GraphDTA - Replication

Jerome Rodrigo

July 23, 2025

Abstract

This paper presents a replication study of the GraphDTA model, which integrates Graph Neural Networks (GNNs) and Convolutional Neural Networks (CNNs) to predict drug-target binding affinity. The primary goal is to validate the original study's findings while deepening my understanding of its computational and mathematical underpinnings. I analyze the model architecture, training process, and dataset used, and explore the theoretical foundations of GNNs and CNNs in the context of bioinformatics. Through this replication, I aim to enhance my knowledge in drug discovery, bioinformatics, and machine learning.

Keywords. Graph Neural Networks · Convolutional Neural Networks · Drug-Target Binding Affinity · Deep Learning · Bioinformatics · Drug Discovery · GraphDTA

1 Introduction

This paper is a math-first walkthrough of reimplementing GraphDTA (Nguyen et al.) to learn how deep learning models can predict drug–target binding affinity. Instead of summarizing results from the original work, I rebuild the full pipeline, tokenizing protein sequences, representing small molecules as graphs, encoding them with CNNs and Graph Neural Networks, and combining the learned embeddings for regression. Along the way I unpack the math behind 1D convolutions, global pooling, message passing on molecular graphs, and the loss/objective used for affinity prediction.

2 High Level Overview

So the GraphDTA pipeline works like this: we take a drug (represented as a SMILES string) and a protein (represented as an amino acid sequence), encode them separately, then combine their embeddings to predict binding affinity. For the drug, we convert the SMILES string into a molecular graph where atoms are nodes and bonds are edges. Then we run this graph through a Graph Neural Network (either GCN or GAT) which learns to represent the molecule as a single vector. For the protein, we one-hot encode each amino acid and run the sequence through a 1D CNN that also outputs a single vector. Finally, we concatenate these two vectors and pass them through a few fully connected layers to get our affinity prediction. The whole thing is trained end-to-end using MSE loss.

3 Protein encoding

3.1 One Hot Encoding

So we use one-hot encoding to represent amino acids in protein sequences. How the function works is that ita takes a sequence of amino acids and converts each amino acid into a one-hot encoded vector. The amino acids are represented by a number in a dictionary corresponding to

their index in the amino acid vocabulary. The one-hot encoded vector is length 22, where the index corresponding to the amino acid is set to 1, and all other indices are set to 0.

3.1.1 How and why one-hot encoding works

So one-hot encoding is a way to represent categorical data as binary vectors. Each category is represented by a vector where one element is set to 1 (the "hot" part/the category that is being represented) and all other elements in the vector are set to 0. It works for neural networks because it allows the model to learn relationships between different categories without assuming any ordinal relationship between them. In the context of protein sequences, one-hot encoding allows us to represent each amino acid as a unique vector, which can then be processed by neural networks. One-hot encoding works for amino acids really well as the feature space is small (22 amino acids) and the relationships between them are not ordinal.

4 Drug encoding

4.1 SMILES to Graph representation

4.1.1 Atoms to Nodes

Following the paper I used RDKit to convert SMILES compounds into molecular graphs. Each node in the graph is represented by a vector of features with most being one-hot encoded. The features include the atom symbol, the number of adjacent hydrogens, the number of adjacent atoms, the implicit value of the atom, and whether the atom is in a aromatic structure.

4.1.2 Bonds to Edges

Each edge in the graph is made by taking the bonds using RDKit. The bond notes the atom indices and the graph is undirected by making edges both ways. Also the matrix is then tranposed to be used in the GNNs.

5 Graph Neural Networks

5.1 Basic Overview

So Graph Neural Networks (GNNs) work similarly to regular neural networks, but they are built for graph-structured data. They learn to understand relationships between nodes in a graph through encoding neighboring node information. Pretty much each layer in the GNN gets information from neighboring nodes and uses it (e.g., by summing or averaging) to update the given node's representation through a shared neural transformation. This is done through a process called message passing, where each node sends and receives messages from its neighbors. The GNN learns to aggregate these messages and update the node representations iteratively. This allows the model to learn rich structure-aware embeddings, which is perfect for tasks like drug-target binding affinity prediction where the relationships between atoms in a molecule are crucial.

5.2 GCNs

So the first approach in the GraphDTA paper is to use a Graph Convolutional Network (GCN). GCNs are a type of GNN that applies convolutional operations on graph-structured data.

5.2.1 High Level Overview

A GCN layer lets each atom average its neighbors + itself, then passes that summary through the same tiny neural step (a shared set of weights) and a ReLU (keeps positives, zeros out negatives which adds some non-linearity which is essential for deep learning). Stacking layers

lets information flow multiple bonds away. Finally, max-pool takes the largest value per feature across all atoms, giving one vector for the molecule.

5.2.2 Technical Overview

We represent each molecule as an undirected graph $G = (V, E)$ with:

- $N = |V|$ atoms (nodes),
- Node features $X \in \mathbb{R}^{\{N \times C\}}$ Where N is the number of atoms and C is the number of features per atoms (13).
- We add self-loops: $\tilde{A} = A + I$ Where the degree matrix is $\tilde{D} \in \mathbb{R}^{N \times N}$ with $\tilde{D}_{ii} = \sum_{j=1}^{N} \tilde{A}_{ij}$
- Adjacency matrix $A \in \{0, 1\}^{\{N \times N\}}$ where $A_{\{i,j\}} = 1$ if atoms i and j are bonded.
- Then we have our Degree matrix $D \in \mathbb{R}^{\{N \times N\}}$ where $D_{\{i,i\}} = \sum_{\{j=1\}}^{\{N\}} A_{\{i,j\}}$ is the degree of node i which is the number of connections a node has plus itself.
- The normalized weights matrix is $S_{ij} = \frac{1}{\sqrt{D_{ii} D_{jj}}}$ if $A_{ij} = 1$ else $S_{ij} = 0$.
  - The reason the we used a normalized weight matrix is to prevent the model from being biased towards nodes with high degrees. This helps the model learn more balanced representations of nodes in the graph.
- A single GCN layer updates node features by degree-aware neighbor averaging followed by a shared linear map and ReLU: $H^{(l+1)} = \mathrm{ReLU}\big(SH^{(l)}W^{(l)}\big)$ where $H^{(l)} \in \mathbb{R}^{N \times C_l}$ and $W^{(l)} \in \mathbb{R}^{C_l \times C_{l+1}}$
  - So what this equation means is that we take the current node features $H^{(l)}$, multiply it by the normalized weights matrix S, and then apply a linear transformation with weights $W^{(l)}$ followed by a ReLU activation function. This allows the model to learn complex relationships between nodes in the graph while preventing overfitting.
- Graph-level readout: after L layers, pool node embeddings with global max to get one vector per molecule: $h_{G[j]} = \max_{\{v \in V\}} H^{(L)}_{\{v,j\}}$ (permutation-invariant).
  - This means that we take the maximum value of each feature across all nodes in the graph to create a single vector representation for the entire molecule. This is important because it allows us to capture the most important features of the molecule.

5.2.3 GCN Notation → Plain English

- $G = (V, E)$: molecule as a graph (atoms V, bonds E); $N = |V|$.
- $H^{(0)} = X \in \mathbb{R}^{N \times C}$: initial node features (13 per atom).
- $A \in \{0, 1\}^{N \times N}$: adjacency; $A_{ij} = 1$ if atoms i and j are bonded.
- $\tilde{A} = A + I$: add self-loops so each atom keeps its own signal.
- $\tilde{D}_{ii} = \sum_{j=1}^{N} \tilde{A}_{ij}$: degree from $\tilde{A}$ (neighbors + self).
- $S = \tilde{D}^{-\frac{1}{2}} * \tilde{A} * \tilde{D}^{-\frac{1}{2}}$: normalized neighbor-averaging weights (if $\tilde{A}_{ij} = 1$, then $S_{ij} = \frac{1}{\sqrt{\tilde{D}_{ii} * \tilde{D}_{jj}}}$, else 0).
- $W^{(l)} \in \mathbb{R}^{C_l \times C_{l+1}}$: shared linear map (like a dense layer) changing feature width.
- Update rule: $H^{(l+1)} = \mathrm{ReLU}\big(S * H^{(l)} * W^{(l)}\big)$ → average neighbors (incl. self) → remix features → keep positives.
- Graph readout (global max): $h_{G[j]} = \max_{v \in V} H^{(L)}_{v,j}$ → one $1 \times C_L$ vector per molecule.
- Shapes at a glance: $H^{(0)} : N \times 13 \to H^{(1)} : N \times 32 \to H^{(3)} : N \times 32 \underset{\text{max-pool}}{\to} h_G : 1 \times 32$.

5.2.4 Results

Table 1: Results from self tests of the GCN model

Dataset	Protein rep.	Compound rep.	CI	MSE
Davis	1D	Graph	0.8523	0.3536
Kiba	1D	Graph	0.7978	0.3089

So the GCN model achieved a similar CI score to the original paper (0.889) but a significantly worser MSE (0.3536 vs 0.139). This may be due to the fact that I had to change the batch size to 64 due to memory constraints. The original paper used a batch size of 512. Ontop of that I wasn't sure of their exact protein encoder structure.

5.3 GATs

Graph Attention Networks are a variant of Graph Neural Networks that use attention mechanisms for feature learning on graphs.

5.3.1 High Level Overview

GATs differ from Graph Neural Networks as they learn which neighbors matter the most, by assigning an attention coefficient to each neighbor. These coefficients are computed through a self-attention mechanism, and by assigning different weights to different neighbors.

5.3.2 GAT (Graph Attention Networks) — Technical Overview

We use the same graph setup: a molecule as an undirected graph G=(V, E) with

- N = |V| atoms (nodes),
- Node features are stored in a matrix $X \in \mathbb{R}^{N \times C}$(here $C = 13$).

Each GAT layer does:

- Linear projection (turn raw features into hidden features)
  - Equation: $z_i = Wx_i$
  - W is a matrix that reshapes or re-weights the input features
    - At first W is initialized randomly and is the matrix of learnable parameters. Throughout training it updates through backpropagation to help the model make better predictions
  - This helps the model learn a better representation of each atom
- Compute edge scores (how important neighbor j is to node i):
  - Equation: $e_{ij} = \text{LeakyReLU}\big(a^T \left[z_i; z_j\right]\big)$
  - What this means:
    - Concatenate node i's hidden vector z_i with neighbor j's hidden vector z_j (denoted as $\left[z_i; z_j\right]$).
    - Multiply by a small vector a (learned parameters).
    - Apply LeakyReLU:
      - Regular ReLU just turns any negative number to zero
      - LeakyReLU = $\max(0, x) + \text{negative\_slope} * \min(0, x)$
        - So numbers below 0 are multiplied by a miniscule negative slope in order to preserve information
- Turn edge scores into attention weights (telling the node how much to "listen" to each neighbor):
  - Equation: $\alpha_{ij} = \text{softmax}_{j \in N(i)}\big(e_{ij}\big)$
  - What this means:

  - We apply softmax over all neighbors $j \in N(i)$ (including i itself if self-loops are added)
  - This normalizes the edge scores so they sum to 1, making them proper attention weights
  - Higher e_{ij} values get higher attention weights, meaning node i will pay more attention to neighbor j

- Aggregate neighbor features weighted by attention:
  - Equation: $h_i^{(l+1)} = \text{ReLU}\Big(\sum_{j \in N(i)} \alpha_{ij} z_j\Big)$
  - What this means:
    - Instead of averaging neighbors like GCN, GAT takes a weighted sum where each neighbor's contribution is scaled by its attention weight
    - Neighbors with higher attention weights contribute more to the updated node representation
    - This allows the model to focus on the most relevant neighbors for each atom

- Multi-head attention (optional but common):
  - We can run K independent attention mechanisms in parallel (called "heads")
  - Each head learns different attention patterns
  - The outputs from all K heads are concatenated to form the final representation.
  - This gives the model more expressive power to capture different types of relationships

- Graph-level readout: after L layers, pool node embeddings with global max to get one vector per molecule: $h_{G[j]} = \max_{v \in V} H_{v,j}^{(L)}$ (same as GCN)

5.3.3 GAT Notation → Plain English

- $G = (V, E)$: molecule as a graph (atoms V, bonds E); $N = |V|$.
- $H^{(0)} = X \in \mathbb{R}^{N \times C}$: initial node features (13 per atom).
- $z_i = W x_i$: linear projection of node i's features into hidden space.
- $e_{ij} = \text{LeakyReLU}\big(a^T \big[z_i; z_j\big]\big)$: edge score measuring how important neighbor j is to node i (where $\big[z_i; z_j\big]$ denotes concatenation).
  - a is a learned attention vector that helps determine importance
  - LeakyReLU allows small negative values to pass through (unlike regular ReLU)
- $\alpha_{ij} = \text{softmax}_{j \in N(i)}\big(e_{ij}\big)$: attention weights (normalized edge scores).
  - These weights tell node i how much to "listen" to each neighbor
  - They sum to 1 over all neighbors, making them a proper probability distribution
- Update rule: $h_i^{(l+1)} = \text{ReLU}\Big(\sum_{j \in N(i)} \alpha_{ij} z_j\Big)$ → weighted sum of neighbors (by attention) → remix features → keep positives.
- Multi-head: run K attention mechanisms in parallel and concatenate results for richer representations.
- Graph readout (global max): $h_{G[j]} = \max_{v \in V} H_{v,j}^{(L)}$ → one $1 \times C_L$ vector per molecule.
- Shapes at a glance: $H^{(0)} : N \times 13 \underset{\text{GAT layer 1}}{\rightarrow} H^{(1)} : N \times (\text{hidden} \times \text{heads}) \underset{\text{GAT layer 2}}{\rightarrow} H^{(2)} : N \times (\text{hidden} \times \text{heads}) \underset{\text{GAT layer 3}}{\rightarrow} H^{(3)} : N \times \text{hidden} \underset{\text{max-pool}}{\rightarrow} h_G : 1 \times \text{hidden}$.

5.3.4 Results

Table 2: Results from self tests of the GAT model

Dataset	Protein rep.	Compound rep.	CI	MSE
Kiba	1D	Graph	0.7956	.3512

So the GAT model uses attention mechanisms to learn which neighbors are most important for each atom, allowing it to focus on the most relevant parts of the molecule. This can potentially lead to better representations compared to GCN's uniform neighbor averaging, especially for molecules where certain atomic interactions are more critical than others.

6 Conclusion

Through this replication, I've learned how graph neural networks can capture molecular structure in a way that string-based representations can't. The GCN and GAT models both show that representing drugs as graphs and using message passing to learn embeddings works well for predicting drug-target binding affinity. While my results don't exactly match the original paper (likely due to implementation differences and hyperparameter choices), the models still achieve reasonable performance with CI scores around 0.80, which shows they're learning meaningful patterns. The attention mechanism in GAT adds an interesting layer of interpretability since we can see which neighbors each atom focuses on, though in practice both approaches work well. Overall, this project gave me a solid understanding of how to apply graph neural networks to molecular data and how to combine different types of neural architectures (CNNs for sequences, GNNs for graphs) for a single prediction task.