In the name of God

GNN final project report

Mohammad mahdi salehi

Abstract

In this project, we perform two tasks of classification and regression on molecular graphs. We used 10 different graph neural network architectures to obtain the vertex feature(Node embedding) for graphs and observed the results. We used BBBP dataset for the classification problem and FreeSolv dataset for the regression problem.

Introduction

As we know, when our data has dependencies, we use graphs to represent them. Depending on the graph and the type of prediction we want to do, we have different tasks. One of those tasks is the problem of classifying graphs, where the dataset has a number of graphs. In the first part of our project, the graphs are the molecules that are in the BBBP dataset, and there is information about the relationship of molecules with proteins; If the molecule is related to protein, it is a member of class 1, otherwise it is a member of class 0. Our goal is to predict a new molecule class. Also, in the second part of the project, the graphs that are molecules are located in the FreeSolv dataset, and our goal is to predict the stability of the solubility of molecules, and in that dataset, the stability energy of the solubility of molecules at a temperature of 25 degrees Celsius is given; Therefore, our problem is regression type.

We used 10 different architectures of neural networks; Both for classification and with a slight change for regression. 3 of them are GCN architecture, 3 GraphSAGE architectures, 3 GAT architectures and one architecture without using ready-made architectures in the DGL library.

Method

First, we explain the classification problem:

First, we install the DGL library and then call the required libraries in the code.

In the next part, we call and unzip the dataset.

In the next cell, The `DGLDatasetClass` class is used for creating a dataset for a classification problem. This class is designed to be used in other classes like `DataLoader` in PyTorch.

In the constructor of the class, the file path of the dataset is passed as an input, and the `dgl.load_graphs` function is used to load the data from the file. Then, labels, masks, and global features of the graphs are extracted from the loaded data.

In the `__len__` method, the length of the dataset is calculated for training and evaluation purposes. In the `__getitem__` method, a single sample from the dataset is returned.

In the next cell, the path of the dataset files is specified, and then the `DGLDatasetClass` class is used to create three datasets: training, validation, and testing. The `len` function is used to print the number of samples in each dataset.

In the next cell, two functions are defined: `collate` and `loader`.

The `collate` function is used to concatenate a sequence of graphs, labels, masks, and globals into a single batch. The input to this function is a list of tuples, where each tuple contains the data of a single sample. This function concatenates the graphs, labels, masks, and globals of all the samples in the batch and returns them as a tuple.

The `loader` function returns three data loaders for the training, validation, and test sets. Each data loader is created using the `DataLoader` class from PyTorch, with the batch size, collate function, drop_last, shuffle, and number of workers specified. The collate function used in the `DataLoader` is the `collate` function defined earlier. Finally, the function returns the three data loaders.

In the next cell, This line of code calls the `loader` function and creates three data loaders for the training, validation, and test sets with a batch size of 32. These data loaders will be used to load the data during the training, validation, and test phases of the model. The data loaders will return batches of size 32 during each iteration of the training, validation, and test loops. We tested batch size 256, 128, 64, 32 and 16 on the first architecture and observed that batch size 32 gives us better results.

In the next cell, some hyperparameters and configurations for the model are defined:

num_tasks is the number of tasks in the dataset. In this case, it is set to 1, as the BBBP dataset has only one task.

global_size is the size of the global feature of each graph.

num_epochs is the number of epochs to train the model.

patience is the number of steps to wait if the model performance on the validation set does not improve.

config is a dictionary that contains some configurations to instantiate the model. These configurations include the size of node features, edge features, and hidden layers.

In the next cell, This part of the code defines a PyTorch module for a Graph Neural Network (GNN) model called `GNN1`.

In the constructor of the class, the configurations for the model and the number of tasks are defined and stored as attributes. The sizes of node features, edge features, and hidden layers are also extracted from the configurations and stored as attributes. Then, two `GraphConv` layers are created for the message passing. The first layer takes the node features as input and produces a hidden representation, which is then passed through a ReLU activation function. The second layer takes the hidden representation as input and produces the final output, which is the predicted label.

The `forward` method takes a molecular graph and a global feature vector as input. The node and edge feature sizes are truncated to the specified sizes using slicing. The message passing is performed using the `GraphConv` layers, and the final node representations are stored as node features in the graph. Finally, the mean of the node features across all nodes is computed and returned as the output of the model.

In the next cell, defines a function called `compute_score` that is used to compute the performance score of the model on the given data.

The input to this function is the trained model, the data loader for the validation or test set, the size of the validation or test set, and the number of tasks in the dataset.

The function first sets the model in evaluation mode using the `eval` method. Then, it uses `roc_auc_score` from the `sklearn.metrics` module as the performance metric. Next, it iterates over the data loader and makes predictions for each batch using the trained model. The predictions, labels, and masks for all batches are concatenated using `torch.cat`. Finally, the function computes the average metric score across all tasks and returns the score as a float.

If the metric cannot be computed for any task, the score for that task is set to 0.

This part of the code defines a function called `loss_func` which is used to compute the loss between the model's output and the true labels.

The input to this function is the model's output, true labels, binary mask, and the number of tasks in the dataset. The function first creates a tensor of ones with the same shape as the output tensor, which is used as positive weights in the binary cross-entropy loss function. Then, it creates a `BCEWithLogitsLoss` object that applies binary cross-entropy with logits. The 'none' reduction parameter is used to keep the loss for each sample separate, and the positive weights are used to give more weight to the positive class to handle class imbalance.

The loss is computed for each sample, but only for the samples specified by the mask. The loss values are summed and divided by the sum of the mask to get the average loss across the samples. Finally, the computed loss is returned as a scalar tensor.

This part of the code defines a function called `train_epoch` which trains the model for one epoch on the given data.

The input to this function is the data loader for the training set, the model to be trained, and the optimizer used to update the model parameters.

The function first initializes the epoch training loss and the number of iterations to 0. Then, it sets the model in training mode using the `train` method. Next, it iterates over the data loader and gets the input samples. The model makes predictions for the input samples, and the loss between the predictions and the true labels is computed using the `loss_func` function. The gradients of the loss with respect to the model parameters are computed using the `backward` method, and the optimizer updates the model parameters using the computed gradients using the `step` method. The epoch training loss is updated, and the number of iterations is incremented.

Finally, the average epoch training loss is computed by dividing the epoch training loss by the number of iterations, and this value is returned as a float.

This part of the code defines a function called `train_evaluate` that trains and evaluates the model.

The function first creates a new instance of the `GNN1` model and an Adam optimizer to update the model parameters during training. It also initializes the best validation score, patience count, and epoch to 0.

Adam with a learning rate of 0.001. We also tried the learning rate of 0.0001 on the first architecture and observed that the rate of 0.001 with a batch size of 32 gives us a better result in less time.

The function then enters a loop to train and evaluate the model for the specified number of epochs. If the patience count has been exceeded, the loop exits. Otherwise, the model is set to training mode, and the `train_epoch` function is called to train the model on the training set for one epoch. The model is then set to evaluation mode, and the `compute_score` function is called to evaluate the model on the validation set and compute the validation score. If the current validation score is better than the best validation score so far, the best validation score is updated, and a checkpoint of the model is saved. Otherwise, the patience count is incremented.

After each epoch, the current epoch number, training loss, and validation score are printed. If a new best validation score is achieved, the message "Save checkpoint" is printed. The function also saves the best model, prints the final average validation score, and returns nothing.

In the next cell defines a function called `test_evaluate` that evaluates the trained model on the test set.

The function first creates a new instance of the `GNN1` model, loads the best model checkpoint from the best model path using the `cloudpickle` library, and loads the saved model state dictionary into the final model. The final model is then set to evaluation mode.

The `compute_score` function is called to evaluate the final model on the test set and compute the test score. The final test score is printed, along with the execution time of the entire program.

Finally, the function returns nothing.

In the next cell the `train_evaluate` function to train and evaluate the model and then calls the `test_evaluate` function to evaluate the trained model on the test set.

The `start_time` variable is used to measure the execution time of the entire program.

When executed, the program should print the training and validation scores for each epoch, the best validation score, the test score of the trained model, and the execution time of the program.

In the following table we train and take results with different hyperparameter for the first architecture:

	Val_score	Test_score	Train_loss
B=256 , L=0.0001	0.368	0.621	0.437
B=128, L=0.0001	0.322	0.506	0.602
B=64 , L=0.0001	0.824	0.621	0.438
B=32, L=0.0001	0.826	0.643	0.420
B=32 , L=0.001	0.831	0.677	0.396
B=16 , L=0.001	0.830	0.646	0.406

B=16 , L=0.0001	0.827	0.620	0.428

B:=batch size , L=learning rate

In the next cell, we define the second architecture, the only difference between this architecture and the first architecture is that we have used 3 layers of neural network, and therefore we used the Relu activation function once again in the forward part.

Other items are the same as before.

In the next architecture, We used 5 GraphConv layers, which in the first input layer is the feature vector as before and goes to the next layer with 64 neurons, the next layer also has 128 neurons, and the next layer has 256 and the next layer is 128 and finally in the final layer which It has 1 neuron (which is the number of our tasks) in this architecture, we used two techniques, Batch Normalization and Dropout with a rate of 0.2. Although we used Dropout only in the second and third layers. Also, as before, we used the Relu activation function.

Next, we used the GraphSAGE architecture. And we trained 3 models exactly the same as before with GraphSAGE instead of GCN.

Also, then we used 3 GAT architectures. And other architectural items are exactly the same as 3 models in GCN, and 3 models in GraphSAGE. The reason for this is to see which architecture gives us better results compared to others.

We designed the next architecture without using DGL libraries. In this architecture, we used 5 layers of convolutional neural network with kernel size 1*1. So that the second layer has 128 neurons, the third layer has 256, the fourth layer has 512, and the fifth layer has 256, as well as three layers of the normal neural network with 128 neurons, 64 neurons, and 32 neurons, and one last layer equal to the number of tasks. we used. Also, we used batch normalization and dropout technique with parameters between 0 and 0.2 in different layers.

In the table below we have the results of all models:

	Val_score	Test_score	Train_loss	Num_epoch
GNN1	0.831	0.677	0.396	16
GNN2	0.848	0.641	0.375	6
GNN3	0.855	0.805	0.238	15
SAGE1	0.856	0.624	0.370	3
SAGE2	0.847	0.792	0.252	21
SAGE3	0.864	0.902	0.112	8
GAT1	0.844	0.688	0.397	15

GAT2	0.816	0.698	0.353	17
GAT3	0.817	0.817	0.307	7
My model	0.619	0.639	0.556	10

As you can see, GraphSAGE3 architecture has given us the best result compared to other models. Also, in any architecture, the second models are better than the first and the third are better than the first and second.

In the relevant article, the best result is 0.919, and we reached the result of 0.902.

The second part of the project is about the regression problem on the FreeSolv dataset.

Most of the code parts are like the classification part. We explain only in the parts where there is a difference.

In the part custom pytorch dataset we have:

A class for a regression dataset. The class inherits from `torch.utils.data.Dataset` and is implemented to be used in neural networks.

The constructor method takes a file path for the dataset, a data transformation object, a boolean flag indicating if the dataset is for training or not, a scaler object from the `preprocessing` module of scikit-learn for normalization, and a boolean flag indicating if normalization should be applied to the labels or not.

The `__len__` method returns the length of the dataset.

The `__getitem__` method returns a single sample from the dataset. If the `scaler_regression` flag is True, the labels are normalized using the fitted scaler object. Otherwise, the labels are returned without normalization.

The `scaler_method` method fits a scaler on the training labels and returns it. If the dataset is not for training, it returns None.

The dataset consists of graphs, and the labels, masks, and global features are extracted from the dataset. The labels are viewed as a 2D tensor, where the number of rows is the number of graphs in the dataset, and the number of columns is the number of targets in the regression problem. The masks and global features are also viewed as 2D tensors, where the number of rows is the number of graphs in the dataset, and the number of columns is the number of nodes in each graph for masks and the number of global features for the global features tensor.

If a scaler object is passed to the constructor, it is used for normalization of the labels. Otherwise, a new scaler object is created using the training labels.

Also, we do the Scale work in the Collate function part and fit it in the last part where we want to do the training and display it.

Also, in this section, we use the MSE cost function. Although we use the sum instead of the mean.

The rest of the cases, including the architectures, are exactly the same as the classification section. With the difference that we did not do it in the parts where batch normalization was done. Because we had errors that we had to change the size of the batches to fix. Also, we used dropout with different values and we came to the conclusion that if we set it equal to zero, we get a better result because we don't suffer from overfitting. We have also used linear layers (simple neural network) in the GAT section; which we had not used in the classification section.

The following table shows the results of different architectures.

	Val_loss	Test_loss	Train_loss	Num_epoch
GNN1	3.950	3.221	0.592	6
GNN2	3.724	3.160	0.440	12
GNN3	2.398	2.722	0.195	55
SAGE1	3.356	2.480	0.101	39
SAGE2	2.810	2.245	0.079	32
SAGE3	1.729	1.565	0.091	18
GAT1	3.184	2.732	0.248	35
GAT2	2.629	2.976	0.266	71
GAT3	2.561	2.847	0.271	47
My model	4.405	4.260	0.951	21

As you can see, the best result is obtained for GraphSAGE3. which is equal to 1.565 and the best result in the article is 1.501. Also, like the classification of the second models, they are better than the first of their kind, and Shum is better than the second and the first. Also, in both classification and regression, GraphSAGE architectures gave the best results.

Conclusion

In both classification and regression, GraphSAGE3 gave the best results. And it is expected because they include batch normalization and dropout and more convolution layers. And also the GraphSAGE architecture itself is a good and used architecture.