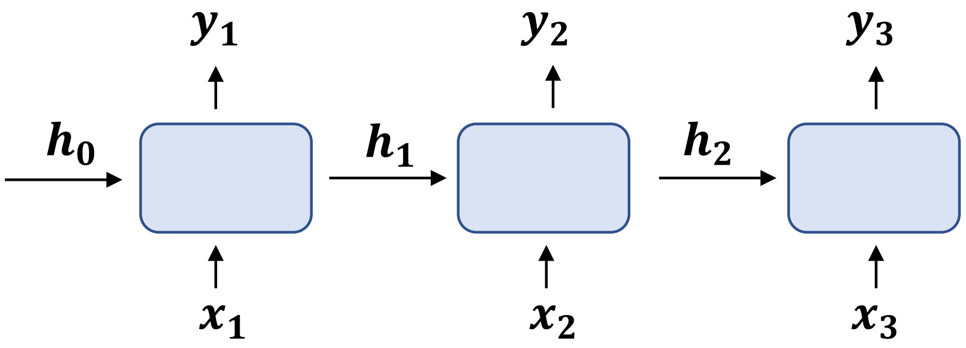
Sequence Learning



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

What are the issues and challenges of RNNs?

Sequence Learning

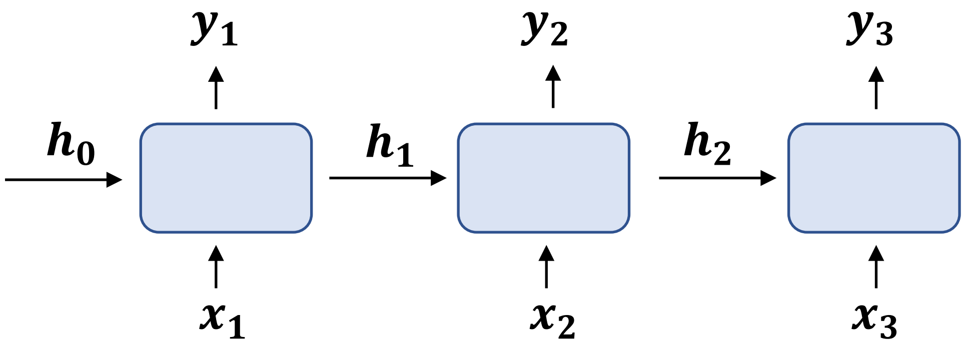


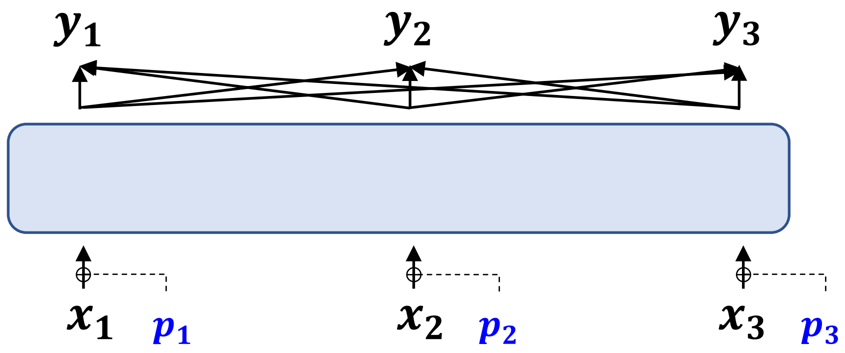
$$h_i = f_W\,(x_i\,, h_{i-1}),\;\; y_i = f_Y\,(h_i)$$

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





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

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

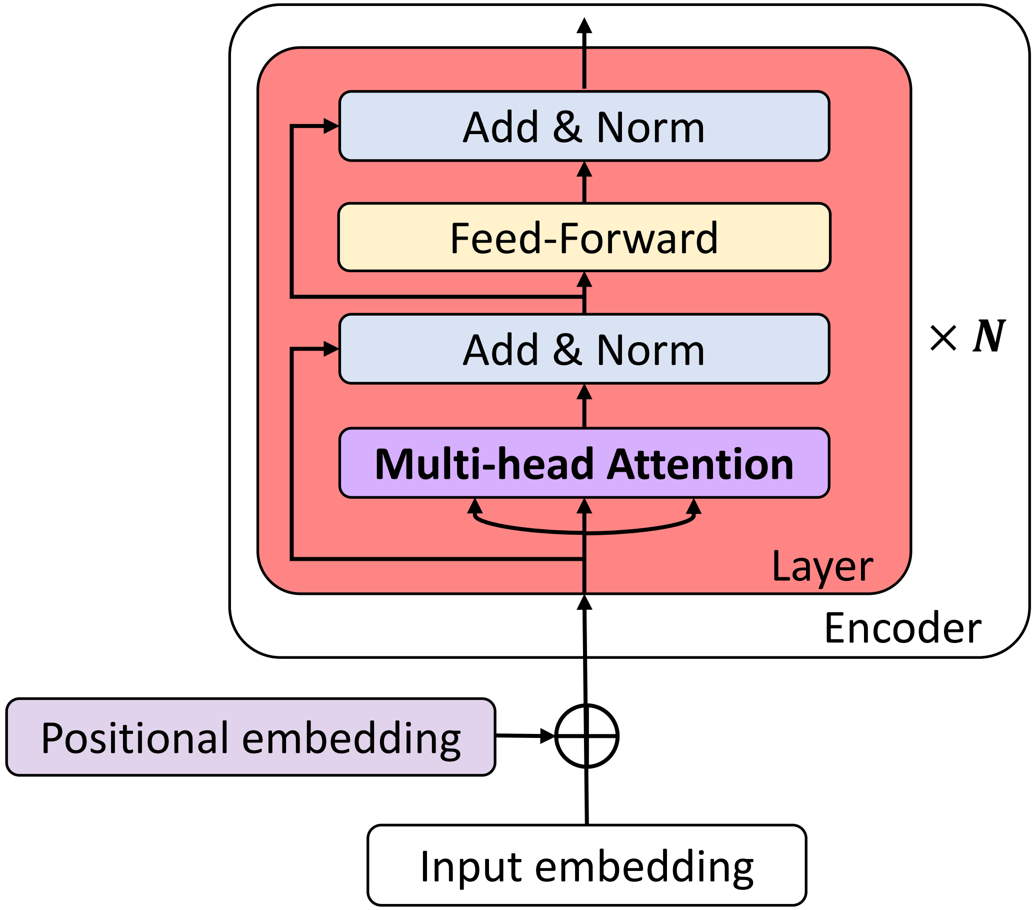
Problems of RNNs

1. parallelization
2. long sequences
3. only recurrence

Solutions by Transformers

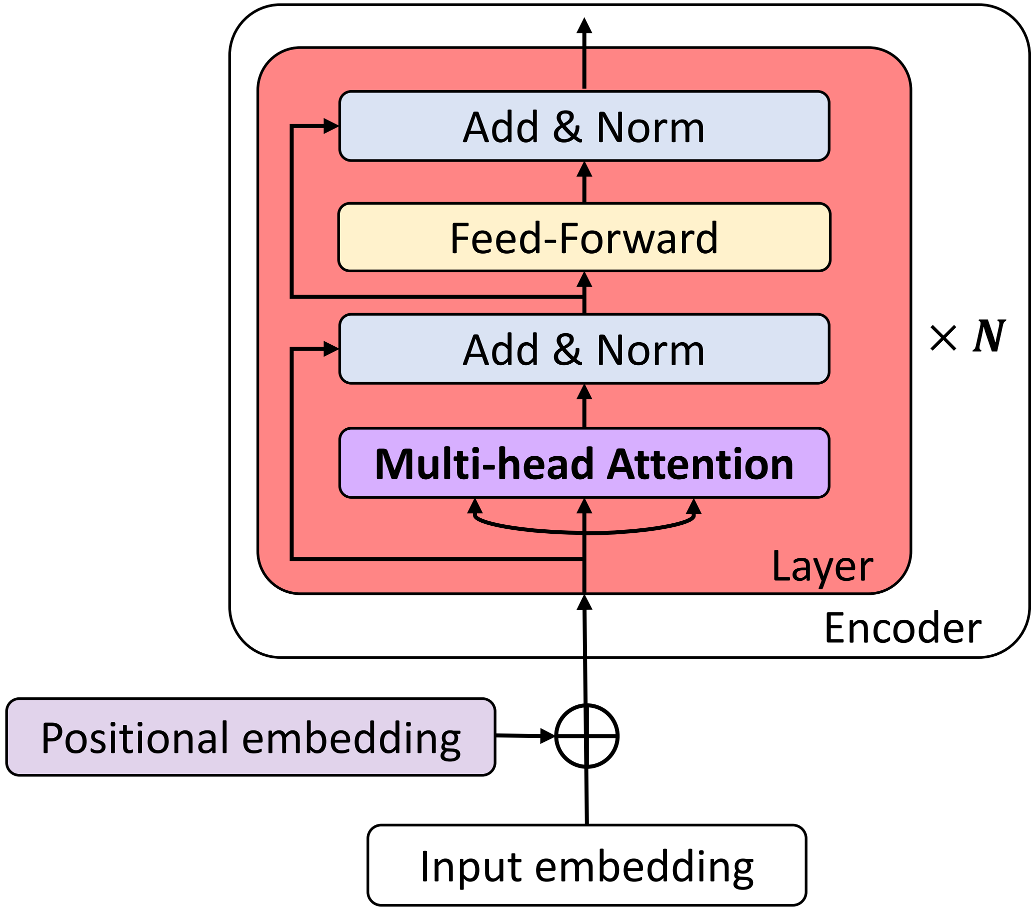
1. **Parallel input**: Input All tokens at the same time
2. **Self-Attention**: Enable attention in long-range
3. **Positional Embeddings p_i**: Model all possible correlations

Transformers — Overview



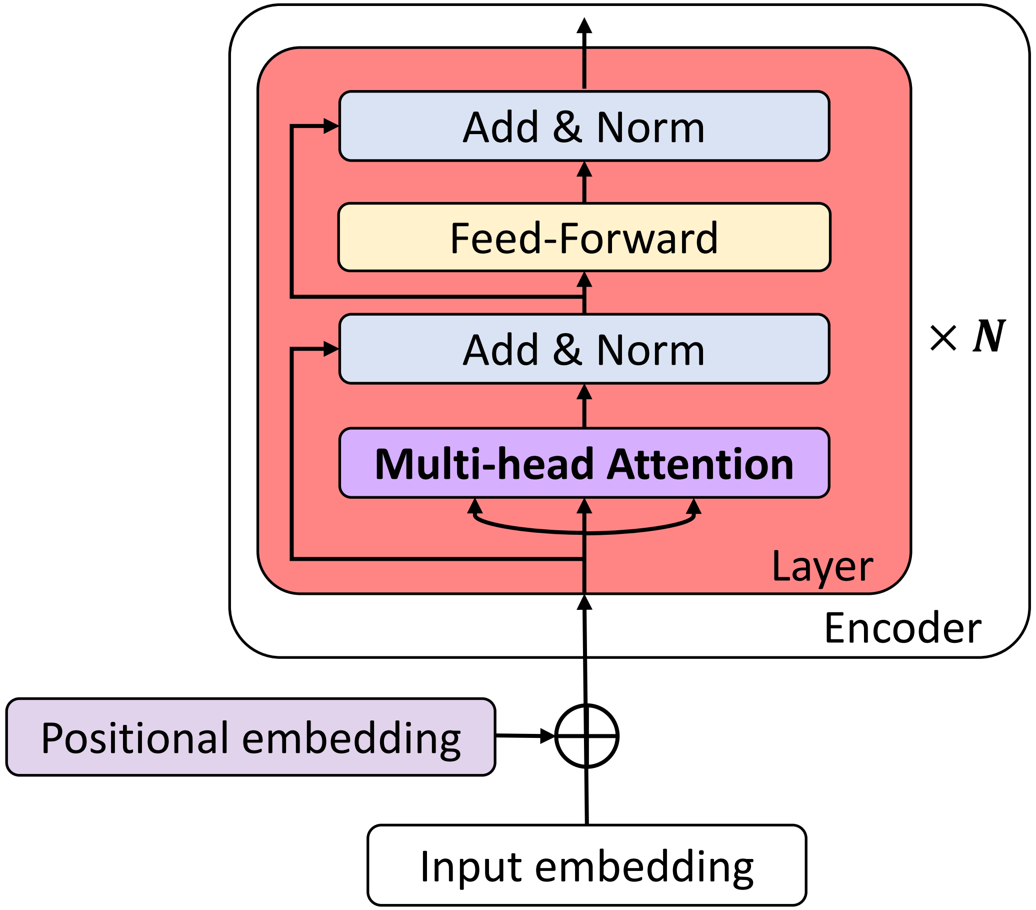
- **Original paper**: Attention is all you need [Vaswani et al., 2017].
- **Key component**: Multi-head self-attention
- **Other components** of a transformer layer: layer normalization, skip connection, position-wise feed-forward layer (FFN, or MLP)
- **Model usage**: Pre-training followed by fine-tuning. The transfered model can be:
  - **Encoder-only** (e.g BERT)
  - **Encoder-Decoder** (e.g BART)
  - **Decoder-only** (e.g GPT)
  - We will show an example later

Transformers — Overview



- **Model usage**: Pre-training followed by fine-tuning. The transfered model can be:
  - **Encoder-only** (e.g BERT)
    - Many-to-one classification / regression
    - Sentiment classification, document classification ...
    - Word / Sentence embeddings for downstream tasks (e.g. recommender system)
  - **Encoder-Decoder** (e.g BART)
  - **Decoder-only** (e.g GPT)
  - We will show an example later

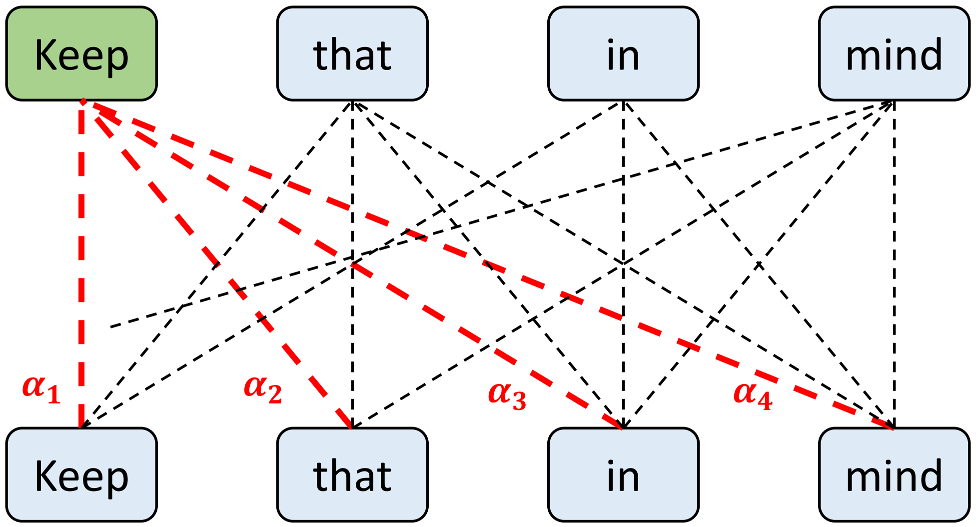
Transformers — Overview



- **Model usage**: Pre-training followed by fine-tuning. The transfered model can be:
  - **Encoder-only** (e.g BERT)
  - **Encoder-Decoder** (e.g BART)
    - Many-to-many use cases
    - Summarization, translation, style transfer ...
  - **Decoder-only** (e.g OpenAI GPT)
    - One-to-many use cases
    - Image / text / code generation, dialogue systems ...
    - GPT-3/4 based apps

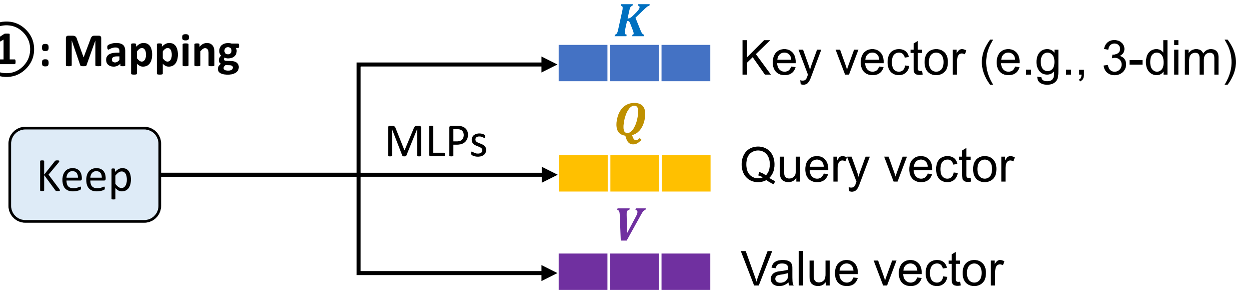
Transformers — Self-Attention (1/5)

Example:



input sequence

Step ①: Mapping



Step ②: Attention

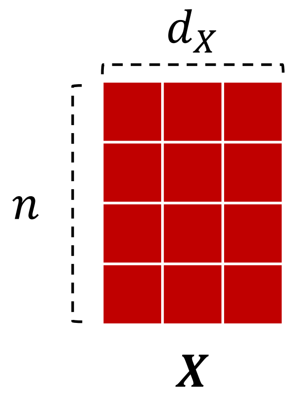
$$\alpha_1, \alpha_2, \alpha_3, \alpha_4 = \textbf{Softmax}\left(\underbrace{Q}_{\text{Keep}} \times \underbrace{K}_{\text{Keep}}\ \underbrace{K}_{\text{that}}\ \underbrace{K}_{\text{in}}\ \underbrace{K}_{\text{mind}}\right)$$

Step ③: Update

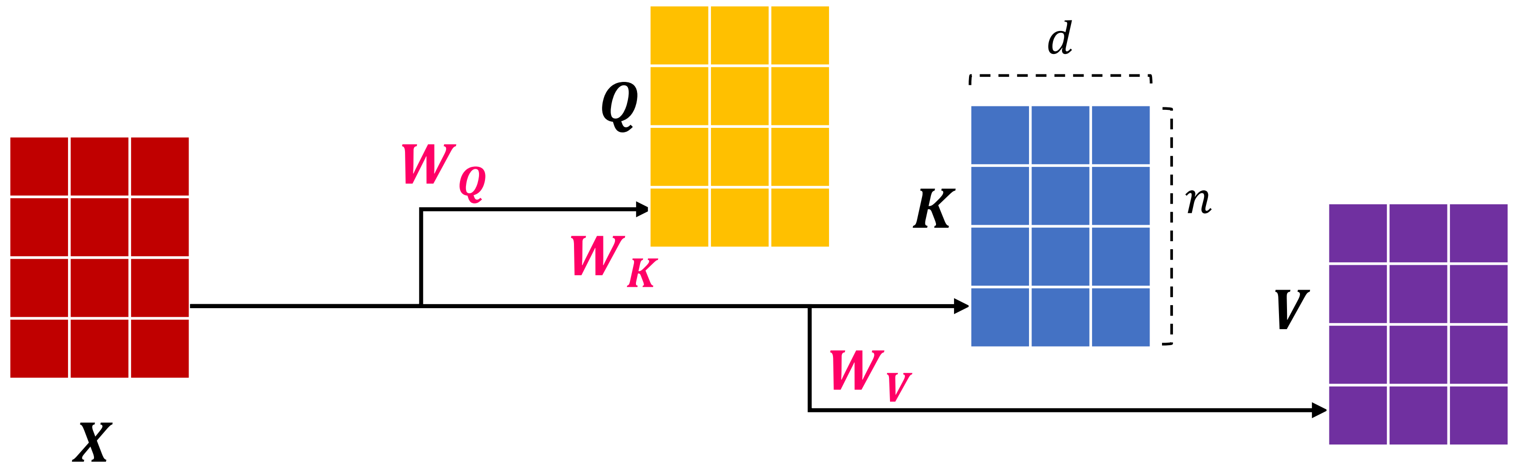
$$\underbrace{V'}_{\text{Keep}} = \alpha_1 \times \underbrace{V}_{\text{Keep}} + \alpha_2 \times \underbrace{V}_{\text{that}} + \alpha_3 \times \underbrace{V}_{\text{in}} + \alpha_4 \times \underbrace{V}_{\text{mind}}$$

Transformers — Self-Attention (2/5)

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

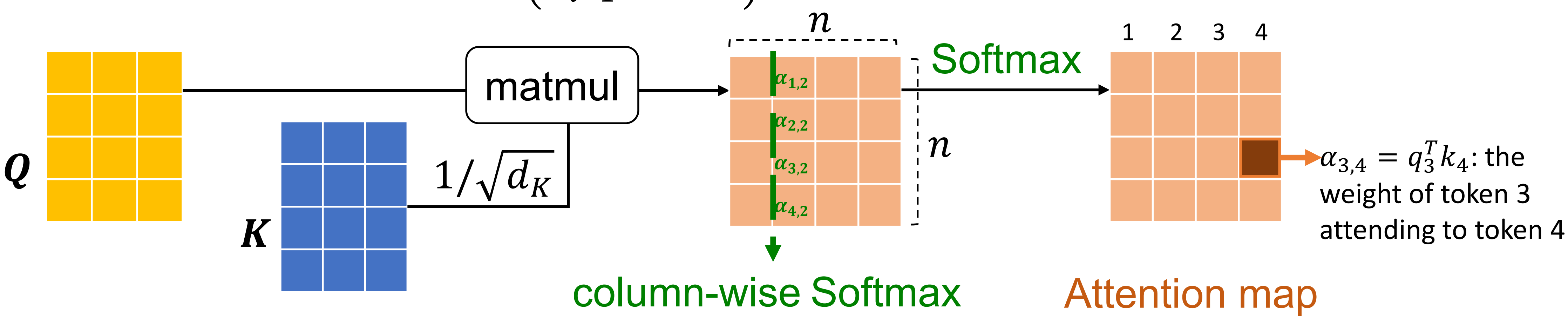


- Step ①: Query $\boldsymbol{Q} = \boldsymbol{X}\boldsymbol{W_Q}$, Key $\boldsymbol{K} = \boldsymbol{X}\boldsymbol{W_K}$, Value $\boldsymbol{V} = \boldsymbol{X}\boldsymbol{W_V}$
  - $W_K \in \mathbb{R}^{d_X \times d_K}$, and thus $\boldsymbol{K} \in \mathbb{R}^{n \times d_K}$
  - We require $d_K = d_Q$,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



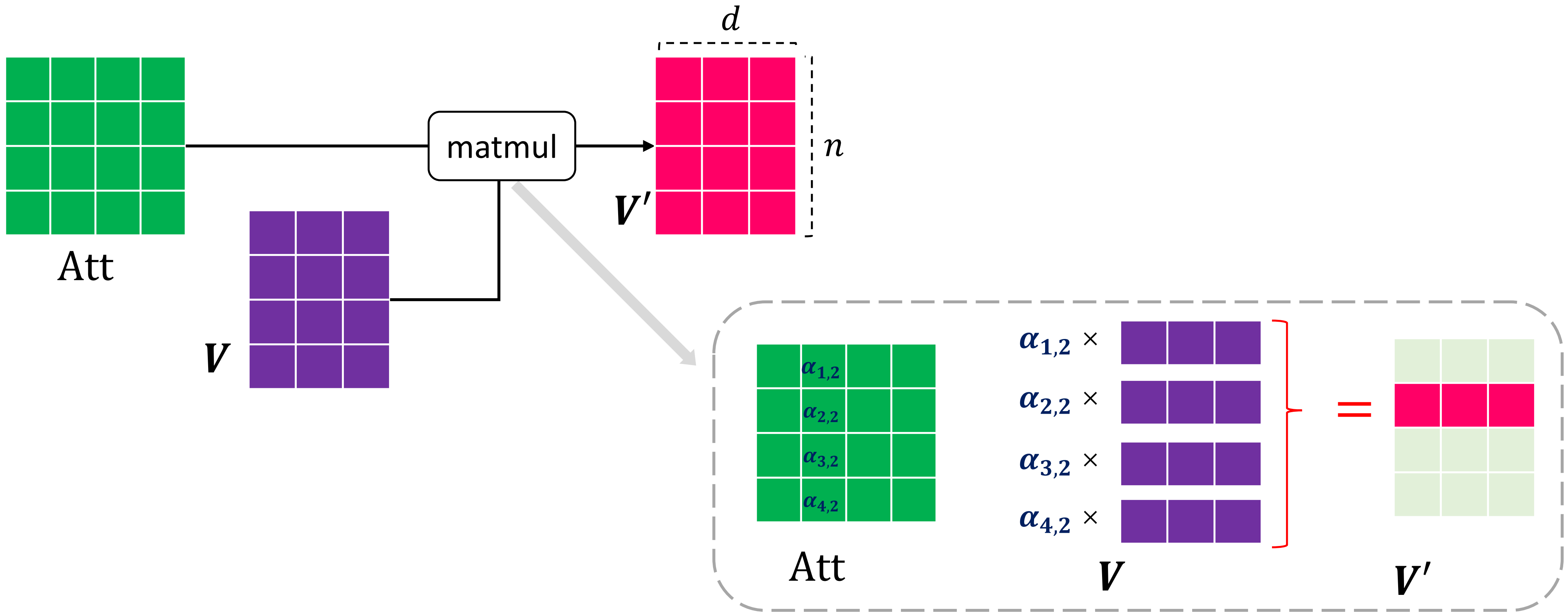
Transformers — Self-Attention (3/5)

- 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,
    - $q \sim N(0,\sigma^2), k \sim N(0,\sigma^2) \rightarrow \text{Var}\left(\sum_{i=1}^{d_K} q[i]k[i]\right) = \sigma^4 d_K$



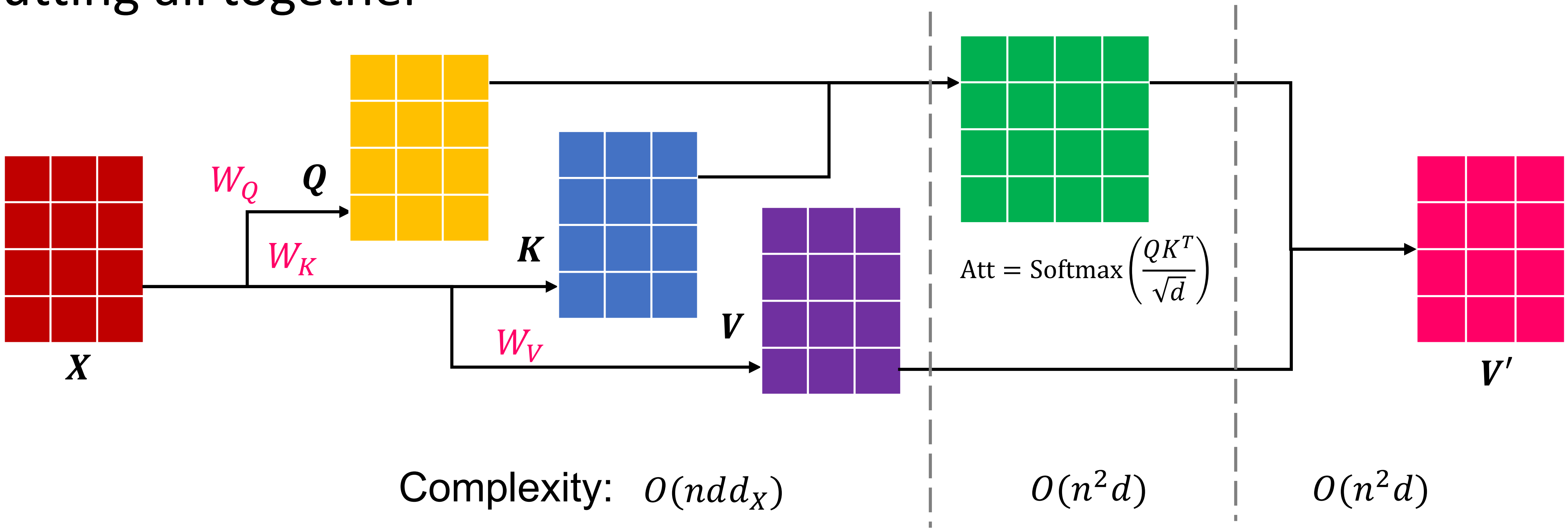
Transformers — Self-Attention (4/5)

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



Transformers — Self-Attention (5/5)

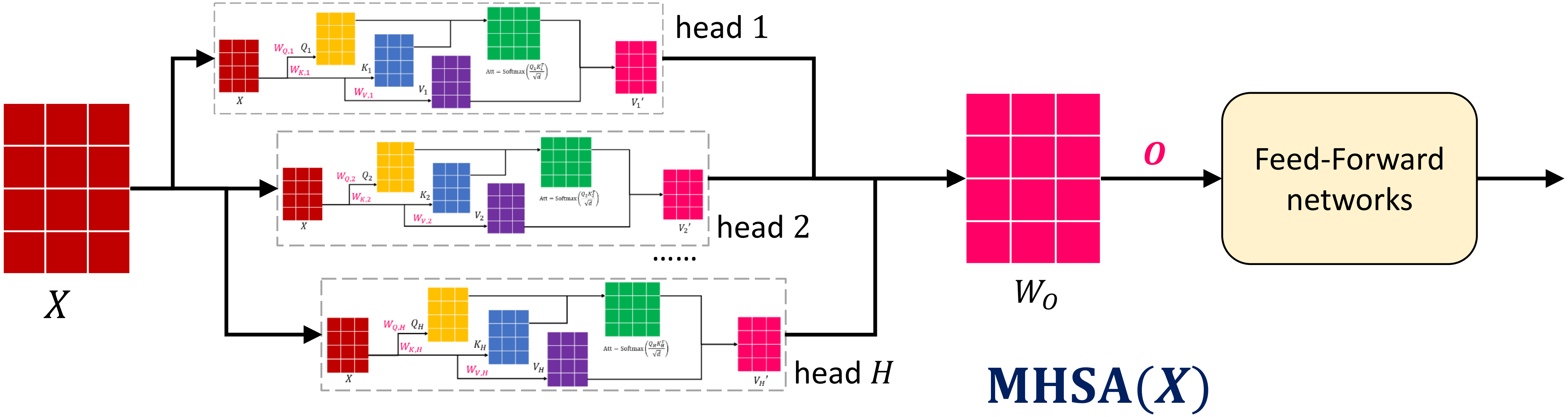
- Putting all together



The computation complexity is quadratic to number of tokens

Transformers — Multi-Head Self-Attention

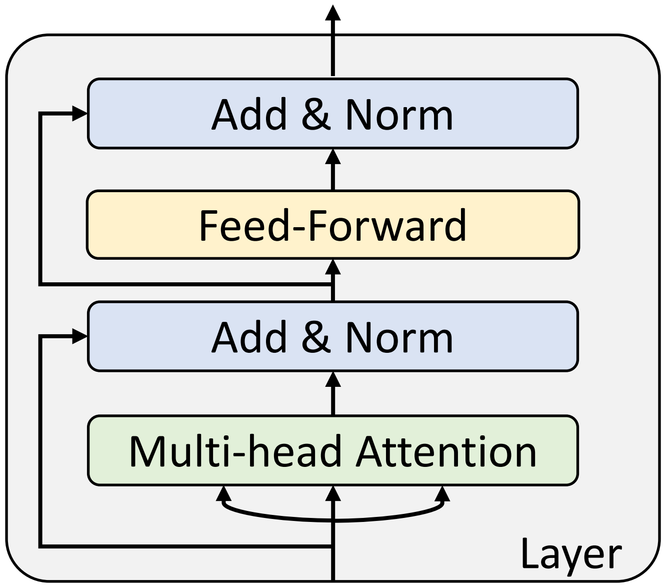
- There are usually **multiple aspects** that a token can attend to.
- We extend the attention to multiple heads, with multiple (Q, K, V) triplets on the same features.
  - The output of multi-head self-attention $O = \text{Concat}([V_1', V_2', \dots, V_H'])W_O$
  - Learnable parameters in each attention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



Transformers — Layer (1)

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

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

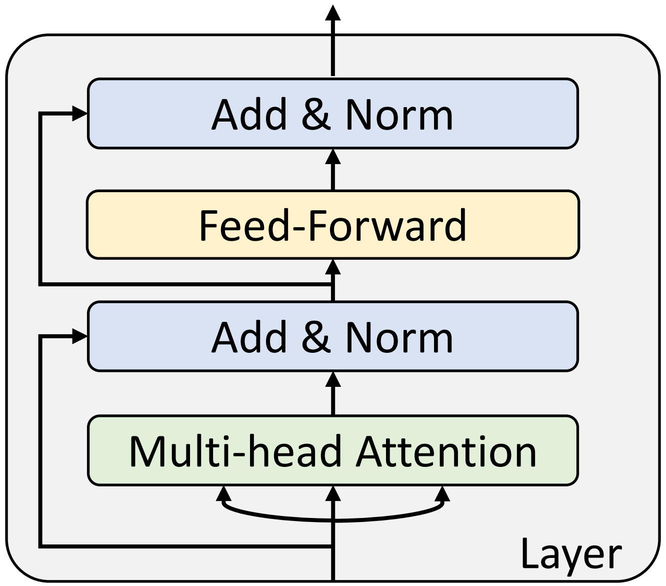


What are some advantages or challenges when trying to explain Transformer-based modes?

Transformers — Layer (2)

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

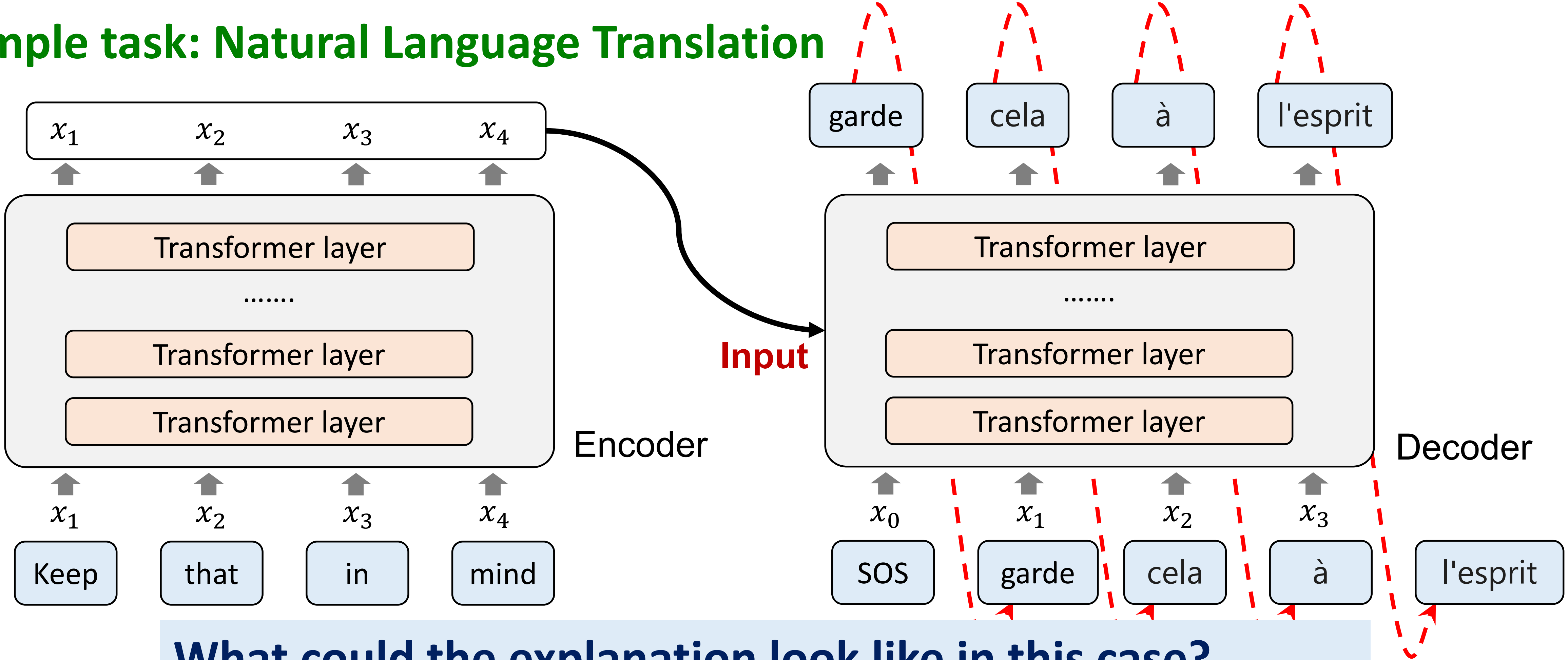
- **Layer Normalization** is used to enable faster training with small regularization and keep features in similar magnitudes.
  - BatchNorm isn't applied because batch size is usually small in Transformers due to GPU memory constraints. Besides, BatchNorm has been shown to lead to worse performance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



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

Transformers —Encoder / Decoder

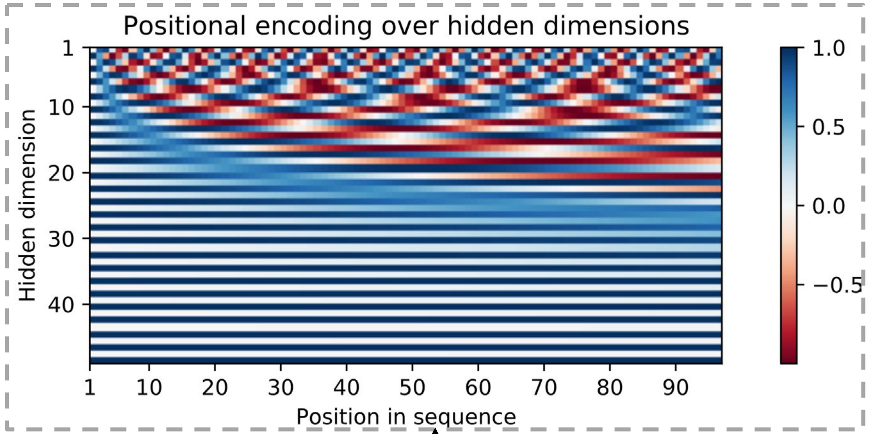
Example task: Natural Language Translation



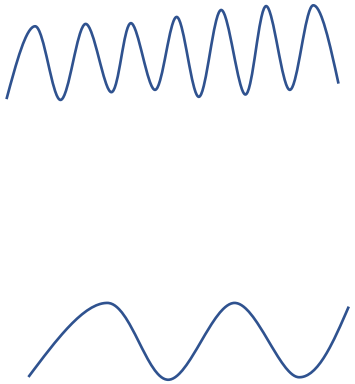
What could the explanation look like in this case?

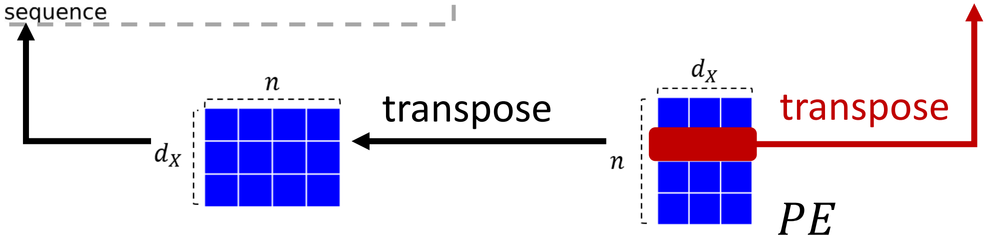
Transformers —Positional Encoding (3)

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

Outline of Today's Lecture

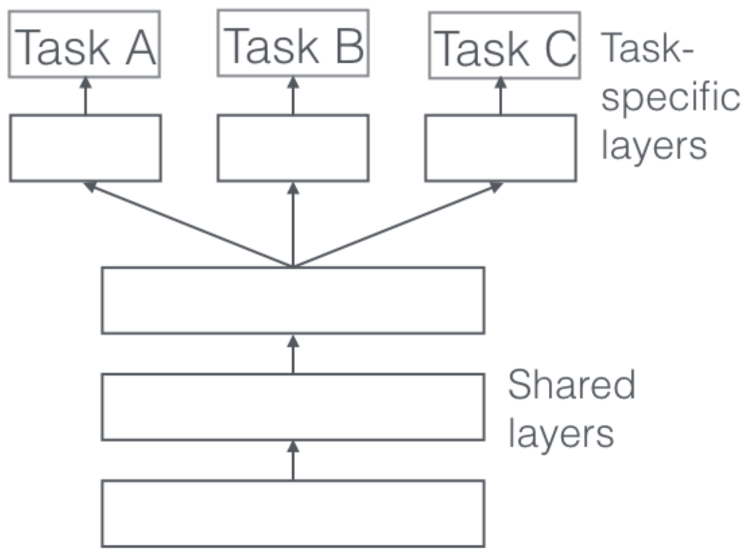
1. Self-Attention and Transformers

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 task



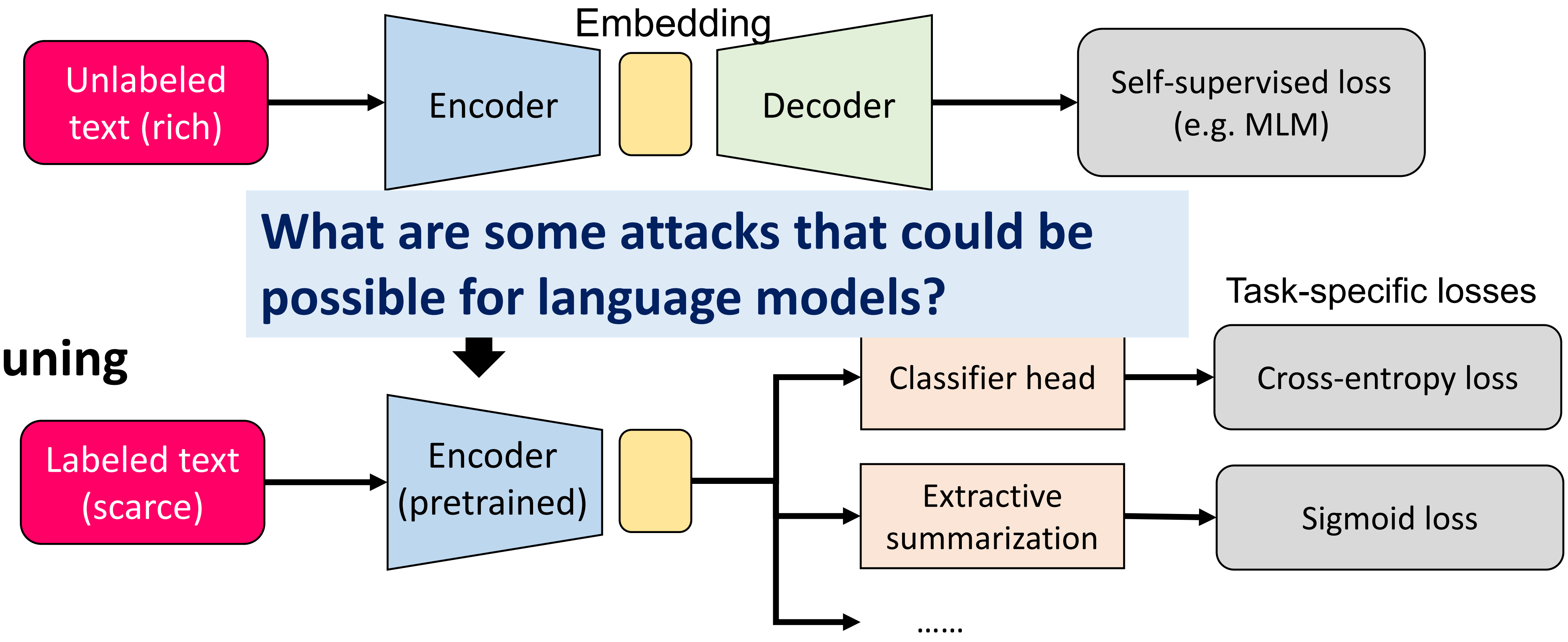
Label Scarcity

- ML models are hungry to data, especially labeled data for supervised task.
- The fast development of computer vision largely benefits from **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.
- **Solution**: **Pre-training** general-purpose language model on unlabeled large corpora (billions of characters) in **self-supervised** setting, then **fine-tuning** on smaller-scale tasks.

Pre-training and Fine-tuning

Encoder-only Pre-training Scenario

- **Pre-training**
- **Fine-tuning**



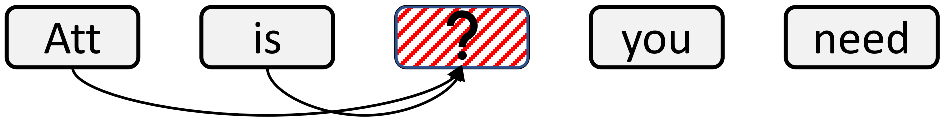
What are some attacks that could be possible for language models?

Transformers in NLP — BERT

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

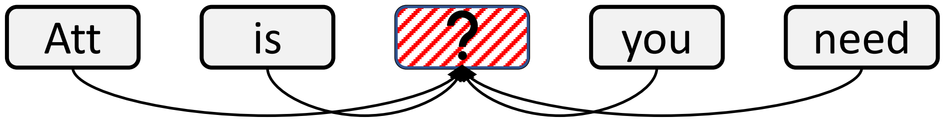
- **Pre-training task** (unsupervised): **Masked Language Model (MLM)**
  - First randomly masking $m\%$ tokens in the input sequence.
    - In BERT, 15% tokens are masked at random (replaced with the special `[MASK]` token)
  - Predicting masked tokens using remaining tokens.
  - Two modes: **Unidirectional**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

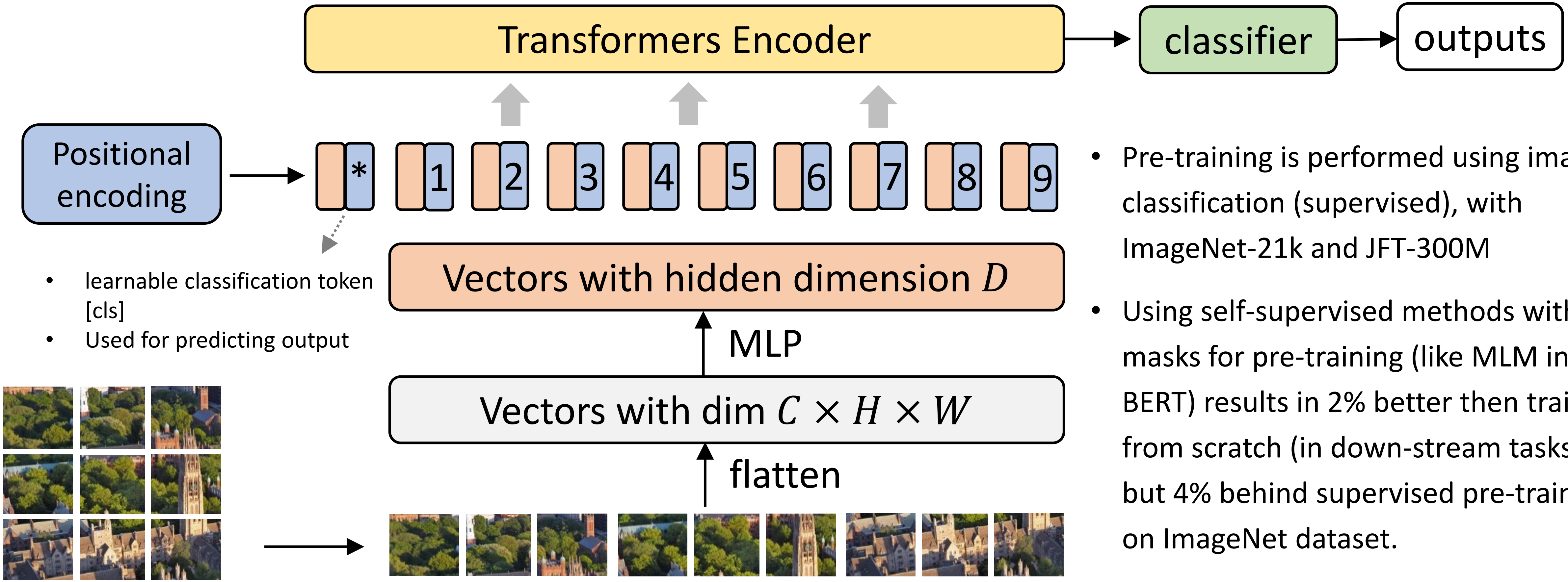
Transformers in NLP — RoBERTa

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.

Outline of Today's Lecture

1. Self-Attention and Transformers

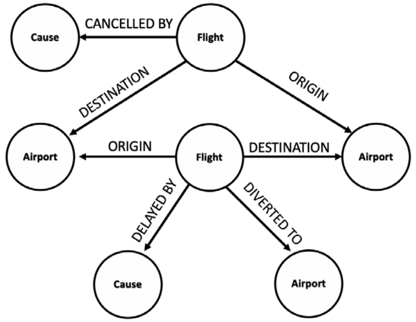
2. Transformers Applications

3. **Graph Neural Networks**

Types of Networks and Graphs (1)

- **Networks (also known as Natural Graphs):**
  - **Social networks:**
    - **Society** is a collection of 7+ billion individuals
  - **Communication and transactions:**
    - Electronic devices, phone calls, financial transactions
  - **Biomedicine:**
    - Interactions between **genes/proteins** regulate life
  - **Brain connections:**
    - Our **thoughts** are hidden in the connections between billions of neurons

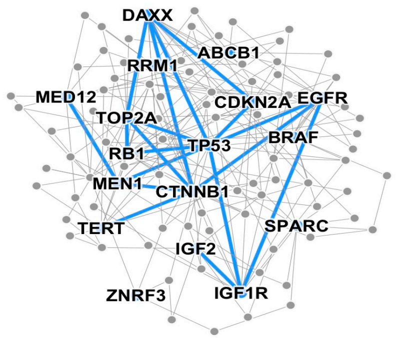
Many Types of Data are Graphs (1)



Event Graphs

Image credit: SalientNetworks

Computer Networks



Disease Pathways

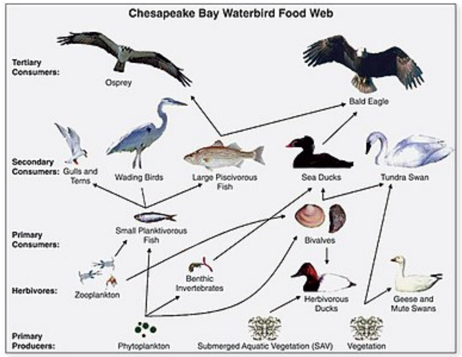


Image credit: Wikipedia

Food Webs

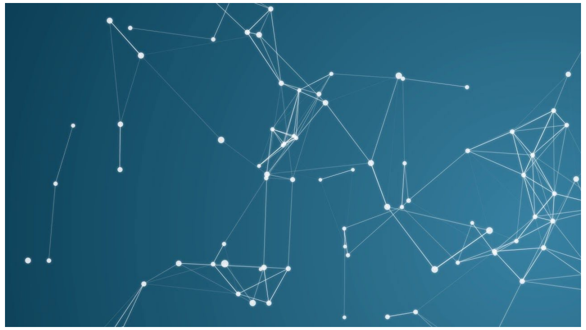


Image credit: Pinterest

Particle Networks

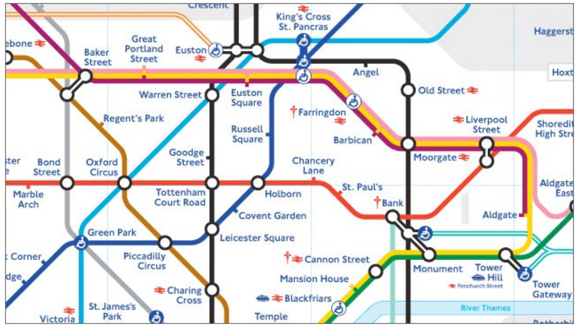


Image credit: visitlondon.com

Underground Networks

Many Types of Data are Graph (2)

Image credit: Medium

Social Networks

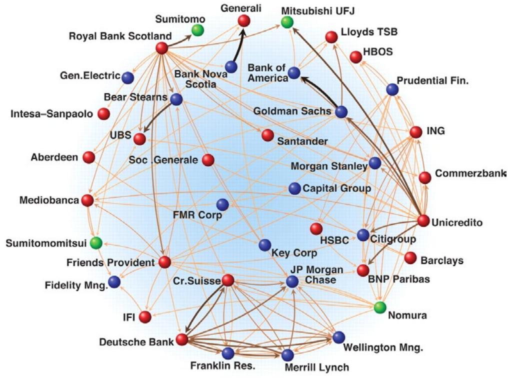
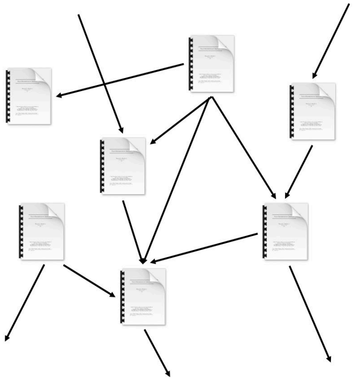


Image credit: Science

Economic Networks

Image credit: Lumen Learning

Communication Networks



Citation Networks

Image credit: Missoula Current News

Internet

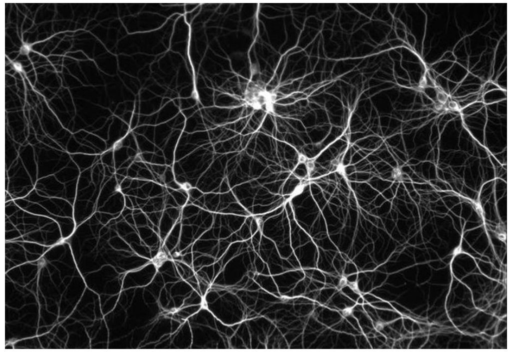
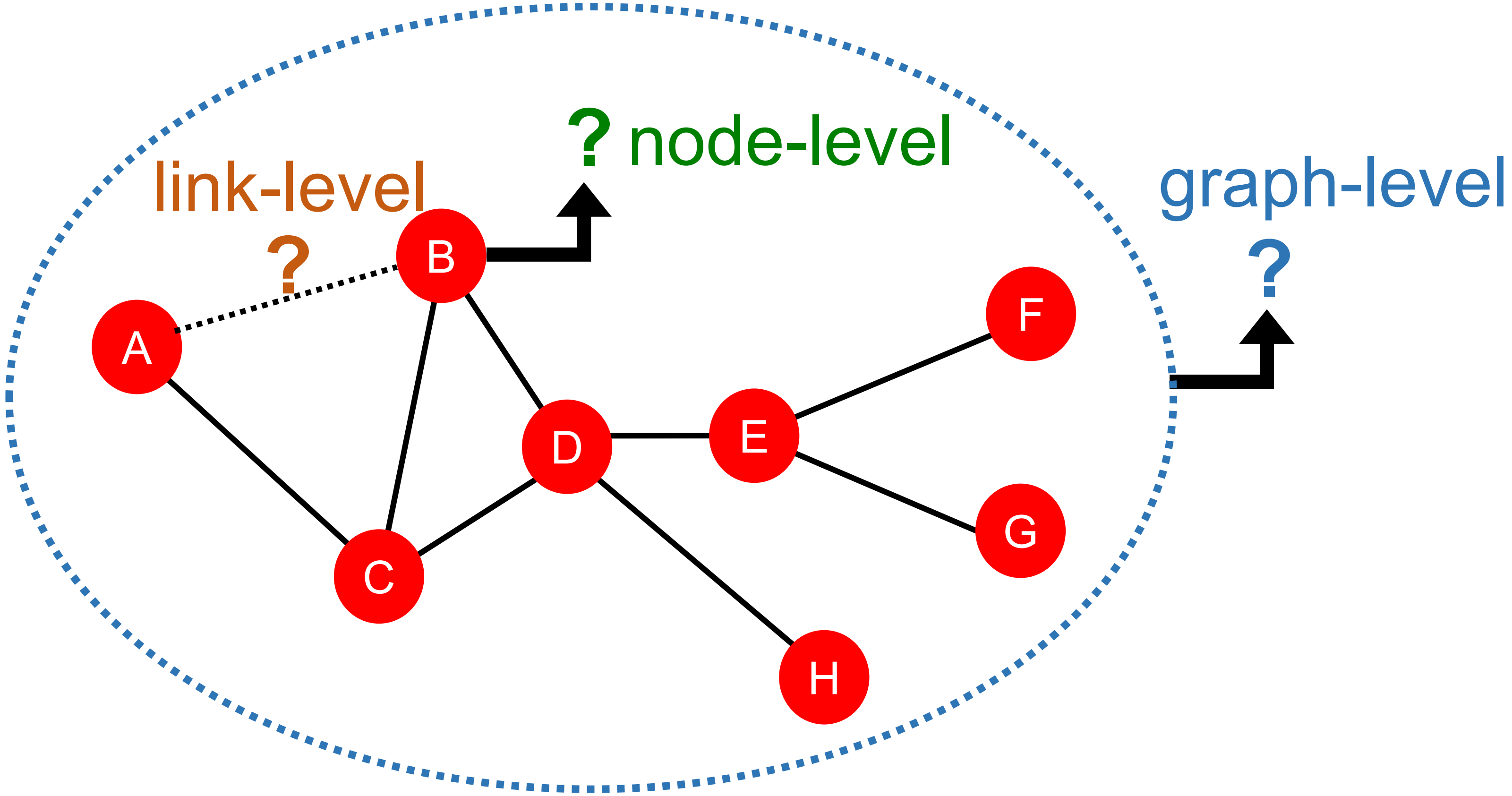


Image credit: The Conversation

Networks of Neurons

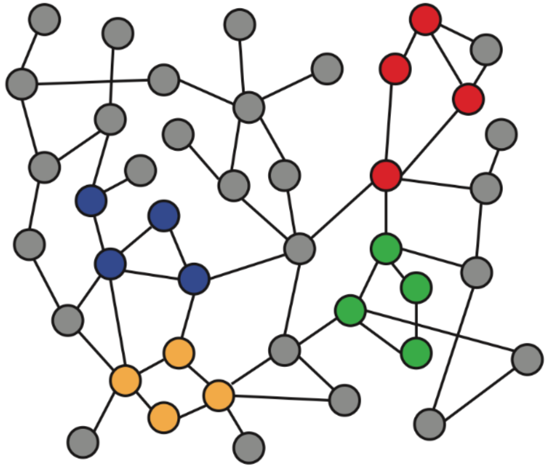
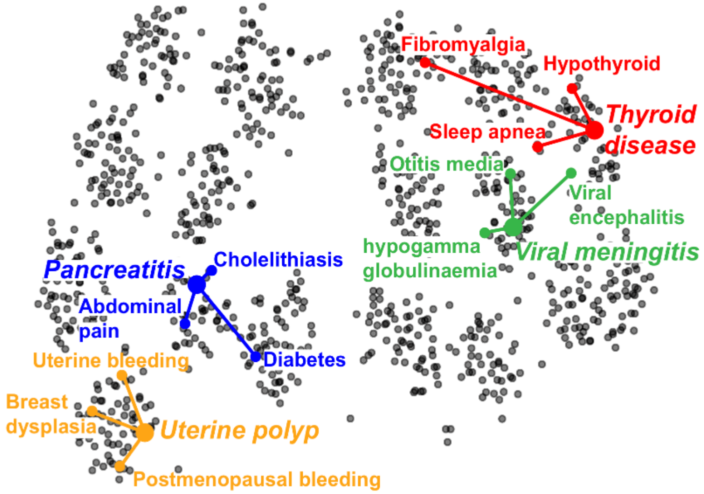
Graph Machine Learning Tasks: Overview

- Node-level prediction
- Link-level prediction
- Graph-level prediction
- Graph generation
    - Generative model



Node Embeddings

- Intuition: Map nodes to d-dimensional **embeddings** (which are "**representations**" of nodes) such that similar nodes in the graph are embedded close together

$f($  $) =$ 

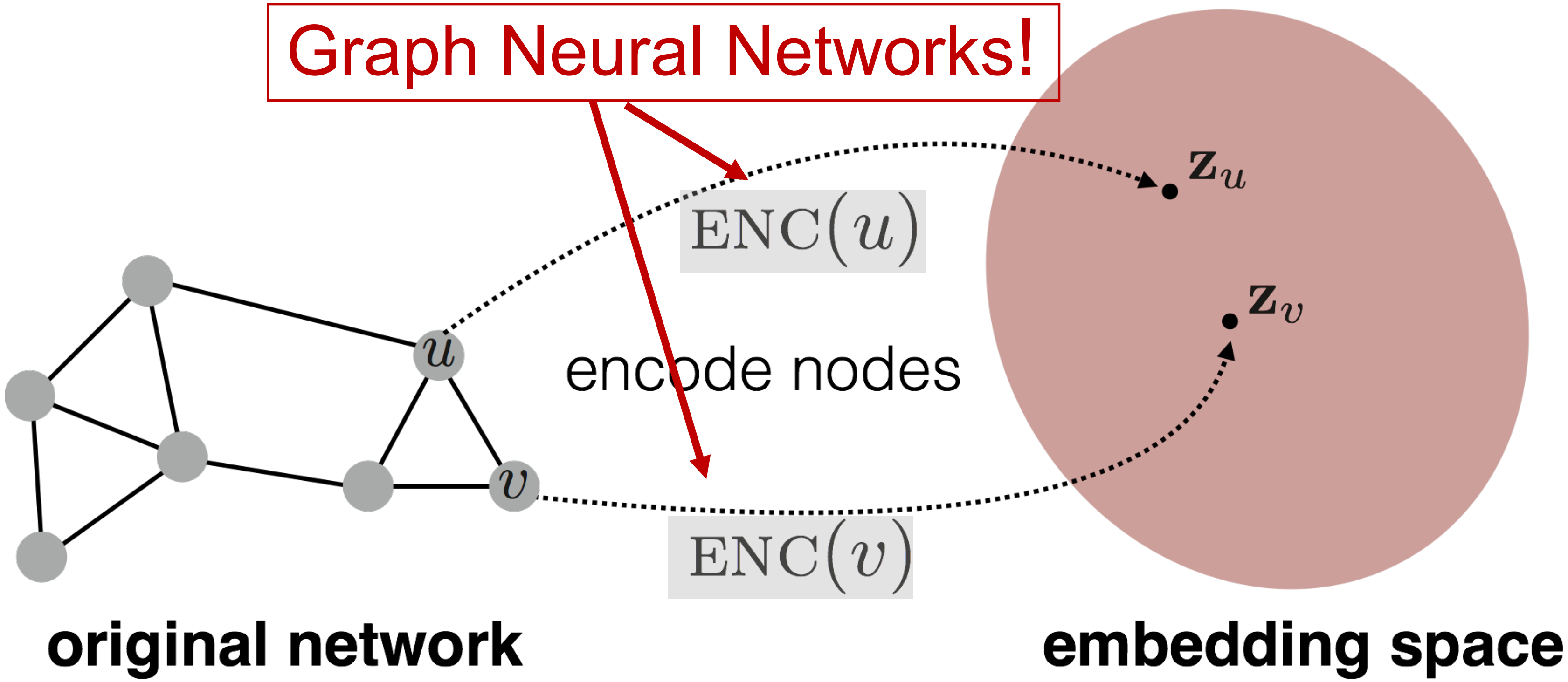
Input graph

2D node embeddings

How to learn the mapping function f?

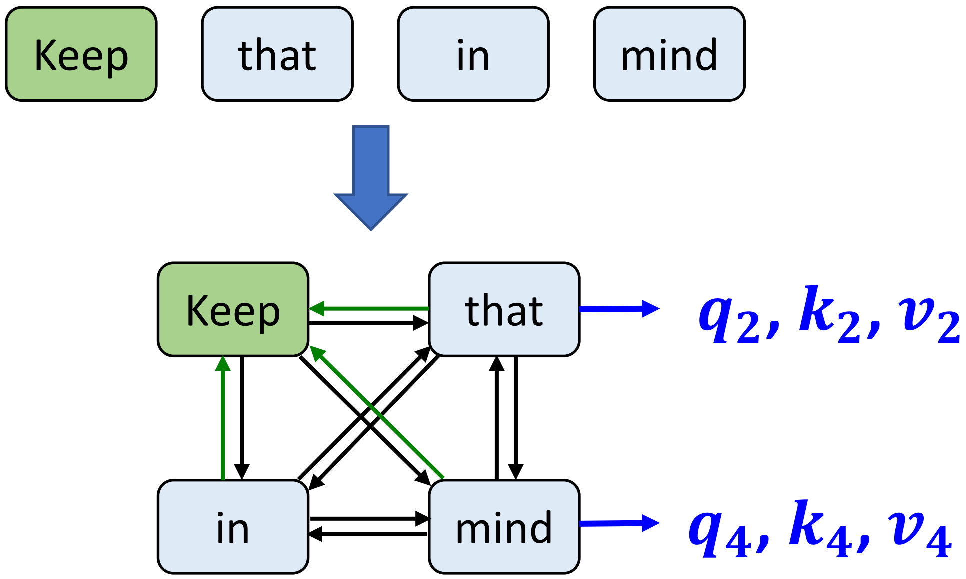
Deep Graph Encoders (1)

$\mathrm{ENC}(\cdot) =$ multiple layers of non–linear transformations based on graph structures

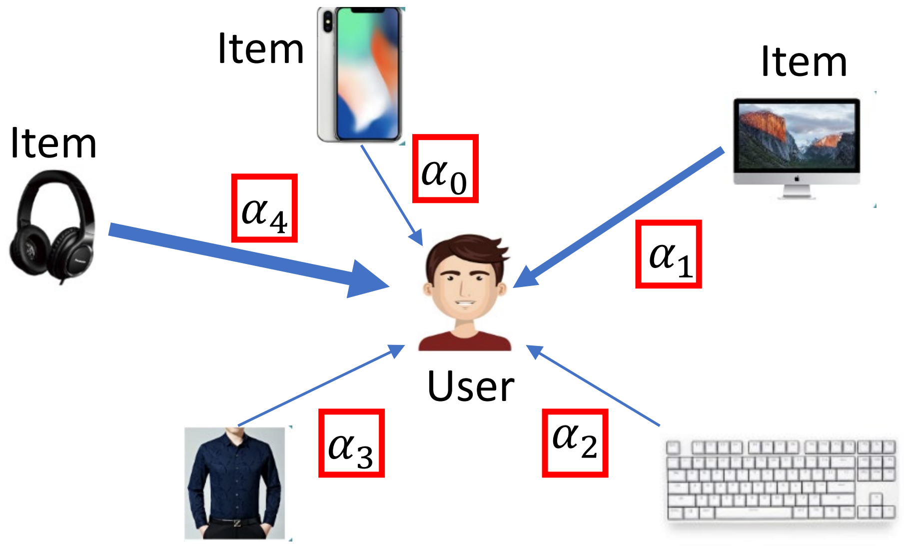


Transformers — in the Language of Graphs (1)

Transformers



GATs



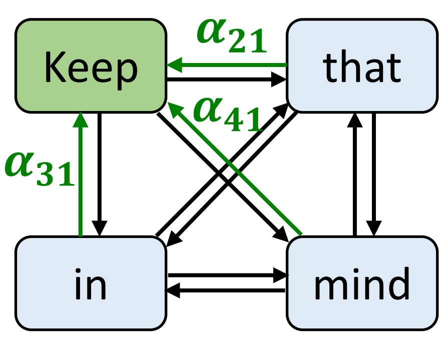
Step ① Mapping: Each node feature x_i is projected to q_i, k_i, v_i.

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)

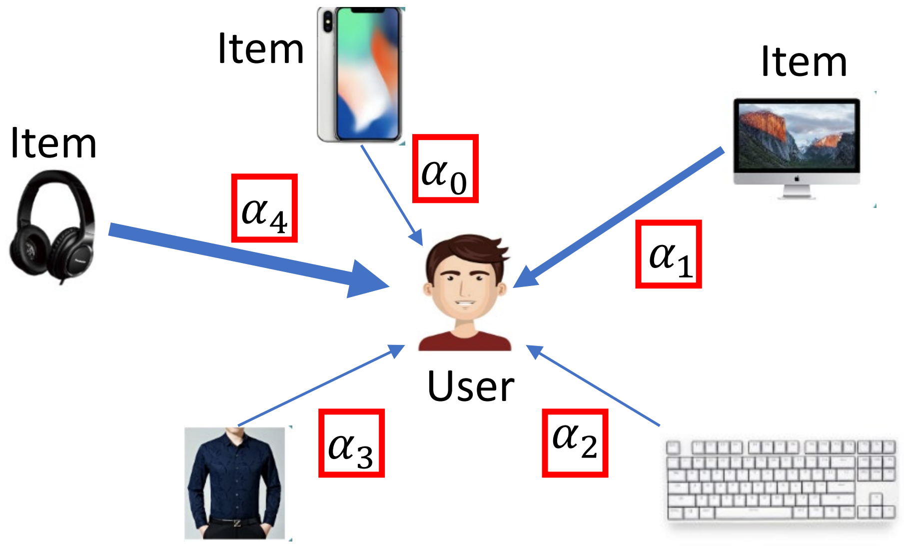
Transformers



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

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

GATs

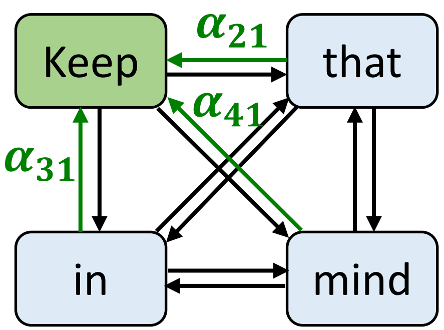


Message computing: transform information of neighbor node to a message

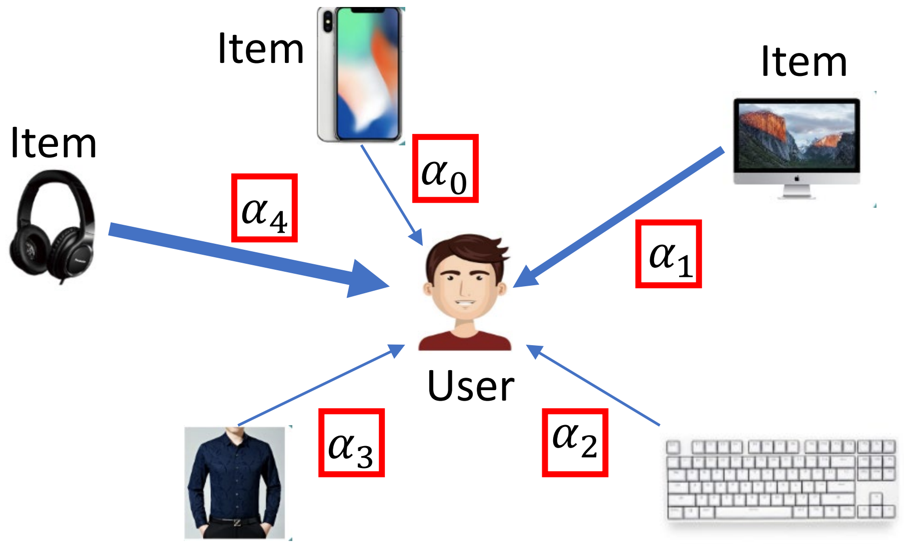
$$\mathbf{m}_u^{(l)} = \alpha_{vu} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)}, u \in N_v$$

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 $\mathrm{LeakyReLU}$

Graph Attention — in the Language of Transformer

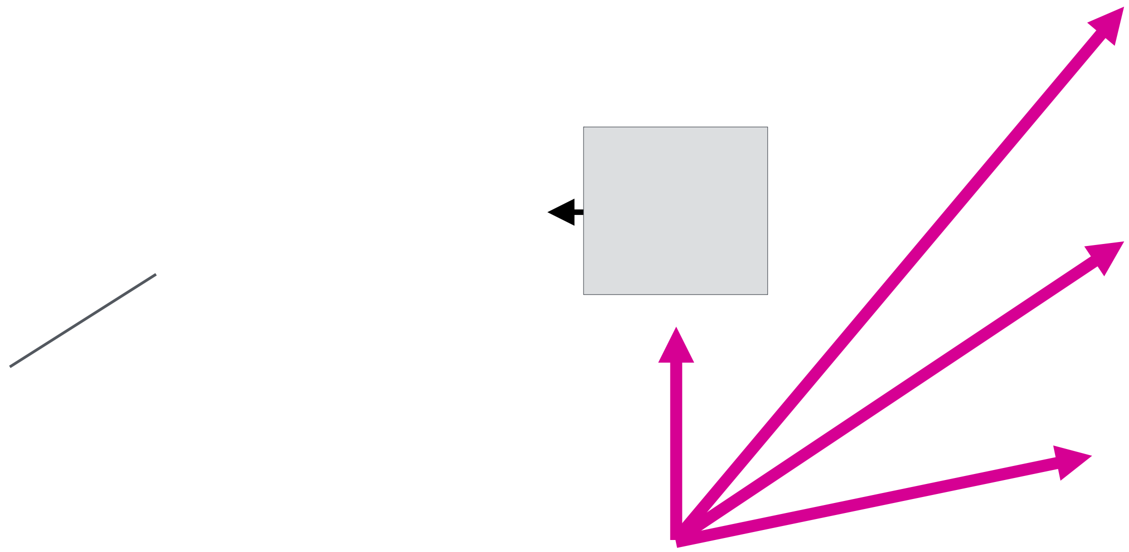
- The above computations do not require the assumption of **the complete graph**.
  - We assume full connectivity, mostly because we do not want to miss any potential token correlations.
- Self-attention can be easily adapted to graph-structured input data where the token correlations are given by the **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 is linear to the edge number $O(E)$

Idea: Aggregate Neighbors (1)

- **Key idea:** Generate node embeddings based on **local network neighborhoods**

Idea: Aggregate Neighbors (2)

- **Intuition:** Nodes aggregate information from their neighbors using neural networks

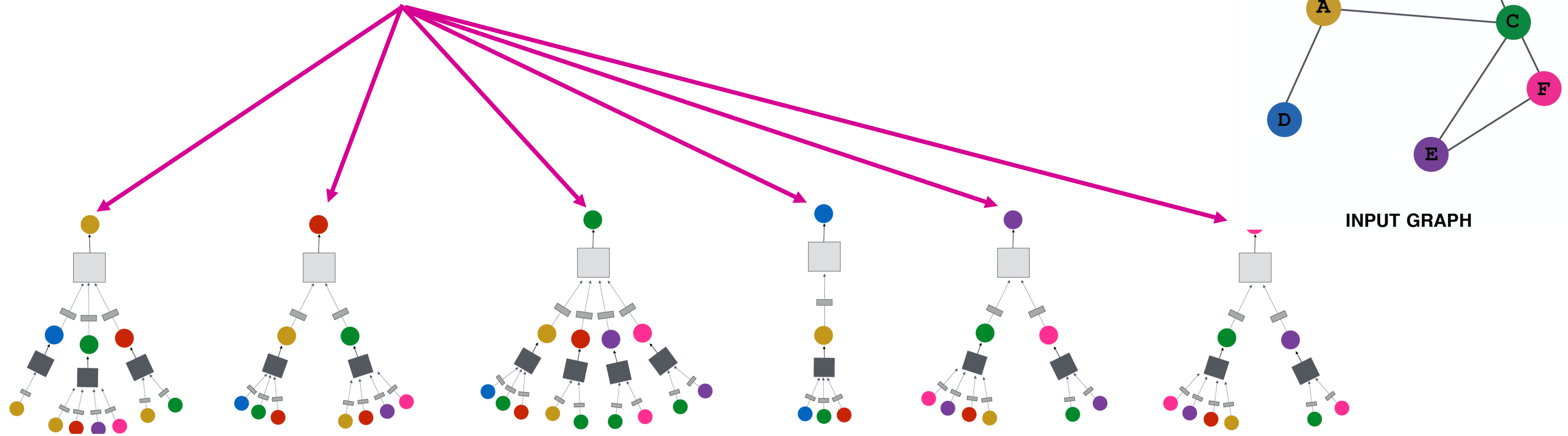


Neural networks

Idea: Aggregate Neighbors (3)

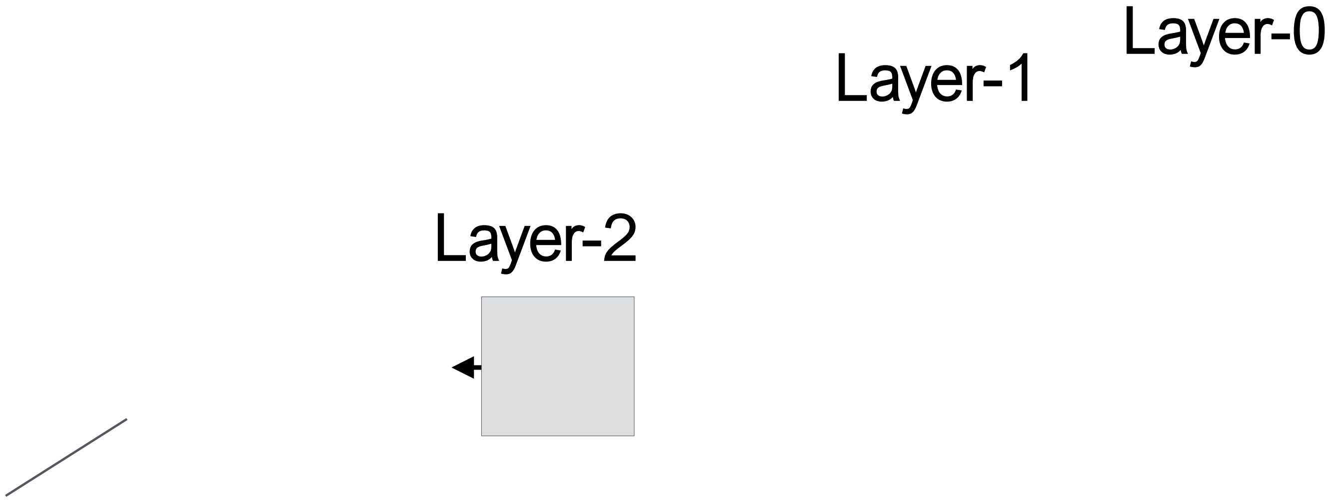
- **Intuition:** Network neighborhood defines a computation graph

Every node defines a computation graph based on its neighborhood!



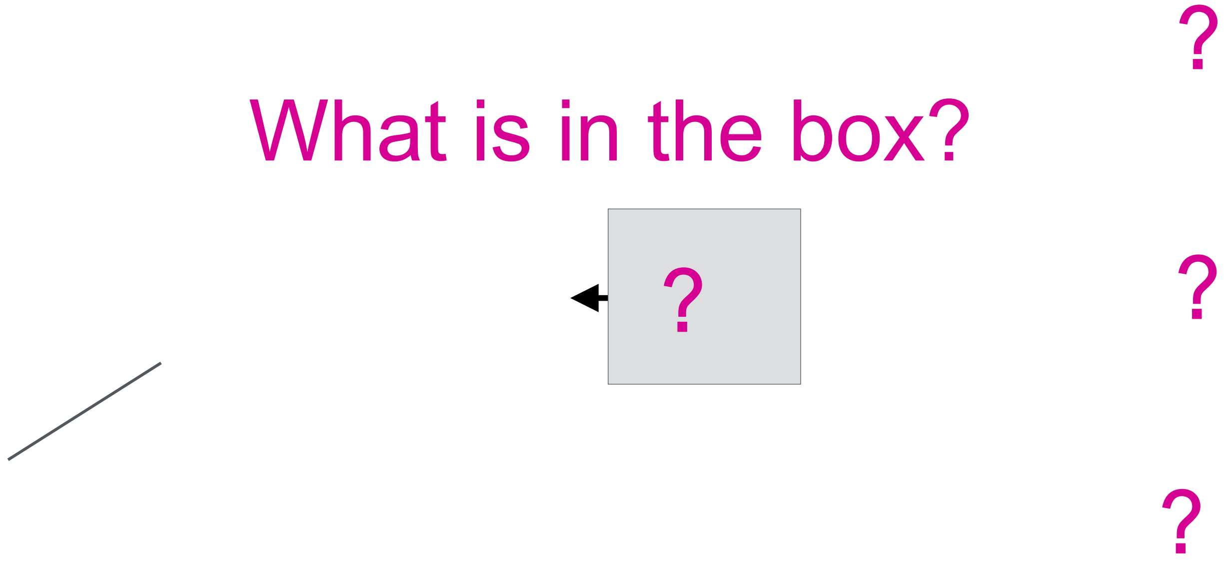
Deep Model: Many Layers

- Model can be of arbitrary depth:
  - Nodes have embeddings at each layer
  - Layer-0 embedding of node u is its input feature, $\boldsymbol{x}_u$
  - Layer-k embedding gets information from nodes that are k hops away



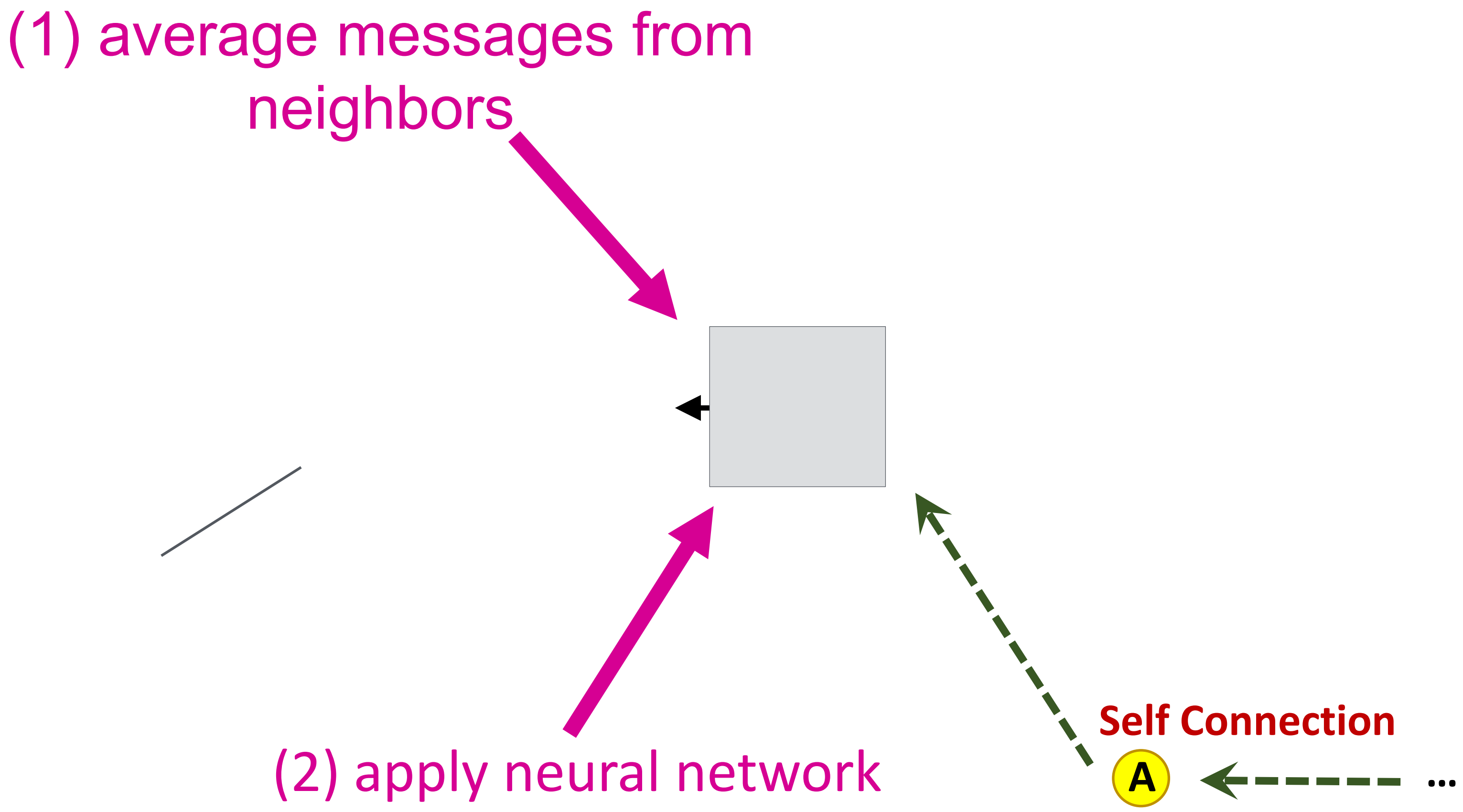
Neighborhood Aggregation (1)

- **Neighborhood aggregation:** Key distinctions are in how different approaches aggregate information across the layers



Neighborhood Aggregation (2)

- **Basic approach:** Average information from neighbors and apply a neural network



Setup

- **Assume we have a graph G:**
  - V is the **vertex set**
  - $\boldsymbol{A}$ is the **adjacency matrix** (assume binary)
  - $\boldsymbol{X} \in \mathbb{R}^{d\times|V|}$ is a matrix of **node features**
  - v: a node in V; $N(v)$: the set of neighbors of v.
  - **Node features:**
    - Social networks: User profile, User image
    - Biological networks: Gene expression profiles, gene functional information
    - When there is no node feature in the graph dataset:
      - Indicator vectors (one-hot encoding of a node)
      - Vector of constant 1: [1, 1, …, 1]

The Math: Deep Encoder

- **Basic approach:** Average neighbor messages and apply a neural network

Initial 0-th layer embeddings are equal to node features

$$\boxed{h_v^0 = x_v}$$

embedding of v at layer l

$$\mathrm{h}_v^{(l+1)} = \boxed{\sigma}\left(\mathrm{W}_l \boxed{\sum_{u \in \mathrm{N}(v)} \frac{\mathrm{h}_u^{(l)}}{|\mathrm{N}(v)|}} + \mathrm{B}_l \boxed{\mathrm{h}_v^{(l)}}\right), \forall l \in \{0, \dots, \boxed{L} - 1\}$$

$$\boxed{z_v} = h_v^{(L)}$$

Weighted sum for each u's message to v

tal number
of layers

Embedding after L layers of neighborhood aggregation

Non-linearity (e.g., ReLU)