50.039 Theory and Practice of Deep Learning

W9-S3 Graph Convolutional Networks

Matthieu De Mari

About this week (Week 9)

1. What are **graph objects**?
2. How do we **define** a graph object **mathematically**?
3. What are **typical graph problems**?
4. How can we **embed a graph object** to later feed it to a Neural Network?
5. What is a **graph convolution** and how des it relate to the concept of image convolution?
6. What are more **advanced problems** and **approaches** on graph convolutional Neural Networks?

Outline

In this lecture
(Continued from previous lecture)

- Graph Convolutional Neural Networks
- Feature engineering in Graph Convolutional Neural Networks

In this lecture, also

- Graph Convolutional Neural Networks with Attention Mechanisms
- Some more advanced graph embeddings

Graph convolutional layer (Kipf)

- **Definition (graph convolutional layer, with conventional spectral propagation rule):**

$$\boldsymbol{H} = \boldsymbol{\sigma}\left(\widehat{\boldsymbol{N}}\boldsymbol{X}\boldsymbol{W}\right) = \boldsymbol{\sigma}\left(\widehat{\boldsymbol{D}}^{-\mathbf{1}}\hat{\mathrm{A}}\boldsymbol{X}\boldsymbol{W}\right)$$

With $\boldsymbol{\sigma}$ an **activation function** (so far we used identity), and $\boldsymbol{W}$ a **weight matrix**, which we will train later on.

- **Definition (graph convolutional layer, with Kipf spectral propagation rule *[Kipf]*):**

$$\boldsymbol{H} = \boldsymbol{\sigma}\left(\widehat{\boldsymbol{N}}^{\boldsymbol{kipf}}\boldsymbol{X}\boldsymbol{W}\right) = \boldsymbol{\sigma}\left(\widehat{\boldsymbol{D}}^{-\frac{\mathbf{1}}{\mathbf{2}}}\hat{\mathrm{A}}\widehat{\boldsymbol{D}}^{-\frac{\mathbf{1}}{\mathbf{2}}}\boldsymbol{X}\boldsymbol{W}\right)$$

- Slightly better than $\widehat{\boldsymbol{D}}^{-\mathbf{1}}\hat{\mathrm{A}}$, especially in oriented graphs.
- $\boldsymbol{M} = \widehat{\boldsymbol{D}}^{-\frac{\mathbf{1}}{\mathbf{2}}}\hat{\mathrm{A}}\widehat{\boldsymbol{D}}^{-\frac{\mathbf{1}}{\mathbf{2}}}$ is called the symmetric Laplacian of the graph.

[Kipf] Kipf, T. N., & Welling, M. (2016). Semi-supervised classification with graph convolutional networks.

A quick word on Laplacians (out of class)

Definitions (some Laplacians): Given a graph G, with adjacency matrix A and degree matrix D, we can define the following matrices:

- The Laplacian matrix:

$$L = D - A$$

- The random walk Laplacian

$$L_{rw} = I - D^{-1}A = D^{-1}L$$

- The symmetric Laplacian

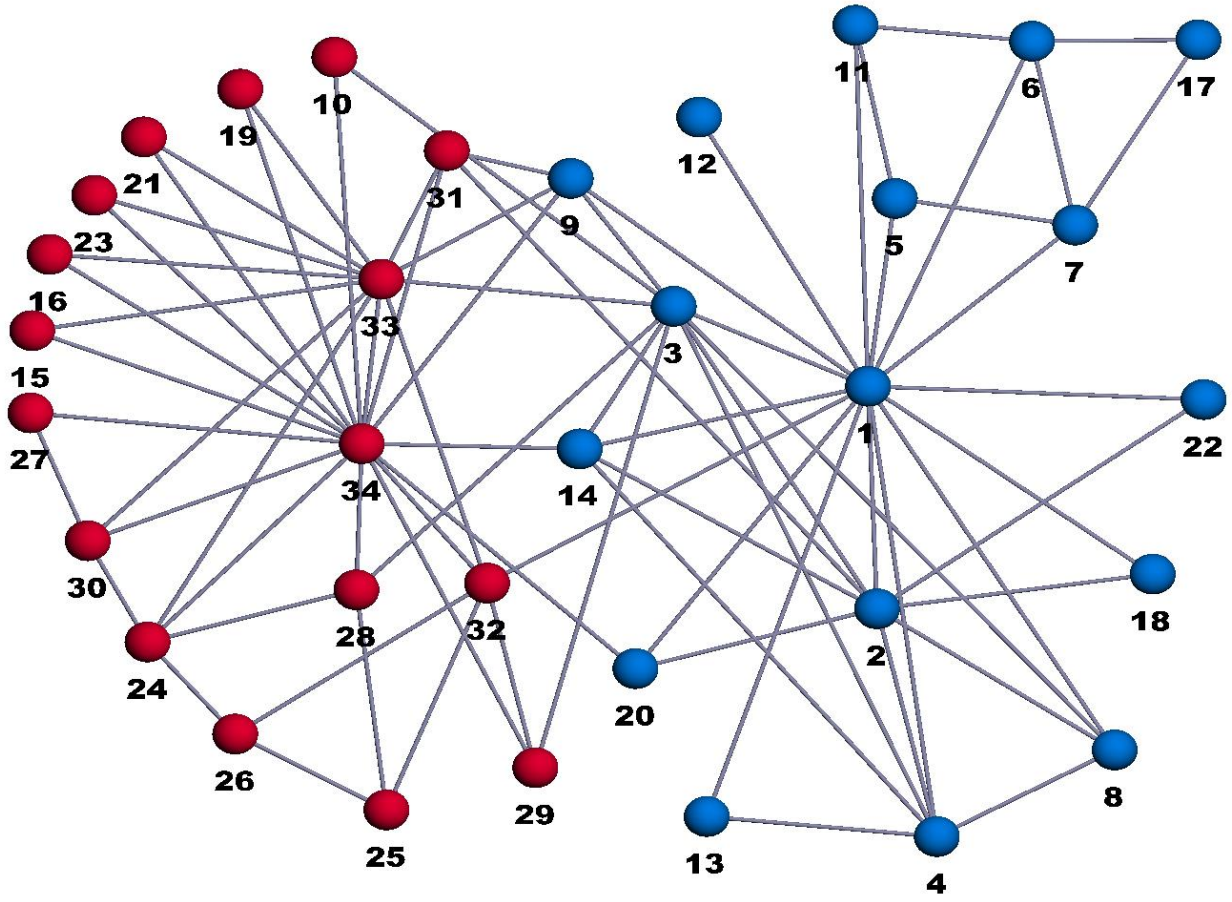
$$L_{sym} = I - D^{-\frac{1}{2}}AD^{-\frac{1}{2}} = D^{-\frac{1}{2}}LD^{-\frac{1}{2}}$$

These matrices have interesting math properties in graph theory, but we will leave those to the curious reader.

A more sophisticated toy example

Zachary Karate Fight club *[ZKC]*:

- 34 members, as nodes, indexed from 1 to 34.
- Mr. Hi, the owner is node 1.
- The Officer/Trainer is node 34.
- All other nodes are members.
- Edges define friendships between individuals.
- **Question:** If Mr. Hi and the officer/trainer decide to fight, what will be the teams?

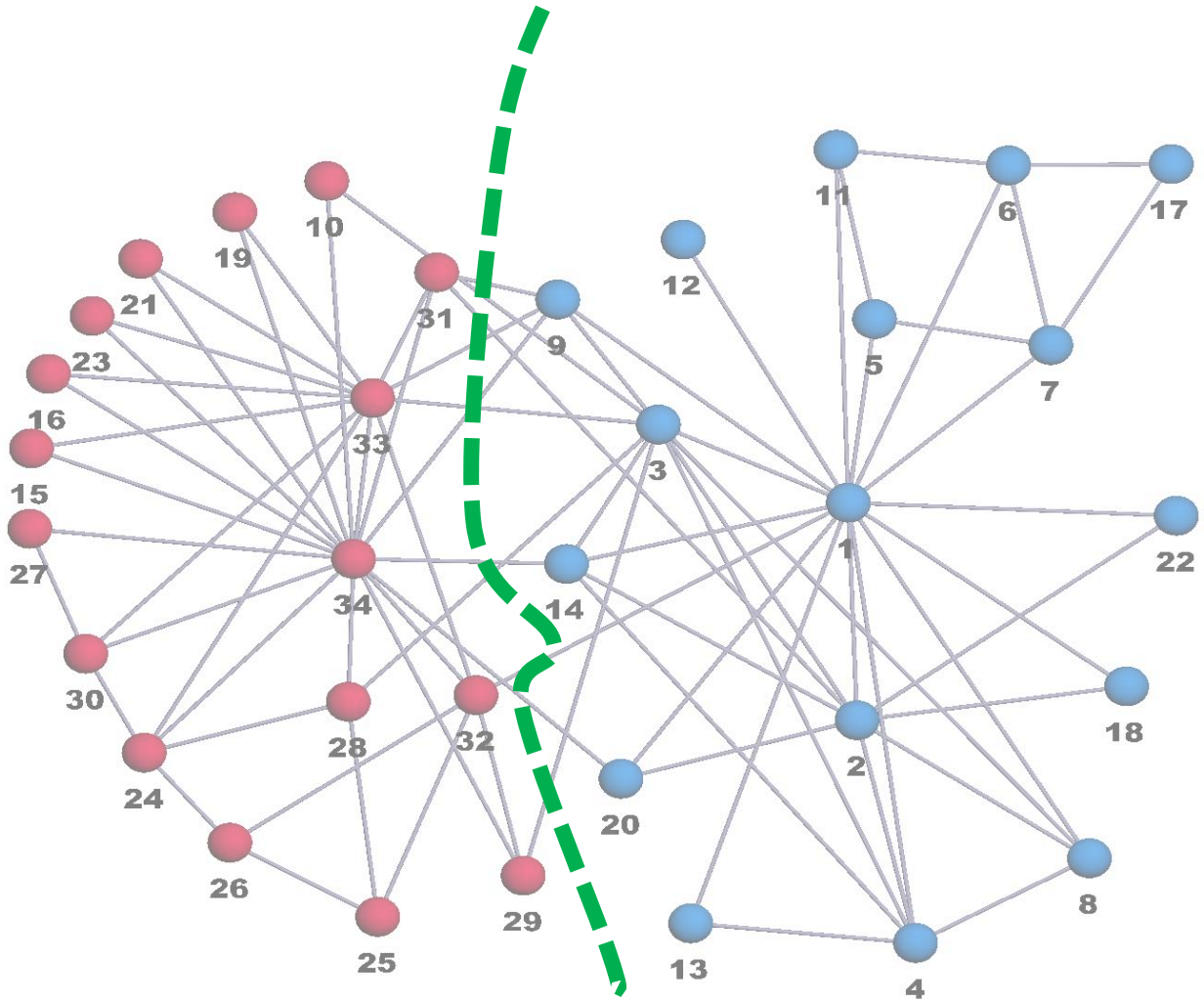


[ZKC] Zachary, W. W. (1977). An information flow model for conflict and fission in small groups.

A more sophisticated toy example

Zachary Karate Fight club *[ZKC]*:

- It consists of a binary classification problem.
- Separate the nodes (as with any classification problem)

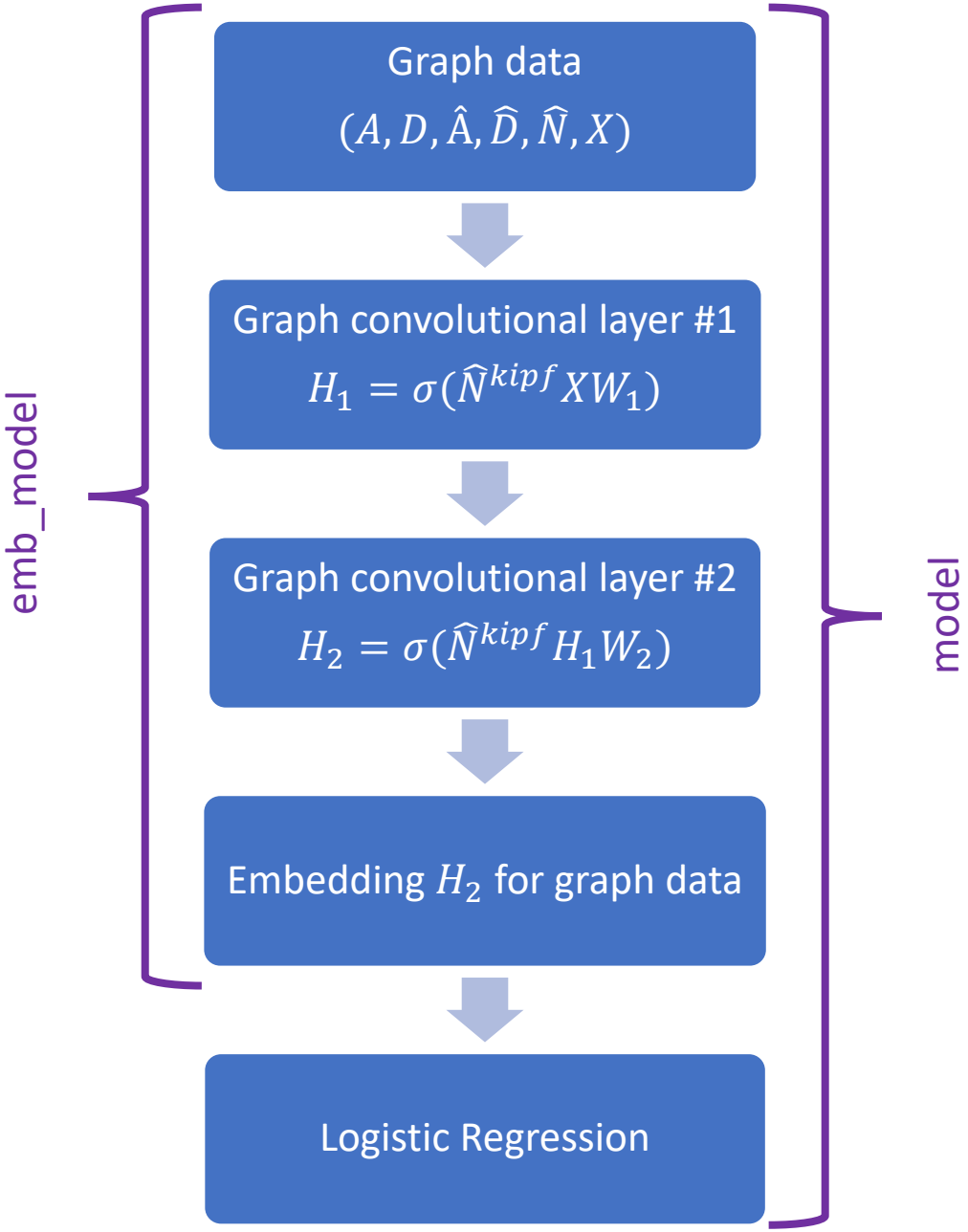


[ZKC] Zachary, W. W. (1977). An information flow model for conflict and fission in small groups.

Building blocks for classifier

- Classification task on graph dataset (ZKC).

- **Structure:**
  - 2 GCN layers (with Kipf propagation rule)
  - 1 Logistic Regression for classification



Building a basic GCN layer and model

- Create our own custom basic GCN layer block, using the PyTorch nn.Module class as superclass.

- Pass the adjacency matrix $\boldsymbol{A}$, compute $\hat{\mathrm{A}}$, $\widehat{\boldsymbol{D}}$, etc.
- Propagation rule:

$$\boldsymbol{H} = \boldsymbol{\sigma}\left(\widehat{\boldsymbol{N}}\boldsymbol{X}\boldsymbol{W}\right) = \boldsymbol{\sigma}\left(\widehat{\boldsymbol{D}}^{-\mathbf{1}}\hat{\mathrm{A}}\boldsymbol{X}\boldsymbol{W}\right)$$

```python
class GCNConv_Layer(nn.Module):
    """
    Standard GCN convolution layer class
    """

    def __init__(self, adj, input_channels, output_channels):
        super().__init__()
        self.A_hat = adj + torch.eye(adj.size(0))
        self.D = torch.diag(torch.sum(adj, 1))
        self.D = self.D.inverse()
        self.A_hat = torch.mm(self.D, self.A_hat)
        self.W = nn.Parameter(torch.rand(input_channels, output_channels))

    def forward(self, X):
        out = torch.relu(torch.mm(torch.mm(self.A_hat, X), self.W))
        return out
```

```python
class Net1(torch.nn.Module):
    """
    Standard GCN model class
    """

    def __init__(self, adj, num_feat, num_hid, num_out):
        super().__init__()
        self.conv1 = GCNConv_Layer(adj, num_feat, num_hid)
        self.conv2 = GCNConv_Layer(adj, num_hid, num_out)

    def forward(self, X):
        X  = self.conv1(X)
        X = self.conv2(X)
        return X
```

Building a Kipf GCN layer and model

- Create our own custom Kipf GCN layer block, using the PyTorch nn.Module class as superclass.

- Pass the adjacency matrix $\boldsymbol{A}$, compute $\hat{\mathrm{A}}$, $\widehat{\boldsymbol{D}}$, etc.
- Propagation rule:

$$\boldsymbol{H} = \boldsymbol{\sigma}\left(\widehat{\boldsymbol{N}}^{\boldsymbol{kipf}}\boldsymbol{X}\boldsymbol{W}\right)$$
$$= \boldsymbol{\sigma}\left(\widehat{\boldsymbol{D}}^{-\frac{\boldsymbol{1}}{\boldsymbol{2}}}\hat{\mathrm{A}}\widehat{\boldsymbol{D}}^{-\frac{\boldsymbol{1}}{\boldsymbol{2}}}\boldsymbol{X}\boldsymbol{W}\right)$$

```python
class GCNKipf_Layer(nn.Module):
    """
    Kipf GCN convolution layer class
    """

    def __init__(self, adj, input_channels, output_channels):
        super().__init__()
        self.A_hat = adj + torch.eye(adj.size(0))
        self.D = torch.diag(torch.sum(adj, 1))
        self.D = self.D.inverse().sqrt()
        self.A_hat = torch.mm(torch.mm(self.D, self.A_hat), self.D)
        self.W = nn.Parameter(torch.rand(input_channels, output_channels))

    def forward(self, X):
        out = torch.relu(torch.mm(torch.mm(self.A_hat, X), self.W))
        return out
```

```python
class Net2(torch.nn.Module):
    """
    Standard GCN model class
    """

    def __init__(self, adj, num_feat, num_hid, num_out):
        super().__init__()
        self.conv1 = GCNKipf_Layer(adj, num_feat, num_hid)
        self.conv2 = GCNKipf_Layer(adj, num_hid, num_out)

    def forward(self, X):
        X  = self.conv1(X)
        X = self.conv2(X)
        return X
```

Trainer definition

Define a training function for our GCN + Logistic Regression model.

- **Loss:** Binary Cross Entropy
- **Optimizer:** SGD, with learning rate 0.01 and momentum 1.

- **Note:** no node features for now, making $X = I$, to make these nodes irrelevant in the forward propagation rule!

```
# No relevant features for nodes
# Using X = identity will make nodes features irrelevant
# and the model will have to learn from adjacency matrix only
X = torch.eye(A.size(0))
```

```
model2 = Net1(A, X.size(0), 10, 2)
criterion = torch.nn.CrossEntropyLoss(ignore_index = -1)
optimizer = optim.SGD(model2.parameters(), lr = 0.01, momentum = 0.9)
loss = criterion(model2(X), ground_truth)
```

```
history2 = []
for i in range(500):
    # Forward pass
    optimizer.zero_grad()
    loss = criterion(model2(X), current)

    # Backprop
    loss.backward()
    optimizer.step()

    # For display later
    l = (model2(X))

    if i%10 == 0:
        history2.append(loss.item())
        print("Cross Entropy Loss (iter = {}): =".format(i), loss.item())
```

```
Cross Entropy Loss (iter = 0): = 0.7158387899398804
Cross Entropy Loss (iter = 10): = 0.6941231489181519
Cross Entropy Loss (iter = 20): = 0.6553307771682739
Cross Entropy Loss (iter = 30): = 0.6119217872619629
Cross Entropy Loss (iter = 40): = 0.5661046504974365
Cross Entropy Loss (iter = 50): = 0.5170614719390869
Cross Entropy Loss (iter = 60): = 0.4643201231956482
Cross Entropy Loss (iter = 70): = 0.40893375873565674
Cross Entropy Loss (iter = 80): = 0.3527054190635681
Cross Entropy Loss (iter = 90): = 0.29891103506088257
Cross Entropy Loss (iter = 100): = 0.2498018741607666
Cross Entropy Loss (iter = 110): = 0.20729894936084747
Cross Entropy Loss (iter = 120): = 0.17220862209796906
Cross Entropy Loss (iter = 130): = 0.1436990201473236
```

Train model (no features)

- Overall, the model is learning but it takes time to do so properly.

- The model attempts to learn from the adjacency matrix and degree matrix only…
- It however has no useful nodes features it could rely on.
- Kipf formula seems to make the training slightly faster.

```python
# No relevant features for nodes
# Using X = identity will make nodes features irrelevant
# and the model will have to learn from adjacency matrix only
X = torch.eye(A.size(0))
```

```python
model2 = Net1(A, X.size(0), 10, 2)
criterion = torch.nn.CrossEntropyLoss(ignore_index = -1)
optimizer = optim.SGD(model2.parameters(), lr = 0.01, momentum = 0.9)
loss = criterion(model2(X), ground_truth)
```

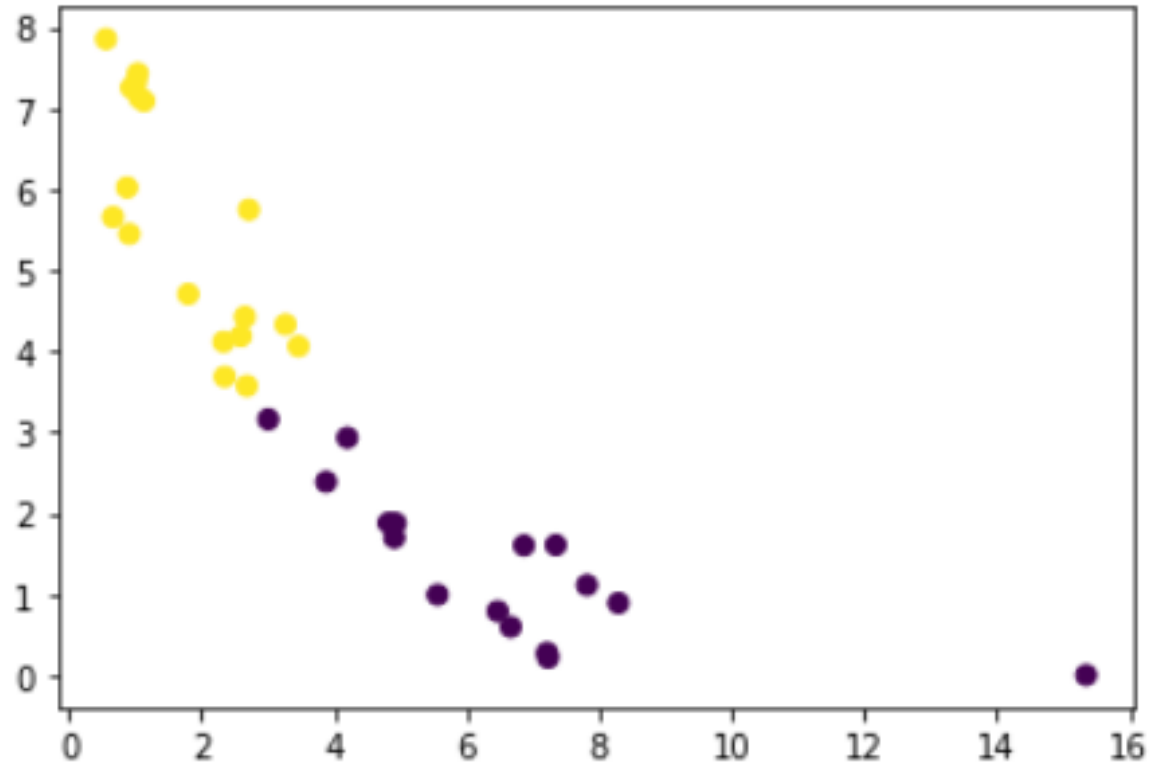
```python
history2 = []
for i in range(500):
    # Forward pass
    optimizer.zero_grad()
    loss = criterion(model2(X), current)

    # Backprop
    loss.backward()
    optimizer.step()

    # For display later
    l = (model2(X))

    if i%10 == 0:
        history2.append(loss.item())
        print("Cross Entropy Loss (iter = {}): =".format(i), loss.item())
```

```
Cross Entropy Loss (iter = 0): = 0.7158387899398804
Cross Entropy Loss (iter = 10): = 0.6941231489181519
Cross Entropy Loss (iter = 20): = 0.6553307771682739
Cross Entropy Loss (iter = 30): = 0.6119217872619629
Cross Entropy Loss (iter = 40): = 0.5661046504974365
Cross Entropy Loss (iter = 50): = 0.5170614719390869
Cross Entropy Loss (iter = 60): = 0.4643201231956482
Cross Entropy Loss (iter = 70): = 0.40893375873565674
Cross Entropy Loss (iter = 80): = 0.3527054190635681
Cross Entropy Loss (iter = 90): = 0.29891103506088257
Cross Entropy Loss (iter = 100): = 0.2498018741607666
Cross Entropy Loss (iter = 110): = 0.20729894936084747
Cross Entropy Loss (iter = 120): = 0.17220862209796906
Cross Entropy Loss (iter = 130): = 0.1436990201473236
```

Train model (no features)

- Overall, the model is learning but it takes time to do so properly.

- The model attempts to learn from the adjacency matrix and degree matrix only…
- It however has no useful nodes features it could rely on.
- Kipf formula seems to make the training slightly faster.

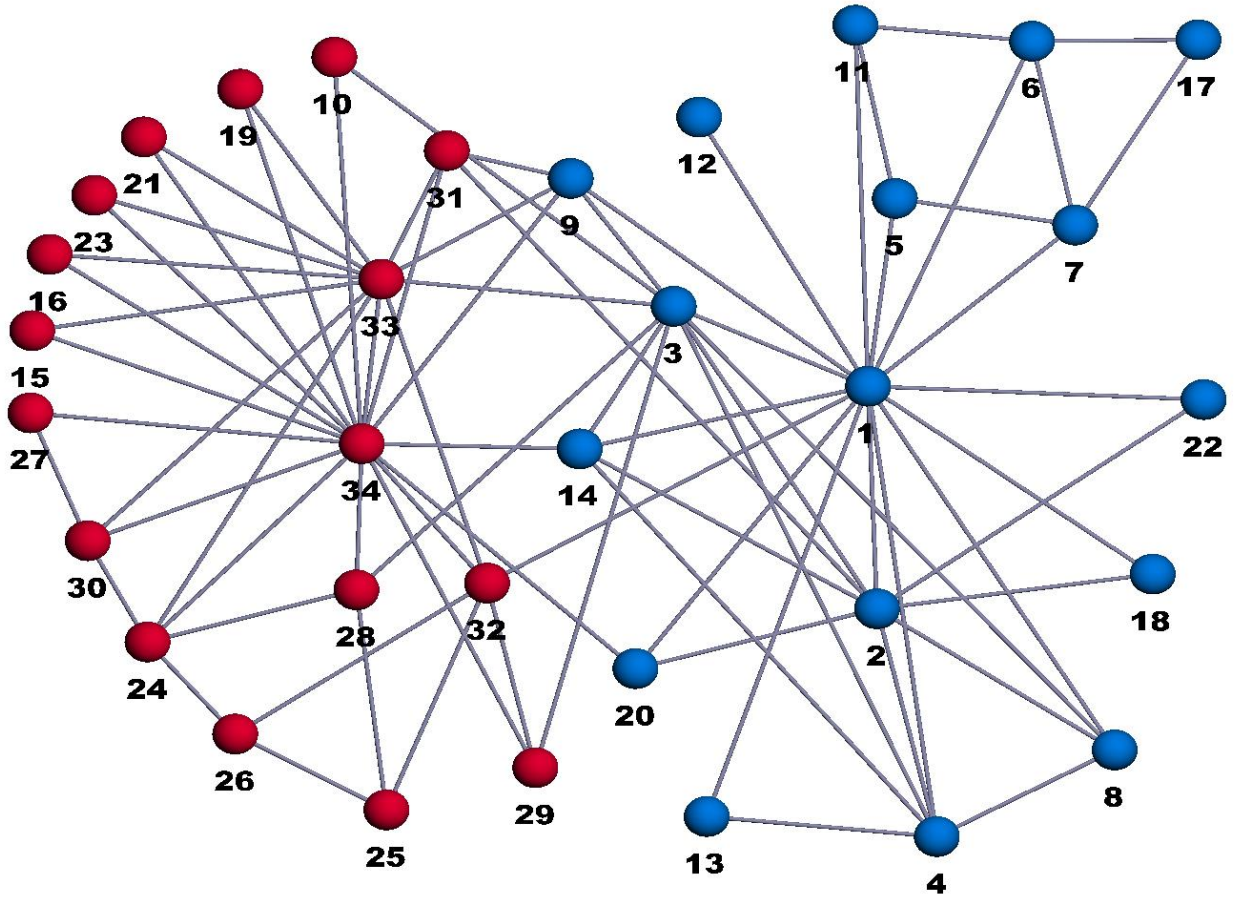


Build model: adding nodes features

- The nodes have no features.

→ Needs feature engineering

- Some interesting features for the nodes would be
  - the hop-distance to Mr. Hi (i.e. node 0),
  - and the hop-distance to the officer/trainer (i.e. node 33).
  - Use a simple BFS to calculate them!



Build model: adding nodes features

- The nodes have no features.

→ Needs feature engineering

- Some interesting features for the nodes would be
  - the hop-distance to Mr. Hi (i.e. node 0),
  - and the hop-distance to the officer/trainer (i.e. node 33).
  - Use a simple BFS to calculate them!

```python
def bfs(adj, start, goal):
    """
    Gives hop-distance between node start and node goal
    for given adjacency matrix.
    Returns zero if start = goal
    or goal not reachable from start.
    """

    if start == goal:
        return float(0)
    queue = [start]
    visited = []
    dist = float(0)

    while(len(queue) > 0):
        dist += 1
        temp = []
        for q in queue:
            neighbours_node = np.argwhere(adj[q]).reshape(1, -1)[0]
            if goal in neighbours_node:
                return dist
            else:
                for n in neighbours_node:
                    not_visited = (n not in visited)
                    not_queue = (n not in queue)
                    not_temp = (n not in temp)
                    if not_visited and not_queue and not_temp:
                        temp.append(n)
        visited.extend(queue)
        queue = temp
    return float(-1)
```

```python
y = bfs(adj, 1, 33)
print(y)
```

Build model: adding nodes features

- The nodes have no features.

→ Needs feature engineering

- Some interesting features for the nodes would be
  - the hop-distance to Mr. Hi (i.e. node 0),
  - and the hop-distance to the officer/trainer (i.e. node 33).
  - Use a simple BFS to calculate them!

```
# Adding relevant features (hop distance to nodes admin and instructor)
node_features = np.array([[bfs(adj, i, 0), bfs(adj, i, 33)] for i in range(34)])
H2 = torch.from_numpy(node_features).float()
print(H2)
```

```
tensor([[0., 2.],
        [1., 2.],
        [1., 2.],
        [1., 2.],
        [1., 3.],
        [1., 3.],
        [1., 3.],
        [1., 3.],
        [1., 1.],
        [2., 1.],
        [1., 3.],
        [1., 3.],
        [1., 3.],
        [1., 3.],
        [1., 1.],
        [3., 1.],
        [3., 1.],
        [2., 4.],
        [1., 3.],
        [3., 1.],
        [1., 1.],
        [3., 1.],
        [1., 3.],
        [3., 1.],
        [3., 1.],
        [2., 2.],
        [2., 2.],
        [3., 1.],
        [2., 1.],
        [2., 1.],
        [3., 1.],
        [2., 1.],
        [1., 1.],
        [2., 1.],
        [2., 0.]])
```

Trainer definition

Define a training function for our GCN + Logistic Regression model.

- **Loss:** Binary Cross Entropy
- **Optimizer:** SGD, with learning rate 0.01 and momentum 1.

- **Note:** let us now try with these relevant node features H_2!

```
history3 = []
for i in range(500):
    # Forward pass
    optimizer.zero_grad()
    pred = model3(H2)
    # Remove first and last nodes predictions
    trimmed_pred = pred[1:-1]
    loss = criterion(trimmed_pred, revised_ground_truth)

    # Backprop
    loss.backward()
    optimizer.step()

    # For display later
    l = (model3(H2))

    if i%10 == 0:
        history3.append(loss.item())
        print("Cross Entropy Loss (iter = {}): =".format(i), loss.item())
```

```
Cross Entropy Loss (iter = 0): = 3.753018617630005
Cross Entropy Loss (iter = 10): = 0.7272272109985352
Cross Entropy Loss (iter = 20): = 0.1322544813156128
Cross Entropy Loss (iter = 30): = 0.09342840313911438
Cross Entropy Loss (iter = 40): = 0.08162444829940796
Cross Entropy Loss (iter = 50): = 0.0526009276509285
Cross Entropy Loss (iter = 60): = 0.047182802110910416
Cross Entropy Loss (iter = 70): = 0.04264695197343826
Cross Entropy Loss (iter = 80): = 0.03863685205578804
Cross Entropy Loss (iter = 90): = 0.03590169921517372
Cross Entropy Loss (iter = 100): = 0.033363137394189835
Cross Entropy Loss (iter = 110): = 0.03127694129943848
Cross Entropy Loss (iter = 120): = 0.0294414721429348
Cross Entropy Loss (iter = 130): = 0.027817633002996445
```

Train model (with some features)

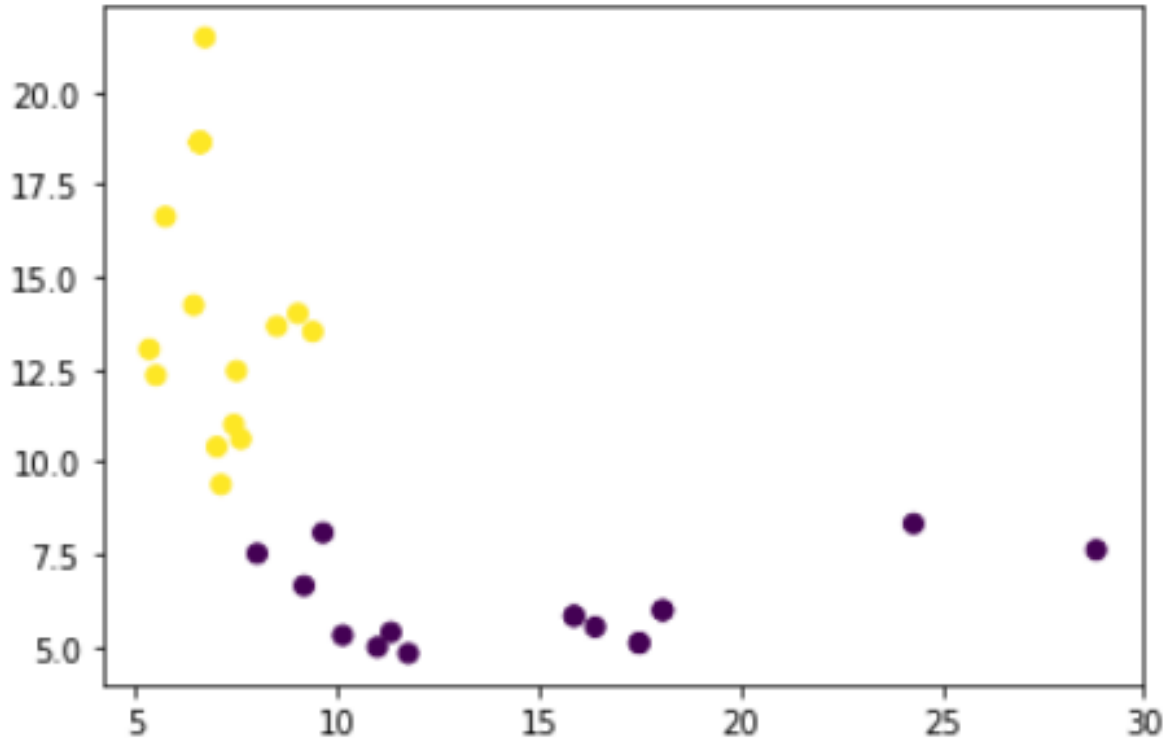
- Overall, the model is now able to train properly.
- The model attempts to learn from the adjacency matrix and degree matrix only.
- The hop-distances carry an information on the proximity of members to Mr. Hi and the Officer/Trainer.

```
history3 = []
for i in range(500):
    # Forward pass
    optimizer.zero_grad()
    pred = model3(H2)
    # Remove first and last nodes predictions
    trimmed_pred = pred[1:-1]
    loss = criterion(trimmed_pred, revised_ground_truth)

    # Backprop
    loss.backward()
    optimizer.step()

    # For display later
    l = (model3(H2))

    if i%10 == 0:
        history3.append(loss.item())
        print("Cross Entropy Loss (iter = {}): =".format(i), loss.item())
```

```
Cross Entropy Loss (iter = 0): = 3.753018617630005
Cross Entropy Loss (iter = 10): = 0.7272272109985352
Cross Entropy Loss (iter = 20): = 0.1322544813156128
Cross Entropy Loss (iter = 30): = 0.09342840313911438
Cross Entropy Loss (iter = 40): = 0.08162444829940796
Cross Entropy Loss (iter = 50): = 0.0526009276509285
Cross Entropy Loss (iter = 60): = 0.047182802110910416
Cross Entropy Loss (iter = 70): = 0.04264695197343826
Cross Entropy Loss (iter = 80): = 0.03863685205578804
Cross Entropy Loss (iter = 90): = 0.03590169921517372
Cross Entropy Loss (iter = 100): = 0.033363137394189835
Cross Entropy Loss (iter = 110): = 0.03127694129943848
Cross Entropy Loss (iter = 120): = 0.0294414721429348
Cross Entropy Loss (iter = 130): = 0.027817633002996445
```

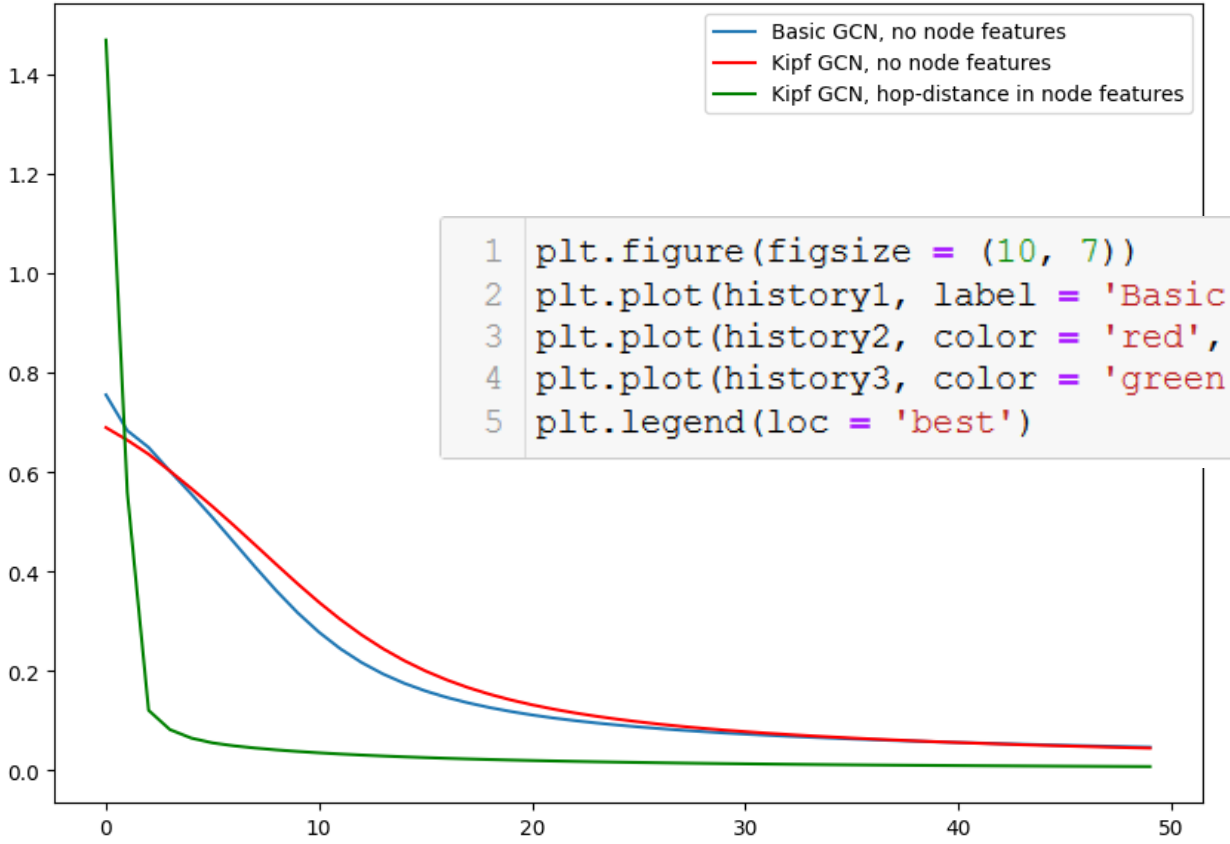
Train model (with some features)

- Overall, the model is now able to train properly.
- The model attempts to learn from the adjacency matrix and degree matrix only.
- The hop-distances carry an information on the proximity of members to Mr. Hi and the Officer/Trainer.



Train model (with some features)

Important message: Additional nodes features make the training much more efficient! Features engineering is key with graphs!




```
plt.figure(figsize = (10, 7))
plt.plot(history1, label = 'Basic GCN, no node features')
plt.plot(history2, color = 'red', label = 'Kipf GCN, no node features')
plt.plot(history3, color = 'green', label = 'Kipf GCN, hop-distance in node features')
plt.legend(loc = 'best')
```

Additional features?

We could think of additional features to help the Graph Convolutional Neural Network learn. For instance:

- Additional node features: for instance, number of close friends to Mr. Hi and Officer/Trainer among the neighbors of the node (close friends = nodes with hop-distance of 1).
- Edge features: for instance, add a random “Friendship” value to the edges, that quantifies friendship between two nodes.
  Open question: Later on, how would you modify your propagation formulas to take into account the edge weights?

Challenge: Leaving the implementation of these extra features as challenge for you to practice!

Limits of the previous approaches

Let us discuss some of the key issues we have observed so far, and some issues we can intuitively guess will be problematic in these vanilla GCNs.

Limits of the previous approaches

Issue #1: Sparse graphs and computational cost of spectral embeddings:

- These approaches rely on adjacency and degree matrix for a given graph.
- On large graphs, it is heavy, computationally speaking.
- Inefficient also, as most elements in the adjacency matrix are zeros.

```
# Adjacency matrix for Zachary graph
adj = np.array([[0,1,1,1,1,1,1,1,1,0,1,1,1,1,0,0,0,1,0,1,0,1,0,0,0,0,0,0,0,0,0,1,0,0],
                [1,0,1,1,0,0,0,1,0,0,0,0,0,1,0,0,0,1,0,1,0,1,0,0,0,0,0,0,0,0,1,0,0,0],
                [1,1,0,1,0,0,0,1,1,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,1,0],
                [1,1,1,0,0,0,0,1,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                [1,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                [1,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                [1,0,0,0,1,1,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                [1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                [1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,1],
                [0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],
                [1,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                [1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                [1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],
                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1],
                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1],
                [0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                [1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1],
                [1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],
                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1],
                [1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1],
                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,1,0,0,1,1],
                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,0,0,1,0,0],
                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,1,0,0],
                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1],
                [0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,1],
                [0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1],
                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,1,0,0,0,0,1,1],
                [0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1],
                [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,1,0,0,0,1,1],
                [0,0,1,0,0,0,0,0,1,0,0,0,0,0,1,1,0,0,1,0,1,0,1,1,0,0,0,0,1,1,1,0,1],
                [0,0,0,0,0,0,0,0,1,1,0,0,0,1,1,1,0,0,1,1,1,0,1,1,0,0,1,1,1,1,1,1,0]])
A = torch.Tensor(adj)
```

Limits of the previous approaches

Issue #1: Sparse graphs and computational cost of spectral embeddings:

- These approaches rely on adjacency and degree matrix for a given graph.
- On large graphs, it is heavy, computationally speaking.
- Inefficient also, as most elements in the adjacency matrix are zeros.

In [5]:
```
nodes_parameters = define_nodes_parameters(G)
print(nodes_parameters)
```

```
{'nodes_number': 34, 'nodes_names': [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
25, 26, 27, 28, 29, 30, 31, 32, 33], 'nodes_features': [], 'labels_list': ['Mr. Hi', 'Officer'], 'node
0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}
```

In [6]:
```
edges_parameters = define_edges_parameters(G, nodes_number = nodes_parameters['nodes_number'])
print(edges_parameters)
```

```
{'edges_number': 78, 'edges_names': [(0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7), (0, 8), (
(0, 13), (0, 17), (0, 19), (0, 21), (0, 31), (1, 2), (1, 3), (1, 7), (1, 13), (1, 17), (1, 19), (1, 2
7), (2, 8), (2, 9), (2, 13), (2, 27), (2, 28), (2, 32), (3, 7), (3, 12), (3, 13), (4, 6), (4, 10), (5,
(6, 16), (8, 30), (8, 32), (8, 33), (9, 33), (13, 33), (14, 32), (14, 33), (15, 32), (15, 33), (18, 3
0, 32), (20, 33), (22, 32), (22, 33), (23, 25), (23, 27), (23, 29), (23, 32), (23, 33), (24, 25), (24,
(26, 29), (26, 33), (27, 33), (28, 31), (28, 33), (29, 32), (29, 33), (30, 32), (30, 33), (31, 32), (3
ency_matrix': array([[0., 1., 1., ..., 1., 0., 0.],
       [1., 0., 1., ..., 0., 0., 0.],
       [1., 1., 0., ..., 0., 1., 0.],
       ...,
       [1., 0., 0., ..., 0., 1., 1.],
       [0., 0., 1., ..., 1., 0., 1.],
       [0., 0., 0., ..., 1., 1., 0.]]), 'degree_matrix': array([[16.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  9.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0., 10., ...,  0.,  0.,  0.],
       ...,
       [ 0.,  0.,  0., ...,  6.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0., 12.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0., 17.]])}
```

In [10]:
```
edges_number = 2*len(edges_parameters['edges_names'])
max_number_edges = (nodes_parameters['nodes_number']**2)
sparstity_ratio = edges_number/max_number_edges
print(sparstity_ratio)
```

```
0.13494809688581316
```

Limits of the previous approaches

Issue #1: Sparse graphs and computational cost of spectral embeddings:

- These approaches rely on adjacency and degree matrix for a given graph.
- On large graphs, it is heavy, computationally speaking.
- Inefficient also, as most elements in the adjacency matrix are zeros.

```
In [5]:
1 nodes_parameters = define_nodes_parameters(G)
2 print(nodes_parameters)

{'nodes_number': 34, 'nodes_names': [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
25, 26, 27, 28, 29, 30, 31, 32, 33], 'nodes_features': [], 'labels_list': ['Mr. Hi', 'Officer'], 'node
0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}
```

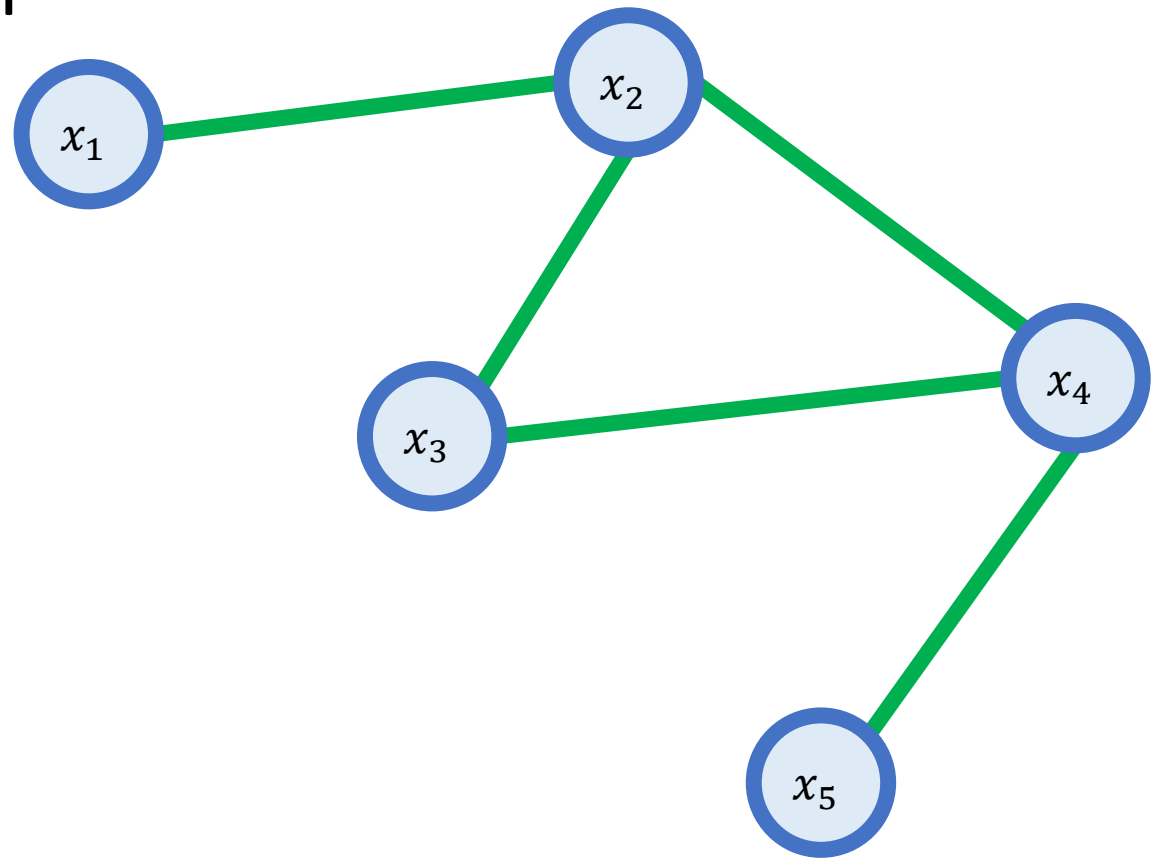
```
In [6]:
1 edges_parameters = define_edges_parameters(G, nodes_number = nodes_parameters['nodes_number'])
2 print(edges_parameters)

{'edges_number': 78, 'edges_names': [(0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7), (0, 8),
(0, 13), (0, 17), (0, 19), (0, 21), (0, 31), (1, 2), (1, 3), (1, 7), (1, 13), (1, 17), (1, 19), (1, 2
7), (2, 8), (2, 9), (2, 13), (2, 27), (2, 28), (2, 32), (3, 7), (3, 12), (3, 13), (4, 6), (4, 10), (5,
(6, 16), (8, 30), (8, 32), (8, 33), (9, 33), (13, 33), (14, 32), (14, 33), (15, 32), (15, 33), (18, 3
0, 32), (20, 33), (22, 32), (22, 33), (23, 25), (23, 27), (23, 29), (23, 32), (23, 33), (24, 25), (24,
(26, 29), (26, 33), (27, 33), (28, 31), (28, 33), (29, 32), (29, 33), (30, 32), (30, 33), (31, 32), (
ency_matrix': array([[0., 1., 1., ..., 1., 0., 0.],
       [1., 0., 1., ..., 0., 0., 0.],
       [1., 1., 0., ..., 0., 1., 0.],
       ...,
       [1., 0., 0., ..., 0., 1., 1.],
       [0., 0., 1., ..., 1., 0., 1.],
       [0., 0., 0., ..., 1., 1., 0.]]), 'degree_matrix': array([[16.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  9.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0., 10., ...,  0.,  0.,  0.],
```

→ Need for scalable embeddings, operating locally for each node, instead of the “general” graph matrices.

Limits of the previous approaches

Issue #2: Spectral embeddings work on fixed graphs structures

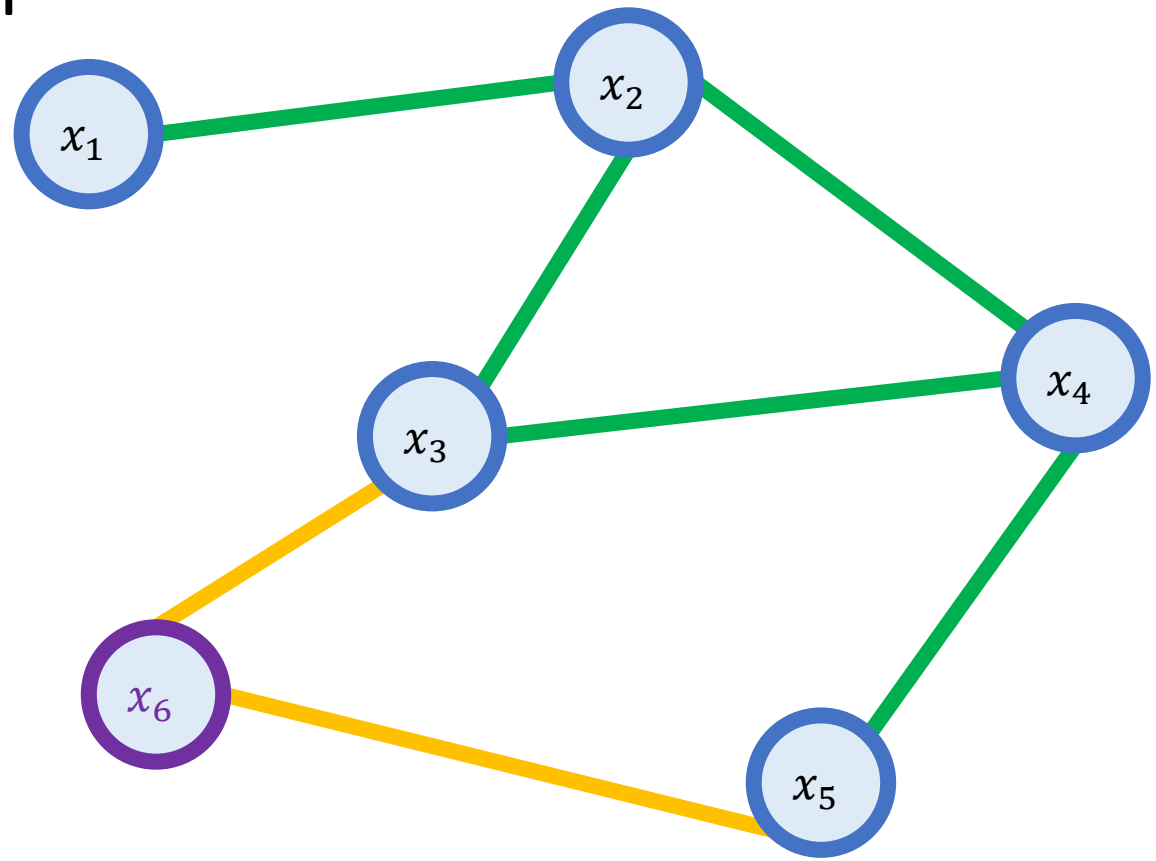
- Our spectral embeddings work for a single given graph.



Limits of the previous approaches

Issue #2: Spectral embeddings work on fixed graphs structures

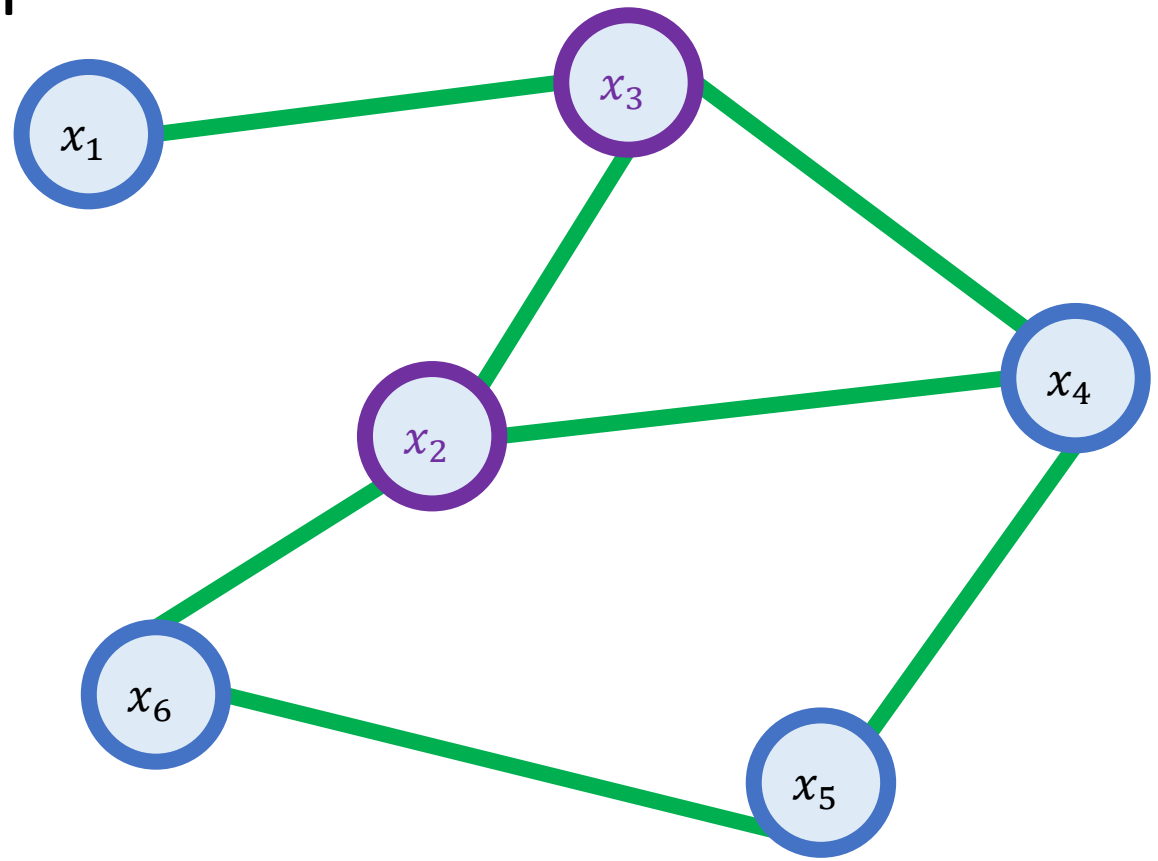
- Our spectral embeddings work for a single given graph.
- Adding a new node in the graph requires to reshape the whole embedding architecture.



Limits of the previous approaches

Issue #2: Spectral embeddings work on fixed graphs structures

- Our spectral embeddings work for a single given graph.
- Adding a new node in the graph requires to reshape the whole embedding architecture.
- Also fails if nodes names are swapped, even though it does not really change the graph structure, so to speak…

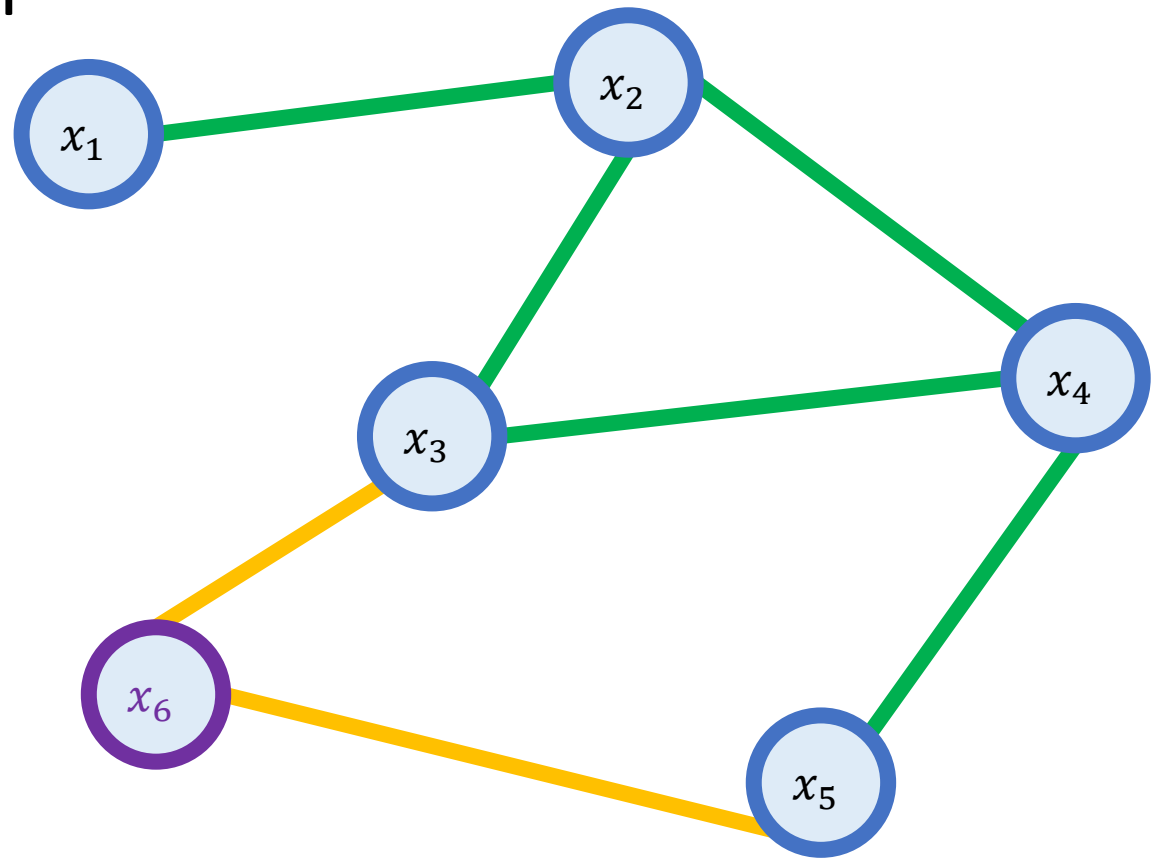


→ Need for scalable embeddings, operating locally for each node, instead of the "general" graph matrices.

Limits of the previous approaches

Issue #2: Spectral embeddings work on fixed graphs structures

- Our spectral embeddings work for a single given graph.
- Adding a new node in the graph requires to reshape the whole embedding architecture.
- Also fails if nodes names are swapped, even though it does not really change the graph structure, so to speak…

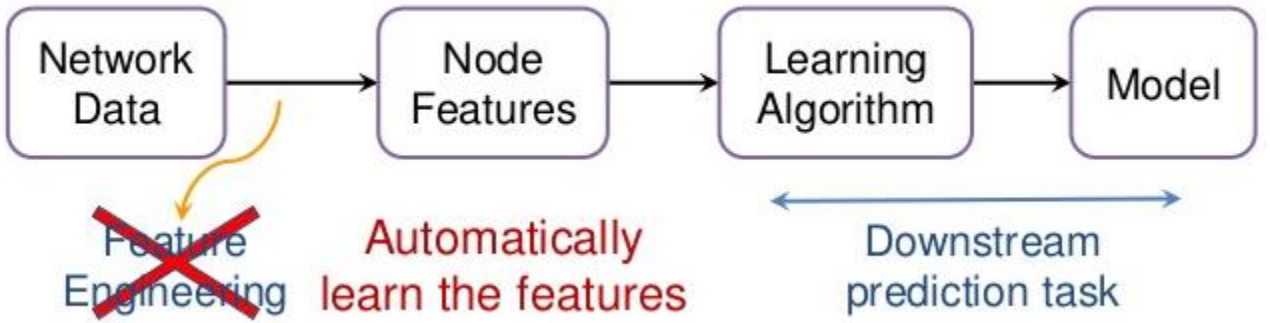


→ Need for scalable embeddings, operating locally for each node, instead of the "general" graph matrices.

Limits of the previous approaches

Issue #3: Critical importance of nodes/edges features

- We have seen the importance of nodes features for proper classification.



→ Need an automated process for learning the optimal features elements in a graph.

Limits of the previous approaches

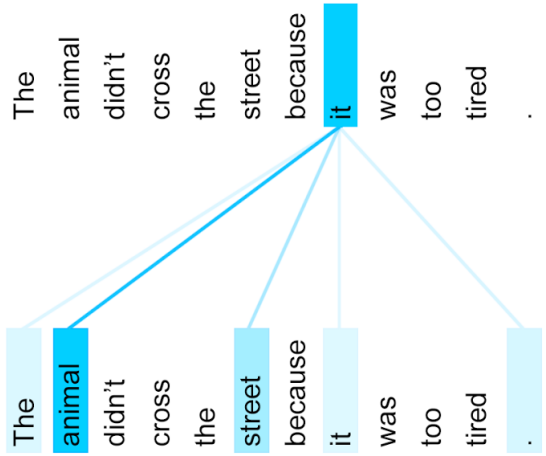
Issue #4: Helping the graph identify more important nodes to propagate

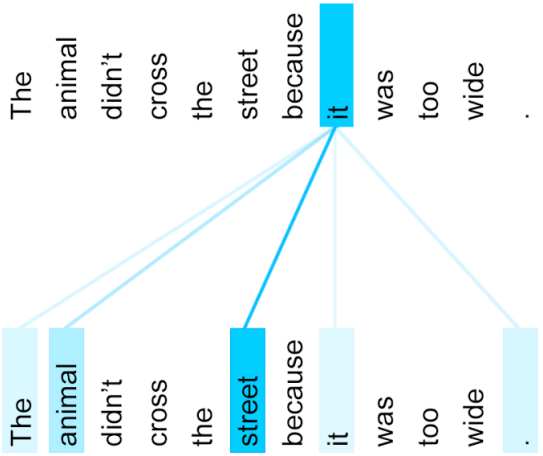
- Attention: selectively concentrating on a few relevant things, while ignoring others in deep neural networks.

Limits of the previous approaches

Issue #4: Helping the graph identify more important nodes to propagate

- Attention: selectively concentrating on a few relevant things, while ignoring others in deep neural networks.
- Attention in NLP: identify words that are most relevant for a given word and propagate meaning to update word embedding.

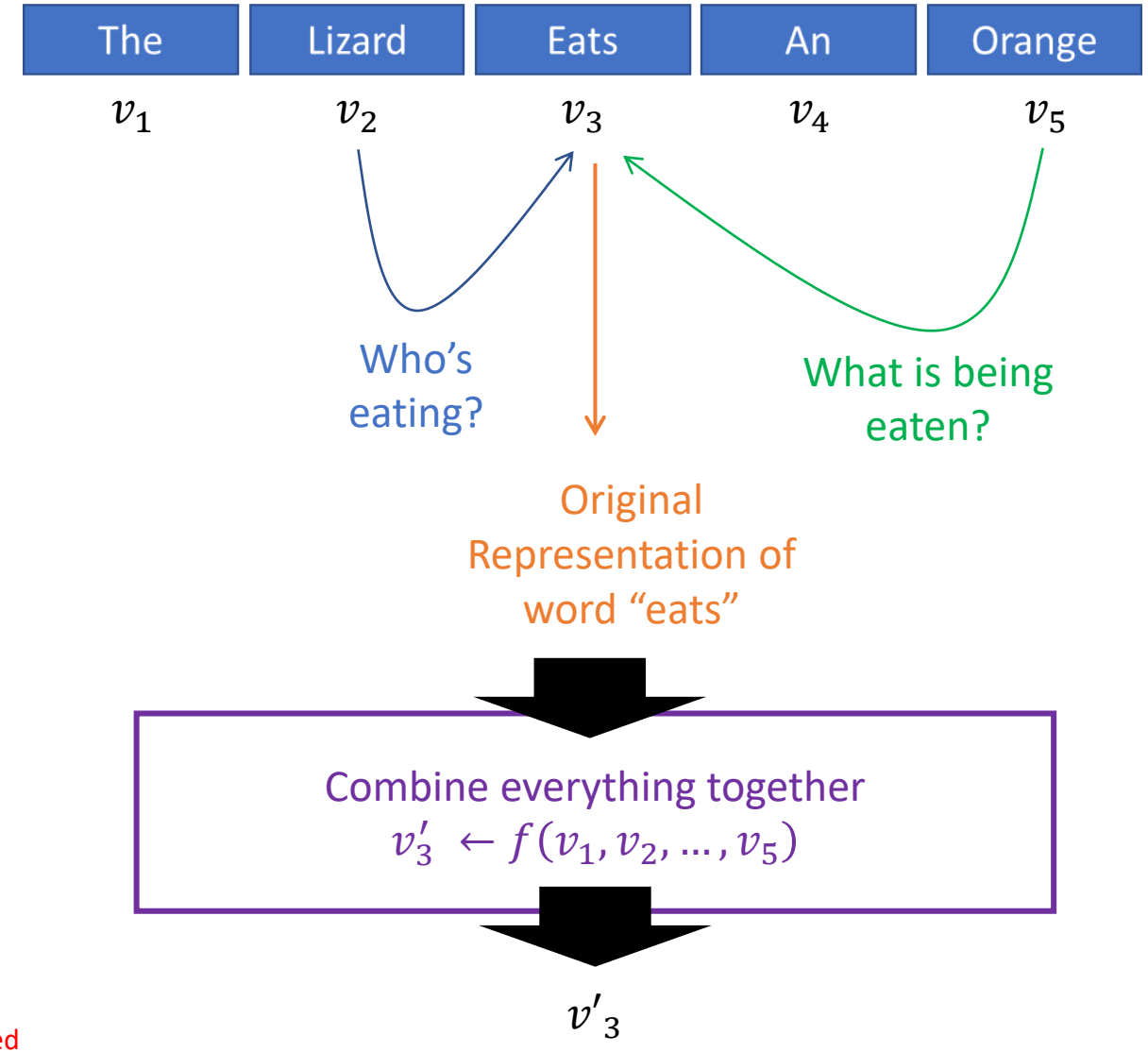




Limits of the previous approaches

Issue #4: Helping the graph identify more important nodes to propagate

- Attention: selectively concentrating on a few relevant things, while ignoring others in deep neural networks.
- Attention in NLP: identify words that are most relevant for a given word and propagate meaning to update word embedding.

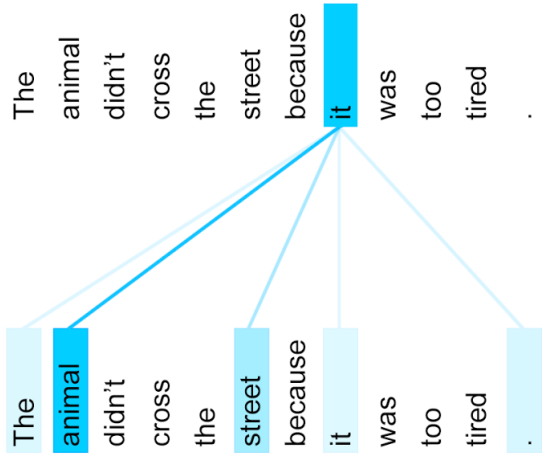


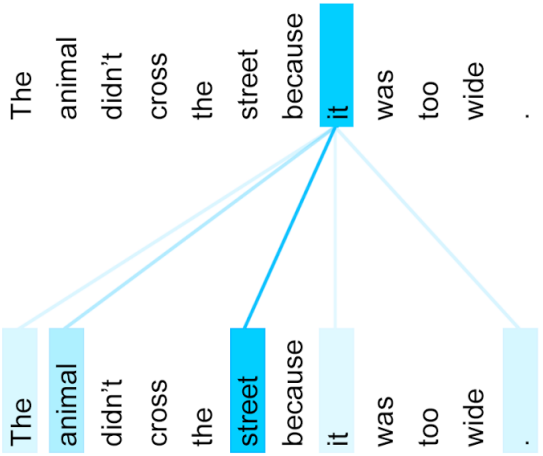
Limits of the previous approaches

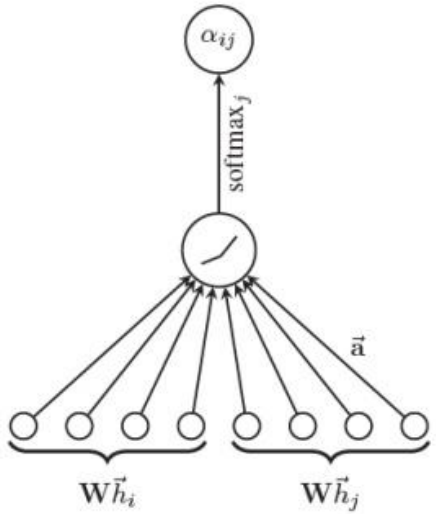
Issue #4: Helping the graph identify more important nodes to propagate

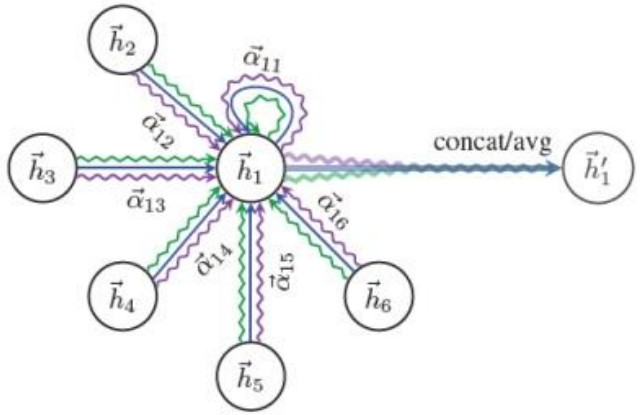
- Attention: selectively concentrating on a few relevant things, while ignoring others in deep neural networks.

→ Question: can we steal the attention concept from NLP and apply it in graphs to identify relevant neighboring nodes features for each node?









In this lecture

Graph Convolutional Networks: on-going research, with many questions yet to be solved.

Out of this lecture:

- Implementing CNN-inspired layers, e.g. Max/Mean Pooling, on graphs. NLP-inspired concepts can also be used.
- Graph-based Generative Adversarial Networks (reusing GANs concepts from next week)
- Etc.

Local/advanced embeddings:

- DeepWalk
- Node2Vec
- Inductive Representation Learning (GraphSAGE)
- Graph Attention Networks

Going deeper: Good white paper, summarizing all concepts of GCN ***[Xu]***

[Xu] Xu, K., Hu, W., Leskovec, J., & Jegelka, S. (2018). How powerful are graph neural networks?.

Local embeddings: DeepWalk

- **Introduced in *[DeepWalk].***

- **Objective:** learn a latent representation/embedding of the adjacency matrix of a graph,
- i.e. a matrix of size $N \times d$, with N the number of nodes, and $d \ll N$.

[DeepWalk] Perozzi, B., Al-Rfou, R., & Skiena, S. (2014, August). Deepwalk: Online learning of social representations.

Local embeddings: DeepWalk

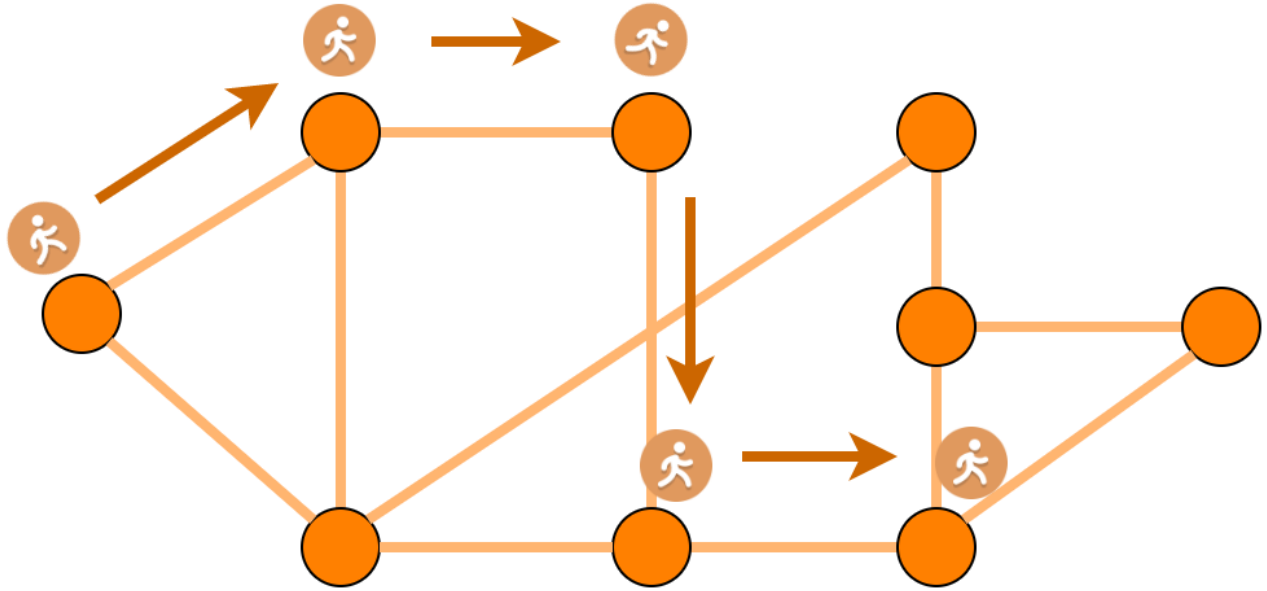
- **Introduced in *[DeepWalk].***
- **Objective:** learn a latent representation/embedding of the adjacency matrix of a graph,
- i.e. a matrix of size $N \times d$, with N the number of nodes, and $d \ll N$.

- **Core idea:** start from a node, and walk randomly in the graph (transition randomly from on node to another using edges)
- **Analogy NLP/Graph:** Short sentences (NLP) = short random walks (Graphs)

[DeepWalk] Perozzi, B., Al-Rfou, R., & Skiena, S. (2014, August). Deepwalk: Online learning of social representations.

Local embeddings: DeepWalk

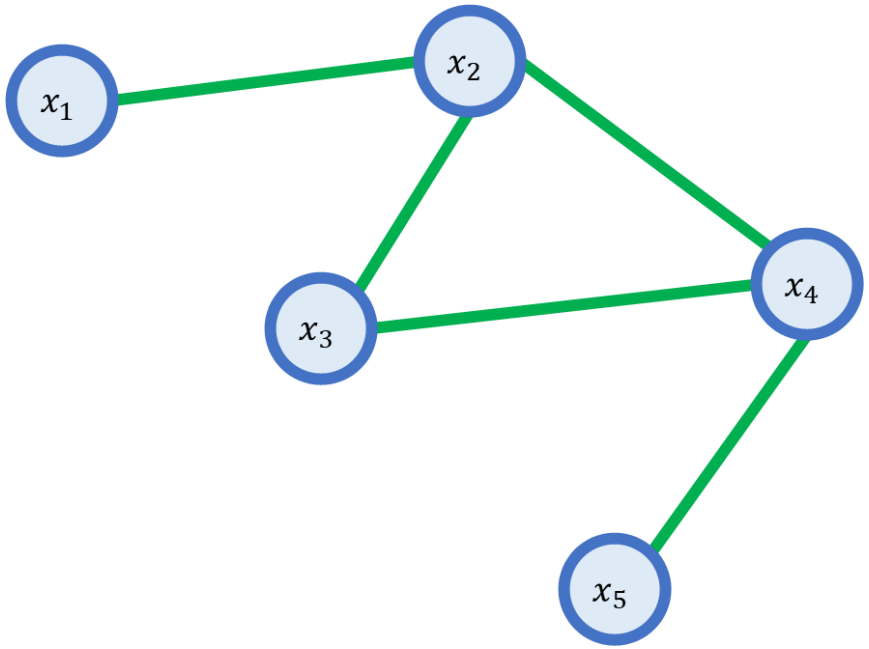
- **Procedure:** For each node in graph, generate γ random walks starting from node, with t transitions. Pick the next node, uniformly among all connected neighbors.
- **Parameters:** γ defines the number of exploration moves, and t how deep each exploration gets in the graph.



- **Note:** No need to know the adjacency matrix. Each node only need to know its list of immediate neighbours.

Local embeddings: DeepWalk

- **Random walks:** you obtain a list of $N\gamma$ sequences $[v_1, \ldots, v_t]$, with
  - $v_1 = x_i$, the starting node x_i,
  - and $\forall k \in [1, t-1], (v_k, v_{k+1}) \in E$.

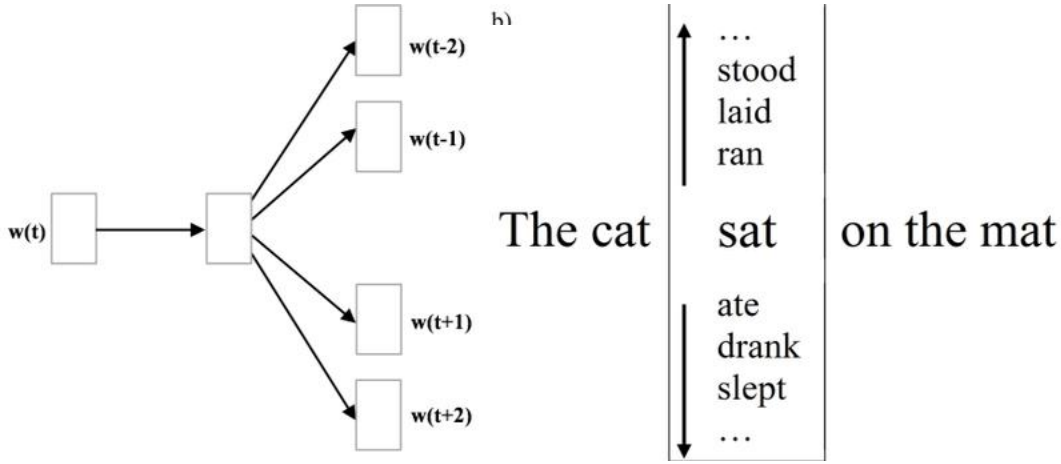


γ indep. random walks

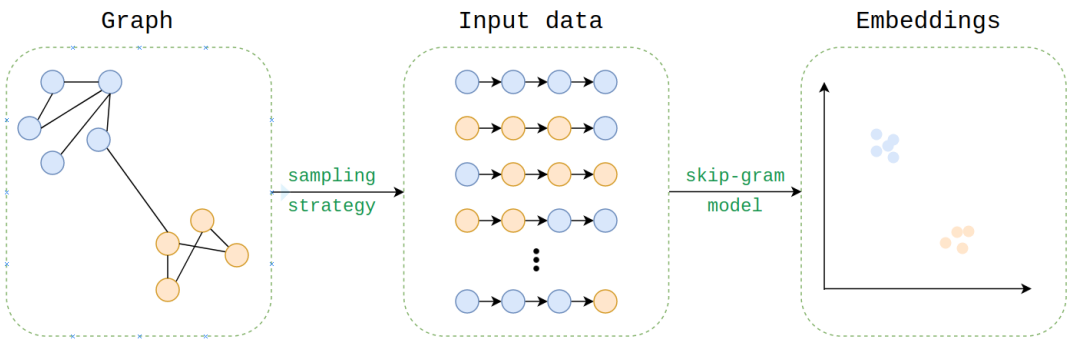
Starting node	t-nodes random walk (e.g., with $t = 4$)
x_1	(x_2, x_3, x_4, x_2)
x_1	(x_2, x_1, x_2, x_1)
…	…
x_1	(x_2, x_3, x_2, x_1)
x_2	(x_3, x_4, x_5, x_4)
…	…
x_3	(x_4, x_2, x_1, x_2)
…	…
x_5	(x_4, x_3, x_4, x_2)

Local embeddings: DeepWalk

- **Random walks:** you obtain a list of $N\gamma$ sequences $[v_1, \ldots, v_t]$, with
  - $v_1 = x_i$, the starting node x_i,
  - and $\forall k \in [1, t-1], (v_k, v_{k+1}) \in E$.



- **Then, SkipGram:** apply SkipGram on all sequences, as in NLP, to obtain an embedding, giving the most likely connected nodes for any input node.



Analogy NLP/Graph for SkipGram

NLP (SkipGram)

- Dataset consists of pairs
  - x_i = middle word,
  - y_i = (context for middle word, defined as a vector of $2k$ surrounding words),
  - Generated from a large corpus of text.
- Predict context (possible surrounding words for a given middle word).
- Extract embedding layer for middle word.

Graph (DeepWalk)

- Dataset consists of pairs
  - x_i = starting node,
  - y_i = (context for node, defined as a vector of t surrounding nodes),
  - Generated randomly by using random walks in the graph.
- Predict context (surrounding nodes visited during random walks), for a given starting node.
- Extract embedding layer for node later.

Analogy NLP/Graph for SkipGram

NLP (SkipGram)

- Dataset consists of pairs
  middle word and
  ng
  middle word).
- Extract embedding layer for middle word.

Definition (the distributional hypothesis of linguistics):

We may rely on a fundamental linguistic assumption
→ Words often appearing in similar contexts/sentences tend to be related to each other.

Graph (DeepWalk)

- Dataset consists of pairs
  - x_i = starting node,
  - y_i = (context for node, defined as a vector of t surrounding nodes),
  - Generated randomly by using random walks in the graph.
- Predict context (surrounding nodes visited during random walks), for a given starting node.
- Extract embedding layer for node later.

Analogy NLP/Graph for SkipGram

NLP (SkipGram)

- Dataset consists of pairs
- Extract embedding layer for middle word.

Definition (the distributional hypothesis of linguistics):

We may rely on a fundamental linguistic assumption
→ Words often appearing in similar contexts/sentences tend to be related to each other.

Graph (DeepWalk)

- Dataset consists of pairs

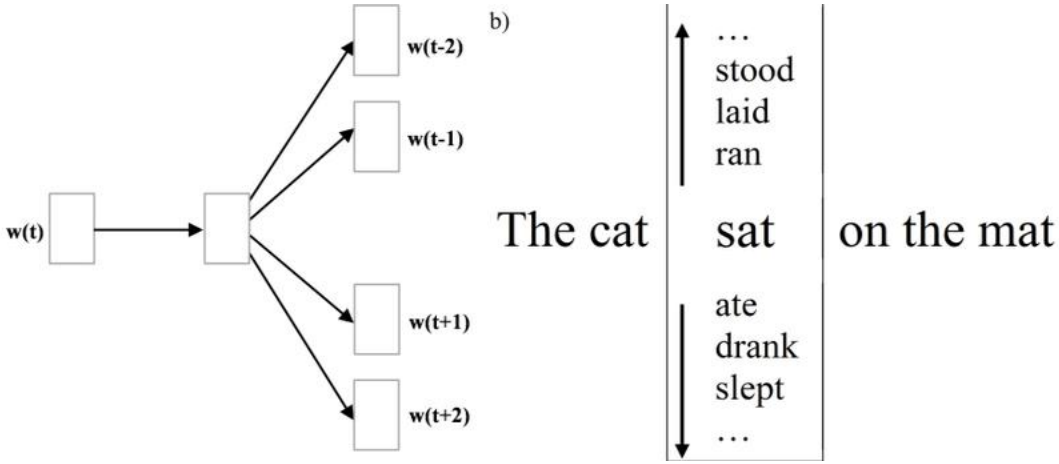
Equivalent (the distributional hypothesis of graphs?!):

Is there an equivalent fundamental graph assumption we could use?
→ Nodes often appearing in similar random walks tend to be related to each other?

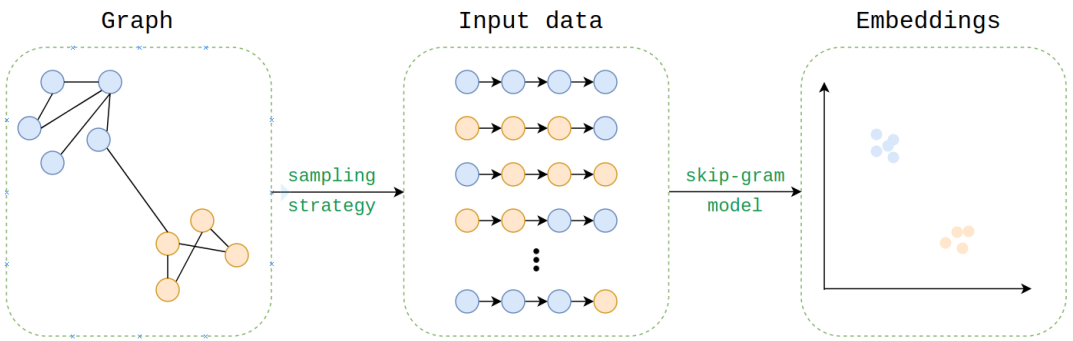
+ Observation (Homophily property of graphs): Connected nodes in a graph tend to have similar features and labels.

Local embeddings: DeepWalk

- **Random walks:** you obtain a list of $N\gamma$ sequences $[v_1, \ldots, v_t]$, with
  - $v_1 = x_i$, the starting node x_i,
  - and $\forall k \in [1, t-1], (v_k, v_{k+1}) \in E$.



- **Then, SkipGram:** apply SkipGram on all sequences, as in NLP, to obtain an embedding, giving the most likely connected nodes for any input node.



Paper with code:
https://paperswithcode.com/paper/deepwalk-online-learning-of-social

Local embeddings: Node2Vec

- **Introduced in *[Node2Vec].***

- **Objective:** learn a latent representation of the adjacency matrix of a graph,
- i.e. a matrix of size $N \times d$, with N the number of nodes, and $d \ll N$.

[Node2Vec] Grover, A., & Leskovec, J. (2016, August). node2vec: Scalable feature learning for networks.

Local embeddings: Node2Vec

- **Introduced in *[Node2Vec].***
- **Objective:** learn a latent representation of the adjacency matrix of a graph,
- i.e. a matrix of size $N \times d$, with N the number of nodes, and $d \ll N$.

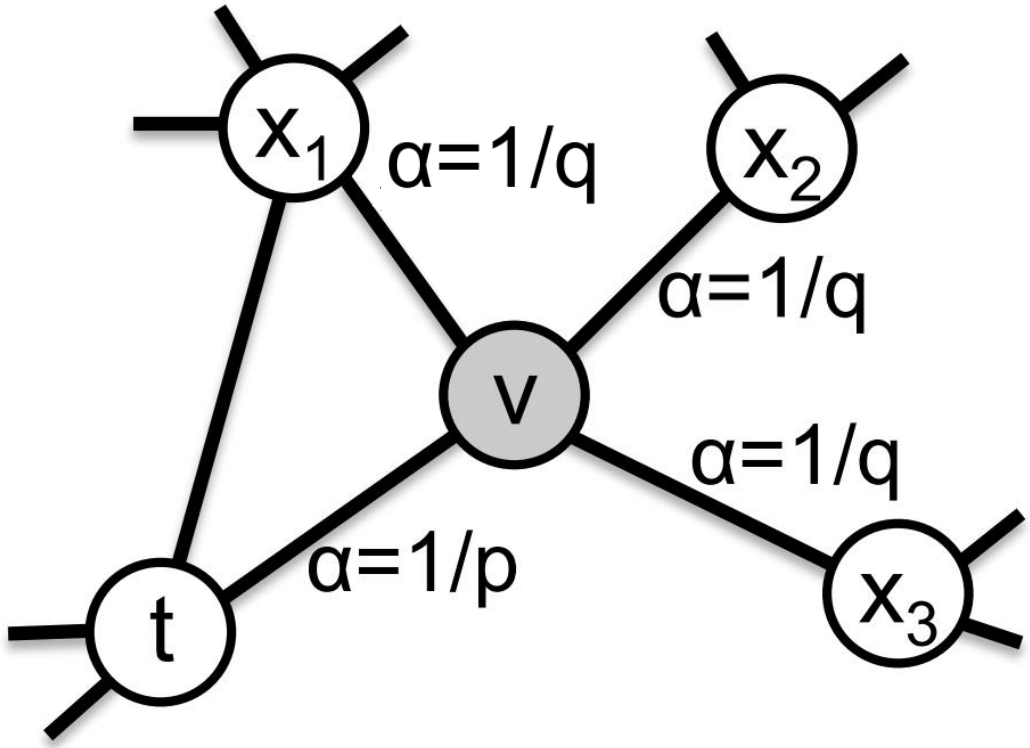
- **Core Idea:** As in DeepWalk, but next node is no longer decided using uniform random, to encourage exploration ($\approx$ tSNE, which will be seen later!).
- **Parameters:**
  - γ, the number of random walks from each node,
  - t, the length of the walk,
  - $1/p$, the probability to return to previously visited node,
  - $1/q$, the probability to visit an unvisited node.
  - Encourage exploration: $1/p < 1/q$

[Node2Vec] Grover, A., & Leskovec, J. (2016, August). node2vec: Scalable feature learning for networks.

Local embeddings: Node2Vec

- **Introduced in *[Node2Vec].***

- Encourage exploration: $1/p < 1/q$
- Note: values of p and q will have to adjust dynamically, based on visited nodes and degree of current node.



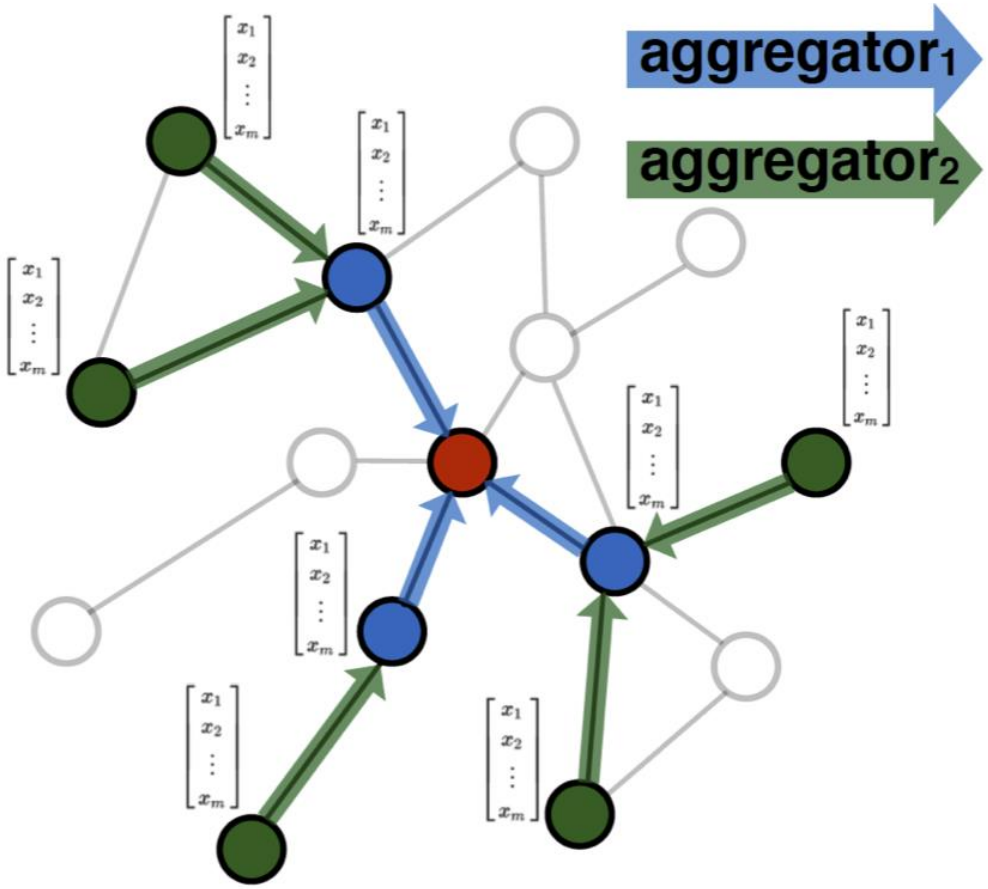
Paper with code:
https://paperswithcode.com/paper/node2vec-scalable-feature-learning-for

[Node2Vec] Grover, A., & Leskovec, J. (2016, August). node2vec: Scalable feature learning for networks.

Advanced Embeddings: GraphSAGE

- Introduced in ***[GraphSAGE].***

- **Core idea:** learn to propagate nodes information/features h_i across the graph to compute new node features h_i'.
- Implemented as a SAGE inductive layer in our Neural Network.



[GraphSAGE] Hamilton, W., Ying, Z., & Leskovec, J. (2017). Inductive representation learning on large graphs.

Advanced Embeddings: GraphSAGE

- Introduced in ***[GraphSAGE].***
- **Core idea:** learn to propagate nodes information/features h_i across the graph to compute new node features h_i'.
- Implemented as a SAGE inductive layer in our Neural Network.

- **Propagation rule:**

$$h_i^{k+1} = \sigma\left(W^k h_i^k, \sum_{j \in nbrs(i)} \sigma\left(Q^k, h_j^k\right)\right)$$

- with $h_i^0 = h_i$, the original node features,
- and $\sum_{j \in nbrs(i)}(\dots)$, an aggregator function (mean, bi-directional RNN, max pooling, etc.)

[GraphSAGE] Hamilton, W., Ying, Z., & Leskovec, J. (2017). Inductive representation learning on large graphs.

Advanced Embeddings: GraphSAGE

- **Propagation rule:**

$$h_i^{k+1} = \sigma\left(W^k h_i^k, \sum_{j \in nbrs(i)} \sigma\left(Q^k, h_j^k\right)\right)$$

- with $h_i^0 = h_i$, the original node features,
- $\sum_{j \in nbrs(i)}(\dots)$, an aggregator function (mean sum, bi-directional RNN, etc.),
- and σ, an activation, preferably ReLU.

Advanced Embeddings: GraphSAGE

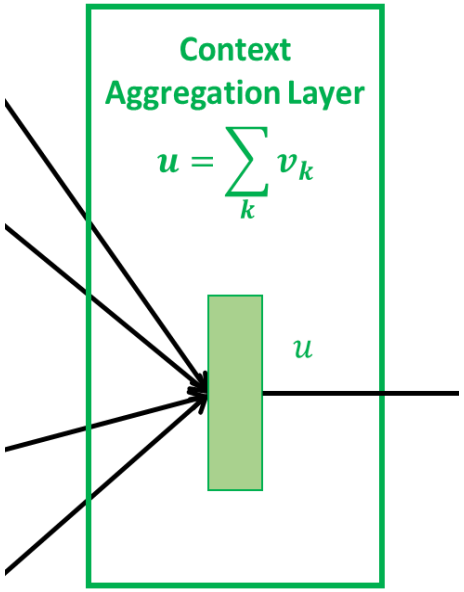
- **Propagation rule:**

$$h_i^{k+1} = \sigma\left(W^k h_i^k, \sum_{j \in nbrs(i)} \sigma\left(Q^k, h_j^k\right)\right)$$

- with $h_i^0 = h_i$, the original node features,
- $\sum_{j \in nbrs(i)}(\dots)$, an aggregator function (mean sum, bi-directional RNN, etc.),
- and σ, an activation, preferably ReLU.

Note: possible aggregators we have seen so far

- Mean sum (as in CBoW)



Advanced Embeddings: GraphSAGE

- **Propagation rule:**

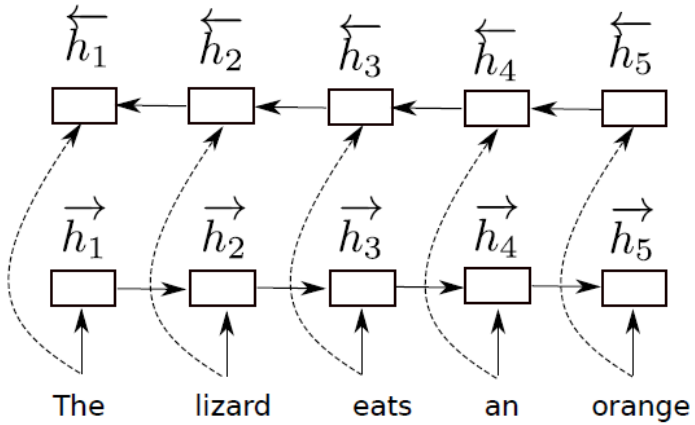
$$h_i^{k+1} = \sigma\left(W^k h_i^k, \sum_{j \in nbrs(i)} \sigma\left(Q^k, h_j^k\right)\right)$$

- with $h_i^0 = h_i$, the original node features,
- $\sum_{j \in nbrs(i)}(\dots)$, an aggregator function (mean sum, bi-directional RNN, etc.),
- and σ, an activation, preferably ReLU.

Note: possible aggregators we have seen so far

- Mean sum (as in CBoW)
- Bi-directional RNN (as in ELMo)

$$h_k = \left[\overleftarrow{h_k}, \overrightarrow{h_k}\right]$$



Advanced Embeddings: GraphSAGE

- **Propagation rule:**

$$h_i^{k+1} = \sigma\left(W^k h_i^k, \sum_{j \in nbrs(i)} \sigma\left(Q^k, h_j^k\right)\right)$$

- with $h_i^0 = h_i$, the original node features,
- $\sum_{j \in nbrs(i)}(\dots)$, an aggregator function (mean sum, bi-directional RNN, etc.),
- and σ, an activation, preferably ReLU.

Note: possible aggregators we have seen so far

- Mean sum (as in CBoW)
- Bi-directional RNN (as in ELMo)
- Many other possibilities (e.g. pooling, etc.)

Advanced Embeddings: GraphSAGE

- **Propagation rule:**

$$h_i^{k+1} = \sigma\left(W^k h_i^k, \sum_{j \in nbrs(i)} \sigma\left(Q^k, h_j^k\right)\right)$$

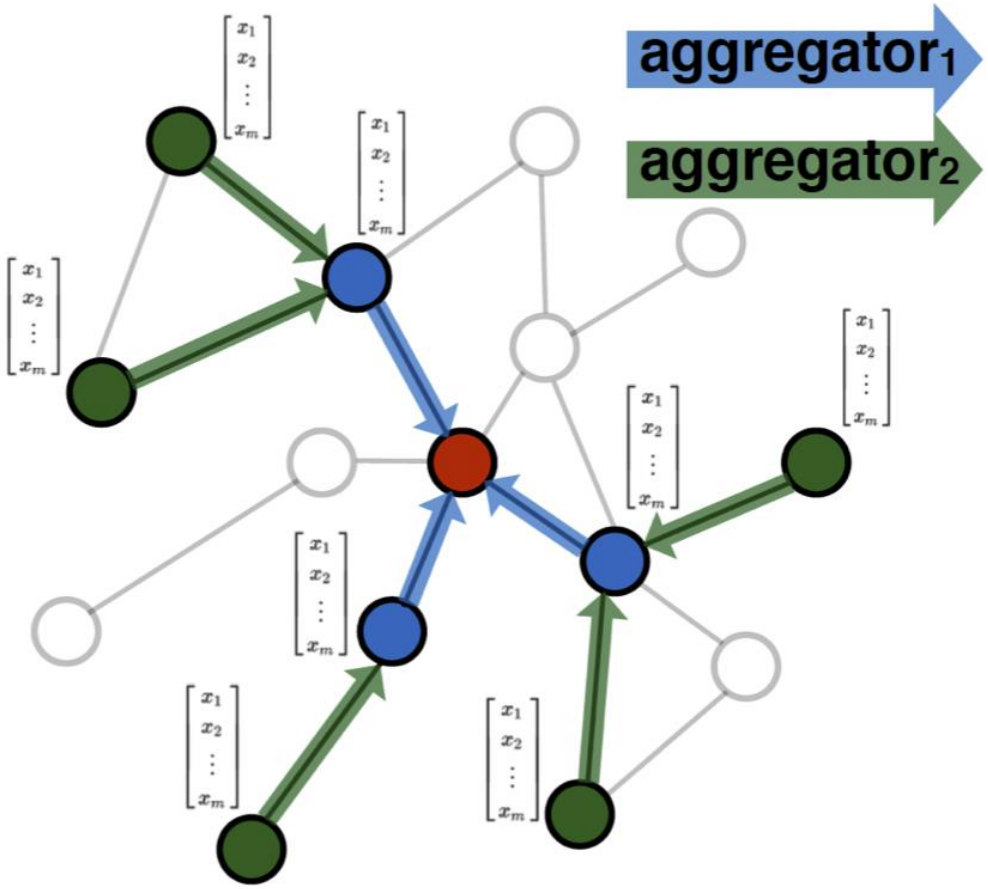
- with $h_i^0 = h_i$, the original node features,
- $\sum_{j \in nbrs(i)}(\dots)$, an aggregator function (mean sum, bi-directional RNN, etc.),
- and σ, an activation, preferably ReLU.

- **Procedure:** supervised training to learn the weights in W^k and Q^k.
- **Advantages:**
  - Model has a constant number of parameters,
  - No need to know the adjacency matrix. Each node only need to know its list of immediate neighbours.
  - Fast and scalable inference,
  - Can be applied to any node in network,
  - Can be applied multiple times in a row to learn deeper features (reuse nodes features further away)

Advanced Embeddings: GraphSAGE

- Introduced in ***[GraphSAGE]*** Hamilton, W., Ying, Z., & Leskovec, J. (2017). Inductive representation learning on large graphs.

- **Paper with code:** https://paperswithcode.com/paper/inductive-representation-learning-on-large

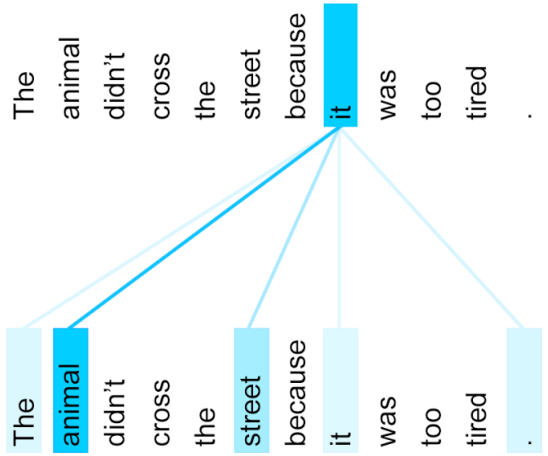


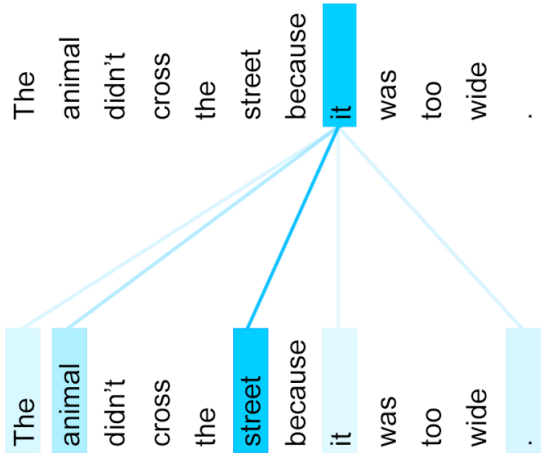
Graph Attention Networks: core idea

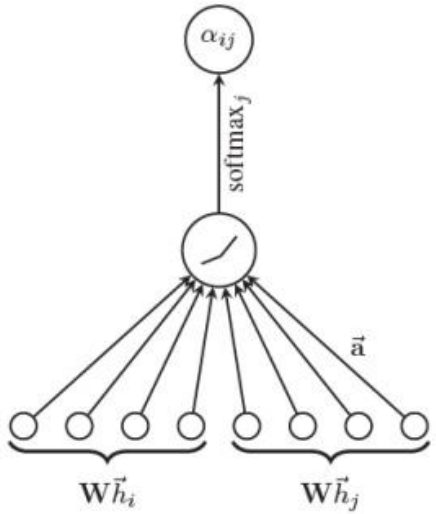
Issue #4: Helping the graph identify more important nodes

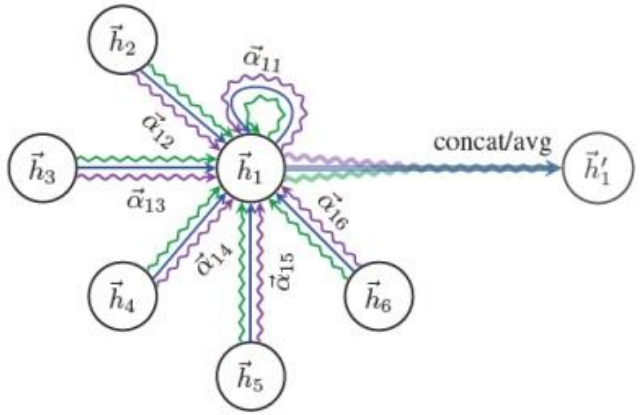
- **Attention:** selectively concentrating on a few relevant things, while ignoring others in deep neural networks.
- **Attention in NLP:** identify words that are most relevant for a given word.

→ Attention in graphs to identify relevant neighboring nodes features for each node?



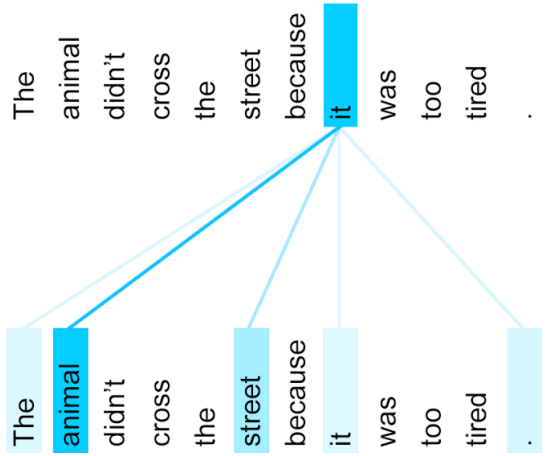


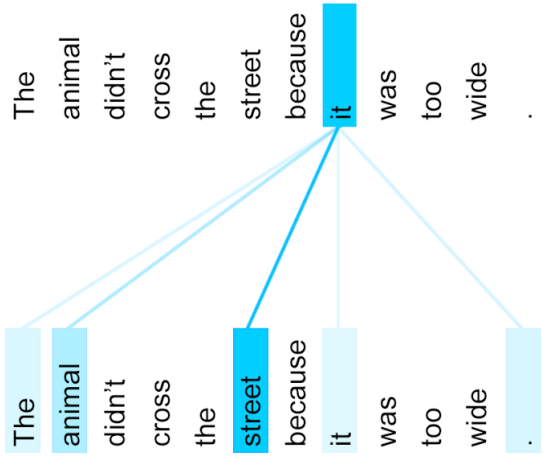


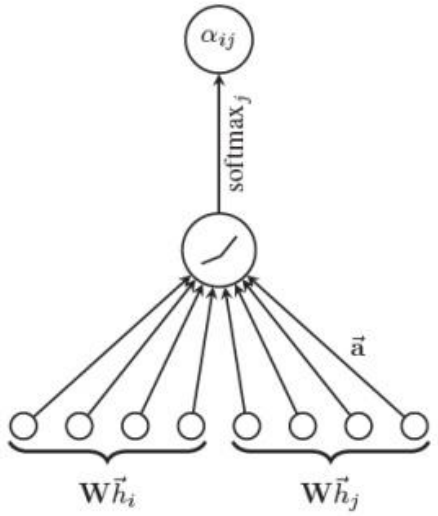


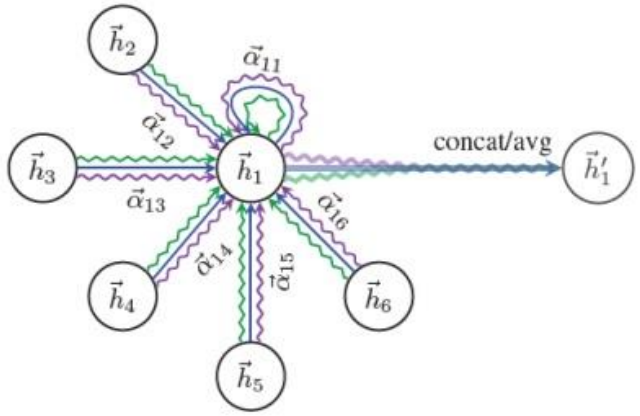
Graph Attention Networks: core idea

- Introduced in ***[GraphSAT].***
- **Core idea:** By analogy with NLP, design a Graph Attention layer that
  - Receives the nodes features $[h_1, \dots, h_i, \dots, h_N]$, for all the nodes $(x_i)_{i \in [1,N]}$ in a given graph, with $h_i \in \mathbb{R}^F$.
  - Outputs a new set of nodes features $[h'_1, \dots, h'_i, \dots, h'_N]$, with $h'_i \in \mathbb{R}^{F'}$.









[GraphSAT] Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., & Bengio, Y. (2017). Graph attention networks.

Graph Attention Networks: implementation

- **Step 1: Linear transformation**
  - Multiply the features h_i of each node x_i in the graph, with a weight matrix W $(\in \mathbb{R}^{F' \times F})$
  - With F, the number of the nodes features in h_i,
  - And $F^{'}$, the number of features expected at the end of the embedding.

Graph Attention Networks: implementation

- **Step 1: Linear transformation**
  - Multiply the features h_i of each node x_i in the graph, with a weight matrix W $(\in \mathbb{R}^{F' \times F})$
  - With F, the number of the nodes features in h_i,
  - And $F^{'}$, the number of features expected at the end of the embedding.

- **Step 2: Compute attention coefficients**
  - Using a shared attentional mechanism a: $\mathbb{R}^{F'} \times \mathbb{R}^{F'} \rightarrow \mathbb{R}$
  - $\forall i, j \in [1, N]$, define the attention coefficient e_{ij} as:

$$e_{ij} = a(Wh_i, Wh_j)$$

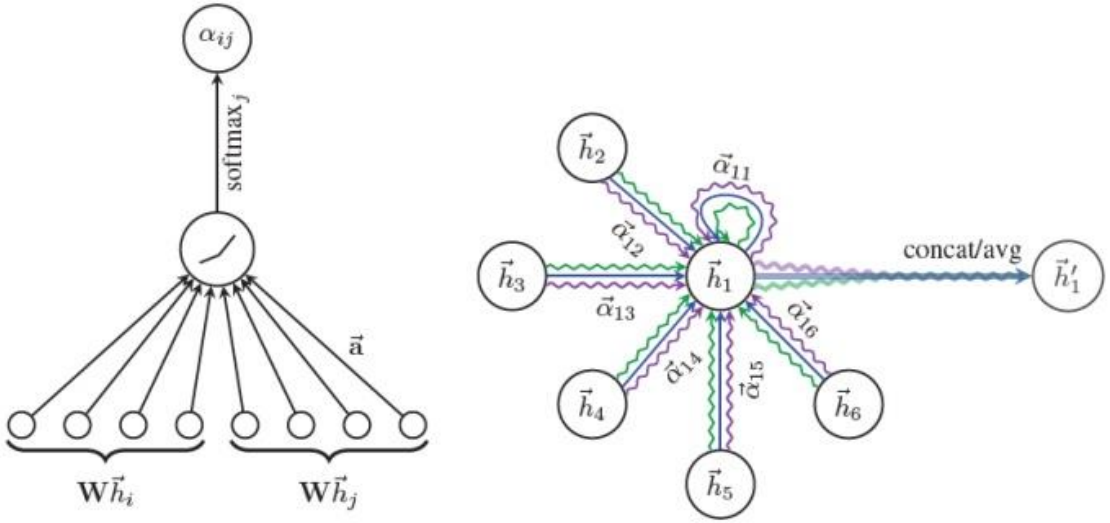
Graph Attention Networks: implementation

- **Step 2(cont'd): Replace a with a feedforward network,**
  - with a weight vector $a \in \mathbb{R}^{2F'}$,
  - and use LeakyReLU as activation.

Graph Attention Networks: implementation

- **Step 2(cont'd): Replace a with a feedforward network,**
  - with a weight vector $a \in \mathbb{R}^{2F'}$,
  - and use LeakyReLU as activation.
- **Step 3: Normalize coefficients with softmax**

$$\alpha_{ij} = \frac{exp(e_{ij})}{\sum_{k \in nbrs(i)} exp(e_{ik})}$$



$$\alpha_{ij} = \frac{exp(e_{ij})}{\sum_{k \in nbrs(i)} exp(e_{ik})} = \frac{exp\left(LeakyReLU(a^T[Wh_i||Wh_j])\right)}{\sum_{k \in nbrs(i)} exp\left(LeakyReLU(a^T[Wh_i||Wh_k])\right)}$$

Graph Attention Networks: implementation

- **Step 4: Propagation rule**

$$h_i' = \sigma\left(\sum_{j \in nbrs(i)} \alpha_{ij} W h_j\right)$$

Graph Attention Networks: implementation

- **Step 4: Propagation rule**

$$h_i' = \sigma\left(\sum_{j \in nbrs(i)} \alpha_{ij} W h_j\right)$$

- **Step 5 (optional): use $K > 1$ heads for attention mechanisms**

$$h_i' = \bigcup_{k=1}^{K} \sigma\left(\sum_{j \in nbrs(i)} \alpha_{ij}^k W^k h_j\right)$$

Graph Attention Networks: implementation

- **Step 4: Propagation rule**

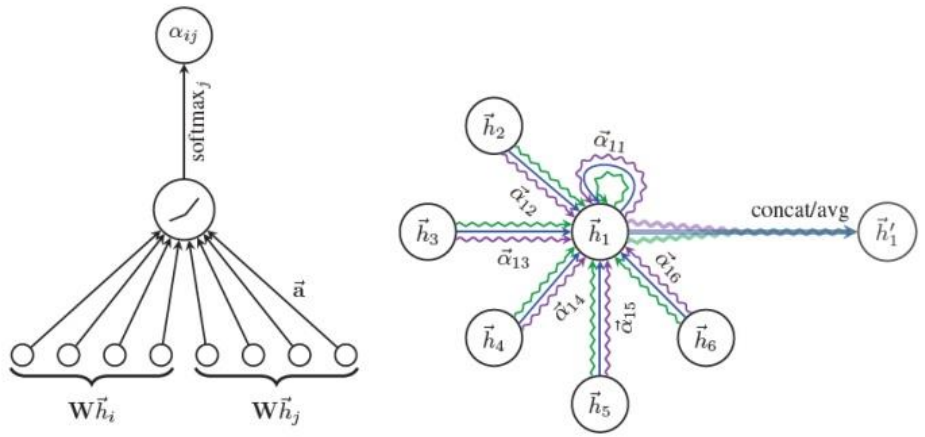
$$h_i' = \sigma\left(\sum_{j \in nbrs(i)} \alpha_{ij} W h_j\right)$$

- **Step 5 (optional): use $K > 1$ heads for attention mechanisms**

$$h_i' = \bigcup_{k=1}^{K} \sigma\left(\sum_{j \in nbrs(i)} \alpha_{ij}^k W^k h_j\right)$$

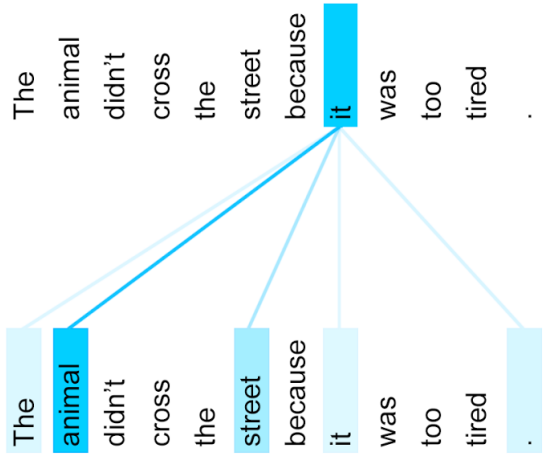
- With $\bigcup_{k=1}^{K}$ …, an aggregation mechanism (such as concatenation, or as shown below, averaging)

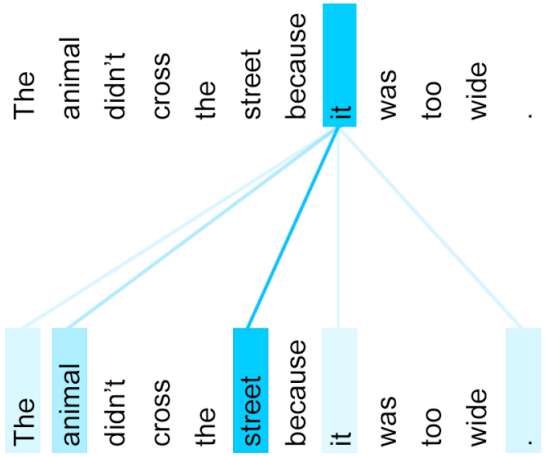
$$h_i' = \frac{1}{K} \sigma\left(\sum_{k=1}^{K} \sum_{j \in nbrs(i)} \alpha_{ij}^k W^k h_j\right)$$



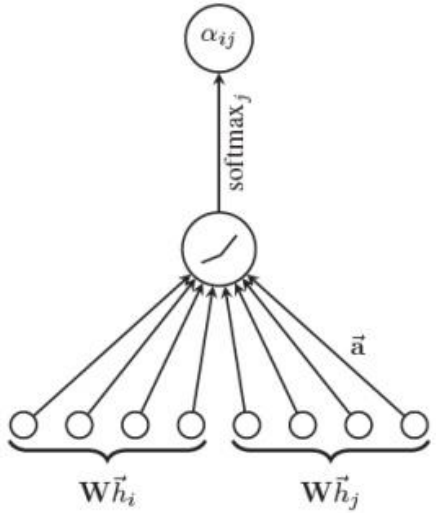
Graph Attention Networks: recap

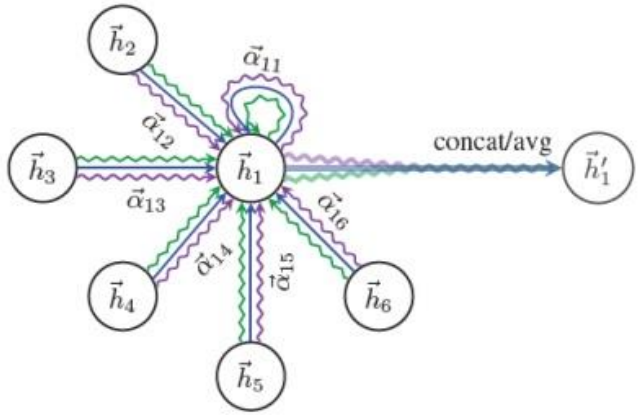
- Introduced in ***[GraphSAT]*** Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., & Bengio, Y. (2017). Graph attention networks.





- **Paper with code:** https://paperswithcode.com/paper/graph-attention-networks





Conclusion

In this lecture

- Building a Graph Convolutional Neural Network for classification
- Training a Graph Convolutional Neural Network
- Feature Engineering for Graph Convolutional Neural Networks

In this lecture, also

- Limits of basic approaches
- Graph Convolutional Neural Networks with Attention Mechanisms
- Some more advanced embeddings
- Open questions in Graph Convolutional Neural Networks

Learn more about these topics

Out of class, for those of you who are curious

- [Kipf] **Kipf** et al., “Semi-supervised classification with graph convolutional networks”, 2016.
- [ZKC] Zachary, “An information flow model for conflict and fission in small groups”, 1977.
- [Xu] Xu et al., “How powerful are graph neural networks?”, 2018.
- [DeepWalk] Perozzi et al., “Deepwalk: Online learning of social representations”, 2014.

Learn more about these topics

Out of class, for those of you who are curious

- [Node2Vec] Grover et al. , “Node2vec: Scalable feature learning for networks”, 2016.
- [GraphSAGE] Hamilton et al., “Inductive representation learning on large graphs”, 2017.
- [GraphSAT] **Veličković**, Cucurull, Casanova, **Romero**, Lio, & **Bengio**, “Graph attention networks”, 2017.

Learn more about these topics

Tracking important names (Track their works and follow them on Scholar, Twitter, or whatever works for you!)

- **Thomas Kipf**: **Research Scientist** at **Google Brain**.
  https://scholar.google.de/citations?user=83HL5FwAAAAJ&hl=en
  https://tkipf.github.io
- **Max Welling**: **Professor** at **University of Amsterdam**.
  https://scholar.google.com/citations?user=8200InoAAAAJ&hl=fr&oi=sra
- **Rianne Van Den Berg: Researcher** at **Microsoft.**
  https://scholar.google.com/citations?user=KARgiboAAAAJ&hl=fr&oi=sra

Learn more about these topics

Tracking important names (Track their works and follow them on Scholar, Twitter, or whatever works for you!)

- **Adriana Romero-Soriano: Researcher** at **Facebook AI**. https://scholar.google.com/citations?user=Sm15FXIAAAAJ&hl=fr&oi=sra
- **Petar Veličković: Researcher** at **DeepMind** and **Affiliated Lecturer** at **Cambridge University**. https://scholar.google.com/citations?user=kcTK_FAAAAAJ&hl=fr&oi=sra
- **Jure Leskovec**: **Professor** at **Stanford University**. https://scholar.google.com/citations?user=Q_kKkIUAAAAJ&hl=fr&oi=sra