

Machine Learning:

Lecture #3

Jennifer Ngadiuba (Fermilab)
University of Pavia, May 8-12 2023

Overview of the lectures

- **Day 1:**

- Introduction to Machine Learning fundamentals
- Linear Models

- **Day 2:**

- Neural Networks
- Deep Neural Networks
- Convolutional Neural Networks

- **Day 3:**

- [Recurrent Neural Networks](#)
- [Graph Neural Networks \(part 1\)](#)

- **Day 4:**

- Graph Neural Networks (part 2)
- Transformers

- **Day 5:**

- Unsupervised learning
- Autoencoders
- Generative Adversarial Networks
- Normalizing Flows

Hands on sessions each day will closely follow the lectures topics

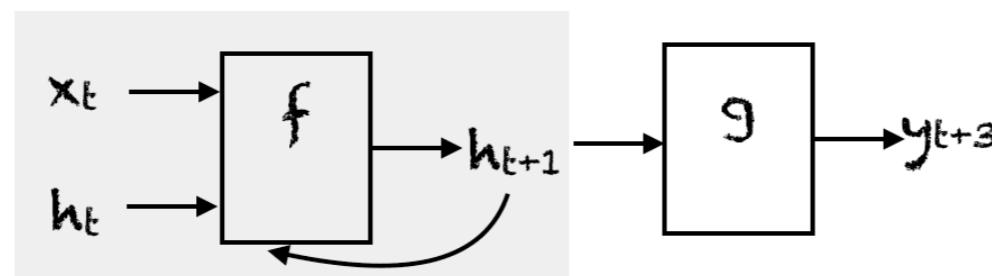
Recurrent Neural Networks

Sequential data

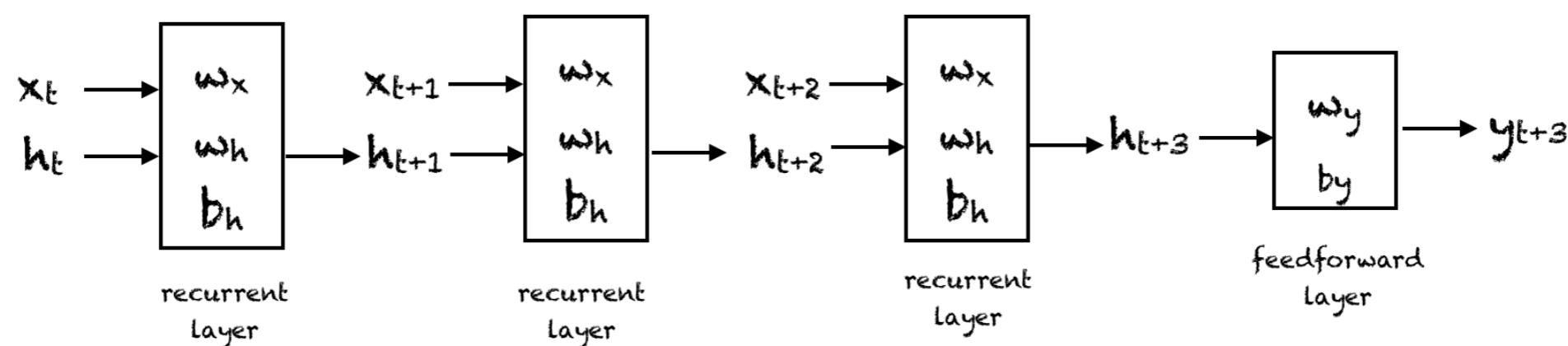
- Many types of data are not fixed in size
- Many types of data have a temporal or sequence-like structure:
 - text
 - video
 - speech
 - DNA
 - ...
- MLPs and CNNs expect fixed size data
- How to deal with sequences?

Recurrent Neural Networks

- A **recurrent neural network (RNN)** is a type of NN which processes sequential data
- They are distinguished by their “memory” as they take information from prior inputs to influence the current input and output



Unfolding the feedback loop in gray for $k=3$



Recurrent Neural Networks

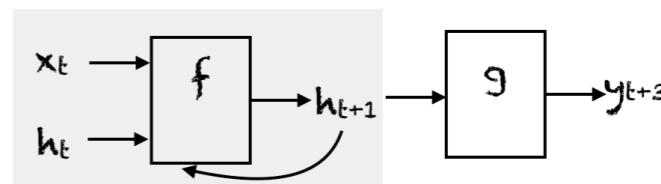
- Given a set \mathcal{X} , let $S(\mathcal{X})$ be the set of sequences, where each element of the sequence $\mathbf{x}_i \in \mathcal{X}$
 - \mathcal{X} could be reals \mathbb{R}^m , integers \mathbb{Z}^m , etc...
 - the sequence $x \in S(\mathcal{X})$ has variable length $T(x) \rightarrow x = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_T\}$
- Use recurrent model that maintains a recurrent state $\mathbf{h}_t \in \mathbb{R}^q$ updated at each time step t :

$$\text{For } t = 1, \dots, T(x): \quad \mathbf{h}_{t+1} = \phi(\mathbf{x}_t, \mathbf{h}_t; \theta)$$

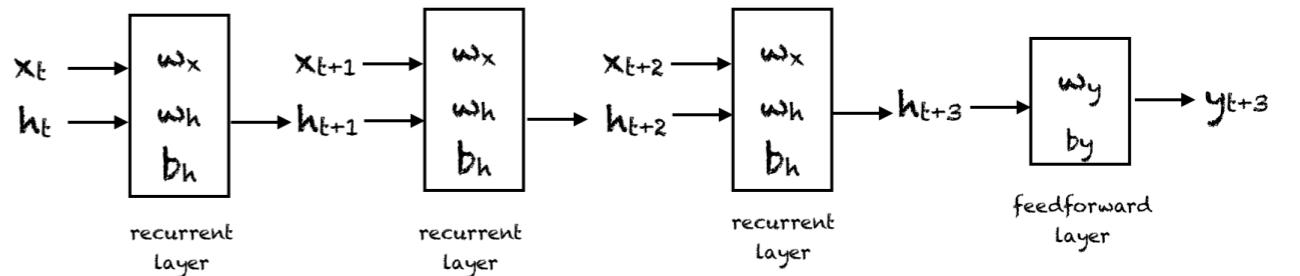
- simplest model: $\phi(\mathbf{x}_t, \mathbf{h}_t; \mathbf{W}, \mathbf{U}) = \sigma(\mathbf{W}\mathbf{x}_t + \mathbf{U}\mathbf{h}_t)$ \longrightarrow single MLP with activation σ
- Predictions can be made at any time step t from the recurrent state:

$$\mathbf{y}_t = \psi(\mathbf{h}_t; \theta)$$

- The hidden state acts as the NN memory — it holds information on previous data the NN has seen

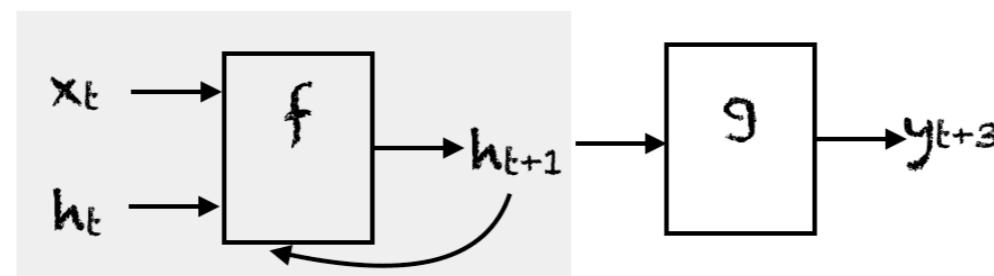


Unfolding the feedback loop in gray for $k=3$

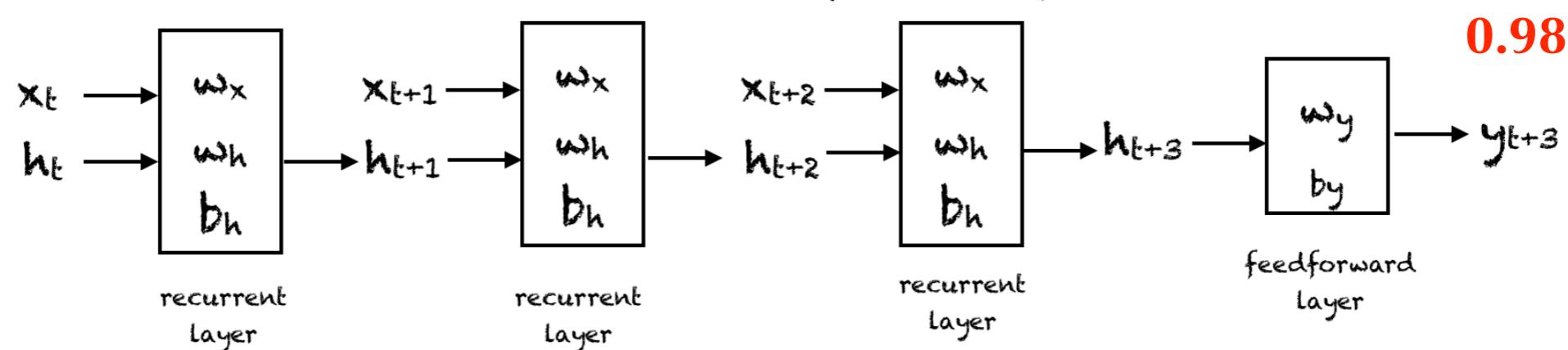


Recurrent Neural Networks

- A **recurrent neural network (RNN)** is a type of NN which processes sequential data
- They are distinguished by their “memory” as they take information from prior inputs to influence the current input and output

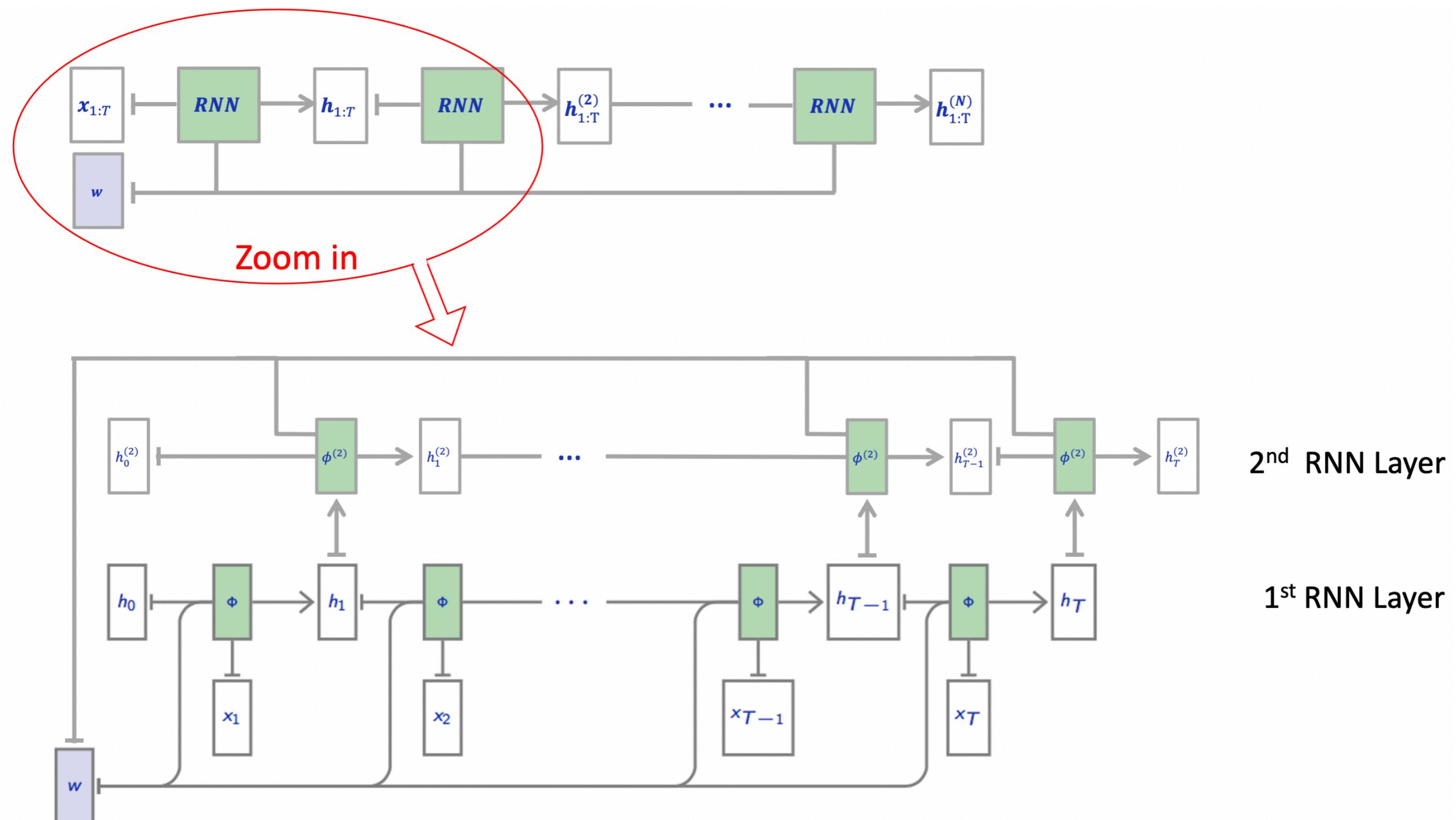


Unfolding the feedback loop in gray for $k=3$

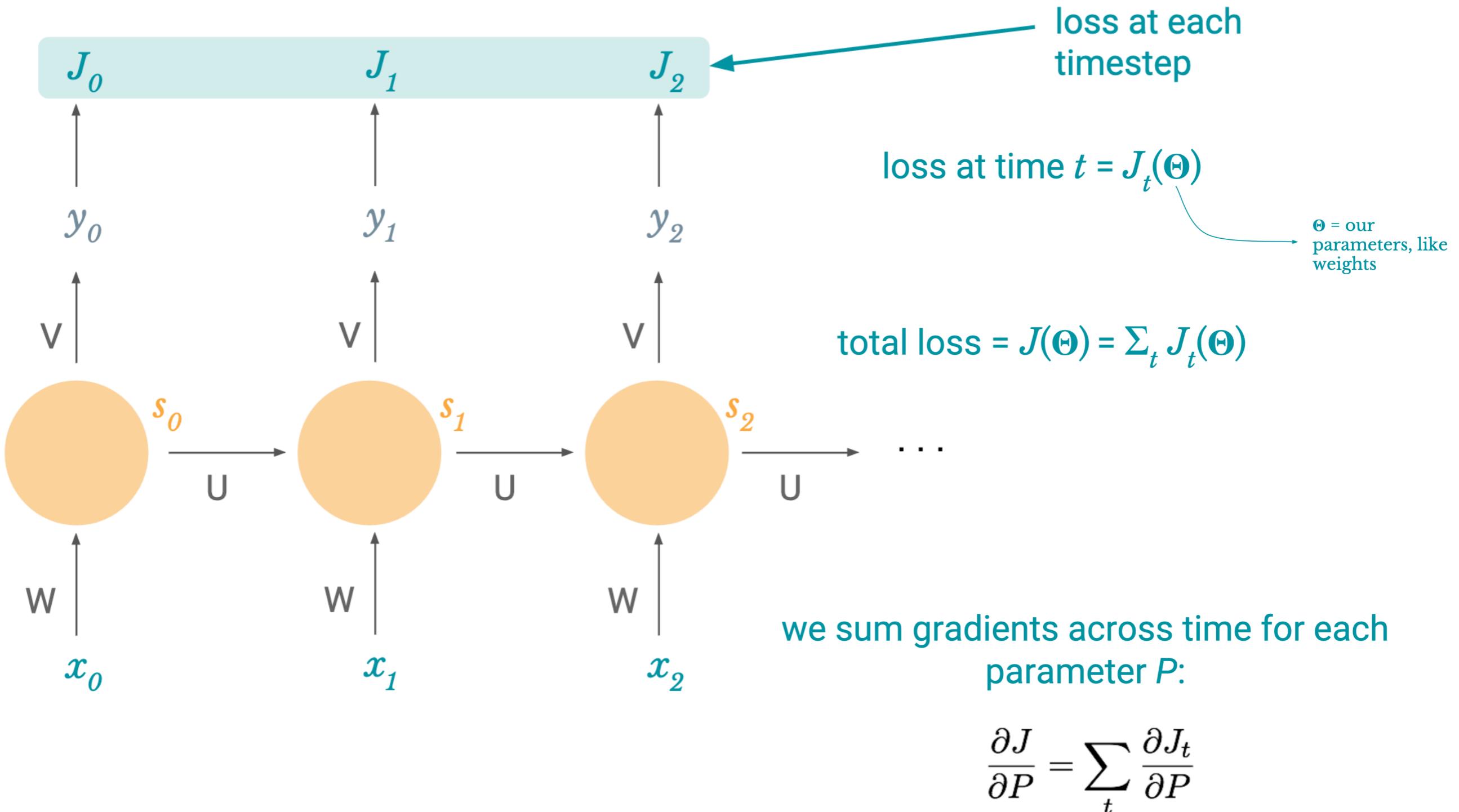


The movie was great

Stacked RNNs



The loss function and gradient

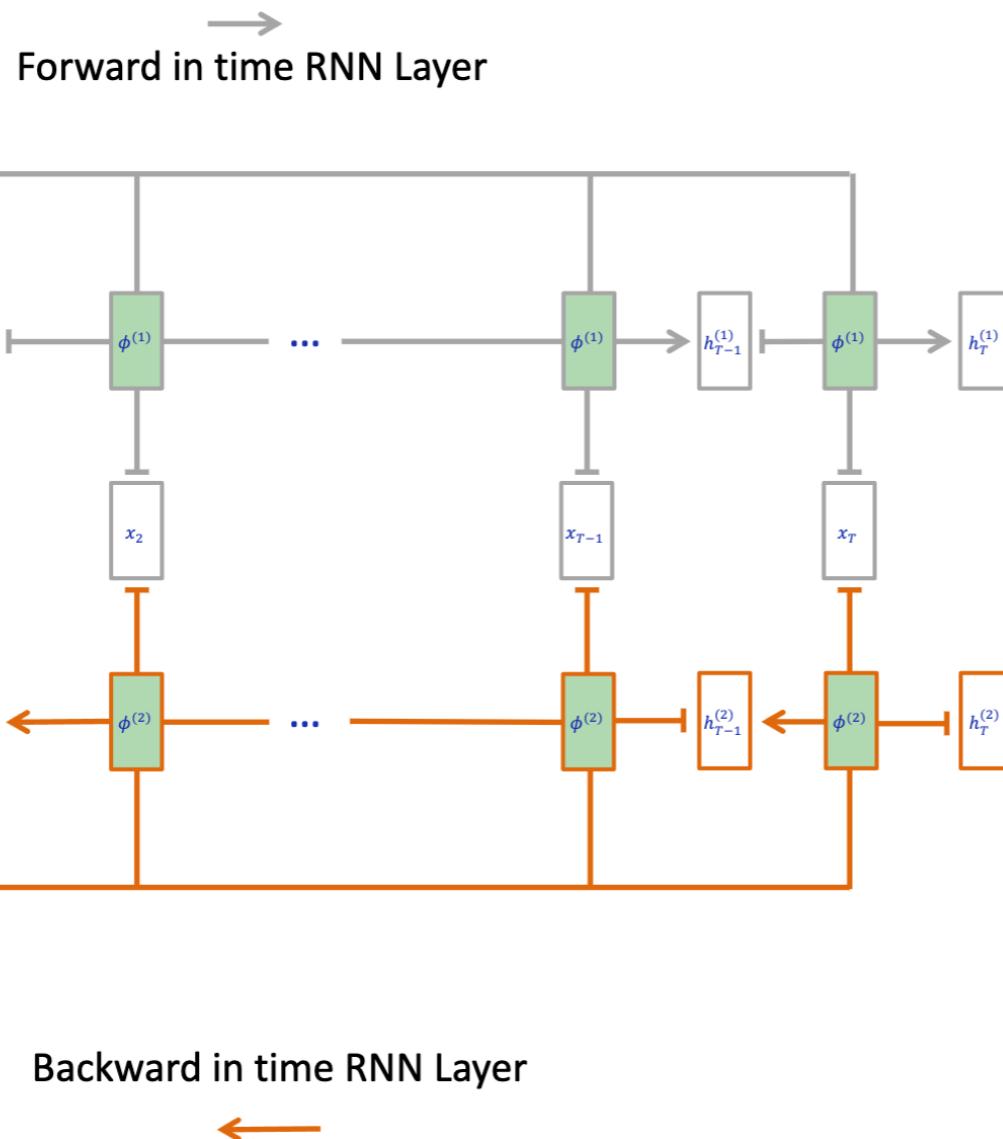


Long Short-Term Memory (LSTM)

- To learn long-term dependency RNNs can grow very big in time → vanishing gradient
 - the layer that get a small gradient update stops and those are typically the earlier layers → short memory

$$\frac{\partial J_n}{\partial W} = \sum_{k=0}^n \frac{\partial J_n}{\partial y_n} \frac{\partial y_n}{\partial s_n} \frac{\partial s_n}{\partial s_k} \frac{\partial s_k}{\partial W}$$

$\frac{\partial s_n}{\partial s_{n-1}} \frac{\partial s_{n-1}}{\partial s_{n-2}} \cdots \frac{\partial s_3}{\partial s_2} \frac{\partial s_2}{\partial s_1} \frac{\partial s_1}{\partial s_0}$

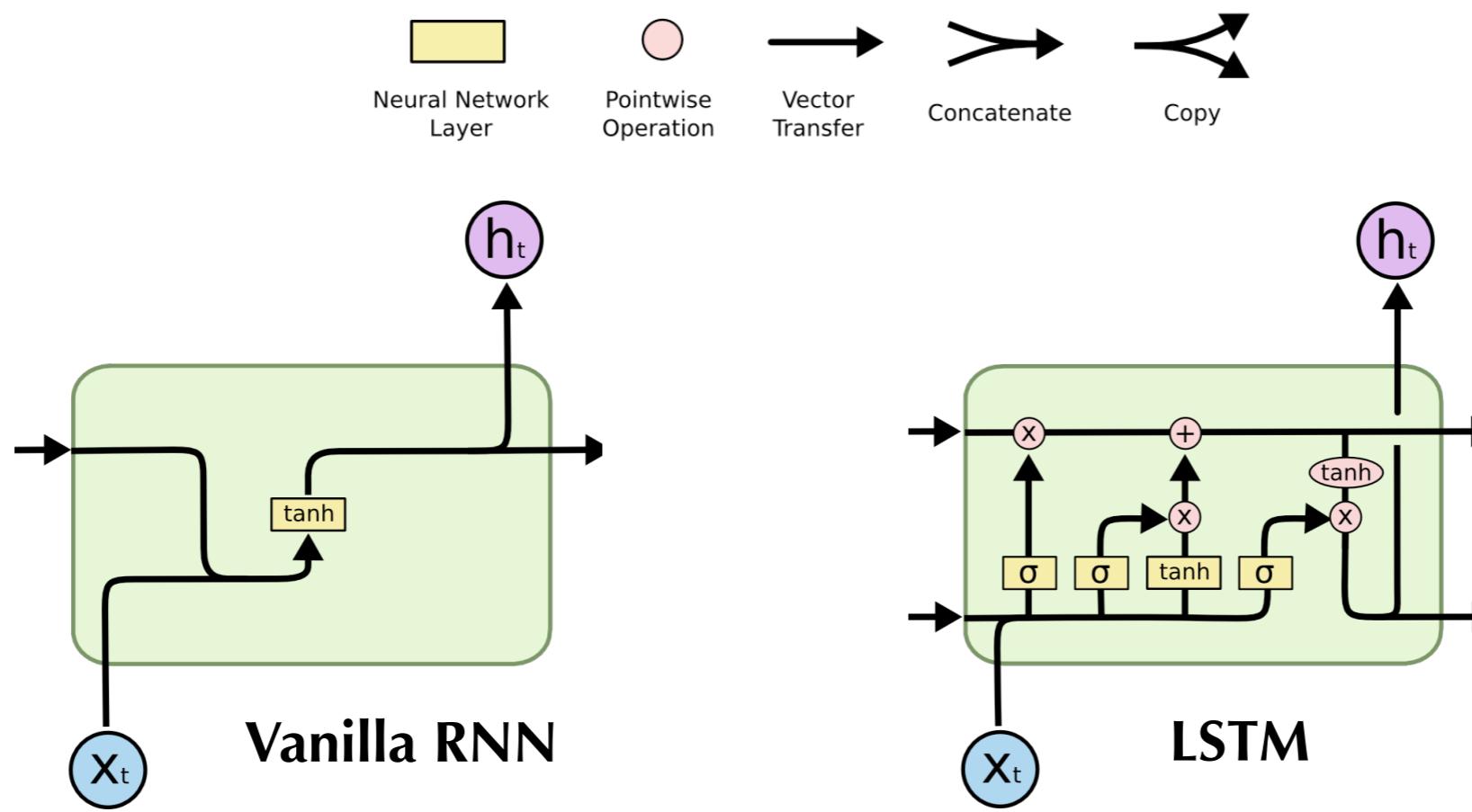


as the gap between timesteps gets bigger, this product gets longer and longer!

[\[Source\]](#)

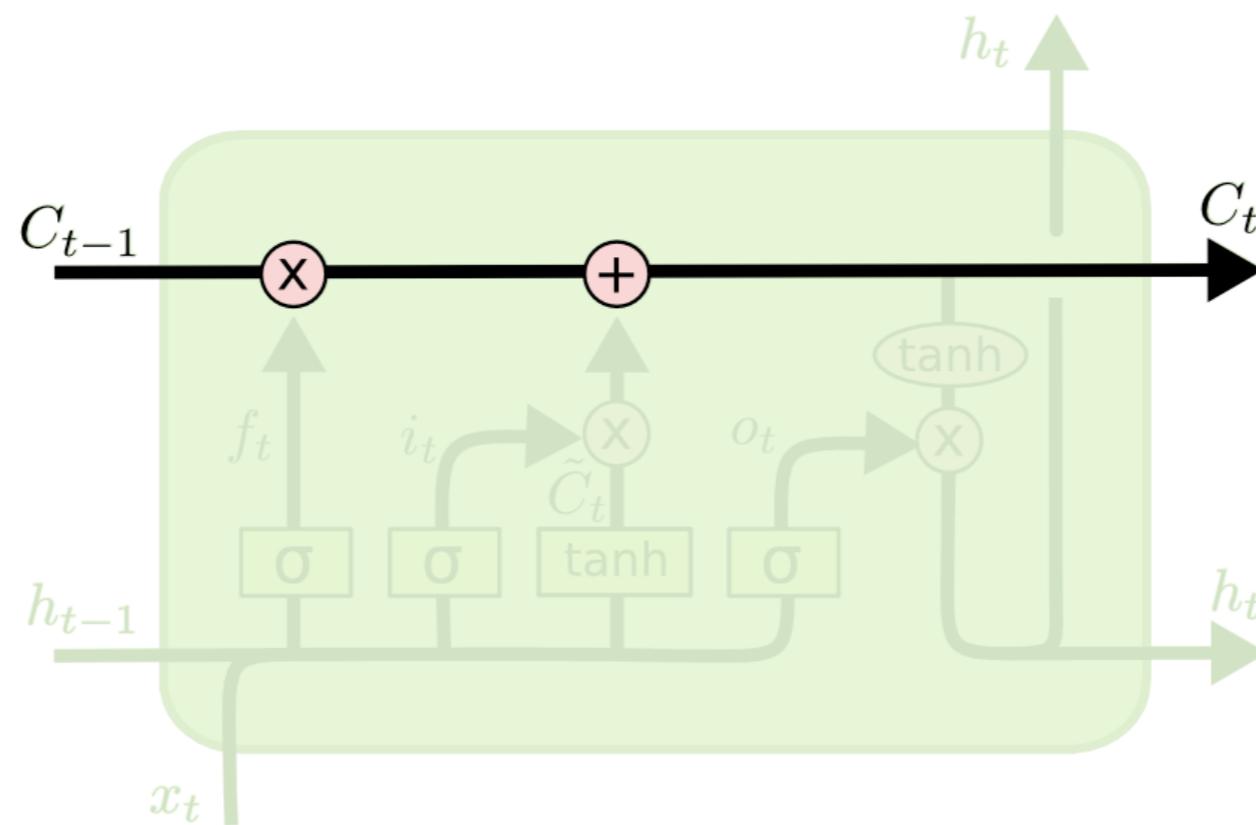
Long Short-Term Memory (LSTM)

- To learn long-term dependency RNNs can grow very big in time → vanishing gradient
 - the layer that get a small gradient update stops and those are typically the earlier layers → short memory
- Breakthrough from [Hochreiter & Schmidhuber \(1997\)](#) : **Long Short-Term Memory NNs**
 - it can learn to keep only relevant information to make predictions, and forget non relevant data



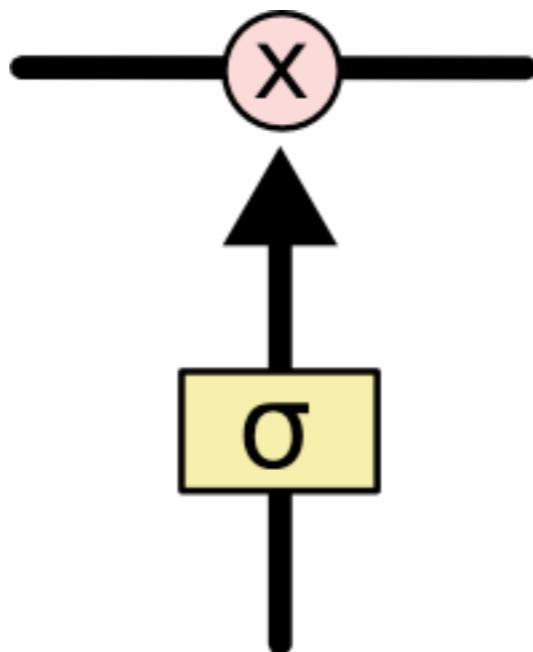
Long Short-Term Memory (LSTM)

- The key to LSTMs is the **cell state** C_t which can thought of as the “long term memory”
 - runs straight down the entire block with only some minor linear interactions
 - allows for information to just flow along it unchanged



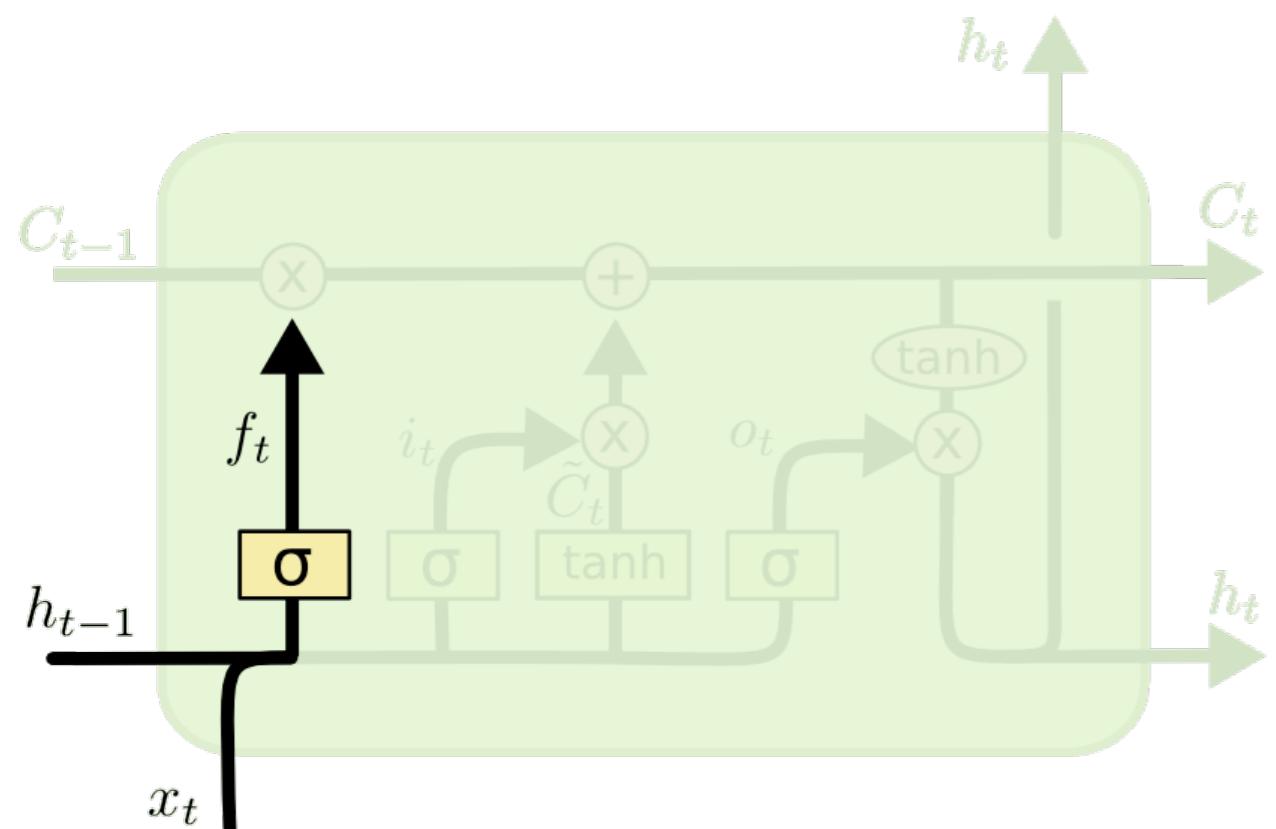
Long Short-Term Memory (LSTM)

- The information that are passed or not to the cell state are controlled by the **gates**
- Gates remove or add information to the cell state through the sigmoid function:
 - a value of zero means “let nothing through”
 - a value of one means “let everything through”
- An LSTM has three of these gates, to protect and control the cell state



Long Short-Term Memory (LSTM)

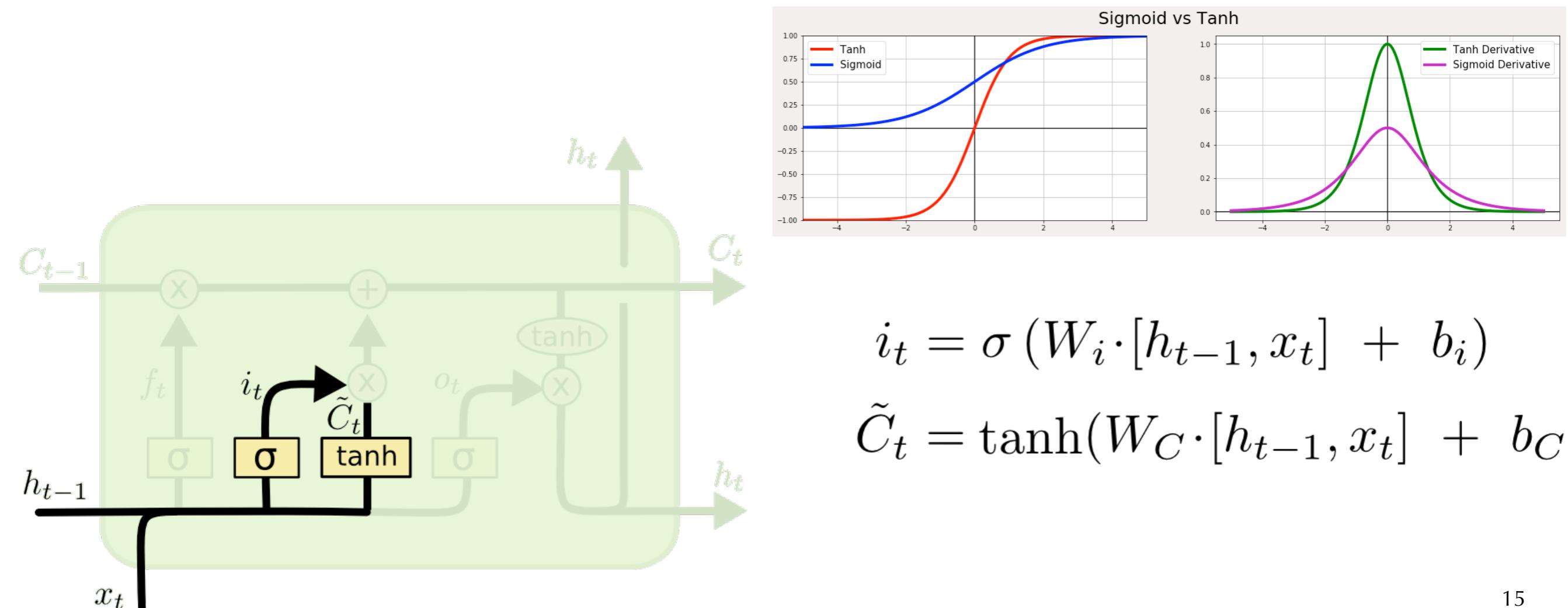
- **Forget gate layer:** decide how much of the past information the cell state should forget
 - allows to selectively update the cell state by forgetting information that is no longer relevant, while retaining information that is important for the current task



$$f_t = \sigma (W_f \cdot [h_{t-1}, x_t] + b_f)$$

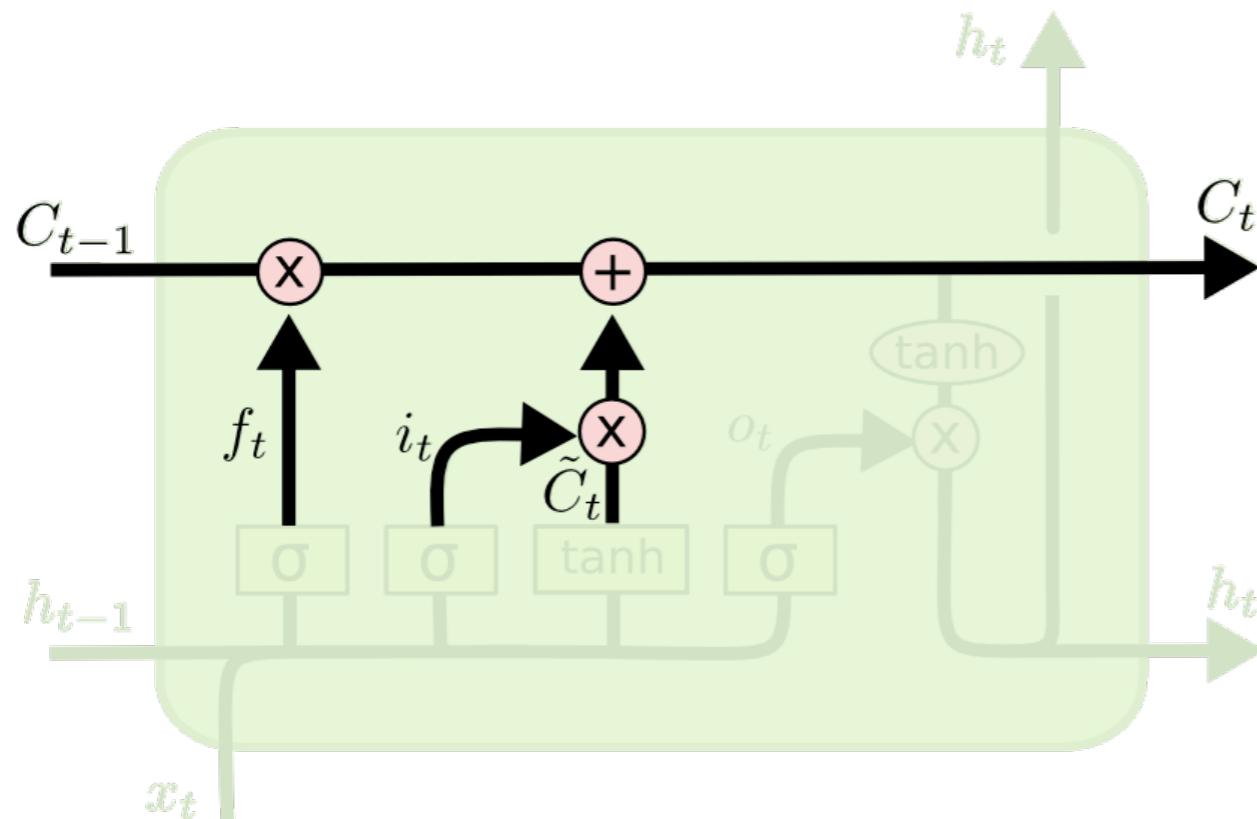
Long Short-Term Memory (LSTM)

- The next step is to decide what new information we are going to store in the cell state
 - the **input gate layer** determines which values from the candidate cell state vector \tilde{C}_t should be added to the current cell state based on the current input and previous hidden state
 - a **candidate cell state vector** \tilde{C}_t is computed with a \tanh layer taking as input the previous hidden state and the current inputs → contains the values that could potentially be added to the cell state if they are deemed relevant by the input gate vector



Long Short-Term Memory (LSTM)

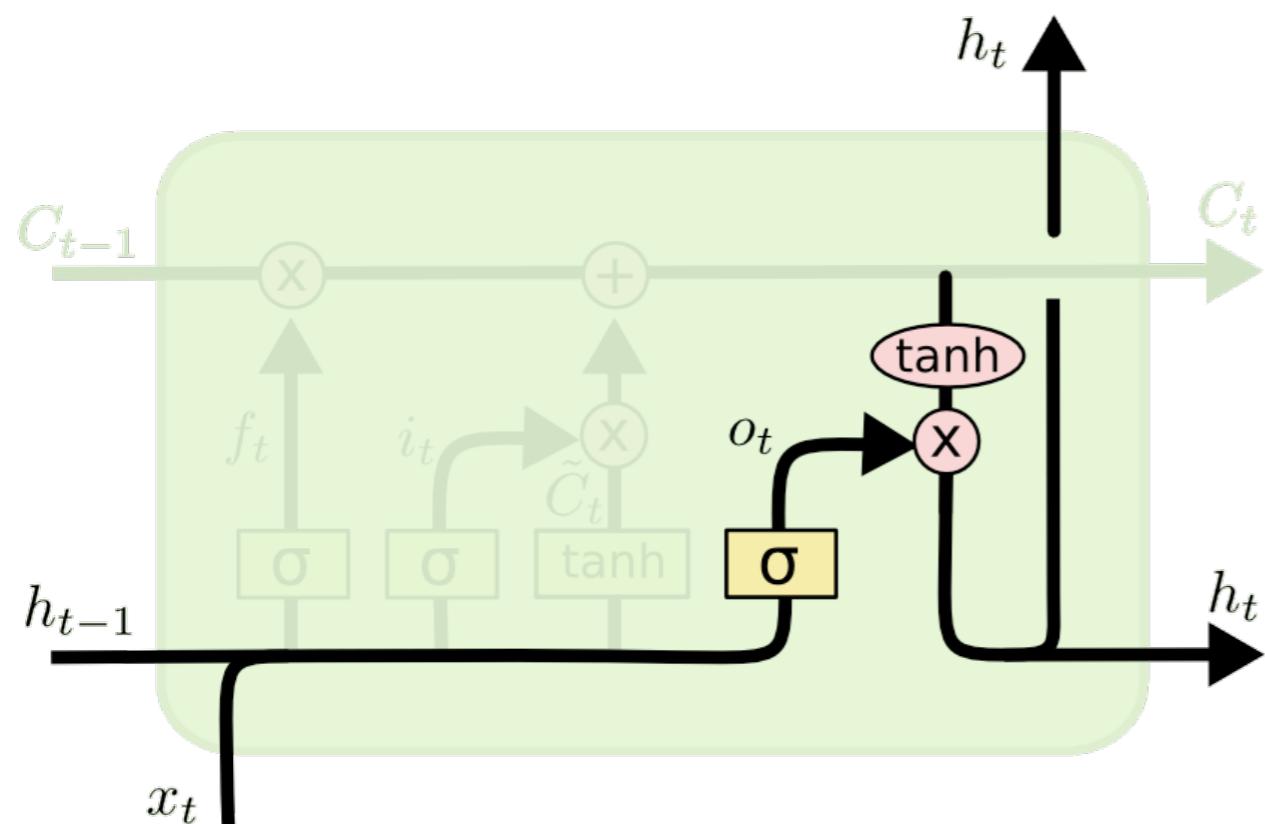
- Now we want to update the old cell state C_{t-1} into the new cell state C_t
 - the previous steps already decided what to do, we just need to actually do it.
- We multiply the old state by f_t — which has forgotten the things we decided to forget earlier
- Then we add $i_t \times \tilde{C}_t$ — this is the new memory update vector (as in standard RNN), scaled by how much we decided to update each state value



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Long Short-Term Memory (LSTM)

- Finally, we need to decide what we are going to expose to the next recurrent unit — the output must be based on the current updated cell state
 - an **output gate** decides which values from the cell state should be output as hidden state at the current time step
 - put the current cell state through \tanh and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to

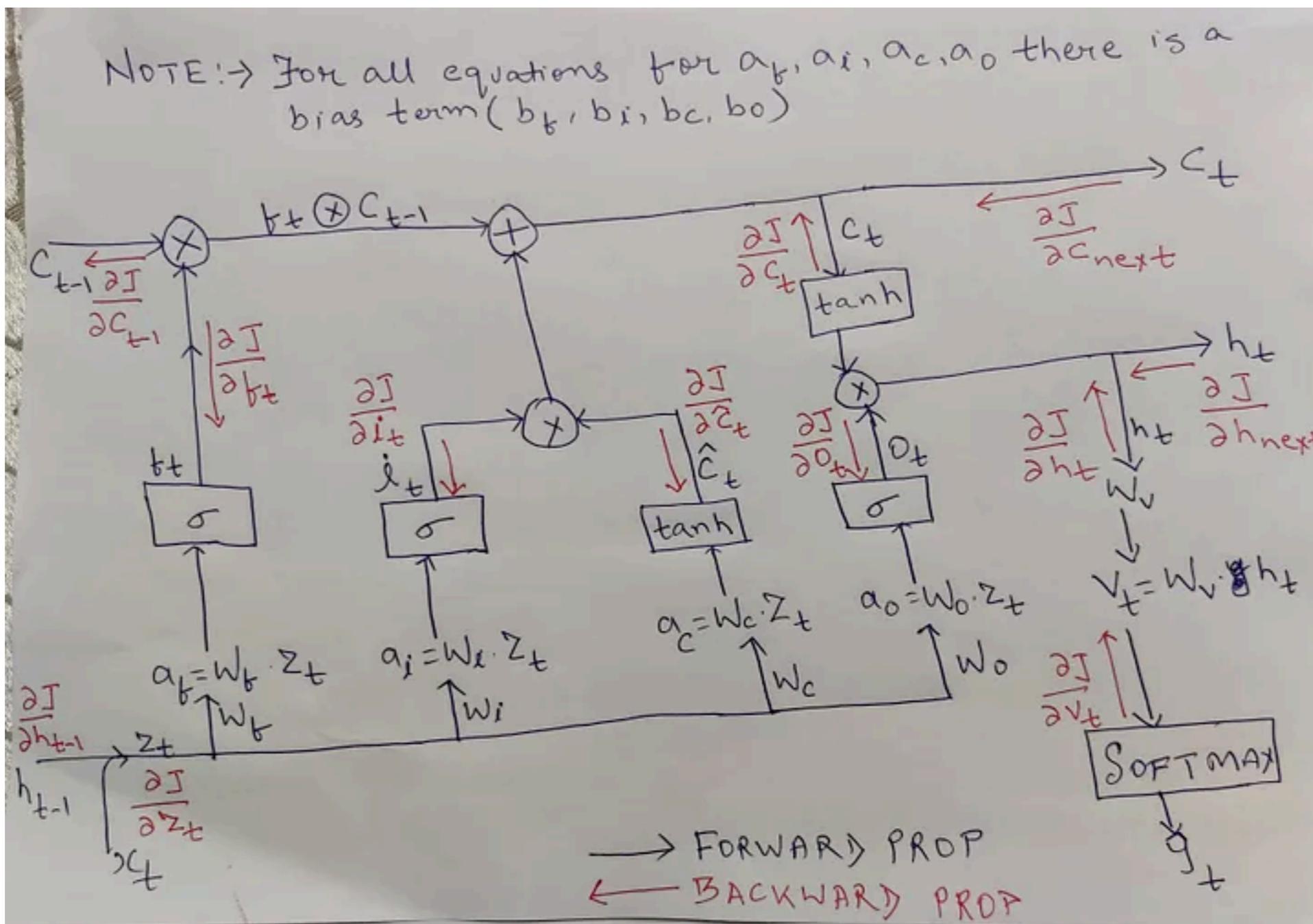


$$o_t = \sigma (W_o [h_{t-1}, x_t] + b_o)$$

$$h_t = o_t * \tanh (C_t)$$

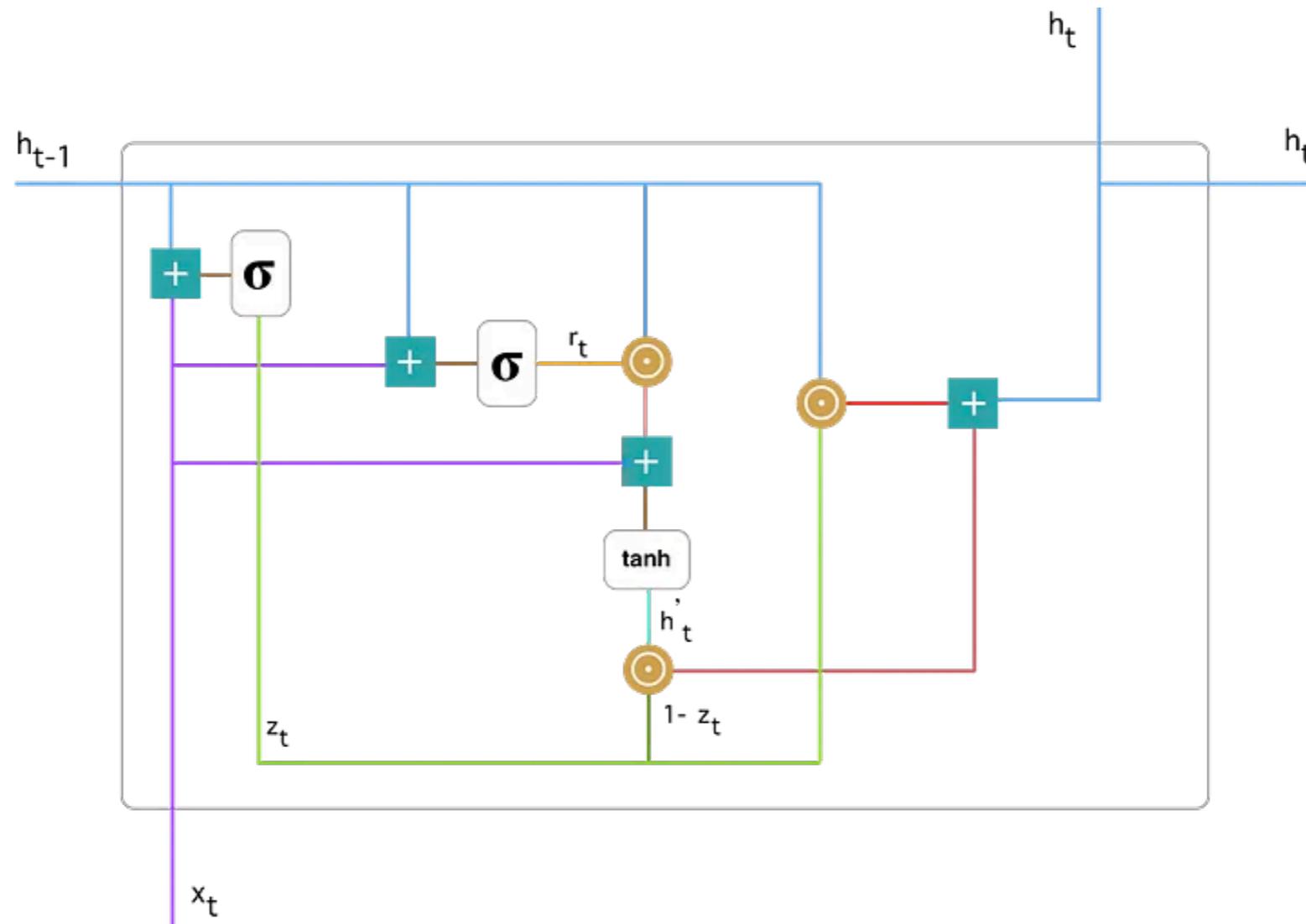
Why do LSTM solve the vanishing gradient problem?

Why do LSTM solve the vanishing gradient problem?



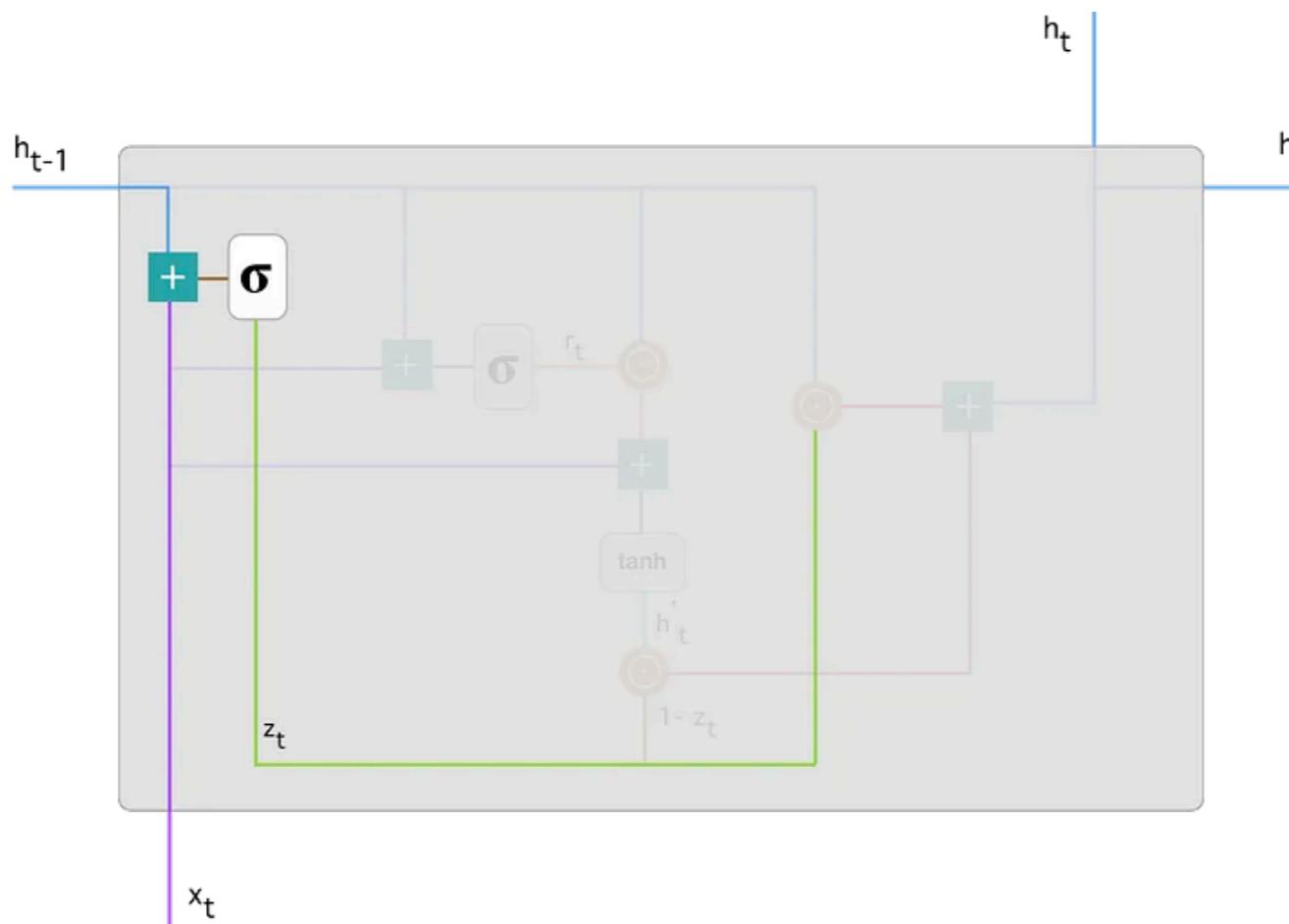
Gated Recurrent Unit (GRU)

- GRUs are another variant of RNNs that solve the vanishing gradient problem
 - younger than LSTM — introduced in 2014 by [Kyunghyun Cho et al.](#)
 - more computationally efficient than LSTM (2 gates instead of 3) but lower accuracy



Gated Recurrent Unit (GRU)

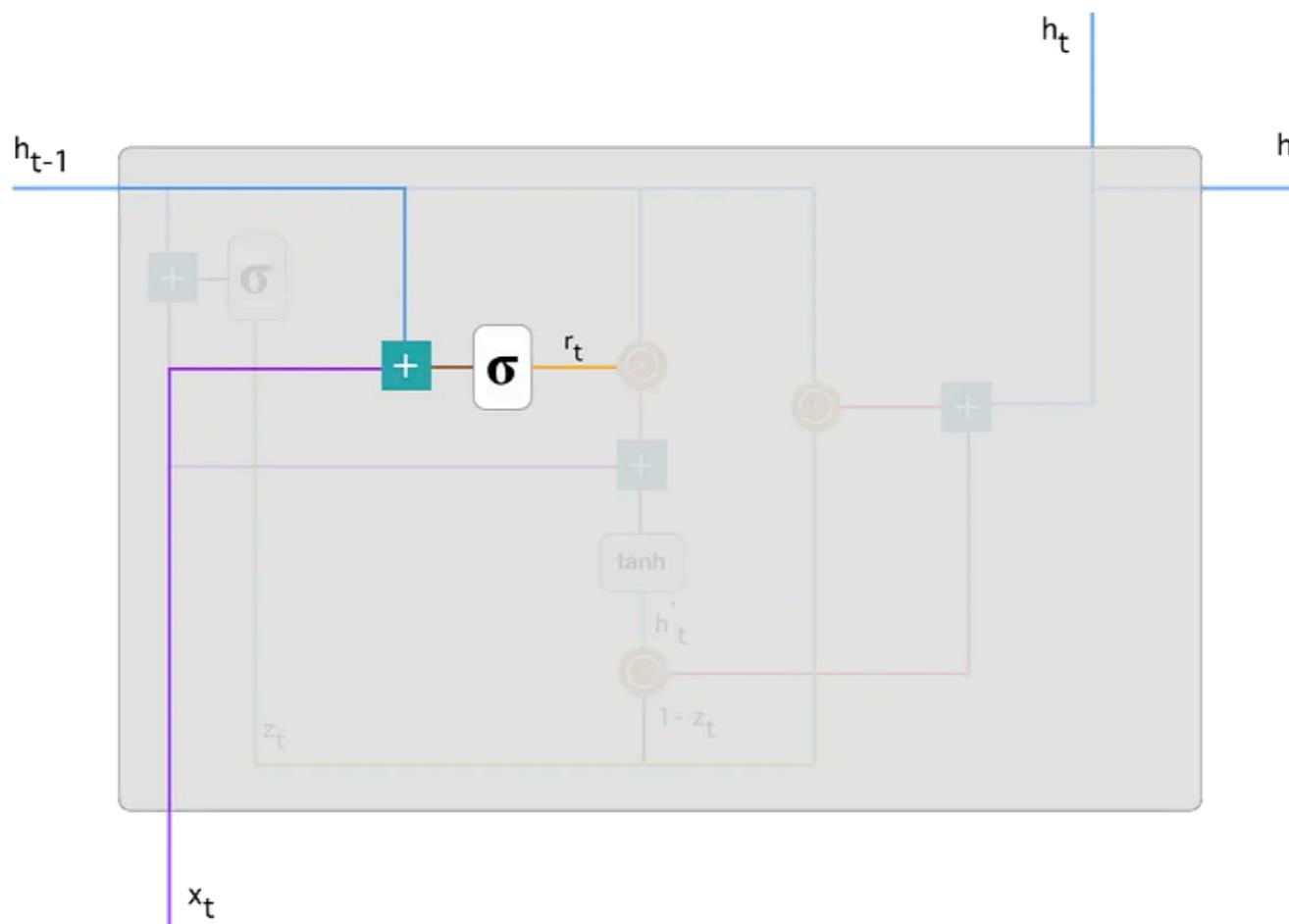
- **Update gate** controls how much of the past information (from previous hidden states) should be retained



$$z_t = \sigma(W^{(z)}x_t + U^{(z)}h_{t-1})$$

Gated Recurrent Unit (GRU)

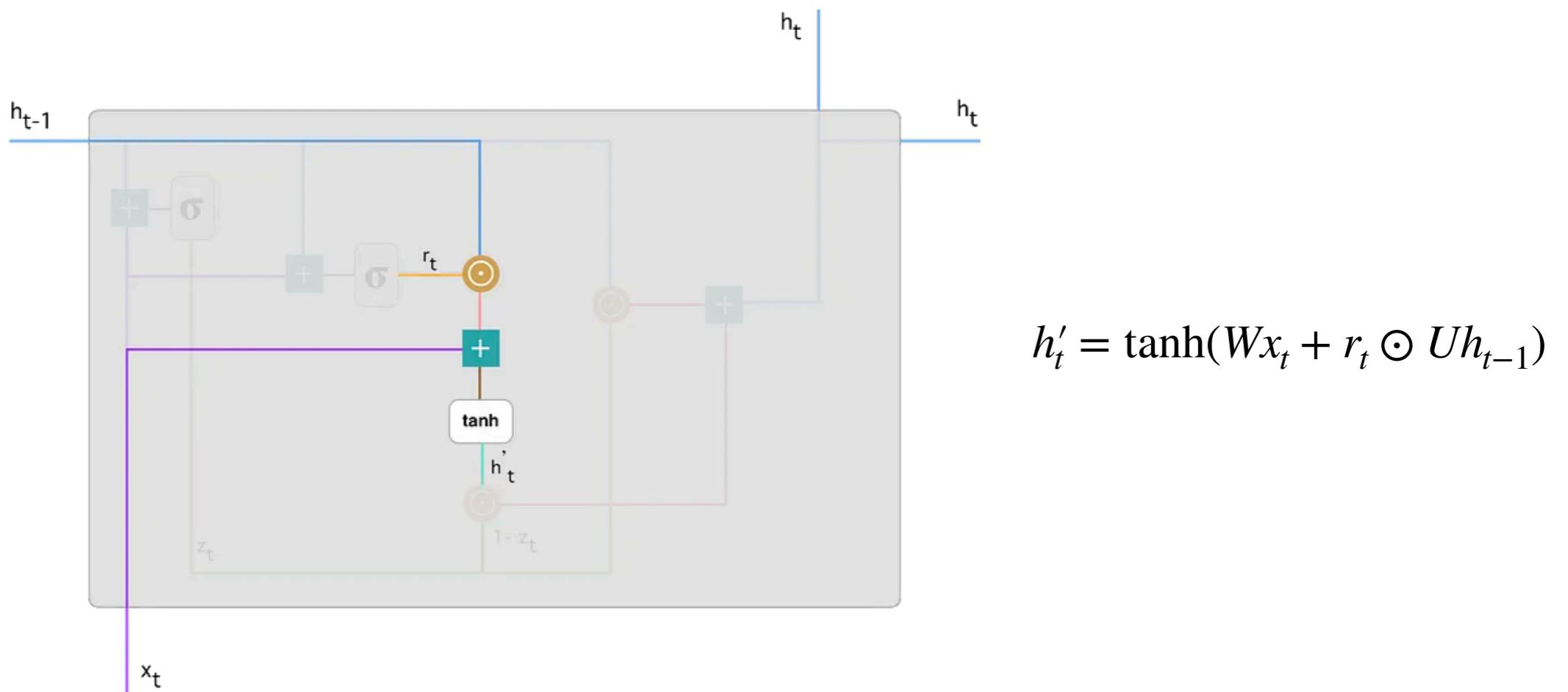
- **Reset gate** determines how much of the previous hidden state should be forgotten and how much of the candidate activation should be used to compute the updated hidden state



$$r_t = \sigma(W^{(r)}x_t + U^{(r)}h_{t-1})$$

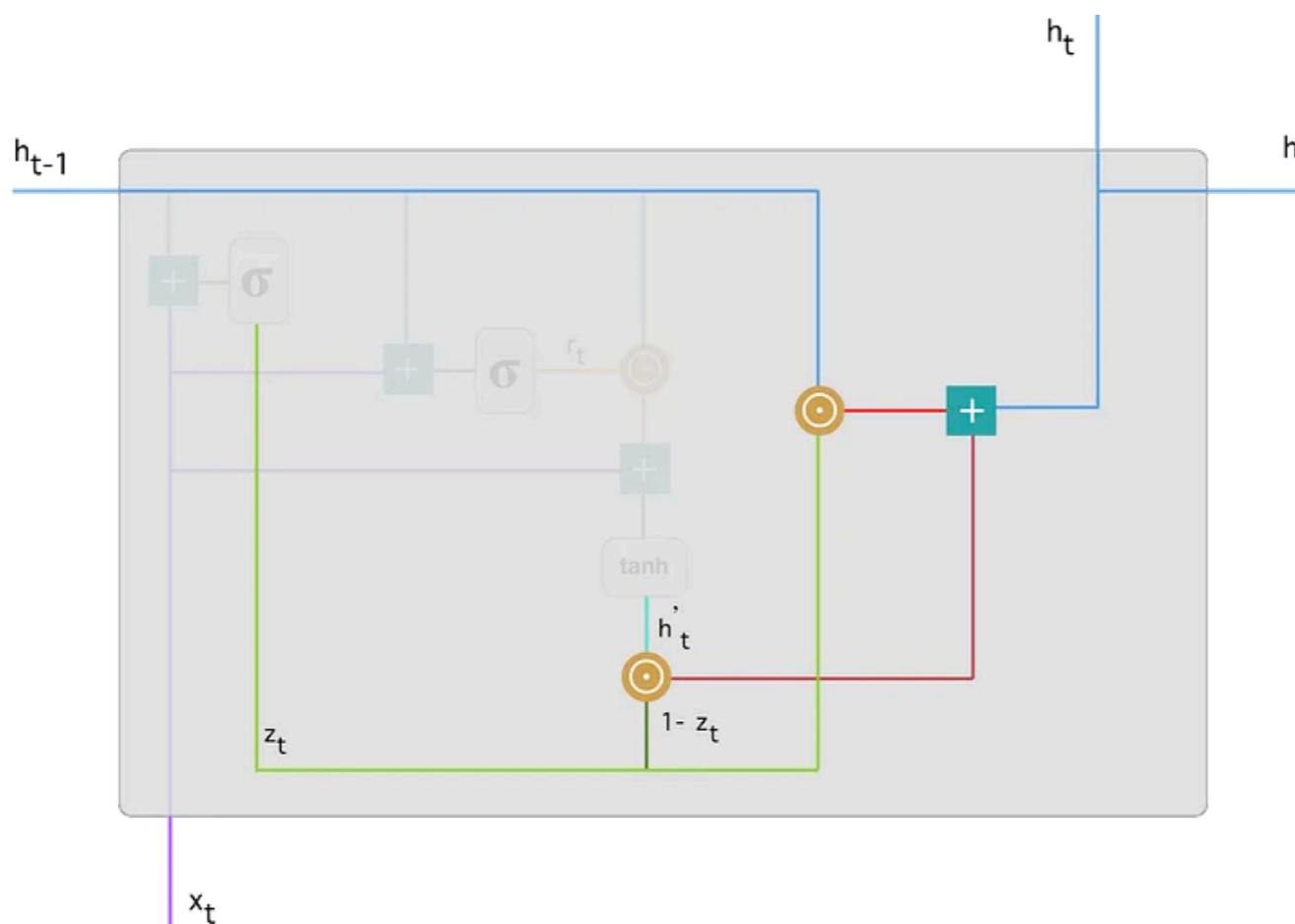
Gated Recurrent Unit (GRU)

- The **candidate activation** h'_t is computed using the reset gate to store the relevant information from the past



Gated Recurrent Unit (GRU)

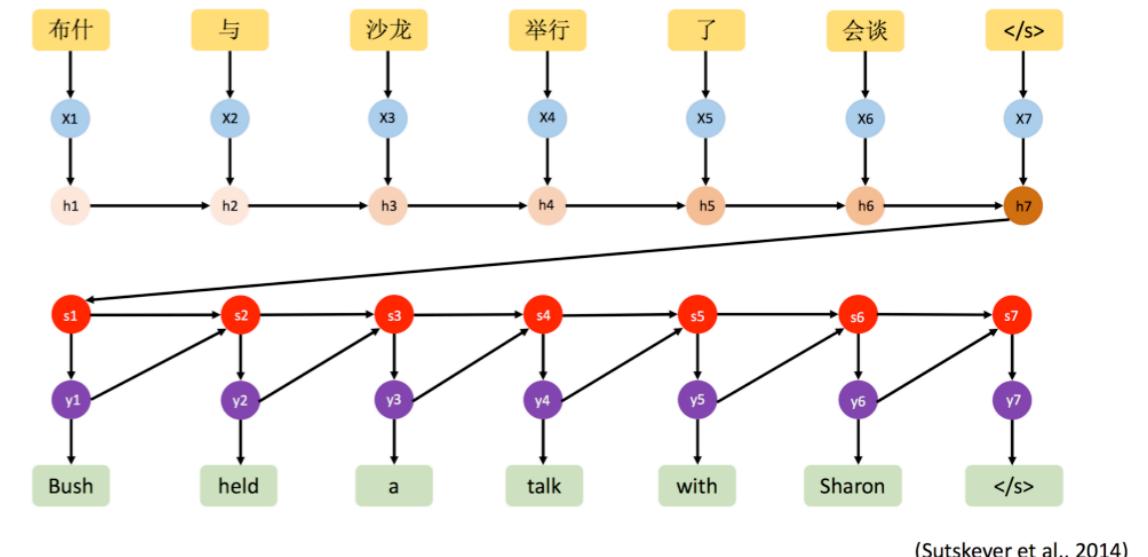
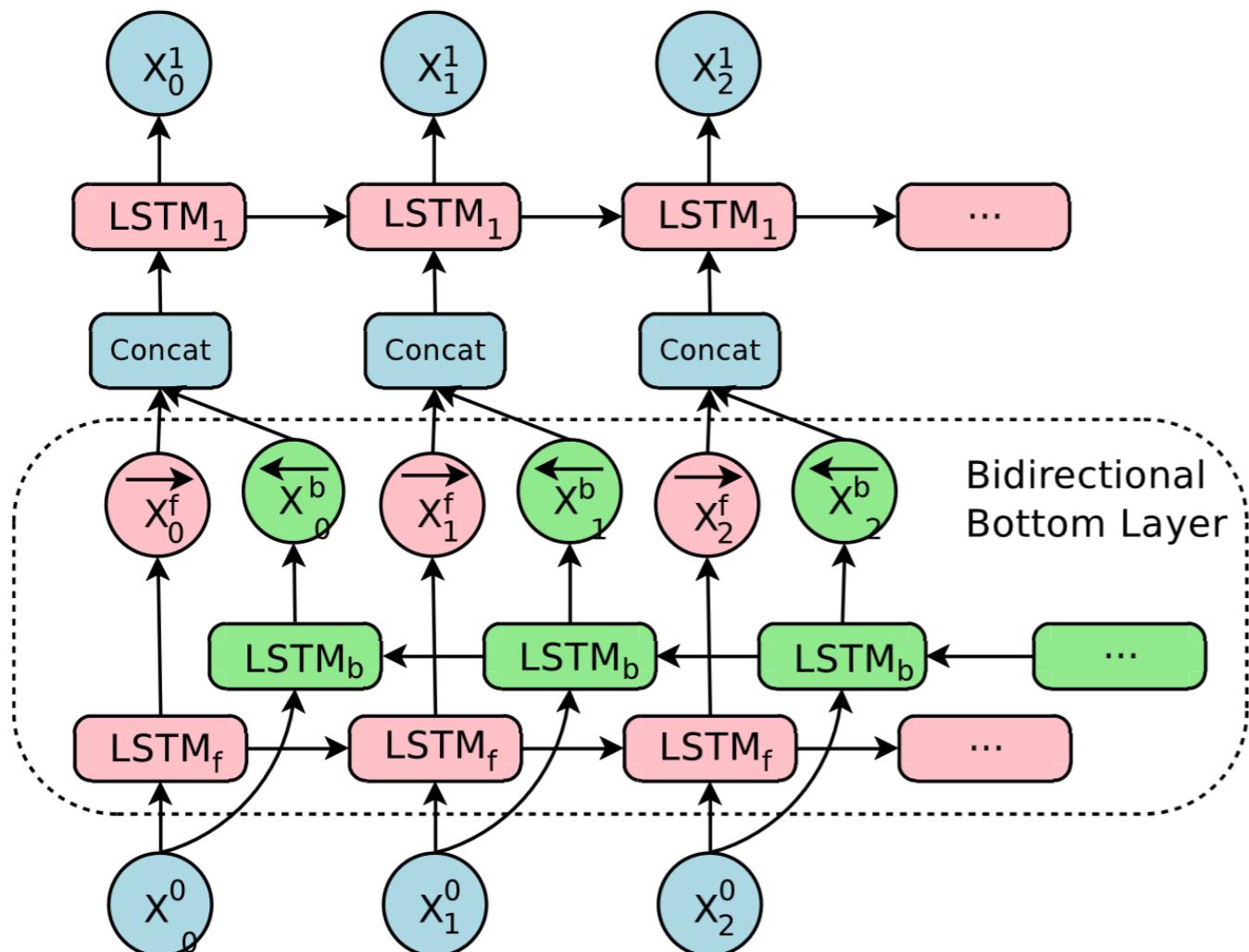
- The **new hidden state** h_t — the vector which holds past and current information and passes it down to the network — is computed with the update gate which decides how much of the candidate activation should be added to the hidden state



$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot h'_t$$

Applications of RNNs

Neural Machine Translation
ex, *Google Translate model*



(Sutskever et al., 2014)

Applications of RNNs

Speech Recognition

ex, Alexa or Siri

The other way around is text-to-speech:
[see Shen et al, 2017](#)

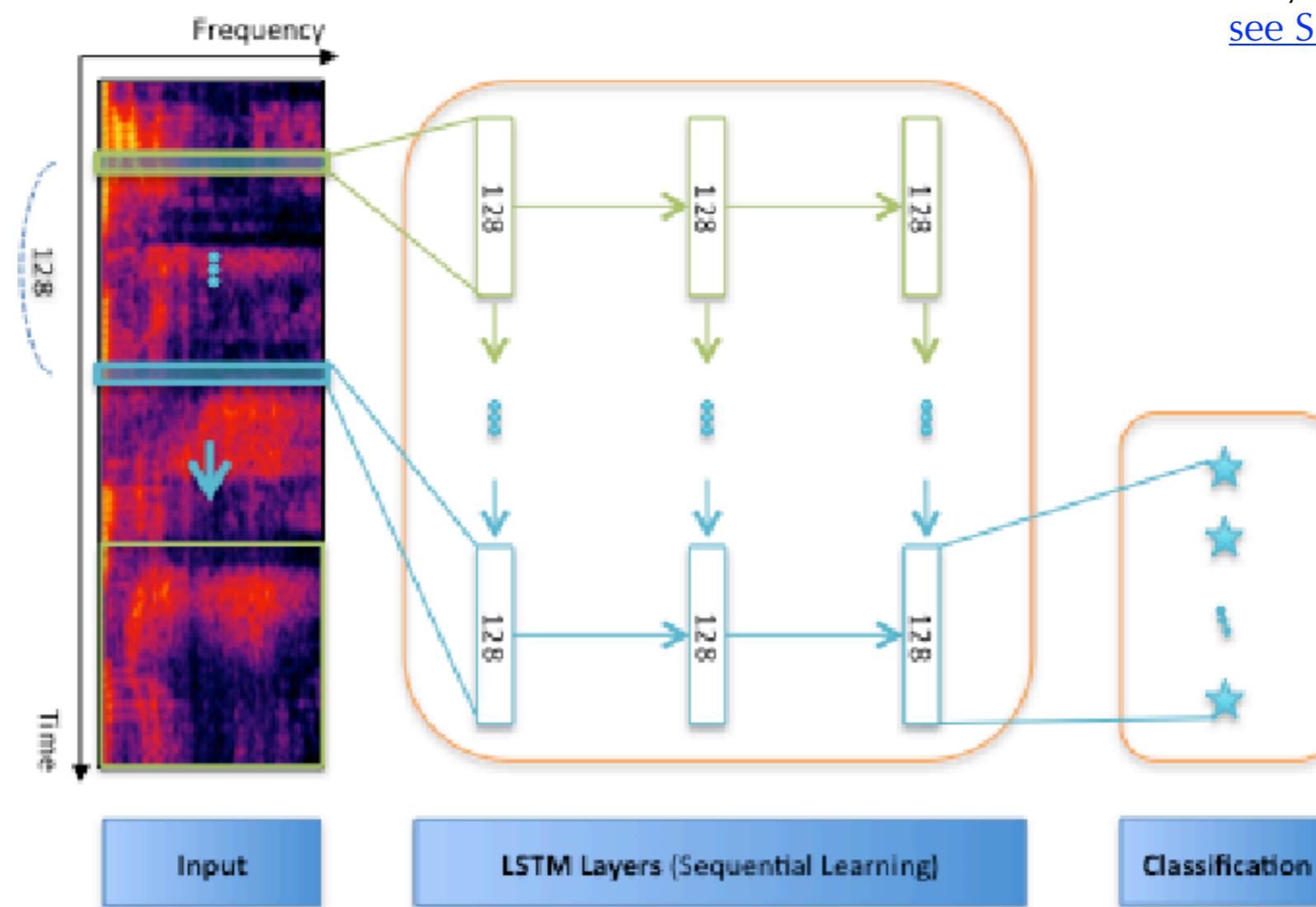
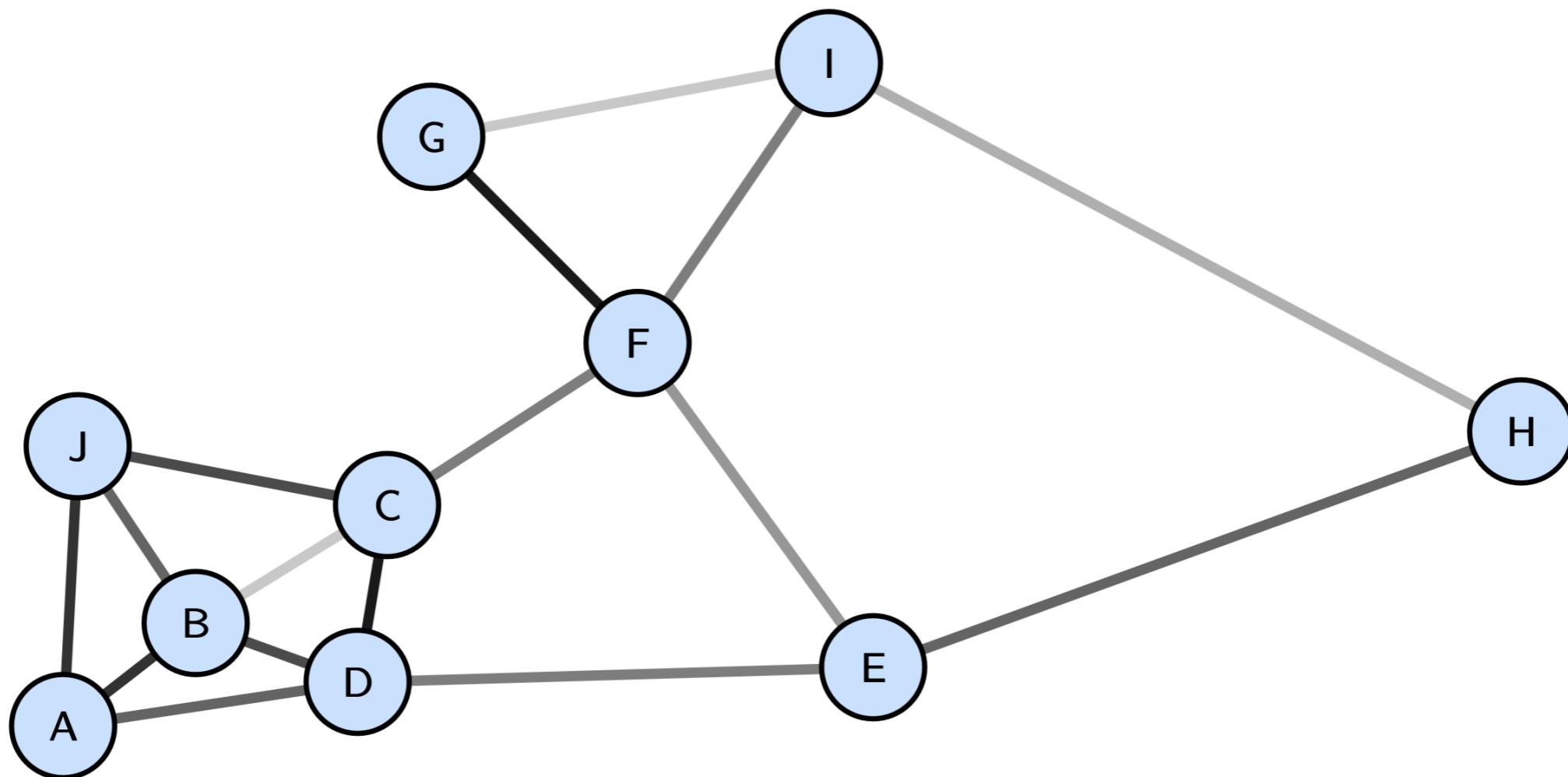


Fig. 4. The proposed LSTM structure for speech emotion recognition

Graph Neural Networks

Graph data representation

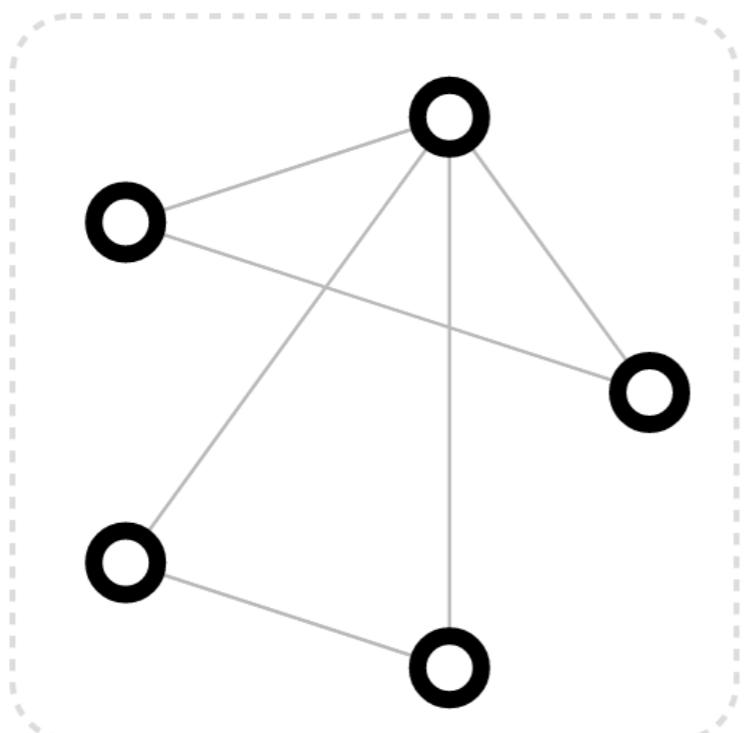
- Data discussed so far had a fixed-size and/or regular-structured connection which we designed appropriate NN architectures to operate on
- What if data have more complex dependencies — i.e. they are connected over a more abstract not-fixed dimension
- **Let's represent the data as a graph and let a neural network learn the actual dimensions that connect the points**



What is a graph?

- A graph $\mathcal{G} = (\mathbf{g}, \mathcal{V}, \mathcal{E})$ represents relational data

- Entities → **Nodes** $v \in \mathcal{V}$ with features $\mathbf{x}_v \in \mathbb{R}^{d_{\mathcal{V}}}$



V Vertex (or node) attributes

e.g., node identity, number of neighbors

E Edge (or link) attributes and directions

e.g., edge identity, edge weight

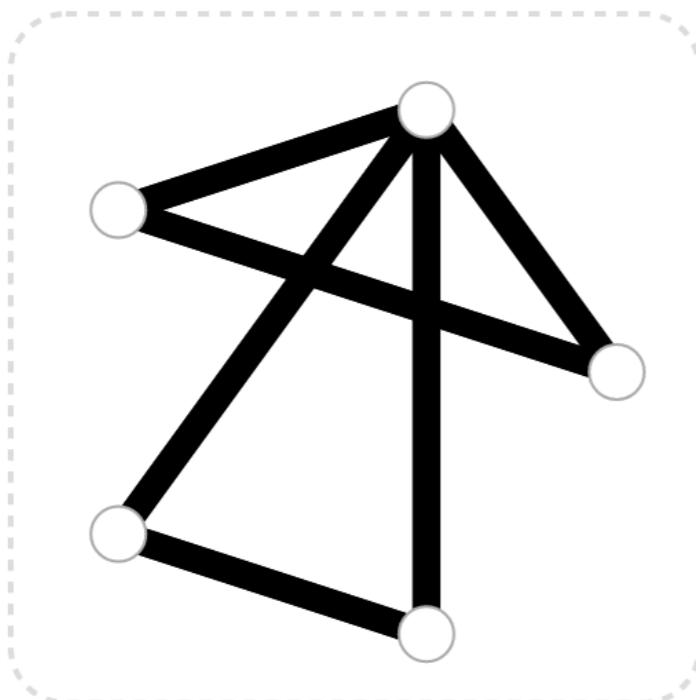
U Global (or master node) attributes

e.g., number of nodes, longest path

What is a graph?

- A graph $\mathcal{G} = (\mathbf{g}, \mathcal{V}, \mathcal{E})$ represents relational data
 - Entities → **Nodes** $v \in \mathcal{V}$ with features $\mathbf{x}_v \in \mathbb{R}^{d_{\mathcal{V}}}$
 - Relations → **Edges** $(u, v) \in \mathcal{E}$ with features $\mathbf{e}_{uv} \in \mathbb{R}^{d_{\mathcal{E}}}$

*Edges define the pairs of nodes that exchange information
Information coming from multiple edges must be “aggregated” at the source node*

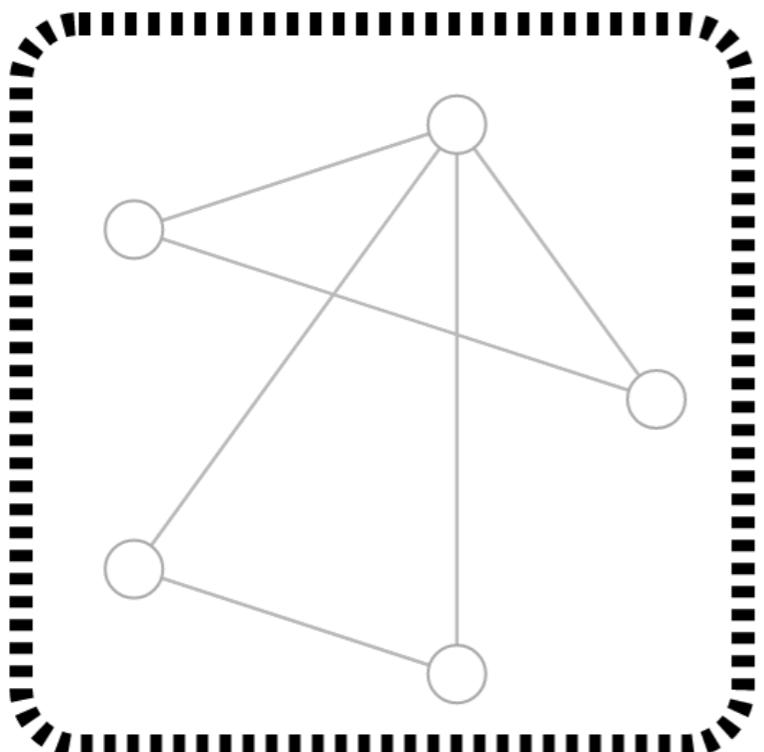


- V** Vertex (or node) attributes
 - e.g., node identity, number of neighbors
- E** Edge (or link) attributes and directions
 - e.g., edge identity, edge weight
- U** Global (or master node) attributes
 - e.g., number of nodes, longest path

What is a graph?

- A graph $\mathcal{G} = (\mathbf{g}, \mathcal{V}, \mathcal{E})$ represents relational data

- Entities → **Nodes** $v \in \mathcal{V}$ with features $\mathbf{x}_v \in \mathbb{R}^{d_{\mathcal{V}}}$
- Relations → **Edges** $(u, v) \in \mathcal{E}$ with features $\mathbf{e}_{uv} \in \mathbb{R}^{d_{\mathcal{E}}}$
- Global graph features $\mathbf{g} \in \mathbb{R}^d$



- V** Vertex (or node) attributes
 - e.g., node identity, number of neighbors
- E** Edge (or link) attributes and directions
 - e.g., edge identity, edge weight
- U** Global (or master node) attributes
 - e.g., number of nodes, longest path

What is a graph?

We can additionally specialize graphs by associating directionality to edges:

Undirected edge



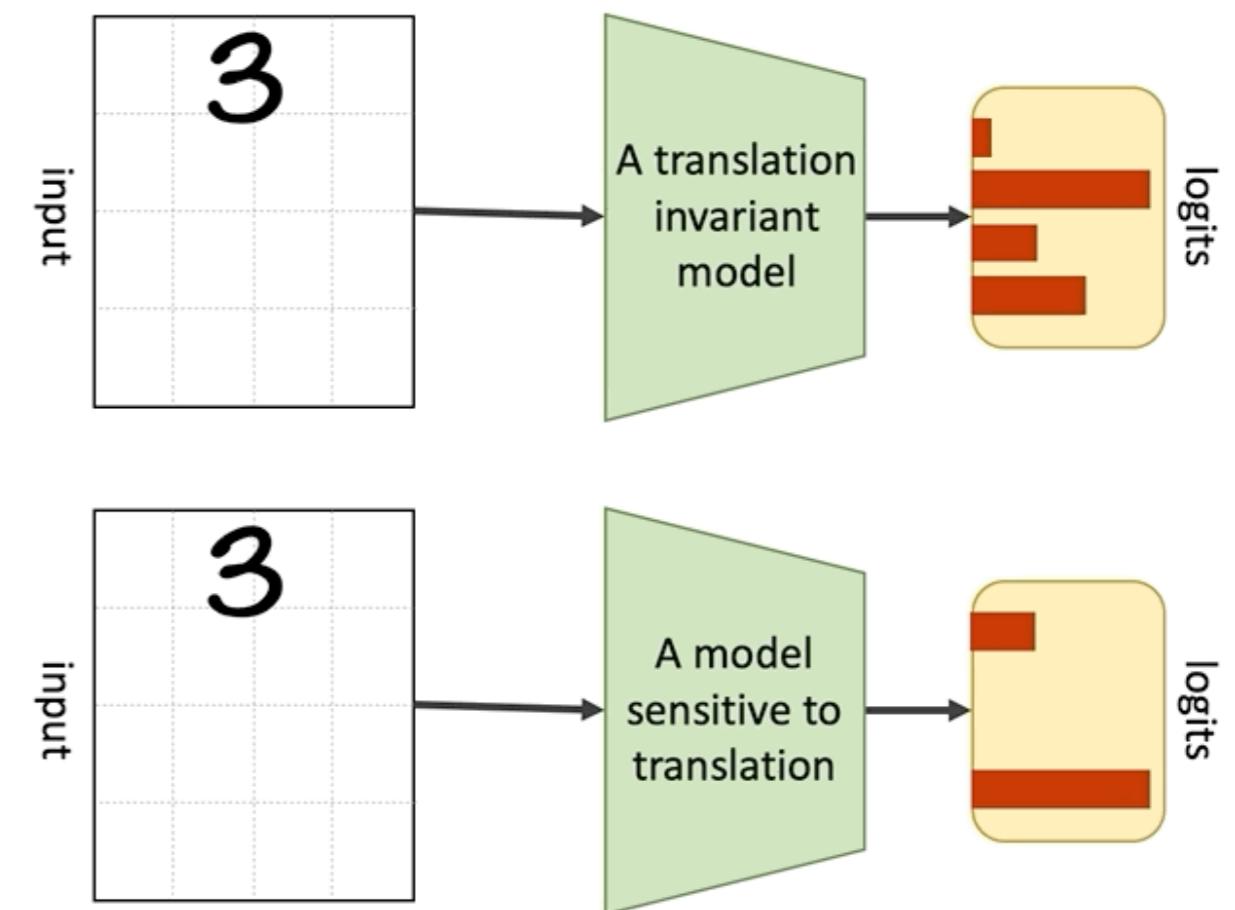
Directed edge



The edges can be directed, where an edge e has a source node, v_{src} , and a destination node v_{dst} . In this case, information flows from v_{src} to v_{dst} . They can also be undirected, where there is no notion of source or destination nodes, and information flows both directions. Note that having a single undirected edge is equivalent to having one directed edge from v_{src} to v_{dst} , and another directed edge from v_{dst} to v_{src} .

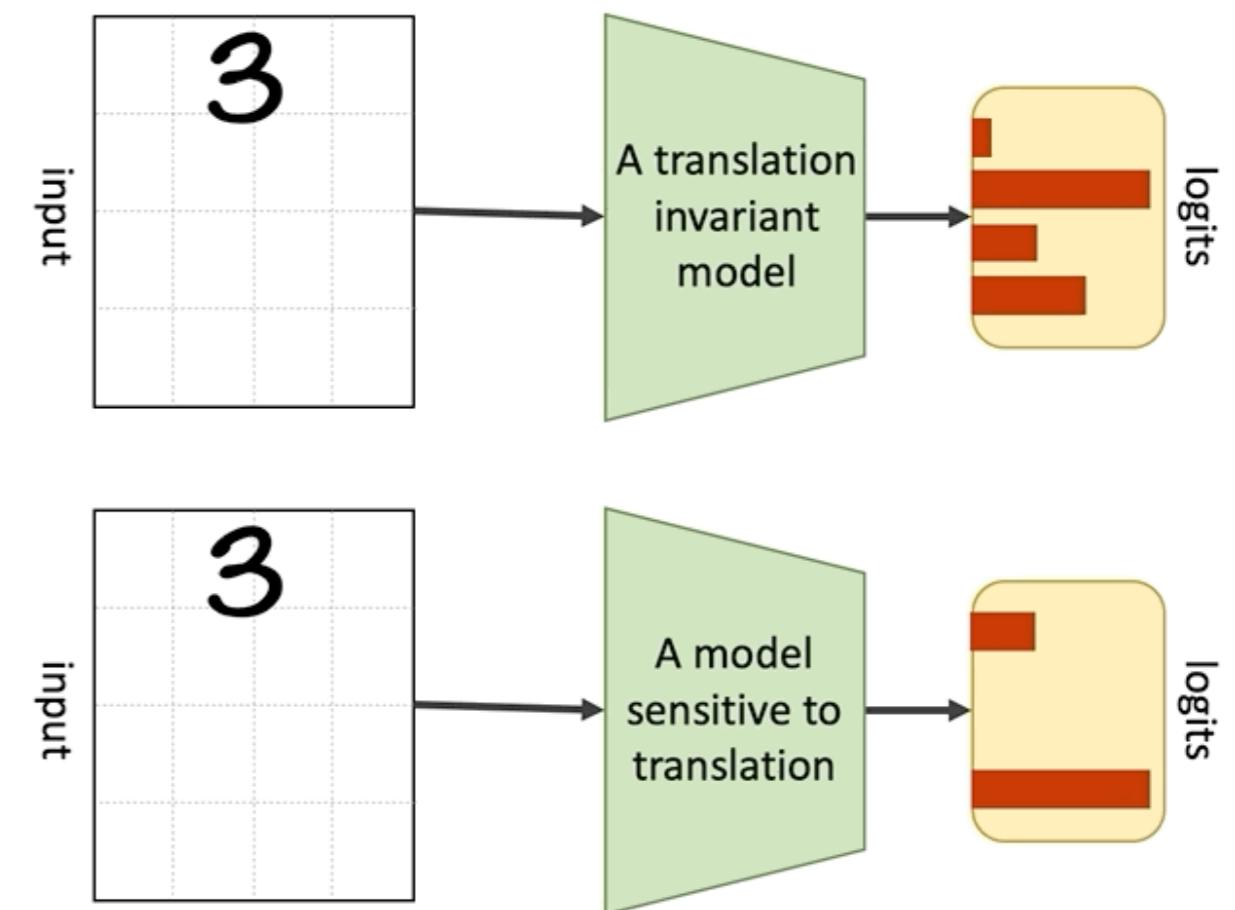
Symmetries

- We have seen that convolutional neural networks respect translational invariance
- Patterns are interesting irrespective of where they are in the image
- **Locality:** neighbouring pixels relate much more strongly than distant ones
- What about arbitrary graphs?



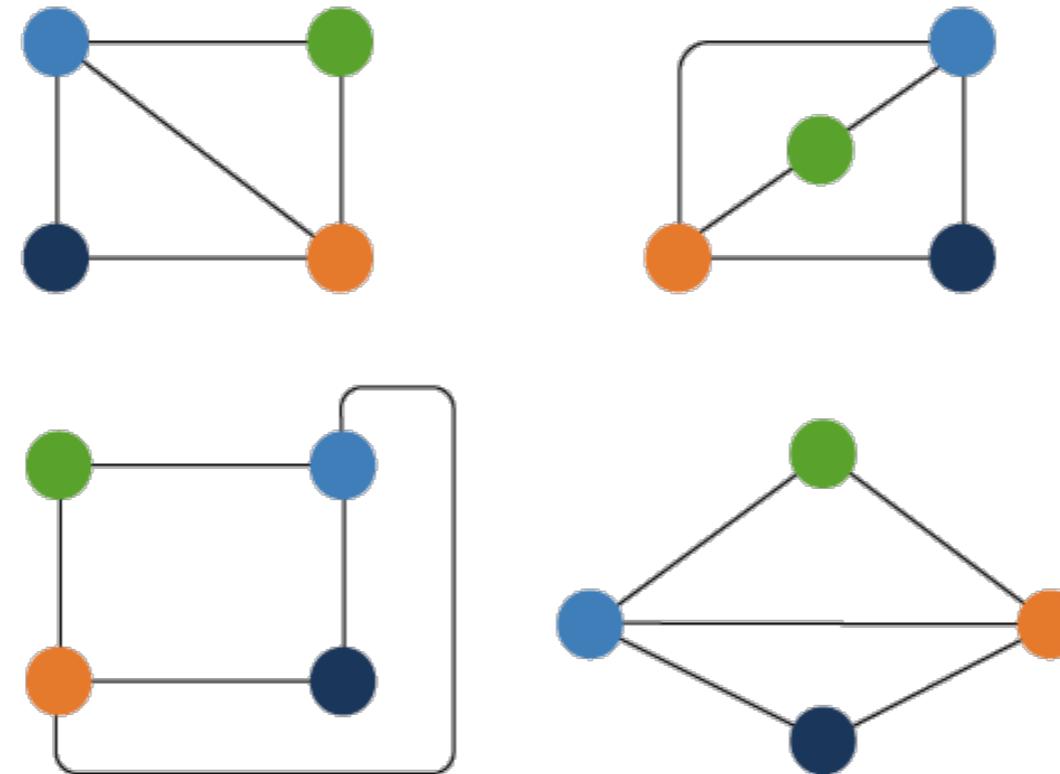
Symmetries

- We have seen that convolutional neural networks respect translational invariance
- Patterns are interesting irrespective of where they are in the image
- **Locality:** neighbouring pixels relate much more strongly than distant ones
- What about arbitrary graphs?



Symmetries

- The nodes of a graph are not assumed to be in any order
- That is, we would like to get the same results for two isomorphic graphs



- To see how to enforce this, we'll go through some formalism...

Learning on sets: setup

- For now, assume the graph **has no edges** (e.g. set of nodes \mathcal{V})
- Let $\mathbf{x}_i \in \mathbb{R}^k$ be the features of node i for $i = 1, 2, \dots, n$
- We can stack them into a node feature matrix of shape $n \times k$:

$$\mathbf{X} = (\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n)^T$$

- That is, the i th row of \mathbf{X} corresponds to the feature vector \mathbf{x}_i of node i
- Note that, by doing so, we have specified a **node ordering!**
 - We would like the result of any neural networks to not depend on this

Permutations

- Operations that change the node order are called **permutations** (there are $n!$ of them)
 - e.g., a permutation $(2, 4, 1, 3)$ means $\mathbf{y}_1 \leftarrow \mathbf{x}_2$, $\mathbf{y}_2 \leftarrow \mathbf{x}_4$, $\mathbf{y}_3 \leftarrow \mathbf{x}_1$, $\mathbf{y}_4 \leftarrow \mathbf{x}_3$
- To stay within linear algebra, each permutation defines an $n \times n$ matrix
 - such matrices are called **permutation matrices**
 - they have exactly one 1 in every row and column, and zeros everywhere else
 - their effect when left-multiplied is to permute the rows of \mathbf{X} , as

$$\mathbf{P}_{(2,4,1,3)}\mathbf{X} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \\ \mathbf{x}_4 \end{bmatrix} = \begin{bmatrix} \mathbf{x}_2 \\ \mathbf{x}_4 \\ \mathbf{x}_1 \\ \mathbf{x}_3 \end{bmatrix}$$

Permutation *invariance*

- Eventually we want to solve tasks (e.g., classification) over graph data through a NN
- This means defining a function $f(x)$ over sets that will not depend on the order (which we want to preserve)
- Equivalently, applying a permutation matrix shouldn't modify the result!
- We arrive at a useful notion of **permutation invariance** — we say that $f(\mathbf{X})$ is permutation invariant if, for all permutation matrices \mathbf{P} if

$$f(\mathbf{P}\mathbf{X}) = f(\mathbf{X})$$

- One very generic form is the *Deep Sets* models [Zaheer et al., NeurIPS '17]:

$$f(\mathbf{X}) = \phi\left(\sum_{i \in \mathcal{V}} \psi(\mathbf{x}_i)\right)$$

- where ψ and ϕ are (learnable) functions, e.g. MLPs.
- the **sum** aggregation is critical! (other choices possible, e.g. **max** or **avg**)

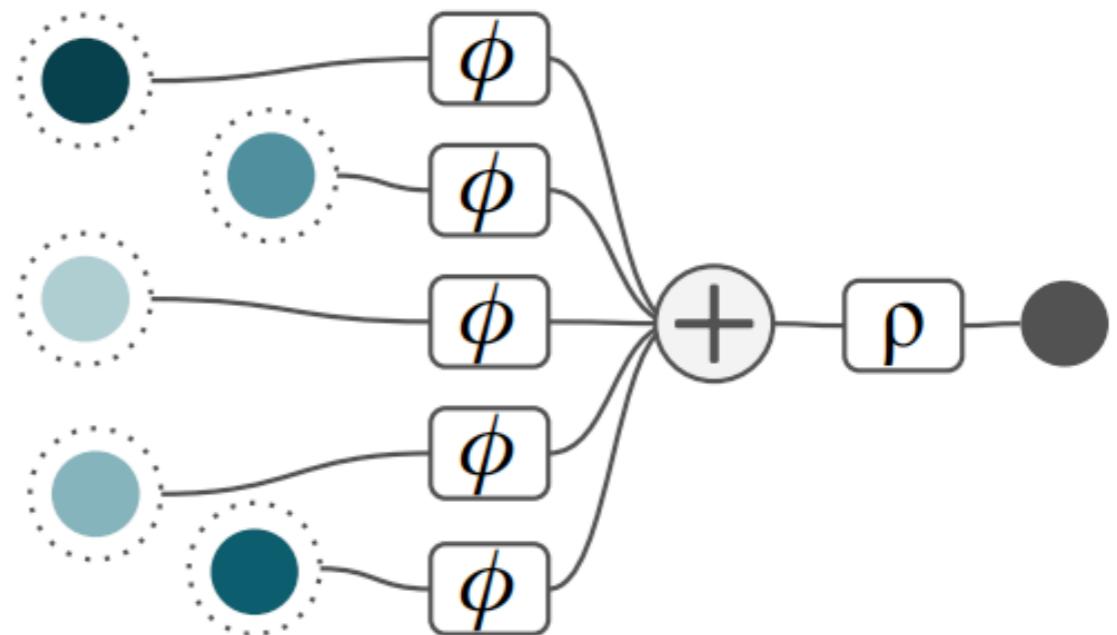
Permutation equivariance

- Permutation-invariant models are the optimal way to obtain graph-level outputs
- What if we would like answers at the **node** level?
- We seek functions that if we permute the nodes, it doesn't matter if we do it before or after the function
- Accordingly we say that $f(\mathbf{X})$ is **permutation equivariant** if, for all permutation matrices \mathbf{P} :

$$f(\mathbf{P}\mathbf{X}) = \mathbf{P}f(\mathbf{X})$$

Permutation symmetries

- The ϕ 's transform each node inputs into a latent vector of lower dimension → this is called “embedding”
- **Permutation invariance:** if I change the order of the nodes the output of the output does not change
 - ensured by having a permutation invariant aggregation function like the **sum**
- **Permutation equivariance =** if I change the order of the nodes the output of ϕ 's does not change
 - ensured by having the same function ϕ per each node embedding (i.e., sharing of weights)



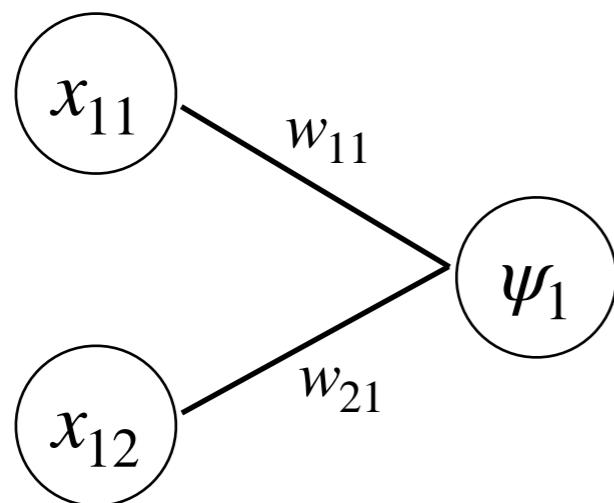
Numerical example

- We have two nodes made of two features:

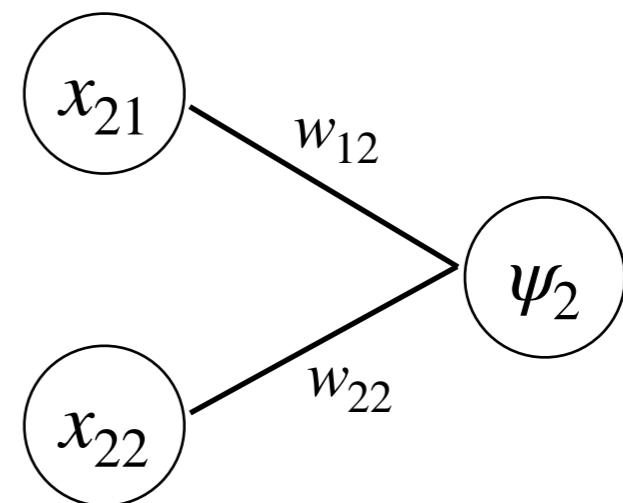
- node 1 has features $x_{11} = 3$ and $x_{12} = 4$
- node 2 has features $x_{21} = 6$ and $x_{22} = 7$

$$f(\mathbf{X}) = \phi\left(\sum_{i \in \mathcal{V}} \psi(\mathbf{x}_i)\right)$$

- The functions ψ_1 and ψ_2 are two distinct MLPs that take as input the two features of the respective node and output two different nodes (= latent representation)



MLP ψ_1 has weights:
 $w_{11} = 1$, $w_{21} = 2$



MLP ψ_2 has weights:
 $w_{12} = 3$, $w_{22} = 4$

Numerical example

- So we now compute the ψ_i functions:

$$\psi_1 = w_{11}x_{11} + w_{12}x_{12} = 1 \times 3 + 2 \times 4 = 11$$

$$\psi_2 = w_{12}x_{21} + w_{22}x_{22} = 3 \times 6 + 4 \times 7 = 46$$

- And the final sum: $\psi_1 + \psi_2 = 11 + 46 = 57$
- If I exchange ψ_1 and ψ_2 the sum does not change ... it would still give $46 + 11 = 57$
- What if exchange the inputs? i.e. node 1 gets the features of node 2:

$$\psi_1 = w_{11}x_{11} + w_{12}x_{12} = 1 \times 6 + 2 \times 7 = 20$$

$$\psi_2 = w_{12}x_{21} + w_{22}x_{22} = 3 \times 3 + 4 \times 4 = 25$$

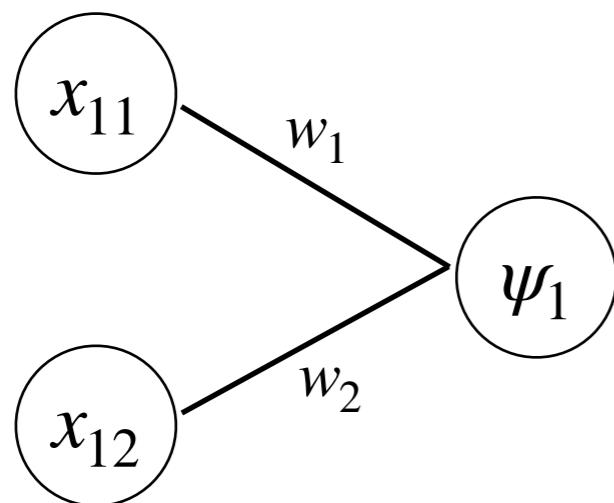
- The final sum is now: $\psi_1 + \psi_2 = 20 + 25 = 45 !!!$
- **Therefore the full function is permutation invariant but not equivariant!**

Numerical example

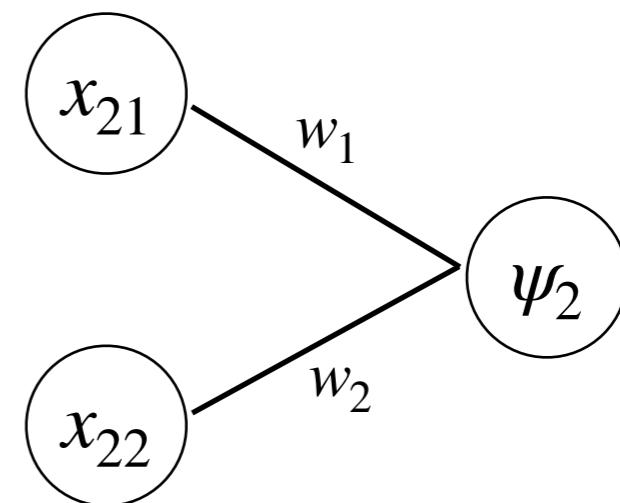
- Now let's instead assume that $\psi_1 = \psi_2$, i.e. they share the same weights:

- $w_{11} = w_{12} = w_1 = 3$

- $w_{21} = w_{22} = w_2 = 4$



MLP ψ_1 has weights:
 $w_1 = 3$, $w_2 = 4$



MLP ψ_2 has weights:
 $w_1 = 3$, $w_2 = 4$

Numerical example

- So we now compute the ψ_i functions:

$$\psi_1 = w_1 x_{11} + w_2 x_{12} = 3 \times 3 + 4 \times 4 = 25$$

$$\psi_2 = w_1 x_{21} + w_2 x_{22} = 3 \times 6 + 4 \times 7 = 46$$

- And the final sum: $\psi_1 + \psi_2 = 25 + 46 = 71$
- If I exchange ψ_1 and ψ_2 the sum does not change ... it would still give $46 + 25 = 71$
- What if exchange the inputs? i.e. node 1 gets the features of node 2:

$$\psi_1 = w_1 x_{11} + w_2 x_{12} = 3 \times 6 + 4 \times 7 = 46$$

$$\psi_2 = w_1 x_{21} + w_2 x_{22} = 3 \times 3 + 4 \times 4 = 25$$

- The final sum is now: $\psi_1 + \psi_2 = 46 + 25 = 71 !!!$
- **Therefore the full function is permutation invariant and also equivariant!**

Learning on graphs

- Now we augment the set of nodes with edges \mathcal{E} between them
- We can represent these edges with an **adjacency matrix**, \mathbf{A} , such that:

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in \mathcal{E} \\ 0 & \text{otherwise} \end{cases}$$

- Further additions (e.g. edge features) are possible but ignored for simplicity
- Our main desire for permutation {in,equi}variance still holds!

Learning on graphs

- The main difference: node permutations now also accordingly act on the edges
- We need to appropriately permute both rows and columns of \mathbf{A}
 - when applying a permutation matrix \mathbf{P} , this amounts to \mathbf{PAP}^T
- We arrive at updated definitions of suitable functions $f(\mathbf{X}, \mathbf{A})$ over graphs:

Invariance: $f(\mathbf{PX}, \mathbf{PAP}^T) = f(\mathbf{X}, \mathbf{A})$

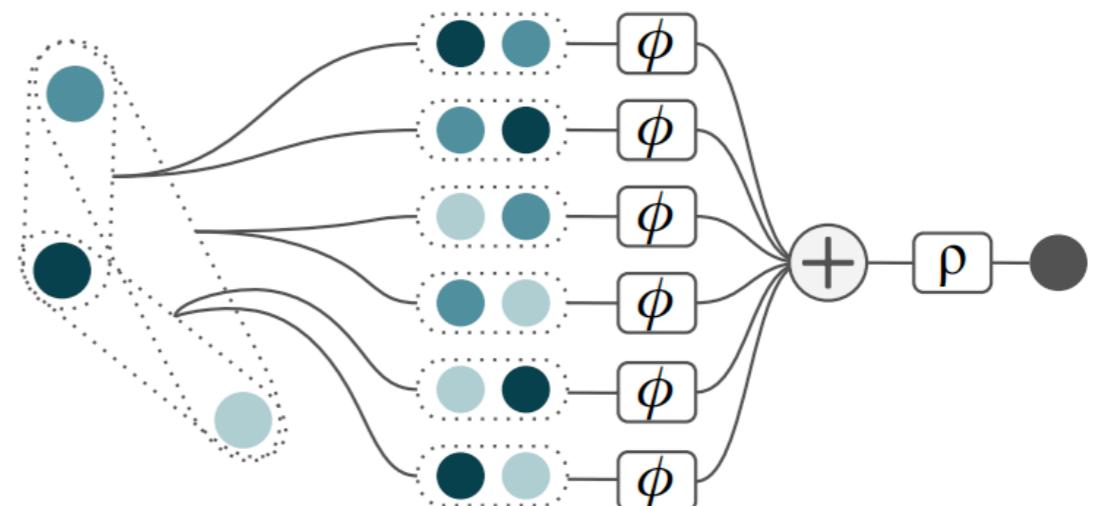
Equivariance: $f(\mathbf{PX}, \mathbf{PAP}^T) = \mathbf{P}f(\mathbf{X}, \mathbf{A})$

Locality on graphs

- On a graph, we enforced equivariance by applying the same function to every node in isolation
- Graphs give us a broader context: a node's **neighbourhood**
 - for a node i , its (1-hop) neighbourhood is commonly defined as follows:
- Accordingly, we can use the multiset of features in the neighbourhood

$$\mathbf{X}_{\mathcal{N}_i} = \{\mathbf{x}_j : j \in \mathcal{N}_i\}$$

- by defining a local function, g ,
as operating over this multiset: $g(\mathbf{x}_i, \mathbf{X}_{\mathcal{N}_i})$



in this example every node is neighbour of each others but they share the same g

Graph Neural Networks

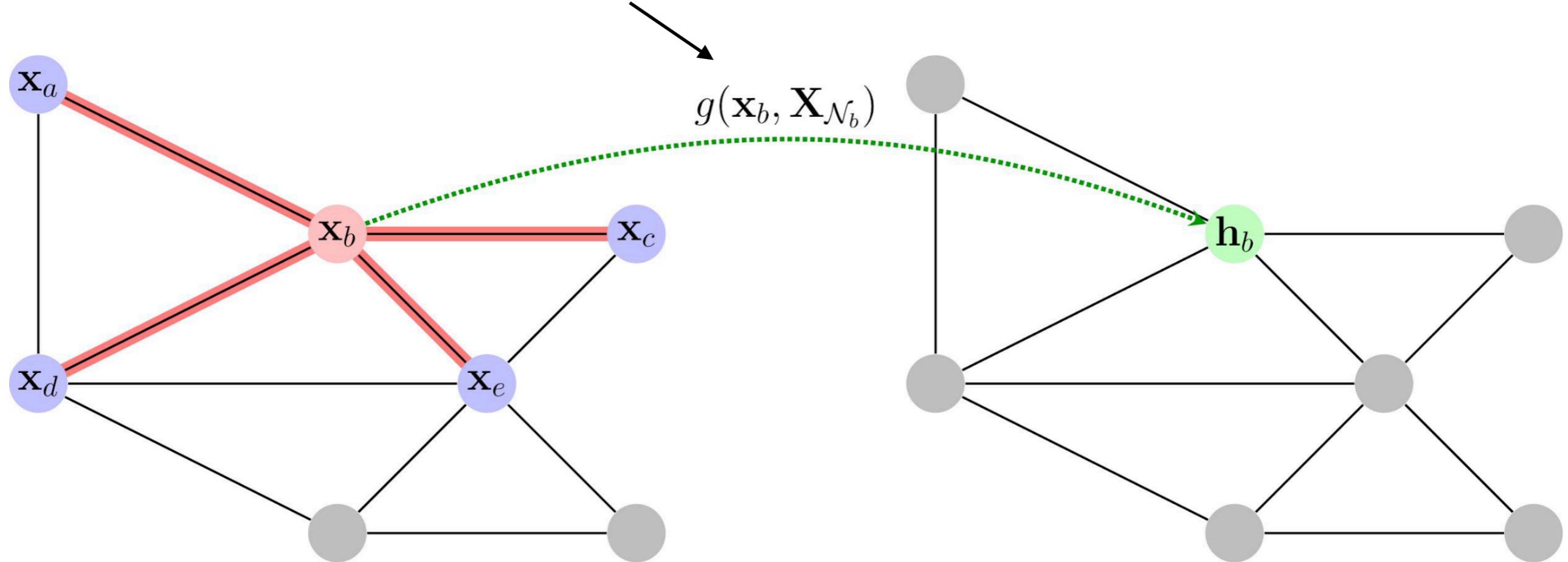
- Now we can construct permutation equivariant functions, $f(\mathbf{X}, \mathbf{A})$, by appropriately applying the same local function, g , over all neighbours

$$f(\mathbf{X}, \mathbf{A}) = \begin{bmatrix} g(\mathbf{x}_1, \mathbf{X}_{\mathcal{N}_1}) \\ g(\mathbf{x}_2, \mathbf{X}_{\mathcal{N}_2}) \\ \vdots \\ g(\mathbf{x}_n, \mathbf{X}_{\mathcal{N}_n}) \end{bmatrix}$$

- To ensure equivariance, we need g to not depend on the order of the vertices in $\mathcal{X}_{\mathcal{N}_i}$
 - Hence, g should be permutation invariant!

GNNs — how does that work?

this is permutation invariant
ensuring equivariance of the full NN

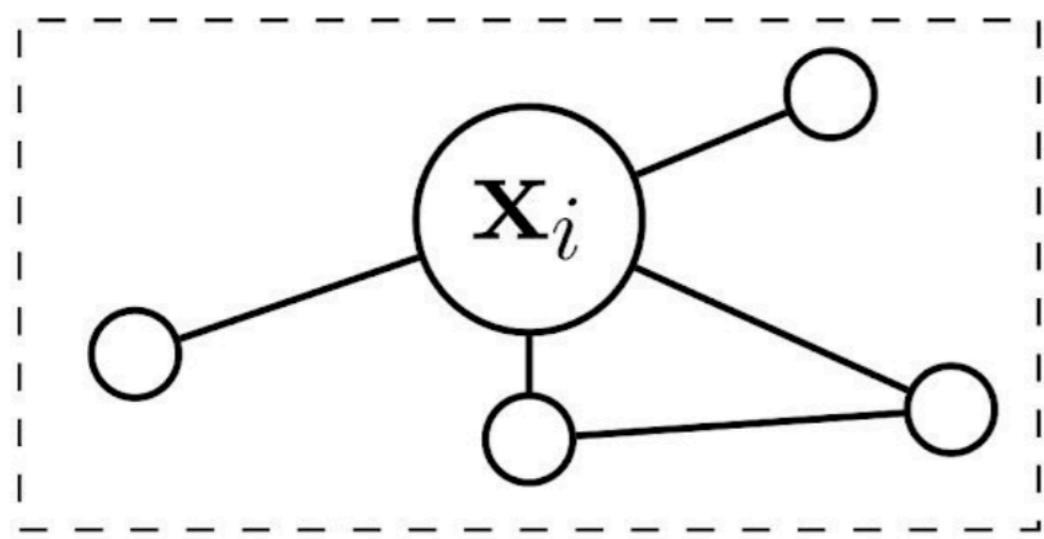


$$\mathbf{X}_{\mathcal{N}_b} = \{\{\mathbf{x}_a, \mathbf{x}_b, \mathbf{x}_c, \mathbf{x}_d, \mathbf{x}_e\}\}$$



How to use GNNs?

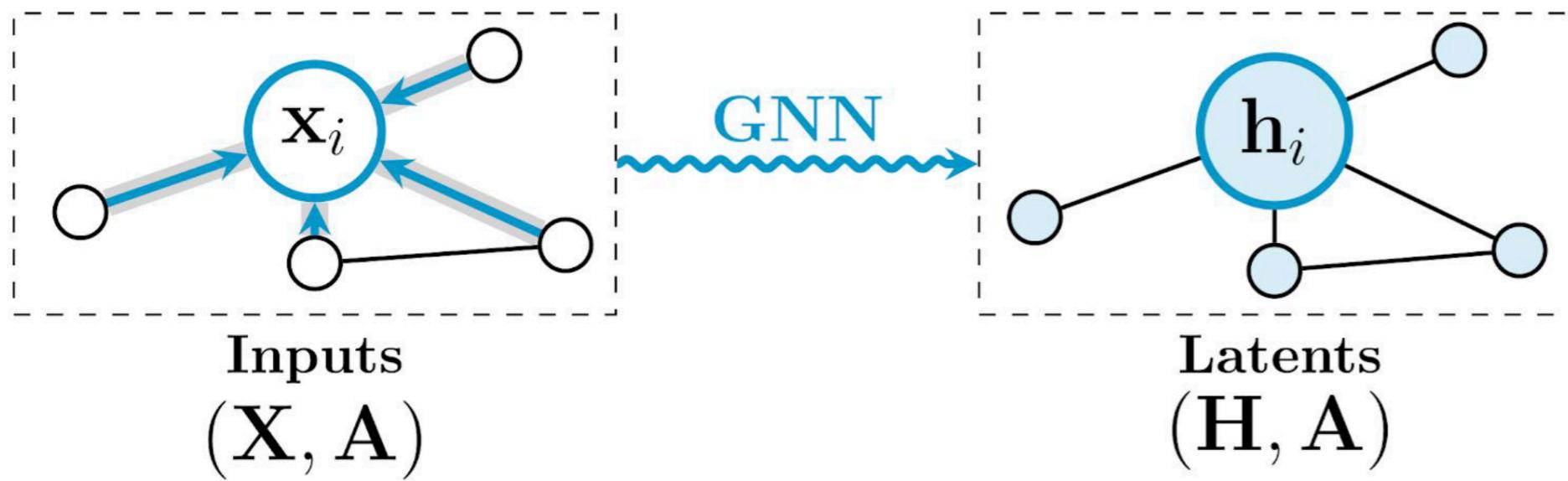
- We start with nodes some of which have connections to others through the adjancency matrix → specified beforehand by edges



Inputs
 (\mathbf{X}, \mathbf{A})

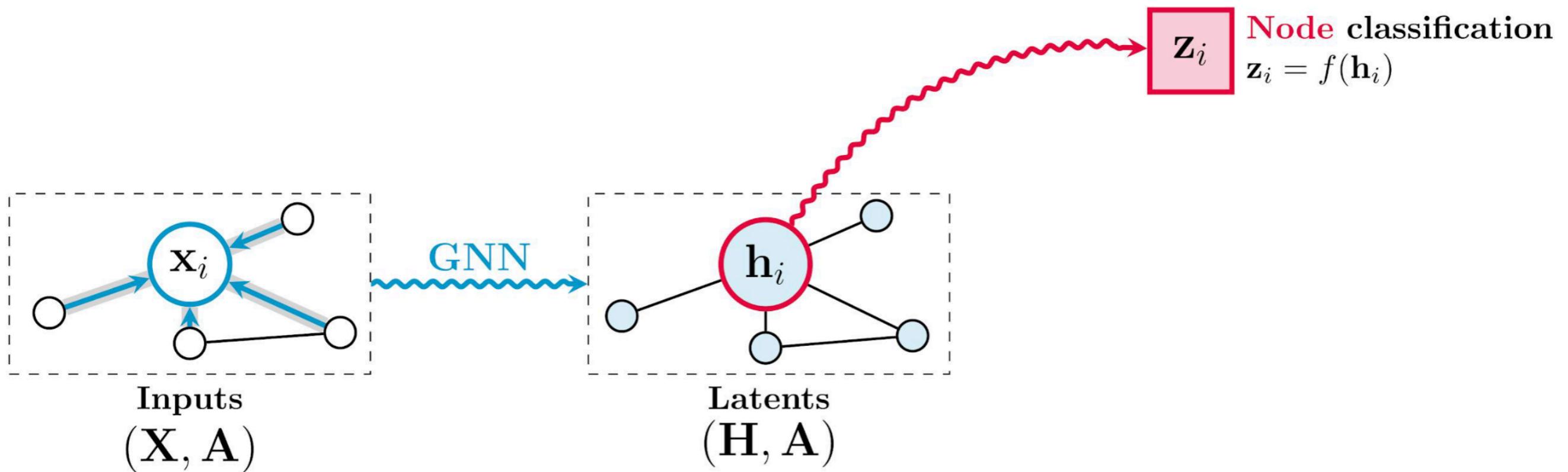
How to use GNNs?

- We start with nodes some of which have connections to others through the adjancency matrix → specified beforehand by edges
- We let the connected nodes (through the edges) to transfer info to the source node \mathbf{x}_i through an MLP (i.e. you learn what info from neighbours are important given \mathbf{x}_i)
 - this creates a new node representation \mathbf{h}_i that takes into account features of neighbours



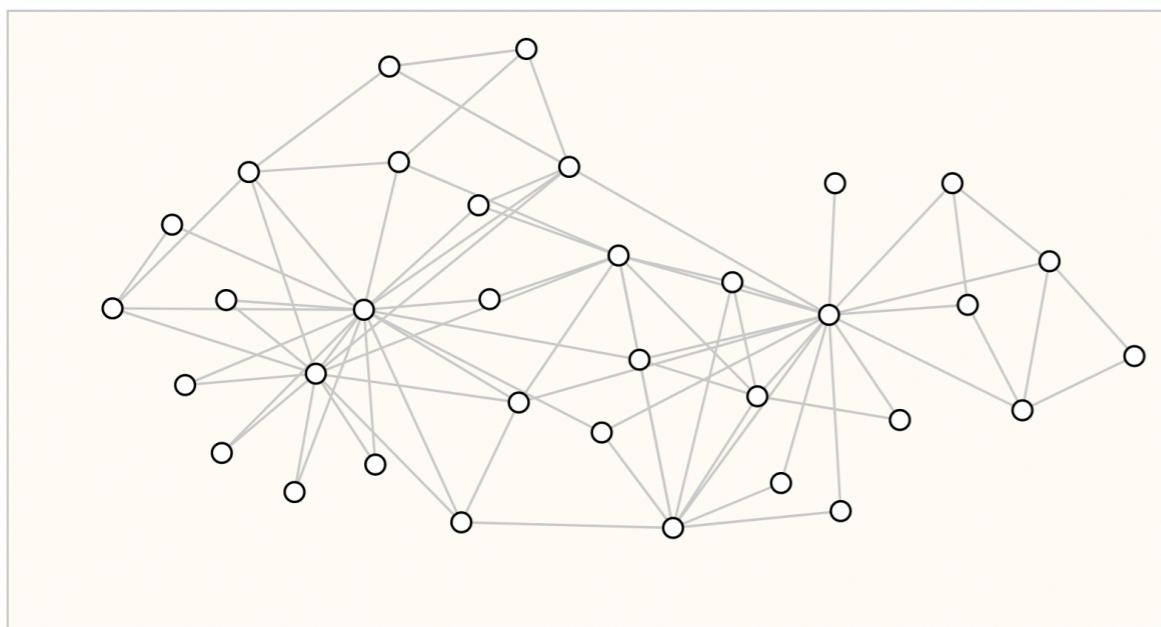
How to use GNNs?

- We start with nodes some of which have connections to others through the adjancency matrix → specified beforehand by edges
- We let the connected nodes (through the edges) to transfer info to the source node \mathbf{x}_i through an MLP (i.e. you learn what info from neighbours are important given \mathbf{x}_i)
 - this creates a new node representation \mathbf{h}_i that takes into account features of neighbours

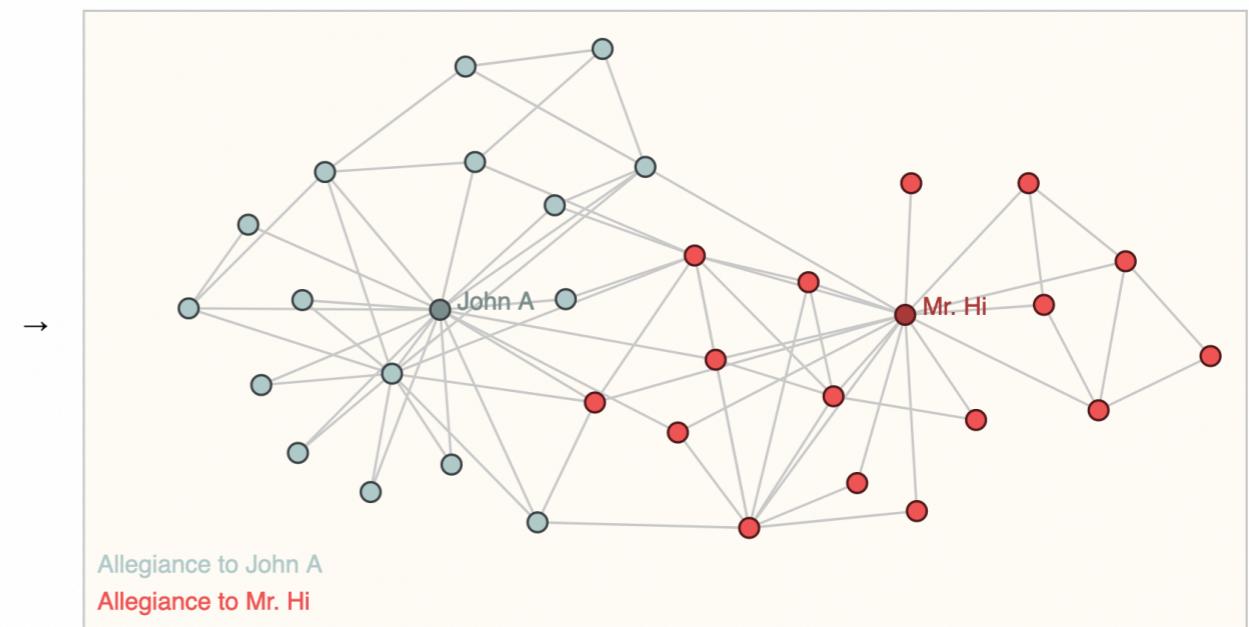


Tasks for graph data

- **Node-level task:** predict some property for each node in a graph
- Classic example: the [Zach's karate club](#) — a social network of a university karate club
 - a feud between Mr. Hi (Instructor) and John H (Administrator) creates a schism in the club
 - the task is to classify whether a given member becomes loyal to either Mr. Hi or John H
 - “distance” between a node to either the Instructor or Administrator is highly correlated to this label



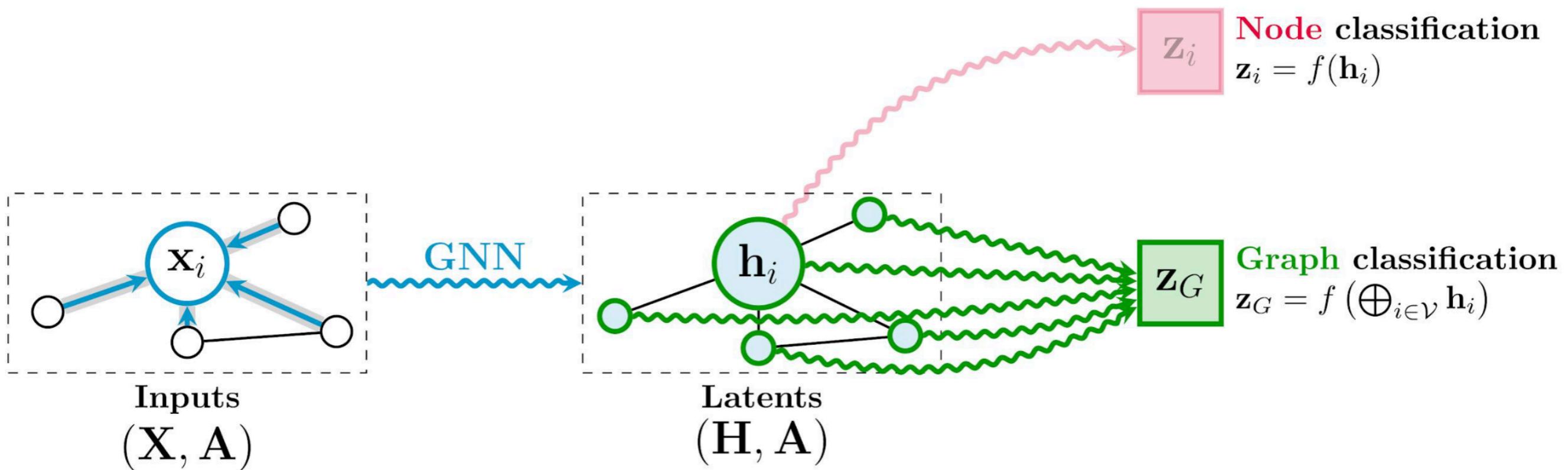
Input: graph with unlabeled nodes



Output: graph node labels

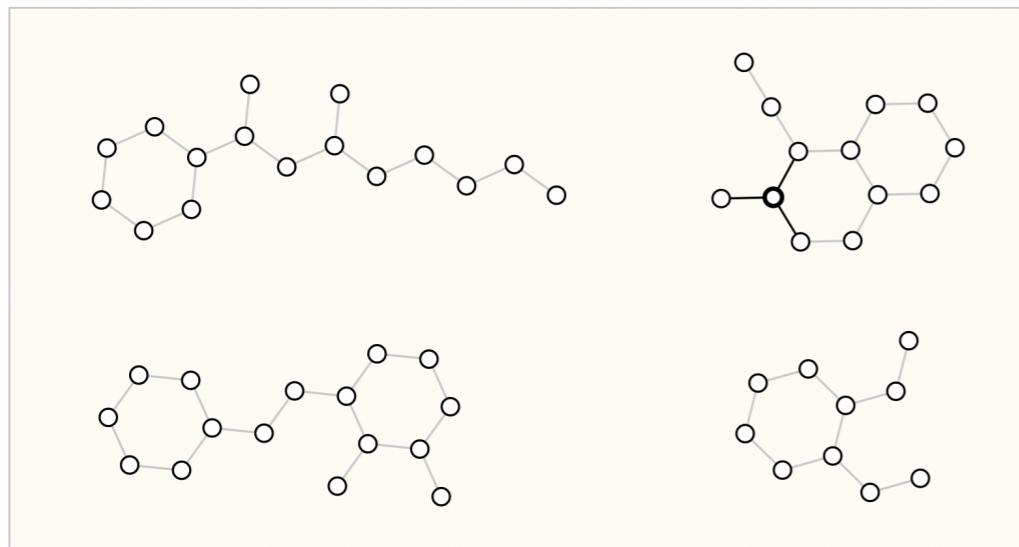
How to use GNNs?

- We start with nodes some of which have connections to others through the adjacency matrix → specified beforehand by edges
- We let the connected nodes (through the edges) to transfer info to the source node \mathbf{x}_i through an MLP (i.e. you learn what info from neighbours are important given \mathbf{x}_i)
 - this creates a new node representation \mathbf{h}_i that takes into account features of neighbours

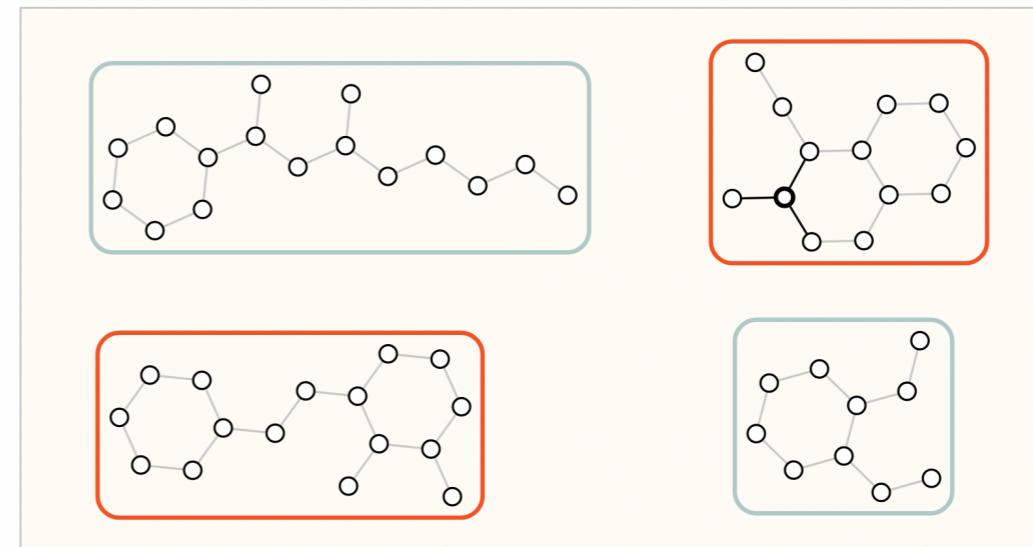


Tasks for graph data

- **Graph-level task:** predict some property for a whole graph
 - example: predict what the molecule smells like or whether it will bind to a receptor implicated in a disease



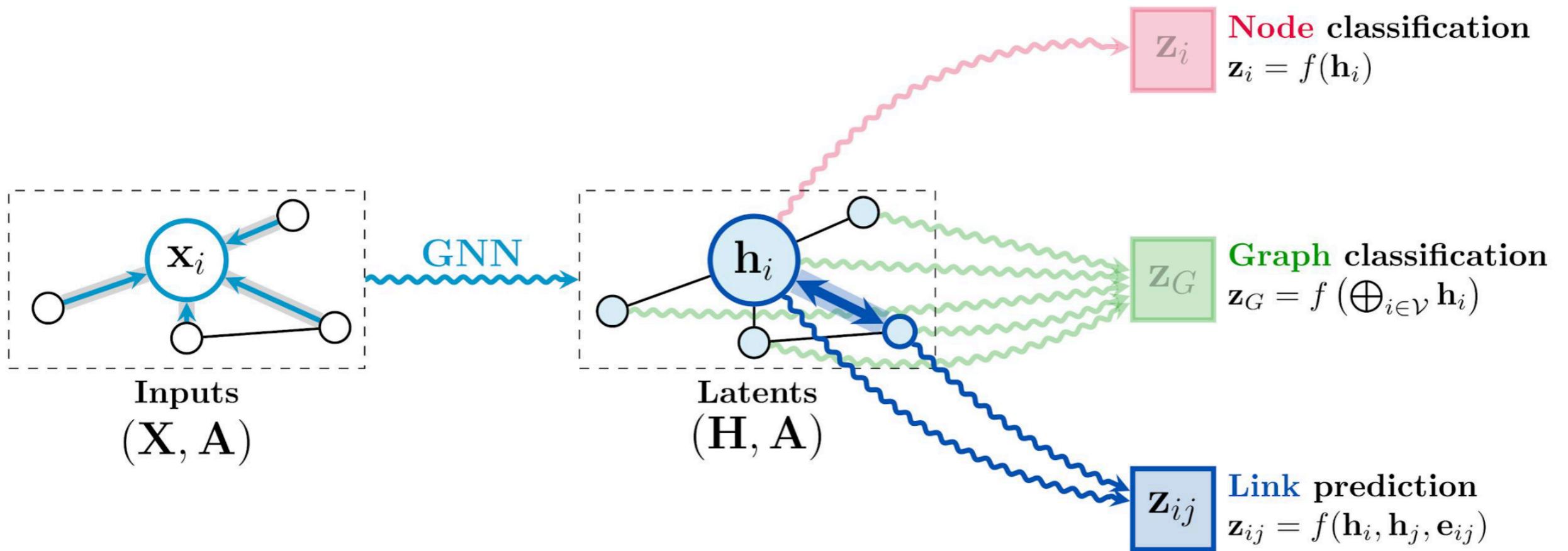
Input: graphs



Output: labels for each graph, (e.g., "does the graph contain two rings?")

How to use GNNs?

- We start with nodes some of which have connections to others through the adjancency matrix → specified beforehand by edges
- We let the connected nodes (through the edges) to transfer info to the source node \mathbf{x}_i through an MLP (i.e. you learn what info from neighbours are important given \mathbf{x}_i)
 - this creates a new node representation \mathbf{h}_i that takes into account features of neighbours

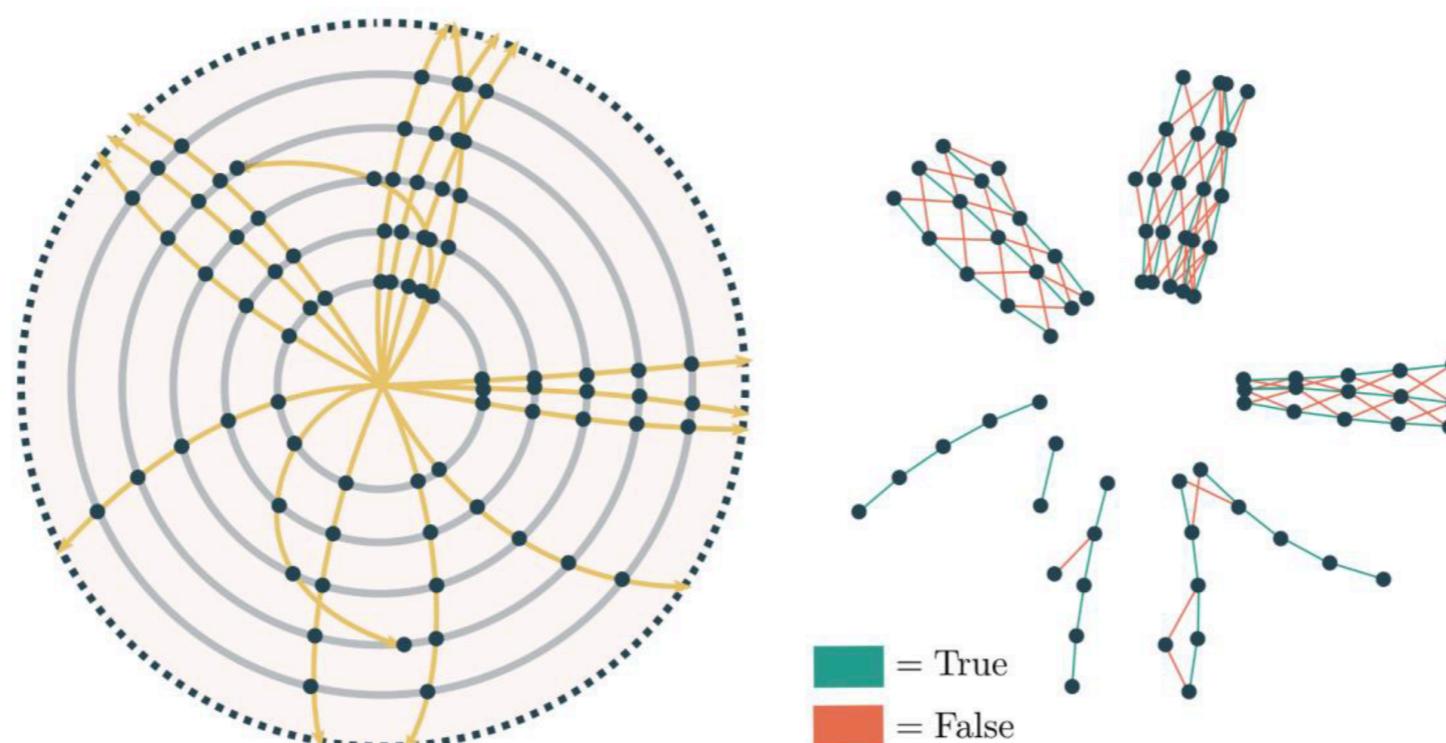


Tasks for graph data

- **Edge-level task:** predict some property or presence of edges in a graph

- if we wish to discover connections between entities, we could consider the graph fully connected and based on their predicted value prune edges to arrive at a sparse graph

- Draw edges to hypothesize various particle trajectories, train a GNN to classify edges



Input data is a 3D
point cloud

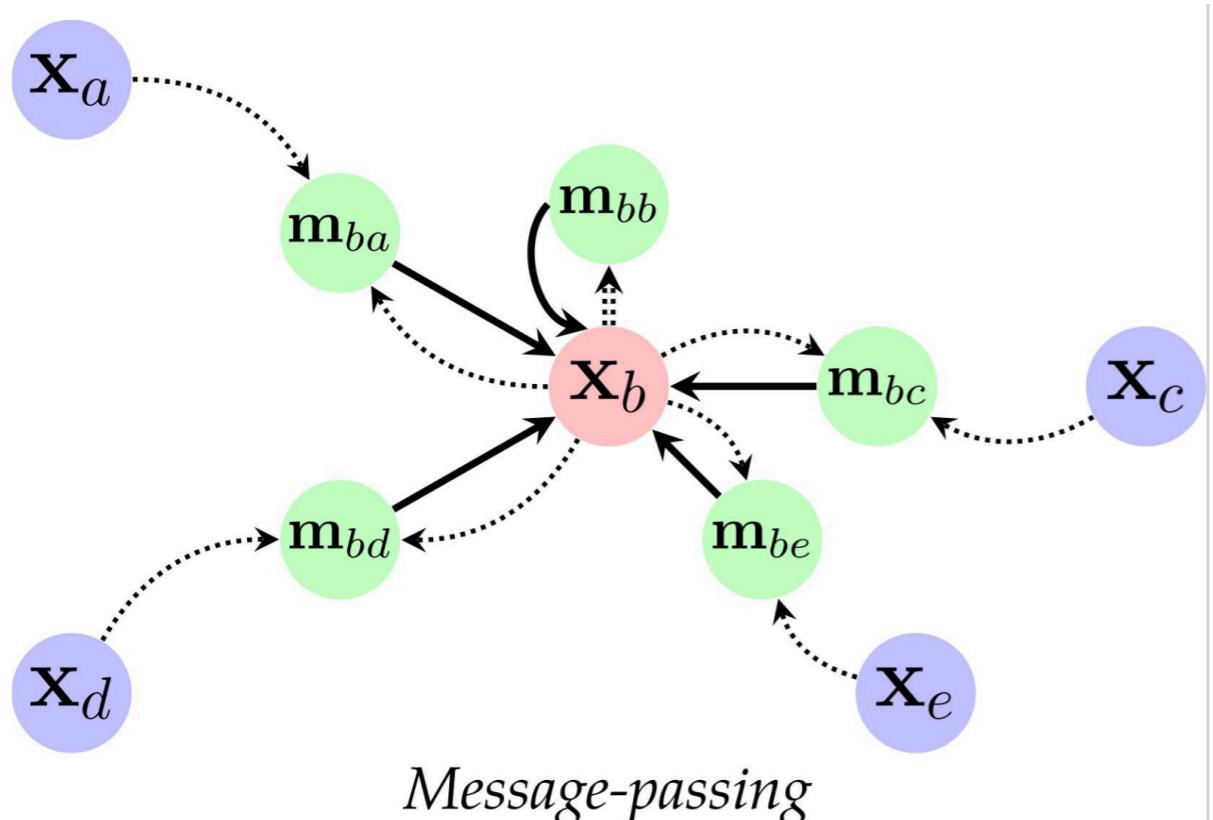
Train on graphs with edge
truth labels

What's in a GNN layer?

- As mentioned, we construct permutation-equivariant functions $f(\mathbf{X}, \mathbf{A})$ over graphs by shared application of a local permutation-invariant
 - we often refer to f as “GNN layer”, g as “diffusion”, “propagation”, “message passing”
- Now we look at ways in which we can actually concretely define g
 - Very intense area of research!
- This is the magic of GNNs — depending how you define g you can enforce basics (permutation) but also additional symmetries (*relational inductive bias*)
 - GNNs are very flexible!

Basic formalism of GNN

- The many GNN flavours are variants of a basic formalism called **Message Passing NN**



This is the basic GNN layer
If you stack more layers you get info from
farther apart

- **message passing:** for target node $i = b$, the “message” is passed at iteration k from neighbouring nodes a, c, d, e to the target node as

$$m_i^{(k)} = \sum_j M(h_i^{(k)}, h_j^{(k)}, e_{ij})$$

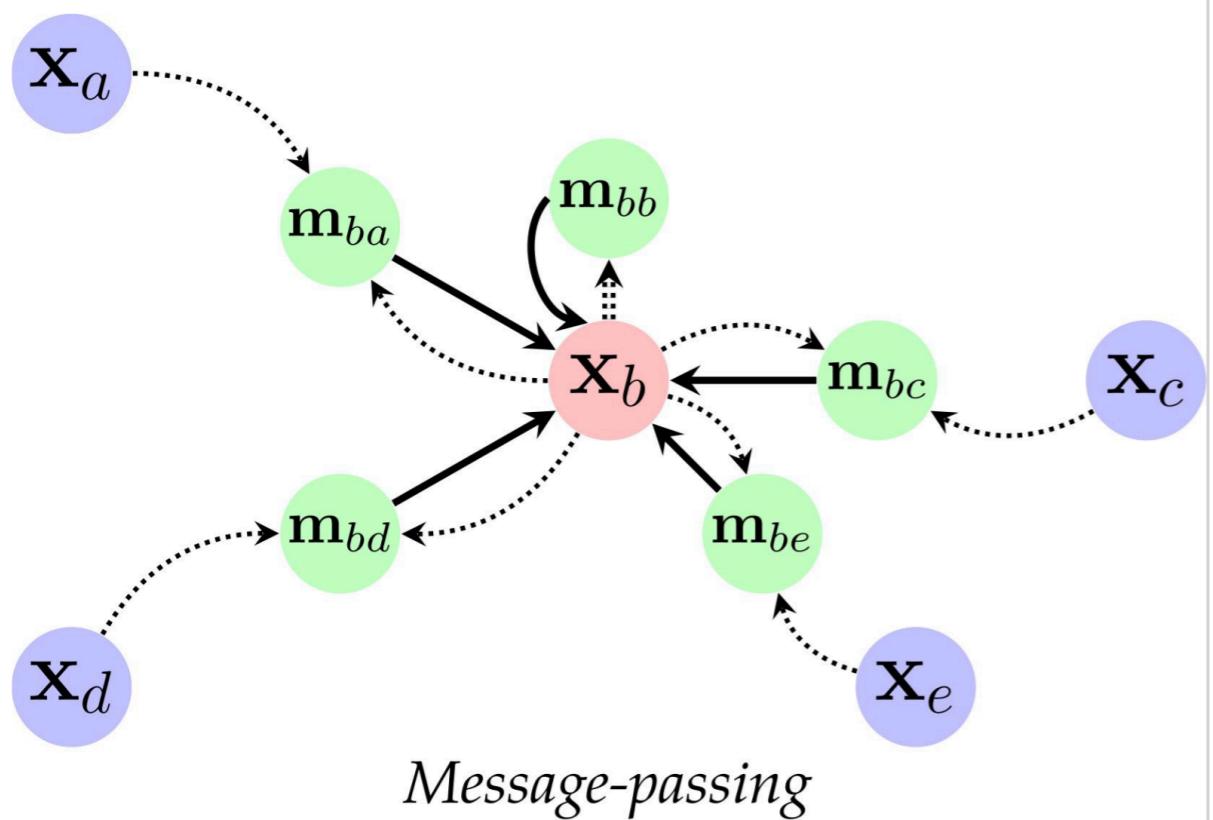
- Node feature update for the target node exploits all those info from neighbours as

$$h_i^{(k+1)} = U(h_i^{(k)}, m_i^{(k)})$$

- The functions M and U are message and updated functions
- The same can be performed on edges

Basic formalism of GNN

- The many GNN flavours are variants of a basic formalism called **Message Passing NN**



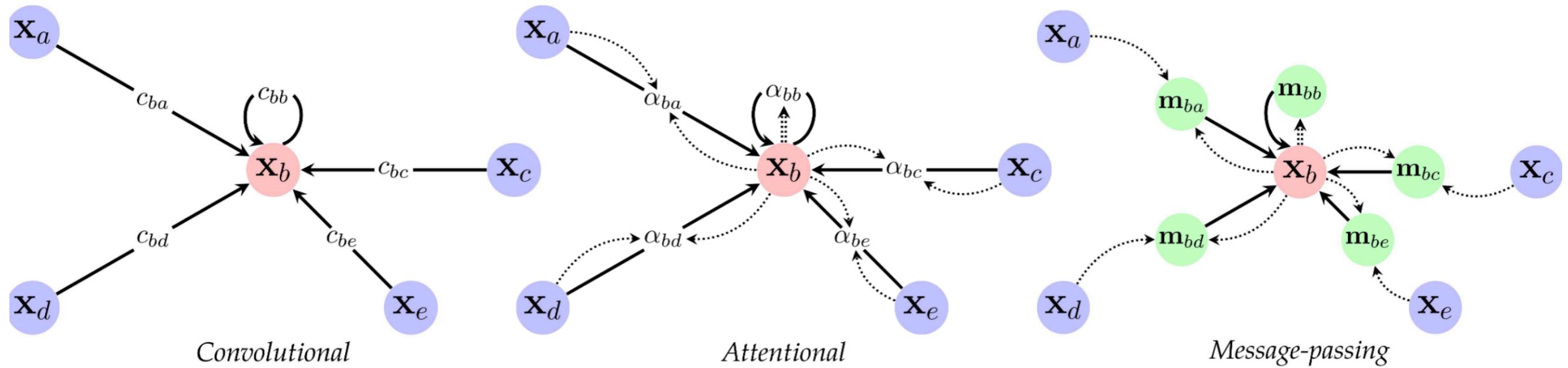
This is the basic GNN layer
If you stack more layers you get info from
farther apart

$$m_i^{(k)} = \sum_j M(h_i^{(k)}, h_j^{(k)}, e_{ij})$$

$$h_i^{(k+1)} = U(h_i^{(k)}, m_i^{(k)})$$

- The functions M and U are message and updated **functions**
 - can be simple MLPs in the simplest setup
 - must preserve permutation in/equivariance
- But depending on how these **functions** are designed one can incorporate different kinds of symmetries and assumptions
- This gives the different GNNs flavours

Basic “flavours” of GNNs



$$\mathbf{h}_i = \phi \left(\mathbf{x}_i, \bigoplus_{j \in \mathcal{N}_i} c_{ij} \psi(\mathbf{x}_j) \right)$$

$$\mathbf{h}_i = \phi \left(\mathbf{x}_i, \bigoplus_{j \in \mathcal{N}_i} a(\mathbf{x}_i, \mathbf{x}_j) \psi(\mathbf{x}_j) \right)$$

$$\mathbf{h}_i = \phi \left(\mathbf{x}_i, \bigoplus_{j \in \mathcal{N}_i} \psi(\mathbf{x}_i, \mathbf{x}_j) \right)$$

Graph Convolutional NN

- In vanilla GCN the weight c_{ij} is a fixed value that normalize the sum of the neighbour embeddings to make the aggregated message of the same size for all nodes

- by using the sum as aggregation function you expect the output to be larger for nodes that have more neighbours → numerical instabilities
- can be solved with the mean, i.e. $c_{ij} = \text{number of neighbours}$, but there are also other options

$$\mathbf{h}_u^{(k)} = \sigma \left(\mathbf{W}^{(k)} \sum_{v \in \mathcal{N}(u) \cup \{u\}} \frac{\mathbf{h}_v}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}} \right)$$

node degree

