Graph Neural Networks

Mihai Alexe

ECMWF Bonn

first.lastname@ecmwf.int

A very fast and evolving landscape

Defining the dataset, split, headline fields and metrics

2020 WeatherBench

2018
Exploring the concept

ECMWF staff
~500km ERA5 to predict future z500.
Similar work from Rasp and Weyn.

Feb 2022
Full medium-range NWP

Keisler - GraphNN
1°, competitive with GFS

NVIDIA – FourCastNet
Fourier+ , 0.25°

O(10^4) faster & more energy efficient than IFS

Dec 2022
Extensive predictions

Deepmind – GraphCast
0.25° 6-hour

Many variables and pressure levels with comparable skill to IFS.

Huawei – PanguWeather
0.25° hourly product

"More accurate tracks" than the IFS.

Nov 2022
Tropical cyclones

Microsoft – ClimaX

Forecasting various lead-times at various resolutions, both globally and regionally

Jan 2023
Global & Limited Area

Apr 2023
7-day+ scores improve

FengWu – China academia + Shanghai Met Bureau
0.25° 6-hour product

Improves on GraphCast for longer leadtimes (still deterministic)

NVIDIA – SFNO
0.25° 6-hour product

Extension of FourCastNet to Spherical harmonics, improved stability

Spherical harmonics

Jun 2023

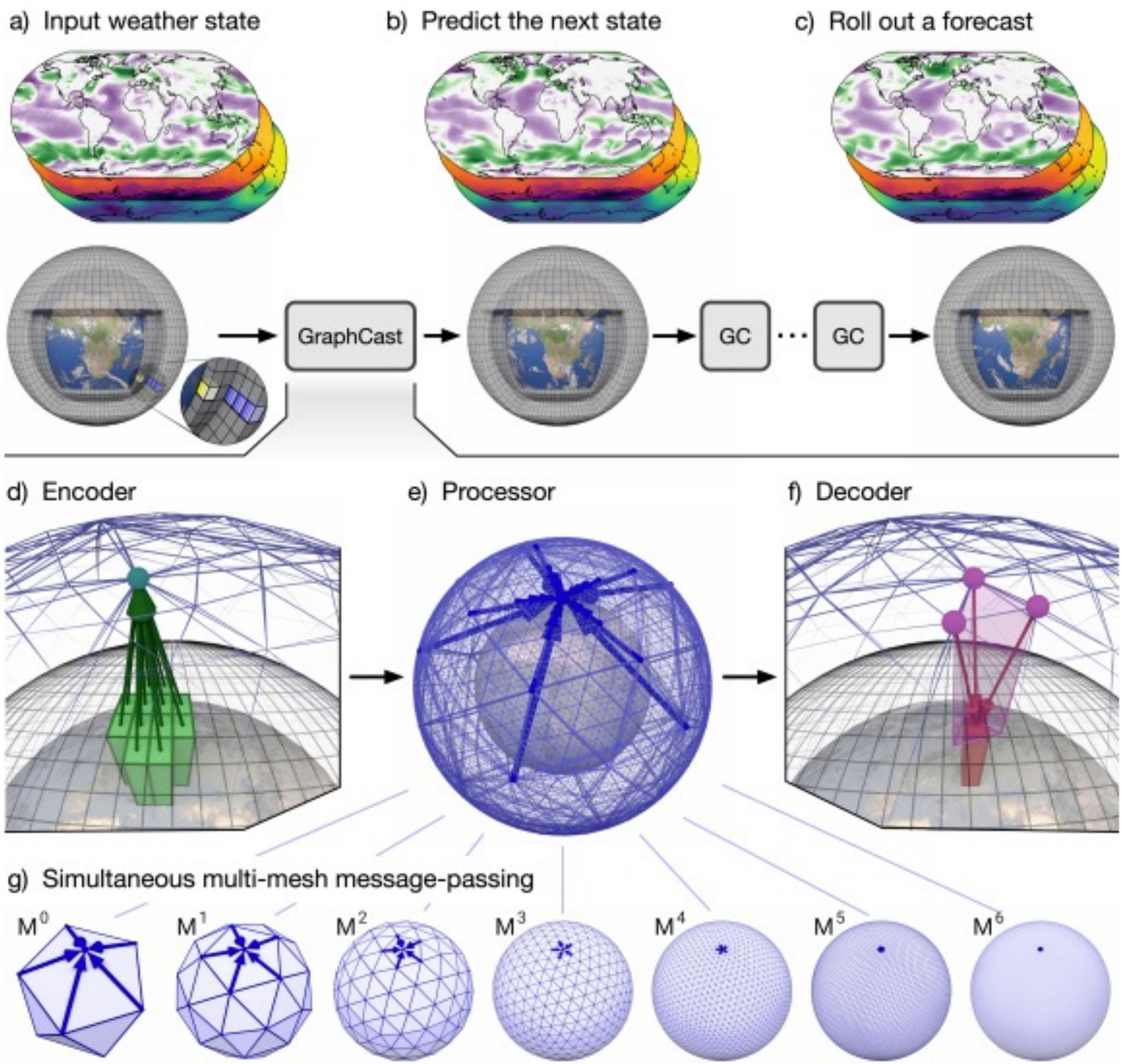
Diffusion modelling

Alibaba – SwinRDM
0.25° 6-hour product

Sharp spatial features

Last months
AIFS
FuXi
AtmoRep
FuXi-extreme
NeuralGCM
GenCast
…

impossible to keep this figure up



https://www.science.org/doi/10.1126/science.adi2336
https://arxiv.org/abs/2212.12794

Refresher on graphs

We define a *graph* as the pair $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ of vertices $v \in \mathcal{V}$ and edges $e_{ij} = (v_i, v_j) \in \mathcal{E}$ with $\mathcal{E} \subseteq V \times V$. The graph connectivity is encoded as an *adjacency matrix* $A \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{V}|}$, with

$$a_{ij} = \begin{cases} 1, & e_{ij} \in \mathcal{E} \\ 0, & e_{ij} \notin \mathcal{E} \end{cases}$$

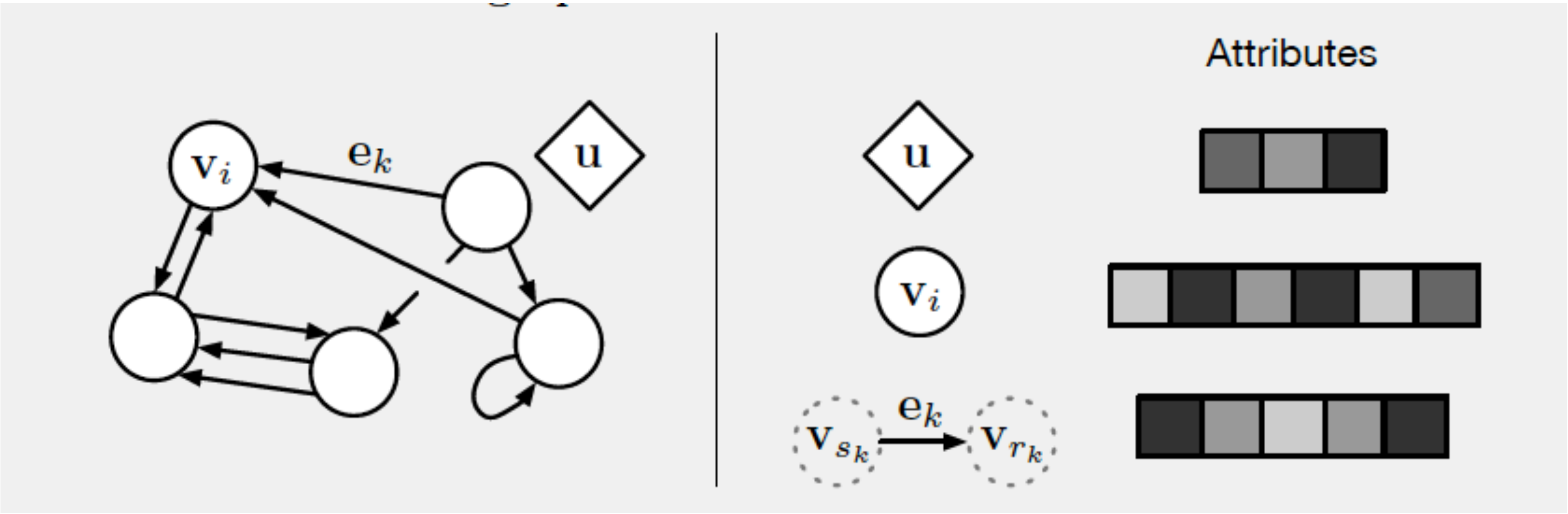


Figure from Battaglia et al., 2018

Graph neural networks

We endow the graph nodes with features - namely for each $\boldsymbol{u} \in \mathcal{V}$ we define a *node feature tensor* $\boldsymbol{x} \in \mathbb{R}^k$. This defines a matrix $\boldsymbol{X} \in \mathbb{R}^{|\mathcal{V}| \times k}$:

$$\boldsymbol{X} = \left[\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_{|\mathcal{V}|}\right]$$

We also define *edge features* $\boldsymbol{x}_{\boldsymbol{uv}} \in \mathbb{R}^l$ and global *graph features* $\boldsymbol{x}_{\mathcal{G}} \in \mathbb{R}^m$.

GNNs are neural networks built to operate on this "graph data".

Quick detour: MLPs

Multi-layer perceptrons

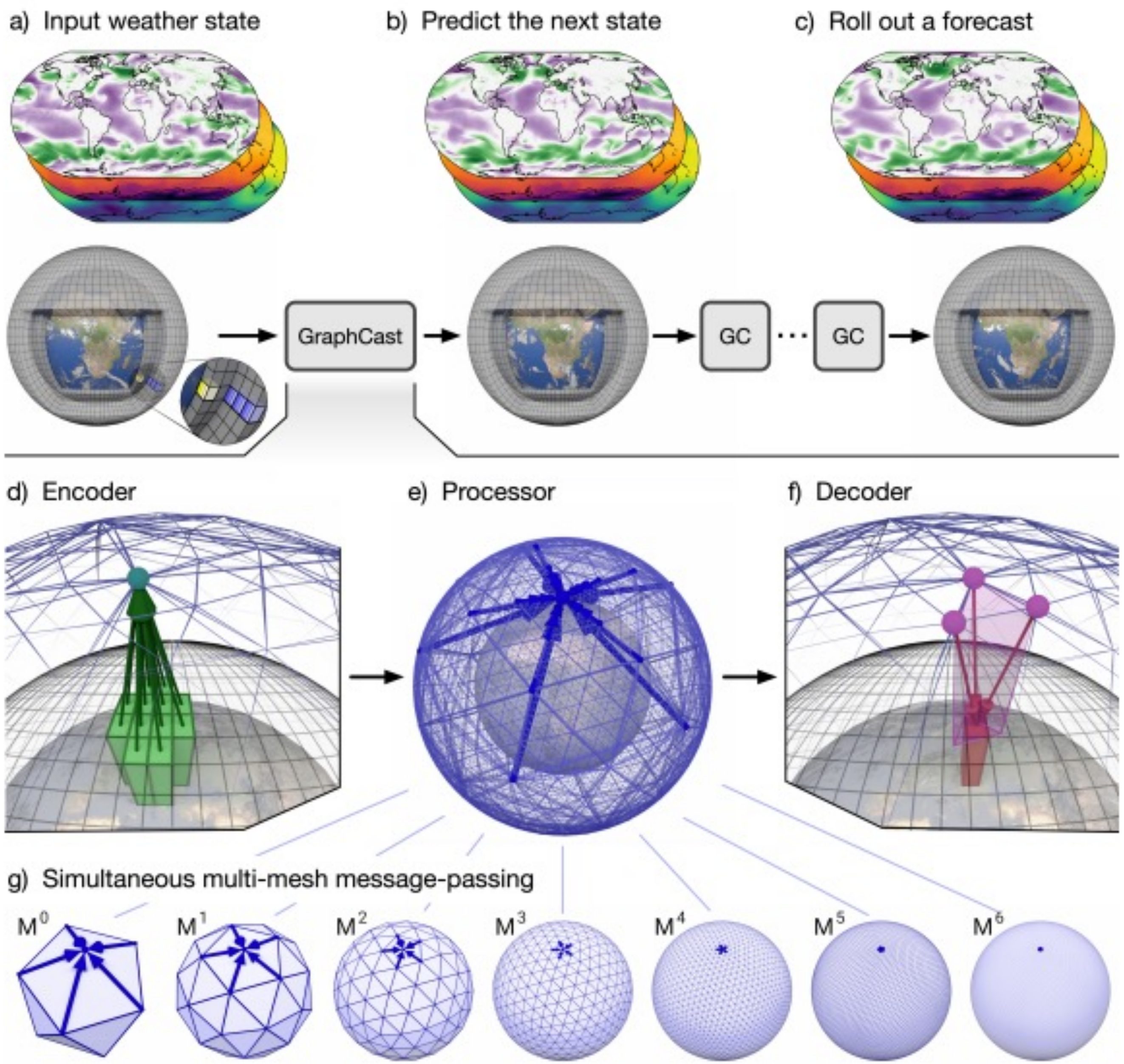
```
LayerNorm(Activation(Linear(x)))
```

```
from torch import nn

def generate_mlp_module(num_inputs: int = 32, hidden_dim: int = 64, num_outputs: int = 32):
    mlp = nn.Sequential(
        nn.Linear(num_inputs, hidden_dim),
        nn.LeakyReLU(0.1),
        nn.Linear(hidden_dim, hidden_dim),
        nn.LeakyReLU(0.1),
        nn.Linear(hidden_dim, num_outputs),
        nn.LeakyReLU(0.1),
        nn.LayerNorm(num_outputs)
    )
    return mlp
```

MLPs will be denoted by Greek letters ϕ, ψ and ρ

What inductive biases should a GNN have?



https://www.science.org/doi/10.1126/science.adi2336
https://arxiv.org/abs/2212.12794

Locality

We want the GNN signal to be stable under small domain deformations.

Standard deep NNs (e.g., CNNs) build large-scale ops from small-scale building blocks (e.g. 3x3 convolutions).

GNN layers should operate locally, too (in neighborhoods).

We can extract neighborhood features and define local functions ϕ (MLPs) operating on them:

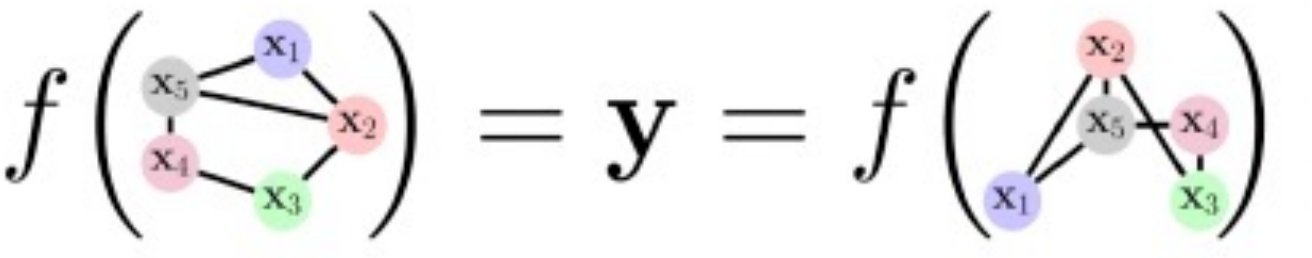
$$\mathbf{X}_{\mathcal{N}_i} = \{\{\, \mathbf{x}_j : j \in \mathcal{N}_i \,\}\}$$

$$\phi(\mathbf{x}_i, \mathbf{X}_{\mathcal{N}_i}).$$

Permutation invariance and equivariance

The specific <u>ordering</u> of nodes / edges should not matter!

Invariance $$f(PX, PAP^T) = f(X, A)$$



Examples: max, sum, min, avg

$\bigoplus$ = any permutation-invariant aggregation op acting on one or more graph nodes / edges

Permutation equivariance

What if we wanted to distinguish between outputs at different nodes?

A permutation-invariant aggregator would not allow us to do that ☹

Instead, we may use a functions that don't change the node ordering.

That is, if we permute nodes using a *permutation matrix P*, it doesn't matter if we do it before or after! ☺

$$\mathbf{F}(\mathbf{X}, \mathbf{A}) = \begin{bmatrix} - & \phi(\mathbf{x}_1, \mathbf{X}_{\mathcal{N}_1}) & - \\ - & \phi(\mathbf{x}_2, \mathbf{X}_{\mathcal{N}_2}) & - \\ & \vdots & \\ - & \phi(\mathbf{x}_n, \mathbf{X}_{\mathcal{N}_n}) & - \end{bmatrix}$$

$$F(PX, PAP^T) = PF(X, A)$$

We then stack multiple equivariant GNN layers to build large-scale operators:

$$\mathbf{F}(\mathbf{X}, \boldsymbol{A}) := \phi\left(\bigoplus_{v \in \mathcal{N}_u} \psi(\boldsymbol{x}_u, \boldsymbol{x}_v, \boldsymbol{x}_{uv})\right)$$

$\bigoplus$ = any permutation-invariant aggregation op acting on one or more graph nodes / edges

We've just defined a GNN layer!

$$\mathbf{F}(\mathbf{X}, \mathbf{A}) := \phi\left(\bigoplus_{v \in \mathcal{N}_u} \psi(\mathbf{x}_u, \mathbf{x}_v, \mathbf{x}_{uv})\right)$$

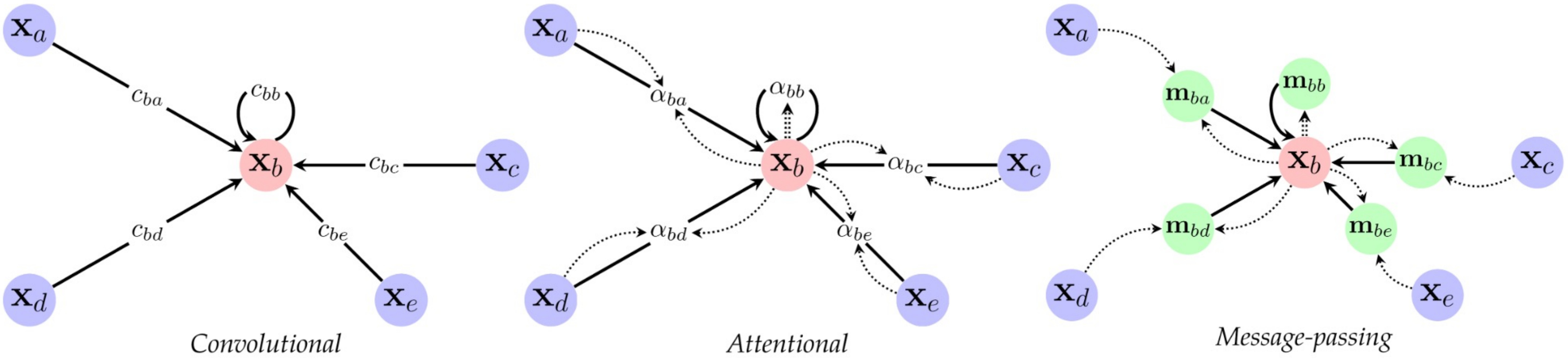
Trainable, shared MLPs

GNN layers are defined by the shared application of local, differentiable and permutation invariant MLPs

The per-edge and per-node functions ϕ and ψ are reused across all edges and all nodes, respectively. This means a GNN can operate on graphs of different sizes ($|\mathcal{V}|$ or $|\mathcal{E}|$) and shapes ($\mathbf{A}$). This is crucial if your graph is dynamic, e.g. it varies with time.

Flavors of GNNs

$$h_i = \phi(x_i, \bigoplus_{j \in \mathcal{N}_i} \alpha(x_i, x_j)\psi(x_j))$$



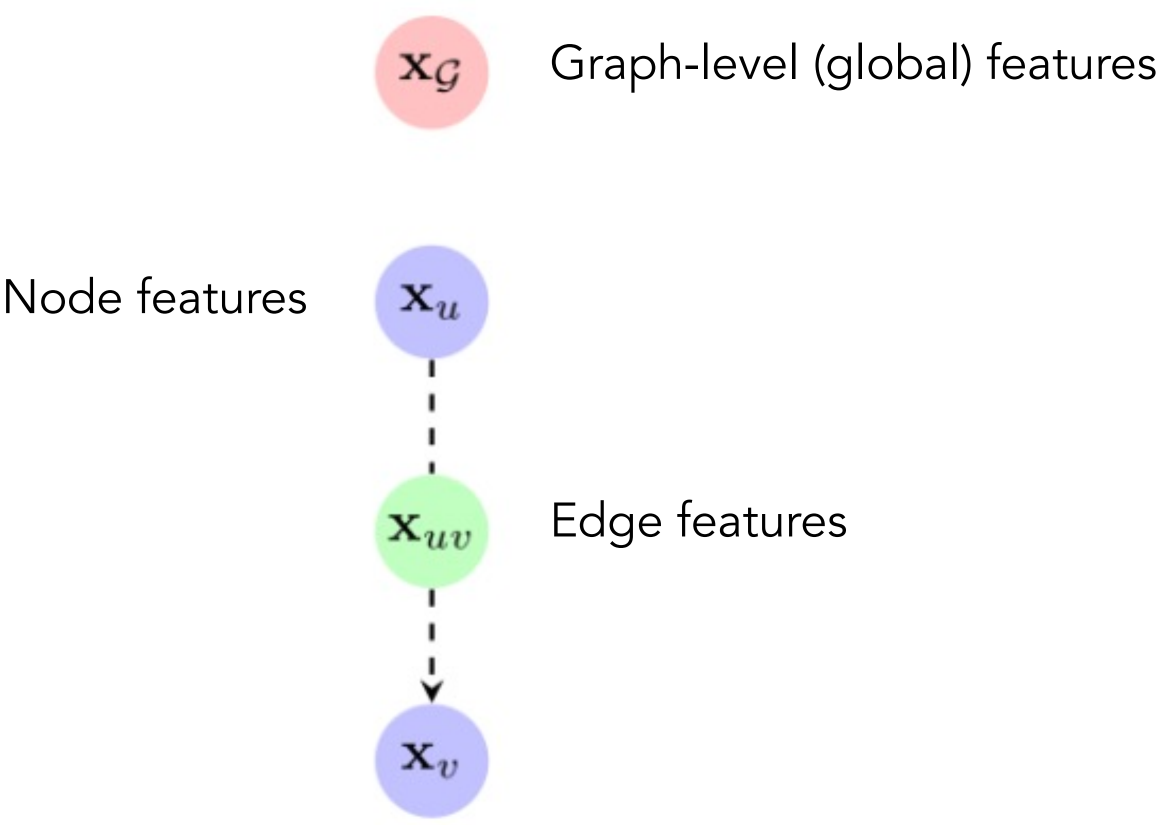
$$h_i = \phi(x_i, \bigoplus_{j \in \mathcal{N}_i} c_{ij}\psi(x_j))$$

$$h_i = \phi(x_i, \bigoplus_{j \in \mathcal{N}_i} \psi(x_i, x_j, e_{ij}))$$

$$\mathbf{m}_{ij} := \psi(x_i, x_j, e_{ij}).$$

Message-passing GNNs

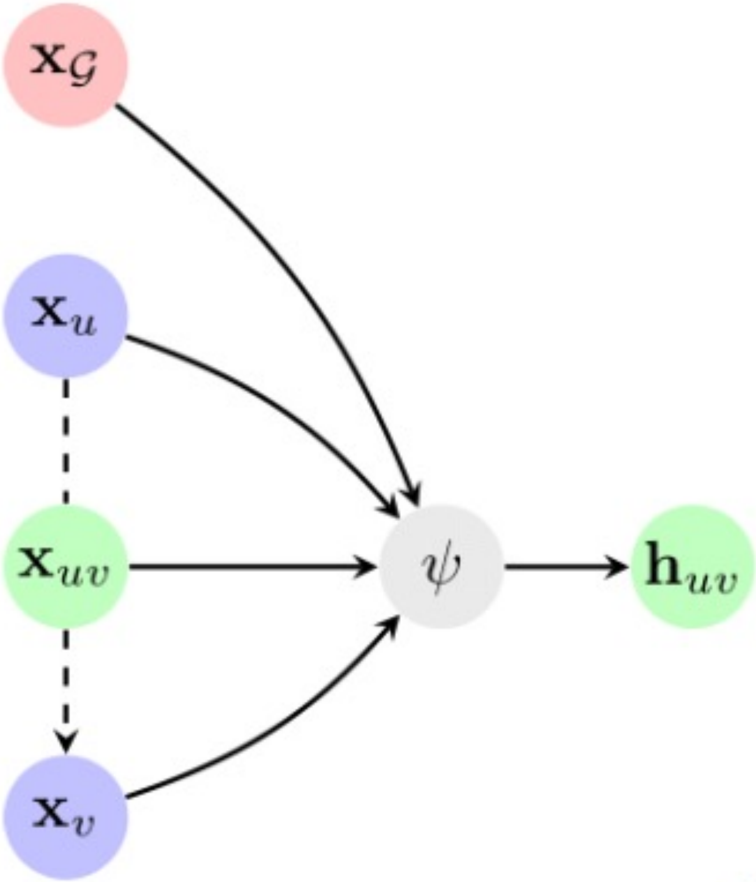
Features (= information associated with elements of our graph)



GNN diagrams taken from (Velickovic, 2022): Geometric deep learning - lecture 7: Graphs and sets (II). https://geometricdeeplearning.com/lectures/

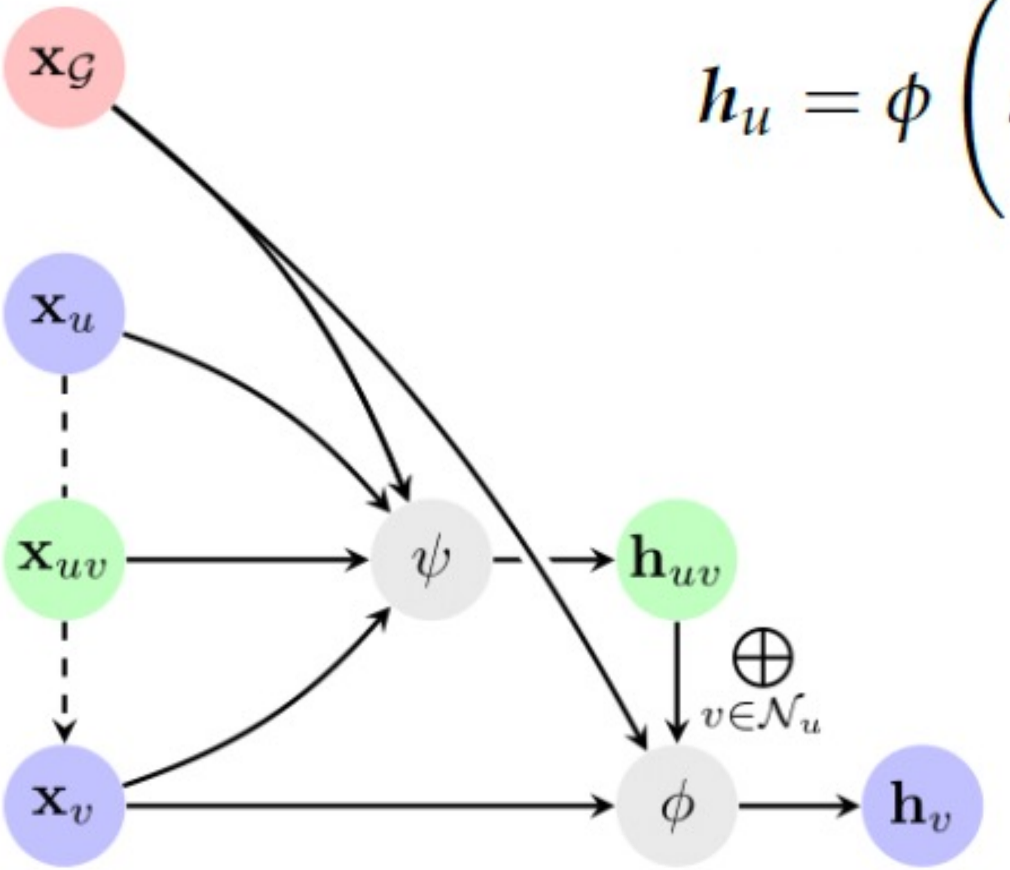
The message-passing algorithm

Step 1: Edge updates



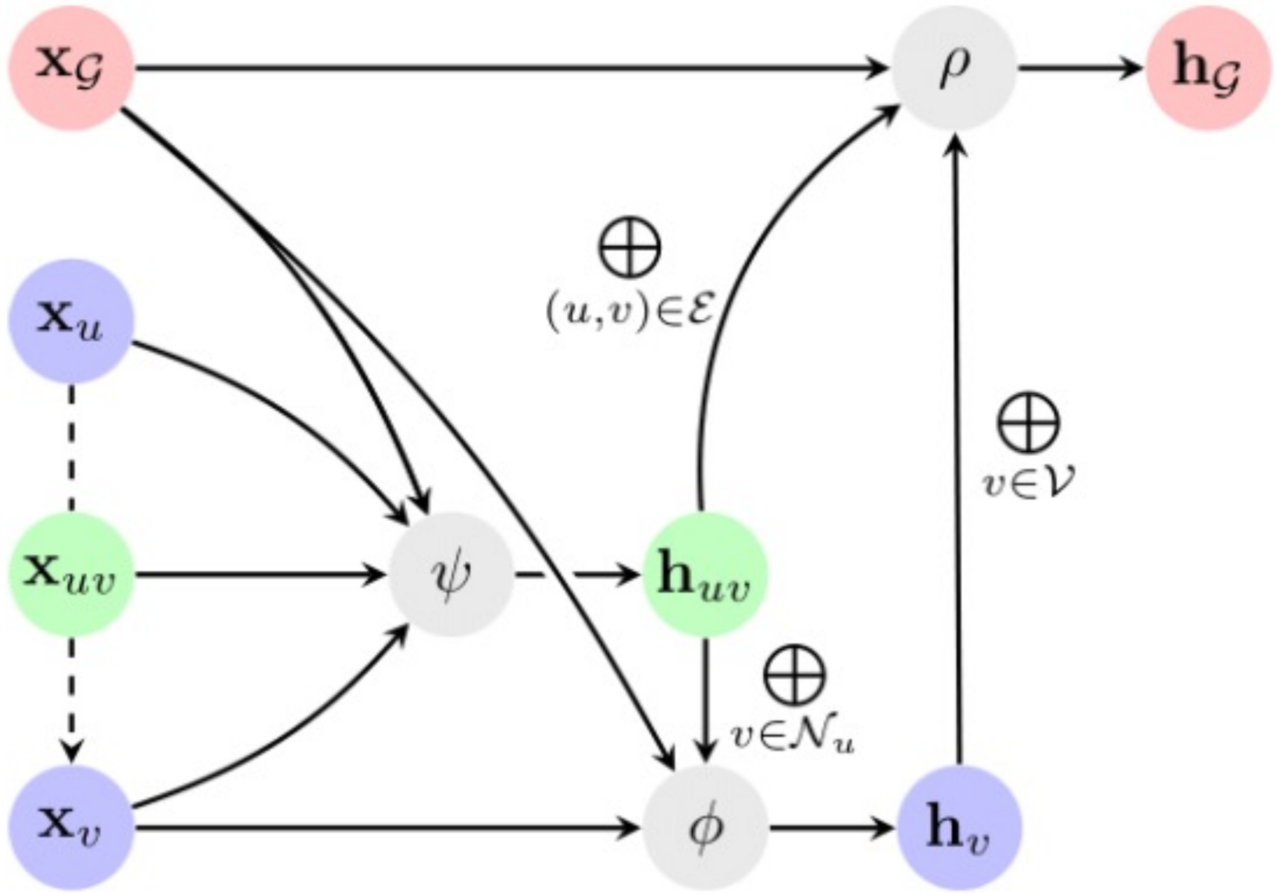
$$\boldsymbol{h}_{uv} = \psi(\boldsymbol{x}_u, \boldsymbol{x}_v, \boldsymbol{x}_{uv}, \boldsymbol{x}_{\mathcal{G}})$$

Step 2: Node updates



$$h_u = \phi\left(x_u, \bigoplus_{u \in \mathcal{N}_v} h_{vu}, x_{\mathcal{G}} \right)$$

Step 3: Graph feature updates

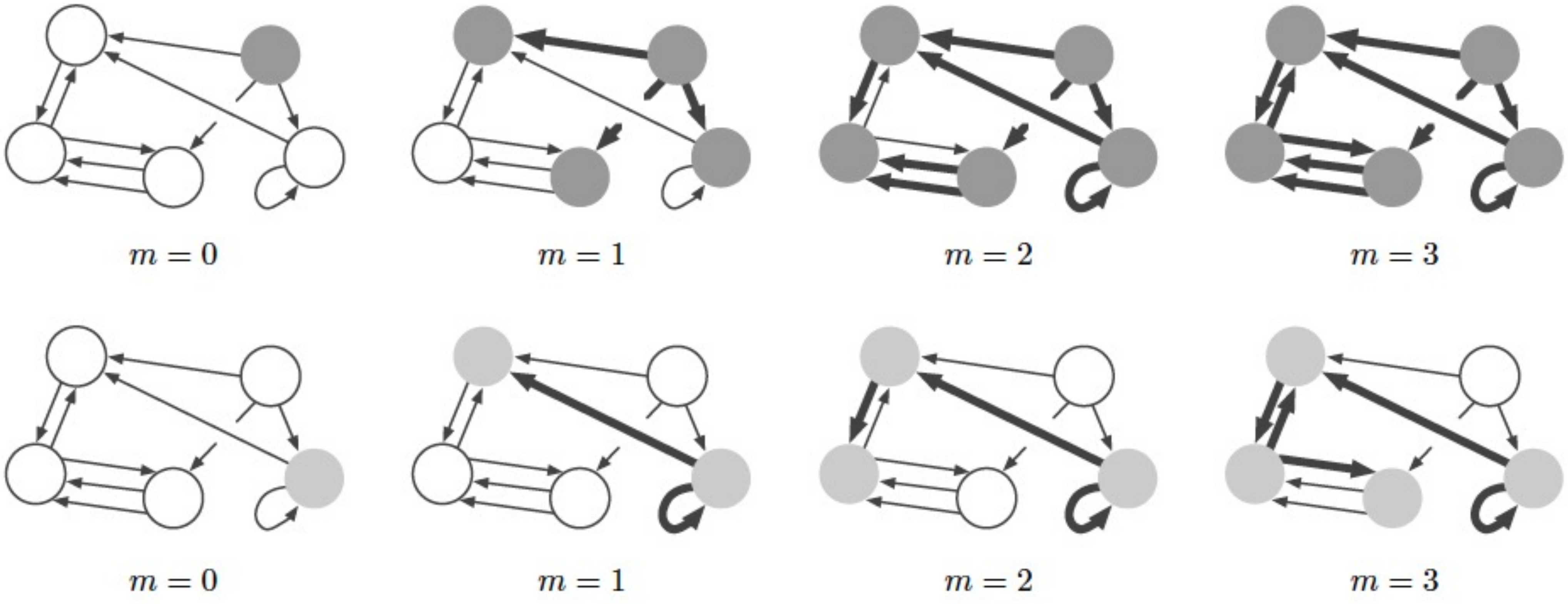


$$h_{\mathcal{G}} = \rho\left(\bigoplus_{u\in\mathcal{V}} h_u, \bigoplus_{(u,v)\in\mathcal{E}} h_{uv}, x_{\mathcal{G}}\right)$$

The message-passing algorithm

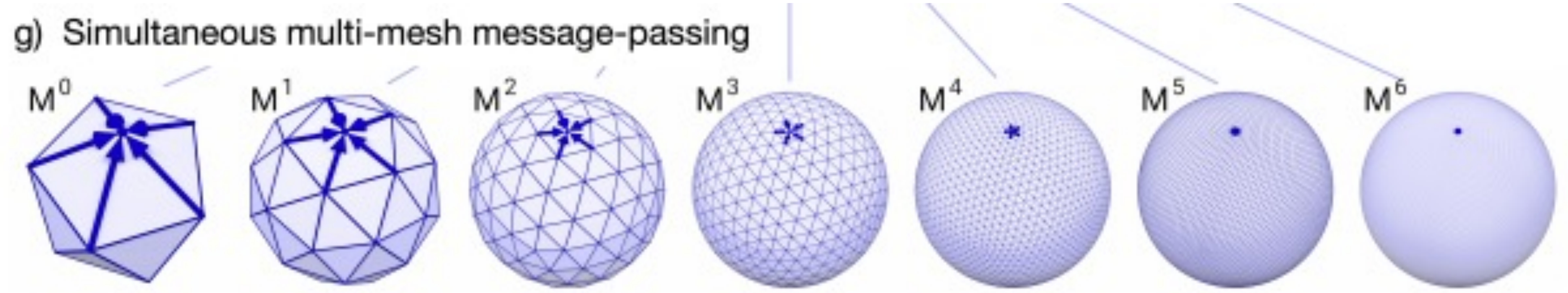
Input: Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ with $\{\boldsymbol{x}_{\mathcal{G}}, \boldsymbol{h}_u, \boldsymbol{h}_{uv}\}$.
for each edge e_{uv} **do**
    Gather sender and receiver nodes $\boldsymbol{x}_u$, $\boldsymbol{x}_v$
    Update edge $\boldsymbol{h}_{uv} \leftarrow \psi(\boldsymbol{x}_u, \boldsymbol{x}_v, \boldsymbol{x}_{uv}, \boldsymbol{x}_{\mathcal{G}})$
end for
for each node u **do**
    Aggregate all incoming edges to u: $\boldsymbol{h}_u^* := \bigoplus_{v,(v,u)\in\mathcal{E}} \boldsymbol{h}_{vu}$
    Compute node-wise features $\boldsymbol{h}_u \leftarrow \phi\left(\boldsymbol{x}_u, \boldsymbol{h}_u^*, \boldsymbol{x}_{\mathcal{G}}\right)$
end for
Aggregate all edges and nodes $\boldsymbol{u}^* := \bigoplus_{u\in\mathcal{V}} \boldsymbol{h}_u$, $\mathbf{e}^* := \bigoplus_{(u,v)\in\mathcal{E}} \boldsymbol{h}_{uv}$
Compute global features $\boldsymbol{h}_{\mathcal{G}} \leftarrow \rho(\boldsymbol{x}_{\mathcal{G}}, \boldsymbol{u}^*, \mathbf{e}^*)$
Output: Graph $\mathcal{G}$ with new $\{\boldsymbol{x}_{\mathcal{G}}, \boldsymbol{h}_u, \boldsymbol{h}_{uv}\}$.

Message passing: information propagation



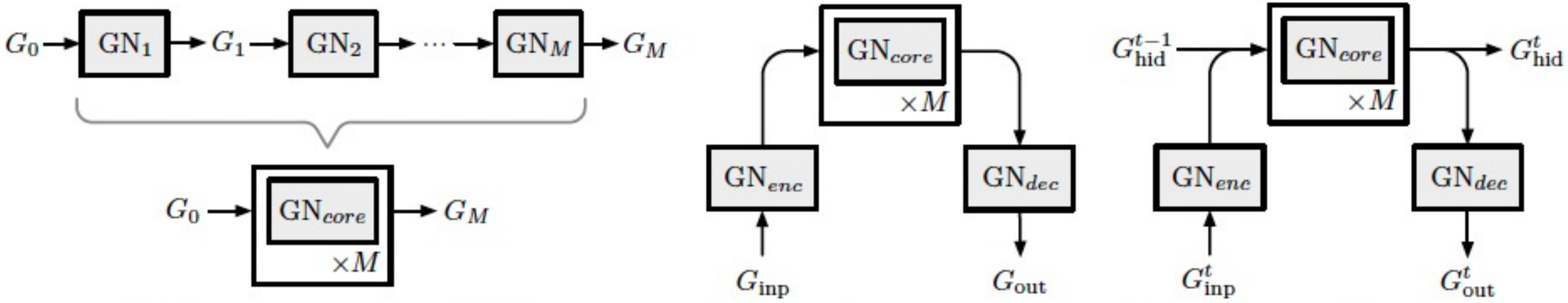
NB: This happens simultaneously for all nodes in the graph!

Figure from ([Battaglia et al., 2018](#))



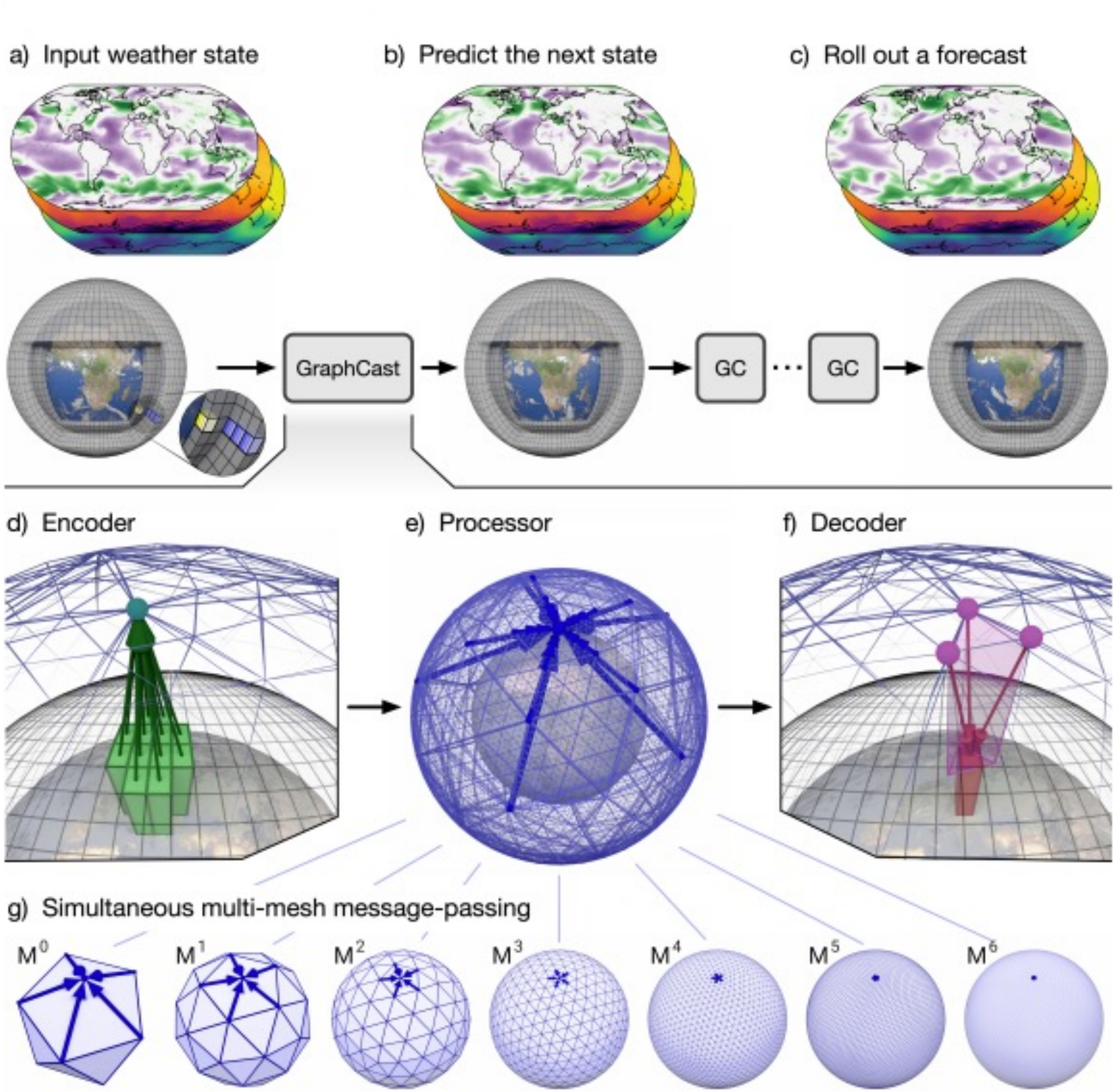
The multi-mesh allows information to propagate faster, across longer distances

GN block structures



(a) Composition of GN blocks (b) Encode-process-decode (c) Recurrent GN architecture

Graphcast and AIFS use both (a) and (b)



https://www.science.org/doi/10.1126/science.adi2336
https://arxiv.org/abs/2212.12794

Software

PyG

https://github.com/pyg-team/pytorch_geometric

DGL DEEP GRAPH LIBRARY

https://www.dgl.ai/

Spektral

https://graphneural.network/

TensorFlow GNN

https://github.com/tensorflow/gnn

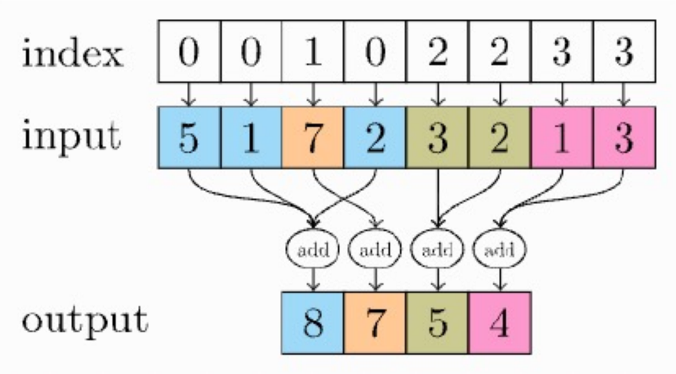
Jraph

https://github.com/google-deepmind/jraph

pytorch-geometric: `MessagePassing`

Pytorch geometric's `MessagePassing` class implements message passing as follows:

1. `message()` implements the MLP ϕ that is to say, it constructs a message $u_j \to u_i$ for each edge in the edge index
2. `update()` implements ϕ; it concatenates all inputs before passing them through the MLP
3. `aggregate()` implements the aggregation logic over a neighborhood, i.e. the $\bigoplus_{j \in \mathcal{N}_i}$ operator using a GPU-accelerated *scatter* operation
4. `propagate()` is the initial call to start propagating a message through the graph

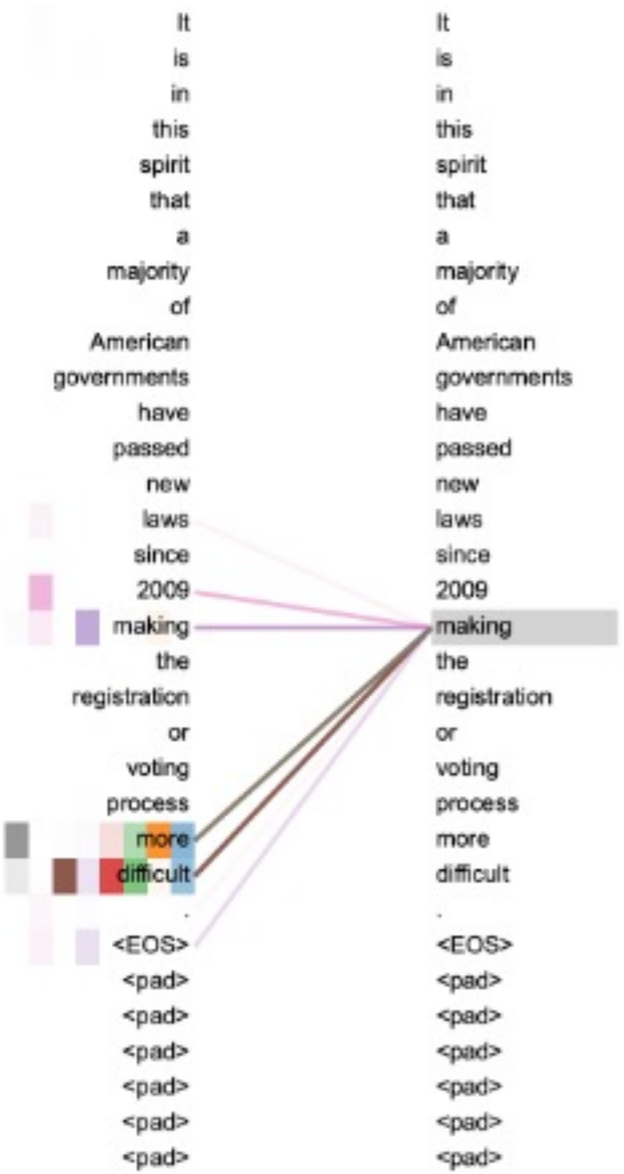


Transformers are fully connected GNNs (+ a positional embedding)

$$\boldsymbol{A} = \mathbb{1}\mathbb{1}^T$$

$$\mathcal{N}_u = \mathcal{V}.$$

$$\boldsymbol{h}_u = \phi\left(\boldsymbol{x}_u, \bigoplus_{v \in \mathcal{V}} \alpha(\boldsymbol{x}_u, \boldsymbol{x}_v)\psi(\boldsymbol{x}_v)\right)$$



Chaitanya Joshi. Transformers are graph neural networks. The Gradient, 2020.

Further references

(Veličković, 2023) https://arxiv.org/pdf/2301.08210.pdf

(Keisler, 2022) https://arxiv.org/abs/2202.07575

(Lam et al., 2023) https://arxiv.org/abs/2212.12794

(Sanchez-Lengeling et al., 2021) https://distill.pub/2021/gnn-intro/

(Veličković, 2023) https://geometricdeeplearning.com/lectures/

(Battaglia et al., 2018) https://arxiv.org/abs/1806.01261

(Sanchez-Gonzalez et al., 2020) https://arxiv.org/abs/2002.09405