Graph Machine Learning

Final Project Report

Meysam Agah

Winter 2023

Table of contents

Introduction **02**

About dataset **03**

Preprocessing **05**

Model defining **08**

Comparison **12**

Intro

At this project our task was to implement a GNN model in order to accurately predict molecular properties.

This project is a part of a paper published by Hossein Hajiabolhassan, Zahra Taheri with title of FunQG: Molecular Representation Learning via Quotient Graphs .

In this project we make a classification and regression on two datasets of main paper.

For classification we use HIV dataset for classification which I'll explain more about it at further sections and ESOL dataset for regression which also I'll explain about it.

This project implementation is provided by using DGL library and some tools from torch modules.

At this project we'll compare effectiveness of different GNN models to predict task of datasets.

About Datasets

Datasets in this project are divided into two sections:

Classification and Regression

Classification:

For classification task there were multiple datasets but I choose HIV datasets which was introduced by the Drug Therapeutics Program (DTP) AIDS Antiviral Screen, which tested the ability to inhibit HIV replication for over 40000 compounds.

Some more information about datasets.

Table 1 Details about molecular graphs, molecular quotient graphs, and functional groups in classification datasets

	HIV dataset
Dataset size	41127
Total #nodes in MG	1049163
Total #nodes in Q-non-FGs	534982
Total #nodes in MQG	325588
Abstraction ratio	0.31
Avg #nodes per mol in MG	25.51
Avg #nodes per mol in Q-non-FGs	13.01
Avg #nodes per mol in MQG	7.92
Total #FGs	178041
Total #unique FGs	4249
Avg #FGs per mol	4.33
#Molecules without FGs	3

Regression:

For regression I choose ESOL dataset which is a small dataset consisting of water solubility data for some compounds.

Some more information about dataset:

Table 2 Details about molecular graphs, molecular quotient graphs, and functional groups in regression datasets

	ESOL dataset
Dataset size	1128
Total #nodes in MG	14991
Total #nodes in Q-non-FGs	7072
Total #nodes in MQG	4751
Abstraction ratio	0.32
Avg #nodes per mol in MG	13.29
Avg #nodes per mol in Q-non-FGs	6.27
Avg #nodes per mol in MQG	4.21
Total #FGs	2639
Total #unique FGs	139
Avg #FGs per mol	2.34
#Molecules without FGs	116

Preprocessing

Let’s go deep in implementation.

For this project first, in order to use DGL we bring it by !pip command.

Then we import these necessary libraries and modules:

os: provides functions for creating and removing a directory (folder), fetching its contents, changing and identifying the current directory, etc. need to import the os module first to interact with the underlying operating system.

dgl: The Deep Graph Library (DGL) is a Python open-source library that helps researchers and scientists quickly build, train, and evaluate GNNs on their datasets.

NumPy: is a Python library used for working with arrays. It also has functions for working in domain of linear algebra, Fourier transform, and matrices.

NetworkX: is a Python language software package for the creation, manipulation, and study of the structure, dynamics, and function of complex networks. It is used to study large complex networks represented in form of graphs with nodes and edges.

Torch: is an open-source ML library used for creating deep neural networks and is written in the Lua scripting language. It's one of the preferred platforms for deep learning research.

Torch.nn: this module is used to train and build the layers of neural networks such as input, hidden, and output. Torch.nn base class helps wrap the torch's parameters, functions, and layers.

dgl.function: This subpackage hosts all the built-in functions provided by DGL.

torch.nn.functional: brings already built-in functions provided by torch.nn

Shutil: this module offers high-level operation on a file like a copy, create, and remote operation on the file.

Data loader: Combines a dataset and a sampler, and provides an iterable over the given dataset.

Cloudpickle: makes it possible to serialize Python constructs not supported by the default pickle module from the Python standard library.

SKlearn.preprocessing: package provides several common utility functions and transformer classes to change raw feature vectors into a representation that is more suitable for the downstream estimators.

Math: for some mathematical functions

Loading datasets:

Now by using shutil module and using original path we upload dataset to notebook.

After this we need to define DGLdataset classes:

For classification:

That's simple by having file address and using bin file we load graphs and labels and masks and globals.

Then performing train and test and split according to file. we bring data from its origin location. at origin location there were 3 scaffold which represent on scaffold in dataset so we choose first one of scaffold0 and then by DGLDataset we bring train and test and validation.

For regression:

For regression it's a bit harder. We need to scale data as well as loading it.

performing train and test and split according to file. we bring data from its origin location. at origin location there were 3 scaffold which represent on scaffold in dataset so we choose first one of scaffold0 and then by DGLDataset we bring train and test and validation.

at this step we need to scale train set using Standard Scaler from SKlearn.preprocessing which we defined earlier at defining Regression dataset class.

then according to scaler which we scaled train set, we scale validation set and test set.

Then we create a collate class which catch graphs, labels, masks, globals from batches tuple.

Then we define loader class:

For classification:

then we define a loader with batch size of 64 since dataset is a bit big. we'll define 3 dataloaders for train and validation and test, batch size for each is 64 and collate_fn is according to collate we defined earlier before defining loader.

For regression:

we define a loader with batch size of 32 since dataset is small.

we'll define 3 dataloaders for train and validation and test, batch size for each is 32 and collate_fn is according to collate we defined earlier before defining loader.

Now we have a train loader and test loader and validation loader.

Model Defining

In order to have a better accuracy we try by using different model which each of them are under different condition.

number of tasks in this dataset is 1 and we set number of epochs to 100 which looks enough for this kind of task. global size is 200 which represents global feature of each graph and patience is 10 which is Number of steps to wait if the model performance on the validation set does not improve.

for configuration it's better to set node_feature_size to 127, edge_feature_size to 12 and hidden size to 100 in order to avoid further problems.

Then since we use 2 graph models Graph Sage and GCN and since we import GCN and define Graph Sage but first let's have a quick brief about what each of them are:

Graph Convolutional Network (GCN):

$$h_i^{(l+1)} = \sigma\left(b^{(l)} + \sum\nolimits_{j\in N(i)} \frac{1}{c_{ji}} h_j^{(l)} W^{(l)}\right)$$

Where N(i) is the set of neighbors of node i, c_{ji} is the product of the square root of node degrees(i.e., $c_{ji} = \sqrt{|N(j)|}\sqrt{|N(i)|}$), and σ is an activation function.

If a weight tensor on each edge is provided, the weighted graph convolution is defined as:

$$h_i^{(l+1)} = \sigma\left(b^{(l)} + \sum_{j\in N(i)} \frac{e_{ji}}{c_{ji}} h_j^{(l)} W^{(l)}\right)$$

Where e_{ji} is the scalar weight on the edge from node j to node i.

Graph SAGE:

$$h_{N(i)}^{(l+1)} = aggregate\left(\left\{h_j^{\ l}, \forall j \in N(i)\right\}\right)$$

$$h_i^{(l+1)} = \sigma\left(W.concat\left(h_i^{\ l}, h_{N(i)}^{\ l+1}\right)\right)$$

$$h_i^{(l+1)} = norm\left(h_i^{(l+1)}\right)$$

If a weight tensor on each edge is provided, the aggregation becomes:

$$h_{N(i)}^{(l+1)} = aggregate\left(\left\{e_{ji} h_j^{\ l}, \forall j \in N(i)\right\}\right)$$

Where e_{ji} is the scalar weight on the edge from node j to node i.

Now after defining graph models we define 20 models according to next page table:

Table 3 all models details

model	Number of layers	Graph	Dropout
Model01	2	GCN	0
Model02	2	Graph Sage	0
Model03	3	GCN	0
Model04	3	Graph Sage	0
Model05	4	GCN	0
Model06	4	GCN	0.1
Model07	4	GCN	0.2
Model08	4	GCN	0.25
Model09	4	Graph Sage	0
Model10	4	Graph Sage	0.1
Model11	4	Graph Sage	0.2
Model12	4	Graph Sage	0.25
Model13	5	GCN	0
Model14	5	GCN	0.1
Model15	5	GCN	0.2
Model16	5	GCN	0.25
Model17	5	Graph Sage	0
Model18	5	Graph Sage	0.1
Model19	5	Graph Sage	0.2
Model20	5	Graph Sage	0.25

Now we define a compute score function:

For regression loss function is RMSE and for classification is ROC-AUC.

Then we define a loss function to calculate loss. For regression we use MSE as criterion of loss function and for classification we use binary cross entropy.

MSE or mean squared error is a measure to calculate error in regression tasks. Formula is:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2$$

Where n is number of datapoints and Y_i is observed values and $\hat{Y}_i$ is predicted value.

Binary cross entropy or BCE is a model metric that tracks incorrect labeling of the data class by a model, penalizing the model if deviations in probability occur into classifying the labels.

$$BCE = \frac{1}{N}\sum_{i=1}^{N} -\left(y_i \times log(p_i) + (1 - y_i) \times log(1 - p_i)\right)$$

Then we define a train epoch function.

Then we define a train evaluate function which its optimizer is Adam optimizer with learning rate of 0.001 for regression first we set best validation to infinity and for classification we set to 0. This is because we want to observe better accuracies after each epoch so if first best val is set to a very bad number it'll change once model is improving.

Another thing in train evaluate is patience count and that allow us to stop operation once model was now improving.

After this we define a test evaluate function to evaluate model improvement on test set.

Model Evaluation and comparison

Now it's time to check how much accuracy each model have and compare with original model from paper.

Table 4 comparison between models

models	Classification			Regression		
	epochs	Valid score	Test score	epochs	Valid score	Test score
Model01	100	0.800	0.734	94	2.928	3.893
Model02	75	0.783	0.684	100	2.635	3.551
Model03	100	0.804	0.741	100	2.858	3.534
Model04	100	0.819	0.770	100	2.194	2.636
Model05	48	0.794	0.737	100	2.771	3.191
Model06	88	0.815	0.778	87	2.794	3.238
Model07	100	0.827	0.784	37	2.901	3.839
Model08	100	0.829	0.780	100	2.770	3.215
Model09	79	0.839	0.791	100	2.005	2.436
Model10	83	0.840	0.777	100	1.831	2.400
Model11	81	0.842	0.776	100	1.828	2.379
Model12	96	0.843	0.787	100	1.909	2.458
Model13	79	0.831	0.777	100	2.748	3.418
Model14	86	0.836	0.778	70	2.827	3.468
Model15	100	0.826	0.781	92	2.763	3.344
Model16	100	0.827	0.774	100	2.749	3.302
Model17	63	0.831	0.778	100	1.690	2.241
Model18	48	0.836	0.768	100	1.641	2.144
Model19	56	0.836	0.786	100	1.724	2.211
Model20	64	0.846	0.783	20	2.907	4.128

According to table 4:

best model for classification is model 09 which has Graph SAGE and 4 layers and dropout = 0. It appears that more than 4 layers doesn't improve model too much Graph SAGE is a bit better than GCN and dropout doesn't necessarily improve model.

best model for regression is model 18 which has 5 layers and using Graph SAGE and dropout is 0.1. it appears that more layers result in better accuracy and dropout doesn’t necessarily improve model and graph sage is better than GCN