Arnau Dastis Fonoll

Drug potency prediction of SARS-Cov2-Mpro based on Graph Convolutional Network

TREBALL DE FI DE GRAU

dirigit per Dr. Francesc Serratosa

Grau d'Enginyeria Informàtica

UNIVERSITAT ROVIRA I VIRGILI
Tarragona

2024

Contents

Abstract

This thesis investigates the prediction of drug potency for the SARS-Cov2 main protease (Mpro) using Graph Convolutional Networks (GCNs). The study is conducted in two phases. Initially, results from the QM7 dataset, which contains molecular structures, are replicated using six neural network models: Graph Attention Networks (GAT), Graph Convolutional Networks (GCN), Graph Isomorphism Networks (GIN), Graph Transformers (GT), Gated Graph Convolutional Networks (GatedGCN), and Message Passing Neural Networks (MPNN). These models, part of the broader class of Graph Neural Networks (GNNs), are chosen due to their ability to handle graph-structured data and capture the complex relationships between molecular components.

In the second phase, the study applies these GNN models to a novel dataset from the Universitat Rovira i Virgili (URV) to predict the binding affinity between drugs and the SARS-Cov2 Mpro enzyme. The goal is to develop a reliable prediction tool that can assist in identifying potential antiviral drugs for COVID-19.

Key findings reveal that while most models performed well on the QM7 dataset, the replication of results for certain models, such as GT and MPNN, was hindered by computational limitations. Furthermore, when applied to the URV dataset, the GNN models exhibited varying levels of accuracy, likely due to the small dataset size (344 samples) and its unique molecular features.

The thesis concludes with suggestions for future research, including optimizing model architectures and exploring the applicability of GNNs in other fields such as bioinformatics and social network analysis. It emphasizes the potential of GNNs in real-world applications, particularly in drug discovery and computational biology.

Keywords: Graph Neural Networks (GNN), Drug potency prediction, SARS-CoV2 main protease (Mpro), Molecular structures, COVID-19, and Binding affinity prediction.

Resum

Aquesta tesi investiga la predicció de la potència de fàrmacs per a la proteasa principal del SARS-CoV2 (Mpro) mitjançant Xarxes Neuronals Convolucionals de Grafs (GCNs). L'estudi es divideix en dues fases. En la primera, es repliquen els resultats del conjunt de dades QM7, que conté estructures moleculars, utilitzant sis models de xarxes neuronals: Xarxes d'Atenció de Grafs (GAT), Xarxes Neuronals Convolucionals de Grafs (GCN), Xarxes d'Isomorfisme de Grafs (GIN), Transformadors de Grafs (GT), Xarxes Convolucionals de Grafs Gated (GatedGCN) i Xarxes Neuronals de Passatge de Missatges (MPNN). Aquests models són part de les Xarxes Neuronals de Grafs (GNNs), triades per la seva capacitat de gestionar dades estructurades i capturar relacions complexes entre components moleculars.

En la segona fase, s'apliquen aquests models de GNN a un conjunt de dades de la Universitat Rovira i Virgili (URV) per predir l'afinitat d'unió entre fàrmacs i l'enzim Mpro del SARS-CoV2, amb l'objectiu de desenvolupar una eina de predicció que faciliti la identificació de possibles fàrmacs antivirals per a la COVID-19.

Els resultats mostren que, tot i que la majoria dels models van rendir bé amb el conjunt de dades QM7, la replicació de certs models com GT i MPNN es va veure limitada per la potència computacional. Quan es van aplicar al conjunt de dades URV, els models van mostrar una precisió variable, degut en part a la mida reduïda del conjunt (344 mostres) i les seves característiques úniques.

La tesi conclou amb propostes per a futures investigacions, com l'optimització de models i l'aplicació de GNNs en altres camps, com la bioinformàtica.

Paraules clau: Xarxes Neuronals de Grafs (GNN), Predicció de la potència de fàrmacs, Proteasa principal del SARS-CoV2 (Mpro), Estructures moleculars, COVID-19 i Binding affinity prediction.

Resumen

Esta tesis investiga la predicción de la potencia de fármacos para la proteasa principal del SARS-CoV2 (Mpro) mediante Redes Neuronales Convolucionales de Grafos (GCNs). El estudio se divide en dos fases. En la primera, se replican los resultados del conjunto de datos QM7, que contiene estructuras moleculares, utilizando seis modelos de redes neuronales: Redes de Atención de Grafos (GAT), Redes Neuronales Convolucionales de Grafos (GCN), Redes de Isomorfismo de Grafos (GIN), Transformadores de Grafos (GT), Redes Convolucionales de Grafos Gated (GatedGCN) y Redes Neuronales de Paso de Mensajes (MPNN). Estos modelos son parte de las Redes Neuronales de Grafos (GNNs), elegidos por su capacidad para manejar datos estructurados y capturar relaciones complejas entre componentes moleculares.

En la segunda fase, se aplican estos modelos de GNN a un conjunto de datos de la Universitat Rovira i Virgili (URV) para predecir la afinidad de unión entre fármacos y la enzima Mpro del SARS-CoV2, con el objetivo de desarrollar una herramienta de predicción que facilite la identificación de posibles fármacos antivirales para el COVID-19.

Los resultados muestran que, aunque la mayoría de los modelos funcionaron bien con el conjunto de datos QM7, la replicación de ciertos modelos como GT y MPNN se vio limitada por la capacidad computacional. Al aplicarse al conjunto de datos URV, los modelos mostraron una precisión variable, debido en parte al reducido tamaño del conjunto (344 muestras) y sus características únicas.

La tesis concluye con propuestas para futuras investigaciones, como la optimización de los modelos y la aplicación de las GNN en otros campos, como la bioinformática.

Palabras clave: Redes Neuronales de Grafos (GNN), Predicción de la potencia de fármacos, Proteasa principal del SARS-CoV2 (Mpro), Estructuras moleculares, COVID-19 y Binding affinity prediction.

1. Introduction

In recent years, the integration of neural networks into diverse fields has had a transformative impact, particularly due to their ability to model complex relationships within large datasets. As the demand for efficient processing of structured data has increased, graph-based neural networks (GNNs) have emerged as a crucial tool, offering advanced methodologies for the analysis and interpretation of data across a wide range of disciplines, including social network analysis and molecular chemistry.

This research investigates the performance of several advanced GNN models, such as Graph Attention Networks (GAT), Graph Convolutional Networks (GCN), and Graph Transformers, on both established benchmark datasets and novel applications. The selection of these models is based on their demonstrated ability to capture intricate patterns within graph-structured data, rendering them essential for contemporary computational challenges.

The theoretical basis for this study is grounded in the contributions of leading scholars in machine learning and artificial intelligence. Of particular relevance is the work of Veličković et al. (2018), whose pioneering research in the development of Graph Attention Networks has profoundly influenced the trajectory of this study. Their research highlights the significance of attention mechanisms in improving both the interpretability and performance of neural networks when applied to graph data, a principle that is central to the models analyzed in this investigation.

Ultimately, this study seeks to advance the understanding and application of graph-based neural networks, emphasizing their potential to address complex, real-world challenges. By providing a comprehensive analysis of various models, this work aims to contribute meaningfully to both the academic discourse and broader societal objectives, illustrating how innovative AI techniques can play a pivotal role in fostering sustainable development and addressing pressing global issues.

2. Theoretical Framework

The theoretical framework of this project focuses on exploring the fundamental concepts and models that underpin neural networks and graph-based neural networks. Neural networks, inspired by the structure and function of the human brain, have become a cornerstone of

modern machine learning due to their remarkable ability to process and learn from vast amounts of data. This section begins with an overview of neural networks, detailing the primary architectures.

Moving beyond traditional neural networks, the framework also delves into graph theory and its application within the realm of neural networks. Graphs, consisting of nodes and edges, provide a powerful mathematical structure for modeling complex relationships between entities, making them essential in domains where data is inherently structured, such as social networks, biological networks, and molecular structures. The discussion progresses to graph-based neural networks, which extend the capabilities of conventional neural networks by leveraging graph structures to capture intricate dependencies and interactions within the data. In this context, various advanced graph neural network (GNN) models are introduced, including Graph Attention Networks (GAT), Graph Convolutional Networks (GCN), Graph Isomorphism Networks (GIN), Graph Transformers (GT), Gated Graph Convolutional Networks(GatedGCN) and Message Passing Neural Networks(MPNN). These models represent the cutting-edge of research in the field, each offering unique mechanisms to enhance the learning process on graph-structured data.

Through this comprehensive overview, the theoretical framework provides a solid foundation for understanding the models and techniques applied in this project, laying the groundwork for the subsequent experimental work and analysis.

2.1. Basic Structure of Neural Networks

The fundamental building block for Neural Networks (NN) is neurons. Faneli et al. (2024) explain in their work that computational units inspired by biological neurons can receive one or more data inputs (typically weighted to represent their importance) then the neuron performs a weighted sum often with a bias term. After that it applies a nonlinear activation function to the value generating the output. The activation function introduces nonlinearity into the network, enabling it to learn and model complex patterns in the data. Common activation functions include the sigmoid, ReLU (Rectified Linear Unit), and tanh functions, each of which transforms the input in a different way.

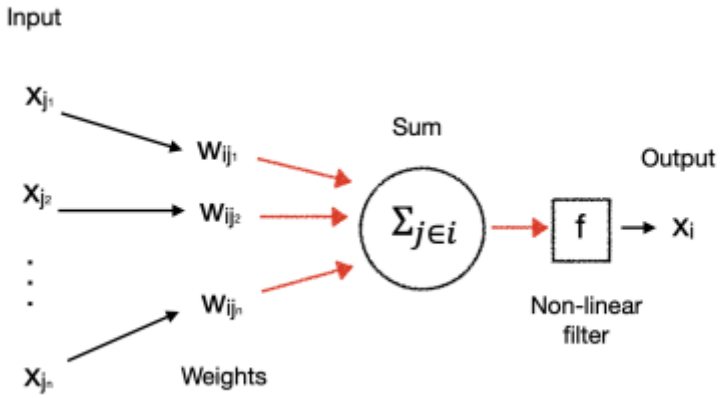


Figure 1. Schematic representation of a neuron from Faneli et al. (2024) showing inputs, weights, function, filter, and output

These neurons can be arranged to form NN, to do so we distinguish three different parts. An input layer responsible for receiving raw data the size of which is determined by the dimension of the data, this layer generally mirrors the entry information to be subsequently processed since it uses a linear function. Following that a sequence of hidden layers of variable size which are responsible for performing transformations of the data applying nonlinear activation functions, these nonlinear transformations enable the network to capture complex patterns and relationships within the data and finally, an output layer responsible for generating the predictions.

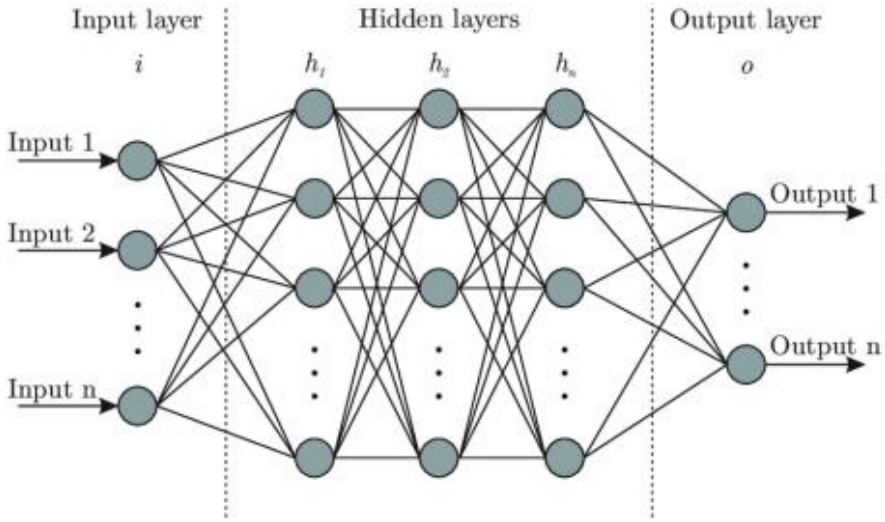


Figure 2. Schematic representation of a NN from Faneli et al. (2024)

To train the network we must change the weights in a way where for a given input it will generate an output with the minimal possible error. To do so usually we follow two main steps:

Forward propagation: in this step we generate an output after passing input data through the network and then we compare it to the expected output. The difference between those is measured usually as mean squared error (MAE).

Backpropagation: In this step the gradient of loss function is calculated with respect to the weights, layer by layer starting on the output and ending on the input adjusting the weight all along, usually techniques like stochastic gradient descent or Adam optimizer are used to do so.

2.2 Graph Neural Networks (GNNs)

Graph Neural Networks (GNNs) are a special kind of NN that were designed to be able to process data in graph form.

2.2.1. Graph Structure and Representation

Graphs are composed by vertices/nodes and edges. Nodes represent the entities being modelled and edges represent the connections between those. Each node can have associated features, and each edge can also. The main goal of the GNN is to learn the state ending hv, this state encodes the information of the neighbouring nodes and then used to produce an output ov

Liu and Zhou (2022) cite Scarselli et al. (2009) in their work to explain that GNNs are designed to learn representations of the nodes, edges, or the entire graph by aggregating information from a node's local neighbourhood (the closest nodes).

2.2.2. Model

The main goal of the GNN is to learn the state ending hv of each node, this state encodes the information of the neighbouring nodes and then used to produce an output ov. To calculate hv a parametric function called local transition function f is used. For the output there is another parametric function the local output function g:

$$\boldsymbol{h_v = f\left(x_v, x_{co[v]}, h_{ne[v],} x_{ne[v]}\right)} \quad (\mathbf{1})$$

$$\boldsymbol{o_v = g\left(h_{v,} x_v\right)} \qquad (\mathbf{2})$$

Here, x represents the input feature, and h represents the hidden state. The set of edges connected to node v is denoted as co[v], while ne[v] represents the set of neighbouring nodes

of v. Therefore, xv, xco[v], hne[v], and xne[v] correspond to the features of node v, the features of its edges, the states of its neighbouring nodes, and the features of those neighbouring nodes, respectively.

Let H, O, X, and XN be matrices formed by stacking all the states, outputs, features, and node features, respectively. These matrices can be expressed in a compact form as:

$$H = F(H, X) \quad (3)$$

$$O = G(H, X_N) \quad (4)$$

Here, F is the global transition function and G is the global output function. Both F and G are the stacked versions of the local transition function f and the local output function g for all nodes in the graph. The value of H is determined as the fixed point of equation (3), and it is uniquely defined under the assumption that F is a contraction map.

According to Banach's fixed point theorem, as referenced by Khamsi and Kirk (2011), the GNN uses the following iterative scheme to compute the node states:

$$H_{t+1} = F(H_t, X) \quad (5)$$

where Ht represents the state matrix at the t iteration. The dynamic system described by equation (5) converges exponentially fast to the solution of equation (3), regardless of the initial value H(0). It is also worth noting that the operations defined by f and g can be understood as feedforward neural networks (FNNs).

The learning algorithm follows a gradient descent strategy in other to learn the parameters of f and g and consists of the following steps:

The node states htv are iteratively updated using Equation (1) until reaching a specific time step T, resulting in an approximate fixed-point solution of Equation (3): H(T)≈H

The gradient of the weights W is computed based on the loss function.

The weights W are then updated according to the computed gradient.

2.3. Models Utilized

Although vanilla GNN is a powerful architecture for modelling graph like data it has some limitations especially with the edges. That is why for this project six different variations of this architecture were chosen: Graph Convolutional Networks (GCN),Graph Attention Networks (GAT) , Graph Isomorphism Network (GIN), Graph Transformer (GT), Message Passing Neural Networks (MPNN) and Gated Graph Convolutional Networks (GatedGCN). These variations could be classified as Convolutional NN(GCN, GIN), Attentional NN (GAT, GatedGCN, GT) and Message-passing NN (MPNN) depending on how they aggregate information from neighbouring nodes and edges:

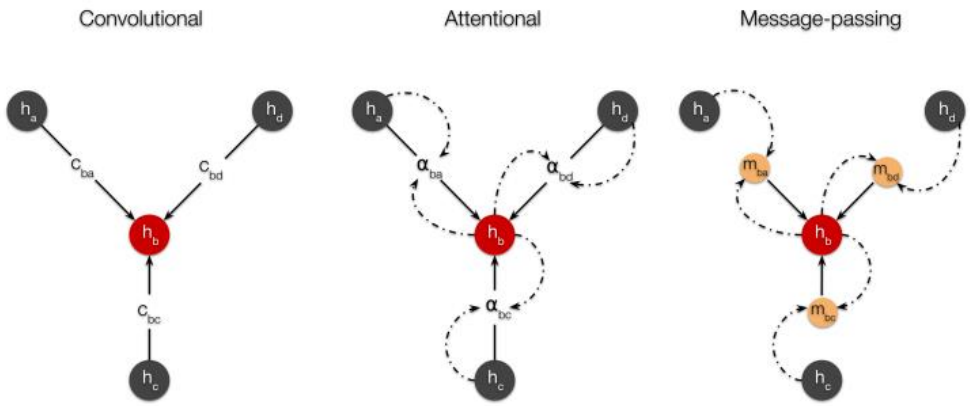


Figure 3. Representation of the 3 main GNN types from Kensert et al. (2023)

2.3.1. Graph Convolutional Networks (GCN)

2.3.1.1. Mechanism

As Dwivedi et al. (2022) state in their work, GCN are probably the simplest of GNN In the context of graph neural networks, it uses symmetric normalization. This method adjusts the contribution of each neighbour's representation by normalizing it based on the degrees of both the node itself (i) and its neighbour (j). Specifically, for a node i, the representation at the next layer (hi(ℓ+1)) is computed using the following steps:

Weighted Aggregation: Each neighbour j of node i contributes to the new representation, but the contribution is weighted by the factor $\frac{1}{\sqrt{\deg(i)\times\deg(j)}}$, where deg(i) and deg(j) are the degrees (number of connections) of nodes i and j, respectively. This normalization ensures that nodes with higher degrees do not disproportionately influence the update.

Linear Transformation: The weighted sum of the neighbours' representations is then passed through a linear transformation represented by the matrix Uℓ, which contains the learnable parameters at the ℓ-th layer.

Non-Linear Activation: Finally, a ReLU (Rectified Linear Unit) activation function is applied to introduce non-linearity to the model, resulting in the updated node representation hi(ℓ+1).

The overall node update equation can be expressed as:

$$h_i^{l+1} = ReLU\left(U^l \frac{1}{\sqrt{\deg(i)\times\deg(j)}} \sum_{j\in N} h_j^l\right) \qquad (6)$$

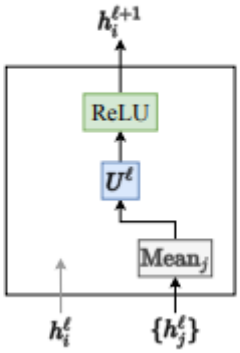


Figure 4. Schematic representation of GCN from Dwivedi et al. (2022).

2.3.2 Graph Isomorphism Network (GIN) mechanism

According to Dwivedi et al. (2022) in their work, Graph Isomorphism Networks (GIN), introduced by Xu et al. (2019), are designed to maximize the expressive power of Graph Neural Networks (GNNs) by leveraging the Weisfeiler-Lehman Isomorphism Test, a method used to determine graph isomorphism. This approach enables GINs to distinguish between different graph structures more effectively than other GNN architectures.

2.3.2.1 Mechanism

The node update process in GINs is carried out through the following equations:
Initial Aggregation:

$$\hat{h}_i^{l+1} = (1+\epsilon)h_i^l + \sum_{j \in Ni} h_j^l \qquad (7)$$

Here, $\hat{h}_i^{l+1}$ represents an intermediate updated representation for node i at the next layer. The term $(1+\epsilon)$ is a learnable parameter that controls the relative weight between the node's current representation h_i^l and the aggregated representation of its neighbors $\sum_{j \in Ni} h_j^l$
Transformation and Normalization:

$$h_i^{l+1} = ReLU\left(U^l\left(ReLU\left(BN\left(V^l \hat{h}_i^{l+1}\right)\right)\right)\right) \qquad (8)$$

In this step, the intermediate node representation $\hat{h}_i^{l+1}$ is first transformed by a linear layer with weights V^l, followed by Batch Normalization (BN) to stabilize training. A ReLU activation function is then applied, introducing non-linearity. The result is further transformed by another linear layer with weights U^l, followed by another ReLU activation to produce the final updated node representation h_i^{l+1}

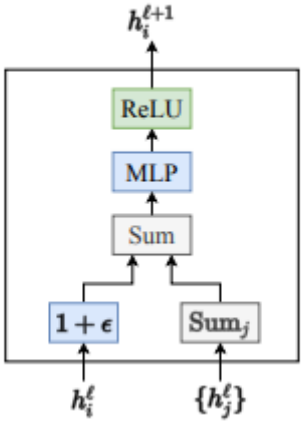


Figure 5. Schematic representation of GIN from Dwivedi et al. (2022).

2.3.3. Graph Attention Networks (GAT)

2.3.3.1 Mechanism

As highlighted by Dwivedi et al. (2022) in their paper, GAT uses an attention mechanism to assign different importance levels to the node neighbours when updating its representation. It also uses a multi-headed architecture allowing the model to capture various aspects of the neighbourhood's information through multiple independent attention mechanisms.

Its equation is the following:

$$h_i^{l+1} = Concat_{k=1}^{K}\left(ELU\left(\sum_{j \in Ni} e_{ij}^{k,l} U^{k,l} h_j^l \right) \right) \qquad (9)$$

where:

U (k, ℓ)∈R2×d/K are the linear projection matrices for the k-th attention head at the ℓ-th layer.

eij (k, ℓ) are the attention coefficients for the k-th head, which determine how much importance is given to the feature of neighbour j when updating node i's representation.

The attention coefficients eij(k, ℓ) are computed as follows:

$$e_{ij}^{k,l} = \frac{\exp\left(\hat{e}_{ij}^{k,l}\right)}{\sum_{j' \in Ni} \exp\left(\hat{e}_{ij'}^{k,l}\right)} \qquad (10)$$

where $\hat{e}_{ij'}^{k,l}$ is an unnormalized attention score given by:

$$\hat{e}_{ij}^{k,l} = LeakyReLU\left(V^{k,l} Concat\left(U^{k,l} h_i^l, U^{k,l} h_j^l \right) \right) \qquad (11)$$

with $V^{k,l} \in \mathbb{R}^{2\times\frac{d}{K}}$ being the learnable weight vector that helps determine the importance of each neighbor.

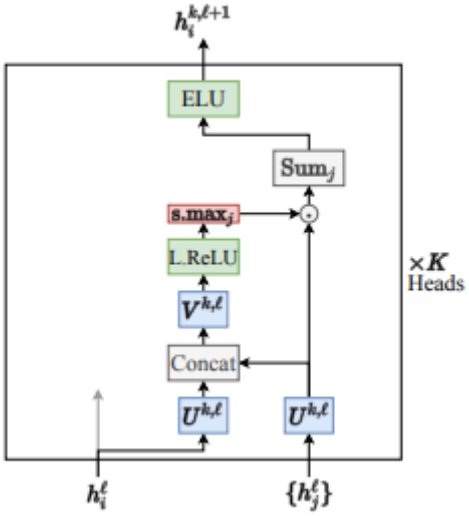


Figure 6. Schematic representation of GAT from Dwivedi et al. (2022).

2.3.4. Gated Graph Convolutional Networks (GatedGCN)

As mentioned by Dwivedi et al. (2022) in their study, the Gated Graph ConvNet (GatedGCN) introduced by Bresson and Laurent (2017) is an advanced variant of Graph Convolutional Networks (GCNs) that incorporates several enhancements such as residual connections, batch normalization, and edge gates. This design creates an anisotropic model, meaning it can treat edges and nodes differently, allowing for more nuanced updates. A key feature of GatedGCN is its explicit update of both node and edge features, setting it apart from other GNNs.

2.3.4.1. Mechanism

The node feature update in GatedGCN is performed as follows:

$$h_i^{l+1} = h_i^l + ReLU\left(BN\left(U^l h_i^l + \sum_{j \in Ni} e_{ij}^l \odot V^l h_j^l\right)\right) \qquad (12)$$

where:

$U^l, V^l \in \mathbb{R}^{d \times d}$ are learnable weight matrices.

$\odot$ denotes the Hadamard product (element-wise multiplication).

e_{ij}^l are edge gates that control the influence of neighbouring node j on node i.

Batch Normalization (BN) is applied to stabilize the learning process, followed by a ReLU activation for non-linearity.

A residual connection is used by adding the initial node feature h_j^l to the updated feature, which helps in mitigating the vanishing gradient problem during training.

The edge gates e_{ij}^l are calculated as:

$$e_{ij}^{l} = \frac{\sigma(\hat{e}_{ij}^{l})}{\sum_{j \in Ni} \sigma(\hat{e}_{ij'}^{l}) + \epsilon} \qquad (13)$$

where:

σ is the sigmoid function, which maps the edge weights to a range between 0 and 1.

ϵ is a small constant added for numerical stability.

$\hat{e}_{ij}^{l}$ is the intermediate edge feature update, defined as:

$$\hat{e}_{ij}^{l} = \hat{e}_{ij}^{l-1} + ReLU\left(BN\left(A^{l}h_{i}^{l-1} + B^{l}h_{j}^{l-1} + C^{l}\hat{e}_{ij}^{l-1}\right)\right) \qquad (14)$$

where $A^{l}, B^{l}, C^{l} \in \mathbb{R}^{d \times d}$ are learnable weight matrices applied to the features of the nodes and the edge from the previous layer.

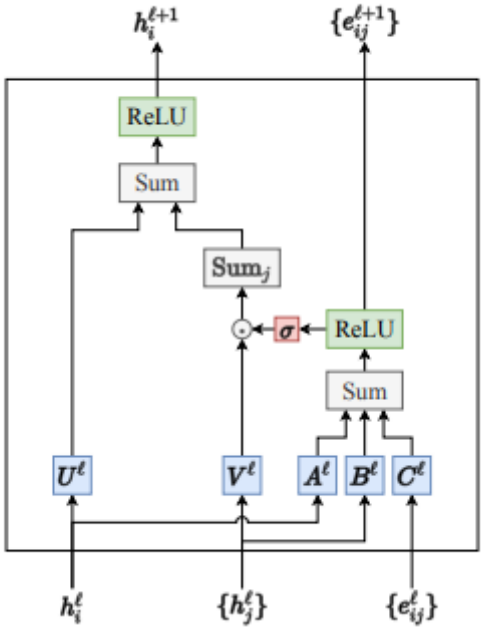


Figure 7. Schematic representation of GatedGCN from Dwivedi et al. (2022).

2.3.5. Graph Transformer (GT)

2.3.5.1. Mechanism

This passage explains the architecture of a Graph Transformer Layer based on the work of Dwivedi & Bresson (2021), which adapts the original Transformer architecture (introduced by Vaswani et al., 2017) to operate on graph data. Here's a breakdown of the key components and equations presented:

1. Graph Transformer Overview:

The Graph Transformer layer builds upon the standard Transformer architecture but is applied to graph structures where nodes represent entities, and edges represent the relationships between them.

Each node in the graph is updated by aggregating information from its neighboring nodes through an attention mechanism, similar to how words attend to each other in a sentence in the original Transformer.

2. Node Update Equations:

The primary equation for updating the representation of a node iii at layer ℓ+1\ell + 1ℓ+1 is:

$$\hat{h}_i^{l+1} = O_h^l \parallel_{k=1}^{H} \left(\sum_{j \in Ni} w_{ij}^{k,l} V^{k,l} h_j^l \right) \qquad (15)$$

$\hat{h}_i^{l+1}$: The updated representation of node i after layer ℓ+1.

Ni: The set of neighboring nodes of node i.

$w_{ij}^{k,l}$: The attention weight that node iii assigns to node j for attention head k.

$V^{k,l}$: The value matrix for attention head k.

h_j^l: The representation of node j from the previous layer ℓ\ellℓ.

O_h^l: A linear transformation matrix for node updates.

3. Attention Mechanism:

The attention weights $w_{ij}^{k,l}$ are calculated using the softmax function:

$$w_{ij}^{k,l} = \left(\frac{Q^{k,l} h_i^l \cdot K^{k,l} h_j^l}{\sqrt{d_k}} \right) \qquad (16)$$

$Q^{k,l}, K^{k,l}$: Query and key matrices for attention head k.

d_k: The dimensionality of the query/key vectors, used to normalize the dot product for numerical stability.

The softmax function ensures that the attention weights sum to 1 across all neighbouring nodes j.

4. Feed Forward Network (FFN) and Normalization:

After computing the attention outputs, the node representations are further processed by a Feed Forward Network (FFN) with residual connections and normalization layers:

$$\hat{\hat{h}}_i^{l+1} = Norm\left(h_i^l + \hat{h}_i^{l+1}\right) \qquad (17)$$

Residual Connection: Adds the original node representation h_i^l (from the previous layer) to the updated node representation $\hat{h}_i^{l+1}$. This helps in training deeper networks by mitigating the vanishing gradient problem.

Normalization: Either Layer Normalization (LayerNorm) or Batch Normalization (BatchNorm) is applied to stabilize the training process.

The resulting vector is then passed through a two-layer FFN:

$$\hat{\hat{h}}_i^{l+1} = W_2^l ReLU\left(W_1^l + \hat{h}_i^{l+1}\right) \qquad (18)$$

W_1^l , W_2^l: Weight matrices of the FFN.

ReLU: A non-linear activation function.

Finally, another residual connection and normalization step produce the final node representation for this layer:

$$h_i^{l+1} = Norm\left(\hat{\hat{h}}_i^{l+1} + \hat{\hat{h}}_i^{l+1}\right) \qquad (19)$$

5. Purpose of the Graph Transformer:

This architecture enables the Graph Transformer to capture complex dependencies between nodes in a graph, similar to how the original Transformer captures dependencies between words in a sentence.

The multi-head attention mechanism allows the model to focus on different aspects of neighboring nodes simultaneously, while the FFN and normalization steps ensure that the model can effectively learn and propagate node representations across layers.

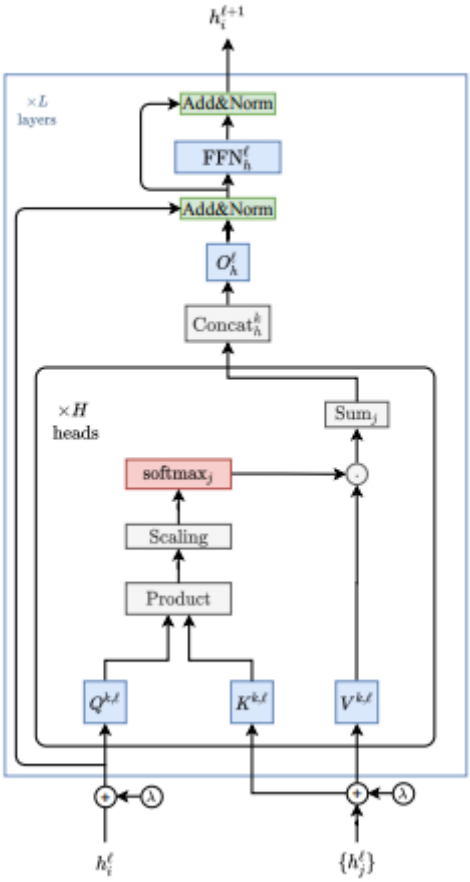


Figure 8. Schematic representation of a GT layer

2.3.6. Message Passing Neural Network (MPNN)

The Message Passing Neural Network (MPNN) described in the paper "Neural Message Passing for Quantum Chemistry" by Gilmer et al. (2017) is a general framework for performing supervised learning on graph-structured data, particularly molecular graphs. Here's a breakdown of the MPNN process:

2.3.6.1. Mechanism

1. Message Passing Phase:

Message Generation: At each node v in the graph, messages are generated from its neighbouring nodes www. The message from node w to node v is computed based on the current hidden states (representations) of both nodes $h_{v,}^{t} h_{w}^{t}$, as well as the edge features e_{vw} between them. This can be expressed as:

$$m_{v}^{(l+1)} = \sum_{w \in N(v)} Mt\left(h_{v,}^{t} h_{w}^{t}, e_{vw}\right) \quad (20)$$

where Mt is the message function, which is typically a neural network designed to combine the node features and edge features.

Message Aggregation: The messages from all neighboring nodes www are aggregated to form a single message for node v. This aggregated message is then used to update the node's hidden state.

2. Node Update Phase:

After aggregating the messages, the node v hidden state is updated using the aggregated message and its current hidden state:

$$h_v^{(l+1)} = Ut\left(h_{v,}^t m_v^{t+1}\right) \qquad (21)$$

where Ut is the update function, another neural network that typically applies a non-linear transformation (like a GRU or other RNN-like mechanism) to combine the current state with the new information from the messages.

3. Readout Phase:

After several rounds of message passing (over T time steps), a readout function R is applied to the final hidden states to generate the output of the network. The readout function must be invariant to permutations of the nodes to maintain the graph's structural properties. This can be expressed as:

$$\hat{y} = R(\{\, h_v^t \mid v \in G \,\}) \qquad (22)$$

where $\hat{y}$ represents the predicted property of the entire graph (such as a molecular property).

Summary:

The MPNN framework effectively allows nodes in a graph to iteratively update their representations by exchanging and aggregating information with their neighbors. This makes MPNNs particularly powerful for learning complex dependencies in graph-structured data, such as predicting molecular properties in quantum chemistry. The use of edge features and the ability to maintain permutation invariance are key aspects of this architecture.

3. Research Design and Objectives

3.1 Objectives

General Research Question: How can drug-binding affinity to the SARS-Cov2 main protease be predicted using Graph Convolutional Networks (GCN)?
Specific Research Question 1. How effective are regression techniques based on GCNs in predicting the binding affinity within a database of drugs and the SARS-Cov2 main protease? Specific Research Question 2. How can the molgraph code be adapted to work with the URV-generated database and accurately predict binding affinity?

General Objective: Develop and implement a Graph Convolutional Network (GCN) to predict the binding affinity between drugs and the SARS-Cov2 main protease using a database generated at Universitat Rovira i Virgili (URV).
Specific Objective 1. Evaluate the effectiveness of GCN-based regression techniques in predicting binding affinity within a specific database of drugs and the SARS-Cov2 main protease. Specific Objective 2. Modify and adapt the molgraph code for compatibility with the URV-generated database, enabling accurate prediction of binding affinity.

Table 1. Summary table and relationship between objectives and research questions

3.2 Research design

This Final Degree Project employs a sequential experimental design methodology because it is developed in two distinct phases: first, the replication and validation of Graph Convolutional Network (GCN) models on a well-established dataset (QM7), and second, the application of these validated models to a new dataset from the Universitat Rovira i Virgili (URV). This approach allows the confirmation of the models' effectiveness before applying them to a new context, ensuring that the results are reliable and relevant.

According to Campbell and Stanley (1963), sequential experimental designs are appropriate when it is necessary to validate a method in an initial phase before extending its application to new contexts, thereby ensuring the robustness and validity of the study.

Figure 9. Outline of the Sequential Experimental Design Method

3. 3 Context

The context of this Final Degree Project is centered on the application of neural networks to graph structures, with a particular focus on predicting the binding affinity of drugs to the main protease of SARS-Cov2. In the field of biotechnology, graphs are utilized to represent complex relationships between entities, such as drugs and proteins. This project lies at the intersection of artificial intelligence and bioinformatics, leveraging advanced Graph Convolutional Networks (GCN) to address pressing challenges in COVID-19 research. The

study involves replicating previous results using the QM7 dataset and applying these models to a novel dataset generated at Universitat Rovira i Virgili (URV).

4. Experimental framework

To conduct the experiments, we implemented in our code **MolGraph**, a powerful framework designed for molecular graph representation learning. MolGraph provides various tools and prebuilt models tailored for graph-based deep learning, particularly suited for tasks such as ligand affinity prediction. Its flexibility allowed us to efficiently implement and test different graph neural network architectures while leveraging state-of-the-art techniques for molecular data processing. The use of this framework significantly facilitated the experimentation process, enabling us to focus on model tuning and performance evaluation.

4.1 Implemented code:

To achieve the research objectives a code was created the main objective of which was to be able to easily create, train and evaluate GNN models and to transform the data gathered by URV regarding Ligands into a format compatible with the models created.

To do so the molgraph library was used to create 3 main functions :

1. train_model(train_database, model_struct, model_name, epochs, batch_size, validation_split=0.1, close_layer=256)

Purpose:

This function trains a graph-based neural network model (using MolGraph) with the provided data. It configures and trains the model according to the specified structure and saves both the training history and the corresponding plots.

Inputs:

train_database: A dictionary containing the training data with keys 'train'['x'] and 'train'['y'], where 'x' is molecular features (in SDF format) and 'y' are the labels (e.g., binding affinities).

model_struct: A list of tuples, where each tuple defines a layer type, the number of layers of that type, and the size of the layers. For example, [['GIN',2,128],['GAT', 2, 64]] would mean two GIN layers with128 units each and two GAT layers with 64 units each.

model_name: A string specifying the name of the model, used to name and organize output files.

epochs: Number of epochs to train the model.

batch_size: Batch size used during training.

validation_split: A float indicating the fraction of training data to use as the validation set.

close_layer: An integer defining the size of the dense layer before the output layer.

Outputs:

Returns the trained model and saves files with the training results.

Detailed Process:

Feature Extraction:

A Featurizer is used to convert the SDF representations of molecules into features usable by the model. Various features such as atom count, hybridization, formal charge, etc., are used.

Molecular Encoding:

A MolecularGraphEncoder3D transforms these features into a format suitable for the graph-based model.

Data Splitting:

The data is split into training and validation sets based on the validation_split.

Preprocessing:

Min-max scaling (MinMaxScaling) is applied to normalize node and edge features.

Model Construction:

A sequential model in Keras is created, adding layers as specified in model_struct.

Node and edge processing layers are added, followed by GAT, GCN, GIN layers, etc., according to the defined structure.

Finally, a readout layer (Readout), a dense layer (close_layer), and an output layer are added.

Compilation and Training:

The model is compiled using the Adam optimizer and mean absolute error loss (MeanAbsoluteError).

Callbacks are configured to adjust the learning rate and stop training early if no improvement is seen.

The model is trained using the training and validation sets.

Training and validation loss results are saved to a text file.

Plot Generation:

Training and validation loss plots are generated and saved using the plot_and_save function.

Code

```
def train_model(train_database, model_struct, model_name, epochs,
batch_size, validation_split=0.1, close_layer=256):
    # Initialize atom encoder with various atomic features
    atom_encoder = Featurizer([
        features.Symbol(),
        features.Hybridization(),
        features.FormalCharge(),
```

```
        features.TotalNumHs(),
        features.TotalValence(),
        features.NumRadicalElectrons(),
        features.Degree(),
        features.ChiralCenter(),
        features.Aromatic(),
        features.Ring(),
        features.Hetero(),
        features.HydrogenDonor(),
        features.HydrogenAcceptor(),
        features.CIPCode(),
        features.ChiralCenter(),
        features.RingSize(),
        features.Ring(),
        features.CrippenLogPContribution(),
        features.CrippenMolarRefractivityContribution(),
        features.TPSAContribution(),
        features.LabuteASAContribution(),
        features.GasteigerCharge(),
    ])

    # Create a 3D molecular graph encoder using atom features
    encoder = MolecularGraphEncoder3D(
        atom_encoder,
        conformer_generator=None,  # qm7 encodes conformers
        edge_radius=None,  # max radius
        coulomb=True,
    )

    # Encode training data
    x_train_full = encoder(np.array(train_database['train']['x']))
    y_train_full = np.array(train_database['train']['y'])

    # Calculate the split index based on validation split ratio
    split_index = int(len(y_train_full) * (1 - validation_split))

    # Split data into training and validation sets
    x_train = x_train_full[:split_index]
    y_train = y_train_full[:split_index]
    x_val = x_train_full[split_index:]
    y_val = y_train_full[split_index:]

    # Get the type specification of training data
    type_spec = x_train.spec

    # Create TensorFlow datasets for training and validation
    train_ds = (
        tf.data.Dataset.from_tensor_slices((x_train, y_train))
        .shuffle(1024)
        .batch(batch_size)
        .prefetch(tf.data.AUTOTUNE)
    )

    val_ds = (
        tf.data.Dataset.from_tensor_slices((x_val, y_val))
        .batch(batch_size)
        .prefetch(tf.data.AUTOTUNE)
    )

    # Preprocess node and edge features using MinMaxScaling
    node_preprocessing = MinMaxScaling(
```

```
        feature='node_feature', feature_range=(0, 1), threshold=True)
    edge_preprocessing = MinMaxScaling(
        feature='edge_feature', feature_range=(0, 1), threshold=True)

    # Adapt preprocessing to training data
    node_preprocessing.adapt(train_ds.map(lambda x, *args: x))
    edge_preprocessing.adapt(train_ds.map(lambda x, *args: x))

    # Initialize the model
    model = tf.keras.Sequential([
        keras.layers.Input(type_spec=type_spec),
        node_preprocessing,
        edge_preprocessing,
    ])

    # Loop over model layers based on model_struct, add graph layers as
specified
    for i in range(len(model_struct)):
      if model_struct[i][0] == "GAT":  # Graph Attention Network
          for j in range(model_struct[i][1]):

model.add(molgraph.layers.GATConv(units=model_struct[i][2],normalization=
'batch_norm', use_edge_features=True)),
              print(f"Layer {j}, Size: {model_struct[i][2]}, Type:
{model_struct[i][0]}")

      elif model_struct[i][0] == "GCN":  # Graph Convolutional Network
        for j in range(model_struct[i][1]):
              model.add(molgraph.layers.GCNConv(units=model_struct[i][2],
normalization='batch_norm', use_edge_features=True)),
              print(f"Layer {j}, Size: {model_struct[i][2]}, Type:
{model_struct[i][0]}")

      elif model_struct[i][0]  == "GIN":  # Graph Isomorphism Network
           for j in range(model_struct[i][1]):
              model.add(molgraph.layers.GINConv(units=model_struct[i][2],
normalization='batch_norm', use_edge_features=True)),
              print(f"Layer {j}, Size: {model_struct[i][2]}, Type:
{model_struct[i][0]}")

      elif model_struct[i][0]  == "GatedGCN":  # Gated Graph Conv Network
           for j in range(model_struct[i][1]):

model.add(molgraph.layers.GatedGCNConv(units=model_struct[i][2],normaliza
tion='batch_norm', use_edge_features=True)),
              print(f"Layer {j}, Size: {model_struct[i][2]}, Type:
{model_struct[i][0]}")

      elif model_struct[i][0]  == "GT":  # Graph Transformer
           for j in range(model_struct[i][1]):
              model.add(molgraph.layers.GTConv(units=model_struct[i][2],
normalization='batch_norm', use_edge_features=True))
              print(f"Layer {j}, Size: {model_struct[i][2]}, Type:
{model_struct[i][0]}")

      elif model_struct[i][0]  == "MPNN":  # Message Passing Neural
Network
          for j in range(model_struct[i][1]):

model.add(molgraph.layers.MPNNConv(units=model_struct[i][2],
normalization='batch_norm', use_edge_features=True))
```

```
            print(f"Layer {j}, Size: {model_struct[i][2]}, Type:
{model_struct[i][0]}")

    # Add the final layers: Readout, Dense hidden, and output layers
    model.add(Readout())
    model.add(keras.layers.Dense(close_layer, 'relu'))
    model.add(keras.layers.Dense(1))

    # Define optimizer and loss function
    optimizer = keras.optimizers.Adam(1e-2)
    loss = keras.losses.MeanAbsoluteError(name='mae')

    # Set up callbacks for learning rate reduction and early stopping
    callbacks = [
        keras.callbacks.ReduceLROnPlateau(
            monitor='val_loss',
            factor=0.1,
            patience=10,
            min_lr=1e-6,
            mode='min',
        ),
        keras.callbacks.EarlyStopping(
            monitor='val_loss',
            patience=20,
            mode='min',
            restore_best_weights=True,
        )
    ]

    # Compile the model
    model.compile(optimizer, loss)

    # Save model structure details to file
    file_name = model_name+"/"+model_name + "predictions_output.txt"
    for i in range(len(model_struct)):
        output_text = f"model: {model_struct[i][0]}, layers:
{model_struct[i][1]}, size: {model_struct[i][2]}\n"

    # Append model details to file
    with open(file_name, "a") as file:
        file.write(output_text)

    print(f"Output has been appended to {file_name}")

    # Train the model
    history = model.fit(
        train_ds,
        callbacks=callbacks,
        validation_data=val_ds,
        epochs=epochs,
        verbose=2,
    )

    # Save loss and validation loss per epoch to a file
    loss_file = model_name+"/"+model_name + "_losses.txt"
    with open(loss_file, "a") as file:
        file.write("Epoch\tLoss\tVal_Loss\n")
        min_length = min(len(history.history['loss']),
len(history.history['val_loss']))
        for epoch in range(min_length):
            loss = history.history['loss'][epoch]
```

```
            val_loss = history.history['val_loss'][epoch]
            file.write(f"{epoch + 1}\t{loss}\t{val_loss}\n")

    # Plot the training history and save it
    plot_and_save((history.history), model_name, "line")

    # Print final training information
    train_info = "Training complete. Model: {}, Epochs: {}, Batch size:
{}".format(model_name, epochs, batch_size)
    print(train_info)

    # Return the trained model
    return model
```

2. test_model(test_database, trained_model, model_name)

Purpose:

This function evaluates the trained model on a test dataset and generates predictions. It also saves the predictions, actual values, and evaluation results to a file.

Inputs:

test_database: A dictionary containing the test data with keys 'test['x'] and 'test['y'].

trained_model: The model previously trained.

model_name: The name of the model, used to organize output files.

Outputs:

Returns the evaluation metrics of the model on the test data.

Detailed Process:

Feature Extraction and Encoding:

As in train_model, a Featurizer and MolecularGraphEncoder3D are used to convert the molecules in the test set into representations suitable for the model.

Evaluation:

The model is evaluated on the test data (x_test and y_test), calculating metrics such as loss.

Prediction Generation:

The model generates predictions for the molecules in the test set.

Saving Results:

Predictions and actual values are saved to a text file.

A scatter plot of predictions versus actual values is also saved.

Code

```
def train_model(train_database, model_struct, model_name, epochs,
batch_size, validation_split=0.1, close_layer=256):
    # Initialize atom encoder with various atomic features
    atom_encoder = Featurizer([
        features.Symbol(),
        features.Hybridization(),
        features.FormalCharge(),
        features.TotalNumHs(),
        features.TotalValence(),
        features.NumRadicalElectrons(),
        features.Degree(),
        features.ChiralCenter(),
        features.Aromatic(),
        features.Ring(),
        features.Hetero(),
        features.HydrogenDonor(),
        features.HydrogenAcceptor(),
        features.CIPCode(),
        features.ChiralCenter(),
        features.RingSize(),
        features.Ring(),
        features.CrippenLogPContribution(),
        features.CrippenMolarRefractivityContribution(),
        features.TPSAContribution(),
        features.LabuteASAContribution(),
        features.GasteigerCharge(),
    ])

    # Create a 3D molecular graph encoder using atom features
    encoder = MolecularGraphEncoder3D(
        atom_encoder,
        conformer_generator=None,  # qm7 encodes conformers
        edge_radius=None,  # max radius
        coulomb=True,
    )

    # Encode training data
    x_train_full = encoder(np.array(train_database['train']['x']))
    y_train_full = np.array(train_database['train']['y'])

    # Calculate the split index based on validation split ratio
    split_index = int(len(y_train_full) * (1 - validation_split))

    # Split data into training and validation sets
    x_train = x_train_full[:split_index]
    y_train = y_train_full[:split_index]
    x_val = x_train_full[split_index:]
    y_val = y_train_full[split_index:]

    # Get the type specification of training data
    type_spec = x_train.spec

    # Create TensorFlow datasets for training and validation
    train_ds = (
        tf.data.Dataset.from_tensor_slices((x_train, y_train))
        .shuffle(1024)
        .batch(batch_size)
        .prefetch(tf.data.AUTOTUNE)
    )
```

```
    val_ds = (
        tf.data.Dataset.from_tensor_slices((x_val, y_val))
        .batch(batch_size)
        .prefetch(tf.data.AUTOTUNE)
    )

    # Preprocess node and edge features using MinMaxScaling
    node_preprocessing = MinMaxScaling(
        feature='node_feature', feature_range=(0, 1), threshold=True)
    edge_preprocessing = MinMaxScaling(
        feature='edge_feature', feature_range=(0, 1), threshold=True)

    # Adapt preprocessing to training data
    node_preprocessing.adapt(train_ds.map(lambda x, *args: x))
    edge_preprocessing.adapt(train_ds.map(lambda x, *args: x))

    # Initialize the model
    model = tf.keras.Sequential([
        keras.layers.Input(type_spec=type_spec),
        node_preprocessing,
        edge_preprocessing,
    ])

    # Loop over model layers based on model_struct, add graph layers as
specified
    for i in range(len(model_struct)):
      if model_struct[i][0] == "GAT":  # Graph Attention Network
          for j in range(model_struct[i][1]):

model.add(molgraph.layers.GATConv(units=model_struct[i][2],normalization=
'batch_norm', use_edge_features=True)),
              print(f"Layer {j}, Size: {model_struct[i][2]}, Type:
{model_struct[i][0]}")

      elif model_struct[i][0] == "GCN":  # Graph Convolutional Network
        for j in range(model_struct[i][1]):
              model.add(molgraph.layers.GCNConv(units=model_struct[i][2],
normalization='batch_norm', use_edge_features=True)),
              print(f"Layer {j}, Size: {model_struct[i][2]}, Type:
{model_struct[i][0]}")

      elif model_struct[i][0]  == "GIN":  # Graph Isomorphism Network
           for j in range(model_struct[i][1]):
              model.add(molgraph.layers.GINConv(units=model_struct[i][2],
normalization='batch_norm', use_edge_features=True)),
              print(f"Layer {j}, Size: {model_struct[i][2]}, Type:
{model_struct[i][0]}")

      elif model_struct[i][0]  == "GatedGCN":  # Gated Graph Conv Network
           for j in range(model_struct[i][1]):

model.add(molgraph.layers.GatedGCNConv(units=model_struct[i][2],normaliza
tion='batch_norm', use_edge_features=True)),
              print(f"Layer {j}, Size: {model_struct[i][2]}, Type:
{model_struct[i][0]}")

      elif model_struct[i][0]  == "GT":  # Graph Transformer
           for j in range(model_struct[i][1]):
              model.add(molgraph.layers.GTConv(units=model_struct[i][2],
normalization='batch_norm', use_edge_features=True))
```

```
            print(f"Layer {j}, Size: {model_struct[i][2]}, Type:
{model_struct[i][0]}")

      elif model_struct[i][0]  == "MPNN":  # Message Passing Neural
Network
          for j in range(model_struct[i][1]):

model.add(molgraph.layers.MPNNConv(units=model_struct[i][2],
normalization='batch_norm', use_edge_features=True))
            print(f"Layer {j}, Size: {model_struct[i][2]}, Type:
{model_struct[i][0]}")

    # Add the final layers: Readout, Dense hidden, and output layers
    model.add(Readout())
    model.add(keras.layers.Dense(close_layer, 'relu'))
    model.add(keras.layers.Dense(1))

    # Define optimizer and loss function
    optimizer = keras.optimizers.Adam(1e-2)
    loss = keras.losses.MeanAbsoluteError(name='mae')

    # Set up callbacks for learning rate reduction and early stopping
    callbacks = [
        keras.callbacks.ReduceLROnPlateau(
            monitor='val_loss',
            factor=0.1,
            patience=10,
            min_lr=1e-6,
            mode='min',
        ),
        keras.callbacks.EarlyStopping(
            monitor='val_loss',
            patience=20,
            mode='min',
            restore_best_weights=True,
        )
    ]

    # Compile the model
    model.compile(optimizer, loss)

    # Save model structure details to file
    file_name = model_name+"/"+model_name + "predictions_output.txt"
    for i in range(len(model_struct)):
        output_text = f"model: {model_struct[i][0]}, layers:
{model_struct[i][1]}, size: {model_struct[i][2]}\n"

    # Append model details to file
    with open(file_name, "a") as file:
        file.write(output_text)

    print(f"Output has been appended to {file_name}")

    # Train the model
    history = model.fit(
        train_ds,
        callbacks=callbacks,
        validation_data=val_ds,
        epochs=epochs,
        verbose=2,
    )
```

```
    # Save loss and validation loss per epoch to a file
    loss_file = model_name+"/"+model_name + "_losses.txt"
    with open(loss_file, "a") as file:
        file.write("Epoch\tLoss\tVal_Loss\n")
        min_length = min(len(history.history['loss']),
len(history.history['val_loss']))
        for epoch in range(min_length):
            loss = history.history['loss'][epoch]
            val_loss = history.history['val_loss'][epoch]
            file.write(f"{epoch + 1}\t{loss}\t{val_loss}\n")

    # Plot the training history and save it
    plot_and_save((history.history), model_name, "line")

    # Print final training information
    train_info = "Training complete. Model: {}, Epochs: {}, Batch size:
{}".format(model_name, epochs, batch_size)
    print(train_info)

    # Return the trained model
    return model
```

3. generate_database_urv_to_format(urv_database_path, k=10)

Purpose:

This function processes a dataset in the URV format (with SDF files and affinities) and converts it into a format suitable for model training, including splitting the data into k partitions for cross-validation.

Inputs:

urv_database_path: The path to the directory containing a directory named Ligand where all the SDF files named are located these files must be named LigandName_ligand.sdf and a file named Ligand_Affinity.txt file the affinity for each ligand is written like so :

LigandName affinity.

LigandName affinity.

…

k: The number of partitions for cross-validation.

Outputs:

Returns a list of k dictionaries, each containing a training and test set ('train' and 'test').

Detailed Process:

Reading Affinities:

The affinity file is read to associate each molecule code with its affinity value.

Processing SDF Files:

All SDF files in the specified folder are read, extracting the corresponding molecules and associating them with their affinities.

A list of molecules (x) and their corresponding affinities (y) is generated.

Generating Datasets:

The data is split into k partitions randomly.

For each partition, a dataset is created that includes both the training set and the test set.

Returning the Result:

A list of k dictionaries is returned, each corresponding to a cross-validation fold.

Code:

```
def generate_database_urv_to_format(urv_database_path, k=10):
    # Path to the folder containing the SDF files
    folder_path = urv_database_path + '/Ligand'

    # Path to the affinity file
    affinity_path = urv_database_path + '/Ligand_Affinity.txt'

    # Dictionary to store affinities with their corresponding ligand
codes
    affinities = {}

    # Read the affinity file and store the affinities in the dictionary
    with open(affinity_path, 'r') as f:
        for line in f:
            parts = line.split()
            if len(parts) == 2:
                code, affinity = parts
                affinities[code] = float(affinity)

    # Initialize lists to store data
    i = 0
    x = []
    y = []
    index = []
    name = []

    # Iterate over all SDF files in the specified folder and associate
them with affinities
    for filename in os.listdir(folder_path):
        if filename.endswith('.sdf'):
            # Extract the ligand code from the filename (assuming format
like '6M2N_ligand.sdf')
            code = filename.split('_')[0]
            sdf_path = os.path.join(folder_path, filename)
            affinity = affinities.get(code)

            if affinity is not None:  # Only include if affinity is
available
                with open(sdf_path, 'r') as file:
```

```
                    sdf_content = file.read()  # Read the entire SDF file
content
                    sdf_content = sdf_content.replace("$$$$", "")  #
Remove unnecessary characters
                    sdf_content = np.str_(sdf_content)  # Convert to
numpy string format
                    x.append(sdf_content)  # Add SDF content to the x
list
                    y.append(affinity)  # Add affinity to the y list
                    index.append(i)  # Add index to the index list
                    name.append(code)  # Add code to the name list
                    i = i + 1

    # Optional: Convert data to a dictionary for easier data handling
    data = {
        'x': x,
        'y': y,
        'index': index,
        'name': name
    }

    total_size = len(x)  # Total number of samples
    # Create random indices for shuffling
    indices = np.arange(total_size)
    np.random.shuffle(indices)

    datasets = []
    partition_size = total_size // k  # Determine partition size for k-
fold cross-validation

    # Create k datasets for cross-validation
    for fold in range(k):
        test_start = fold * partition_size
        test_end = test_start + partition_size

        # Assign data to train and test sets using the random indices
        dataset = {
            'train': {key: [value[i] for i in indices[:test_start]] +
[value[i] for i in indices[test_end:]] for key, value in data.items()},
            'test': {key: [value[i] for i in
indices[test_start:test_end]] for key, value in data.items()},
        }

        datasets.append(dataset)  # Append the dataset to the datasets
list

    return datasets  # Return the list of datasets
```

These functions as well as auxiliary ones are designed to work together, facilitating the training, testing, and evaluation of deep learning models generated in this project.

For more information about the code checkout the annex 1.

4.1 Replication of QM7 Results

The QM7 dataset, which contains molecular structures and their properties, was used to evaluate the performance of the graph models. The objective was to replicate the original results to validate the effectiveness of various graph neural network models.

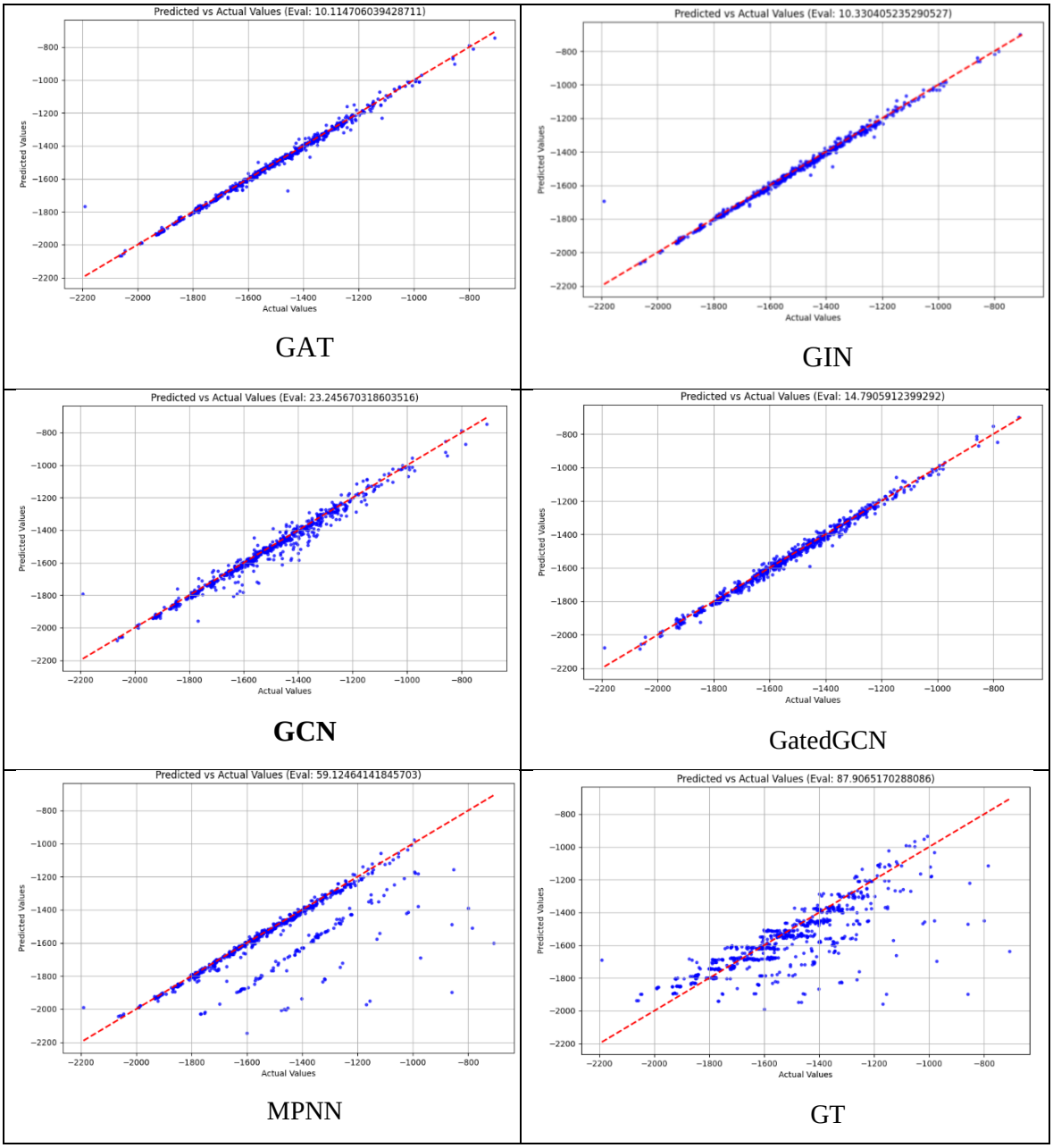


Figure10. Scatter plots from the models tested. X axis represents real values and Y axis values predicted by the models. A perfect model should have a perfect diagonal represented in the graphic by a dotted red line.

	Original paper	Replication
GAT	17.9556	10.1147
GCN	18.9011	23.2456
GT	7.5603	87.9065

GatedGCN	10.2520	14.7905
GIN	19.2505	10.3304
MPNN	15.0094	59.1246

Table 2. Summary table of the values extracted between the original paper Kensert et al. (2023) and the created models.

We managed to get close to all the results except for the GT and MPNN probably do to a lack of computational power since we were limited by my home computer.

4.2. Results from URV Database

For each model, we performed four separate iterations of training and testing, where each iteration used a different randomly selected test set. The MAE was recorded for each iteration, followed by calculation of the mean MAE, standard deviation (SD), and relative standard deviation (SD/mean * 100). The latter metric provides insight into the variability of the model's performance relative to its mean accuracy.

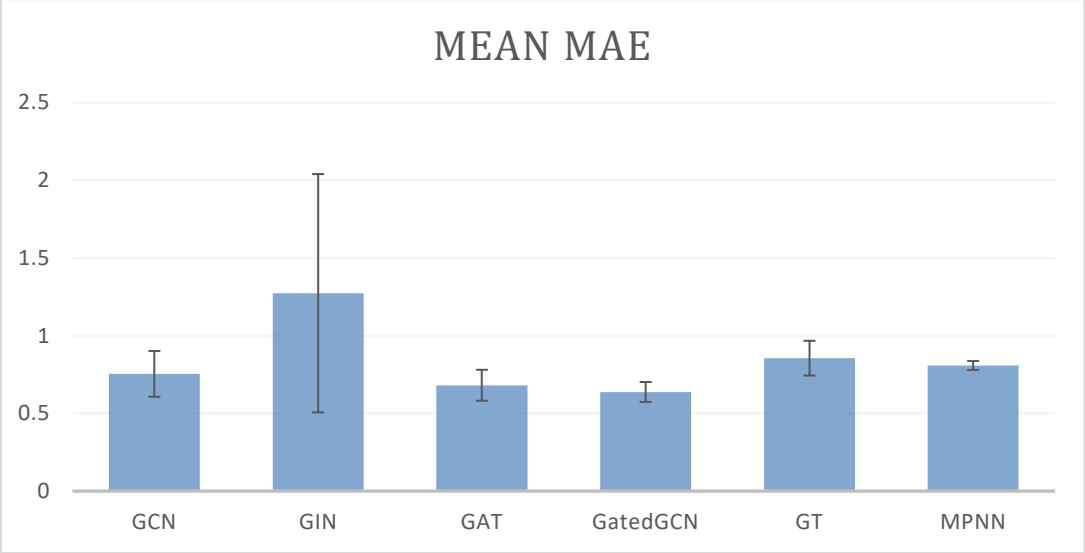


Figure 11. Table showing the mean mae of the six different models implemented with error bars.

4.2.1. Model Accuracy (Mean MAE)

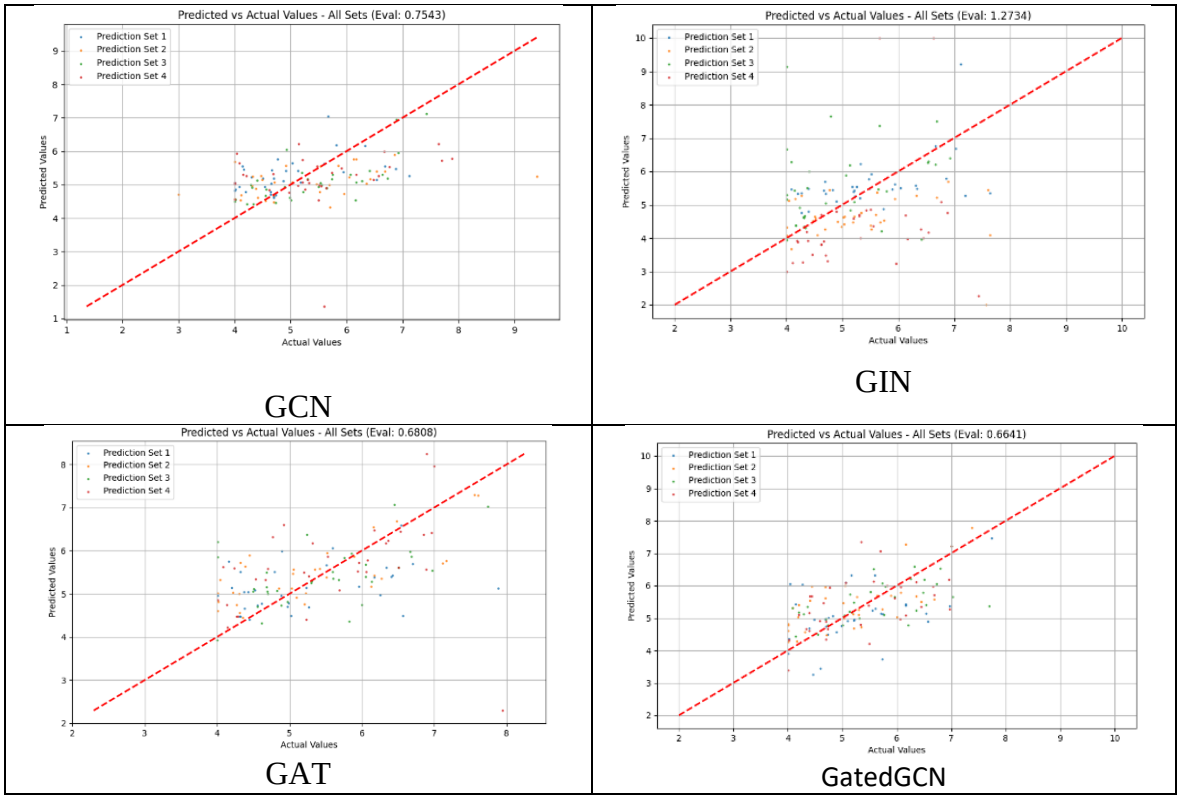
Among the models, **GatedGCN** exhibited the lowest average MAE (0.6378), suggesting that it is the most accurate in predicting ligand affinity. The second-best performer in terms of average MAE was **GAT**, with a value of 0.6808, followed by **MPNN** with a value of 0.8083.

On the other hand, **GIN** had the highest mean MAE (1.2734), indicating relatively poor performance, which can be attributed to either overfitting or the inability of GIN to generalize well on this dataset.

4.2.2 Model Stability (Standard Deviation and Relative Standard Deviation)

Stability across different test sets is a critical factor, particularly when working with small datasets. A low standard deviation and relative standard deviation indicate that a model is less sensitive to changes in the test set composition, making it more reliable in real-world applications.

- **MPNN** demonstrated the highest stability, with a standard deviation of only 0.0286 and a relative standard deviation of 3.53%. This suggests that while the average performance of MPNN is slightly higher than GatedGCN, it is extremely consistent across different test splits.
- **GatedGCN** also showed strong stability, with a relative standard deviation of 10.07%, further reinforcing its reliability alongside its excellent performance.
- **GAT** and **GT** exhibited moderate variability, with relative standard deviations of 14.65% and 13.07%, respectively. These models still demonstrate fairly consistent performance but may be somewhat more sensitive to different test data configurations.
- **GCN** had a relative standard deviation of 19.53%, indicating higher variability, though its performance is still reasonable in comparison to models like GIN.
- **GIN** was the most unstable model, with a standard deviation of 0.7666 and a relative standard deviation of 60.20%. This suggests that GIN's performance varies widely depending on the test set, making it unreliable for this specific task.



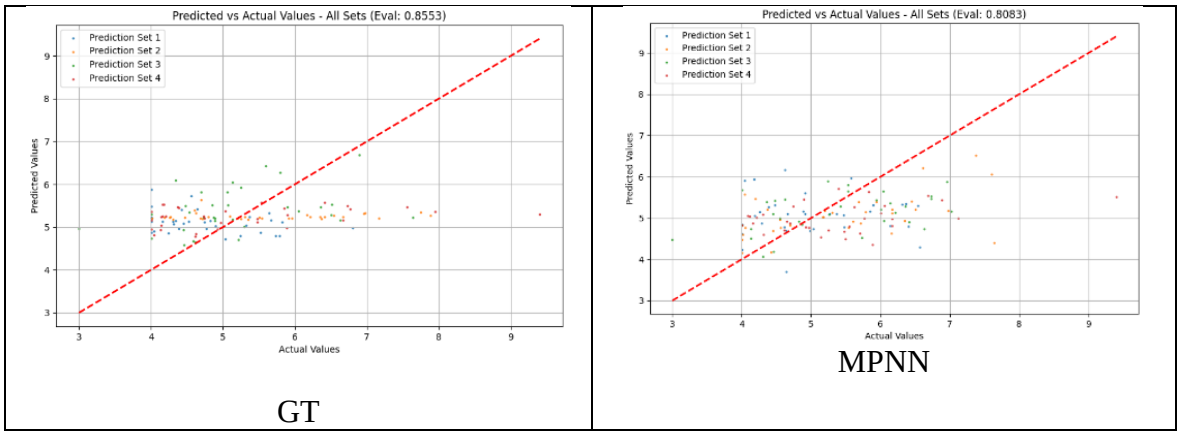


Figure 12 Scatter plots for each model tested X axis represents real values and Y axis values predicted by the models. A perfect model should have a perfect diagonal represented in the graphic by a dotted red line. Each attempt is coloured differently.

Despite the results, the scatter plots reveal a low level of understanding in the models' predictions. This is likely due to the small size of the dataset and the large size of the molecules being analysed which forced us to build smaller models with fewer and smaller layers. The complexity of the molecular structures would typically require larger and deeper models to fully capture the intricate relationships, but due to the limited computational power available, we had to compromise, which likely affected the models' performance and their ability to generalize.

5. Conclusions

This study explored the application of Graph Convolutional Networks (GCNs) and other Graph Neural Network (GNN) models to predict the binding affinity of drugs to the SARS-Cov2 main protease (Mpro). The research was divided into two phases: the replication of results using the QM7 dataset and the application of GNN models to a dataset provided by the Universitat Rovira i Virgili (URV). The following conclusions can be drawn from the analysis:

1. **Replication of Results**: The replication of results using the QM7 dataset generally demonstrated the efficacy of GNN models for molecular data analysis, though some models, such as GT and MPNN, struggled due to computational constraints. Most models performed comparably to those in prior studies, validating the potential of GNNs in chemical property prediction.
2. **Application to SARS-Cov2 Data**: When applied to the URV dataset, the GNN models exhibited varied accuracy. The best performing model, GatedGCN, achieved a mean absolute error (MAE) of 0.6378, suggesting that it is the most promising model for predicting drug-protein binding affinity. The GAT and MPNN models also performed well, but models like GIN showed higher errors and lower stability, indicating that they may not be as suited for this particular task.

3. **Challenges**: The small size of the URV dataset (344 samples) posed significant challenges in training the models effectively. The large size of the molecules being analyzed, combined with the need for smaller models due to limited computational resources, likely contributed to the observed variance in accuracy. Larger datasets and more powerful computational tools would be necessary to fully realize the potential of these models.
4. **Future Research**: Future work should focus on optimizing the model architectures to handle larger and more complex molecular structures. Additionally, exploring hybrid approaches that combine the strengths of different GNN models could further improve accuracy. The applicability of GNNs in other fields, such as social network analysis and bioinformatics, should also be explored to diversify the impact of this technology.

   In conclusion, while this study demonstrates the potential of GNNs, particularly GatedGCN, for drug potency prediction, there are limitations in the current approach that need to be addressed to improve reliability and performance. Further research and computational advancements will be critical in refining these models for real-world drug discovery applications.

References

Campbell, D. T., & Stanley, J. C. (1963). Experimental and quasi-experimental designs for research. Houghton Mifflin Company.

Dwivedi, V. P., & Bresson, X. (2021). A generalization of transformer networks to graphs. AAAI'21 Workshop on Deep Learning on Graphs: Methods and Applications (DLG-AAAI'21). arXiv. https://arxiv.org/abs/2012.09699

Dwivedi, V. P., Joshi, C. K., Luu, A. T., Laurent, T., Bengio, Y., & Bresson, X. (2022). Benchmarking graph neural networks. arXiv. https://arxiv.org/abs/2003.00982v5

Fanelli, D., Bindi, L., Chicchi, L., Pereti, C., Sessoli, R., & Tommasini, S. (2024). A short introduction to neural networks and their application to Earth and Materials Science. arXiv. https://arxiv.org/pdf/2408.11395

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., & Dahl, G. E. (2017). Neural message passing for quantum chemistry. Proceedings of the 34th International Conference on Machine Learning, PMLR 70. https://arxiv.org/abs/1704.01212

Liu, Z., & Zhou, J. (2022). Introduction to graph neural networks. Tsinghua University Press.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., & Bengio, Y. (2018). Graph attention networks. In International Conference on Learning Representations (ICLR) 2018. Retrieved from https://arxiv.org/abs/1710.10903

Kensert, A., Desmet, G., & Cabooter, D. (2023). MolGraph: A Python package for the implementation of molecular graphs and graph neural networks with

TensorFlow and Keras. *A preprint*. University of Leuven (KU Leuven) and Vrije Universiteit Brussel (VUB). Retrieved from https://arxiv.org/pdf/2208.09944

Anex

Annex 1: REDME and main example

README

Overview

This project involves training and testing a machine learning model for molecular property prediction using graph-based neural networks. The primary tools used include TensorFlow, RDKit, and MolGraph. The project includes several functions to process data, train models, evaluate models, and visualize results.

Dependencies

Ensure you have the following packages installed:

%pip install rdkit molgraph tensorflow matplotlib

Functions

1. plot_and_save

Purpose:

To save plots of the training and validation loss, as well as scatter plots of predicted vs actual values.

Inputs:

- data: The data to be plotted (either history of losses or predicted vs actual values).
- model_name: The name of the model for directory creation.
- plot_type: The type of plot to generate ('scatter' or 'line').

Outputs:

Saves the plots in specified directories.

2. train_model

Purpose:

To train a graph neural network model on a given training dataset.

Inputs:

- train_database: The training dataset.
- model_struct: The structure of the model.
- model_name: The name of the model.
- epochs: Number of epochs for training.
- batch_size: The size of batches for training.
- validation_split: The fraction of data to be used for validation.
- close_layer: Size of the final dense layer.

Outputs:

Returns the trained model and saves loss plots and model details.

3. test_model

Purpose:

To evaluate a trained model on a test dataset and generate predictions.

Inputs:

- test_database: The test dataset.
- trained_model: The trained model to be evaluated.
- model_name: The name of the model.

Outputs:

Saves predictions and evaluation metrics in a text file and scatter plot of predictions vs actual values.

4. generate_database_urv_to_format

Purpose:

To generate a dataset for training and testing from given URV SDF files and affinities.

Inputs:

- urv_database_path: Path to the URV database containing SDF files and affinity file.
- k: Number of folds for cross-validation.

Outputs:

Returns a list of datasets partitioned for cross-validation.

5. read_predictions_file

Purpose:

To read predictions from a file.

Inputs:

- file_path: Path to the predictions file.

Outputs:

Returns predictions, actual values, and evaluation metrics.

6. read_loss_file

Purpose:

To read loss values from a file.

Inputs:

- file_path: Path to the loss file.

Outputs:

Returns training and validation loss values.

7. get_next_filename

Purpose:

To get the next available filename for saving plots.

Inputs:

- base_path: Path to the directory.
- base_name: Base name for the files.
- extension: File extension.

Outputs:

Returns the next available filename.

8. generate_plot

Purpose:

To generate overall plots of predictions and loss values.

Inputs:

- predictions: List of predictions.
- vals: List of actual values.
- evals: List of evaluation metrics.
- all_loss: List of training loss values.
- all_val_loss: List of validation loss values.
- model_name: Name of the model.

Outputs:

Saves overall plots in the specified directory.

9. genera_plots_totals

Purpose:

To generate total plots for predictions and losses.

Inputs:

- model_name: Name of the model.

Outputs:

Saves overall plots for predictions and losses.

Example Usage

The following code demonstrates the main steps to train and test a model using the provided functions:

```
# Define model name and structure
model_name = "mod2"
model_struct = [["GIN", 6, 256]]

# Generate dataset
generated_database = generate_database_urv_to_format("Ligand", 10)

# Create model directory
os.makedirs(model_name, exist_ok=True)

# Train and test the model on multiple folds
for i in range(4):
    error = f"{i}"
    trained_model = train_model(generated_database[i], model_struct,
model_name, epochs=5, batch_size=15, validation_split=0.10)
    error = error + ";" + str(test_model(generated_database[i],
trained_model, model_name))
    with open(model_name + "recopilation.txt", "a") as file:
        file.write(error + "
")
    genera_plots_totals(model_name)

# Save the trained model
model_path = os.path.join(model_name, model_name)
trained_model.save(model_path)
```

Generated Files Explanation

When you run the `main.py` script, several files are generated in the process:

Training and Validation Loss Plots:

These plots are saved in the directory `model_name/model_name_plots`.
Training loss plot: `model_name/model_name_plots/validation/model_name_loss_X.png`
Validation loss plot:
`model_name/model_name_plots/validation/model_name_loss_validation_X.png`
`X` represents the iteration number.

Predictions and Actual Values Scatter Plots:

These plots are saved in the directory `model_name/model_name_plots/scatter`.
File name format:
`model_name/model_name_plots/scatter/model_name_predicted_vs_actualY.png`
`Y` represents the iteration number.

Prediction Output Text File:

This file is saved in the root of the `model_name` directory.
File name: `model_name/model_namepredictions_output.txt`
It contains the predictions, actual values, and evaluation metrics.

Losses Text File:

This file is saved in the root of the `model_name` directory.
File name: `model_name/model_name_losses.txt`
It contains the training and validation loss values for each epoch.

Recopilation Text File:

This file is saved in the root of the `model_name` directory.
File name: `model_name_recopilation.txt`
It contains the error information for each iteration.

Saved Model:

The trained model is saved in the root of the `model_name` directory.
File name: `model_name/model_name`
It contains the saved model which can be loaded for future use.