Final Report: CSE 881

Team Overview

Team Members and Roles

1. **Varuntej Kodandapuram** - Specialized in the model development, including defining the GCN model architecture, implementing the forward propagation logic and helped in training epochs.
2. **Shreyas Srinivas Bikumalla** - Focused on data preprocessing and setting up the environment, including handling the adjacency matrix, node features, and labels from external files also managed the report generation.
3. **Keerthi Gogineni** - Handled the training process, setting up the training-validation split, training loop, and optimization parameters. Also responsible for monitoring the training progress and logging.

Technical Approach and Code Explanation

Data Preparation and Loading

The project employs a Graph Convolutional Network (GCN) using PyTorch and PyTorch Geometric libraries to handle graph-structured data. This approach is beneficial as it allows the incorporation of relational information among data points.

- **Adjacency Matrix (adj.npz**): Represents the graph structure, crucial for modelling node relationships. The file containing the adjacency matrix is loaded in a compressed format designed to hold sparse matrices. The network structure with nodes connected by edges is shown by this matrix. The matrix's entries each indicate whether two nodes are connected or not.
- **Node Features (features.npy**): Attributes of each node, used as input features for the models. It contains which contains numerical data for each node in the graph.
- **Labels (labels.npy):** Target classification labels for supervised learning.
- **Training/Testing Splits (splits.json):** Defined the division of data into training and testing sets to evaluate model performance accurately.

Code Snippet: -

```
import torch
from torch_geometric.data import Data
import numpy as np
import scipy.sparse as sp
from torch_geometric.nn import GCNConv
import torch.nn.functional as F
import json
from torch_geometric.utils import dense_to_sparse

adj_matrix = sp.load_npz('./adj.npz')
features = np.load('./features.npy')
labels = np.load('./labels.npy')
splits = json.load(open('./splits.json'))
train_idx, test_idx = splits['idx_train'], splits['idx_test']

features = torch.FloatTensor(features)
labels = torch.LongTensor(labels)

full_labels = -1 * torch.ones(size=(features.shape[0],), dtype=torch.int64)
full_labels[train_idx] = labels

edge_index, _ = dense_to_sparse(torch.Tensor(adj_matrix.toarray()))

data = Data(x=features, edge_index=edge_index, y=full_labels)

data.train_mask = torch.zeros(data.num_nodes, dtype=torch.bool)
data.test_mask = torch.zeros(data.num_nodes, dtype=torch.bool)
data.val_mask = torch.zeros(data.num_nodes, dtype=torch.bool)

num_train = int(len(train_idx) * 0.75)

train_indices = train_idx[:num_train]
val_indices = train_idx[num_train:]

data.train_mask[train_indices] = True
data.val_mask[val_indices] = True
data.test_mask[test_idx] = True

num_classes = (data.y.max() + 1).item()

class GCN(torch.nn.Module):
    def __init__(self):
        super(GCN, self).__init__()
        self.conv1 = GCNConv(data.num_node_features, 128)
        self.conv2 = GCNConv(128, 128)
        self.conv3 = GCNConv(128, num_classes)
```

Data conversion

We used PyTorch, the standard data structure for storing multi-dimensional data, the feature and label matrices are transformed into PyTorch tensors. During model testing and training, this conversion makes computation more efficient.

The features are changed to torch. FloatTensor for the gradient operations that call for floating-point precision,

Labels are converted to torch.LongTensor, suitable for classification tasks where labels are typically integers representing different classes.

Data objects

The data is encapsulated using a Data object from the PyTorch Geometric library. This object contains the features (x), edge index (edge_index), and labels (y), as well as extra masks indicating which nodes belong to the training, validation, and testing sets.

The masks are important during the training and evaluation stages because they ensure that the model only trains on the training nodes and assesses on the right sets of nodes, leaving the testing nodes unnoticed during training.

- **train_mask**: A Boolean tensor that specifies which nodes are included in the training dataset.
- **val_mask**: A Boolean tensor representing nodes in the validation set. The training indices can be used to determine the division between training and validation nodes, which are usually further split.
- **test_mask**: A Boolean tensor that specifies the nodes utilized to test the model.

Code Snippet: -

```
data = Data(x=features, edge_index=edge_index, y=full_labels)

data.train_mask = torch.zeros(data.num_nodes, dtype=torch.bool)
data.test_mask = torch.zeros(data.num_nodes, dtype=torch.bool)
data.val_mask = torch.zeros(data.num_nodes, dtype=torch.bool)

num_train = int(len(train_idx) * 0.75)

train_indices = train_idx[:num_train]
val_indices = train_idx[num_train:]

data.train_mask[train_indices] = True
data.val_mask[val_indices] = True
data.test_mask[test_idx] = True

num_classes = (data.y.max() + 1).item()
```

Model Definition

• **First Layer:** This layer applies an initial transformation to a higher-dimensional space using the raw features of each node. The graph structure serves as a guide for the transformation; thus, a node's final features depend on both its own and its neighbours' features. The model gains non-linearity via the ReLU activation function, which aids in the learning of increasingly intricate patterns.

• **Second Layer:** This layer processes the features transformed by the preceding layer. It continues to integrate and improve node characteristics based on their local graph neighbourhoods. The continual employment of ReLU contributes to the maintenance of non-linear learning capabilities. Dropout is used after activation to minimize overfitting by randomly removing units features throughout the training process, making the model resistant to noise in the data.

.

• **Third Layer:** The final layer of the GCN architecture generates logits for each class, which are subsequently processed by a log SoftMax activation function. This function transforms logits to probabilities by normalizing the outputs so that their exponentials amount to one. The output is a probability distribution over classes for each node, which is appropriate for multi-class classification problems. This layer essentially transfers the graph's refined features to specific classes before producing the final prediction based on the aggregated and transformed information passed through the network.

Training Process:

1. Utilized the Adam optimizer with carefully selected hyperparameters (learning rate: 0.001, weight decay: 0.005). The optimizer was chosen for its ability to handle sparse gradients, which are typical in graph data due to connection patterns. It combines the advantages of two earlier stochastic gradient descent extensions: Adaptive Gradient Algorithm and Root Mean Square Propagation.

2. Negative Log-Likelihood (NLL) Loss: This is used to compare the model's predictions to the actual labels within the labelled graph nodes, which are the training nodes. The loss is estimated by calculating the negative log of the probability assigned to the true class by the SoftMax function in the output layer.

Hyperparameter Tuning Outputs (epochs):

```
 Evaluate the model on the test set
odel.eval()
red = model(data).argmax(dim=1)
est_preds = pred[test_idx]
est_preds[0:10]
print('Test predictions:', test_preds.tolist())
```

```
Training with lr=0.01, weight_decay=0.005, dropout_rate=0.3
Validation Accuracy: 0.8387
Training with lr=0.01, weight_decay=0.005, dropout_rate=0.5
Validation Accuracy: 0.8407
Training with lr=0.01, weight_decay=0.005, dropout_rate=0.7
Validation Accuracy: 0.8387
Training with lr=0.01, weight_decay=0.001, dropout_rate=0.3
Validation Accuracy: 0.8266
Training with lr=0.01, weight_decay=0.001, dropout_rate=0.5
Validation Accuracy: 0.8246
Training with lr=0.01, weight_decay=0.001, dropout_rate=0.7
Validation Accuracy: 0.8226
Training with lr=0.01, weight_decay=0.0001, dropout_rate=0.3
Validation Accuracy: 0.8145
Training with lr=0.01, weight_decay=0.0001, dropout_rate=0.5
Validation Accuracy: 0.8125
Training with lr=0.01, weight_decay=0.0001, dropout_rate=0.7
Validation Accuracy: 0.8064
Training with lr=0.005, weight_decay=0.005, dropout_rate=0.3
Validation Accuracy: 0.8387
Training with lr=0.005, weight_decay=0.005, dropout_rate=0.5
Validation Accuracy: 0.8407
Training with lr=0.005, weight_decay=0.005, dropout_rate=0.7
Validation Accuracy: 0.8448
Training with lr=0.005, weight_decay=0.001, dropout_rate=0.3
Validation Accuracy: 0.8286
Training with lr=0.005, weight_decay=0.001, dropout_rate=0.5
```

Learning Rate: This determines how frequently the model weights are adjusted during training. A high learning rate could cause the model to converge too rapidly.

Weight Decay is a regularization strategy that discourages big weights in the model by imposing a penalty on the loss function that is proportionate to the weight size. This can assist prevent overfitting by keeping the model simple.

Dropout Rate: This is another regularization strategy that randomly 'drops out' or sets to zero a percentage of the features during training. We ran it multiple times so to fine tune the model parameters and selected the best resultant parameters for the model.

Training process

- **Forward Pass:** The model executes a forward pass, which includes computing the graph convolutions specified in the layers section. Each layer processes node features and passes them through the graph structure using the learned weights.

- **Backpropagation:** After calculating the loss, backpropagation is used to calculate the gradients of the loss function in relation to each weight in the model. The gradients represent the direction and size of the change required to minimize the loss.
  The optimizer then adjusts the weights based on the gradients and learning rate.

- **Dropout:** To avoid overfitting during training, dropout is used after each ReLU activation in the hidden layers by randomly omitting a fraction of feature detectors (nodes) for each training case.

Evaluation

After training, the model conducts a forward pass on the validation and test datasets (by the val_mask and test_mask, respectively). This process generates anticipated labels based on the learnt weights and then computes the metrices.

Then Using the performance measurements, finetune the model's hyperparameters to increase performance. This could entail experimenting with various combinations and repeating the training and evaluation processes.

Further To avoid overfitting, training can be stopped when validation performance deteriorates or stops increasing after a given number of epochs.

Then after finetuning we sent the code for evaluation.

Code Snippet:

```
    def forward(self, data):
        x, edge_index = data.x, data.edge_index
        x = F.relu(self.conv1(x, edge_index))
        x = F.dropout(x, training=self.training)
        x = F.relu(self.conv2(x, edge_index))
        x = F.dropout(x, training=self.training)
        x = self.conv3(x, edge_index)
        return F.log_softmax(x, dim=1)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = GCN().to(device)
data = data.to(device)

optimizer = torch.optim.Adam(model.parameters(), lr=0.01, weight_decay=5e-3)

model.train()
for epoch in range(200):
    optimizer.zero_grad()
    out = model(data)
    loss = F.nll_loss(out[data.train_mask], data.y[data.train_mask], ignore_index=-1)
    loss.backward()
    optimizer.step()
    if epoch % 50 == 0:
        print('Epoch {0}: {1}'.format(epoch, loss.item()))

model.eval()
pred = model(data).argmax(dim=1)
correct = (pred[data.val_mask] == data.y[data.val_mask]).sum()
acc = int(correct) / int(data.val_mask.sum())
print(f'Accuracy: {acc:.4f}')
```

Solutions and Performance Analysis

The model had an accuracy of 83.7% and 83.4% on the test set. This consistent performance across both datasets implies that the model generalizes effectively to new, previously unseen data, providing a viable solution to the problem.

Key takeaways included the importance of data quality, the significance of strategic hyperparameter finetuning, and the advantages of regularization techniques like dropout and early stopping.

This project's findings include the importance of graph neural networks, regularization approaches, and thorough data preparation. Experience has shown that topics such as feature engineering and model architecture improvising are crucial for future progress.

Future Directions

We will try focus on exploring deeper or alternative neural network architectures, enhancing feature engineering techniques, and potentially using more extensive datasets to improve the robustness and accuracy of the model. We would also like to apply the experience gained into new domains like Bioinformatics, Cybersecurity, and finance.

We have few key points where we would like to spearhead our project: -

- **Advanced Feature Engineering**: Learn and understand in detail about extracting delicate information from complex datasets.

- **Expanded Model Testing**: Use a variety of algorithms and designs to identify the most successful solutions and use the experience gained to boost accuracy.

- **Data Augmentation Techniques:** Improve model generalization and robustness by artificially expanding training datasets.

Currently we used GCN model implementation where we achieved the accuracy of 83.7 % but in future, we might also try other strategies for the insights and try to improve the model accuracy to reach the optimal solution for the project.