dortmund university

SFB 876 Providing Information by Resource-Constrained Data Analysis

Tutorial on Graph Neural Networks

Workshop on Geometry and Machine Learning with Applications to Biomedical Engineering - University College London

Matthias Fey

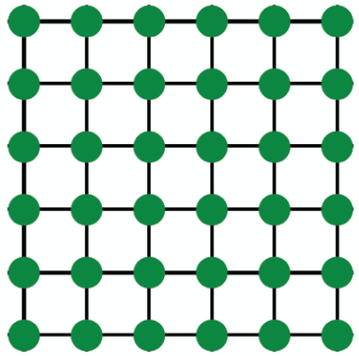
matthias.fey@udo.edu

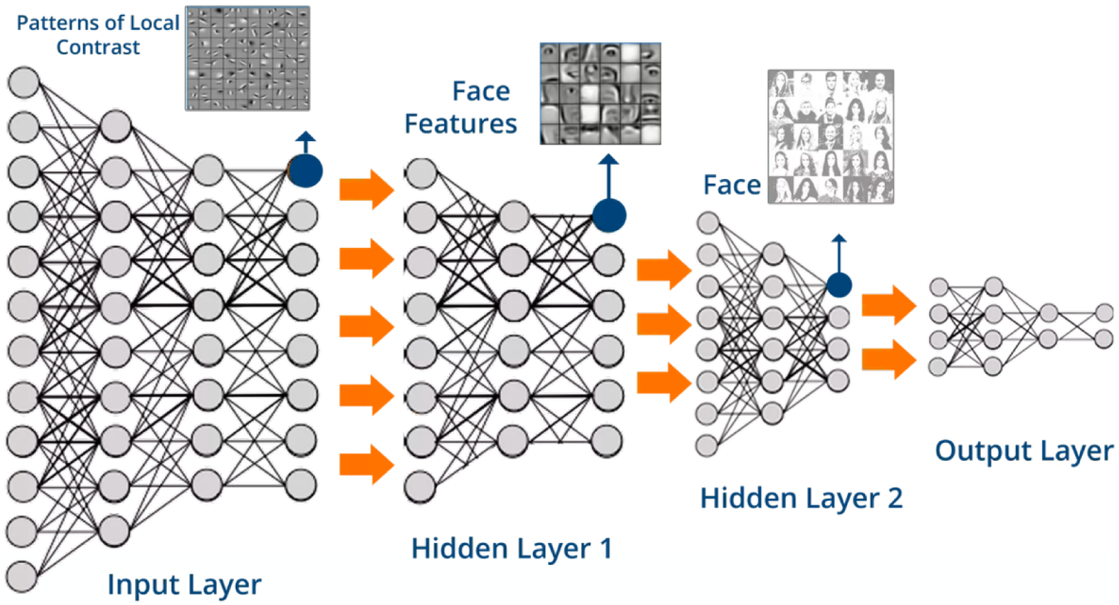
rusty1s.github.io

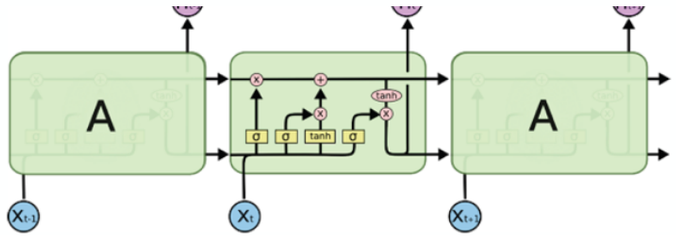
/rusty1s

Introduction

The **modern deep learning toolbox** is designed for simple **sequences and grids**

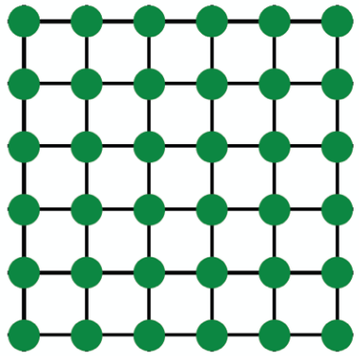


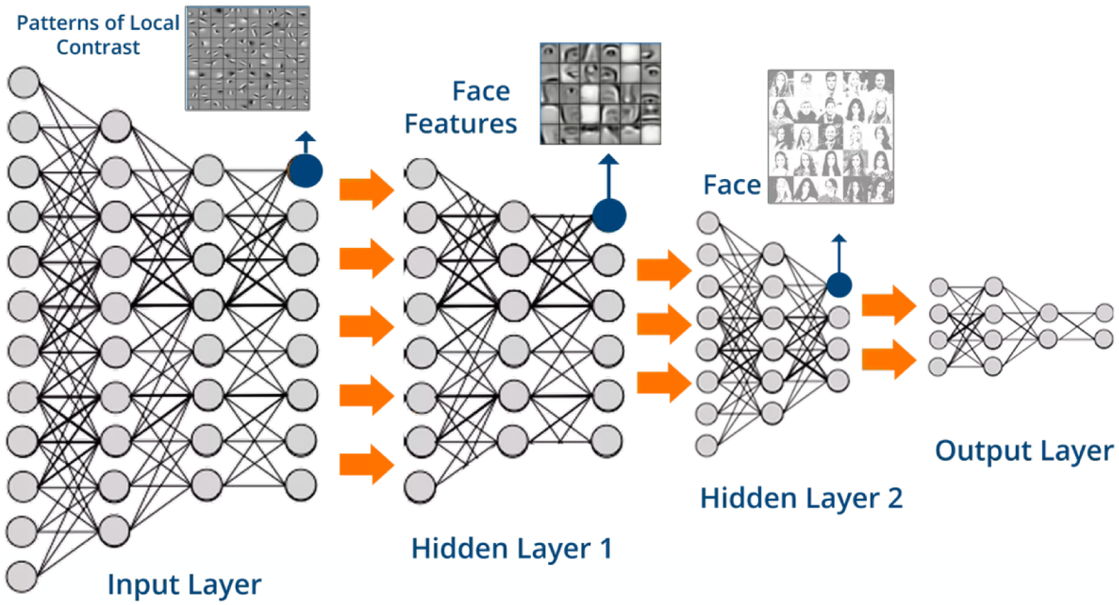


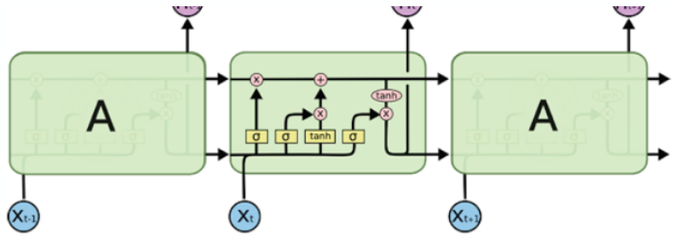


Introduction

The **modern deep learning toolbox** is designed for simple **sequences and grids**







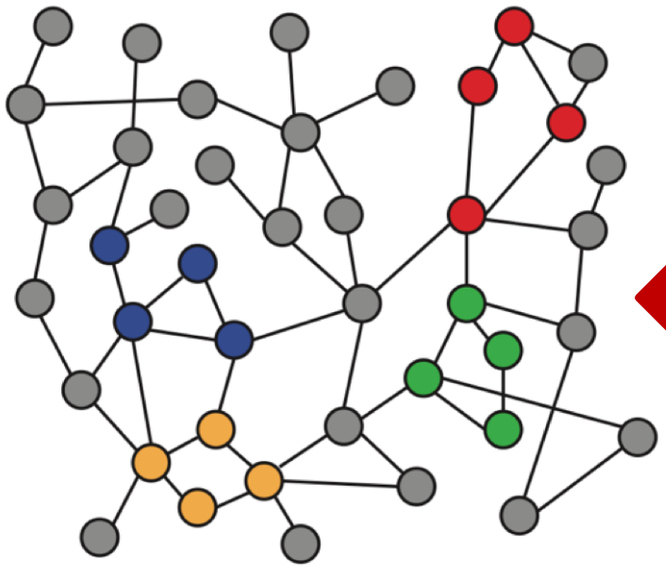
Not everything can be represented as such!

How can we develop neural networks that are much more broadly applicable?

Introduction

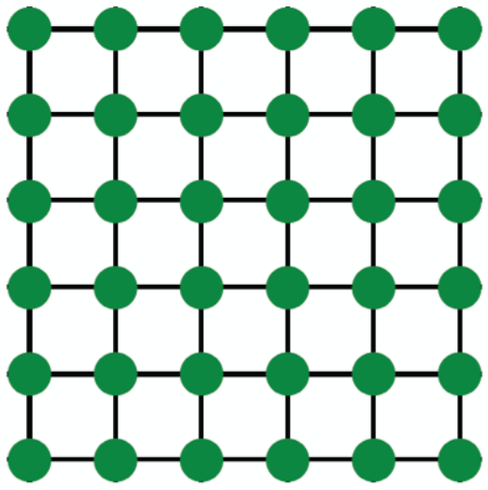
(Mostly) everything can be represented as a **graph**

- Going beyond classic neural networks via **Graph Neural Networks**



Graph

vs.



Image

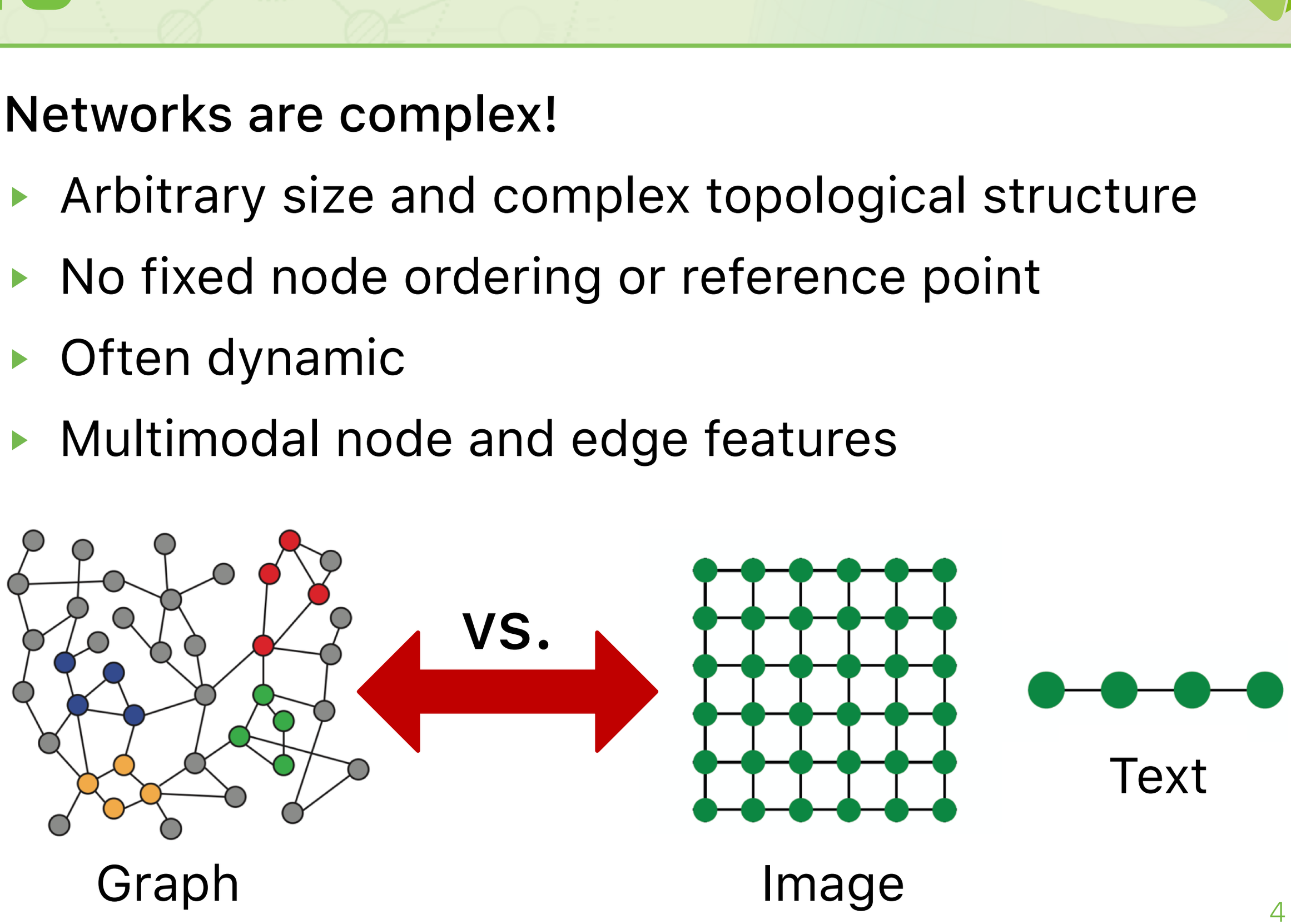
Text

How can we enable learning on arbitrary graphs?

Introduction

Networks are complex!

- Arbitrary size and complex topological structure
- No fixed node ordering or reference point
- Often dynamic
- Multimodal node and edge features



Introduction

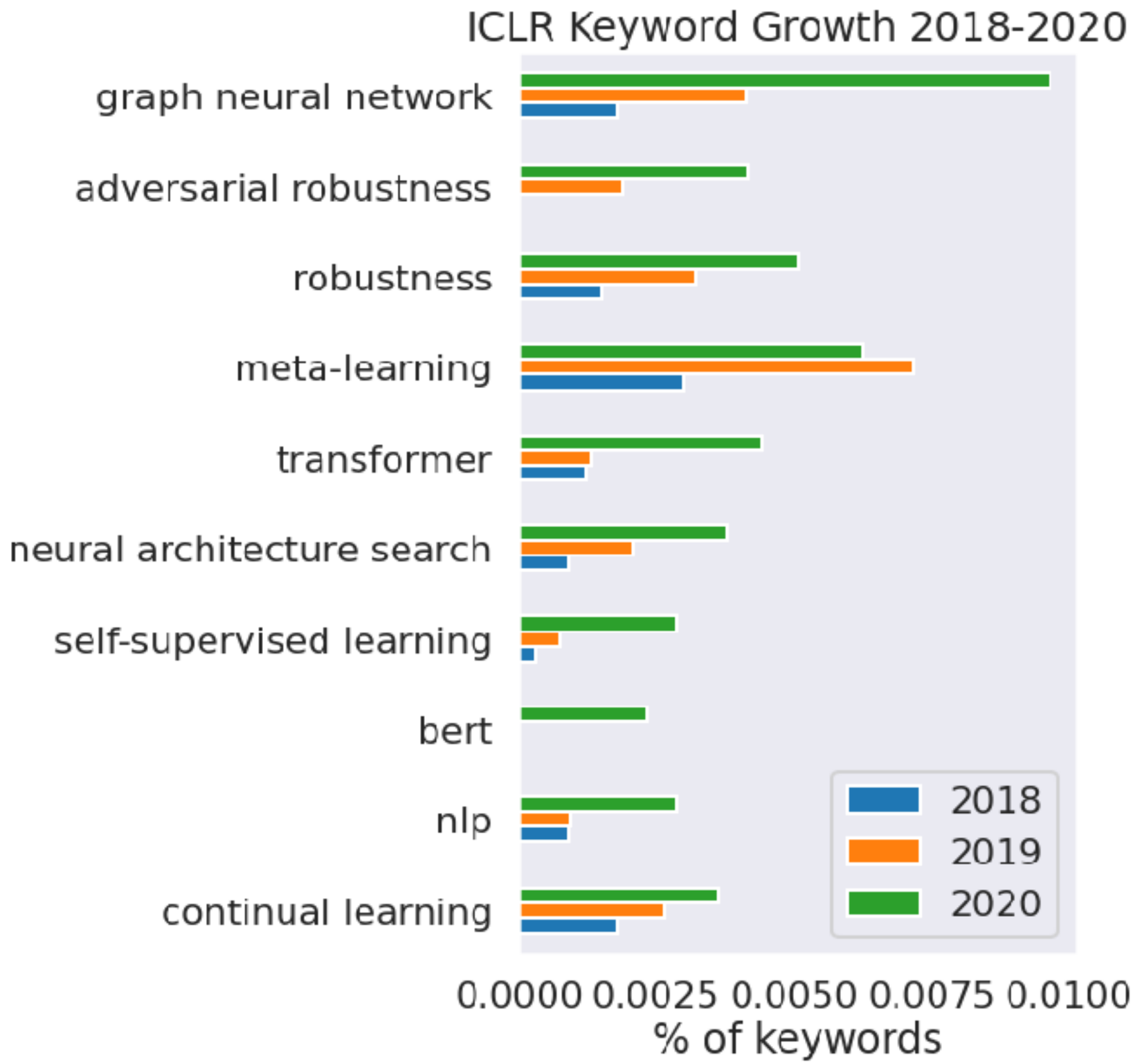
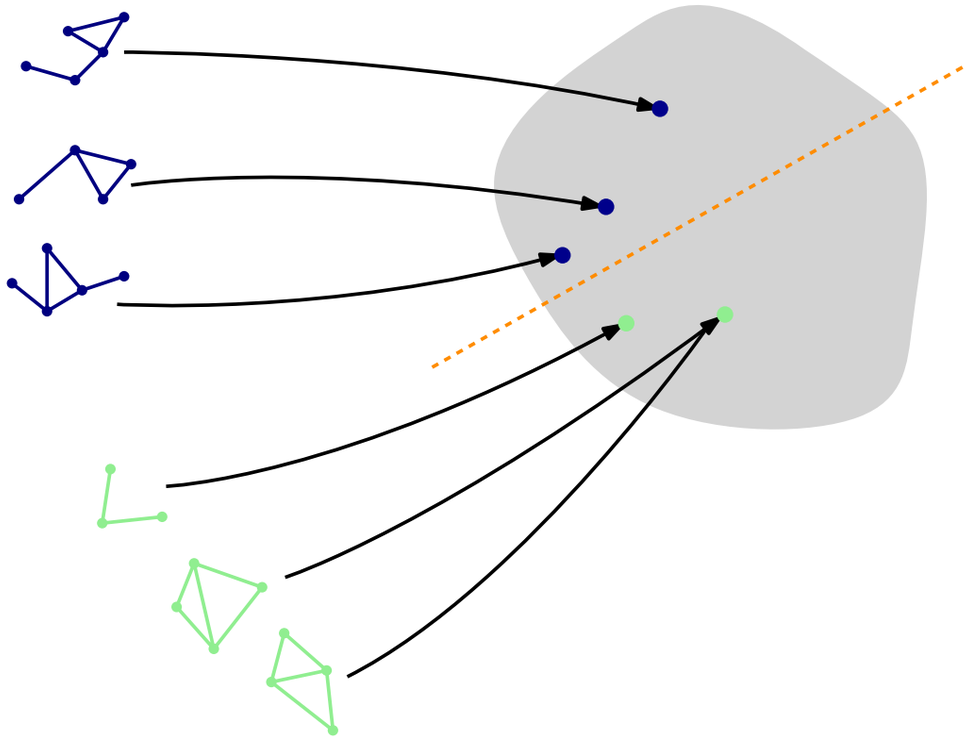


Table of Contents



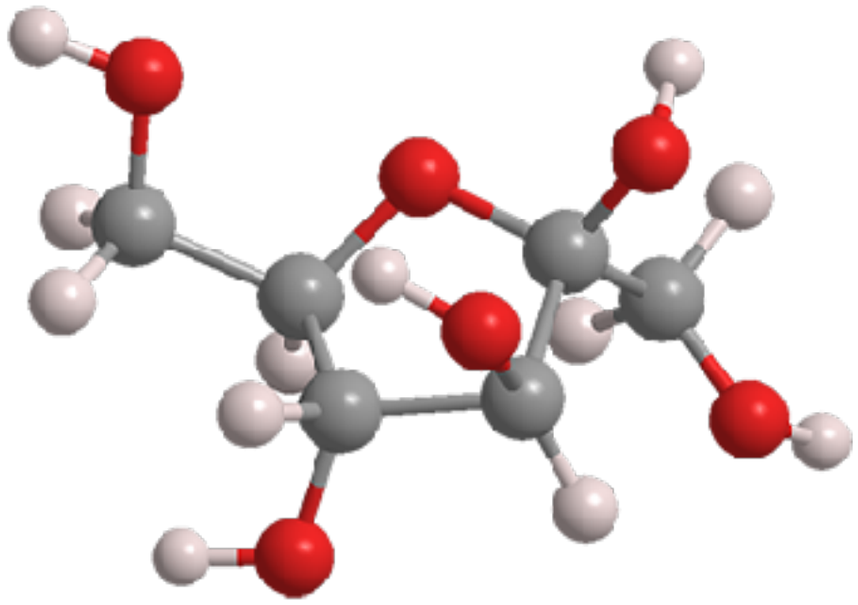
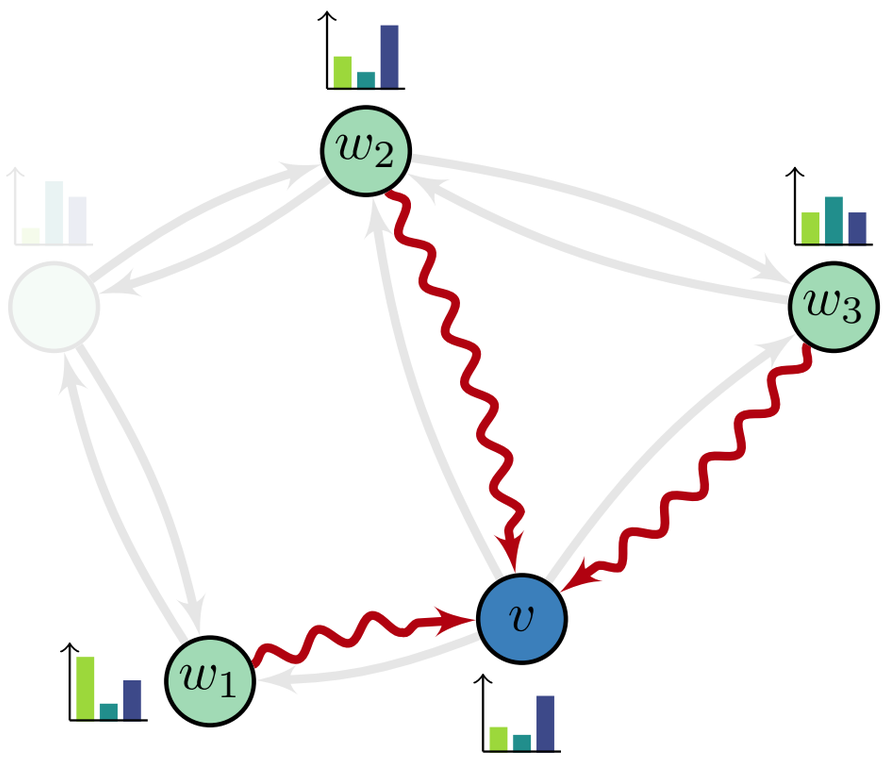


Table of Contents



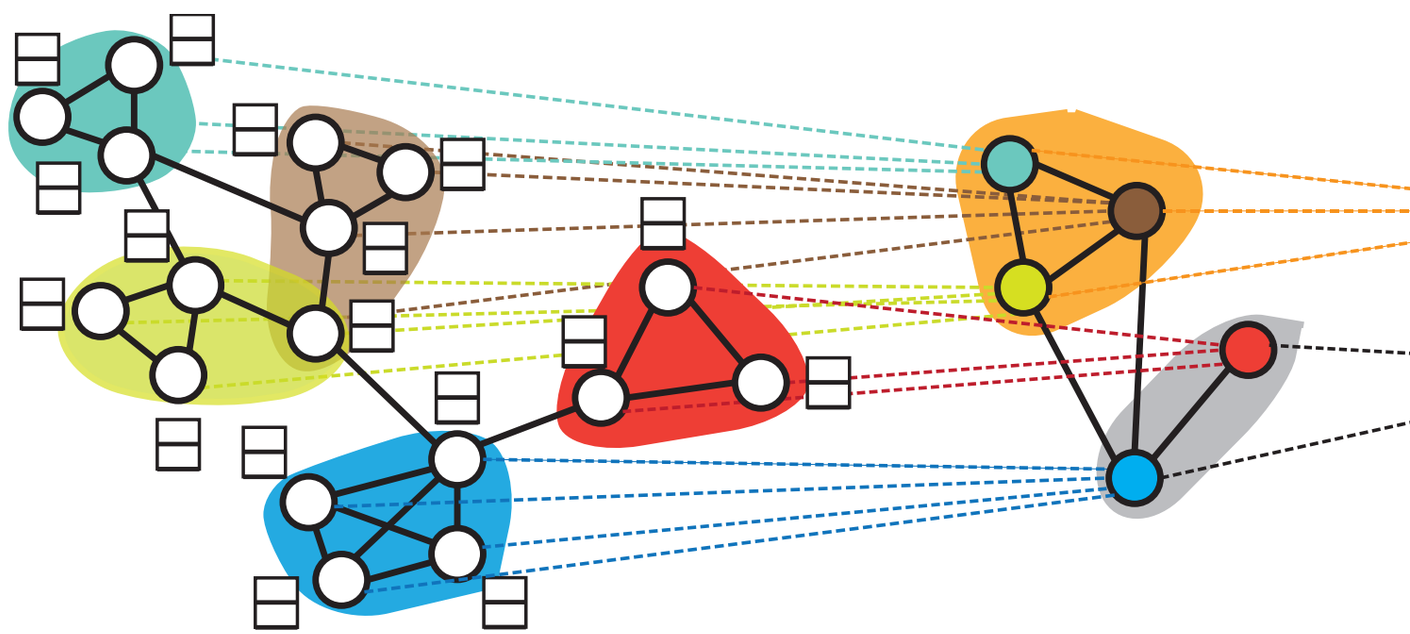


Table of Contents

1. **Learning on Graphs** and its Applications
2. What are **Graph Neural Networks (GNNs)**?
3. **Relational Structure Discovery**
   - *How can we discover objects and their relations?*

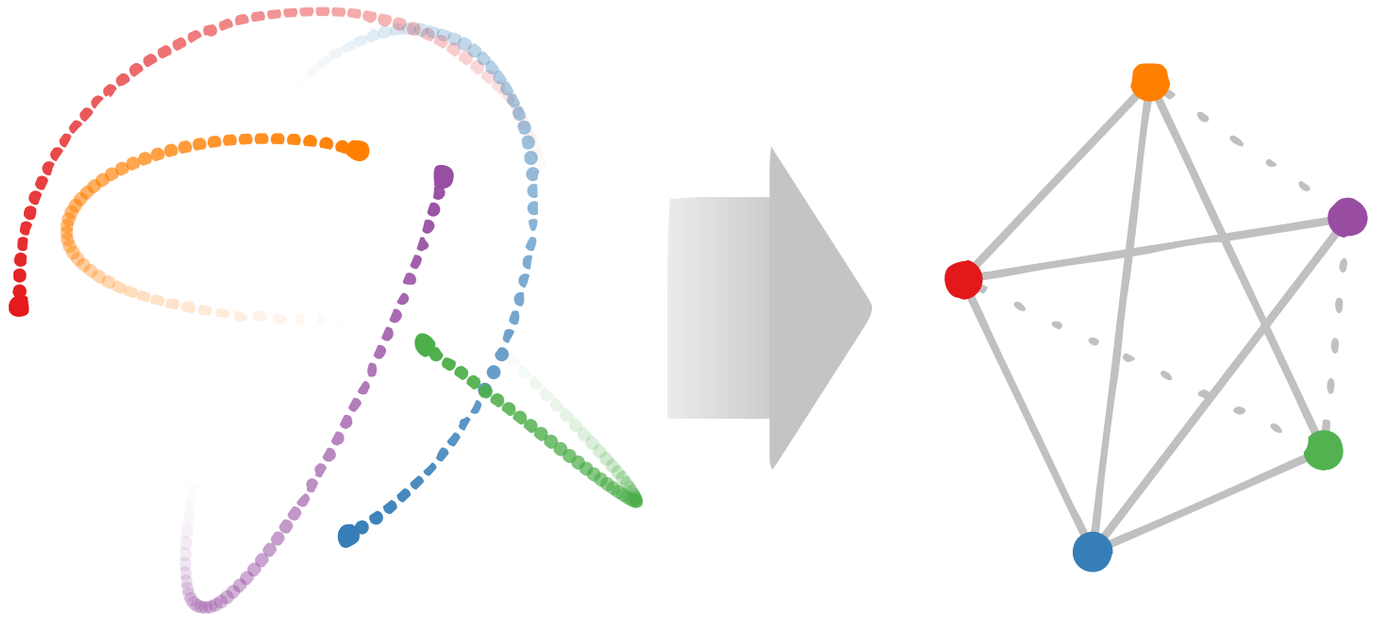
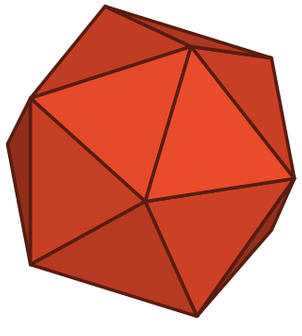


Table of Contents

1. **Learning on Graphs** and its Applications
2. What are **Graph Neural Networks (GNNs)**?
3. **Relational Structure Discovery**
   - *How can we discover objects and their relations?*
4. **Implementing** Graph Neural Networks



PyTorch geometric

Aim: Map nodes/graphs into an **embedding space**, in which they can be distinguished for a given task

Graph: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$

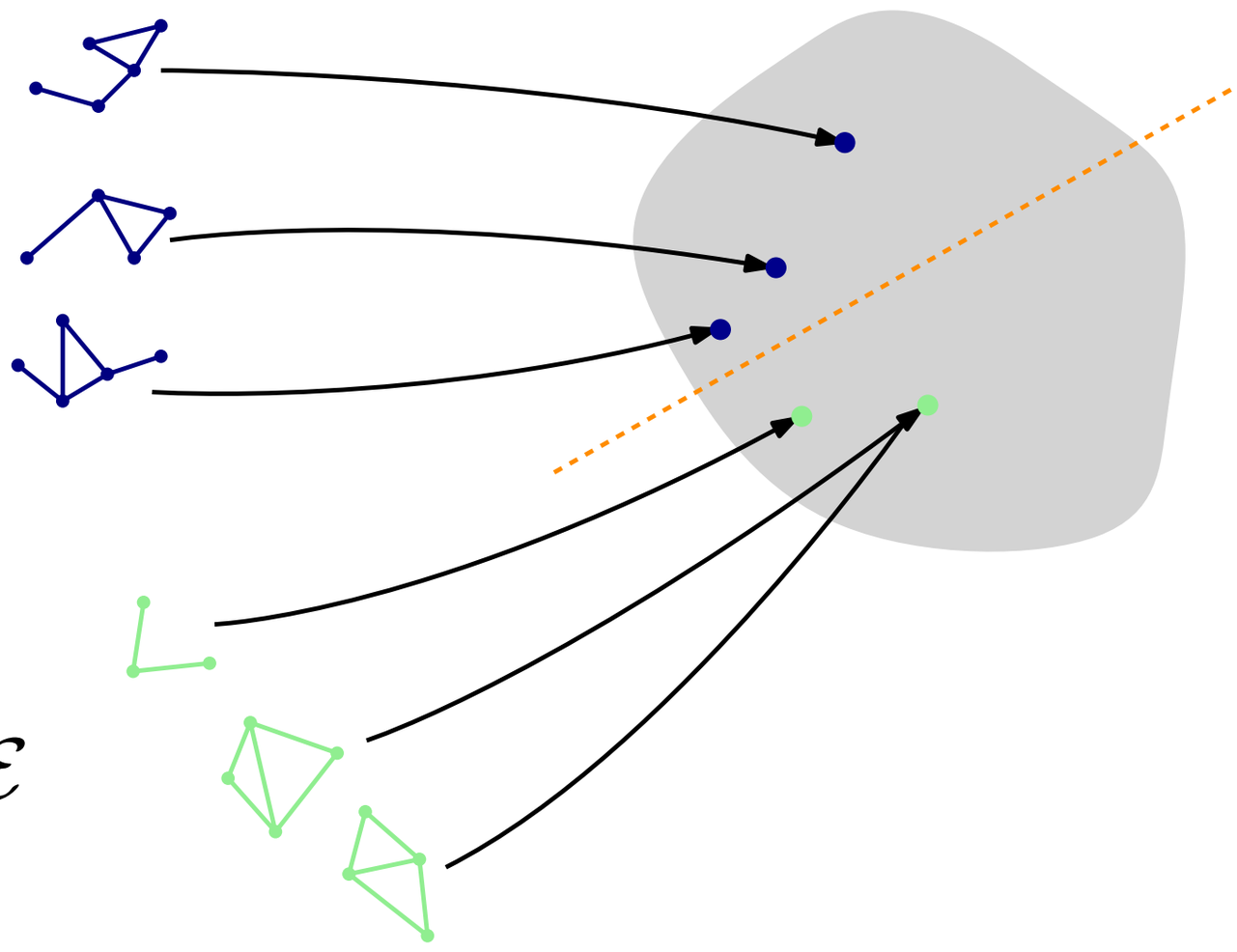
or $\mathbf{A} \in \{0,1\}^{|\mathcal{V}| \times |\mathcal{V}|}$

Node features:

$\mathbf{x}_v \in \mathbb{R}^F$ for all $v \in \mathcal{V}$

Edge features:

$\mathbf{e}_{v,w} \in \mathbb{R}^d$ for all $(v,w) \in \mathcal{E}$



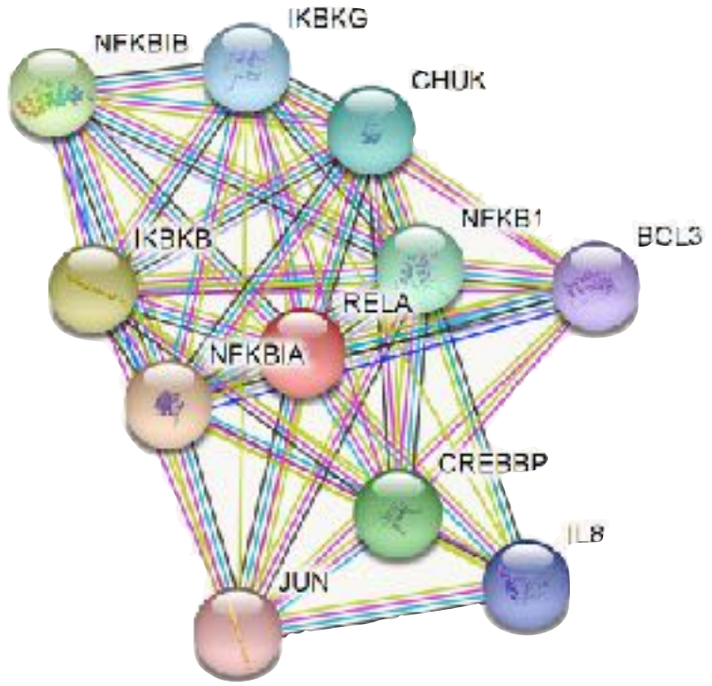
(Non-Exhaustive) Applications

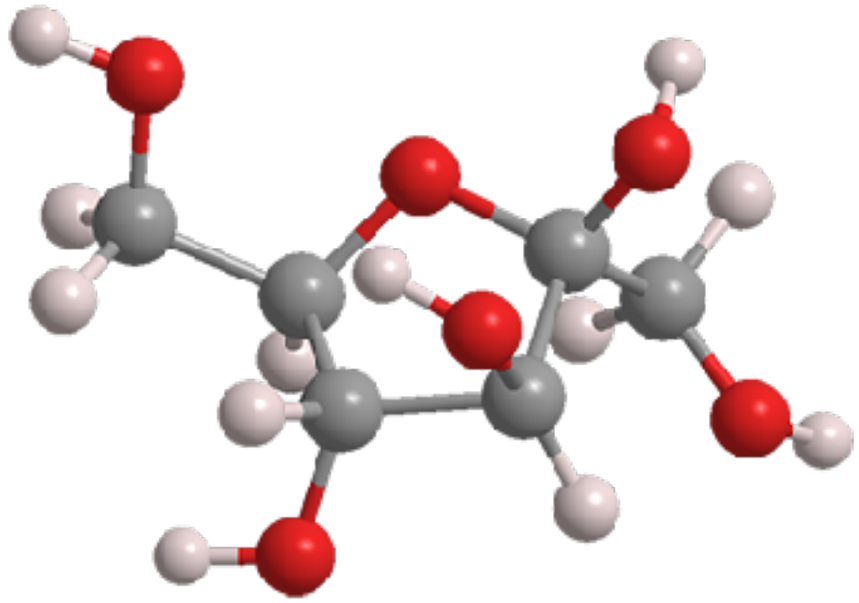
Social Networks

(Non-Exhaustive) Applications

Social Networks

Protein-Protein-Interactions



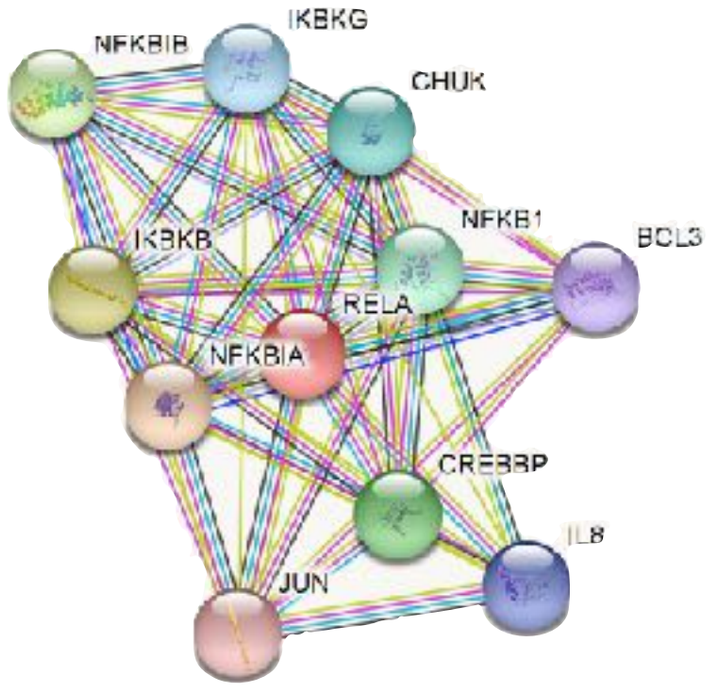


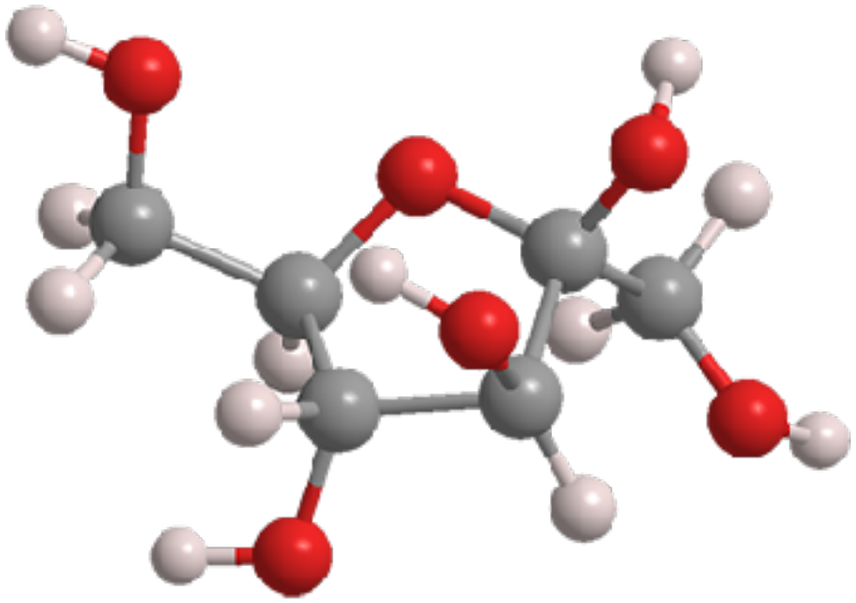
Predicting molecular properties

(Non-Exhaustive) Applications

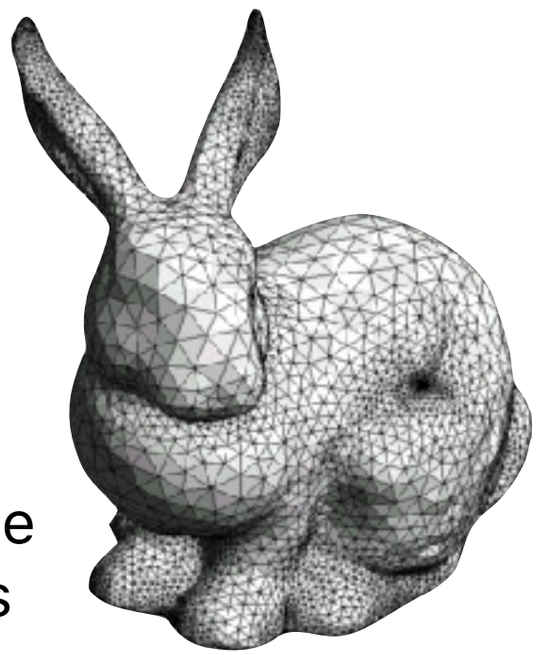
Social Networks

Protein-Protein-Interactions





Predicting molecular properties

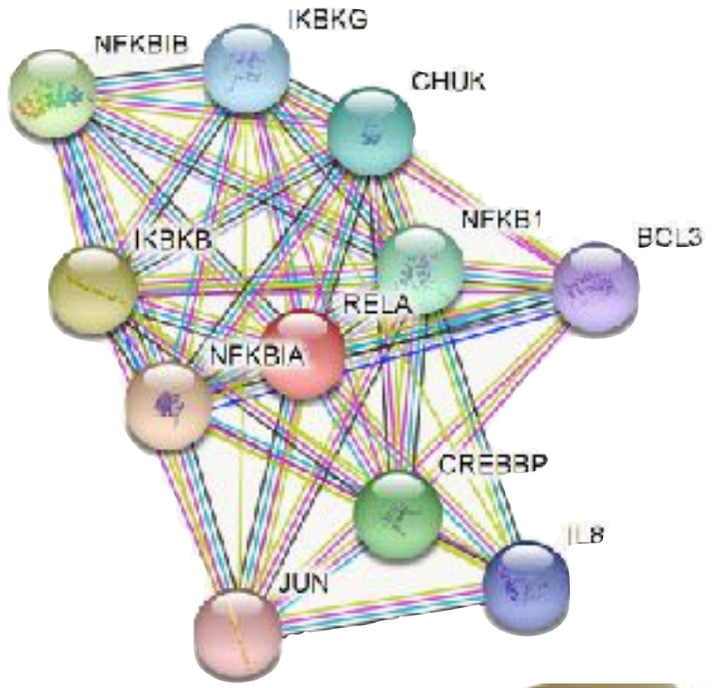


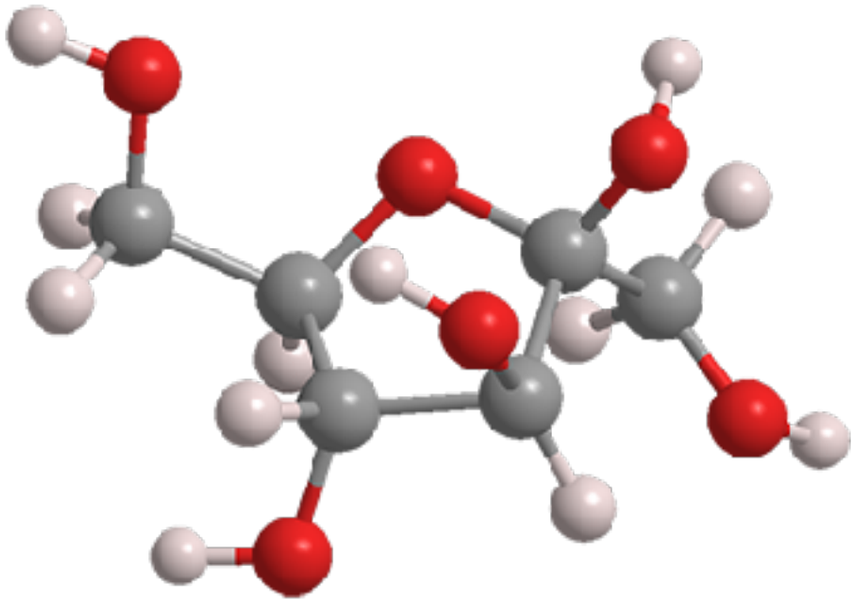
3D Shape Analysis

(Non-Exhaustive) Applications

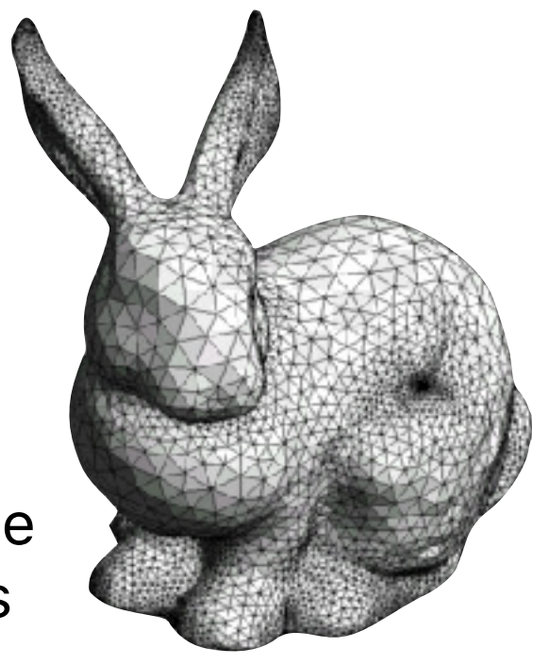
Social Networks

Protein-Protein-Interactions





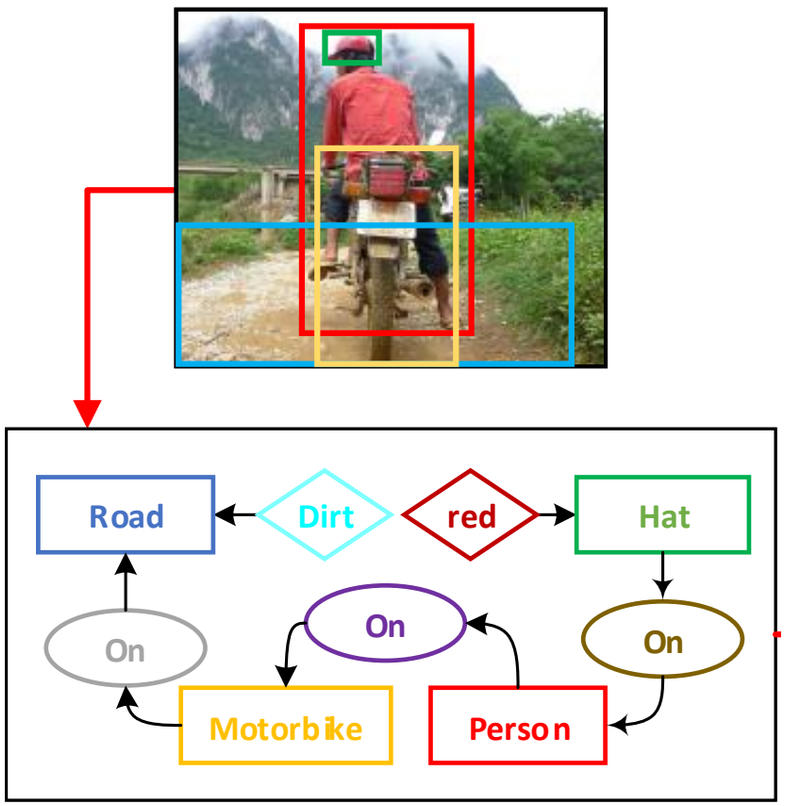
Predicting molecular properties



3D Shape Analysis

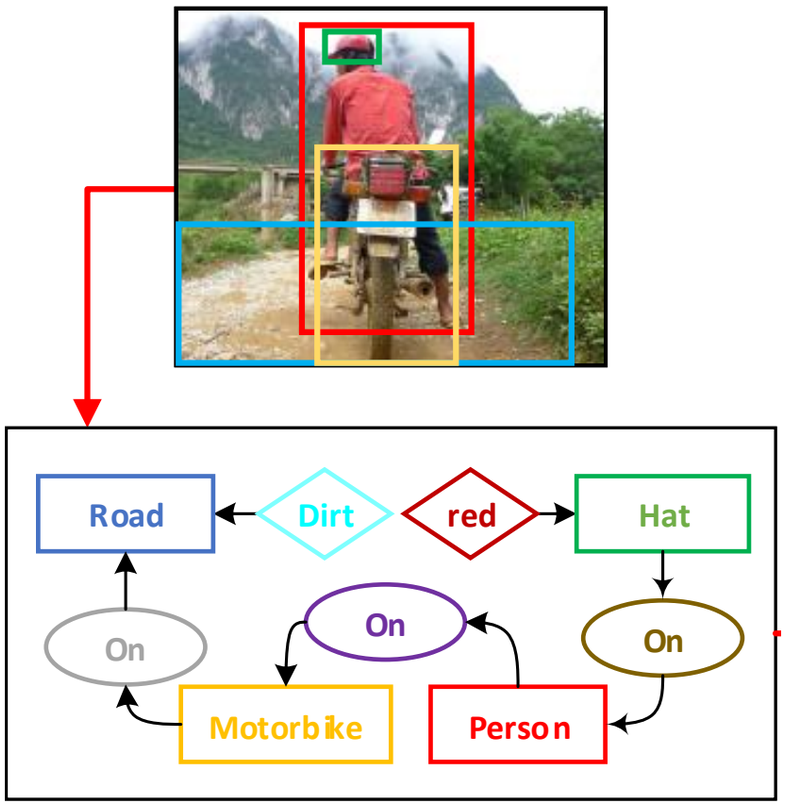
Human Pose Estimation

(Non-Exhaustive) Applications

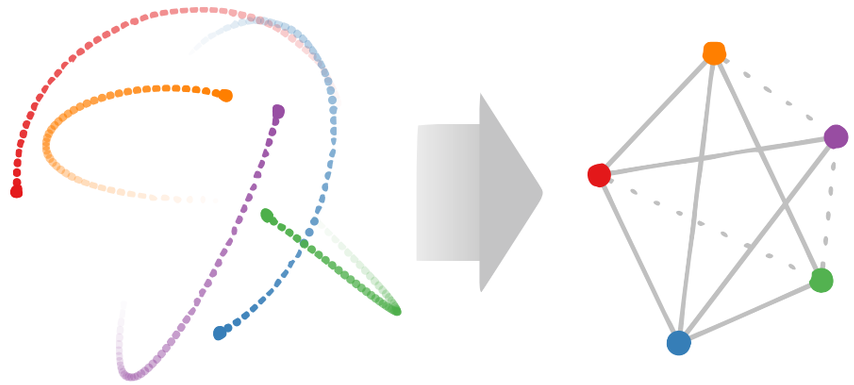


Understanding Scenes

(Non-Exhaustive) Applications

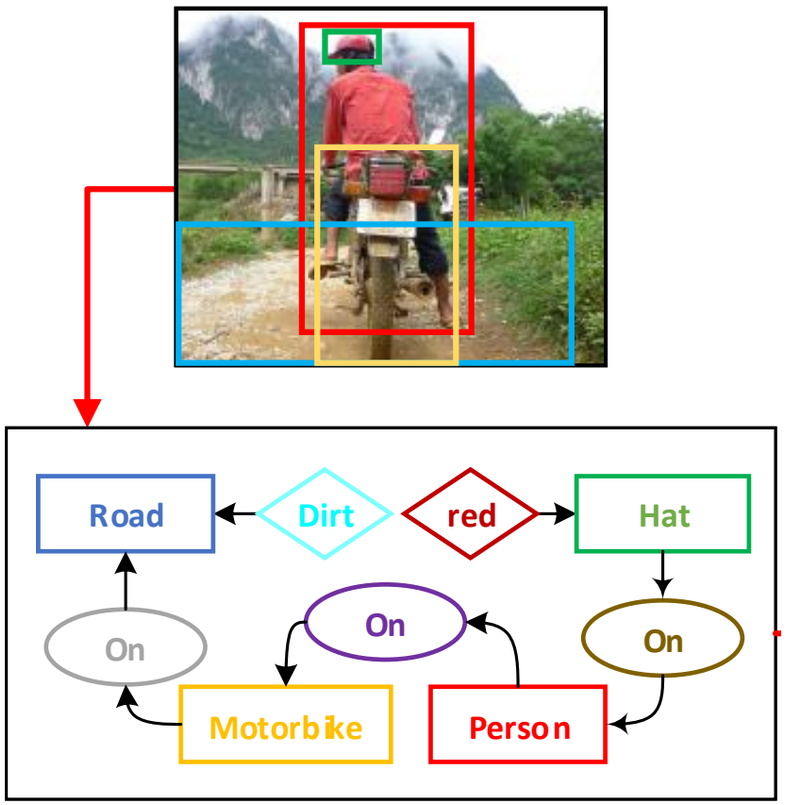


Understanding Scenes

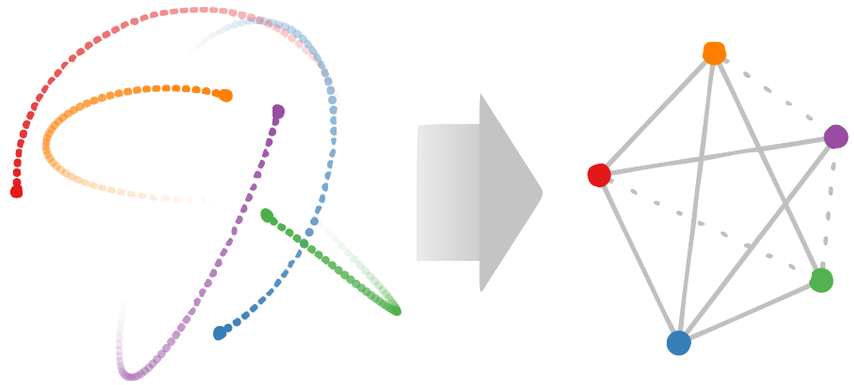


Understanding the dynamics in physical systems

(Non-Exhaustive) Applications

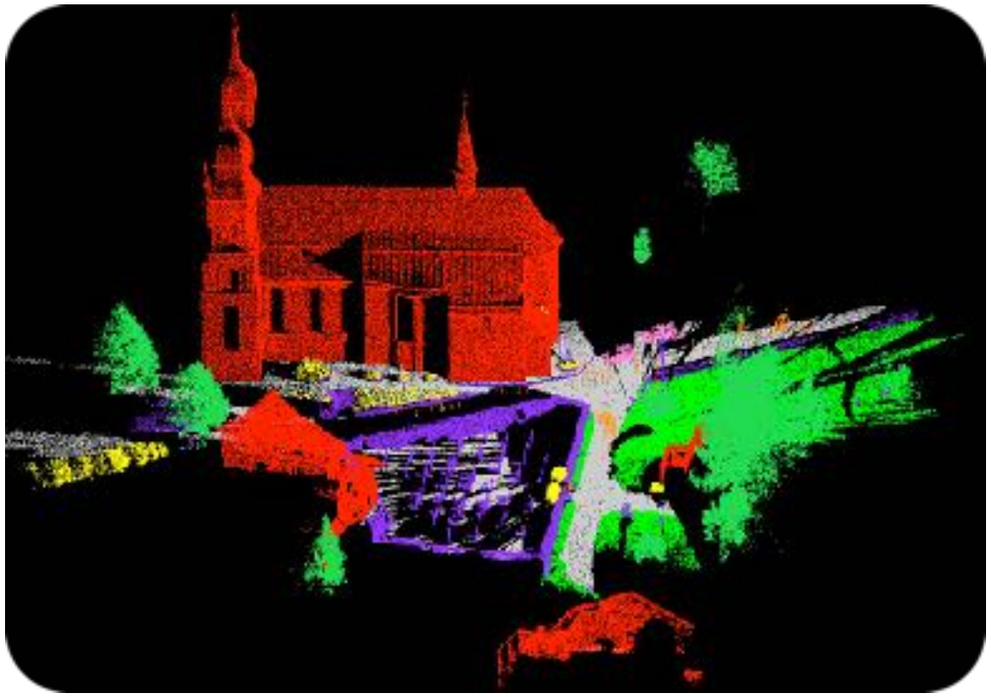


Understanding Scenes



Understanding the dynamics in physical systems

Autonomous driving



Segmenting Point Clouds

Graph Neural Networks (GNNs)

Graph Neural Networks borrow ideas from classical neural networks and **generalize** them to graphs:

Locality

- Learn local patterns of the graph

Weight sharing

- Learn universally applicable patterns of the graph

Graph Neural Networks (GNNs)

Graph Neural Networks borrow ideas from classical neural networks and **generalize** them to graphs:

Locality

- Learn local patterns of the graph

Weight sharing

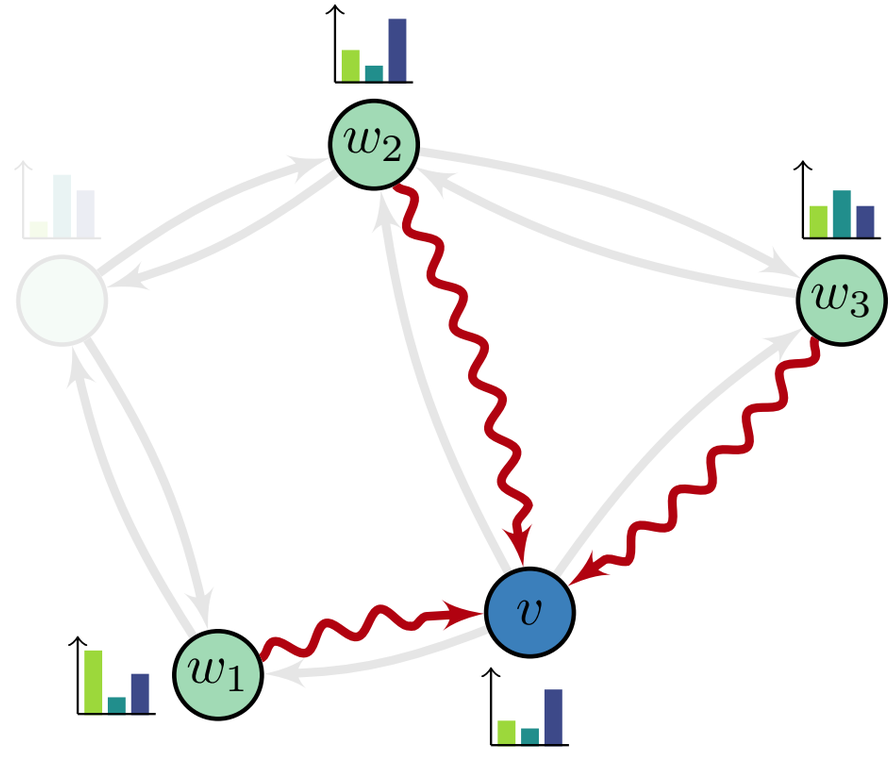
- Learn universally applicable patterns of the graph

A **Graph Neural Network** updates node representations by repeatedly **transforming** and **aggregating neighboring node representations**

Message Passing Scheme

1. Each neighbor sends a **message**:

$$\mathbf{m}_{w,v}^{(\ell)} = \text{MESSAGE}(\mathbf{h}_v^{(\ell-1)}, \mathbf{h}_w^{(\ell-1)})$$



Message Passing Scheme

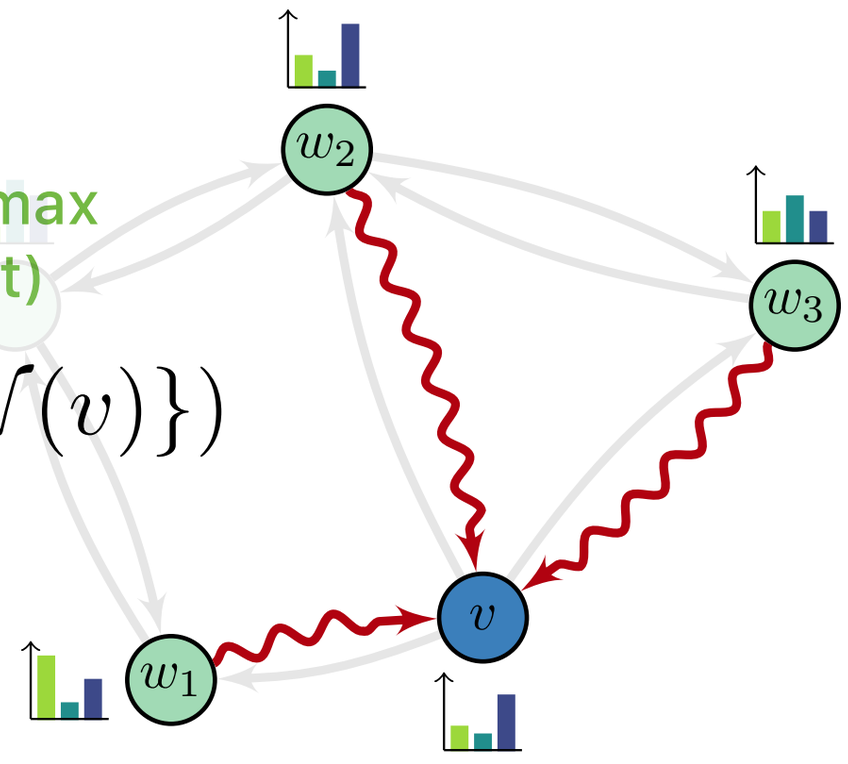
1. Each neighbor sends a **message**:

$$\mathbf{m}_{w,v}^{(\ell)} = \mathrm{MESSAGE}(\mathbf{h}_v^{(\ell-1)}, \mathbf{h}_w^{(\ell-1)})$$

2. Messages are **aggregated** across all neighbors:

typically sum, mean or max (permutation-invariant)

$$\mathbf{a}_v^{(\ell)} = \mathrm{AGGREGATE}(\{\mathbf{m}_{w,v}^{(\ell)} : w \in \mathcal{N}(v)\})$$



Message Passing Scheme

1. Each neighbor sends a **message**:

$$\mathbf{m}_{w,v}^{(\ell)} = \mathrm{MESSAGE}(\mathbf{h}_v^{(\ell-1)}, \mathbf{h}_w^{(\ell-1)})$$

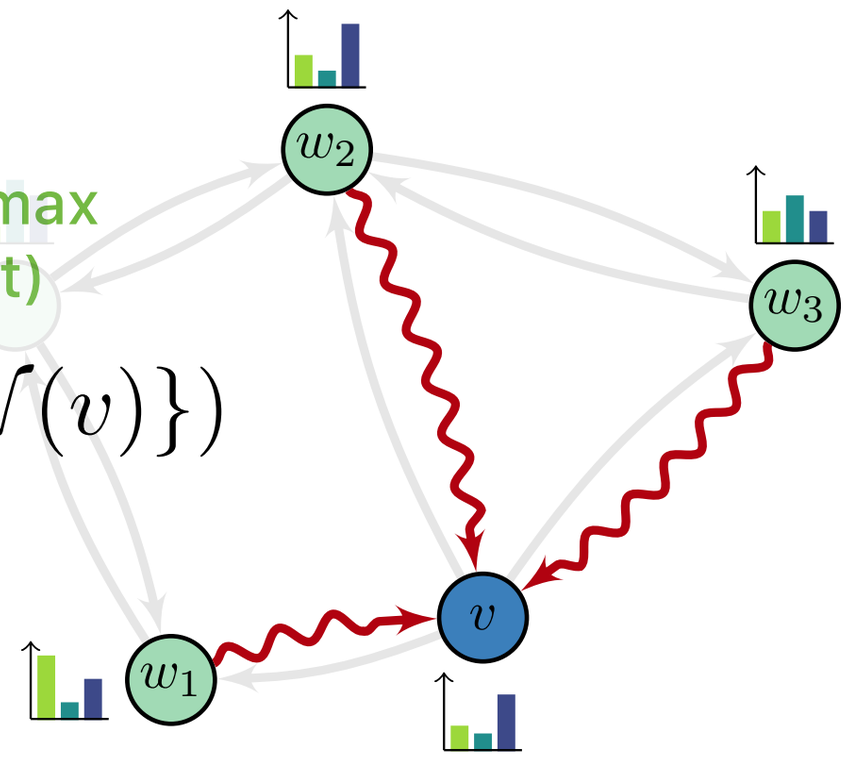
2. Messages are **aggregated** across all neighbors:

typically sum, mean or max (permutation-invariant)

$$\mathbf{a}_v^{(\ell)} = \mathrm{AGGREGATE}(\{\mathbf{m}_{w,v}^{(\ell)} : w \in \mathcal{N}(v)\})$$

3. Neighbor information is used to **update** node represenation:

$$\mathbf{h}_v^{(\ell)} = \mathrm{UPDATE}(\mathbf{h}_v^{(\ell-1)}, \mathbf{a}_v^{(l)})$$

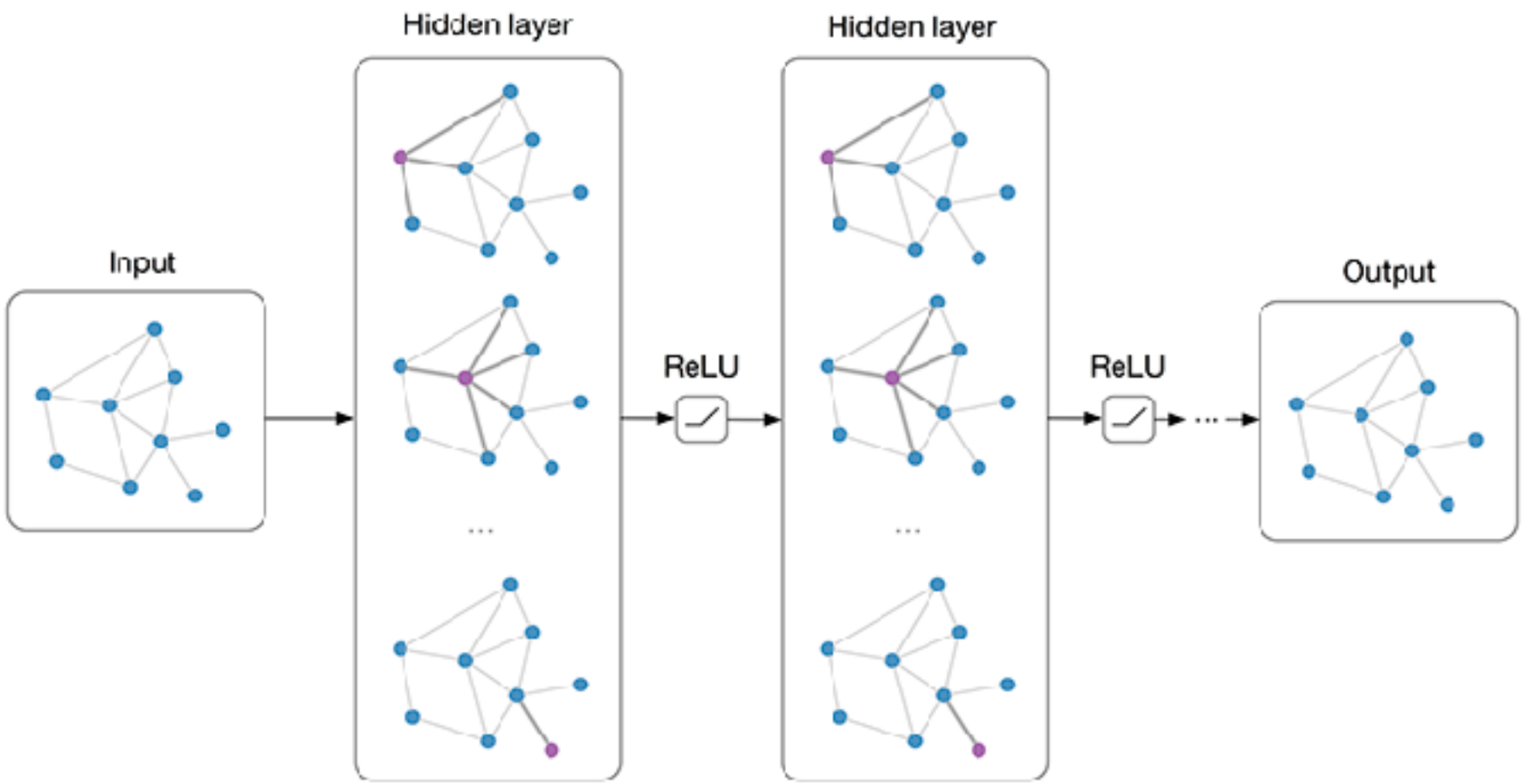


Graph Neural Networks (GNNs)

Message passing functions are **trainable** and **differentiable**

Each node uses the same set of **shared parameters**

Message Passing operators can be **stacked**:
After L layers, a node has aggregated L-hop neighborhood



Graph Neural Networks (GNNs)

Graph Neural Networks compute **localized node embeddings** $\mathbf{h}_v^{(L)} \in \mathbb{R}^{|\mathcal{V}| \times h}$ for a given task:

Node Classification:

Prediction based on node embedding

End-to-end training!

$$\phi(\mathcal{G}, v) = \mathrm{MLP}(\mathbf{h}_v^{(L)})$$

Graph Classification:

Prediction based on **global readout** of node embeddings

$$\phi(\mathcal{G}) = \mathrm{MLP}(\sum_{v \in \mathcal{V}} \mathbf{h}_v^{(L)})$$

Link Prediction:

Prediction based on **pairs** of node embeddings

$$\phi(\mathcal{G}, v, w) = \mathrm{MLP}(\mathbf{h}_v^{(L)}, \mathbf{h}_w^{(L)})$$

Popular Implementations

$$\mathbf{h}_v^{(\ell)} = \sigma\left(\mathbf{W}_1\mathbf{h}_v^{(\ell-1)} + \mathbf{W}_2\sum_{w\in\mathcal{N}(v)} C_{v,w}\mathbf{h}_w^{(\ell-1)}\right)$$

Non-Linearity

Skip-connection

Trainable parameters

Normalization coefficient

Xu et al.: **How Powerful are Graph Neural Networks?** (ICLR '19)
Hamilton et al.: **Inductive Representation Learning on Large Graphs** (NIPS '17)
Kipf and Welling: **Semi-Supervised Classification with Graph Convolutional Networks** (ICLR '17)
Velickovic et al.: **Graph Attention Networks** (ICLR '18)

Popular Implementations

$$\mathbf{h}_v^{(\ell)} = \sigma\left(\mathbf{W}_1\mathbf{h}_v^{(\ell-1)} + \mathbf{W}_2\sum_{w\in\mathcal{N}(v)} C_{v,w}\mathbf{h}_w^{(\ell-1)}\right)$$

Non-Linearity

Skip-connection

Trainable parameters

Normalization coefficient

Normalization can be either ...

... static: $C_{v,w} = 1$

Xu et al.: **How Powerful are Graph Neural Networks?** (ICLR '19)
Hamilton et al.: **Inductive Representation Learning on Large Graphs** (NIPS '17)
Kipf and Welling: **Semi-Supervised Classification with Graph Convolutional Networks** (ICLR '17)
Velickovic et al.: **Graph Attention Networks** (ICLR '18)

Popular Implementations

$$\mathbf{h}_v^{(\ell)} = \sigma\left(\mathbf{W}_1\mathbf{h}_v^{(\ell-1)} + \mathbf{W}_2\sum_{w\in\mathcal{N}(v)} C_{v,w}\mathbf{h}_w^{(\ell-1)}\right)$$

Non-Linearity

Skip-connection

Trainable parameters

Normalization coefficient

Normalization can be either ...

... static: $C_{v,w} = 1$

... structure-dependent: $C_{v,w} = |\mathcal{N}(v)|^{-1}$

Xu et al.: **How Powerful are Graph Neural Networks?** (ICLR '19)
Hamilton et al.: **Inductive Representation Learning on Large Graphs** (NIPS '17)
Kipf and Welling: **Semi-Supervised Classification with Graph Convolutional Networks** (ICLR '17)
Velickovic et al.: **Graph Attention Networks** (ICLR '18)

Popular Implementations

$$\mathbf{h}_v^{(\ell)} = \sigma\left(\mathbf{W}_1\mathbf{h}_v^{(\ell-1)} + \mathbf{W}_2\sum_{w\in\mathcal{N}(v)} C_{v,w}\mathbf{h}_w^{(\ell-1)}\right)$$

Non-Linearity

Skip-connection

Trainable parameters

Normalization coefficient

Normalization can be either ...

... static: $C_{v,w} = 1$

... structure-dependent: $C_{v,w} = |\mathcal{N}(v)|^{-1}$

... data-dependent (learned), aka **Attention**

Xu et al.: **How Powerful are Graph Neural Networks?** (ICLR '19)
Hamilton et al.: **Inductive Representation Learning on Large Graphs** (NIPS '17)
Kipf and Welling: **Semi-Supervised Classification with Graph Convolutional Networks** (ICLR '17)
Velickovic et al.: **Graph Attention Networks** (ICLR '18)

Popular Implementations

The message passing scheme is really flexible:

It can model **anisotropic transformations:**

$$\text{MESSAGE}(\mathbf{h}_v^{(\ell)}, \mathbf{h}_w^{(\ell)}) = \text{MLP}(\mathbf{h}_v^{(\ell)}, \mathbf{h}_w^{(\ell)} - \mathbf{h}_v^{(\ell)})$$

Wang et al.: **Dynamic Graph CNN for Learning on Point Clouds?** ('19)
Schlichtkrull et al.: **Modeling Relational Data with Graph Convolutional Networks** ('17)
Fey et al.: **SplineCNN: Fast Geometric Deep Learning with Continuous B-Spline Kernels** (CVPR '18)

Popular Implementations

The message passing scheme is really flexible:

It can model **anisotropic transformations:**

$$\text{MESSAGE}(\mathbf{h}_v^{(\ell)}, \mathbf{h}_w^{(\ell)}) = \text{MLP}(\mathbf{h}_v^{(\ell)}, \mathbf{h}_w^{(\ell)} - \mathbf{h}_v^{(\ell)})$$

It can model different **edge relation types:**

$$\text{MESSAGE}(\mathbf{h}_v^{(\ell)}, \mathbf{h}_w^{(\ell)}) = \mathbf{W}_{R(v,w)}\mathbf{h}_w^{(\ell)}$$

Wang et al.: **Dynamic Graph CNN for Learning on Point Clouds?** ('19)
Schlichtkrull et al.: **Modeling Relational Data with Graph Convolutional Networks** ('17)
Fey et al.: **SplineCNN: Fast Geometric Deep Learning with Continuous B-Spline Kernels** (CVPR '18)

Popular Implementations

The message passing scheme is really flexible:

It can model **anisotropic transformations:**

$$\text{MESSAGE}(\mathbf{h}_v^{(\ell)}, \mathbf{h}_w^{(\ell)}) = \text{MLP}(\mathbf{h}_v^{(\ell)}, \mathbf{h}_w^{(\ell)} - \mathbf{h}_v^{(\ell)})$$

It can model different **edge relation types:**

$$\text{MESSAGE}(\mathbf{h}_v^{(\ell)}, \mathbf{h}_w^{(\ell)}) = \mathbf{W}_{R(v,w)}\mathbf{h}_w^{(\ell)}$$

It can incorporate **edge features:**

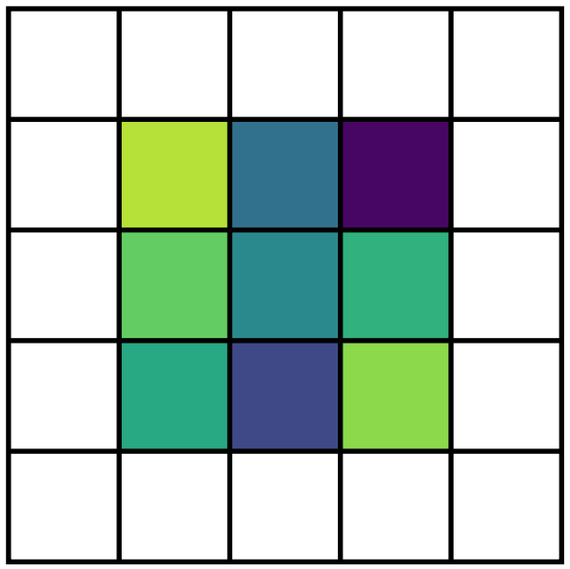
$$\text{MESSAGE}(\mathbf{h}_w^{(\ell)}, \mathbf{e}_{w,v}) = \text{MLP}(\mathbf{e}_{w,v})\mathbf{h}_w^{(\ell)}$$

Wang et al.: **Dynamic Graph CNN for Learning on Point Clouds?** ('19)
Schlichtkrull et al.: **Modeling Relational Data with Graph Convolutional Networks** ('17)
Fey et al.: **SplineCNN: Fast Geometric Deep Learning with Continuous B-Spline Kernels** (CVPR '18)

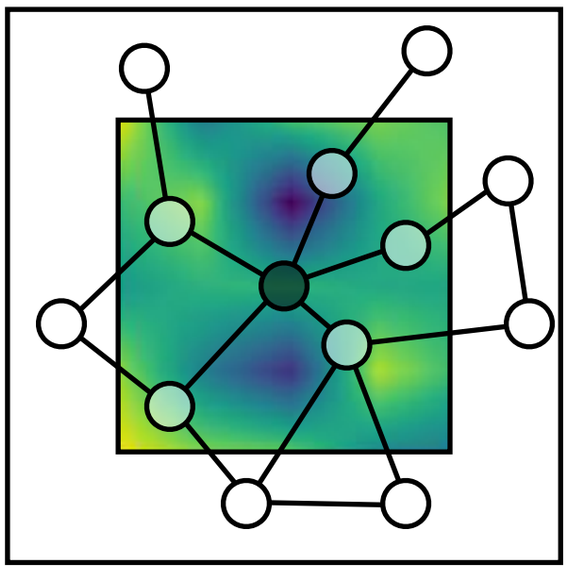
Popular Implementations

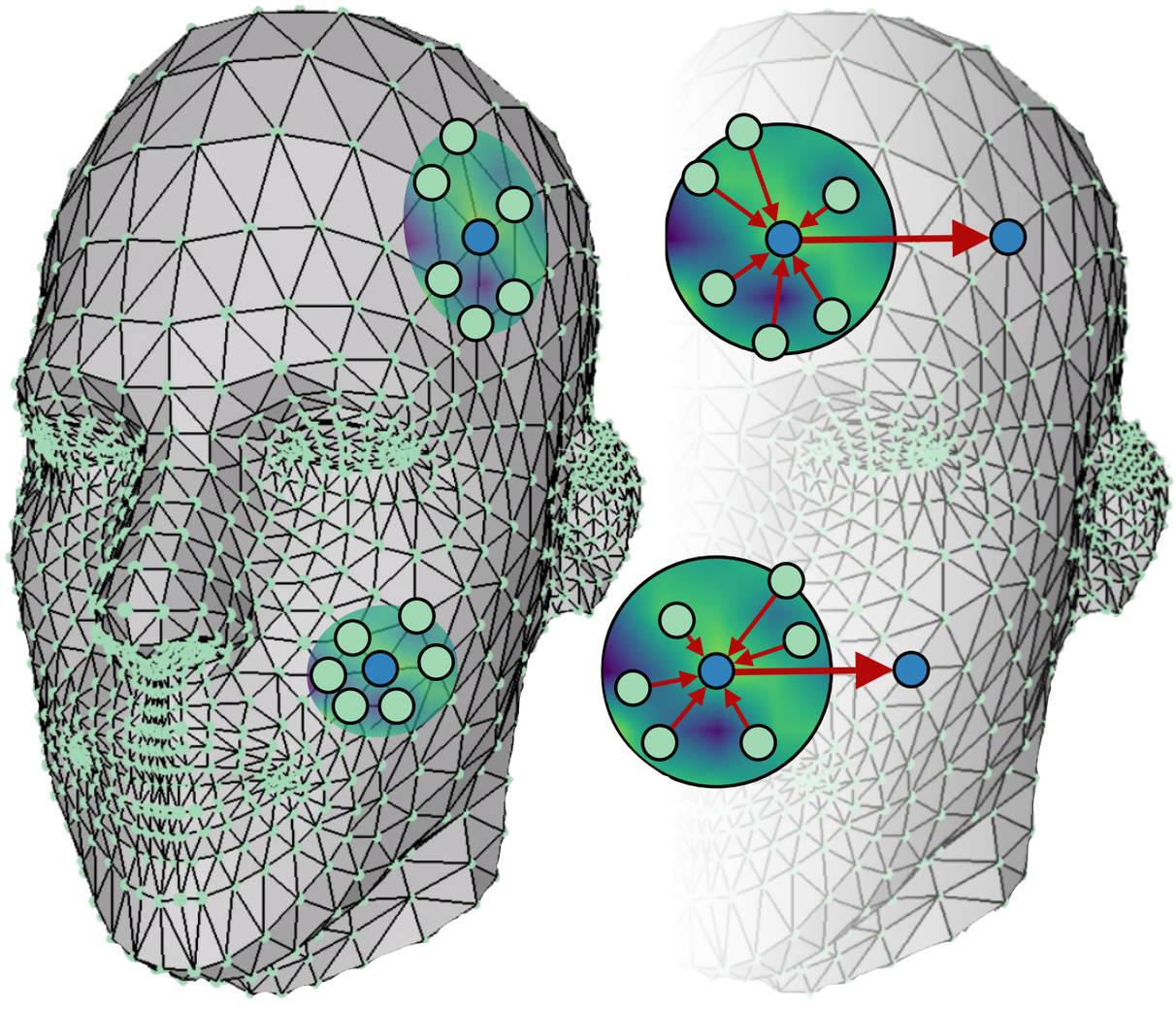
$$\text{MESSAGE}(\mathbf{h}_w^{(\ell)}, \mathbf{e}_{w,v}) = \text{MLP}(\mathbf{e}_{w,v})\mathbf{h}_w^{(\ell)}$$

CNN



GNN





Relational Structure Discovery

Assumption: We know what the **nodes** and **edges** are

Kipf and Welling: **Graph Auto-Encoders** (NIPS-W '16)
Schlichtkrull et al.: **Knowledge Graph Completion** ('17)
Kipf et al.: **Neural Relational Inference for Interacting Systems** (ICML '18)
Kipf et al.: **Contrastive Learning of Structured World Models** (ICLR '20)

Relational Structure Discovery

Assumption: We know what the **nodes** and **edges** are

What if we have incomplete knowledge about relations?

- *Link prediction (graph completion)*
- *Infer links from scratch (relational inference)*

Kipf and Welling: **Graph Auto-Encoders** (NIPS-W '16)
Schlichtkrull et al.: **Knowledge Graph Completion** ('17)
Kipf et al.: **Neural Relational Inference for Interacting Systems** (ICML '18)
Kipf et al.: **Contrastive Learning of Structured World Models** (ICLR '20)

Relational Structure Discovery

Assumption: We know what the **nodes** and **edges** are

What if we have incomplete knowledge about relations?

- *Link prediction (graph completion)*
- *Infer links from scratch (relational inference)*

What if we do not even know about nodes?

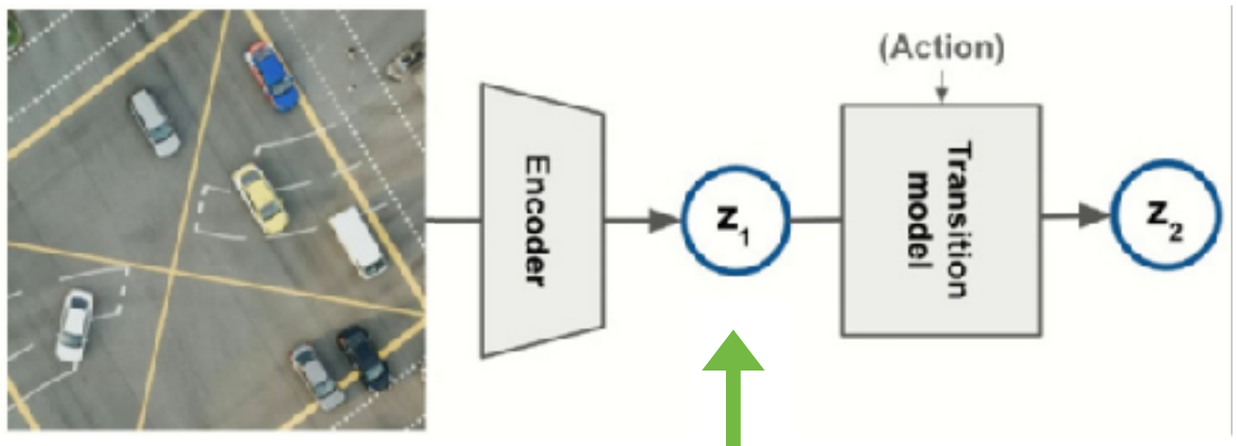
- *Object / entity / event discovery*
- *Grouping / clustering of low-level features*

Kipf and Welling: **Graph Auto-Encoders** (NIPS-W '16)
Schlichtkrull et al.: **Knowledge Graph Completion** ('17)
Kipf et al.: **Neural Relational Inference for Interacting Systems** (ICML '18)
Kipf et al.: **Contrastive Learning of Structured World Models** (ICLR '20)

What if we do not even know about nodes?

What if we do not even know about nodes?

"Unstructured"
World Model



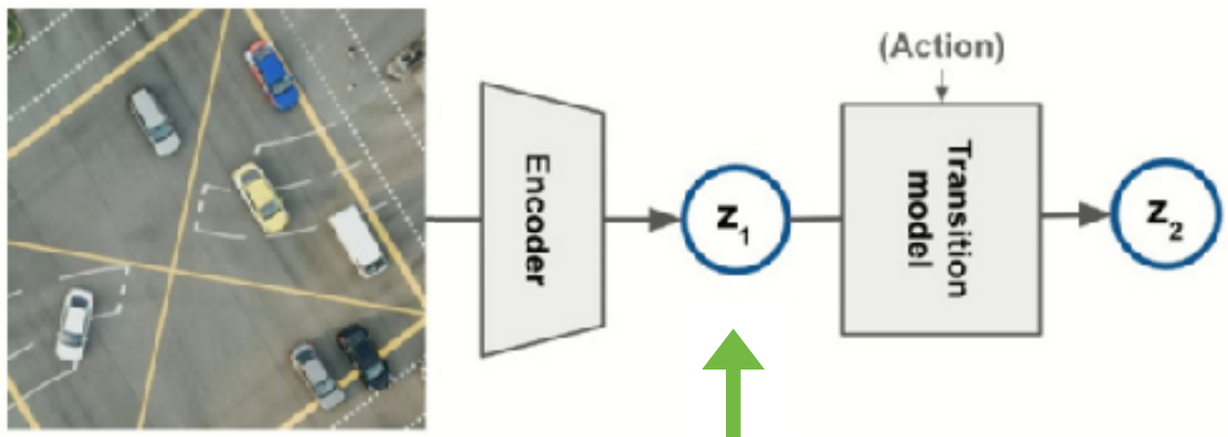
$t = 1$ $t = 2$

Representation is
fully distributed
and **unstructured**

Object-Centric Learning

What if we do not even know about nodes?

"Unstructured" World Model



$t = 1$ $t = 2$

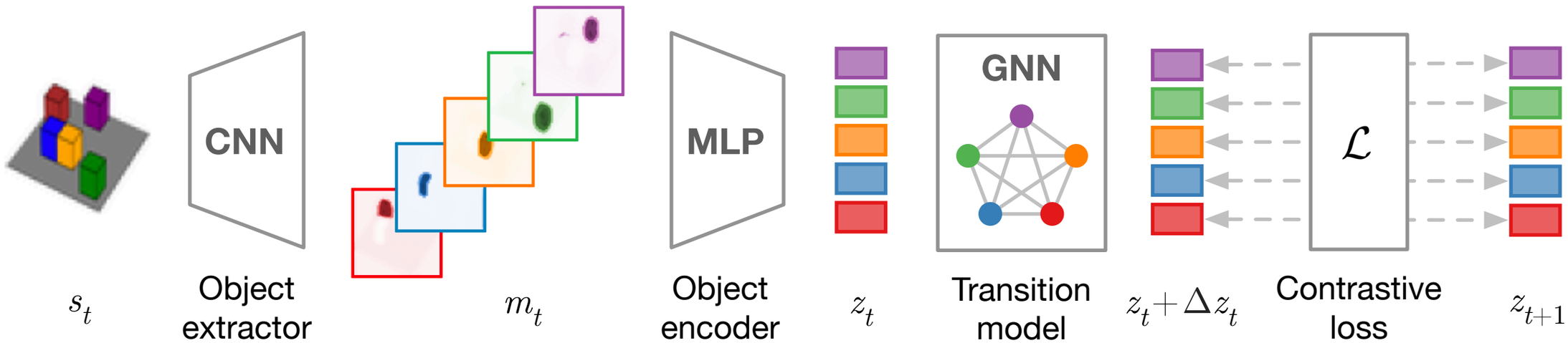
Representation is **fully distributed** and **unstructured**

"Structured" World Model

$t = 1$ $t = 2$

Representation is **fully distributed**, but **structured as a set**

Fully self-supervised "structured" world models



Encoder:

- **"Object discovery" module** that maps observations to a **set of vectors**

Transition model:

- Models **pairwise interactions** between objects using a **GNN**

Training: Using a **contrastive loss** in **latent space** based on a **random state** in the dataset

Implementing GNNs

Implementing GNNs

Implementing GNNs

Graph Neural Networks introduce their own unique challenges regarding implementation

Most of the deep learning frameworks are designed for dense representations!

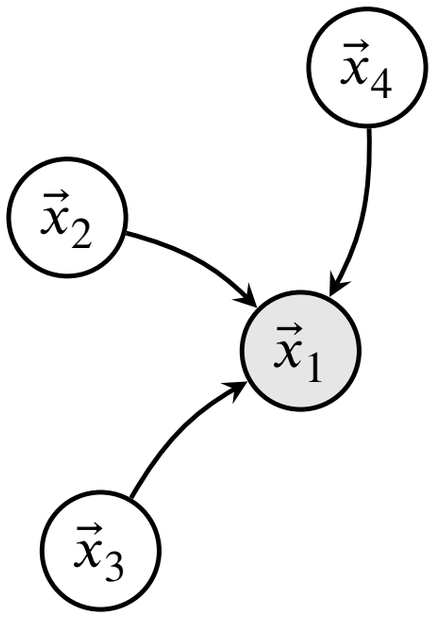
- *How can we achieve* ***effective parallelization*** *on* ***irregular*** *and* ***sparse graphs****?*
- *How can we provide* **mini-batching capabilities** *for graphs of* ***potentially different sizes****?*

Implementing GNNs

Given a *sparse* graph $\mathcal{G} = (\mathbf{X}, (\mathbf{I}, \mathbf{E}))$ with

- node features $\mathbf{X} \in \mathbb{R}^{|\mathcal{V}| \times F}$
- edge indices $\mathbf{I} \in \{1, \ldots, N\}^{2 \times |\mathcal{E}|}$
- *optional* edge features $\mathbf{E} \in \mathbb{R}^{|\mathcal{E}| \times D}$

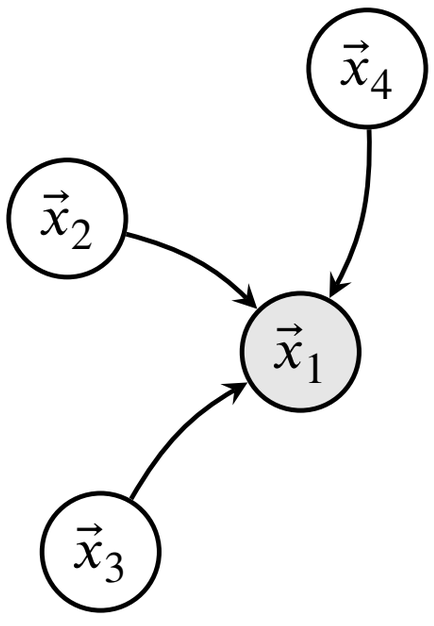
$$\mathbf{X} = \begin{bmatrix} \vec{x}_1 \\ \vec{x}_2 \\ \vec{x}_3 \\ \vec{x}_4 \end{bmatrix}$$



$$\mathbf{I} = \begin{bmatrix} 2 & 1 \\ 3 & 1 \\ 4 & 1 \end{bmatrix}^\top$$

Given a *sparse* graph $\mathcal{G} = (\mathbf{X}, (\mathbf{I}, \mathbf{E}))$

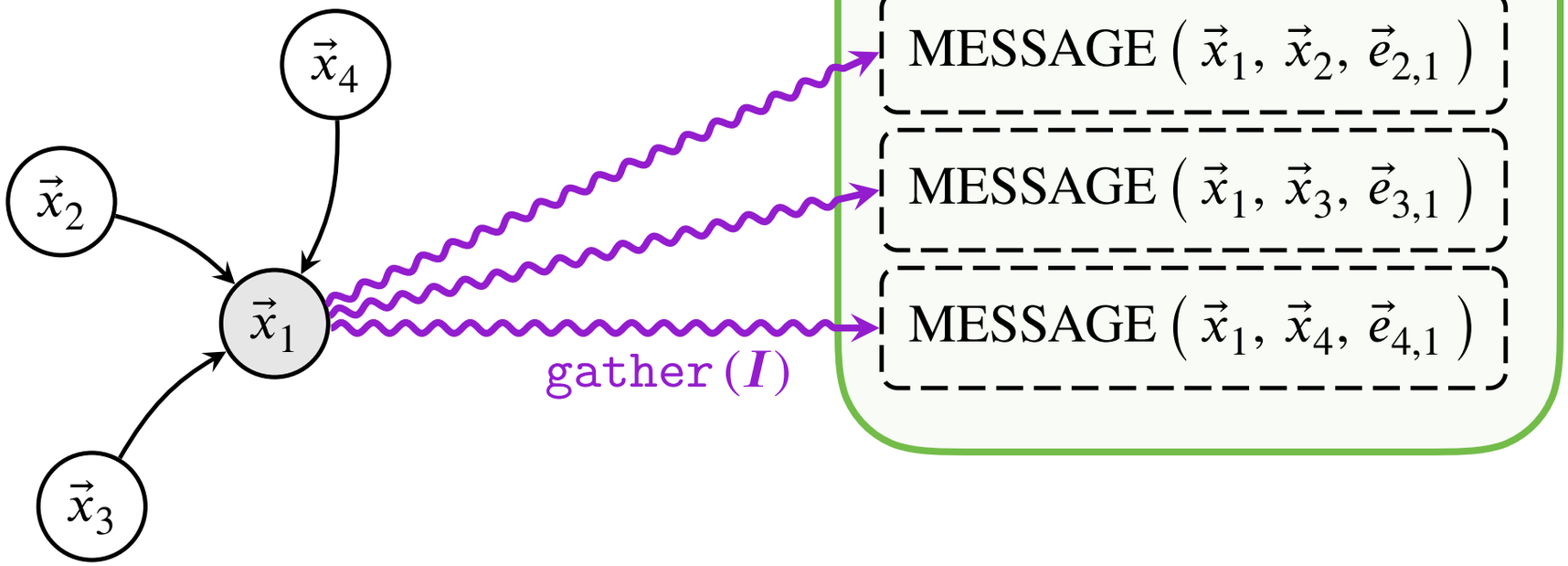
$$\mathbf{X} = \begin{bmatrix} \vec{x}_1 \\ \vec{x}_2 \\ \vec{x}_3 \\ \vec{x}_4 \end{bmatrix}$$



$$\mathbf{I} = \begin{bmatrix} 2 & 1 \\ 3 & 1 \\ 4 & 1 \end{bmatrix}^\top$$

Given a *sparse* graph $\mathcal{G} = (\mathbf{X}, (\mathbf{I}, \mathbf{E}))$

$$\mathbf{X} = \begin{bmatrix} \vec{x}_1 \\ \vec{x}_2 \\ \vec{x}_3 \\ \vec{x}_4 \end{bmatrix}$$

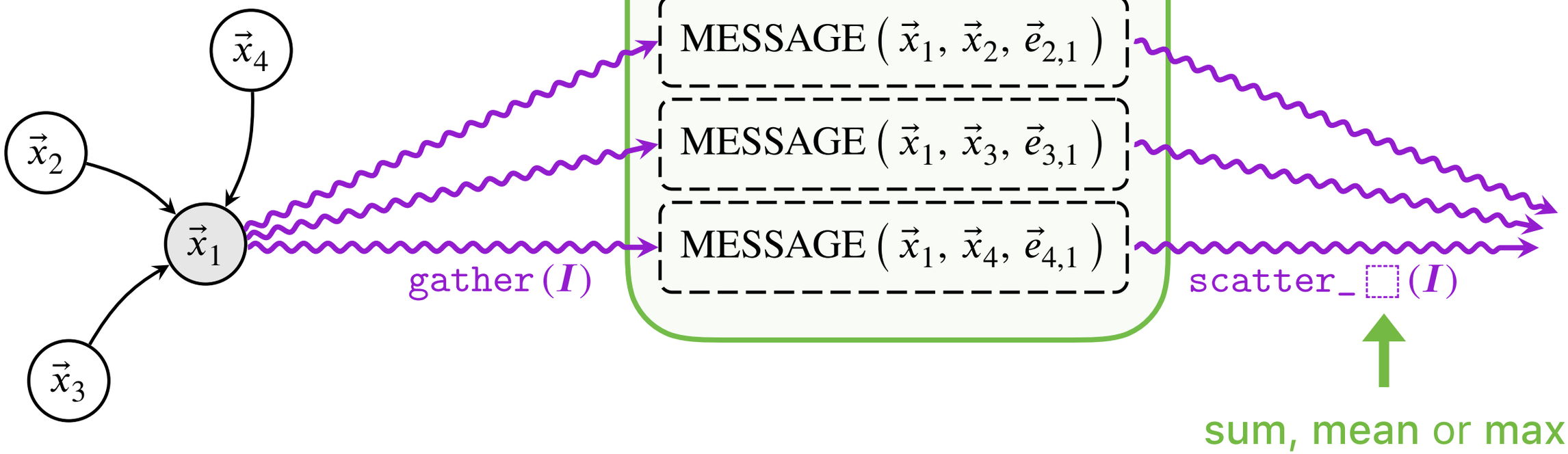


$$\mathbf{I} = \begin{bmatrix} 2 & 1 \\ 3 & 1 \\ 4 & 1 \end{bmatrix}^\top$$

Implementing GNNs

Given a *sparse* graph $\mathcal{G} = (\mathbf{X}, (\mathbf{I}, \mathbf{E}))$

$$\mathbf{X} = \begin{bmatrix} \vec{x}_1 \\ \vec{x}_2 \\ \vec{x}_3 \\ \vec{x}_4 \end{bmatrix}$$

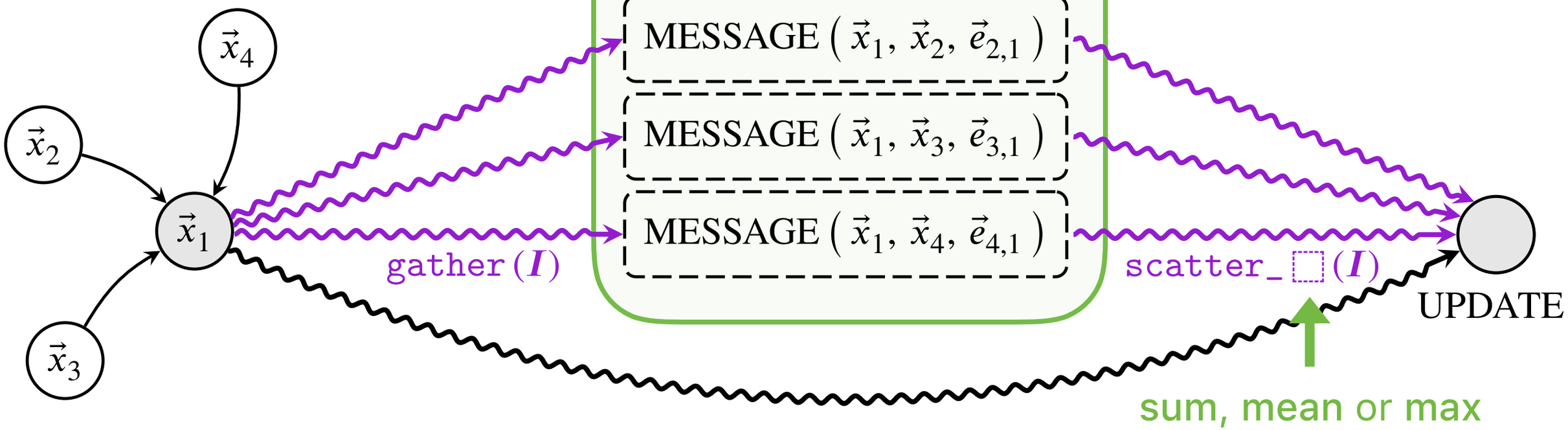


$$\mathbf{I} = \begin{bmatrix} 2 & 1 \\ 3 & 1 \\ 4 & 1 \end{bmatrix}^\top$$

Implementing GNNs

Given a *sparse* graph $\mathcal{G} = (\mathbf{X}, (\mathbf{I}, \mathbf{E}))$

$$\mathbf{X} = \begin{bmatrix} \vec{x}_1 \\ \vec{x}_2 \\ \vec{x}_3 \\ \vec{x}_4 \end{bmatrix}$$

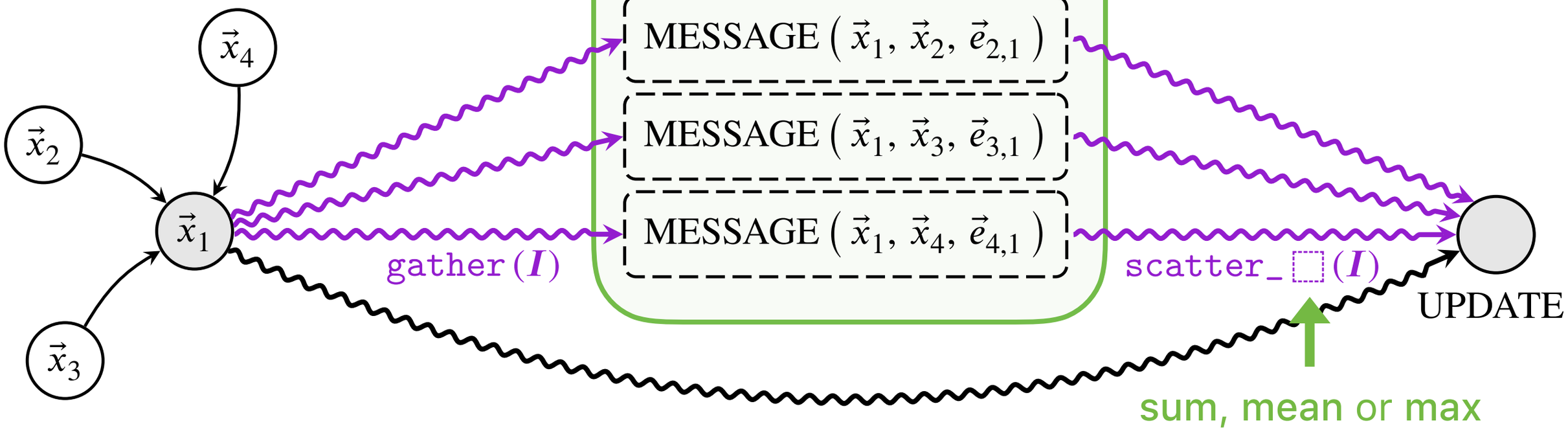


$$\mathbf{I} = \begin{bmatrix} 2 & 1 \\ 3 & 1 \\ 4 & 1 \end{bmatrix}^\top$$

Implementing GNNs

Given a *sparse* graph $\mathcal{G} = (\mathbf{X}, (\mathbf{I}, \mathbf{E}))$

$$\mathbf{X} = \begin{bmatrix} \vec{x}_1 \\ \vec{x}_2 \\ \vec{x}_3 \\ \vec{x}_4 \end{bmatrix}$$

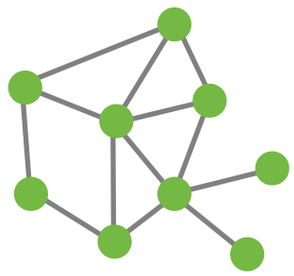


$$\mathbf{I} = \begin{bmatrix} 2 & 1 \\ 3 & 1 \\ 4 & 1 \end{bmatrix}^\top$$

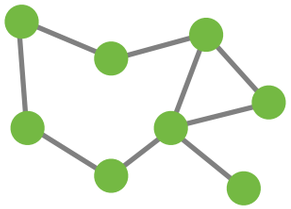
Flexible implementation via gather/scatter operations!

Mini-batch handling of different graphs:

$$\mathcal{G}_1 = (\mathbf{X}_1, \mathbf{A}_1)$$



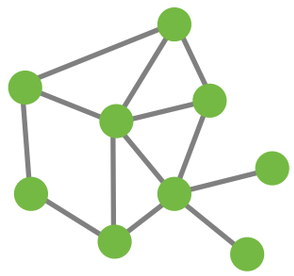
$$\mathcal{G}_2 = (\mathbf{X}_2, \mathbf{A}_2)$$



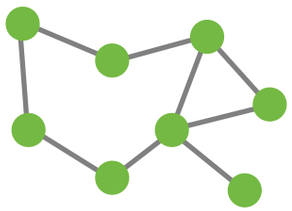
$$\text{GNN}\left(\begin{bmatrix} \mathbf{A}_1 & \\ & \mathbf{A}_2 \end{bmatrix}, \begin{bmatrix} \mathbf{X}_1 \\ \mathbf{X}_2 \end{bmatrix}\right) = \begin{bmatrix} \mathbf{X}'_1 \\ \mathbf{X}'_2 \end{bmatrix}$$

Mini-batch handling of different graphs:

$\mathcal{G}_1 = (\mathbf{X}_1, \mathbf{A}_1)$



$\mathcal{G}_2 = (\mathbf{X}_2, \mathbf{A}_2)$



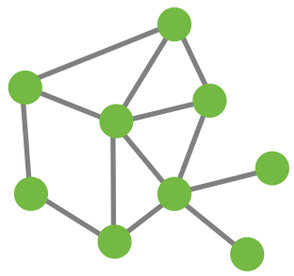
$$\text{GNN}\left(\begin{bmatrix} \mathbf{A}_1 & \\ & \mathbf{A}_2 \end{bmatrix}, \begin{bmatrix} \mathbf{X}_1 \\ \mathbf{X}_2 \end{bmatrix}\right) = \begin{bmatrix} \mathbf{X}'_1 \\ \mathbf{X}'_2 \end{bmatrix}$$

- *no GNN modifications needed*
- *no memory/computation overhead*
- *supports examples of different size*

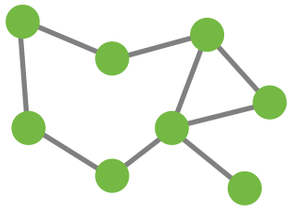
Implementing GNNs

Mini-batch handling of different graphs:

$\mathcal{G}_1 = (\mathbf{X}_1, \mathbf{A}_1)$



$\mathcal{G}_2 = (\mathbf{X}_2, \mathbf{A}_2)$

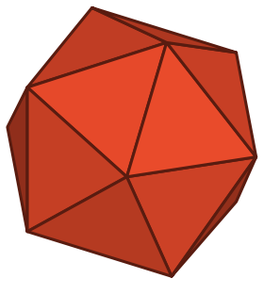


$$\mathrm{GNN}\left(\begin{bmatrix} \mathbf{A}_1 & \\ & \mathbf{A}_2 \end{bmatrix}, \begin{bmatrix} \mathbf{X}_1 \\ \mathbf{X}_2 \end{bmatrix}\right) = \begin{bmatrix} \mathbf{X}'_1 \\ \mathbf{X}'_2 \end{bmatrix}$$

assignment vector $\begin{bmatrix} 0,\ldots,0 & 1,\ldots,1 \end{bmatrix}$

readout function

- *no GNN modifications needed*
- *no memory/computation overhead*
- *supports examples of different size*



PyTorch geometric

A **Python library** built upon **PyTorch** to enable **deep learning on graphs:**

- simplifies implementing and working with **Graph Neural Networks**
- **omits the need of re-writing** common logic (operators, datasets, batching, ...)
- bundles **fast implementations** from published papers

PyTorch Geometric

GNNs

Cheby GCN SAGE PointNet MoNet MPNN GAT
SplineCNN AGNN EdgeCNN S-GCN R-GCN PointCNN
ARMA APPNP GIN GIN-E CG GatedGCN NMF TAG
Signed-GCN DNA PPFNet FeaST Hyper-GCN GravNet

Pooling

Set2Set SortPool DiffPool MinCUT Graclus
VoxelGrid TopK SAG EdgePool ASAP

Models

(V)GAE ARG(V)A DGI Node2Vec GraphUNet
GeniePath SchNet DimeNet MetaPath2Vec ReNet

Utilities

GNNExplainer SIGN GDC ClusterGCN DropEdge
GraphSAINT NeighborSampling GraphSizeNorm JK

1. Data Loading Routine

```
from torch_geometric.datasets import PPI
dataset = PPI('~/datasets/PPI', split='train')

print(dataset)
>>> PPI(20)
print(dataset[0])
>>> Data(edge_index=[2, 32318], x=[1767, 50], y=[1767, 121])
```

1. Data Loading Routine

```
from torch_geometric.datasets import PPI
dataset = PPI('~/datasets/PPI', split='train')

print(dataset)
>>> PPI(20)
print(dataset[0])
>>> Data(edge_index=[2, 32318], x=[1767, 50], y=[1767, 121])

from torch_geometric.data import DataLoader
loader = DataLoader(dataset, batch_size=2, shuffle=True)

print(next(iter(loader)))
>>> Batch(batch=[3422], edge_index=[2, 82434], x=[3422, 50], y=[3422, 121])
```

2. Defining the Network Architecture

```
from torch_geometric.nn import GCNConv

class Net(torch.nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = GCNConv(dataset.num_features, 256)
        self.conv2 = GCNConv(256, dataset.num_classes)

    def forward(self, x, edge_index):
        x = self.conv1(x, edge_index)
        x = x.relu()
        x = self.conv2(x, edge_index)
        return x
```

3. Training the Model

```
model = Net().cuda()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

def train():
    model.train()

    for data in loader:
        data = data.cuda()
        logits = model(data.x, data.edge_index)
        loss = compute_loss(logits, data.y)
        loss.backward()
        optimizer.step()
```

Training on giant graphs:

Training on giant graphs:

```
def train(data):  # Full-batch training
    logits = model(data.x, data.edge_index)
    loss = compute_loss(logits, data.y)
```

```
RuntimeError: CUDA error: out of memory
```

Training on giant graphs:

```
def train(data):  # Full-batch training
    logits = model(data.x, data.edge_index)
    loss = compute_loss(logits, data.y)
```

```
RuntimeError: CUDA error: out of memory
```

```
+ loader = ClusterLoader(  # Cluster-GCN
+     ClusterData(data, num_parts=128), batch_size=32)
```

```
~ def train(loader):  # Mini-batch training
+     for data in loader:
          out = model(data.x, data.edge_index)
          loss = compute_loss(logits, data.y)
```

Message Passing Interface

```
class MyOwnConv(MessagePassing):
    def __init__(self):
        super(MyOwnConv, self).__init__(aggr='add')
```

add, mean or max aggregation

Message Passing Interface

```
class MyOwnConv(MessagePassing):
    def __init__(self):
        super(MyOwnConv, self).__init__(aggr='add')

    def forward(self, x, edge_index, edge_weight):
        return self.propagate(edge_index, x=x,
                              edge_weight=edge_weight)
```

add, mean or max aggregation

pass every-thing needed for propagation

Message Passing Interface

```python
class MyOwnConv(MessagePassing):
    def __init__(self):
        super(MyOwnConv, self).__init__(aggr='add')

    def forward(self, x, edge_index, edge_weight):
        return self.propagate(edge_index, x=x,
                              edge_weight=edge_weight)

    def message(self, x_j, x_i, edge_weight):
        return x_j * edge_weight
```

add, mean or max aggregation

pass every-thing needed for propagation

Node features get **automatically lifted** to source (`_j`) and target (`_i`) nodes

PyTorch Geometric

- uniform implementations of over **40 GNN operators/models**
- extendable by using a simple **message passing interface**
- access to over **100 benchmark datasets**
- **automatic mini-batching for graphs with different sizes**
- deterministic and differentiable **pooling operators**
- basic and more sophisticated **readout functions**
- useful transforms for **augmentation**, **point sampling**, ...
- leverages **dedicated CUDA kernels**

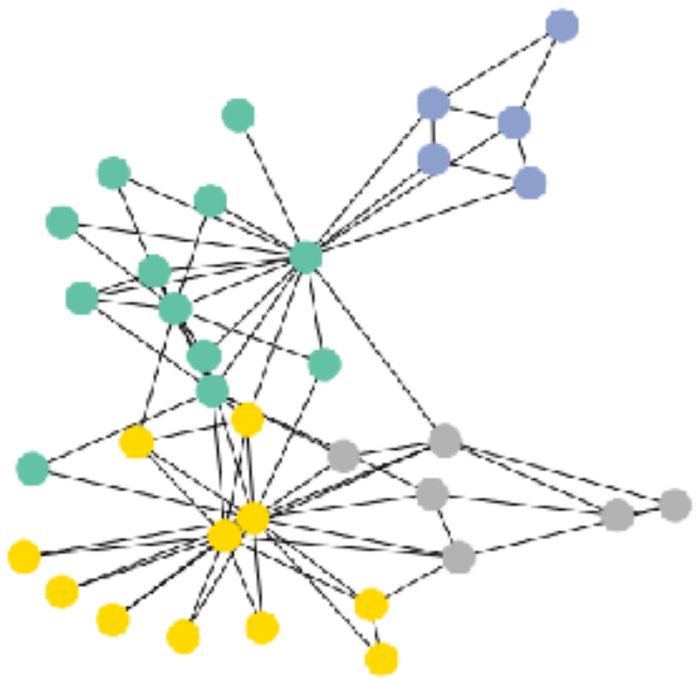
/rusty1s/pytorch_geometric

Embedding the Karate Club Network!

PyTorch geometric colab

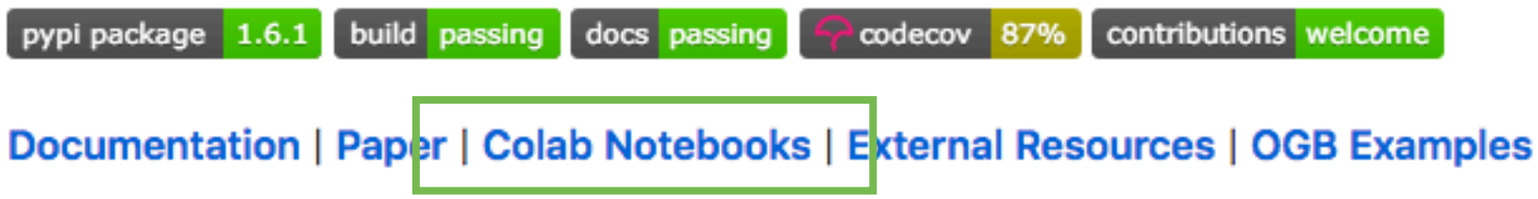
pypi package 1.6.1 | build passing | docs passing | codecov 87% | contributions welcome

Documentation | Paper | Colab Notebooks | External Resources | OGB Examples

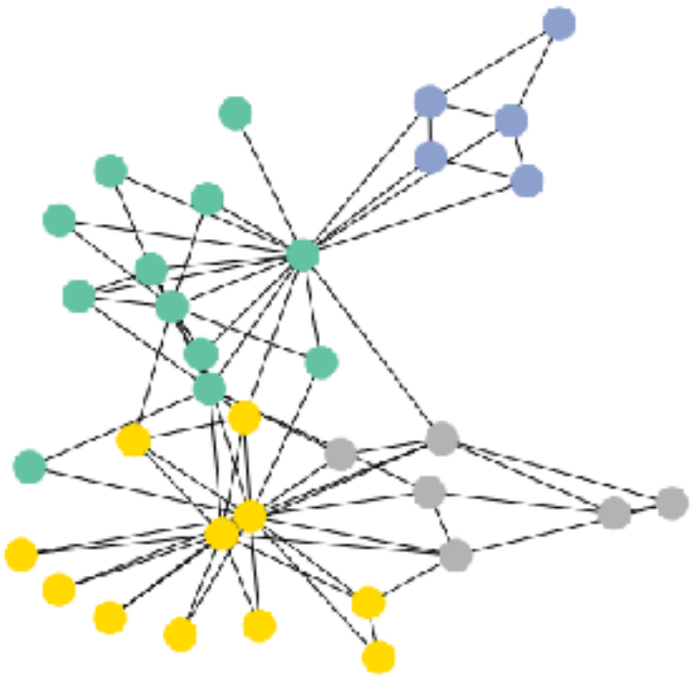


Embedding the Karate Club Network!

PyTorch geometric colab



Documentation | Paper | Colab Notebooks | External Resources | OGB Examples



Exercise:

Run through and complete
the **Point Cloud Classification Colab**

Conclusion

Graph Neural Networks are one of the hottest research fields in deep learning!

- GNNs learn **localized node embeddings** by following a simple **message passing scheme**
- Exciting research opportunities
  Expressivity, Pooling, Scalability, ...
- GNNs can help us to reason about **relational structure** in our world
- Exciting applications in
  computer vision, NLP, biology, chemistry, physics, ...

dortmund university

SFB 876 Providing Information by Resource-Constrained Data Analysis

Tutorial on Graph Neural Networks

Workshop on Geometry and Machine Learning with Applications to Biomedical Engineering - University College London

Matthias Fey

matthias.fey@udo.edu

rusty1s.github.io

/rusty1s

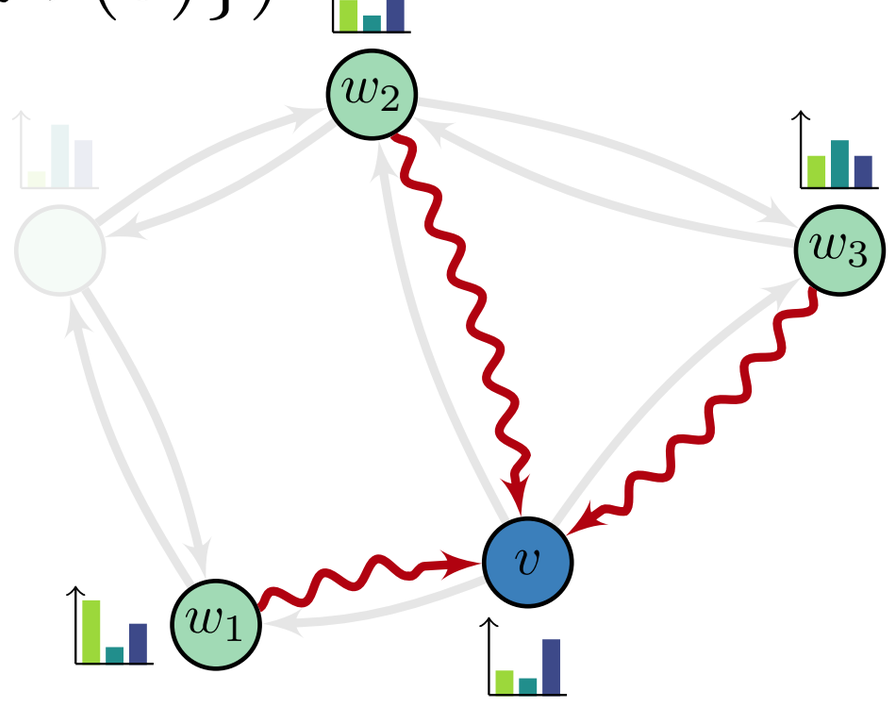
Conclusion

- GNNs learn **localized node embeddings** by following a simple **message passing scheme**

$$\mathbf{m}_{w,v}^{(\ell)} = \mathrm{MESSAGE}(\mathbf{h}_v^{(\ell-1)}, \mathbf{h}_w^{(\ell-1)})$$

$$\mathbf{a}_v^{(\ell)} = \mathrm{AGGREGATE}(\{\mathbf{m}_{w,v}^{(\ell)} : w \in \mathcal{N}(v)\})$$

$$\mathbf{h}_v^{(\ell)} = \mathrm{UPDATE}(\mathbf{h}_v^{(\ell-1)}, \mathbf{a}_v^{(l)})$$



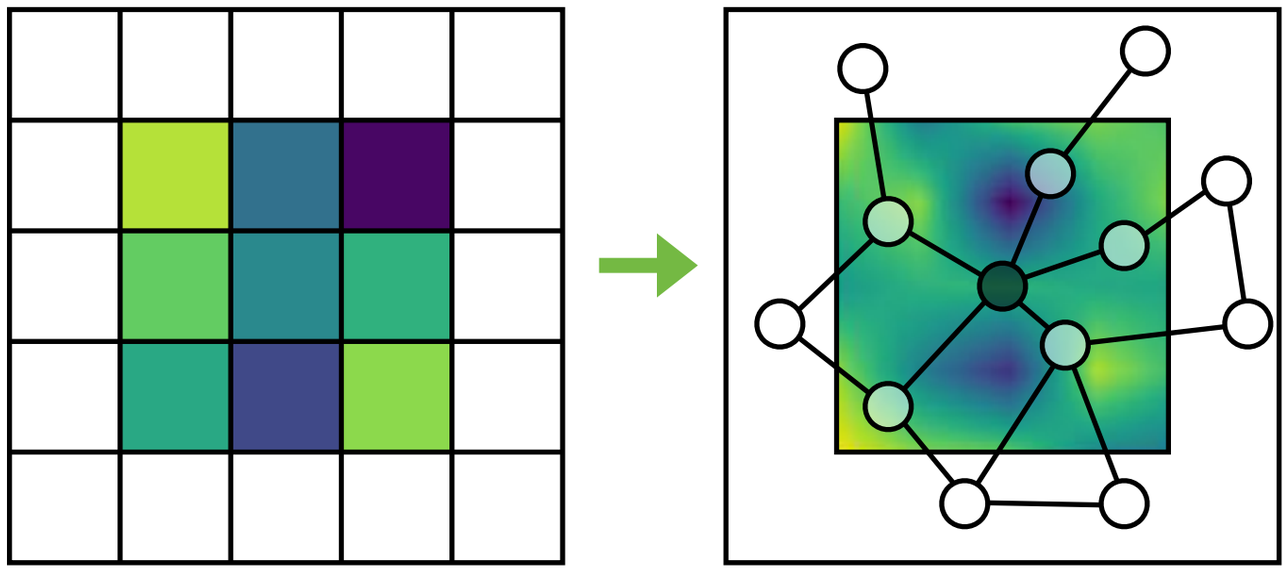
Conclusion

- GNNs learn **localized node embeddings** by following a simple **message passing scheme**

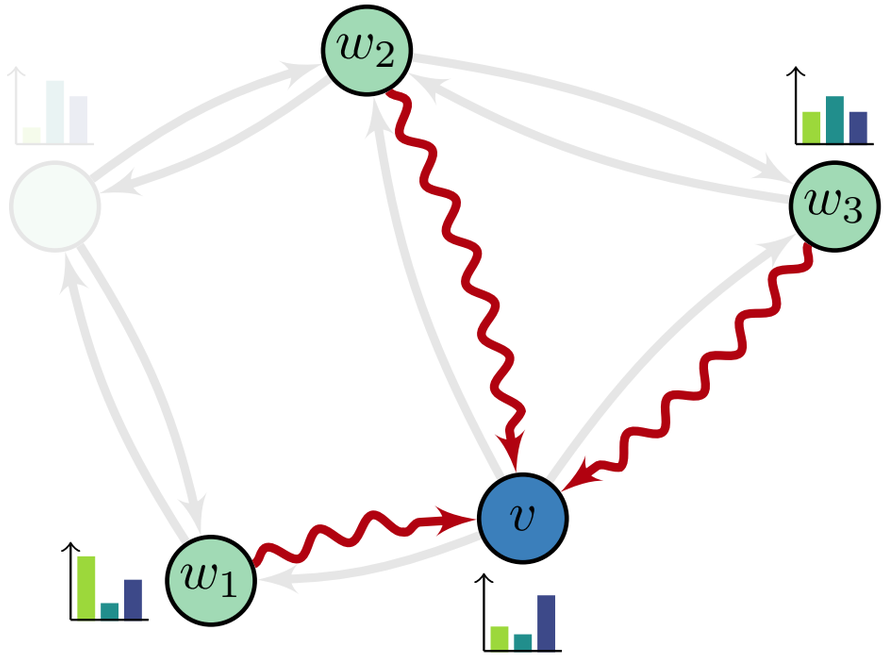
$$\mathbf{m}_{w,v}^{(\ell)} = \text{MESSAGE}(\mathbf{h}_v^{(\ell-1)}, \mathbf{h}_w^{(\ell-1)})$$

$$\mathbf{a}_v^{(\ell)} = \text{AGGREGATE}(\{\mathbf{m}_{w,v}^{(\ell)} : w \in \mathcal{N}(v)\})$$

$$\mathbf{h}_v^{(\ell)} = \text{UPDATE}(\mathbf{h}_v^{(\ell-1)}, \mathbf{a}_v^{(l)})$$



CNN GNN

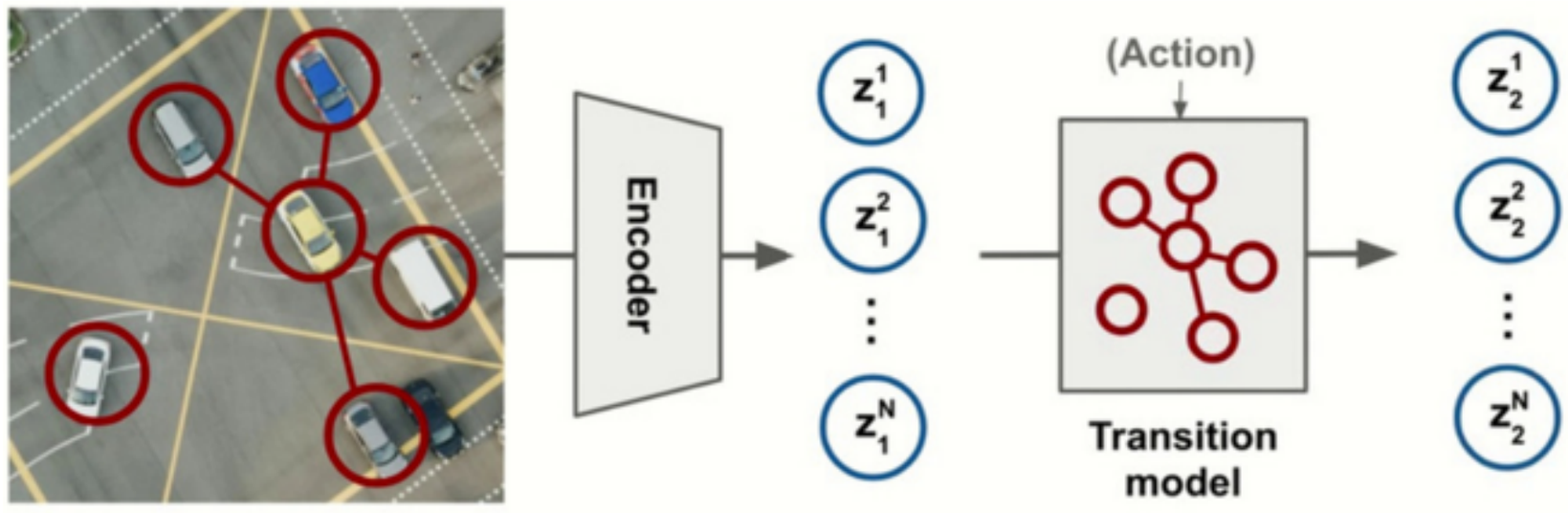


Conclusion

- GNNs can help us to reason about **relational structure** in our world

A small step towards more general AI?

- Combination of CNNs and GNNs for "object-centric" learning



Kipf et al.: Contrastive Learning of Structured World Models (ICLR '20)

Conclusion

- Exciting research opportunities
  Expressivity, Pooling, Scalability, ...

How powerful are Graph Neural Networks in distinguishing graph structures?

- Different aggregations yield different power
- **Higher-order variants**

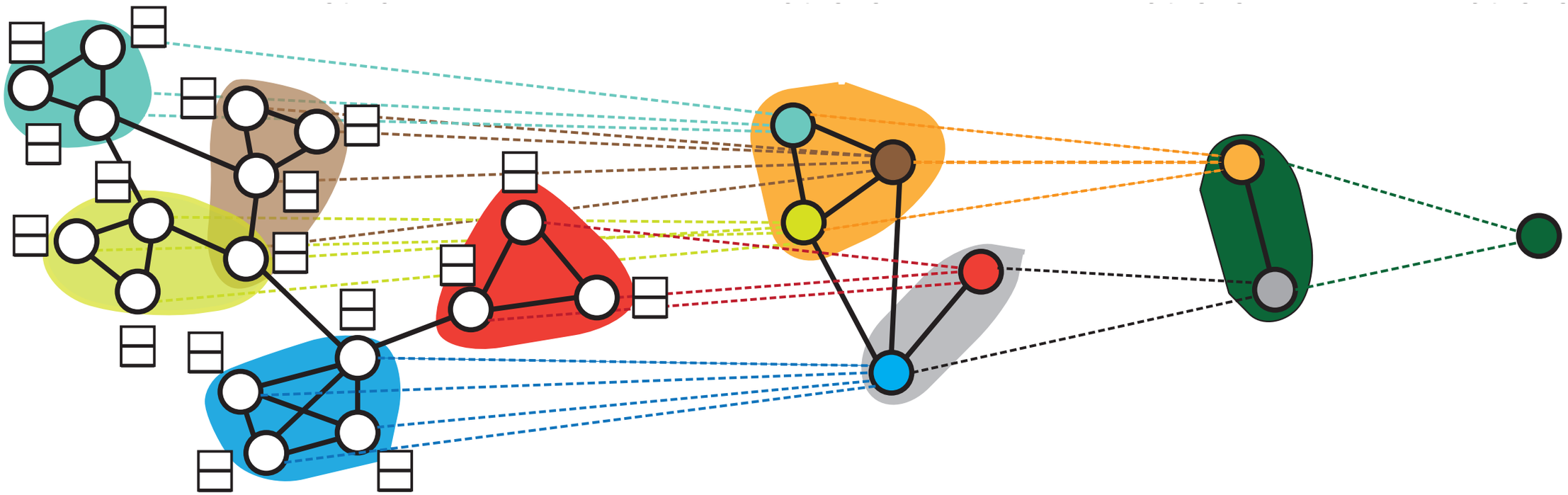
Maron et al.: **Provably Powerful Graph Neural Networks** (NeurIPS '19)
Morris et al.: **Weisfeiler and Leman Go Neural: Higher-order Graph Neural Networks** (AAAI '19)

Conclusion

- Exciting research opportunities
  Expressivity, **Pooling**, Scalability, ...

Can we reason about hierarchies in graphs?

Can we generalize the concepts of pooling known from classical CNNs?

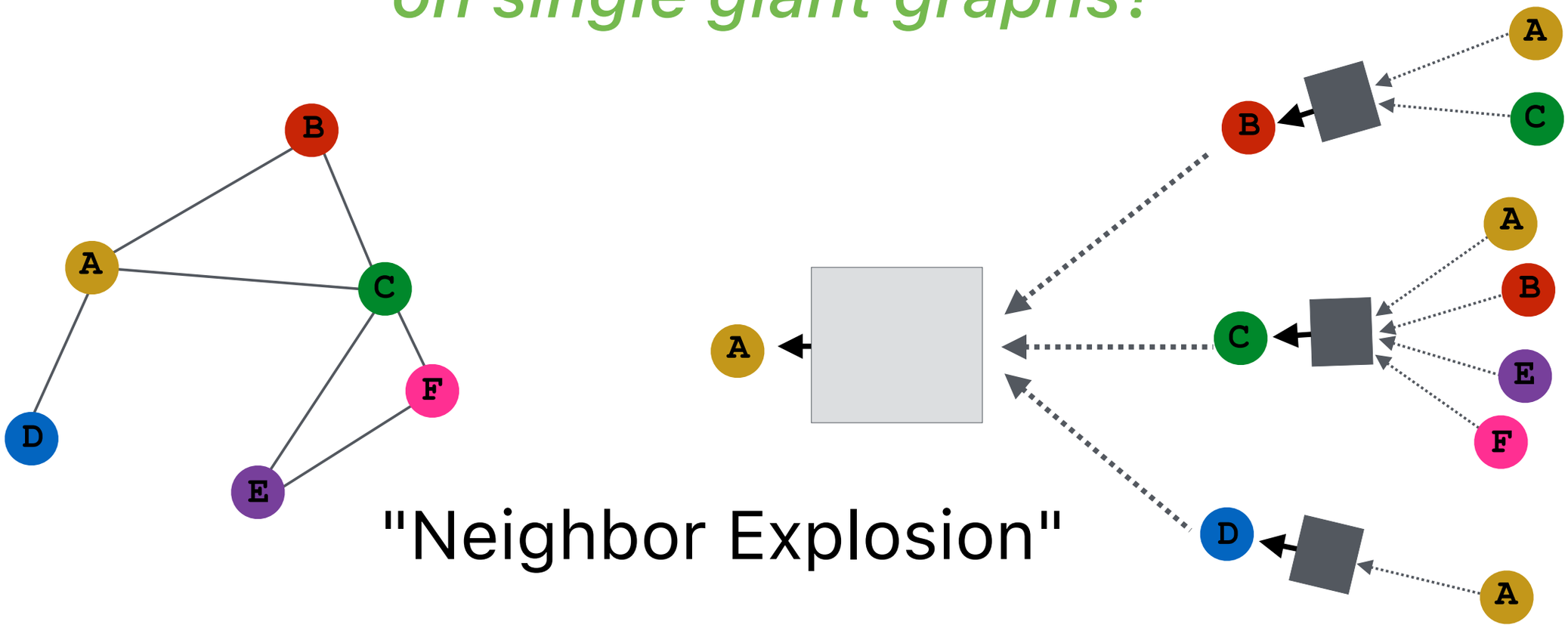


Ying et al.: **Hierarchical Graph Representation Learning with Differentiable Pooling** (NeurIPS '18)
Cangea et al.: **Towards Sparse Hierarchical Graph Classifiers** (NeurIPS-W '18)

Conclusion

- Exciting research opportunities
  Expressivity, Pooling, **Scalability**, ...

How can we train GNNs in a mini-batch fashion on single giant graphs?



Hamilton et al.: **Inductive Representation Learning on Large Graphs** (NIPS '17)
Chiang et al.: **Cluster-GCN: An Efficient Algorithm for Training Deep and Large GCNs** (KDD '19)
Zeng et al.: **GraphSAINT: Graph Sampling Based Inductive Learning Method** (ICLR '20)

Conclusion

- Exciting research opportunities
  Expressivity, Pooling, Scalability, ...

There are still a lot of other things to explore!

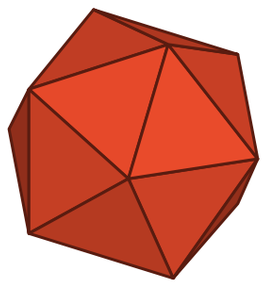
- **Temporal** GNNs
- **Pre-Training** of GNNs
- **Generative models** for graphs
- ...

Conclusion

- GNNs can be **implemented efficiently**

```
class GNN(torch.nn.Module):
    def __init__(self):
        super(GNN, self).__init__()
        self.conv1 = GCNConv(128, 256)

def forward(self, x, edge_index):
        x = self.conv1(x, edge_index)
        x = x.relu()
        ...
```



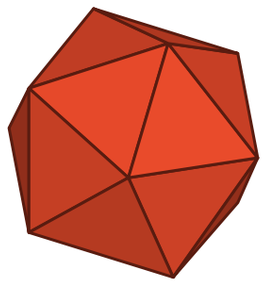
PyTorch geometric

colab

/rusty1s/pytorch_geometric

Conclusion

- GNNs can be **implemented efficiently**



PyTorch geometric

```
class GNN(torch.nn.Module):
    def __init__(self):
        ...

    def forward(self, x, edge_index):
        x = self.conv1(x, edge_index)
        x = x.relu()
        ...
```

Any Questions?

/rusty1s/pytorch_geometric