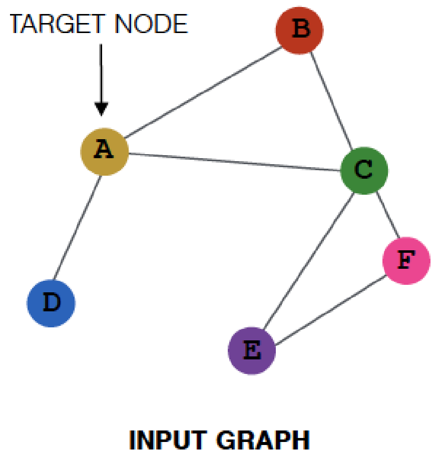
# IT5429E-1-24 (24.1A01)(Fall 2024): Graph Analytics for Big Data

# Week 10: Graph Transformer

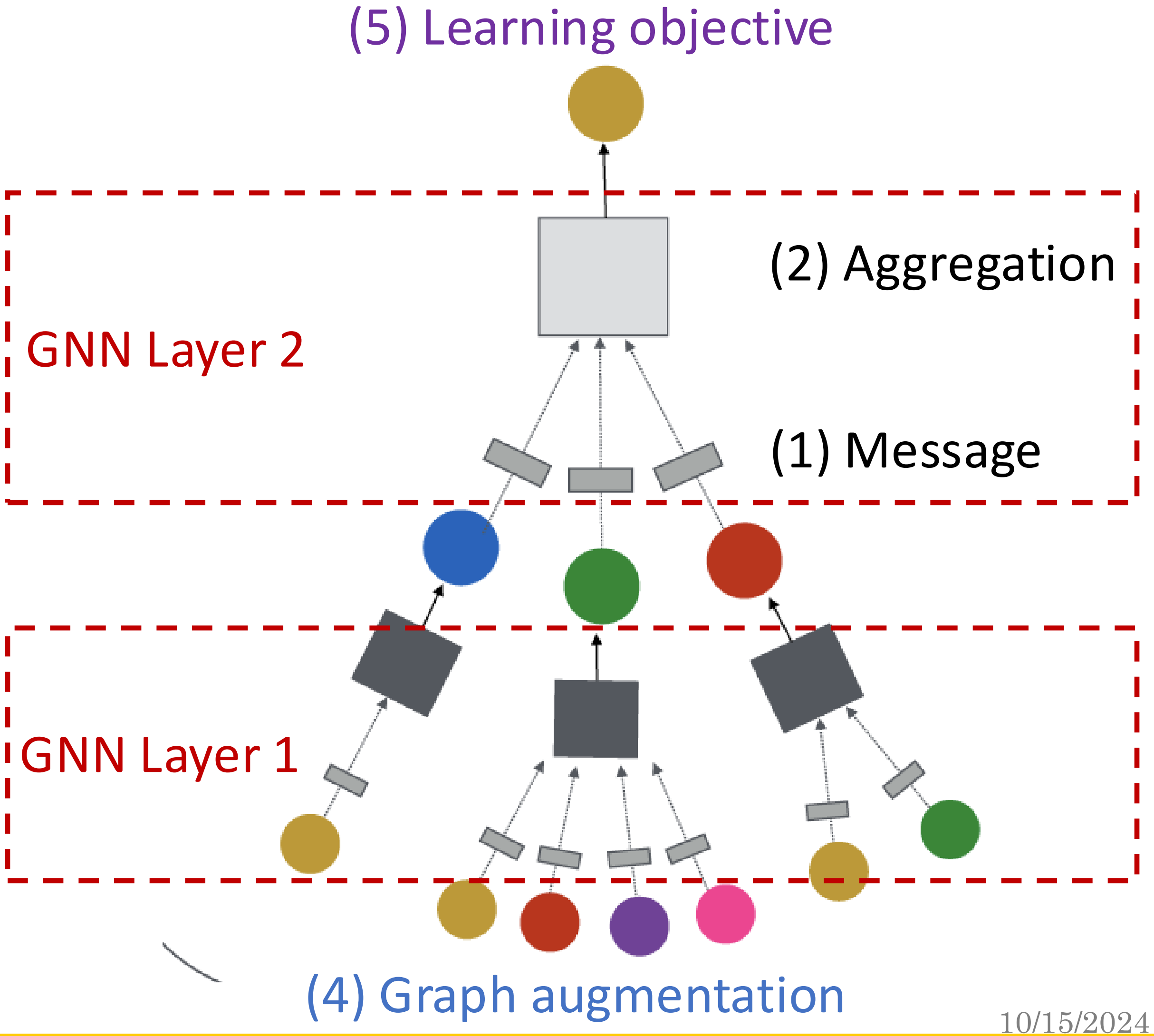
Instructor: Thanh H. Nguyen

Many slides are adapted from https://web.stanford.edu/class/cs224w/

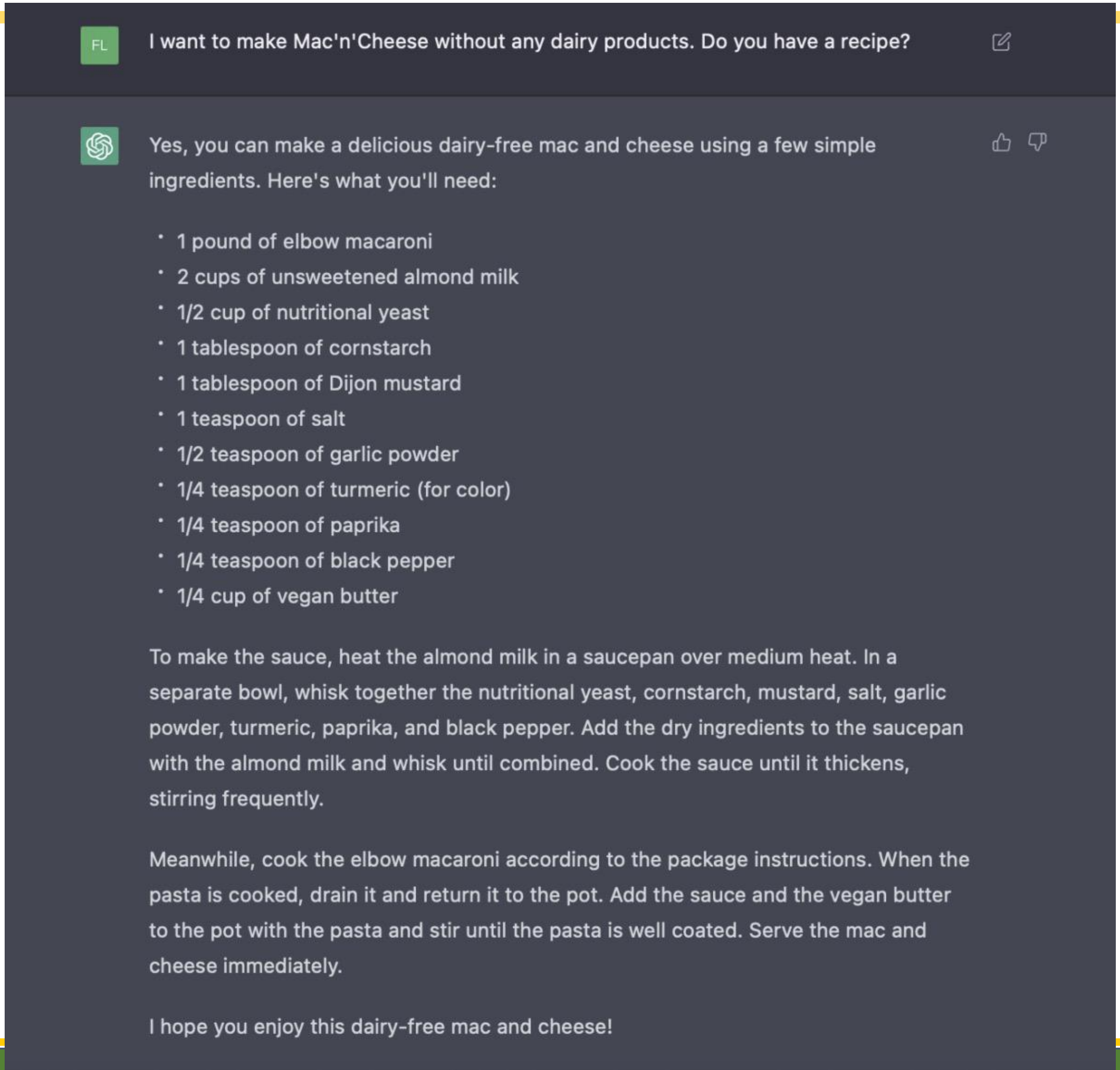
# Recap: A General GNN Framework



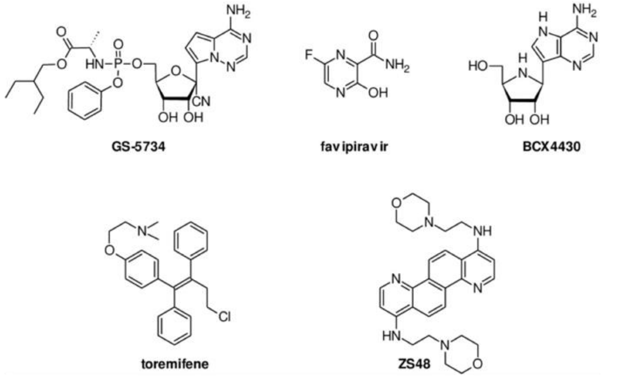

(3) Layer connectivity



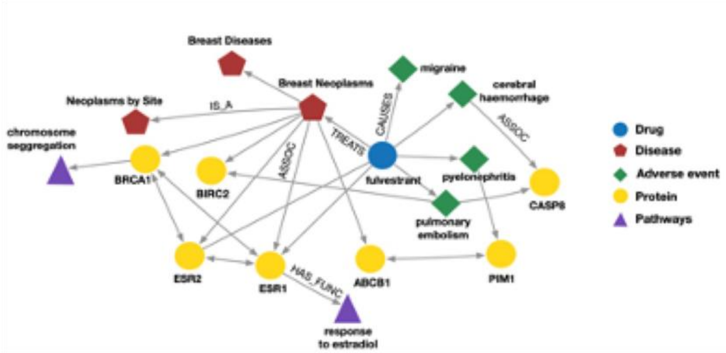
# Transformers:
# The Backbone of Generative AI



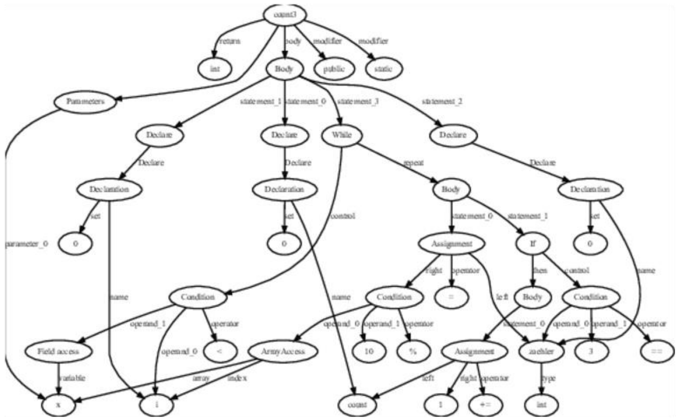
# Goal: Bring Power of Transformer to Graphs




Small molecules




Knowledge graphs



Code graphs

Image credit: ResearchGate

Communication networks

Image credit: Lumen Learning

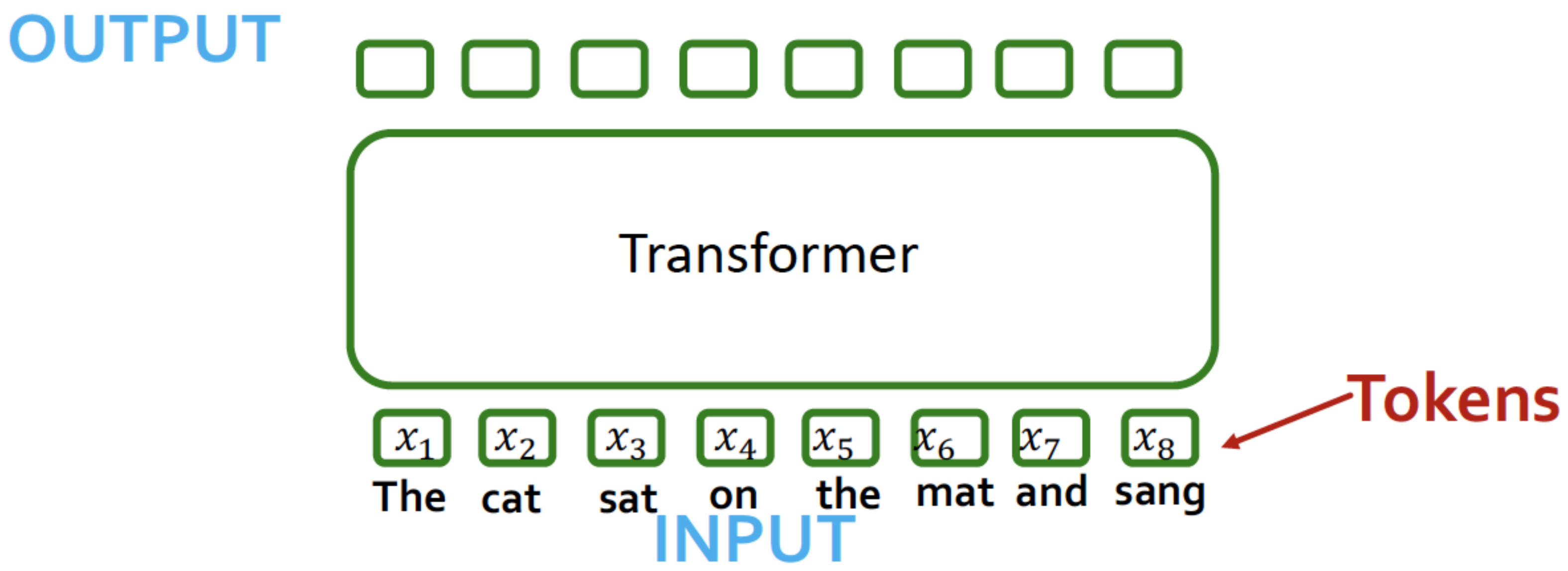
x

# Outline

- Part 1:
  - Introducing Transformers
  - Relation to message passing GNNs
- Part 2:
  - A new design landscape for graph Transformers
- Part 3:
  - Sign invariant Laplacian positional encodings for graph Transformers

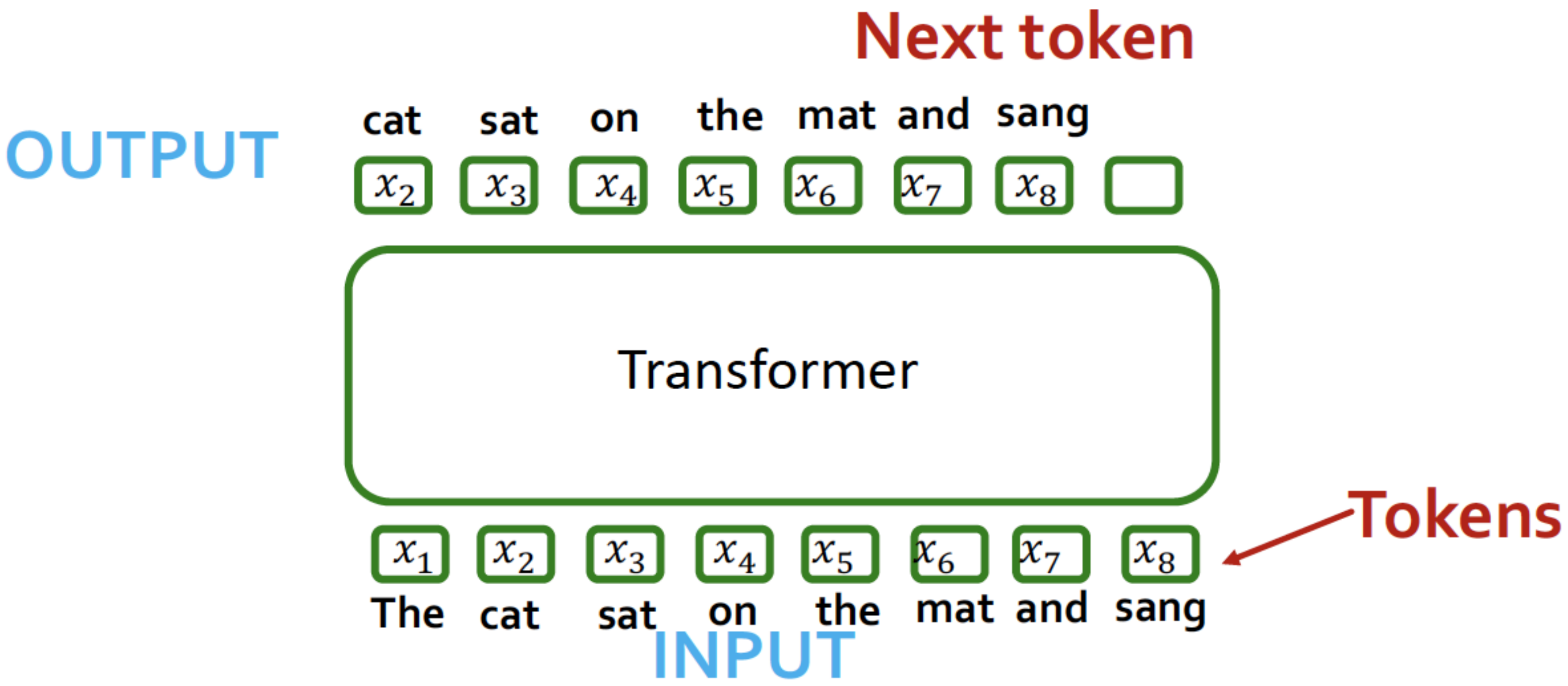
# Transformer Ingest Tokens

- Transformers map 1D sequences of vectors to 1D sequences of vectors



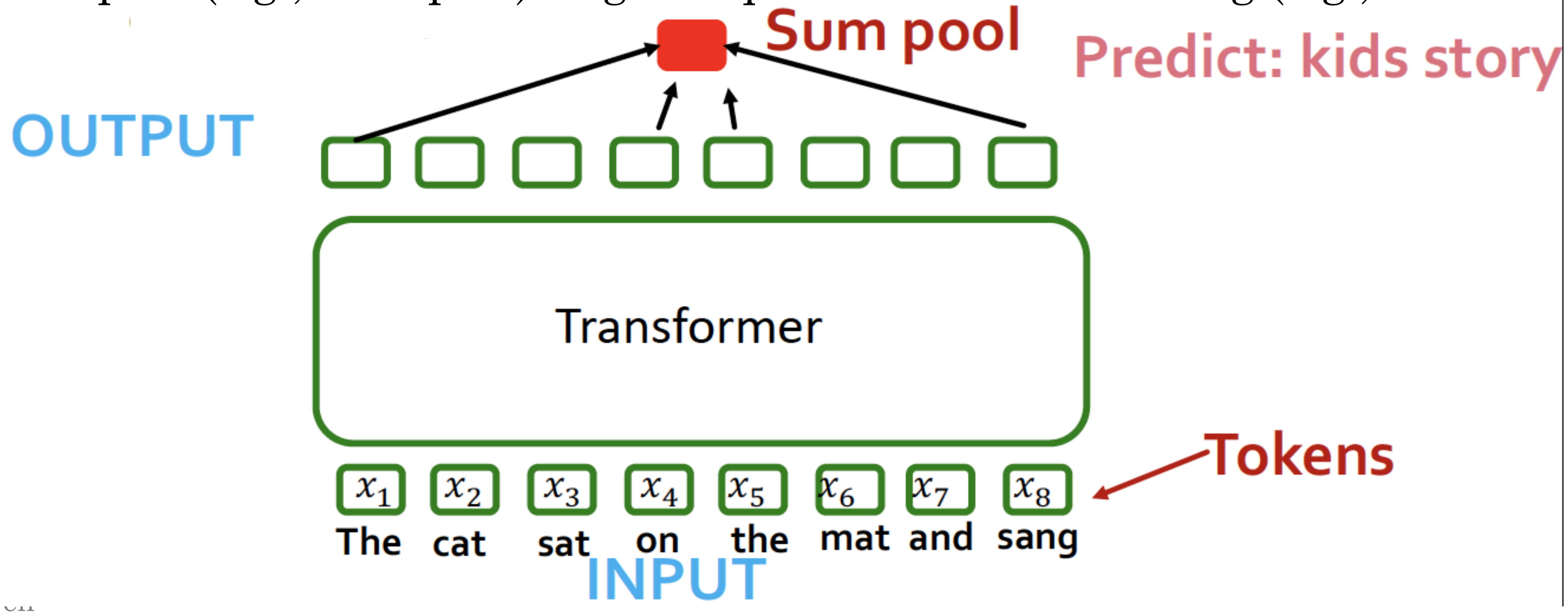
# Transformer Ingest Tokens

- Transformers map 1D sequences of vectors to 1D sequences of vectors
  - Tokens describe a ”piece” of data – e.g., a word
- What output sequence?
  - Option 1: next token => GPT



# Transformer Ingest Tokens

- Transformers map 1D sequences of vectors to 1D sequences of vectors
  - Tokens describe a ”piece” of data – e.g., a word
- What output sequence?
  - Option 1: next token => GPT
  - Option 2: pool (e.g., sum-pool) to get sequence level-embedding (e.g., for classification task)

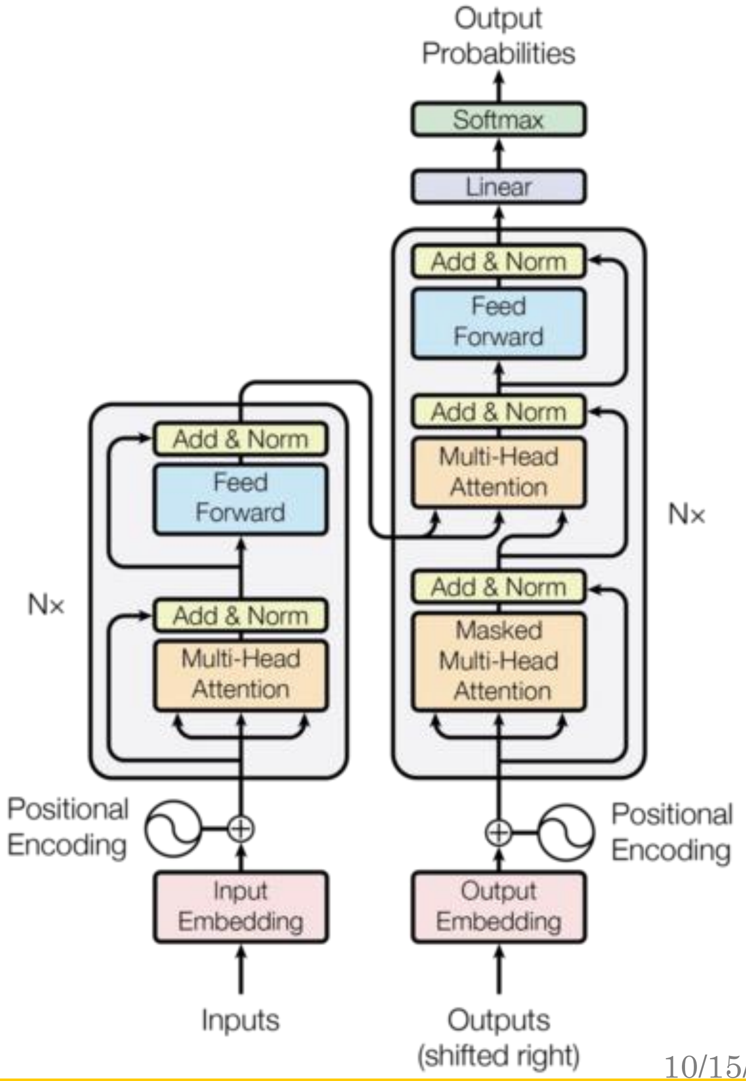


# Transformer Blueprint

- How are tokens processed?

- Lots of components
  - Normalization
  - Feed forward networks
  - Positional encoding
  - Multi-head self-attention

- What does self-attention block do?



# Self-Attention

- Intuition
  - As the model processes each word, self attention allows it to look at other positions in the input sequence for clues that can help lead to a better encoding for this word.

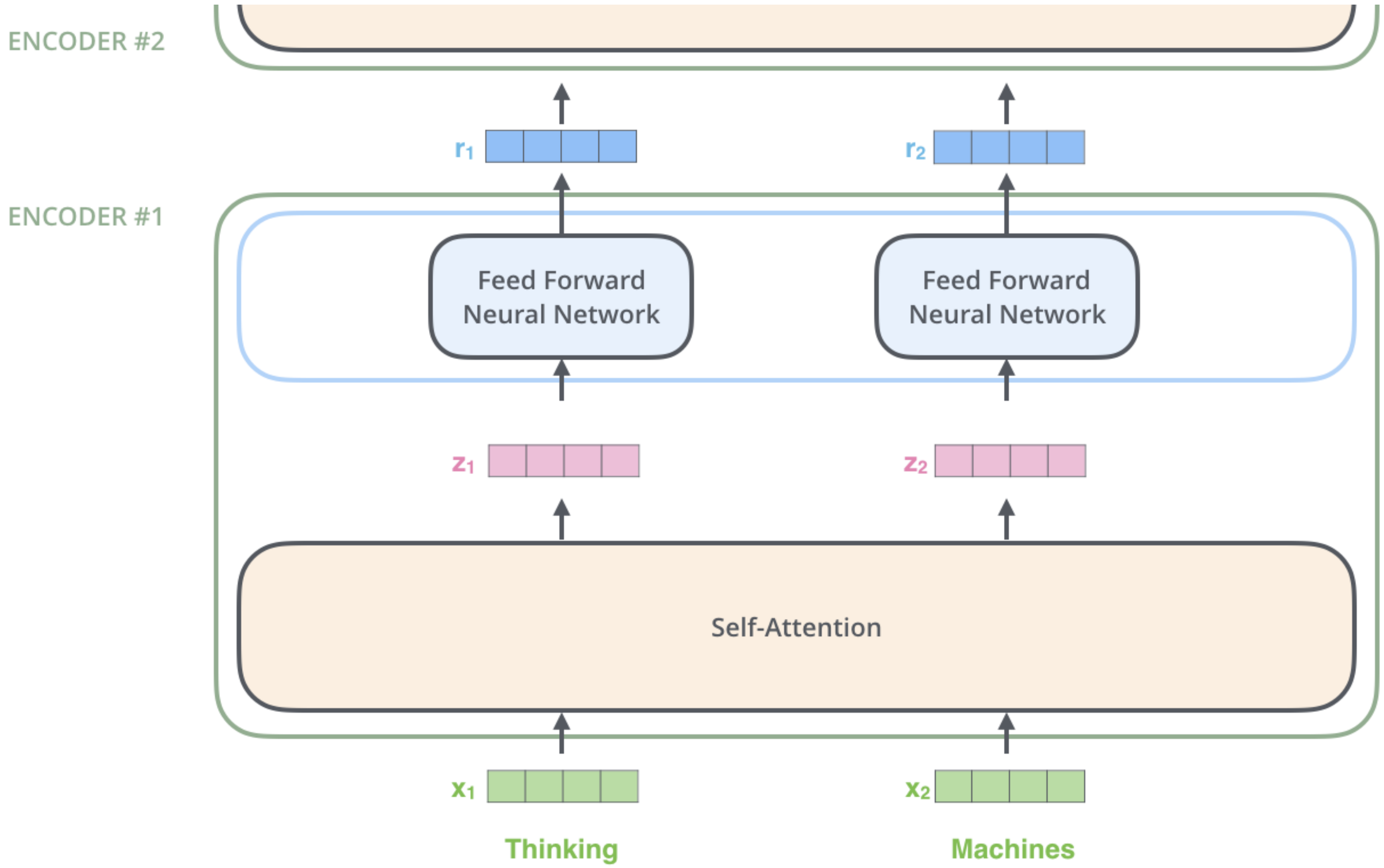
- Example:

The animal didn't cross the street because it was too tired.

  - When a model processes the word "it", self-attention allows it to associate "it" with "animal".

# Self-Attention

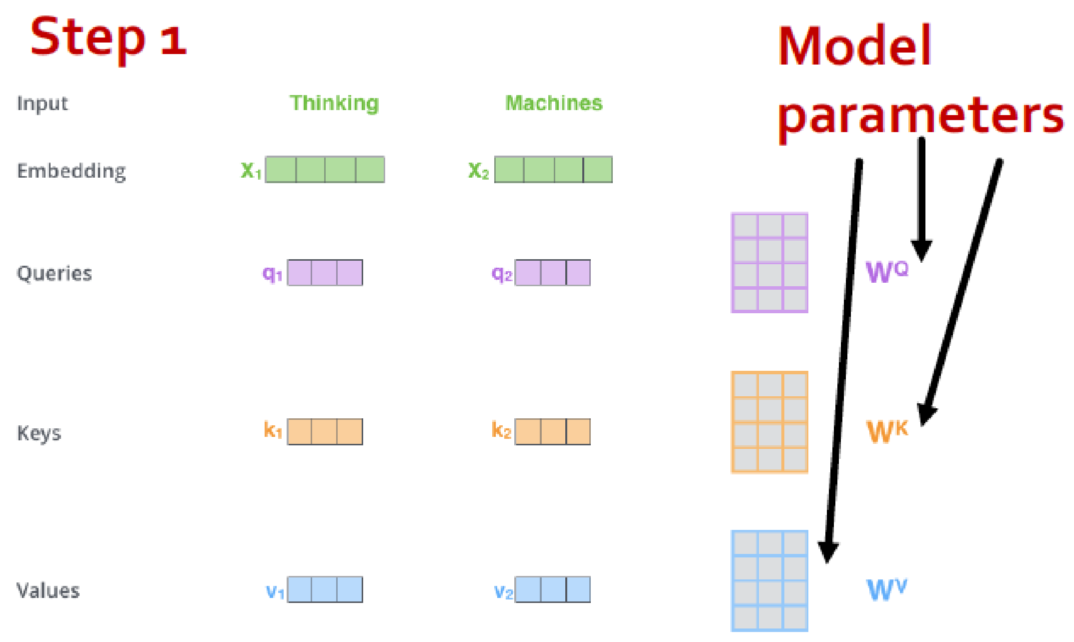
- Before "multi-head" self-attention, what is "single head" self-attention?




See: Illustrated Transformer tutorial, https://jalammar.github.io/illustrated-transformer/

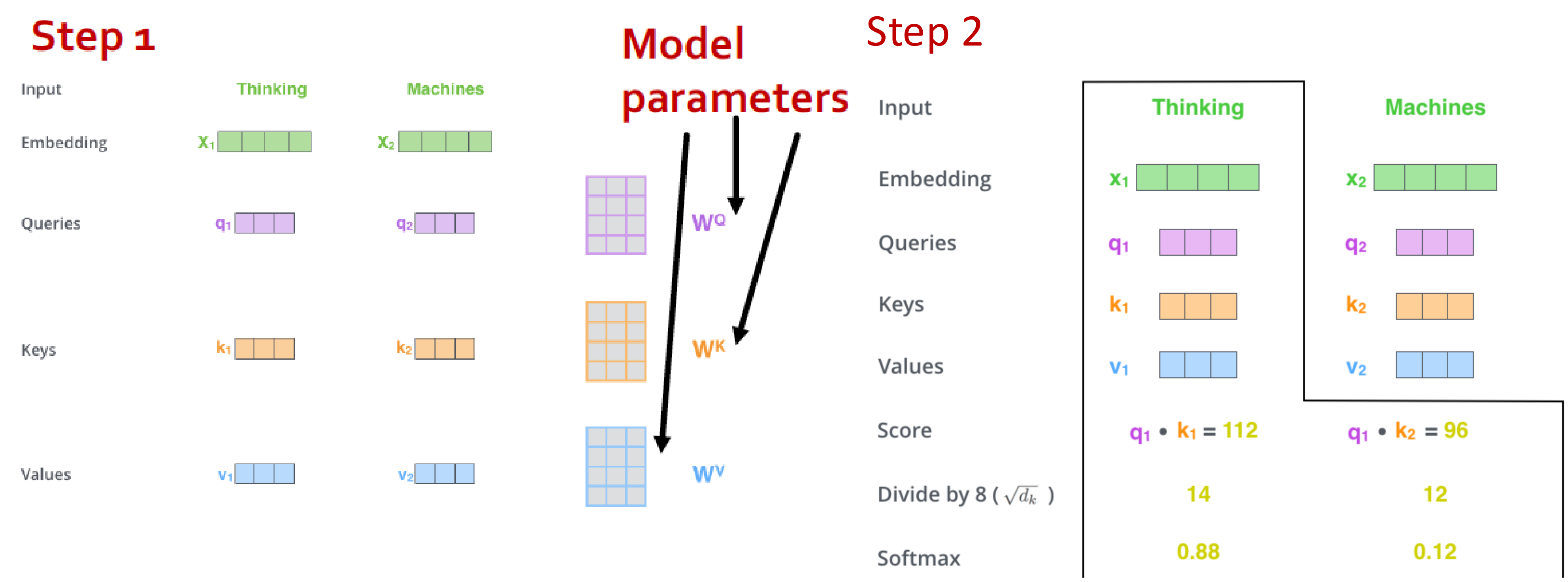
# Self-Attention

- Step 1: compute "key, value, query" for each input



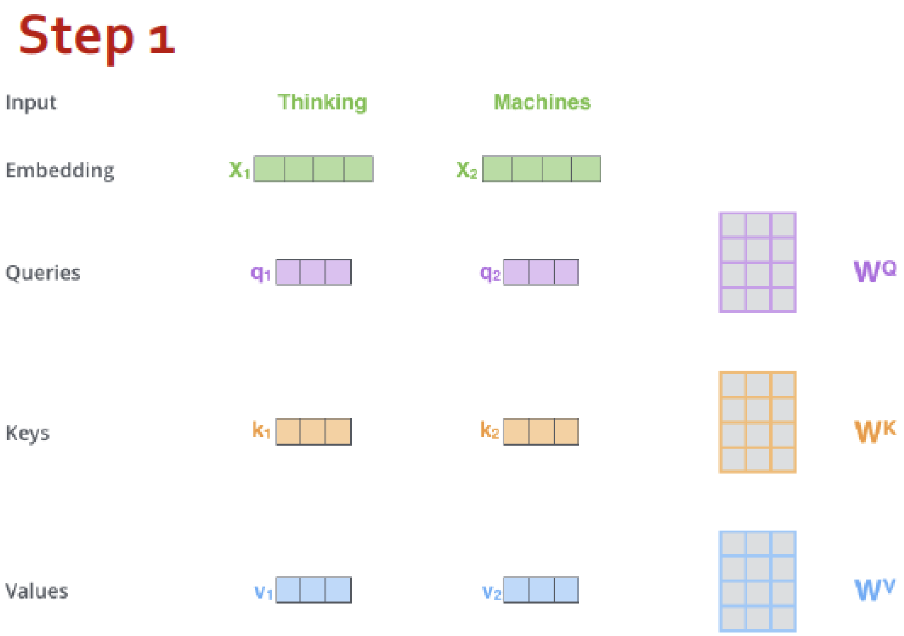
# Self-Attention

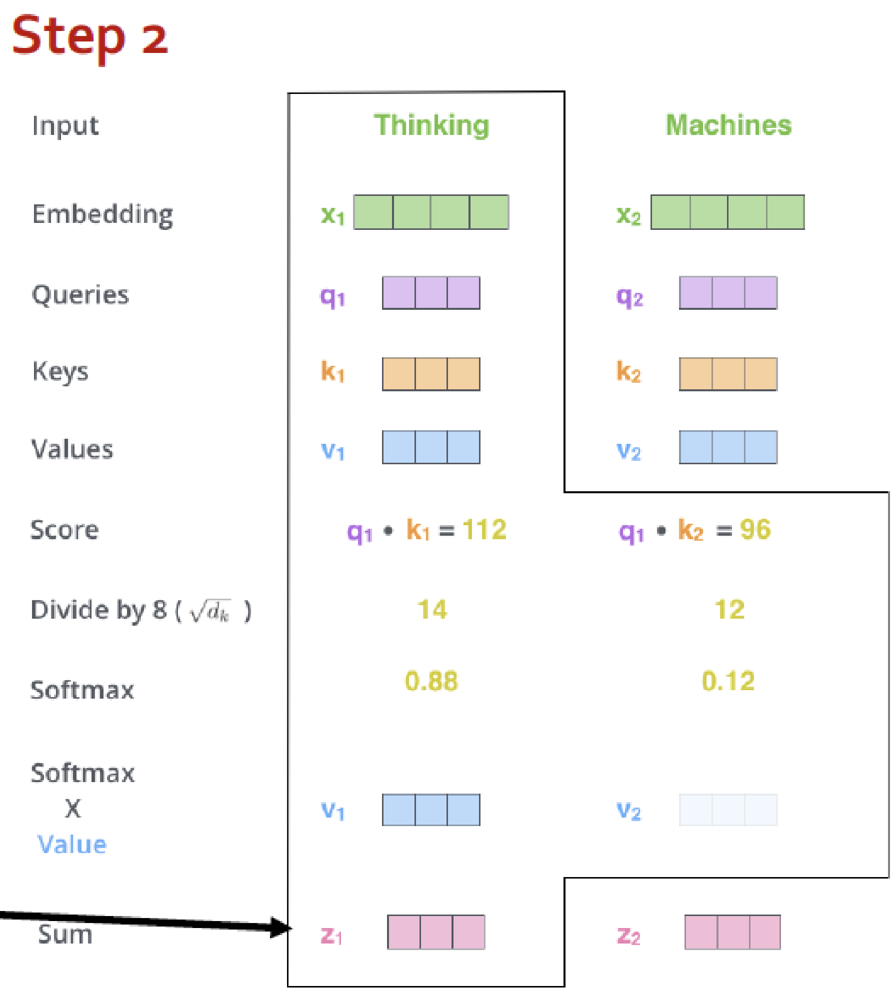
- Step 1: compute "key, value, query" for each input
- Step 2 (just for $x_1$): compute scores between pairs, turn into probabilities (same for $x_2$)



# Self-Attention

- Step 1: compute "key, value, query" for each input
- Step 2 (just for $x_1$): compute scores between pairs, turn into probabilities (same for $x_2$)
- Step 3: get new embedding $z_1$ by weighted sum of $v_1, v_2$




Step 3

$$z_1 = 0.88v_1 + 0.12v_2$$

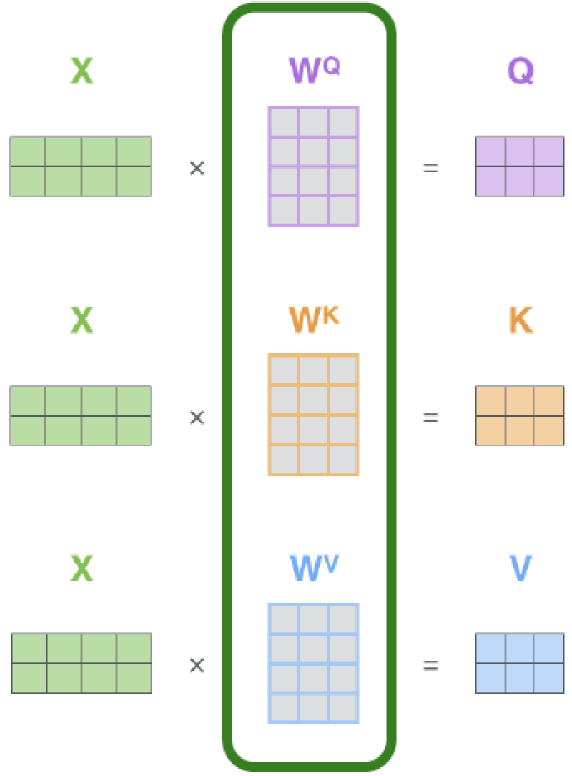
# Self-Attention

Large similarities will cause softmax to saturate and give vanishing gradients.
Recall a . b = |a||b| cos(angle)
Suppose that a and b are constant vectors of dimension D.

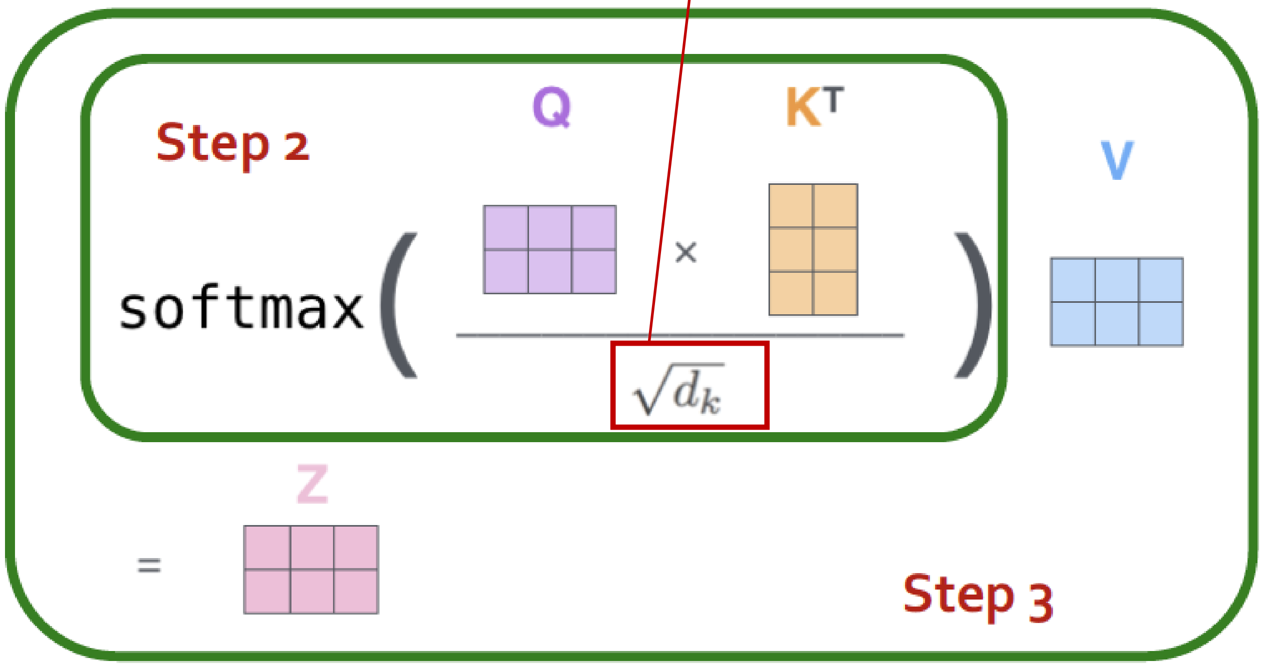
Then |a| $=(\sum_i a^2)^{\frac{1}{2}} = a\sqrt{D}$

- Same calculation in matrix form

Step 1

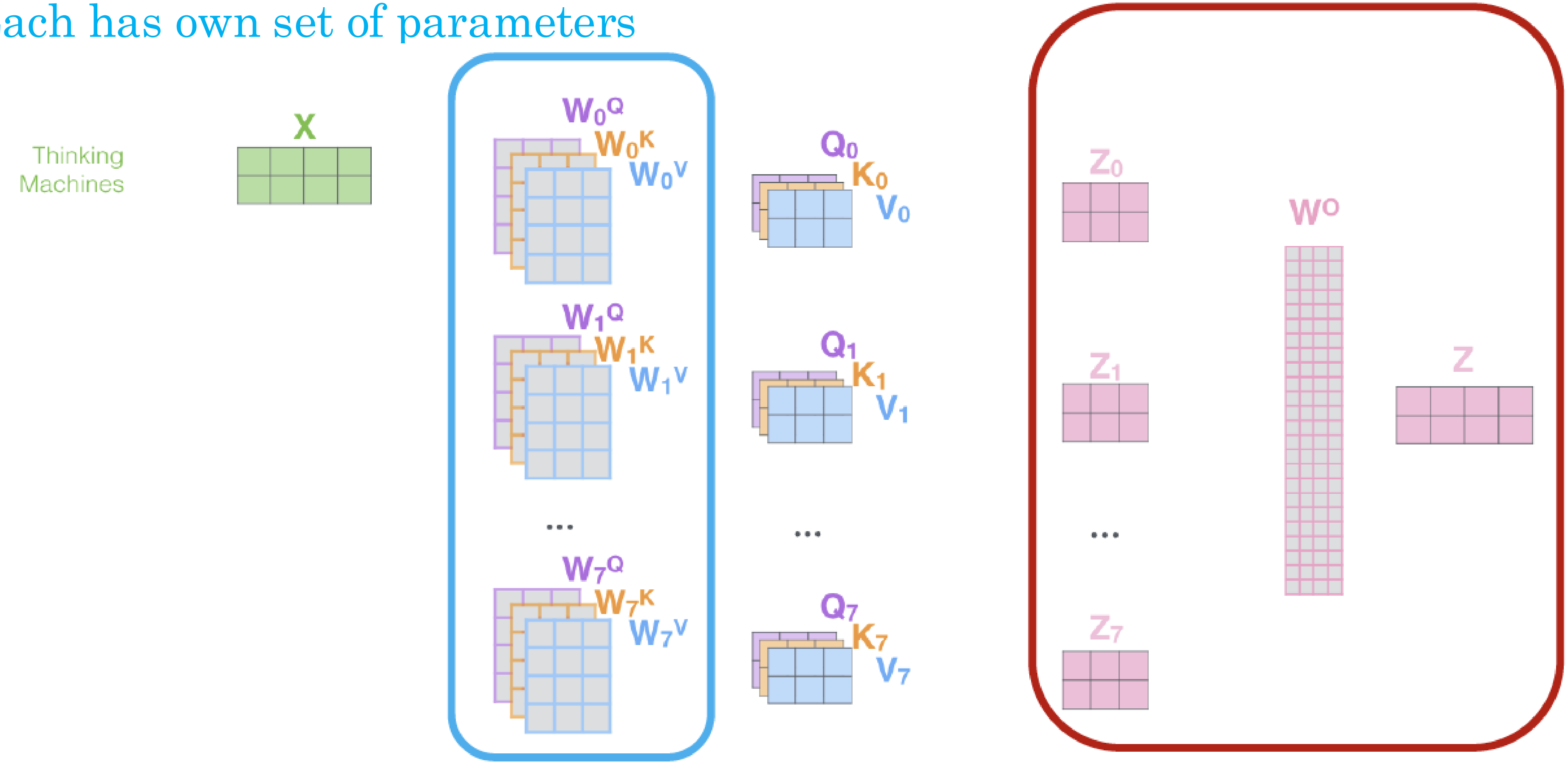



Model parameters



# Multi-Head Self-Attention

- Do many self-attentions in parallel, and combine
- Different heads can learn different “similarities” between inputs
- Each has own set of parameters



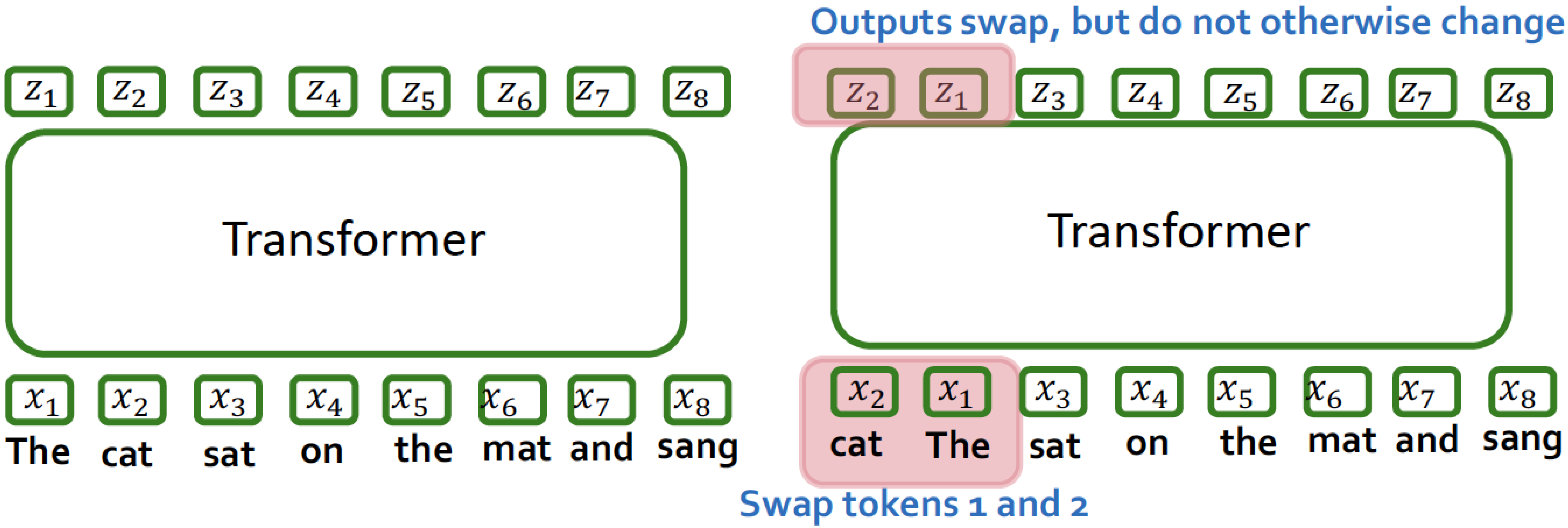
# Token Ordering

# Token Ordering

- First recall update formula: $$z_1 = \sum_{j=1}^{n} softmax(q_1^T k_j) v_j$$
- Key Observation: order of tokens does not matter!!!

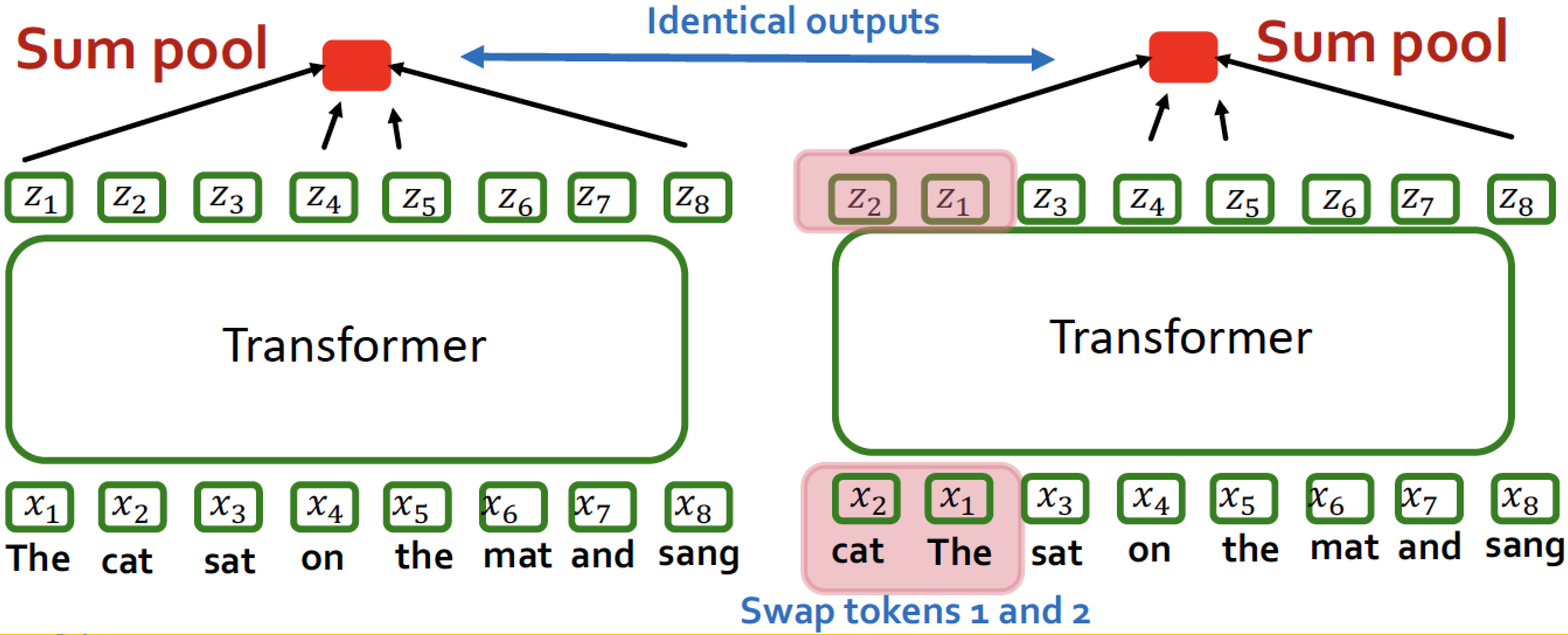
# Token Ordering

- First recall update formula: $z_1 = \sum_{j=1}^{n} softmax(q_1^T k_j) v_j$
- Key Observation: order of tokens does not matter!!!



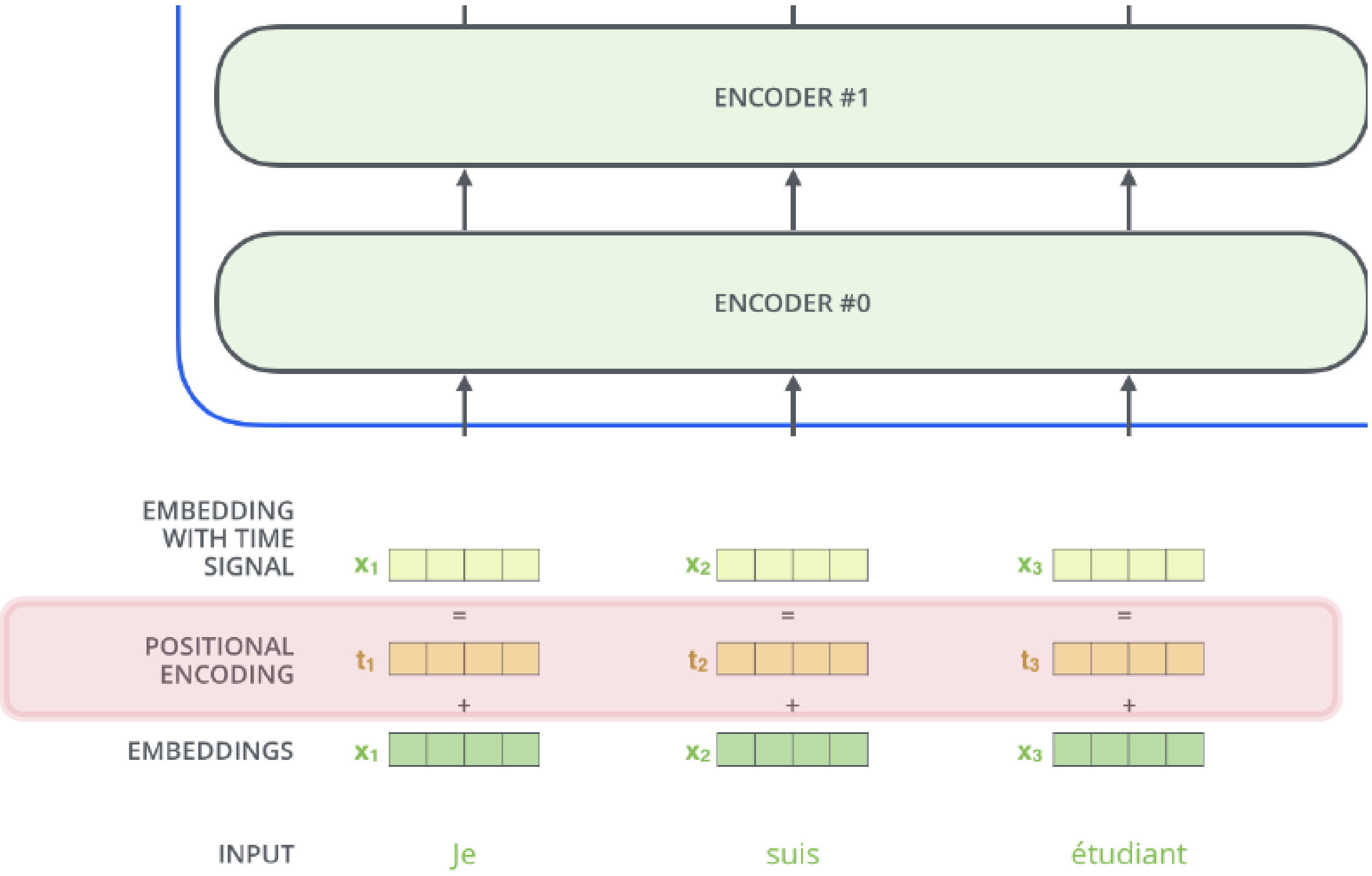
# Token Ordering

- This is a problem
- Same predictions no matter what order the words are in!
  - How to fix?

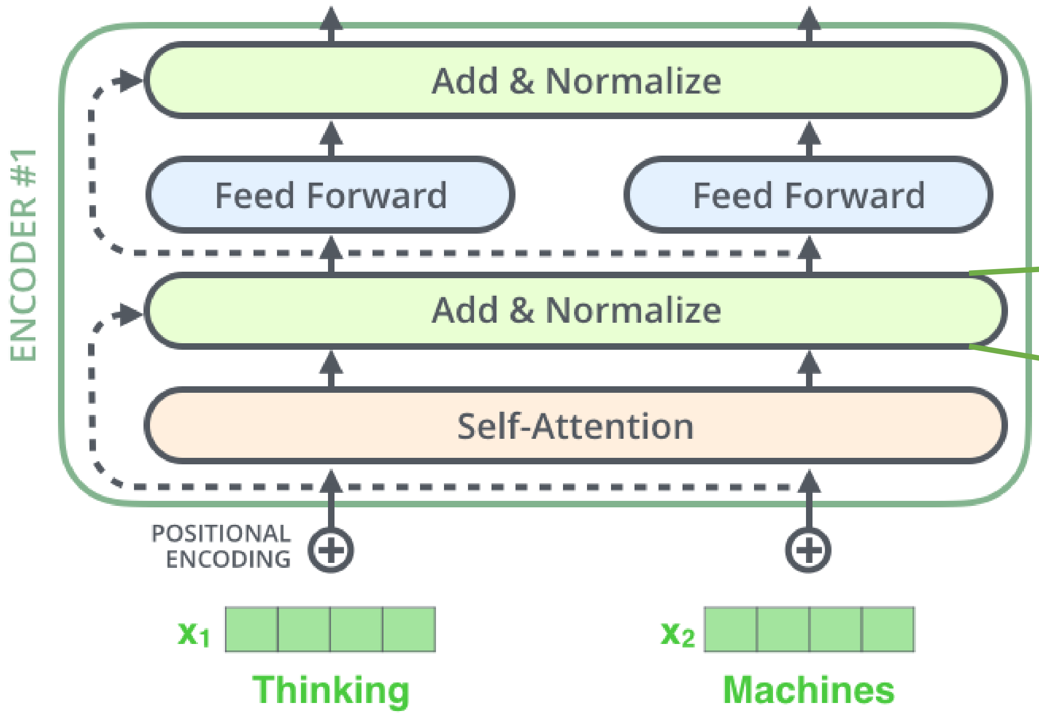


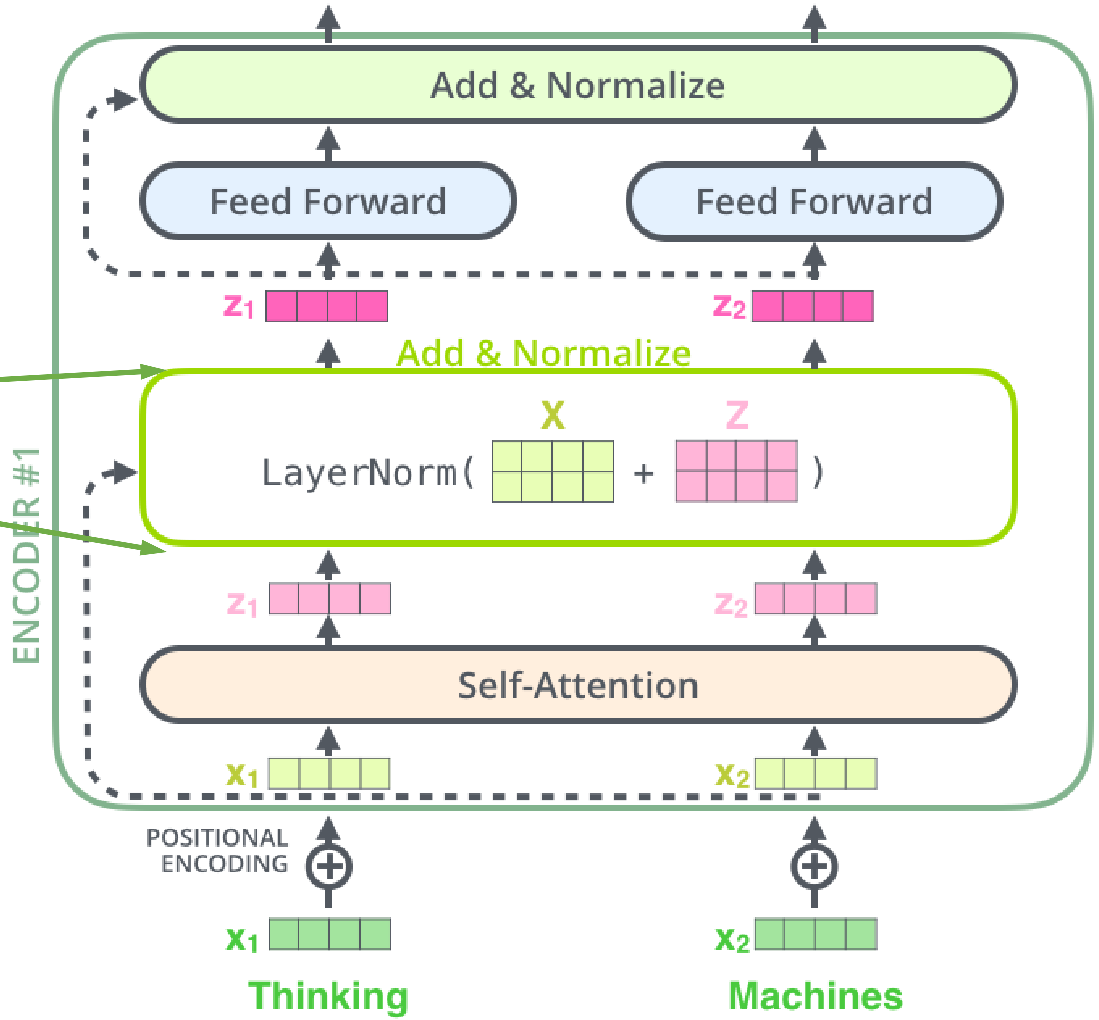
# Positional Encodings

- Transformer doesn't know order of inputs
- Extra positional features needed so it knows that
  - Je = word 1,
  - suis = word 2
  - etc.
- For NLP, positional encoding vectors are learnable parameters

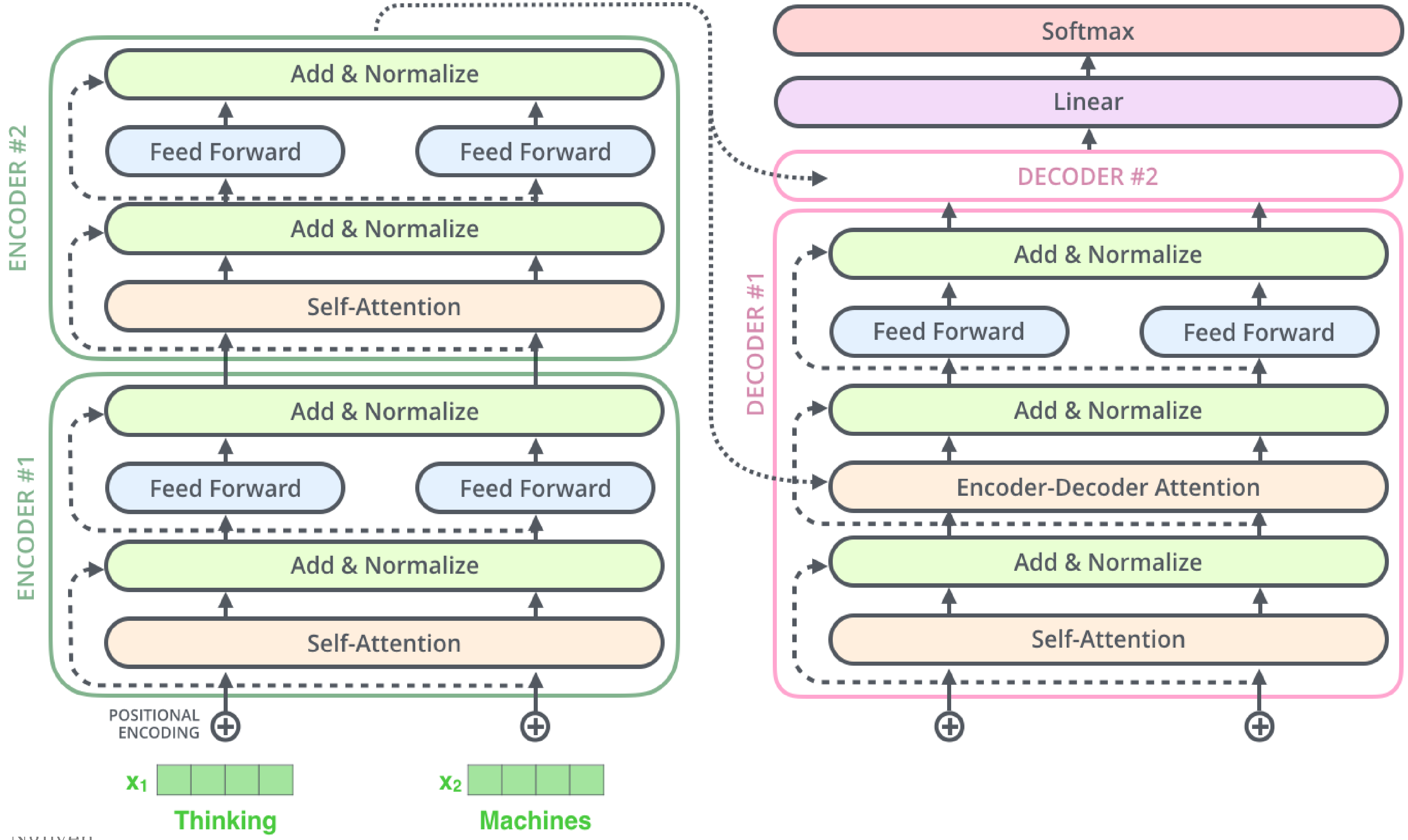


# Residual Connection & Normalization

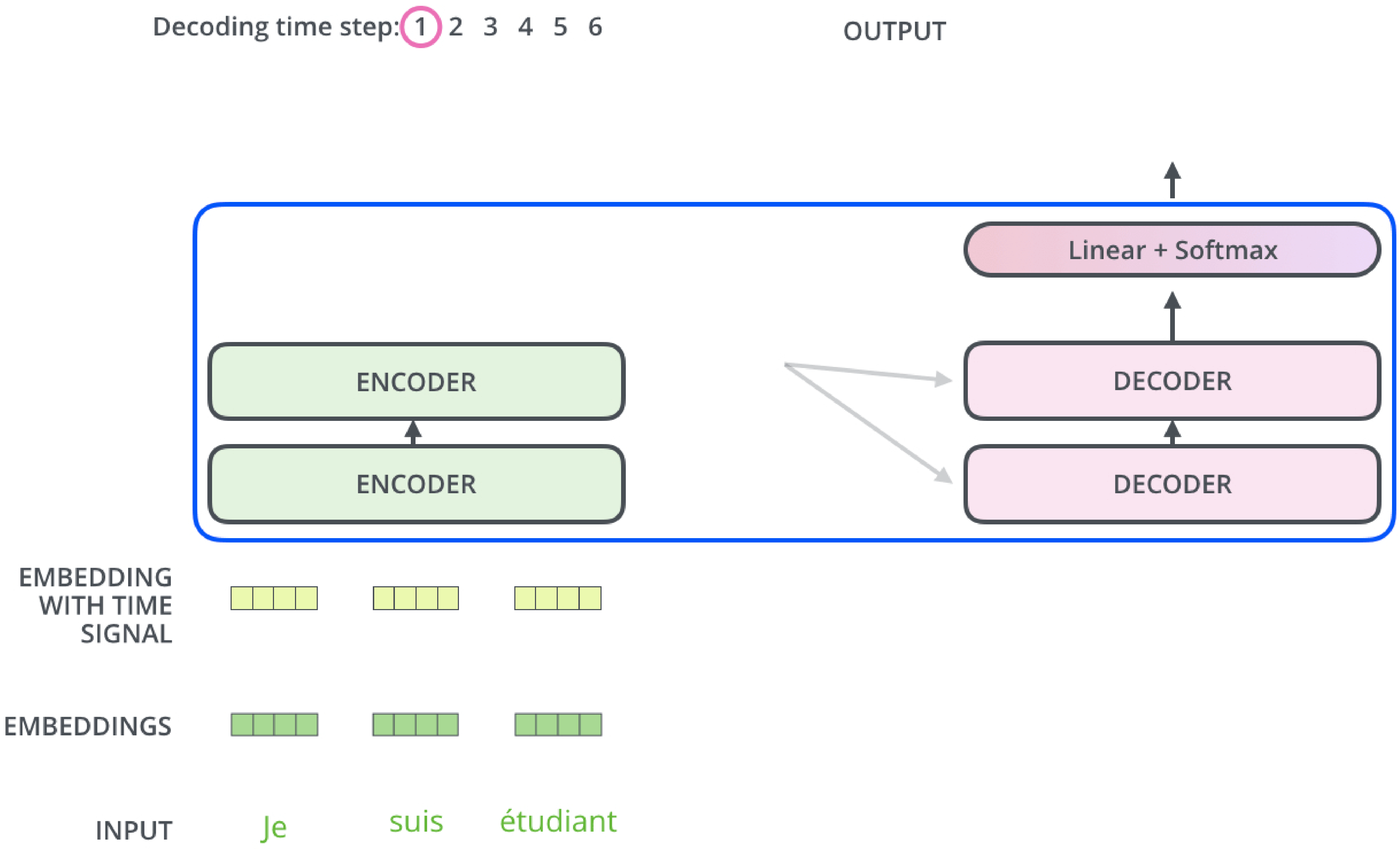


# Transformer: Encoder    Transformer: Decoder

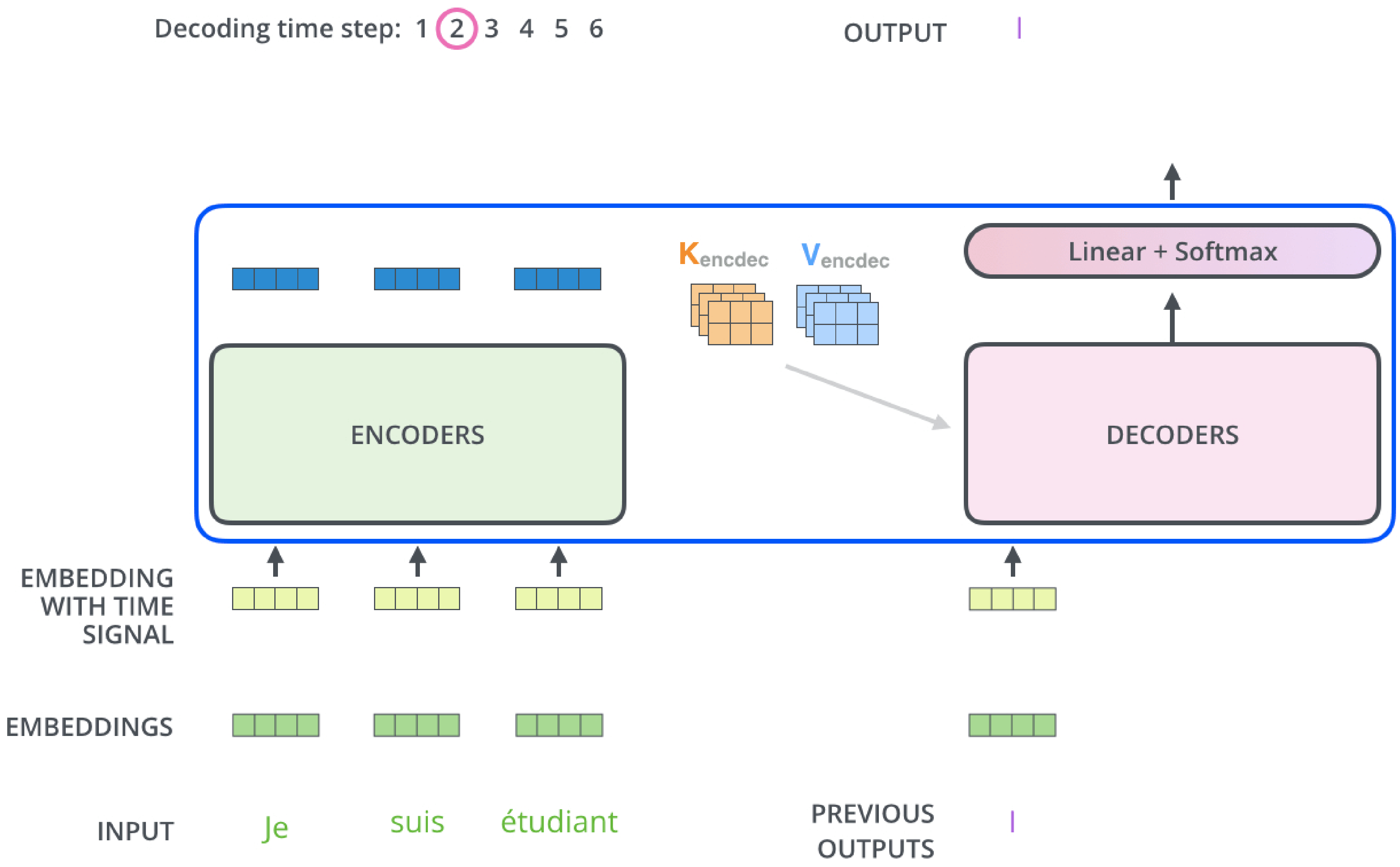


# Decoder



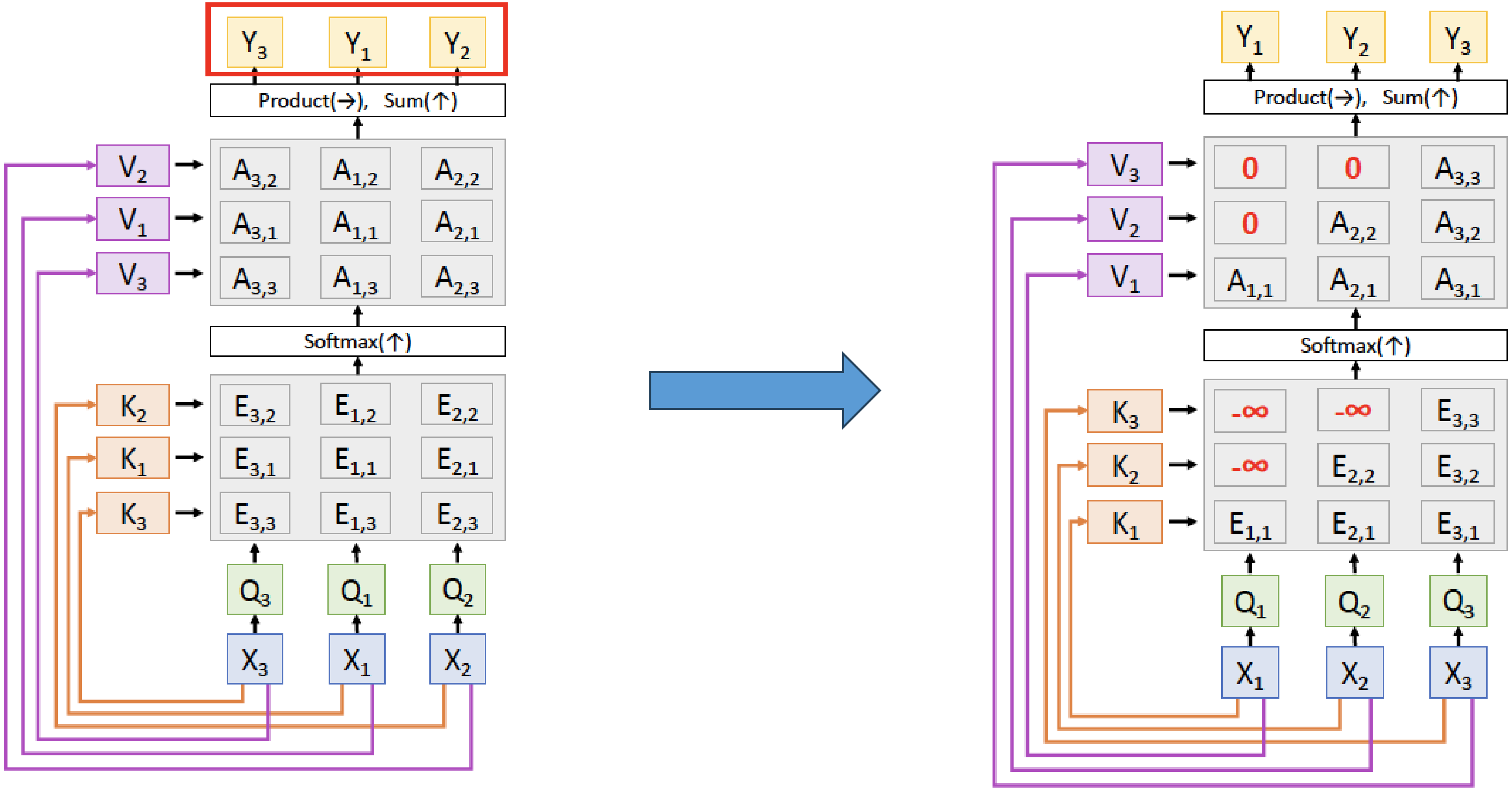
Decoding time step: 1 2 3 4 5 6
OUTPUT
Linear + Softmax
ENCODER
ENCODER
DECODER
DECODER
EMBEDDING WITH TIME SIGNAL
EMBEDDINGS
INPUT
Je
suis
étudiant

# Decoder



Decoding time step: 1 2 3 4 5 6
OUTPUT
I
K encdec
V encdec
Linear + Softmax
ENCODERS
DECODERS
EMBEDDING WITH TIME SIGNAL
EMBEDDINGS
INPUT
Je
suis
étudiant
PREVIOUS OUTPUTS
I

# Decoder: Masked Self-Attention Layer

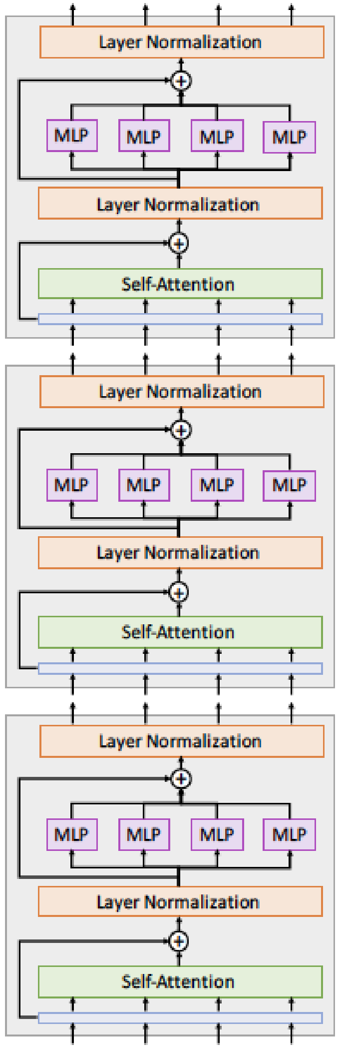


# The Transformer: Transfer Learning

- "ImageNet Moment for Natural Language Processing"

- Pretraining:
  - Download a lot of text from the internet
  - Train a giant Transformer model for language modeling

- Finetuning:
  - Fine-tune the Transformer on your own NLP task



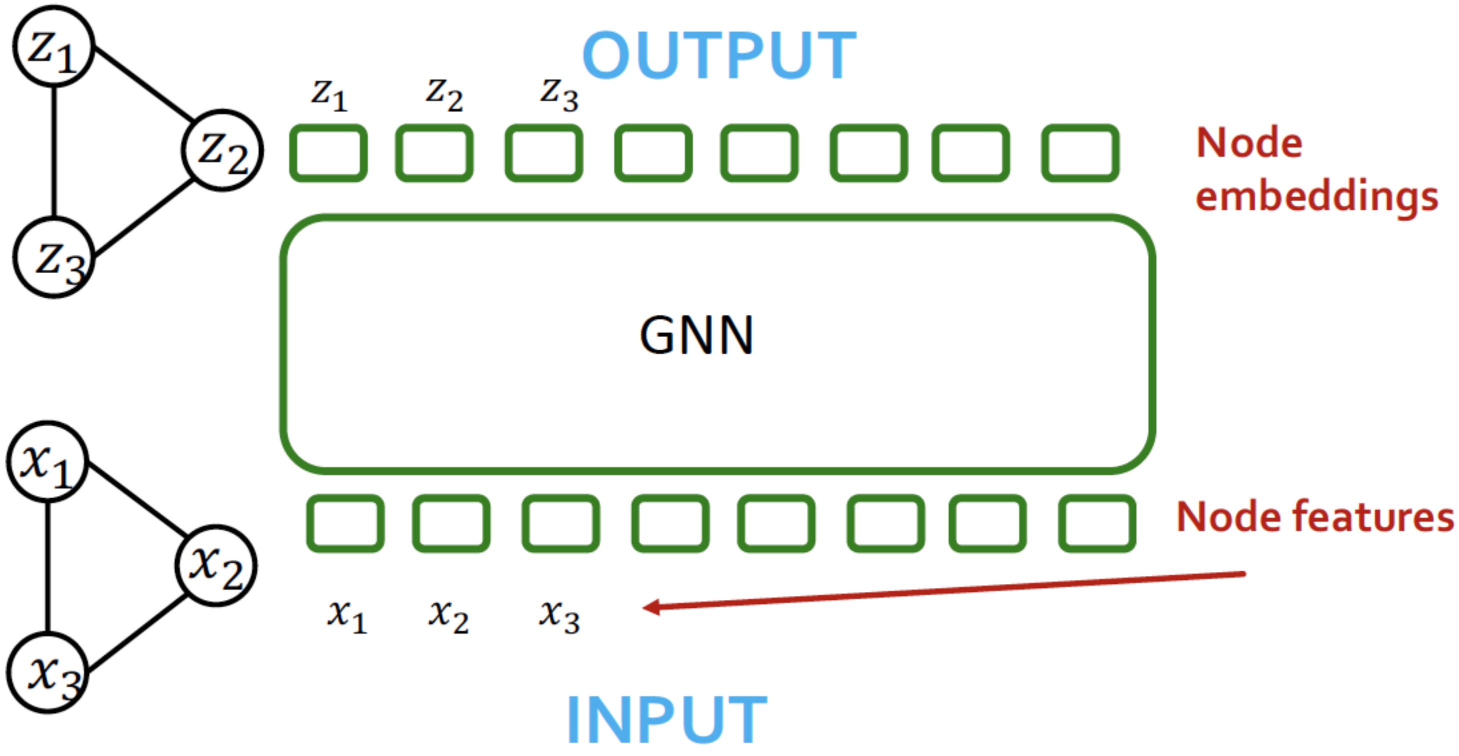
# Scaling Up Transformers

| Model | Layers | Width | Heads | Params | Data | Training |
|---|---|---|---|---|---|---|
| Transformer-Base | 12 | 512 | 8 | 65M | | 8x P100 (12 hours) |
| Transformer-Large | 12 | 1024 | 16 | 213M | | 8x P100 (3.5 days) |
| BERT-Base | 12 | 768 | 12 | 110M | 13 GB | |
| BERT-Large | 24 | 1024 | 16 | 340M | 13 GB | |
| XLNet-Large | 24 | 1024 | 16 | ~340M | 126 GB | 512x TPU-v3 (2.5 days) |
| RoBERTa | 24 | 1024 | 16 | 355M | 160 GB | 1024x V100 GPU (1 day) |
| GPT-2 | 48 | 1600 | ? | 1.5B | 40 GB | |
| Megatron-LM | 72 | 3072 | 32 | 8.3B | 174 GB | 512x V100 GPU (9 days) |
| Turing-NLG | 78 | 4256 | 28 | 17B | ? | 256x V100 GPU |
| GPT-3 | 96 | 12,288 | 96 | 175B | 694GB | ? |
| Gopher | 80 | 16,384 | 128 | 280B | 10.55 TB | 4096x TPUv3 (38 days) |

# Transformers for Graph Data

# Comparing Transformers with GNN

- Similarity: GNNs also take in a sequence of vectors (in no particular order) and output a sequence of embeddings
- Difference: GNNs use message passing, Transformer uses self-attention



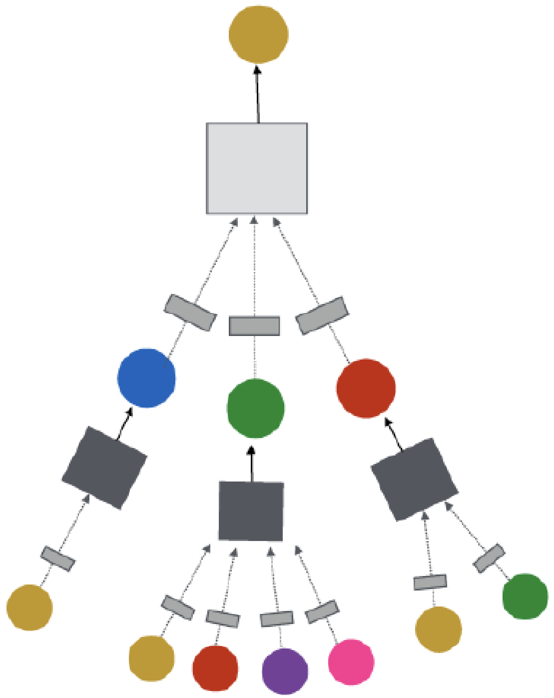
# Comparing Transformers with GNN

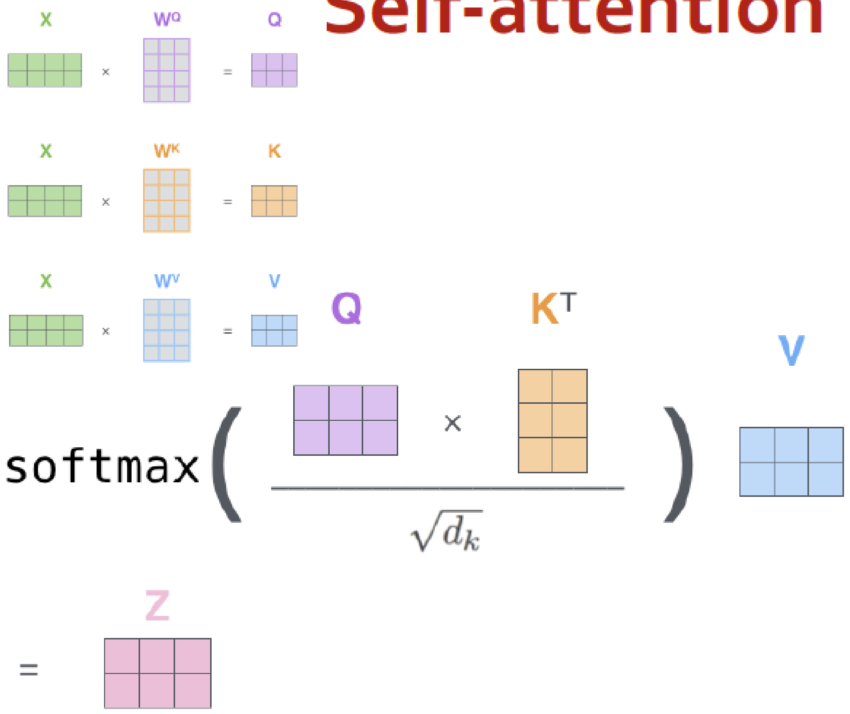
- Difference: GNNs use message passing, Transformer uses self-attention
- Are self-attention and message passing really different?

Message Passing

Vs.

Self-attention





# Self-Attention versus Message Passing

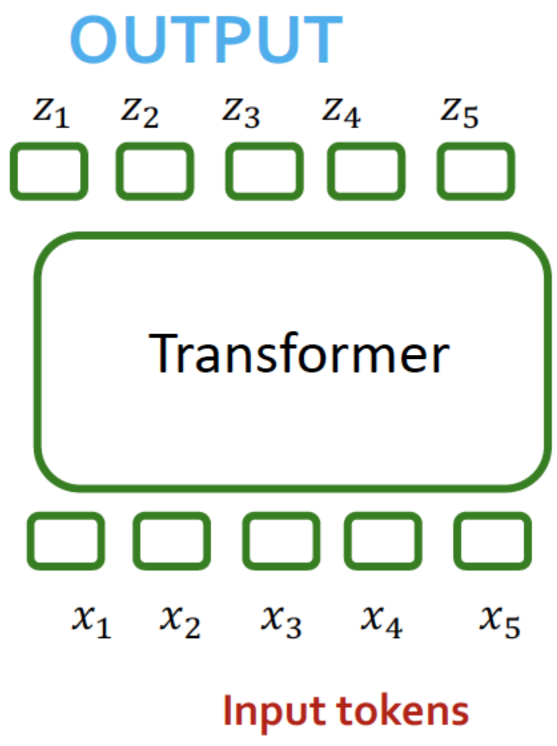
# Interpreting the Self-Attention Update

- Recall formula for attention update

$$
\begin{aligned}
Att(X) &= \mathrm{softmax}(K^T Q)V \\
&= \mathrm{softmax}\Big((XW^K)^T(XW^Q)\Big)(XW^V)
\end{aligned}
$$

Input stored row-wise

$X = [\cdots x_i \cdots]$



# Interpreting the Self-Attention Update

- Recall formula for attention update

$$
\begin{aligned}
Att(X) &= \text{softmax}(K^T Q)V \\
&= \text{softmax}\Big((XW^K)^T (XW^Q)\Big)(XW^V)
\end{aligned}
$$

Input stored row-wise

$X = [\cdots x_i \cdots]$

- This formula gives the embedding for all tokens simultaneously
- What if we simplify to just token $x_1$?

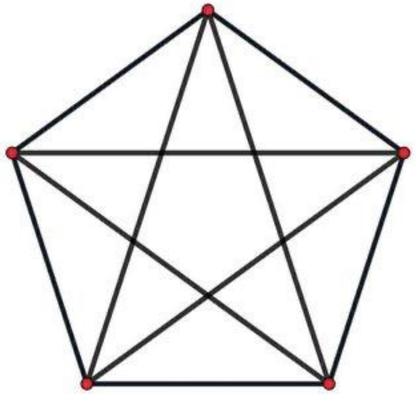
$$z_1 = \sum_{j=1}^{5} softmax_j\left(q_1^T k_j\right) v_j$$

How to interpret this?

- Steps for computing new embedding for token 1:
  1. Compute message from j: $MSG(x_j) = (v_j, k_j) = (W^V x_j, W^K x_j)$
  2. Compute query for 1: $q_1 = W^Q x_1$
  3. Aggregate all messages: $Agg\left(\{MSG(x_j) : j\}, q_1\right) = \sum_{j=1}^{n} softmax\left(q_1^T k_j\right) v_j$

# Self-Attention as Message Passing

- Takeaway: Self-attention can be written as message + aggregation – i.e., it is a GNN!
- But so far there is no graph – just tokens.
  - So what graph is this a GNN on?
- Clearly tokens = nodes, but what are the edges?
- Key observation:
  - Token 1 depends on (receives "messages" from) all other tokens
  - ➢ The graph is fully connected!
- Alternatively: if you only sum over $j \in N(i)$, you get ~GAT

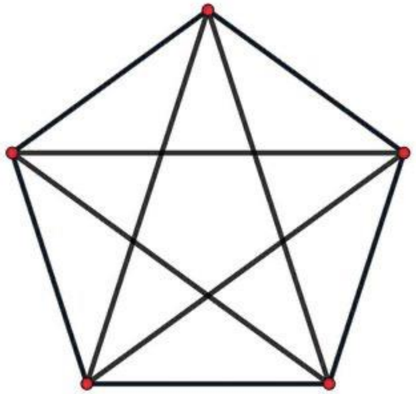


$$\boxed{z_1 = \sum_{j=1}^{n} softmax(q_1^T k_j) v_j}$$

- Steps for computing new embedding for token 1:
  1. Compute message from j: $MSG(x_j) = (v_j, k_j) = (W^V x_j, W^K x_j)$
  2. Compute query for 1: $q_1 = W^Q x_1$
  3. Aggregate all messages: $Agg(\{MSG(x_j) : j\}, q_1) = \sum_{j=1}^{n} softmax(q_1^T k_j) v_j$

# Self-Attention as Message Passing

- Takeaway 1: Self-attention is a special case of message passing
- Takeaway 2: It is message passing on the fully connected graph
- Takeaway 3: Given a graph $G$, if you constrain the self-attention softmax to only be over ***j*** adjacent to ***i*** nodes, you get ~GAT!

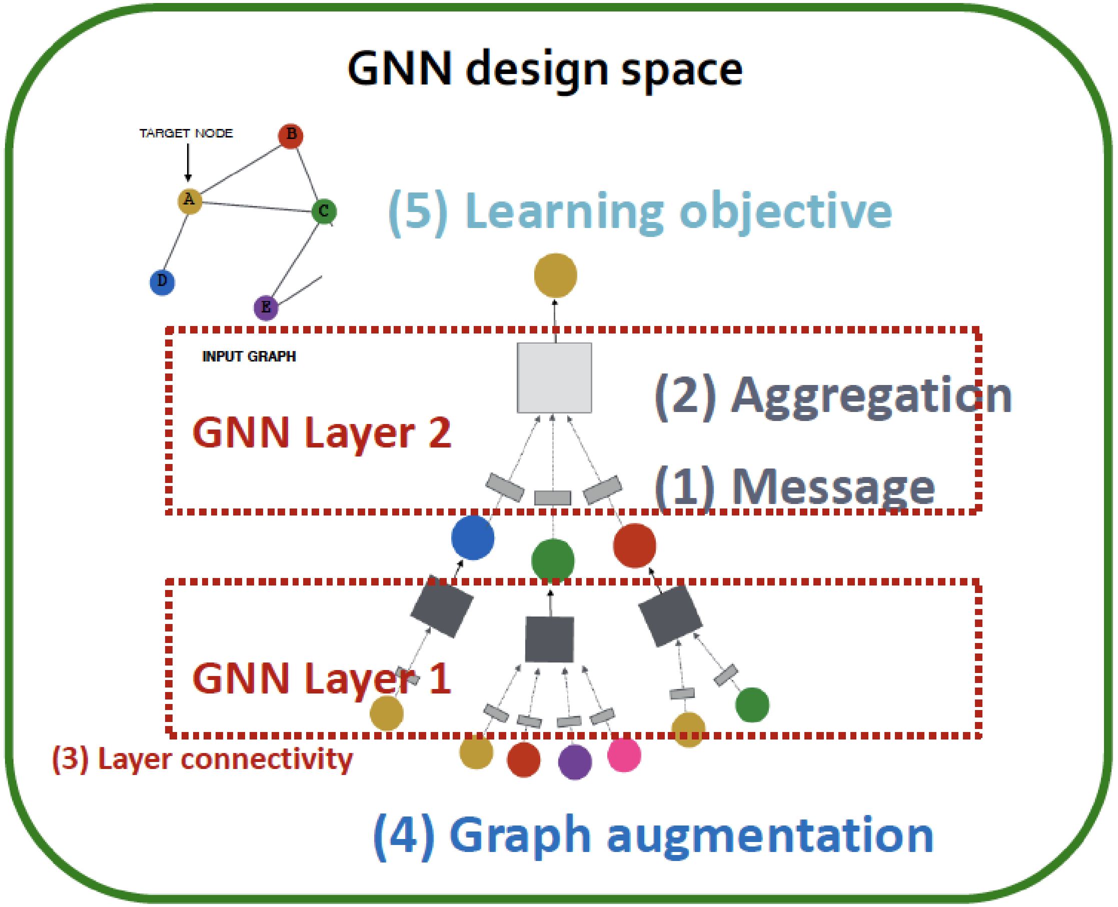


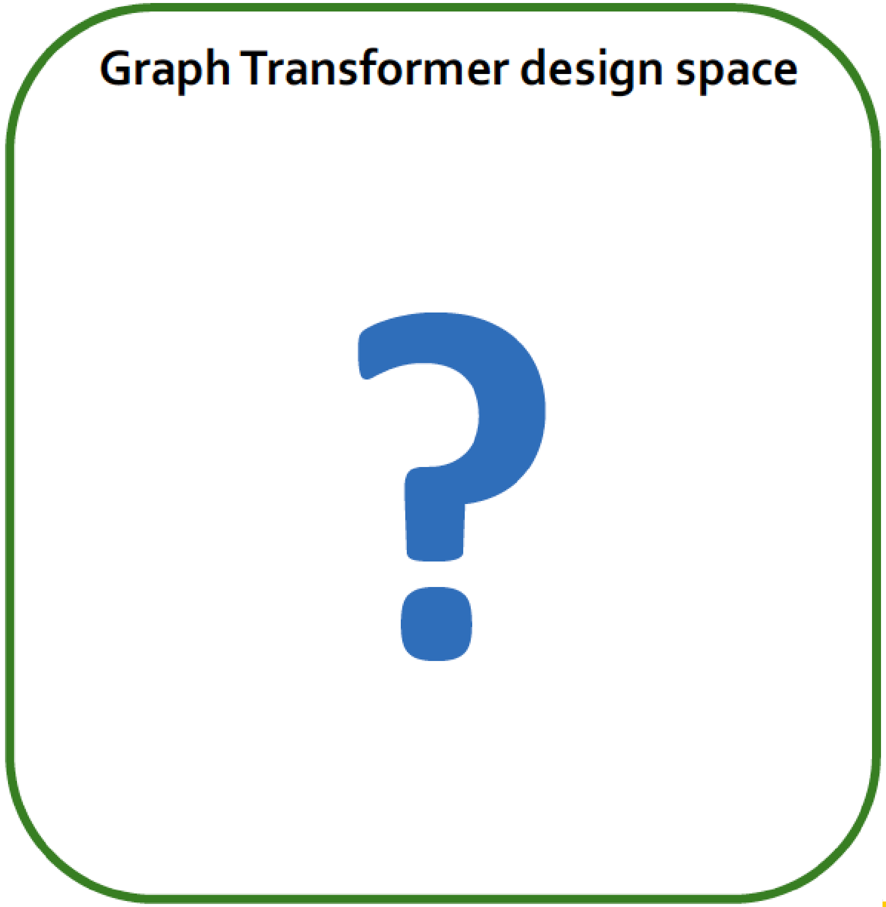
- Steps for computing new embedding for token 1:
  1. Compute message from j: $MSG(x_j) = (v_j, k_j) = (W^V x_j, W^K x_j)$
  2. Compute query for 1: $q_1 = W^Q x_1$
  3. Aggregate all messages: $Agg(\{MSG(x_j): j\}, q_1) = \sum_{j=1}^{n} softmax(q_1^T k_j) v_j$

# A New Design Landscape for Graph Transformer

# Recap: A General GNN Framework

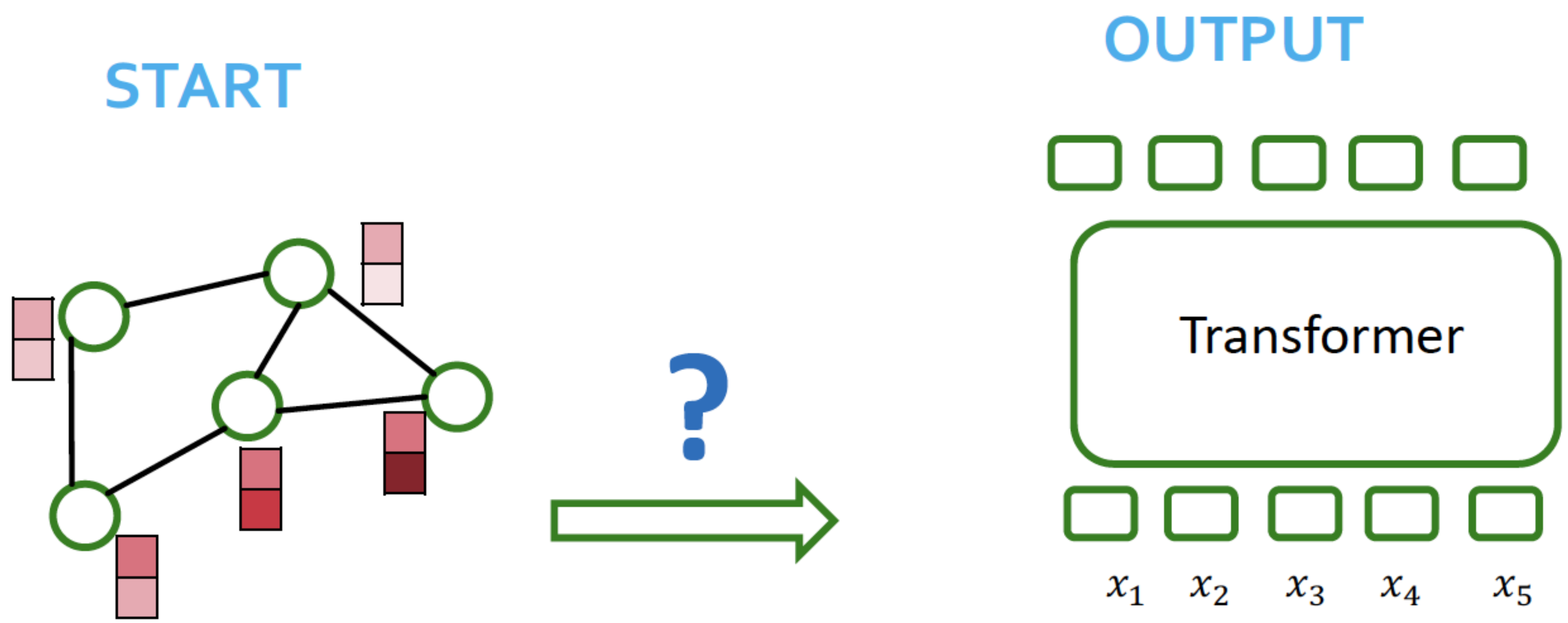
- We know a lot about the design space of GNNs
- What does the corresponding design space for Graph Transformers look like?



# Processing Graphs with Transformer

- We start with graph(s)
- How to input a graph into a Transformer?



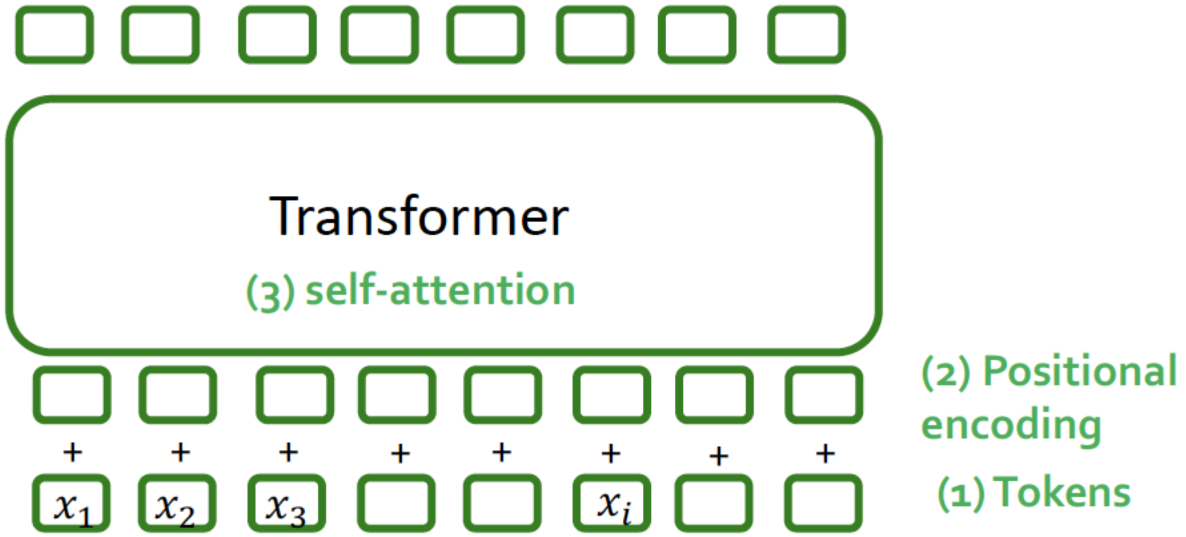

J. You, R. Ying, J. Leskovec. [Design Space of Graph Neural Networks](), NeurIPS 2020c

# Components of a Transformer

- Key components of Transformer
  1. tokenizing
  2. positional encoding
  3. self-attention

*How to chose these for graph data?*

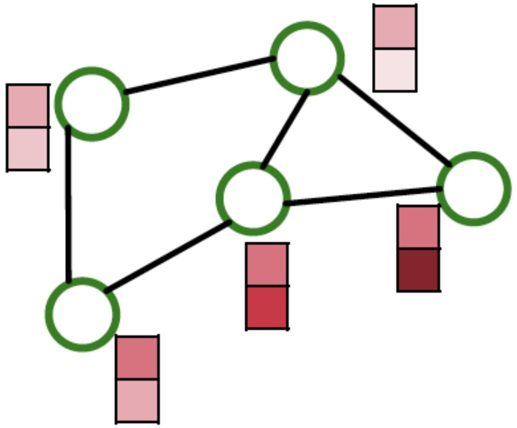
- Key question: What should these be for a graph input?



# Processing Graphs with Transformers

- A graph Transformer must take the following inputs:
  1. Node features?
  2. Adjacency information?
  3. Edge features?
- Key components of Transformer
  1. tokenizing
  2. positional encoding
  3. self-attention

- There are many ways to do this
- Different approaches correspond to different "matchings" between graph inputs (1), (2), (3) transformer components (1), (2), (3)





J. You, R. Ying, J. Leskovec. Design Space of Graph Neural Networks, NeurIPS 2020c

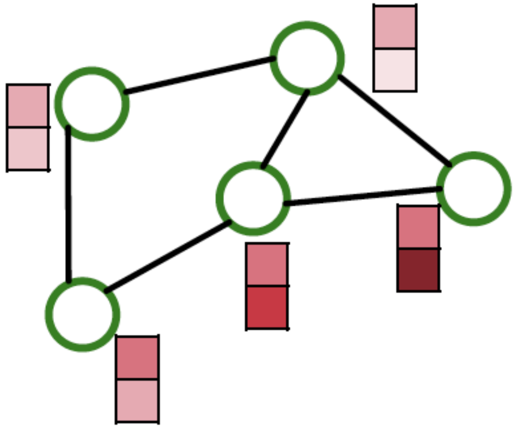
# Processing Graphs with Transformers

- A graph Transformer must take the following inputs:
  1. Node features?
  2. Adjacency information?
  3. Edge features?

- Key components of Transformer
  1. tokenizing
  2. positional encoding
  3. self-attention

Today

- There are many ways to do this
- Different approaches correspond to different "matchings" between graph inputs (1), (2), (3) transformer components (1), (2), (3)

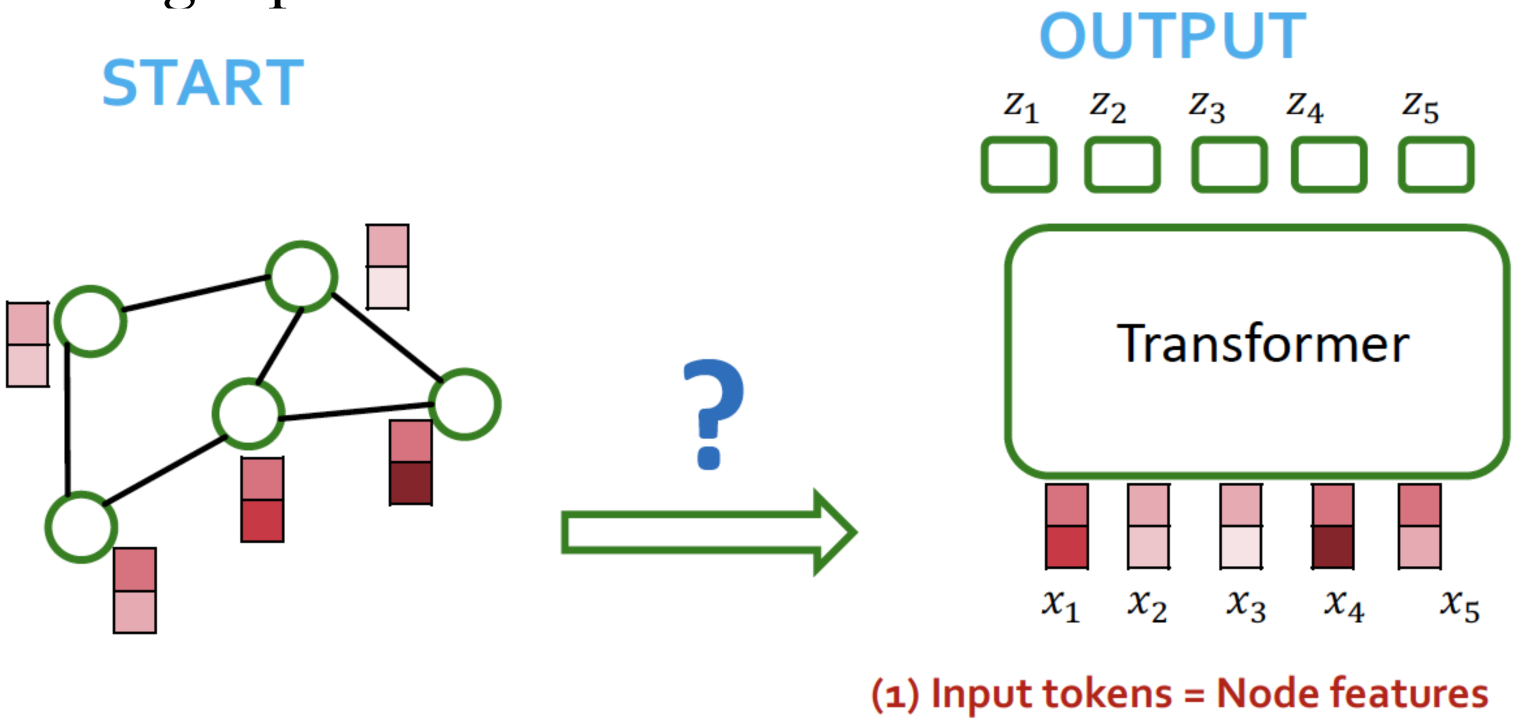




J. You, R. Ying, J. Leskovec. Design Space of Graph Neural Networks, NeurIPS 2020c

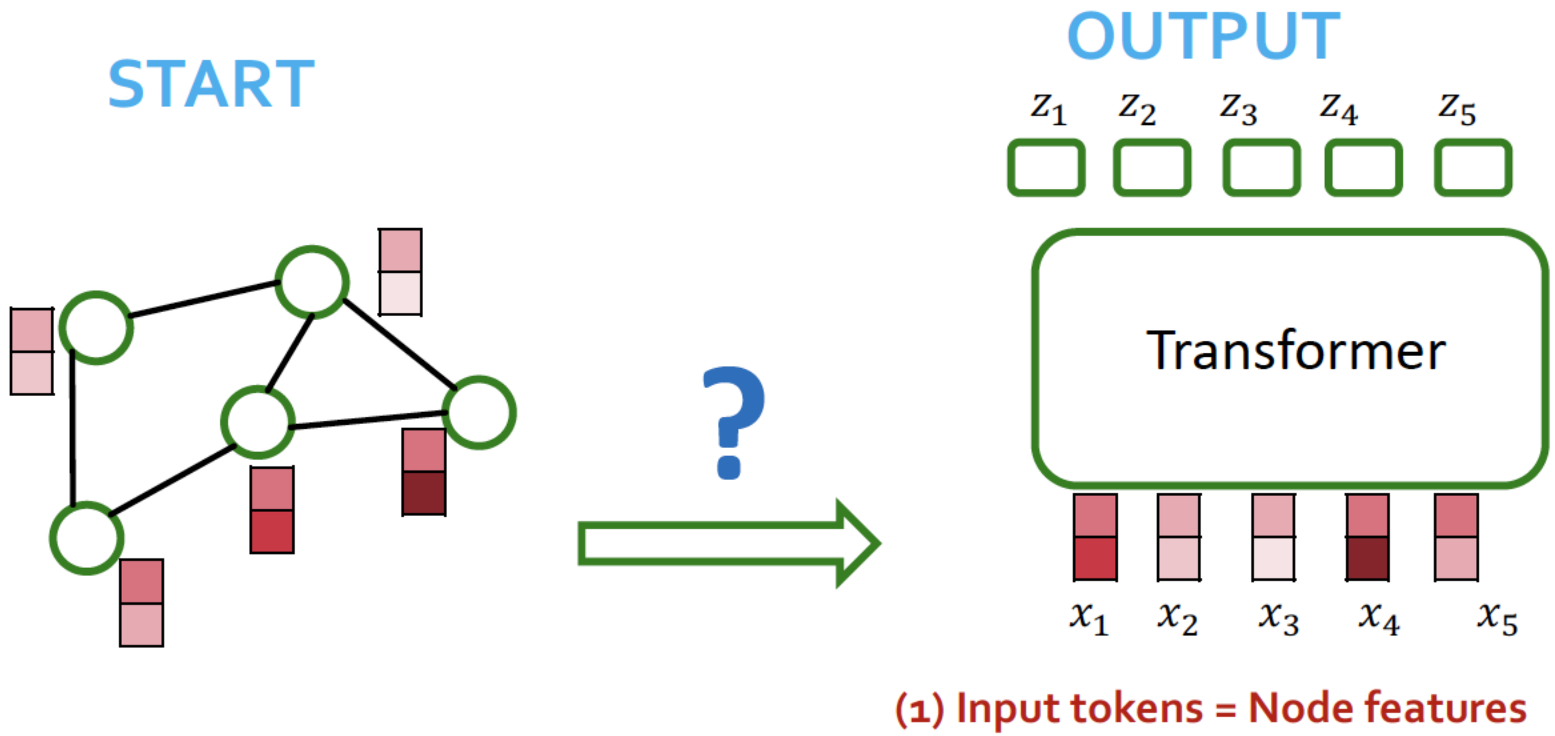
# Nodes as Tokens

- Q1: what should our tokens be?
- Sensible Idea: node features = input tokens
- This matches the setting for the "attention is message passing on the fully connected graph" observation



# Processing Graphs with Transformers

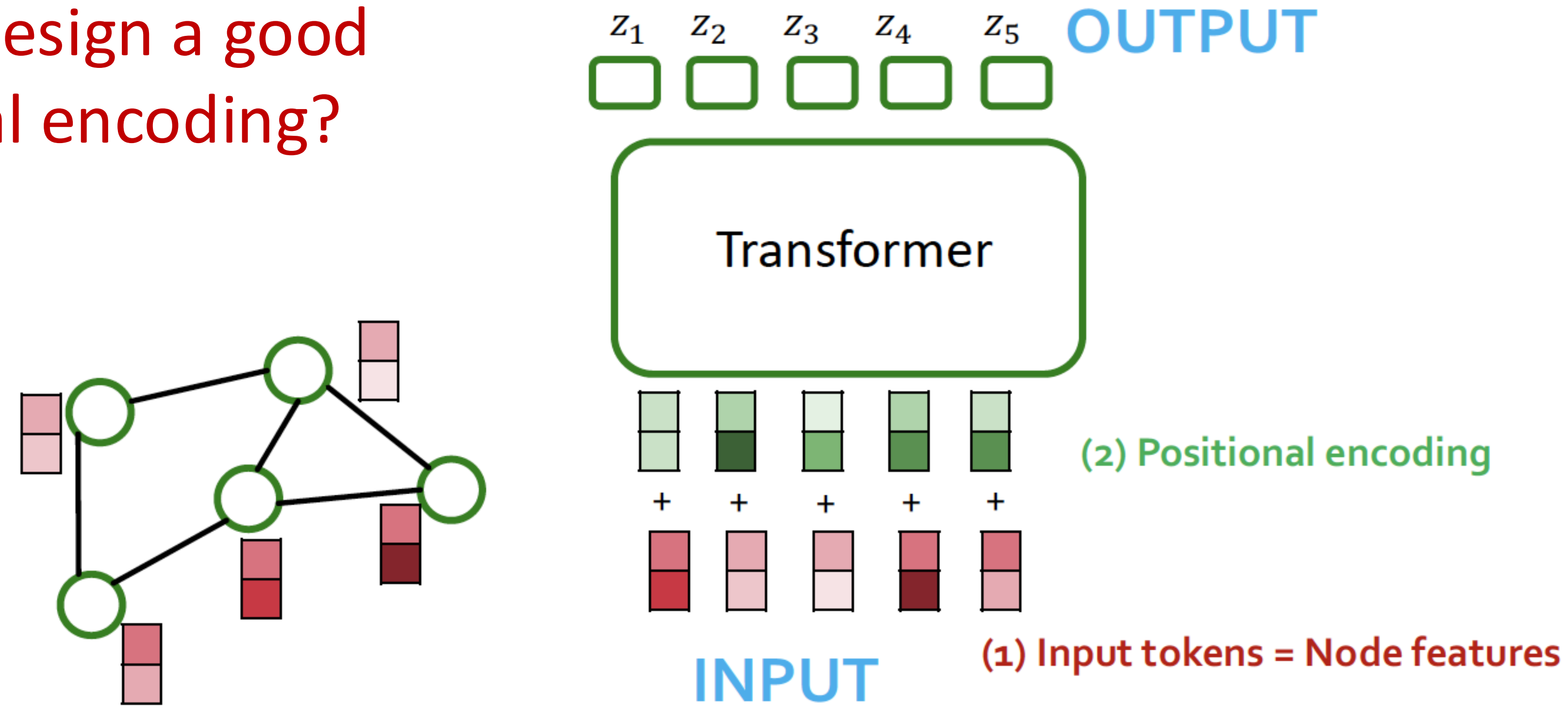
- Problem? We completely lose adjacency info!
- How to also inject adjacency information?



# How to Add Back Adjacency Info?

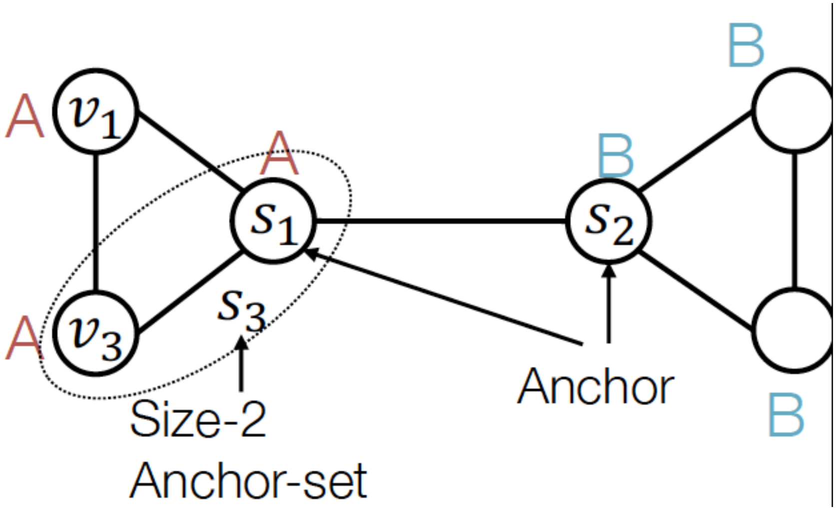
- Idea: Encode adjacency info in the positional encoding for each node
- Positional encoding describes where a node is in the graph

How to design a good positional encoding?



# Option 1: Relative Distances

- Last lecture: positional encoding based on relative distances
- Similar methods based on random walks
- This is a good idea! It works well in many cases
- Especially strong for tasks that require counting cycles




Relative Distances

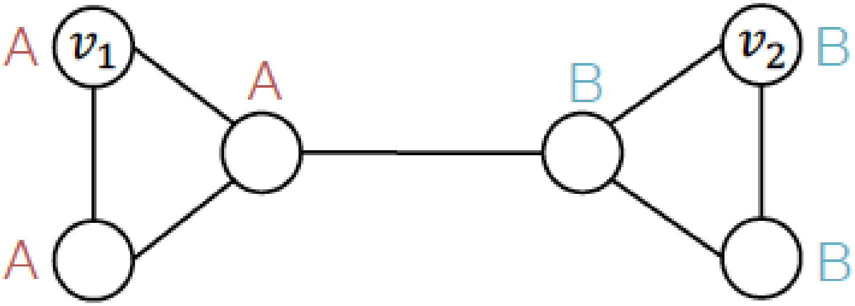
Positional encoding for node $v_1$ =

| | $s_1$ | $s_2$ | $s_3$ |
|---|---|---|---|
| $v_1$ | 1 | 2 | 1 |
| $v_3$ | 1 | 2 | 0 |

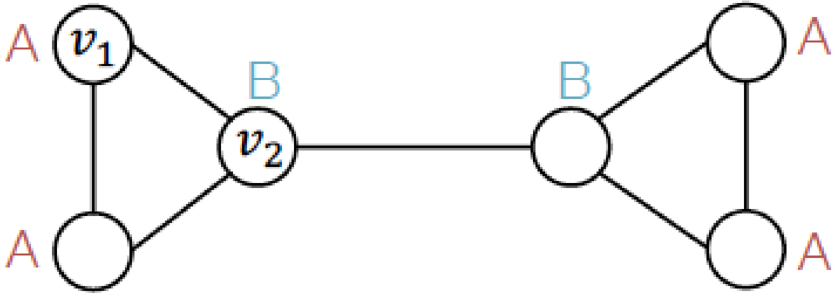
Anchor $s_1$, $s_2$ cannot differentiate node $v_1$, $v_3$, but anchor-set $s_3$ can

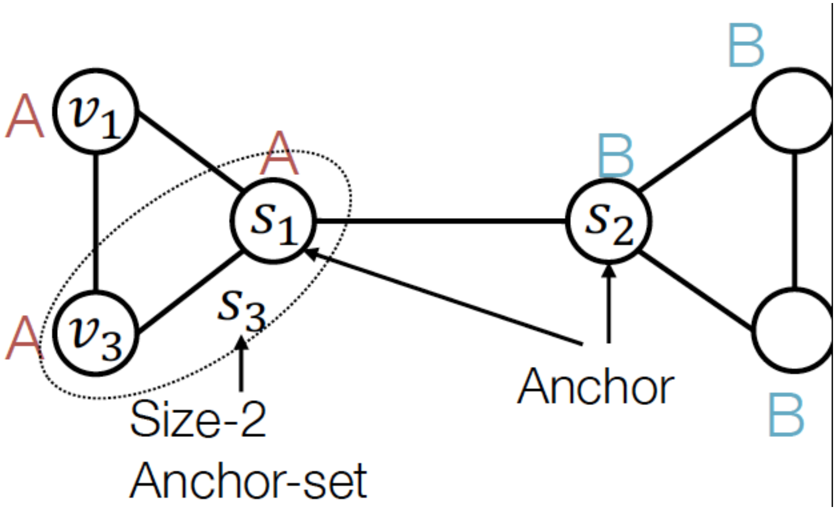
# Option 1: Relative Distances

- Last lecture: Relative distances useful for position-aware task




- But not suited to structure-aware tasks




Relative Distances

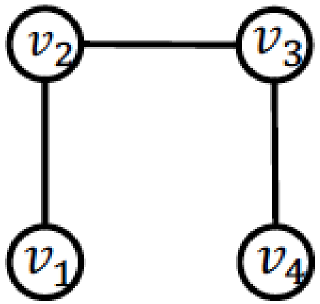
Positional encoding for node $v_1$ =

| | $s_1$ | $s_2$ | $s_3$ |
|---|---|---|---|
| $v_1$ | 1 | 2 | 1 |
| $v_3$ | 1 | 2 | 0 |

Anchor $s_1$, $s_2$ cannot differentiate node $v_1$, $v_3$, but anchor-set $s_3$ can

# Option 2: Laplacian Eigenvector Positional Encodings

- What other ways to make positional encoding?
- Draw on knowledge of Graph Theory (many useful and powerful tools)
- Key object: Laplacian Matrix L = Degrees - Adjacency
  - Each graph has its own Laplacian matrix
  - Laplacian encodes the graph structure
  - Several Laplacian variants that add degree information differently




$$L = \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\text{Degree of each node}} - \underbrace{\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}}_{\text{Adjacency}}$$

# Laplacian Eigenvector Positional Encodings

- Laplacian matrix captures graph structure
- Its eigenvectors inherit this structure
- This is important because eigenvectors are vectors (!) and so can be fed into a Transformer
- Eigenvectors with small eigenvalue = local structure, large eigenvalue = global symmetries

Refresher

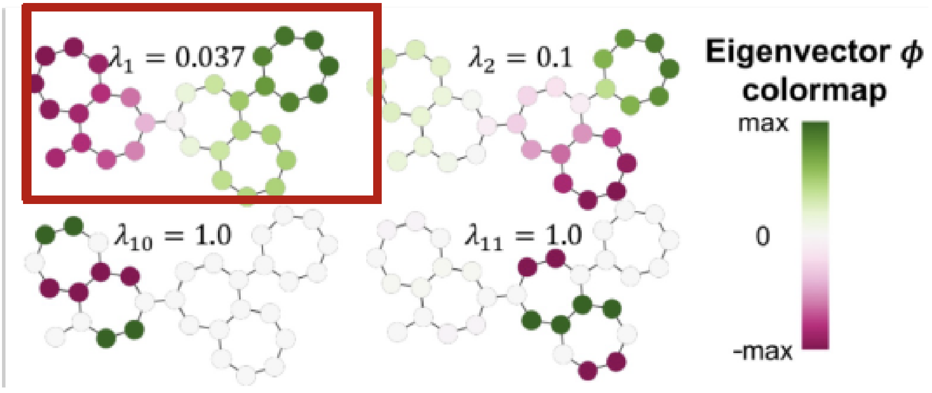
$\boxed{\text{Eigenvector: } v \text{ such that } Lv = \lambda v}$

$L: n \times n$ matrix

$v: n$ dimensional vector

$\lambda:$ Scalar eigenvalue

Visualize one eigenvector

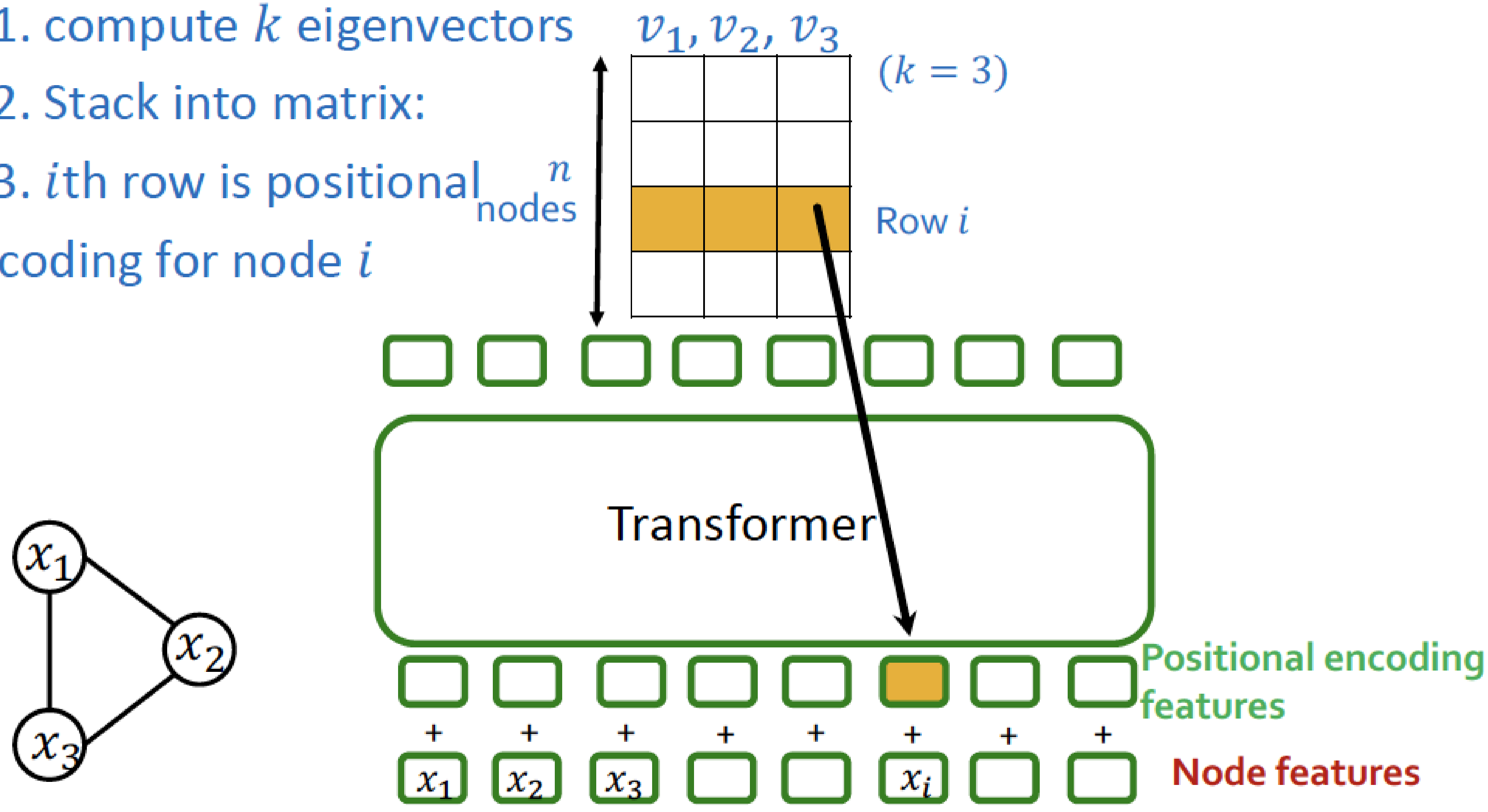



(Figure from Kreuzer* and Beaini* et al. 2021)

# Laplacian Eigenvector Positional Encodings

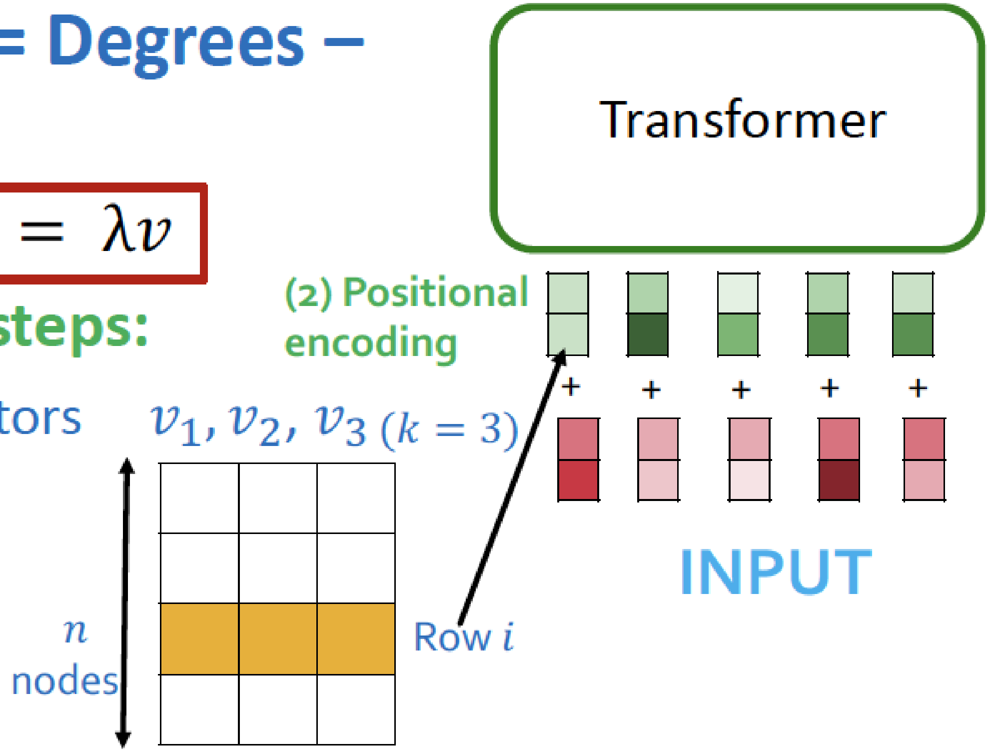
**Positional encoding steps:**

- 1. compute $k$ eigenvectors
- 2. Stack into matrix:
- 3. $i$th row is positional encoding for node $i$



# Summary: Laplacian Eigenvector Positional Encodings

- Laplacian Matrix **L = Degrees – Adjacency**
- $\boxed{\text{Eigenvector: } v \text{ such that } Lv = \lambda v}$
- **Positional encoding steps:**
  - 1. compute $k$ eigenvectors $v_1, v_2, v_3$ $(k = 3)$
  - 2. Stack into matrix:
  - 3. $i$th row is positional encoding for node $i$




- Laplacian Eigenvector positional encodings can also be used with message-passing GNNs
  - **This helps for same reasons as relative-distance based positional encodings in previous lecture**

# Laplacian Eigenvectors in Practice

- Task: given a graph, predict YES if it has a cycle, NO otherwise
- Recall, message-passing cannot solve this task!
- "PE" indicates using Laplacian Eigenvector Pos. Enc.

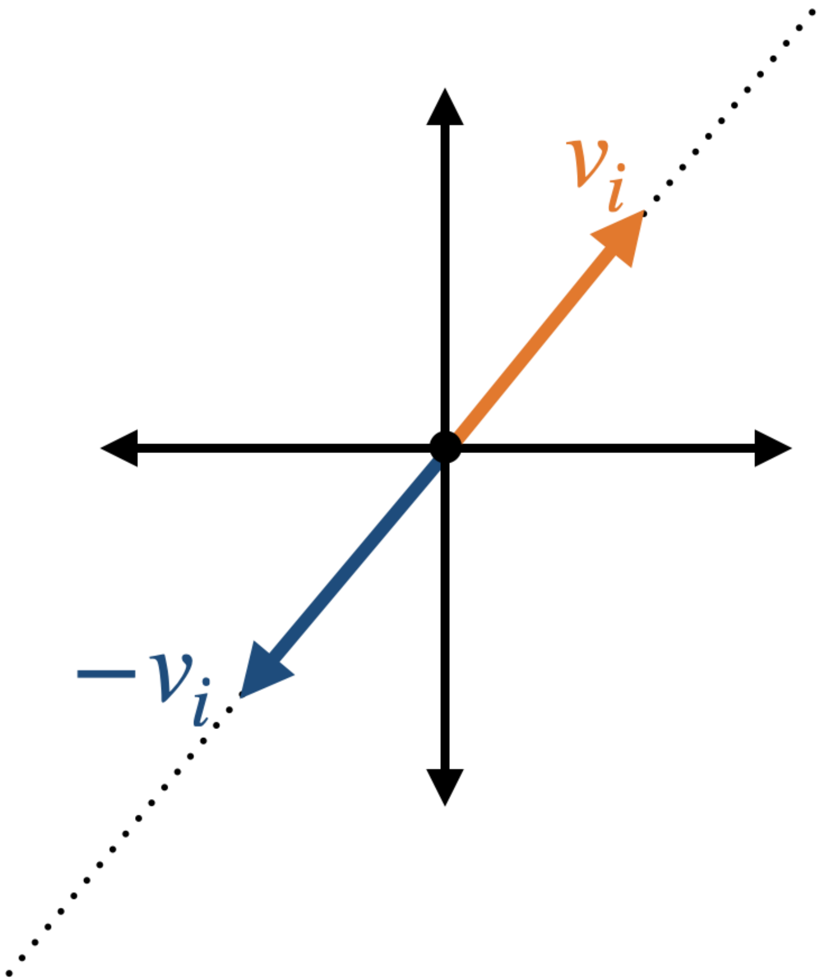
| Train samples → | | | 200 | 500 | 1000 | 5000 |
|---|---|---|---|---|---|---|
| **Model** | $L$ | **#Param** | **Test Acc±s.d.** | | | |
| GIN | 4 | 100774 | 70.585±0.636 | 74.995±1.226 | 78.083±1.083 | 86.130±1.140 |
| GIN-PE | 4 | 102864 | **86.720±3.376** | **95.960±0.393** | **97.998±0.300** | **99.570±0.089** |
| GatedGCN | 4 | 103933 | 50.000±0.000 | 50.000±0.000 | 50.000±0.000 | 50.000±0.000 |
| GatedGCN-PE | 4 | 105263 | **95.082±0.346** | **96.700±0.381** | **98.230±0.473** | **99.725±0.027** |

# Laplacian Eigenvector Positional Encodings

- Laplacian Eigenvector positional encodings work!
- But is this the best we can do?
  - Hint: no
- Q: What is the problem with the current approach?
  - A1: Eigenvectors are not arbitrary vectors
  - A2: They have special structure that we have been ignoring!
- To use eigenvectors properly we must account for their structure in our models

# Eigenvector Sign Ambiguity

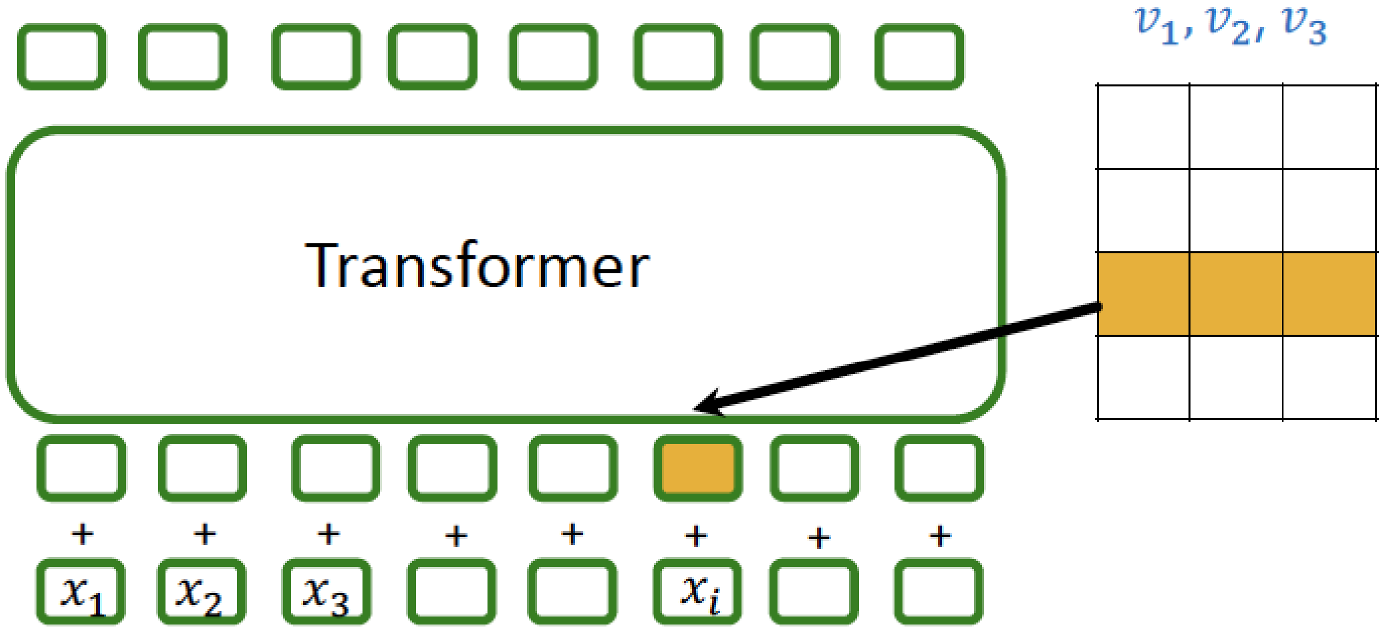
- Suppose $v$ is a Laplacian eigenvector
  - So $Lv = \lambda v$
- But this means:
  - Also $L(-v) = \lambda(-v)$
- So $-v$ is also a Laplacian eigenvector
- The choice of sign is arbitrary!



# Sign Ambiguity is a Problem

- Both
- But when we use them as positional encodings we pick one arbitrarily
- Why does this matter for positional encodings?

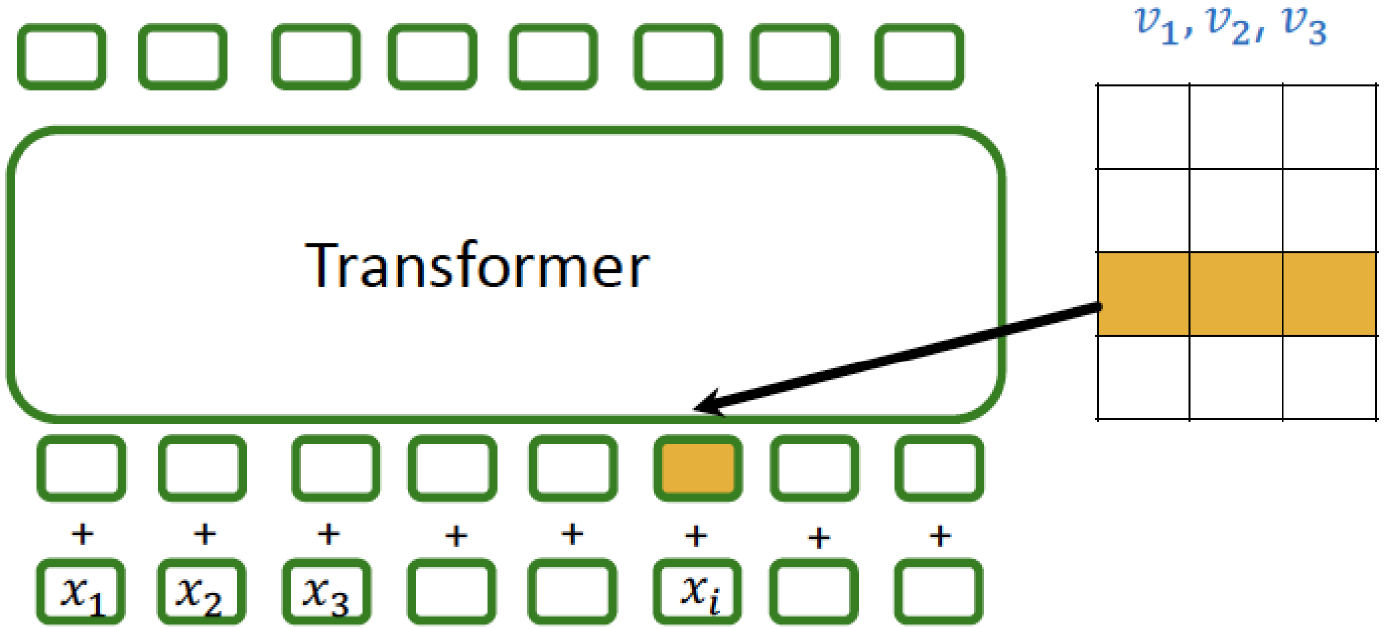
*What if we picked the other sign?*



# Sign Ambiguity is a Problem

- Both
- But when we use them as positional encodings we pick one arbitrarily
- Why does this matter for positional encodings?

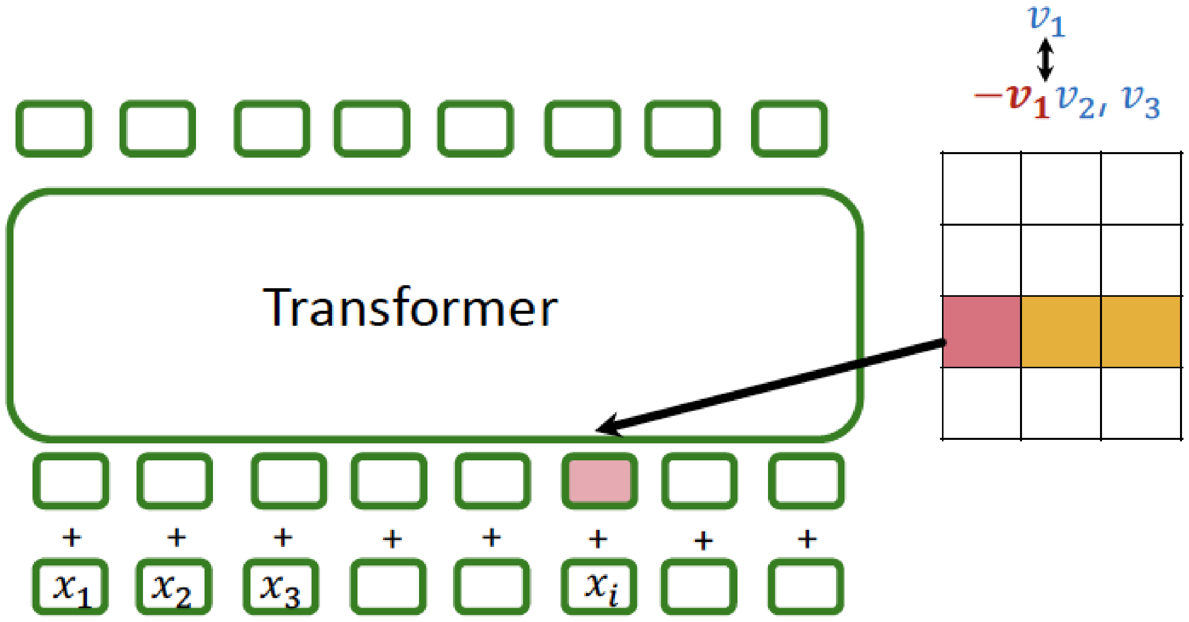
*What if we picked the other sign?*



# Sign Ambiguity is a Problem

- What if we picked the other sign choice?
- Then the input PE changes
- => The models predictions will change!
- For $k$ eigenvectors there are $2^k$ sign choices

*$2^k$ different predictions for the same input graph!*

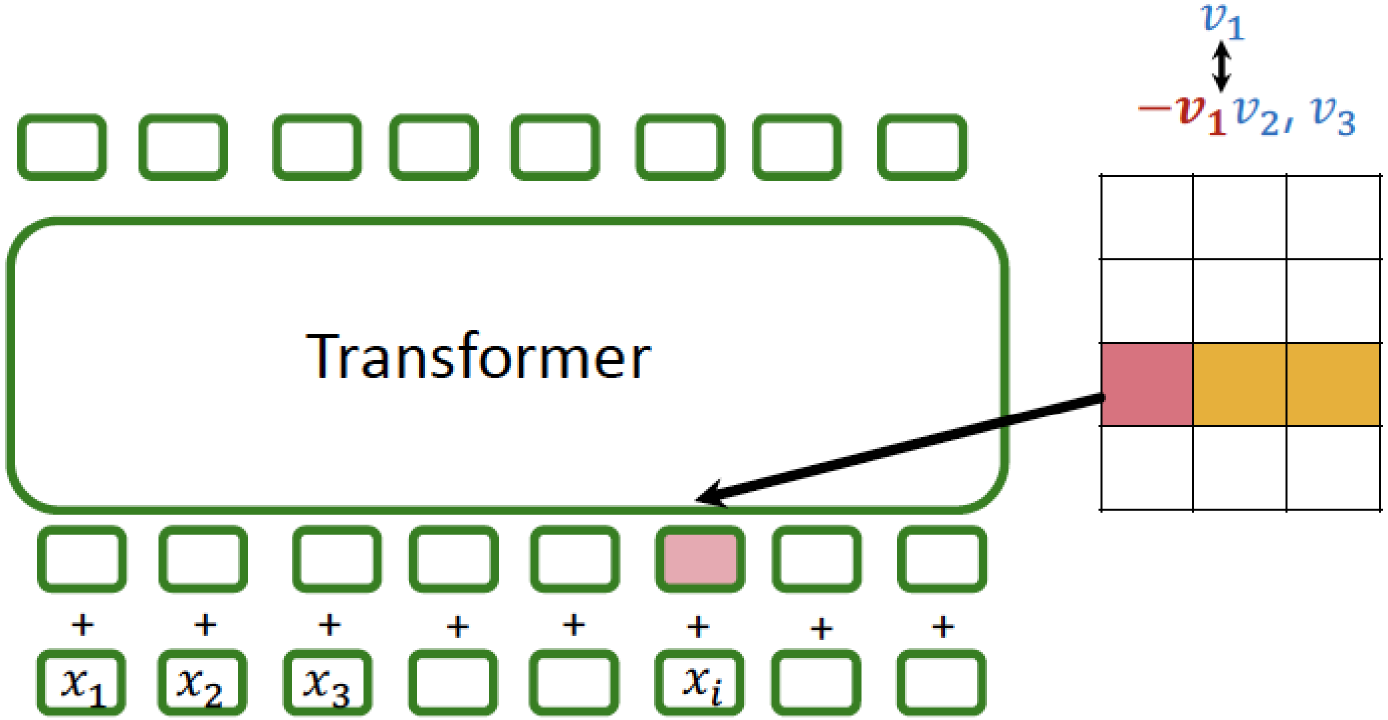


# How to Fix Sign Ambiguity

- Simple Idea: randomly flip the signs of eigenvectors during training
  - I.e., data augmentation
  - Model will learn to not use the sign information
  - Issue: exponentially many sign choices is very difficult to learn

# How to Fix Sign Ambiguity

- Better Idea: build a neural network that is invariant to sign choices!
- Since it is invariant, the predictions will no longer depend on the sign choice



# Sign Invariant Neural Networks

- **Goal:** design a neural network $f(v_1, v_2, \ldots v_k)$ such that:
  - $f(v_1, v_2, \ldots v_k) = f(\pm v_1, \pm v_2, \ldots \pm v_k)$ for all $\pm$ choices
  - $f$ is "expressive": note that $f(v_1, v_2, \ldots v_k) = 0$ is sign invariant... but it's a terrible neural network architecture
- **Warmup: one eigenvector**
  - What about $f(v_1)$ such that $f(v_1) = f(-v_1)$ ?

# Sign Invariant Neural Networks

- Warmup: one eigenvector
- Goal: design a neural network $f(v_1)$ such that $f(v_1) = f(-v_1)$
- Proposition: $f$ satisfies $f(v_1) = f(-v_1)$ if and only if there is a $\phi$ such that $f(v_1) = \phi(v_1) + \phi(-v_1)$

# Sign Invariant Neural Networks

- Warmup: one eigenvector
- Goal: design a sign invariant neural network $f(v_1, v_2, \dots v_k)$ in two steps:
  - Step 1: sign invariant $f_i(v_i)$ for each $i$
  - Step 2: COMBINE individual eigenvector embeddings into one:

$$f(v_1, v_2, \dots v_k) = AGG(f_1(v_1), \dots, f_k(v_k))$$

# Sign Invariant Neural Networks

- Warmup: one eigenvector
- Goal: design a sign invariant neural network $f(v_1, v_2, \ldots v_k)$ in two steps:
  - Step 1: sign invariant $f_i(v_i)$ for each $i$
  - Step 2: COMBINE individual eigenvector embeddings into one:

$$f(v_1, v_2, \ldots v_k) = AGG(f_1(v_1), \ldots, f_k(v_k))$$

**Use model for one eigenvector**

$$f(v_1, v_2, \ldots v_k)$$
$$= AGG(\phi_1(v_1), +\phi_1(-v_1), \ldots, \phi_k(v_k), +\phi_k(-v_k))$$

Combine using another neural net $AGG = \rho$

# Sign Invariant Neural Networks

- Overall model:

$$f(v_1, v_2, \cdots, v_k) = \rho((\phi_1(v_1), +\phi_1(-v_1), \ldots, \phi_k(v_k), +\phi_k(-v_k)\,)$$

- Introducing $k$ distinct neural nets is costly…
- Let's minimize extra parameters by sharing one $\phi$

$$f(v_1, v_2, \ldots v_k)$$
$$= \rho(\phi(v_1), +\phi(-v_1), \ldots, \phi(v_k), +\phi(-v_k))$$

$\rho, \phi =$ any neural network (MLP, GNN etc.)

SignNet

# Sign Invariant Neural Networks

- **Recall Goal:** design a neural network $f(v_1, v_2, \ldots v_k)$ such that:
  - $f(v_1, v_2, \ldots v_k) = f(\pm v_1, \pm v_2, \ldots \pm v_k)$ for all $\pm$ choices
    - **SignNet is sign invariant.**
  - $f$ is "expressive"
    - **Is SignNet expressive?**

$$f(v_1, v_2, \ldots v_k)$$
$$= \rho(\phi(v_1), +\phi(-v_1), \ldots, \phi(v_k), +\phi(-v_k))$$

$\rho, \phi =$ any neural network (MLP, GNN etc.)

**SignNet**

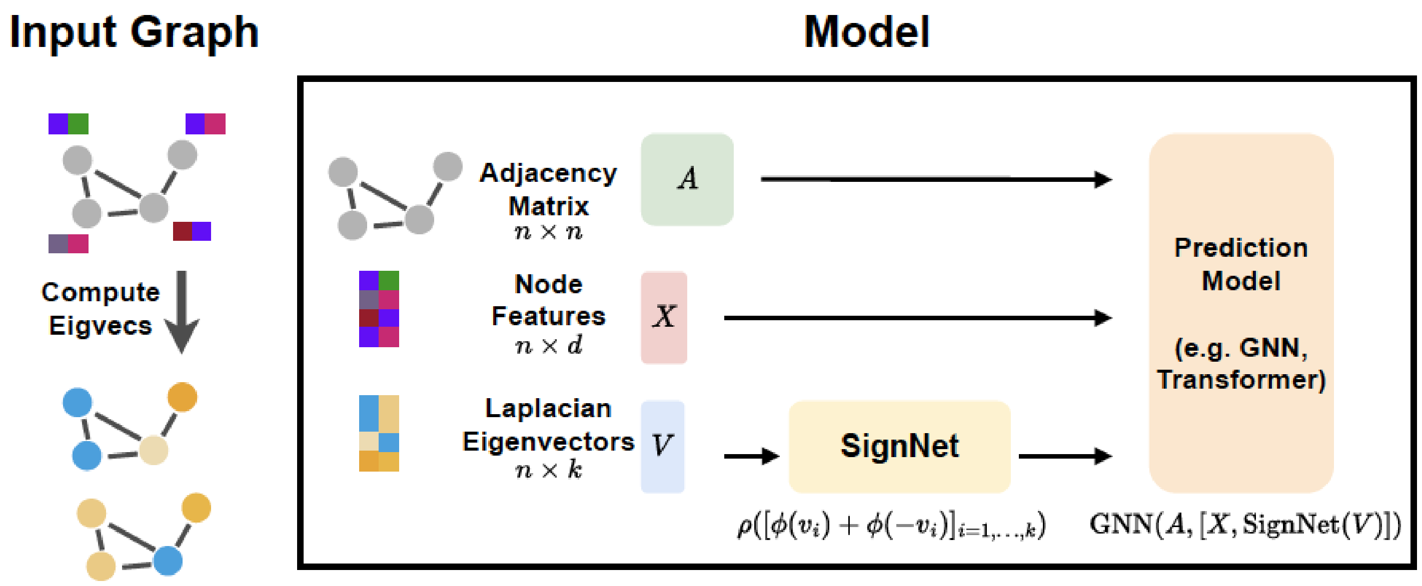
# Sign Invariant Neural Networks

- **Recall Goal:** design a neural network $f(v_1, v_2, \dots v_k)$ such that:
  - $f(v_1, v_2, \dots v_k) = f(\pm v_1, \pm v_2, \dots \pm v_k)$ for all $\pm$ choices
    - **SignNet is sign invariant.**
  - $f$ is "expressive"
    - **Is SignNet expressive?**

> **Theorem:** if $f$ is sign invariant, then there exist functions $\rho, \phi$ such that
>
> $$f(v_1, v_2, \dots v_k) = \rho(\phi(v_1), +\phi(-v_1), \dots, \phi(v_k), +\phi(-v_k))$$
>
> **SignNet can express all sign invariant functions!!**

# SignNet in Practice

- How to use SignNet in practice?
  - Step 1: Compute eigenvectors
  - Step 2: get eigenvector embeddings using SignNet
  - Step 3: concatenate SignNet embeddings with node features X
  - Step 4: pass through main GNN/Transformer as usual.
  - Step 5: Backpropagate gradients to train SignNet + Prediction model jointly.

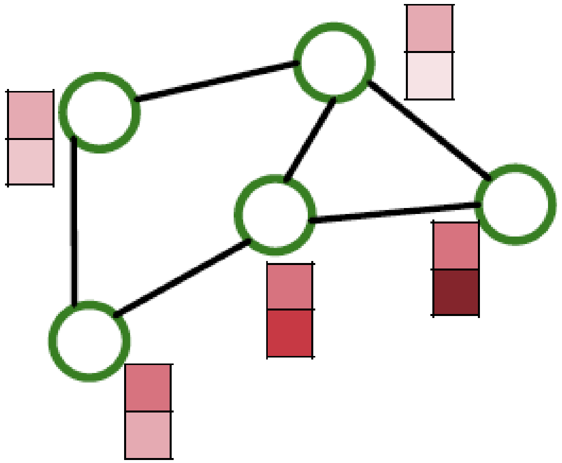


# Processing Graphs with Transformers

- A graph Transformer must take the following inputs:
  - (1) Node features?
  - (2) Adjacency information?
  - (3) Edge features?

- Key components of Transformer
  - (1) tokenizing
  - (2) positional encoding
  - (3) self-attention

So far



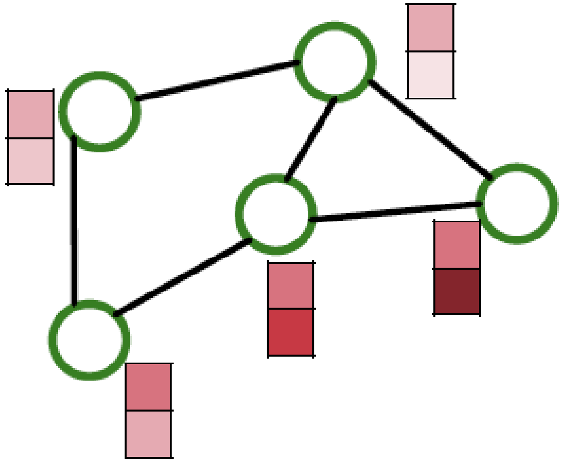
- There are many ways to do this
- Different approaches correspond to different "matchings" between graph inputs (1), (2), (3) transformer components (1), (2), (3)

# Processing Graphs with Transformers

- A graph Transformer must take the following inputs:
  - (1) Node features?
  - (2) Adjacency information?
  - (3) Edge features?

Left to do

- Key components of Transformer
  - (1) tokenizing
  - (2) positional encoding
  - (3) self-attention



- There are many ways to do this
- Different approaches correspond to different "matchings" between graph inputs (1), (2), (3) transformer components (1), (2), (3)

# Edge Features in Self-Attention

- Not clear how to add edge features in the tokens or positional encoding
- How about in the attention? $Att(X) = softmax(K^T Q)V$
- $[k_{ij}] = K^T Q$ is an n x n matrix. Entry $k_{ij}$ describes "how much" token j contributes to the update of token i



Do Transformers Really Perform Bad for Graph Representation? Ying et al. NeurIPS 2021

# Edge Features in Self-Attention

- Not clear how to add edge features in the tokens or positional encoding
- How about in the attention? $Att(X) = softmax(K^T Q)V$
- $[k_{ij}] = K^T Q$ is an n x n matrix. Entry $k_{ij}$ describes "how much" token j contributes to the update of token i
- Idea: adjust $\boldsymbol{k}_{ij}$ based on edge features. Replace with $\boldsymbol{k_{ij}} + \boldsymbol{c_{ij}}$ where $\boldsymbol{c_{ij}}$ depends on the edge features


[Do Transformers Really Perform Bad for Graph Representation?](#) Ying et al. NeurIPS 2021

# Edge Features in Self-Attention

- Not clear how to add edge features in the tokens or positional encoding
- How about in the attention? $Att(X) = softmax(K^T Q)V$
- $[k_{ij}] = K^T Q$ is an n x n matrix. Entry $k_{ij}$ describes "how much" token j contributes to the update of token i
- Idea: adjust $\boldsymbol{k}_{ij}$ based on edge features. Replace with $\boldsymbol{k_{ij}} + \boldsymbol{c_{ij}}$ where $\boldsymbol{c_{ij}}$ depends on the edge features

**Implementation:**

Learned parameters $w_1$

- If there is an edge between $i$ and $j$ with features $e_{ij}$, define $c_{ij} = w_1^T e_{ij}$
- If there is no edge, find shortest edge path between $i$ and $j$ $(e^1, e^2, \dots e^N)$ and define $c_{ij} = \sum_n w_n^T e^n$

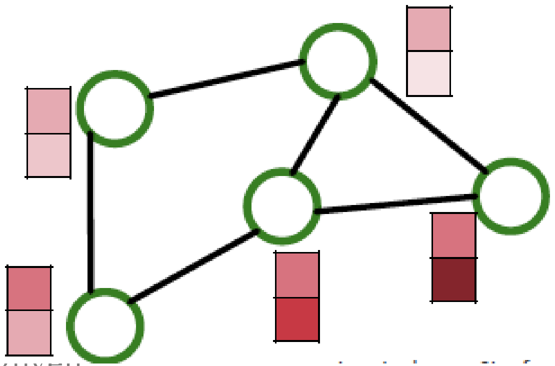
Learned parameters $w_1, \dots, w_N$

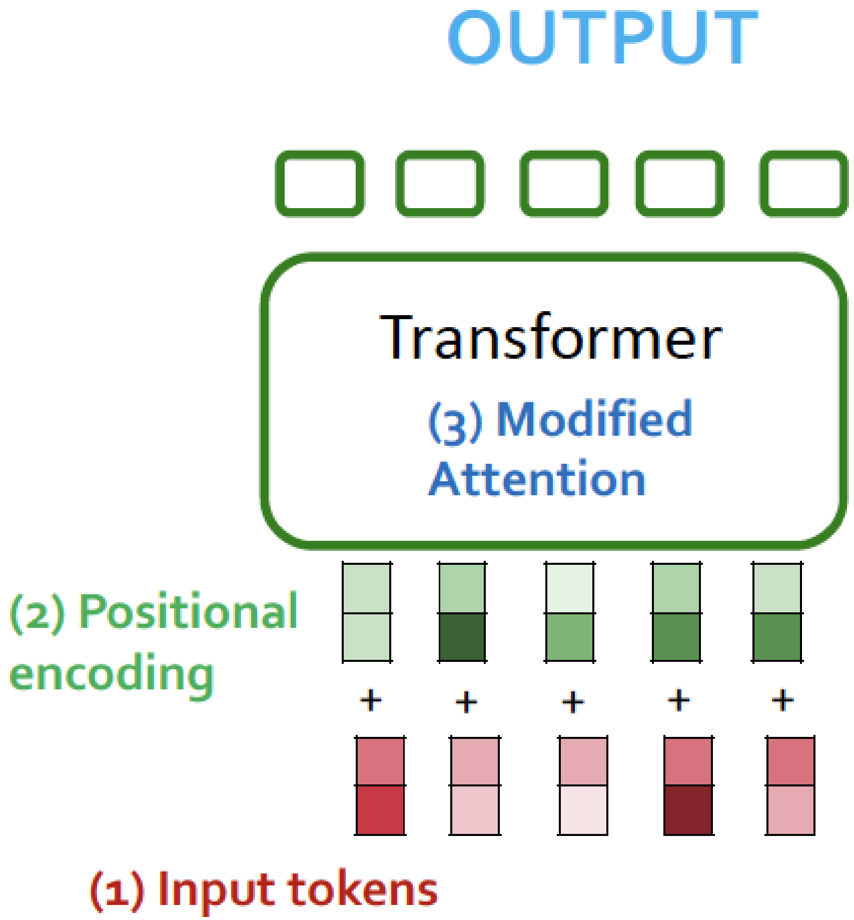


[Do Transformers Really Perform Bad for Graph Representation?](#) Ying et al. NeurIPS 2021

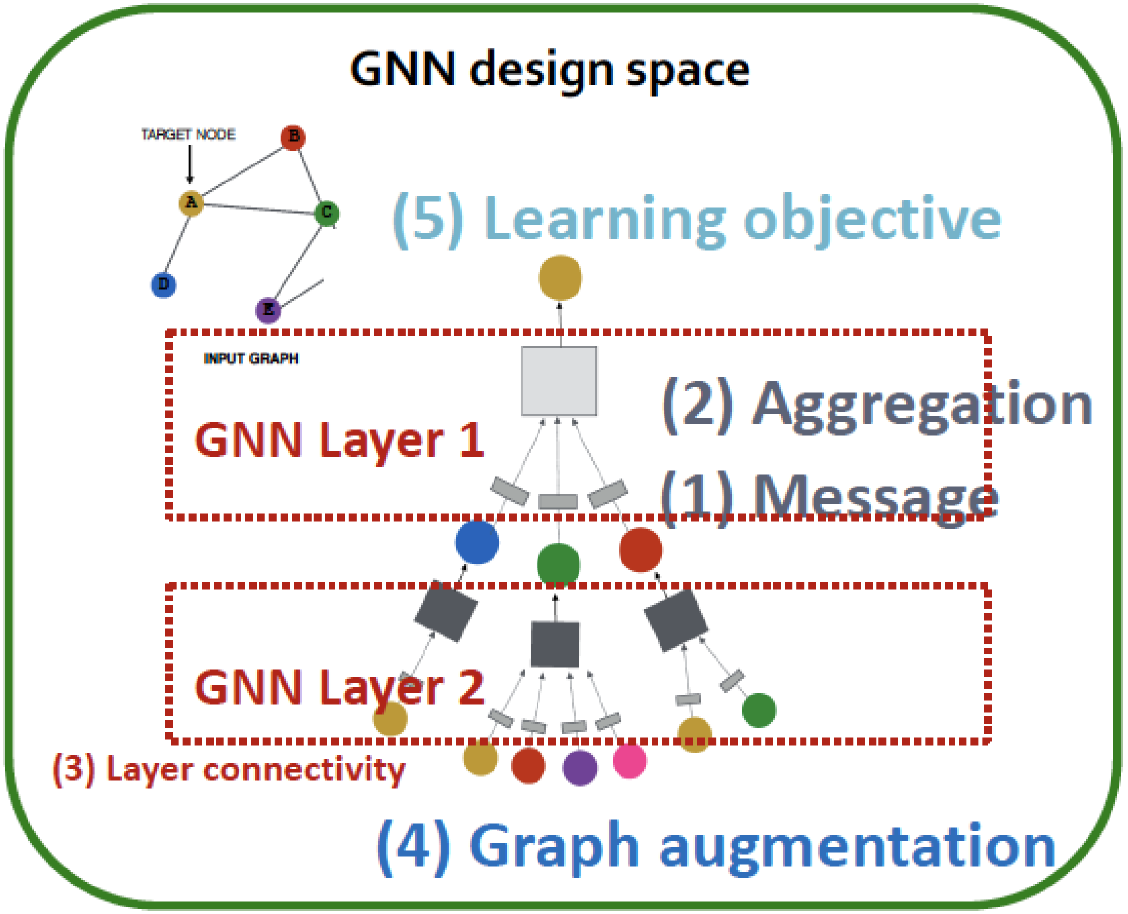
# Summary: Graph Transformer Design Space

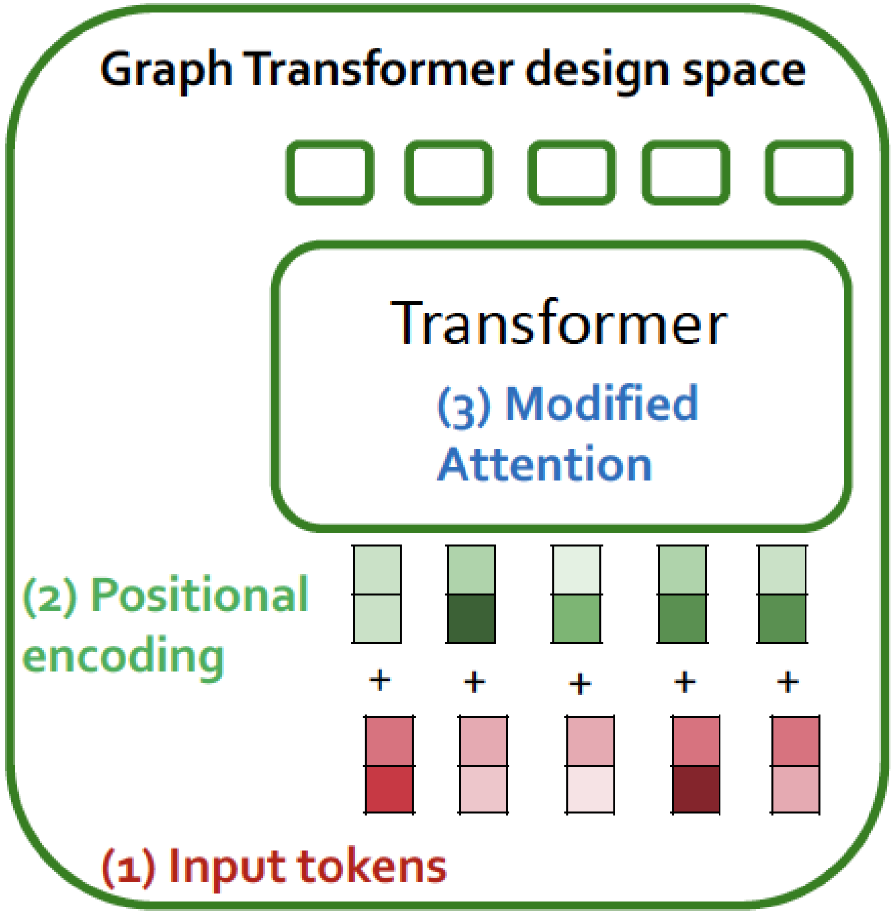
- (1) Tokenization
  - Usually node features
  - Other options, such as subgraphs, and node + edge features (not discussed today)
- (2) Positional Encoding
  - Relative distances, or Laplacian eigenvectors
  - Gives Transformer adjacency structure of graph
- (3) Modified Attention
  - Reweight attention using edge features





# Summary: Graph Transformer Design Space



# Class Project

# List of Project Proposals (23 Students?)

1. Le Xuan Nam, Do Ngoc Trung. "*Substructure–substructure interactions for drug–drug interaction prediction*".
2. Dao Phan Khai, Dang Quang Thang, Nguyen Doan Hieu. "*A Graph-based Approach for Android Malware Detection*".
3. Le Ngoc Lam, Dao Duc Manh. "*Completing a Temporal Knowledge Base Graph on Diachronic Entity Embedding and Applications*".
4. Philippe Dufresne. "*Application of Graph Neural Networks in Media Content Recommendation Systems*".
5. Tran Tien Bang, Nguyen Duc Manh. "*Recommender Systems*".
6. Nguyen Tieu Anh. "*Friend Recommendation*".
7. Do Hoang Dung, Nguyen Thanh Son. "*Stock Price Prediction*".
8. Tong Ngoc Anh, Phung Thu Hang. "*Book Recommendation System with Graph-Based Models*".
9. Tran Xuan Tung. "*Graph Neural Network For Receipt Information Extraction*".
10. Uvin Wijesinghe. "Enhancing Temporal Relational Graph Neural Networks for Stock Prediction".
11. Nguyen Tuan Dung, Nguyen Minh Chau. "*Temporal Health Event Prediction with Dynamic Disease Graphs Modelling*".
12. Dao Minh Tuan, Nguyen Danh Phuc. "*Recommender Systems*".
13. Vu Duc Anh, Nguyen Duc Manh. "*Applying Collaborative Filtering Graph Neural Network (CF-GNN) for recommender system of video social media platforms*".

# Class Projects: Components

- Project proposal (10%)
  - Deadline: has already passed

- Project report (60%)
  - Deadline: October 21st, 2024.

- Project presentation (20%)
  - Tuesdays, October 22nd and October 29th.

# Class Projects: Project report

- Writing (40 points): 10-15 pages
  - Motivation & explanation of data/task (9 points)
  - Appropriateness & explanation of model(s) (9 points)
  - Insights + results (9 points)
  - Figures (9 points)
  - Code snippets (4 points)
- Submission:
  - Format: NeurIPS2024 LaTeX style
  - Message me the file on the class Slack channel.
- Colab (20 points)
  - Code: correctness, design (10 points)
  - Documentation: class/function descriptions, comments in code (10 points)

# Class Projects: Project presentation

- Present at class with Q&A

- Time:
  - Group of 1 student: 15-20 minutes
  - Group of 2 students: 25 minutes
  - Group of 3 students: 30 minutes