GNNs stand out because they can integrate the embedding process directly into the learning algorithm itself... As the network processes inputs through its layers, the embeddings are refined and updated, making the learning phase and the embedding phase inseparable. This means that GNNs learn the most informative representation of the graph data during training time.

```python
import NetworkX as nx
from Node2Vec import Node2Vec
books_graph = nx.read_gml('PATH_TO_GML_FILE')
node2vec = Node2Vec(books_graph, dimensions=64,
 walk_length=30, num_walks=200, workers=4)
model = node2vec.fit(window=10, min_count=1,\
batch_words=4)
embeddings = {str(node): model.wv[str(node)]\
 for node in gml_graph.nodes()}
```
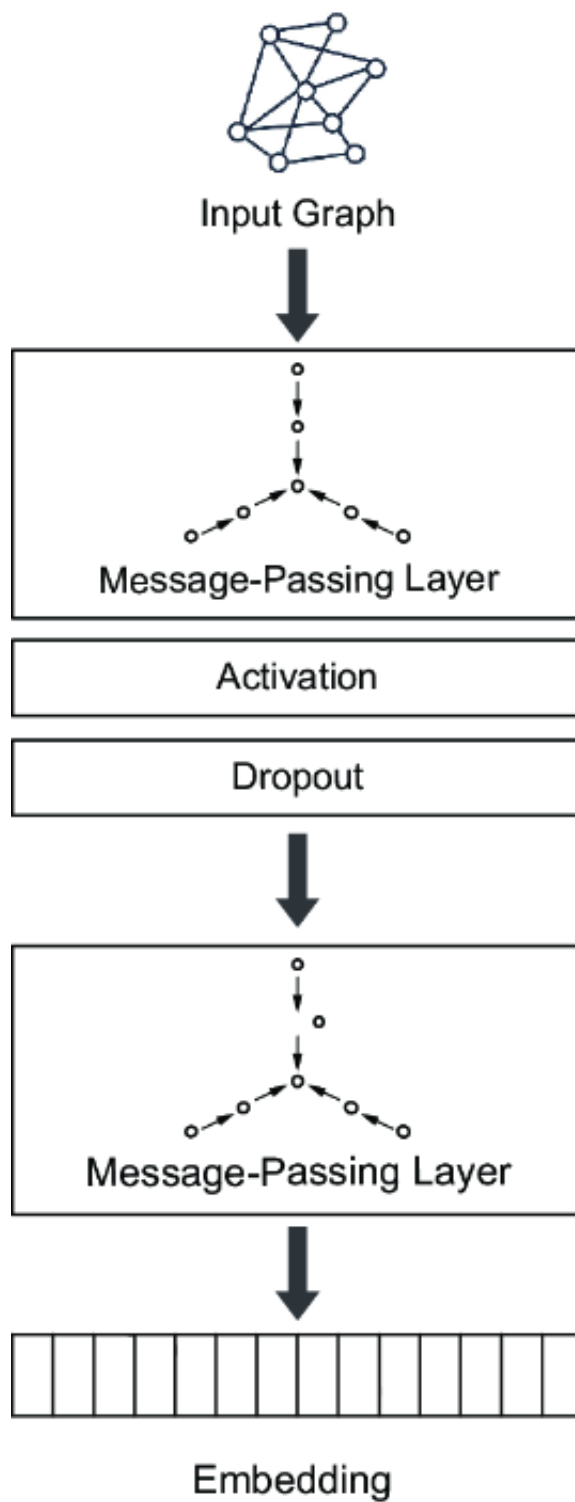
```python
 node_embedding = model.wv['Losing Bin Laden']
 print(node_embedding)
```

```python
node_embeddings = [embeddings[str(node)] \
for node in gml_graph.nodes()]
node_embeddings_array = np.array(node_embeddings)

umap_model = umap.UMAP(n_neighbors=15, min_dist=0.1,
n_components=2, \
random_state=42)
umap_features = umap_model.fit_transform\
(node_embeddings_array)

plt.scatter(umap_features[:, 0], \
umap_features[:, 1], color=node_colors, alpha=0.7)
```

Our SimpleGNN class inherits from torch.nn.Module and is composed of two GCNConv layers, which are the building blocks of our GNN. This architecture is shown in figure 2.7, consisting of the first layer, a message passing layer (self.conv1), an activation (torch.relu), a dropout layer (torch.dropout), and a second message passing layer.

Input Graph

Message-Passing Layer

Activation

Dropout

Message-Passing Layer

Embedding

```
import torch
```

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

class SimpleGNN_embeddings(torch.nn.Module):
    def __init__(self, num_features, hidden_channels):
        super(SimpleGNN_embeddings, self).__init__()
        self.conv1 = GCNConv(num_features, hidden_channels)
        self.conv2 = GCNConv(hidden_channels, hidden_channels)


    def forward(self, x, edge_index):
        x = self.conv1(x, edge_index)
        x = torch.relu(x)
        x = F.dropout(x, p=0.5, training=self.training)
        x = self.conv2(x, edge_index)
        return x
```

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class SimpleMLP(nn.Module):
    def __init__(self, in_dim, hidden=64, out_dim=NUM_CLASSES):
        super().__init__()
        self.fc1 = nn.Linear(in_dim, hidden)
        self.fc2 = nn.Linear(hidden, hidden)
        self.fc3 = nn.Linear(hidden, out_dim)
        self.dropout = nn.Dropout(p=0.5)

    def forward(self, x):
        # x: [num_nodes, in_dim]  (or [batch, in_dim] for non-graph data)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
        x = self.fc3(x)      # logits
        return x
```

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

class SimpleGCN(nn.Module):
    def __init__(self, in_dim, hidden=64, out_dim=NUM_CLASSES):
        super().__init__()
        self.conv1 = GCNConv(in_dim, hidden)
        self.conv2 = GCNConv(hidden, hidden)
        self.conv3 = GCNConv(hidden, out_dim)
        self.dropout = nn.Dropout(p=0.5)

    def forward(self, x, edge_index):
        # x: [num_nodes, in_dim]
        # edge_index: [2, num_edges] (COO format)
        x = F.relu(self.conv1(x, edge_index))
        x = self.dropout(x)
        x = F.relu(self.conv2(x, edge_index))
        x = self.conv3(x, edge_index)   # logits
        return x
```

GCNs act as message-passing layers that are critical in constructing embeddings.

```
data = from_NetworkX(gml_graph)
```

Method from_NetworkX specifically translates the edge lists and node/edge attributes into PyTorch tensors.

When no node features are available or they aren't informative,.... Xavier initialization, which sets the initial node features with values drawn from a distribution that keeps the variety of activations consistent across layers.

This technique ensures that the model starts with a balanced representation, preventing problems such as vanishing or exploding gradients.

```
data.x = torch.randn((data.num_nodes, 64), dtype=torch.float)
'nn.init.xavier_uniform_(data.x) '
```

```
model = SimpleGNN(num_features=data.x.shape[1],
hidden_channels=64)
```

Specifically, model.eval() turns off certain behaviors specific to training, such as dropout, which randomly deactivates some neurons to prevent overfitting, and batch normalization, which normalizes inputs across a mini-batch. By disabling these features, the model provides consistent and deterministic outputs, ensuring that the evaluation accurately reflects its true performance on unseen data.

we employ torch.no_grad(), which ensures that the computational graph that records operations for backpropagation isn't constructed, preventing us from accidentally changing performance.

```
model.eval()
with torch.no_grad():
    gnn_embeddings = model(data.x, data.edge_index)
```

```
gnn_embeddings_np = gnn_embeddings.detach().cpu().numpy()
```

Semi-supervised learning

```
labels = []
for node, data in gml_graph.nodes(data=True):
    if data['value'] == 'c':
        labels.append('right')
    elif data['value'] == 'l':
        labels.append('left')
    else:
        labels.append('neutral')
labels = np.array(labels)

random.seed(52)

indices = list(range(len(labels)))
```

```
labelled_percentage = 0.2

labelled_indices = random.sample(indices, \
int(labelled_percentage * len(labels)))

labelled_mask = np.zeros(len(labels), dtype=bool)
unlabelled_mask = np.ones(len(labels), dtype=bool)

labelled_mask[labelled_indices] = True
unlabelled_mask[labelled_indices] = False

labelled_labels = labels[labelled_mask]
unlabelled_labels = labels[unlabelled_mask]

label_mapping = {'left': 0, 'right': 1, 'neutral': 2}
numeric_labels = np.array([label_mapping[label] for label in
labels])
```

```
X_train_gnn = gnn_embeddings[labelled_mask]
Y_train_gnn = numeric_labels[labelled_mask]

X_n2v = np.array([embeddings[str(node)] \
for node in gml_graph.nodes()])
X_train_n2v = X_n2v[labelled_mask]
y_train_n2v = numeric_labels[labelled_mask]
```

Random Forest

```
clf_gnn = RandomForestClassifier()
clf_gnn.fit(X_train_gnn, y_train_gnn)

clf_n2v = RandomForestClassifier()
clf_n2v.fit(X_train_n2v, y_train_n2v)
```

Emb	Acc	F1
GNN	83%	82%

| N2V | 85% | 81% |

End to End

```python
class SimpleGNN_inference(torch.nn.Module):
    def __init__(self, num_features, hidden_channels):
        super(SimpleGNN, self).__init__()
        self.conv1 = GCNConv(num_features, hidden_channels)
        self.conv2 = GCNConv(hidden_channels, hidden_channels)

    def forward(self, x, edge_index):
        # First Graph Convolutional layer
        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x = F.dropout(x, training=self.training)

        # Second Graph Convolutional layer
        x = self.conv2(x, edge_index)
        predictions = F.log_softmax(x, dim=1)

        return x, predictions
```

```python
for epoch in range(3000):
    optimizer.zero_grad()

    _, out = model(data.x, data.edge_index)

    out_masked = out[data.train_mask]

    loss = loss_fn(out_masked, train_labels)
    loss.backward()
    optimizer.step()

    if epoch % 10 == 0:
        print(f'Epoch {epoch}, Log Loss: {loss.item()}')
```

Model	GNN Accuracy	GNN F1 Score
Two-layer, randomized features	82.27%	82.14%
Two-layer, N2V features	87.79%	88.10%
Four-layer, randomized features	86.58%	86.90%
Four-layer, N2V features	88.99%	89.29%

Model	Data Input	Accuracy	F1 Score
Random forest	Embedding from GNN	83.33%	82.01%
Random forest	Embedding from N2V	84.52%	80.72%
Two-layer simple GNN	Graph with randomized node features	82.27%	82.14%
Two-layer simple GNN	Graph with n2v embeddings as node features	87.79%	88.10%
Four-layer simple GNN	Graph with randomized node features	86.58%	86.90%
Four-layer simple GNN	Graph with n2v embeddings as node features	88.99%	89.29%