UNIVERSITE PARIS SACLAY

Master in Computer Science (AI & DS track)

Report of TER (Travail d’étude et de recherche)
May 2023

Graph Neural Networks for glassy dynamics modeling

Efoe Etienne Blavo
Diego Andres Torres Guarin

Graph Neural Networks for glassy dynamics modeling

Efoe Etienne Blavo
Diego Andres Torres Guarin

May 1, 2023

1 Introduction

Graph Neural Networks (GNNs) are a class of neural networks that have recently emerged as a powerful tool for analyzing and processing data that are represented as graphs. Compared to traditional deep learning methods, GNNs are able to effectively extract and use the information contained in the structure of a graph, which enables them to get the highest performance on these tasks. Some of the fields where GNNs have made remarkable progress include social network analysis, physics, chemistry or discrete optimization (Zhou et al. 2020). As such, GNNs are a very attractive concept to learn, allowing us to solve new problems, break the state of the art in already established domains, or make important contributions in this rapidly evolving field.

The main objective of this TER is to acquire the fundamental and practical knowledge necessary to apply GNNs to real world scenarios, as well as implementing recently developed models or even creating ones of our own. This involves learning about graphs in general, their mathematical description and their practical representations in programming languages. Then, we try to replicate the state of the art in a relevant problem involving structural glasses. These materials are a particular form of solid whose crystalline structure is neither regular nor convenient, but which today have many applications in the food industry, cosmetics and personal care, optics and photonics. The data and model architecture are taken from the paper published by one of the divisions of DeepMind (Bapst et al. 2020), the very famous laboratory that has lead to big leaps in machine learning.

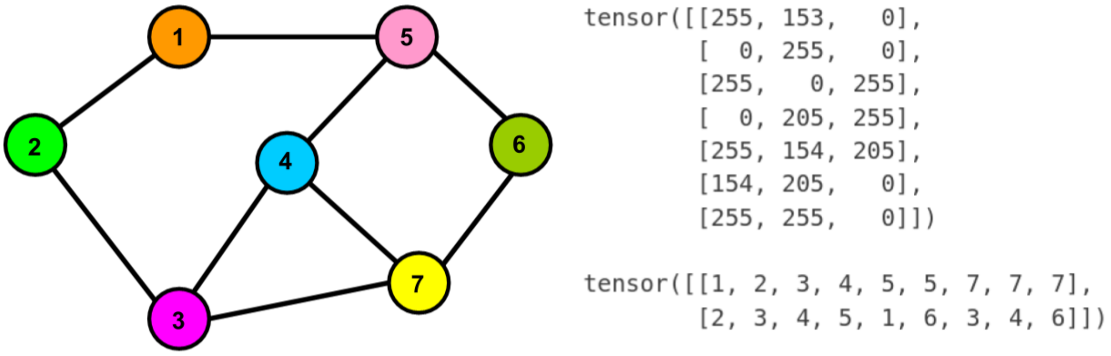
In the following pages we will explain the concepts that we have learned throughout the TER, serving both purposes of showing our understanding by explaining the ideas in our own words and also providing the appropriate context to understand the results on the main challenge.

2 Graph Neural Networks theory

2.1 Graphs and their representation

In order to understand GNNs we should first get an idea of what a graph is. Intuitively, a graph is a set of points that are linked to each other. The points are called nodes or vertices and the links between them are called edges. Mathematically, we describe a graph as a tuple of sets $G = (V, E)$, where $V = \{v_1, ..., v_n\}$ is the set of vertices and $E = \{(v_i, v_k), ...\}$ is the set of edges, which is formed by tuples of vertices, indicating that the two are connected in the graph. We can have more expressive graphs by considering nodes and edges that have properties associated to them, which we will represent as vectors.

This mathematical way to describe a graph will be slightly modified when actually implementing them in code. The set of vertices will be replaced by a 2-dimensional array, of dimensions $n \times d$ (n



```
tensor([[255, 153,   0],
        [  0, 255,   0],
        [255,   0, 255],
        [  0, 205, 255],
        [255, 154, 205],
        [154, 205,   0],
        [255, 255,   0]])

tensor([[1, 2, 3, 4, 5, 5, 7, 7, 7],
        [2, 3, 4, 5, 1, 6, 3, 4, 6]])
```

(a) The usual way to visually represent a graph

(b) Numerical representation of the graph

Figure 1: A simple graph consisting of colors as nodes (hence encoded in RGB format), and it's connectivity matrix (represented in the edge-index format)

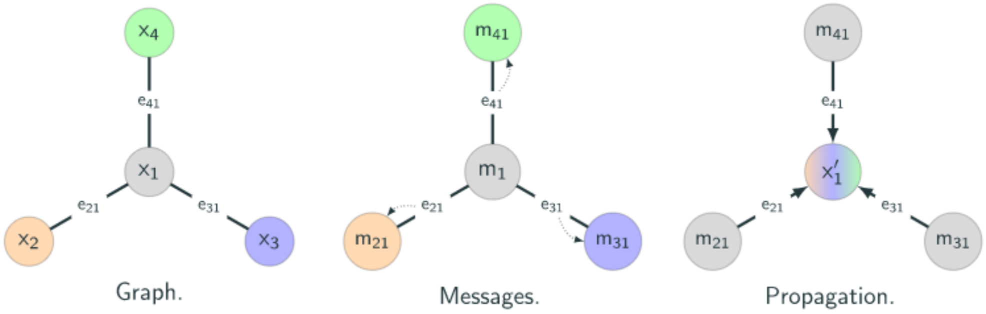


Figure 2: Diagram of the general functioning of an MPN

being the number of nodes in the graph and d the dimension of the vector representation of the node features). As for the set of edges, it will also be described as a 2-dimensional array, but this time of size $2 \times m$ (m is the number of edges). The elements inside this array will be integers indicating the index of the node it refers to. Finally, we will have an additional array of size $m \times k$ to represent the vector features of the edges (of size k). To make this more clear, we provide an example of a graph in figure 1

2.2 Relevant GNN architectures and techniques

Graph Neural Networks are a special kind of neural network that can handle graphs as input data, taking advantage of the spatial structure of the graph and connectivity to make better predictions. The architecture of the GNNs will depend on the kind of task that we want to perform, according to the object for which the predictions are made: nodes, edges or the graph as a whole. Nonetheless, there are general architectures that provide a powerful framework to build expressive GNN that can model a variety of problems.

2.2.1 Message Passing Networks

Message Passing Networks (MPNs) are a type of GNN that work by exchanging information (or messages) between the nodes of a graph, updating their vector representations. These messages are computed by considering the information from the node's neighbors and edges, even considering global information from the entire graph. MPNs are more naturally described in the context of a directed graph, where the notion of sender and receiver node makes sense. However, we can also apply it to undirected graphs by considering it as a directed graph where each node is always connected to its neighbors as sender and receiver at the same time. We will omit this detail in the following explanation by assuming a directed graph.

(1) The first step of a message passing layer consists of the computation of the messages from the senders to the receivers. This is done by means of a multilayer perceptron (MLP) that receives as input the feature vectors of the sender node, the receiver, and the edge between them.

$$\text{Message}(i \to j) = \text{MLP}(x_i, x_j, e_{ij}) \tag{1}$$

The output of this MLP will be another feature vector, whose dimension we can control according to the complexity of the problem.

(2) The next step is then to aggregate all these incoming messages for each receiver node. Generally, the aggregation function can be any that is differentiable and invariant to permutations, but it is very often the sum, max, or the mean of the messages.

(3) Finally, we grab the aggregated messages sent to the receivers and use them together with the current feature vector of the node to compute its updated version, again using an MLP.

Combining these steps together, the update rule for the node features is given by

$$x_{(i)}^{k} = \text{MLP}\left(x_i^{(k-1)}, \text{Agg}_{j\in\mathcal{N}(i)}\left(\text{MLP}(x_i^{(k-1)}, x_j^{(k-1)}, e_{ij})\right)\right) \tag{2}$$

This is only one update of a message passing layer, and in a full MPN you chain several of these together to get an improved version of the node embeddings.

2.2.2 Graph Convolutional Neural Networks

Graph Convolutional Neural Networks (Kipf and Welling 2017) are a very simple type of MPN. The first simplification comes in the message computation, in which the MLP is replaced by a linear function. Similarly, we fix the aggregation operation as the sum of the messages, and we also drop the last MLP. The resulting update equation is given by

$$x_i^{(k)} = \sum_{j\in\mathcal{N}(i)\cup\{i\}} \frac{1}{\sqrt{\deg(i)}\sqrt{\deg(j)}} W^T x_j^{(k-1)} + b. \tag{3}$$

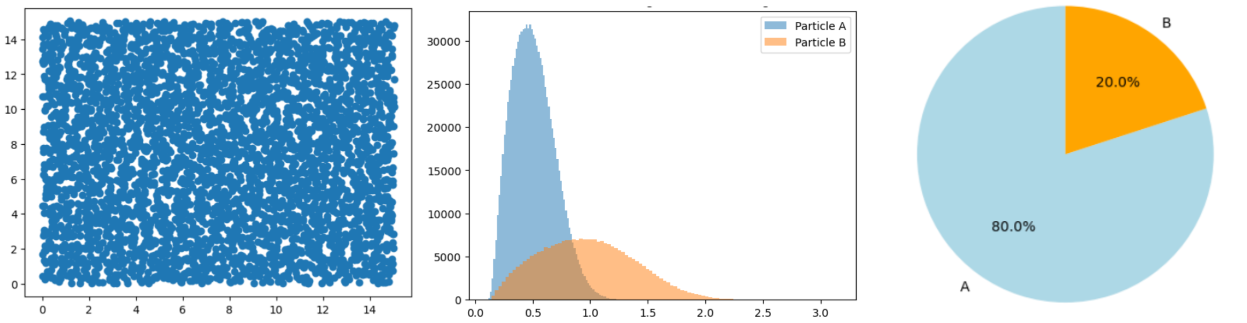
As it is expected, the several simplifications made in this architecture make it less expressive to model complex relationships in the graph. This is especially true if we're interested in graphs where the edges have features that prove useful for the task at hand, since in this formulation of GCNs they are not taken into account at all.

It is essential to emphasize that all these discussed operations, such as message passing and graph convolutional networks, are implemented using **PyTorch Geometric** (Fey and Lenssen 2019). This library, built on top of PyTorch, significantly simplifies and optimizes performance when constructing GNNs.

3 The Bapst dataset

3.1 Description of the dataset

After gaining an understanding of graph representations and Graph Neural Network (GNN) theory, as well as exploring several datasets (see Appendix), we now focus on our primary dataset. This dataset was created by the authors of (Bapst et al. 2020) and was generated using the molecular dynamics software, LAMMPS. The dataset consists of a combination of large (A) and small (B) particles that interact through a Lennard-Jones potential with a ratio of 6 to 12.



(a) First graph node's position (b) Distribution of targets in the training set (c) Distribution of nodes features

Figure 3: Data exploration graphs of the Bapst dataset

3.2 Data Exploration

In this study, we conduct an exploratory analysis to gain insights into the data in a more intuitive manner. Upon examining a graph representation of the dataset, we observe that it comprises 4096 nodes and 166848 edges. The dataset features two types of nodes, representing large particles denoted by A and small particles denoted by B (see Figure 3). Additionally, the dataset includes three edge features that correspond to the vector of relative positions between the particles.

Let the positions of Particle 1 and Particle 2 be represented by vectors:

$$\mathbf{r}_1 = (x_1, y_1, z_1),$$
$$\mathbf{r}_2 = (x_2, y_2, z_2).$$

The relative position vector between these particles is:

$$\mathbf{r}_{12} = \mathbf{r}_2 - \mathbf{r}_1 = (x_2 - x_1, y_2 - y_1, z_2 - z_1).$$

Although we attempted to visualize a sample graph in a similar manner to the Zinc dataset, we did not achieve success. Instead, we obtained a representation of the nodes' positions within a graph (see Figure 3).

We attempt to visualize the distribution of targets in our training dataset (see Figure 3). As observed, the cage-relative mean-square displacement, which quantifies particle mobility in glassy systems, ranges from 0.1 to 1 for A particles and from 0.1 to 2 for B particles.

Indeed, it is the difference in their target distributions that make the problem more interesting if we focus on predicting the mobility for only one type of particle. The authors of the paper choose particles of type A, and the pearson correlation coefficient as their metric to evaluate the model predictions. In this study, we will follow the same goals and will try to replicate their results.

4 Glassy dynamics modelling

4.1 Model architecture and design choices

We implement the architecture used by DeepMind in the paper (Bapst et al. 2020). It is based on the general message passing framework with some additional layers and details.

4.1.1 Graph Encoder

The first step of the pipeline consists of a graph encoder, which takes care of computing higher dimensional representations of the node and edge features. The encoding is performed separately for nodes and edges, using an MLP with 2 hidden layers of 64 neurons and the ReLU activation.

4.1.2 Message Passing Layer

In the context described in section 2.2.1, the message sent to the receiver nodes is simply the edge connecting it with its senders, which implies a simplification from the more general message passing framework. After the aggregation stage, which is done using the addition function, the messages are processed by an MLP

In this architecture we also update the feature vector of the edges by concatenating its current feature vector with the vectors of the two nodes it connects and then passing it through another MLP.

The two MLPs used in the message passing layer will be referred to as the *node updater* and *edge updater*, following exactly the same architecture but not sharing the same weights. This architecture consists of 2 linear layers of 64 neurons and the ReLU after each one of them. It is also important to mention that the node and edge updates are performed in ”parallel” meaning that both are computed using their previous versions and not updating one and then using this new version to update the other.

Each time that one of these layers is applied, information flows to each node from its neighbors, so that consecutive applications allow for more distance nodes to communicate through their common neighbors. For example, 2 of these layers mean that the prediction for a node will be affected by the state of its neighbors *and* the neighbors of its neighbors. In this particular dataset, 7 layers are enough to cover the whole graph, so the further addition of layers doesn't make sense from an information point of view. This is the number of layers that the authors of the paper use, and there's no reason for us to use a different one.

4.1.3 Node Decoder

After a consecutive application of message passing layers, it is time to create the prediction for each node. This is what the node encoder takes care of. It is simply an MLP with 2 linear layers and the ReLU activation function in between. It takes as input a vector of size 64 and outputs a single number, so it is applied independently to each node.

4.1.4 Skip connections

Additionally, before passing any feature vector through the MLPs we concatenate them with their original versions (coming directly from the encoder). The purpose of this is to be able to effectively propagate the gradient to the previous layers, therefore increasing the learning speed.

4.2 Training and validation setup

As it is mandatory in almost every machine learning project, we start by splitting the dataset in training, validation and testing sets. We employ an 80%-10%-10% split (corresponding to 320, 40 and 40 graphs respectively), also taking care of seeding the code so that we can reproduce the results.

Before feeding the data to the network we apply 2 small yet relevant preprocessing steps. The first one consists of one-hot encoding the node features so that they can be correctly used by the graph encoder. Secondly, we rescale the target variable so that it has a mean of 0 and a variance of 1, which is recommended to speed up the training of the network.

As we're only interested in evaluating the model on A particles, we apply a mask to the MSE loss function so that only the predictions for these particles are taken into account. Nonetheless, it is very important to clarify that we still pass the whole graph through the network, as the information from the other particles will be crucial to have good performance.

5 Results

Before training with the architecture described in section 4 on this dataset, we tried using the simpler, faster GCN. Nonetheless, given the limited information that it uses, we got very poor results that we don't consider worth mentioning in more detail (0.05 of R^2 on the validation set). Similarly, in a couple of times we tried to modify the architecture of the model by 1) using always the same message passing layer 7 times in a recurrent manner and 2) first updating the edges and then updating the nodes. Since these modification didn't improve the performance of the model we came back to the original architecture and further improved from there.

5.1 Model performance on the Bapst dataset

The first iterations of training yielded a lower performance than expected given the results shown in the paper. More specifically, we got a correlation of around 0.35 on the validation set by training for 50 epochs with a learning rate of 0.005 and a step learning rate scheduler with step size 4 and gamma of 0.3 (StepLR).

Even though this trial wasn't successful, the analysis of its learning curves (shown in figure 4), allowed us to find a way to improve the results. The sudden decrease in the performance of the model at epoch 23 suggested that the learning rate was too big, making the optimizer circle around and occasionally diverging.

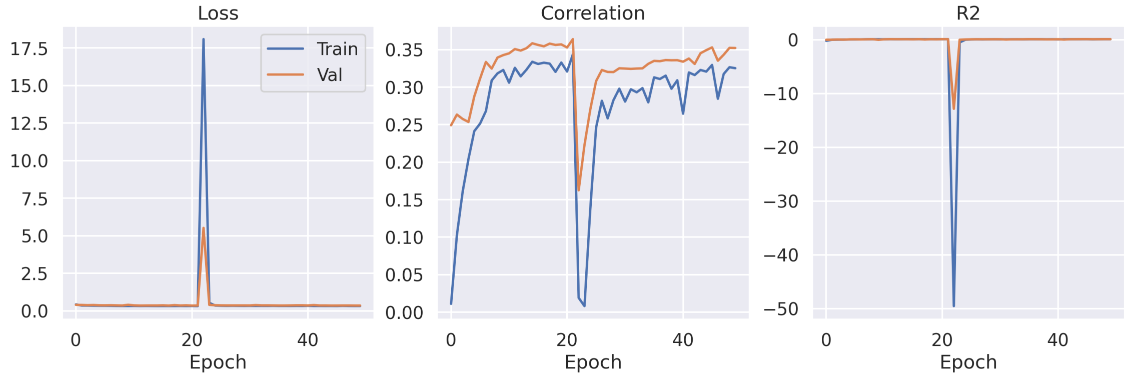


Figure 4: Learning curves for the first successful iteration. Epochs: 50, lr=0.005, StepLR(step size: 4, gamma: 0.3). It may seem unusual for the validation correlation to be higher than the training one, but it is because of the lower number of samples that it has. Indeed, this is a pattern that will be present in almost all of the results shown in this section.

With this in mind, we decided to decrease the learning rate to 0.0001, double the number of epochs, and increase the step size of the learning rate scheduler to 10. These changes naturally increased the training time (from 33 minutes to 1 hour and 6 minutes), but proved to be exactly what had to be done (see figure 5). The correlation on the validation set increased to around 0.44, a significant increase of 25.7% with respect to the previous setting. Most importantly, the new learning curves had the advantage of showing a longer tendency in the behavior of the loss, and that tendency showed that more improvement was to be gained by training for more epochs.

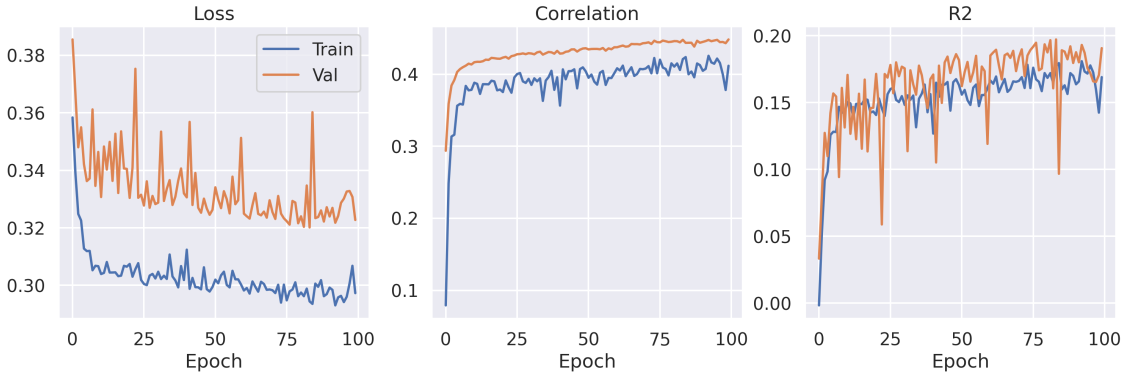


Figure 5: Learning curves for an improved iteration. Epochs: 100, lr=0.0001, StepLR(step size: 10, gamma: 0.3)

Therefore, we doubled again the number of epochs, going up to 200. As we were looking for more stability in the training, we also increased the batch size from 4 to 5 graphs (the largest value that the Kaggle infrastructure was able to handle before running out of memory). These changes also improved the results, taking the correlation up to 0.471. Again, the tendency we saw in the learning curves (that we do not show here for practical reasons), indicated that further training would yield more performance.

As such, we decided to go for 400 epochs, leaving the rest of the hyperparameters unchanged. Then, we realized that we could continue the training with the same model, without the need to start from scratch again. Hence, we further trained this model for 50, 100, and 400 more epochs, trying to get as much performance as possible (see figure 6). In the first stage we used a learning rate of 10^{-4}, which we then lowered to 10^{-5} for the 2nd and 3rd stages, and finally for the last stage we used 10^{-6}. We can confidently say that we reached a point very close to the top performance of the model with these data, as the learning curves show a plateau with low fluctuations.

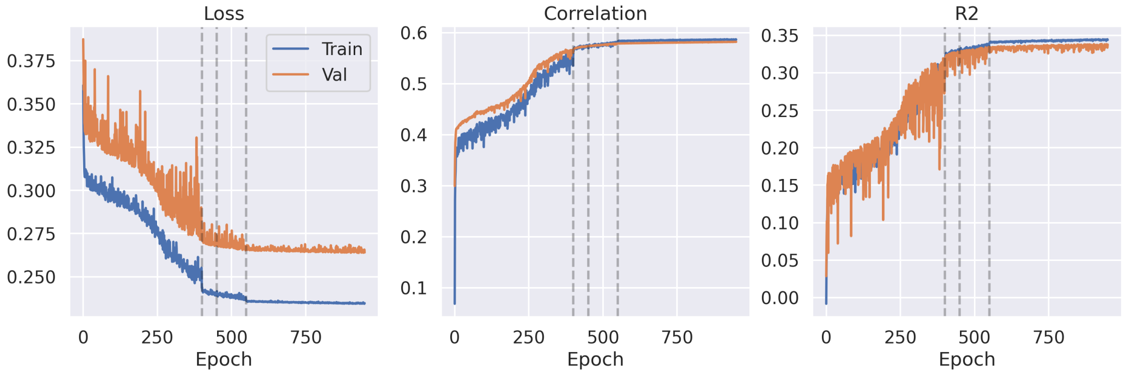


Figure 6: Learning curves for the final iteration of training. We trained the model in 4 stages of 400, 50, 100 and 400 epochs, marked by the vertical dashed lines.

6 Conclusions

As can be seen in this report, the TER involved several stages of learning new concepts, new techniques, and then applying them to try to replicate a research paper. In the first few weeks of the project we spent most of the time getting used with the theory, notations and architectures behind graph neural networks. We learned that in the context of GNNs you could have different predicitve goals, such as graph, edge or node level predictions (which could be regression, classification or even more complex problems). Similarly, we continued with a more hands-on approach by making our first step with the Pytorch geometric library. We implemented simple GNNs from a very low level and tested our models with a toy dataset (KarateClub), and a more serious one (ZINC).

Once our knowledge and skills with GNNs were sharp enough, we decided to tackle a recent problem that is still relevant and researched upon by very important institutions such as DeepMind. For this problem we had to implement the same architecture presented by this very same research laboratory, which proved to be a much more interesting task than the previous ones faced throughout the TER. For starters, the overall neural network involved several stages (encoder, GNN, decoder), and the use of skip connections. Moreover, we found this model to be quite unstable, leading in some cases to a constant prediction or temporary divergences in the optimization algorithm. This made us think more deeply about how to train a neural network, and at some point we manually supervised the training of the model, decreasing the learning rate when we didn't notice any more progress. In the end we found that reducing the learning rate, training for longer and using a learning rate scheduler were keys to getting significant gains in performance.

References

Bapst, V., T. Keck, A. Grabska-Barwińska, et al. (2020). "Unveiling the predictive power of static structure in glassy systems." In: *Nature Physics* 16, pp. 448–454. DOI: 10.1038/s41567-020-0842-8.

Fey, Matthias and Jan Eric Lenssen (2019). *Fast Graph Representation Learning with PyTorch Geometric*. arXiv: 1903.02428 [cs.LG].

Irwin, John J, Khanh G Tang, Jennifer Young, Chinzorig Dandarchuluun, Benjamin R Wong, Munkhzul Khurelbaatar, Yurii S Moroz, John Mayfield, and Roger A Sayle (2020). "ZINC20—a free ultralarge-scale chemical database for ligand discovery." In: *Journal of chemical information and modeling* 60.12, pp. 6065–6073.

Kipf, Thomas N. and Max Welling (2017). *Semi-Supervised Classification with Graph Convolutional Networks*. arXiv: 1609.02907 [cs.LG].

Zhou, Jie, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun (2020). "Graph neural networks: A review of methods and applications." In: *AI Open* 1, pp. 57–81.

A Modeling the ZINC dataset

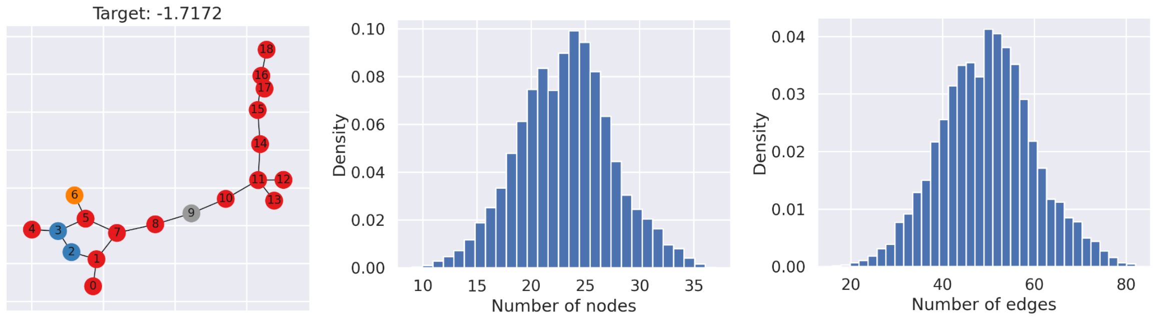
As part of the process to familiarize ourselves with GNNs and Pytorch Geometric we worked with a ”toy” dataset. As it is a different dataset from the original objective of the TER we decided to show the results in the appendix.

A.1 Description of the dataset

The ZINC database is a public database containing data about molecular compounds (Irwin et al. 2020). In it’s latest version, it has over 230 million samples. A smaller version of the database is available through Pytorch geometric, specifically designed to help users get started using the library modules and classes. In this version, the modeling task is a regression problem on the whole graph, where the target label is a quantity related to the solubility of the molecule. As we are working with molecules, the node features are the elements of the atoms, and the edge features are the type of bond between them (more on this dataset at the Pytorch geometric documentation).

A.2 Data Exploration

In order to better understand the data, we performed a thorough exploratory data analysis, and in figure 7 we show some of the most important results. We found that the distributions of nodes and edges are essentially gaussian, with means of 23.1 and 49.8 respectively. As for the elements present in the molecules, we found 21 distinct values (from 0 to 20). Element 0 is the most common one with 70%, followed by 2 and 1 with around 10%. We also found that there are 3 types of edges in the dataset, with around 74% being the first type, 25% the second type and about 0.2% the third one. From this information we can already hypothesize that the model will have problems generalizing to graphs where the least common elements and edges are present, and in a scenario where performance is crucial a preprocessing step to balance this problem should be performed.



(a) Sample from the ZINC dataset

(b) Histogram of the number of nodes in each graph

(c) Histogram of the number of edges in each graph

Figure 7: Data exploration graphs of the ZINC dataset

A.3 Results

The architecture chosen for this dataset was the Graph Convolutional Network. We employ several convolutional layers, and in between we apply activation functions so that we get more expressive power. Subsequently, after all the layers have been used, we aggregate the resulting node representations using the average. This will give us a feature vector for each graph, which we will then pass through an MLP.

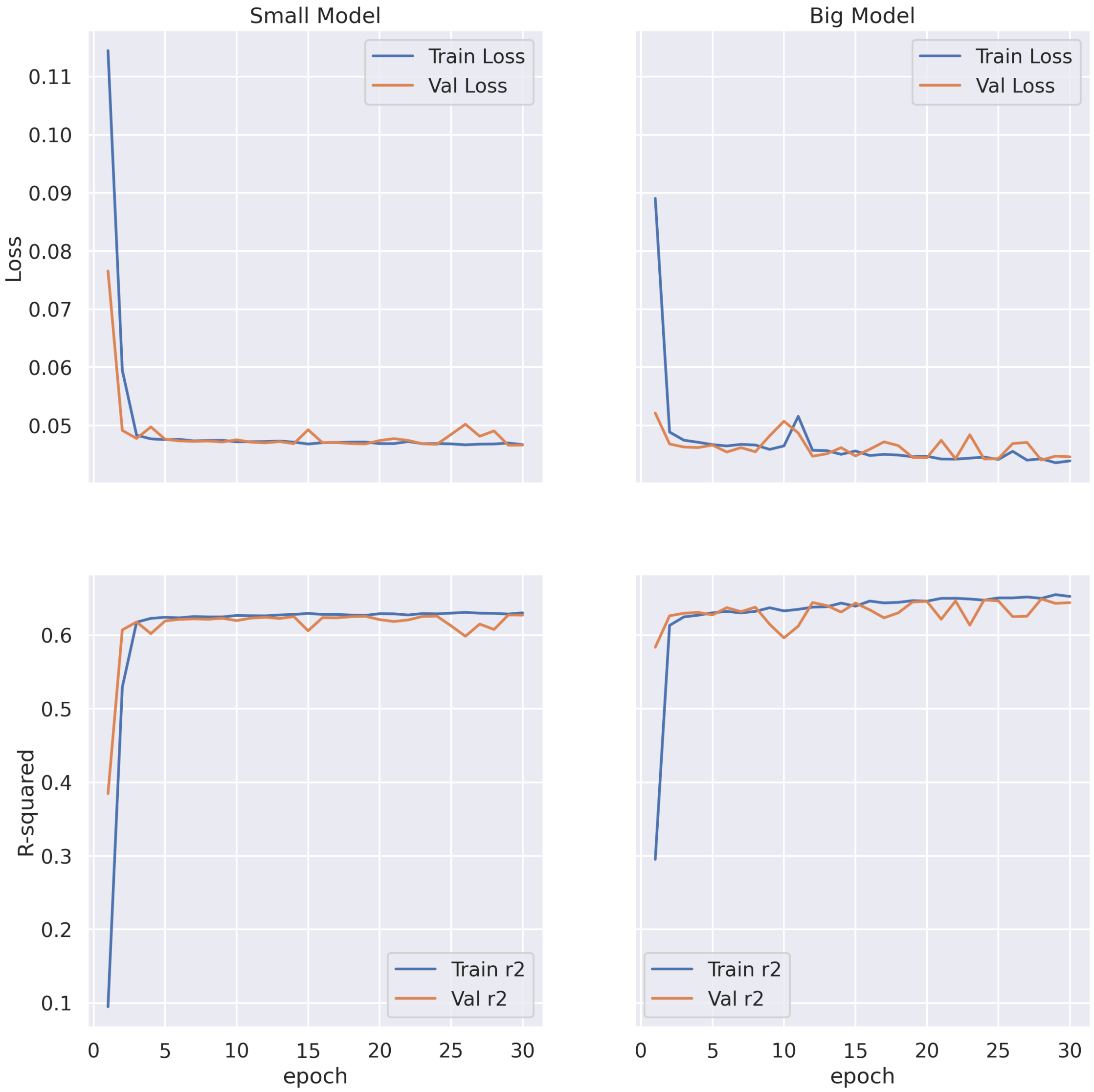


Figure 8: Loss and R^2 for train and validation datasets using the 2 different models

We did several iterations of hyperparameter tuning, focusing especially on the architecture of the network. We started with a very small model consisting of 2 convolutional layers with 16 neurons each, and an MLP with 2 hidden layers with 16 neuros as well. We then progressively increased the number and size of the layers, getting marginal improvements each time. In the final iteration, when we realized that bigger models were not having significantly better results, we trained a model with 5 convolutional layers (2 of 128 neurons and 3 of 64 neurons) followed by an MLP with 4 hidden layers (64, 64, 32, and 16 layers). This increase in model complexity was also accompanied by a decrease in the learning rate, going from 0.001 to 0.0005.

The best R^2 that we found with the smaller model was 0.627, which increased to 0.649 with the big one.