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

Part I

Introduction and Fundamentals

Lecture 04

Nodes Classification with Vanilla Graph Neural Networks

Explore the foundational concepts of classifying nodes using Vanilla Graph Neural Networks (GNNs). Understand the principles behind GNNs, including message passing, graph structure, and the role of adjacency matrices. Gain hands-on experience implementing Vanilla GNNs, which capture the graph's topology, and see how this architecture outperforms traditional Multilayer Perceptrons (MLPs) in node classification tasks.

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

Graph Datasets Features

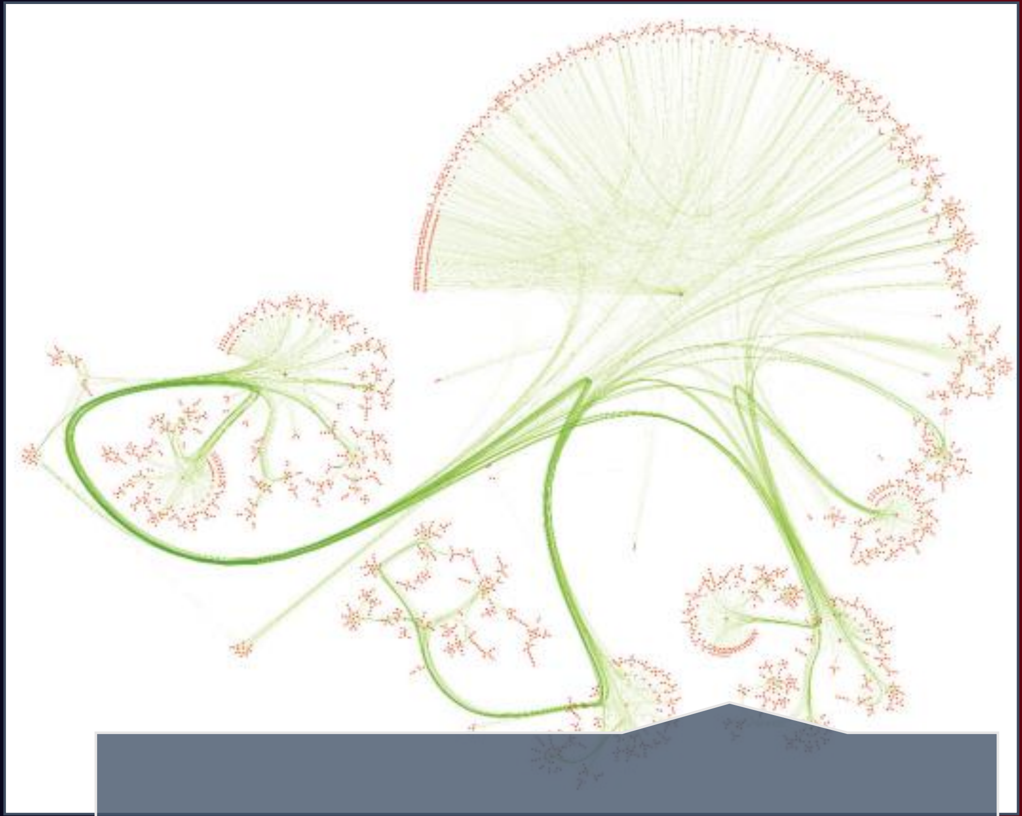
Graph Data Richness: Real-world graph datasets are often more intricate than simpler examples like Zachary's Karate Club, with larger numbers of nodes, edges, and additional features that can enhance analysis.

Diverse Information: These datasets go beyond just structural connections; they encompass various types of data, such as text, categories, and numerical attributes associated with nodes and edges.

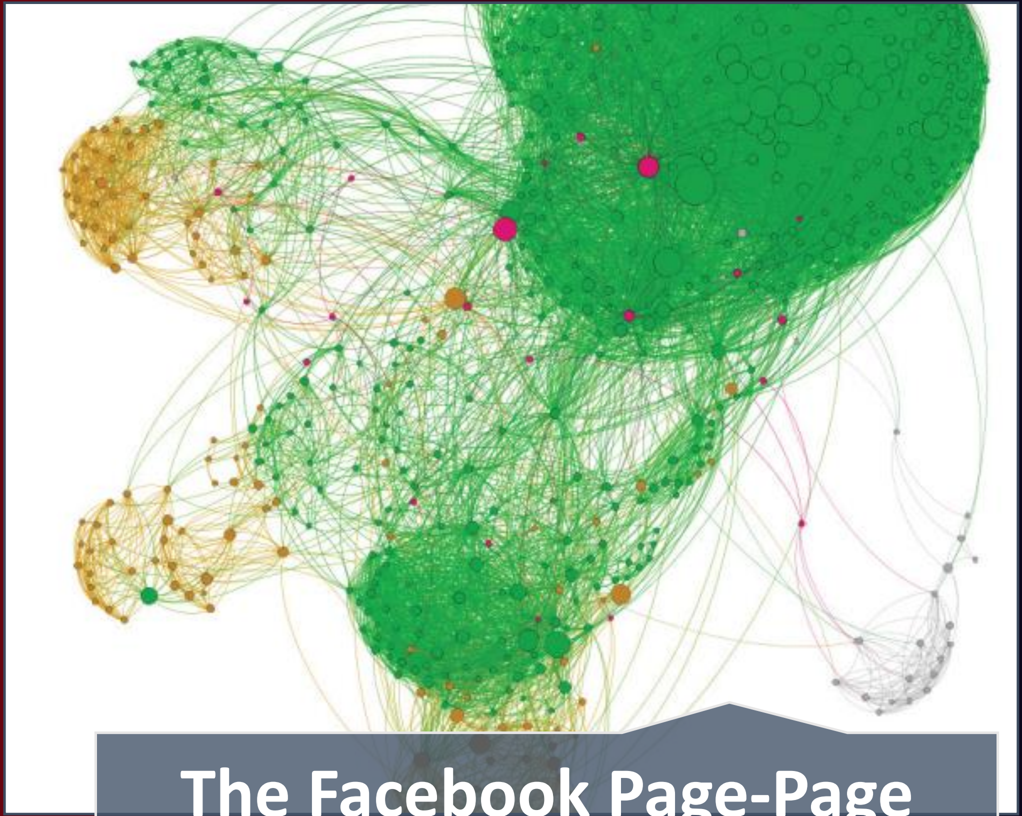
Essential for Analysis: Proficiency in handling complex graph datasets is vital for performing comprehensive graph analysis and applying machine learning approaches effectively.

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

Introducing the Selected Datasets



The Cora Dataset



The Facebook Page-Page dataset

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

The Cora Dataset

Cora Dataset Characteristics:

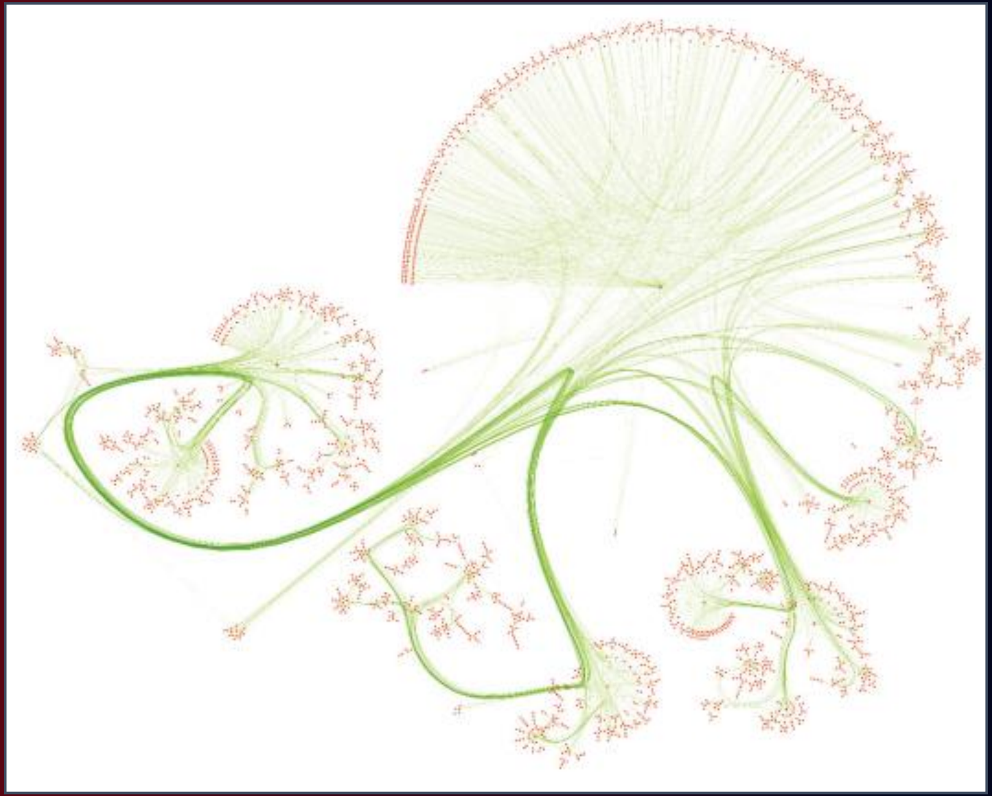
Publication Network: Cora represents a network of 2,708 scientific publications.

Classification Objective: The dataset aims to classify each publication into one of seven classes.

Citation Network: The citation network comprises 5,429 links connecting the publications.

Feature Representation: Each publication is described using a binary word vector, with 0 indicating word absence and 1 indicating word presence.

Word Dictionary: The dataset's dictionary includes 1,433 unique words.



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

The Cora Dataset Analysis with Pytorch Geometry

PyTorch Geometric Usage

Planetoid Class: **PyTorch Geometric** provides the Planetoid class for downloading and managing well-known graph datasets.

```
# Import necessary libraries
from torch_geometric.datasets import Planetoid
```

Dataset Download: It enables the straightforward download of the Cora dataset for analysis within **PyTorch Geometric**.

```
# Download the Cora dataset using Planetoid
dataset = Planetoid(root=".", name="Cora")
```

Data Structure: The dataset comprises a single graph stored within a dedicated data variable.

```
# Cora only has one graph
data = dataset[0]
```

Dataset Information Access: **PyTorch Geometric** can provide key information about the dataset, including the number of graphs, nodes, features, and classes.

```
# Print general information about the dataset
print(f'Dataset: {dataset}')
print('---------------')
print(f'Number of graphs: {len(dataset)}')
print(f'Number of nodes: {data.x.shape[0]}')
print(f'Number of feat.: {dataset.num_features}')
print(f'Number of classes: {dataset.num_classes}')
```

```
Dataset: Cora()
---------------
Number of graphs: 1
Number of nodes: 2708
Number of features: 1433
Number of classes: 7
```

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

The Cora Dataset Analysis with Pytorch Geometry

PyTorch Geometric Usage

Planetoid Class: **PyTorch Geometric** provides the Planetoid class for downloading and managing well-known graph datasets.

```
# Import necessary libraries
from torch_geometric.datasets import Planetoid
```

Dataset Download: It enables the straightforward download of the Cora dataset for analysis within **PyTorch Geometric**.

```
# Download the Cora dataset using Planetoid
dataset = Planetoid(root=".", name="Cora")
```

Data Structure: The dataset comprises a single graph stored within a dedicated data variable.

```
# Cora only has one graph
data = dataset[0]
```

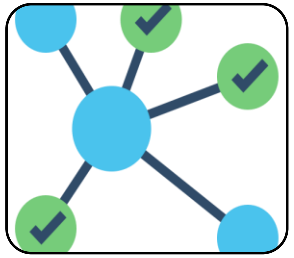
Additional Dataset Information: Using dedicated functions from **PyTorch Geometric**, detailed information about the graph's properties might be accessed. For example, the code checks if the edges are directed, if there are isolated nodes, and if the graph has loops.

```
# Check various properties of the graph
print(f'Graph:')
print('------')
print(f'Edges are directed: {data.is_directed()}')
print(f'Graph has isolated nodes:
{data.has_isolated_nodes()}')
print(f'Graph has loops: {data.has_self_loops()}')
```

```
Graph:
------
Edges are directed: False
Graph has isolated nodes: False
Graph has loops: False
```

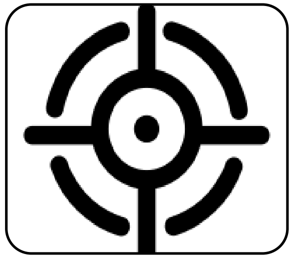
DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz

Classifying Nodes with Neural Networks



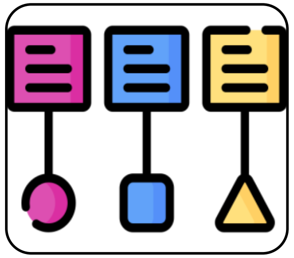
Node Features:

- Additional attributes associated with nodes in a graph. These features can represent a wide range of information, including user profiles, numerical values, text data, or any relevant attributes.



Purpose:

- These node features provide additional information about nodes, enhancing the potential for downstream tasks.



Node Classification:

- In a neural network context, node features are considered as a regular tabular dataset.

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


Classifying Nodes with Neural Networks

01 Node Features in Graphs

02 Overview of a Vanilla Neural Network (VNN)

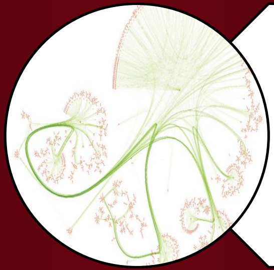
03 Implementing a Multilayer Perceptron (MLP)

04 Model Training

05 Model Evaluation

06 Testing on Cora Dataset

For the Cora Dataset Graphs

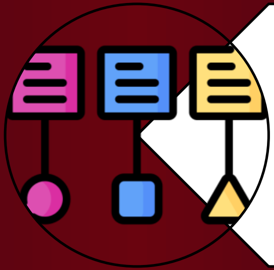


Node Features:

- Each publication in the **Cora dataset** is described as a binary vector of 1,433 unique words.

Binary Bag of Words:

- The binary vector uses values of **0** and **1** to indicate the absence or presence of specific words.



Classification Goal:

- The objective is to classify each **node** (**publication**) into one of the seven predefined **categories**.

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

Classifying Nodes with Neural Networks

Definition:

- A **VNN**, also known as a Multilayer Perceptron (**MLP**), is a traditional neural network that consists of multiple layers, including an input layer, one or more hidden layers, and an output layer.

Use of Node Features:

- We consider node features in the **Cora dataset** as a **regular tabular** dataset.

Node Classification:

- We employ a simple **VNN** to train and classify nodes using the node features.

Model Architecture:

- The VNN does not incorporate the network topology.

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

Classifying Nodes with Neural Networks

Use of Node Features: Preparing Data

```
# Import necessary libraries
import pandas as pd
# Create a DataFrame from the node features and label
# Convert PyTorch tensor data.x to a NumPy array and create a
DataFrame
df_x = pd.DataFrame(data.x.numpy())
# Add a 'label' column to the DataFrame containing node labels
df_x['label'] = pd.DataFrame(data.y)
df_x
```

	0	1	2	3	4	5	6	7	8	9	...	1424	1425	1426	1427	1428	1429	1430	1431	1432	label
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	3
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	4
2	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	4
3	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
4	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	3
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...

Node 1 Node Features Class Label

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

Classifying Nodes with Neural Networks

01 Node Features in Graphs

02 Overview of a Vanilla Neural Network (VNN)

03 Implementing a Multilayer Perceptron (MLP)

04 Model Training

05 Model Evaluation

06 Testing on Cora Dataset

Overview:

- We'll create an MLP class with key methods for initialization, forward pass, training, and evaluation.

Forward Pass:

- The forward method processes node features through linear layers and a softmax function for classification.

Metric:

- We'll use **accuracy** as a metric to measure the model's performance.

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

Classifying Nodes with Neural Networks

Overview of MLP Class

```python
# Import necessary libraries
import pandas as pd
import torch
from torch.nn import Linear
import torch.nn.functional as F

# Create a new class named MLP for our Multilayer Perceptron
class MLP(torch.nn.Module):
    def __init__(self, dim_in, dim_h, dim_out):
        # Initialize the MLP class with input, hidden, and
          output layer dimensions
    def forward(self, x):
        # Perform the forward pass of the MLP
    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

Classifying Nodes with Neural Networks

Overview of MLP Class: **Init() and forward()**

```python
# Create a new class named MLP for our Multilayer Perceptron
class MLP(torch.nn.Module):
    # Initialize the MLP class with input, hidden, and output
      layer dimensions
    def __init__(self, dim_in, dim_h, dim_out):
        super().__init__()
        self.linear1 = Linear(dim_in, dim_h)
        self.linear2 = Linear(dim_h, dim_out)

    # Perform the forward pass of the MLP
    def forward(self, x):
        x = self.linear1(x)  # Apply the first linear layer
        x = torch.relu(x)    # Apply a Rectified Linear Unit
                               (ReLU) activation function
        x = self.linear2(x)  # Apply the second linear layer
        return F.log_softmax(x, dim=1)  # Return log-softmax
                                    of the result for classification
```

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

Classifying Nodes with Neural Networks

Overview of MLP Class: **accuracy()**

```python
# Create a new class named MLP for our Multilayer Perceptron
class MLP(torch.nn.Module):
    # Continued
    # Definning the metric
    def accuracy(self, y_pred, y_true):
        # Calculate the accuracy of predictions
        return torch.sum(y_pred == y_true) / len(y_true)
```

The **accuracy** function counts the number of correct predictions in the batch and divides it by the total number of data points to determine the accuracy of the model.

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

Classifying Nodes with Neural Networks

Model Training : fit()

```
# Create a new class named MLP for our Multilayer Perceptron
class MLP(torch.nn.Module):
    # Continued
    # Train the model
    def fit(self, data, epochs):
        # Define the loss function
        criterion = torch.nn.CrossEntropyLoss()
        # Initialize the optimizer
        optimizer = torch.optim.Adam(self.parameters(),
                                     lr=0.01,
                                     weight_decay=5e-4)
        # Set the model to training mode
        self.train()
```

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

Classifying Nodes with Neural Networks

Model Training : fit() --Continued

```python
# Create a new class named MLP for our Multilayer Perceptron
class MLP(torch.nn.Module):
    # Train the model
    def fit(self, data, epochs):
        # Training loop for the specified number of epochs
        for epoch in range(epochs+1):
            # Zero out gradient prepare for a new iteration
            optimizer.zero_grad()
            # Perform forward pass to get predictions
            out = self(data.x)
            # Calculate the loss
            loss = criterion(out[data.train_mask],
                             data.y[data.train_mask])
            # Calculate accuracy
            acc =
            self.accuracy(out[data.train_mask].argmax(dim=1),
                          data.y[data.train_mask])
            # Computes gradients of the loss
            loss.backward()
            # Adjusts the model's params (weights and biases)
            optimizer.step()
```

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

Classifying Nodes with Neural Networks

Model Training: fit() --Continued

```
# Create a new class named MLP for our Multilayer Perceptron
class MLP(torch.nn.Module):
    # Train the model
    def fit(self, data, epochs):
        # Training loop for the specified number of epochs
        for epoch in range(epochs+1):
            # Continued
            # Print loss and accur for train data every 20 epochs
            if epoch % 20 == 0:
                val_loss = criterion(out[data.val_mask],
                                     data.y[data.val_mask])
                # Calculate validation loss
                val_acc=
                    self.accuracy(out[data.val_mask].argmax(dim=1),
                                  data.y[data.val_mask])
                # Calculate validation accuracy
                print(f'Epoch {epoch:>3} | Train Loss: {loss:.3f} |
                      Train Acc: {acc*100:>5.2f}% | Val Loss:
                      {val_loss:.2f} | Val Acc: {val_acc*100:.2f}%')
```

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

Classifying Nodes with Neural Networks

Model Evaluation: **test()**

```
# Create a new class named MLP for our Multilayer Perceptron
class MLP(torch.nn.Module):
    # Evaluate the model
    def test(self, data):
        # Set the model to evaluation mode
        self.eval()
        # Calculate model predictions (forward pass) for the
          given data.
        out = self(data.x)
        # Calculate accuracy for test data
        acc = self.accuracy(out.argmax(dim=1)[data.test_mask],
                            data.y[data.test_mask])
        return acc
```

The accuracy is computed by comparing the predicted class labels (based on the maximum value along dimension 1 (**Argmax for Prediction**), which selects the class with the highest probability) to the true class labels in the test data.

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

Classifying Nodes with Neural Networks

Model Testing on Cora Dataset:

```
# Create and train the MLP
# Create an instance of the MLP class.
# It takes three arguments:
# - dataset.num_features: Number of input features (input
dimension)
# - 16: Number of hidden units (you can adjust this as needed)
# - dataset.num_classes: Number of output classes (output
dimension)
mlp = MLP(dataset.num_features, 16, dataset.num_classes)
print(mlp)
```

```
MLP(
  (linear1): Linear(in_features=1433, out_features=16, bias=True)
  (linear2): Linear(in_features=16, out_features=7, bias=True)
)
```

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

Classifying Nodes with Neural Networks

Model Testing on Cora Dataset:

```python
# Train the MLP model on the given data for a specified number of epochs (100 in this case).
mlp.fit(data, epochs=100)
```

```
Epoch   0 | Train Loss: 1.953 | Train Acc: 15.00% | Val Loss: 2.02 | Val Acc: 7.00%
Epoch  20 | Train Loss: 0.118 | Train Acc: 100.00% | Val Loss: 1.48 | Val Acc: 50.60%
Epoch  40 | Train Loss: 0.014 | Train Acc: 100.00% | Val Loss: 1.61 | Val Acc: 49.80%
Epoch  60 | Train Loss: 0.008 | Train Acc: 100.00% | Val Loss: 1.58 | Val Acc: 50.40%
Epoch  80 | Train Loss: 0.009 | Train Acc: 100.00% | Val Loss: 1.47 | Val Acc: 51.00%
Epoch 100 | Train Loss: 0.009 | Train Acc: 100.00% | Val Loss: 1.41 | Val Acc: 51.00%
```

```python
# Test the model and get accuracy
test_acc = mlp.test(data)
print(f'MLP test accuracy: {test_acc*100:.2f}%')
```

```
MLP test accuracy: 51.90%
```

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

Classifying Nodes with Vanilla GNNs

Overview of Vanilla GNN:

Definition-- Vanilla GNN:

- A foundational type of **Graph Neural Network (GNN)** used to analyze graph-structured data.
- Incorporates both **node features** and **topological information** in graph analysis.

Role of Vanilla GNNs:

- Addresses **the complexity of graph datasets** by considering **node features**.
- Perform **node embeddings** through a process of **message passing**.

Key Features:

- **Linear Transformation:** Utilizes a **linear layer** to capture **node relationships**.
- **Information Aggregation:** Collects information from both the node and its neighbors (**message passing**).

Thought Process:

- Foundation for understanding **advanced GNN architectures**.

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

Classifying Nodes with Vanilla GNNs

Overview of Vanilla GNN:

Fundamental Concepts of Vanilla GNN

Local Transition Function (Message Passing):

- Shared function, ψ, for nodes to update their states using local and neighbor information (**Aggregation**).
- The function is used to update the representation of a node in the graph.

$$\psi_{W}\left(h_i^{(k)}, h_{\aleph_i}^{(k)}\right) = h_i^{(k+1)}$$

Learned Parameters

New Hidden State of node v_i

Actual Hidden State of node v_i

Actual Hidden States of Neighbors

Scarselli, F., Gori, M., Tsoi, A., Hagenbuchner, M. & Monfardini, G. 2009, 'The graph neural network model', IEEE Transactions on Neural Networks, vol. 20, no. 1, pp. 61-80 1.

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

Classifying Nodes with Vanilla GNNs

Overview of Vanilla GNN:

Fundamental Concepts of Vanilla GNN

Global Transition Function (Pooling):

- An efficient layer that updates all **node states** at once using the entire **state vector** and **adjacency matrix**.

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

W: Learned Parameters

A: Adjacency Matrix

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

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

Scarselli, F., Gori, M., Tsoi, A., Hagenbuchner, M. & Monfardini, G. 2009, 'The graph neural network model', IEEE Transactions on Neural Networks, vol. 20, no. 1, pp. 61-80 1.

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

Classifying Nodes with Vanilla GNNs

Implementing the Vanilla GNN Layer:

```python
# Import necessary libraries
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.nn import Linear

class VanillaGNNLayer(nn.Module):
    def __init__(self, dim_in, dim_out):
        super(VanillaGNNLayer, self).__init__()
        # Initialize a linear transformation layer without bias
        self.linear = Linear(dim_in, dim_out, bias=False)

    def forward(self, x, adjacency):
        # Apply the linear transformation to the input node
          features
        x = self.linear(x)
        # Perform a sparse matrix-vector multiplication with the
          adjacency matrix
        x = torch.sparse.mm(adjacency, x)
        return x
```

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

Classifying Nodes with Vanilla GNNs

Implementing the Vanilla GNN:

```
# Create a new class named VanillaGNN for our GNN
class VanillaGNN(nn.Module):
    def __init__(self, dim_in, dim_h, dim_out):
        # Initialize the VGNN class with input, hidden, and
          output layer dimensions
    def forward(self, x):
        # Perform the forward pass of the VGNN
    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

Classifying Nodes with Vanilla GNNs

Implementing the Vanilla GNN -- **Init() and forward() :**

```python
# Create a new class named VanillaGNN for our GNN
class VanillaGNN(nn.Module):
    # Initialize the VanillaGNN class with input, hidden, and
      output layer dimensions
    def __init__(self, dim_in, dim_h, dim_out):
        super().__init__()
        # Initialize 2 VanillaGNNLayer instances for the GNN
          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):
        h = self.gnn1(x, adjacency)  # Apply the 1st GNN layer
        h = torch.relu(h)  # Apply ReLU activation
        h = self.gnn2(h, adjacency)  # Apply the 2nd GNN layer
        return F.log_softmax(h, dim=1)  # Log softmax for
                                          classification
```

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

Classifying Nodes with Vanilla GNNs

Implementing the Vanilla GNN – **accuracy():**

```
# Create a new class named VanillaGNN for our GNN
class VanillaGNN(nn.Module):
    # Continued
    # Definning the metric
    def accuracy(self, y_pred, y_true):
        # Calculate the accuracy of classification
        return torch.sum(y_pred == y_true) / len(y_true)
```

The **accuracy** function counts the number of correct predictions in the batch and divides it by the total number of data points to determine the accuracy of the model.

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

Classifying Nodes with Vanilla GNNs

Implementing the Vanilla GNN – fit():

```python
# Create a new class named VanillaGNN for our GNN
class VanillaGNN(nn.Module):
    # Continued
    # Train the model
    def fit(self, data, epochs, adjacency):
        # Define the loss function
        criterion = torch.nn.CrossEntropyLoss()
        # Initialize the optimizer
        optimizer = torch.optim.Adam(self.parameters(),
                                     lr=0.01,
                                     weight_decay=5e-4)
        # Set the model to training mode
        self.train()
```

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

Classifying Nodes with Vanilla GNNs

Implementing the Vanilla GNN – fit():

```
class VanillaGNN(nn.Module):
    # Continued
    # Train the model
    def fit(self, data, epochs, adjacency):
         # Conitnued
         # Training loop for the specified number of epochs
         for epoch in range(epochs + 1):
            # Zero out gradients to prepare for a new iteration
            optimizer.zero_grad()
           # Perform forward pass to get predictions
            out = self(data.x, adjacency)
            # Calculate the loss for the training set
            loss = criterion(out[data.train_mask],
                             data.y[data.train_mask])
            # Perform backpropagation to compute gradients
            loss.backward()
            # Update model parameters using the optimizer
            optimizer.step()
```

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

Classifying Nodes with Vanilla GNNs

Implementing the Vanilla GNN – fit():

```
class VanillaGNN(nn.Module):
    # Train the model
    def fit(self, data, epochs, adjacenc):
        # Training loop for the specified number of epochs
        for epoch in range(epochs+1):
           # Continued
            # Print loss and accur for train data every 20 epochs
            if epoch % 20 == 0:
              # Calculate validation loss
              val_loss = criterion(out[data.val_mask],
                                   data.y[data.val_mask])
              # Calculate validation accuracy
              val_acc =
                     self.accuracy(out[data.val_mask].argmax(dim=1),
                                  data.y[data.val_mask])
              print(f'Epoch {epoch:>3} | Train Loss: {loss:.3f} |
                      Train Acc: {acc*100:>5.2f}% | Val Loss:
                      {val_loss:.2f} | Val Acc: {val_acc*100:.2f}%')
```

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

Classifying Nodes with Vanilla GNNs

Implementing the Vanilla GNN – **test():**

```
class VanillaGNN(nn.Module):
    # Evaluate the model
    def test(self, data, adjacency):
        # Set the model to evaluation mode
        self.eval()
        # Calculate model predictions (forward pass) for the
          given data and adjacency matrix.
        out = self(data.x, adjacency)
        # Calculate accuracy for test data
        acc = self.accuracy(out.argmax(dim=1)[data.test_mask],
                            data.y[data.test_mask])
        return acc
```

The accuracy is computed by comparing the predicted class labels (based on the maximum value along dimension 1 (**Argmax for Prediction**), which selects the class with the highest probability) to the true class labels in the test data.

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

Classifying Nodes with Vanilla GNNs

Preparing the Adjacency Matrix:

```
from torch_geometric.utils import to_dense_adj

# Convert the edge index to a dense adjacency matrix
adjacency = to_dense_adj(data.edge_index)[0]

# Add self-loops to the adjacency matrix
adjacency += torch.eye(len(adjacency))

# Resulting adjacency matrix
adjacency
```

- **Self-Loops In Adjacency:** Self-loops are included in the adjacency matrix.
- **Information Consideration:** Ensures that every node's information is included.
- **Message Passing Enhancement:** Self-loops enhance the accuracy of message passing

- **Common Practice:** Converting edge indices to dense adjacency matrices is standard in GNNs.
- **Efficient Message Passing:** Facilitates efficient message propagation between nodes.
- **Consistency:** Provides a consistent graph representation for various topologies.

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

Classifying Nodes with Vanilla GNNs

Building, Training, and Testing the Vanilla GNN:

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

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

```
VanillaGNN(
  (gnn1): VanillaGNNLayer(
    (linear): Linear(in_features=1433, out_features=16, bias=False)
  )
  (gnn2): VanillaGNNLayer(
    (linear): Linear(in_features=16, out_features=7, bias=False)
  )
)
```

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

Classifying Nodes with Vanilla GNNs

Building, Training, and Testing the Vanilla GNN:

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

```
Epoch   0 | Train Loss: 2.227 | Train Acc: 74.70% | Val Loss: 2.10 | Val Acc: 20.00%
Epoch  20 | Train Loss: 0.058 | Train Acc: 74.70% | Val Loss: 1.60 | Val Acc: 74.40%
Epoch  40 | Train Loss: 0.008 | Train Acc: 74.70% | Val Loss: 2.23 | Val Acc: 73.20%
Epoch  60 | Train Loss: 0.003 | Train Acc: 74.70% | Val Loss: 2.46 | Val Acc: 71.80%
Epoch  80 | Train Loss: 0.002 | Train Acc: 74.70% | Val Loss: 2.40 | Val Acc: 71.20%
Epoch 100 | Train Loss: 0.002 | Train Acc: 74.70% | Val Loss: 2.29 | Val Acc: 71.80%
```

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

```
GNN test accuracy: 72.50%
```

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

Models Comparison

Dataset	MLP	Vanilla GNN
Cora	51.90%	72.50% +39.38%

MLP has poor accuracy on Cora.

Vanilla GNN surpasses the MLP in performance.

- Including topological information in node features is crucial for accuracy.
- GNN considers the entire neighborhood of each node.
- This leads to a **39.38%** boost in accuracy compared to a tabular dataset.
- The architecture, while basic, provides a guideline for building improved models.

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

Models Comparison

MLP (Multilayer Perceptron):

- **Feedforward Neural Network**: Applies a simple feedforward neural network to input data.
- **Fully Connected Layers**: Uses fully connected layers, not specific to graph topology.
- **No Message Passing**: Does not incorporate neighbor information during training.
- **Tabular Data**: Treats node features as tabular data, ignoring graph structure.

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.

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

THANK YOU