Graph Neural Networks and Graph Convolutional Networks: an application on protein-protein interactions

Greta Grassmann, Lorenzo Spagnoli

Complex Networks Project
Bologna University

ABSTRACT

The Graph Neural Network based models, like the Graph Convolutional Network, have recently gained increasing popularity in various domains related to graph analysis, thanks to their power in modeling the dependencies between the nodes belonging to a graph. In a simplified approach each node of the graph is described by a set of features. The network uses the features from each node, as well as that of its neighbouring ones, to infer a state embedding which contains the information of the neighborhood of each node. In a more complete approach this information propagation between nodes (or *message passing*) which is generally guided by the connections alone, takes into account also the features of the edges along which it happens.
Graph Convolutional Networks are a subclass of these networks that uses convolution operators to aggregate the information from the neighborhood of a node. In this project we will study their application for protein interface prediction: in this case each protein is represented by a graph, where each node corresponds to a residue (or amino acid). The proposed method produces a new embedding for the residues of the ligand and receptor protein, and learns to classify them as interacting or non interacting pairs. Starting from a basic approach that does not consider the internal structure of the proteins, which is to be intended as it neglects the edges' features, we tried to improve its classification precision. Firstly, we added the edge's features, but then we focused on trying to improve the performance of a model without them by changing some characteristics of the network architecture. In particular, we compared the use of different activation functions and the change of the way in which the weights for the ligand and receptor proteins' residues are learned.

Keywords: Graph Neural Network, Graph Convolutional Network, Protein-protein interaction.

1 INTRODUCTION

Many underlying relationships among data in several areas of science can be naturally represented in terms of graph structures. The simplest kinds of graphs include single nodes and sequences, but in several applications the information is organized in more complex structures such as trees, acyclic graphs or cyclic graphs.
Traditional machine learning applications apply to the graph structured data a pre-processing phase in which the graph structure of the information is mapped in low-dimensional vectors (*graph embedding*). However, these approaches are computationally inefficient and cannot be generalized: since they usually go from a discrete structure to a continuous space, they are trained on specific structures and the same embedding won't work if the graph itself is evolving in time (e.g. new nodes or edges being created with certain probability) nor will it work on a graphs built with different properties. Moreover, important information (like the topological dependency of information on each node) may be lost during this stage: in order to incorporate graph structured information in the data processing step, it is necessary to encode the underlying graph structure using the topological relationships among the nodes.
Convolutional Neural Networks (CNNs) are able to extract multi-scale localized spatial features and compose them to construct highly expressive representation, but they can only operate on data which has a "regular data structure" (e.g. images, which are a grid of pixel values). When the data has non Euclidean distance the CNN approach is unfit for analysis.

Often graphs have interesting properties that are invariant under permutations of the node indices.This is another reason due to which CNNs, that implicitly give order in the feature aggregation phase, are not a desirable tool in the context of graph analysis. Another set of commonly used architectures contains Recursive Neural Networks and Markov Chains: both of these can be applied to graph problems yet they are somewhat inefficient in performance. There is a deep learning based method, however, that operates on graph domain which generalizes these two approaches: the Graph Neural Network (GNN). In particular GNNs are an evolution of Recursive Neural Networks in that they can process a broader class of graphs and do not need a preprocessing step, while compared to random walk theory they add a learning algorithm and enlarge the class of processes that can be modeled.
GNNs can deal directly with graph structured information in most of the practically useful types of graphs (like acyclic, cyclic, directed and undirected) by implementing a function that maps a graph G and each one of its nodes into a s-dimensional Euclidean space. Thanks to this feature GNNs maintain the invariant nature of the data by basing the feature aggregation step on the structure of the graph itself .
Each edge in GNNs represents the information of dependency between two nodes and guides the propagation, instead of being considered just as a feature of the nodes like in standard neural networks. The target of a GNN is to learn a state embedding which contains the information of the neighborhood of each node. This node state embedding is an s-dimensional vector and can be used to produce an output such as the node label. In simple terms, GNNs update each node using a weighted sum of the states of their (k-hop)neighborhood, where k depends on the number of layers in the network [1].
The GNNs were first proposed in (1) in 2009, with an algorithm which is often regarded as the original GNN and is here described in Section 2.
Since then, several modifications have been introduced, regarding for example the propagation step of the original GNN model. Each of them uses different aggregating function to gather information from the neighbors of a node and specific updaters.

In this regard, a particular interest is increasing in generalizing the convolutional approach of CNNs to the graph domain. CNNs are based on local connections, shared weights and the use of multi-layer; all these characteristics are fundamental for solving problems in the graph domain, because graphs are typical locally connected structures, shared weights reduce the computational cost compared with traditional spectral graph theory and multi-layer structures can deal with hierarchical patterns. Therefore Graph Convolutional Networks (GCNs) have been proposed. This kind of network is discussed in more details in Section 3.2, and for this project will be applied to the study of protein-protein interactions.

2 THE ORIGINAL GRAPH NEURAL NETWORK

GNN is a type of neural network which directly operates on the graph structure and is based on an information diffusion mechanism.
A graph G is a pair (N,E), where N is the set of nodes and E is the set of edges; the nodes represent objects or concepts, and the edges the relationships among them. Each concept is defined by its features and the concepts related to it. Thus we can attach a state $x_n \in \mathbb{R}^s$ to each node n which is based on the information contained in its neighborhood. This state contains a representation of the concept denoted by that node and can be used to produce as output o_n a decision about the concept. G is processed by a set of units, each one corresponding to a node of the graph, which are linked according to the graph connectivity. These units update their states and exchange information until they reach a stable equilibrium. In order for this stable equilibrium to always exists and be unique the diffusion mechanism has to be constrained.

[1] Here k-hop neighbourhood indicates the set of nodes that can be reached, starting from a center node, in k steps. Each layer of the GNN can be thought of as adding a step to the diameter of the neighbourhood, which means that a k-layered convolutional neural network will aggregate, for each node, the information derived from the k-hop neighbourhood defined by the graphs connectivity.

2.1 The model

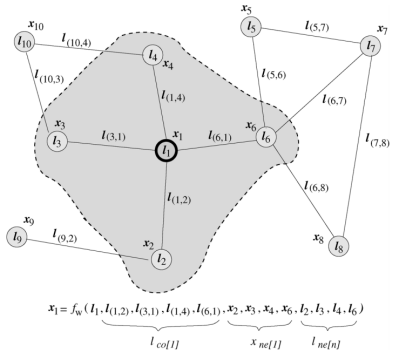

Figure 1. Graph and the neighborhood of a node: the state x_1 of node 1 depends on the information contained in it.

The state and the output referred to a neuron n are defined respectively as:

$$x_n = f_w\big(l_n, l_{co[n]}, x_{ne[n]}, l_{ne[n]}\big) \qquad o_n = g_w\big(x_n, l_n\big). \tag{1}$$

Where l_n denotes the features (or labels) of node n, $l_{co[n]}$ the features of the edges connecting with n, $x_{ne[n]}$ the embedding of its neighboring nodes and $l_{ne[n]}$ their features. The function f_w is the *local transition function* that projects these inputs onto a s-dimensional space and expresses the dependence of the node on its neighborhood. g_w is the *local output function* that describes how the output is produced.

The dependence on the neighboring nodes as a set $ne[n]$ represents the intention to learn a function that is order-independent.

Neighborhood: Different notions of neighborhood can be adopted. For example we could remove the labels $l_{ne[n]}$ since their information is implicitly contained in $x_{ne[n]}$, or include in the neighborhood nodes that are two or more links away from n. In the following we will consider as neighbors the nodes connected to n by an arc and base our discussion on the form defined by Equation 1.

Directed graph: When dealing with directed graphs the function f_w can take as input also a variable that represents the direction of each arc. The following discussion still applies for undirected and directed links and even for graphs with a mix of them.

Positional graph: For positional graph, the positions of the neighbors must be given as an additional input to f_w. This can be achieved if the information contained in $x_{ne[n]}$, $l_{co[n]}$ and $l_{ne[n]}$ is sorted according to neighbors' positions and is properly padded with null values in correspondence to non-existing neighbors. However, it is useful to replace f_w in Equation 1 with

$$x_n = \sum_{u \in ne[n]} h_w\big(l_n, l_{(n,u)}, x_u, l_u\big). \tag{2}$$

Where $ne[n]$ is the set of the neighbors of n and $l_{(n,u)}$ are the labels attached to the edge between each of them and n. h_w is a parametric *transition function*, which is not affected by the position and the number of nodes. This is the so called *non-positional form*.

Different kinds of objects: Equation 1 describes a particular model where all nodes share the same implementation. In general the *transition and output functions* and their parameters may depend on the node n: to represent different kind of objects different mechanisms may be needed. In this case each kind of nodes k_n has its own *transition function* f^{k_n}, *output function* g^{k_n} and a set of parameters w_{k_n}. Equations 1 can then be rewritten as:

$$x_n = f^{k_n}_{w_{k_n}}\big(l_n, l_{co[n]}, x_{ne[n]}, l_{ne[n]}\big) \qquad o_n = g^{k_n}_{w_{k_n}}\big(x_n, l_n\big). \tag{3}$$

In the following discussion Equation 1 will be considered.

Next, we can construct the vectors x, o, l and l_N by stacking all the states, all the outputs, all the labels and all the node labels respectively. Then Equation 1 can be rewritten in a compact form:

$$x = F_w(x,l) \qquad\qquad o = G_x(x,l_N). \tag{4}$$

Where the *global transition function* F_w is a stacked version of N instances of f_w, while the *global output function* G_w is a stacked version of N instances of g_w.
We are interested in the case where x and o are uniquely defined, so that Equations 4 define a map which takes the graph G as input and returns an output o_n for each node: the Banach fixed point theorem then guarantees the existence of a unique solution if F_w is a contraction map[2] with respect to the state. As already anticipated this property can be enforced by an appropriate implementation of the *transition function*.

2.2 Implementation

In order to implement the GNN model we need a method to solve Equations 1 (described in Section 2.2.1), a learning algorithm to adapt f_w and g_w, or in other words to estimate the parameters w, using examples from the training data set (described in Section 2.2.2) and an implementation of f_w an g_w (described in Section 2.2.3).

2.2.1 Computation of the state

We can apply the Banach fixed point theorem and rewrite Equation 1 as an iterate update process. Such operation is referred to as *message passing* or *neighbouring aggregation*. To compute the state the following iterative scheme must be computed:

$$x(t+1) = F_w(x(t),l). \tag{5}$$

Where $x(t)$ is the t^{th} iteration of x. This dynamical system converges exponentially fast to the solution of Equation 4 for any initial value $x(0)$. Consequently, the outputs and the states can be computed by iterating:

$$x_n(t+1) = f_w\big(l_n, l_{co[n]}, x_{ne[n]}(t), l_{ne[n]}\big) \qquad\qquad o_n(t) = g_w\big(x_n(t), l_n\big). \tag{6}$$

As already anticipated, this can be interpreted as the representation of a network consisting of units which compute f_w and g_w, that will be called *encoding network*.

2.2.2 The learning algorithm

Learning in GNNs consists of estimating the parameters w of f_w and g_w looking at a set of given training examples. With the target information t_n for a specific node n for the supervision, the *Least Absolute Deviations loss function* (L1$loss$) can be rewritten as:

$$L1loss = \sum_{i=1}^{p} (t_i - o_i) \tag{7}$$

where p is the number of supervised nodes.
The learning algorithm is based on a gradient-descent technique and is composed of the following steps:

1. The states $x_n(t)$ are iteratively updated by Equation 6 until a time T. They approach the fixed point solution of Equation 4: $x(T) \approx x$.
2. The gradient of Equation 7 is computed.
3. The weights w are updated according to the gradient computed in the preceding step, until the output reaches a desired accuracy or some other stopping criterion is achieved.

[2] F_w is a contraction map if there exists μ, $0 < \mu, 1$, such that $||F_w(x,l) - F_w(y,l)|| \leq \mu||x-y||$ holds for any x, y.

2.2.3 Transition and Output function implementation

The implementation of g_w does not need to fulfill any particular constraint; on the other hand, as already anticipated, the implementation of f_w determines the number and the existence of the solutions of Equation 1. The GNN is based on the assumption that the design of f_w is such that F_w is a contraction map with respect to the state x.

2.3 Limitations

The original proposal of GNN has several limitations, pointed out in (2):

- Firstly, it is inefficient to update the hidden states of nodes iteratively for the fixed point. If the assumption of fixed point is relaxed, it is possible to design a multi-layer GNN to get a stable representation of a node and its neighborhood.
- GNN uses the same parameters in the iteration: varying them in each different layer serves as a hierarchical feature extraction method.
- It uses edge information in a sub-optimal way: the information relative to edges should be used separated form that of nodes and not lumped together. For example in a knowledge graph the message propagation through different edges should depend on the type and features of the edge itself.

Several variants of this original model have been proposed to extend its representation capability. In the following sections one will be analyzed in detail, the Graph Convolutional Network (GCN), with the presentation of a particular cases: protein interface prediction (discussed in Section 3).

3 PROTEIN INTERFACE PREDICTION USING GRAPH CONVOLUTIONAL NETWORKS

3.1 Biological overview

Proteins are large biomolecules (or macromolecules) consisting of one or more long chains of amino acid residues. Proteins perform a vast array of functions within organisms, including catalysing metabolic reactions, DNA replication, responding to stimuli, providing structure to cells and organisms, and transporting molecules from one location to another. Proteins differ from one another primarily in their sequence of amino acids, which is dictated by the nucleotide sequence of their genes, and which usually results in protein folding into a specific 3D structure that determines its activity. They perform their function through a complex network of interactions with other proteins.

The most commonly implemented machine learning methods to predict the interfaces between them are based on hand-crafted features. The generalization of the convolution operator to this kind of graph data has also been explored.

3.2 Graph Convolutional Networks

Graph Convolutional Networks are a typical structure of the GNN model. They learn to integrate node features based on labels and link structures, by generalizing the standard notion of convolution over a regular grid (representing a sequence or an image) to convolution over a graph structures. They can be thought of as fully-connected neural networks plus a neighborhood aggregation component (3). The former computes a non-linear feature projection, whereas the latter mixes the feature of each node with those of its neighbors. The objective is to design a convolutional operator that can be applied to graph without a regular structure, without imposing a particular order on the neighbors of a given node. The convolution is performed by the convolutional layers for which a number of filters must be specified. The filters are what actually detects specific patterns; the deeper the network goes, the more sophisticated patterns these filters can pick-up.
The GCN has also other kind of layers, that we are going to describe in Section 4: to make useful predictions, it will need for example the usual dense, pooling layer structure used for CNNs.

GCNs can be applied on structured objects that can naturally be modeled as graphs, for example for protein interface prediction (4), a problem described in Sections 3.3 and 3.4. The resulting structure information

is useful for predicting a variety of property of proteins, including their function and interaction with other proteins or with DNA or RNA.
By performing a convolution over the local neighborhood of a node, and stacking multiple layers of convolution we can learn a latent representation that integrates information across the graph that represent the 3D structure of a protein of interest. Then a follow-up neural network architecture combines this learned features across pairs of proteins and classifies pairs of amino acid residues as part of an interface or not.
We remark again that the graph operator described in the following Section maintains the structure of the graph, and does not need downsampling of the inputs or pooling: this is necessary for the protein interface prediction problem, where we classify pairs of nodes from different graphs rather than entire graph.

3.3 Graph convolution on protein structures

Each residue in a protein is a node in a graph. The neighborhood of that node used in the convolutional operator is the set $ne[n]$ of k closest residues as determined by the mean distance between their atoms in the protein structure. Each node has features computed from its amino acid sequence and structure. Edges have features describing for example the relative distance and angle between residues.

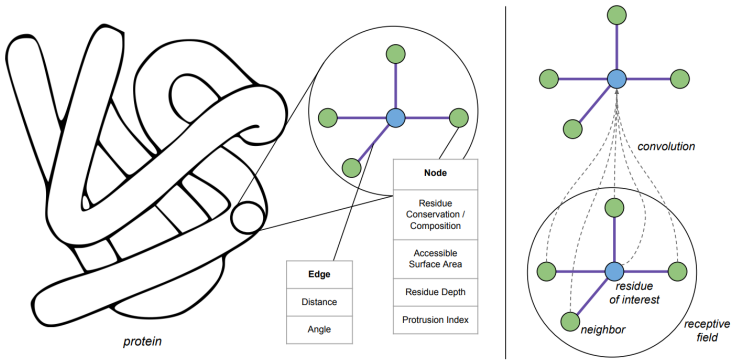

Figure 2. Schematic description of the convolution operator which has as receptive field a set of neighboring residues, and produces an activation which is associated with the center residue.

A possible convolutional operator could be defined so as to provide the following output of a set of filters in a neighborhood of a node of interest n, that we are going to call *center node*:

$$x_n = \sigma\Big(W^c l_n + \frac{1}{|ne[n]|}\sum_{j\in ne[n]} W^N l_j + b\Big). \tag{8}$$

The notation already introduced in Section 2 is still valid. Moreover, we define W^c as the weight matrix associated with the *center node*, W^N as the weight matrix associated with its neighbors and b is a vector of biases, one for each filter. σ is a non-linear activation function.
In order to provide for some differentiation between neighbors, we can incorporate the features of the edges between the *center node* and its neighbors by introducing W^E as the weight matrix associated with edge features:

$$x_n = \sigma\Big(W^c l_n + \frac{1}{|ne[n]|}\sum_{j\in ne[n]} W^N l_j + \frac{1}{|ne[n]|}\sum_{j\in co[n]} W^E l_j + b\Big). \tag{9}$$

Multiple layers of these graph convolution operators can be used: in this case the model learns features that characterize the graph at increasing level of abstraction and the information is allowed to propagate through the graph across region of increasing size.

3.4 Pairwise classification architecture

To solve the protein interface prediction problem we need to classify pairs of nodes from two separate graph representing the ligand and the receptor protein. The examples are composed of pairs of proteins, one receptor and one ligand: the data are a set of N labeled pairs $\{((lig_i, r_i), y_i)\}_{i=1}^{N}$, where lig_i is a residue (node) in the ligand, r_i is a residue (node) in the receptor protein and $y_i \in \{-1, 1\}$ is the associated label that indicates if the two are interacting or not.

As schematically shown in Figure 3, each neighborhood of a residue in the two proteins is processed using one or more graph convolutional layers, that learn their feature representations. The weights of the two "legs" of this network architecture are usually shared. The activations generated by the convolutional layers are merged by concatenating them in a representation of residue pairs. Finally these resulting features are passed through one or more regular dense layers before classification.

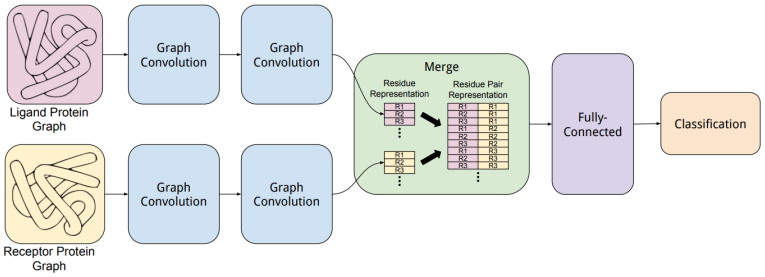

Figure 3. Schematic description of the pairwise classification architecture.

Since the role of ligand and receptor is arbitrary, the scoring function should be learned independently of the order in which the two residues are presented to the network. For this reason the final evaluation will be performed by taking into account both a ligand-receptor and a receptor-ligand structure.

3.5 Dataset

The data can be found in (5) and is composed of python tuples in the form $(train\_list,\ test\_list)$, where each element of $train\_list$ and $test\_list$ is a dictionary containing data data for a single protein complex. These data are derived from protein complexes in the Docking[3] Benchmark Dataset (DBD) version 5.0, which is the standard benchmark dataset for assessing docking and interface prediction method. These complexes are selected subsets of structures from the Protein Data Bank (PDB). The PDB is a database for the three dimensional structural data of large biological molecules, such as proteins and nucleic acids, typically obtained by X-ray crystallography, Nuclear Magnetic Resonance (NMR) spectroscopy, or increasingly cryo-electron microscopy. These structures contain the atomic coordinates of each amino acid residue in each protein, that typically ranges in length from 29 to 1979 residues (with a median of 203.5). Each node and edge in the graph representing a protein has features associated with it that are computed from the protein's sequence and structure. The numbers of node in a neighborhood is fixed in the dataset as 20.

Protein sequence can indicate the propensity of a residue to form an interface because each amino acid exhibits unique electro-chemical and geometrical properties. The level of conservation of a residue[4] in alignments against similar proteins provides information too: surface residues that participate in an interface tend to be more conserved. The identity and conservation of a residue are quantified by 20 features that capture the relative frequency of each of the 20 amino acids in alignments to similar proteins. Each node has a certain number of features computed from the protein structure; for example surface accessibility, a measure of its protrusion, its distance from the surface and the counts of amino acids within $8\mathring{A}$ in two directions.

The primary edge feature is a Radial Basis Function[5] (RBF) of the distance between two features, calculated as the average distance between their atoms. To know the relative orientation of the two residues,

[3] Docking aims at predicting the structure of a complex given the 3D structures of its components.

[4] Conserved sequences are identical or similar sequences in nucleic acids (DNA and RNA) or proteins. Conservation indicates that a sequence has been maintained by natural selection.

[5] A radial basis function (RBF) is a real-valued function whose value depends only on the distance between the input and some fixed point, either the origin, so that $\phi(x) = \phi(||x||)$, or some other fixed point so that $\phi(x) = \phi(||x - c||)$.

the angle between the normal vectors of the amide plane of each residue is calculated as well.
All node and edge features are normalized, except for the residue conservation features, which were standardized.
For the test set the 55 more recently added complexes were used. The remaining were divided into training (140 complexes) and validation (35 complexes) sets in the case of (4), while in our implementation all 175 were used for the training alone.
The validation set was used to select the best possible feature representations and model hyperparameters, like the edge distance feature RBF kernel standar deviation, the negative to positive example ratio, the number of convolutional layers, the number of filters, the neighborhood size, the pairwise residue representation, the number of dense layers after merging, the optimization algorithm, the learning rate, the dropout probability, the minibatch size and the number of epochs. The resulting values are the one we used.

Because in any given complex the majority of residue pairs do not interact, the negative examples were downsized in the training dataset to obtain a 10:1 ratio of negative and positive examples. Positive examples are residue pairs that participate in the interface, negative examples are pairs that do not.

4 OUR IMPLEMENTATION

To implement a simple and easy to understand graph convolutional deep learning method to predict interfaces between protein residues, we started from the code proposed in (6). This is a simplified version of the code proposed in (7) by the authors of (4): both are based on a deep learning architecture that uses graph convolutions, but the code here studied is not complicated by the internals of protein structure in that it does not take into account the edges' features.
Our implementations and modifications of the original code can be found in (8).
The studied data are the ones described in Section 3.5. The values like the number of filters (256), the size of the minibatches (128 paired examples), as well as all the following constant parameters (like the number of dense layer chosen as two or the negative to positive example ratio as $1 : 10$) were inspired by the result of a validation set performed by the original authors of (4). In this work no extensive search over all the space of possible feature representation and model hyperparameters will be done, however three particular cases will be analyzed:

- The difference between a model which includes the edges' features and one that does not.
- The difference between the implementation of a *tanh* function in place of a ReLU as activation functions for the convolutional layers.
- The difference between an architecture in which the two "legs"[6] of the network share the weights and one in which they do not.

[6]Legs has to be intended as the two convolutional branches of the whole network, one dedicated to ligands, the other to receptors

4.1 Basic implementation

We started from the original version, which implements, with the ReLU function as σ, Equation 8: the graph convolution does not consider the edge features. The code implements the network architecture showed in Figure 4.

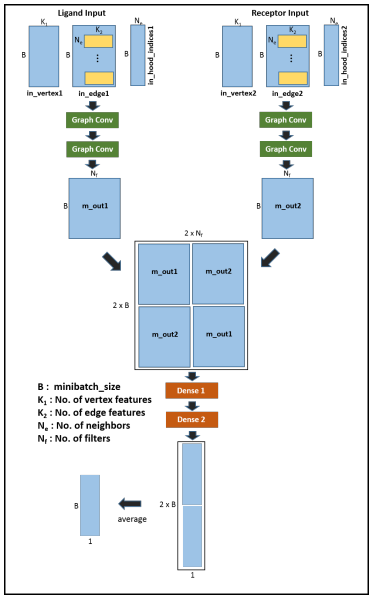

Figure 4. Schematic representation of the network architecture.

The weights matrices are tensors of dimension $(number\ of\ features) \times (number\ of\ filters)$ elements that multiply the data presented in a tensor with dimension $(number\ of\ nodes) \times (number\ of\ feature)$. Ligand end receptor residues are fed separately into the two "legs" of the network's architecture that are composed by two convolutional layers. To each residue (or node) in the minibatch is associated one value for each filter. The two matrices of dimension *(minibatch size)* $\times$ *(number of filters)* given as ouput are unified in a single matrix of dimension *2·(minibatch size)* $\times$ *2·(number of filters)*; in this way both the ligand-receptor and receptor-ligand pairs are considered, as required in Section 3.4.
This matrix is than fed to two dense layers. The first one acts almost as a square matrix *2·(number of filters)* $\times$ *2·(number of filters)*, while the second one has size *2·(number of filters)* $\times$ *1s*. The output is then a *2·minibatch size* vector: we have associated to each couple combination a single value, given by a combination of the different filters'values.
The classification is performed on an average of the outputs of the fully-connected dense layers for the ligand-receptor and receptor-ligand pairs. For each residues pair a label $+1/-1$ indicates if there is interaction or not respectively. Not all pairs of residues are present in the labels because of the downsampling described in Section 3.5.
It should be noted that in this case the weights W^c and W^N and the filters b are shared, on every layer, between the two "legs" of the architecture[7].

The models resulting from the training are then used for the testing, which as already anticipated is performed on 55 proteins pairs. For each model, at the end of training the average loss (defined as the average of the Categorical Cross-Entropy loss of each minibatch), is saved together with the weights resulting from the training. After the testing of a model, the average of the Categorical Cross-Entropy loss of the 55 proteins is saved as average loss.

4.2 Edges features addition

To understand if this model prediction can be improved, the original code has been modified by adding a matrix W^E that takes into consideration the edges features, so that Equation 9 is now implemented in place of Equation 8. To compare the two approaches we start by looking at their average loss function for increasing number of epochs in the two training. The result is shown in Figure 5.

The average loss of the model without the edges' features is always lower and decreases faster too. This is in line with the expectations, since it has far less parameter than the model that considers the edges: a model with less parameters can be tuned more rapidly.
With regards to the model with less information (without edges) being more precise, this may be explained by the fact that we did not perform a validation to find some new parameters more suitable for the new model that includes the edges. It's also possible that the aforementioned slow variation hides a better performance with increased number of epochs. We've decided, partially forced by the limitations of our computers, that it wasn't worth the eventual training time to discover if and when this hypothesis is verified.

[7] This means that each layer has a single set of parameters, trained for ligand and used then on receptor as well

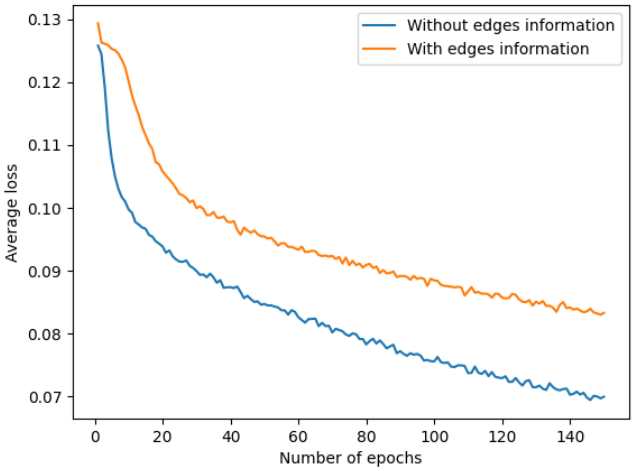

Figure 5. Average loss for increasing number of epochs for both the training in which edges features are and are not considered in the convolution.

As a next step, we compare the results of the two approaches for the testing. Figure 6 shows the comparison between the two average losses (on the top) and the values relative to the area under the Receiver Operating Characteristic (ROC) curve (on the bottom) for increasing number of epochs. Although they do not differ much in their behaviour, two observations arise looking at the average losses for the two kind of training: the first one is that for $\#epochs < 60$ the average loss of the models with edges information is more stable and does not present high peaks. The second one is that for $\#epochs > 60$ it presents values higher than the ones of the models without edges information.
The stability and the initial lower values when there is information from the edges depend on the higher number of parameters: a model trained with more parameter yields more stable results when presented with new examples. On the other hand, since the model with edges has more parameters rather than it's opponent, it's plausible that the worse performance in testing is due to some combination of early overfitting of the former and slow convergence of the second. Since a good implementation of a model that takes into account the edges' features was already performed in (7), in the following Sections the research for new parameters more suitable for the edges-including model will not be performed. Instead, we will present some new attempts to improve the results yielded by a model that does not have access to the edges' information.

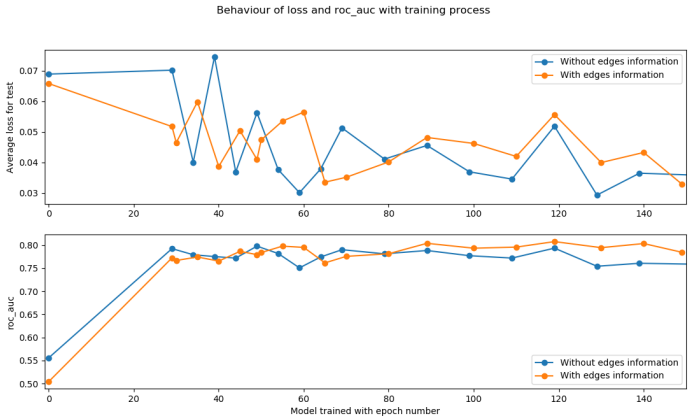

Figure 6. Top: average loss in testing models built with increasing number of epochs. Bottom: values relative to the area under the ROC curve in testing models with increasing number of epochs. Models both take into account and don't take into account edge features.

4.3 Activation function

As a first attempt we swap the ReLU activation function σ with a *tanh* function in the two convolutional layers. The ReLU activation function is still implemented in the dense layers, since the output on which we want to perform the classification has to be a positive number.
Figure 7 shows the average loss for the training with increasing number of epochs of the models with $\sigma =$ ReLU and $\sigma = tanh$.

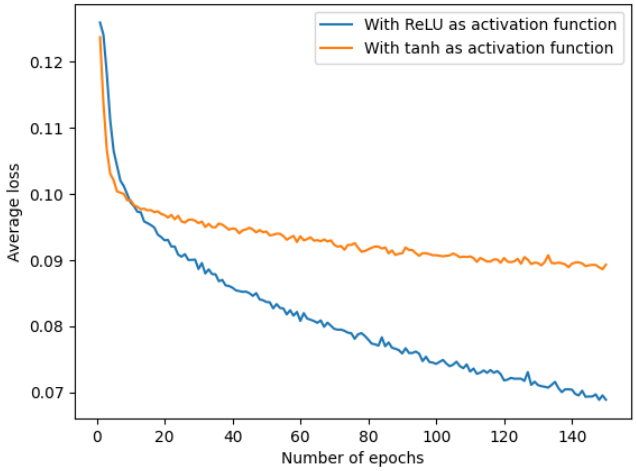

Figure 7. Average loss of the training for increasing number of epochs, for the model with a ReLU activation function and the one with a *tanh* function.

Looking at this plot, it appears that the performance of the model with $\sigma = tanh$ is worse than the one of the original model. It's evident that the average loss for *tanh* not only has a much slower rate of change but also is almost double that of a model with the ReLU. Furthermore it should be noted that the computation time is consistently longer for the *tanh* model. To have a more complete view we can look at the comparison of the two models' testing, showed in Figure 8.

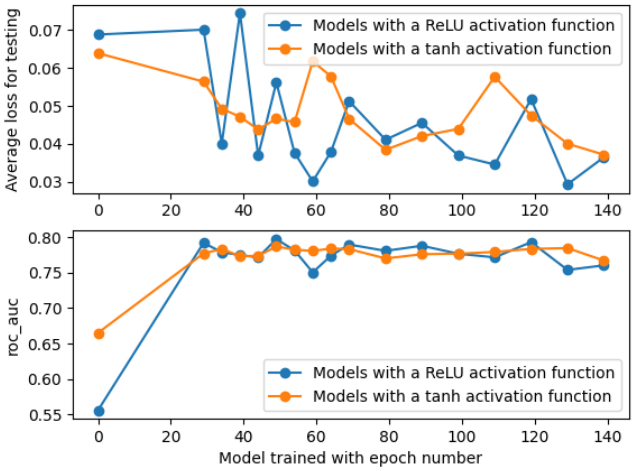

Figure 8. Top: average loss of the testing for increasing number of epochs, for the model with a ReLU activation function and the one with a *tanh* function. Bottom: area under the ROC curve relative to the testing for increasing number of epochs, for the model with a ReLU activation function and the one with a *tanh* function.

It can be noted that the average loss of the testing for the models with a *tanh* activation function trained for a number of epochs up to 60 is more stable and has a lower mean value. The better performance could be explained by the comparison between the shapes of a *tanh* and a ReLU function: given the same inputs, a ReLU function can produce a much higher output compared to the *tanh*. As a consequence, a model with a *tanh* as activation function will have smaller parameters and their values will also change slower(and so it will be more stable). It's also reasonable to assume that, since in the end all parameters are squished in the [-1,1] range, there is a set of sub-optimal yet acceptable parameters which are in the domain in which *tanh* stagnates. This would explain why the performance for low number of epochs is better than the ReLU activated network.

4.4 Independent weights

Next, we test if the implementation of non-shared weights between the two "legs" of the architecture leads to an improvement in the model predictions. Indeed the two "legs" process different kind of proteins, respectively a receptor and a ligand. Since they have different functioning, it would makes sense for them to have different morphology and as a consequence for the corresponding graphs to have different weights.
Figure 9 shows the average loss of the training for increasing number of epochs for the original model and one that has independent weights.

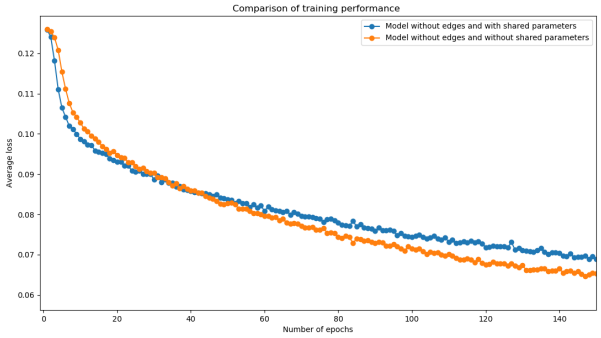

Figure 9. Training performance for training procedure with and without sharing parameters for the two legs of the network.

Looking at the plot is seems that the improvement is rather small, but it must be remembered that for the new model with independent weights the hyper-parameters that best suited the original model have been used. It seems reasonable to assume that a better choice of hyper-parameters, which could be obtained with an ex novo validation procedure, might lead to even better performances and predictions. Figure 10 shows instead the average loss and the value of the integral of ROC curve relative to a set of models trained with increasing number of epochs.

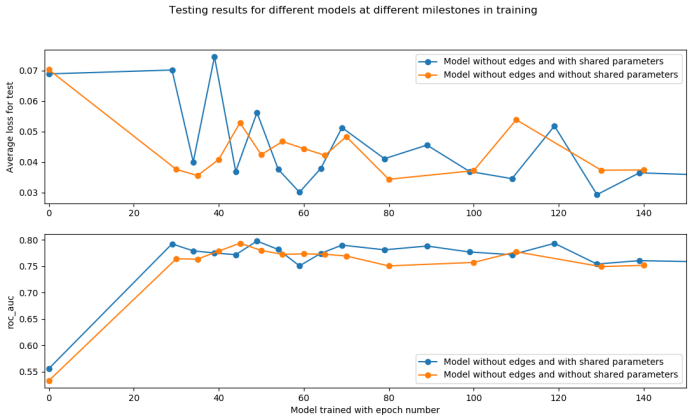

Figure 10. Top: average loss of the testing for increasing number of epochs, for the model with weights shared between the convolutional layers and the one with independent weights. Bottom: area under the ROC curve of the testing for increasing number of epochs, for the model with weights shared between the convolutional layers and the one with independent weights.

The separation between the weights in the two ”legs” increased significantly the running time of both training and testing: although this model could lead to the best classification, its cost in terms of computational time makes it a non optimal option.

5 CONCLUSIONS

There are a few takeaways that can be summarized at this point in the elaborate. First of all we have presented a general introduction to GNNs and how their limitations call for different implementations to improve performances. We have also quickly explained how CNNs are ideal to extract patterns of increasing complexity within organized structures, as well as how they are limited when applied to graph structures. Consequently, we have then focused our attention on a peculiar way to adapt the convolutional approach to graph structures by exploiting the connectivity of the graph itself.
Having taken inspiration from ref.s (4) and (6) we have then tried different software implementations. Regarding our results there are a couple of points that can be made.
First of all our implementation is definitely slower than what is reported in (4). The causes for this are twofold: first our implementation is focused on adapting the concept without regards for performance. Second, since we have used our personal computers, the computing power at our disposal was reasonably much smaller than what available for (4). Overall, our implementation takes somewhere close to 1-2 hrs for training a single model and a further 2-4 hrs for testing the model obtained at a fixed number of epochs. When testing models from different epochs the run times easily got up to 10-13 hrs, whereas in (4) it seems that at most the implementation runs for 1hr and 20min.
The second point can be made looking at the values relative to the area under the ROC curve, which quantifies how often in the testing set the model obtained at fixed epochs predicts true positives or false positives. Our naive implementation obtains on average a value in the interval [0.75, 0.83]. When compared to the best results reported in the original article (4) of 0.891 it seems that our results are not that much worse than what was obtained. Our results are actually comparable with those obtained by some of the competing models reported in (4).

REFERENCES

[1] Scarselli, F. et al. *The Graph Neural Network Model.* IEEE Transactions on Neural Networks 20 (2009): 61-80.

[2] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li and Maosong Sun. *Graph Neural Networks: A Review of Methods and Applications.* 2019, arXiv 1812.08434.

[3] Xie, Yiqing and Li, Sha and Yang, Carl and Wong, Raymond and Han, Jiawei. (2020). *When Do GNNs Work: Understanding and Improving Neighborhood Aggregation.* 1303-1309. 10.24963/ijcai.2020/181.

[4] Fout, Alex et al. *Protein Interface Prediction using Graph Convolutional Networks.* NIPS (2017).

[5] https://zenodo.org/record/1127774.X6ak0yySk2x

[6] https://github.com/pchanda/Graph_convolution_with_proteins/

[7] https://github.com/fouticus/pipgcn/

[8] https://github.com/gretagrassmann/ComplexNetworks2