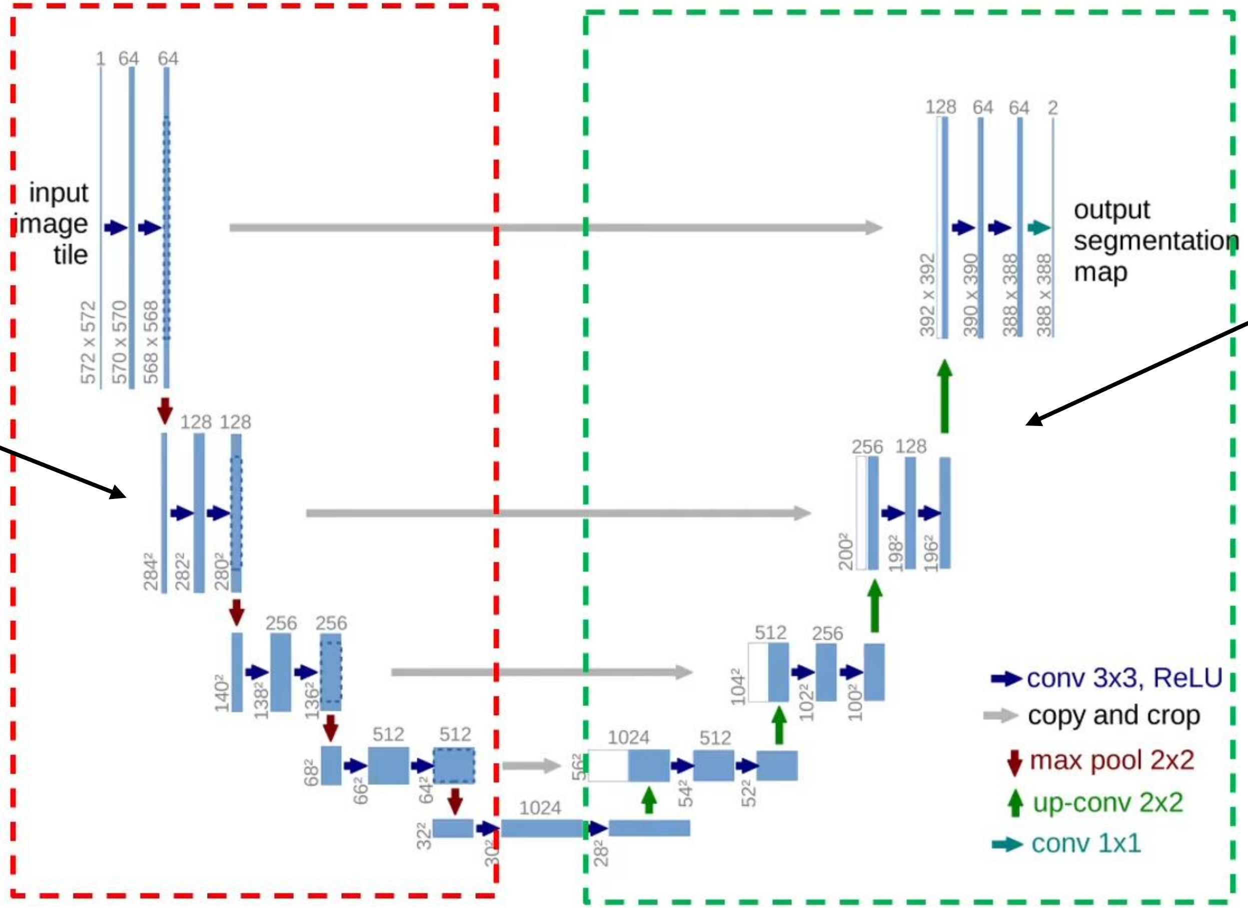
# Graph U-Net

Ibrahim Aitkazin

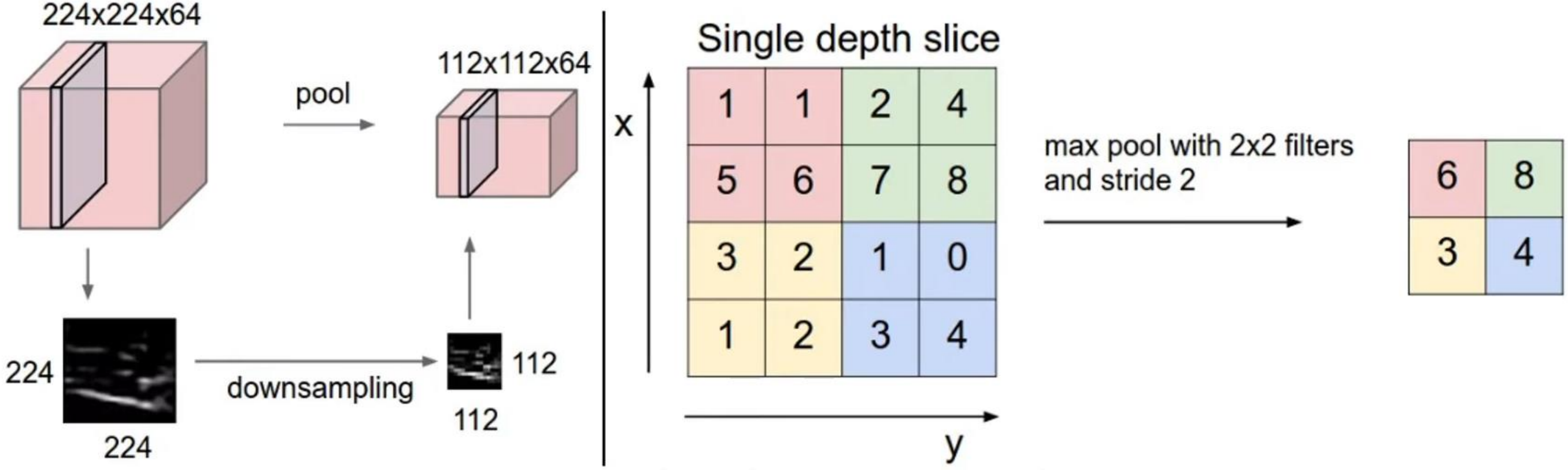
# U-net

Contracting path
to capture context




Expansive path
for precise
localization

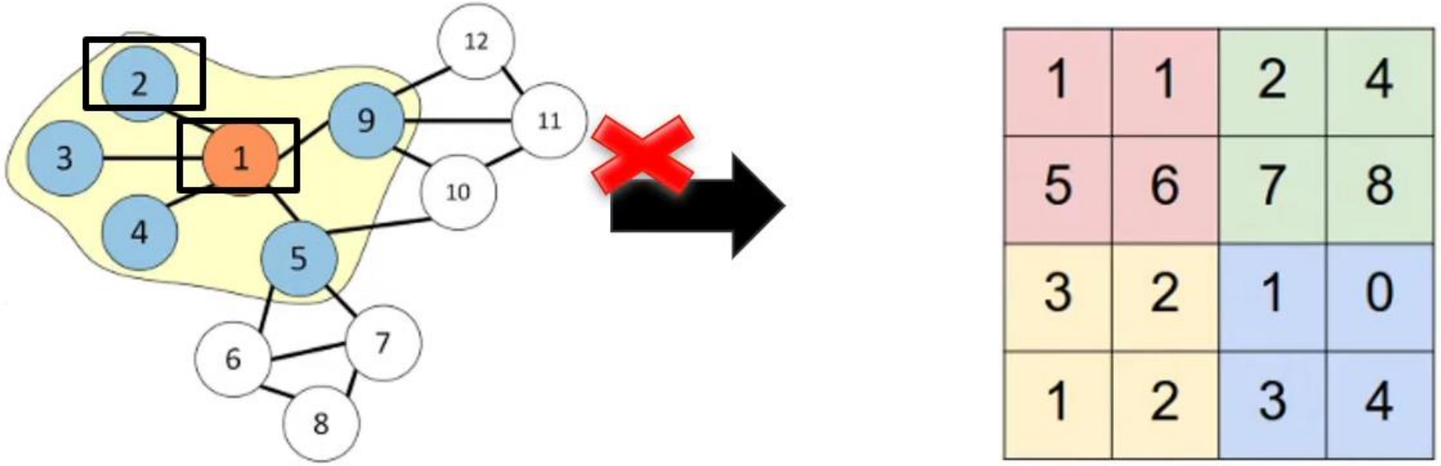
# Pooling




Pooling progressively reduce the spatial size of the representation to reduce the number of parameters and computation in the network.

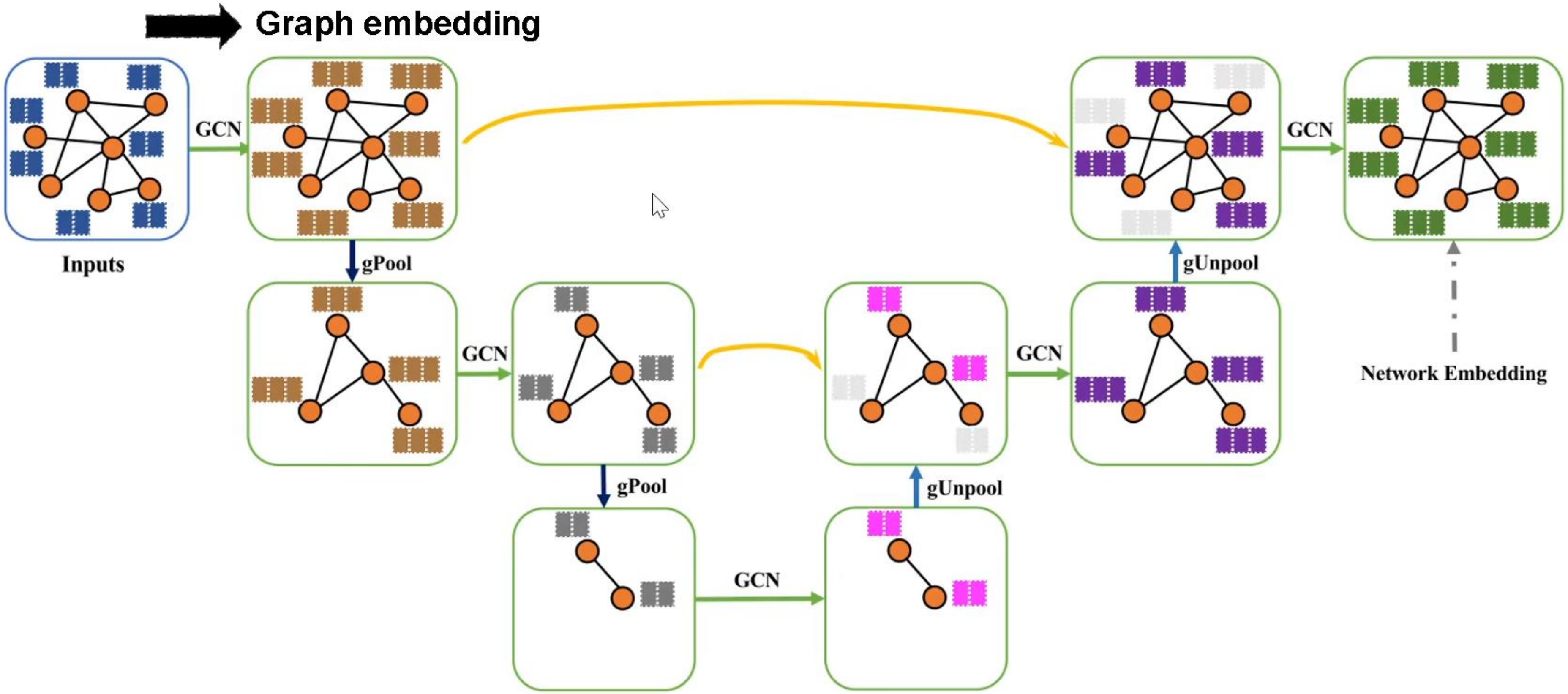
# Problem

Pooling can not measure the local information among feature vectors




However, we cannot directly apply these pooling operations to graphs. In particular, there is no locality information among nodes in graphs. Thus the partition operation is not applicable on graphs. The global pooling operation will

# Graph U-net



Graph embedding
Inputs
GCN
gPool
GCN
gPool
GCN
gUnpool
GCN
gUnpool
GCN
Network Embedding

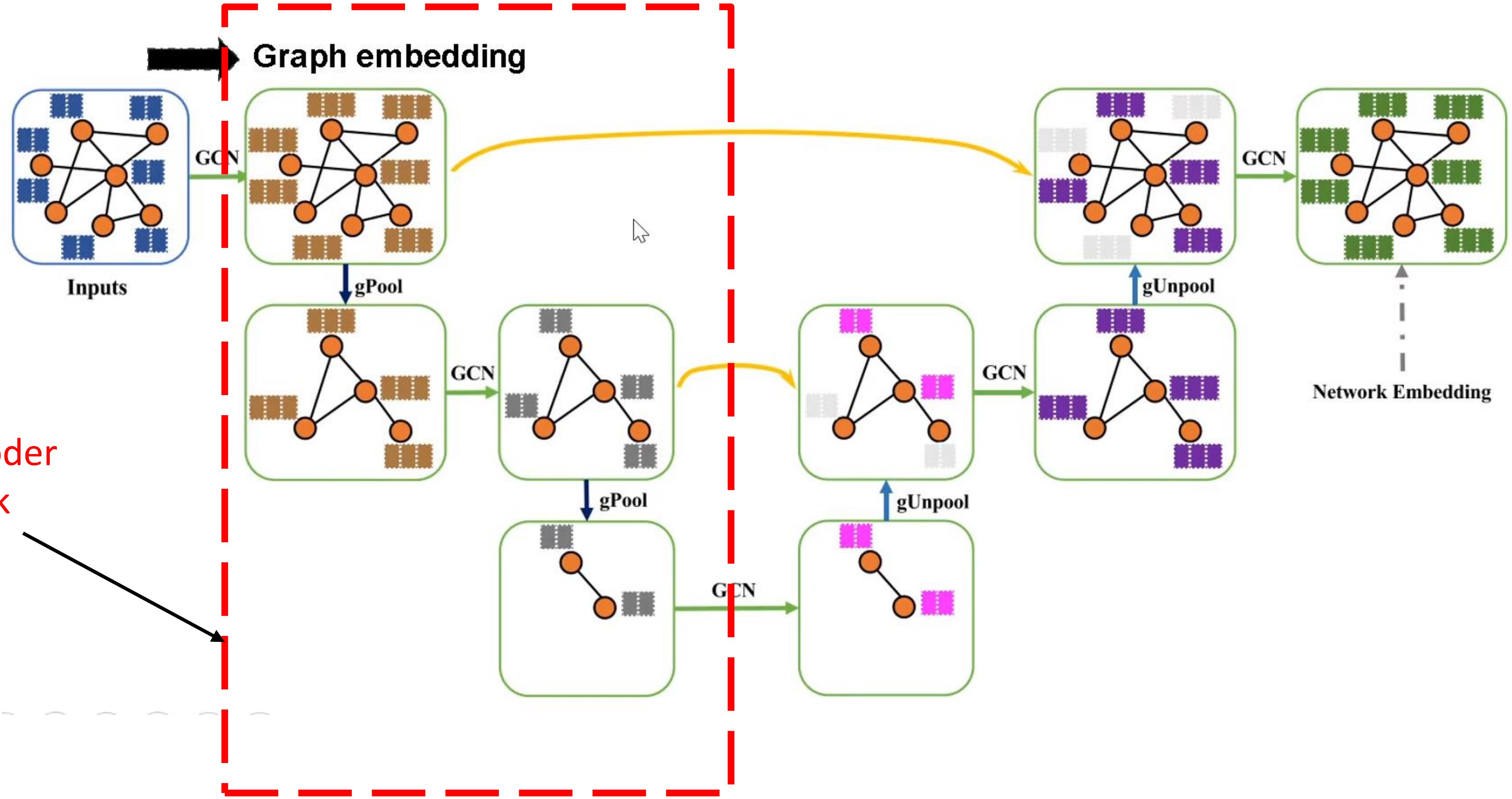
# Encoder

Reducing graph size

Aggregate first order neighbor information

$$X_{\ell+1} = \sigma(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}X_\ell W_\ell)$$

**Inputs -> GCN -> gPool -> GCN....-> gPool -> GCN**



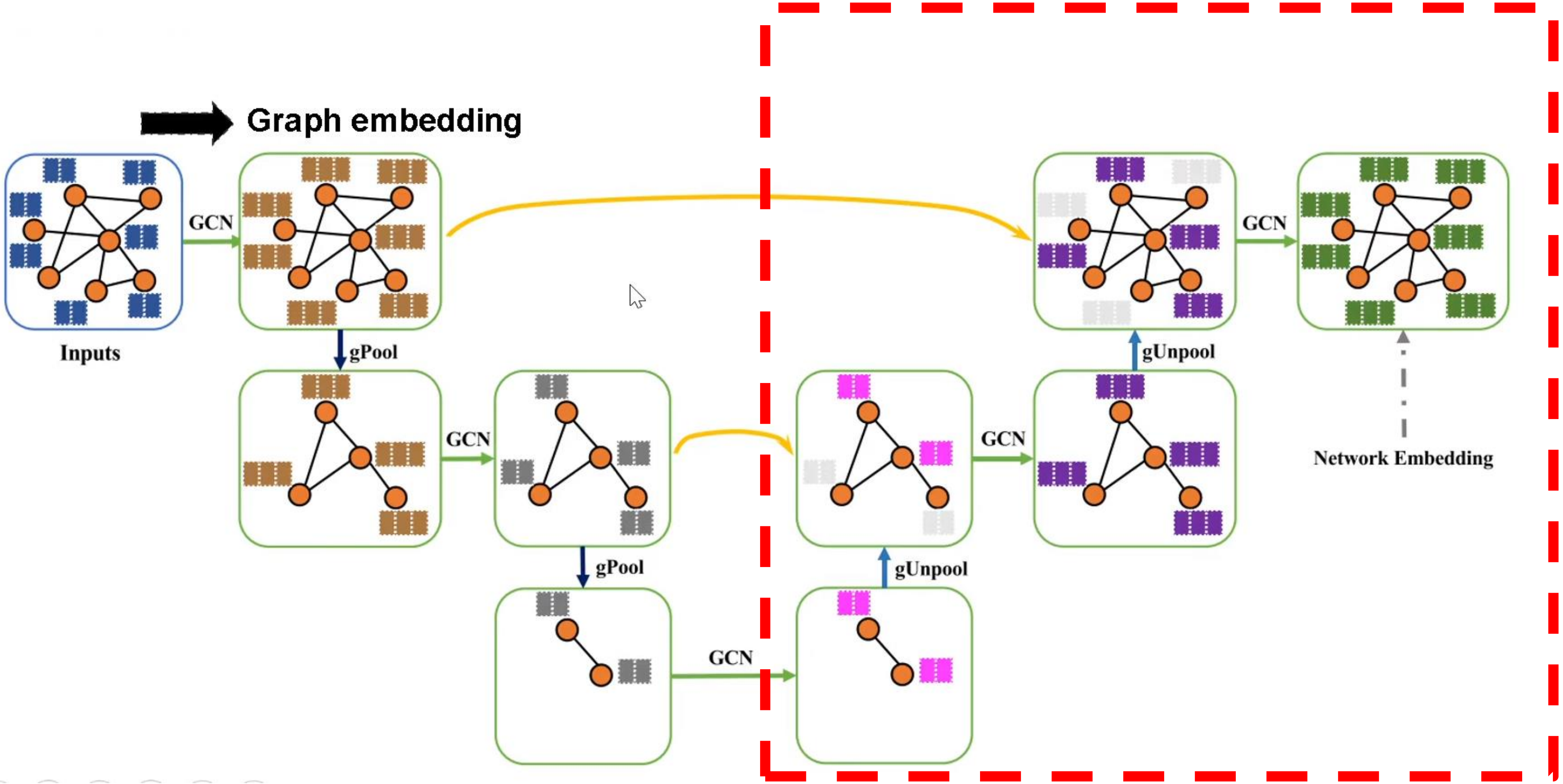

Encoder
block

# Decoder

Restore graph into higher resolution

gUnpool -> GCN....-> $\boxed{\text{gUnpool -> GCN}}$ → **Aggregate first order neighbor information**

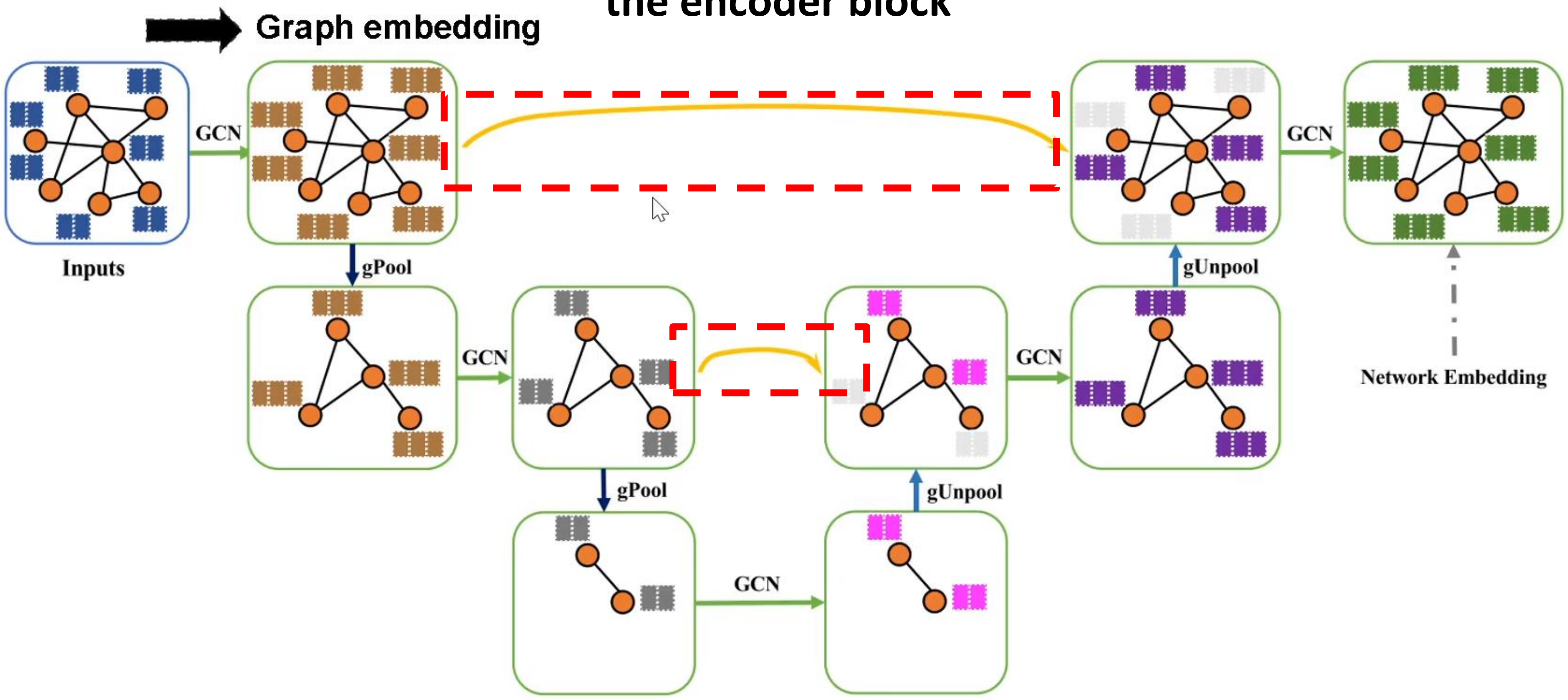
$$X_{\ell+1} = \sigma(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}X_\ell W_\ell)$$




**Decoding block**

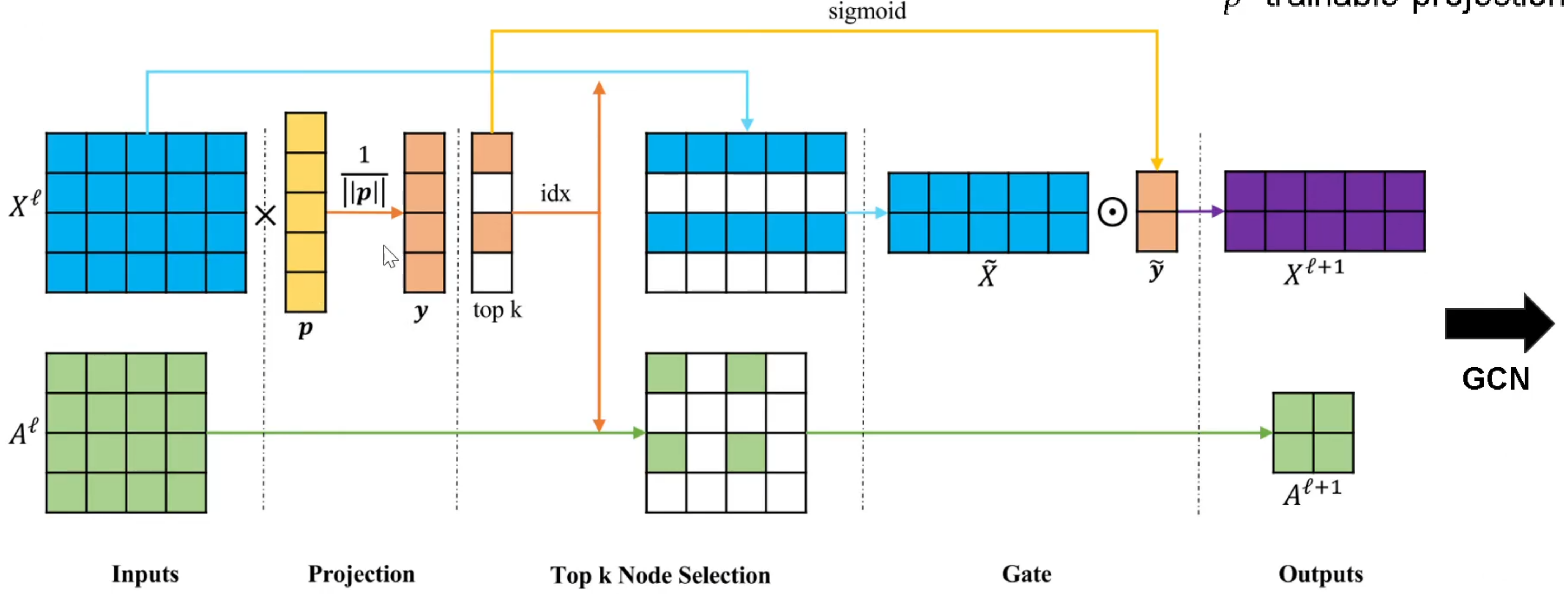
# Skip Connection

**For blocks in the same level, encoder block uses skip connection to fuse the low-level features from the encoder block**



# Graph Pooling

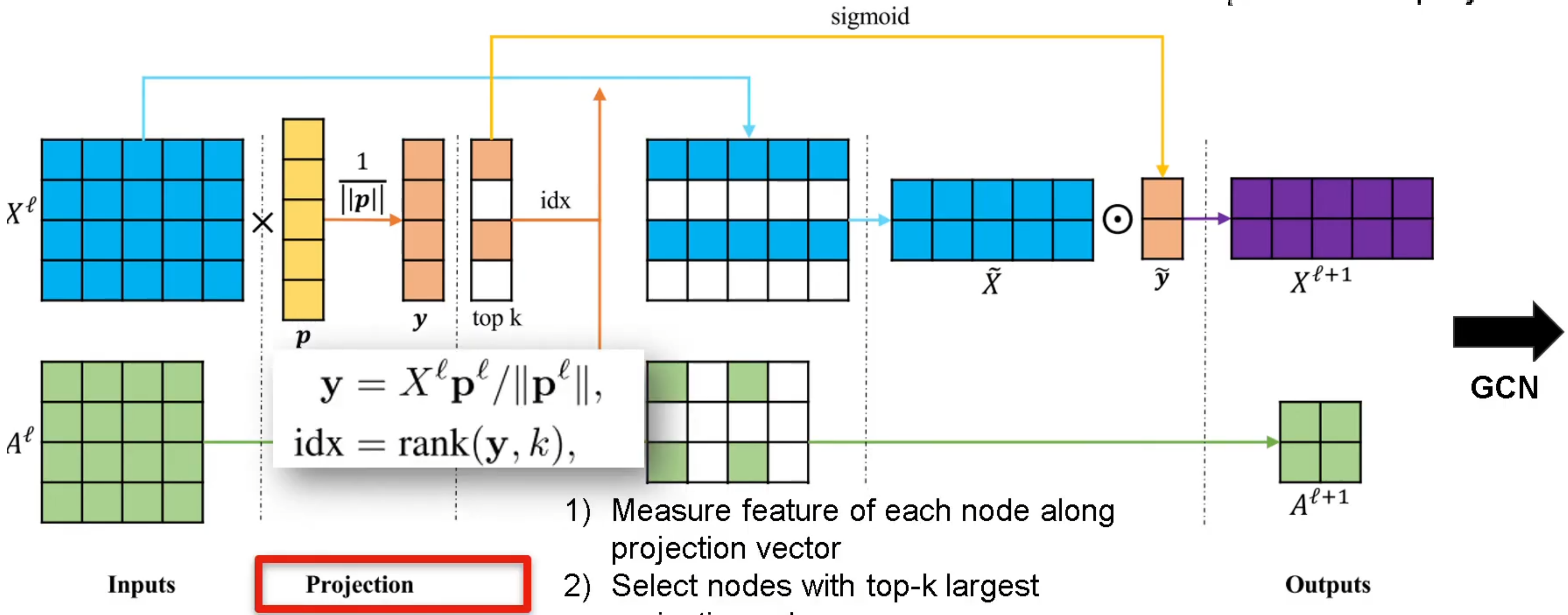
$X$ – feature matrix
$A$ – Adjacency matrix
$p$- trainable projection vector



# Graph Pooling

1. Projection

$X$ – feature matrix
$A$ – Adjacency matrix
$p$- trainable projection vector

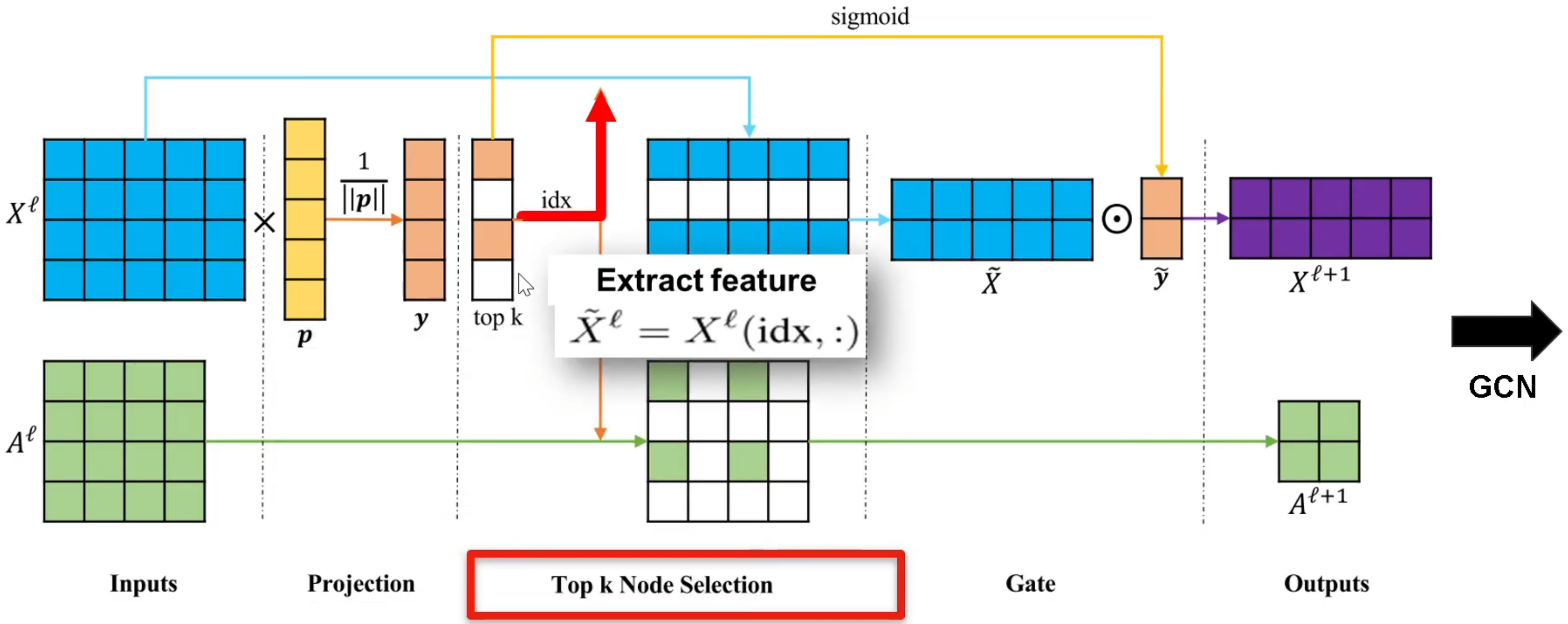



1) Measure feature of each node along projection vector
2) Select nodes with top-k largest projection value

# Graph Pooling

2. Top k Node Selection( Extract feature )

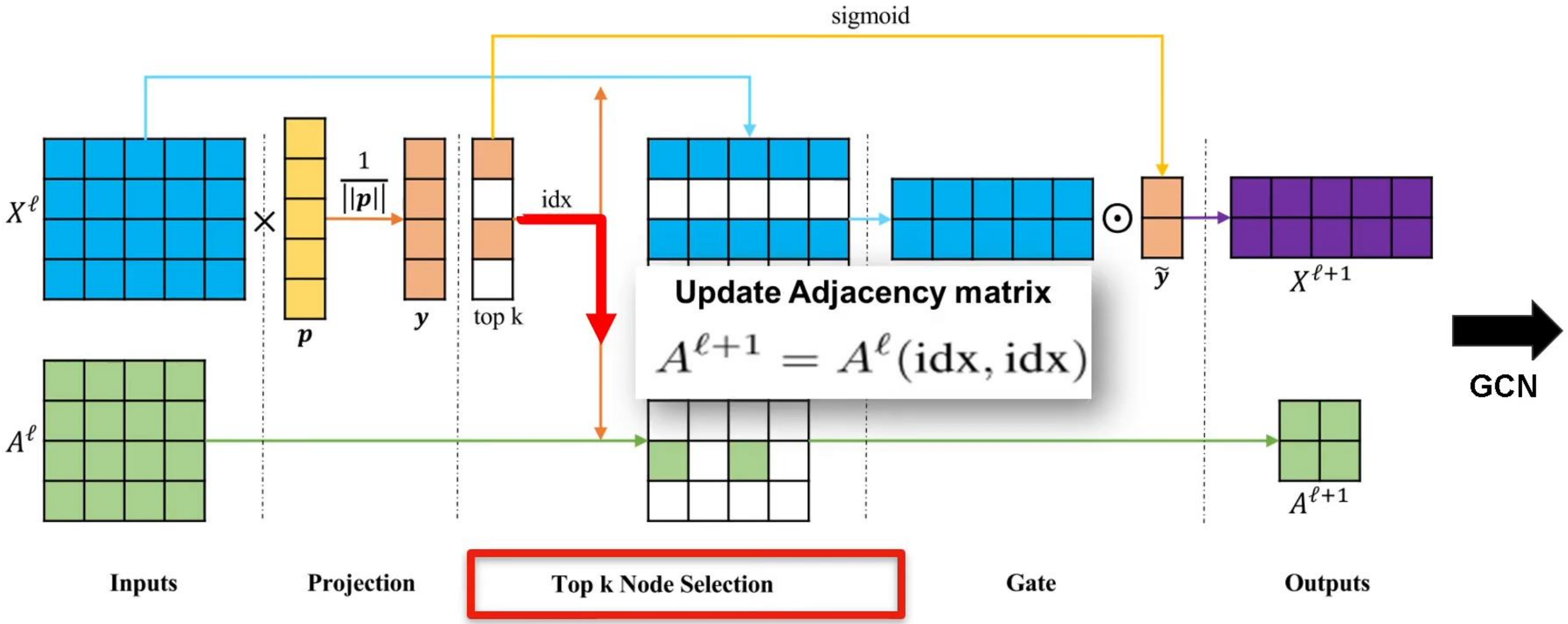
$X$ – feature matrix
$A$ – Adjacency matrix
$p$- trainable projection vector



# Graph Pooling

$X$ – feature matrix
$A$ – Adjacency matrix
$p$- trainable projection vector

3. Top k Node Selection( Update Adjacency matrix)

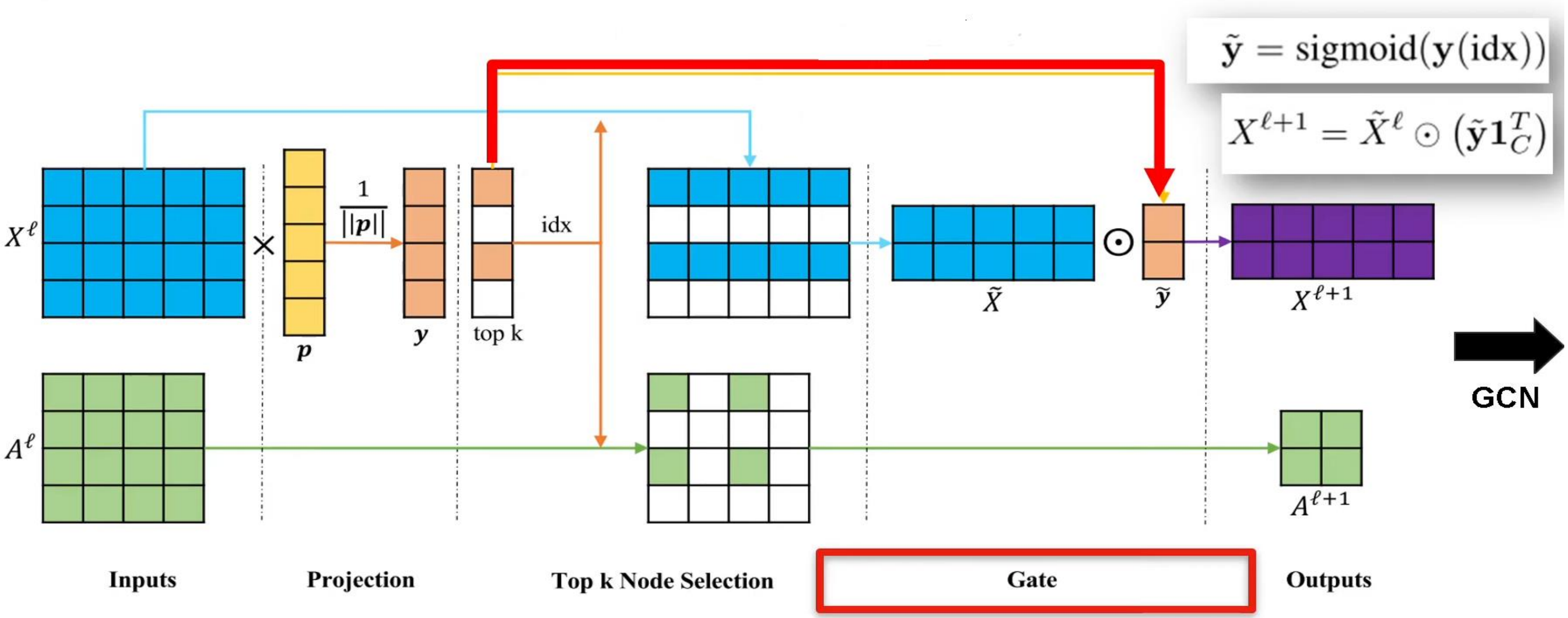


# Graph Pooling

4. Gate

Control information flow

$X$ – feature matrix
$A$ – Adjacency matrix
$p$- trainable projection vector

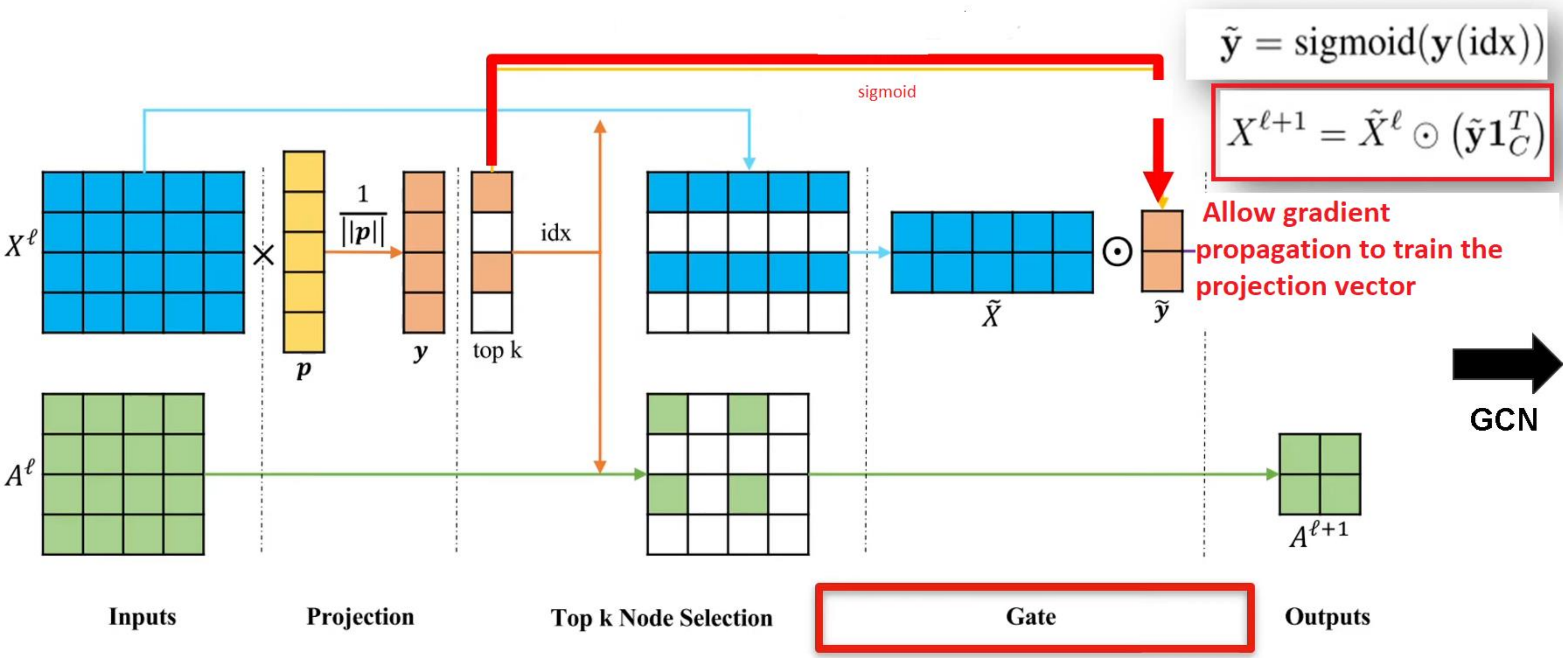


# Graph Pooling

$X$ – feature matrix
$A$ – Adjacency matrix
$p$- trainable projection vector

4. Gate(sigmoid)

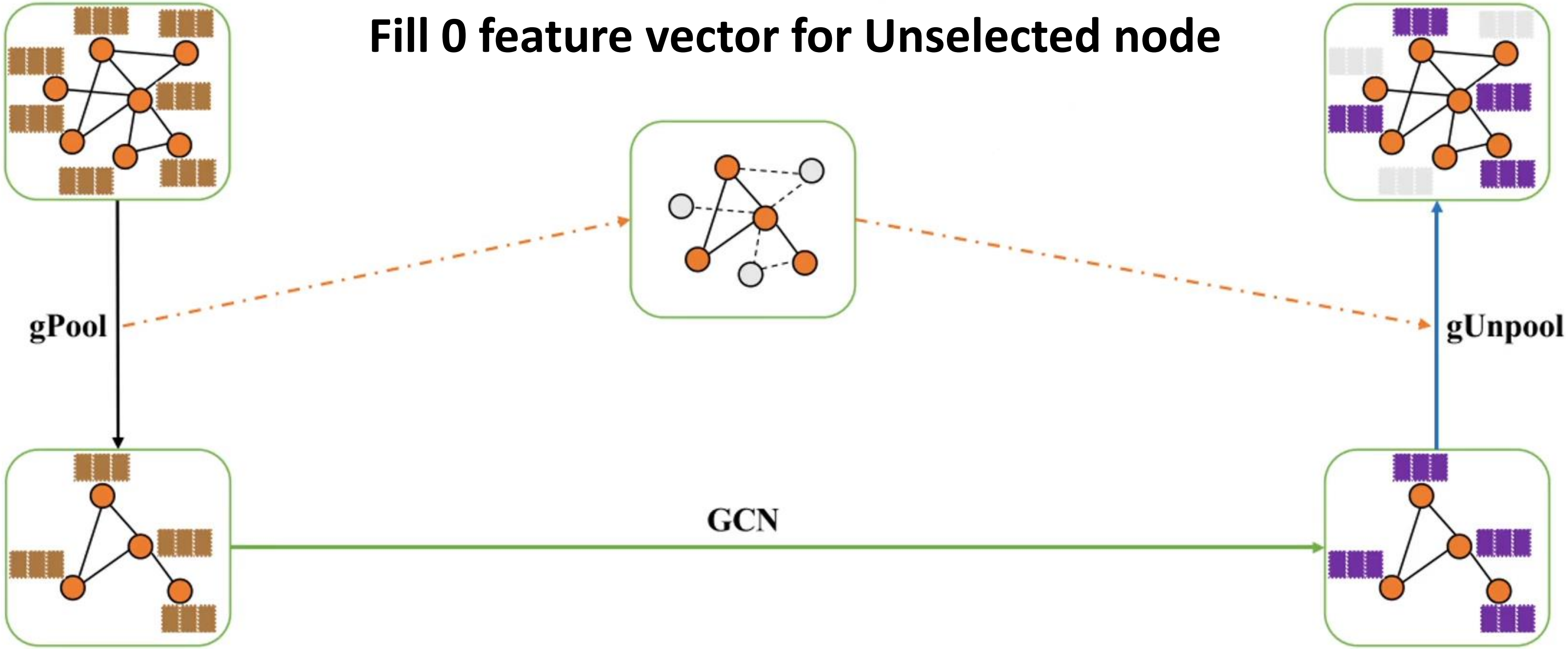
## Control information flow



# Graph Unpooling

$$X^{\ell+1} = \text{distribute}(0_{N \times C}, X^{\ell}, \text{idx})$$

**Fill 0 feature vector for Unselected node**



# Tricks

- Use 2nd-order adjacency matrix to avoid too sparse connectivity after gPooling

$$A^2 = A^{\ell} A^{\ell}, \qquad A^{\ell+1} = A^2(\mathrm{idx}, \mathrm{idx})$$

- Emphasize each node's own feature

$$\hat{A} = \hat{A} + 2I$$

# Experiments (described in the paper)

## 1. Datasets

- **Node classification**

*Table 1.* Summary of datasets used in our node classification experiments (Yang et al., 2016; Zitnik & Leskovec, 2017). The Cora, Citeseer, and Pubmed datasets are used for transductive learning experiments.

| Dataset | Nodes | Features | Classes | Training | Validation | Testing | Degree |
|---|---|---|---|---|---|---|---|
| Cora | 2708 | 1433 | 7 | 140 | 500 | 1000 | 4 |
| Citeseer | 3327 | 3703 | 6 | 120 | 500 | 1000 | 5 |
| Pubmed | 19717 | 500 | 3 | 60 | 500 | 1000 | 6 |

- **Inductive learning experiment (Labels of parts of nodes are unknow)**

*Table 2.* Summary of datasets used in our inductive learning experiments. The D&D (Dobson & Doig, 2003), PROTEINS (Borgwardt et al., 2005), and COLLAB (Yanardag & Vishwanathan, 2015) datasets are used for inductive learning experiments.

| Dataset | Graphs | Nodes (max) | Nodes (avg) | Classes |
|---|---|---|---|---|
| D&D | 1178 | 5748 | 284.32 | 2 |
| PROTEINS | 1113 | 620 | 39.06 | 2 |
| COLLAB | 5000 | 492 | 74.49 | 3 |

# Experiments (described in the paper)

## 2. Performance

*Table 3.* Results of transductive learning experiments in terms of node classification accuracies on Cora, Citeseer, and Pubmed datasets. g-U-Nets denotes our proposed graph U-Nets model.

| Models | Cora | Citeseer | Pubmed |
|---|---|---|---|
| DeepWalk (Perozzi et al., 2014) | 67.2% | 43.2% | 65.3% |
| Planetoid (Yang et al., 2016) | 75.7% | 64.7% | 77.2% |
| Chebyshev (Defferrard et al., 2016) | 81.2% | 69.8% | 74.4% |
| GCN (Kipf & Welling, 2017) | 81.5% | 70.3% | 79.0% |
| GAT (Veličković et al., 2017) | $83.0 \pm 0.7\%$ | $72.5 \pm 0.7\%$ | $79.0 \pm 0.3\%$ |
| **g-U-Nets (Ours)** | $\mathbf{84.4 \pm 0.6\%}$ | $\mathbf{73.2 \pm 0.5\%}$ | $\mathbf{79.6 \pm 0.2\%}$ |

*Table 4.* Results of inductive learning experiments in terms of graph classification accuracies on D&D, PROTEINS, and COLLAB datasets. g-U-Nets denotes our proposed graph U-Nets model.

| Models | D&D | PROTEINS | COLLAB |
|---|---|---|---|
| PSCN (Niepert et al., 2016) | 76.27% | 75.00% | 72.60% |
| DGCNN (Zhang et al., 2018) | 79.37% | 76.26% | 73.76% |
| DiffPool-DET (Ying et al., 2018) | 75.47% | 75.62% | **82.13%** |
| DiffPool-NOLP (Ying et al., 2018) | 79.98% | 76.22% | 75.58% |
| DiffPool (Ying et al., 2018) | 80.64% | 76.25% | 75.48% |
| **g-U-Nets (Ours)** | **82.43%** | **77.68%** | 77.56% |

# Experiments (described in the paper)

## 3. Network structure study

- **Network depth (Works for shallow network and consistent with U-net)**

*Table 7.* Comparison of different network depths in terms of node classification accuracy on Cora, Citeseer, and Pubmed datasets. Based on g-U-Nets, we experiment with different network depths in terms of the number of blocks in encoder and decoder parts.

| Depth | Cora | Citeseer | Pubmed |
|---|---|---|---|
| 2 | $82.6 \pm 0.6\%$ | $71.8 \pm 0.5\%$ | $79.1 \pm 0.3\%$ |
| 3 | $83.8 \pm 0.7\%$ | $72.7 \pm 0.7\%$ | $79.4 \pm 0.4\%$ |
| 4 | $\mathbf{84.4 \pm 0.6\%}$ | $\mathbf{73.2 \pm 0.5\%}$ | $\mathbf{79.6 \pm 0.2\%}$ |
| 5 | $84.1 \pm 0.5\%$ | $72.8 \pm 0.6\%$ | $79.5 \pm 0.3\%$ |

- **Parameter number (Add small parameters for large improvement)**

*Table 8.* Comparison of the g-U-Nets with and without gPool or gUnpool layers in terms of the node classification accuracy and the number of parameters on Cora dataset.

| Models | Accuracy | #Params | Ratio of increase |
|---|---|---|---|
| g-U-Nets without gPool or gUnpool | $82.1 \pm 0.6\%$ | 75,643 | 0.00% |
| **g-U-Nets (Ours)** | $\mathbf{84.4 \pm 0.6\%}$ | 75,737 | 0.12% |

# Graph U-Net in PyTorch Geometric

```
dataset = Planetoid(root='tmp/Cora', name='Cora')
data = dataset[0]
```

```
device = 'cpu'
model, data = Net().to(device), data.to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.01, weight_decay=0.001)
```

Data Loading

```
class Net(torch.nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        pool_ratios = [2000 / data.num_nodes, 0.5]
        self.unet = GraphUNet(dataset.num_features, 32, dataset.num_classes,
                              depth=3, pool_ratios=pool_ratios)

    def forward(self):
        edge_index, _ = dropout_adj(data.edge_index, p=0.2,
                                    force_undirected=True,
                                    num_nodes=data.num_nodes,
                                    training=self.training)
        x = F.dropout(data.x, p=0.92, training=self.training)

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

# Graph U-Net in PyTorch Geometric

```
for epoch in range(1, 201):
    model.train()
    optimizer.zero_grad()
    loss = F.nll_loss(model()[data.train_mask], data.y[data.train_mask])
    loss.backward()
    optimizer.step()

    model.eval()
    logits, accs = model(), []
    for _, mask in data('train_mask', 'val_mask', 'test_mask'):
        pred = logits[mask].max(1)[1]
        acc = pred.eq(data.y[mask]).sum().item() / mask.sum().item()
        accs.append(acc)
    acc_arr.append(acc)

    train_acc, val_acc, tmp_test_acc = accs
    if val_acc > best_val_acc:
        best_val_acc = val_acc
        test_acc = tmp_test_acc


    epoch_arr.append(epoch)
    loss_arr.append(loss.item())
    log = 'Epoch: {:03d}, Loss :{:.4f}, Accuracy :{:.4f}, Train: {:.4f}, Val: {:.4f}, Test: {:.4f}'
    print(log.format(epoch, train_acc, best_val_acc, test_acc,loss.item(),acc))
```

Training

Testing

# Experiment result

- Data: Cora
- Epoch: 200

```
Epoch: 001, Loss :0.5000, Accuracy :0.4280, Train: 0.4290, Val: 1.9623, Test: 0.4290
Epoch: 002, Loss :0.6071, Accuracy :0.5200, Train: 0.5530, Val: 1.8873, Test: 0.5530
Epoch: 003, Loss :0.6786, Accuracy :0.5640, Train: 0.5970, Val: 1.8642, Test: 0.5970
Epoch: 004, Loss :0.7500, Accuracy :0.6120, Train: 0.6270, Val: 1.7876, Test: 0.6270
Epoch: 005, Loss :0.7929, Accuracy :0.6400, Train: 0.6540, Val: 1.6981, Test: 0.6540
Epoch: 006, Loss :0.8429, Accuracy :0.6580, Train: 0.6930, Val: 1.6356, Test: 0.6930
Epoch: 007, Loss :0.8857, Accuracy :0.6900, Train: 0.7160, Val: 1.5279, Test: 0.7160
Epoch: 008, Loss :0.9071, Accuracy :0.7200, Train: 0.7350, Val: 1.4898, Test: 0.7350
Epoch: 009, Loss :0.9286, Accuracy :0.7520, Train: 0.7650, Val: 1.4498, Test: 0.7650
Epoch: 010, Loss :0.9500, Accuracy :0.7560, Train: 0.7750, Val: 1.4351, Test: 0.7750
Epoch: 011, Loss :0.9643, Accuracy :0.7700, Train: 0.7750, Val: 1.2631, Test: 0.7750
Epoch: 012, Loss :0.9643, Accuracy :0.7740, Train: 0.7770, Val: 1.1646, Test: 0.7770
Epoch: 013, Loss :0.9571, Accuracy :0.7820, Train: 0.7810, Val: 1.1423, Test: 0.7810
Epoch: 014, Loss :0.9643, Accuracy :0.7820, Train: 0.7810, Val: 1.0445, Test: 0.7760
Epoch: 015, Loss :0.9643, Accuracy :0.7840, Train: 0.7850, Val: 1.1282, Test: 0.7850
Epoch: 016, Loss :0.9571, Accuracy :0.7840, Train: 0.7850, Val: 0.8989, Test: 0.7840
Epoch: 017, Loss :0.9643, Accuracy :0.7840, Train: 0.7850, Val: 0.8642, Test: 0.7750
Epoch: 018, Loss :0.9714, Accuracy :0.7840, Train: 0.7850, Val: 0.8773, Test: 0.7720
Epoch: 019, Loss :0.9643, Accuracy :0.7840, Train: 0.7850, Val: 0.8510, Test: 0.7720
Epoch: 020, Loss :0.9786, Accuracy :0.7840, Train: 0.7850, Val: 0.8236, Test: 0.7660
```

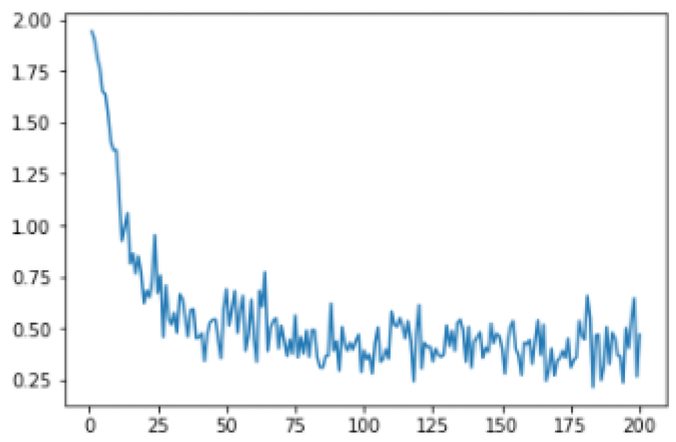
```
Epoch: 182, Loss :1.0000, Accuracy :0.7960, Train: 0.7950, Val: 0.3310, Test: 0.7870
Epoch: 183, Loss :1.0000, Accuracy :0.7960, Train: 0.7950, Val: 0.3709, Test: 0.7840
Epoch: 184, Loss :1.0000, Accuracy :0.7960, Train: 0.7950, Val: 0.4090, Test: 0.7890
Epoch: 185, Loss :1.0000, Accuracy :0.7960, Train: 0.7950, Val: 0.2991, Test: 0.7880
Epoch: 186, Loss :1.0000, Accuracy :0.7960, Train: 0.7950, Val: 0.3706, Test: 0.7910
Epoch: 187, Loss :1.0000, Accuracy :0.7960, Train: 0.7950, Val: 0.5192, Test: 0.7850
Epoch: 188, Loss :1.0000, Accuracy :0.7960, Train: 0.7950, Val: 0.4711, Test: 0.7860
Epoch: 189, Loss :1.0000, Accuracy :0.7960, Train: 0.7950, Val: 0.4144, Test: 0.7890
Epoch: 190, Loss :1.0000, Accuracy :0.7960, Train: 0.7950, Val: 0.2995, Test: 0.7890
Epoch: 191, Loss :1.0000, Accuracy :0.7960, Train: 0.7950, Val: 0.4400, Test: 0.7890
Epoch: 192, Loss :1.0000, Accuracy :0.7960, Train: 0.7950, Val: 0.3971, Test: 0.7850
Epoch: 193, Loss :1.0000, Accuracy :0.7960, Train: 0.7950, Val: 0.3801, Test: 0.7730
Epoch: 194, Loss :1.0000, Accuracy :0.7960, Train: 0.7950, Val: 0.4296, Test: 0.7700

Epoch: 195, Loss :1.0000, Accuracy :0.7960, Train: 0.7950, Val: 0.4701, Test: 0.7700
Epoch: 196, Loss :1.0000, Accuracy :0.7960, Train: 0.7950, Val: 0.3588, Test: 0.7710
Epoch: 197, Loss :1.0000, Accuracy :0.7960, Train: 0.7950, Val: 0.5546, Test: 0.7690
Epoch: 198, Loss :1.0000, Accuracy :0.7960, Train: 0.7950, Val: 0.3649, Test: 0.7620
Epoch: 199, Loss :1.0000, Accuracy :0.7960, Train: 0.7950, Val: 0.4322, Test: 0.7680
Epoch: 200, Loss :1.0000, Accuracy :0.7960, Train: 0.7950, Val: 0.4251, Test: 0.7710
```

# Experiment result

## Loss graph

```
plt.savefig("loss_viz.png", dpi=300)
plt.clf()
# Epoch-Accuracy 시각화
plt.plot(epoch_arr, loss_arr)
```

```
[<matplotlib.lines.Line2D at 0x190ea240970>]
```




## Accuracy graph

```
plt.savefig("acc_viz.png", dpi=300)
plt.clf()
plt.plot(epoch_arr, acc_arr, 'r')
```

```
[<matplotlib.lines.Line2D at 0x190fcd8fc70>]
```