College of Science

Lecturer: Dr. Taheri

Spring 1402

Final project report

Graph mining course

Zahra Haghshenas

1 .Introduction

The purpose of this work report is to provide details of a project aimed at predicting molecular properties using a Graph Neural Network (GNN) model. This project uses the BBBP (Blood-Brain Barrier Penetration) dataset, which records the permeability of compounds to the blood-brain barrier. This report covers the description of the dataset, an overview of the code used, and the output obtained from running the code on the dataset.

2 .Description of the dataset

The BBBP dataset consists of molecular data related to blood-brain barrier permeability of compounds. The dataset contains 2039 compounds, each represented by a set of molecular descriptors. The goal is to predict whether a given compound is permeable to the blood-brain barrier, indicating its ability to cross this physiological barrier. The dataset is classified as a classification task, where the target variable is binary (penetrable or nonpenetrable). The data set was extracted from the study by Martinez et al. (2012) on modeling and predicting the permeability of the blood-brain barrier. The authors used a Bayesian approach to model the penetration of the silicon blood-brain barrier

3 . Code review

At first the code focuses on the initial setup and definition of the Graph Neural Network (GNN) model using the DGL (Deep Graph Library) library .

3.1 .Importing Libraries:

- The code starts by importing necessary libraries. These include torch (PyTorch), torch.nn (PyTorch's neural network module), torch.nn.functional

(functions for neural network operations), dgl (Deep Graph Library), and time (for tracking execution time).

3.2 .Defining the GNN Model:

- Next, we define the GNN model as a subclass of the `nn.Module` class. In Python, creating a custom neural network model typically involves inheriting from the `nn.Module` class and implementing the `__init__` and `forward` methods.

- In the `__init__` method of the GNN model, we initialize various attributes such as the configuration (`config`) dictionary, the number of tasks (`num_tasks`), the sizes of node features and edge features, and the hidden size of the GNN layers.

- In this part, the GNN model consists of two `GraphConv` layers (`conv1` and `conv2`). The `GraphConv` layer is a basic graph convolutional layer provided by DGL, which performs message passing and graph convolution operations.

3.3 .The Forward Pass:

- The `forward` method of the GNN model defines the forward pass logic for the GNN.

- Inside the `forward` method, we pass the molecular graph (`mol_dgl_graph`) and global features (`globals`) as input arguments.

- First,we truncate the node features and edge features of the input graph to the desired sizes using slicing operations (`mol_dgl_graph.ndata["v"]` and `mol_dgl_graph.edata["e"]`).

- Then, we apply the first `GraphConv` layer (`conv1`) to the input graph and node features (`mol_dgl_graph`, `mol_dgl_graph.ndata["v"]`).

- The resulting node features are passed through a ReLU activation function using `F.relu.`

- Next, we apply the second `GraphConv` layer (`conv2`) to the graph and the activated node features.

- Finally, we update the node features of the graph with the computed values (`mol_dgl_graph.ndata["h"] = h`) and compute the mean of the node features using `dgl.mean_nodes` with the target feature name "h."

3.4 .Setting Up Configuration and Model Initialization:

- After defining the GNN model, we set up a configuration dictionary (`config`) to hold various settings or hyperparameters.

- Then, we instantiate an instance of the GNN model (`gnn`) by passing the configuration and the number of tasks to it (`GNN(config, num_tasks=1)`).

In the continuation of the code consists of several components related to training, evaluating, and testing a graph based model .

3.5.Loss Function**: The `loss_func` function calculates the loss for a multi task binary classification problem. It uses the binary cross entropy loss (`BCEWithLogitsLoss`) with specified positive weights. The function applies a mask to filter out padded or irrelevant elements, and the final loss is averaged over the valid elements.

3.6.Training Function**: The `train_epoch` function performs one training epoch for the graph based model. It iterates over the batches in the training data, computes the model's predictions and the training loss, performs backpropagation to compute the gradients, updates the model's parameters,

and accumulates the training loss. The function returns the average training loss per iteration for the epoch.

3.7. Train and Evaluate Function**: The `train_evaluate` function trains and evaluates the graph based model for a specified number of epochs. It keeps track of the best validation score and saves the corresponding model checkpoint. During training, it prints the current epoch, training loss, and validation score. If the validation score improves, it saves the checkpoint. If the validation score does not improve for a certain number of epochs (defined by the patience count), the training process stops. After training, the function copies the checkpoint files of the best model to a separate directory. Finally, it prints the final results, including the average validation score.

3.8.Test Evaluation Function**: The `test_evaluate` function loads the best model checkpoint, initializes a model with the same configuration, loads the model's state from the checkpoint, evaluates the model on the test dataset, and prints the test score and execution time.

3.9.DGLDatasetClass Modification**: In this part, the `DGLDatasetClass` is modified to include the `edge_features` attribute. This modification allows loading and accessing edge features from the dataset.

3.10.GNN Model Modification**: The `GNN` model class is modified to include additional functionality for message passing and global feature transformation. The model takes a DGL graph (`mol_dgl_graph`) and global features (`globals`) as inputs and returns the aggregated node features.

By calling the `train_evaluate()` and `test_evaluate()` functions one after the other, the code performs both training and testing of the graph based model.

The `start_time` variable is used to calculate and print the total execution time for both operations.

in this part the code we provided introduces different variations of the SAGEConv module for graph convolutional operations. Each variation uses a different message function and reduce function to perform message passing and aggregation.

3.11.SAGEConv: The `SAGEConv` class defines a SAGEConv layer that uses the "copy" message function and "mean" reduce function. It copies the node features to messages and computes the mean of the aggregated messages from neighboring nodes. The class extends the `nn.Module` class and implements the forward method to perform the graph convolution operation.

3.12.GNN Model with SAGEConv**: The `GNN` class extends the `nn.Module` class and initializes a GNN model. It has two SAGEConv layers (`self.conv1` and `self.conv2`) for performing graph convolution operations. The forward method of the `GNN` class applies the SAGEConv layers to the node features and performs ReLU activation between the layers.

3.13.SAGEConv1**: The `SAGEConv1` class defines a SAGEConv layer that uses the "add" message function and "sum" reduce function. It performs element-wise addition of the node features and computes the sum of the aggregated messages from neighboring nodes.

3.14.GNN Model with SAGEConv1**: The `GNN` class is modified to use the `SAGEConv1` layer instead of the original `SAGEConv` layer. The rest of the code remains the same.

3.15.SAGEConv2**: The `SAGEConv2` class defines a SAGEConv layer that uses the "add" message function and "sum" reduce function. It performs element-wise addition of the node features and computes the sum of the aggregated

messages from neighboring nodes. This implementation uses the `send_and_recv` function for message passing and aggregation.

3.16.GNN Model with SAGEConv2**: The `GNN` class is modified to use the `SAGEConv2` layer instead of the original `SAGEConv` layer. The rest of the code remains the same.

3.17.SAGEConv3**: The `SAGEConv3` class defines a SAGEConv layer that uses the "div" message function and "mean" reduce function. It performs element-wise division of the node features and computes the mean of the aggregated messages from neighboring nodes.

3.18.GNN Model with SAGEConv3**: The `GNN` class is modified to use the `SAGEConv3` layer instead of the original `SAGEConv` layer. The rest of the code remains the same.

3.19.SAGEConv4**: The `SAGEConv4` class defines a SAGEConv layer that uses the "sub" message function and "mean" reduce function. It performs element-wise subtraction of the node features and computes the mean of the aggregated messages from neighboring nodes.

3.20.GNN Model with SAGEConv4**: The `GNN` class is modified to use the `SAGEConv4` layer instead of the original `SAGEConv` layer. The rest of the code remains the same.

3.21.SAGEConv5**: The `SAGEConv5` class defines a SAGEConv layer that uses the "mul" message function and "mean" reduce function. It performs element-wise multiplication of the node features and computes the mean of the aggregated messages from neighboring nodes.

3.22.GNN Model with SAGEConv5**: The `GNN` class is modified to use the `SAGEConv5` layer instead of the original `SAGEConv` layer. The rest of the code remains the same.

3.23. SAGEConv6: This module uses element-wise multiplication ("mul") as the message function and computes the sum ("sum") of aggregated messages from neighboring nodes as the reduce function.

3.24. SAGEConv7: This module also uses element-wise multiplication ("mul") as the message function but computes the minimum ("min") of aggregated messages from neighboring nodes as the reduce function.

3.25. SAGEConv8: This module uses element-wise multiplication ("mul") as the message function and computes the maximum ("max") of aggregated messages from neighboring nodes as the reduce function.

3.26. SAGEConv9: This module extends the SAGEConv module by adding additional layers. It uses element-wise multiplication ("mul") as the message function and computes the sum ("sum") of aggregated messages from neighboring nodes as the reduce function. It also incorporates batch normalization layers between the convolutional layers.

3.27. SAGEConv10: This module is similar to SAGEConv9 but uses element-wise addition ("add") as the message function and computes the maximum ("max") of aggregated messages from neighboring nodes as the reduce function.

The GNN class is updated accordingly to include the different SAGEConv modules in its architecture.

After each variation of the SAGEConv module is defined, the `train_evaluate()` and `test_evaluate()` functions are called to train, evaluate, and test the corresponding GNN models.

4 code output

After running the code on the BBBP dataset, the following results were obtained:

"Average valid score" shows the average performance of the model in the validation set

It gives how well it generalizes to unseen data. "Test score" the performance of the model

The test set shows and estimation of its ability to predict blood barrier permeability-

Brain provides compounds. Execution time indicates the duration of code execution and acquisition

The results are

model	Average Valid Score:	Test Score:	Execution time:
GCN	0.824	0.636	89.537 seconds
Covsage: message copy and reduce mean	0.843	0.565	39.441 seconds
Covsage: message add and reduce sum	0.848	0.634	38.615 seconds
Covsage: message add and reduce sum	0.864	0.565	28.719 seconds
Covsage: message div and reduce max	0.000	0.000	9.678 seconds
Covsage: message sub and reduce mean	0.504	0.404	10.609 seconds
Covsage: message mul and reduce mean	0.814	0.614	94.291 seconds
Covsage: message mul and reduce sum	0.639	0.472	10.532 seconds
Covsage: message mul and reduce min	0.815	0.617	95.197 seconds

Covsage: message mul and reduce max	0.800	0.615	96.035 seconds
add layers 3 layers message mul and reduce sum	0.879	0.835	56.280 seconds
add layers 4 layers message mul and reduce sum	0.822	0.788	61.389 seconds
add layers 4 layers message add and reduce max	0.818	0.813	34.850 seconds

5.Result:

In conclusion, this report describes the work of a project focused on predicting the permeability of the blood-brain barrier of compounds using a graph neural network model. This code uses DGL and PyTorch libraries to process the BBBP dataset, train the model and evaluate its performance. Based on the provided output criteria,

The best model is 3 layers message mul and reduce sum, which has more test points than other models .

These results show that the model shows a reasonable level of performance in predicting the permeability of the blood-brain barrier.