Stanford CS224W: Graph Transformers

CS224W: Machine Learning with Graphs

Charilaos Kanatsoulis and Jure Leskovec, Stanford University

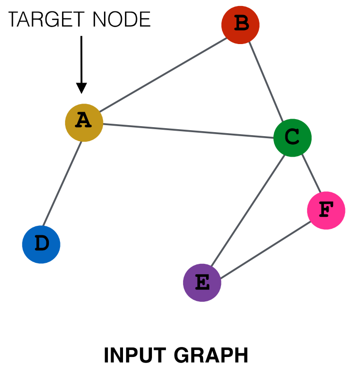
http://cs224w.stanford.edu

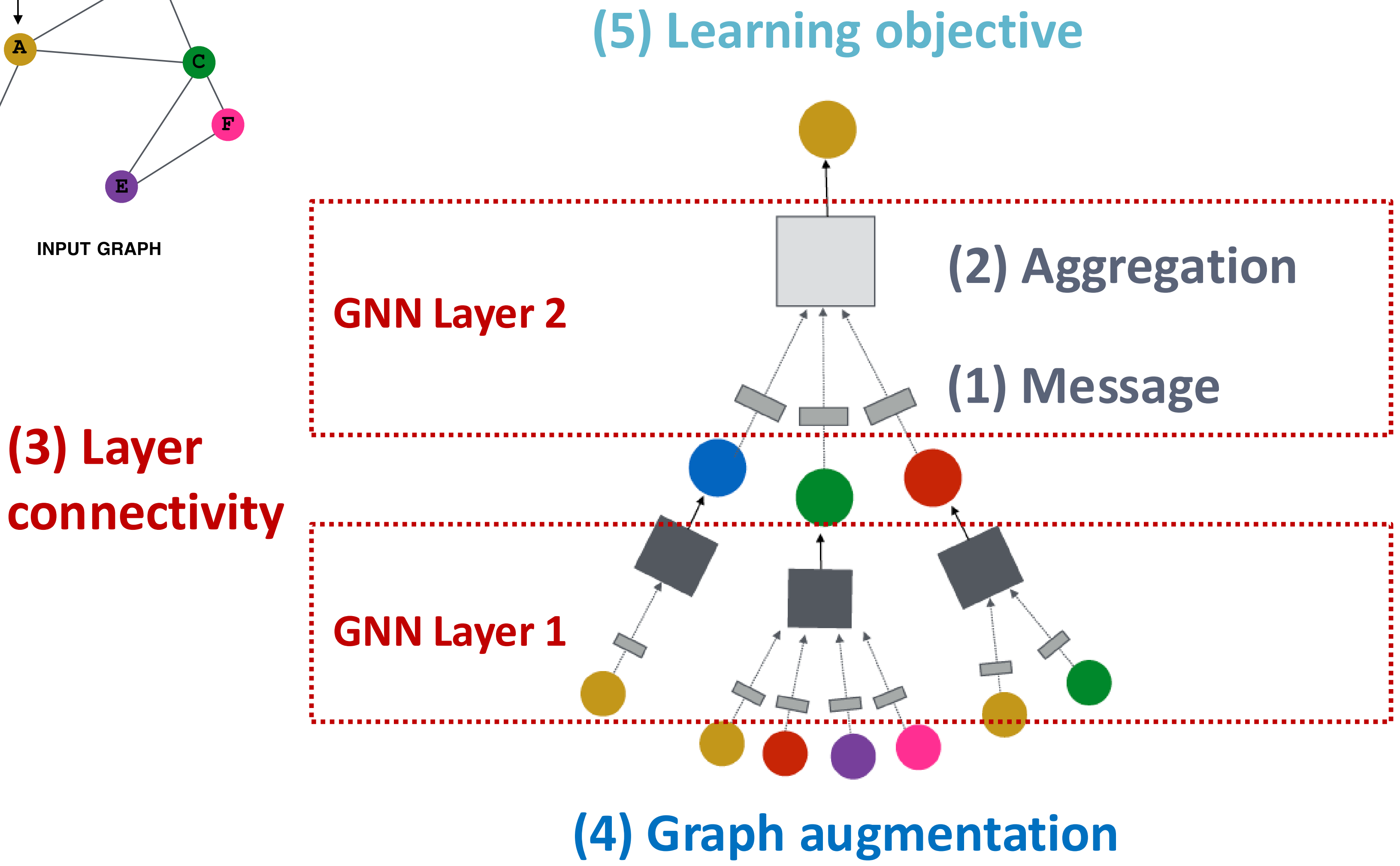
Announcements

- **Homework 1** due tonight at 11:59PM
  - Late submissions accepted until end of day Monday 10/20
- **Project Proposal** due Tuesday 10/21
- **Colab 2** due Thursday 10/23
- **Regrade request deadlines**
  - **Colab 1**: Thursday 10/23

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

Recap: A General GNN Framework



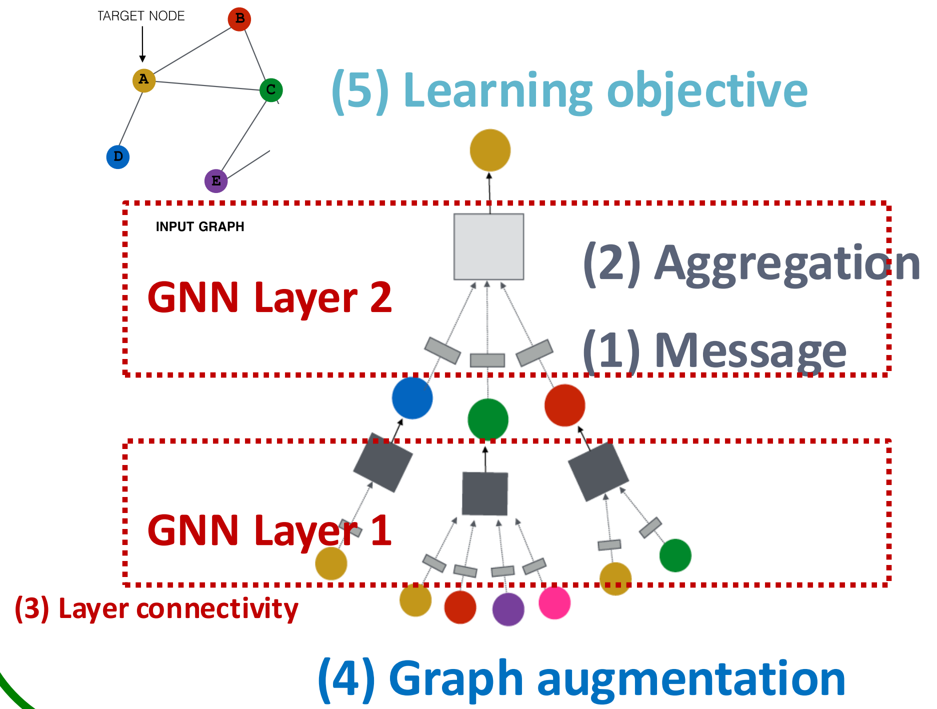


J. You, R. Ying, J. Leskovec. Design Space of Graph Neural Networks, NeurIPS 2020c

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?**

GNN design space



Graph Transformer design space

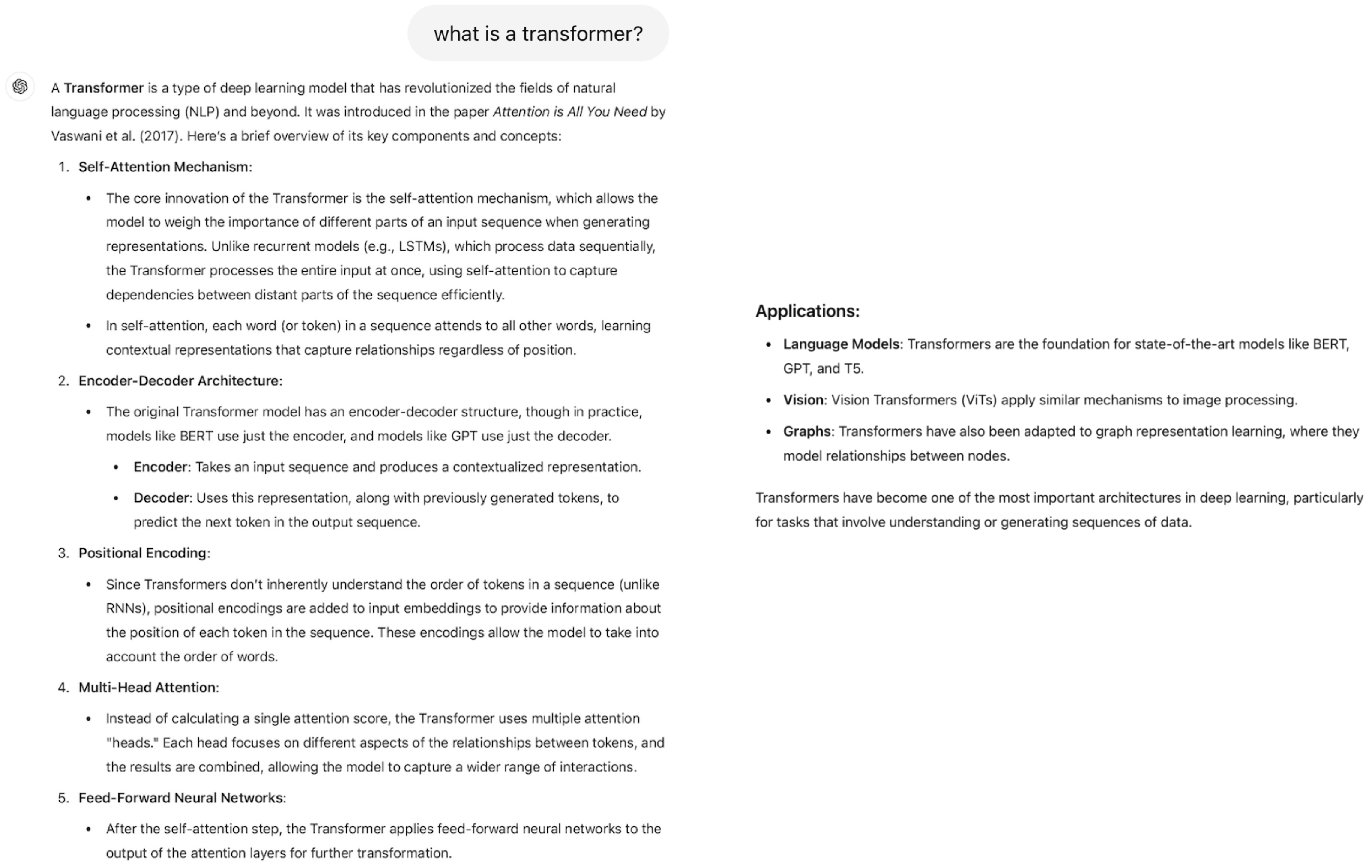
Stanford CS224W: Intro to Transformers

CS224W: Machine Learning with Graphs

Charilaos Kanatsoulis and Jure Leskovec, Stanford University

http://cs224w.stanford.edu

The Backbone of Generative AI



what is a transformer?

A **Transformer** is a type of deep learning model that has revolutionized the fields of natural language processing (NLP) and beyond. It was introduced in the paper *Attention is All You Need* by Vaswani et al. (2017). Here's a brief overview of its key components and concepts:

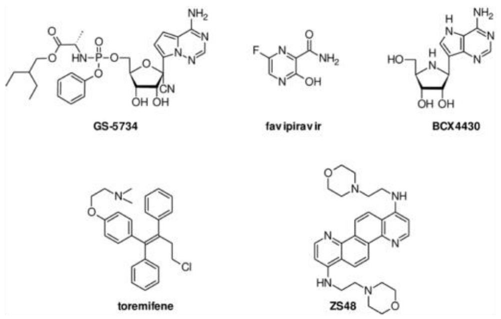
1. **Self-Attention Mechanism**:
   - The core innovation of the Transformer is the self-attention mechanism, which allows the model to weigh the importance of different parts of an input sequence when generating representations. Unlike recurrent models (e.g., LSTMs), which process data sequentially, the Transformer processes the entire input at once, using self-attention to capture dependencies between distant parts of the sequence efficiently.
   - In self-attention, each word (or token) in a sequence attends to all other words, learning contextual representations that capture relationships regardless of position.
2. **Encoder-Decoder Architecture**:
   - The original Transformer model has an encoder-decoder structure, though in practice, models like BERT use just the encoder, and models like GPT use just the decoder.
     - **Encoder**: Takes an input sequence and produces a contextualized representation.
     - **Decoder**: Uses this representation, along with previously generated tokens, to predict the next token in the output sequence.
3. **Positional Encoding**:
   - Since Transformers don't inherently understand the order of tokens in a sequence (unlike RNNs), positional encodings are added to input embeddings to provide information about the position of each token in the sequence. These encodings allow the model to take into account the order of words.
4. **Multi-Head Attention**:
   - Instead of calculating a single attention score, the Transformer uses multiple attention "heads." Each head focuses on different aspects of the relationships between tokens, and the results are combined, allowing the model to capture a wider range of interactions.
5. **Feed-Forward Neural Networks**:
   - After the self-attention step, the Transformer applies feed-forward neural networks to the output of the attention layers for further transformation.

Applications:

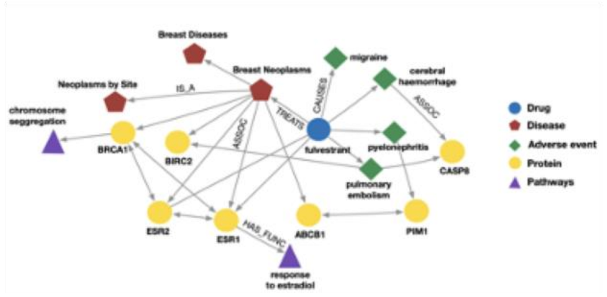
- **Language Models**: Transformers are the foundation for state-of-the-art models like BERT, GPT, and T5.
- **Vision**: Vision Transformers (ViTs) apply similar mechanisms to image processing.
- **Graphs**: Transformers have also been adapted to graph representation learning, where they model relationships between nodes.

Transformers have become one of the most important architectures in deep learning, particularly for tasks that involve understanding or generating sequences of data.

Goal: Bring Power of Transformers to Graphs



Small molecules



Knowledge graphs

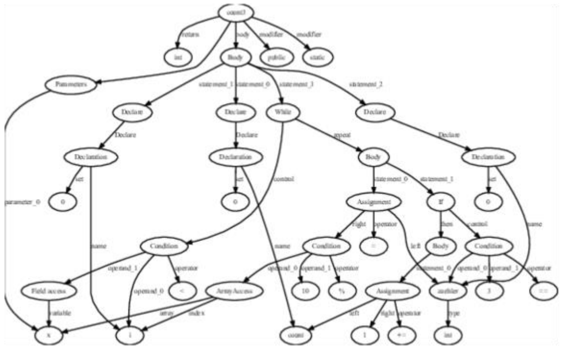

Code graphs

Image credit: ResearchGate

Communication networks

Image credit: Lumen Learning

x

- **There is lots of multi-billon node/graph scale data to learn from**

Plan for Today

- **Part 1:**
  - Introducing Transformers
  - Relation to message passing GNNs
- **Part 2:**
  - A new design landscape for graph Transformers
- **Part 3 (time permitting):**
  - PEARL: Learning Efficient Positional Encodings with GNNs

Stanford CS224W: Transformers

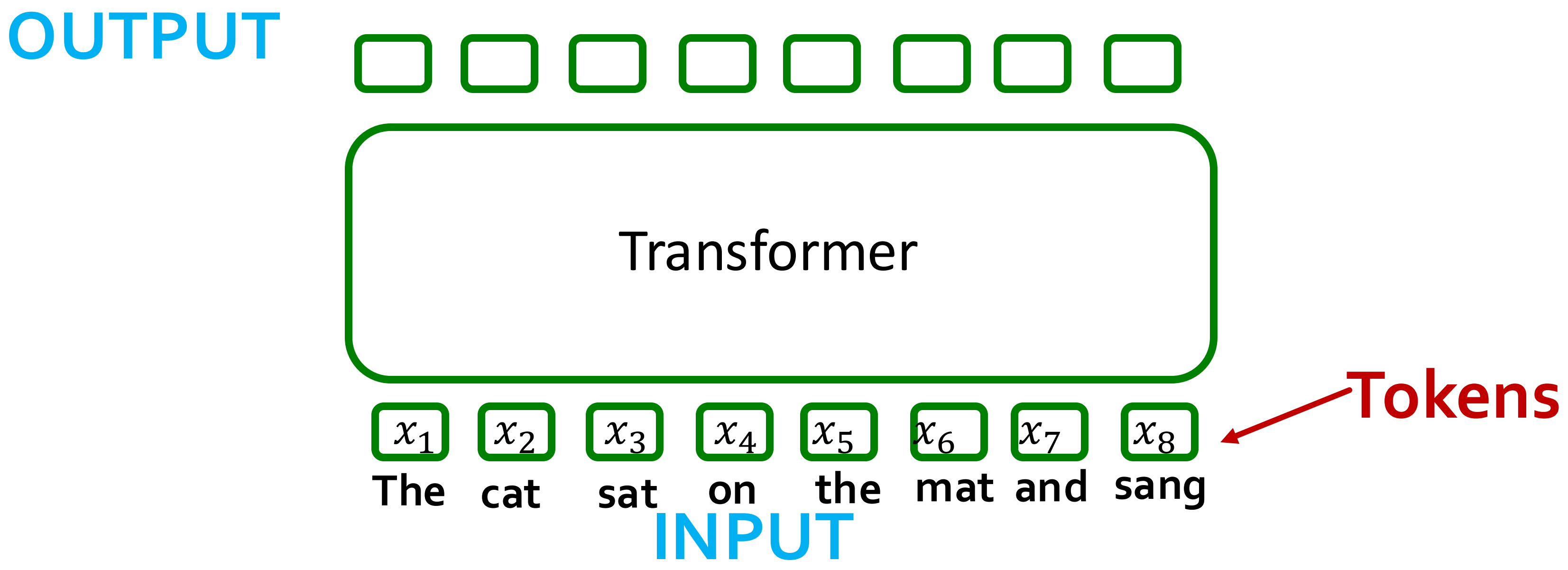
CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

http://cs224w.stanford.edu

Transformers Ingest Tokens

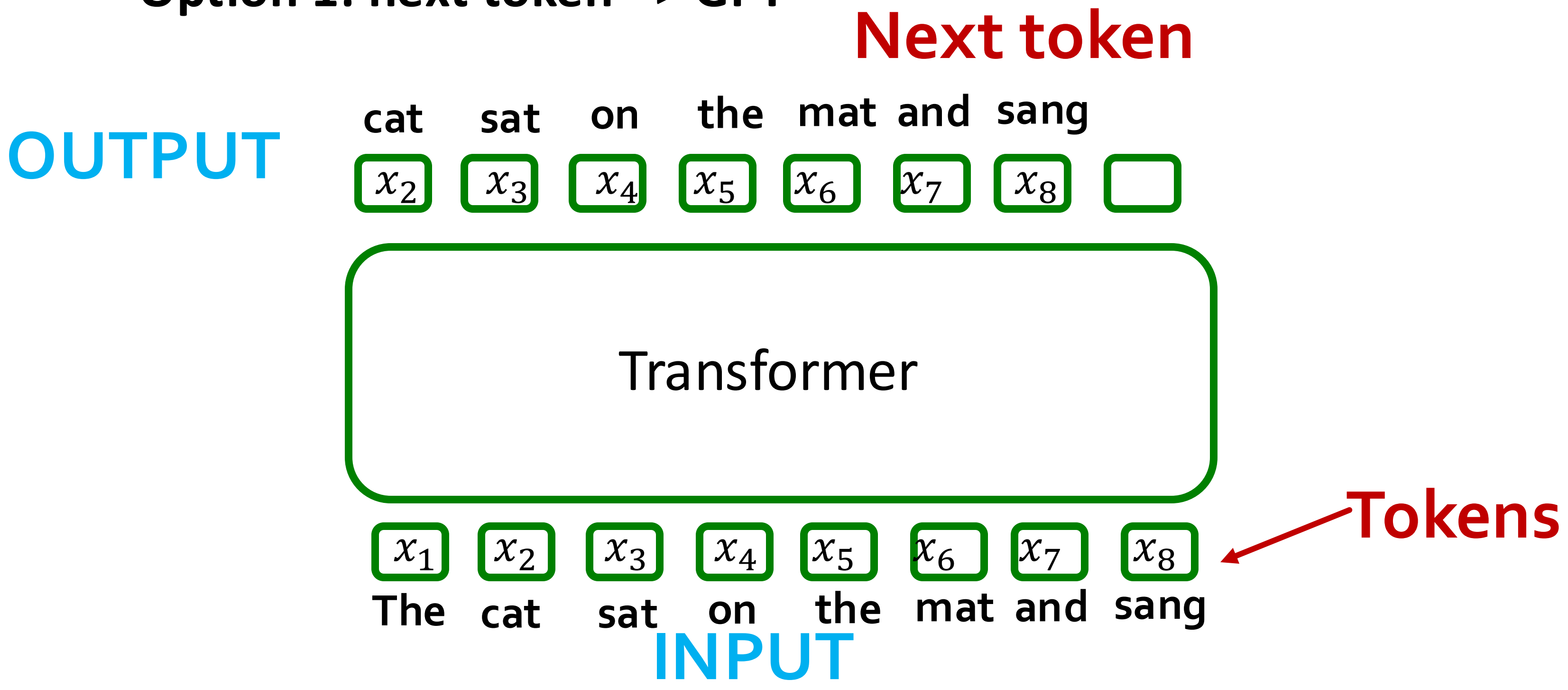
- **Transformers map 1D sequences of vectors to 1D sequences of vectors**



OUTPUT
Transformer
Tokens
x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8
The cat sat on the mat and sang
INPUT

Transformers Ingest Tokens

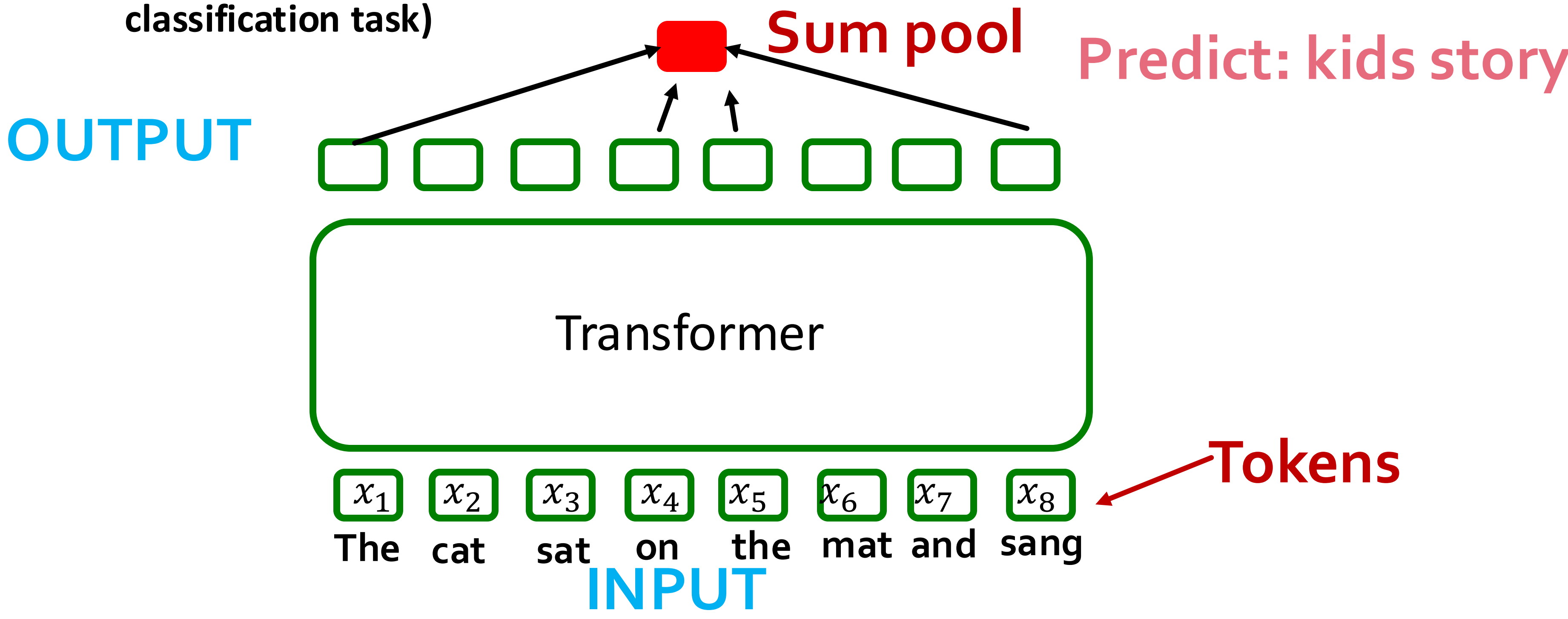
- Transformers map 1D sequences of vectors to 1D sequences of vectors known as tokens
  - Tokens describe a ”piece” of data – e.g., a word
- What output sequence?
  - Option 1: next token => GPT



Next token
OUTPUT
cat sat on the mat and sang
x_2 x_3 x_4 x_5 x_6 x_7 x_8
Transformer
Tokens
x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8
The cat sat on the mat and sang
INPUT

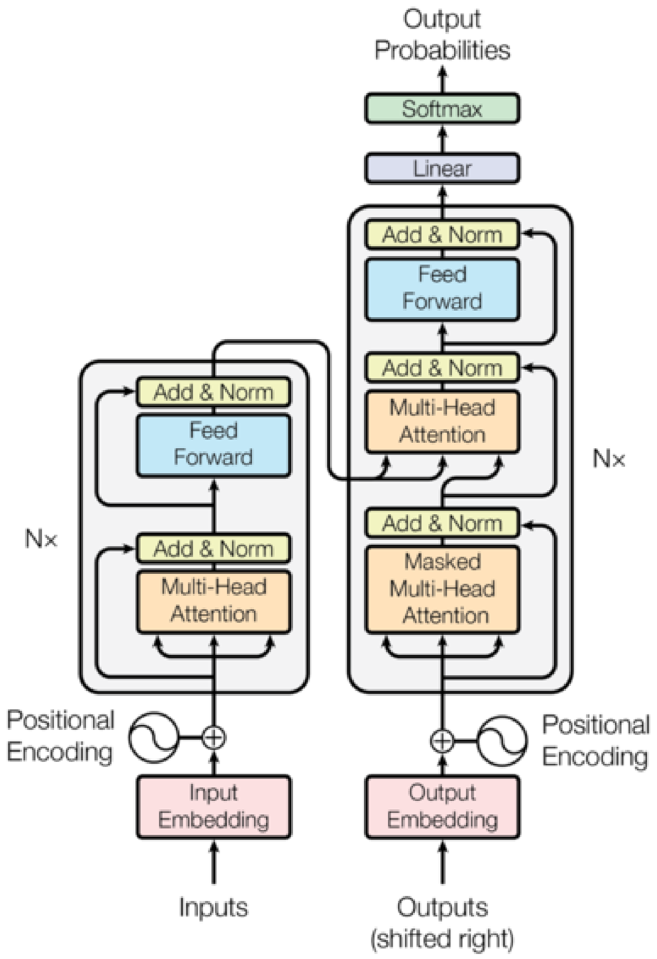
Transformers Ingest Tokens

- Transformers map 1D sequences of vectors to 1D sequences of vectors known as tokens
  - 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)


Sum pool
Predict: kids story
OUTPUT
Transformer
Tokens
x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8
The cat sat on the mat and sang
INPUT

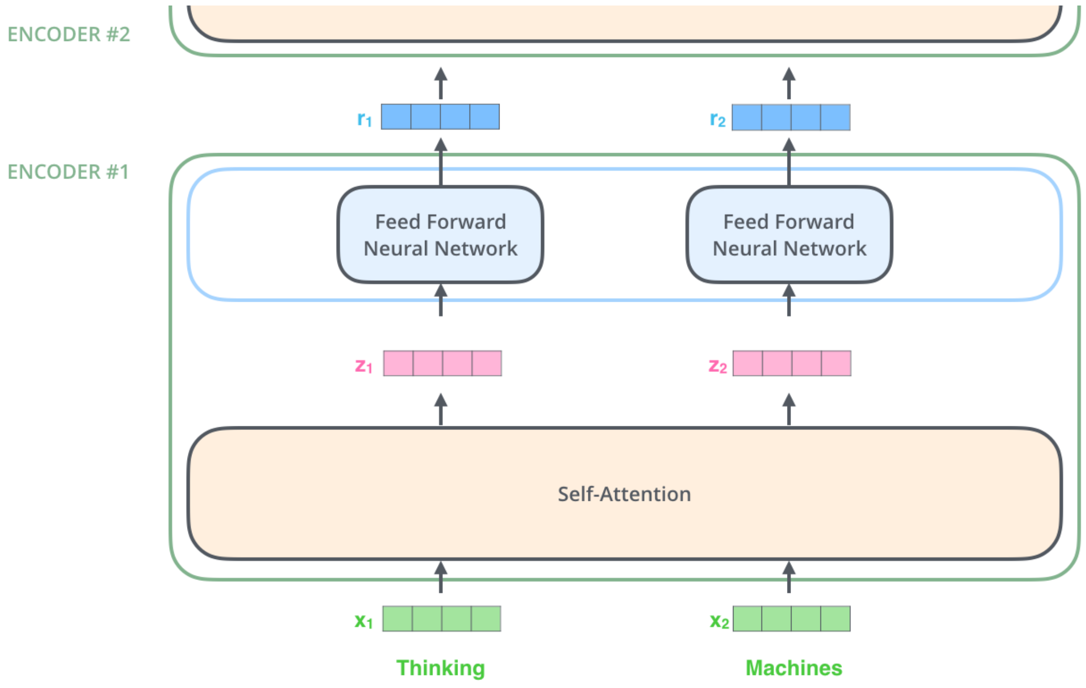
Transformer Blueprint

- How are tokens processed?
- Lots of components
  - Normalization
  - Feed forward networks
  - Positional encoding (more later)
  - Multi-head self-attention
- What does self-attention block do?



Self-attention

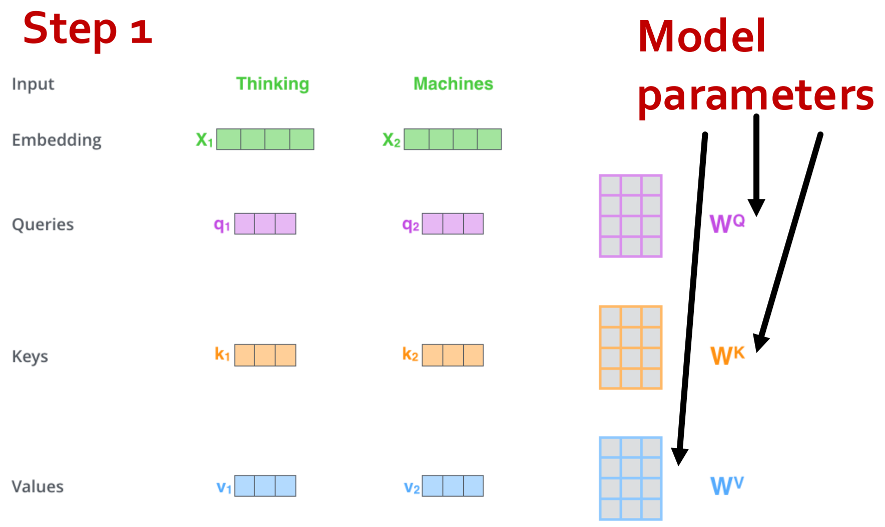
- **Before "multi-head" self-attention, what is "single head" self-attention?**



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

Self-attention

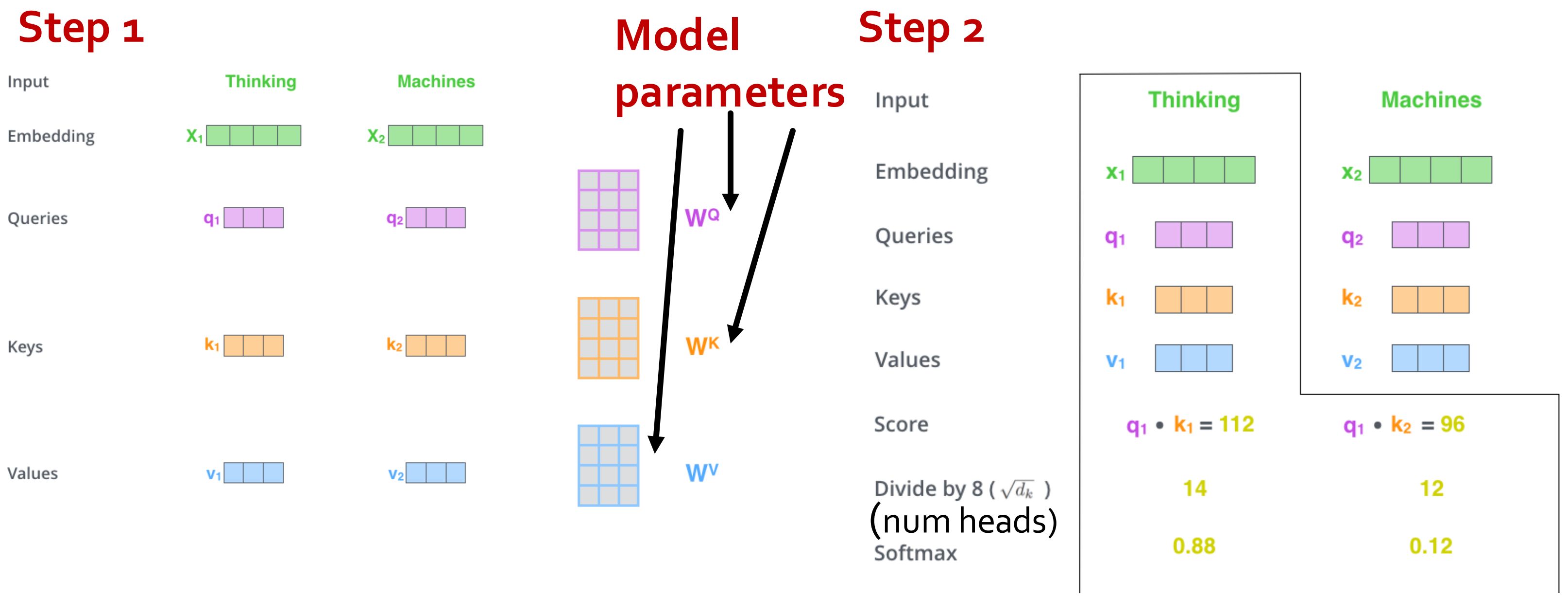
- **Step 1:** compute "key, value, query" for each input



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

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)

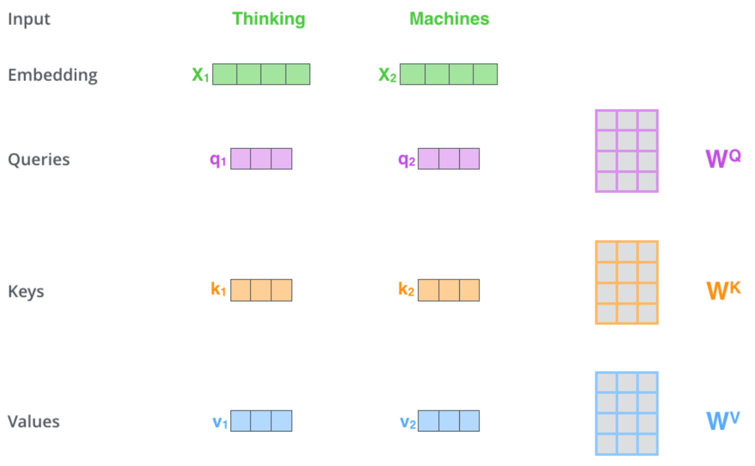


See: Illustrated Transformer tutorial, https://jalammar.github.io/illustrated-transformer/

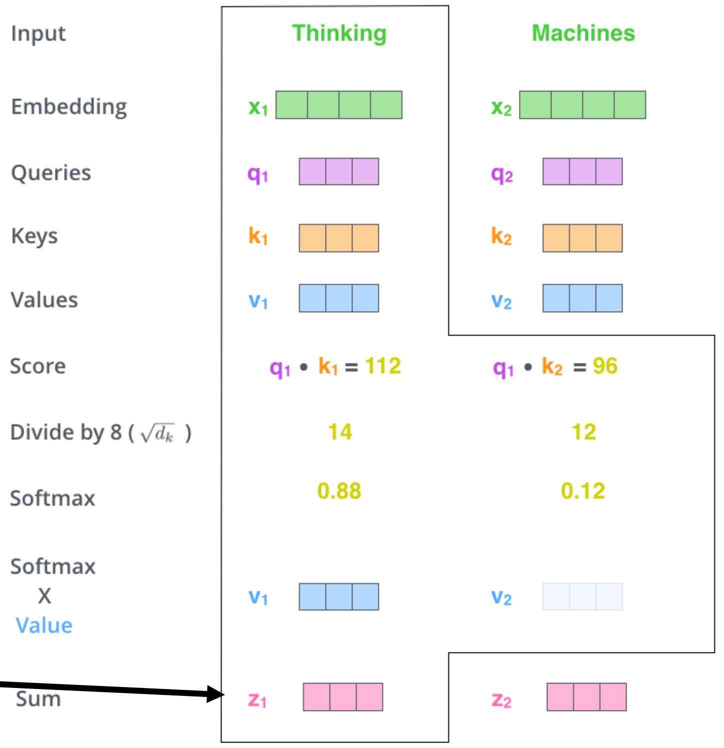
Self-attention

- **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 1



Step 2



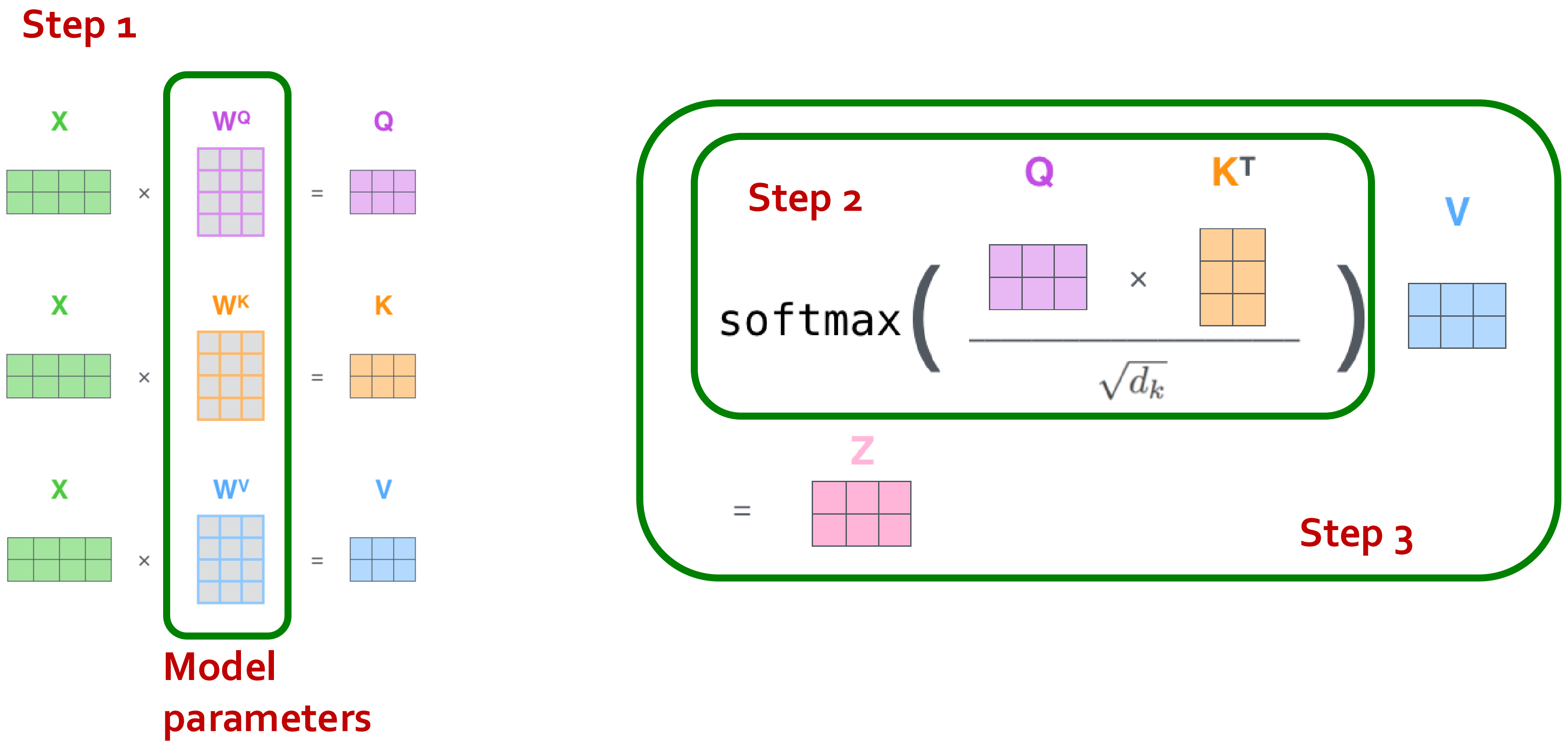
Step 3

$$z_1 = 0.88v_1 + 0.12v_2$$

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

Self-attention

- **Same calculation in matrix form**



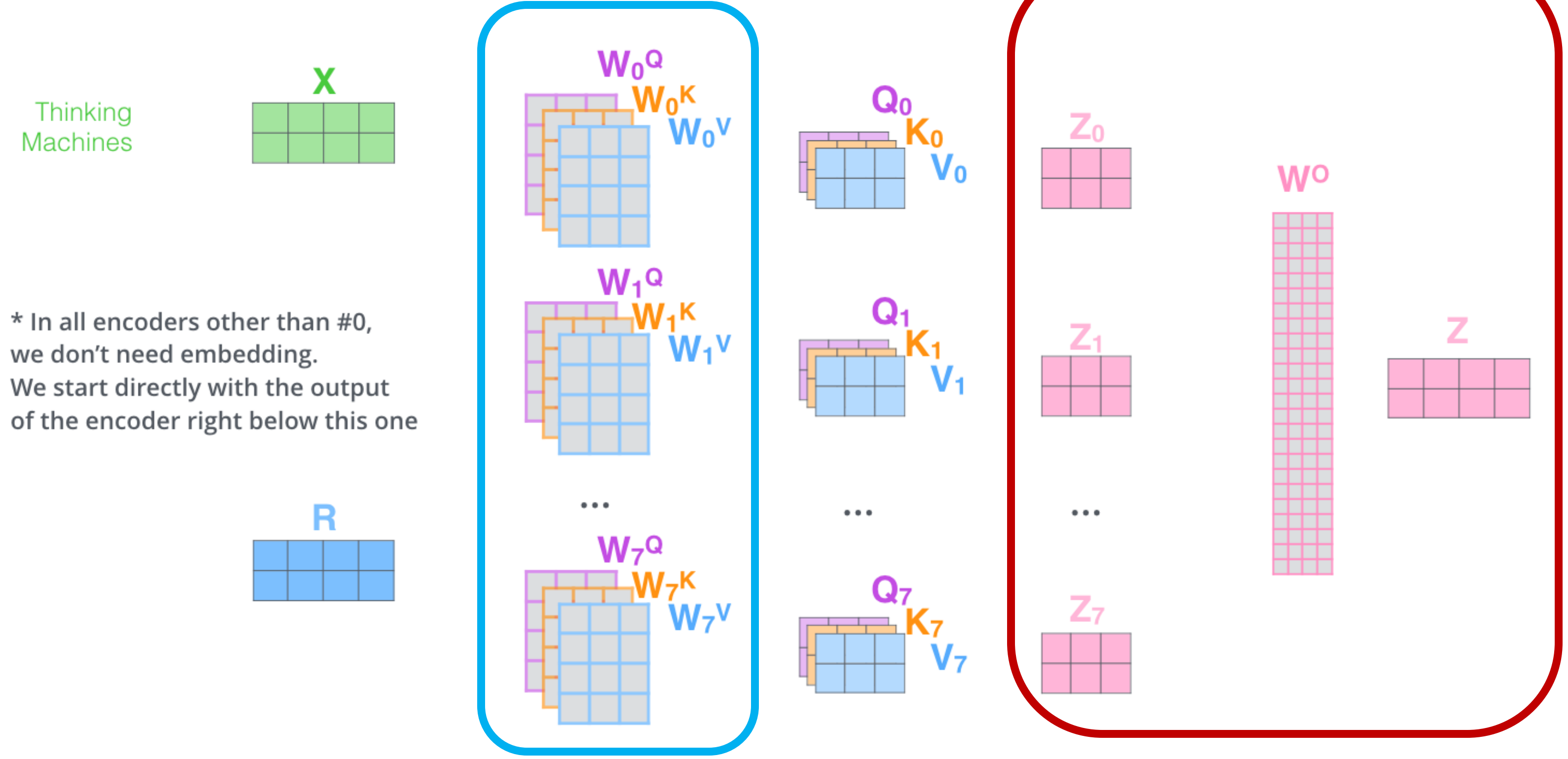
$$X \times W^Q = Q, \quad X \times W^K = K, \quad X \times W^V = V$$

$$\text{softmax}\left(\frac{Q \times K^T}{\sqrt{d_k}}\right) V = Z$$

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

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

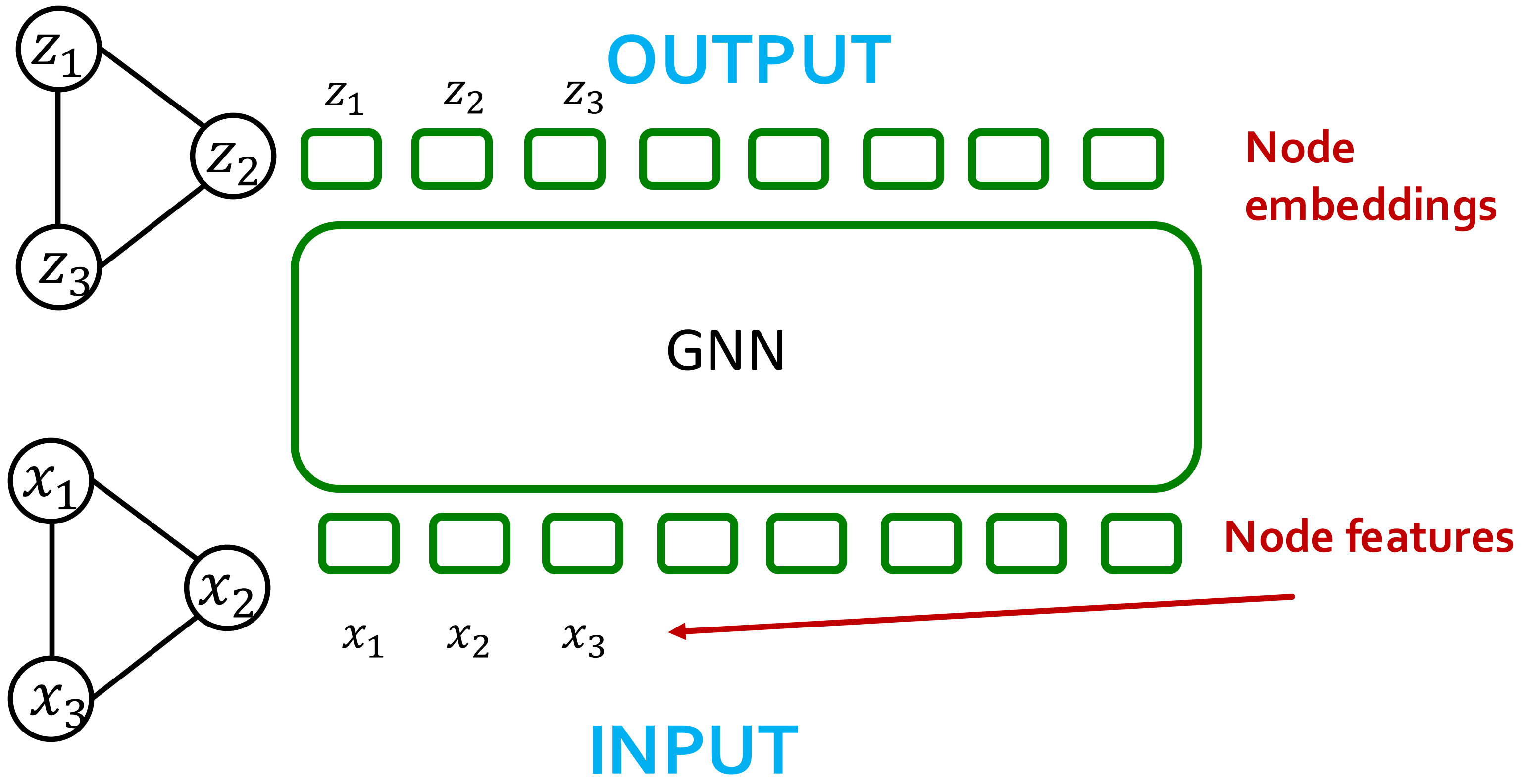


* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one

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

Comparing Transformers and GNN

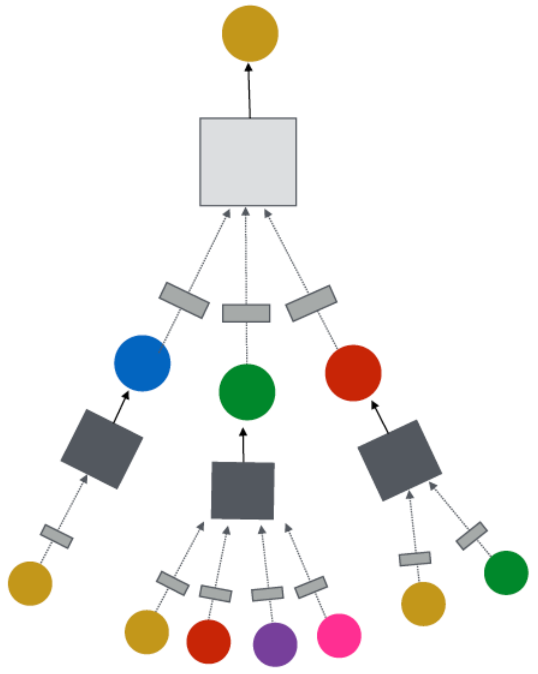
- **Similarity:** GNNs also take in a sequence of vectors (in no particular order) and output a sequence of embeddings
- **Difference:** GNNs use **message passing**, Transformer uses **self-attention**

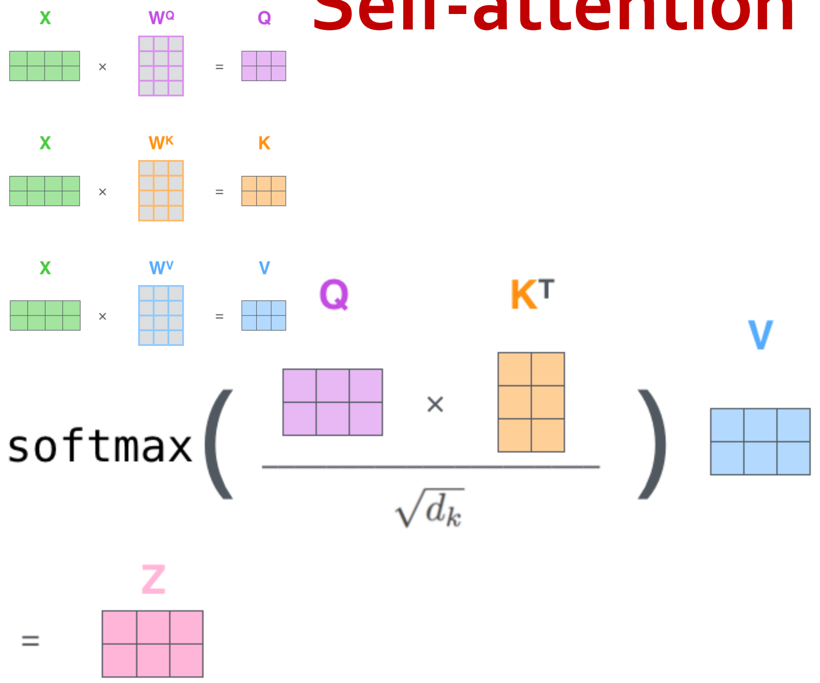


Comparing Transformers and GNN

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

Message Passing Vs. **Self-attention**





$$\text{softmax}\left(\frac{Q \times K^T}{\sqrt{d_k}}\right) V = Z$$

Stanford CS224W: Self-attention vs. message passing

CS224W: Machine Learning with Graphs
Jure Leskovec, Stanford University
http://cs224w.stanford.edu

Interpreting the Self-Attention Update

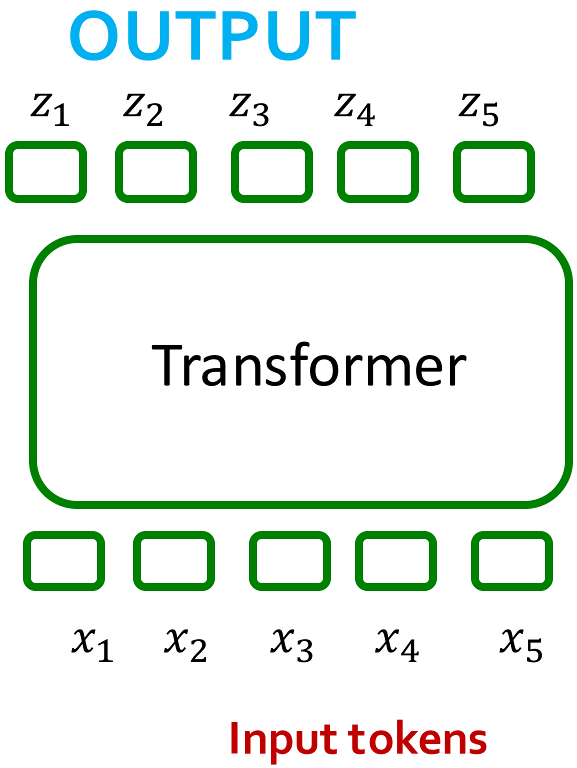
- Recall Formula for attention update:

$$Att(X) = softmax(QK^T)V$$

$$Q = XW^Q, K = XW^K, V = XW^V$$

Inputs stored row-wise

$$X = \left[— x_i — \right]$$



Interpreting the Self-Attention Update

- Recall Formula for attention update:

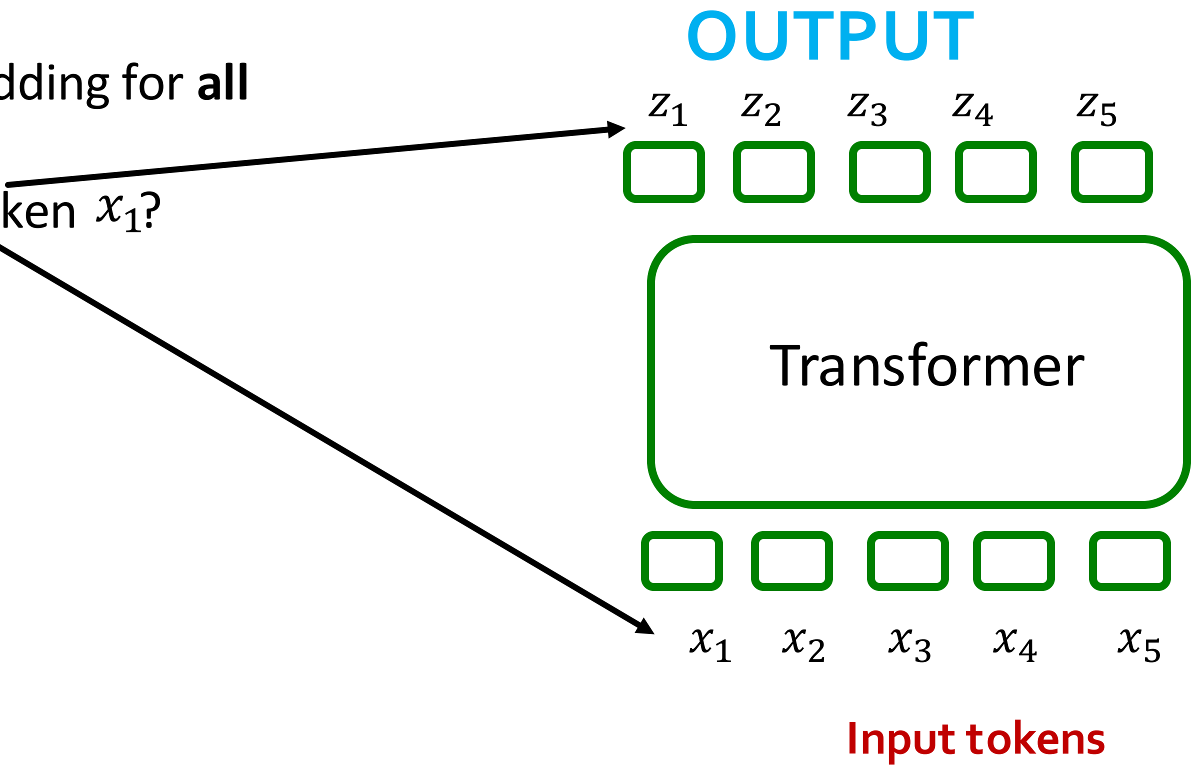
$$Att(X) = softmax(QK^T)V$$

$$Q = XW^Q, K = XW^K, V = XW^V$$

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

Inputs stored row-wise

$$X = \begin{bmatrix} - x_i - \end{bmatrix}$$



Interpreting the Self-Attention Update

- Recall Formula for attention update:

$$Att(X) = softmax(QK^T)V$$

$$Q = XW^Q, K = XW^K, V = XW^V$$

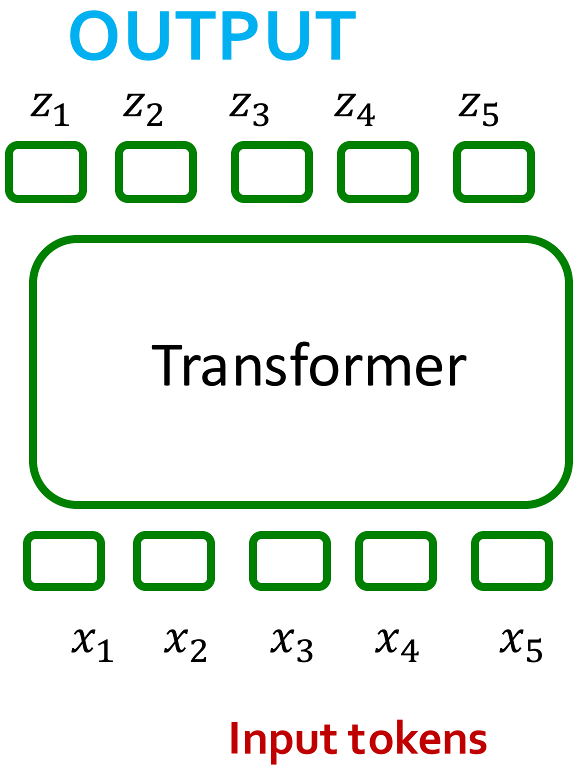
Inputs stored row-wise

$$X = \begin{bmatrix} — x_i — \end{bmatrix}$$

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

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

How to interpret this?



Interpreting the Self-Attention Update

$$Att(X) = softmax(QK^T)V$$

$$Q = XW^Q, K = XW^K, V = XW^V$$

Inputs stored row-wise

$$X = \begin{bmatrix} — x_i — \end{bmatrix}$$

- This formula gives the embedding for **all tokens** simultaneously
- If we simplify to just token x_1 what does the update look like?

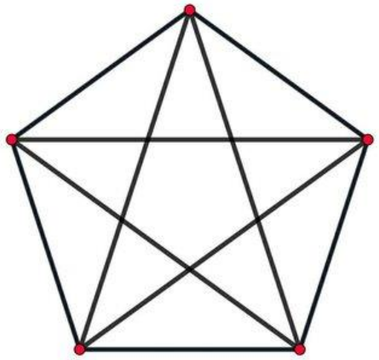
$$z_1 = \sum_{j=1}^{5} softmax_j(q_1^T k_j) v_j$$

How to interpret this?

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

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**

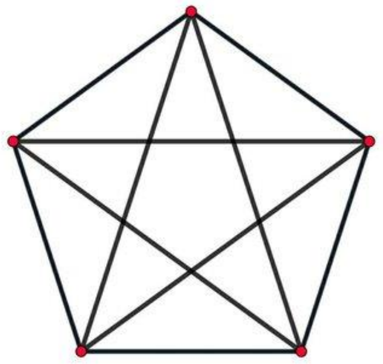


$$z_1 = \sum_{j=1}^{5} softmax_j(q_1^T k_j) v_j$$

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

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:** $(v_j,\ k_j) = MSG(x_j) = (W^V x_j, W^K x_j)$
  - **2. Compute query for 1:** $q_1 = MSG(x_1) = W^Q x_1$
  - **3. Aggregate all messages:** $Agg(q_1, \{MSG(x_j) : j\}) = \sum_{j=1}^{n} softmax_j(q_1^T k_j) v_j$

Plan for Today

- **Part 1:**
  - Introducing Transformers
  - Relation to message passing GNNs
- **Part 2:**
  - A new design landscape for graph Transformers
- **Part 3 (time permitting):**
  - PEARL: Learning Efficient Positional Encodings with GNNs

Stanford CS224W: A New Design Landscape for Graph Transformers

CS224W: Machine Learning with Graphs

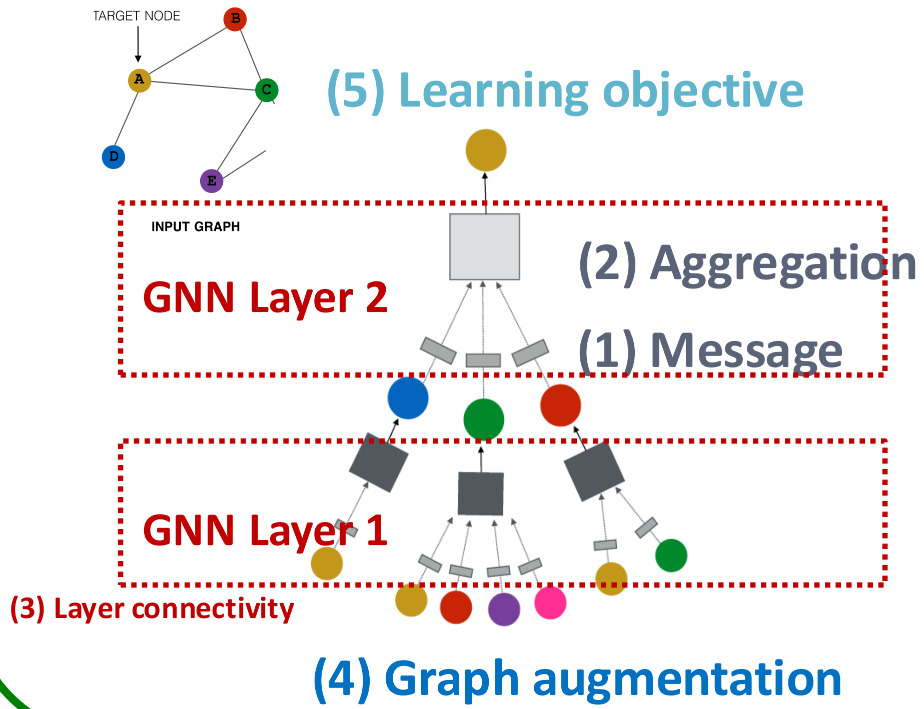
Jure Leskovec, Stanford University

http://cs224w.stanford.edu

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?**

GNN design space

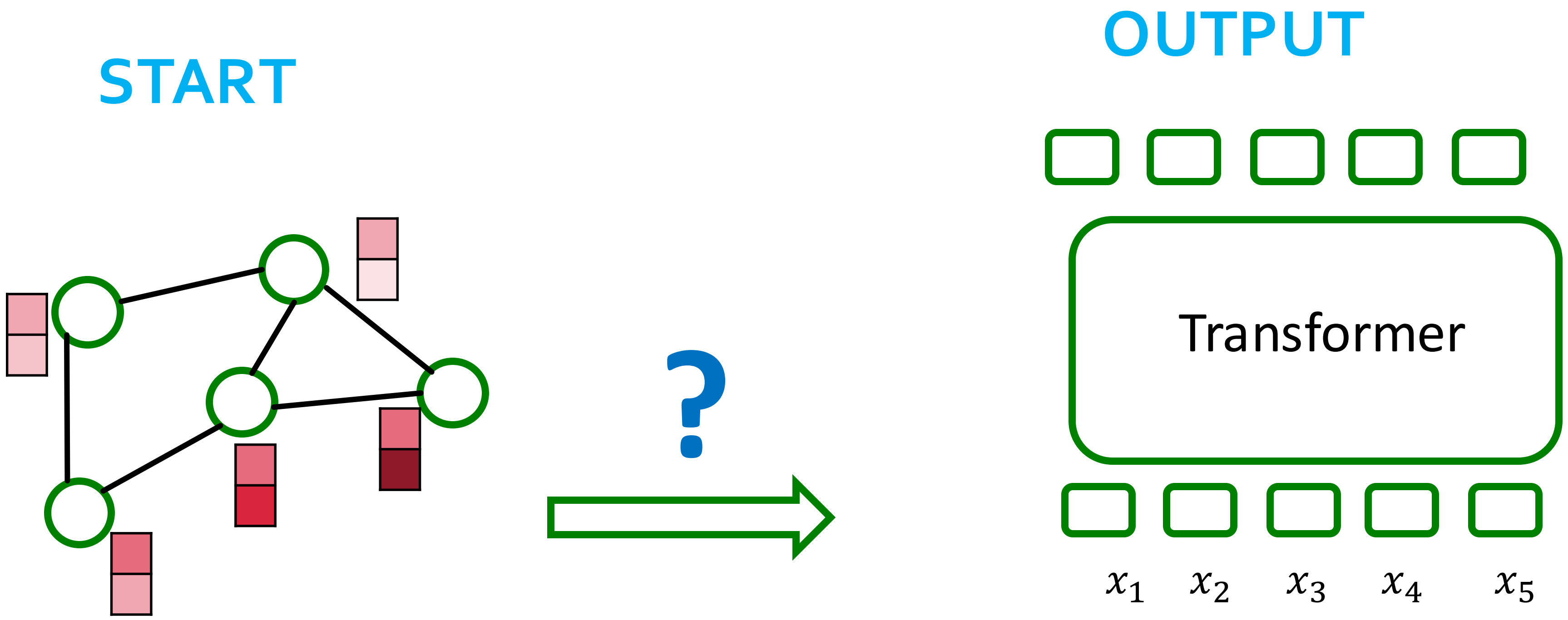


Graph Transformer design space

?

Processing Graphs with Transformers

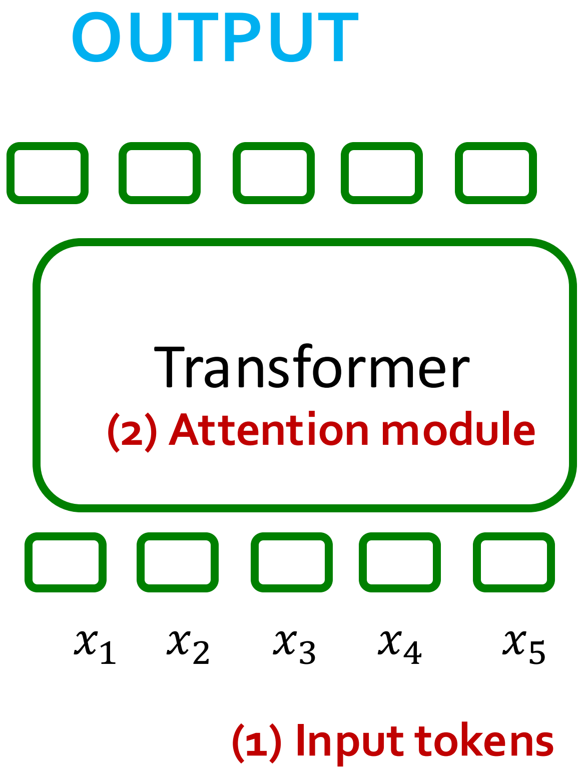
- We start with graph(s)
- How to input a graph into a Transformer?



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

Components of a Transformer

- To understand how to process graphs with Transformers we must:
  - Understand the key components of the Transformer. Seen already:
    - 1) tokenizing,
    - 2) self-attention
  - Decide how to make suitable **graph versions** of each



A final key piece: token ordering

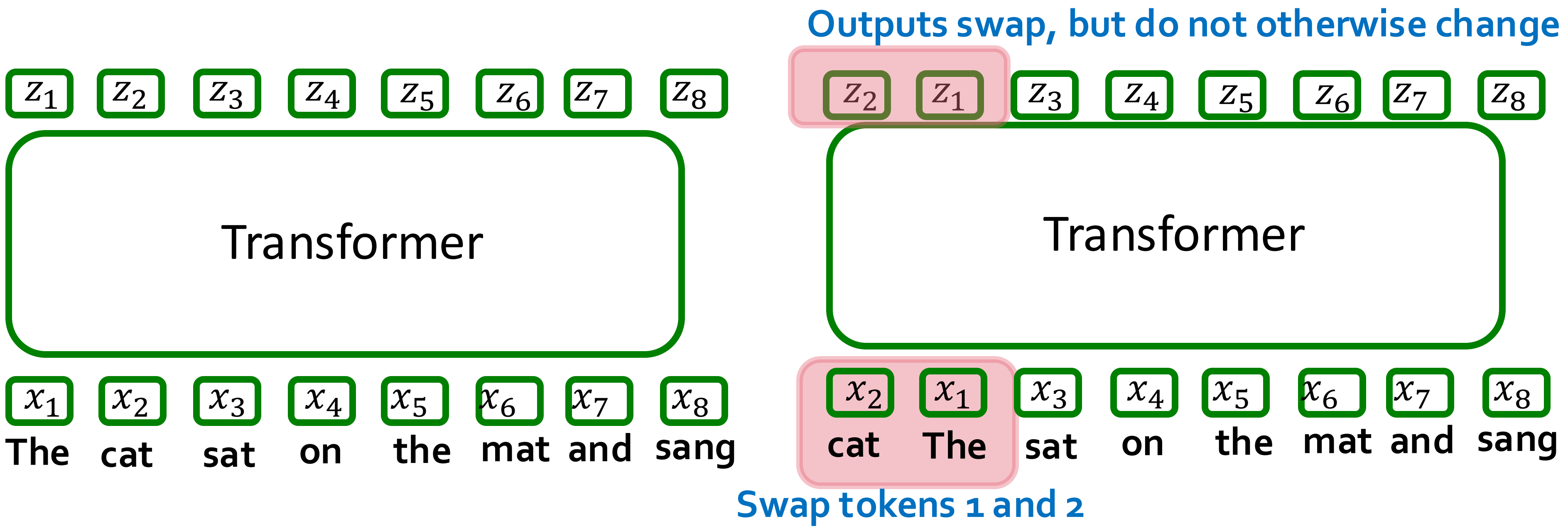
- There is one other key missing piece we have not yet discussed…

A final key piece: token ordering

- There is one other key missing piece we have not yet discussed …
- **First recall update formula** $$z_1 = \sum_{j=1}^{5} softmax_j\left(q_1^T k_j\right) v_j$$
- **Key Observation: order of tokens does not matter!!!**

A final key piece: token ordering

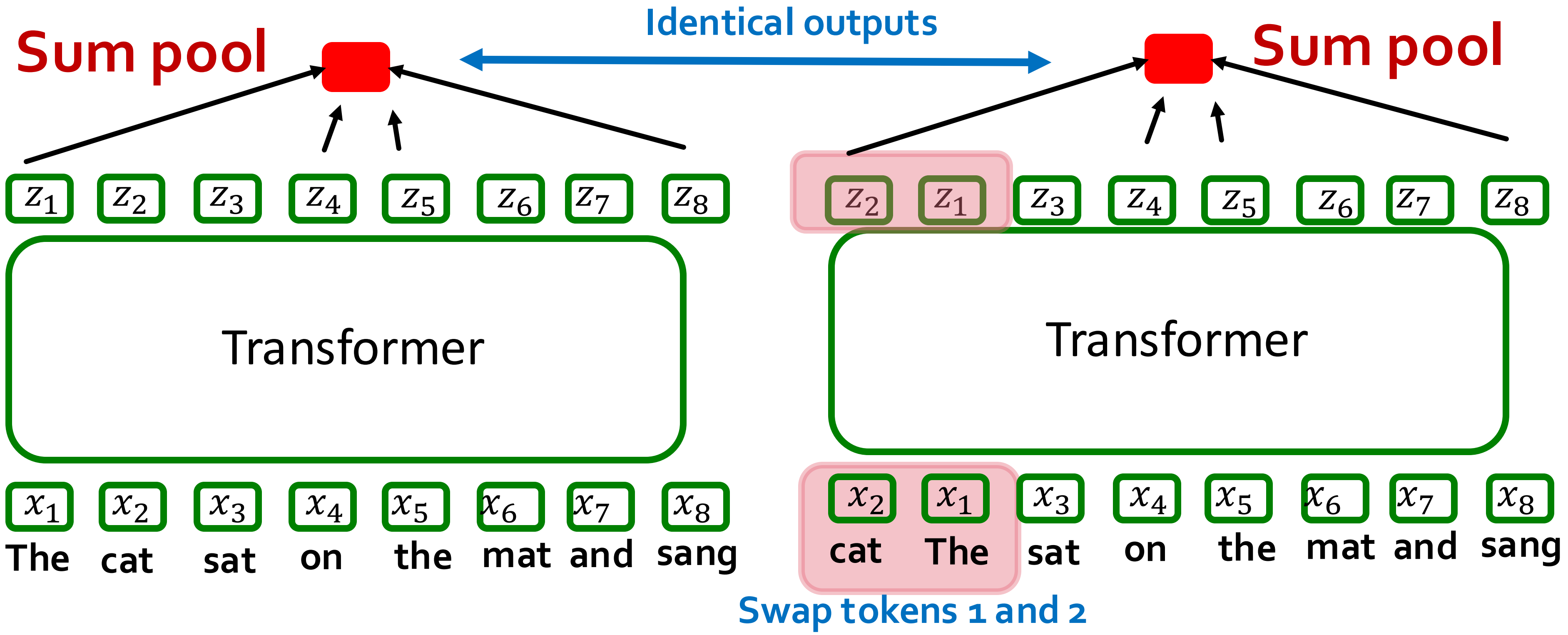
- There is one other key missing piece we have not yet discussed …
- **First recall update formula** $$z_1 = \sum_{j=1}^{5} softmax_j(q_1^T k_j) v_j$$
- **Key Observation: order of tokens does not matter!!!**



Outputs swap, but do not otherwise change
z_1 z_2 z_3 z_4 z_5 z_6 z_7 z_8
Transformer
x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8
The cat sat on the mat and sang
z_2 z_1 z_3 z_4 z_5 z_6 z_7 z_8
Transformer
x_2 x_1 x_3 x_4 x_5 x_6 x_7 x_8
cat The sat on the mat and sang
Swap tokens 1 and 2

A final key piece: Token ordering

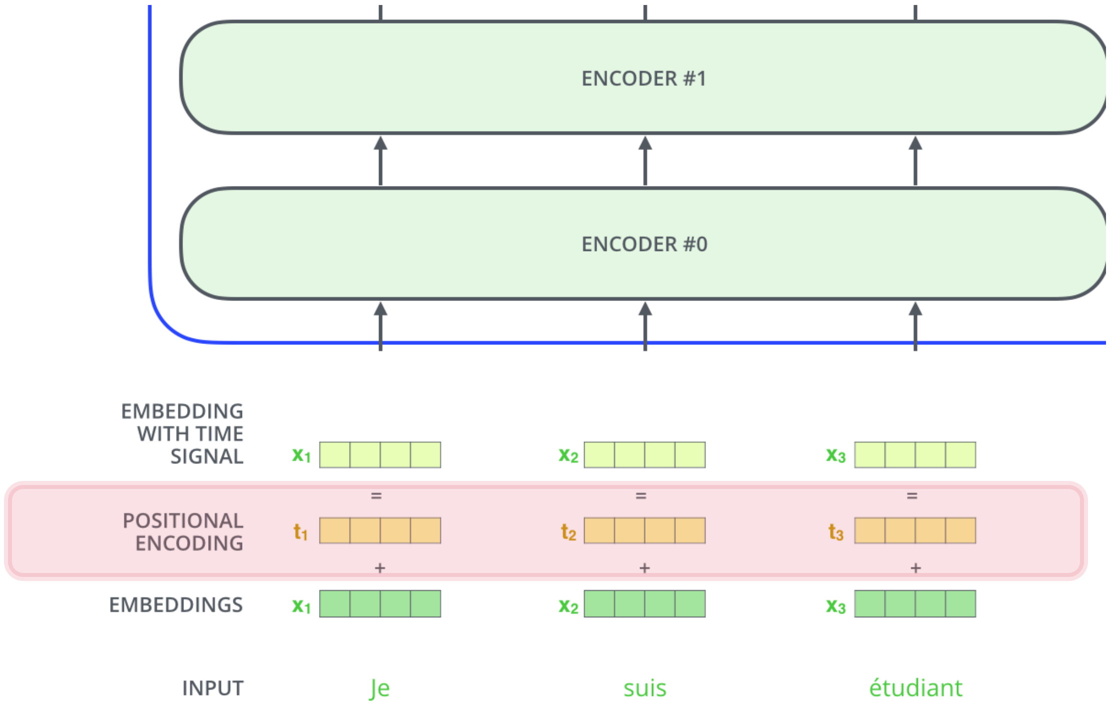
- **This is a problem**
- **Same predictions** no matter what order the words are in! (A "bag of words" prediction model)...
  - How to fix?



Sum pool
Identical outputs
Sum pool
z_1 z_2 z_3 z_4 z_5 z_6 z_7 z_8
Transformer
x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8
The cat sat on the mat and sang
z_2 z_1 z_3 z_4 z_5 z_6 z_7 z_8
Transformer
x_2 x_1 x_3 x_4 x_5 x_6 x_7 x_8
cat The sat on the mat and sang
Swap tokens 1 and 2

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



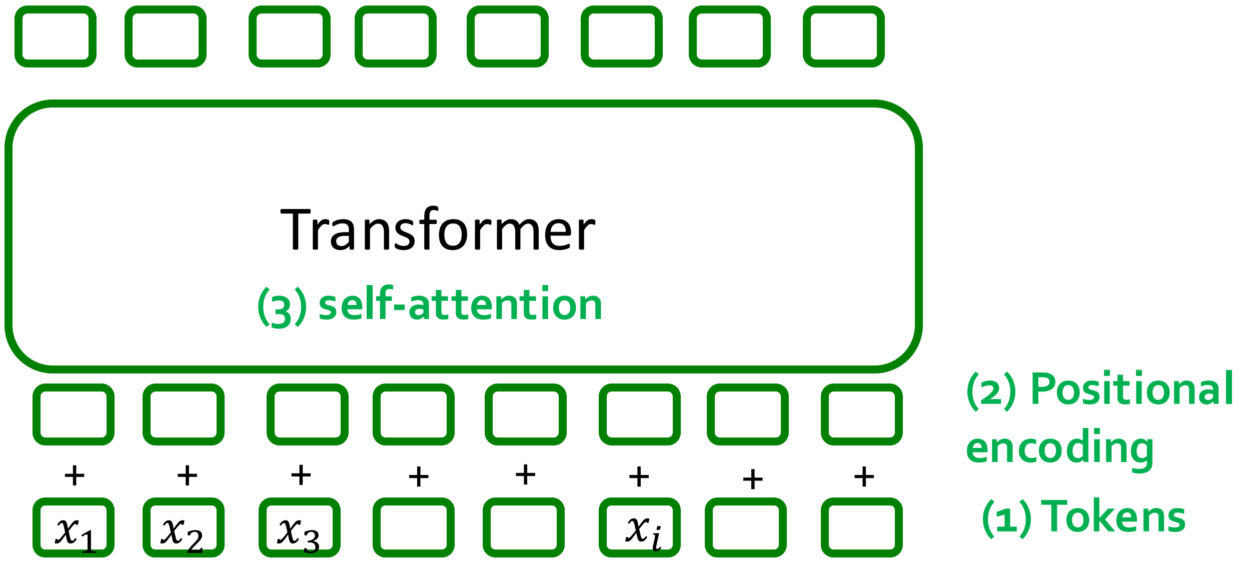
J. You, R. Ying, J. Leskovec. [Design Space of Graph Neural Networks](), NeurIPS 2020

Components of a Transformer

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

How to chose these for graph data?

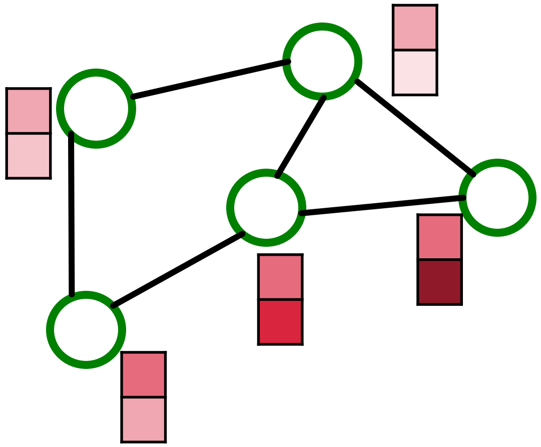
- **Key question:** What should these be for a graph input?



Processing Graphs with Transformers

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

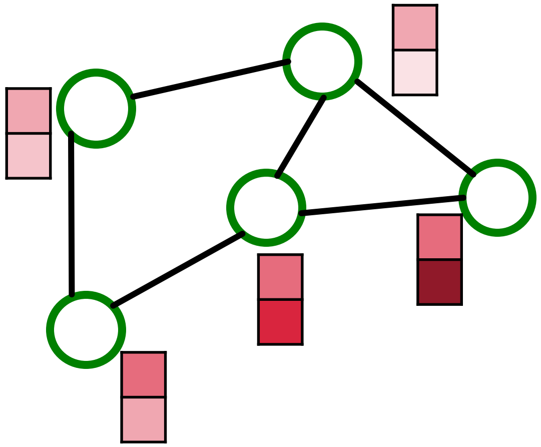
- Key components of Transformer
  - **(1) tokenizing**
  - **(2) positional encoding**
  - **(3) self-attention**



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**



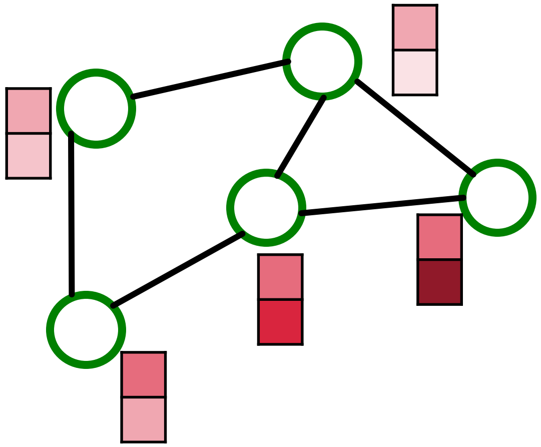
- There are many ways to do this
- Different approaches correspond to different “matchings” between graph inputs **(1), (2), (3)** transformer components **(1), (2), (3)**

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**

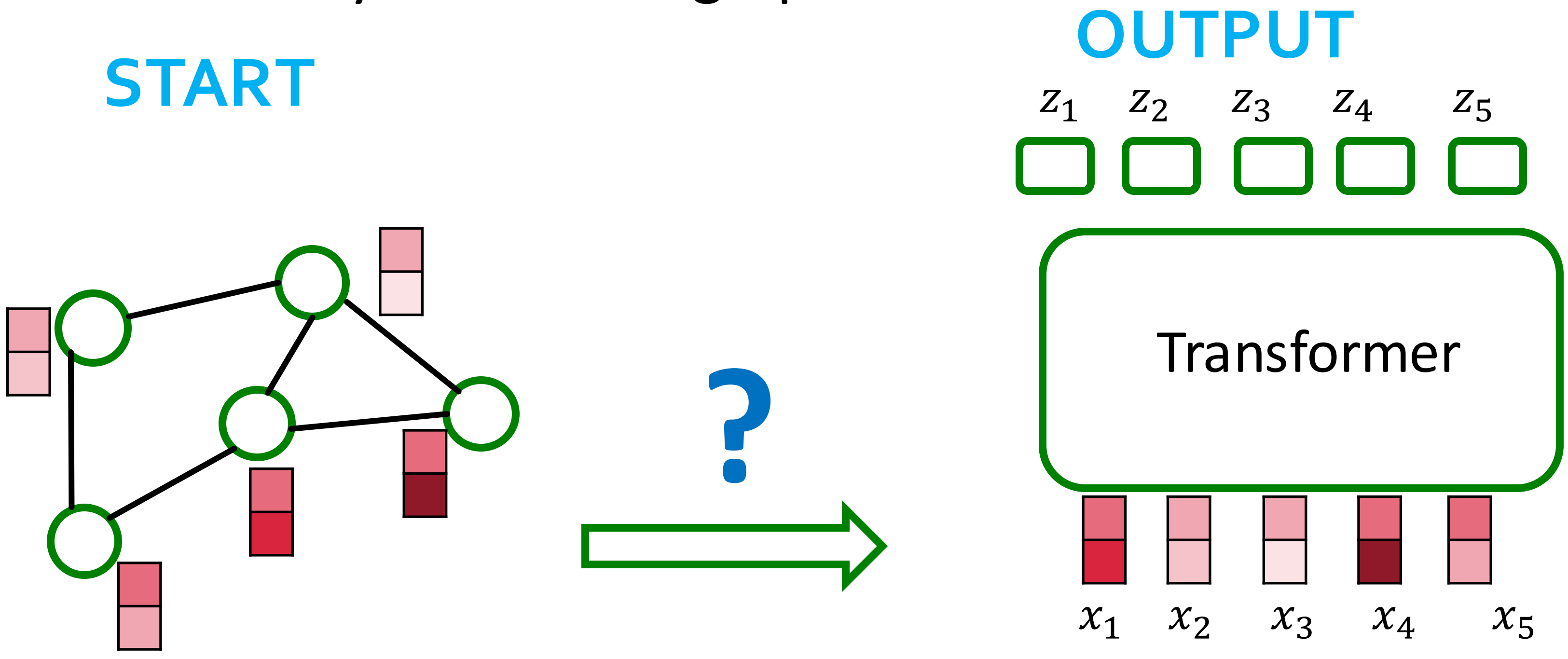
Today



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

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

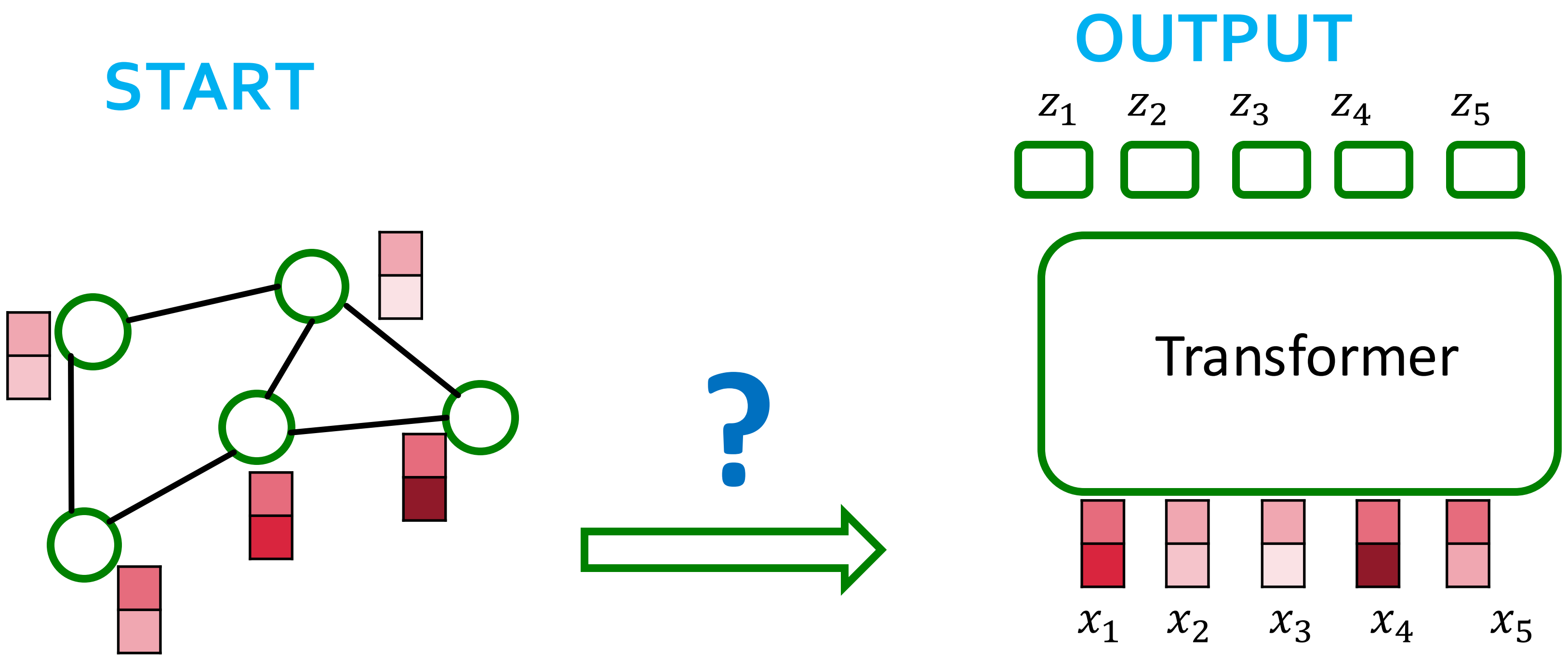


(1) Input tokens = Node features

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

Processing Graphs with Transformers

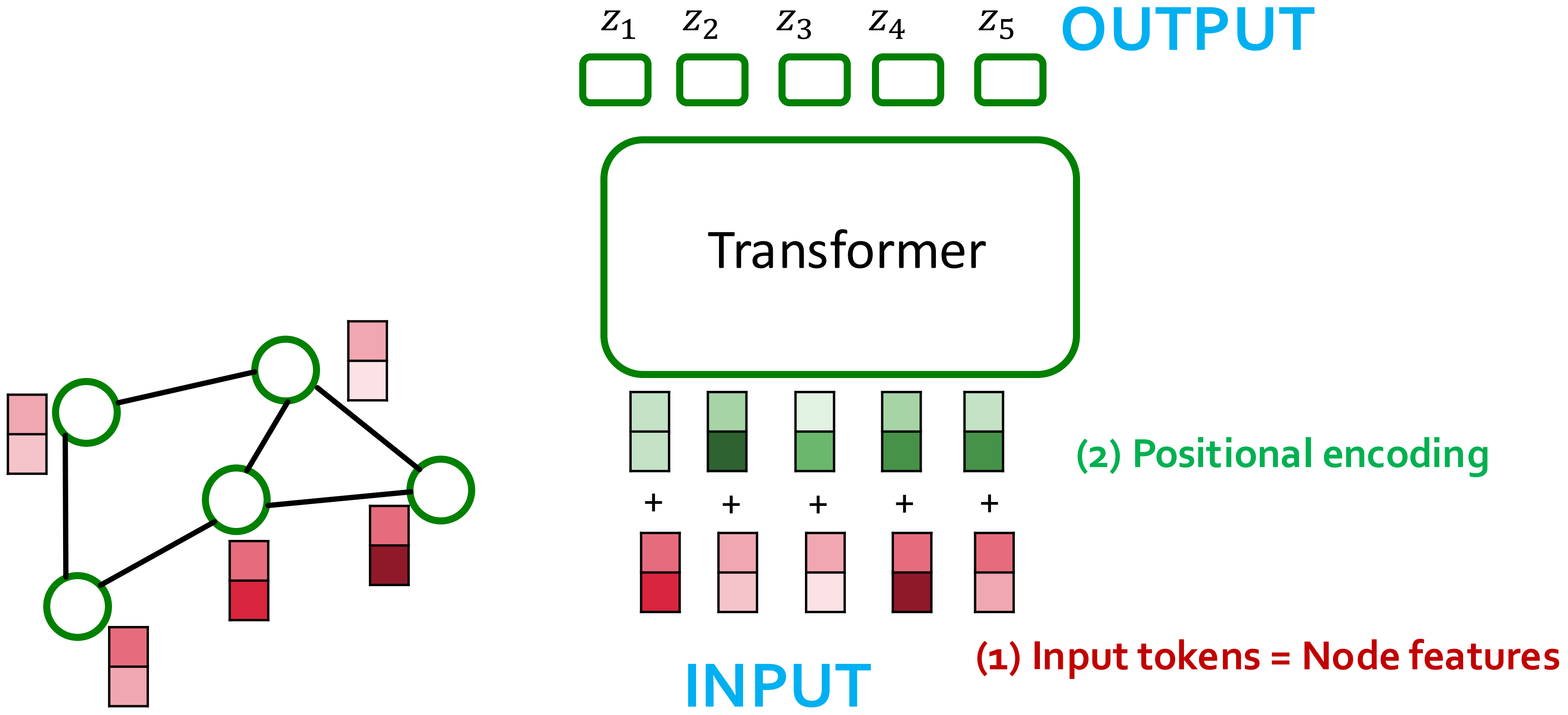
- **Problem? We completely lose adjacency info!**
- **How to also inject adjacency information?**



(1) Input tokens = Node features

How to Add Back Adjacency Info?

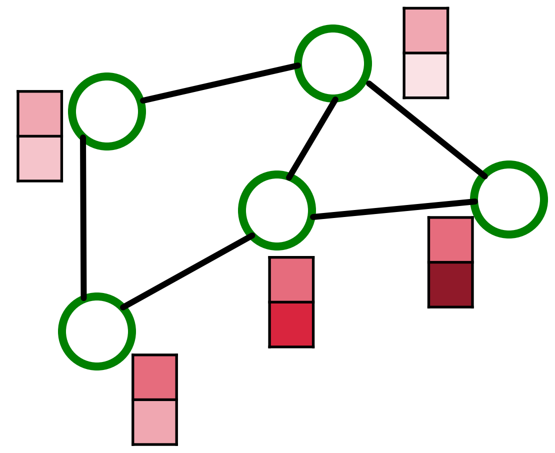
- **Idea:** Encode adjacency info in the **positional encoding** for each node
- Positional encoding describes **where** a node is in the graph

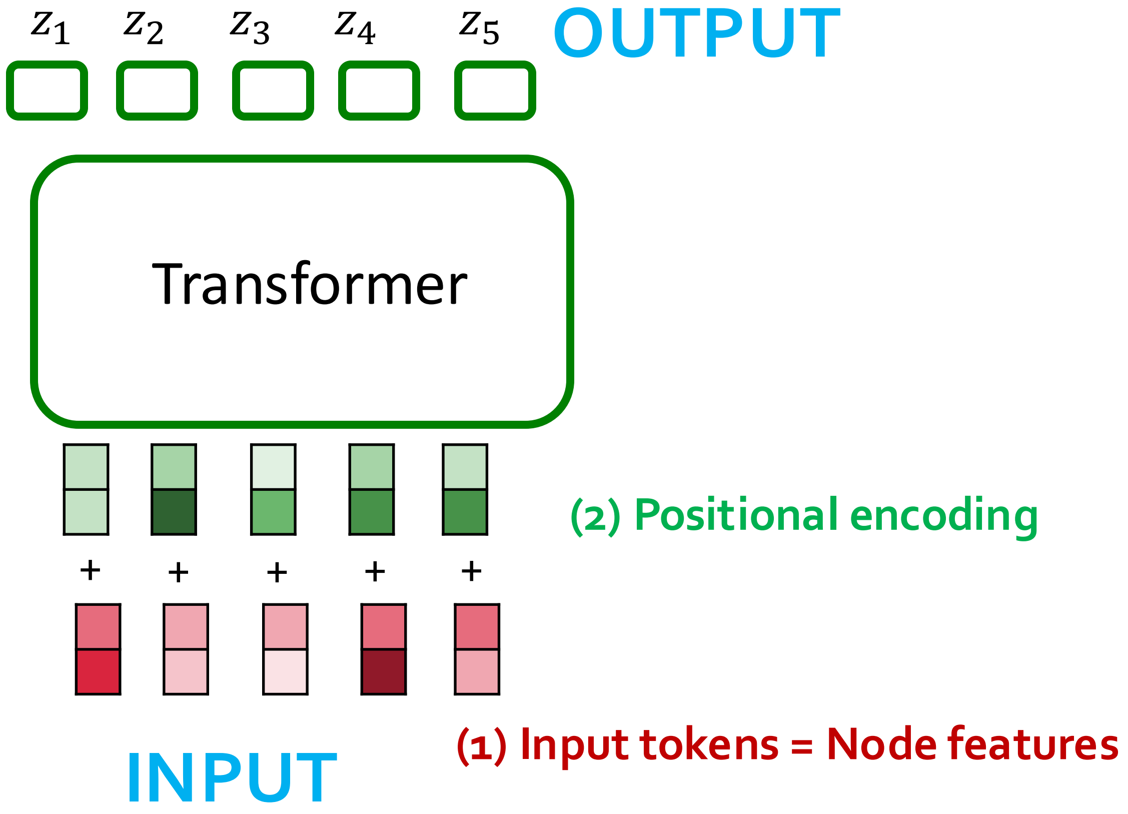


How to Add Back Adjacency Info?

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

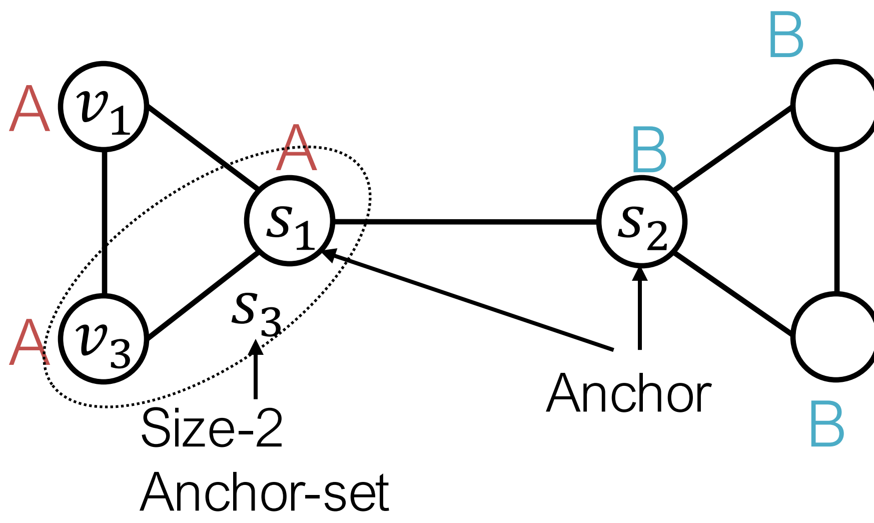
Q2: How to design a good positional encoding?





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

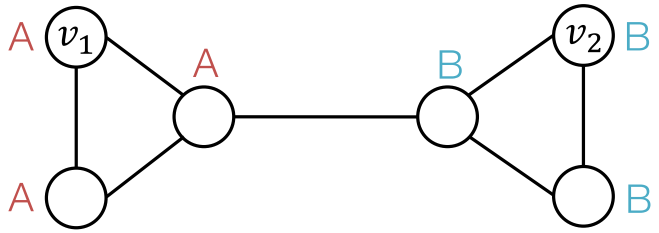
Positional encoding for node v_1 🟩 =

	s_1	s_2	s_3
v_1	1	2	1
v_3	1	2	0

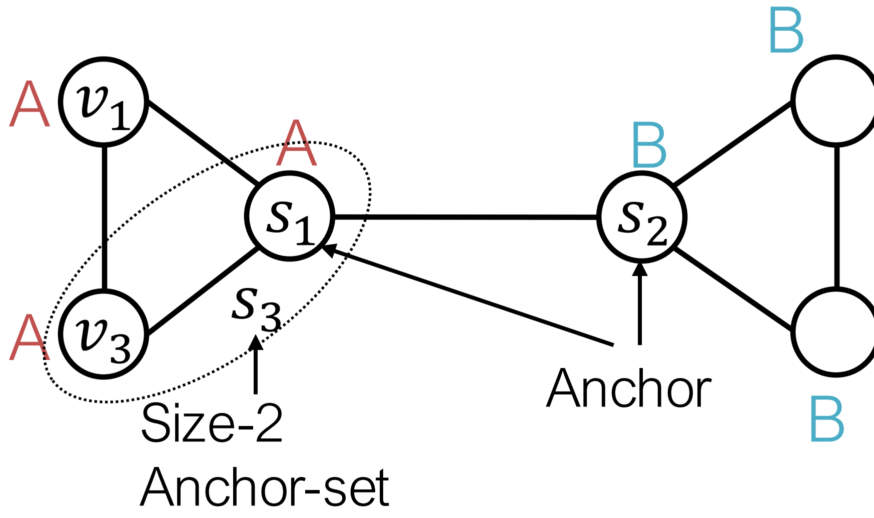
Anchor s_1, s_2 cannot differentiate node v_1, v_3, but anchor-set s_3 can

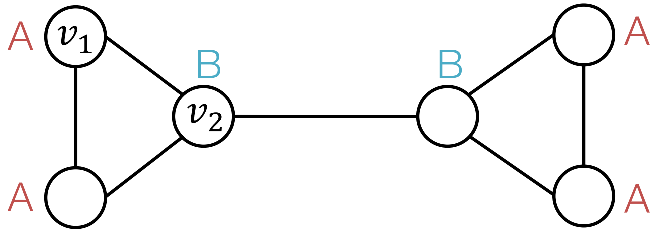
Option 1: Relative distances

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



- But not suited to structure-aware tasks





Positional encoding for node v_1 🟩 =

Relative Distances

	s_1	s_2	s_3
v_1	1	2	1
v_3	1	2	0

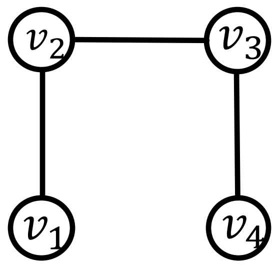
Anchor s_1, s_2 cannot differentiate node v_1, v_3, but anchor-set s_3 can

Option 2: Laplacian Eigenvector Positional Encodings

- What other ways to make positional encoding?

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



$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} - \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Degree of each node **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 = global structure, large eigenvalue = local symmetries

Refresher

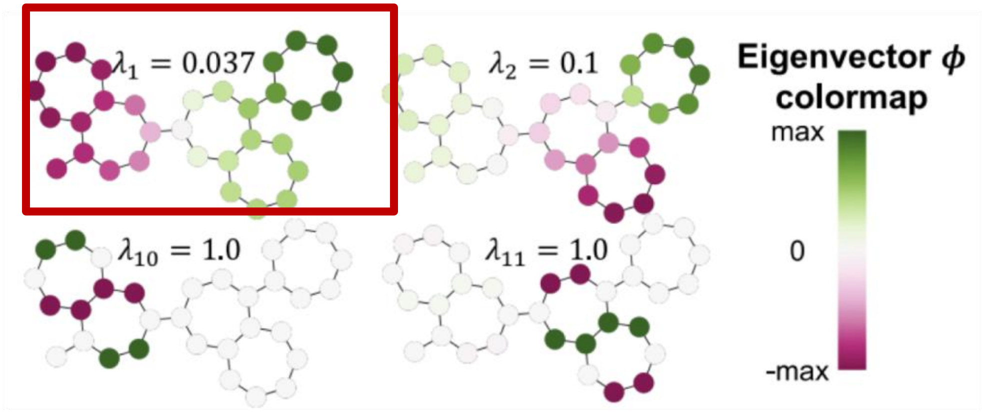
Eigenvector: v such that $Lv = \lambda v$

L: $n \times n$ matrix

v: n dimensional vector

λ: Scalar eigenvalue

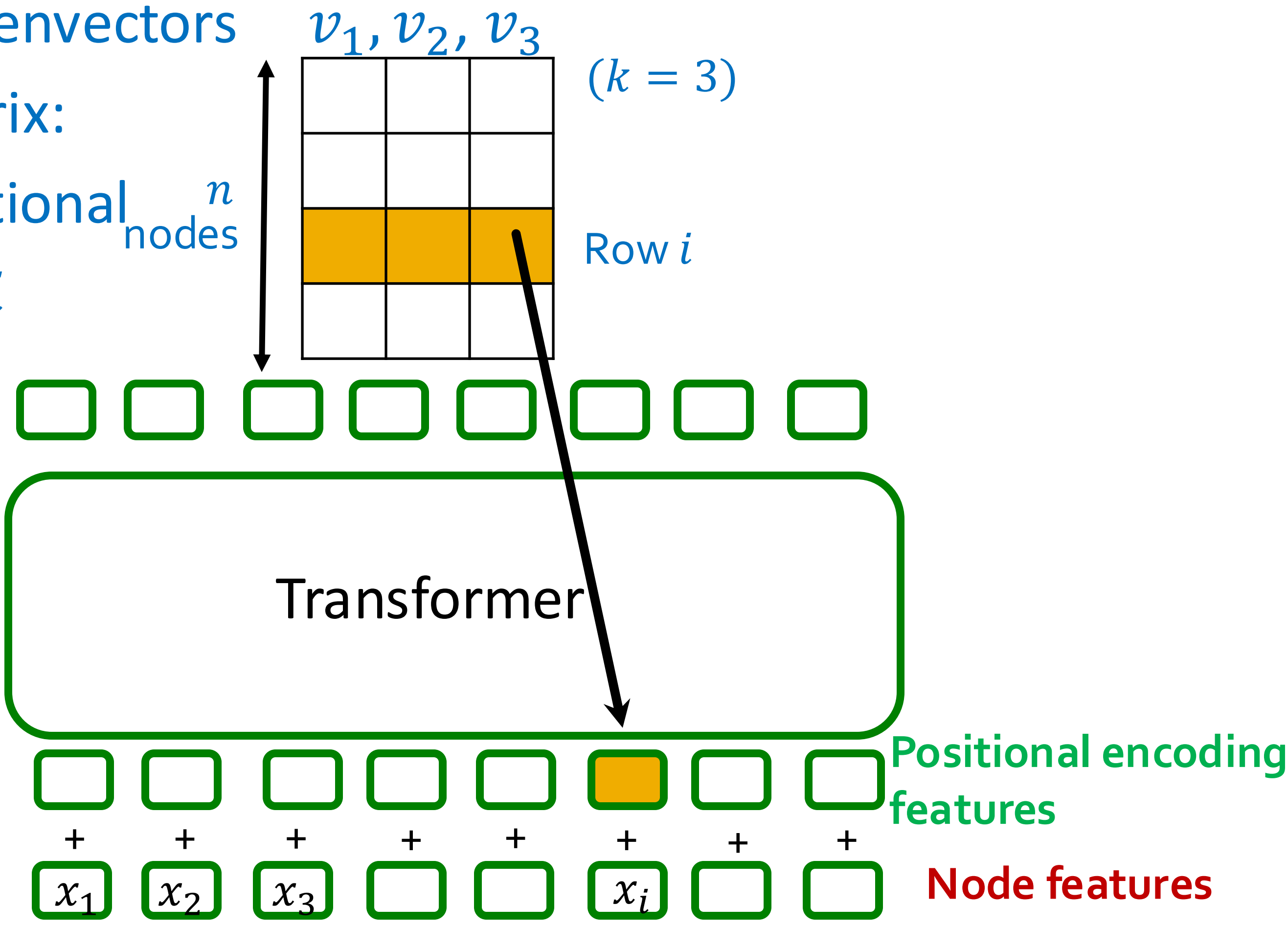
Visualize one eigenvector

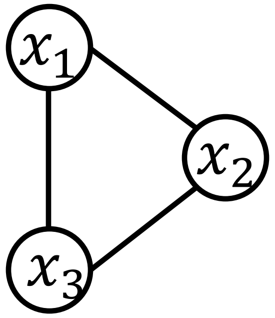


(Figure from Kreuzer* and Beaini* et al. 2021)

Laplacian Eigenvector Positional Encodings

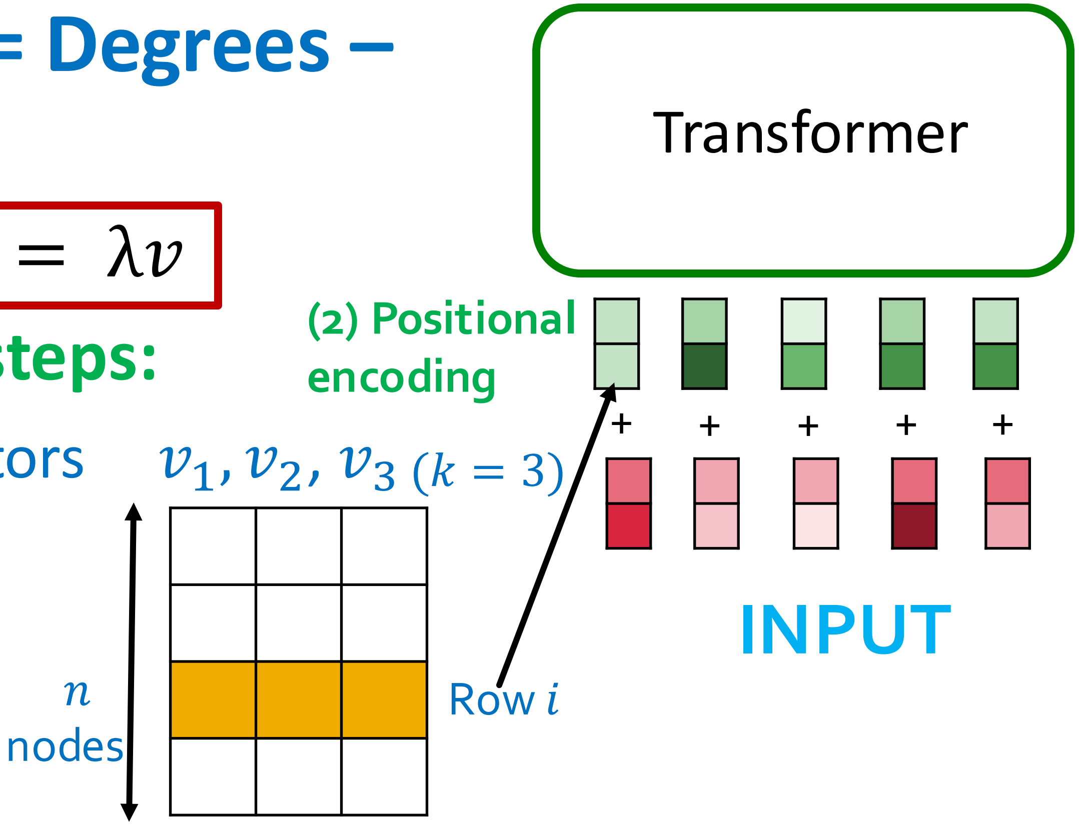
- **Positional encoding steps:**
  - 1. compute k eigenvectors
  - 2. Stack into matrix:
  - 3. ith row is positional encoding for node i





Summary: Laplacian Eigenvector Positional Encodings

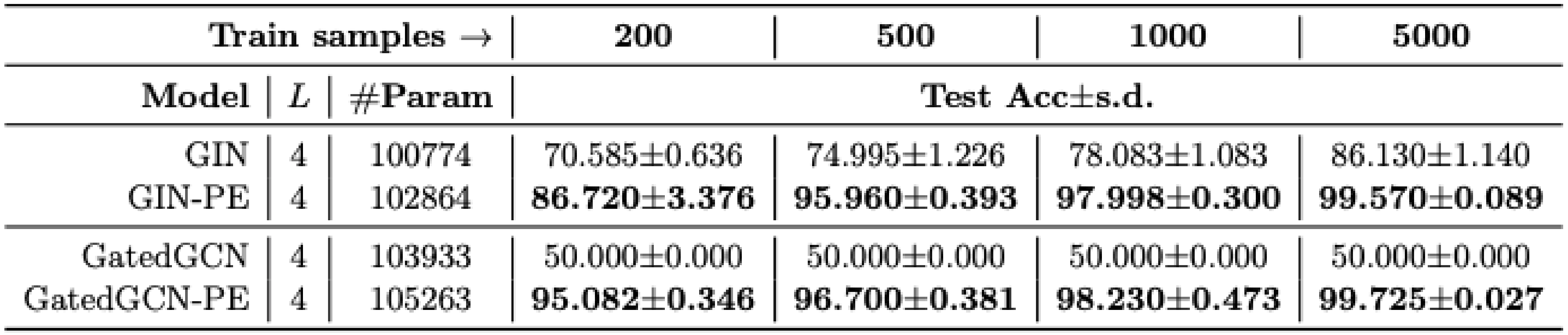
- Laplacian Matrix **L = Degrees – Adjacency**
- Eigenvector: v such that $Lv = \lambda v$
- **Positional encoding steps:**
  - 1. compute k eigenvectors
  - 2. Stack into matrix:
  - 3. ith row is positional encoding for node i



- Laplacian Eigenvector positional encodings can also be used with message-passing GNNs
  - **This helps for same reasons as structural and 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
- "PE" indicates using Laplacian Eigenvector Pos. Enc.



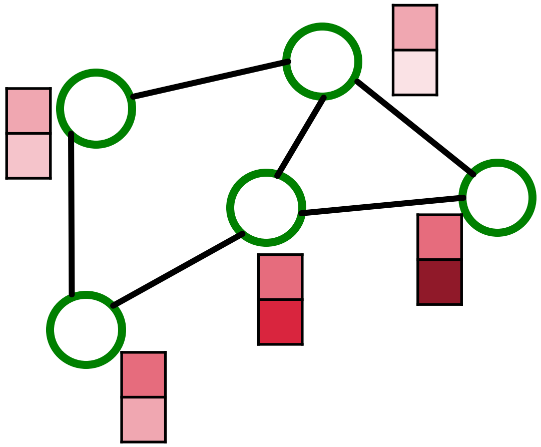
		Train samples →	200	500	1000	5000
Model	L	#Param	Test Acc±s.d.			
GIN	4	100774	70.585±0.636	74.995±1.226	78.083±1.083	86.130±1.140
GIN-PE	4	102864	**86.720±3.376**	**95.960±0.393**	**97.998±0.300**	**99.570±0.089**
GatedGCN	4	103933	50.000±0.000	50.000±0.000	50.000±0.000	50.000±0.000
GatedGCN-PE	4	105263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

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



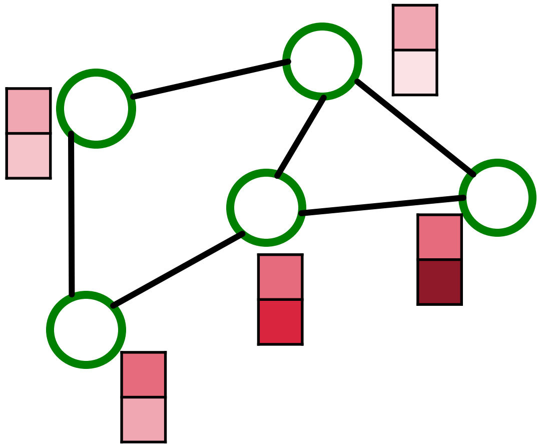
- There are many ways to do this
- Different approaches correspond to different "matchings" between graph inputs **(1), (2), (3)** transformer components **(1), (2), (3)**

Processing Graphs with Transformers

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

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

Left to do



- 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? $Att(X) = softmax(QK^T)V$
- $[a_{ij}] = QK^T$ is an n x n matrix. Entry a_{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? $Att(X) = softmax(QK^T)V$
- $[a_{ij}] = QK^T$ is an n x n matrix. Entry a_{ij} describes "how much" token j contributes to the update of token i
- **Idea: adjust $\boldsymbol{a_{ij}}$ based on edge features. Replace with $\boldsymbol{a_{ij} + c_{ij}}$ 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? $Att(X) = softmax(QK^T)V$
- $[a_{ij}] = QK^T$ is an n x n matrix. Entry a_{ij} describes "how much" token j contributes to the update of token i
- **Idea: adjust a_{ij} based on edge features. Replace with $a_{ij} + c_{ij}$ where c_{ij} depends on the edge features**
- **Implementation:**

  Learned parameters w_1

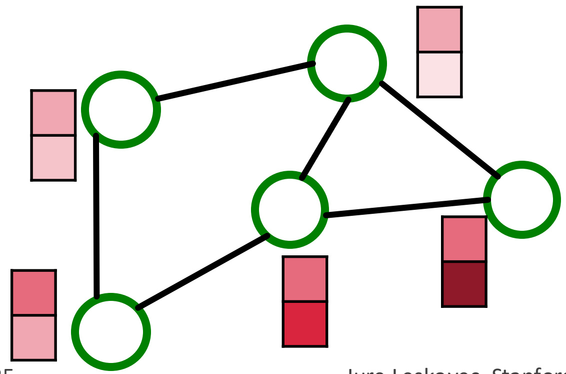
  - **If there is an edge between i and j with features e_{ij}, define $c_{ij} = w_1^T e_{ij}$**
  - **If there is no edge, find shortest edge path between i and j $(e^1, e^2, \dots e^N)$ and define $c_{ij} = \sum_n w_n^T e^n$**

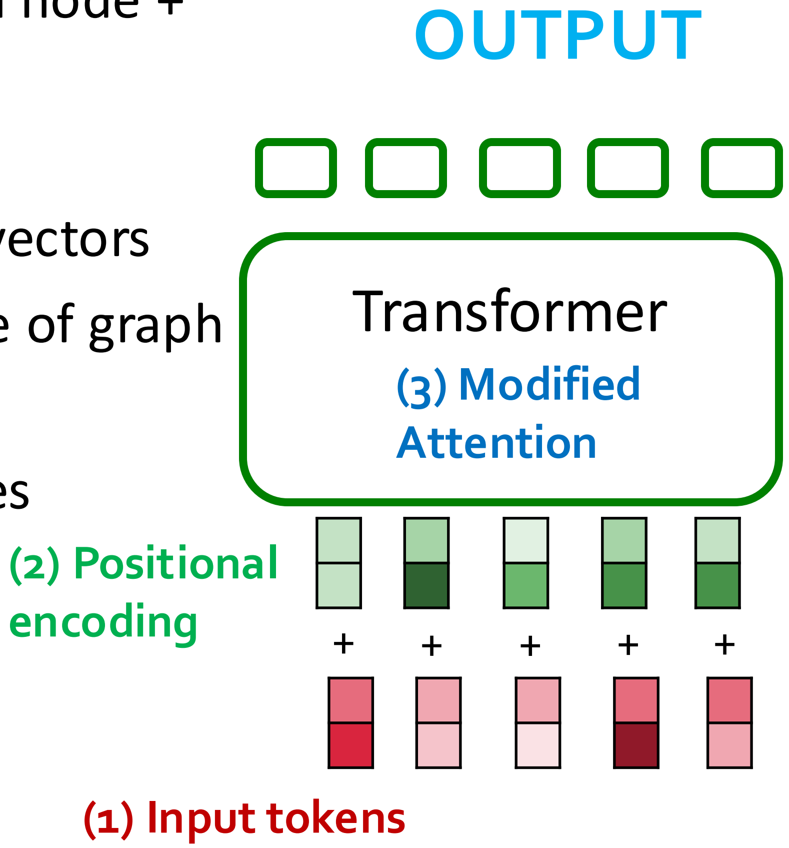
  Learned parameters $w_1, \dots, w_N$

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

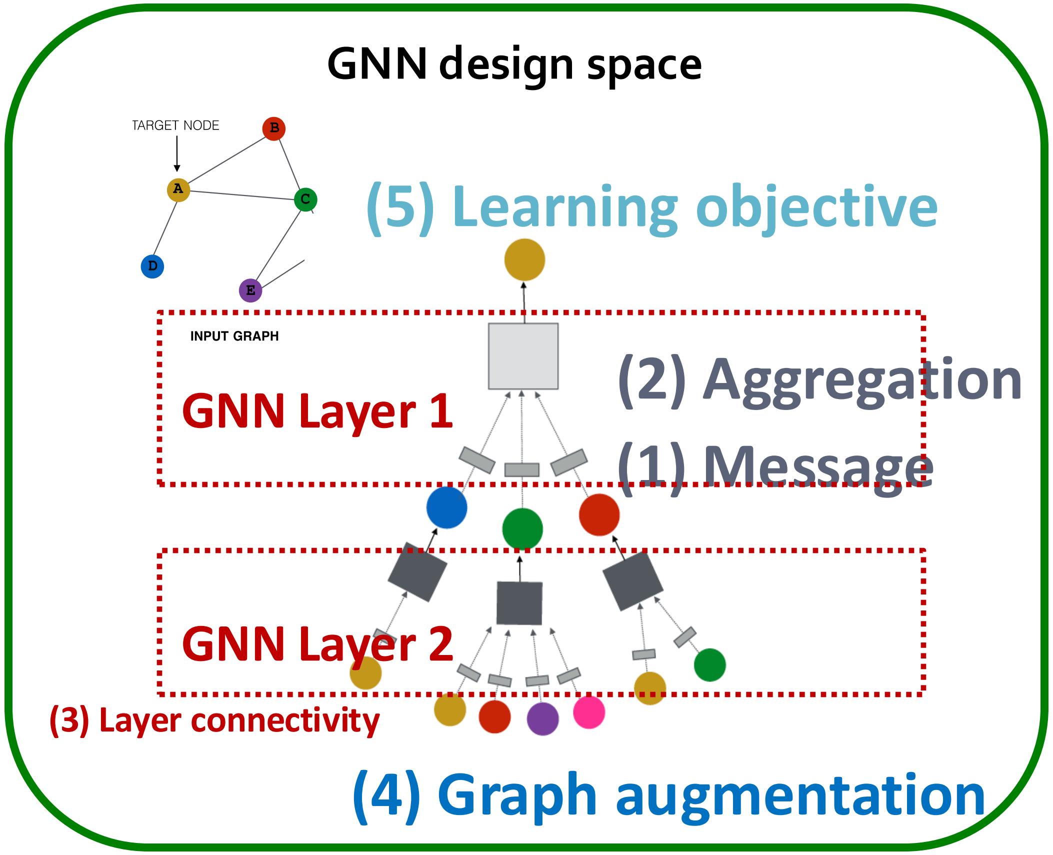
Summary: Graph Transformer Design Space

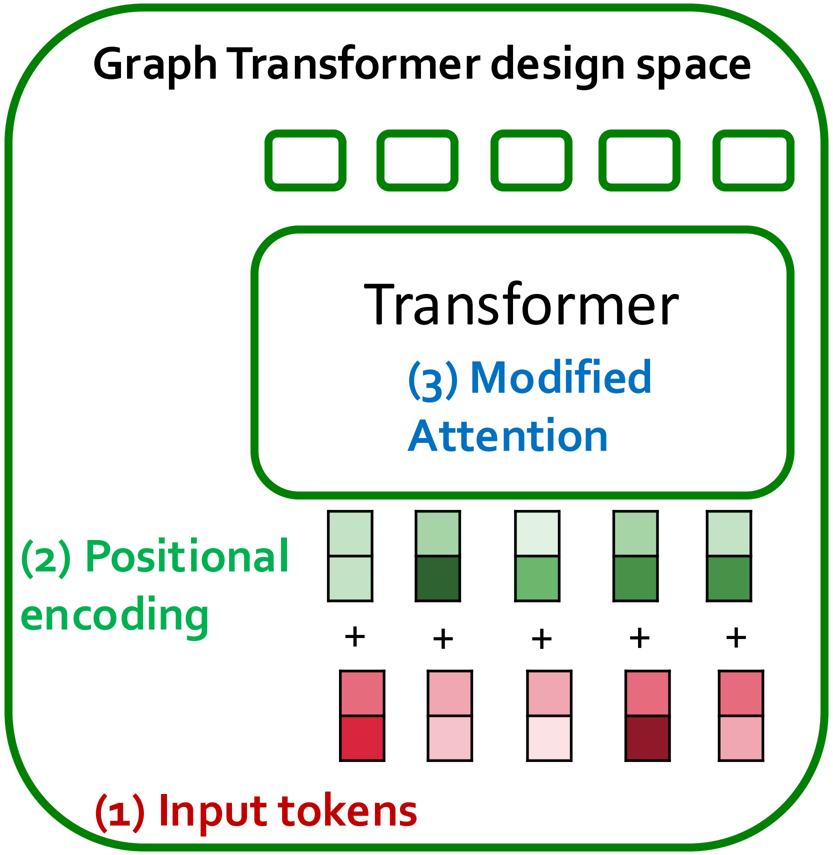
- **(1) Tokenization**
  - Usually node features
  - Other options, such as subgraphs, and node + edge features (not discussed today)
- **(2) Positional Encoding**
  - Relative distances, or Laplacian eigenvectors
  - Gives Transformer adjacency structure of graph
- **(3) Modified Attention**
  - Reweight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





Summary: Graph Transformer Design Space





Plan for Today

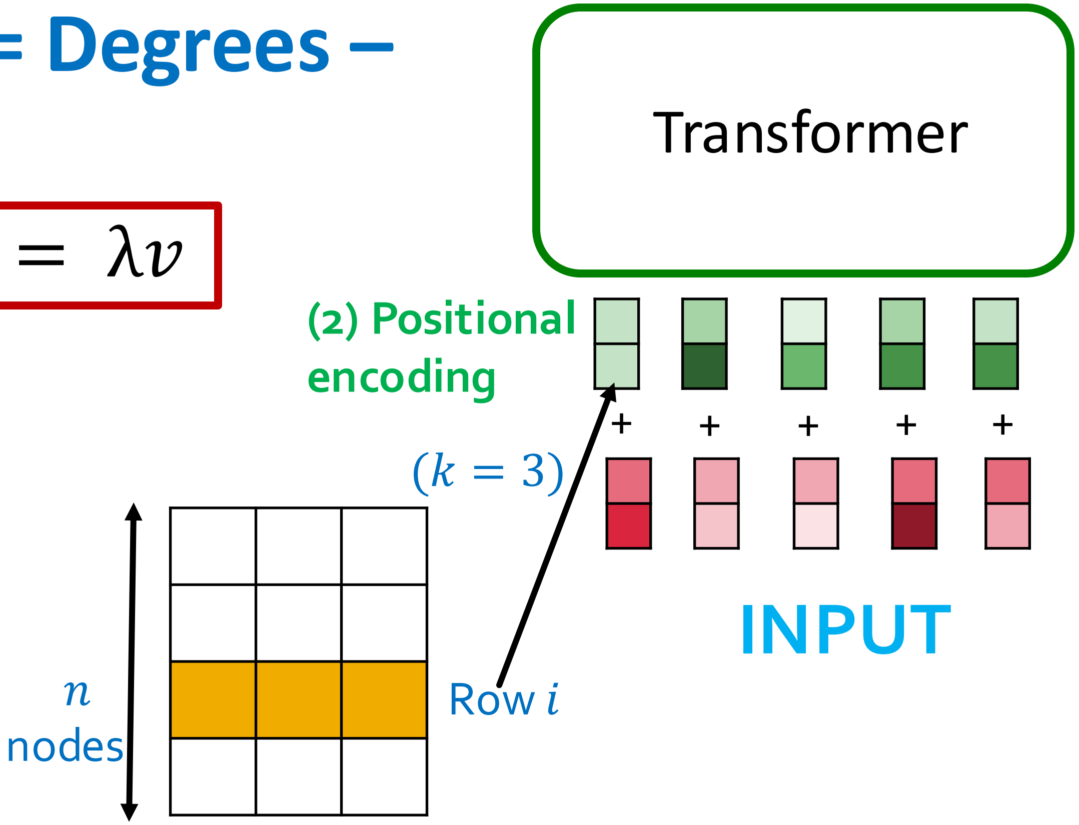
- **Part 1:**
  - Introducing Transformers
  - Relation to message passing GNNs
- **Part 2:**
  - A new design landscape for graph Transformers
- **Part 3 (time permitting):**
  - PEARL: Learning Efficient Positional Encodings with GNNs

Stanford CS224W: Powerful Positional Encodings for Graph Transformers

CS224W: Machine Learning with Graphs
Jure Leskovec, Stanford University
http://cs224w.stanford.edu

Recall: Laplacian Eigenvector Positional Encodings

- Laplacian Matrix **L = Degrees – Adjacency**
- Eigenvector: v such that $Lv = \lambda v$



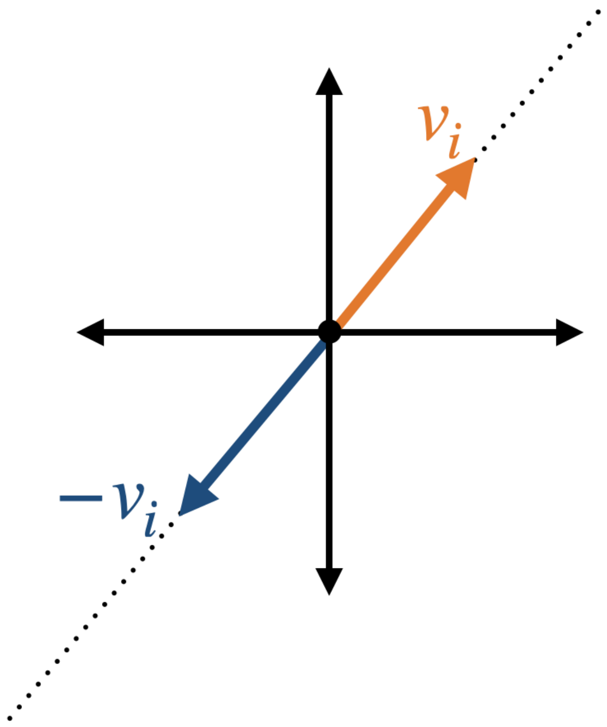
Laplacian Eigenvector Positional Encodings

- 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

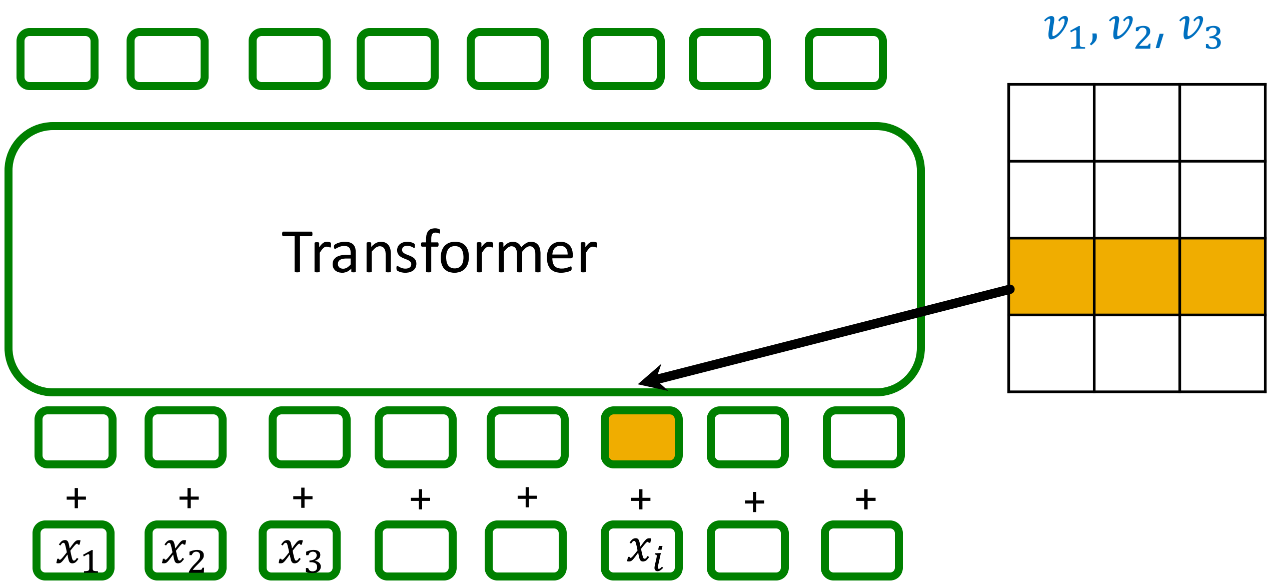
- Suppose v is a Laplacian eigenvector
  - So $Lv = \lambda v$
- But this means:
  - Also $L(-v) = \lambda(-v)$
- So $-v$ is also a Laplacian eigenvector

The choice of sign is arbitrary!



Sign Ambiguity is a Problem

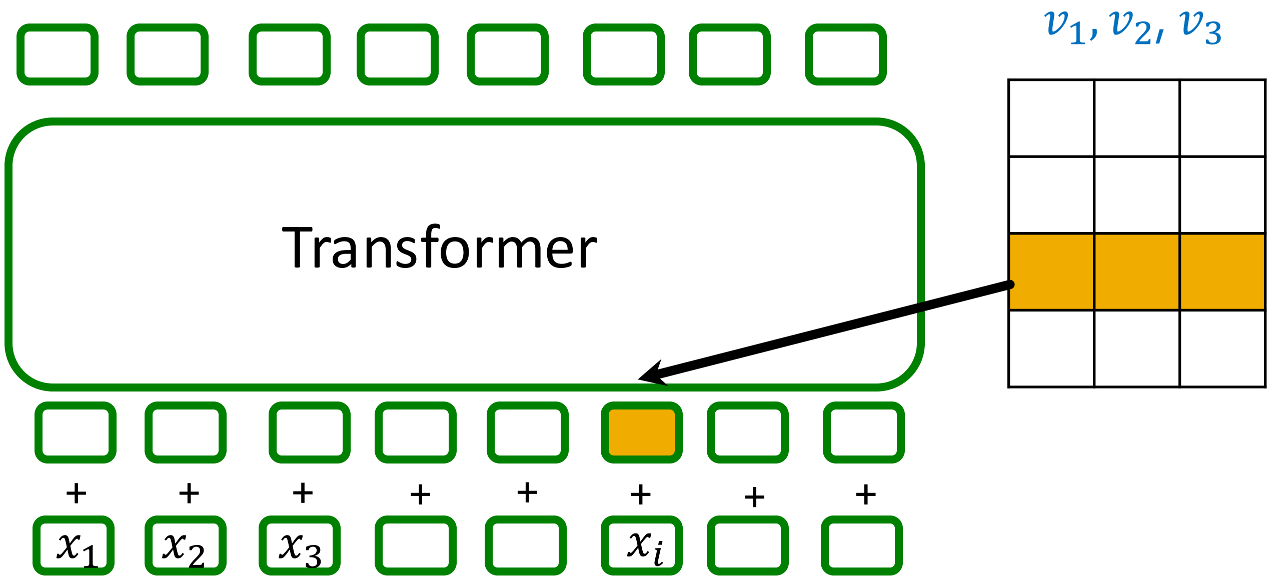
- Both v and $-v$ are eigenvectors
- But when we use them as positional encodings we **pick one arbitrarily**
- **Why does this matter for positional encodings?**



Sign Ambiguity is a Problem

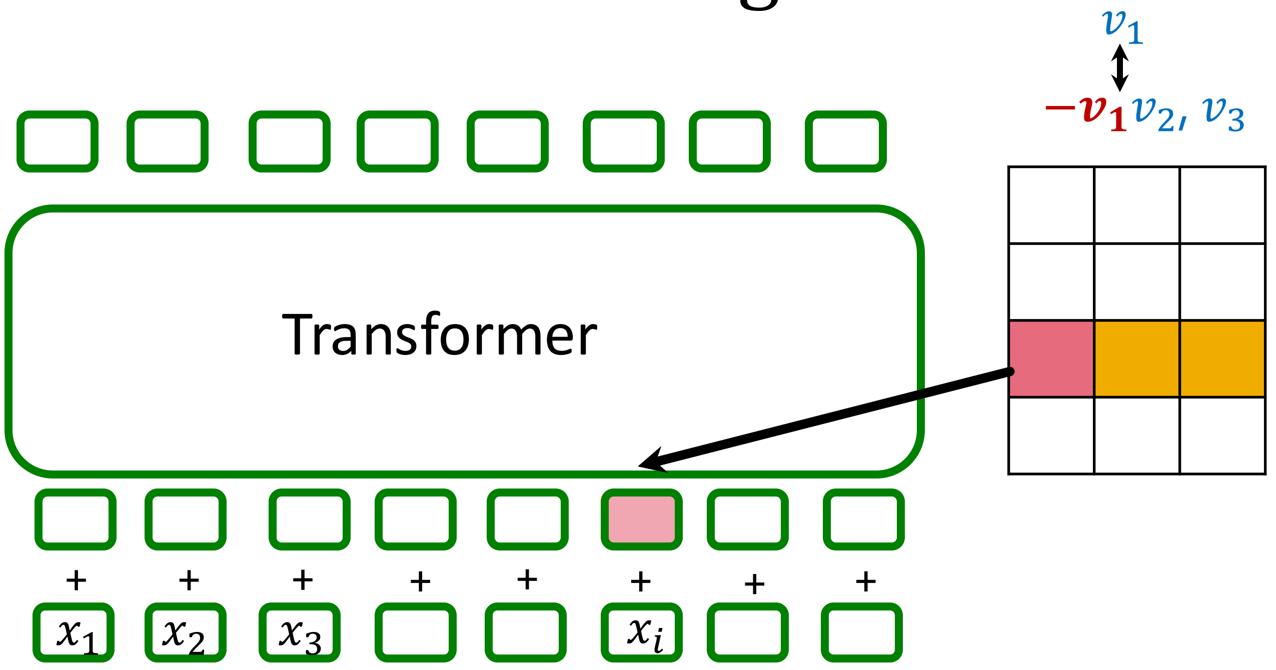
- Both v and $-v$ are eigenvectors
- But when we use them as positional encodings we **pick one arbitrarily**
- **Why does this matter for positional encodings?**

- **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!

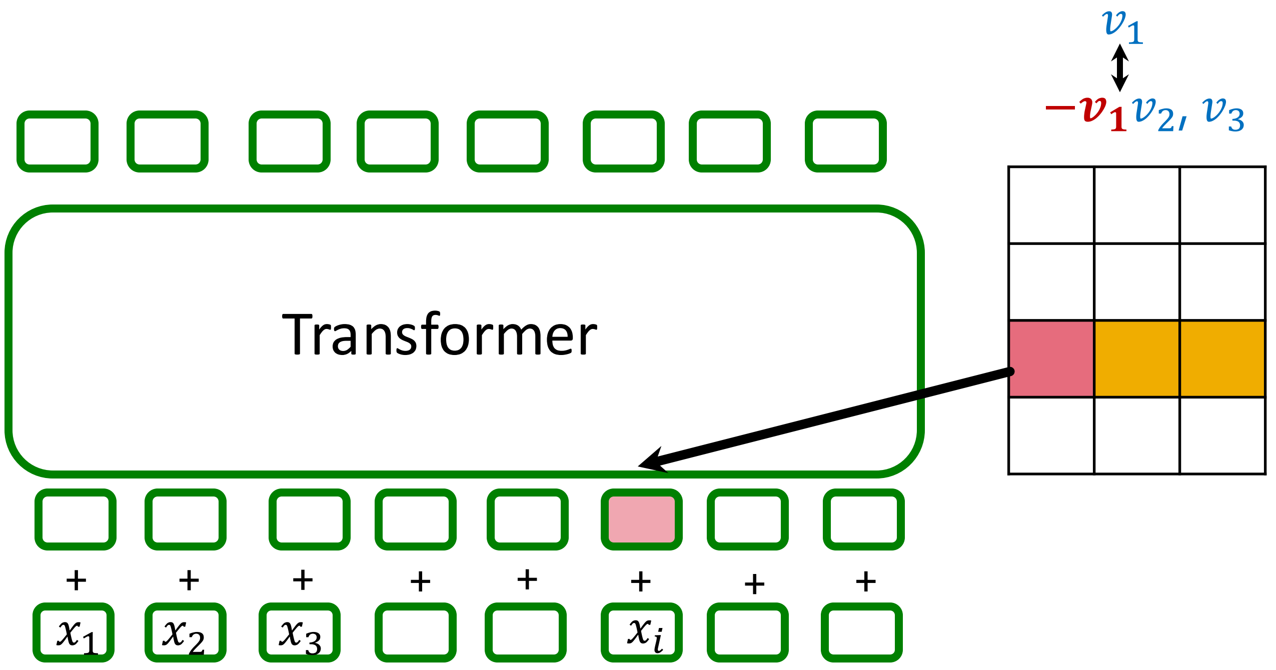


How to fix sign ambiguity

- **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



Laplacian Eigenvector Positional Encodings

- Q: What is the problem with this approach?
  - A1: Computing eigenvectors has cubic complexity
  - A2: Storing them needs quadratic space!
- **Can we do something better?**

C. Kanatsoulis, E. Choi et al. Learning Efficient Positional Encodings with Graph Neural Networks, ICLR 2025

Stanford CS224W: PEARL: Learning Efficient PEs with GNNs

CS224W: Machine Learning with Graphs
Charilaos Kanatsoulis and Jure Leskovec, Stanford University
http://cs224w.stanford.edu

GNNs are nonlinear functions of eigenvectors

- **Recall the GIN update:**

$$\boldsymbol{c}_v^{(l+1)} = \mathrm{MLP}\left((1+\epsilon)\, \boldsymbol{c}_v^{(l)} + \sum_{u \in \mathcal{N}(v)} \boldsymbol{c}_u^{(l)} \right)$$

  - We can consider a single-layer MLP
  - Write the color update in a matrix form:

$$\boldsymbol{C}^{(l+1)} = \sigma\left(\sum_{k=0}^{1} \boldsymbol{A}^k \boldsymbol{C}^{(l)} \boldsymbol{W}_k^{(l)} \right) = \sigma\left(\sum_{k=0}^{1} \boldsymbol{V} \boldsymbol{\Lambda}^k \boxed{\boldsymbol{V}^T \boldsymbol{C}^{(l)}} \boldsymbol{W}_k^{(l)} \right)$$

  - $\boldsymbol{C}^{(l)} \in \mathbb{R}^{N \times d}, \quad \boldsymbol{C}^{(l)}[v,:] = \boldsymbol{c}_v^{(l)}$
  - Where $A \epsilon \{0,1\}^{N \times N}$ is the adjacency matrix of the graph, and:

$$\boldsymbol{A} = \boldsymbol{V} \boldsymbol{\Lambda} \boldsymbol{V}^T$$

GNNs are nonlinear functions of eigenvectors

- **Recall the GIN update:**

$$\boldsymbol{c}_v^{(l+1)} = \mathrm{MLP}\left((1+\epsilon)\, \boldsymbol{c}_v^{(l)} + \sum_{u \in \mathcal{N}(v)} \boldsymbol{c}_u^{(l)} \right)$$

  - We can consider a single-layer MLP

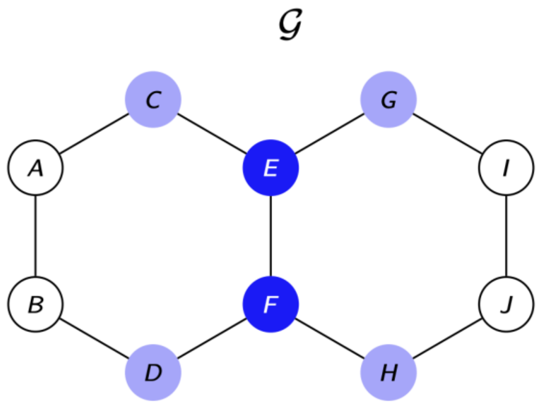
$$\boldsymbol{c}_v^{(l+1)}[f] = \sigma\left(\sum_{n=1}^{N} w\,[n, f]\, \boldsymbol{v}_n \right) [v]$$

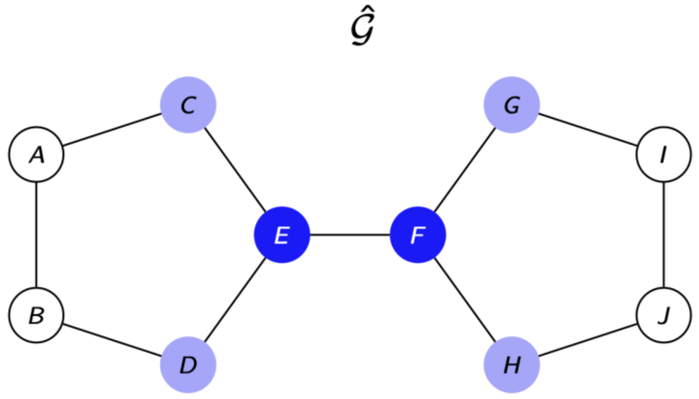
  - Where:

$$w\,[n, f] = \sum_{i=1}^{d} \sum_{k=0}^{1} \lambda_n^k \boldsymbol{W}_k[i, f] \langle \boldsymbol{v}_n, \boldsymbol{C}^{(l)}\,[:, i] \rangle$$

$$\boldsymbol{C}^{(l)} \in \mathbb{R}^{N \times d}, \quad \boldsymbol{C}^{(l)}\,[v, :] = \boldsymbol{c}_v^{(l)}$$

Limitation of the WL kernel



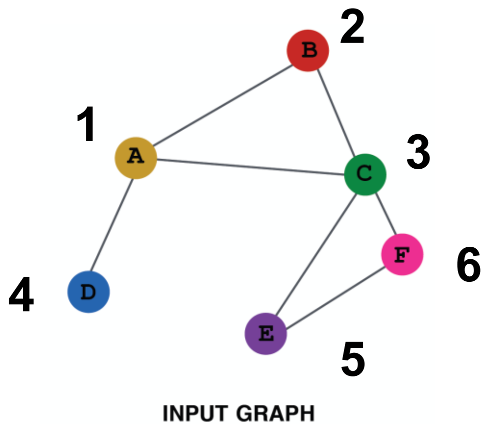


- **The WL kernel** colors **inherit the graph symmetries**.
- **Symmetric colors** are associated with limitations involving the **spectral decomposition of the graph**.

Random samples as node ID's

Can we break these symmetries?

- **Standard approach:**
- **Assign unique IDs to nodes**
  - These IDs can be converted into **one-hot vectors**



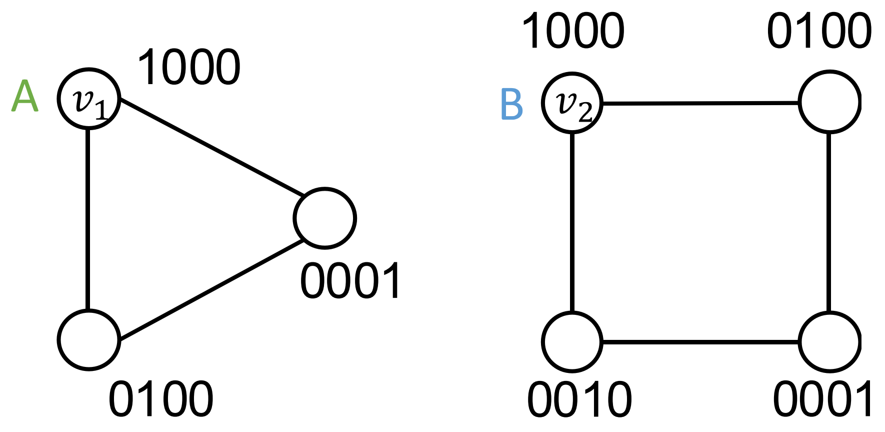
One-hot vector for node with ID=5

$$\underbrace{[0, 0, 0, 0, \overset{\substack{\text{ID = 5}\\ \downarrow}}{1}, 0]}_{\text{Total number of IDs = 6}}$$

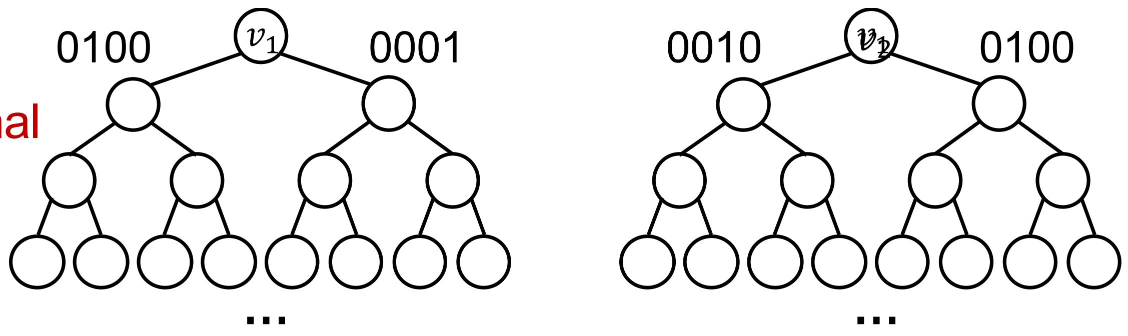
Naïve Solution is not Desirable

- **A naïve solution:** One-hot encoding
  - Encode each node with a different ID, then we can always differentiate different nodes/edges/graphs

Input graphs



Computational graphs

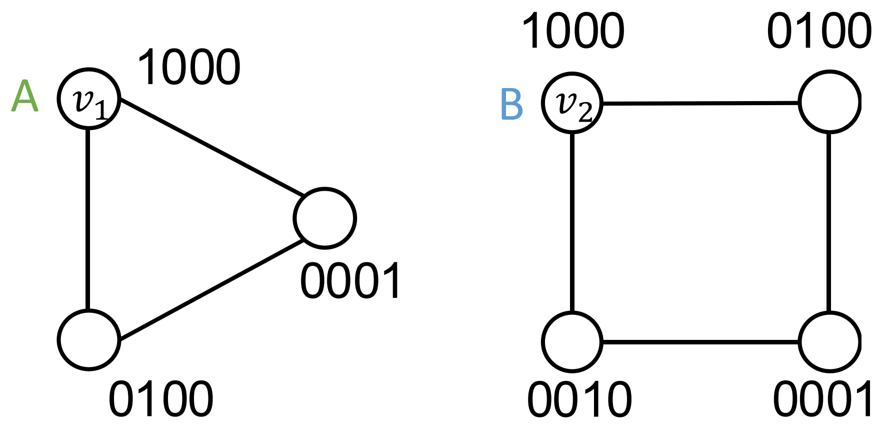


Computational graphs are clearly different if each node has a different ID

Naïve Solution is not Desirable

- **A naïve solution:** One-hot encoding
  - Encode each node with a different ID, then we can always differentiate different nodes/edges/graphs

Input graphs

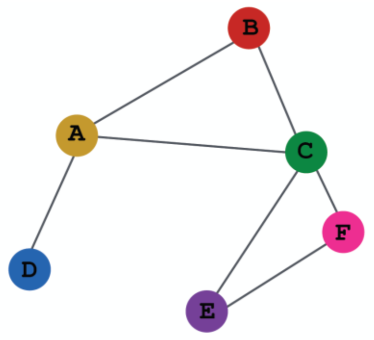


  - **Issues:**
    - **Not scalable**: Need $O(N)$ feature dimensions (N is the number of nodes)
    - **Not inductive:** Cannot generalize to new nodes/graphs

PEARL Positional Encodings

Can we learn powerful PEs with GNNs only?

- **Assign unique IDs to nodes**
  - These IDs are represented by **random samples**
  - **Each node will be represented by a different set of random variables**

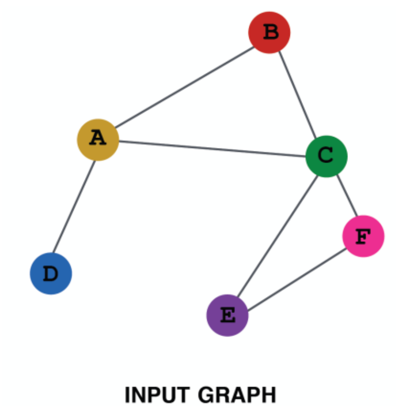


INPUT GRAPH

Random samples for node 3

$$\underbrace{[0.2, 1.5, -2.3, -10.1]}_{\text{Total number of random samples = 4}}$$

Independent Processing of samples



Node A [3.3, -1.7, -1.2, -0.1]

Node B [-0.1, -5.4, 3.0, -9.8]

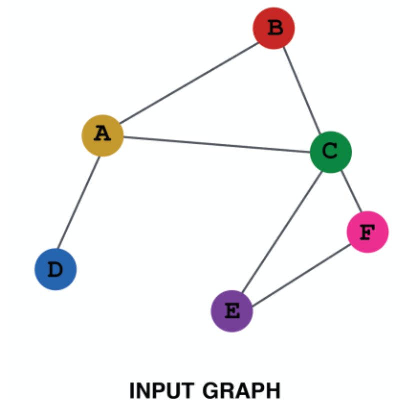
Node C [0.2, 1.5, -2.3, -10.1]

Node D [0.5, 1.9, -12.7, 11.1]

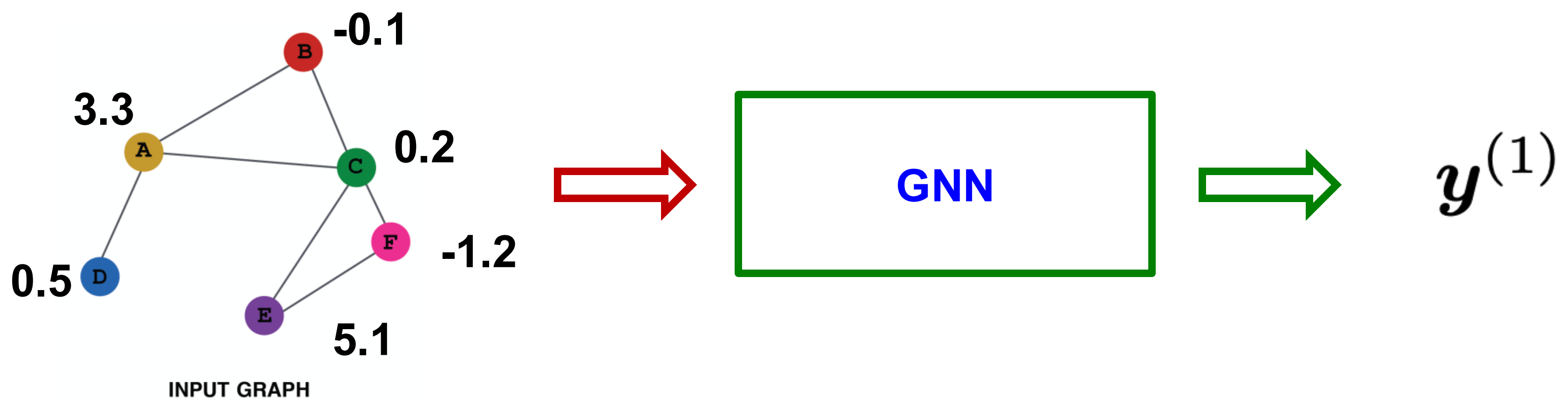
Node E [5.1, -0.7, -2.9, -13.5]

Node F [-1.2, 7.5, -0.3, -7.9]

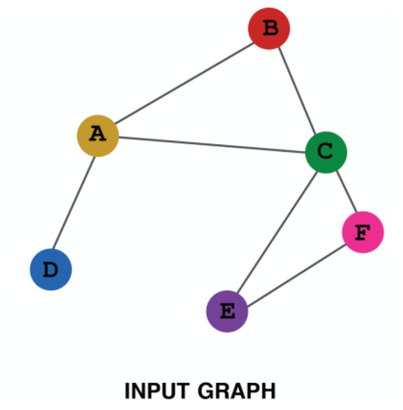
Independent Processing of samples



Node A [3.3, -1.7, -1.2, -0.1**]**
Node B [-0.1, -5.4, 3.0, -9.8**]**
Node C [0.2, 1.5, -2.3, -10.1**]**
Node D [0.5, 1.9, -12.7, 11.1**]**
Node E [5.1, -0.7, -2.9, -13.5**]**
Node F [-1.2, 7.5, -0.3, -7.9**]**



Independent Processing of samples



Node A [3.3, **-1.7**, -1.2, -0.1]

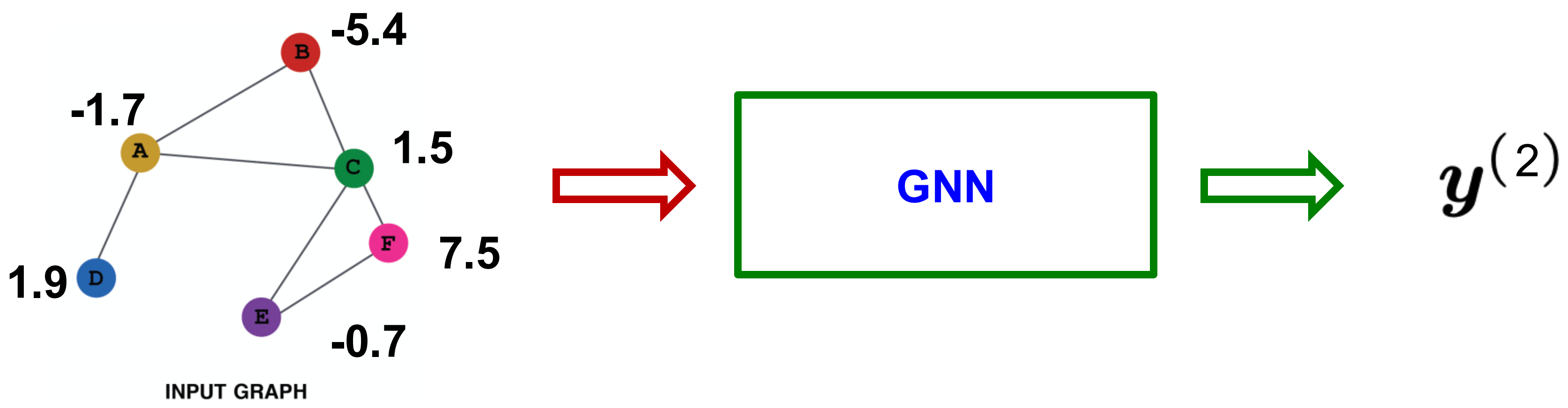
Node B [-0.1, **-5.4**, 3.0, -9.8]

Node C [0.2, **1.5**, -2.3, -10.1]

Node D [0.5, **1.9**, -12.7, 11.1]

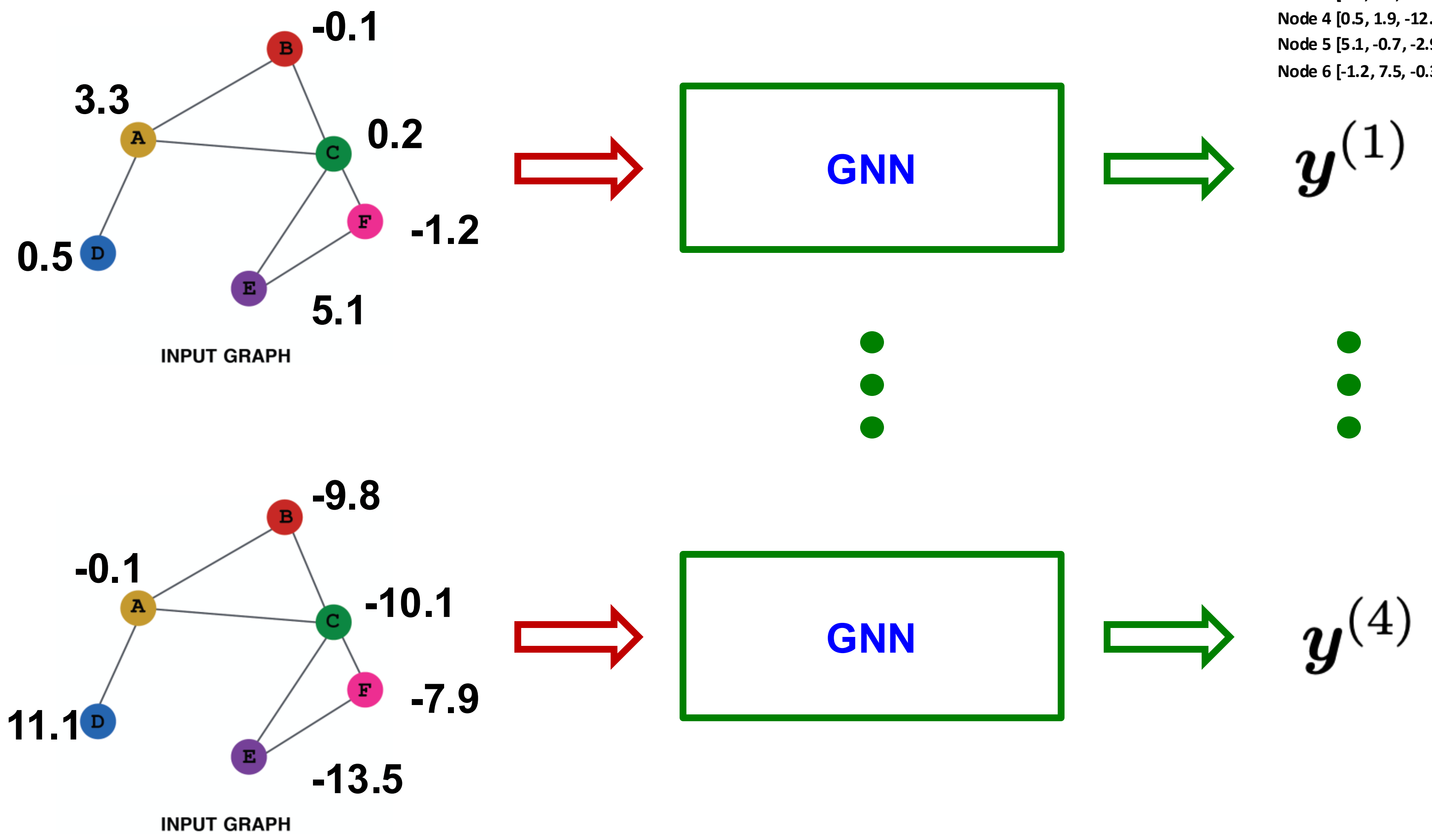
Node E [5.1, **-0.7**, -2.9, -13.5]

Node F [-1.2, **7.5**, -0.3, -7.9]



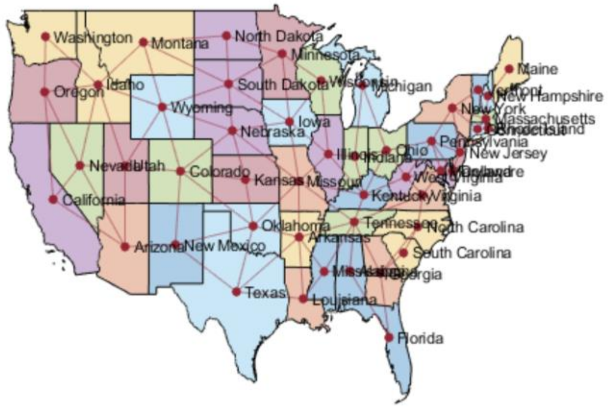
Independent Processing of samples

Node 1 [3.3, -1.7, -1.2, -0.1]
Node 2 [-0.1, -5.4, 3.0, -9.8]
Node 3 [0.2, 1.5, -2.3, -10.1]
Node 4 [0.5, 1.9, -12.7, 11.1]
Node 5 [5.1, -0.7, -2.9, -13.5]
Node 6 [-1.2, 7.5, -0.3, -7.9]



An approximate analogy

Sample one person per state

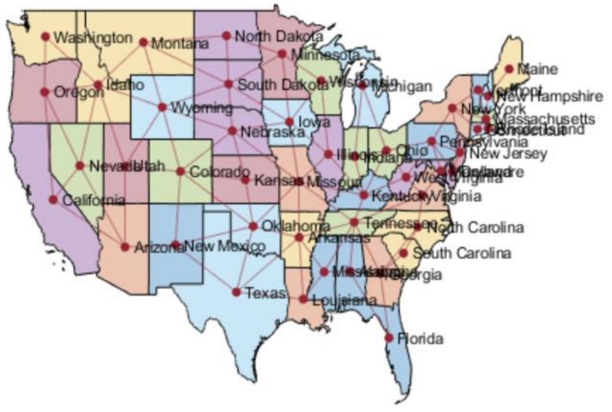


INPUT GRAPH

$\Rightarrow$ Set of questions $\Rightarrow$ $y^{(1)}$

⋮ ⋮

Sample one person per state

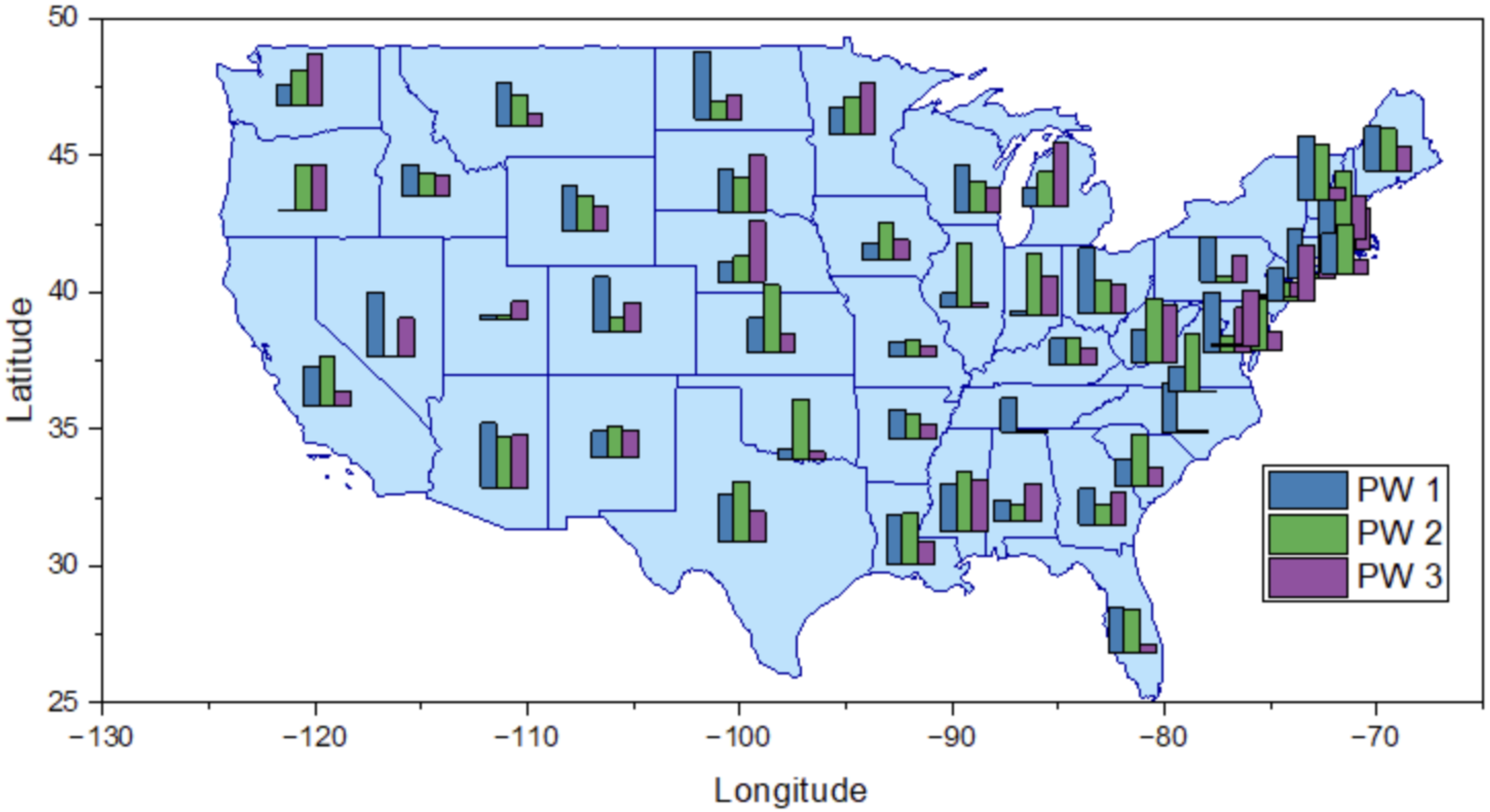


INPUT GRAPH

$\Rightarrow$ Set of questions $\Rightarrow$ $y^{(4)}$

https://blogs.mathworks.com/images/loren/2017/polygonmaps2_01.png

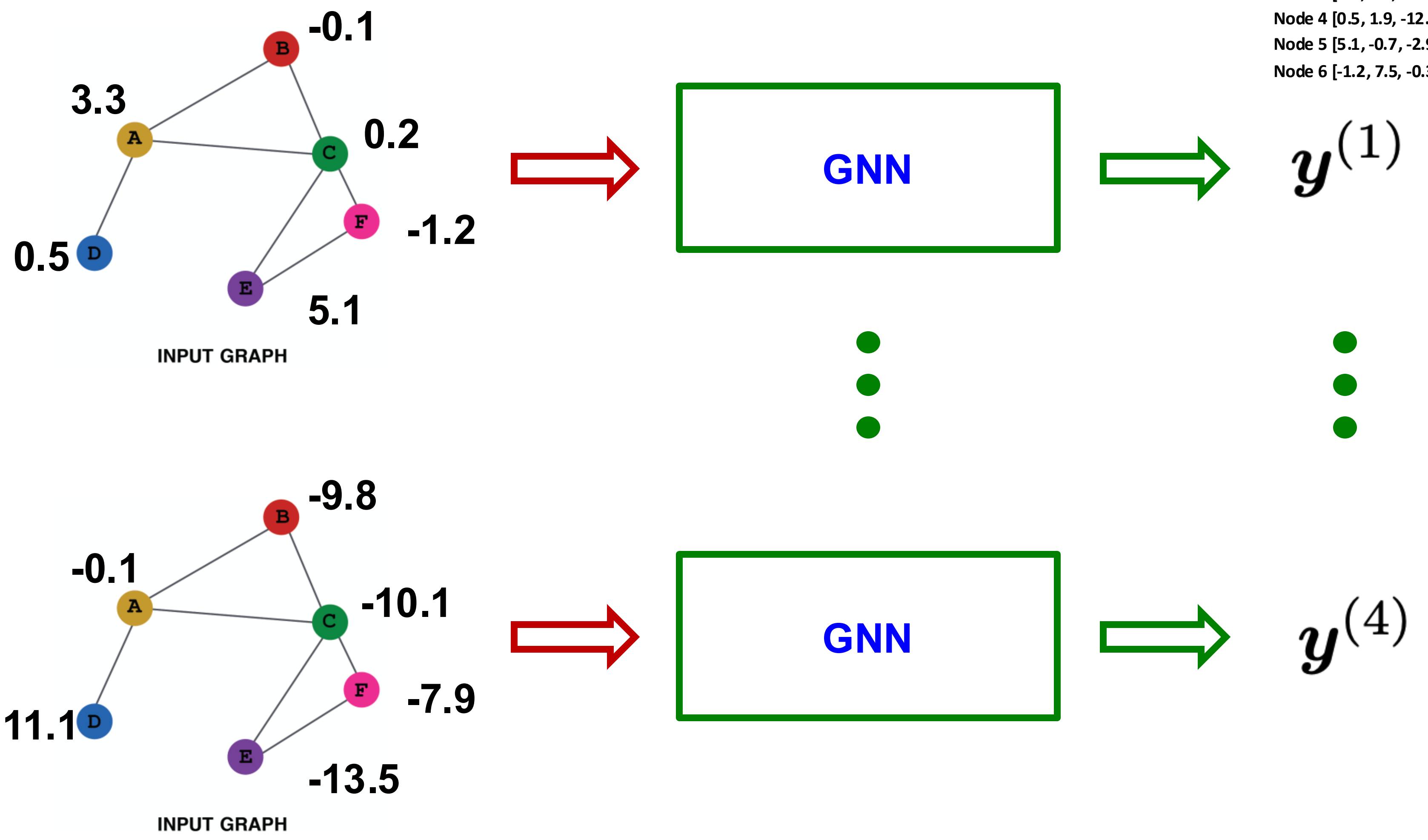
An approximate analogy



https://www.originlab.com/doc/Origin-Help/Bar-Map

Independent Processing of samples

Node 1 [3.3, -1.7, -1.2, -0.1]
Node 2 [-0.1, -5.4, 3.0, -9.8]
Node 3 [0.2, 1.5, -2.3, -10.1]
Node 4 [0.5, 1.9, -12.7, 11.1]
Node 5 [5.1, -0.7, -2.9, -13.5]
Node 6 [-1.2, 7.5, -0.3, -7.9]



C. Kanatsoulis, et al. Learning Efficient Positional Encodings with Graph Neural Networks, ICLR 2025

Counting Cycles with GNNs

- **To maintain inductive capability the final output:**

$$\boldsymbol{y} = \mathbb{E}\left[\boldsymbol{y}^{(m)}\right]$$

  - Which in practice is computed as:

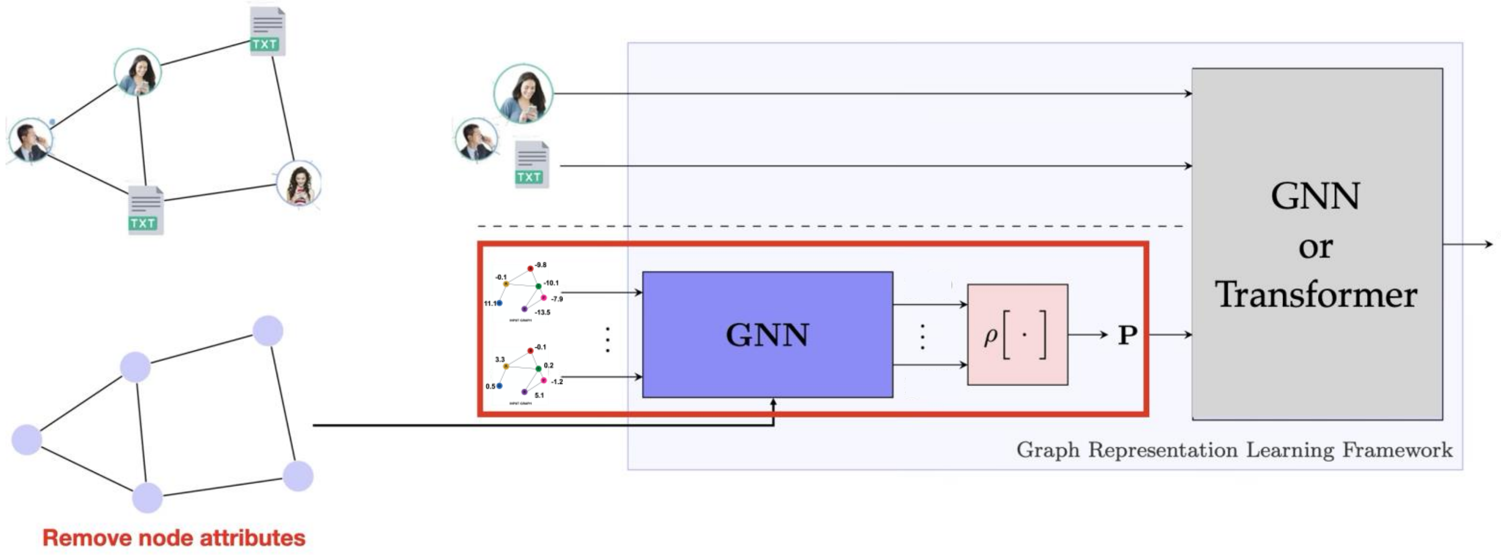
$$\boldsymbol{y} = \frac{1}{M}\sum_{m=1}^{M}\boldsymbol{y}^{(m)}$$

Theorem: PEARL is strictly more expressive than the WL-test.

PEARL can count cycles 7-node cycles with near linear complexity.

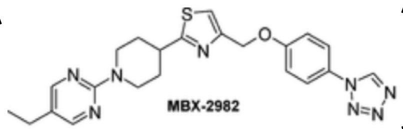
PEARL in practice

- **How to use PEARL: in practice?**
  - Step 1: Sample node ids from a probability distribution.
  - Step 2: Process each set of node samples independently via a GNN.
  - Step 3: Summarize the outputs via empirical expectation.
  - Step 4: concatenate PEARL embeddings with node features X.
  - Step 5: pass through main GNN/Transformer as usual.
  - Step 6: Backpropagate gradients to train PEARL + Prediction model jointly.

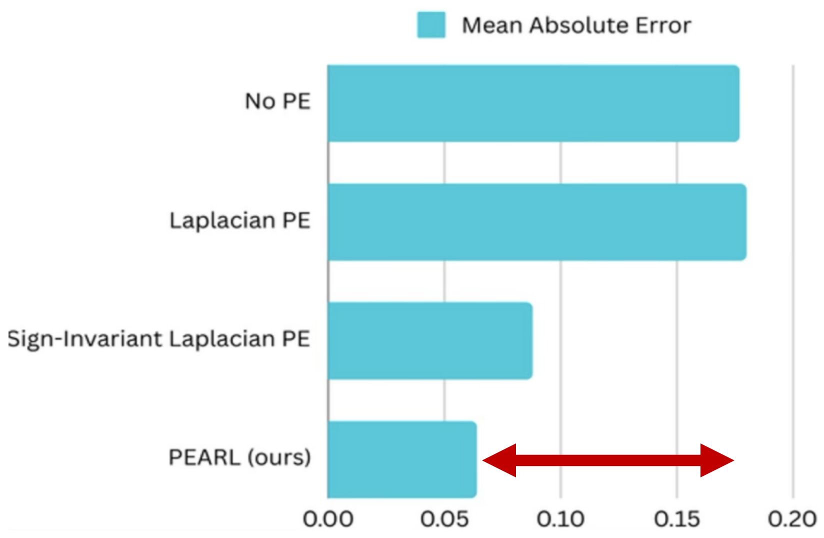


Small molecule property prediction with SignNet

- **Task:** given a small molecule, predict its **solubility**

$f($  $) =$ solubulity

60% reduction in test error

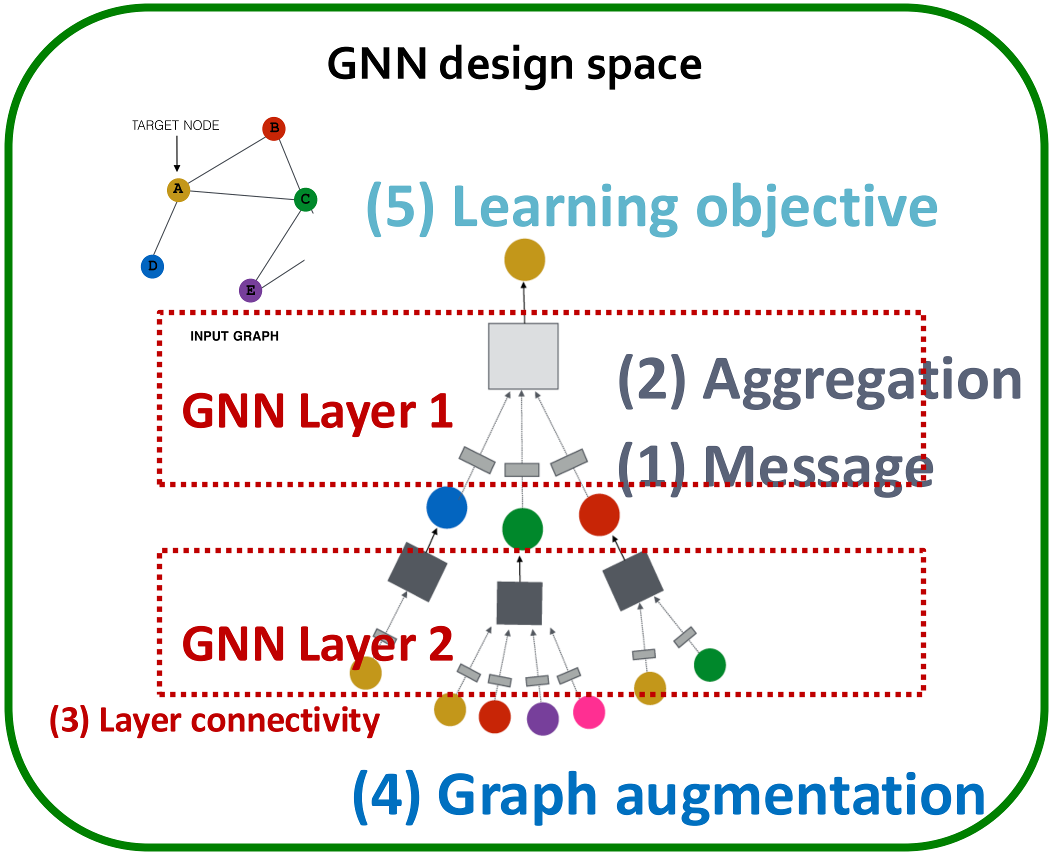


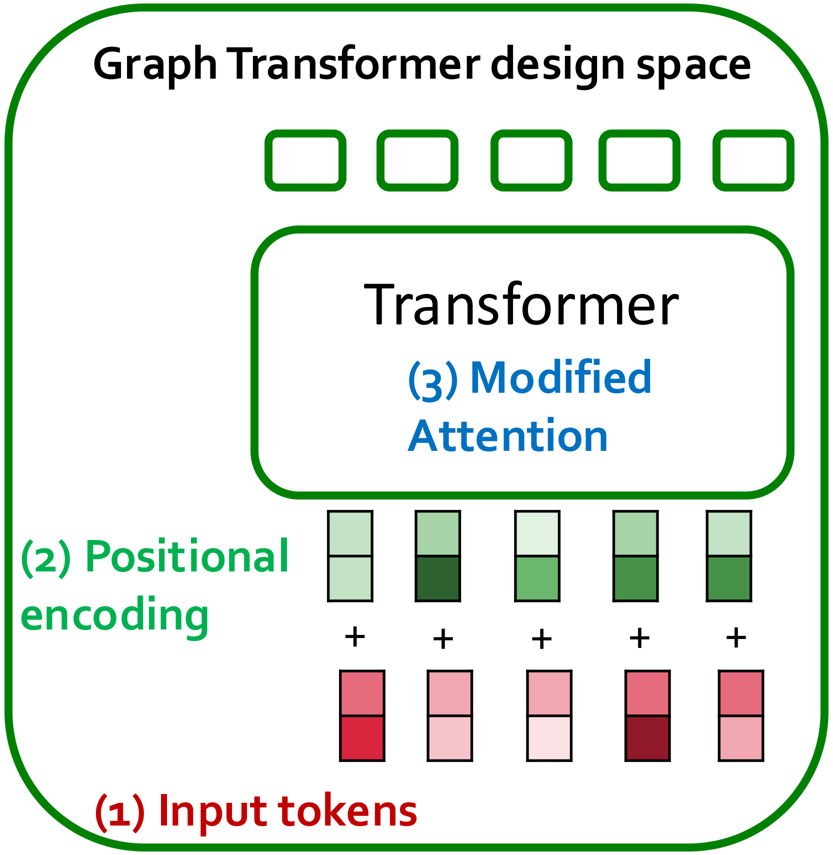
Plan for Today

- **Part 1:**
  - Transformers to message passing on fully connected graph
- **Part 2:**
  - New design landscape for graph Transformers
    - Tokenization
    - Positional encoding
    - Modified self-attention
- **Part 3:**
  - PEARL: positional encodings for graph Transformers

Summary: Graph Transformer Design Space

- New design space for graph Transformers


GNN design space
TARGET NODE
A
B
C
D
E
INPUT GRAPH
(5) Learning objective
GNN Layer 1
(2) Aggregation
(1) Message
GNN Layer 2
(3) Layer connectivity
(4) Graph augmentation


Graph Transformer design space
Transformer
(3) Modified Attention
(2) Positional encoding
+
(1) Input tokens