Universitat Autònoma
de Barcelona

Facultat de ciències

Treball de fi de grau

TFG2122_095 Adapting graph transformer embeddings to enhance small molecule bioactivity descriptors

Direcció:
Patrick Aloy Calaf
Tutorització:
Ricard Gelabert Peiri

Alumne:
Teodor Parella i Dilmé
NIU:
1491702

Juny 2022

Treball de fi de grau realitzat a l'Institut de Recerca Biomèdica de Barcelona i presentat a la Facultat de Ciències de la Universitat Autònoma de Barcelona per a l'obtenció del Grau en Química

Resum analític

Els descriptors moleculars juguen un paper fonamental en quimioinformàtica, ja que permeten retenir informació estructural i de bioactivitat de compostos químics. Els models d'aprenentatge profund poden ser entrenats sobre descriptors disponibles amb la finalitat de predir propietats en compostos desconeguts, sent doncs eines poderoses en el descobriment de fàrmacs. Recentment, models basats en xarxes neuronals de grafs han mostrat prediccions excel·lents mitjançant l'ús d'informació estructural en forma de grafs moleculars. En aquest treball presentem una col·lecció de models basats en incrustacions de transformador de grafs, fent possible prediccions a tots els nivells de bioquímica quan només es coneix l'estructura molecular d'un compost. Els models construïts estableixen una referència comparativa sobre metodologies prèvies basades en empremtes de connectivitat, permetent aplanar el camí cap a prediccions eficients de signatures per a molècules petites.

Resumen Analítico

Los descriptores moleculares juegan un papel fundamental en quimioinformática, ya que permiten retener información estructural y de bioactividad de compuestos químicos. Los modelos de aprendizaje profundo pueden ser entrenados sobre descriptores disponibles con la finalidad de predecir propiedades sobre compuestos desconocidos, siendo entonces herramientas poderosas para el descubrimiento de fármacos. Recientemente, modelos basados en redes neuronales de grafos han ofrecido predicciones excelentes mediante el uso de información estructural en forma de grafos moleculares. En este trabajo presentamos una colección de modelos basados en incrustaciones de transformador de grafo, permitiendo predicciones a todos los niveles de bioquímica cuando sólo se conoce la estructura molecular de un compuesto. Los modelos construidos establecen una referencia comparativa sobre metodologías anteriores basadas en huellas de conectividad, permitiendo allanar el camino hacia la predicción eficiente de signaturas en moléculas pequeñas.

Analytical abstract

Molecular descriptors play a fundamental role in cheminformatics, enabling relevant information encoding of a compound's structural and bioactivity features. Deep learning models can directly learn on a known dataset's descriptors to do further predictions on unknown data, hence serving as powerful theoretical tools for compounds' properties forecast and drug discovery. Recently, models based on graph neural networks have outperformed alternative methodology on molecular datasets by exploiting overall structural information in molecular graphs. Here, we present a collection of models based on graph transformer embeddings that enable predictions to all levels of biochemistry when just a compound's structure is known. Built graph-based models serve as a benchmark on previous extended connectivity fingerprints approaches, continuing to pave the way towards efficient chemistry-to-signature prediction on arbitrary small molecules.

Aknowledgements

I am grateful to you Patrick, for enabling me this outstanding opportunity in a research environment. This work would not have been possible without you Martino: thanks for all the knowledge and passion transmitted. Also, I would like to thank every single member of the structural biology and bioinformatics group. Thanks a lot Ricard, for the mentoring and supervision of the thesis.

Finally, I would like to thank Barcelona's institute for research in biomedicine for enabling me to use its facilities through the "a future in biomedicine 2021" call.

Teo Parella

Table of contents

List of abbreviations

BP	backpropagation.
CC	Chemical Checker.
CTS	chemistry-to-signature.
DL	deep learning.
ECFP-N	extended connectivity fingerprints up to N bonds.
ECFP4	extended connectivity fingerprints up to four bonds.
GIN	graph isomorphism network.
GNN	graph neural networks.
GROVER	graph representation from self-supervised message passing transformer.
Gtransformer	GNN transformer.
InChI	international chemical identifier.
LR	learning rate.
MF	morgan fingerprint.
MLP	multilayer perceptron.
MSE	mean squared error.
NN	neural network.
ReLU	rectified linear unit.
Sig0	signature 0.
Sig1	signature 1.
Sig2	signature 2.
Sig3	signature 3.
Sig4	signature 4.
SMILES	simplified molecular-input line-entry system.

1 Introduction

Molecular descriptors play a fundamental role in cheminformatics by enabling information storage of compounds' chemical and bioactivity properties. However, bioactivity information is not always available for any given compound. To overcome this drawback, machine and deep learning (DL) models can be trained efficiently on a known compounds' descriptors database to predict further properties from unseen molecular compounds. Such models provide useful insights into uncharacterized compounds and serve as powerful tools in drug discovery.

Here, we pay special attention to the Chemical Checker (CC), consisting of a database resource of structural, bioactivity, and pharmacological information in form of descriptors [1]. These result from encoding available information of major drug databases in a convenient computer-interpretable form for comparison and supervised/unsupervised DL tasks. Recently, a CC-based tool consisting of a siamese neural network was proposed to complete the missing information on the CC signatures [2]. Ultimately, general signature 3 (Sig3) descriptors provided complete information on the CC database, as these were obtained for any integrated compound.

Chemistry-to-signature (CTS) signaturizers were built to infer Sig3 information in all levels of biology from any small molecule structural information. Despite the predictability potential of CTS signaturizers, they infer Sig3 from simple extended connectivity fingerprints up to four bonds (ECFP4). Such easy-to-compute descriptors just encode structural information in form of simple substructures, missing rich stereochemical and extended structure features present in a compound's architecture. Therefore, ECFP4 descriptors are expected to hinder overall signaturizers potential, as they are an information passing bottleneck for further model training.

Lately, models based on graph neural networks (GNN) have proven enhanced predictability performance in cheminformatics by using graph encoding of overall molecular structures as input. In particular, a benchmarking GNN transformer (Gtransformer) called graph representation from self-supervised message passing transformer (GROVER), has outperformed other GNN-based models [3]. In the present work, we will attempt to enhance CTS signaturizers performance in each CC area by using GROVER, hopefully exploiting the maximum available information within a compound structure. We will adapt GROVER graph embeddings in a multitask to directly infer Sig3, hence providing a benchmark to previous CTS ECFP4-based signaturizers methodology.

2 Elemental theory on supervised learning

2.1 Chemical information

In cheminformatics, dealing with a molecule requires convenient information encoding in order to make it interpretable by a computer, ideally with the lowest memory usage and easiest manipulation. Although the International Union of Pure and Applied Chemistry compound nomenclature can be interpreted in a straightforward manner by a human, computers are not efficient at achieving performance with it and rely on better approaches for identifying compounds.

In the computational field, a widespread chemical structure identifier is the canonical simplified molecular-input line-entry system (SMILES), consisting of short variable-length character strings that specify compounds with no ambiguity. SMILES representation achieves this in a user-friendly and comprehensive way, but the correspondence is not injective, meaning that many SMILES may correspond to the same molecule. Other useful structural information encodings that solve the non-injective and fixed-length problems exist, such as the international chemical identifier (InChI) or the InChI key. In particular, while dealing with a molecular database, such structural identifiers are saved in a list and retrieved when properties and information want to be computed.

From the molecular structure identifier, the compound is recreated and information may be collected in form of molecular descriptors or fingerprints, key to be used in upcoming machine learning steps. Usually, they consist in a fixed length vector belonging to an n-dimensional chemical space $\mathbf{v} = (v_1, ..., v_i, ..., v_n) \in \mathbb{R}^{\mathrm{n}}$, where each variable contains specific information. In the one-hot encoding, each variable takes boolean values $v_i = 0, 1$, but continuous variable $v_i \in [-1, 1]$ following a distribution (usually bimodal) is also used in many descriptors to maximize encrypted information entropy.

Extended connectivity fingerprints up to N bonds (ECFP-N), also known as morgan fingerprint (MF), are popular descriptors thanks to circular atom neighborhoods information encoding and rapid computing time. They encrypt the presence/absence of specific connectivities and substructures within a molecular structure, up to a neighborhood of N bonds (usually $2-4$) [Fig.1]. Other sorts of molecular geometry-based descriptors enhancing ECFP-N performance are also available, such as the GNN-based and transformers-based descriptors [4].

Overall, chemical descriptors offer a rich and easy-to-manipulate information handling in computation for large compound databases. They are not only optimal for feeding an upcoming model, but also for performing a direct comparison between different compounds to search for similarities and clusterings. For instance, similarity can be related through inner product operations in the chemical space, such as euclidean distance, Tanimoto similarity or cosine similarity, among others.

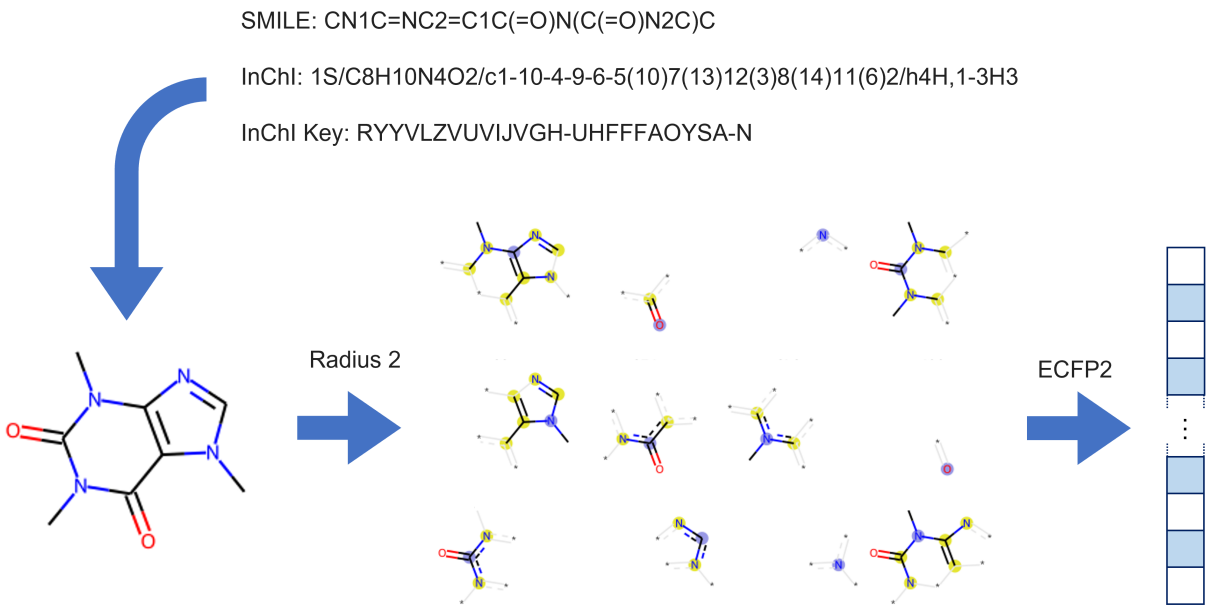

Figure 1: Retrieving process of caffeine ECFP4 descriptors. The molecule is rebuilt from stored canonical SMILES, InChI, or InChI key structural identifiers. Radius 2 substructures within the molecule are computed, and substructure presence/absence is encrypted in an ECFP4 vector, storing the molecular structural information.

2.2 Deep learning neural networks and backpropagation

A neural network (NN), also called multilayer perceptron (MLP), is a DL structure that enables a generic multivariable function $F(\mathbf{x}) = \mathbf{y}$ approximation through functional parametrisation. It is a powerful prediction tool, as it enables the identification of complex data relations in extensive databases that are not intuitive from a human perspective. In a broad sense, a NN learns features over a dataset and uses acquired knowledge to do further predictions on unseen data.

Artificial neurons/perceptrons are the building blocks of NNs, and consist of mathematical functions accomplishing the task of minimal information processing units. In analogy to real neurons, they take several input values/stimuli from other units, process the data, and emit an output accordingly. Overall, a NN has an architecture based on an input layer $\mathbf{x} = \mathbf{y}_0$, M hidden layers (L_i neurons at layer i), and an output layer $\mathbf{y}_{M+1}$, all based on neurons [Fig.2]. The output value y_i^k associated with the k^{th} neuron at layer i is computed from the values of the neurons in the previous layer. In particular, weights $w_{i-1,k'}^{i,k}$ are associated between the neuron k at layer i and connected neurons k' at layer $i-1$. Through an activation function σ and a bias u_i^k, y_i^k is generally computed by feeding forward:

$$y_i^k = \sigma\left(u_i^k + \sum_{k'=1}^{L_{(i-1)}} w_{i-1,k'}^{i,k} y_{i-1}^{k'}\right) \quad . \tag{1}$$

The activation function plays a protagonist hyper-parameter role in the NN performance. As hyper-parameter, we refer to a parameter used for controlling the training: results will most certainly show dependence on it, but it should be initially set and not trainable. Among the most popular activation functions, we find the rectified linear unit (ReLU) $\sigma_{ReLU}(x) = max(0, x)$, hyperbolic tangent, sigmoid $\sigma_{Sig.}(x) = (1 + e^{-x})^{-1}$ and Heaviside step $\sigma_{H.S.}(x) = \theta(x)$, where $\theta(x > 0) = 1$, $\theta(x \leq 0) = 0$.

For the training step, initialization weights and biases are usually set with uniform or Gaussian distributions. The network performs all intermediate calculations through convenient matrix operations, resulting in random outputs at first. Feeding an input dataset $\{\mathbf{x} = \mathbf{y}_0\}$ and desired target values $\{\mathbf{y}_{Tar.}\}$, NN predicted outputs $\{\mathbf{y}_{M+1}\}$ are compared with the targets through a loss function $\phi(\mathbf{y}_{M+1}, \mathbf{y}_{Tar.})$. The NN is then readjusted in a process called backpropagation (BP), consisting of a local minimum gradient descent optimization of the loss function in the weight space. Typical loss functions are the mean squared error (MSE) and the binary cross-entropy. To perform BP, the effect of the trainable parameters on the final loss function has to be found. This is achieved by computing the loss function total derivative with respect to the weights and biases. In particular, for $w^{i,k}_{i-1,k'}$, u^k_i:

$$\frac{d\phi}{dw^{i,k}_{i-1,k'}} = \frac{d\phi}{dy^k_i}\frac{dy^k_i}{dw^{i,k}_{i-1,k'}} \quad , \tag{2}$$

$$\frac{d\phi}{du^k_i} = \frac{d\phi}{dy^k_i}\frac{dy^k_i}{du^k_i} \quad . \tag{3}$$

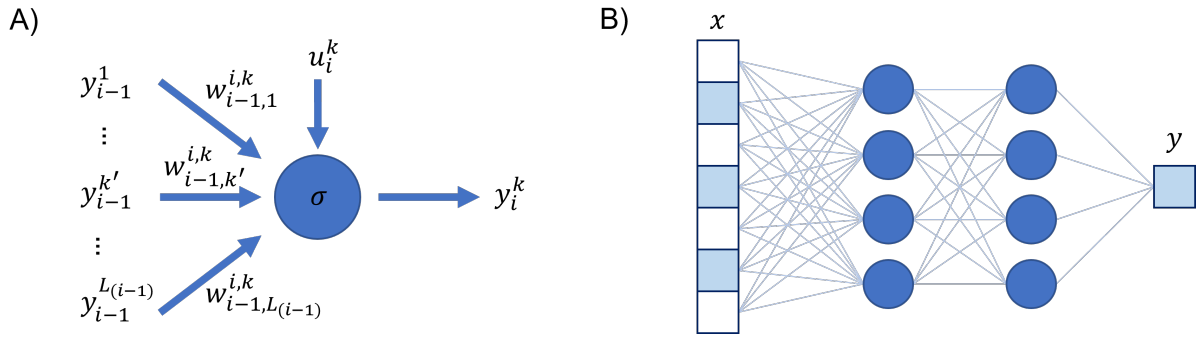

Figure 2: **(A)** Scheme of a k^{th} neuron/perceptron at layer i. It takes as input a bias u^k_i and previous layer $i-1$ neuron values and associated weights $\{y^{k'}_{i-1}, w^{i,k}_{i-1,k'}\}_{k'}$. The neuron operates on the values and applies an activation function σ to yield y^k_i as output. **(B)** Exemplification of a simple fully-connected NN architecture. It has a 7-dimension input $\mathbf{x}$ and a single-valued output $\mathbf{y}$. It performs the task with two hidden layers of $L_1 = L_2 = 4$ neurons each. Weights will be updated during the training to extract the complex relations between $\mathbf{x}$ values and output $\mathbf{y}$. This single task can be used for both classification or regression and can be easily adapted to multitasks if $\mathbf{y}$ has a different dimension.

Loss function derivatives are easily computed for all weights and biases using the chain rule iteratively on corresponding eq.(1), resulting in a simple matrix algebraic problem on the computer. Selection of the loss and activation functions hyperparameters will inherit particular function derivatives to be implemented. Eventually, a weight w^t and a bias u^t at iteration $t+1$ are updated as

$$w^{t+1} = w^t - \lambda \frac{d\phi^t}{dw^t} \quad , \tag{4}$$

$$u^{t+1} = u^t - \lambda \frac{d\phi^t}{du^t} \quad , \tag{5}$$

being λ the learning rate (LR) hyperparameter, a key ingredient in the gradient descent speed and performance. Departing from a consistent dataset, BP can shift the NN predictions to a generalised function $F(\mathbf{x}) = \mathbf{y}$ by minimising the loss function ϕ. In cheminformatics we can exploit the potential of the information encoded in a molecular descriptor as input $\mathbf{x} = \mathbf{y}_0$, to predict further complex molecular properties $\mathbf{y}_{M+1}$ as outputs of the NN.

2.3 Dataset handling: sets and cross-validation splits

In order to effectively train a NN, we must start from a consistent and large database containing the model input data $\mathbf{x}$ the and features to be predicted $\mathbf{y}$ for each element of the set. The overall dataset is split into two different subsets: the train set and the validation set. The former will contain the major part of the data (about 60-80%) and will be explicitly used during the NN training and BP. In particular, the NN trainable parameters will be updated after computing the averaged loss function over a specified batch size number of elements on the training set. In addition, the training process will take place for a specified number of epochs on the training set, meaning that the data will be used multiple times to train the NN. After each epoch, it will be important (as will be seen later on) to compute the model performance on the training set itself, as it may provide relevant information along the training process.

On the other hand, model performance will be evaluated on a validation set (consisting of the rest of the data) after each epoch to follow its evolution. Data included in it will not have been seen during the training nor perform BP while predicting its outputs. In this case, the average value of the loss function (and/or other metrics) will just be computed to assess the prediction efficiency of the NN after each epoch. As a clarification, sometimes a third test set may also be defined and used on a final evaluation after the overall model is trained. This is particularly interesting in some cases, as it might allow a fair comparison between models on the exact same data.

Depending on the data and parameters used during the training, the search for the local minima in the weight/bias space can lead to different results. Replications on the same model structure

but with different data selections will perform differently, so a cross-validation process is usually considered to hit the best-performing model. Cross-validation consists of multiple splits of the overall dataset in different train-validation subsets, each to be used on different replicates of the same model architecture. A common test set is usually defined in these cases, which ultimately enables a fair evaluation between the models on the exact same data. Eventually, after performing training and evaluation on all the splits, the model offering the best performance will be kept and used for further predictions on unseen data. On top of this, cross-validation can be an efficient method to deal with overfitting and selection bias events.

2.4 Selection bias and overfitting

Major drawbacks arising from model training that hinder predictability performance are selection bias and overfitting. The former consists of a predictability skew due to the training of a not-enough representative dataset. It may also appear when the training data is not randomly shuffled, opening the possibility of skew on the gradient descent process and complicating the achievement of the local minima target. Therefore, as the original data may be organized in some sort of way, it is important to perform a random shuffle at each epoch to avoid possible selection bias effects. On the other hand, overfitting is an issue more complicated to deal with, arising from a model over-training and usually being the limiting factor of DL performance. While training a dataset from scratch, the predictability of the model increases up to a point. If extensive training epochs are made, the model continues minimizing the loss function for the training set, learning it excellently. However, the predictability performance on the validation

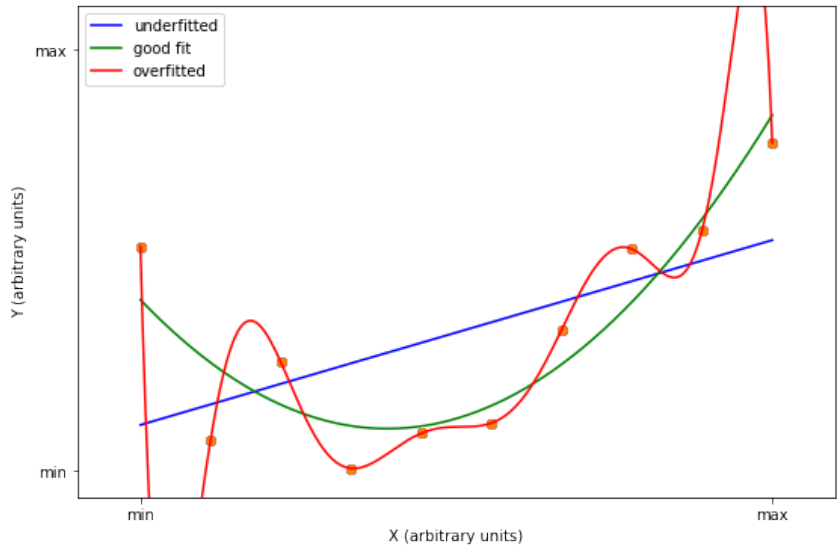

Figure 3: Comprehensive overfitting exemplification for 10 training points in a model with $dim(\mathbf{y}) = dim(\mathbf{x}) = 1$. An undertrained model will underfit the training data resulting in poor predictability and plenty of improvement margin. An overtrained model will adjust very well to the training data with a minimized loss function but will have reduced predictability.

set may begin to diminish, reducing generalized prediction effectivity and ruining the model's main purpose [Fig.3]. At this point where the model is way too complex and only performs well on the training data, we say that the model is overfitted. Overfitting effects are highly appreciable when dealing with small and hard-to-generalize datasets. Ultimately, while training a model we need to assess the performance on both train and test sets during the epochs, as we might appreciate how the model complexity evolves along with the training. If at some point we begin to experience overfitting effects, we might tune some hyperparameters to reduce overfitting effects or keep the model offering the best performance on the validation set.

3 State of the art and methodology

3.1 The Chemical Checker database and signaturization

The CC [1] is a resource based on a molecular database (currently around 1 million molecules) that processes their available bioactivity data in different areas of increasing complexity. Overall, information on 25 spaces (From A-E and 1-5) is compressed in convenient descriptors readily usable for comparison and DL. In particular, one can extract precomputed descriptors in the areas of chemistry (A), protein receptors and targets (B), biological pathways (C), cell-based assays(D) and clinical outcomes (E). Three different signatures may be retrieved for each space, corresponding to different sections of the CC data curation pipeline. signature 0 (Sig0) corresponds to raw data extracted from public databases, with varying dimensionality among compounds as feedback on an experimental feature might be given for one compound, but not for another. Although Sig0 information is explicit, it may not result convenient for DL as it is also sparse, heterogeneous, and unprocessed. Data is initially processed through principal component analysis and latent semantic analysis projections. These enable the extraction of the main Sig0 features and allow a first data curation step in the form of signature 1 (Sig1). Nevertheless, Sig1 still has a variable length, so a network embedding of k nearest neighbors clustering is performed. This results in the signature 2 (Sig2), an acceptably short fixed-length vector that may be used for DL. The further processes are performed on the data, the more convenient and noisy they are for usage. Also, not every signature in every space can be retrieved, as initial information in Sig0 is not always available.

On the other hand, Signaturizer tool allows extending predictions on CC information descriptors to any small molecule not included in the original database [2]. Available CC Sig2 descriptors are clustered together through a similarity score, making a large number of triplet groups consisting in an anchor, a positive similar molecule (from the cluster), and a negative molecule (from elsewhere in the chemical space) each. A siamese neural network predicts the output for the three triplet elements and performs BP with triplet loss as loss function, which maximizes the anchor's output similarity with the positive molecule and difference with the negative. Thanks to this process, predictions are made on missing CC data to achieve Sig3 fixed 128-D descriptors available for any CC molecule. Despite this being a slightly noisy prediction, Sig3 possess relevant and reliable molecular information ready to be used in further computational scenarios.

Sig3 is then used to train the so-called CTS Signaturizers, the key elements of this work. In them, Sig3 is considered as ground-truth targets $\mathbf{y}$, inferred from simple ECFP4 structural information input $\mathbf{x}$. Such an approach allows the prediction $\mathbf{y}_{pred.}$ of signature 4 (Sig4) for any small molecule included or not in the CC database. CTS Signaturizers are powerful models that enable information forecast to any level of chemistry given a compound structure.

3.2 Graph neural networks and transformer models

A graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$ is a mathematical structure consisting on a set of vertices $\mathcal{V} = \{\mathbf{v}_i\}$ and edges $\mathcal{E} = \{\mathbf{e}_{ij}\}$ connecting vertices i, j pairwise. In particular, undirected graph edges connect vertices in a symmetrical way, and all the graph elements contain some information features ($\mathbf{v}_i \in \mathbb{R}^{\mathrm{n}}$, $\mathbf{e}_{ij} \in \mathbb{R}^{\mathrm{m}}$). A molecule can be intuitively represented as a graph, with vertices and edges as nuclei and bonds respectively, each with local corresponding features [Fig.4A]. GNNs are adapted variations of NNs, which often result in enhanced performance by using graphs as inputs [5–8]. In a GNN, the molecular graph is subject to a process called message passing (or neighborhood aggregation), where contiguous vertices and edge features are updated sequentially as a function Φ, Φ' of the immediate neighbouring set $\mathcal{N}_i$ of bonds and features of nuclei i or bond ij:

$$\mathbf{v}_i^{t+1} = \Phi(\mathbf{v}_i^t, \{\mathbf{v}_j^t, \mathbf{e}_{ij}^t\} \in \mathcal{N}_i) \quad , \tag{6}$$

$$\mathbf{e}_{ij}^{t+1} = \Phi'(\mathbf{e}_{ij}^t, \mathbf{v}_i^t, \mathbf{v}_j^t, \{\mathbf{e}^t\} \in \mathcal{N}_i \cup \mathcal{N}_j) \quad . \tag{7}$$

Consequently, at each step/layer of the GNN each element learns from its surroundings, enabling it to reach k-hop neighbor information at layer k. All features are updated in a complex manner by learning information about neighboring elements and contextualizing their situation within the graph. After several GNN layers, resulting vertices/edges might be subject to a readout process for node/edge or overall graph tasks. Multitask graph regression is of special interest in this work, and is usually achieved by progressively coarsening the resulting graph by applying pooling operations [9–11]. Eventually, a desired embedded target vector $\mathbf{y}$ results as GNN output. This method enables an arbitrary-shaped graph (i.e. molecule) to be input in the GNN, resulting always in fixed multitask output $\mathbf{y}$.

In the present work, we are using GROVER, a state-of-the-art Gtransformer based model that has proven to outperform other modern approaches. Such Gtransformer enables to exploit further graph information through attention-based building blocks [12]. Initially, graph features are linearized and input to a dynamic message-passing network to extract matrices on queries $\mathbf{Q}$, keys $\mathbf{K}$ and values $\mathbf{V}$, which are fed into the transformer. The transformer block consists of attention operations using projection matrices $\{\mathbf{W}_i\}_i$, which are part of the trainable section of the Gtransformer:

$$Attention(\mathbf{Q}\mathbf{W}_i^{\mathbf{Q}}, \mathbf{K}\mathbf{W}_i^{\mathbf{K}}, \mathbf{V}\mathbf{W}_i^{\mathbf{V}}) = Softmax(\mathbf{Q}\mathbf{W}_i^{\mathbf{Q}}(\mathbf{K}\mathbf{W}_i^{\mathbf{K}})^T \frac{1}{\sqrt{d}})\mathbf{V}\mathbf{W}_{\mathbf{i}}^{\mathbf{V}} \tag{8}$$

Being d the dimension of $\mathbf{Q}$ and $\mathbf{K}$. Multiple attention layers outputs are concatenated and output as a multi-head attention operation:

$$Multihead(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = Concat.(\{Attention(\mathbf{Q}\mathbf{W}_i^{\mathbf{Q}}, \mathbf{K}\mathbf{W}_i^{\mathbf{K}}, \mathbf{V}\mathbf{W}_i^{\mathbf{V}})\}_i)\mathbf{W}^{Global} \tag{9}$$

The resulting output is aggregated and concatenated to initial graph pooled features and passed through a final MLP that enables the acquisition of the desired node, edge, or overall graph embeddings and classification/regression tasks. This GROVER Gtransformer has been recently proposed as a benchmark, outperforming other GNN-based methods. Alternative state-of-the-art methods implementing GNNs exist such as graph isomorphism network (GIN) [13].

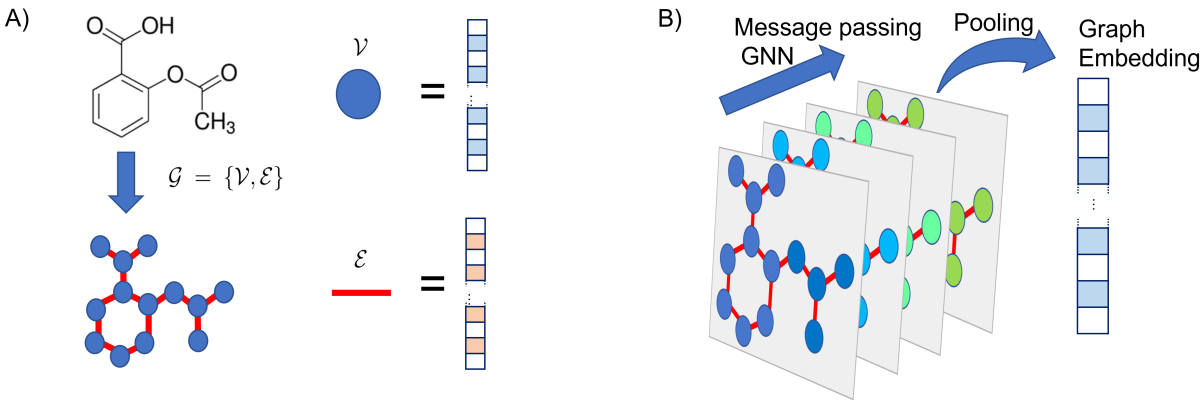

Figure 4: **(A)** Exemplification of aspirine in graph encoding. Atoms and bonds are represented as vertices $\mathcal{V}$ and edges $\mathcal{E}$ respectively, encoding own information in vector format. **(B)** Intuitive simplified overview of the mechanism used by a GNN. In the message passing, each element of the graph is updated acording to the surrounding information at each layer. Ultimately, a pooling operation can be applied to obtain fixed dimension embeddings for any graph.

4 Results and discussion

4.1 Database preparation

Current 1009212 available compounds in the CC database have been collected in form of SMILES, InChI and InChI keys in a CSV file. In order to perform scaffold splits, Murcko scaffolds representing core structures of compounds have been computed using RDKit [14, 15]. This step was not performed in the original signaturizers, but must be considered since stereoisomers (all with equal scaffolds) must be included either in the train or validation set. Otherwise, similar data would be present on both sets, and the validation process would be skewed due to partial similarity between them. Overall, 10 different splits have been performed with 80% of the molecules in the training set and 20% in the validation set (no test set), taking into account precomputed scaffolds [Fig.5]. Such splits have been computed using the group shuffle split function in scikit-learn [16]. In a simplified python version, the process would resemble the following:

```
import pandas as pd
import numpy as np
from rdkit.Chem.Scaffolds import MurckoScaffold
from sklearn.model_selection import GroupShuffleSplit

DataPath='./cc_universe.csv' #CC CSV file path.
data = pd.read_csv(DataPath,sep='\t') #SMILES, InChI and InChI keys entries
AllSmiles=data['SMILES'].tolist()
AllScaffolds=[MurckoScaffold.MurckoScaffoldSmiles(smi) for smi in AllSmiles]
data['MurckoScaffold']=AllScaffolds #Store scaffolds

gss = GroupShuffleSplit(n_splits=10, train_size=.8, random_state=1)
Train,Validation=[],[] #element i will hold train/validation indices of split i

for train_idx, test_idx in gss.split(np.ones(shape=(len(MurckoVec), 1)),np.ones
    (shape=(len(MurckoVec), 1)), AllScaffolds):
    Train.append(train_idx) #Keep train indices of all splits
    Validation.append(test_idx) #Keep validation indices of all splits

for split in range(len(Train)):
    Current_Split_Sets=[] #empty list to hold train/validation sets
    for i in range(len(Smilesvec)):
        if (i in Train[split]):
            Current_Split_Sets.append(1) #Include a 1 if in the train set
        elif (i in Validation[split]):
            Current_Split_Sets.append(0) #Include a 0 if in the validation set
    data['Split_{0}'.format{split}]=Current_Split_Sets

SavePath='./CrossValidations.csv'
data.to_csv(SavePath,sep='\t',encoding='utf-8',index=False)
```

	InChIKey	InChI	SMILES	MurckoScaff...	Split_0	Split_1	Split_2	Split_3	Split_4	Split_5	Split_6	Split_7	Split_8	Split_9
1	AAAAEENPAALFRN...	InChI=1S/C17H...	COc1cc(C(C)C)...	N=c1[nH]cc...	0.0	1.0	1.0	1.0	1.0	1.0	0.0	1.0	1.0	0.0
2	AAAAZQPHATYWO...	InChI=1S/C32H...	CCOc1cc2[nH]...	c1ccc2sc(C...	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.0	1.0
3	AAABHMIRDIOYOK...	InChI=1S/C18H...	Cn1cc([N+](=...	O=C(Nc1ccc...	0.0	1.0	1.0	1.0	1.0	1.0	0.0	1.0	1.0	1.0
4	AAABMNXUOFPYQ...	InChI=1S/C25H...	C=C1C[C@](O...	C=C1CCOC(...	1.0	0.0	1.0	1.0	1.0	0.0	1.0	0.0	1.0	1.0
5	AAABTPAECTZDET...	InChI=1S/C22H...	Cc1ccc(O)cc1N...	c1ccc(Nc2cc...	1.0	1.0	0.0	1.0	1.0	0.0	0.0	1.0	1.0	1.0
6	AAACBXVBBDAYR...	InChI=1S/C19H...	OC(=Nc1cccnc...	C(=Nc1cccn...	1.0	1.0	1.0	1.0	1.0	1.0	0.0	1.0	0.0	0.0
7	AAACGYYPWMUUF...	InChI=1S/C21H...	COC(=O)C[C@...	O=C(NCc1c...	0.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
8	AAADJKPSAGHJNF-...	InChI=1S/C28H...	COc1cc(/C=C/c...	C(=Cc1cc(C...	1.0	0.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
9	AAADKYXUTOBAG...	InChI=1S/C10H...	CC1(C)[C@@H...	S=C1CC2CC...	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.0	1.0	1.0
10	AAADMRIIAVPPNA-...	InChI=1S/C24H...	O=C(c1ccc(SC(...	O=C(c1cccc...	1.0	1.0	1.0	1.0	1.0	0.0	1.0	1.0	1.0	1.0
11	AAADPBLPXCELKR...	InChI=1S/C31H...	CN1CCN(Cc2cc...	C(=NCc1ccc...	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
...														
17	AAAFZMYJJHWUPN...	InChI=1S/C5H1...	O=P(O)(O)OC[...	C1CCOC1	1.0	1.0	0.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
18	AAAFZMYJJHWUPN...	InChI=1S/C5H1...	O=P(O)(O)OC[...	C1CCOC1	1.0	1.0	0.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
...														

Figure 5: Generated CSV file containing 10 scaffold-based splits for all the compounds in the CC database. Each molecule is included either in the train set (1) or the validation set (0) in each split. Equal scaffolds belong to the same sets in all splits (ex. molecule 17 and 18).

Corresponding Sig3 for all spaces (available for all molecules in the CC database), have also been retrieved from the CC library using a similar code to:

```
from chemicalchecker import ChemicalChecker
cc = ChemicalChecker()

data = pd.read_csv("./CrossValidations.csv", sep='\t')
InchiKeys=data['InChIKey'].values.tolist() #InChI keys are needed for Sig3

for i in ['A','B','C','D','E']:
    for j in range(1,6): #Iterate over all CC spaces
        s3 = cc.signature('{0}{1}.001'.format(i,j),'sign3') #Load database
        inks, signs = s3.get_vectors(InchiKeys) # Retrieve signatures 3
        np.save('./{0}{1}_sig3.npy'.format(i,j),signs)
```

ECFP4 signatures have also been computed using an equivalent code to:

```
from rdkit import Chem
import rdkit.DataStructs as DataStructs

data = pd.read_csv("~/CrossValidations.csv",sep='\t')
AllSmiles=data['SMILES'].tolist()
All_Morgans=[]

for i in range(len(AllSmiles)):
    current_mol=Chem.rdmolfiles.MolFromSmiles(AllSmiles[i]) #Build mol object
    current_bit=Chem.GetMorganFingerprintAsBitVect(current_mol,radius=2,nBits
    =2048) #Compute ECFP4 descriptors with 2048D
    arr = np.zeros((0,), dtype=np.int8) #target format
    DataStructs.ConvertToNumpyArray(current_bit, arr) #Copy ECFP4 in np format
    All_Morgans.append(arr) #Store current ECFP4

np.save('~/cc_morgans.npy', np.array(All_Morgans))
```

Ultimately, the 3 generated files encoding splits, Sig3 and ECFP4 have resulted in alphabetical InChI key order. Afterwards, data has been randomized with equal random seeds in all the files to prevent selection bias. For each descriptor, split, space, and set, an h5py file with its own architecture has been generated [17]. Each file encodes sub-datasets with batches of 1000 compounds. This structure will be convenient during the training process, as batches of molecules will be uploaded using the upcoming data generators in order to not overload memory.

To compute graphs, semantic motif label has been extracted with the GROVER provided file $scripts/save\_feature.py$, with $fgtasklabel$ as feature generator. Atom/bond contextual property has also been retrieved with $scripts/build\_vocab.py$. At this point graph featurization has been performed from SMILES using $scripts/save\_features.py$. Graph node encoding has included a total of 128 features, including information on the atom type, mass, charge, aromaticity, chirality, hybridization, number of bonds and surrounding hydrogen atoms. Edges have included 12 bond features, including type, conjugation, and cis/trans stereoisomery presence. Generation of the overall GROVER node embeddings has been performed in 1000 molecule batches, resulting in a total of 49 molecules in the CC database unable to be computed and whose batch has been killed. In such cases, the whole batch has been recomputed one molecule at a time, and molecules with incomplete graphs have been eliminated from all the datasets.

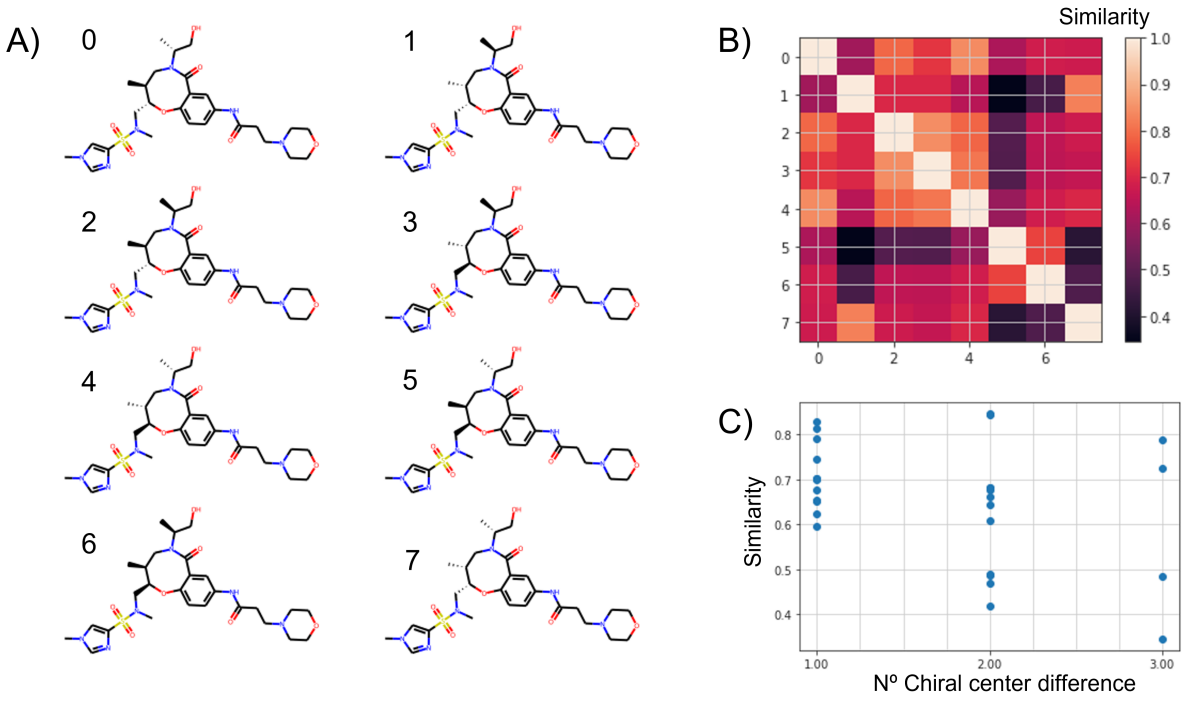

Figure 6: **(A)** The $2^3 = 8$ possible stereoisomers of a small molecule with 3 chiral centers. **(B)** Similarity between GROVER embeddings visualized as 1 - euclidean distance. **(C)** Decreasing similarity trend between graph embedings with increased chiral center difference number.

Molecular graphs' node embeddings have been computed with *main.py* script, using the built-in $predict$ function and the large GROVER model, all provided in Github repository. Overall, the graph transformer has provided 5000-dimension continuous variable embeddings for almost all CC molecules, which again have been stored in a new h5py file following the order and batches of the other descriptors. Despite ECFP4 being identical for stereoisomers, GROVER node embeddings prove to encode relevant stereochemical information that may be exploited to predict properties in pharmacological areas [Fig.6].

Although GROVER outperforms other GNN-based models, GIN graph embeddings have also been computed and stored in h5py files for comparison with the GROVER case.

4.2 Rebuilding ECFP4 signaturizers

CTS signaturizers using similar conditions on the original paper have been reconstructed using $tensorflow : KERAS$ [18]. The main reason for this decision is to do a fair comparison between the ECFP4 and Gtransformer-based signaturizers by considering the same splits. A NN with 2048-dimension input layer, 128-dimension output layer, and hidden layers of 1024, 512, 256 neurons, has been used in a regression task with ECFP4 descriptors as input and Sig3 as output. Set hyperparameters correspond to a dropout ratio of 0.1 (10% of the weights are kept), LR of 0.001 and ReLU activations to all layers with the exception of the last one, where $tanh$ has been used to allow Sig3 negative values inference. The $adam$ optimizer has been used with MSE as loss function and Pearson's correlation coefficient as an additional metric to track performance. NN architecture has been constructed with the following code:

```
from tensorflow import keras
from tensorflow.keras.layers import Dropout, Lambda, Dense, Activation
from tensorflow.keras import backend as K

model = keras.Sequential() #Build a Keras sequential model

model.add(layers.Dense(1024, input_dim=2048))
model.add(Dropout(0.1))
model.add(Activation('relu'))
model.add(Dense(512))
model.add(Dropout(0.1))
model.add(Activation('relu'))
model.add(Dense(256))
model.add(Dropout(0.1))
model.add(Activation('relu'))
model.add(Dense(128))
model.add(Activation('tanh')) #For inferring negative output values
model.add(Lambda(lambda x: K.l2_normalize(x, axis=-1))) #Normalization layer

model.compile(loss='mse', optimizer='adam', metrics=[corr]) #correlation metric
```

Since we are dealing with a huge dataset, injecting the whole data into memory would result in a memory error. In this case, we will use data generators, which will load data in 1000 compound batches, as already prepared in the h5py files. More specifically, the core structure of two example data generators retrieving split 0 ECFP4 and Sig3 of B1 space are:

```
import h5py
import random

Morgan_path='./Morgan/Split_0/' #Path to trainable split 0 ECFP4
Sig3_path='./Sig3/Split_0/B1/' #Path to trainable split 0 B1 space Sig3

def TrainGenerator_Morgan(int(seed)): #generator
    random.seed(seed)
    h5p=h5py.File('{0}/Train.h5'.format{Morgan_path},'r')
    Number_Batches=len(list(h5p.keys()))
    CurrentTrainOrder=random.sample([i for i in range(Number_Batches)],
    Number_Batches) #Random batch retrieval order
    h5p.close()
    for Batch in CurrentTrainOrder:
        h5f=h5py.File('{0}/Train.h5'.format(Morgan_path),'r')
        CurrentSigs=h5f[str(Batch)][:]
        h5f.close()
        yield np.array(CurrentSigs) #Returns a batch per generator iteration

def TrainGenerator_SIG3(seed):
    random.seed(seed)
    h5p=h5py.File('{0}/Train.h5'.format{Sig3_path},'r')
    Number_Batches=len(list(h5p.keys()))
    CurrentTrainOrder=random.sample([i for i in range(Number_Batches)],
    Number_Batches)
    h5p.close()

    for Batch in CurrentTrainOrder:
        h5f=h5py.File('{0}/Train.h5'.format{Sig3_path},'r')
        CurrentSigs=h5f[str(Batch)][:]
        h5f.close()
        yield np.array(CurrentSigs)
```

Overall, above generators allow DL implementation in the current dataset. On top of the mentioned generators, two additional ones retrieving ECFP4 and Sig3 on the validation set are recquired for the model validation. Epoch training and testing both on the training and validation data are implemented through the following functions:

```
def TrainEppoch(int(seed)):
    TRAIN_MORGAN=TrainGenerator_Morgan(seed) #Initialize generators
    TRAIN_SIG3=TrainGenerator_SIG3(seed)

    h5p=h5py.File('{0}/Train.h5'.format{Morgan_path},'r')
    Number_Batches=len(list(h5p.keys()))
    h5p.close() #Read total number of h5py load iterations to perform
```

```
    for i in range(Number_Batches):
        model.fit(next(TRAIN_MORGAN), next(TRAIN_SIG3), epochs=1, batch_size=200, verbose=2) #Fit data into the model to perform BP

def ValidateEppoch(which_epoch='validation'):
    if which_epoch == 'validation': #Validate performance on the validation set
        VALIDATE_MORGAN=ValidationGenerator_Morgan()#Initialize generators
        VALIDATE_SIG3=ValidationGenerator_SIG3()

        h5p=h5py.File('{0}/Test.h5'.format{Morgan_path},'r')
        Number_Batches=len(list(h5p.keys()))
        h5p.close()

    elif which_epoch == 'train': #Validate performance on the train set
        Tseed=random.randint(1, 100)
        VALIDATE_GROVER=TrainGenerator_Morgan(Tseed)
        VALIDATE_SIG3=TrainGenerator_SIG3(Tseed)

        h5p=h5py.File('{0}/Train.h5'.format{Morgan_path},'r')
        Number_Batches=len(list(h5p.keys()))
        h5p.close()

    CorrVec,LossVec=[],[]

    for i in range(Number_Batches):
        score = model.evaluate(next(VALIDATE_MORGAN), next(VALIDATE_SIG3), batch_size=1000, verbose=0) #Validate loss and correlation
        LossVec.append(score[0])
        CorrVec.append(score[1])

    return np.mean(LossVec),np.mean(CorrVec) #Return average
```

Finally, the following main function implements overall training:

```
def main():
    OutPath='./Models/Split_0/B1/' #Where model and scores will be saved
    epochs=50

    for i in range(epochs):
        TrainEppoch(i) #Execute train on a single epoch with generated function
        EppochLossValidationSet,EppochCorrValidationSet=ValidateEppoch(which_epoch='validation')
        EppochLossTrainSet,EppochCorrTrainSet=ValidateEppoch(which_epoch='train')
        scores=[EppochLossValidationSet,EppochCorrValidationSet,EppochLossTrainSet,EppochCorrTrainSet]
        np.save('{0}/Scores/Scores_Eppoch_{1}.npy'.format(OutPath,i),np.array(scores)) #Store MSE and correlation on both sets each epoch

    model.save('{0}Model_B1_Split0.h5'.format(OutPath)) #Save final model
```

For all 10 data splits, ECFP4-Sig3 CTS models have been built for each CC space and split (250 models in total). Each model has been run in a computer cluster of 1 GPU and 6 CPU nodes. Ultimately, the model at each space offering the best performance on the validation set at any epoch has been kept as the CTS model of that space [Fig.7]. Performance has been evaluated with Pearson's correlation, resulting in an average of $r^2 = (0.755 \pm 0.076)$ for the final selected signaturizers. Average correlation compares well to the one obtained in the original paper $r^2 = (0.769 \pm 0.074)$, considering that here we have performed scaffold splitting. Models seem to rapidly achieve maximum prediction performance at early epochs, which is in part expected when using one-hot encoding descriptors as more simple relations may be directly extracted. Afterward, we appreciate mild overfitting effects, slightly lowering the models' performance along the epochs.

4.3 Graph based chemistry-to-signature signaturizers

The previous process of ECFP4-Sig3 model construction has been repeated, but now using graph embeddings as model input instead. This process has been undertaken both with GROVER and GIN. In an analogous way, graph embeddings prepared in the data section have been retrieved with adapted data generators for training/validation. Model architecture has been escalated due to the increased complexity of the grover embeddings (bimodal continuous distributions). More specifically, two additional layers of 4000 and 2048 neurons have been included in front of the model architecture of the ECFP4 case. Initially, gradient descent initialization has been complicated due to the increased complexity of the model. To overcome this obstacle, batch size has been raised to 500 and batch normalization layers have been added between each neuron layer.

For each structural descriptor, a total of 25 final models have been retrieved after 10-fold cross-validation, one for each CC space [Fig.7]. Average maximum correlation achieved has resulted $r^2 = (0.761 \pm 0.076)$ for the GROVER selected models, and $r^2 = (0.614 \pm 0.083)$ for the GIN ones. We appreciate that GROVER embeddings slightly improve the predictability with respect to the ECFP4 used in this work. Still, it probably results slightly lower than the original signaturizers because no scaffold splitting was performed there. We see that GIN embedding-based models offer poor performance. This was initially expected because of the low dimensionality of these embeddings, hence capturing less information. Also, GROVER benchmark is proven to outperform GIN predictability in the original publication.

Overfitting effects have not been experienced in any of the two graph models. This is explained due to the large dataset used and the continuous variable encoding in graph embeddings. Models have been run up to 50 epochs, and in some GROVER cases further training periods suggest a slight increase in the validation predictability. Despite this, the models have taken a large amount of time to be computed, and slight predictability improvement does not suggest to significantly outperform ECFP4 models.

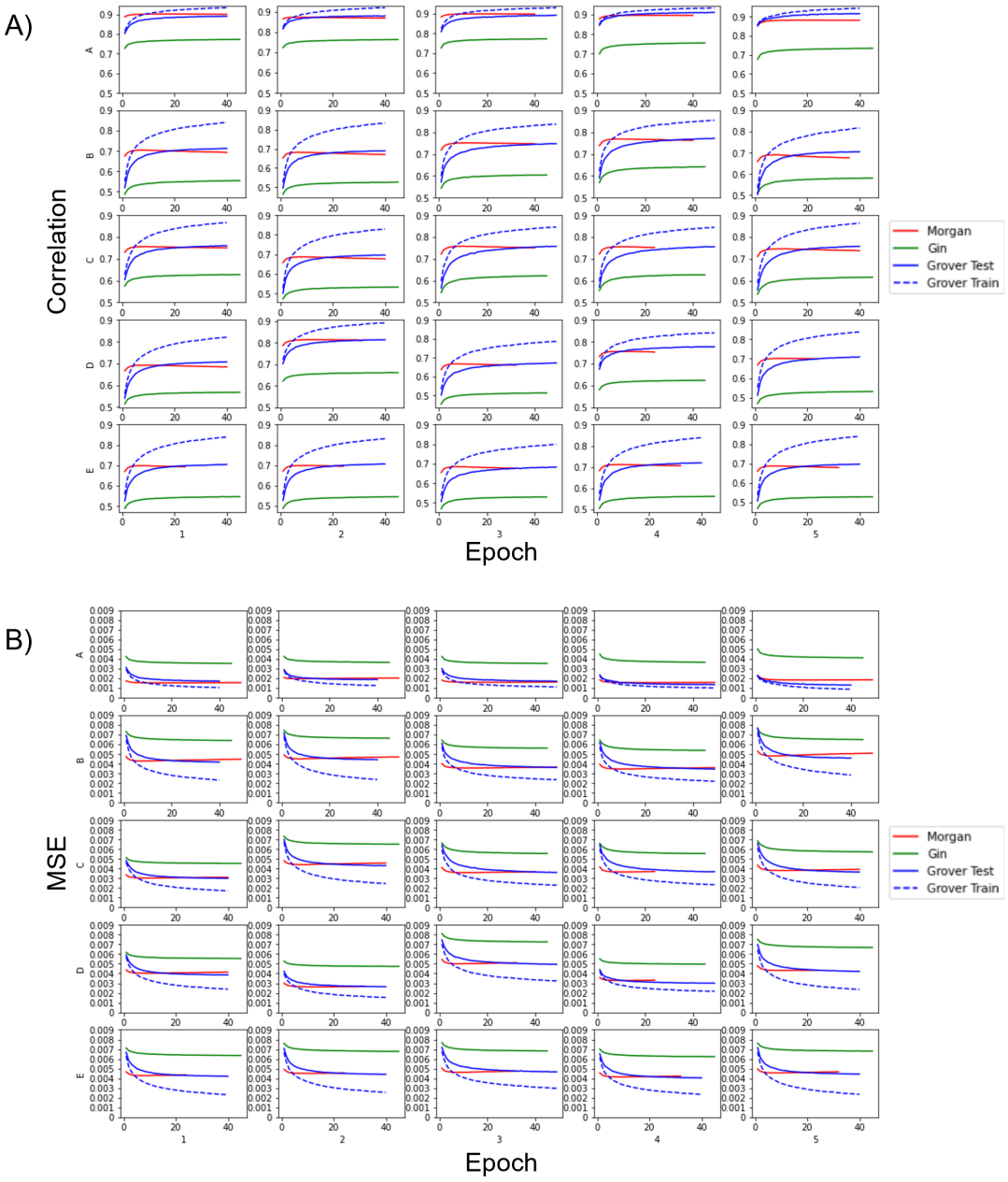

Figure 7: ECFP4 (MF), GROVER and GIN best models performance. Metric as function of the epochs for all the CC spaces. Train metrics on ECFP4 and GIN have not been included to not overcrowd the plots. **(A)** Average Pearson's correlation metric **(B)** average MSE loss funtion.

4.4 Enhancement with GROVER finetuning

Even though we have only obtained slightly enhanced performance on the graph embedding-based signaturizers, a finetuning task approach on the models provided by GROVER has been attempted. In this case, we have tried to directly influence the Gtransformer architecture by implementing a multitask to obtain Sig3 as the output of the Gtransformer (no intermediate graph embeddings). Despite the huge effort in making the finetuning task work, gradient descent has not been able to initialize on the training data. Due to the large amount of parameters to optimize in the Gtransformer, memory occupancy scales rapidly, limiting a batch size of approximately 20 molecules with the maximum allocation memory. Batch size included in the training has a strong effect in making the gradient descent initialization effective, but in this case the hyperparameter is restricted. Initial, maximum, and final LR have also been modified to try to overcome the problem. The finetuning task has not been able to be implemented in the end due to the limited duration of the project, but we encourage further work to focus on this finetuning direction. This is a command example to implement the finetuning task by executing the *./main.py* script.

```
import subprocess
import os

LoadModelPath='./grover_large.pt' #Model to be finetuned
SaveModelPath='./finetuned_B1.pt'
Main_path='./main.py' #Main file implementing finetunning
TrainDataPath='./data_B1.csv' #Structral indentifiers and multitask output
TrainFeaturesPath='./Features.npz' #Precomputed graph features

cmdTrain = "python {0} finetune --data_path {1} \
                        --features_path {2} \
                        --save_dir {3} \
                        --checkpoint_path {4} \
                        --dataset_type regression \
                        --split_type scaffold_balanced \
                        --ensemble_size 128 \
                        --num_folds 10 \
                        --no_features_scaling \
                        --activation tanh \
                        --split_sizes 0.8 0.15 0.05 \
                        --batch_size 20 \
                        --ffn_hidden_size 300 \
                        --epochs 50 \
                        --init_lr 0.000075 \
                        --max_lr 0.0005 \
                        --final_lr 0.000075 \
                        --metric r2".format(Main_path,TrainDataPath,
    TrainFeaturesPath, SaveModelPath, LoadModelPath) #Main command in CMD format

subprocess.Popen(cmdTrain, shell=True, env=os.environ).wait() #CMD execution
```

5 Conclusions and outlook

In this work, we have attempted to improve CTS signaturizers. Originally, these allowed CC Sig3 inference from simple ECFP4 structural descriptors, hindering relevant information passing in the models. In order to exploit the structural information of compounds, we have adapted graph transformer embeddings from GROVER and GIN. In particular, such graphs have served as input to trainable models, predicting Sig3 for all CC areas.

On the one hand, we have successfully rebuilt ECFP4 signaturizers to compare their performance with the upcoming graph embedding-based ones. 10-fold cross-validation on the CC data has been performed keeping the main architectural features. The best model for each space has been kept, resulting in a $r^2 = (0.755 \pm 0.076)$ correlation on the final models. Correlation has compared well to the original signaturizers, and we attribute a slightly lower correlation to the current ones due to the scaffold-based cross-validation splits. Such models have rapidly achieved maximum predictability performance in a few epochs, which can be attributed to the simple one-hot information encoding of ECFP4.

On the other hand, graph embedding models have been constructed in a similar manner as well. Complications in initializing gradient descent steps have been relevant in these models, up to a point where the batch size has been significantly increased and batch normalization layers have been included between neuron layers. After keeping the best models in a 10-fold cross-validation training, average correlation in all spaces has resulted in $r^2 = (0.761 \pm 0.076)$ for GROVER models and $r^2 = (0.614 \pm 0.083)$ for GIN models. As expected, GROVER embeddings have outperformed GIN ones as stated in the original GROVER publication. No overfitting effects have been appreciated, most probably because of the large dataset and complex continuous-variable data in the initial embeddings.

An attempt to directly finetune Gtransformer architecture has resulted in failure due to its complexity and the current project's scope. Gradient descent has not been able to initialize properly, as GPU cluster memory limit has been achieved with a maximum batch size of approximately 20 molecules. Modifying the LR has been attempted as well. Batch size has not been able to be increased higher than 20 and no batch normalization layers have been able to be implemented within the transformer architecture during finetuning.

We strongly encourage future studies to try to successfully implement direct finetuning of GROVER Gtransformer. Despite this being a more elaborate procedure, impacting directly on the Gtransformer would probably result in enhanced predictability performance. Ultimately, a pipeline to retrieve Sig3 prediction from SMILES could be implemented by automatizing graph features generation, model import, and signatures prediction.

6 Bibliography

(1) Duran-Frigola, M.; Pauls, E.; Guitart-Pla, O.; Bertoni, M.; Alcalde, V.; Amat, D.; Juan-Blanco, T.; Aloy, P. *Nature Biotechnology* **2020**, *38*, 1087–1096.

(2) Bertoni, M.; Duran-Frigola, M.; Badia-i-Mompel, P.; Pauls, E.; Orozco-Ruiz, M.; Guitart-Pla, O.; Alcalde, V.; Diaz, V. M.; Berenguer-Llergo, A.; Brun-Heath, I. e. a. *Nature Communications* **2021**, *12*, DOI: 10.1038/s41467-021-24150-4.

(3) Rong, Y.; Bian, Y.; Xu, T.; Xie, W.; Wei, Y.; Huang, W.; Huang, J. *Advances in Neural Information Processing Systems* **2020**, *33*.

(4) Atz, K.; Grisoni, F.; Schneider, G. *Nature Machine Intelligence* **2021**, *3*, 1023–1032.

(5) Sanchez-Lengeling, B.; Reif, E.; Pearce, A.; Wiltschko, A. B. *Distill* **2021**, https://distill.pub/2021/gnn-intro, DOI: 10.23915/distill.00033.

(6) Daigavane, A.; Ravindran, B.; Aggarwal, G. *Distill* **2021**, https://distill.pub/2021/understanding-gnns, DOI: 10.23915/distill.00032.

(7) Wu, Z.; Pan, S.; Chen, F.; Long, G.; Zhang, C.; Yu, P. S. *IEEE Transactions on Neural Networks and Learning Systems* **2021**, *32*, 4–24.

(8) Zhou, J.; Cui, G.; Zhang, Z.; Yang, C.; Liu, Z.; Sun, M. Graph Neural Networks: A Review of Methods and Applications, cite arxiv:1812.08434, 2018.

(9) Zhang, M.; Cui, Z.; Neumann, M.; Chen, Y. *Proceedings of the AAAI Conference on Artificial Intelligence* **2018**, *32*.

(10) Ying, R.; You, J.; Morris, C.; Ren, X.; Hamilton, W. L.; Leskovec, J. Hierarchical Graph Representation Learning with Differentiable Pooling, 2018.

(11) Lee, J.; Lee, I.; Kang, J. In *Proceedings of the 36th International Conference on Machine Learning*, ed. by Chaudhuri, K.; Salakhutdinov, R., PMLR: 2019; Vol. 97, pp 3734–3743.

(12) Veličković, P.; Cucurull, G.; Casanova, A.; Romero, A.; Liò, P.; Bengio, Y. Graph Attention Networks, 2017.

(13) Xu, K.; Hu, W.; Leskovec, J.; Jegelka, S. In *International Conference on Learning Representations*, 2019.

(14) Bemis, G. W.; Murcko, M. A. *Journal of Medicinal Chemistry* **1996**, *39*, 2887–2893.

(15) RDKit: Open-source cheminformatics, http://www.rdkit.org, [Online; accessed 11-April-2013].

(16) Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; Vanderplas, J.; Passos, A.; Cournapeau, D.; Brucher, M.; Perrot, M.; Duchesnay, E. *Journal of Machine Learning Research* **2011**, *12*, 2825–2830.

(17) Collette, A., *Python and HDF5*; O'Reilly: 2013.

(18) Chollet, F. et al. Keras https://github.com/fchollet/keras.