DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz

Part II

# Basic GNNs & Applications

Lecture 05

# Introduction to Graph Convolution Networks (GCN)

- Learn GCN fundamentals and its significance in graph-based data analysis.
- Understand message passing and smart normalization in GCNs.
- Apply GCNs to various graph-based tasks for valuable insights.

DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz


# Challenges and Limitations of Vanilla GNN

## Recall of Vanilla GNN

### Fundamental Concepts of Vanilla GNN

**Global Transition Function (Message Passing):**

- The vanilla GNN employs a simple layer-wise propagation rule:

$$\Psi_{W}\left(H^{(k)}, A\right) = H^{(k+1)}$$

Learned Parameters

$$H^{(k)} = \begin{bmatrix} h_1^T \\ h_2^T \\ \vdots \\ h_{|v|}^T \end{bmatrix}$$

Adjacency Matrix

$$H^{(k+1)} = \begin{bmatrix} \psi_W(h_1^{(k)}, h_{\aleph_1}^{(k)})^T \\ \psi_W(h_2^{(k)}, h_{\aleph_2}^{(k)})^T \\ \vdots \\ \psi_W(h_{|v|}^{(k)}, h_{\aleph_{|v|}}^{(k)})^T \end{bmatrix}$$

$$H^{(k+1)} = \Psi\left(A H^{(k)} W^{(k)}\right)$$

Layer (k+1)

Activation Function

Wight Matrix at Layer (k)

## Self-Loop Enforcing

$$\tilde{A} = A + I$$

$$H^{(k+1)} = \Psi\left(\tilde{A} H^{(k)} W^{(k)}\right)$$

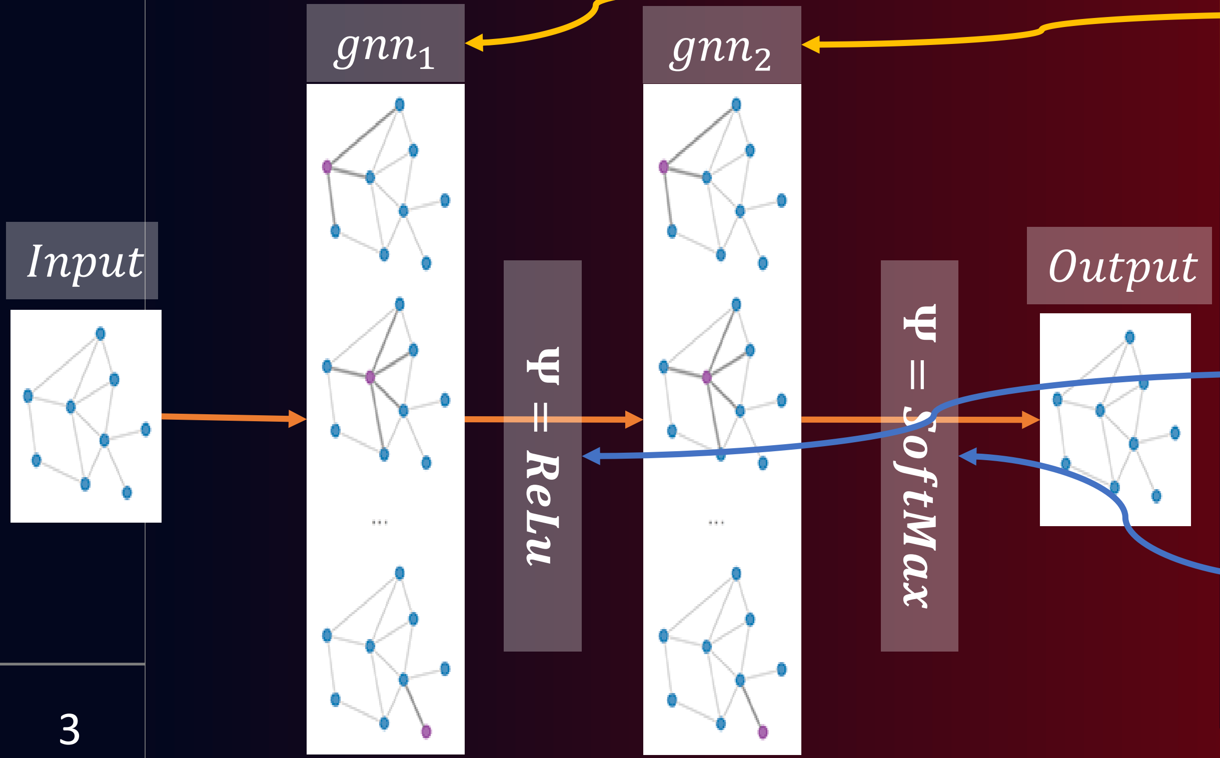
DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz



# Challenges and Limitations of Vanilla GNN

## Recall of Vanilla GNN

| Dataset | MLP | Vanilla GNN |
|---|---|---|
| Cora | 51.90% | 72.50% |
| Facebook | 75.21% | 84.85% |




```python
# Create a new class named VanillaGNN for our GNN
class VanillaGNN(nn.Module):
    # Initialize the VanillaGNN
    def __init__(self, dim_in, dim_h, dim_out):
        super().__init__()
        # Initialize 2 VanillaGNNLayer layers
        self.gnn1 = VanillaGNNLayer(dim_in, dim_h)
        self.gnn2 = VanillaGNNLayer(dim_h, dim_out)

    # Perform the  Forward pass of the VanillaGNN
    def forward(self, x, adjacency):
        # Apply the 1st GNN layer
        h = self.gnn1(x, adjacency)
        # Apply ReLU activation
        h = torch.relu(h)
        # Apply the 2nd GNN layer
        h = self.gnn2(h, adjacency)
        # Return Log softmax for classification
        return F.log_softmax(h, dim=1)

    # ................ Other def Functions ...........
```

DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz



# Challenges and Limitations of Vanilla GNN

| | |
|---|---|
| **Unbalanced Neighbor Counts** | In real-world graphs, nodes often have varying numbers of neighbors. |
| | **Example:** Node 1 has 3 neighbors, while node 2 has only 1. |
| **Unanticipated Problem:** | Vanilla GNN layers treat all neighbors equally with a simple aggregation operation. |
| | No consideration for the difference in neighbor counts. |
| **Consequence: Inconsistent Embeddings** | Node 1 (with 1,000 neighbors) and Node 2 (with only 1 neighbor) create embeddings with vastly distinct scales. |
| | This scale variation hinders meaningful comparisons between embeddings. |

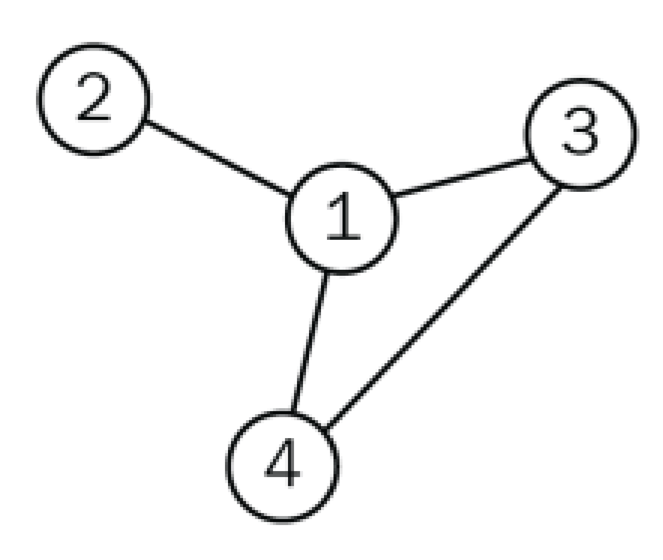



Fig. – Simple graph where nodes have different numbers of neighbors

$\boldsymbol{A}$ or $\tilde{\boldsymbol{A}}$ is typically not normalized.

DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz



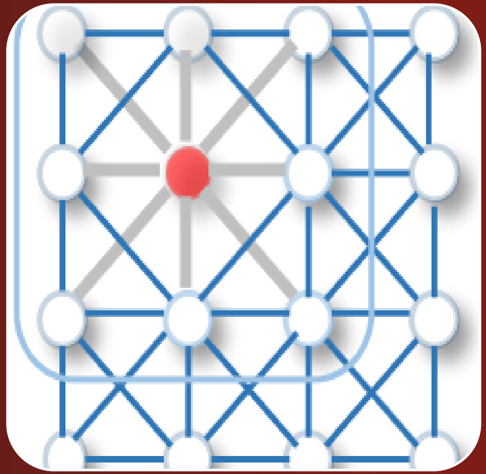
# Graph Convolutional Networks (GCN)

## Inspiration From 2d Conv

**Addressing These Limitations:**

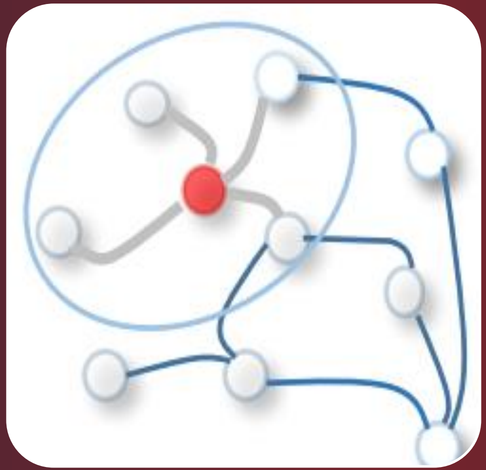
Researchers have proposed more advanced methods like **Graph Convolutional Networks** (**GCNs**).

**GCNs** aim to tackle the issues **of vanilla GNNs** through **normalization** and more **sophisticated weight assignments**.



**2D Convolution:**

- Treats each image pixel as a node.
- Neighbors determined by fixed grid size (The filter size ).
- Computes weighted average of pixel values with fixed-size, ordered neighbors.
- Neighbors are ordered and consistently sized.



**Graph Convolution:**

- Aims to obtain a hidden representation of a target node.
- Simple approach: average the node features of the target node and its neighbors.
- Graph neighbors are unordered and variable in size.
- Adaptable to diverse and irregular data structures.

DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz



# Graph Convolutional Networks (GCN)

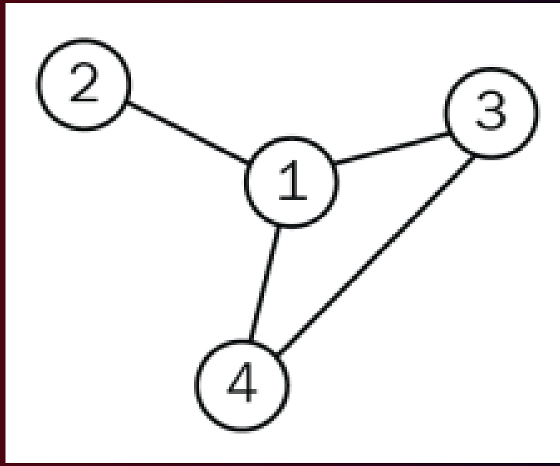
## Normal Normalization

Normalizing $\boldsymbol{A}$ or $\widetilde{\boldsymbol{A}}$ such that all rows sum to **one** ➔
Taking the average of neighboring node features:

$$\boldsymbol{D}^{-1}\boldsymbol{A} \quad \text{or} \quad \widetilde{\boldsymbol{D}}^{-1}\widetilde{\boldsymbol{A}}$$

Where:
- $\boldsymbol{D}$ is the diagonal node degree matrix of $\boldsymbol{A}$
- $\widetilde{\boldsymbol{D}}$ is the diagonal node degree matrix of $\widetilde{\boldsymbol{A}}$

$$\widetilde{\boldsymbol{A}} = \boldsymbol{A} + \boldsymbol{I}$$




```
A_delta:
[[1. 1. 1. 1.]
 [1. 1. 0. 0.]
 [1. 0. 1. 1.]
 [1. 0. 1. 1.]]
```

```
Adjacency Matrix A:
[[0 1 1 1]
 [1 0 0 0]
 [1 0 0 1]
 [1 0 1 0]]
```

```
D_delta:
[[4. 0. 0. 0.]
 [0. 2. 0. 0.]
 [0. 0. 3. 0.]
 [0. 0. 0. 3.]]
```

```
Degree Matrix D:
[[3 0 0 0]
 [0 1 0 0]
 [0 0 2 0]
 [0 0 0 2]]
```

DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz



# Graph Convolutional Networks (GCN)

## Normal Normalization

Normalizing $A$ or $\tilde{A}$ such that all rows sum to **one** ➔
Taking the average of neighboring node features:

$$D^{-1}A \quad \text{or} \quad \tilde{D}^{-1}\tilde{A}$$

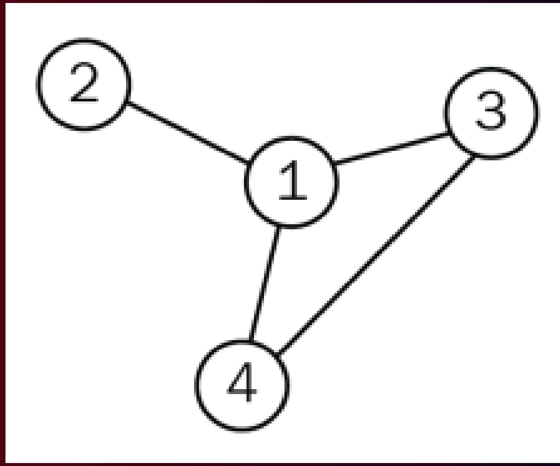
Where:

- $D$ is the diagonal node degree matrix of $A$
- $\tilde{D}$ is the diagonal node degree matrix of $\tilde{A}$

$$\tilde{A} = A + I$$

This normalizes neighboring node features

Rows Sum to 1




```
D^{-1}:
[[0.33 0.   0.   0.  ]
 [0.   1.   0.   0.  ]
 [0.   0.   0.5  0.  ]
 [0.   0.   0.   0.5 ]]
```

```
D_delta^{-1}:
[[0.25 0.   0.   0.  ]
 [0.   0.5  0.   0.  ]
 [0.   0.   0.33 0.  ]
 [0.   0.   0.   0.33]]
```

```
D^{-1}.A:
[[0.   0.33 0.33 0.33]
 [1.   0.   0.   0.  ]
 [0.5  0.   0.   0.5 ]
 [0.5  0.   0.5  0.  ]]
```

$D^{-1}A$

```
D_delta^{-1}.A_delta:
[[0.25 0.25 0.25 0.25]
 [0.5  0.5  0.   0.  ]
 [0.33 0.   0.33 0.33]
 [0.33 0.   0.33 0.33]]
```

$\tilde{D}^{-1}\tilde{A}$

DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz



# Graph Convolutional Networks (GCN)

## Symmetric Normalization

**Normal Normalization**

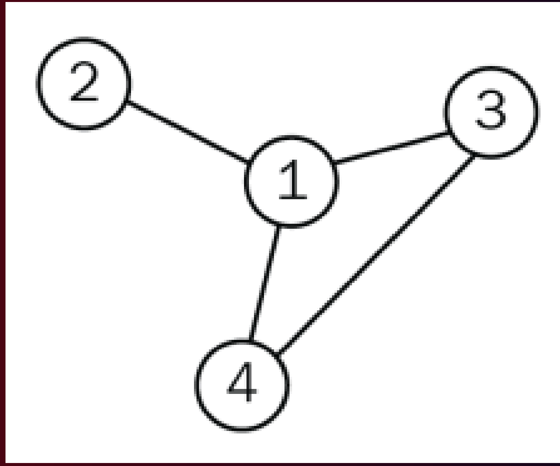
The multiplication $\tilde{D}^{-1}\tilde{A}$ results in an **imbalance**, where nodes with **higher degrees** have more **influence** in the **message passing**.

**Symmetric Normalization**

We use instead: $\tilde{D}^{-1/2}\tilde{A}\,\tilde{D}^{-1/2}$

$\tilde{D}^{-1/2}$ is the diagonal matrix with the **square root** of node degrees.

This form of normalization ensures that the **propagation weights** are equally distributed among neighboring nodes.




```
Symmetric Normalization (D^(-1/2) * A * D^(-1/2)):
[[0.   0.58 0.41 0.41]
 [0.58 0.   0.   0.  ]
 [0.41 0.   0.   0.5 ]
 [0.41 0.   0.5  0.  ]]
```

Kipf, Thomas N., and Max Welling. "Semi-supervised classification with graph convolutional networks." arXiv preprint arXiv:1609.02907 (2016).

DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz



# Graph Convolutional Networks (GCN)

## Overview of GCN

### Fundamental Concepts of GCN

**Global Transition Function (Message Passing):**

- The GCN employs a specific layer-wise propagation rule:

$$\Psi_{W}(H^{(k)}, [\widetilde{A}, \widetilde{D}]) = H^{(k+1)}$$

- $W$: Learned Parameters
- $H^{(k)}$: Current State at Layer $(k)$
- $\widetilde{A}$: Adjacency Matrix With Self-Loop Enforcement
- $\widetilde{D}$: Diagonal Degree Matrix of $\widetilde{A}$
- $H^{(k+1)}$: Next State at Layer $(k+1)$

$$H^{(k+1)} = \Psi(\widetilde{D}^{-1/2}\widetilde{A}\widetilde{D}^{-1/2}H^{(k)}W^{(k)})$$

DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz



# Graph Convolutional Networks (GCN)

01 __init__()

02 forward()

## Manual Implementing of the GCN Layer:

```
# …Import necessary libraries
class GCNLayer(nn.Module):
    def __init__(self, in_features, out_features):
        # Initialize the GCNLayer with input and output feature
          dimensions.
        super(GCNLayer, self).__init__()
        # Create a learnable weight parameter.
        # This weight matrix will be fine-tuned during training.
        self.weight =
                    nn.Parameter(torch.FloatTensor(in_features,
                                                   out_features))
        # Initialize the weight matrix using the Xavier (Glorot)
          initialization method.
        # Xavier initialization helps with the convergence of
          the neural network.
        nn.init.xavier_uniform_(self.weight)
        # Rest of the methods...
```

DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz



# Graph Convolutional Networks (GCN)

01 __init__()

02 forward()

## Manual Implementing of the GCN Layer:

```
# …Import necessary libraries
class GCNLayer(nn.Module):
    def forward(self, x, adjacency):
        # Compute symmetric normalization
        # Calculate the degree of each node
        degree = torch.sum(adjacency, dim=1)
        # Calculate the reciprocal square root of the degree
        degree_sqrt_inv = 1.0 / torch.sqrt(degree)
        # Create a diagonal matrix with the degree_sqrt_inv
        D_sqrt_inv = torch.diag(degree_sqrt_inv)
        # Apply symmetric normalization to the adjacency matrix
        adjacency = torch.mm(torch.mm(D_sqrt_inv, adjacency),
                             D_sqrt_inv)
        # Compute the support (feature transformation)
        support = torch.mm(x, self.weight)
        # Perform the graph convolution using the normalized
          adjacency
        output = torch.spmm(adjacency, support)
        return output
```

DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz



# Graph Convolutional Networks (GCN)

01 Class Overview

02 __init__()

03 forward()

## Implementation the GCN:

```
# Create a new class named GCN
class GCN(nn.Module):
    def __init__(self, dim_in, dim_h, dim_out):
        # Initialize the GCN class with input, hidden, and
          output layer dimensions
    def forward(self, x):
        # Perform the forward pass of the GCN
    def accuracy(self, y_pred, y_true):
        # Calculate the accuracy of predictions
    def fit(self, data, epochs):
        # Train the model
    def test(self, data):
        # Evaluate the model
```

DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz



# Graph Convolutional Networks (GCN)

01 Class Overview

02 __init__()

03 forward()

## Implementation the GCN:

```
# Create a new class named GCN
class GCN(nn.Module):
    def __init__(self, dim_in, dim_h, dim_out):
         super().__init__()
         self.gcn1 = GCNLayer(dim_in, dim_h)
         self.gcn2 = GCNLayer(dim_h, dim_out)
```

## Implementation the GCN With the Built-In GCNConv Module:

```
from torch_geometric.nn import GCNConv
# Create a new class named GCN
class GCN(nn.Module):
    def __init__(self, dim_in, dim_h, dim_out):
         super().__init__()
         self.gcn1 = GCNConv(dim_in, dim_h)
         self.gcn2 = GCNConv(dim_h, dim_out)
```

DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz



# Graph Convolutional Networks (GCN)

01 Class Overview

02 __init__()

03 **forward()**

## Implementation the GCN:

```
# Create a new class named GCN
class GCN(nn.Module):
    def forward(self, x, adjacency):
         h = self.gcn1(x, adjacency)
         h = torch.relu(h)
         h = self.gcn2(h, adjacency)
         return F.log_softmax(h, dim=1)
```

DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz



# Graph Convolutional Networks (GCN)



## Building, Training, and Testing the GCN With the Cora Dataset:

```
# Create a GCN instance with specified input, hidden, and output dimensions
gnn = GCN(dataset.num_features, 16, dataset.num_classes)

# Print the model architecture
print(gcn)
```

```
GCN(
  (gcn1): GCNLayer()
  (gcn2): GCNLayer()
)
```

DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz



# Graph Convolutional Networks (GCN)

## Building, Training, and Testing the GCN With the Cora Dataset – Manual GCN Layer



```
# Train the GCN model on the given data for a specified number of epochs (100 in this case) and the adjacency matrix.
gcn.fit(data, epochs=100, adjacency=adjacency)
```

```
Epoch   0 | Train Loss: 2.762 | Train Acc: 20.00% | Val Loss: 2.77 | Val Acc: 12.00%
Epoch  20 | Train Loss: 0.931 | Train Acc: 82.86% | Val Loss: 1.53 | Val Acc: 55.80%
Epoch  40 | Train Loss: 0.187 | Train Acc: 100.00% | Val Loss: 0.89 | Val Acc: 76.60%
Epoch  60 | Train Loss: 0.055 | Train Acc: 100.00% | Val Loss: 0.77 | Val Acc: 76.20%
Epoch  80 | Train Loss: 0.038 | Train Acc: 100.00% | Val Loss: 0.76 | Val Acc: 76.00%
Epoch 100 | Train Loss: 0.034 | Train Acc: 100.00% | Val Loss: 0.76 | Val Acc: 76.60%
```

```
# Test the model and get accuracy
test_acc = gcn.test(data, adjacency=adjacency)
print(f'\nGCN test accuracy: {test_acc*100:.2f}%')
```

```
GCN test accuracy: 80.30%
```

DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz



# Graph Convolutional Networks (GCN)



## Building, Training, and Testing the GCN With the Cora Dataset – **Built-In GCNConv Module**

```
# Train the GCN model on the given data for a specified number of epochs (100 in this case) and the adjacency matrix.
gcn.fit(data, epochs=100, adjacency=adjacency)
```

```
Epoch   0 | Train Loss: 1.932 | Train Acc: 15.71% | Val Loss: 1.94 | Val Acc: 15.20%
Epoch  20 | Train Loss: 0.099 | Train Acc: 100.00% | Val Loss: 0.75 | Val Acc: 77.80%
Epoch  40 | Train Loss: 0.014 | Train Acc: 100.00% | Val Loss: 0.72 | Val Acc: 77.20%
Epoch  60 | Train Loss: 0.015 | Train Acc: 100.00% | Val Loss: 0.71 | Val Acc: 77.80%
Epoch  80 | Train Loss: 0.017 | Train Acc: 100.00% | Val Loss: 0.71 | Val Acc: 77.00%
Epoch 100 | Train Loss: 0.016 | Train Acc: 100.00% | Val Loss: 0.71 | Val Acc: 76.40%
```

```
# Test the model and get accuracy
test_acc = gcn.test(data, adjacency=adjacency)
print(f'\nGCN test accuracy: {test_acc*100:.2f}%')
```

```
GCN test accuracy: 79.70%
```

DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz



# Models Comparison

| Dataset | MLP | Vanilla GNN | GCN |
|---|---|---|---|
| Cora | 51.90% | 72.50%<br>+20.60% | 79.70%<br>+27.80% |

DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz



# Models Comparison

## Vanilla GNN:

- **Graph-Based Layers**: Employs graph-based layers for iterative node embedding updates.
- **Adjacency Matrix**: Incorporates topological information from the graph's adjacency matrix.
- **Message Passing**: Utilizes message passing to aggregate information from neighboring nodes.
- **Graph Structure**: Considers the entire neighborhood of each node, capturing graph structure.

## GCN (Graph Convolutional Network):

- **Graph-Based Layers**: Similar to Vanilla GNN, it employs graph-based layers for iterative node embedding updates.
- **Smart Normalization**: Improves upon Vanilla GNN by correctly normalizing features during message passing.
- **Message Passing**: Also utilizes message passing to aggregate information from neighboring nodes.
- **Graph Structure**: Like Vanilla GNN, it considers the entire neighborhood of each node, capturing graph structure.

DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz

# THANK YOU