Heribert Pascual Saldaña

Graph Convolutional Neural Networks applied to classify cancer types

Final Master's Project

Directed by: Dr. Carme Julià Ferré

UNIVERSITAT ROVIRA i VIRGILI

Tarragona
2021

Abstract

The main objective of this master's thesis was to reproduce the results published in the paper “Classification of Cancer Types Using Graph Convolutional Neural Networks” written by Ricardo Ramirez et al., since the use of these networks is a relatively new concept, and many researchers are focused nowadays on its use for many purposes. The paper’s goal is to classify tumour tissues correctly using convolutional graph networks (aka. GCNN). In the paper’s results section, we can see the precision and errors of four types of GCNNs, created by different interactions between genes. Using the GCNN with the best results, a study has been carried out by the authors on the data, to know the gene type relevance of the classification result. This is a way of knowing which genes can be an indicator of a specific type of tumour, it can also be useful for diagnosing certain types of tumours that, due to their proximate location, can be confused.

Once the code began to be analysed in some depth, because it caused problems, some anomalies were detected. This reason made it necessary to carry out an in-depth analysis of the whole code and its libraries, where problems were detected, some of them errors, different use of the usual techniques or problems with the files that contained the results. This motivated the modification of many parts of the code for its correct operation and to be able to analyse the final results. A parallel code has also been generated; this code follows the techniques commonly used to demonstrate that the GCNNs can be used for this purpose in an unambiguous way.

With the modifications made, the initial objective of this master's thesis has been partially achieved. All types of GCNNs have been created to measure their precision, and confusion matrices have been extracted from all GCNNs. The problem has arisen when studying the relevance of each gene, since the code supplied with the paper did not contain this part. As this part of the objectives could not be achieved, it was decided to add new ones, such as generating the second code to validate the authors' approach. A code that performs the gene alterations analysis in a coarser way was also made, but that can be valid since it is based on whether there is a change in the prediction or not, instead of applying the commented thresholds on the paper, that are hard to acquire with the actual code structure.

In addition, the fact of having created the second code that uses a different technique is used during this document to make several comparisons about the results of the original and the modified code, which gives more robustness to the conclusions.

It is recommended to read this master's thesis to see in detail the number of problems and complications that have had to be overcome to reach the final conclusion. This is, that the code provided by the authors in the repository should not really be the one used to extract the results of their conclusions. Or in case it really is, the conclusions could not be considered correct at all, since there are some important errors. You can also see throughout this master's thesis some inconsistencies between what is exposed in the paper and what the code actually does, this is what explains this controversial final conclusion.

Table of contents

Table of figures

Introduction

In the following sections, we can read the background subsection that puts the reader in the context of this master's thesis. Also, the thesis's motivation, were the main reasons that have led me to carry out this thesis are exposed. Then, in the objectives section, we can read a description of the proposed objectives, since I have not presented a new solution, but rather I have analysed an innovative proposal.

Background

This master's thesis is based on the analysis of the paper "Classification of Cancer Types Using Graph Convolutional Neural Networks" **[1]** written by Ramirez, Ricardo; Chiu, Yu Chiao; Hererra, Allen; Mostavi, Milad; Ramirez, Joshua; Chen, Yidong; Huang, Yufei and Jin, Yu Fang. The objective of which is to classify different types of tumorous tissues with their respective cancer type and to analyse which markers are more characteristic for each cancer. To reach their goal, they work with RNA sequences extracted from tumours. For those who are not very experts in the field, as is my case, it can be said that if DNA is the basis of the genome, RNA is the message. That is, genes are made of transcribing the genetic code into RNA, and then the RNA into protein.

The cancer disease is quite common nowadays, in the global cancer observatory webpage **[2]**, we can find so much data related to it. From this source I've extracted the plot in the Figure 1, that shows the incidence of cancer worldwide with divisions by continents in the year 2020. If we access its page, putting the pointer over the outermost divisions, we can see the data at a regional level. With this estimation, a cumulative incidence of about 247 cases per 100,000 inhabitants can be calculated. That, if we compare it with the trendiest disease nowadays, Covid-19, this incidence level is considered high risk. Taking into account that cancer isn't contagious, we have a very high, and worrying incidence comparing with a contagious respiratory infection, that is able to spread in a fast way.

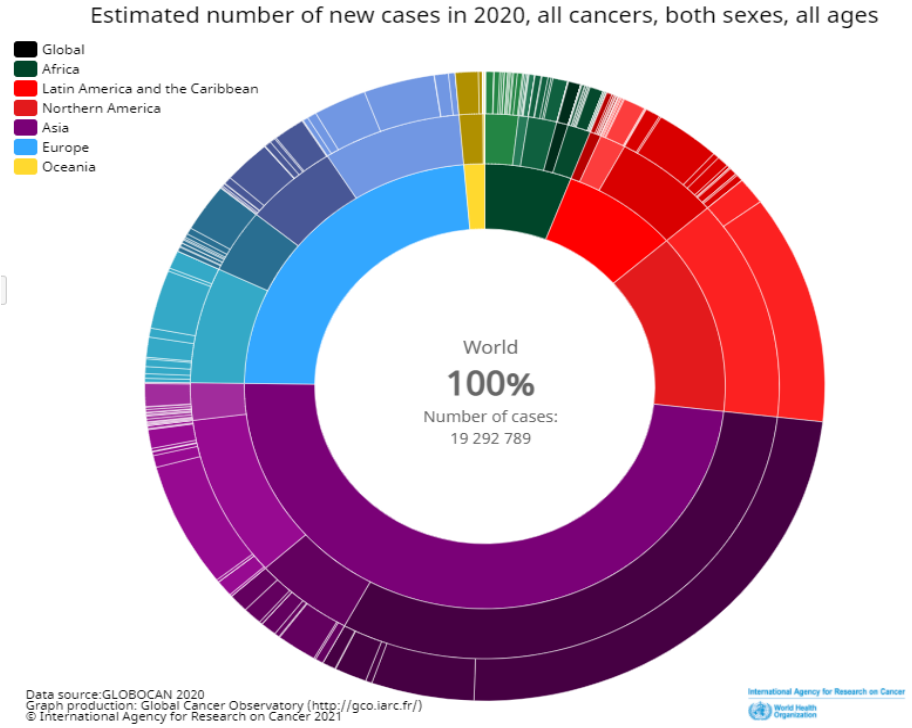

Figure 1 Cancer cases in year 2020

It is also interesting to know the mortality of the disease, for this the same source has been consulted and the results can be seen in Figure 2. All these deaths, are not a direct consequence of the new cases in the same year, but making a rough calculation, it can be said that more than half of cancer cases in the world end in death.

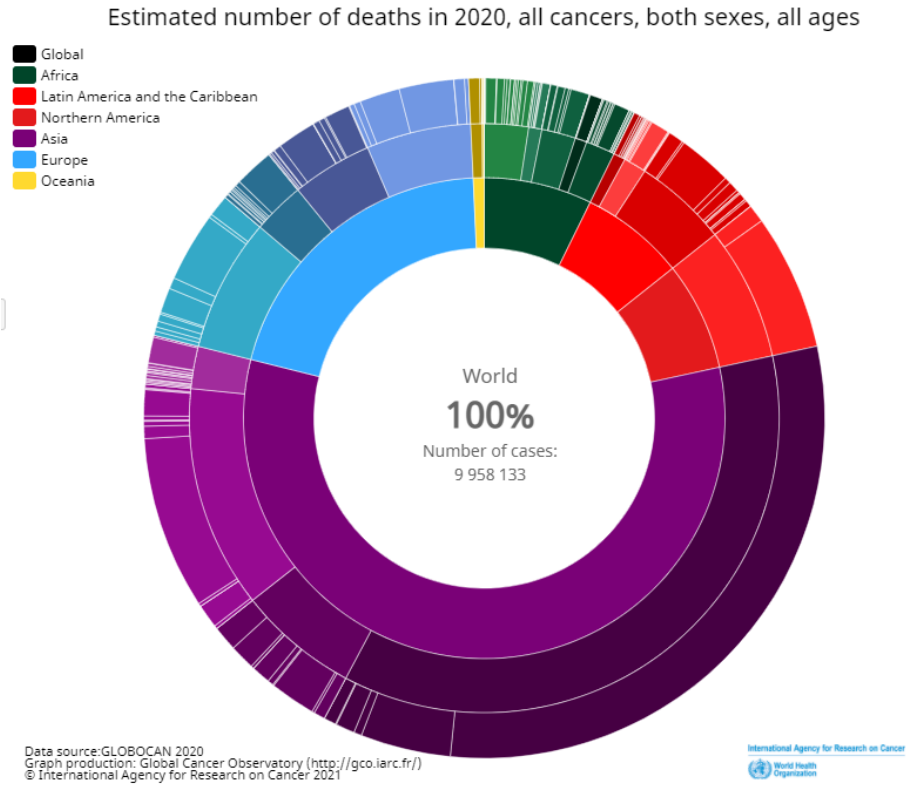

Figure 2 Estimated cancer deaths in 2020

To know in depth the history of cancer, a fast reading of **[3]** is highly recommended. To learn more about its history, it is recommended to read **[4]**, where you can read the history of cancer, of which documentation has been found from the time of the ancient Egyptians dating back to the year 3000 BCE.

Cancer research boomed in the late 19th and early 20th centuries. Nowadays, various factors that can produce the disease and their states have been known, and treatments and diagnostic techniques have been developed. However, the most relevant discoveries have been developed in recent decades, especially in the fields of DNA and RNA. Moreover, the incredible technological advance in the field of computing capacity aids to still have research ground to go beyond what we will see later in the state-of-the-art section. At the moment, in addition to having exceptionally trained professionals, good technical means, and relatively effective (many times quite hurting for the patient) treatments, prevention and early detection are the best techniques. This last factor helps the disease, minimally affect the patient, in terms of quality of life and treatment. That is why today much effort is being focused on finding easy and inexpensive ways to detect as early as possible. Due to these facts, a quick cure can be easier and faster would be developed, which is not only good for the patient, but also for the institutions, which could save many costs, in exchange for investment in research to enhance the early detection.

For these reasons, the study carried out in the paper object of this thesis is relevant. Since, the deeper the global knowledge in all aspects of the disease, the greater the capacity to

understand it and thus be able to detect and attack it earlier. In the state-of-the-art section delve into some novel techniques that have been developed as a result of various genetic studies. Which is interesting, because if they are tested and standardized, they could help to save lives, and alleviate much suffering in the world, moreover, doing the work of our medical professionals more bearable.

Motivation

I chose this project because it was full of challenges for me. From the main concept, genetics, to which I have never made any approximation, although it is very interesting to me. The fact of deepening and improving my skills in a programming such as Python, which I am not much used to, and nowadays is widely used to perform preliminary tests before a doing an efficient final implementation. This fact, saves a lot of development time before deciding with the continuity of a project. In addition to the introduction to convolutional networks using graphs, which caught my attention because there are hundreds of applications for graphs in everyday life, as I saw in complex networks master's subject. All these reasons reinforced by the main subject, tumour classification, encouraged me to accept this challenge. Also, I thought that this thesis can be a great opportunity for me to learn about other techniques that can improve people's lives, and open my mind to new applications. In the end, the more techniques and applications I can know, the better I can work to improve the society in which I live.

Master's thesis goal

The main goal of this thesis is to understand and explain in a more understandable way the paper **[1]** written by Ricardo Ramirez's and how it's code, shared in a GitHub link that it's based on graph convolutional neural networks (aka. GCNN) works, and try to reproduce the results shown in the paper **[1]**. To achieve this goal, it has been proposed to:

- Reproduce the results.
- Understand the input data to know how the GCNN is generated.
- Understand the gene input data.
- Compare the results.

All of this, doing the small necessary modifications to the code to get it work. Also, giving everything ready for any other computer scientist, that reading some details in this document, and the comments in the code, can understand it quickly to use it. Giving the possibility to go deeper into it if desired, without having to waste time to get everything working, and avoiding it to following in detail the entire code.

State of art

To know where we are in this master's thesis, I have decided to divide this section into two subsections, since both have many recent innovations. One is the use of convolutional neural networks using graphs, a recent topic that is being researched a lot. The second one, is the use of computer science and artificial intelligence in cancer diagnostics.

Graph convolutional networks GCNN

Before focusing on the GCNN, we must know the origin and utilities of the CNN, since the GCNN are an evolution of the CNNs. To write this section, I have relied on the survey **[5]** where we can read some history about the convolutional neural networks (aka. CNN), the different types of networks that have been created over time, an interesting experimental analysis and different types of applications for the CNNs. There are also data extracted from the research article **[6]** in which we can read interesting and concise details of the internal architectures, and details of their use in computer vision and speech recognition.

As can be read in the introductions of both papers cited above, the first use of the word convolution was used by LeCun et al. **[7]**. When they introduced the artificial neural network LeNet-5, which was used to perform a classification of handwritten postal codes. As we have seen, CNNs come from the convergence of artificial intelligence and the term convolution. A good convolution explanation from scratch can be found in **[8]** for all those aren't used to this term.

In the introduction of **[6]** we can see the history of the CNN idea. The architecture originally comes from a study of the structure of the visual system of cats **[9]** developed by Hubel and Wiesel in 1962. In 1980 Fukusima proposed the Neocognitron **[10]**, a model inspired in Hubel and Wiesel research. This is known as the forerunner of CNNs, hierarchically organizing neurons to perform an image conversion. After the Neocognitron, the next notable proposal was from Lecun et al. named LeNet-5 **[7]**, Where the word convolution was used for the first time. The proposal is based on a CNN able to recognize postal codes written by hand.

As can be seen in the **[5]** introduction, the existence of CNNs is closely linked to artificial neural networks (aka. ANNs). The beginnings of ANN can be found in "A logical Calculus of Ideas Immanent in Nervous Activity" **[11]** a mathematical model proposed by McCulloch and Pitts in 1943. Some years later, in 1960, Rosenblatt proposed a single layer perceptron **[12]** adding the ability to learn to the McCulloch and Pitts model. In this case, this architecture is not capable of solving linearly inseparable problems, but it was a big innovation in these times. In 1986 Hinton et al. **[13]** proposed a multilayer neural network that was trained in the now widely known method of back propagation. A year later, in 1987, Waibel et al. They proposed a network that was called Time delay neural network **[14]** (aka. TDNN), which can be seen as a one-dimensional convolutional network, focused on speech recognition. Zhang then proposed a network that was the first two dimensionally network, a close one, to that today is known as CNN. Finally, we are back to the point where we started, in 1989, with the network created by LeCun, LeNet-5 **[7]**. Having connected the history of AANs and CNNs, ANNs in general will be left aside as they are outside the scope of this document, since the field of artificial intelligence has evolved closely, linked to the increase in computing capacity that it has

occurred in recent years. For this reason, a large amount of literature has been generated and it is important not to lose the main thread, the CNNs that leads us to the GCNNs.

From LeCun's contribution, the research in CNNs was frozen, for some years there aren't any remarkable developments. In 2005 GPUs began to be used for machine learning, this put the researchers' focus again on this subject. As we can see from the image extracted from **[5]** in Figure 3 below, beyond 2012, different types of remarkable networks were created with different goals. Due to the high number of networks and their variations, they will not be commented one by one so as not to extend the section more than necessary. To delve into these networks, a reading of **[5]** is highly recommended, where all its main features are explained.

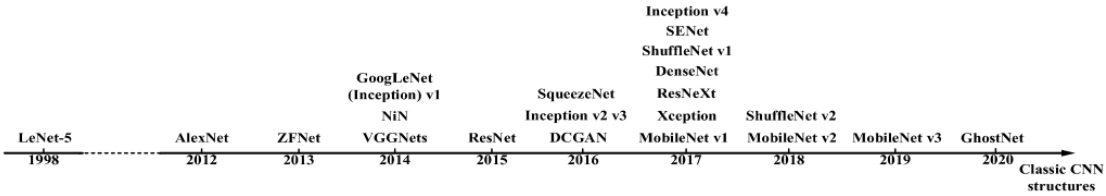

Figure 3 CNN timeline

Now we arrive to the master's thesis topic, the graph-based convolutional neural networks, that are very recent. If we look in the literature, we can see how the first documents we can find date from the beginning of 2017. The two most relevant papers were written by two referents in this field, Doctor Thomas N. Kipf with his contribution **[15]**, and Michaël Defferrard et al. with the paper **[16]**, in this last one, we can find the explanation and the code used by Ricardo Ramirez in the paper that is being analysed in this master's thesis. In both we can see very similar proposals. Getting into the exact detail of each operation is very complicated. As a quick introduction it is recommended to see the talk by Xavier Bresson, a Michaël Defferrard co-author in **[16]** and **[17]**, or if you want to go deeper into the GCNN theory, review the Thomas N. Kipf doctoral thesis in **[18]**. Some proposals based on GCNN can be found, such as Chunyu Wang one, "A Cancer Survival Prediction Method Based on Graph Convolutional Network" **[19]**, which presents a new model, which combines the genomic point of view, like the one in the paper that is being analysed, in this master thesis, and clinical data of the patients to obtain the prediction of cancer survival.

Machine learning in cancer

To write this section, the following surveys and articles have been reviewed **[20] [21] [22] [23] [24]**. Below there's a short summary of the most relevant ideas for this master's thesis, for a better understanding of the author's global objective in the field of tumour cell research.

First, before getting into more technical concepts, we have to know some details about cancer. Cancer is a disease that can be found in many parts of the body. The human body is made up of trillions of cells that are being continuously renewed. Some of those cells, die or become old through the time, then those cells, should be replaced by new ones, this is a common process in our body. When a person is affected by cancer, this process is altered, and the patient develops new cells when they are not necessary. These extra cells help the older ones to survive, when they should be dead. Then, these new cells, are grouped together, and those continuing with the reproduction, generating what is known as tumour. Those tumours help the cancer disease spread to the rest of the body. It can be

read in the WHO website **[24]**, that in 2020 there have been about ten million deaths caused by cancer. In this interesting article, where much data can be consulted, such as mortality due to types of cancer, the risk factors that can cause it, advices to prevent it, data on detection and treatment.

As seen in the GCNNs section, there is a lot of work done in the field of computer vision, from the work of LeCun et al. in 1989 recognizing handwritten postal codes, through all other networks, until today. Until some time ago, the field of artificial vision was very focused on cancer visual detection, a field which is highly developed as we can see In **[23]**. In this survey it can be read many different methods of image segmentation for the detection of some types of tumours, tumour image classification, tumour detection, a comparison of the precision of different techniques, and a summary of the public data sets available with images of tumours. In the review article **[22]**, you can also read the summary of several techniques that have been tested in public data sets, with special emphasis on common tumours. Its reading is highly recommended due to the explanations detail, and the level of detail and the illustration's quality. However, it will not be seen in detail, since it is also based on artificial vision that is not the aim of this document.

The use of artificial vision tools, helps the clinicians to detect tumours using diagnostic imaging. These tumours when are detected, are routinely biopsied for a later analysis. This generates a great inconvenience and loss of time, both for the patient, who must carry out several diagnostic tests in the hospital, and for the clinicians and auxiliary personnel, who have to perform them. In addition, there are certain areas that are very difficult to access to perform a needle puncture, such as some internal organs of the rib cage. Other problems that we can find are that the sample obtained with the needle is too small or that the tumour due to its small size is difficult to biopsy. This, along with other more technical details explained in **[25]**, have led to a search for a new diagnostic concept, as can be the liquid biopsy.

To perform a liquid biopsy, only one or more blood samples are needed from the person to be analysed as is detailed in **[25]**. Then, in the analysis of this blood, circulating cells of the tumours can be observed if they exist. Furthermore, their genetic codes can be differentiated to recognize the specific tumour type, or whether the tumour is in a metastatic state. In turn, this is of great help in the diagnosis, given its ease, and in the preparation of the treatment, given its precision. Taking into account that genes are a sequence of markers, and a human may spend a huge amount of time to classify it, with the added problem that when the cancer is in early the stages there is not so much DNA in the blood. This, leads us to think of how far can be reached using the advances in the field of artificial intelligence mentioned above, using only the genetic markers, that basically is data, not images. We can find a solution similar to the first one proposed by the author in **[1]** that is focused in the blood analysis, in article **[26]**, where artificial intelligence is applied in the early detection of colorectal cancer.

Ricardo Ramirez et al. proposal analysis

On paper **[1]** Ricardo Ramirez et al. uses GCNN (convolutional graph neural networks) to classify the genetic code of thirty-three tumour tissues, also adding a normal category for healthy tissues. Doctor Ramirez has been working in this field for some time, which will be introduced in the previous work subsection. Then, in the summary subsection a coarse view of the work done by the code is presented in a pleasant way. And after this small introduction, we will go into the detail of the code, following the best result execution, the protein-to-protein interaction with singletons. The code is followed from the input data section, where the origin of the data, and its structure will be analysed to understand the GCNN data feed. In the GCNN preparation section, is explained how the data is prepared to create the GCNN. Later, in GCNN section it will be analysed following the code into the libraries from the data through the pooling, until the GCNN is trained. Then, a function modified by the authors in the Michaël Defferrard library **[27]** is commented, because is in charge of performing the GCNN test. Using other library, the test function, also is in charge to perform the post modelling analysis. The original code is attached near the comments to follow it easily, but also can be found in **[28]**.

Previous work

We can read how Ricardo Ramirez et al. has been working on the classification of tumorous tissues for some time. In paper "Convolutional neural network models for cancer type prediction based on gene expression" **[29]** published in 2019, he presents us with three different types of convolutional networks designed for that purpose. In this approach the author requires a specific order according to the chromosomal position of the gene. In the results section you can see how all three achieve an accuracy of around 95%. As can be seen in the abstract, we can find the code referring to this paper in **[30]**.

When the authors set up the experiment for the new paper the main goal was design a model to classify the tumorous tissues using the GCNNs, create a model able to classify using the co-expression interaction, and know which gene markers are more relevant to classify a tumorous tissue. For this he prepared the data to generate four different graphs. Two of them are based on the protein-to-protein interaction (aka. PPI), explained in the next section. Another is based on PPI but adding the markers that have no direct interaction with the others. The same strategy is applied to those that are related by the co-expression graph, for one it uses only those that are related to each other, and for the other it uses all of them, as in the second case of PPI, whether they are related or not. From this point on, the code begins to be analysed, from data entry, through the preparation of the graphs, their execution and the reproduction of the results that the author achieved between 89.9% and 94.7%.

Summary

Starting the explanation from the input data, the process that has been carried out in the analysed code, first loads an adjacency matrix. The fact that one gene is related to another depends on the pre-processing step that has been carried out in Matlab, which can be the result of a Spearman correlation in the case of the co-expression relationship, or extracted from the BioMart database, which uses a protein identifier.

So, the next step is to pass the adjacency matrix loaded by the coarsening function, which is the one that performs the pooling using the metis algorithm **[31]**. Then, a list is returned

containing the two graphs, the original, and the pooled one. Then, a fully connected layer with 1024 neurons is attached between the coarsened graph and the output.

In Figure 4 we can see on the left the original graph for PPIS and on the right the real result of how it is actually used in the code. These graphs are obtained using the networkX library to extract the network in Pajek format. Then, using Pajek software, a 3D file was generated to finally generate the 3D model with view3Dscene software. Only the PPI is presented because this operation is very time consuming due to the number of nodes and links.

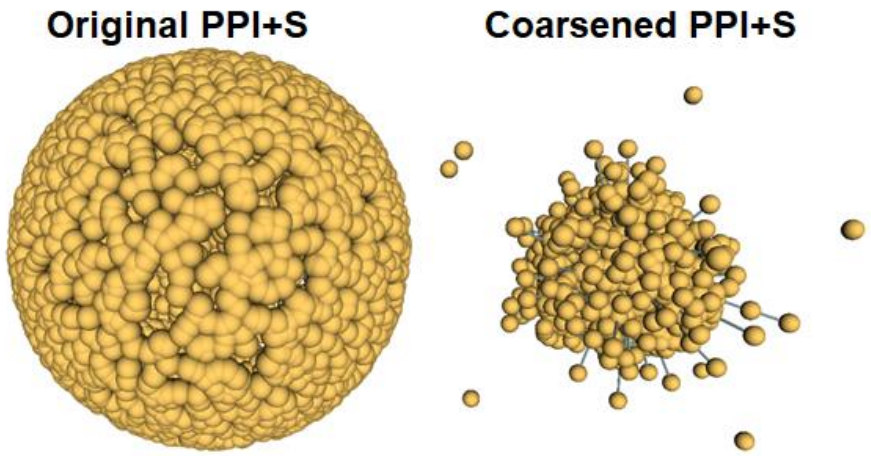

Figure 4 3D GCNN graphs extracted from code

Then, another file is loaded from which a matrix and an array are extracted, the matrix contains the data with which the GCNN will be fed, and the array the labels with the tumour type for each array in the matrix. At this point, the data are processed to fit into the pooled graph, then, the data and the labels are divided into 80% for the GCNN training, and the remaining 20% is used for validation, and for the final test.

Then, after indicating the corresponding parameters to create the GCNN, the cgcnn function from the model's library is used. This is responsible for the creation of the structure of the GCNN, and therefore receives the list with the graphs and the parameters. Once the CGNN is created, it is trained using the 80 percent of the data discussed above, over several epochs. Periodically, it uses the validation data (the remaining 20%), to contrast the result of the training up to that moment and adjust the learning rate. At the end of the training, the GCNN is ready to classify one set of data in a class. In Figure 5, a scheme of the structure creation can be seen.

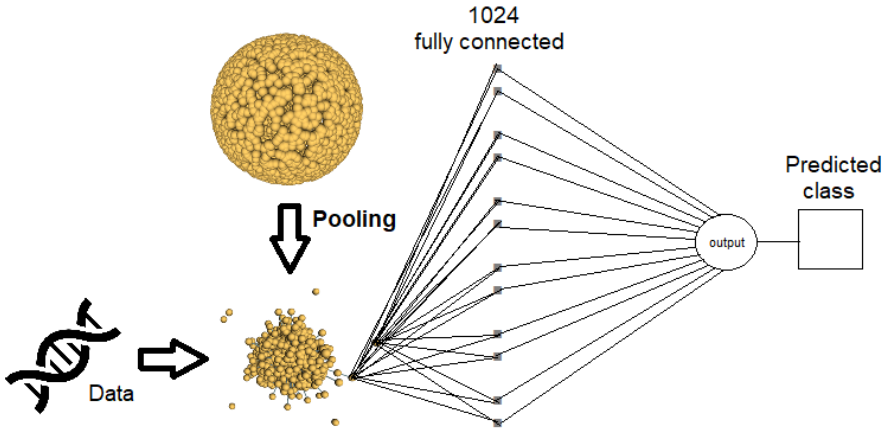

Figure 5 Ricardo Ramirez's proposal structure scheme

The first step, once the GCNN is created, is to test with the test data set, which, being the same as the validation data set, should not give a very different accuracy. Now the post-modelling analysis begins. To do this, from the input gene matrix, one is modified at each step, and all the data to be predicted are sent. In this way, it is possible to know what affectation this gene generates by counting the erroneous predictions generated, or using other techniques. In this case, the authors have set the gene to one and have repeated the predictions on the test data, then they have set it to zero and have done the same. All this for all the genes of the graph that is being analysed. In Figure 6 we can see it in a more graphic way.

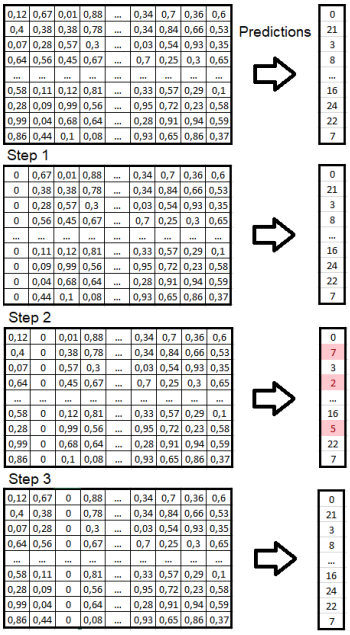

Figure 6 Knock-down example

These predictions are accumulated in two matrices, one of the values is modified to zero, and another for the ones. At the end of the process, these matrices are saved for later analysis. In the following subsections the code is followed commenting how it works step by step.

Input data

The data used to carry out the experiment, becomes from the same databases than in experiment of the 2019 in paper [29], they have been extracted the genetic data and the class from the TGCA [32] RNA-seq to perform the GCNN training and testing. The original measure unit used is FPKM (Fragments Per Kilobase Million) and the author applied a logarithm to base two after adding one to the value, then the values greater than 0.6 are considered. Then, these values are normalized between 0 and 1. We can know the process in more detail on the web **[33]** or in the paper **[34]**.

The paper explains how four different types of GCNNs are created. However, the way of how they are created can be divided in two. One of these groups is based the protein-to-protein interaction, and the other on the co-expression relationship between genes. In the next lines a deeper explanation about each one can be seen.

The PPI graph relationship, can be summarized in a very rough way with the following sentence. "*A group of proteins working together to perform a biological function not necessarily are in direct contact, but their relation may be of regulation or influence"*. To create the PPI graph, the authors used the BioMart databased **[35]** to get the corresponding Ensemble protein Id and human protein interactions were downloaded from STRING **[36]**. Then, due to limitations of each database, at the end remained 4.444 that are represented in an adjacency matrix, where a 1 is places if there's interaction between them. The PPI graph that includes the singleton nodes has all the 7091 genes. In Figure 7 we can see the data that are loaded examined from Matlab. In the first variable Adj_list we see what an adjacency matrix looks like, but this is not used in the code. Then we see row i col with 53372 doubles. In row we have integers from 0 to 4443 that are repeated and are increasing. In col we have integers from 166 to 1076 in no apparent order. In value we can also observe 53372 doubles, which all contain 1s. Finally, after observing the operation of the code, row and col are adjacency lists and value should be the weight of the edge, which as in this case is not taken into account, it is one.

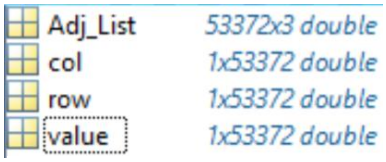

Figure 7 Adjacency PPI file

The co-expression graph can be also summarized with the next sentence from the web **[26][37]**. *"The co-expression network is defined as an undirected graph where the nodes correspond to genes, and the edges indicate significant co-expression relationships. The pairs of genes are considered in a systemic perspective of cooperation, including co-regulation, activation/suppression, and indirect control through the action of siRNA, miRNA, proteins, metabolites, and epigenetic mechanisms*". To create the co-expression graph, the authors first processed the data using Matlab applying a Spearman correlation. Then, when the correlation result is greater than 0.6, a one is written in the adjacency matrix, otherwise a zero is written. Finally, the result is an adjacency matrix with 3866 elements. The matrix that contains the singleton nodes, again has 7091 elements. In the Figure 8 we can see the Matlab data that are loaded in the Python code. Into this file we can see some different data representations. This file is used for both co-expression graphs, the one that has singleton

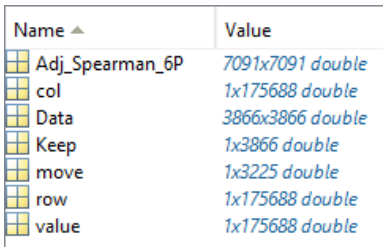

Figure 8 Adjacency COEX

nodes and the one that hasn't. A deeper explanation about gene relations used in both types can be found in **[38]**.

Now it's time to start to follow the code to check the data movements, the GCNN creation and test the result at the end. The first step that I did, was to prepare all the environment to meet all code requirements, and some code modifications had carried out before being able to run the code. All details about these problems are being explained in a detailed way in the section work done.

Graph preparation

Using the original code to be able to comment on it more easily, and the PPI + singleton graph that's the one the original code is prepared, after the initial imports, and the preparation of some flags, from line 34 we can see how a .mat file is loaded into the test variable. Commented, on line 36 we can see that it loads another file. These lines are attached below for a better understanding.

```
33:#For PPI and PPI-singleton model change file location
34:test = sio.loadmat('C:/Users/RJ\Desktop/exp_fpkm_pancan/processed/PPI_filtered/Adj_Filtered_List_0Con.mat')
35:# for Correlaton model change file location
36:#test = sio.loadmat('C:/Users/RJ\Desktop/exp_fpkm_pancan/processed/CoExpression/Adj_Data/Adj_Spearman_6P.mat')
```

As we can see, the path is absolute, so to reproduce the code we will have to change it. The files needed as inputs, can be found in **[39]** under an author's Google Drive, this path is relative, and the files should be in the folders assigned to the function. The mat file is the block of data that has been commented on in the input data section that comes from the pre-processing that the authors comment on in the methods and materials section. The loadmat function belongs to the Python ScyPy library and generates a Python dictionary where the variable names are the keys and the saved arrays are the values. In the Figure 9 Adjacency comparison we can see at left the data used for PPI graph creation, and in the right the data used for the co-expression graph generation.

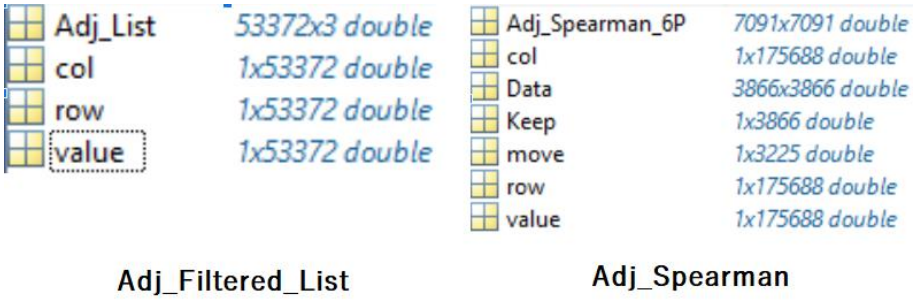

Figure 9 Adjacency comparison

Now we follow with the next three code lines.

```
37:row = test['row'].astype(np.float32)
38:col = test['col'].astype(np.float32)
39:value = test['value'].astype(np.float32)
```

In the line 37 the authors load in the variable row, an array with 53372 doubles (forced by astype function). Checking it with the debugger, we can see that all of those doubles are integers, and they have a minimum of 0, and a maximum of 4443. In Figure 10 can be seen the "row" variable analysis, performed with the debugger, where the values are repeated. Also, In the same figure it can be observed the minimums and maximums discussed before. This led me to Matlab again to be able to view the data one by one more comfortably. Observing them from Matlab it is found that the values are repeated, and are ordered in increasing order.

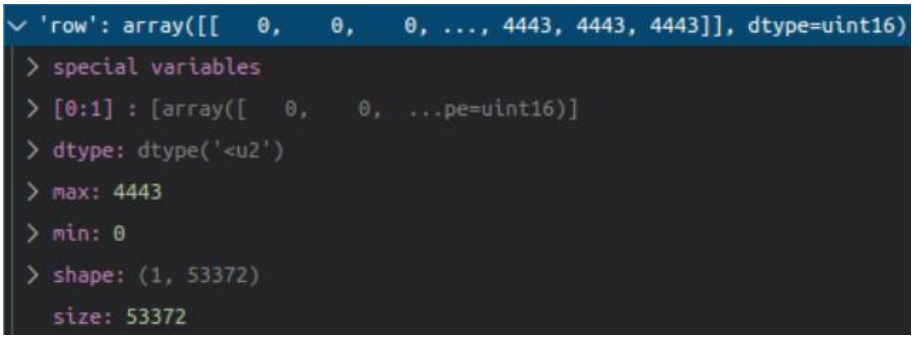

Figure 10 Debugging ROW variable

The fact of not directly loading the adjacency matrix in the Matlab file, led me to delve deeper into the data. In order to have more information, I generated a histogram that we can see in Figure 11 which will be useful for a later analysis.

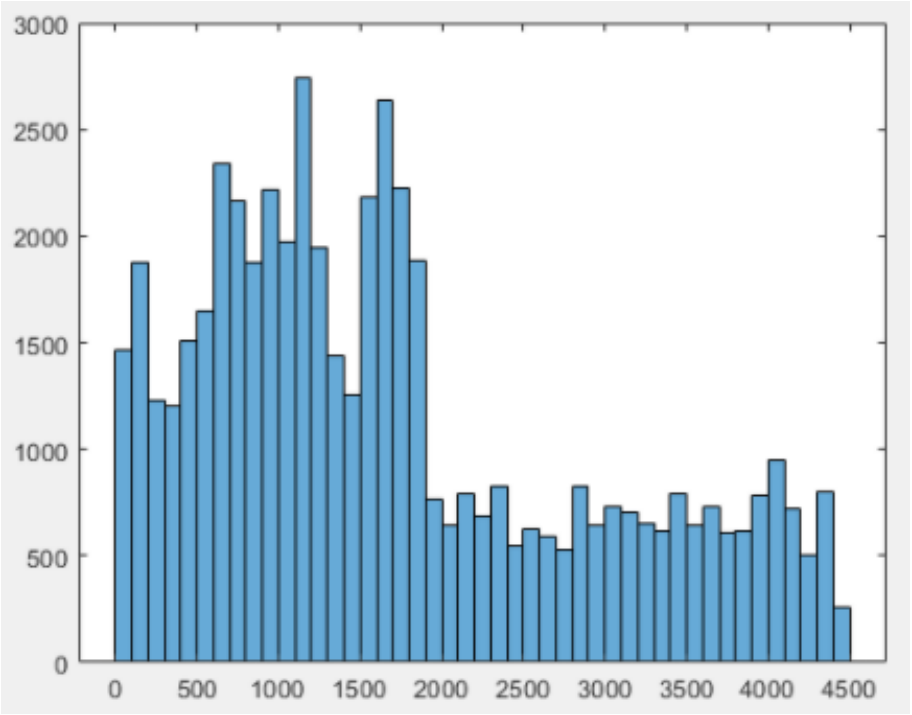

Figure 11 ROW histogram

Now continuing with line 38 of the code, we see that it performs the same operation for the variable col, where it stores the information of the key col. It also contains 53372 doubles according to Matlab, but again analyzing the variable with the debugger, we can

see in Figure 12 that it is filled with integers. These have a minimum of 0 and a maximum of 4443, just like in row.

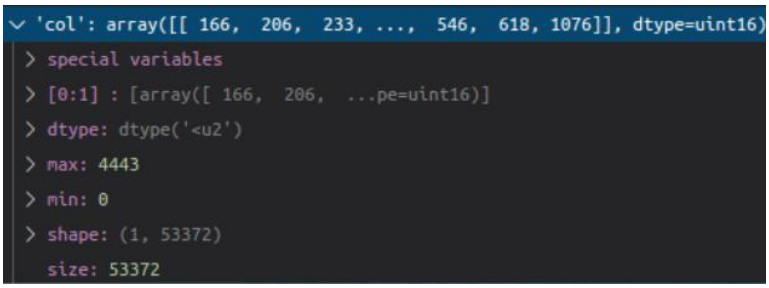

Figure 12 Debugging COL variable

In this case, we can see how they are not ordered as in row, but repeated as in the previous one. In Figure 13 we can see the data histogram.

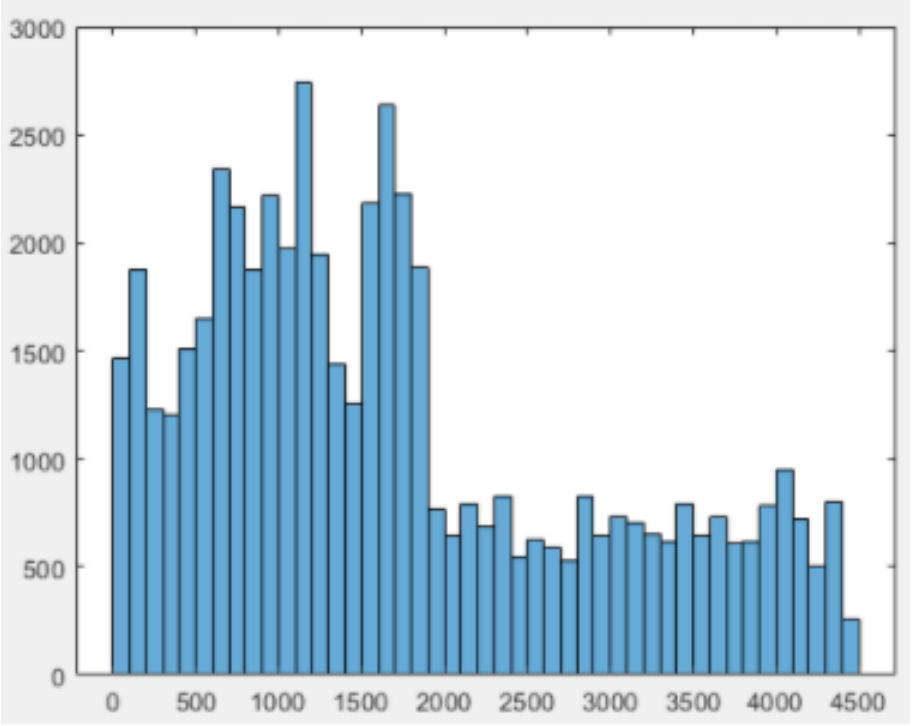

Figure 13 COL histogram

In Figure 14 it can be observed an overlay of the histograms, based on the row histogram, with the original plot blue color, and superimposed on it, in black color and with a 60% of transparency the col one. In this figure it is clear that the values of row and col are the same, but distributed in a different way.

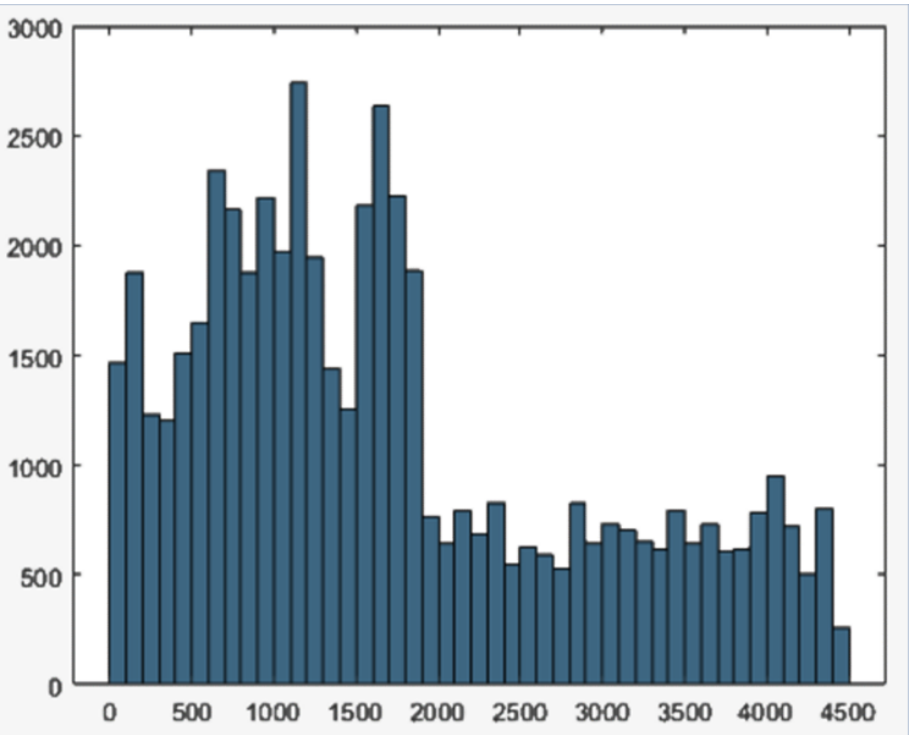

Figure 14 Histograms overlay

Now, on line 39, the values of the value data block are stored in the value variable. As we can see in Figure 15, the variable **value contains an array of 53372 elements, which are all ones.**

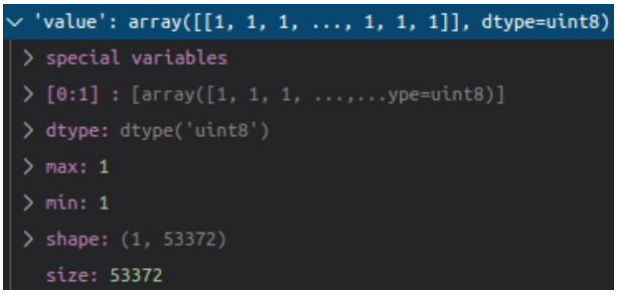

Figure 15 Debugging value variable

Now the data has been loaded into the program, the authors carry out some movements that we can see in the following lines of code to later be commented on.

```
40:M, k = row.shape
41:row = np.array(row)
42:row = row.reshape(k)
43:row = row.ravel()
44:col = np.array(col)
45:col = col.reshape(k)
46:col = col.ravel()
47:value = np.array(value)
48:value = value.reshape(k)
49:value = value.ravel()
```

In line 40, the shape of the array is obtained using the Numpy function shape **[40]**. As can be seen in the previous Figure 10 where the row variable was analysed in the debugger, it has a shape of (1, 53372), that is, we have an array of one element, which in its first position contains another 53372 elements. After applying the function shape, the value of the variable M becomes 1, and that of the variable K is 53372. In line 41 can be seen how the row data type is passed to the Numpy format using the array function **[41]**, this function creates a Numpy array to be able to use all the functions offered by the Numpy library. In line 42 the Numpy function reshape **[42]** is applied to the data with parameter k that in the PPI case is 53372. Now, we have an array with 53372 elements. In line 43 the Numpy function ravel **[43]** is applied to the col array. This action doesn't have so much sense, because after doing the reshape with a shape of 53372, a flat array is obtained, then after the reshape, the result is the same. These same steps are repeated for the keys col and value and are stored in variables with the same name.

GCNN preparation

Now the code follows with the next lines.

```
50:A = scipy.sparse.coo_matrix((value, (row, col)),shape = (4444,4444))
# change size for model being used 4444 for both PPI and 3866 for
51:graphs, perm = coarsening.coarsen(A, levels=FLAGS.coarsening_levels, self_connections=True)
```

In line 50 we can see how the result of the coo_matrix function from the SciPy library is assigned to variable A, which returns a sparse matrix in coordinate format. Analysing the parameters that are sent in this case (and also in the case that we were working with the co-expression adjacency matrix), we see that value is sent as the first parameter is the array of ones that will be the values in the row-col positions in the result sparse matrix. In the second one, the values of row and col sent together, which are those that contain the data positions. We also see how a shape is passed to the function. This shape, in the original code, is hardcoded, with a comment indicating that it should be modified if the type of graph is changed, but it is cut off. In the code that has been modified, it has been prepared so that this is done automatically according to the parameter received by the program. In Figure 16 we can see the example extracted from the API of the function, where it is clearly seen that the first element of row is 0, the first of col is 0 and the first of data is 4. This tells us that in position [0] [0] we will have a 4.

```
>>> # Constructing a matrix using ijv format
>>> row  = np.array([0, 3, 1, 0])
>>> col  = np.array([0, 3, 1, 2])
>>> data = np.array([4, 5, 7, 9])
>>> coo_matrix((data, (row, col)), shape=(4, 4)).toarray()
array([[4, 0, 9, 0],
       [0, 7, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 5]])
```

Figure 16 Scipy sparese matrix example

After having analysed these movements, all the strange things that had been seen in the data loading and processing are clear like the identic histograms in row and col, because authors are working with an unweighted and undirected graph. From the pre-processing in Matlab, the authors were already preparing this sparse matrix, although in the paper they do not explain exactly how they do it, it is understood that they did to do not increase the explanation complexity in an irrelevant part.

In line 51 the variables graphs and perms are set with the coarsen function result from the coarsening library **[27]** created by Michaël Defferrard. This function receives the sparse matrix A, prepared before, the coarsening levels flag that it's set to 1 in line 26, and the last parameter that allows self-connections. Before going deeper into this function, the returned values are a list of graphs that are stored in a list in the first variable. In this list if we check it, we have in the position 0 the original graph, in the 1 the original graph coarsened 1 time, and if the level of coarsening is higher than 1, we get this number of graphs, coarsened one step by each position. Below we can see attached the function code with some commented parts removed.

```
def coarsen(A, levels, self_connections=False):
    """
    Coarsen a graph, represented by its adjacency matrix A, at multiple
    levels.
    """
    graphs, parents = metis(A, levels)
    perms = compute_perm(parents)
    for i, A in enumerate(graphs):
        p1 = parents[i-1]
        M, M = A.shape

        if not self_connections:
            A = A.tocoo()
            A.setdiag(0)

        if i < levels:
            A = perm_adjacency(A, perms[i])

        A = A.tocsr()
        A.eliminate_zeros()
        graphs[i] = A
        Mnew, Mnew = A.shape
        print('Layer {0}: M_{0} = |V| = {1} nodes ({2} added),'
              '|E| = {3} edges'.format(i, Mnew, Mnew-M, A.nnz//2))

    return graphs, perms[0] if levels > 0 else None
```

As can be seen, the first thing it does is to call another function from the same library called metis, passing the received graph and the requested coarsen levels as a parameter. This function is what performs the coarsening. It is not attached due to its length, what it does is return a list of coarsened graphs the times indicated using the metis algorithm

[31], and a list that contains the coarsened graph parents. So, we can see how it creates the perms variable with the result of the compute_perms function, sending by parameter the parents obtained previously. If we look at the function, we can see how its creator tells us that it returns a list of indices to reorder the adjacency matrix where the union of two parents forms a binary tree. Going again to the coarsen function, for each graph resulting from the metis function, the code eliminates the zeros to obtain a sparse matrix, and sets to zero the diagonal if self-loops are not allowed. For each generated graph, it prints a small summary on the screen, in the attached lines it can be seen an example output from the code.

```
Layer 0: M_0 = |V| = 7448 nodes (3004 added),|E| = 26686 edges
Layer 1: M_1 = |V| = 3724 nodes (0 added),|E| = 25775 edges
```

Finally, returns the previously mentioned lists, one with all the generated graphs, and perms[0], which contains the parents of the original graph.

```
52:L = [graph.laplacian(A, normalized=True,renormalized=True) for A in graphs]
```

In line 52, it can be seen as for each graph contained in the variable graphs, using another function from Defferrard's library, it creates the Laplacian matrix of the graph. Saving all the results in a list of Laplacian matrices called L. Then, from line 53 to 57 some useless variables that are created to generate the final L are deleted.

At this point, the genetic data from pre-processed Matlab file are loaded, as can be seen in the attached code below.

```
59:Data = sio.loadmat('C:/Users/RJ/Desktop/exp_fpkm_pancan/processed/Final/Data/Block_PPIA.mat')
60:Data1 = Data['Block'][0,0]
61:Data2 = Data['Block'][0,1]
62:Data3 = Data['Block'][0,2]
63:Data4 = Data['Block'][0,3]
64:Data5 = Data['Block'][0,4]
65:D1= Data1['D'].astype(np.float32)
66:D2= Data2['D'].astype(np.float32)
67:D3= Data3['D'].astype(np.float32)
68:D4= Data4['D'].astype(np.float32)
69:D5= Data5['D'].astype(np.float32)
70:L1= Data1['L'].astype(np.float32)
71:L2= Data2['L'].astype(np.float32)
72:L3= Data3['L'].astype(np.float32)
73:L4= Data4['L'].astype(np.float32)
74:L5= Data5['L'].astype(np.float32)
```

It can be observed how the Matlab data path is hardcoded again. To execute the code for any other graph, that is not the one for PPI + singleton, we must follow the instructions found in the readme file of the data, which can be downloaded from **[44]**. In this link we can find the adjacency matrices and the data of the genes. In Figure 17 we can see the Block_PPIA.mat file loaded into Matlab. The other data files follow the same structure, but with different number of data, as is explained in the materials and methods section in the different type of graphs subsections in the paper **[1]**.

Fields	P	L	D
1	2227x1 double	2227x1 double	7091x2227 double
2	2208x1 double	2208x1 double	7091x2208 double
3	2151x1 double	2151x1 double	7091x2151 double
4	2264x1 double	2264x1 double	7091x2264 double
5	2220x1 double	2220x1 double	7091x2220 double

Figure 17 Genetic PPI data in Matlab

P are blocks with integers ranging from 50 to 1100. L are integers ranging from 0 to 33, which are the labels for the 34 types of cancer. Finally, D are matrices with doubles ranging from 0 to 0.8 which are the values of the normalized and preprocessed genes. From line 60 to 64 loads the data blocks, and then, in lines 65 to 69, load the normalized data in different variables that start with D, and then from line 70 to 74 loads the labels in the same way into variables that begin with L. Then, the code goes through the next lines.

```
76:Train_Data = np.transpose(np.hstack((D1,D2,D3,D4)))
77:Val_Data = np.transpose(D5)
78:Test_Data = np.transpose(D5)
79:Train_Label = (np.vstack((L1,L2,L3,L4)))
80:Val_Label = (L5)
81:Test_Label = (L5)
82:Test_Label = Test_Label.ravel()
83:Train_Label = Train_Label.ravel()
84:Val_Label = Val_Label.ravel()
```

On line 76 the Train_data variable is generated. To do this, the Numpy transpose **[45]** function is used on the data resulting from sending the variables D1 to D4 to the Numpy function hstack **[46]**. What is being done at this point is to change the format of the data from the rows of the matrix to columns and then using the transpose to change the data from the axis in a reverse way. In short, it is a way to gather the data of the variables D1, D2, D3, D4 as if this was a single list.

On line 77 and 78 a similar operation is carried out to adjust the form to match Train_data in the Val_data (validation) and Test_data (test) variables. In this case, we see that the hstack function is not used, since there are no data blocks to join. At this point, it is striking that the names of the variables that are usually used for the validation of the CNNs (or GCNNs in this case) and for the test, have the same data both. We will analyse this fact later, when the code of the library that creates the GCNN is analysed, so as not to draw hasty conclusions. In line 79 we can see how the labels are stacked in rows using the Numpy function vstack **[47]** on the same blocks used in line 76. In lines 80 and 81 the same labels are assigned to the test and the validation, in accordance with the data in lines 77 and 78. Finally, it uses the Numpy ravel **[43]** function that after checking with the debugger does not make any changes to the data, since they have been previously flattened.

GCNN

The code follows as can be seen in the following lines.

```
86:Train_Data = coarsening.perm_data(Train_Data, perm)
87:Val_Data = coarsening.perm_data(Val_Data, perm)
88:Test_Data = coarsening.perm_data(Test_Data, perm)
```

In lines 86, 87 and 88 the data that has just been loaded into the variables Train_data, Val_data and Test_data is passed through the perm_data function together with the list of relatives. This function returns the data to be fitted in the generated graph input. Quoting the comment of the function within the library, "Permute data matrix, i.e., exchange node ids, so that binary unions form the clustering tree". Then, the code follows in the next lines.

```
91:C = 34  # number of classes

93:common = {}
94:common['dir_name']       = 'PPI/'
95:common['num_epochs']     = 20
96:common['batch_size']     = 200
97:common['decay_steps']    = 17.7 # * common['num_epochs'] since not used use as in momentum
98:common['eval_frequency'] = 10 * common['num_epochs']
99:common['brelu']          = 'b1relu'
100:common['pool']           = 'apool1'

102:model_perf = utils.model_perf()

104:common['regularization'] = 0
105:common['dropout']        = 1
106:common['learning_rate']  = .005
107:common['decay_rate']     = 0.95
108:common['momentum']       = 0

110:common['F']              = [1]
111:common['K']              = [1]
112:common['p']              = [2]
113:common['M']              = [1024,C]
```

At this point, a Python dictionary is prepared with various parameters that are required for the creation of the GCNN. Here we can see how the Chebyshev polynomials are used as it can be read in the paper. The main code las lines are the next ones. Note that an empty object from class model_perf that is within utils library is created at line 102.

```
116:if True:
117:    name = 'Run1'
118:    params = common.copy()
119:    params['dir_name'] += name
120:#    params['filter'] = 'chebyshev5'
```

```
121:    params['filter'] = 'chebyshev2'
122:    params['brelu'] = 'b1relu'
123:    model_perf.test(models.cgcnn(L, **params), name, params, Train_Data, Train_Label, Val_Da
ta, Val_Label, Test_Data, Test_Label)

125:model_perf.show()

127:if False:
128:    grid_params = {}
129:    data = (train_data, train_labels, val_data, val_labels, test_data, test_labels)
130:    utils.grid_search(params, grid_params, *data, model=lambda x: models.cgcnn(L,**x))
```

In line 116 there's a conditional that always is executed. Within this part, the dictionary that had been prepared previously is copied into the variable params, and some more parameters are added up to line 122. Finally, all the data that has been prepared are thrown into the test function. This can be found in the libraries folder, file utils.py in line 291, which will be analysed in detail later. In line 125 the program shows us a plot where the training loss function and the epochs used to validate the GCNN are shown, an example can be seen in Figure 18.

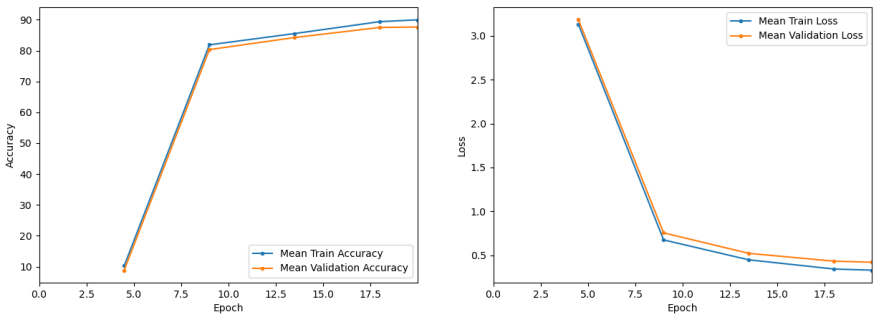

Figure 18 Original code training statistics plot

Finally, on line 127 we have a conditional that will never be executed. This part of the code calls a function from the utils library that searches for the best parameters by training the GCNN several times with different parameters and then reports all the information. It is a very commonly used technique for this purpose in Python.

Test function

Now that we have the generated the pooled graph, the dictionary with the parameters and the data for inputs and labels prepared, we are going to delve into the core function of the code, which belongs to the Defferrard **[27]** library. As mentioned before, the class used can be found in the file inside the lib folder, with the name utils.py, on line 282 of the code that is being analysed, since the authors have made changes to it. Before continuing, we should go back. If we look at the code discussed above, on line 123, we can find the call to the test function that is attached below.

```
123:    model_perf.test(models.cgcnn(L, **params), name, params, Train_Data, Train_Label, Val_Da
ta, Val_Label, Test_Data, Test_Label)
```

It can be seen that the first parameter that is sent to the function, is a cgcnn object, it receives the list with Laplacian matrices and the params variable preceded by two asterisks, this symbol in Python unpacks a dictionary. That is, all the keys and their values

are sent as if they were different variables. In order to know exactly what the code is doing; we must check the cgcnn class contained in the models.py file.

In order to know the type of object that is being created, first we will see below a summary of the data that is being sent in the dictionary that is unpacked, recovered from the code portion of the PPI.py file between line 93 and 122.

```
'dir_name' = 'PPI/'; 'num_epochs' = 20; 'batch_size' = 200;
'decay_steps' = 17.7; 'eval_frequency' = 10 * 'num_epochs';
'brelu' = 'b1relu'; 'pool' = 'apool1'; 'regularization' = 0;
'dropout' = 1; 'learning_rate' = .005; 'decay_rate' = 0.95;
'momentum' = 0; 'F' = [1]; 'K' = [1]; 'p' = [2]; 'M' = [1024,C];
'dir_name' += 'Run1'; 'filter' = 'chebyshev2'; 'brelu' = 'b1relu'
```

Now it's time to check the variables that are sent to generate the instance and those that the library assigns by default when not receiving them, if there are any.

```
def __init__(self, L, F, K, p, M, filter='chebyshev5', brelu='b1relu', pool='mpool1',num_epochs=
20, learning_rate=0.1, decay_rate=0.95, decay_steps=None, momentum=0.9,regularization=0, dropout
=0, batch_size=100, eval_frequency=200, dir_name=''):
```

Comparing the two pieces of code we can see how all the parameters required by the object are sent within the dictionary, in most cases, modifying the ones that it has by default. If we analyse the class found in the lib folder, model.py file on line 722, first of all, there are around 40 comment lines that explain each variable in the class. To summarize its functionality, this class prepares all the parameters that are necessary for the creation of the GCNN. At the beginning, it performs some checks to ensure that the data is correct, and then prepares all the instance variables as necessary with those that have been passed to it when initializing the instance. At the end, it shows us on the screen a summary of the structure of the GCNN. The output of this part of the code is attached below.

```
NN architecture
  input: M_0 = 7448
  layer 1: cgconv1
    representation: M_0 * F_1 / p_1 = 7448 * 1 / 2 = 3724
    weights: F_0 * F_1 * K_1 = 1 * 1 * 1 = 1
    biases: F_1 = 1
  layer 2: fc1
    representation: M_2 = 1024
    weights: M_1 * M_2 = 3724 * 1024 = 3813376
    biases: M_2 = 1024
  layer 3: logits (softmax)
    representation: M_3 = 34
    weights: M_2 * M_3 = 1024 * 34 = 34816
    biases: M_3 = 34
```

At the end, the instantiation of the class calls the build_graph function that we will analyse next, since it is one of the most relevant parts of the GCNNs library.
The following function can be found in the models.py file, on line 168.

```
def build_graph(self, M_0):
        """Build the computational graph of the model."""
        self.graph = tf.Graph()
        with self.graph.as_default():

            # Inputs.
            with tf.name_scope('inputs'):
                self.ph_data = tf.placeholder(tf.float32, (self.batch_size, M_0), 'data')
                self.ph_labels = tf.placeholder(tf.int32, (self.batch_size), 'labels')
                self.ph_dropout = tf.placeholder(tf.float32, (), 'dropout')

            # Model.
            op_logits = self.inference(self.ph_data, self.ph_dropout)
            self.op_loss, self.op_loss_average = self.loss(op_logits, self.ph_labels, self.regul
arization)
            self.op_train = self.training(self.op_loss, self.learning_rate,
                    self.decay_steps, self.decay_rate, self.momentum)
            self.op_prediction = self.prediction(op_logits)

            # Initialize variables, i.e. weights and biases.
            self.op_init = tf.global_variables_initializer()

            # Summaries for TensorBoard and Save for model parameters.
            self.op_summary = tf.summary.merge_all()
            self.op_saver = tf.train.Saver(max_to_keep=5)

        self.graph.finalize()
```

As soon as the function begins, it can be seen how a TensorFlow library a Graph object **[48]** is created (the references are made to the new version 2.5, since the 1.4 used by the code is obsolete and the links to the documentation are not working). In newer versions of TensorFlow, using the graph function directly is no longer allowed. To understand what a graph object is in TensorFlow, the best way to explain it that I have found is to reading directly the documentation. The most relevant text taken from **[48]** is : “Graphs are used by tf.functions to represent the function's computations. Each graph contains a set of tf.Operation objects, which represent units of computation; and tf.Tensor objects, which represent the units of data that flow between operations." We can find a very detailed explanation in **[49]**. In short, this is the empty structure, for now, of what the GCNN will end up being. Now we need to define the information flow between the inputs and the different elements that it will contain according to the information with which it has been prepared. Next, it is observed how it prepares the space that will be used by the data with the TensorFlow placeholder **[50]** function. Next, it prepares the internal operations of the class, using its own functions. If we look at them one by one in detail, we can see that all of them are based on TensorFlow. We will not go into so much detail, since in that case the extension of this document would be too long, in addition to require a great knowledge of the TensorFlow library. Next, the global variables of the TensorFlow graph are initialized, and then the operations to save the training and the summaries are performed. Finally, it uses a function from TensorFlow graph class called

finalize. This function finalizes the object, now is no longer modifiable, because it has the read-only attribute.

Now, knowing all the data that are passed to the test function, it's time to analyse the next steps. As we have seen previously, on line 102 of the main code, an empty instance of the models_perf class is created, which is found in the utils.py file, on line 282 of the code provided by the authors. The test function can be found on line 291. Attached below, we can see the function beginning, which is very similar to the code from the original Defferrard library **[27]**.

```
 def test(s, model, name, params, train_data, train_labels, val_data, val_labels, test_data, tes
t_labels):
        s.params[name] = params
        s.fit_accuracies[name], s.fit_losses[name], s.fit_time[name], s.train_accuracies[name],
s.train_losses[name], s.epoch[name]= \
                model.fit(train_data, train_labels, val_data, val_labels)
        string, s.train_accuracy[name], s.train_f1[name], s.train_loss[name] = \
                model.evaluate(train_data, train_labels, Name = 'Train')
        print('train {}'.format(string))
        string, s.test_accuracy[name], s.test_f1[name], s.test_loss[name] = \
                model.evaluate(test_data, test_labels, Name = 'Test')
        print('test  {}'.format(string))
        string, s.val_accuracy[name], s.val_f1[name], s.val_loss[name] = \
                model.evaluate(val_data, val_labels,con = True, Name = 'Val')
        print('Validation  {}'.format(string))
        s.names.add(name)
```

If we compare the file provided by the authors, with the original one from Defferrard, it is observed that both in the variables of the instantiation, as in the test function, they have added some extra lines to carry out the validation of the GCNN.

As soon as the function begins, we see how the fit function of the model sent as a parameter is called, which corresponds to the GCNN object that we have analysed a few lines above. Therefore, to go into more depth, we should go again to the models.py file contained within the libraries folder. The fit function is found on line 95 of it. From lines 96 to 103 of the function, we can see how various preparations are performed. Some of them refer to time, summaries, folders, etc. Following the code, from lines 106 to 112, empty lists and some variables are prepared, where the results of the training analysis will be stored.
Then a loop is started, the duration of the loop depends on the epochs, the form of the training data and the batch size. In each iteration of the loop, the input data is prepared, the output labels in their corresponding position, determined by a random permutation. Then, in line 124, it can be seen how the batch of data is applied to the training with the TensorFlow session. Next, we see a conditional, which evaluates the precision periodically (each 200 steps in this case), saving the result in the summary and displaying it on the screen. Below we can see the output of the terminal belonging to this part.

```
step 200 / 885 (epoch 4.52 / 20):
  learning_rate = 2.84e-03, loss_average = 2.67e-01
  validation accuracy: 91.31 (2027 / 2220), f1 (weighted): 90.58, loss: 4.37e-01
  time: 147s (wall 13s)
```

```
step 400 / 885 (epoch 9.04 / 20):
  learning_rate = 1.62e-03, loss_average = 1.56e-01
  validation accuracy: 93.24 (2070 / 2220), f1 (weighted): 92.63, loss: 3.43e-01
  time: 291s (wall 26s)
step 600 / 885 (epoch 13.56 / 20):
  learning_rate = 9.20e-04, loss_average = 1.35e-01
  validation accuracy: 93.92 (2085 / 2220), f1 (weighted): 93.43, loss: 3.06e-01
  time: 435s (wall 38s)
step 800 / 885 (epoch 18.08 / 20):
  learning_rate = 4.97e-04, loss_average = 1.23e-01
  validation accuracy: 94.37 (2095 / 2220), f1 (weighted): 93.98, loss: 2.96e-01
  time: 574s (wall 50s)
step 885 / 885 (epoch 20.00 / 20):
  learning_rate = 4.05e-04, loss_average = 1.18e-01
  validation accuracy: 94.10 (2089 / 2220), f1 (weighted): 93.44, loss: 2.89e-01
  time: 645s (wall 57s)
validation accuracy: peak = 94.37, mean = 93.39
train accuracy: 96.29 (8522 / 8850), f1 (weighted): 95.84, loss: 1.36e-01
time: 14s (wall 2s)
test  accuracy: 94.10 (2089 / 2220), f1 (weighted): 93.44, loss: 2.89e-01
time: 5s (wall 1s)
Validation  accuracy: 94.10 (2089 / 2220), f1 (weighted): 93.44, loss: 2.89e-01
```

It's important to emphasize that, in lines 131 and 132 where the TensorFlow graph is evaluated, the collected results come from the processed training data and the validation data, we can see these lines below, that as each one has the same data, are doing exactly the same.

```
stringV, accuracyV, f1, lossV = self.evaluate(val_data, val_labels, sess)
stringT, accuracyT, f1T, lossT = self.evaluate(train_data, train_labels, sess)#
```

Outside of the conditional, a print can be observed, which is found in the last line of the last output shown above, it indicates the final results of the training. These results are returned at the fit function end. So, we return to the utils.py code, to the test function and continue from line 293, in which we had diverted to the library. Now the code evaluates the generated model using the models evaluate method of the class. It uses the data that is passed to it as a parameter, and saves the confusion matrix in Matlab format, using the part of the name corresponding to the actual period (test, train, val). Once the GCNN is trained, and the results have been tested with the test and validation data, it can be used.

Post modelling gene perturbation

Continuing with the test function in line 305, from the utils.py library, it can be seen how the code regarding the post modelling analysis begins. This section is difficult to explain since in it, several problems are detected. To better explain it, let's look it at Figure 19, which is a copy of figure 3 on page 5 of the paper **[1]** that is being analysed. This figure shows us the proposal that the authors make to determine the relevance of a gene for a specific type of tumour. To achieve this algorithm, the authors have relied on the algorithm proposed in **[51]** by TaeJin Ahn et al.

Gene-Effect Score Algorithm Calculating Individual gene's contribution per class
Input: $X \in \mathbb{R}^{nxd}$, matrix of *n* number of samples and *d* genes
Input: Y $\in \mathbb{R}^{nx1}$, *vector of the true class*
Output: $y \in \mathbb{R}^{cxd}$, matrix of each gene contribution of *c* number of classes

```
1:  for sample i 1 to n do
2:     Put X[i,:] in trained GCN and get predictions p
3:     for gene j 1 to d do
4:        modify X[i,j] = 0
5:        Put X[i,:] in trained GCN and get predictions p0'
6:        modify X[i,j] = 1
7:        Put X[i,:] in trained GCN and get predictions p1'
8:        Restore X to original input
9:        contribution = max(abs(p(Y(i)) – p0'(Y(i)),abs(p(Y(i)) – p1'(Y(i))))
10:       if p(Y(i)) > Threshold
11:          y(Y(i),j) += contribution
12:       end
13:    end
14: end
15:
16: for class k 1 to c do
17:     min = minimum(y(k,:))
18:    max= maximum(y(k,:))
19:    for gene j 1 to d do
20:       y(k,j) = (y(k,j)-min)/(max-min)
21:    end
22: end
```

Figure 19 Post modelling gene perturbation pseudocode

But now, focusing into this code part, the following lines that are attached below.

```
305:### starting the dropout code
306: Knockdown = []
307: Knockup = []
308: for i in range(7100):
309:     if i % 50 == 0:
310:         print(' On gene number')
311:         print(i)
312:     Datadown = test_data
313:     Datadown[:,i]=0
314:     Knockdown.append(model.evaluate(Datadown, test_labels, pre = True))
315:     Dataup = test_data
316:     Dataup[:,i]=0
317:     Knockup.append(model.evaluate(Dataup, test_labels, pre = True))
318: sio.savemat('Knockdown.mat', {'Knockdown':Knockdown})
319: sio.savemat('Knockup.mat', {'Knockup':Knockup})
```

As can be seen in this piece of code, the calculations between lines 1 to 7 of the pseudocode are being more or less carried out, using the test data. Once the predictions for each gene have been made, they are stored raw in a list, which contains all the predictions, to finally be saved in a Matlab file. It should be noted that in the code provided by the authors, we can see a hardcoded 7100 in line 308 that can produce some error in some circumstances, this fact, will be commented later into the problems section. Also, seems that there's an important error, since if the code is observed carefully, the part that goes from line 312 to 314 is identical to the one that goes from line 314 to 317, it seems that the zero in line 316 should be a one. The only difference is the name of the variable where the result is finally stored, but it should be the same. In the absence of a detailed check, since the code provided does not offer any tool to analyse it, and the paper does not comment on this either, the two matrices resulting from this part of the code

should be the same. After observing this, it seems that to obtain the final data, they have used some other code, either Python or Matlab, which is not provided by the authors.

Returning from the function to the main code PPI.py, in which it calls the show method of the model_perf object. This function simply prints some data about the model in the terminal, and displays a plot on the screen with information about the training of the model. It also saves a confusion matrix using the test data. And at the end, the terminal prints the peaks in validation data, the max and the max of the ending ones.

```
  accuracy        F1             loss        time [ms]  name
test  train   test  train   test     train
93.92 95.46   93.24 94.93   2.81e-01 1.58e-01    61   Run1
[92.19254097 94.25732353 95.10904633 95.45747838]
peak
94.87951807228916
ending peak
94.87951807228916
```

Work done

As seen in the analysis section, there are several things that are not completely correct. From hardcoded values, important errors, some strange application of the usual techniques and the files where the output data are saved incorrectly. Also, the fact that parts of the code are based on very old libraries is an added problem. This has led me to have to modify many parts of the code, create some new ones to compare using the usual techniques to compare the results. All of this will be explained in detail in the discussion section below, and the results will be compared in the next main section.

In the following subsections, the most relevant modifications made or the code files added are exposed. Below are the sources from where you can download the original and modified code, some comments on the software and hardware used during the tests. Then, how to use these programs, from download to execution.

Code added or modified

This has been an important part of the master's thesis, since at first the objective was to validate the results, but as I progressed in understanding the code, I saw unusual things and had to investigate the content of the libraries. Finally, I have had to know how each library works, each function and in many cases, each call to third-party libraries, to find a justification for how each thing is done. In the following subsections we can see a brief description of the modified or generated files.

PPI.py

Since the code supplied in the repository only works with the protein-to-protein graph, the option to pass as a parameter the type of graph we want to generate has been added. In the terminal, when the program is executed, we can add the COEX parameter, the results will be generated by means of a GCNN that is based on the interaction of co-expressions between genes, without using the genes not related to any other. If instead the parameter is COEXS, the genes that are not linked by the co-expression relationship with others are added, but without edge, also known as singletons. If we pass the PPI parameter, the GCNN graph will be based on the protein-to-protein interaction. If the parameter is PPIS, as in the previous case, all nodes will be added, even the non-existent interactions. To do this, many tests have been carried out, since each type of GCNN has different hyperparameters and acquires different data for genes and graph creation. As has already been said, the way to make the types of graph work I had to find some things in the paper, some others in the code comments, often incomplete. A folder called input data has also been added to hold all needed inputs, in which all the data downloaded from **[44]** should be added, since due to its size it cannot be stored in GitHub. As a result of the execution, a confusion matrix is created in PNG format like the one in Figure 23, and a CSV file with the same data for later analysis. After executing the post-modelling stage of the data, the plots that can be seen in Figure 35 and Figure 37 are also generated, at the same time these data are also saved in CSV format for future use.

RandomPPI.py

A duplicate of the modified PPY.py file has also been generated that allows modifying the input data. Since the original code always uses the same data, and also uses the same block for the validation and the training of the GCNN. The main problem is that while the GCNN is trained, the training adjusts the parameters depending on the periodic validation results. Then, when the trained GCNN is tested, is classifying data already

seen, at this way, the result should be good in any way. For this reason, in this new version, all the data has been grouped in a block and it's sorted randomly, and after selecting the percentage to use for train using a variable, the code divides the train data, and then, splits the remaining data in two blocks, one to do the validation, and other for the final test. Once the data is divided, the code proceeds, using the same code as the original, and leaving the same outputs than the PPI.py code. The results obtained by this code are those shown in its corresponding results section in order to compare the two versions.

OnlyGCNN.py

This code uses the modified library code presented in the next subsection. It is a copy of PPI.py, but instead of using the utils.py library to carry out the whole process, it uses the modified utilsNo.py library to create just the GCNN, do the test and generate the confusion matrix. Once this is done, it ends without carry out the post-processing. This file has been created in order to perform tests with different percentages of train, validation and test data. To be able to average the precision of several executions and quickly get the confusion matrix for all kinds of graphs, or train data sizes. As a result, in this case, only the confusion matrix in PNG and CSV versions are created.

Lib/models.py

In this library, the creation of confusion matrices has been completely modified, since it is not necessary to change the programming language to generate a confusion matrix. Therefore, a plot has been created directly so that the confusion matrix appears directly in a well-formatted PNG file. A TensorFlow parameter has also been added together with the multiprocessing library, so that it can take full advantage of the number of processors on the machine where the code is executed.

Lib/utils.py

This is the library file in which a greater number of modifications have been carried out because it is where the authors create the GCNN and for some reason, they have inserted the post-modelling analysis of the test function. First, the error that caused knockdown and knock-up to do the same has been fixed, in order to be able to perform the analysis of the results correctly later. The way in which the results data are saved has also been modified, since for some reason, perhaps related to libraries, the output data saved was useless.

In this library, an attempt was also made to parallelize the post-processing, but it was discarded for two reasons. The first is that when parallelizing in a very fine way, the overhead of creating and transferring information to memory was slower than doing it with serialized code. The second reason is that when the post modelling analysis is carried out, after the first 2000 genes, the GCCN is returning always the classes 0, or 28. For the parallelization problem, a parameter was added to the model's library, that allows to take full advantage of the processors, since TensorFlow is already prepared to run in parallel. For the second one, we can see some code commented because I had to execute the code many times acquiring the data in blocks of 1000 genes.

Lib/utilsNo.py

As previously stated, this is a copy of the utils library modified to exit the code after generating the confusion matrices.

Lib/Coarsening.py

In this library line 241 has simply been commented, which contains an assertion that made it impossible to execute the co-expression graph.

postprocessing.py

This code has been made to read the CSV files generated from the post modelling analysis in the main code, and it generates plots like the ones in Figure 35 and Figure 37 for a better visual understanding.

GitHub original project

The original code can be cloned from the repository that Ricardo Ramirez offers us in **[28]**. The data of the genes, and the adjacency matrices to generate the four graphs, can be downloaded from **[44]**, as indicated in the repository files. To use this code, the better option is using the repository explained in the next subsection, where the original code is also supplied, only with the modifications needed to compute the many different graphs. If you want to use the original, without any modification, you will have to spend a large amount of time for it to work, since you will have to install the appropriate libraries, and make several modifications to several files. This information has not been added to memory, as they are small modifications, scattered throughout the code, which any person minimally used to code, should be able to fix.

GitHub modified project

In this repository **[52]** you can find all the files necessary for the execution of the final code delivered with this master thesis. To execute it, you must follow the indications in the final code section below. In the following lines the repository structure is explained.

As can be seen in the Figure 20, repository has 7 folders and 5 files in its root.

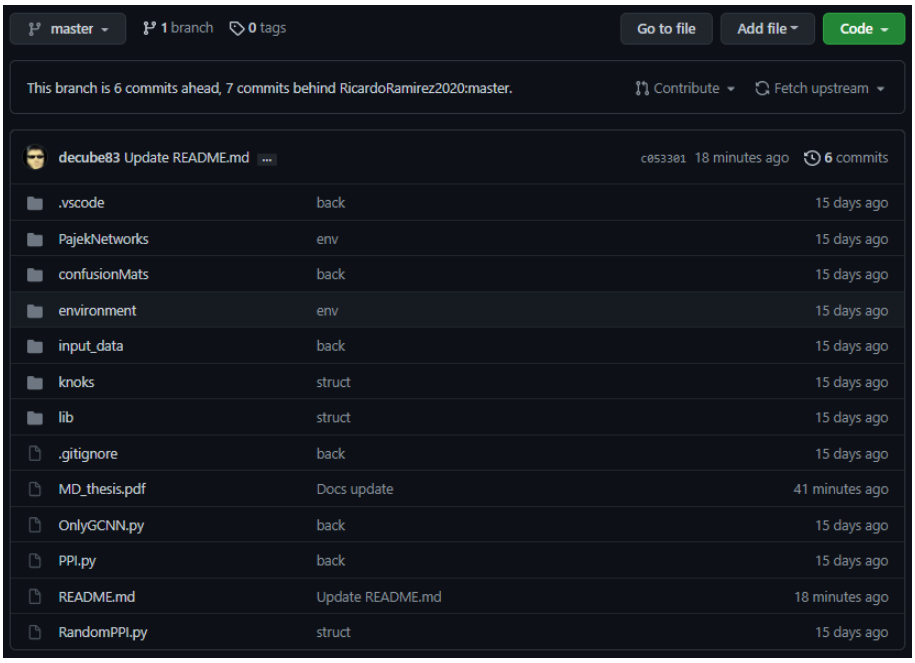

Figure 20 Modified code repository

- **RandomPPI.py**: Code explained in final code section.
- **PPI.py**: original code, explained in final code section.
- **OnlyGCNN.py**: Code, explained in final code section.
- **MD_thesis.pdf**: The final documentation.
- **.gitignore**: File to ignore some useless project files in GIT
- **Lib**: Contains the libraries used in the project.
- **Knocks**: Place to move knock-up and down files once are created.
- **Input data**: Place for the .mat files, explained into folder with a readme.
- **Environment**: Contains the YML file to build the Conda environment.
- **ConfusionMats**: Folder to place the generated confusion matrices to analyze.
- **PajekNetworks**: Contains the Pajek files extracted from one code execution.
- **.vscode**: Sample Visual Studio configuration to run code remotely.

Conda environment

Due to the use of obsolete libraries, if someone wants to execute the code, it is highly recommended to create a virtual environment. In this way, the projects that are in progress on the machine will not be affected by the downgrade of the libraries. In the repository that contains the code that accompanies this thesis **[52]**, into the environment folder, there's the environment.yml file, which can be used to create a Conda virtual environment with the necessary libraries. In case you want to execute the code, you simply have to create a virtual environment with the libraries defined in it. In this way, in case someone wants to delve into this code, they will not have to waste time again, simply to get a code execution. A detailed use explanation about it, can be found in the final code section.

Hardware

The entire project has been carried out working on a remote high-performance computer. It has two Xeon processors, and 132GB of RAM, using an Ubuntu server 18 operating system. The exact model is not indicated as it is custom made by the SuperMicro company and it is not necessary to detail the components one by one. This has been done because it is better to use a machine located in an air-conditioned environment, and prepared to perform intensive calculations, than a desktop computer that can be more punished by these calculations.

On this machine, a code execution using the protein interaction graph takes about two hours and twenty minutes. The difference by running it with a desktop computer, should not be very large, since TensorFlow only took advantage of 30% of the calculation capacity in this high-performance computer, because the library itself adjusts to what it needs, and not because it has an excessive computing power capacity it will go quicker.

IDE

To work on this code, at least in the remote way that I did, I recommend using the Microsoft Visual Studio Code IDE. Since information is very easily found to make connections to remote machines, to execute the code remotely, and install extensions that can be useful. In addition to having a good debugger, which can be some time spending to configure remotely, but once done, it avoids many headaches, and speeds up the work. An example configuration to run code in remote can be found in the .vscode folder.

Final code

This section explains, step by step, the process that should be followed to execute the final codes that are attached to this document in a Linux environment. To run it in Widows, some details in the environment set up can be different.

Clone repository

To begin, we must copy the code found in the **[52]** repository. For this we can download all the code in a zip directly from the page, or having git installed, it can be copied to a folder by entering it and entering in the terminal:

"git clone https://github.com/decube83/GCN_Cancer.git"

Virtual environment

To prepare the virtual environment using Conda, we must first have the package installed. To recreate the environment used, we must write in the terminal *"conda env create -f environment.yml"* where environment.yml is the file located inside the environment folder. This will create an environment called tfm. To use it, we must activate it every time we access it, with the command *"conda activate tfm"*. Then, we will already have all the libraries in the version necessary for the code to work.

Main code PPI.py

The original code has been modified to be able to pass as a parameter the type of graph to use in the GCNN creation. We have the next options, COEX to use the adjacency matrix for the genetic relationship based on the co-expressions, COEXS to use the same by adding the genes that do not have any link after the pre-processing that the author performed in Matlab. PPI to generate the graph based on the interaction between proteins and PPIS for interaction between proteins adding the genes without direct relationship. If we want to obtain results similar to those analysed in the paper, we must *enter "python3 PPI.py PPIS"* in the terminal. The genetic input data differ depending on the type of graph chosen, but the code has been prepared to load it right with the input parameter, as well as the different parameters for the creation of the GCNN. Furthermore, if we want to carry out tests with different amounts of data for training, validation and testing, we can use the file randomPPI.py file in which the percentage of data destined to these variables can be easily adjusted. It is only necessary to modify the variable trainPercentage in line 157 in which we can see that is set to 70, which is the percentage of data used for training. The data for validation and testing are divided in two equal parts. The last code presented is the OnlyGCNN.py. The goal of this code is to generate the GCNN, check its accuracy, and generate the confusion matrix from the test data. Within this file we can find three blocks prepared for commenting and uncommenting. One of them, as can be read in the comments, works in a very similar way to the original. The way of dividing the data has simply been changed to check that the data movements performed were correct and similar results are achieved. The second block behaves like the first, but choosing the data randomly. Once the data size is selected, it uses the same data for the test and validation, as in the original approximation. The last block allows us to modify the proportion of data destined for each block, in addition, these are randomly mixed to add one more point of uncertainty.

Except for the code where only the GCNN is generated, where the creation of it can be modified, the behaviour of the others is similar to the original, with the modifications that have had carried out for its correct operation, obviously.

Post modelling analysis postprocessing.py

This code, that can be found in the knock folder, is in charge to generate the plots from the two CSVs resulting from the post-modelling analysis results files created using the main code. To do this, after executing the main code, it is recommended to rename the output files knockDown.csv and knockUp.csv to one that makes it more recognizable as PPIS_knockUp.csv and the same for the down (it is a way to save many different files and generate the results at once). So, to run this code, we must have these files in the same folder as the code. To execute it, we should type "python3 posprocessing.py PPIS_knock" (following the renaming exposed before. If we have renamed using COEX_knockUp, whe should use COEX_knock). Once code finishes, it creates four files that start with the name passed by parameter. Two are CSV that can be used for further analysis, one of them that finish with the name "_genes affecting", contains an array with the same elements than columns in the input data. It means that, modifying the gene at position 0, the number of predictions affected is the one that there's in the first position. In the second position we can see the number of wrong predictions modifying the gene 1, and so on. The second CSV file named "tumours_affected" contains an array with 34 elements that are tumour classes following the order used in the paper and in this document. The number in each position means that this class is being affected by n number of gene modification. The PNG files created are the ones shown in the results section under the post modelling subsection. The results are the same than the CSV ones, but in a graphic way, that helps to understand better the results.

Confdiff.py

This code goal is comparing two confusion matrices. The code can be found in confusionMats folder. The routes for the two confusion matrix CVS should be sent as a parameter, moreover it requires a third parameter that the code will append to the final filename to distinguish it. The code loads the first parameter route matrix, and from this, subtracts in each cell the second matrix values. The result is a new matrix that in each cell contains a positive value in a cell where the first matrix has a greater number, a negative value if the bigger value is in the second matrix, and a dot if the values are equal. The final filename is "xxx_difference.png" if we send as third parameter xxx. We can see the code outputs in result section in the subsections ending with different title.

Results

In the following subsections the results obtained from the different parts of the code are exposed. We can see in the first subsection the results of the precision from the different GCNNs created. The following subsection shows the confusion matrices obtained from the creation of several GCNNs. Finally, in the last one, the results of the final data post modelling are shown. Details of the exposed results can be found in the introduction to each of the subsections.

GCNN test results

Figure 21 below shows the means and standard deviation for the accuracy in ten runs of the GCNN creation and test for the different types of input data detailed below. The leftmost column shows the types of graph used. In the top row, first we have the results obtained after the execution of the original code. Following, we have the results of training the GCNN by selecting the data randomly from the entire set of input data, in this code the technique of using the same data for validation and for testing has also been used, to be able to compare the results with the original ones. In the modified code column, we have the results of using 70% of the entire input data set, randomly selected, as training data. A 15% is used as validation and the other 15% as test, also randomly selected. In this last code, the test results are based on data that the GCNN has never seen.

ACCURACY

10 execution averages

	Original code	Original code using random selected data	Modified code using random data 70% train, 15% validation, 15% test
Protein-to-protein interaction	81.94 ± 8.66	84.67 ± 5.37	87.76 ± 1.85
Protein-to-protein interaction with singletons	93.82 ± 1.17	93.96 ± 0.55	93.81 ± 0.66
Co-expression	92.57 ± 2.73	91.72 ± 2.68	80.25 ± 2.68
Co-expression with singletons	93.32 ± 0.41	93.05 ± 0.78	93.07 ± 0.37

Figure 21 Accuracy values obtained in code executions

In the Figure 22 below, we can see the results published by the authors.

	Co-expression +singleton graph	Co-expression graph	PPI+singleton graph	PPI graph
Mean ± std	94.23% ± 0.146	94.24% ± 0.251	94.61% ± 0.107	88.98% ± 0.883
Peak	94.43%	94.67%	94.71%	89.99%
Mean precision	91.39%	92.06%	92.76%	87.75%
Mean recall	92.30%	91.39%	92.19%	83.79%
Mean training loss	0.19	0.51	0.2	0.38
Mean validation loss	0.30	1.05	0.49	0.91

Figure 22 Paper's GCNNs results

Confusions by GCNN

The following subsections show the confusion matrices obtained on the predictions of each type of GCNN, one for the original code, and another one for the modified code. Both contain the results on the entire data set. Based on these two matrices, a third matrix has been generated, which contains the result of subtracting from the values of the original matrix, those of the modified matrix. In this way, in the position where the number is positive, it means that the value is higher in the original. On the other side, if the value is negative, the highest value corresponds to the matrix generated with the modified code. If the values are the same, it has been marked with a point to facilitate its observation.

Protein-to-protein interaction graph original

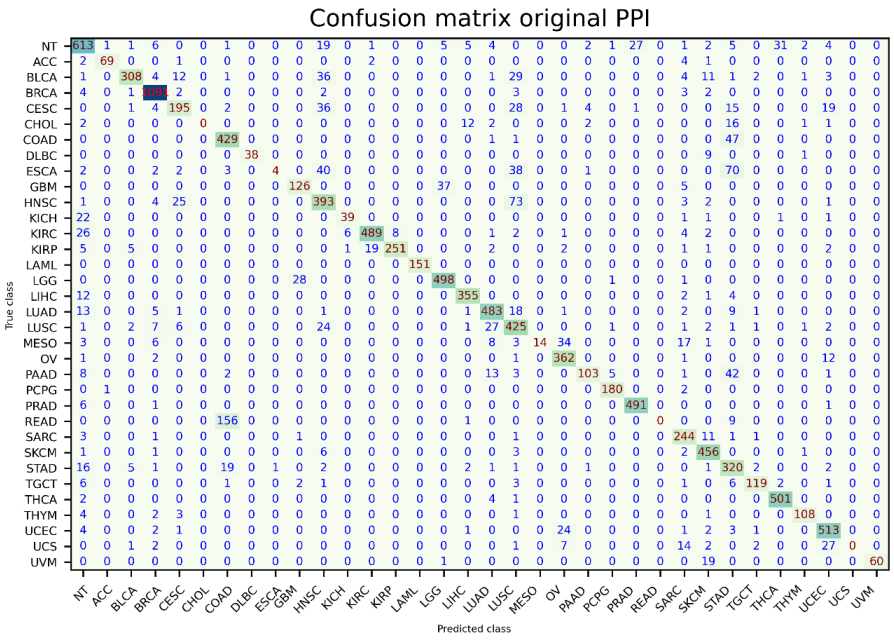

Figure 23 Original code PPI GCNN confusion matrix

Protein-to-protein interaction graph modified

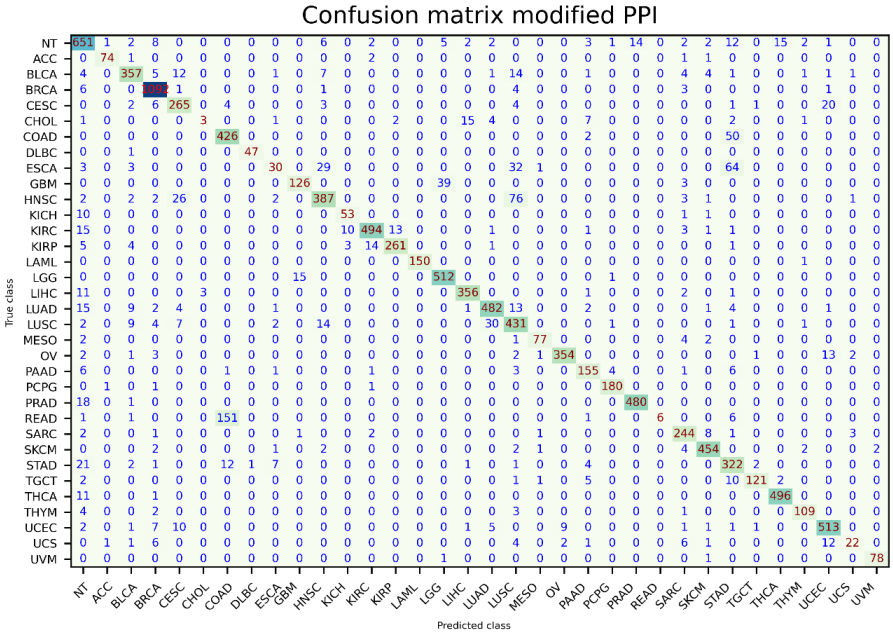

Figure 24 Modified code PPI GCNN confusion matrix

Protein-to-protein difference

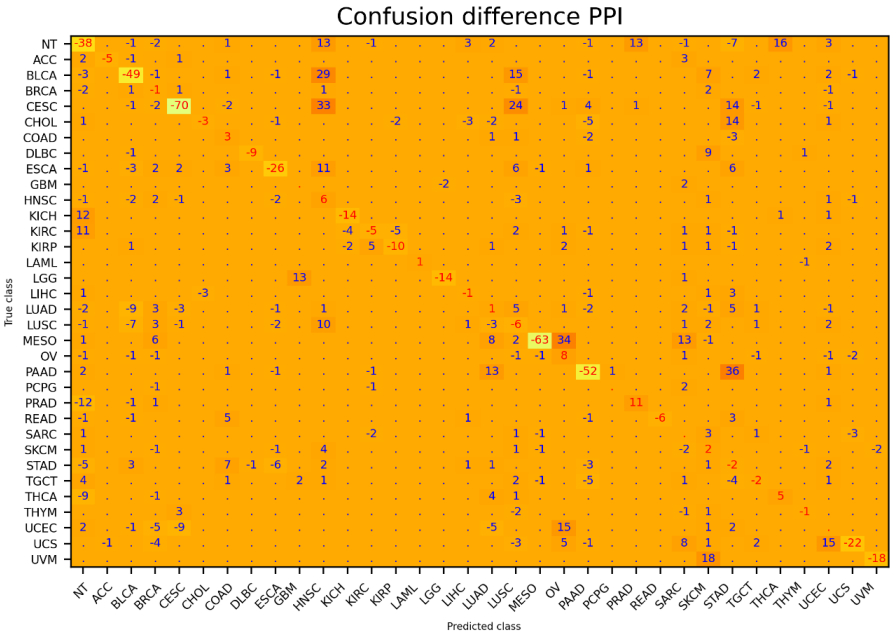

Figure 25 PPI original GCNN - PPI modified GCNN

Protein-to-protein interaction with singletons graph original

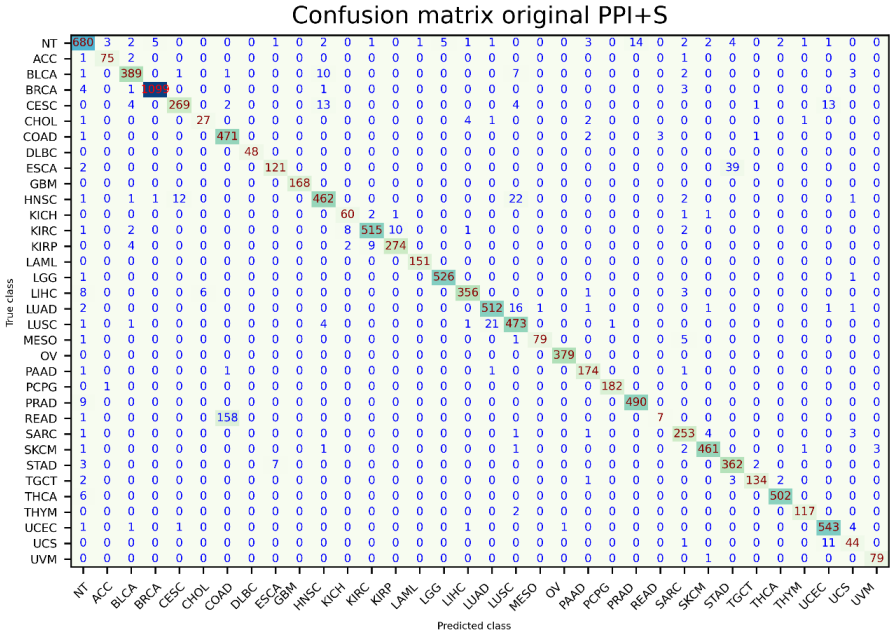

Figure 26 Original code PPI+S GCNN confusion matrix

Protein-to-protein interaction with singletons modified

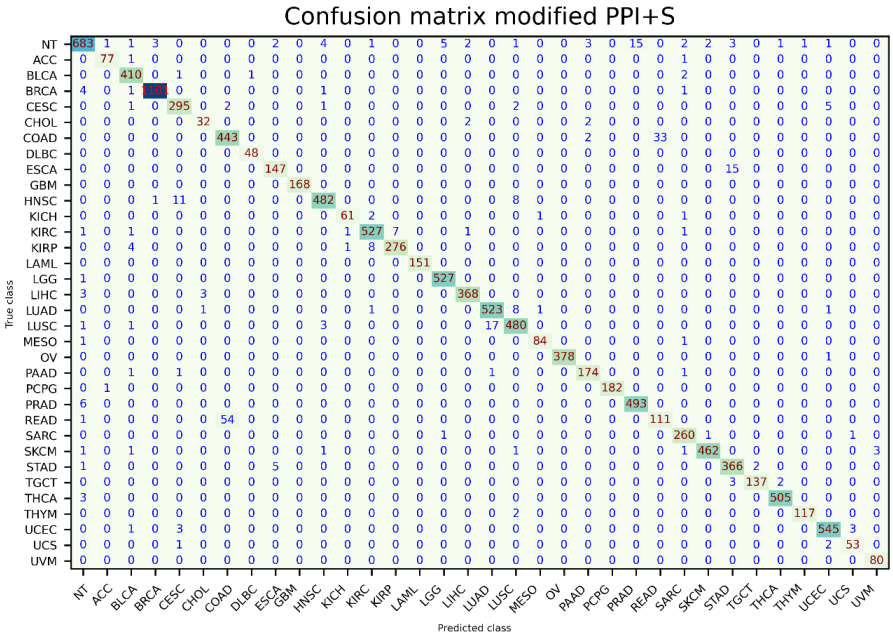

Figure 27 Modified code PPI+S GCNN confusion matrix

Protein-to-protein interaction with singletons difference

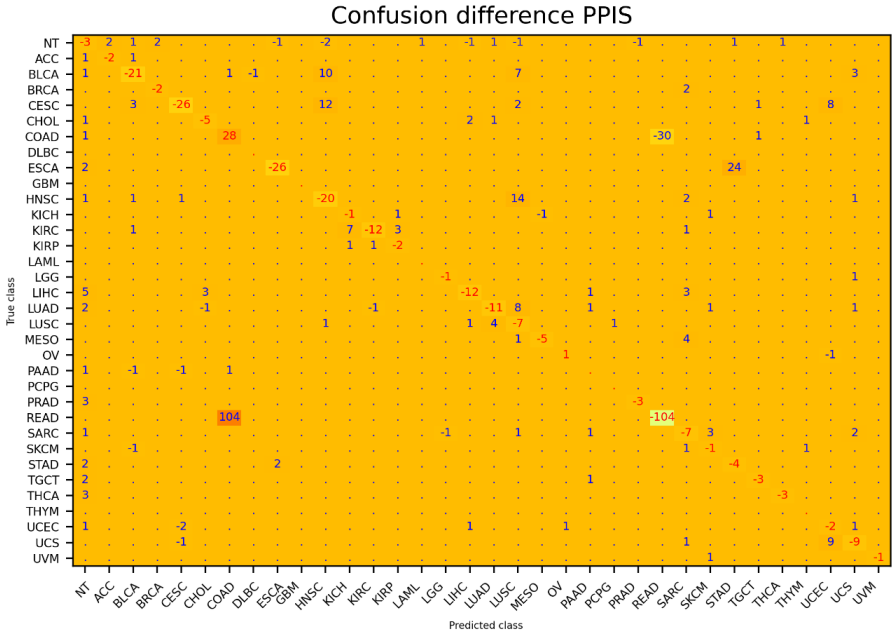

Figure 28 Original PPI+S GCNN - Modified PPI+S GCNN

Co-expression graph original

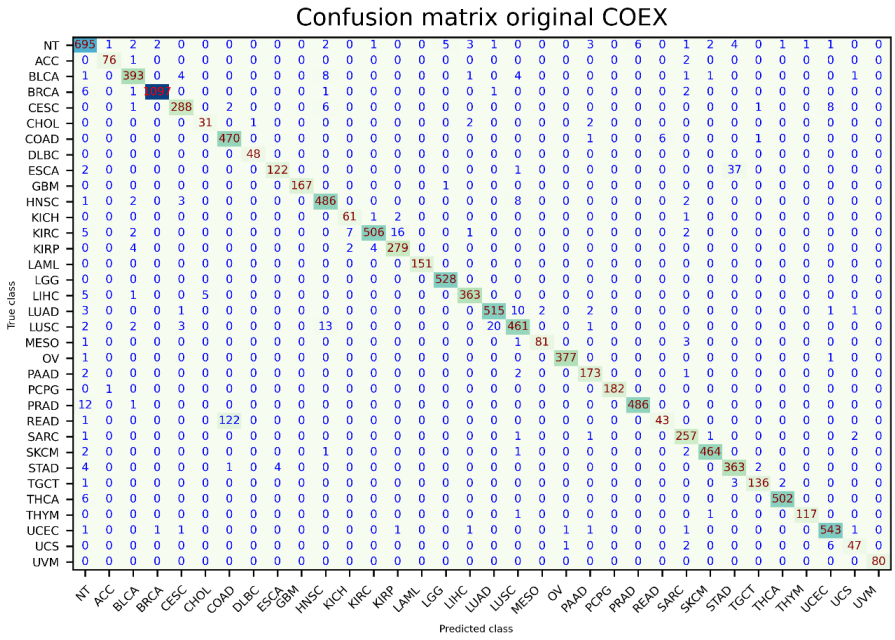

Figure 29 Original code co-expression GCNN confusion matrix

Co-expression graph modified

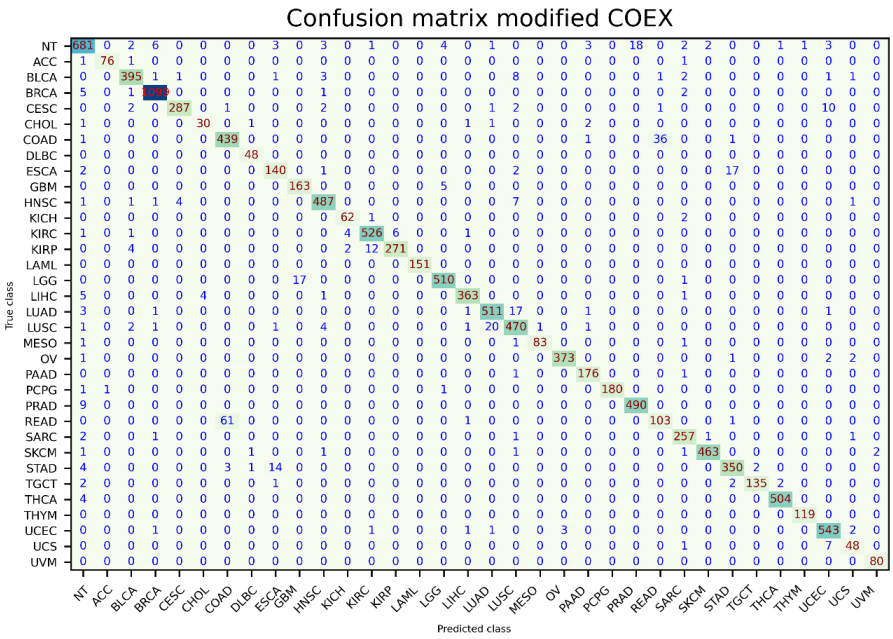

Figure 30 Modified code co-expression GCNN confusion matrix

Co-expression difference

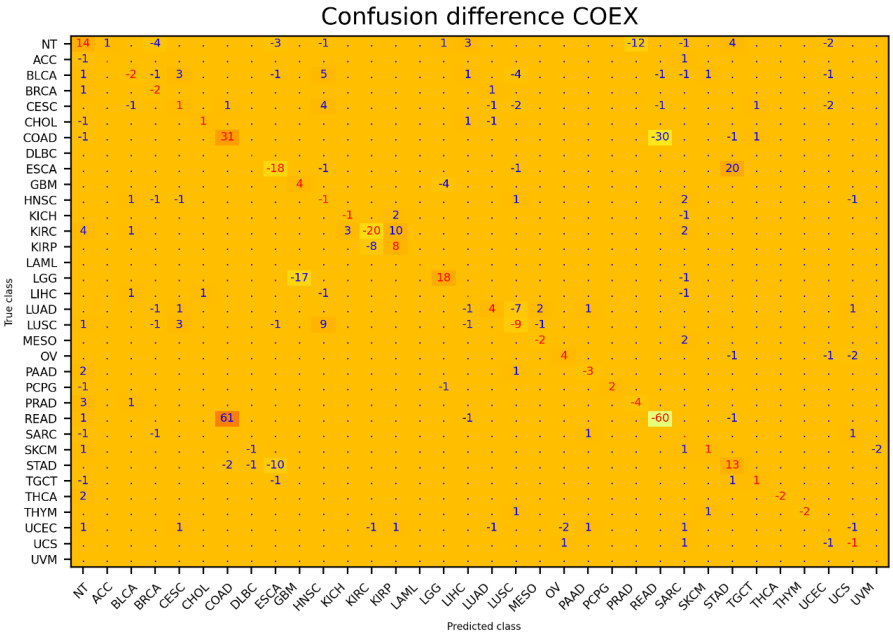

Figure 31 Original COEX GCNN- Modified COEX GCNN

Co-expression graph with singletons original

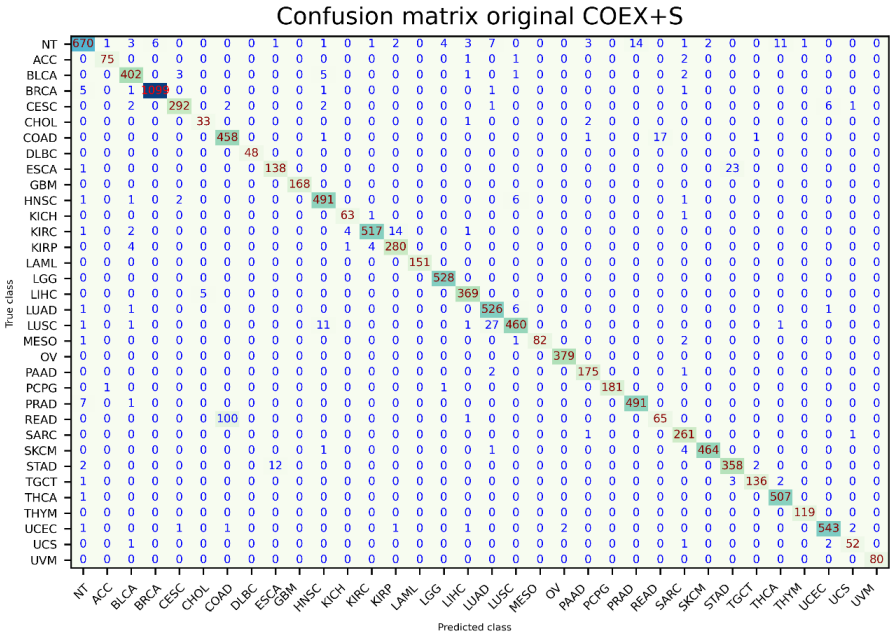

Figure 32 Original code co-expression with singletons GCNN confusion matrix

Co-expression graph with singletons modified

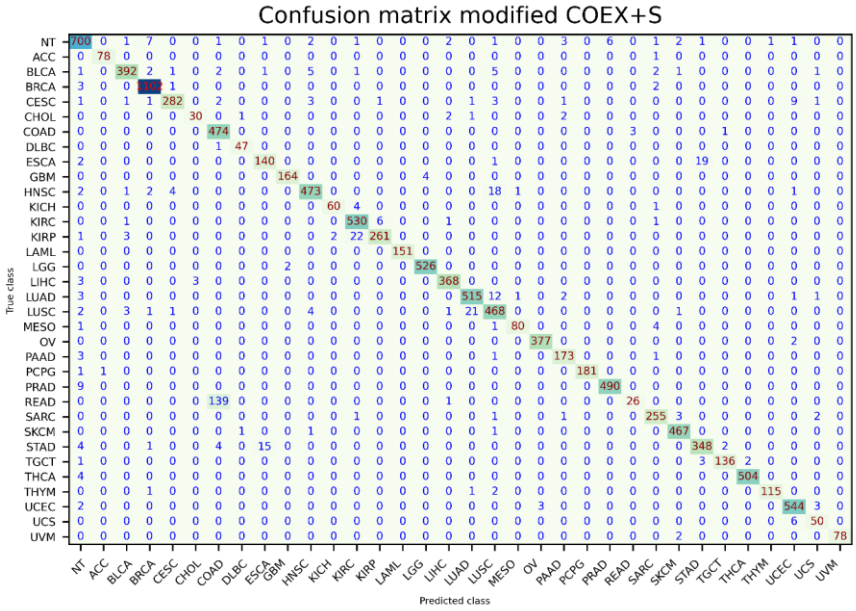

Figure 33 Modified code co-expression with singletons GCNN confusion matrix

Co-expression graph with singletons modified

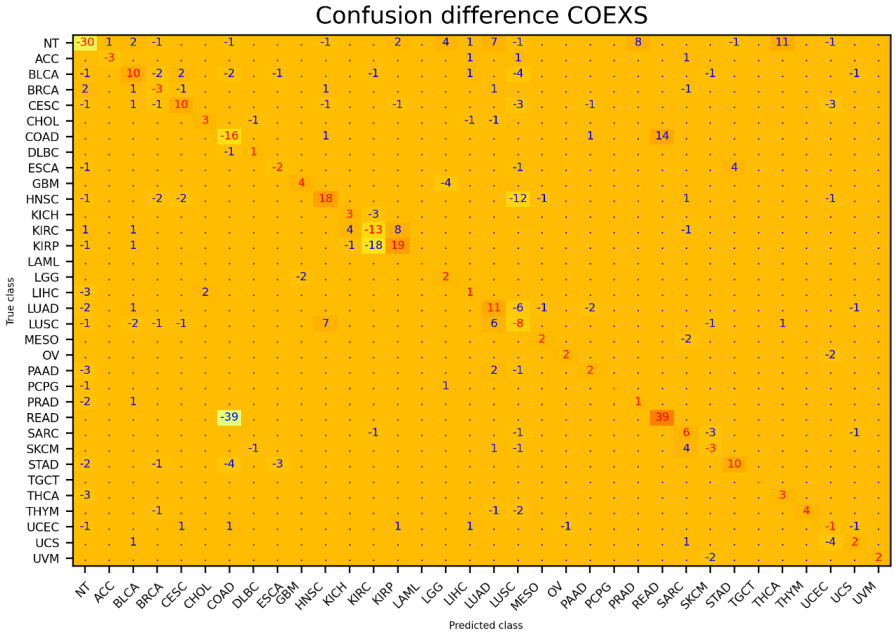

Figure 34 Original COEX+S GCNN - modified COEX+S GCNN

Post modelling analysis

This section shows the results of the post-modelling of the data. As seen previously, the value of a gene is set to zero and a prediction is executed, and then it is set to one and another prediction is made. The result is then compared with the labels. Due to the various problems found in the code, this study has only been carried out for the two types of graph with the best results, the co-expression with singletons and the protein-to-protein interaction with singletons. In addition, to continue with the comparison of the previous section, it has been done in the original code and for the modified one.

In the following subsections we can see the generated plots. In the one titled as classes affected by gene, it's represented the number of times that a label classification has been affected by a modification in one of the genes. To compute the result, the gen in position n, where n can be any gene from the first one to the last one in the input data chain, is set to zero in one data block, and to one in another. Then, a classification is made, if some of both data block result differs from the original class, an error is counted for the original class label. In the second type of plots presented, titled genes affecting predictions, the results affected by each gene in the input data chain that has been modified are seen. A notable difference in data is seen between the two types of graph, this is due to the fact that in the pre-processing carried out by the authors in Matlab, there are more relevant genes in the protein-to-protein interaction than the co-expression relationship.

PPI with singletons original classes affected by gene

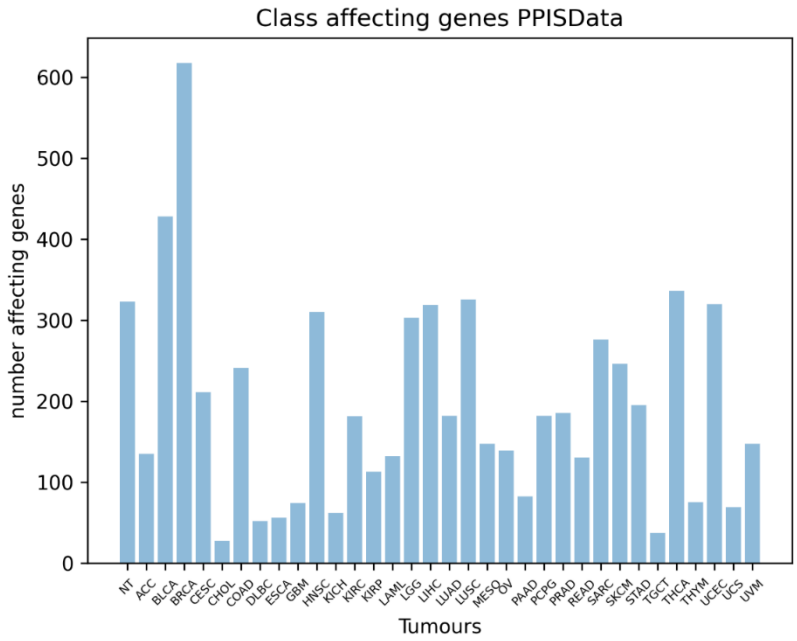

Figure 35 Tumours affected by n genes for original code GCNN PPI+S

PPI interaction with singletons modified classes affected by gene

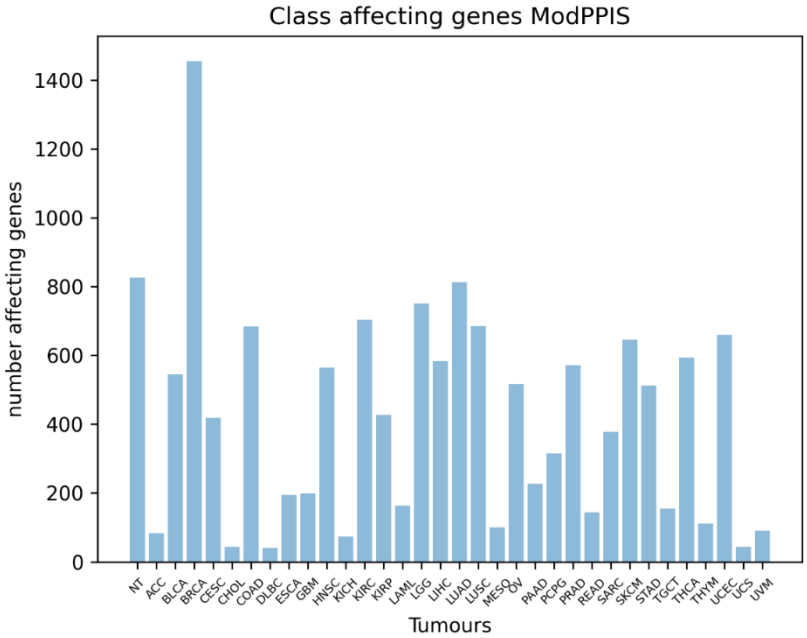

Figure 36 Tumours affected by n genes for modified code GCNN PPI+S

PPI interaction with singletons original genes affecting predictions

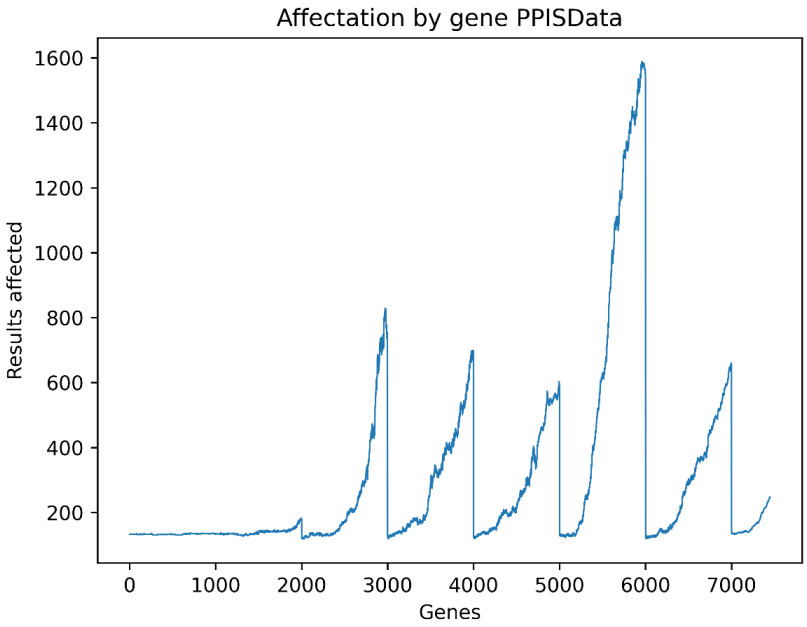

Figure 37 Predictions affected by gene modification original code PPI+S

PPI interaction with singletons modified genes affecting predictions

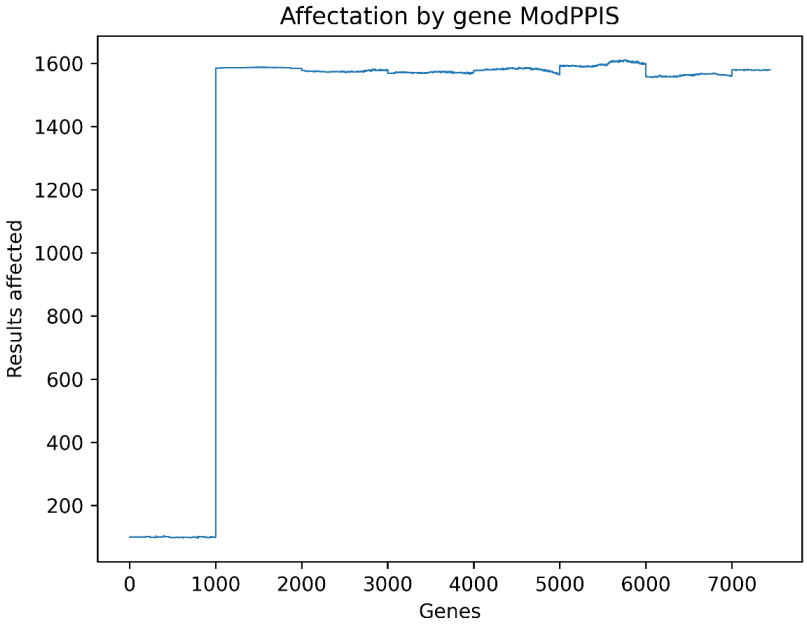

Figure 38 Predictions affected by gene modification modified code PPI+S

Co-expression with singletons original classes affected by gene

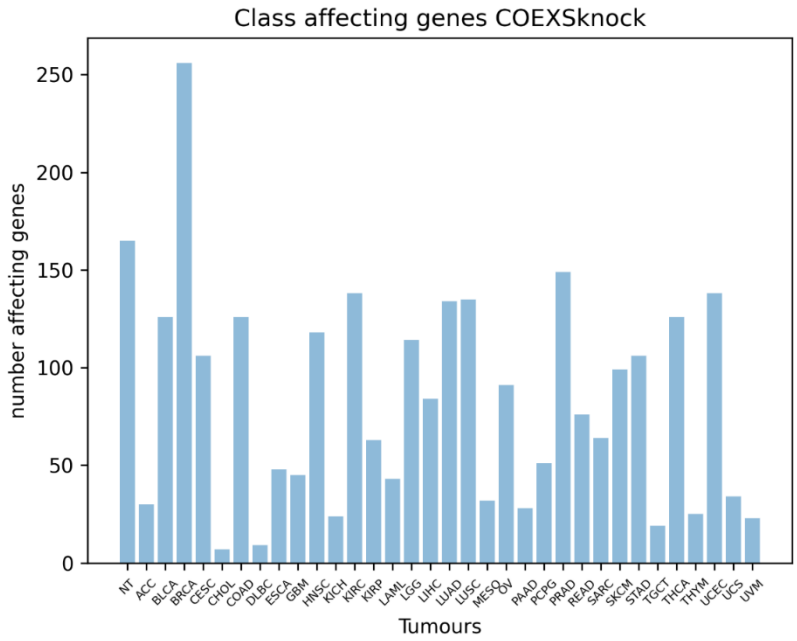

Figure 39 Tumours affected by n genes for original code GCNN COEX+S

Co-expression with singletons modified classes affected by gene

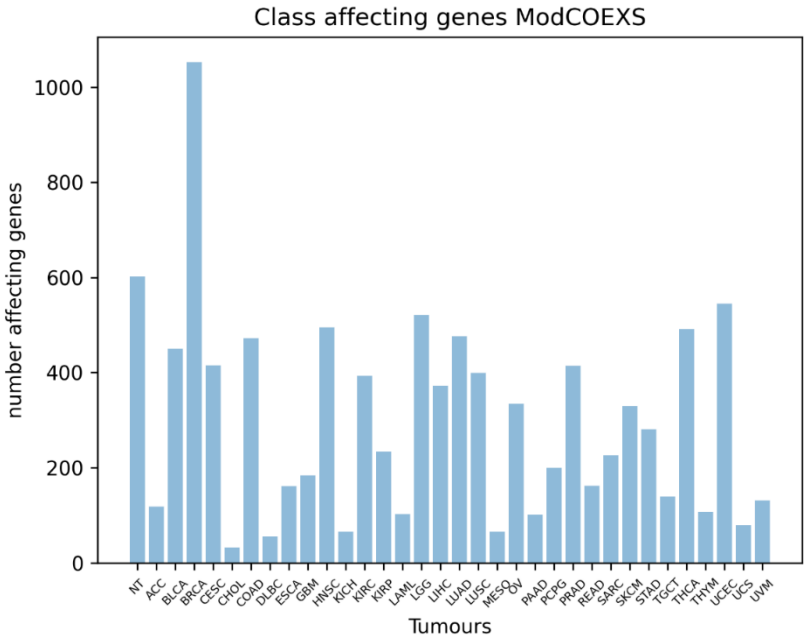

Figure 40 Tumours affected by n genes for modified code GCNN COEX+S

Co-expression with singletons original genes affecting predictions

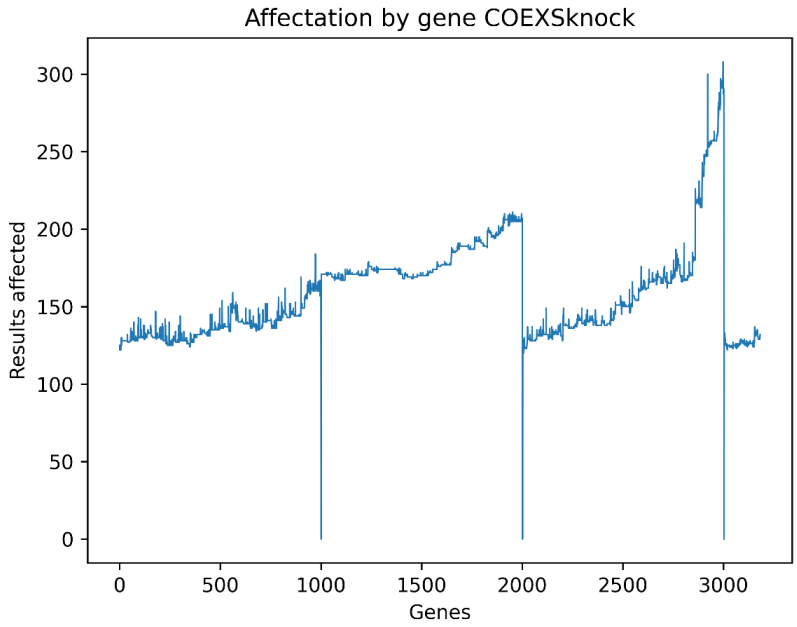

Figure 41 Predictions affected by gene modification original code COEX+S

Co-expression with singletons modified genes affecting predictions

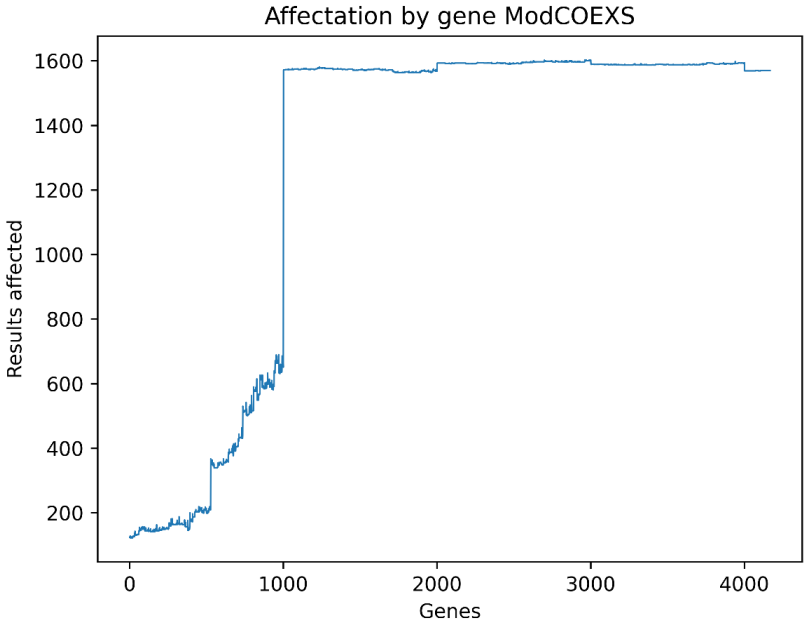

Figure 42 Predictions affected by gene modification modified code PPI+S

Results comparison

The following subsections comment the results of the three previous subsections, and the last one presents a final conclusion of this comparison of results.

Test results

As can be seen in Figure 21, between the original model and the one modified to create the GCNN with random data, there is not much difference. The results from the random data code in general have a lower precision, but it can be observed that the standard deviation is smaller in most cases. This indicates that the GCNN generation is somewhat more stable, perhaps because the type of data contained in the block does not contain a good mix of classes.

If we compare the results of the original code, that can be seen Figure 22 with the modified ones in Figure 21, it is observed that generally for all types of graph, the original code one has a better average precision. In this case, it is also observed how the standard deviation is smaller in the modified code. I take this opportunity to remember that the results are calculated based on test data, which the original code has already shown to the GCNN, and the modified one has not. Also, the parameters of the modified code, were only adjusted to get a good result in the graph protein-to-protein interaction with singletons. We see that the accuracy of this type of graph is somewhat lower than that of the original code and much lower than the one published by the authors, but the standard deviation is half that of the original code in our tests. Regarding this topic, the results obtained empirically, it is seen that they are very far from those that accompany the publication, especially the standard deviations.

Confusion comparison

In this section I want to focus more on the PPIS matrices, but taking a look at the others, we can see that the usual technique of using different data for validation and testing gives good results (Figure 24, Figure 27, Figure 30, Figure 33), in front to the one used by the authors (Figure 23, Figure 26, Figure 29, Figure 32). In most matrices that contain the difference of the results, the values of the diagonal are negative. In protein-to-protein interaction in Figure 25 the result of the diagonal sum is -318, in the version that includes the singleton genes in Figure 28 it is -264, in the co-expression one in Figure 31 the result is -17, and the only one that has been positive is the co-expression with singleton genes in Figure 31 with a +76.

Taking a closer look to the protein-to-protein interaction with singletons matrices and, considering a confusion greater than ten, being a relevant error, a total of ten errors can be found in the original code confusion matrix in Figure 26, and a total of six in the modified code in Figure 27. Of these six, four are in the same classes as the original, but with lower values, and one with one misclassifying error more. The last one, is an error quite high, that has been generated in the modified code confusing the COAD class with the READ. In the original code we see this same error in reverse, READ is confused with COAD. This, as discussed in the paper, may be due to their proximity, since COAD is colon adenocarcinoma, and READ is rectum's adenocarcinoma, being both very close. In reference to minor errors, it can be seen how the GCNN, which has been created with the usual technique, reduces and even eliminates most of them. To verify this, the output file in CSV has been used and, through the use of Microsoft Excel, the zeros have been counted in each matrix to know the number of correct cells. The result is 994 zeros for

the original code, times 1027 for the modified one. Which results in a difference of about 33 fewer errors, which are small, but they also count.

Tumour and gene affectation comparison

In the plots showing the predictions alterations for each tumour due to the gene nth modification (Figure 35, Figure 36, Figure 39, Figure 40), it can be observed how the GCNN of the code provided by the authors, always has a result at least very similar to the one created for this thesis, and generally, the modified one gets the double of alterations or more, than the original one. In comparison, the modified GCNN, only gives a slightly better result in the UCS and UVM classes. Both GCNNs cannot be directly compared, since they have been trained on different data sets, but it can be said that certain classes of tumour are more sensitive to variations as the global shape in the plot bars are quite similar.

In the plots in Figure 37 and Figure 38, it can be seen how in both protein-to-protein interaction graph with singletons, in the first thousand genes, both are close to one hundred different results. However, the modified GCNN in Figure 38, from this point, modifies around 1600 predictions for each gene, with slight variations till last gene above this number. On the other hand, the original code in Figure 37, remains in a hundred and a few, with five peaks, where the worst one reaches the modified GCNN stationary level. In the co-expression plots in Figure 41 and Figure 42, we see a similar behaviour. The peaks are less elevated for the case of the original GCNN, and one very similar to the previous one for the modified one, in this case, the alteration of results can be seen I a progressive way from gene 600 to the gene 1000, to reach the stationary level that is maintained till the last gene.

From the set of plots, it can be deduced that the GCNN proposed by the authors focuses more on certain genes to classify the tissue, while the one proposed in this master's thesis needs to know more about the full genes set, rather than in specific genes.

Results conclusion

Taking into account what was observed in the three previous sections, it can be said that the GCNN created with the author’s code is more focused in a certain type of genes to do the predictions, than the one that has been proposed using the usual techniques. From what has been seen both in the confusion matrices and in the global accuracy, it seems that the new proposal works slightly better, moreover knowing that part of the data that GCNN is classifying has never been seen. This leads us to think that there is no one better than another, you just have to prepare it for our final purpose. Perhaps the author's proposal is better to carry out an analysis of the gene’s relevance, and the proposal in this master's thesis works better by classifying the tissues in a more realistic environment. I personally think that delving into the research would be the right choice, having a clear path to follow, and conducting the experiments accordingly to achieve stronger conclusions.

Discussion

After analysing the code in detail, performing several code executions, and having carried out many modifications that have been exposed throughout this document, we can reach the conclusion that the code provided is not really the one used in the written paper. And if it were, the conclusions drawn could not be considered valid due to the important errors.

As it has been commented in previous sections, I contacted Yu-Fang Jin to ask him if there was any other code that they had been used, since the code, is very focused on giving the results presented in the paper only for the protein-to-protein interaction graph with singletons, without the ability to test the others. In addition, when making the modifications to get the code work, the execution ended in an assertion of a library when the co-expression graph was used. The answer to this question, was that there were five authors, and it worked correctly for all of them. This specific case cannot be attributed to a library, since it is an assertion in the graph library, which implies that the input data have some form not fully compatible with the library. Furthermore, the post-modelling of the results is incomplete.

The first problem is that we must trust that the input data is correct, because it is provided by the authors, and its reproducibility is very complicated, since it requires collecting records from many sources, and the sites that have these records, require many accreditations to download the data. Then, we have that when the post-modelling is executed, the same value is assigned to the up and down modified gene value, when one should have a zero and the other a one. Moreover, the number of genes in the protein-to-protein graphs post modelling isn't right, as in the paper said that there are 7091 genes, in the code the loop reaches the gene 7099, and after modifying it, to be able to compute all graphs without changes, and getting the size of the variable, the code process 7447 genes. Finally, the final step code is not provided, since only a Matlab file is generated, which in my case, perhaps due to the version of the libraries, is full of threes. But in any case, the 140 minutes it takes to execute the code are unnecessary if you only want to show that the result of the GCNN is the same than the one exposed in the paper. We can see some explanations about these problems, in the problems handled subsection below.

Entering into a more technical field, there are things done in a somewhat peculiar way. For example, the GCNN, when training modifies the learning rate based on the validations that it performs periodically. Using the same data for the validation as for the test, ensure a good final result. Personally, I would see it as a correct approximation in the case that the main objective of the paper was the post-processing of the data to know the relevance of the genes, but then I do not see right the assertion that the code is ready to be used in the cancer diagnosis, since it is not correctly tested with never seen data. The use of the same data as validation and test, was also asked of Yu-Fang Jin and his answer was very generic. He commented that 80% of the data had been used for training and 20% for testing. Which is not false, but not entirely true either, since the GCNN in the original code uses 80% to train, but it also modifies the training parameters in function of this 20% of the data used for validation, and when the training finishes, the test accuracy is performed with the same data. This was what led me to set myself the challenge of being able to doing it in a more usual way, since it would be closer to a real environment. For that reason, I modified the code, and I spent many time trying to achieve a result close to the paper one. Finally, as seen in the results section, the precision is not the same, but is quite close, the results variations are somewhat more stable in the creation of the GCNN. This second version could be proposed for diagnostic use. The proposal by the authors as

well, but it would be necessary to test it with other data that the GCNN have never seen, since the genetic code of tumours may have small variations that may affect the result.

As has been seen, the GCNN created with only a part of the data depends much more on different genes, than on a few in particular. This is a point that requires further research before asserting that it can be used. In addition to the added problem that in the medical environment 94% accuracy is very low, it can be used as a diagnostic aid, under human supervision. Since doctors are quite reluctant to blindly use artificial intelligence because it is difficult for them to understand how it works. In addition, as it can be read in the state of the art, there is ongoing research to be able to detect tumours through their biological remains in the blood and not from the tumour tissue, that makes the extraction more suitable. This does not mean that this study is irrelevant, the more knowledge we have from all sides, the better we will be able to understand this disease in order to detect and treat it better.

One last technical and important detail, that I would like to highlight, is that the paper says that 5-fold cross validation is used. After analysing the entire operation of the code, reviewing the TensorFlow instructions step by step, the GCNN is simply fed by the same data in different order several times, the validation data always being the same as those sent from the main code. After consulting with several people more expert than me in this field, in order not to make a mistake when writing these lines, it was concluded that the GCNN was fed in a basic and typical way.

Another aspect that I would like to note, which has not been discussed yet, is that the repository is somewhat messy. Upon downloading it, we can see how there is a Jupyter notebook with graph library use examples, along with an example that uses the mnist library and also the caches generated by the Python executions. These are small details, but they do not give a polish aspect to a publication, where only the strictly necessary files should be seen.

After commenting on the more general aspects, we are going to comment the results obtained in the original code and in the modified one, to which the parameters have been adjusted to obtain a good result in the protein-to-protein interaction graph with singletons. After several tests carried out, it seems that the authors focused basically on the GCNN parameters, but after obtaining good results, they did not delve into the structure. When performing tests, other structures with different levels of pooling were found that gave good results by changing the final fully connected layer of 1024 neurons. It is a subject in which it would be necessary to deepen, beyond carrying out a GridSearch. Focusing on the results, it can be seen how the GCNN created with less data is somewhat less accurate than the one that uses all of them, but as previously said, the success is on data that has never been seen, in contrast to the original, that is classifying previously seen data. This is a way to demonstrate that the GCNN works as expected, also we should keep in mind that the major part of the doctors would be reluctant to trust in it without anything more.

The final result exposed in the paper cannot be contrasted by the lack of code. However, to compare the two proposals, the same post-processing has been carried out on the original and modified GCNN. As it can be seen, the original one focuses more on some genes, probably because these relationships exist, and are more relevant. On the other hand, the one that has trained with less data, generates more errors when a gene is

modified. This is another aspect that should be explored, since it could come from an overfitting in the original code, or it could be the reality. Due to this same fact, the errors by type of tumour are greater, since the modified GCNN depends on a bigger set of genes to classify the result.

Problems handled

After the main discussion, in the following sections the most relevant problems in a deep way, some of them exposed before, that have been detected throughout the code analysis and some bugs will be exposed in detail.

Data

Leaving aside the first problems of understanding the Matlab data, I tried to get the original data, to try to do the same processing that the authors describe in **[29]**. I found the problem when I try to find the exact data. The author's explanation is correct, but the way to obtain the data is explained in a very generic way, finally, I had to blindly trust in the data provided, because some sources needs some special requirements to get a registration like **[53]**, or the data has changed. Moreover, the code to process it isn't provided. If the analysis of the input data is observed, we see how the authors simply comment that they work with an adjacency matrix to generate the graphs. This is correct, but I was very confused by the fact that something similar to an adjacency matrix exists, but the code uses the variables col, row, and value. Being value a huge array full of ones. Until I use the debugger, the input data was so weird for me.

Old software versions

A problem that has made me waste a big amount of time has been the use of obsolete libraries, such as TensorFlow 1.4, among others. I will not comment on the details one by one, because the list can be very long, and it is not relevant. To solve this, as we seen within the work done section, a solution is presented. This is based on the original code, modified in all the necessary points, ready to be executed under a virtual environment from which the configuration of the exact libraries necessary for the execution of the code will be available in the master's thesis repository. This way, in case someone wants to test the code, to analyse it in more depth, or wants to try an improvement, he should not waste time again to get the code to work as it should.

Dubious code

As it can be seen in the analysis of the code, there are some pieces of code that do not exactly match what is written in the paper. As we have seen, the code, once the problems with the libraries have been saved, is executed correctly, and gives a result similar to those exposed in the paper. The code, is only prepared to execute it on the PPI graph plus singletons. Observing the original code, we can see some annotations, some of them cut, on how to execute the co-expression graph, but really, more things than the explained ones have to be changed within the library, due to the hardcoded values that have been commented during the analysis. To solve this, the code that will be provided with this document will be executed according to an input parameter. I have relied on the comments in the paper, which refer to the data, in the comments of the code and in the analysis carried out, since without knowing the data movements well, this would have been impossible for me. Another of the complicated details that I found analysing the code, is the fact of how they loaded the data of the genes, as can be seen in the analysis section.

Data validation and test

As it can be seen in the original code analysis, the block of data that are assigned to the validation is the same that is used for the test. This helps to achieve a good result, since at the end, when you are performing the test, you are classifying the genetic code of tumours that the GCNN has already seen. After consulting with more expert people in this field, I decided to create the modified code, with which the results generated are compared with the originals in the results section, and its use is explained in the work done section. Since usually, if there isn't a data shortage problem, it is not common to perform this practice. In addition, when the GCNN is trained, the hyperparameters are modified based on periodic validations, therefore, to ensure that the result is valid, it must be tested with genetic codes that the GCNN have never seen, to know if it can be adapted to it giving the right result.

5-fold cross validation

In the paper it is commented the use of 5-fold cross validation, and in the original code on line 75 of the PPI.py file, we can see the k fold cross validation preparation comment. After reviewing the code, the k-fold, or the five, has not been seen anywhere. Moreover, as I have commented previously, the authors gather all the data and the labels in their respective variables, and throw it into the library. Then, it is impossible for the k fold cross validation to be used. But, as I have a limited experience in this field, as I am just starting out, I decided to ask again for more experienced people. For this, I met with two students and a teacher from the master. The main conclusion drawn from these meetings has been that the GCNN is trained with a fixed train data set, several times and is always validated with the same validation set (which is also that of the final test). Therefore, there is any cross validation in the code.

Output data

Another important problem that I have found is that as the code was, probably due to a library problem, the knockdown and knock-up files only contained threes in all positions. To solve this, an output in CSV has been generated, parsing the result to strings, since this format is widely accepted by all programming languages and spreadsheets. First, I've used the Numpy library, obtaining the same bad result, a lot of threes, and the way that worked was saving it using the string type. Also, I had a similar problem with the confusion matrices, and I had to save in CSV and use Matplotlib to generate a confusion matrix in a PNG file to observe the results in a better way.

Post modelling analysis

In the post-modelling analysis, as seen above, there is an important error. The code provided does not follow the pseudo-code exposed in the paper. First of all, the bug that made the two calculations performed on the same value has been fixed. To do this, the zero on line 316 of the utils.py file has simply been changed to one. Moreover, the last part of the pseudocode isn't performed in the code.

In this part of the code, it has also been necessary to change the way files are saved, since it is faster and cleaner to process them directly in Python, than to transfer them to Matlab, just because this adds uncertainly to the result, because it can't be checked. Even more, when the code does not provide all the data that the pseudo-code exposed in the paper needs. The first step was to create the files with the output data in CSV. Once the operation has been tested, the code for the four types of graphs has been executed. Then,

reviewing the results, it is seen how from the modification in the 2200 gene approximately, the GCNN begins to return only zeros, or some twenty-eight. After observing that most of the results did not seem correct, the execution has been verified only in the last genes, obtaining consistent results in the terminal. After carrying out several code modifications and tests, the best solution found has been to reload the GCNN and save the data in blocks of 1000 in separate code executions, because stopping it by some minutes between blocks doesn't work, and the bad results become from the GCNN prediction. Due to the laborious nature of this task, the code has only been executed on the graphs of the two types with the singletons, which are the ones that give the best results, both for the original code, and for the GCNN created with the usual technique. Therefore, 8 executions have been made on the code with the PPI + singletons graph, and 5 executions for the co-expressions + singletons.

Another detail is that in the post-processing original code, 7100 genes are modelled (the loop with the hardcoded limit in the test function). This means that when the post-analysis is performed with the co-expression graph, the loop goes beyond the size of the array, generating a stack trace. To solve this, the loop has been referenced to the input data size. This has led us to see that 7448 genetic data inputs in the execution of the code with the protein-to-protein interaction graph with singletons. If we look at the Figure 43 extracted from the paper **[1]**, it can be seen that a total of 7091 genes are available as the authors say. Using the graph extracted from the code to perform the 3D rendering, using the Pajek format, it is contrasted that the graph actually contains 7448 nodes (genes), as can be seen in Figure 44, which is a capture of the file generated from the adjacency matrix used to create the protein-to-protein interaction graph.

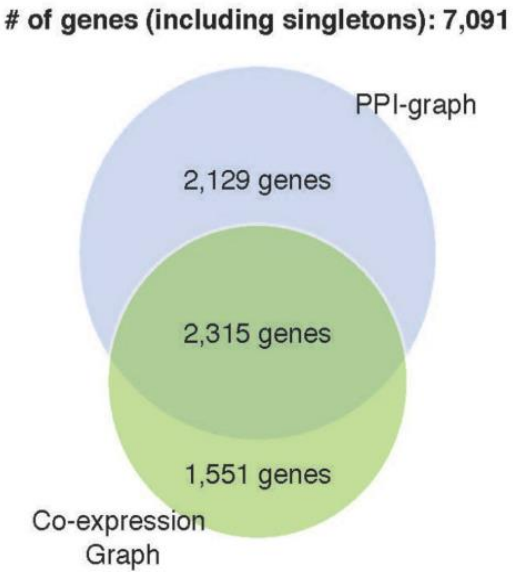

Figure 43 Paper's figure 4 genes for graph

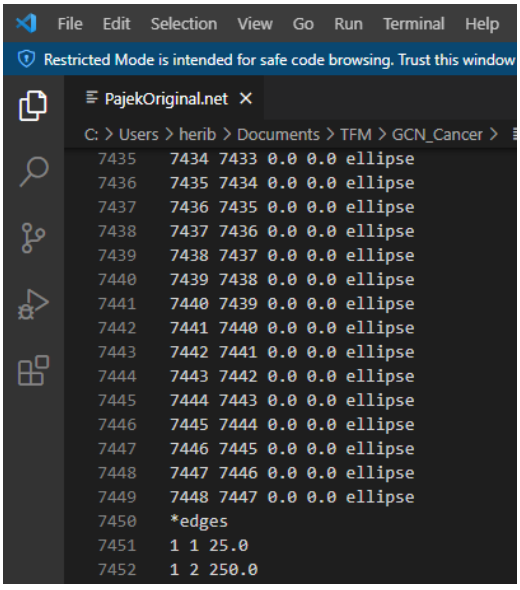

Figure 44 Pajek file from code graph in last nodes

Therefore, it is neither the 7100 that had been hardcoded, nor the 7448. After trying to find where this new number of genes come from, I have given up, since it does not appear anywhere, and it becomes directly from the input data loaded.

Post modelling results

Once we have the predictions for the graphs, it remains to obtain the results. As the code to compute the final result to contrast the conclusions, create the code is an option, but all the data required by the pseudocode isn't collected, and as the time necessary to modify all the code and libraries to be able to do it is so huge, I decided to look for a simpler approach, but one that can be valid. As seen in the results section, a plot was first generated in which the number of genes that affect each class of tumour (giving a wrong prediction without take into account any prediction probability) can be seen based on the test data. And with the same data, processed in another way, a plot that indicates how many times each tumour class is being affected for a gene modification.

Conclusions

As mentioned above, the code provided in the paper does not fully correspond to what is stated in the paper, since it is said that the pre-processing of the data is done in Matlab, so far correct. However, there is no trace of the code used in the post-modelling of the data, beyond the section where the value of the genes is modified. Therefore, it has only been possible to contrast the part of GCNNs accuracy results. Therefore, we can say that not all the master's thesis initial objectives have been met. Reviewing those that have been achieved, it has been explained how the GCNN of the various types exposed in the paper are created in an understandable way, how the GCNN work, what type of data is used in this study and the results have been verified. Also, the creation of the GCNNs is done, with values relatively close to those exposed in the paper.

Since the subject is interesting and new to me, in the course of all the work I did, I added some goals, since I felt that the work was very incomplete by not working nothing in the way it should. The first major goal, was to modify the data blocks with which the GCNN is trained to do it in a more realistic way, trying to achieve maximum accuracy and add a way to be able to compare the results with the original code ones, to be able to extract some conclusions. This goal is achieved, since it is admissible to have a slightly worse precision, but on data never seen by the GCNN. The second major objective has been to carry out an analysis of the resulting predictions. This has been the most laborious one, since I have encountered many problems through the code. I have not been able to replicate the analysis proposed in the paper, but I have made an approximation that gives an idea of the dependence of each GCNN on each gene position. However, I have not been able to say what genes exactly are, since the paper does not describe more than some considered as relevant, and I have not been able to acquire the input data from the sources. This is not conclusive, since many more tests should have to be done, but the idea presented in the paper it's a first approach as the authors says.

Having tried to overcome all these master's thesis problems, I can say that it has been very useful for my learning. Since the problems that I have encountered, have made me go further, in the understanding of genetic issues, as in the field of convolutional neural networks, their techniques and the most commonly used libraries.

Future work

As a first approximation we can see how the result of applying the GCNN to this field is satisfactory, but in the paper, at least from my point of view, there are too many flanks to deal with.

A first point to continue, is creating all the code in a unique coding language, Python, Matlab or any other. Using the latest libraries and giving a ready environment, to unequivocally demonstrate that it works. This code should compute from the first data processing, where the correlation between the genes is computed to create the GCNN structure, to the GCNN creation, train, prediction and post processing. Always using well known techniques, or otherwise, justifying the reason why a special method has been used, and show it that works correctly. In this way, more solid conclusions can be achieved.

From some of the tests carried out during the analysis of the code, there are other structures of the GCNNs that give good results, with different pooling levels, or with a smaller fully connected layer. I think it would be good to try to modify the structure of these to know their behaviour, since the one presented works well for PPI, but perhaps with some modification, the one of co-expressions are able to achieve better results. The paper does not comment much on the task carried out on the structure, only what it is. And all GCNNs are created following the same structure in the paper, with some changes in the hyperparameters.

Another idea is to carry out tests with different correlation coefficients in the interactions between genes for creating the GCNN structure, as the author proposes in the paper, since these values have been chosen for some reason, I understand that they choose empirically between some values. If so, it would have been a good idea to publish these results, or at least, explain why 0.8 and not 0.75, or any other number.

Another good path to continue with this research, would be the creation of a more tumour focused specialized code. The one presented to us works with tumorous tissues, for this a biopsy is required, which makes the GCNN ability to distinguish between a cervical tumour and a breast tumour less relevant, as the zones are quite far to confuse between them. One option that could have a future, would be to contact doctors who are specialized in this field, and since it has been shown that the results are better with a GCNN than with other approaches presented by other authors, focus the input data on specific tumour groups that can be confused due to their proximity. As the authors state in the paper, sometimes there are tumours that can be found very close together, they are very similar and their treatments are completely different. The objective would be to detect these tumours, and generate specialized GCNNs in them, which can give greater precision and be of real help in the diagnosis.

Another objective that can be proposed is using the code proposed in the second paragraph of this section, carried out in the same language and processing all the data, to carry out the same study that is presented in the paper's conclusions, to see which genes can be more relevant in each tumour. To do this, I would propose doing it on more than one approximation to be able to ensure that a gene is clearly relevant. Since only the one that has been done, if it is based on the code provided, raises some doubts. Moreover, if we compare the two GCNNs analysed in this document, we can see how each one relies on different genes. This could help the development of other future studies that will help

the cancer diagnosis research to go further, cause as more knowledge is achieved in the genetics field, there are other new techniques that are being developed can evolve in consequence.

Acknowledgments

I want to especially thank the director of this master's thesis, **Dr. Carme Julià Ferré**, for the effort she has made guiding me throughout all the work carried out. From the beginning, when it didn't seem as complicated as it turned out, to the helpful advices she gave me in the moment where the planned work got off track. I also want to thank the times that she has contacted third parties to consult some details, where some questions are not being excessively relevant, but helped me to improve the understanding of this master's thesis. To finish, I appreciate the quick responses to any problem and all revisions and meetings that we had, it helped me a lot. Without all these contributions, this master's thesis would be much simpler than it has ended up being.

I also want to thank Dr. Sergio Gómez Jiménez for the time he has dedicated to meet with me several times in this master's thesis course. His experience, and his comments about the theory and the techniques used have been very useful to me and have helped me obtain a better result in less time.

Finally, I want to acknowledge my co-workers in the ANA / CRAAX laboratory, Martí Miquel, Andrés Prieto and Ayaz Hussain, for the good times we have spent discussing the subject of this master's thesis, the techniques used and the reasons. These moments have helped me to think about some aspects of the thesis, and using with their comments, I have been able to explain some parts in a more understandable way.

Glossary

ACC:	Adrenocortical cancer
BLCA:	Bladder urothelial carcinoma
BRCA:	Breast invasive carcinoma
CESC:	Cervical and endocervical cancer
CHOL:	Cholangiocarcinoma
COAD:	Colon adenocarcinoma
COEX:	Co-expression
DLBC:	Diffuse large B-cell lymphoma
ESCA:	Esophageal carcinoma
GBM:	Glioblastoma multiforme
GCNN:	Graph convolutional neural network
HNSC:	Head and neck squamous cell carcinoma
KICH:	Kidney chromophobe
KIRC:	Kidney clear cell carcinoma
KIRP:	Kidney papillary cell carcinoma
LAML:	Acute myeloid leukaemia
LGG:	Low grade glioma
LIHC:	Liver hepatocellular carcinoma
LUAD:	Lung adenocarcinoma
LUSC:	Lung squamous cell carcinoma
MESO:	Mesothelioma
OV:	Ovarian serous cystadenocarcinoma
PAAD:	Pancreatic adenocarcinoma
PCPG:	Pheochromocytoma and paraganglioma
PPI:	Protein-protein interaction
PRAD:	Prostate adenocarcinoma
READ:	Rectum adenocarcinoma
SARC:	Sarcoma
SKCM:	Skin cutaneous melanoma
STAD:	Stomach adenocarcinoma
TCGA:	The Cancer Genome Atlas
TGCT:	Testicular germ cell tumor
THCA:	Thyroid carcinoma
THYM:	Thymoma
UCEC:	Uterine corpus endometrioid carcinoma
UCS:	Uterine carcinosarcoma
UVM:	Uveal melanoma

References

[1] R. Ramirez, Y.C. Chiu, A. Hererra, M. Mostavi, J. Ramirez, Y. Chen, Y. Huang, Y.F. Jin, Classification of Cancer Types Using Graph Convolutional Neural Networks, Front. Phys. 8 (2020) 1–14. https://doi.org/10.3389/fphy.2020.00203.

[2] Global Cancer Observatory, (n.d.). https://gco.iarc.fr/ (accessed 4 August 2021).

[3] I.B. Weinstein, K. Case, The History of Cancer Research: Introducing an AACR Centennial Series, Cancer Res. 68 (2008) 6861–6862. https://doi.org/10.1158/0008-5472.CAN-08-2827.

[4] S.I. Hajdu, A note from history: Landmarks in history of cancer, part 1, Cancer. 117 (2011) 1097–1102. https://doi.org/10.1002/CNCR.25553.

[5] Z. Li, W. Yang, S. Peng, F. Liu, A Survey of Convolutional Neural Networks: Analysis, Applications, and Prospects, (2020). https://doi.org/10.1109/tnnls.2021.3084827.

[6] A. Shamsaldin, P. Fattah, T. Rashid, N. Al-Salihi, A Study of The Convolutional Neural Networks Applications, UKH J. Sci. Eng. 3 (2019) 31–40. https://doi.org/10.25079/ukhjse.v3n2y2019.pp31-40.

[7] Y. Lecun, L. Eon Bottou, Y. Bengio, P.H. Abstract|, Gradient-Based Learning Applied to Document Recognition, n.d.

[8] Convolution - Artificial Inteligence, (n.d.). https://leonardoaraujosantos.gitbook.io/artificial-inteligence/machine_learning/deep_learning/convolution (accessed 9 August 2021).

[9] D.H. Hubel, T.N. Wiesel, Receptive fields, binocular interaction and functional architecture in the cat's visual cortex, J. Physiol. 160 (1962) 106–154. https://doi.org/10.1113/jphysiol.1962.sp006837.

[10] K. Fukushima, Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position, Biol. Cybern. 36 (1980) 193–202. https://doi.org/10.1007/BF00344251.

[11] W.S. Mcculloch, W. Pitts, A logical calculus nervous activity, Bull. Math. Biol. 52 (1990) 99–115.

[12] V. Crawford, Simulation Experiments, Essays Econ. Theory (Routledge Revivals). (2020) 42–43. https://doi.org/10.4324/9781315717791-15.

[13] Back propagation proposal paper, (n.d.). http://www.cs.utoronto.ca/~hinton/absps/naturebp.pdf (accessed 23 June 2021).

[14] Waivel, TDNN Waivel, (1987). https://isl.anthropomatik.kit.edu/pdf/Waibel1987a.pdf (accessed 23 June 2021).

[15] T.N. Kipf, M. Welling, Semi-supervised classification with graph convolutional networks, 5th Int. Conf. Learn. Represent. ICLR 2017 - Conf. Track Proc. (2017) 1–14.

[16] M. Defferrard, X. Bresson, P. Vandergheynst, Convolutional neural networks on graphs with fast localized spectral filtering, Adv. Neural Inf. Process. Syst. (2016) 3844–3852.

[17] Xavier Bresson, Xavier Bresson: 'Convolutional Neural Networks on Graphs' - YouTube, (n.d.). https://www.youtube.com/watch?v=v3jZRkvIOIM (accessed 19 May 2021).

[18] T. Kipf, Deep Learning with Graph-Structured Representations, n.d. https://pure.uva.nl/ws/files/46900201/Thesis.pdf (accessed 19 May 2021).

[19] C. Wang, J. Guo, N. Zhao, Y. Liu, X. Liu, G. Liu, M. Guo, A Cancer Survival Prediction Method Based on Graph Convolutional Network, IEEE Trans. Nanobioscience. 19 (2020) 117–126. https://doi.org/10.1109/TNB.2019.2936398.

[20] N. Cherian Kurian, A. Sethi, A. Reddy Konduru, A. Mahajan, S.U. Rane, A 2021 update on cancer image analytics with deep learning, WIREs Data Min. Knowl.

Discov. 11 (2021) 1–31. https://doi.org/10.1002/widm.1410.
[21] N. Malik, V.B. Bharat, S.P. Tiwari, J. Singla, Study of detection of various types of cancers by using deep learning: A survey, Int. J. Adv. Trends Comput. Sci. Eng. 8 (2019) 1228–1233. https://doi.org/10.30534/ijatcse/2019/31842019.
[22] T. Saba, Recent advancement in cancer detection using machine learning: Systematic survey of decades, comparisons and challenges, J. Infect. Public Health. 13 (2020) 1274–1289. https://doi.org/10.1016/j.jiph.2020.06.033.
[23] C. Li, X. Li, X. Li, M. Rahaman, X. Li, J. Wu, Y. Yao, M. Grzegorzek, A State-of-the-art Survey of Artificial Neural Networks for Whole-slide Image Analysis: from Popular Convolutional Neural Networks to Potential Visual Transformers, n.d.
[24] WHO, Cancer, (2021). https://www.who.int/news-room/fact-sheets/detail/cancer (accessed 23 June 2021).
[25] K. Pantel, C. Alix-Panabières, ScienceDirect Liquid biopsy : Potential and challenges 5, Mol. Oncol. 0 (2015) 371–373.
[26] N. Wan, D. Weinberg, T.Y. Liu, K. Niehaus, D. Delubac, A. Kannan, B. White, E.A. Ariazi, M. Bailey, M. Bertin, N. Boley, D. Bowen, J. Cregg, A.M. Drake, R. Ennis, S. Fransen, E. Gafni, L. Hansen, Y. Liu, G.L. Otte, J. Pecson, B. Rice, G.E. Sanderson, A. Sharma, J.S. John, C. Tang, A. Tzou, L. Young, G. Putcha, I.S. Haque, Machine learning enables detection of early-stage colorectal cancer by whole-genome sequencing of plasma cell-free DNA, BioRxiv. (2018) 1–10. https://doi.org/10.1101/478065.
[27] GitHub - mdeff/cnn_graph: Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering, (n.d.). https://github.com/mdeff/cnn_graph (accessed 19 May 2021).
[28] R. Ramirez, GitHub - RicardoRamirez2020/GCN_Cancer, (2020). https://github.com/RicardoRamirez2020/GCN_Cancer (accessed 24 June 2021).
[29] M. Mostavi, Y.C. Chiu, Y. Huang, Y. Chen, Convolutional neural network models for cancer type prediction based on gene expression, BMC Med. Genomics. 13 (2020) 1–13. https://doi.org/10.1186/s12920-020-0677-2.
[30] GitHub - chenlabgccri/CancerTypePrediction: This is the repository for paper titled as 'Convolutional neural network models for cancer type prediction based on gene expression'., (n.d.). https://github.com/chenlabgccri/CancerTypePrediction (accessed 19 May 2021).
[31] G. Karypis, V. Kumar, Metis: Unstructured Graph Partitioning and Sparse Matrix Ordering, Unstructured Graph Partinioning and Sparse Matrix Ordering. (1995) 1–16.
[32] The Cancer Genome Atlas Program - National Cancer Institute, (n.d.). https://www.cancer.gov/about-nci/organization/ccg/research/structural-genomics/tcga (accessed 19 May 2021).
[33] RPKM, FPKM and TPM, clearly explained | RNA-Seq Blog, (n.d.). https://www.rna-seqblog.com/rpkm-fpkm-and-tpm-clearly-explained/ (accessed 19 May 2021).
[34] K.R. Kukurba, S.B. Montgomery, Topic Introduction RNA Sequencing and Analysis, (2015). https://doi.org/10.1101/pdb.top084970.
[35] BioMart Central Portal: an open database network for the biological community - PubMed, (n.d.). https://pubmed.ncbi.nlm.nih.gov/21930507/ (accessed 17 June 2021).
[36] STRING: functional protein association networks, (n.d.). https://string-db.org/ (accessed 17 June 2021).
[37] From protein-protein interactions to protein co-expression networks: a new perspective to evaluate large-scale proteomic data, (n.d.). https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5359264/ (accessed 24 June 2021).
[38] D. Vella, I. Zoppis, G. Mauri, P. Mauri, D. Di Silvestre, From protein-protein interactions to protein co-expression networks: a new perspective to evaluate

large-scale proteomic data, Eurasip J. Bioinforma. Syst. Biol. 2017 (2017). https://doi.org/10.1186/s13637-017-0059-z.

[39] GCN Classification: Google Drive, (n.d.). https://drive.google.com/drive/folders/1_Cnvab7mIwCrNJyY-J4aR2ck9i72KH8t (accessed 29 July 2021).

[40] numpy.shape — NumPy v1.22.dev0 Manual, (n.d.). https://numpy.org/devdocs/reference/generated/numpy.shape.html (accessed 24 June 2021).

[41] numpy.array — NumPy v1.21 Manual, (n.d.). https://numpy.org/doc/stable/reference/generated/numpy.array.html (accessed 24 June 2021).

[42] numpy.reshape — NumPy v1.21 Manual, (n.d.). https://numpy.org/doc/stable/reference/generated/numpy.reshape.html (accessed 24 June 2021).

[43] numpy.ravel — NumPy v1.21 Manual, (n.d.). https://numpy.org/doc/stable/reference/generated/numpy.ravel.html (accessed 24 June 2021).

[44] GCN Classification: Google Drive, (n.d.). https://drive.google.com/drive/folders/1_Cnvab7mIwCrNJyY-J4aR2ck9i72KH8t (accessed 24 June 2021).

[45] numpy.transpose — NumPy v1.21 Manual, (n.d.). https://numpy.org/doc/stable/reference/generated/numpy.transpose.html (accessed 24 June 2021).

[46] numpy.hstack — NumPy v1.21 Manual, (n.d.). https://numpy.org/doc/stable/reference/generated/numpy.hstack.html (accessed 24 June 2021).

[47] numpy.vstack — NumPy v1.21 Manual, (n.d.). https://numpy.org/doc/stable/reference/generated/numpy.vstack.html (accessed 24 June 2021).

[48] tf.Graph | TensorFlow Core v2.5.0, (n.d.). https://www.tensorflow.org/api_docs/python/tf/Graph (accessed 26 June 2021).

[49] Introduction to graphs and tf.function | TensorFlow Core, (n.d.). https://www.tensorflow.org/guide/intro_to_graphs (accessed 26 June 2021).

[50] tf.compat.v1.placeholder | TensorFlow Core v2.5.0, (n.d.). https://www.tensorflow.org/api_docs/python/tf/compat/v1/placeholder (accessed 26 June 2021).

[51] T. Ahn, T. Goo, C.H. Lee, S. Kim, K. Han, S. Park, T. Park, Deep Learning-based Identification of Cancer or Normal Tissue using Gene Expression Data, Proc. - 2018 IEEE Int. Conf. Bioinforma. Biomed. BIBM 2018. (2019) 1748–1752. https://doi.org/10.1109/BIBM.2018.8621108.

[52] GitHub - decube83/GCN_Cancer, (n.d.). https://github.com/decube83/GCN_Cancer (accessed 29 July 2021).

[53] ncbi.nlm.nih.gov, (n.d.). https://www.ncbi.nlm.nih.gov/projects/gap/cgi-bin/GetPdf.cgi?document_name=GeneralAAInstructions.pdf (accessed 8 February 2021).