Report 3

Project topic

The objective of this project is to classify proteins into one of the 2 classes based on their structure.

Team

1. Roman Voronov

   ro.voronov@innopolis.university

Github Link

The progress of last weeks

During last weeks, I was coding the Graph U-Net model from this paper to apply it to the problem.

Next, I will describe the model and the training process

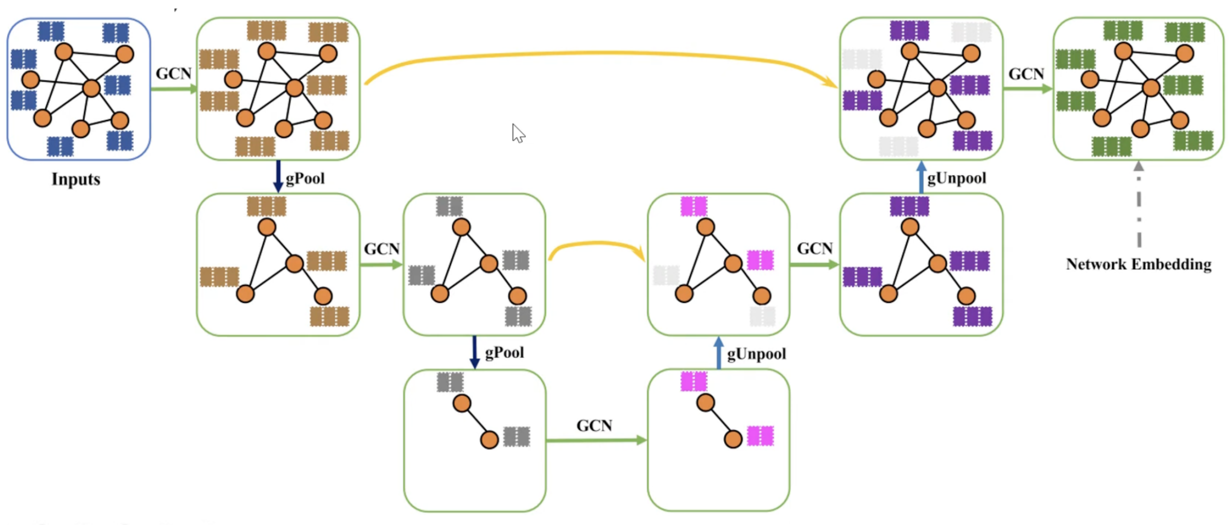
The structure of the whole model

The model has 3 parts.

1. We apply convolution to all nodes in order to change the feature dimensionality of nodes
2. We apply Graph U-Net to graph to get better embeddings for the nodes. This subpart is described later.
3. Classification part, where we aggregate node embeddings to get graph embedding and apply linear layers to do a classification

U-Net structure

We will use an encoder-decoder model to obtain embeddings called **Graph U-Net**. The model structure is very similar to the structure of U-Net for images. Below you can see its visualisation:



- Two consecutive encoder blocks with pooling layer and GCN (Graph Convolutional Network) layer
- Two consecutive decoder blocks with unpooling layer and GCN layer
- A final GCN layer that generates the final embedding of nodes
- Skip connections between layers on the same level

Description of layers:

1. **GCN**. This layer does message passing between neighbors nodes and transforms feature vectors to new feature vector by the following formula:

$$X_{l+1} = \sigma \cdot (\check{D}^{-0.5}\hat{A}\check{D} \cdot X_l \cdot W_l)$$

where $\hat{A} = A + 2I$ is used to add self-loops in the input adjacency matrix A *(Note that we have $2I$ to give more weight to node's own features)*, X_l is the feature matrix of layer l. The GCN layer uses the diagonal node degree matrix $\check{D}$ to normalize $\hat{A}$. W_l is a trainable weight matrix that applies a linear transformation to feature vectors. As a non linearity, authors proposed to use exponential linear unit

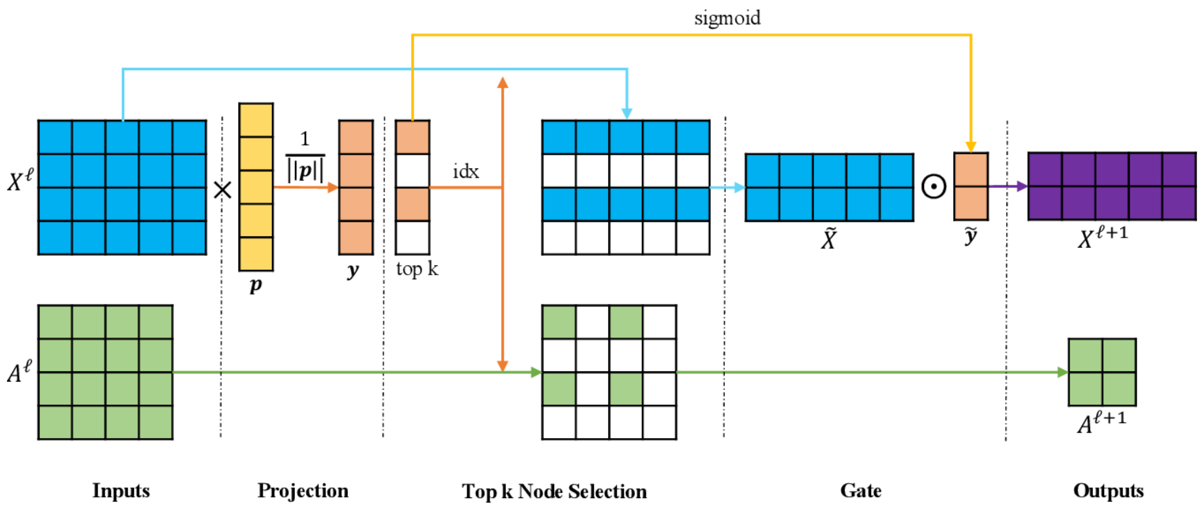
Note: this layer is already implemented in pytorch_geometric, so it is used.

All the next layers were implemented by scratch

2. **Pooling layer**. The purpose of this layer is to decrease the number of nodes. The layer has a trainable vector p as a parameter. Given a node i, we obtain scalar projection of its feature vector on vector p as $y_i = \frac{x_i p}{||p||} \cdot y_i$ measures how much information of node i can be retained when projected onto the direction of p.

   Next, the pooling is performed: we pick k nodes with highest scalar projection values and remove remaining nodes from the graph. By sampling nodes, we wish to preserve as much information as possible from the original graph.

   After such selection, the **gate stage** is coming. Here, we element-wise multiply the feature vectors of remaining nodes by $sigmoid(y_i)$, where y_i are obtained earlier. As a result, we have our new graph with different structure and new features for fewer nodes we had before the layer.



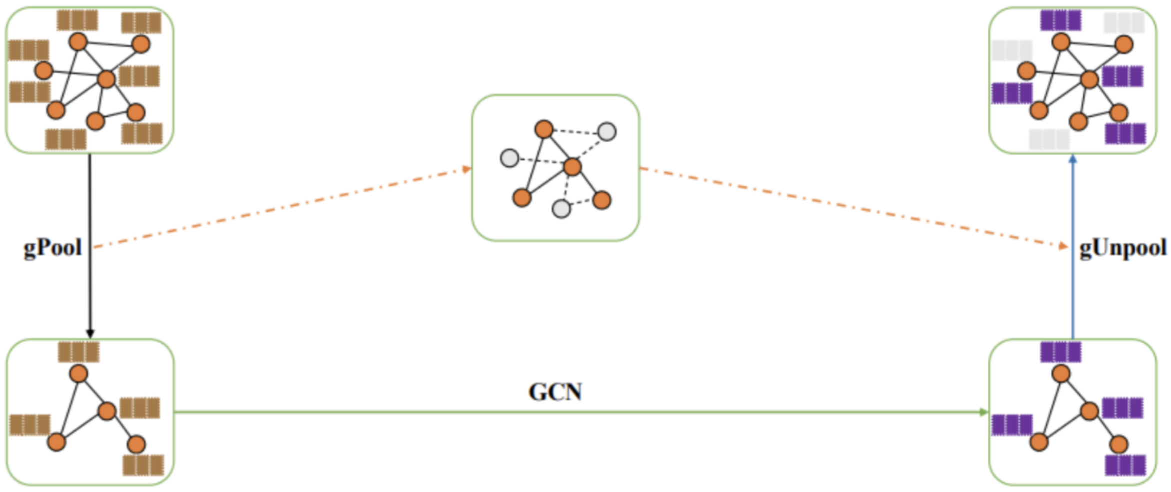
💡 **Very important thing about pooling layer:**
Since we remove some edges during the pooling layer, we lose connectivity. What is worse, some nodes can become isolated. To increase connectivity in a reduced graph, we use $k - th$ power of the graph. This operation builds links between nodes whose distances are at most k hops. **In the original work, authors used $k = 2$, so did I**

3. **Unpooling layer**. The purpose of this layer is to restore the original graph resolution. During the pooling layer, we record the locations of removed nodes and

during the corresponding unpooling layer we reconstruct them as can be seen in the figure below. Authors propose the following formula for reconstructing:

$$X^{l+1} = distribute(0_{N \times C}, X^l, idx)$$

Here, $0_{N \times C}$ is an empty feature matrix for **new** graph (zeros as features). We distribute row vectors from X^l into $0_{N \times C}$ matrix according to indexes in idx. Row vectors whose indices not included in idx remain zero

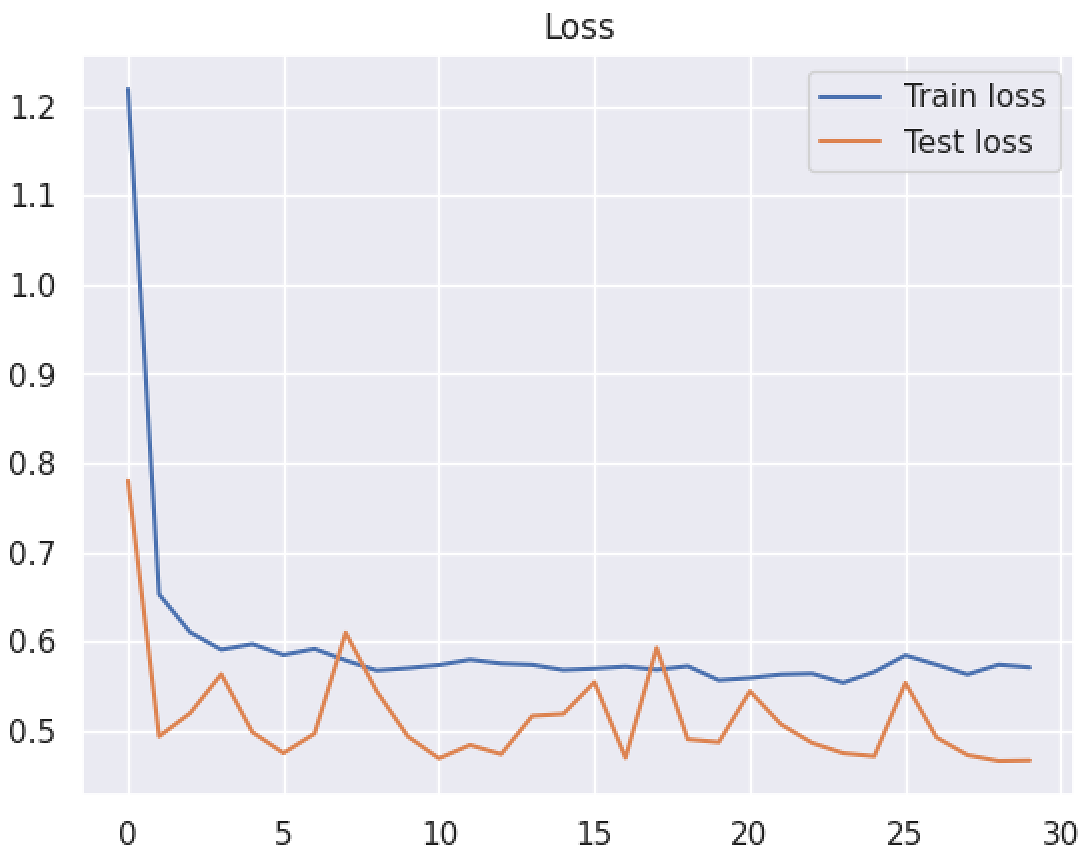


Training process

I used the same division to test and train sets as earlier in order to compare accuracies: stratified division 30 to 70. The best accuracy on the test set is 78% and 72% on train set. In the paper, authors got 77% that is close to my result. Remember that the previous model with use of only GCNs, random poolings and convolution with attention, I have achieved only 72% of accuracy of test. We see the huge increase (huge in case of this problematic dataset).

Batch size used is 64, optimizer used is Adam with default learning rate, the loss is **NLL Loss.**

Here is the loss plot



Plans for next weeks

I want to write an article on some resource about using Graph U-Net like habr or medium, because now the internet has only the paper and filthy official github page that is very hard to follow. This article will include the newbie-easy description of the model as here, and the guide in colab about the model structure and how to use it.

Or I will just code new model and test it. Still think…