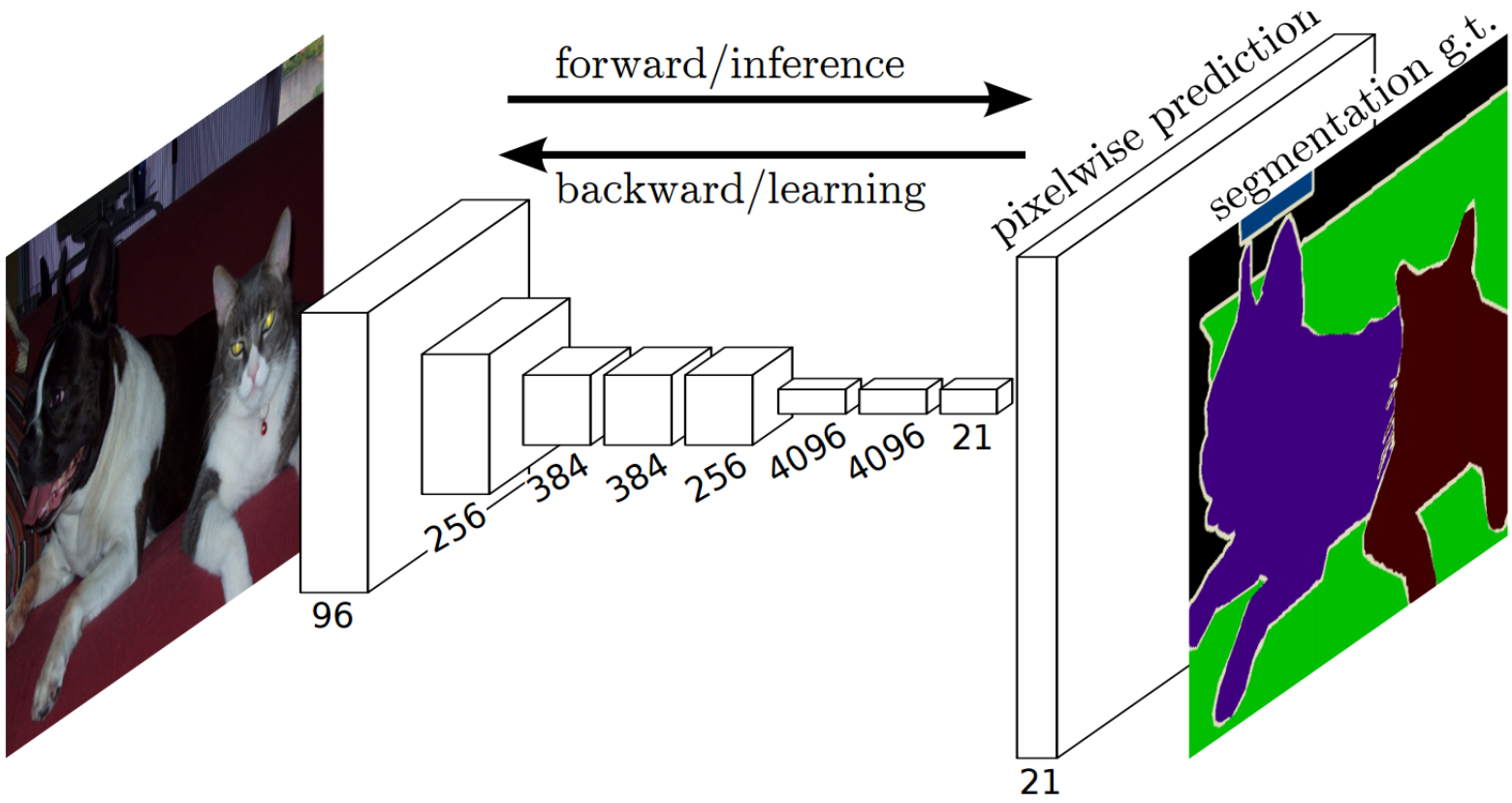
Graph Neural Networks

Soumith Chintala

Neural Networks



Types of typical operators

Convolution

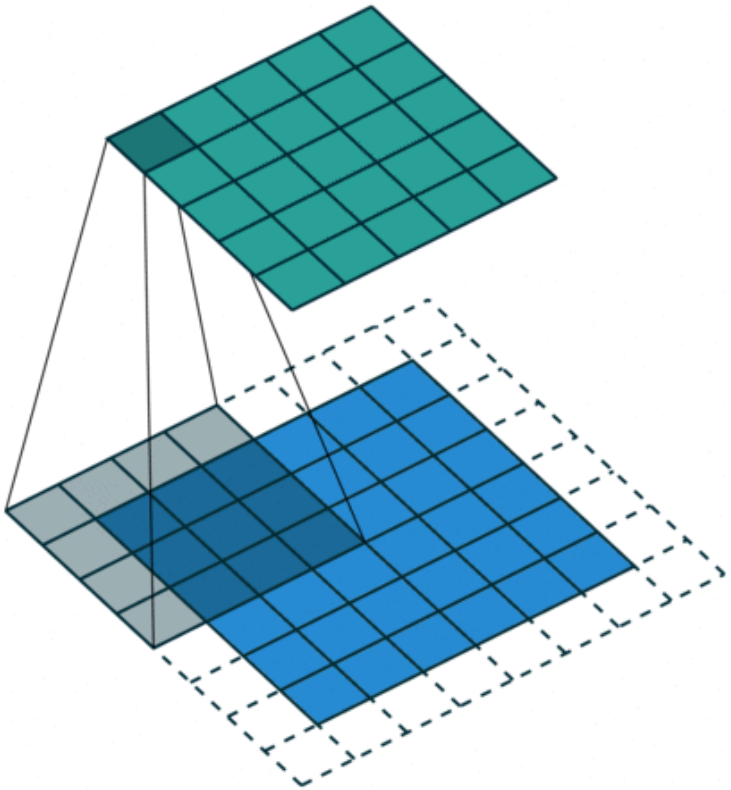
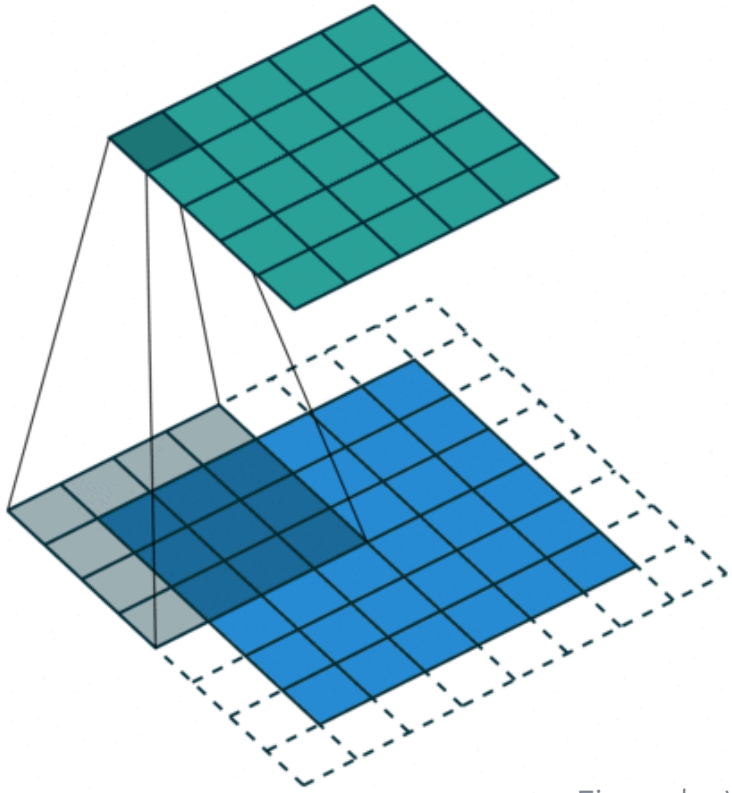


Figure by Vincent Dumolin: https://github.com/vdumoulin/conv_arithmetic

Types of typical operators

Convolution



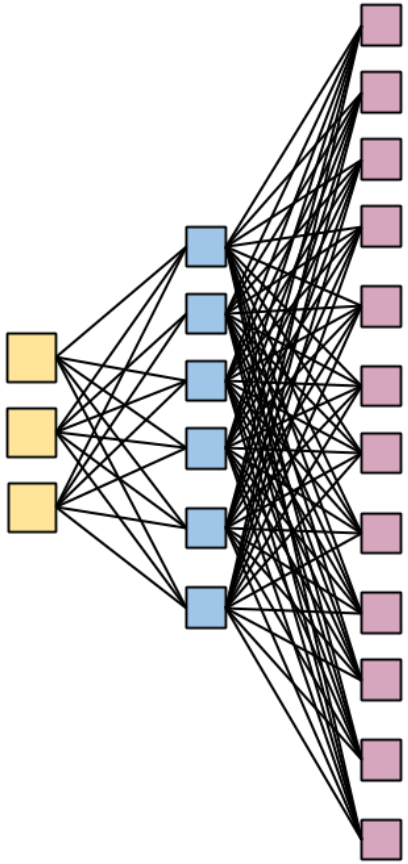


Figure by Vincent Dumolin: https://github.com/vdumoulin/conv_arithmetic

Types of typical operators

Convolution

```
for oc in output_channel:
    for ic in input_channel:
        for h in output_height:
            for w in output_width:
                for kh in kernel_height:
                    for kw in kernel_width:
                        output_pixel += input_pixel * kernel_value
```

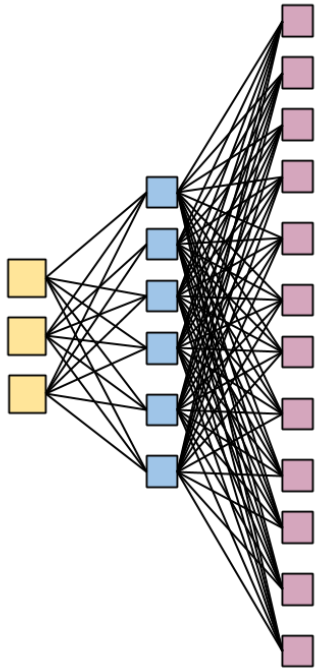
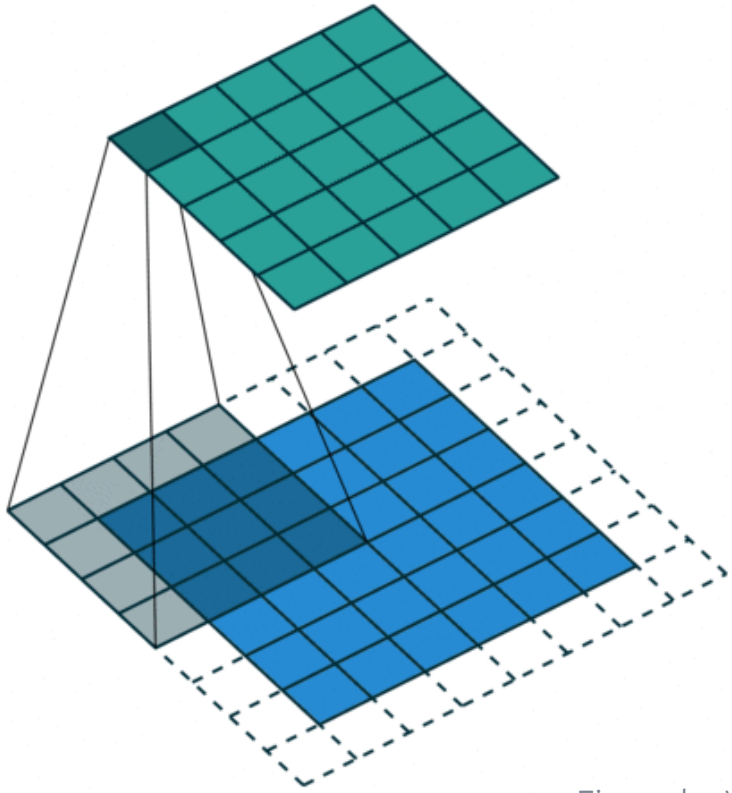


Figure by Vincent Dumolin: https://github.com/vdumoulin/conv_arithmetic

Types of typical operators

Pooling



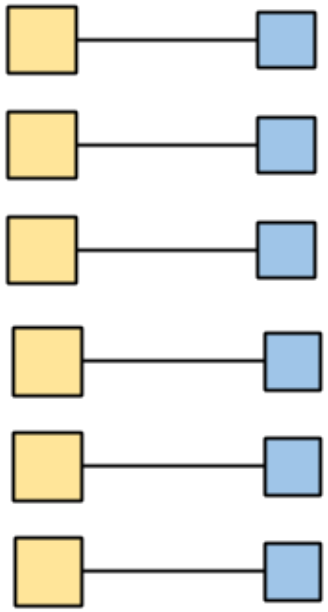
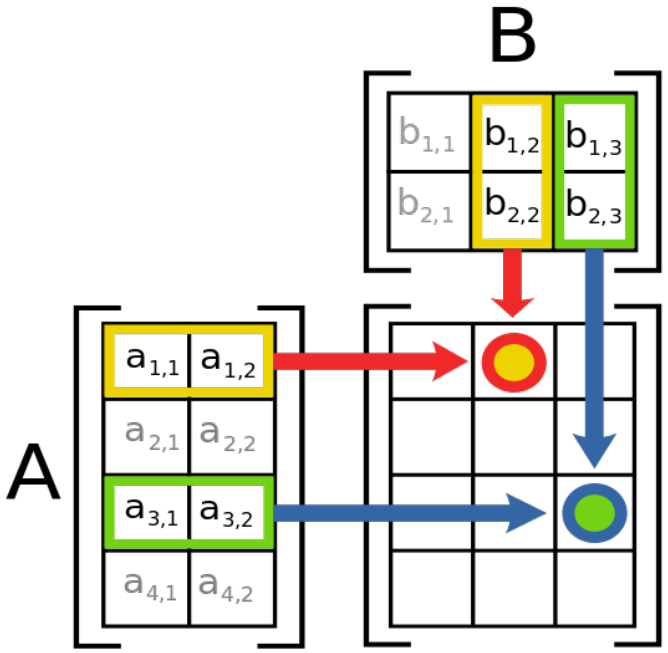


Figure by Vincent Dumolin: https://github.com/vdumoulin/conv_arithmetic

Types of typical operators

Matrix Multiply



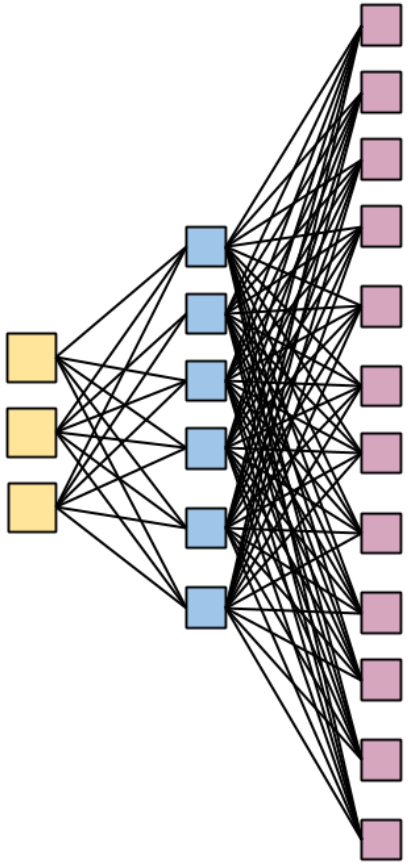
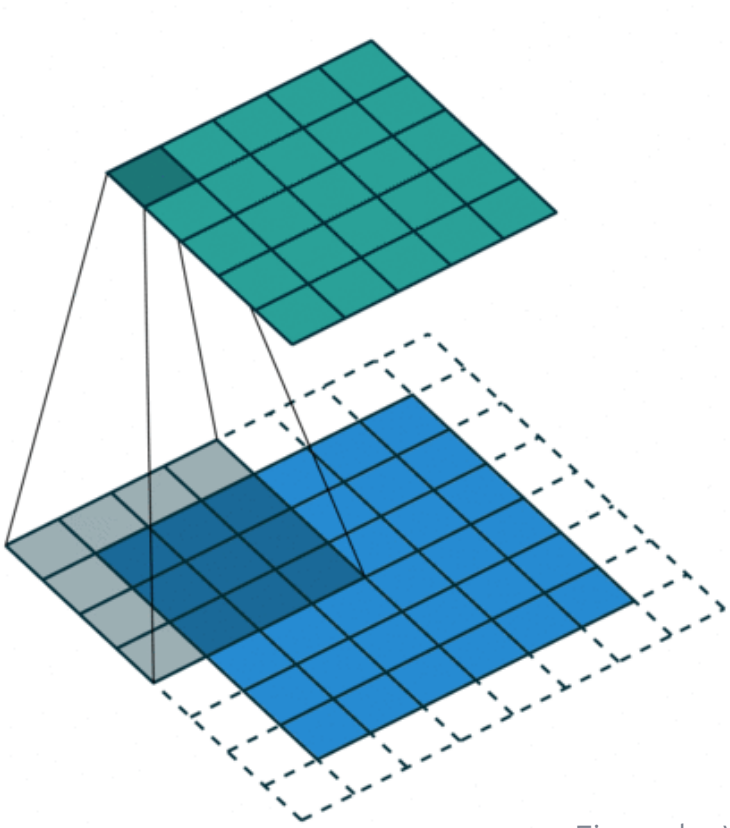


Figure by Wikipedia: https://en.wikipedia.org/wiki/Matrix_multiplication

Convolution



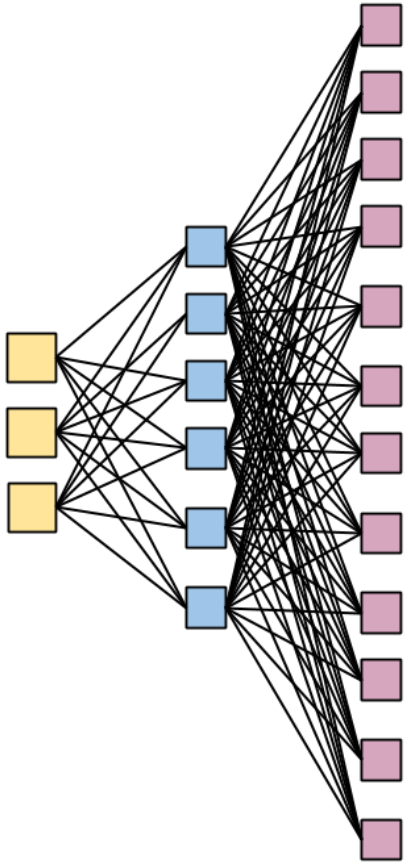
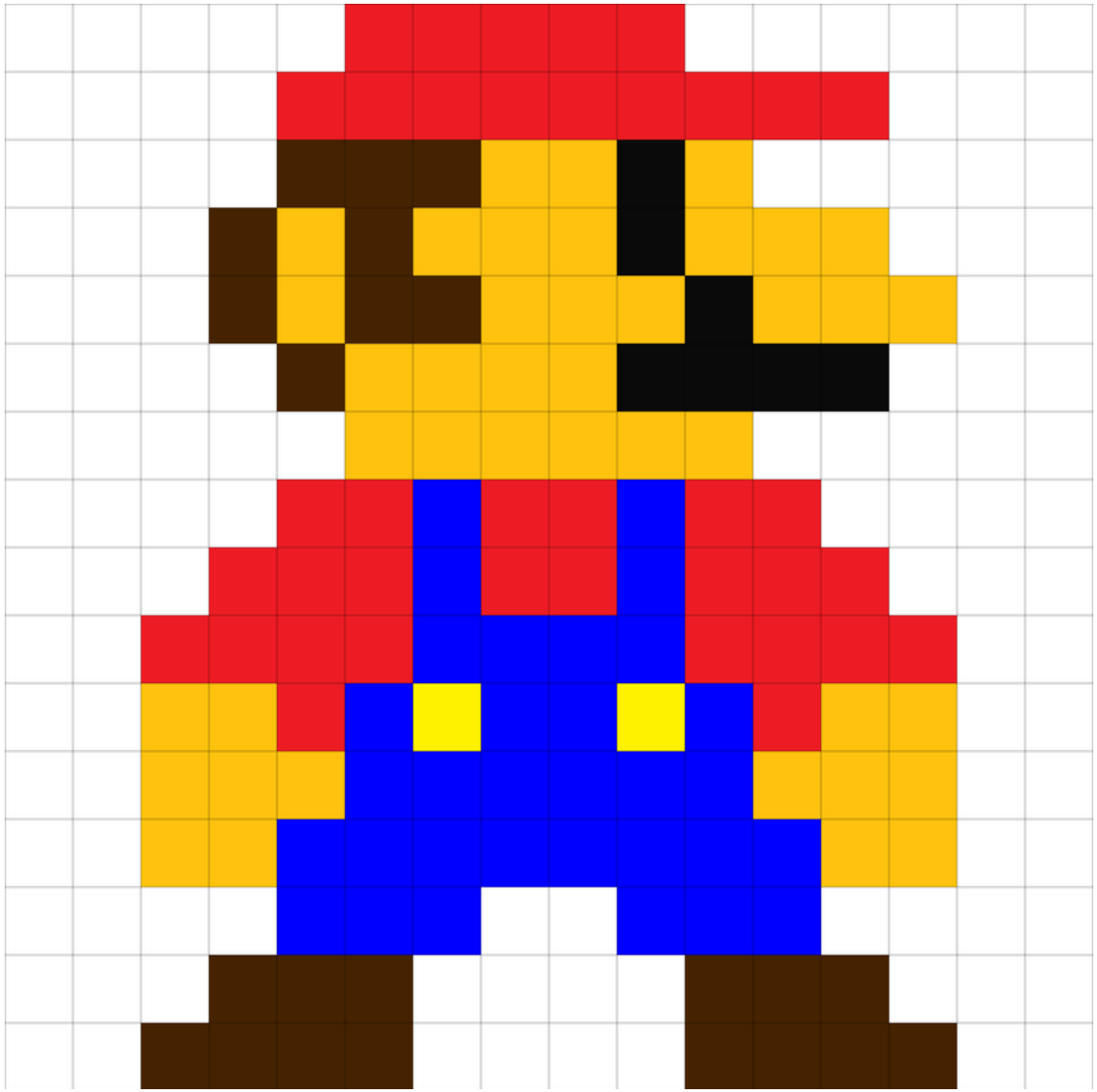
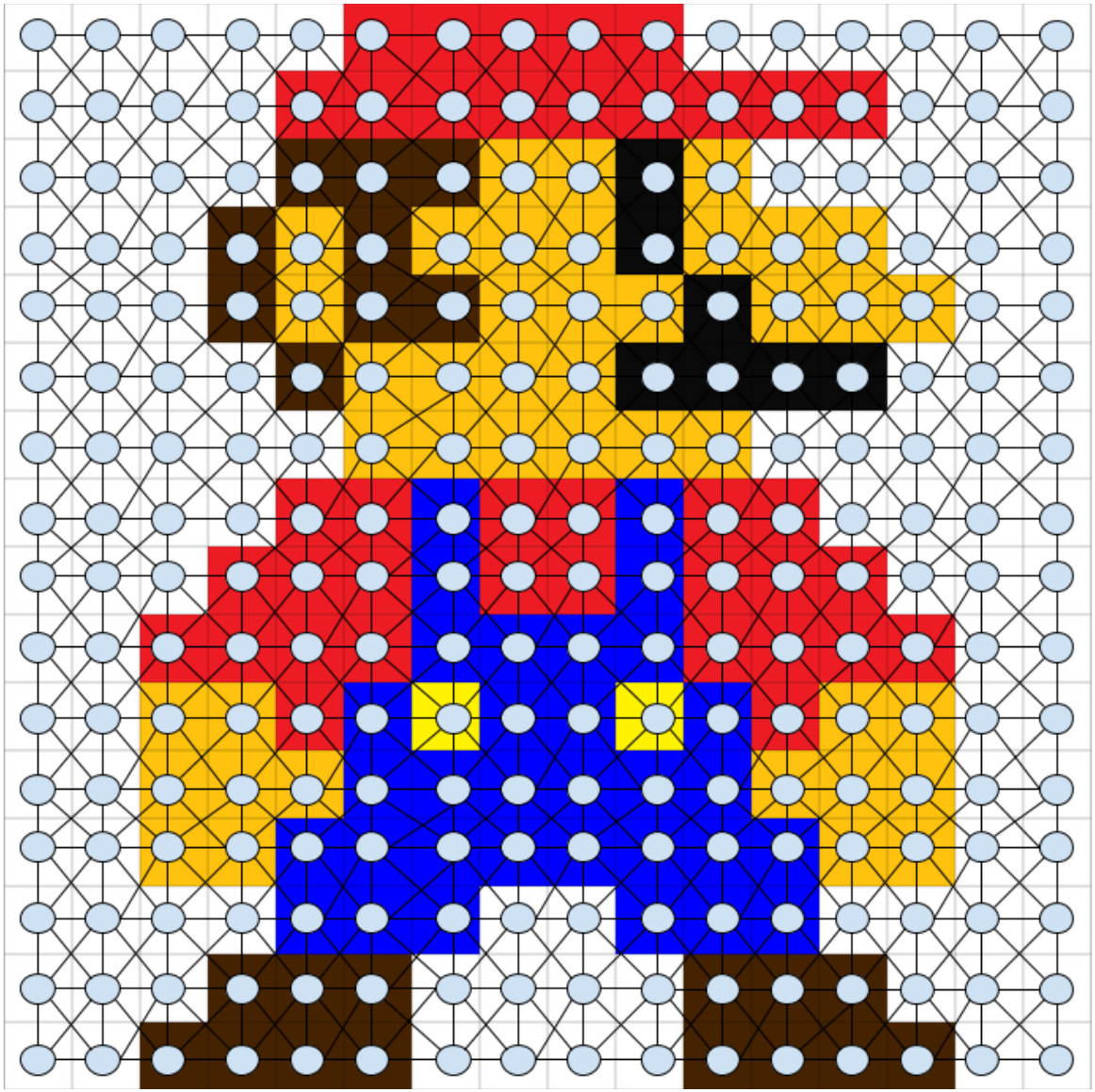
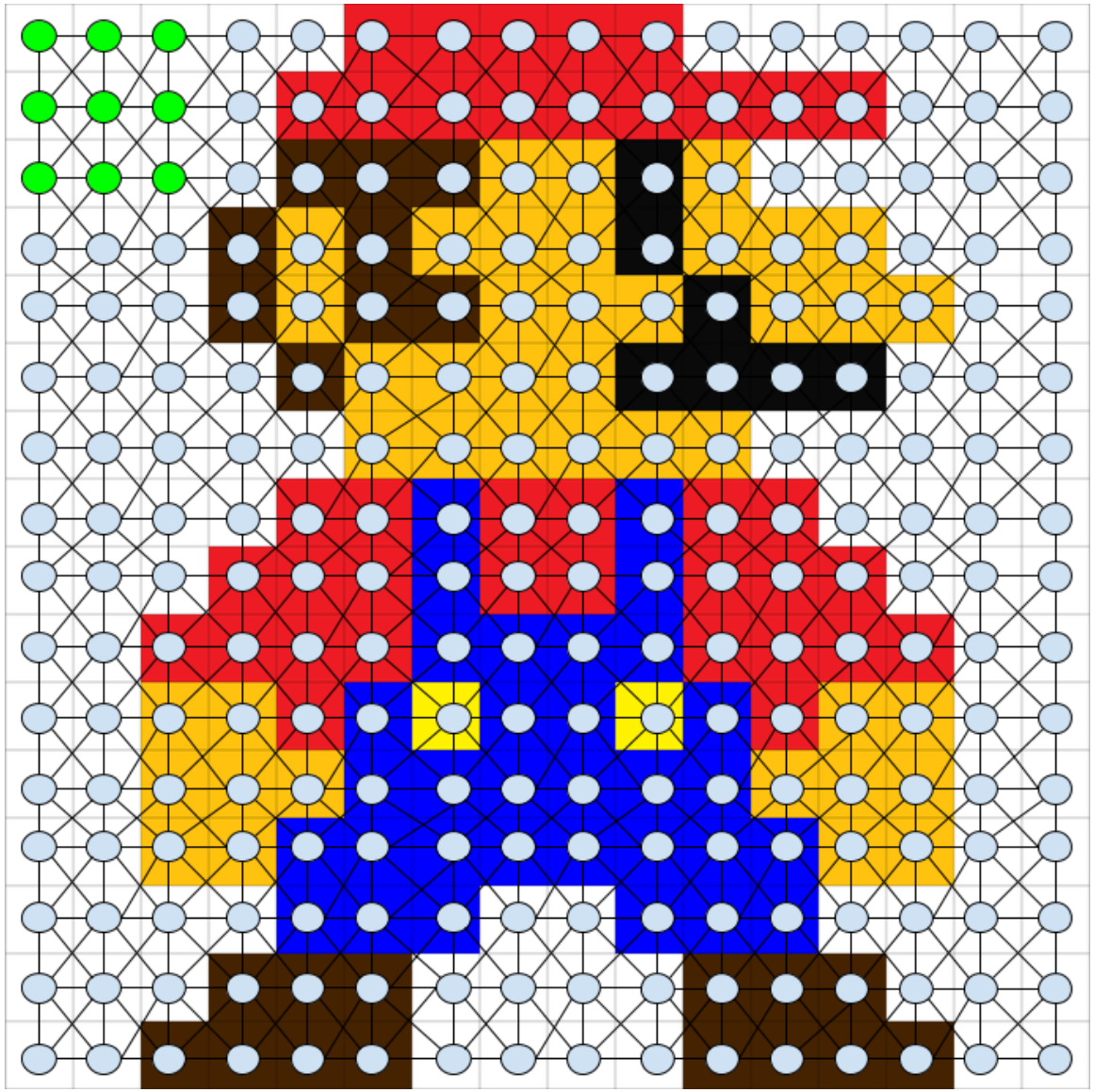
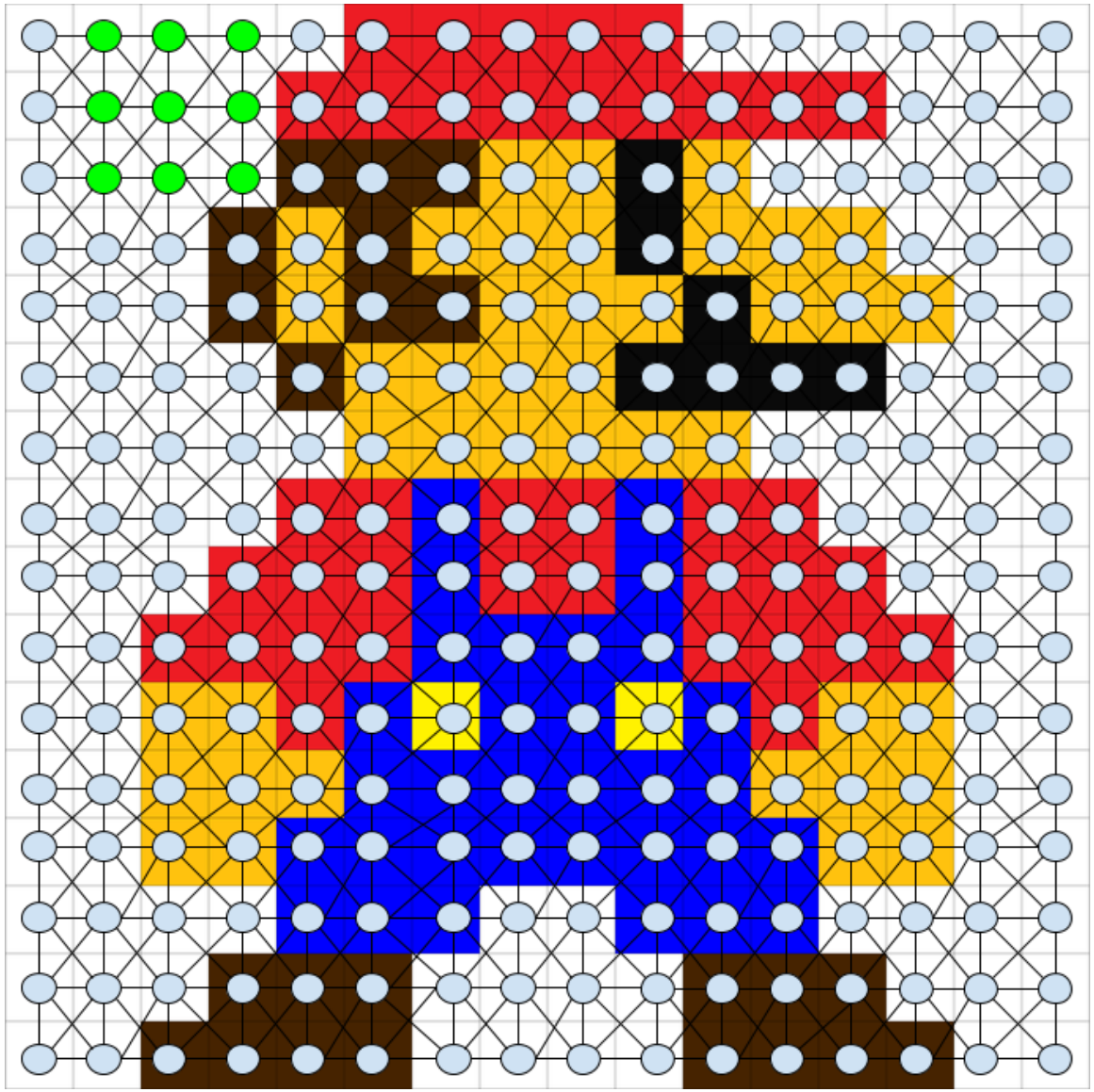


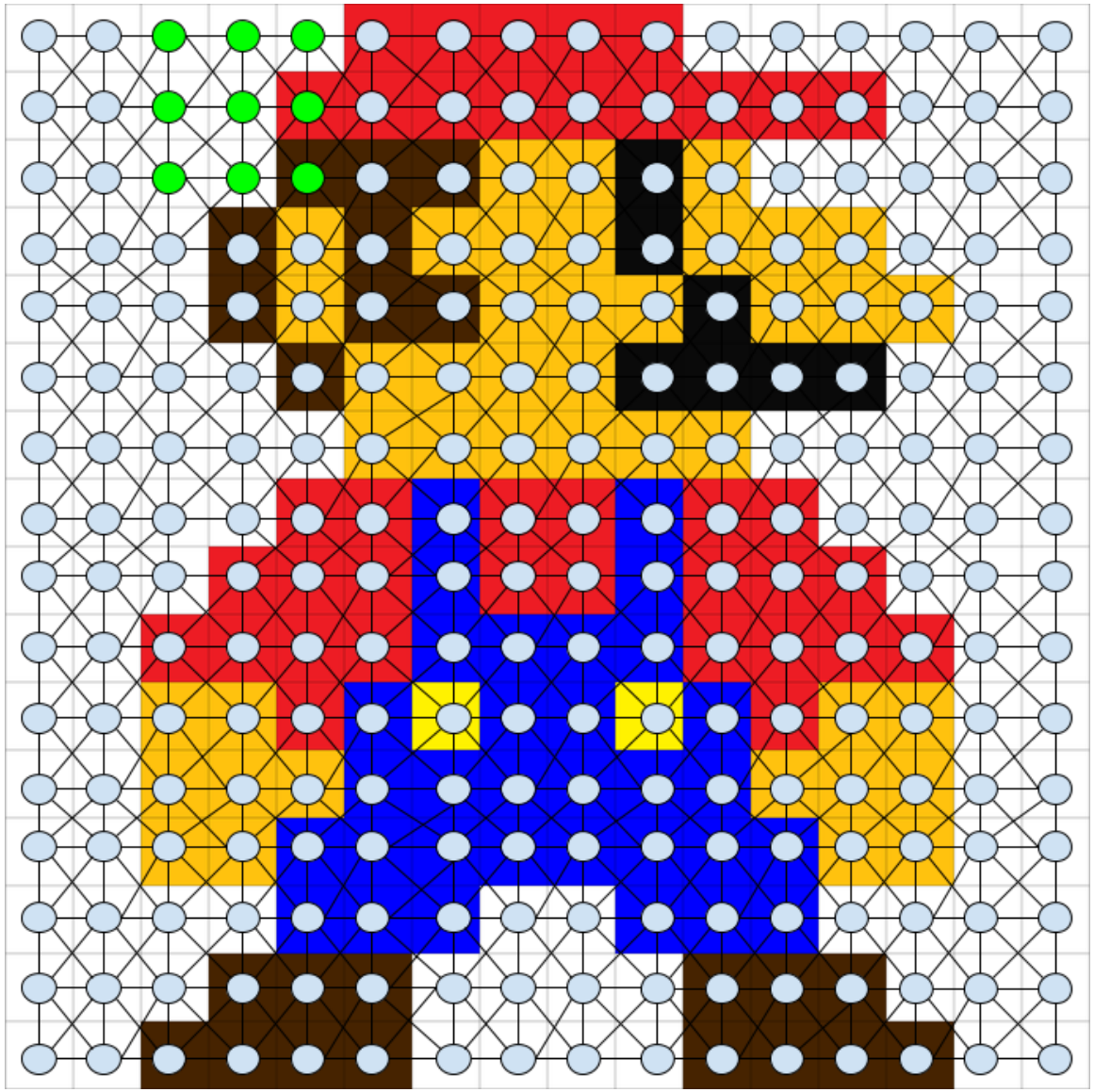
Figure by Vincent Dumolin: https://github.com/vdumoulin/conv_arithmetic

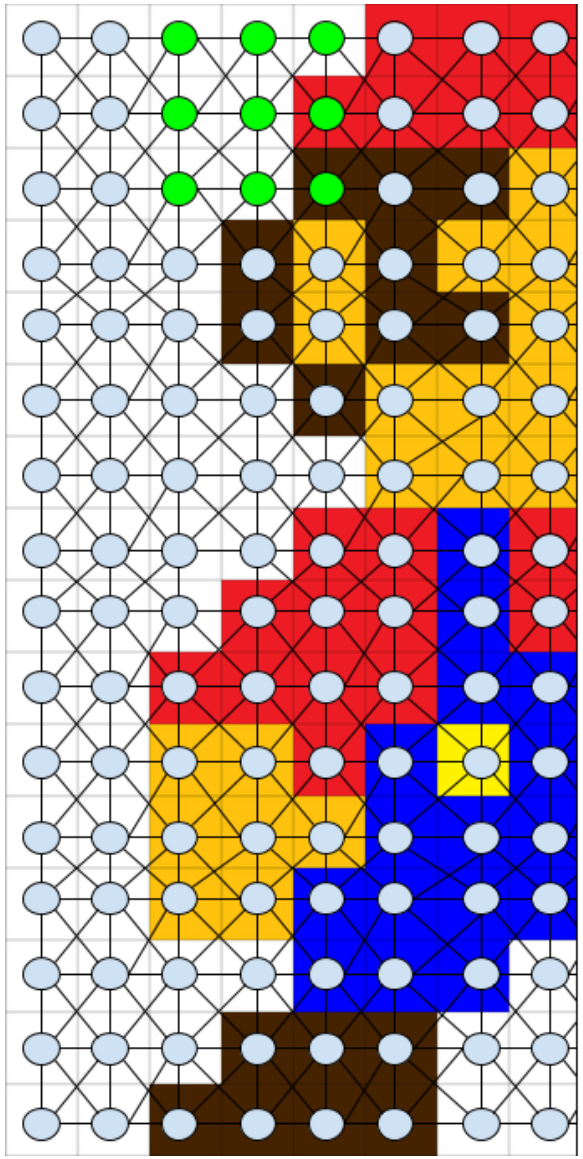


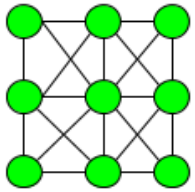








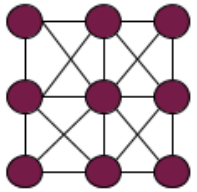




input

channels x height x width

(3 x 3 x 3)

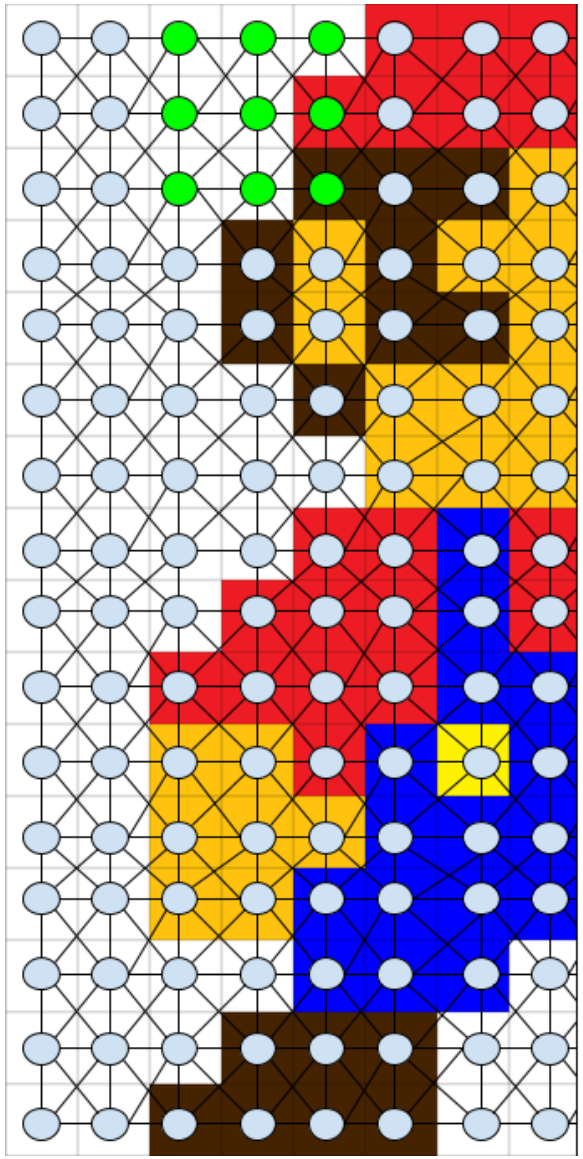


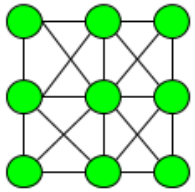
weight

channels x height x width

(3 x 3 x 3)

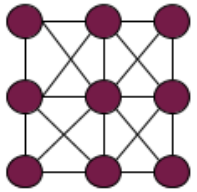
output[i] = sum(input[i-1:i+1, i-1:i+1]^T * weight)





input

channels x height x width
(3 x 3 x 3)



weight

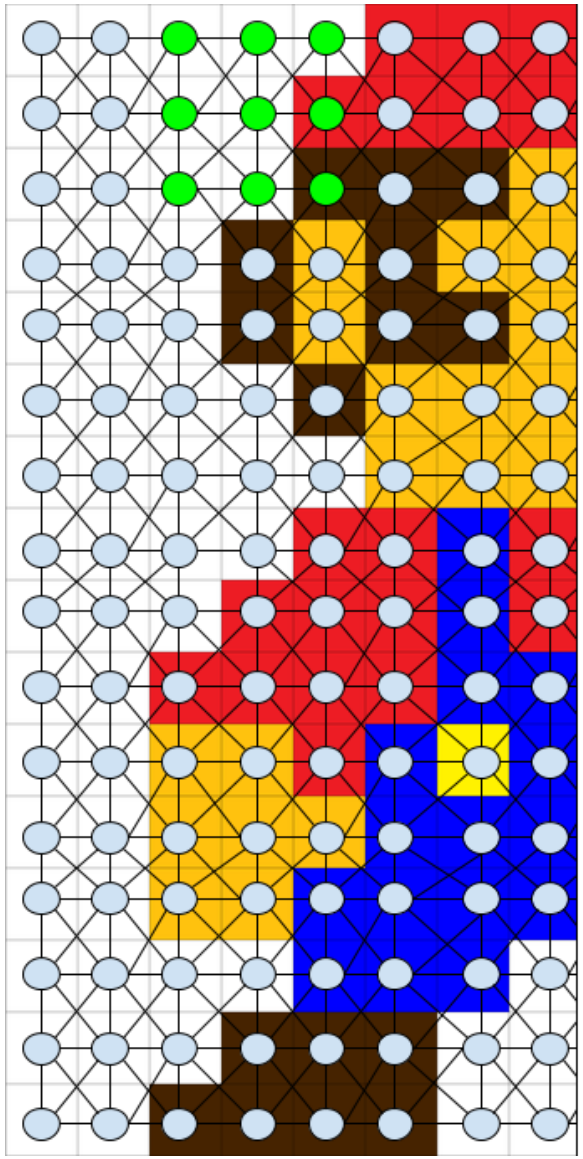
channels x height x width
(3 x 3 x 3)

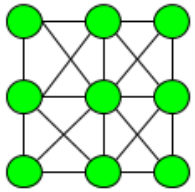
Step 1: Compute Node Features

f[0, 0] = inp[0, 0] * w[0, 0]
f[0, 1] = inp[0, 1] * w[0, 1]
f[0, 2] = inp[0, 2] * w[0, 2]

f[1, 0] = inp[1, 0] * w[1, 0]
f[1, 1] = inp[1, 1] * w[1, 1]
f[1, 2] = inp[1, 2] * w[1, 2]

f[2, 0] = inp[2, 0] * w[2, 0]
f[2, 1] = inp[2, 1] * w[2, 1]
f[2, 2] = inp[2, 2] * w[2, 2]

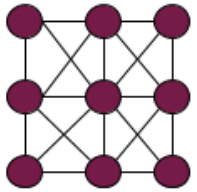




input

channels x height x width

(3 x 3 x 3)



weight

channels x height x width

(3 x 3 x 3)

Step 2: Message Passing

Pass features to connected edges

Mailbox[0, 0] = {f(0, 0), f(0, 1), f(1, 0), f(1, 1)}

Mailbox[0, 1] = {f(0, 0), f(0, 1), f(0, 2),
f(1, 0), f(1, 1), f(1, 2)}

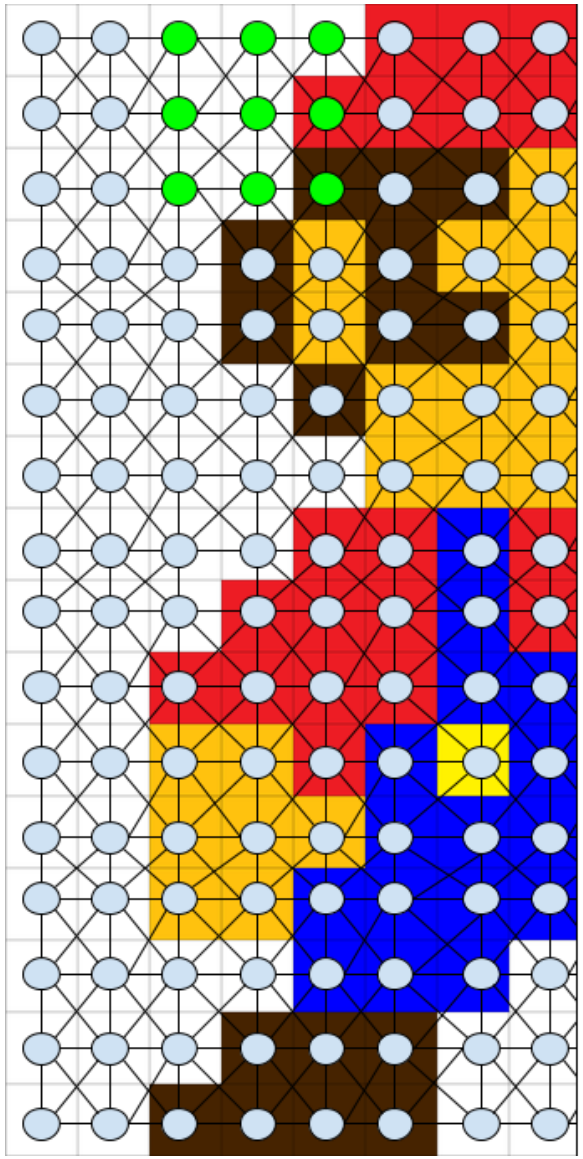
...

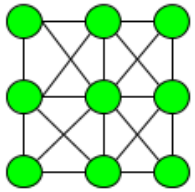
...

Mailbox[1, 1] = {f(0, 0), f(0, 1), f(0, 2),
f(1, 0), f(1, 1), f(1, 2),
f(2, 0), f(2, 1), f(2, 2)}

...

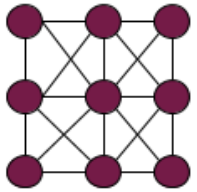
...





input

channels x height x width
(3 x 3 x 3)



weight

channels x height x width
(3 x 3 x 3)

Step 3: Aggregation

output[0, 0] = sum(mailbox)

...

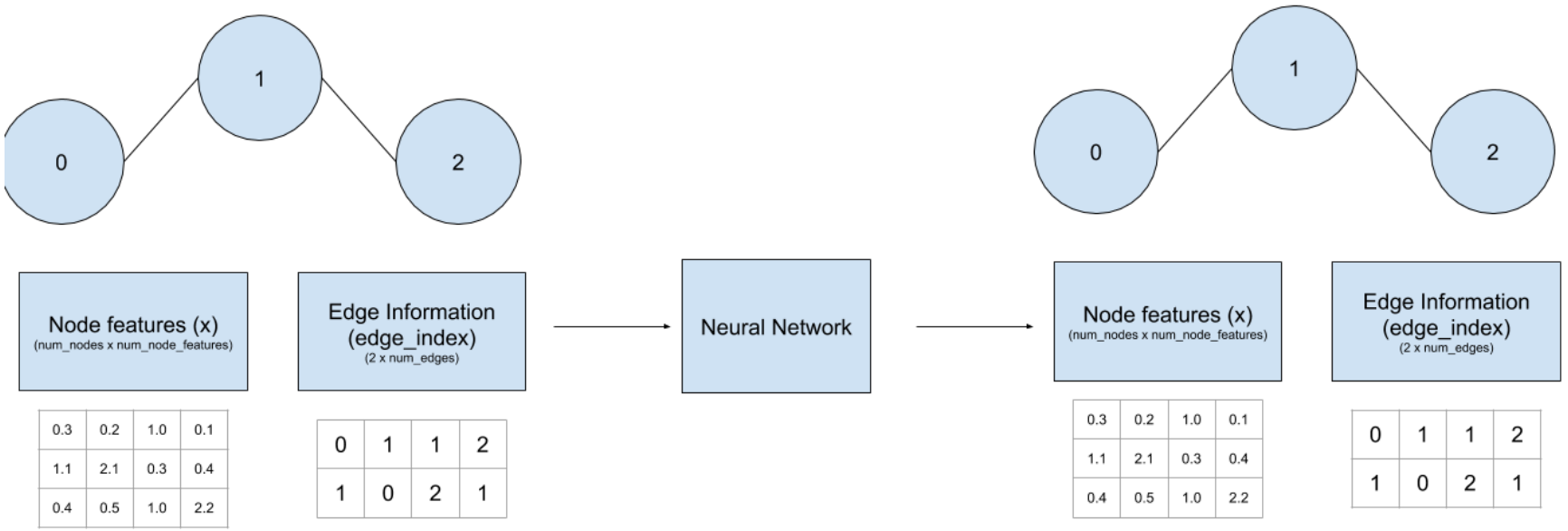
...

output[1, 1] = sum{f(0, 0), f(0, 1), f(0, 2),
f(1, 0), f(1, 1), f(1, 2),
f(2, 0), f(2, 1), f(2, 2)}

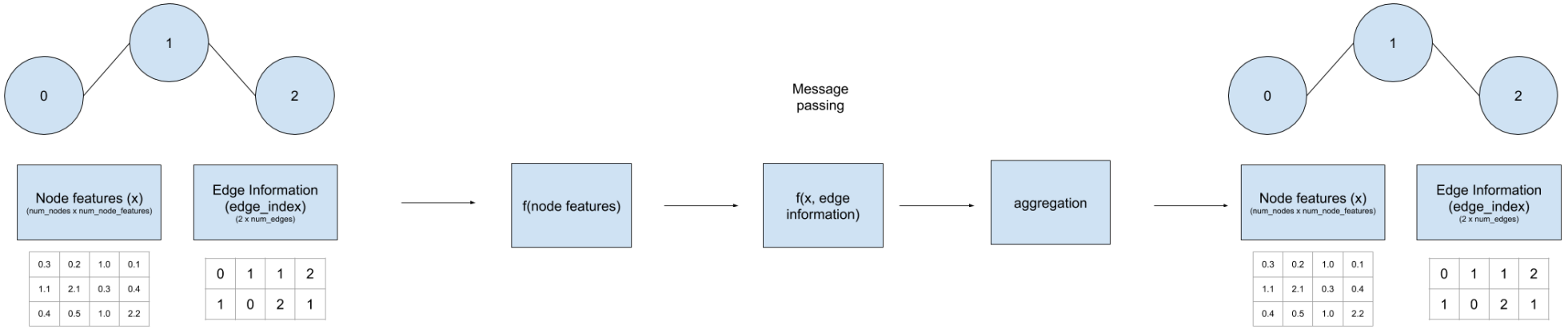
...

...

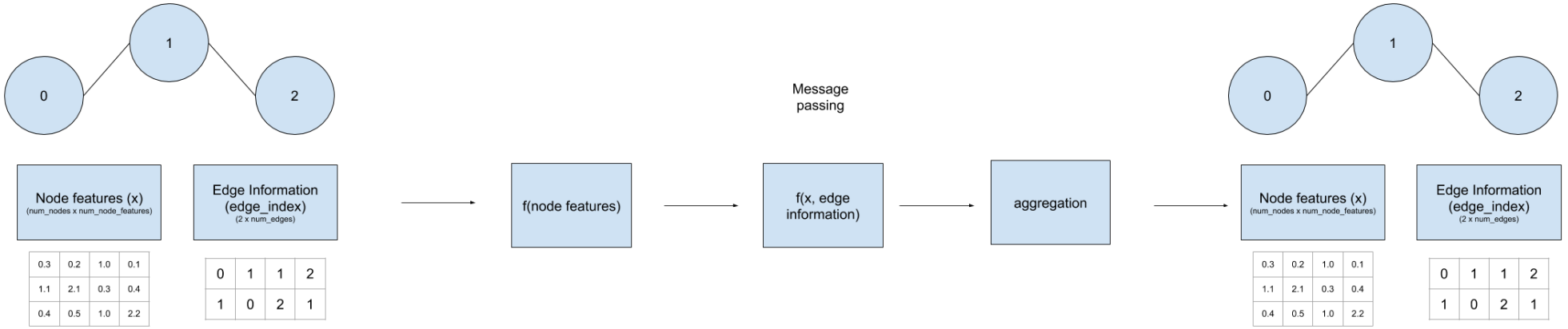
Graph Neural Networks



Graph Neural Networks



Graph Neural Networks



Why?

- Sparse data: eg. 3D meshes

Why?

- Sparse data: eg. 3D meshes

FAUST Dataset (Bogo et. al.)

Figure 1: FAUST dataset: Example scans of all 10 subjects (all professional models) showing the range of ages and body shapes. A sampling of the poses shows the wide pose variation.

Why?

- Sparse data: eg. 3D meshes

DynamicFAUST Dataset (Bogo et. al.)

Figure 1: **Dynamic FAUST**. We present a new 4D dataset containing more than one hundred dynamic performances of 10 subjects. We provide raw 3D scans (meshes) at 60 frames per second and dense ground-truth correspondences between them, obtained with a novel technique that combines shape and appearance to obtain accurate temporal mesh registration.

Why?

- Sparse data: eg. 3D meshes

SHREC (Deformable shapes) Cosmo et. al.

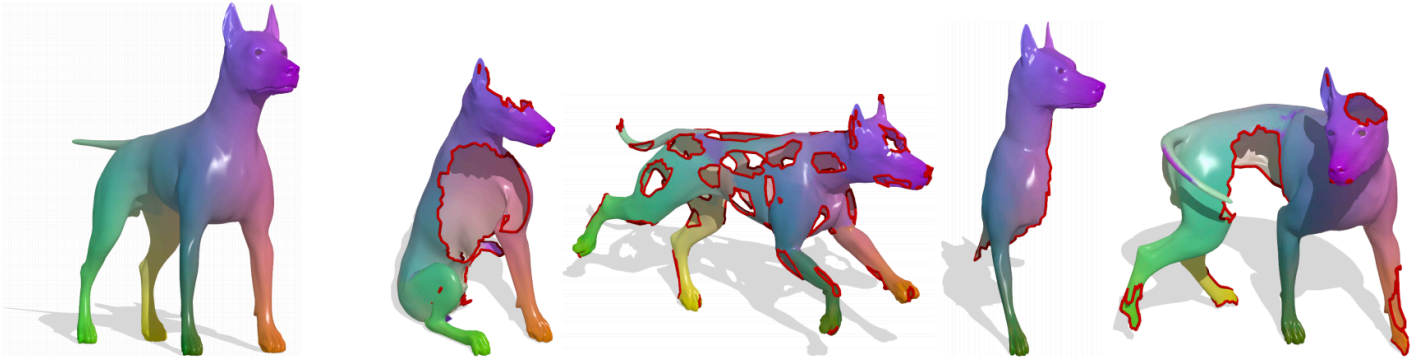


Figure 1: *A subset of shapes from the benchmark. Participants are required to produce a point-wise correspondence (either sparse or dense) between a full template in a neutral pose (left) and its deformed versions with missing parts. The dataset includes 8 shape classes, for a total of 599 3D models. In this figure corresponding points have the same color, while shape boundaries are marked by a red contour.*

Why?

- Sparse data: eg. 3D meshes

CoMA 3D Faces Ranjan et. al.

Fig. 8. Samples from the dataset

Why?

- Sparse data: eg. indoor scans

S3DIS Large-scale Indoor Spaces (Armeni et. al.)

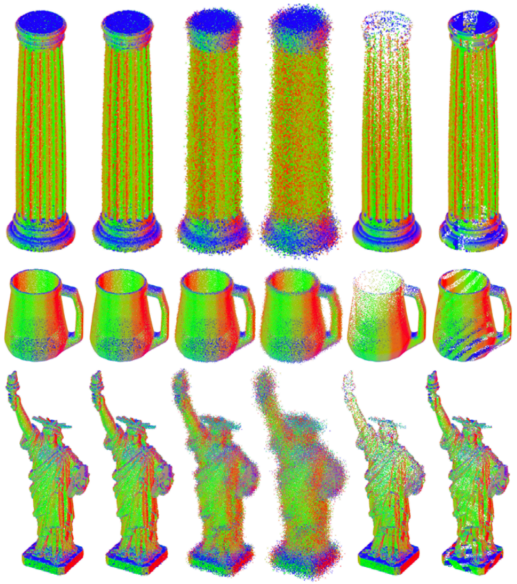
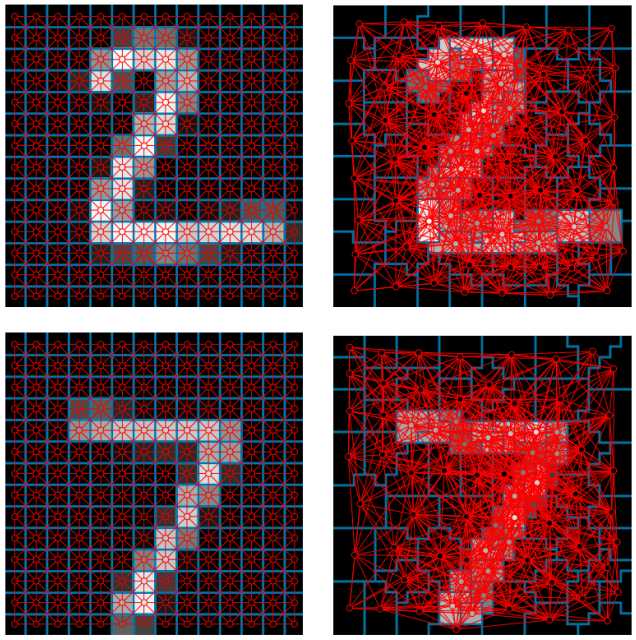


Figure 6: *A few point clouds from our dataset with their variants. The base shape is shown in the left column, followed by variants with three noise levels (*0.0025, 0.012 *and* 0.024*), and two non-uniform sampling schemes. Shapes are colored according to their unoriented normals.*

Why?

- Manifold learning: eg. pre-processed super-pixels

MNIST Superpixels (Mo



Regular grid Superpixels

Figure 2. Representation of images as graphs. Left: regular grid (the graph is fixed for all images). Right: graph of superpixel adjacency (different for each image). Vertices are shown as red circles, edges as red lines.

Why?

- Manifold learning: molecular properties

MoleculeNet (Wu et. al.)

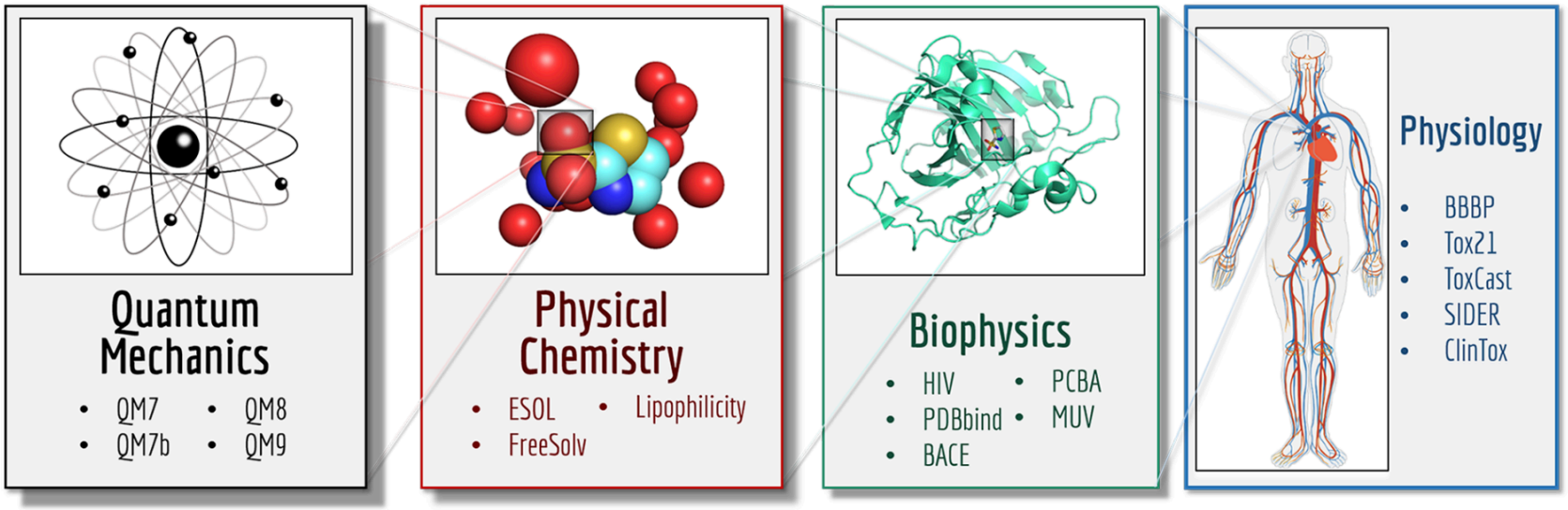


Figure 2: Tasks in different datasets focus on different levels of properties of molecules.

Why?

- Social and Interaction Graphs
  - Reddit social graph (users, posts, comments, upvotes are nodes)
  - Amazon Reviews dataset (review, rating, user are nodes)
  - Citation Networks (authors, papers are nodes, references are edges)
  - Bitcoin Network (transaction cliques, trust models, etc.)

Common Preprocessing Tricks

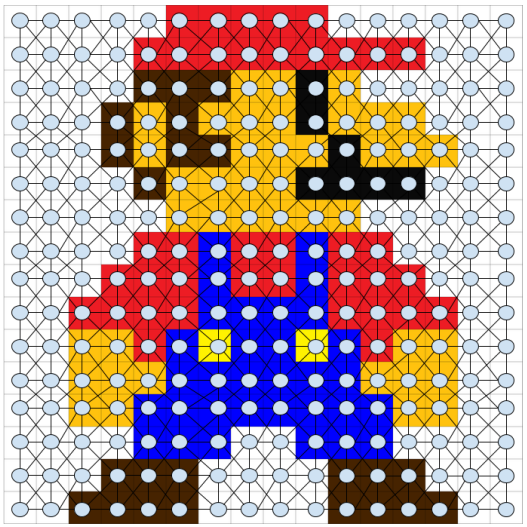
- Hand-crafted node features

Common Preprocessing Tricks

- Hand-crafted node features
- Only look at K Nearest Neighbors

Common Preprocessing Tricks

- Hand-crafted node features
- Only look at K Nearest Neighbors



Common Preprocessing Tricks

- Hand-crafted node features
- Only look at K Nearest Neighbors
- Random sampling of K edges

Common Preprocessing Tricks

- Hand-crafted node features
- Only look at K Nearest Neighbors
- Random sampling of K edges

Common Layers in literature

- GCNConv (Kipf. et. al.)

Semi-Supervised Classification with Graph Convolutional Networks

Thomas N. Kipf, Max Welling

(Submitted on 9 Sep 2016 (v1), last revised 22 Feb 2017 (this version, v4))

We present a scalable approach for semi-supervised learning on graph-structured data that is based on an efficient variant of convolutional neural networks which operate directly on graphs. W the choice of our convolutional architecture via a localized first-order approximation of spectral graph convolutions. Our model scales linearly in the number of graph edges and learns hidden l representations that encode both local graph structure and features of nodes. In a number of experiments on citation networks and on a knowledge graph dataset we demonstrate that our appr outperforms related methods by a significant margin.

Comments: Published as a conference paper at ICLR 2017
Subjects: **Machine Learning (cs.LG)**; Machine Learning (stat.ML)
Cite as: **arXiv:1609.02907 [cs.LG]**
(or **arXiv:1609.02907v4 [cs.LG]** for this version)

Common Layers in literature

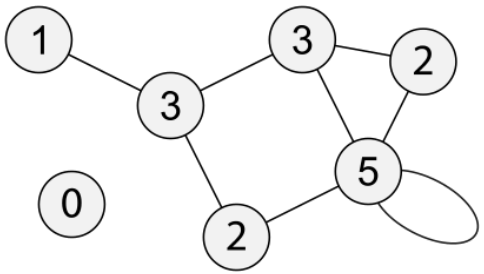
- GCNConv (Kipf. et. al.)
- Semi-supervised task (few labels)

Common Layers in literature

- GCNConv (Kipf. et. al.)
- Semi-supervised task (few labels)

$$\mathbf{X}' = \hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-1/2}\mathbf{X}\mathbf{\Theta},$$

where $\hat{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ denotes the adjacency matrix with inserted self-loops and $\hat{D}_{ii} = \sum_{j=0} \hat{A}_{ij}$ its diagonal degree matrix.



Common Layers in literature

- GCNConv (Kipf. et. al.)
- Semi-supervised task (few labels)
- Results on Citation Networks and Knowledge Graphs

$$\mathbf{X}' = \hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-1/2}\mathbf{X}\mathbf{\Theta},$$

where $\hat{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ denotes the adjacency matrix with inserted self-loops and $\hat{D}_{ii} = \sum_{j=0} \hat{A}_{ij}$ its diagonal degree matrix.

Common Layers in literature

- GCNConv (Kipf. et. al.)
- Semi-supervised task (few labels)
- Results on Citation Networks and Knowledge Graphs
- Citation: BoW for documents + list of citation links between documents

Common Layers in literature

- GraphConv (Morris. et. al.)

Weisfeiler and Leman Go Neural: Higher-order Graph Neural Networks

Christopher Morris, Martin Ritzert, Matthias Fey, William L. Hamilton, Jan Eric Lenssen, Gaurav Rattan, Martin Grohe

(Submitted on 4 Oct 2018 (v1), last revised 16 Nov 2018 (this version, v2))

In recent years, graph neural networks (GNNs) have emerged as a powerful neural architecture to learn vector representations of nodes and graphs in a supervised, end-to- have only been evaluated empirically---showing promising results. The following work investigates GNNs from a theoretical point of view and relates them to the 1-dimens isomorphism heuristic (1-WL). We show that GNNs have the same expressiveness as the 1-WL in terms of distinguishing non-isomorphic (sub-)graphs. Hence, both algorith shortcomings. Based on this, we propose a generalization of GNNs, so-called k-dimensional GNNs (k-GNNs), which can take higher-order graph structures at multiple scale order structures play an essential role in the characterization of social networks and molecule graphs. Our experimental evaluation confirms our theoretical findings as well information is useful in the task of graph classification and regression.

Comments: Extended version with proofs, accepted at AAAI 2019
Subjects: **Machine Learning (cs.LG)**; Artificial Intelligence (cs.AI); Computer Vision and Pattern Recognition (cs.CV); Neural and Evolutionary Computing (cs.NE); Machine Learning (stat.ML)
Cite as: arXiv:1810.02244 **[cs.LG]**
(or arXiv:1810.02244v2 **[cs.LG]** for this version)

Common Layers in literature

- GraphConv (Morris. et. al.)

$$\mathbf{x}_i' = \mathbf{\Theta}_1 \mathbf{x}_i + \sum_{j \in \mathcal{N}(i)} \mathbf{\Theta}_2 \mathbf{x}_j .$$

Common Layers in literature

- GraphSAGE (Hamilton. et. al.)

Inductive Representation Learning on Large Graphs

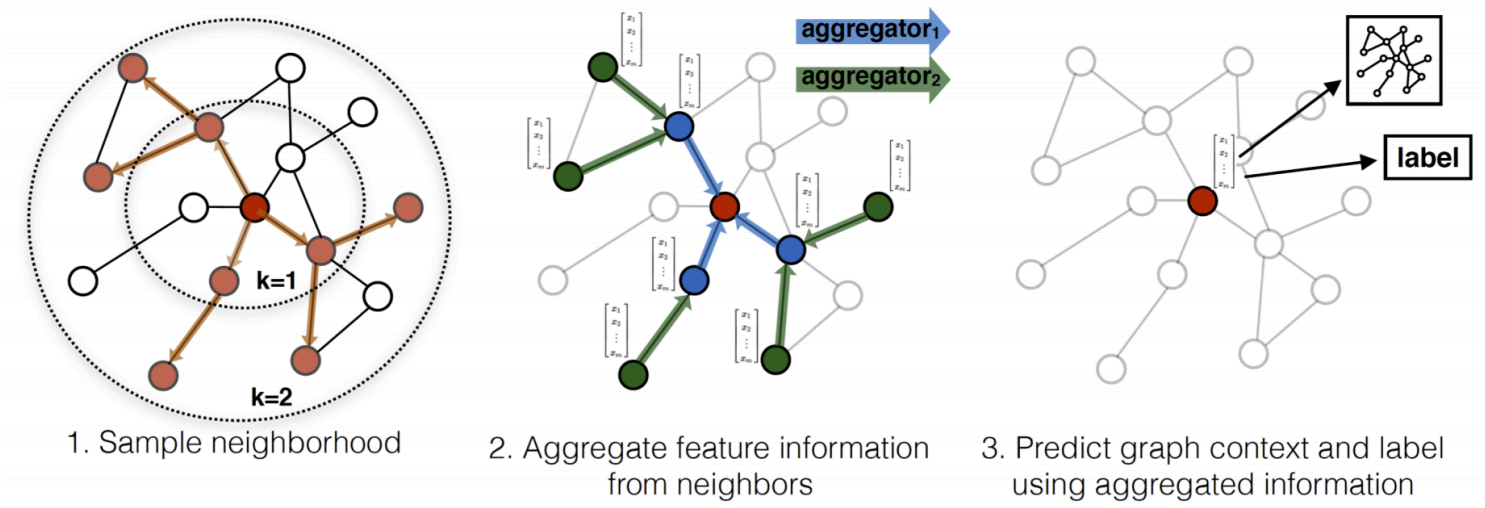
William L. Hamilton, Rex Ying, Jure Leskovec

(Submitted on 7 Jun 2017 (v1), last revised 10 Sep 2018 (this version, v4))

Low-dimensional embeddings of nodes in large graphs have proved extremely useful in a variety of prediction tasks, from content recommendation to identi approaches require that all nodes in the graph are present during training of the embeddings; these previous approaches are inherently transductive and do we present GraphSAGE, a general, inductive framework that leverages node feature information (e.g., text attributes) to efficiently generate node embeddings individual embeddings for each node, we learn a function that generates embeddings by sampling and aggregating features from a node's local neighborhoo on three inductive node-classification benchmarks: we classify the category of unseen nodes in evolving information graphs based on citation and Reddit pos generalizes to completely unseen graphs using a multi-graph dataset of protein-protein interactions.

Common Layers in literature

- GraphSAGE (Hamilton. et. al.)



1. Sample neighborhood

2. Aggregate feature information from neighbors

3. Predict graph context and label using aggregated information

Common Layers in literature

- GraphSAGE (Hamilton. et. al.)

Algorithm 1: GraphSAGE embedding generation (i.e., forward propagation) algorithm

Input : Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$; input features $\{\mathbf{x}_v, \forall v \in \mathcal{V}\}$; depth K; weight matrices $\mathbf{W}^k, \forall k \in \{1, ..., K\}$; non-linearity σ; differentiable aggregator functions $\text{AGGREGATE}_k, \forall k \in \{1, ..., K\}$; neighborhood function $\mathcal{N} : v \rightarrow 2^{\mathcal{V}}$

Output : Vector representations $\mathbf{z}_v$ for all $v \in \mathcal{V}$

$\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$;
for $k = 1...K$ **do**
  for $v \in \mathcal{V}$ **do**
    $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$;
    $\mathbf{h}_v^k \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k)\right)$
  end
  $\mathbf{h}_v^k \leftarrow \mathbf{h}_v^k / \|\mathbf{h}_v^k\|_2, \forall v \in \mathcal{V}$
end
$\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$

Common Layers in literature

- GraphSAGE (Hamilton. et. al.)

$$\hat{\mathbf{x}}_i = \boldsymbol{\Theta} \cdot \text{mean}_{j \in \mathcal{N}(i) \cup \{i\}}(\mathbf{x}_j)$$

$$\mathbf{x}'_i = \frac{\hat{\mathbf{x}}_i}{\|\hat{\mathbf{x}}_i\|_2}.$$

Common Layers in literature

- GraphSAGE (Hamilton. et. al.)
- classifying academic papers into subjects based on citations
- classify reddit posts to subreddits
- classify protein functions across PPI graphs

Common Layers in literature

- Graph Attention Networks (Veličković et. al.)

Graph Attention Networks

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, Yoshua Bengio

(Submitted on 30 Oct 2017 (v1), last revised 4 Feb 2018 (this version, v3))

We present graph attention networks (GATs), novel neural network architectures that operate on graph-structured data, leveraging masked self-attentional layers to address the shortcomings of prior methods based on graph convolutions or their approximations. By stacking layers in which nodes are able to attend over their neighborhoods' features, we enable (implicitly) specifying different weights to different nodes in a neighborhood, without requiring any kind of costly matrix operation (such as inversion) or depending on knowing the graph structure upfront. In this way, we address several key challenges of spectral-based graph neural networks simultaneously, and make our model readily applicable to inductive as well as transductive problems. Our GAT models have achieved or matched state-of-the-art results across four established transductive and inductive graph benchmarks: the Cora, Citeseer and Pubmed citation network datasets, as well as a protein-protein interaction dataset (wherein test graphs remain unseen during training).

Comments: To appear at ICLR 2018. 12 pages, 2 figures
Subjects: **Machine Learning (stat.ML)**; Artificial Intelligence (cs.AI); Machine Learning (cs.LG); Social and Information Networks (cs.SI)
Cite as: arXiv:1710.10903 **[stat.ML]**
(or arXiv:1710.10903v3 **[stat.ML]** for this version)

Common Layers in literature

- Graph Attention Networks (Veličković et. al.)

The graph attentional operator from the "Graph Attention Networks" paper

$$\mathbf{x}'_i = \alpha_{i,i}\mathbf{\Theta}\mathbf{x}_i + \sum_{j \in \mathcal{N}(i)} \alpha_{i,j}\mathbf{\Theta}\mathbf{x}_j,$$

where the attention coefficients $\alpha_{i,j}$ are computed as

$$\alpha_{i,j} = \frac{\exp\left(\text{LeakyReLU}\left(\mathbf{a}^\top [\mathbf{\Theta}\mathbf{x}_i \,\|\, \mathbf{\Theta}\mathbf{x}_j]\right)\right)}{\sum_{k \in \mathcal{N}(i)\cup\{i\}} \exp\left(\text{LeakyReLU}\left(\mathbf{a}^\top [\mathbf{\Theta}\mathbf{x}_i \,\|\, \mathbf{\Theta}\mathbf{x}_k]\right)\right)}.$$

Common Layers in literature

- Graph Attention Networks (Veličković et. al.)

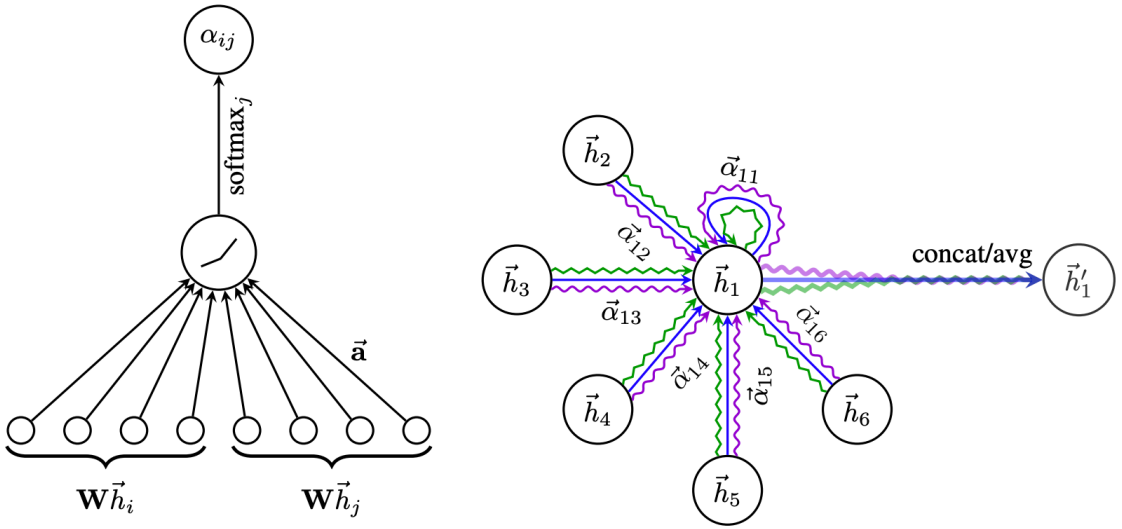


Figure 1: **Left:** The attention mechanism $a(\mathbf{W}\vec{h}_i, \mathbf{W}\vec{h}_j)$ employed by our model, parametrized by a weight vector $\vec{\mathbf{a}} \in \mathbb{R}^{2F'}$, applying a LeakyReLU activation. **Right:** An illustration of multi-head attention (with $K = 3$ heads) by node 1 on its neighborhood. Different arrow styles and colors denote independent attention computations. The aggregated features from each head are concatenated or averaged to obtain $\vec{h}'_1$.

Common Layers in literature

- Graph Attention Networks (Veličković et. al.)
- Citation Networks
- Protein-Protein Interaction

Graph Neural Networks

- Available software
  - PyTorch Geometric: https://github.com/rusty1s/pytorch_geometric
  - DGL: https://www.dgl.ai/
  - Deepmind GraphNets: https://github.com/deepmind/graph_nets
  - Rolling your own using TensorFlow, SciPy or PyTorch sparse operations

PyTorch Geometric

PyTorch Geometric

.Author

Matthias Fey
rusty1s

PhD student @ TU Dortmund University - Interested in Representation Learning on Graphs and Manifolds; PyTorch, CUDA, Vim and macOS Enthusiast

Dortmund, Germany

matthias.fey@tu-dortmund.de

http://rusty1s.github.io

PyTorch Geometric

Data Representation

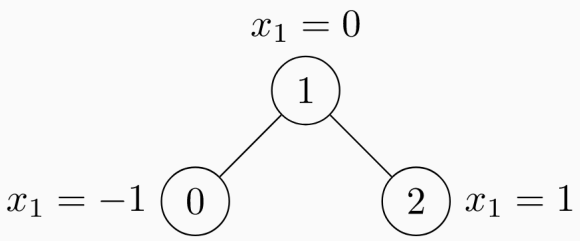
A graph is used to model pairwise relations (edges) between objects (nodes). A single graph in PyTorch Geometric is described by an instance of `torch_geometric.data.Data`, which holds the following attributes by default:

- `data.x`: Node feature matrix with shape `[num_nodes, num_node_features]`
- `data.edge_index`: Graph connectivity in COO format with shape `[2, num_edges]` and type `torch.long`
- `data.edge_attr`: Edge feature matrix with shape `[num_edges, num_edge_features]`
- `data.y`: Target to train against (may have arbitrary shape)
- `data.pos`: Node position matrix with shape `[num_nodes, num_dimensions]`

PyTorch Geometric

- Data Representation

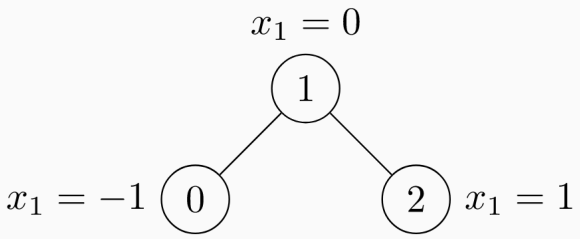
```
import torch
from torch_geometric.data import Data

edge_index = torch.tensor([[0, 1, 1, 2],
                           [1, 0, 2, 1]], dtype=torch.long)
x = torch.tensor([[-1], [0], [1]], dtype=torch.float)

data = Data(x=x, edge_index=edge_index)
>>> Data(x=[3, 1], edge_index=[2, 4])
```



PyTorch Geometric

- Data Representation

```
import torch
from torch_geometric.data import Data

edge_index = torch.tensor([[0, 1],
                           [1, 0],
                           [1, 2],
                           [2, 1]], dtype=torch.long)
x = torch.tensor([[-1], [0], [1]], dtype=torch.float)

data = Data(x=x, edge_index=edge_index.t().contiguous())
>>> Data(x=[3, 1], edge_index=[2, 4])
```



PyTorch Geometric

- Data Representation

```
print(data.keys)
>>> ['x', 'edge_index']

print(data['x'])
>>> tensor([[0.0],
            [1.0],
            [2.0]])

for key, item in data:
    print('{} found in data)
>>> x found in data
>>> edge_index found in data

'edge_attr' in data
>>> False

data.num_nodes
>>> 3
```

```
data.num_edges
>>> 4

data.num_features
>>> 1

data.contains_isolated_nodes()
>>> False

data.contains_self_loops()
>>> False

data.is_directed()
>>> False
```

```
# Transfer data object to GPU.
device = torch.device('cuda')
data = data.to(device)
```

PyTorch Geometric

- Common Datasets
- - over 60 graph kernel benchmark datasets
  - - IMDB-BINARY, REDDIT-BINARY, PROTEINS
  - - Cora, CiteSeer, PubMed, Cora-Full
  - - Coauthor CS/Physics
  - - Amazon Computers / Photo datasets
  - - molecule datasets QM7b, QM9
  - - protein-protein interaction graphs from Hamilton et. al.
  - - temporal datasets:
  - - Bitcoin-OTC
  - - ICEWS
  - - GDELT

Embedded datasets:
- MNIST superpixels
- FAUST
- ModelNet10/40
- ShapeNet
- COMA 3D faces dataset
- PCPNet

PyTorch Geometric

- Common Datasets

```
from torch_geometric.datasets import TUDataset

dataset = TUDataset(root='/tmp/ENZYMES', name='ENZYMES')
>>> ENZYMES(600)

len(dataset)
>>> 600

dataset.num_classes
>>> 6

dataset.num_features
>>> 21
```


PyTorch Geometric

- Common Datasets

If you are unsure whether the dataset is already shuffled before you split, you can randomly permutate it by running:

```
dataset = dataset.shuffle()
>>> ENZYMES(600)
```

This is equivalent of doing:

```
perm = torch.randperm(len(dataset))
dataset = dataset[perm]
>> ENZYMES(600)
```

PyTorch Geometric

- Transforms / Pre-processing

```
from torch_geometric.datasets import ShapeNet

dataset = ShapeNet(root='/tmp/ShapeNet', category='Airplane')

data[0]
>>> Data(pos=[2518, 3], y=[2518])
```

We can convert the point cloud dataset into a graph dataset by generating nearest neighbor graphs from the point clouds via transforms:

```
import torch_geometric.transforms as T
from torch_geometric.datasets import ShapeNet

dataset = ShapeNet(root='/tmp/ShapeNet', category='Airplane',
                   pre_transform=T.KNNGraph(k=6))

data[0]
>>> Data(edge_index=[2, 17768], pos=[2518, 3], y=[2518])
```

PyTorch Geometric

- Example: Graph ConvNets (Kipf et. al. 2016)

```
from torch_geometric.datasets import Planetoid

dataset = Planetoid(root='/tmp/Cora', name='Cora')
>>> Cora()
```


PyTorch Geometric

- Example: Graph ConvNets (Kipf et. al. 2016)

```python
import torch
import torch.nn.functional as F
from torch_geometric.nn import GCNConv

class Net(torch.nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = GCNConv(dataset.num_features, 16)
        self.conv2 = GCNConv(16, dataset.num_classes)

    def forward(self, data):
        x, edge_index = data.x, data.edge_index

        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x = F.dropout(x, training=self.training)
        x = self.conv2(x, edge_index)

        return F.log_softmax(x, dim=1)
```

PyTorch Geometric

- Example: Graph ConvNets (Kipf et. al. 2016)

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = Net().to(device)
data = dataset[0].to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.01, weight_decay=5e-4)

model.train()
for epoch in range(200):
    optimizer.zero_grad()
    out = model(data)
    loss = F.nll_loss(out[data.train_mask], data.y[data.train_mask])
    loss.backward()
    optimizer.step()
```

PyTorch Geometric

- Example: Graph ConvNets (Kipf et. al. 2016)

```
model.eval()
_, pred = model(data).max(dim=1)
correct = pred[data.test_mask].eq(data.y[data.test_mask]).sum().item()
acc = correct / data.test_mask.sum().item()
print('Accuracy: {:.4f}'.format(acc))
>>> Accuracy: 0.8150
```

PyTorch Geometric

- Example: Graph ConvNets (Kipf et. al. 2016)

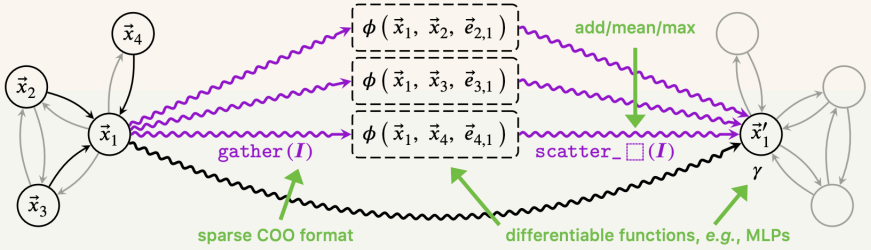
```python
import torch
import torch.nn.functional as F
from torch_geometric.nn import GCNConv

class Net(torch.nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = GCNConv(dataset.num_features, 16)
        self.conv2 = GCNConv(16, dataset.num_classes)

    def forward(self, data):
        x, edge_index = data.x, data.edge_index

        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x = F.dropout(x, training=self.training)
        x = self.conv2(x, edge_index)

        return F.log_softmax(x, dim=1)
```

PyTorch Geometric

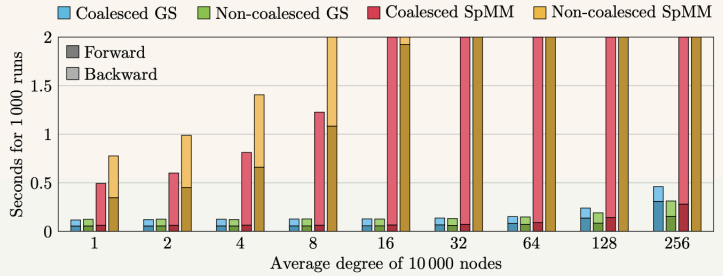
- Message-Passing

An Intuitive Message Passing Interface based on Gather and Scatter Operations

$$\vec{x}_i' = \gamma\left(\vec{x}_i, \square_{j \in \mathcal{N}(i)} \phi\left(\vec{x}_i, \vec{x}_j, \vec{e}_{j,i}\right)\right)$$




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

    def forward(self, x, edge_index, e=None):
        return self.propagate(edge_index, x=x, e=e)

    def message(self, x_i, x_j, e):
```



Sparse-Matrix Multiplication (SpMM) needs *coalesced* sparse tensors → backward pass is inherently slow

Gather/Scatter (GS):

- ✓ input does not need to be coalesced
- ✓ can integrate central node and edge information
- × non-deterministic by nature on GPU

PyTorch Geometric

- Message-Passing

class **MessagePassing**(*aggr='add', flow='source_to_target'*) [source]

Base class for creating message passing layers

$$\mathbf{x}_i' = \gamma_{\Theta} \left(\mathbf{x}_i, \square_{j \in \mathcal{N}(i)} \, \phi_{\Theta} \left(\mathbf{x}_i, \mathbf{x}_j, \mathbf{e}_{i,j} \right) \right),$$

where $\square$ denotes a differentiable, permutation invariant function, *e.g.*, sum, mean or max, and γ_{Θ} and ϕ_{Θ} denote differentiable functions such as MLPs. See here for the accompanying tutorial.

Parameters:

- **aggr** (*string*, *optional*) – The aggregation scheme to use (`"add"` , `"mean"` or `"max"`). (default: `"add"`)
- **flow** (*string*, *optional*) – The flow direction of message passing (`"source_to_target"` or `"target_to_source"`). (default: `"source_to_target"`)

PyTorch Geometric

- Message-Pass

message(*x_j*) [source]

Constructs messages in analogy to ϕ_{Θ} for each edge in $(i, j) \in \mathcal{E}$. Can take any argument which was initially passed to `propagate()`. In addition, features can be lifted to the source node i and target node j by appending `_i` or `_j` to the variable name, *.e.g.* `x_i` and `x_j`.

propagate(*edge_index*, *size=None*, ***kwargs*) [source]

The initial call to start propagating messages.

Parameters:

- **edge_index** (*Tensor*) – The indices of a general (sparse) assignment matrix with shape `[N, M]` (can be directed or undirected).
- **size** (*list* or *tuple*, *optional*) – The size `[N, M]` of the assignment matrix. If set to `None`, the size is tried to get automatically inferrred. (default: `None`)
- ****kwargs** – Any additional data which is needed to construct messages and to update node embeddings.

update(*aggr_out*) [source]

Updates node embeddings in analogy to γ_{Θ} for each node $i \in \mathcal{V}$. Takes in the output of aggregation as first argument and any argument which was initially passed to `propagate()`.

PyTorch Geometric

- Example: Graph ConvNets (Kipf et. al. 2016)

The GCN layer is mathematically defined as

$$\mathbf{x}_i^{(k)} = \sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{1}{\sqrt{\deg(i)} \cdot \sqrt{deg(j)}} \cdot \left(\mathbf{\Theta} \cdot \mathbf{x}_j^{(k-1)} \right),$$

where neighboring node features are first transformed by a weight matrix $\mathbf{\Theta}$, normalized by their degree, and finally summed up. This formula can be divided into the following steps:

1. Add self-loops to the adjacency matrix.
2. Linearly transform node feature matrix.
3. Normalize node features in ϕ.
4. Sum up neighboring node features (`"add"` aggregation).
5. Return new node embeddings in γ.

PyTorch Geometric

- Example: Graph ConvNets (Kipf et. al. 2016)

```python
import torch
from torch_geometric.nn import MessagePassing
from torch_geometric.utils import add_self_loops, degree

class GCNConv(MessagePassing):
    def __init__(self, in_channels, out_channels):
        super(GCNConv, self).__init__(aggr='add')  # "Add" aggregation.
        self.lin = torch.nn.Linear(in_channels, out_channels)

    def forward(self, x, edge_index):
        # x has shape [N, in_channels]
        # edge_index has shape [2, E]

        # Step 1: Add self-loops to the adjacency matrix.
        edge_index, _ = add_self_loops(edge_index, num_nodes=x.size(0))

        # Step 2: Linearly transform node feature matrix.
        x = self.lin(x)

        # Step 3-5: Start propagating messages.
        return self.propagate(edge_index, size=(x.size(0), x.size(0)), x=x)
```

```python
def message(self, x_j, edge_index, size):
    # x_j has shape [E, out_channels]

    # Step 3: Normalize node features.
    row, col = edge_index
    deg = degree(row, size[0], dtype=x_j.dtype)
    deg_inv_sqrt = deg.pow(-0.5)
    norm = deg_inv_sqrt[row] * deg_inv_sqrt[col]

    return norm.view(-1, 1) * x_j

def update(self, aggr_out):
    # aggr_out has shape [N, out_channels]

    # Step 5: Return new node embeddings.
    return aggr_out
```

PyTorch Geometric

- List of implemented methods

  - **SplineConv** from Fey *et al.*: SplineCNN: Fast Geometric Deep Learning with Continuous B-Spline Kernels (CVPR 2018)
  - **GCNConv** from Kipf and Welling: Semi-Supervised Classification with Graph Convolutional Networks (ICLR 2017)
  - **ChebConv** from Defferrard *et al.*: Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering (NIPS 2016)
  - **NNConv** adapted from Gilmer *et al.*: Neural Message Passing for Quantum Chemistry (ICML 2017)
  - **GATConv** from Veličković *et al.*: Graph Attention Networks (ICLR 2018)
  - **SAGEConv** from Hamilton *et al.*: Inductive Representation Learning on Large Graphs (NIPS 2017)
  - **GraphConv** from, *e.g.*, Morris *et al.*: Weisfeiler and Leman Go Neural: Higher-order Graph Neural Networks (AAAI 2019)
  - **GatedGraphConv** from Li *et al.*: Gated Graph Sequence Neural Networks (ICLR 2016)
  - **GINConv** from Xu *et al.*: How Powerful are Graph Neural Networks? (ICLR 2019)
  - **ARMAConv** from Bianchi *et al.*: Graph Neural Networks with Convolutional ARMA Filters (CoRR 2019)
  - **SGConv** from Wu *et al.*: Simplifying Graph Convolutional Networks (CoRR 2019)
  - **APPNP** from Klicpera *et al.*: Predict then Propagate: Graph Neural Networks meet Personalized PageRank (ICLR 2019)
  - **AGNNConv** from Thekumparampil *et al.*: Attention-based Graph Neural Network for Semi-Supervised Learning (CoRR 2017)
  - **RGCNConv** from Schlichtkrull *et al.*: Modeling Relational Data with Graph Convolutional Networks (ESWC 2018)
  - **SignedConv** from Derr *et al.*: Signed Graph Convolutional Network (ICDM 2018)
  - **DNAConv** from Fey: Just Jump: Dynamic Neighborhood Aggregation in Graph Neural Networks (ICLR-W 2019)

PyTorch Geometric

- List of implemented methods

  - **EdgeConv** from Wang *et al.*: Dynamic Graph CNN for Learning on Point Clouds (CoRR, 2018)
  - **PointConv** (including **Iterative Farthest Point Sampling**, dynamic graph generation based on **nearest neighbor** or **maximum distance**, and **k-NN interpolation**) from Qi *et al.*: PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation (CVPR 2017) and PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space (NIPS 2017)
  - **XConv** from Li *et al.*: PointCNN: Convolution On X-Transformed Points (official implementation) (NeurIPS 2018)
  - **PPFConv** from Deng *et al.*: PPFNet: Global Context Aware Local Features for Robust 3D Point Matching (CVPR 2018)
  - **GMMConv** from Monti *et al.*: Geometric Deep Learning on Graphs and Manifolds using Mixture Model CNNs (CVPR 2017)
  - **HypergraphConv** from Bai *et al.*: Hypergraph Convolution and Hypergraph Attention (CoRR 2019)
  - A **MetaLayer** for building any kind of graph network similar to the TensorFlow Graph Nets library from Battaglia *et al.*: Relational Inductive Biases, Deep Learning, and Graph Networks (CoRR 2018)
  - **GlobalAttention** from Li *et al.*: Gated Graph Sequence Neural Networks (ICLR 2016)
  - **Set2Set** from Vinyals *et al.*: Order Matters: Sequence to Sequence for Sets (ICLR 2016)
  - **Sort Pool** from Zhang *et al.*: An End-to-End Deep Learning Architecture for Graph Classification (AAAI 2018)
  - **Dense Differentiable Pooling** from Ying *et al.*: Hierarchical Graph Representation Learning with Differentiable Pooling (NeurIPS 2018)
  - **Graclus Pooling** from Dhillon *et al.*: Weighted Graph Cuts without Eigenvectors: A Multilevel Approach (PAMI 2007)

PyTorch Geometric

- List of implemented methods

  - **Voxel Grid Pooling** from, *e.g.*, Simonovsky and Komodakis: Dynamic Edge-Conditioned Filters in Convolutional Neural Networks on Graphs (CVPR 2017)
  - **Top-K Pooling** from Gao and Ji: Graph U-Net (ICLR 2019 submission) and Cangea *et al.*: Towards Sparse Hierarchical Graph Classifiers (NeurIPS-W 2018)
  - **Local Degree Profile** from Cai and Wang: A Simple yet Effective Baseline for Non-attribute Graph Classification (CoRR 2018)
  - **Jumping Knowledge** from Xu *et al.*: Representation Learning on Graphs with Jumping Knowledge Networks (ICML 2018)
  - **Deep Graph Infomax** from Veličković *et al.*: Deep Graph Infomax (ICLR 2019)
  - All variants of **Graph Auto-Encoders** from Kipf and Welling: Variational Graph Auto-Encoders (NIPS-W 2016) and Pan *et al.*: Adversarially Regularized Graph Autoencoder for Graph Embedding (IJCAI 2018)
  - **RENet** from Jin *et al.*: Recurrent Event Network for Reasoning over Temporal Knowledge Graphs (ICLR-W 2019)

DGL

DGL (https://dgl.ai)

- Team

New York University: Minjie Wang (*Project Lead*), Lingfan Yu, Quan Gan, Jake Zhao

NYU Shanghai: Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang

AWS: Da Zheng (*Project Lead*), Haibin Lin, Chao Ma, Damon Deng

Fudan University: Qipeng Guo

CQUPT: Hao Zhang

HKUST: Ziyue Huang

DGL

We benchmark both GCN and GAT models on several popular datasets following their original settings. The testbed is one AWS p3.2xlarge instance with NVIDIA V100 GPU (16GB memory).

Dataset	#V	#E	Models	DGL(v0.2) Times(s)	PyG Time(s)	DGL(v0.3) Time(s)
Cora	3K	11K	GCN GAT	0.685 9.727	0.482 1.248	0.619 1.389
Citeseer	3K	9K	GCN GAT	0.670 9.018	0.490 1.254	0.631 1.363
Pubmed	20K	889K	GCN GAT	0.694 26.186	0.485 1.509	0.603 1.381
Reddit	**232K**	**114M**	**GCN**	**OOM**	**OOM**	**25.30**

Thank You