Predicting Mutation Effects on Protein-Protein Binding Affinities Using Graph Neural Networks

Dezsö Babai

BACHELOR THESIS
Faculty of Physics
Technical University Munich

Submitted by: Dezsö Babai
Supervisor: Prof. Dr. Martin Zacharias

September 20, 2022

Contents

1 Introduction **3**

2 Proteins **5**
2.1 Amino Acids 5
2.2 Proteins 5
2.3 Protein Stability 7

3 Deep Learning **8**
3.1 Feedforward Neural Networks 8
3.2 Training Neural Networks 10
3.2.1 Forward pass 10
3.2.2 Backpropagation 11
3.3 Recurrent Neural Networks 11
3.3.1 Simple Recurrent Neural Networks 11
3.3.2 Long-Short Term Memory Networks 12
3.3.3 Gated Recurrent Unit 13
3.4 Training Recurrent Neural Networks 14
3.4.1 Back propagation Through Time 14

4 Graphs **15**

5 Graph Neural Networks **17**
5.1 Tasks on Graphs 17
5.2 Fundamental Graph Neural Network Architectures 18
5.3 Message Passing Networks 20
5.4 Training Graph Neural Networks 20
5.5 Gated Graph Neural Networks 21
5.5.1 Propagation model 21
5.5.2 Examples 22
5.5.3 Virtual Edges 23

6 Results **25**
6.1 PDB Files 25
6.2 Proposed model 26
6.3 Processing of the raw data set 26
6.3.1 Extraction of relevant residues 27
6.3.2 Creation of graphs 27
6.4 Training of the model 28

6.5 Analysis of the results 28
6.6 Comparison to other methods 32

7 Conclusion **33**

8 Appendix A - Code **34**
8.1 make_dataset.py 34
8.2 mutagenesis.py 37
8.3 utils.py 40
8.4 new_atomic.py 44
8.5 model.py 48
8.6 train.py 50
8.7 Further Remarks 53

Introduction

In almost every biological process, in a large number of genetic disorders and diseases protein-protein complexes play a decisive role. To understand the role of Protein-Protein Interactions (PPI) and the effect of mutations in such processes, we have to be able to analyze them in a fast and computationally feasible way. A promising and insightful way to analyze the impact of such mutations on protein complexes, lies in the analysis of the protein structure. [1]

A metric to measure the change in protein-protein complex stability upon point mutations is the $\Delta\Delta G$ (ddG). There is already a wide range of methods, which are more or less exact. Most of these methods apply Molecular Dynamics (MD), other models use statistical mechanics. The ideas behind these methods are empirically determined, they generally use a scoring function which is either Semi-Empirical, Statistical or Physics-Based. Semi-Empirical methods are for example, FoldX [2], BindProfX [3] in combination with MD simulations. Statistical methods are iSEE [4], JayZ and EasyE [5]. A Physics-Based method is DXCOMPLEX [6]. [7]

An example of an energy function is the Rosetta All-Atom Energy Function [8] which is already widely used in molecular modeling, or the MMFF94s forcefield introduced in [9].

As shown in Figure (1.1) have a relatively poor performance. Most of these methods, as known from MD simulations, also need a long time to compute.

Promising tools and methods can be found in the area of machine learning (ML) for example using Neural Networks [10], Convolutional Neural Networks, Support Vector Machines [11] or Graph Neural Networks [12], or other machine learning tools [13] which are faster, more accurate and universal methods to compute descriptors of protein complexes including their stability.

In this thesis we will explore the possibilities provided by graph neural network in depth. We start in the first chapter with a short introduction to amino acids and proteins, in the second chapter we take a deep dive into neural networks, then in the third chapter the fundamentals of graphs, and in the fourth chapter graph neural networks will be introduced. In the final chapter, we apply our newly gained knowledge to the use case of learning to calculate the $\Delta\Delta G$ values of the SKEMPI v.2 dataset with a graph neural network similar to that proposed in [12] and analyze our results.

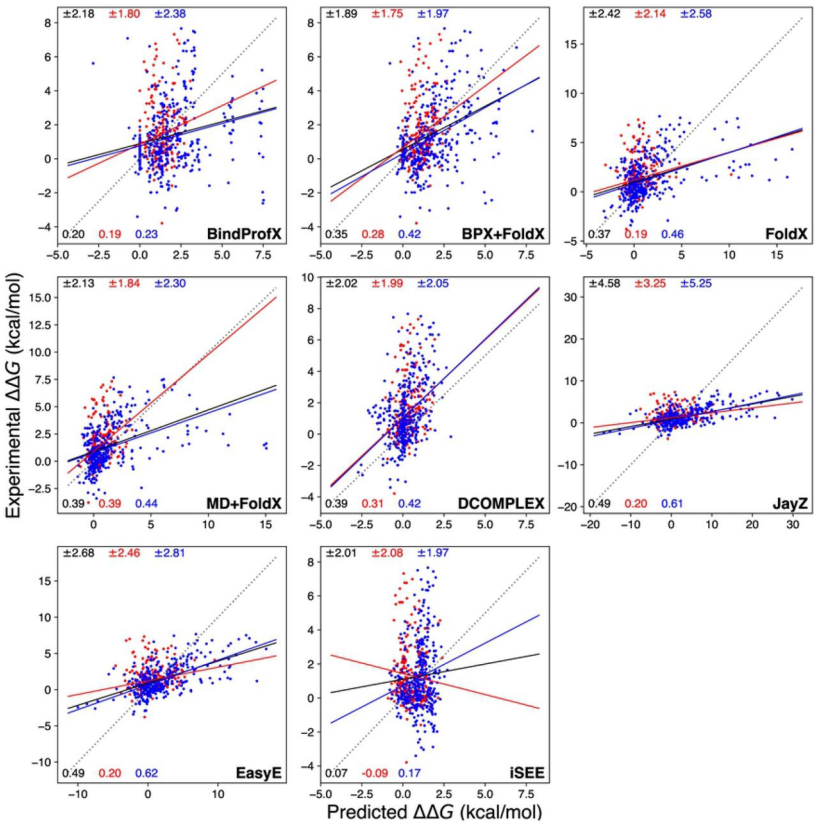


Figure 1.1: Calculated $\Delta\Delta G$ values (x-axis) compared to the experimental $\Delta\Delta G$ values (y-axis) for each listed method. The red and blue dots are two separate data sets. The red and blue lines are linear fits. The dotted line is the y=x line, if the models could perfectly predict the $\Delta\Delta G$ values, every point would be on this line. *Reproduced from* [7]

Proteins

2.1 Amino Acids

Amino acids are the basic building block for peptids and proteins. They are organic compounds. Every amino acid is constructed from the same four types of chemical groups: an amino group (NH_3^+), a hydrogen atom, an acid residue (COO^-), and a side-chain $\boldsymbol{R}$, which determines the type of amino acid one is dealing with. Each amino acid can be organized into four possible groups: negative (acidic), positive (basic), polar/neutral and nonpolar/hydrophobic. There are 20 standard amino acids found in proteins, plus Selenoicysteine and Pyrrolysine which also can be found in some liveforms. Both Selenocysteine and Pyrrolysine are somewhat rare, therefore for our calculations we will only take the standard 20 amino acids into account. A depiction of the 20 standard amino acids can be seen in Figure (2.1). [14]

2.2 Proteins

Peptids and proteins are formed by amino acids after condensation and forming of peptide bonds, this process is also called polymerization. The formula of the reaction can be seen in Figure (2.2).

Proteins have four structural levels: primary structure, secondary structure, tertiary structure and quaternary structure:

- **Primary Structure:** The primary structure of a protein is also known as its sequence (of amino acids). It is the simplest structure.
- **Secondary Structure:** The secondary structure refers to locally folded structures, which can be either an α-helix or a β-sheet. These structures are held in shape by hydrogen bonds.
- **Tertiary Structure:** The tertiary structure is the overall folded shape of a protein, which is arises as the result of energy minimization.
- **Quaternary Structure:** The quaternary structure is a structure where multiple protein chains interact with each other, forming a protein-protein complex.

The entirety of these structures defines the possible rule or behavior of a protein under certain circumstances. These substructures can be seen in Figure (2.3).

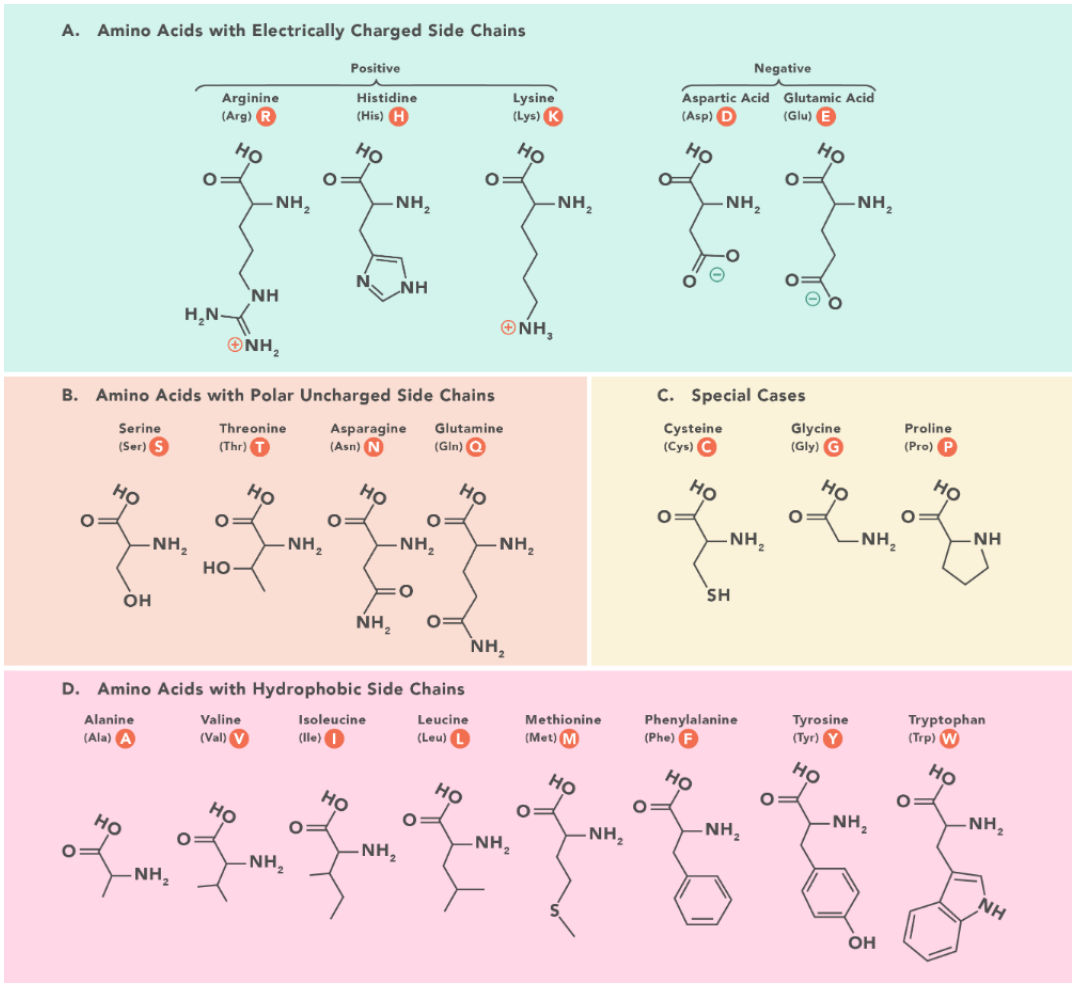


Figure 2.1: The structures of the 20 proteinogenic amino acids, annotated with their names and their three- and one letter codes. *Reproduced from* [15]

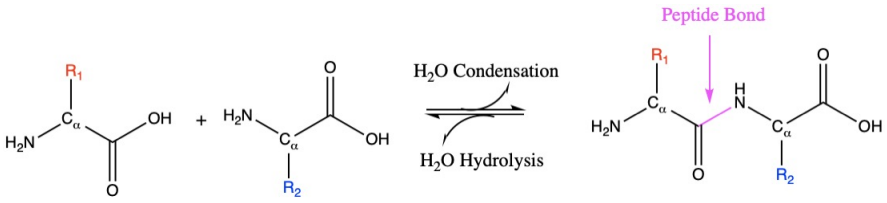


Figure 2.2: The reaction of forming (condansation) or destroying (hydrolisis) peptide bonds. *Reproduced from* [16]

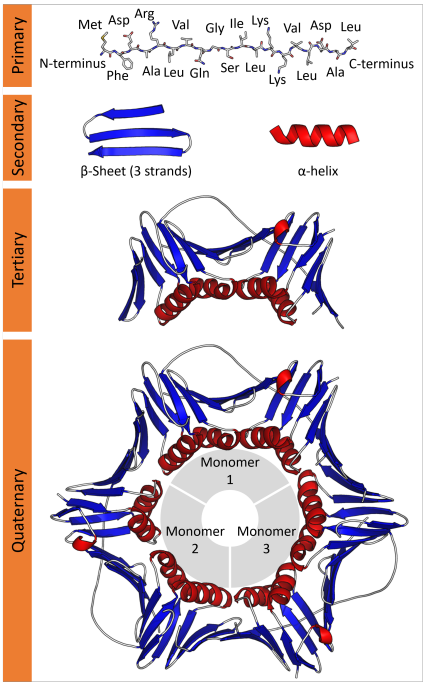


Figure 2.3: The structural units of proteins shown on PCNA ([17]): primary, secondary, tertiary and quaternary. *Reproduced from* [18]

2.3 Protein Stability

In the area of Protein-Protein Interactions, protein stability is an important factor. These interactions and the stability of protein complexes are determined by various forces: electrostatic, hydrogen bonds, van der Waals, disulphide bonds. The stability is small, typically in the range of 5-10 kcal/mol. [19]

Nevertheless, protein complex stability is also an important factor when it comes to treatment or identification of diseases and disorders. [20]

One of the factors that can affect protein complex stability are point mutations. The metric used for measuring the impact of these point mutations is the so called $\Delta\Delta G$ value (the change in the change of the Gibbs free energy). Its value is calculated by subtracting the Gibbs free energy change of complex formation ($\Delta G^{WT}_{complex}$) of the wild type and the Gibbs free energy change of complex formation with the mutation present ($\Delta G^{mut}_{complex}$):

$$\Delta\Delta G = \Delta G^{mut}_{complex} - \Delta G^{WT}_{complex} \tag{2.1}$$

The $\Delta\Delta G$ value of protein complexes can be in three ranges:

- $\Delta\Delta G < 0$: The mutation stabilizes the protein complex.
- $\Delta\Delta G \approx 0$: The mutation is neutral with almost no impact on the stability.
- $\Delta\Delta G > 0$: The mutation destabilizes the protein complex.

Deep Learning

Deep Learning (DL) is a subfield of machine learning algorithms, which use Artificial Neural Networks (ANN) to solve tasks like regression and classification. The word "deep", refers to the multiple layers of neurons contained within Artificial Neural Networks. Artificial Neural Networks were firstly modeled as an abstraction of real, biological neurons, but they can also be derived in a purely mathematical way. In the past decade, with the immense growth of computational power, deep learning has become a daily tool for research and commerce. The following sections will introduce the basic concepts of Deep Learning in a broad mathematical sense.

3.1 Feedforward Neural Networks

Feedworward networks are the simplest form of neural network. In this type of neural network data can only pass in one direction, which makes it the simplest type of neural network. Their fundamental building blocks are artificial neurons which are modelled as a simplified form of real neurons.

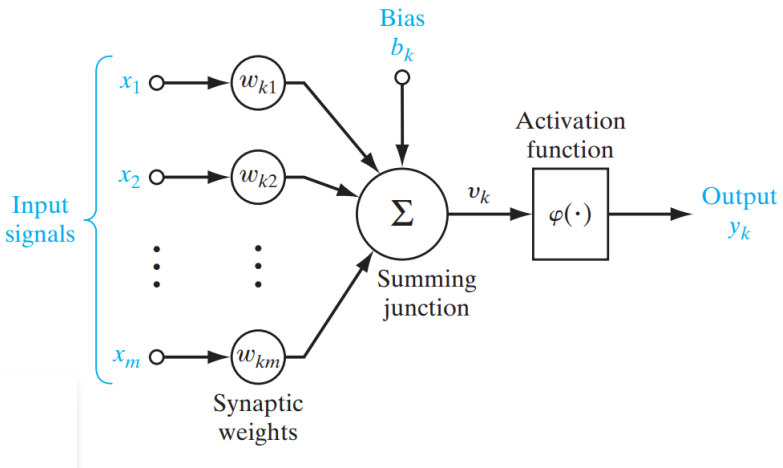


Figure 3.1: Representation of a neuron used in Artificial Neural Networks. *Reproduced from* [21]

The outputs of each neuron are calculated as follows:

$$y_k = \varphi\Big(\sum_{i=1}^{N} w_{ki} \cdot x_i + b_k\Big), \tag{3.1}$$

where y_k is the output corresponding to the kth neuron in a layer, w_{ki} is a weight, x_i the input and b_k a constant called bias, and $\varphi(.)$ is an element vise nonlinearity called activation function. Equation (3.1). can also be written in vector form:

$$\mathbf{y} = \varphi\Big(\mathbf{W}\mathbf{x} + \mathbf{b}\Big) \tag{3.2}$$

In equation (3.2) one clearly sees the resemblance to a linear model, with the difference that a nonlinearity is added. This slight change, makes neural networks - as shown in [22]; capable of approximating any function.

The choice of activation function is an important part of every model. Every activation function has to be differentiable, we want to avoid exploding outputs or gradients and vanishing gradients. Some of the most frequently used activation functions can be seen in Figure (3.2).

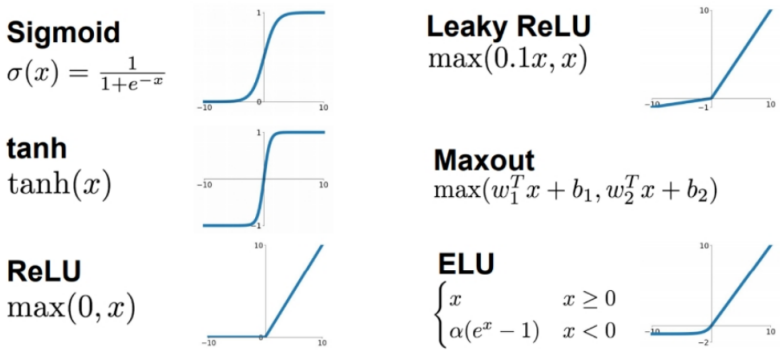


Figure 3.2: Most widely used activation functions *Reproduced from* [23]

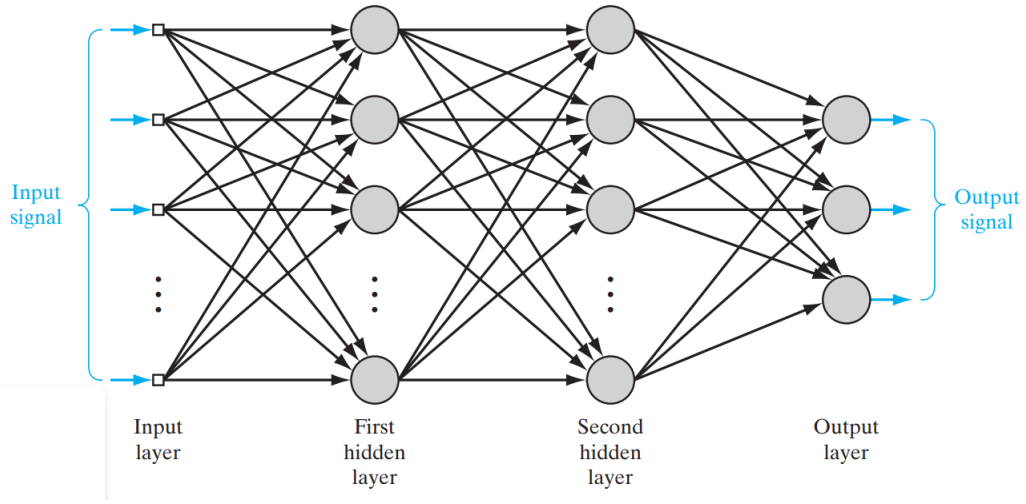


Figure 3.3: Graphical representation of a Multi-layer Feedforward Network (a Multi-Layer Perceptron). The inputs propagate from left to right. *Reproduced from* [21]

3.2 Training Neural Networks

Training neural networks requires an optimization algorithm, and is done using back-propagation. The training of neural networks consists of two steps:

1. Forward propagation
2. Backward propagation (Backpropagation)

The whole training procedure generally involves a large amount of arithmetic operations. For this reason we can use Graphical Processing Units (GPU) and Data Parallelism to speed up the process. GPUs where at first invented to accelerate image processing, thus can perform matrix multiplications in an efficient and fast way, opposed to general purpose CPUs. As Deep Learning models consists of these kind of operations GPUs are also capable of accelerating this process. Without GPUs, training of complex models in reasonable time would not be possible.

3.2.1 Forward pass

In the forward pass the inputs are fed through the neural network, the network makes a prediction. Finally an error is calculated to measure the performance of our model. This error is calculated using a predefined loss function which suits our problem the best. Some example loss functions for problems can be seen in Table (3.1)

Problem	Loss function
Two-class classification	Binary Cross Entropy, Hinge (L1)
Multiclass classification	Crossentropy, Hinge (L1)
Regression problems	Mean Squared Error (MSE/L2), Log-Cosh

Table 3.1: Different loss functions suited for classification and regression problems.

3.2.2 Backpropagation

In the backward pass each weight is upgraded starting from the output layer, with the goal to minimize the error. That is, we try to minimize a Loss function $\mathcal{J}(\mathbf{W})$, according to the parameters W which we want to learn. Gradient descent is first-order iterative optimization algorithm. The parameters $\mathbf{W}$ are updated at each iteration by taking a step towards the direction of the negative gradient:

$$\mathbf{W_{new}} = \mathbf{W_{old}} - \alpha \nabla_{\mathbf{W}_{old}} \mathcal{J}, \tag{3.3}$$

where $\nabla_{\mathbf{W}_{old}}$ is the gradient, α is the learning rate with ν¿0 determining the step size towards the direction. The gradients for each layer are calculated using the chain rule, which simplifies the process to a chain of matrix multiplications. The loss function is usually summed over a minibatch:

$$\mathcal{J}(\mathbf{W}) = \sum_{i=1}^{N_i} \mathcal{J}_i(\mathbf{W}) \tag{3.4}$$

Calculating the gradient $\nabla_{\mathbf{W}_{old}} \mathcal{J}(\mathbf{W}_{old})$ over all samples would be computationally and time expensive, for this reason usually a variant of stochastic gradient descent is used, and the loss is computed on a minibatch (holding a batch of samples together, e.g. 8, 16, 32, etc., usually powers of 2) like in equation (3.4). This way the gradient is estimated, which not only saves time, but favors generalization. After dividing the whole training data set into minibatches, the so called training loop iterates through it multiple times. One iteration through the whole data set is called an epoch.

The previously mentioned variants of gradient descent used in this work are Adam and Adadelta. Other variants are for example Adagrad and RMSProp. These methods use different techniques to achieve better convergence, either with an adaptive learning rate or physically inspired properties like momentum or friction.

3.3 Recurrent Neural Networks

Certain tasks (e.g. voice recognition, natural language processing) need to handle sequential data. These sequences can have varying ordering and length, which traditional neural network architectures only hardly can handle. Recurrent Neural Networks (RNN) try to overcome this issue by introducing multiple loops, allowing information from previous time steps to exist at later points. This kind of "memory" makes RNNs extremely powerful in many applications.

3.3.1 Simple Recurrent Neural Networks

The most simple RNN is a fully recurrent neural network with one gate, its outputs are described by the following equations:

$$\mathbf{y}_t = f\big(\mathbf{W}_y \mathbf{h}_t\big) \tag{3.5}$$

$$\mathbf{h}_t = \sigma\big(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t\big) \tag{3.6}$$

A graphical representation of these equations can be seen in Figure (3.4).

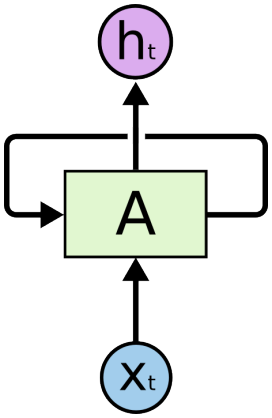


Figure 3.4: Graphical representation of a standard fully recurrent neural network. h_t is the hidden state at time step t, x_t is the input at time step t. *Reproduced from* [24]

As seen in equation (3.6), each output is directly influenced by the output of the previous time step. This works especially well for short sequences, with no long-term dependencies.

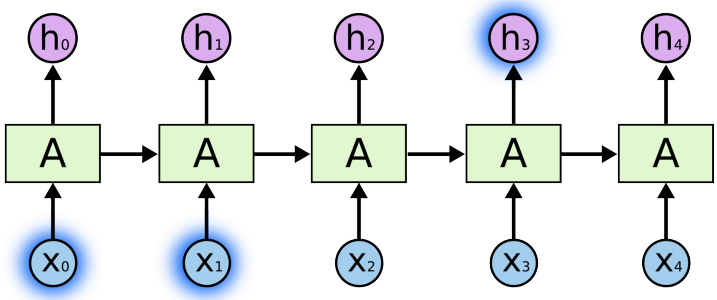


Figure 3.5: An illustration of short-term relationships between different time steps of processing a sequencia via a simple RNN. *Reproduced from* [24]

3.3.2 Long-Short Term Memory Networks

In the case of long-term dependencies in sequences, simple RNNs fail to establish correlation between distant points of the sequence as illustrated in Figure (3.6).

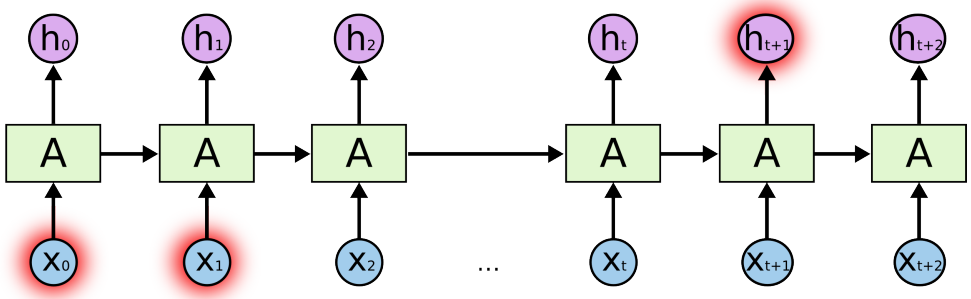


Figure 3.6: An illustration of long-range relationships between different time steps of processing a sequencia via a simple RNN. *Reproduced from* [24]

To solve the problem caused by long-term dependencies, Long Short Term Memory Networks (LSTM) were introduced in [25] and gained great popularity over time. The outputs are computed as follows:

$$\mathbf{i}_t = \sigma\big(\mathbf{W}_i \cdot [\mathbf{x}_t, \mathbf{h}_{t-1}]\big) \tag{3.7}$$
$$\mathbf{f}_t = \sigma\big(\mathbf{W}_f \cdot [\mathbf{x}_t, \mathbf{h}_{t-1}]\big) \tag{3.8}$$
$$\mathbf{c}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot tanh\big(\mathbf{W}_c \cdot [\mathbf{x}_t, \mathbf{h}_{t-1}]\big) \tag{3.9}$$
$$\mathbf{o}_t = \sigma\big(\mathbf{W}_o \cdot [\mathbf{x}_t, \mathbf{h}_{t-1}]\big) \tag{3.10}$$
$$\mathbf{h}_t = \mathbf{o}_t \odot tanh\big(\mathbf{c}_t\big), \tag{3.11}$$

where i_t, f_t and o_t are the input gate, forget gate and output gate, c_t is the cell state and h_t the hidden state. A graphic representation of these equations (gates) can be seen in Figure (3.7).

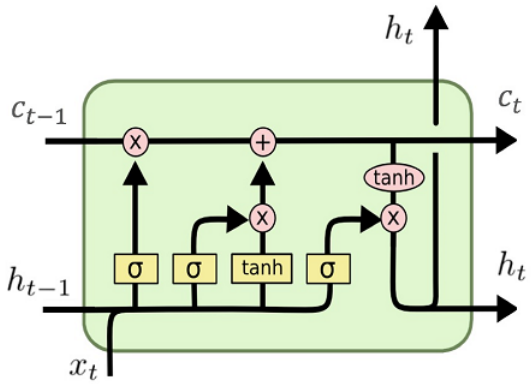


Figure 3.7: Graphical representation of a single LSTM Cell. The yellow squares are element-wise activation functions, the red circles element-wise linear functions. c^{t-1} is the cell state from the previous time step, x_t the current input. *Reproduced from* [24]

Similarly to simple RNNs, only the outputs of the previous time steps influence directly the new outputs. The crucial difference is made by the introduced gating mechanisms. The previous input, output and hidden state, all get assigned an own single layer neural network, with learnable weights. Also two important gates are introduced: the forget gate and the cell state. The forget gate assigns a number between 0 and 1 to every previous cell state, thus deciding what to forget and what to remember, thus being a form of memory. The cell state is a state carried along in every time step, representing some kind of global context.

3.3.3 Gated Recurrent Unit

Another alternative RNN is the Gated Recurrent Unit (GRU). It is almost identical to the LSTM Network, with a slight change - the forget gate (memory) is missing. The outputs are calculated as follows:

$$\mathbf{r}_t = \sigma\big(\mathbf{W}_{ir}\mathbf{x}_t + \mathbf{W}_{hr}\mathbf{h}_{(t-1)} + \mathbf{b}_{hr}\big) \tag{3.12}$$

$$\mathbf{z}_t = \sigma\big(\mathbf{W}_{iz}\mathbf{x}_t + \mathbf{W}_{hz}\mathbf{h}_{(t-1)} + \mathbf{b}_{hz}\big) \tag{3.13}$$

$$\mathbf{n}_t = tanh\big(\mathbf{W}_{in}\mathbf{x}_t + \mathbf{b}_{in} + \mathbf{r}_t \odot \big(\mathbf{W}_{hn}h_{(t-1)} + \mathbf{b}_{hn}\big)\big) \tag{3.14}$$

$$\mathbf{h}_t = \big(1 - \mathbf{z}_t\big) \odot \mathbf{n}_{(t-1)} + \mathbf{z}_t \odot \mathbf{h}_{(t-1)} \tag{3.15}$$

where z_t and r_t are the update gate and reset gate, $\tilde{h}$ is the candidate output and h_t is the output. A graphical representation of these equations (gates) can be seen in Figure (3.8). [26]

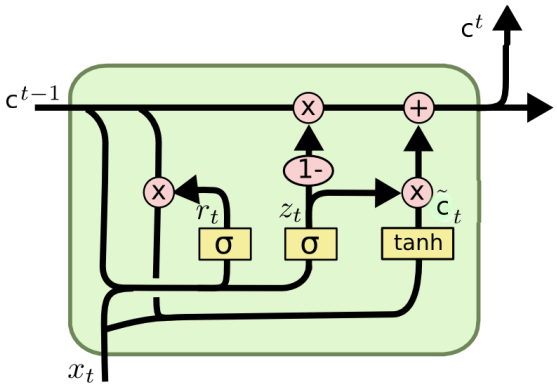


Figure 3.8: Graphical representation of the GRU Cell. The yellow squares are element-wise activation functions, the red circles element-wise linear functions. c^{t-1} is the cell state from the previous time step, x_t the current input. *Reproduced from* [24]

3.4 Training Recurrent Neural Networks

3.4.1 Back propagation Through Time

The back propagation through time (BPTT) algorithm was introduced in [27] to extend the backpropagation algorithm onto recurrent neural networks. Every recurrent neural network can be unfolded into a multilayer feedfordward network, as shown in Figure (3.9). This procedure can be generelized for any type of RNN.

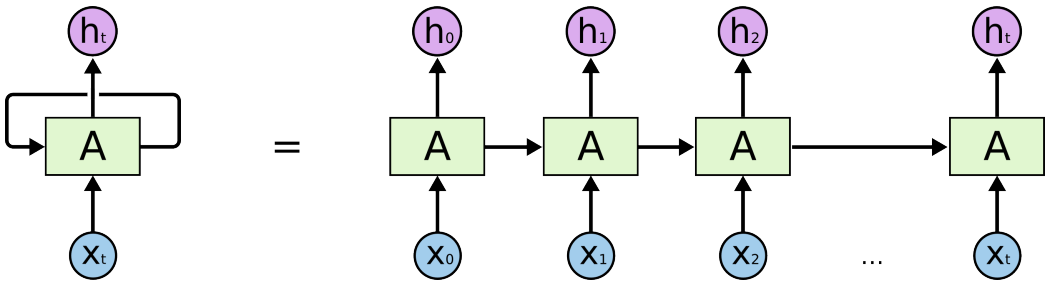


Figure 3.9: Unrolling of an RNN. *Reproduced from* [24]

After unrolling the RNN which we want to train, we can use backpropagation in the same manner as before.

Graphs

Many problems have a (nonlinear) network structure, especially real-life problems. This could be for example an electricity or railway network of a city. Graphs can be used to represent these network structures, such as molecules. In this section graphs will be introduced as a data structure, without any excursion on graph theory and algorithms on graphs.

Definition 1 (Graphs)**.** *A graph is a data structure which can be denoted as a set* $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$*, where* $\mathcal{V} = \{v_1, v_2, ..., v_N\}$ *is a set of nodes with* $|\mathcal{V}| = N$ *and* $\mathcal{E} = \{e_1, e_2, ..., e_M\}$ *with* $|\mathcal{E}| = M$ *is a set of edges. [28]*

An example of a graph can be seen in Figure (4.1). In this case

$$\mathcal{V} = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10$$

and

$$\mathcal{E} = \{a_{ij} \text{ if node i is connected to node j, with } i, j \in \mathcal{V}\}$$

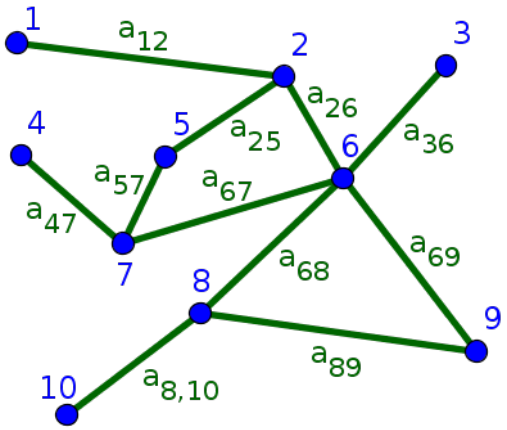


Figure 4.1: An example graph. Nodes are blue dots, edges are green lines. *Reproduced from* [29]

A multitude of objects can be represented as graphs. This includes social networks, images, and for our convenience - also molecules. In molecules the atoms are treated as nodes and the edges between them are defined by the chemical bonds between them.

The connections between the nodes of a graph be represented by an Adjacency Matrix:

Definition 2 (Adjacency Matrix)**.** *For a given Graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$ the corresponding adjacency matrix is denoted as $\mathbf{A} \in \{0,1\}^{NxN}$, with each entry of $\mathbf{A}$ representing the connectivity of two nodes in the following manner:*

$$\mathbf{A}_{ij} = \begin{cases} 1 & \textit{if nodes } v_i \textit{ and } v_j \textit{ are adjacent} \\ 0 & \textit{if nodes } v_i \textit{ and } v_j \textit{ are not adjacent} \end{cases}$$

[28]

The adjacency matrix for our previous example shown in Figure (4.1) is shown in equation (4.1).

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{4.1}$$

The memory needed to store an adjacency matrix is proportional to its size N^2, and accessing each of its entries results in a computational drawback when the adjacency matrix becomes large. Therefore a third representation will be used in our computations to represent connectivity, the so-called edge list.

Definition 3 (Edge List)**.** *The edge list $\mathbf{E}$ is a sparse matrix $\mathbb{B} \in 1, 2, ..., N^{2xM}$ with two rows, each row containing the indices of the connected edges respectfully.*

The edge list corresponding to the adjacency matrix shown in equation (4.1) is looks like follows:

$$\mathbf{E}= \begin{bmatrix} 0 & 1 & 1 & 1 & 2 & 3 & 4 & 4 & 5 & 5 & 5 & 5 & 5 & 6 & 6 & 6 & 7 & 7 & 7 & 8 & 8 & 9 \\ 1 & 0 & 4 & 5 & 5 & 6 & 1 & 6 & 1 & 2 & 6 & 7 & 8 & 3 & 4 & 5 & 5 & 8 & 9 & 5 & 7 & 7 \end{bmatrix}$$

The edge list clearly holds the same amount of information as the adjacency matrix but in a compact, memory-efficient way.

Graph Neural Networks

There are many cases, where similarities between two objects does not imply a similarity between the properties of these objects. The underlying structure of this type of data is a non-euclidean space, where machine learning techniques which only operate in an euclidean space have no chance to analyze the data correctly in the long run. Protein structures are this type of data. Each protein has a different sequence, a different folded structure. Although the different interactions which shape them and some structural parts are similar between proteins, generally each protein behaves differently.

To apply deep learning to non-euclidean spaces, Geometric Deep Learning was proposed in [30].

One type of Geometric Deep Learning methods are Graph Neural Networks (GNN). They have gained large popularity due to their universality and expressive power. [28]

In the following sections we will introduce the main ideas behind Graph Neural Networks focusing on the fundamental ideas and Message Passing Networks.

5.1 Tasks on Graphs

There are three different levels of tasks on graphs one may want to solve:

- Node Level tasks
- Edge Level tasks
- Graph Level tasks

A Node level task could be segmentation, an edge level task could be edge detection. In graph level tasks we try to predict something that characterizes the entire graph, for example in our case, we want to analyze molecular graphs of protein structures.

5.2 Fundamental Graph Neural Network Architectures

The most simple GNN used to learn embeddings on graphs consists of three different MLPs, each one applied to a different component. This method can be used to learn a context vector for each embedding, but it does not really use the connectivity yet. A depiction of the model can be seen in Fig. (5.1)

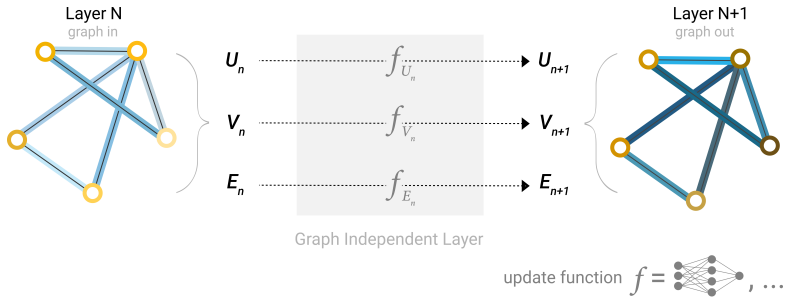


Figure 5.1: The most simple GNN. Seperate MLPs applied to each embedding. U, V and E are the node, edge and global embeddings. The function f is an MLP, the index refers to it being applied to $\mathbf{U}_n$, $\mathbf{V}_n$ or $\mathbf{E}_n$. *Reproduced from* [31]

To get information out of this graph, the so-called pooling operation is used. Pooling is a procedure in which we concatenate the embeddings into a matrix and then aggregate them in a predefined manner (e.g. mean, sum, product, max). A corresponding depiction can be seen in figure (5.2). After this operation the final predicitions can be made, usually with an MLP.

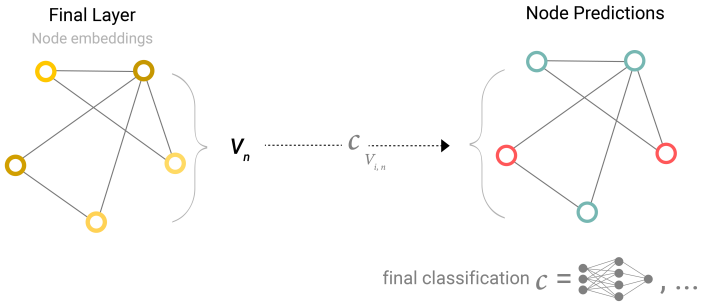


Figure 5.2: Depiction of the pooling operation. $\mathbf{V}_n$ are node features, c is an MLP applied to $\mathbf{V}_n$. *Reproduced from* [31]

These two fundamental ideas regarding processing of graph based data with neural netorks leave us with three simple worklflows for the previously introduced levels of tasks, each applying the previously introduced layers to a graph.

- Node level tasks. We only have information about the edges and want to predict node level features. The corresponding model is shown in Figure (5.3).

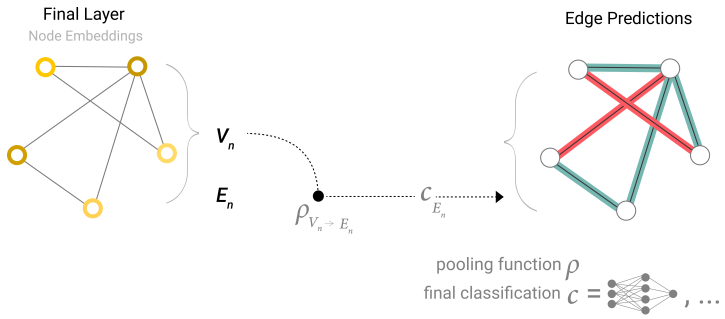


Figure 5.3: A simple GNN solving a node level task. $\mathbf{V}_n$ are the node features, $\mathbf{E}_n$ are the edge features, ρ is the pooling function, c is an MLP applied to the node features. *Reproduced from* [31]

- Edge level tasks. We only have information about the nodes and want to predict edge level features. The corresponding model is shown in Figure (5.4.

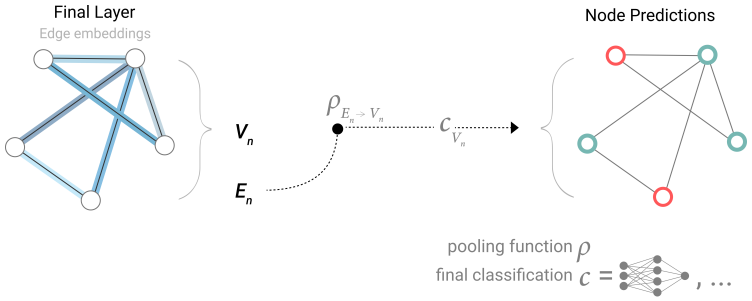


Figure 5.4: A simple GNN solving an edge level task. $\mathbf{V}_n$ are the node features, $\mathbf{E}_n$ are the edge features, ρ is the pooling function, c is an MLP applied to the edge features. *Reproduced from* [31]

- Global tasks. In this case we can use both edge level or node level features, or just one to predict a global feature of a graph. The corresponding model is shown in Figure (5.5).

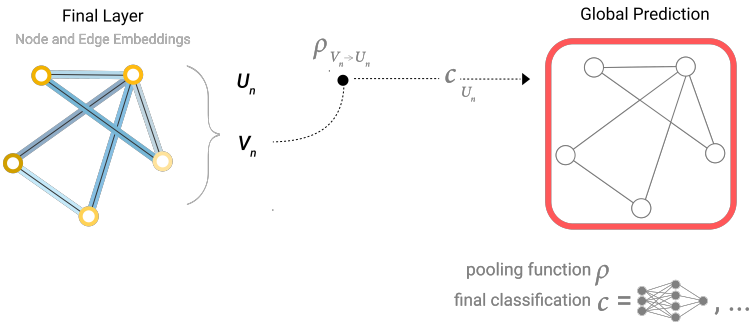


Figure 5.5: A simple GNN solving a global level task. $\mathbf{V}_n$ are the node features, $\mathbf{E}_n$ are the edge features, ρ is the pooling function, c is an MLP applied to the edge features or the node features. *Reproduced from* [31]

5.3 Message Passing Networks

Many problems that can be represented with graphs, intrinsically involve information exchange between the nodes, for example molecular property prediction. Atoms influence eachother's "embeddings", creating the molecules with the properties we now. To allow this kind of information exchange, Message Passing Neural Networks (MPNN) were introduced in [32].

Message passing works in a relative simple manner: for every node in a graph, the neighboring embeddings have to be gathered and then aggregated into a single embedding vector, aggregation happens in the same manner as previously described. Finally these new feature vectors are passed through a neural network.

Although a sophisticated approach, message passing has still its flaws. A lot of message passing steps may be required for information to pass from a node A to a node B which is far away. To deal with this, one may use virtual edges i.e. artificial edges that make information passing faster. This method proves to be really useful when working with molecular graphs, and especially effective paired with neural networks, because neural networks can learn which edges to disregard if they're not important.

5.4 Training Graph Neural Networks

As the nodes in graphs can have arbitrary connections between them, simple backpropagation can't be used in classic graph neural networks. The recurrent algorithm used in such cases is the Almeida-Pineda Algorithm[1] described in [34], which requires the function which makes predictions about the graph to be a contraction map. This is a hard requirement and it restricts the performance of graph neural networks immensely. Therefore other methods were invented using recurrent neural networks (LSTM, GRU, RNN), which release this requirement but also bring instability in terms of convergence into the frame. As more complex RNNs are performing good on filtering information and memorising long range correlations in sequences, we will use a sequence based approach to analyze the forces between atoms in proteins.

[1] This was also explained in some manner in Richard Feynmans senior thesis about forces in molecules [33].

5.5 Gated Graph Neural Networks

In case of Gated Graph Neural Networks (GGNN) as proposed previously we give up the requirement that the propagation function should be a contraction map. It uses the previously introduced Gated Recurrent Units instead of relying on the convergence of the Almeida-Pineda algorithm. The recurrence of the GRU is unrolled for a fixed number of steps T, which can also be understood as the number of layers being used. Finally backpropagation in time is used in the learning step. [35]

5.5.1 Propagation model

The propagation in such a network works in the following manner:

1. The feature vectors of the nodes are padded with zeros ($\mathbf{h_v^{(1)}}$), to match their size to the adjency matrix $\mathbf{A}$. (Equation (5.1))
2. The message passing step is performed between connected nodes (Equation (5.2)).
3. The GRU is applied to each of the nodes feature vectors. (Equation (5.3))

$$\mathbf{h}_v^{(1)} = [\mathbf{x}_v^T, \mathbf{0}]^T \tag{5.1}$$

$$\mathbf{a}_v^{(t)} = \mathbf{A}_v \Big[\mathbf{h}_1^{(t-1)} \Big| \mathbf{h}_2^{(t-1)} \quad ... \quad \mathbf{h}_{\mathcal{V}}^{(t-1)} \Big]^T \tag{5.2}$$

$$\mathbf{h}_v^{(t)}, \mathbf{h}_n = GRU\big(\mathbf{a}_v^{(t)}, \mathbf{h}_0\big) = GRU\big(\mathbf{a}_v^{(t)}, \mathbf{0}\big) \tag{5.3}$$

The equations used to calculate the output at the time step t is the following:

$$\mathbf{h}_v^{(1)} = [\mathbf{x}_v^T, \mathbf{0}]^T \tag{5.4}$$

$$\mathbf{a}_v^{(t)} = \mathbf{A}_v \Big[\mathbf{h}_1^{(t-1)} \Big| \mathbf{h}_2^{(t-1)} \quad ... \quad \mathbf{h}_{\mathcal{V}}^{(t-1)} \Big]^T \tag{5.5}$$

$$\mathbf{r}_v^t = \sigma\Big(\mathbf{W}_r \mathbf{a}_v^{(t)} + \mathbf{U}_r \mathbf{h}_v^{(t-1)} \Big) \tag{5.6}$$

$$\mathbf{z}_v^t = \sigma\Big(\mathbf{W}_z \mathbf{a}_v^{(t)} + \mathbf{U}_z \mathbf{h}_v^{(t-1)} \Big) \tag{5.7}$$

$$\mathbf{n}_v^{(t)} = tanh\Big(\mathbf{W} \mathbf{a}_v^{(t)} + \Big(\mathbf{r}_v^{(t)} \odot \mathbf{h}_v^{(t-1)} \Big) \Big) \tag{5.8}$$

$$\mathbf{h}_v^{(t)} = \big(\mathbf{1} - \mathbf{z}_v^t \big) \odot \mathbf{h}_v(t-1) + \mathbf{z}_v^t \odot \mathbf{n}_v^{(t)} \tag{5.9}$$

Where $\mathbf{x_v}$ is a node feature vector, $\mathbf{h_v^{(t)}}$ are the padded feature vectors at step t, $\mathbf{A}$ is the adjency matrix, $r_v^{(t)}$ is the reset gate, $z_v^{(t)}$ is the update gate, $n_v^{(t)}$ and $h_v^{(t)}$ are the candidate output and the output of the gated recurrent unit.

Based on the previous section about recurrent neural networks, one immediately sees the equivalence of the last four equations to a Gated Recurrent Unit with bias set to zero, and the equations can now be written in much simpler form:

First the node features are padded with zeros (eq. 5.1). Then a message passing step is performed between nodes, which can be seen in equation R D—V—×2D—V— (5.2), where $\mathbf{A} \in \mathbb{R}^{DVxDV}$ is is the matrix describing the communication between the nodes (adjency matrix). In the last step a GRU is applied to the updated node features for

a certain mount of timesteps T, to extract more relevant features. Here $\mathbf{a}_v^{(t)}$ is the final output of the *GRU*, $\mathbf{h}_n$ the final hidden state. Another useful feature of this model is that the edges can have certain weights to describe importance of connections, which comes in handy when dealing with different types of bonds. The weighting happens trough multiplying each entry of the adjency matrix with an edge weight.

5.5.2 Examples

Undirected Graph

For an example we will take a look at a water molecule (H_2O) and the corresponding graph and adjency matrix. For simplicity the only node feature used is the one-hot encoded type of atom.

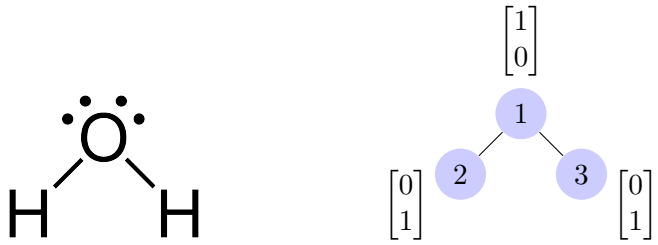


Figure 5.6: A water molecule (left) and the corresponding graph.

The corresponding adjency matrix can easily deduced from the graph seen in fig (5.6):

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \tag{5.10}$$

Now we concatenate the feature vectors to a matrix $\mathbf{H}$ as seen in equation (5.5):

$$\mathbf{H} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 0 \end{bmatrix} \tag{5.11}$$

The message passing step can now be perfomed in form of multiplication of the to matrices:

$$\mathbf{M} = \mathbf{AH} = \begin{bmatrix} 2 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} \tag{5.12}$$

The results of the message passing procedure can be seen in Figure (5.7)

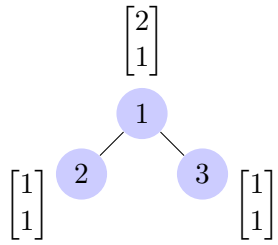


Figure 5.7: Resulting graph after message passing between the nodes.

In the next step the GRU is applied to each feature vector. An example output can be seen in figure (5.8).

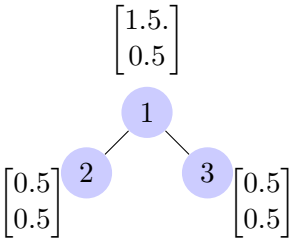


Figure 5.8: Resulting graph after applying the GRU on the nodes.

Finally the feature vectors are pooled. In case of molecular property prediction like prediction of binding affinity it is intuitive to use a sum pool. So, the final output of the GNN is:

$$\begin{bmatrix} 2.5 \\ 1.5 \end{bmatrix}$$

This procedure is then repeated for T steps. After T steps, a final pooling is applied and a readout function which is usually an MLP. Limiting ourselves to molecules containing three atoms in this example, this model could be used to determine binding angles (water: 104.5°) of such molecules or intramolecular forces.

5.5.3 Virtual Edges

By introducing virtual edges between atoms, electrostatic, van der Waals, etc. forces can be simulated, leading to more expressive models:

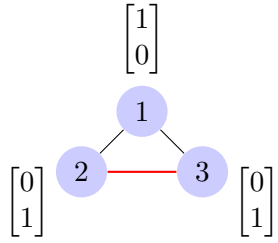


Figure 5.9: The graph representation of a water molecule with an added virtual edge between the two hydrogen atoms.

Directed graphs

The above procedure can be generalized for directed graphs by using a matrix A $\in \mathbb{R}^{DVx2DV}$, where D is the size of a feature vector, V is the number of node. This matrix double the size of the previous matrix. The first half describes incoming edges, the second half outgoing edges.

Results

Our aim was to accurately predict the effect of point mutations on protein-protein complex stability. To do this we first have to get a feeling for proteins, the data format (PDB files) they are saved in, process the atomic and molecular information and create the corresponding molecular graphs and feature vectors. This process requires a deeper understanding about the underlying protein structures and how to handle them computationally and effectively. Opposed to this, the final task of designing a proper model, capable of solving our task is a far more intuitive task.

In the following few sections these steps will be broken down into reasonable subprocedures and explained in depth, starting with the data set that was used.

The data set used in this paper was the SKEMPI v2.0 database [36] (containing 4943 single point mutations).

6.1 PDB Files

The PDB File format is used to store data of proteins and biological macromolecules, for example atomic coordinates, and at which chain and residue an atom is. An excerpt of an example PDB File is shown in figure (6.1)

1	ATOM	1	N	SER	A	1	12.880	5.246	-4.370	1.00 44.14	N
2	ATOM	2	CA	SER	A	1	13.781	5.819	-3.336	1.00 45.16	C
3	ATOM	3	C	SER	A	1	13.894	4.890	-2.125	1.00 48.42	C
4	ATOM	4	O	SER	A	1	14.895	4.186	-1.978	1.00 53.15	O
5	ATOM	5	CB	SER	A	1	13.275	7.200	-2.886	1.00 46.62	C
6	ATOM	6	OG	SER	A	1	14.332	8.106	-2.580	1.00 42.51	O
7	ATOM	7	N	LEU	A	2	12.871	4.869	-1.268	1.00 46.55	N
8	ATOM	8	CA	LEU	A	2	12.913	4.037	-0.052	1.00 46.73	C
9	ATOM	9	C	LEU	A	2	11.851	2.953	0.145	1.00 42.13	C
10	ATOM	10	O	LEU	A	2	10.714	3.068	-0.302	1.00 40.38	O
11	ATOM	11	CB	LEU	A	2	12.910	4.931	1.195	1.00 47.50	C
12	ATOM	12	CG	LEU	A	2	14.131	5.819	1.429	1.00 48.89	C
13	ATOM	13	CD1	LEU	A	2	14.167	6.186	2.891	1.00 45.96	C
14	ATOM	14	CD2	LEU	A	2	15.422	5.103	1.000	1.00 49.50	C
15	ATOM	15	N	ASP	A	3	12.253	1.903	0.846	1.00 40.68	N
16	ATOM	16	CA	ASP	A	3	11.388	0.777	1.156	1.00 41.06	C
17	ATOM	17	C	ASP	A	3	11.970	0.123	2.411	1.00 39.23	C
18	ATOM	18	O	ASP	A	3	12.746	-0.824	2.334	1.00 40.54	O
19	ATOM	19	CB	ASP	A	3	11.375	-0.209	-0.011	1.00 41.53	C
20	ATOM	20	CG	ASP	A	3	10.345	-1.301	0.159	1.00 43.03	C
21	ATOM	21	OD1	ASP	A	3	10.022	-1.645	1.311	1.00 43.43	O
22	ATOM	22	OD2	ASP	A	3	9.858	-1.819	-0.865	1.00 50.47	O

Figure 6.1: An excerpt from a PDB file.

6.2 Proposed model

As seen in research papers like [10], [37], [38], [39] or [13], many deep learning architectures were trained successfully to solve similar problems, sometimes performing better, sometimes not. Starting from a practical point of view, the most intuitive idea to solve our problem, would be to take a look at both at the wild type and the mutant type of the protein then somehow compare the differences, define some kind of measure and predict the corresponding $\Delta\Delta G$ value. The proposed model, depicted in figure (6.2) does exactly this operation in a fast and surprisingly exact way.

The main idea behind our proposed model is taken from [12]. A depiction of the model can be seen in Figure (6.2). The steps of residue extraction and feature extraction had to be fully hard-coded to fit our special use case. These two parts are generally the hardest, as one needs to get a feeling for the data and the information contained. The processing step consists of the definition of a model, varying its hyperparameters like number of layers, and optimizing it through a training algorithm. At this step, finding the right hyperparameters is the most difficult task.

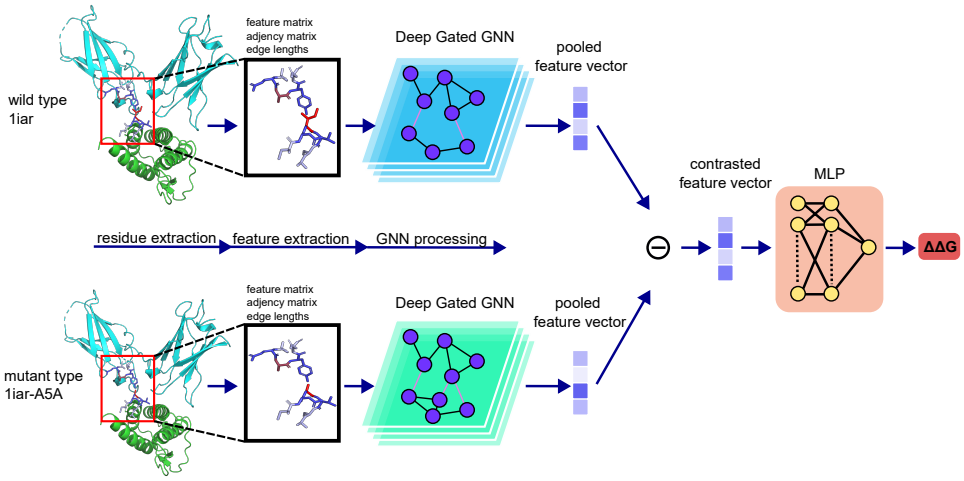


Figure 6.2: The proposed model to compute the $\Delta\Delta G$ value of a point mutation. In the first two steps the data is extracted from the protein structures, then two seperate GNNs process the generated graphs, followed by pooling. Finally the two pooled feature vectors are substracted and an MLP makes a final prediction of the $\Delta\Delta G$ value.

In the following sections the working process of the model will be described in detail.

6.3 Processing of the raw data set

The procedure of the creation of the processed data set can be thorn down to three points which have to be completed for each mutation:

1. Create the mutated version and save both mutant and wild-type using PyMol [40] in the PDB format (Code in section 8.2)

2. Extract the region of the mutation both from the mutant and the wild-type, save them in the PDB format. (Code in section 8.3)

3. Extract node features, edge lengths, adjency matrix and store them. (code in section 8.3)

6.3.1 Extraction of relevant residues

Assuming that the impact of point mutations on the $\Delta\Delta G$ value and other properties decrease with distance from the actual mutated residue, only residues in close neighborhood to the mutated residue will experience the impact of the mutation. That is, we don't have to process the whole protein structure, which favors speed of computation.

For each mutation there is an identifier, which defines our mutating residue, consisting of a chain identifier and a residue number. As an example the mutation **LI38G** on the protein *1CSE* means, that the 38th amino acid, which is Leucine (L) on chain I, mutates to Glycine. To process these mutations they were saved in an Excel file in the following format:

	pdb_id	mut_id	ddg
1	1cse	LI38G	2.25
2	1cse	LI38S	1.17
3	1cse	LI38P	6.67
...	...	...	...

Table 6.1: An example on how to declare mutations in the data set. (index.xlsx)

At first the center of geometry is calculated for every residue in the protein. Then we calculate the distance for every residue from the mutating residue, and for each chain we find the closest one within a cutoff distance of 12Å. This leaves us with the most important residues taking part in interactions between the separate chains. Finally for each of these residues the neighbors are extracted up to the second nearest on the same chain. This extracted "molecule" is then saved as a PDB File for further processing.

6.3.2 Creation of graphs

The graphs were constructed using the Graphein package ([41]) and its changed atomic edge construction function. The slight change of the atomic edge construction function was, that besides the basic atomic graph of the extracted residues, edges of long range interactions between atoms of different chains were introduced, if the distance of the two atoms was lower than a cutoff distance of 9Å. These virtual edges made information exchange between separate chains possible, "simulating" intermolecular forces.

In the final step for every node (atom) a feature vector was defined and stored in a feature matrix. The feature matrix, edge index, the length of every edge (bonds/virtual edges) were stored both for the Wild-type extracted residues and the mutants. A feature vector is rather simple in our case. It contains the one-hot[1] encoded identifier of the

[1]One-hot encoding is used in ML to quantify categorical data. It produces a vector with length equal to the possible categories. Every value in the vector is 0 except the *ith* entry which is 1 and represents the category that the corresponding object belongs to.

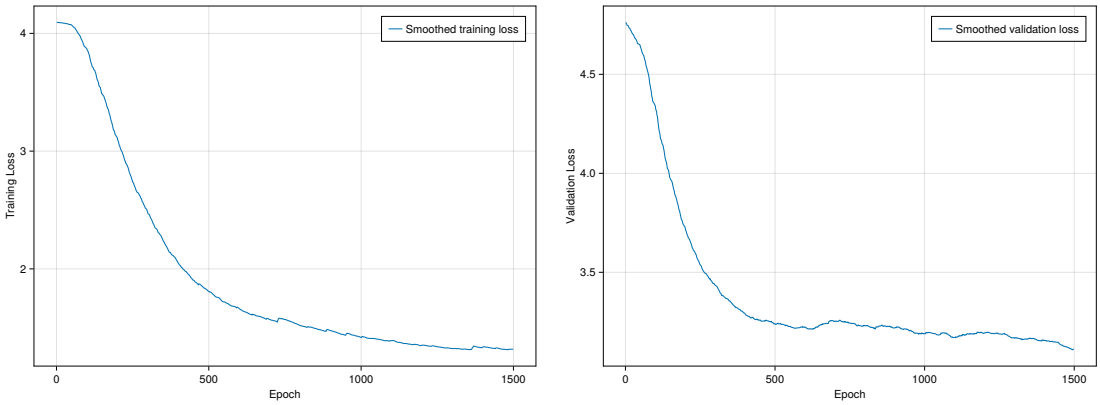
corresponding atom, the number of valence electrons, and the electronegativity. Other features can bea easily added by changing the corresponding function in section (8.3).

6.4 Training of the model

For training, the data set consisting of 4943 samples has been split up into a train and validation data set (90%-10%). The loss function was chosen to be the Mean Squared Error (MSE), as it suits our regression problem the most. For the optimization the Adam optimizer was used and an exponential learning rate scheduler to ensure a faster convergence. The model was trained until the loss stopped decreasing (around 1500 epochs).The performance was also validated for each epoch on the validation data set. The time needed for the training procedure was about one day, depending on how big the extracted part of the protein was and the batch size. The best batch size for training turned out to be 32.

6.5 Analysis of the results

An example for losses as a function of epochs in a training run can be seen in Figure (6.7), they are decreasing functions which is expected for a successful learning process.



(a) Training loss plotted against the epochs (b) Validation loss plotted against the epochs

Figure 6.3: Caption

After 1500 epochs the accuracy on the validation set no longer increased. The performance of the model on the training set can be seen in Figure (6.4), is really good. The regressed line is almost identical to the y=x line, therefore the model preforms really well, even better than most of the classical methods as we will see. To measure the correlation between the experimental $\Delta\Delta G$ values and the predicted $\Delta\Delta G$ values, we also calculate the Pearson Correlation Coefficient (PCC) when we pass a data set through our model. For the training data set the PCC is $r = 0.81$.

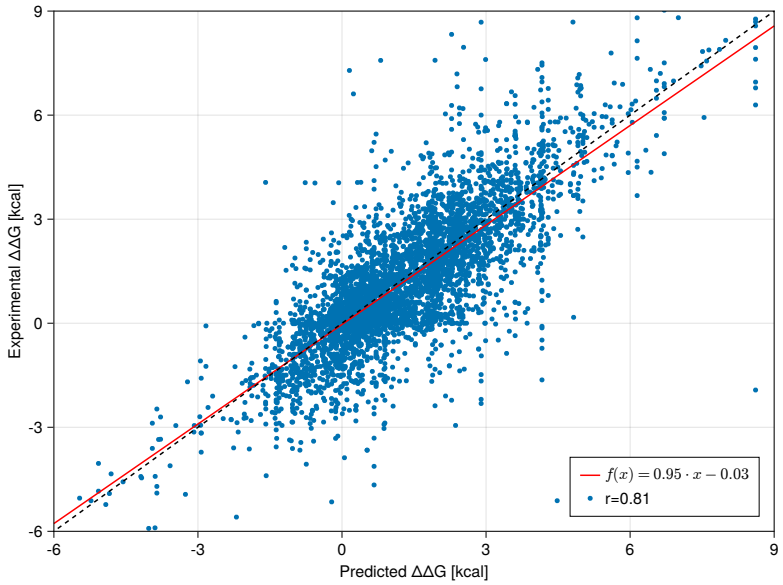


Figure 6.4: Predicted $\Delta\Delta G$ values (x-axis) compared to experimental $\Delta\Delta G$ values (y-axis) on the training set. The red line is a linear fit on the data points. The black dashed line is the y=x line, if the model could perfectly predict the $\Delta\Delta G$ values, every point would be on this line, r is the Pearson Correlation Coefficient.

As previously mentioned, the model was also validated on previously not seen data. The performance on the validation set can be seen in Figure (6.5). It performs clearly worse, but given the information it has, and the assumptions we made at the data set creation, it is still surprisingly good. The PCC here is significantly lower than before, $r = 0.44$.

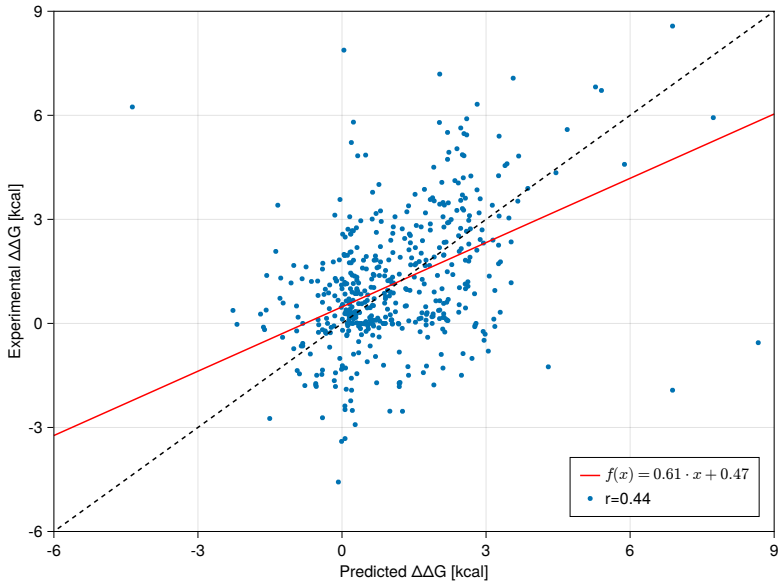
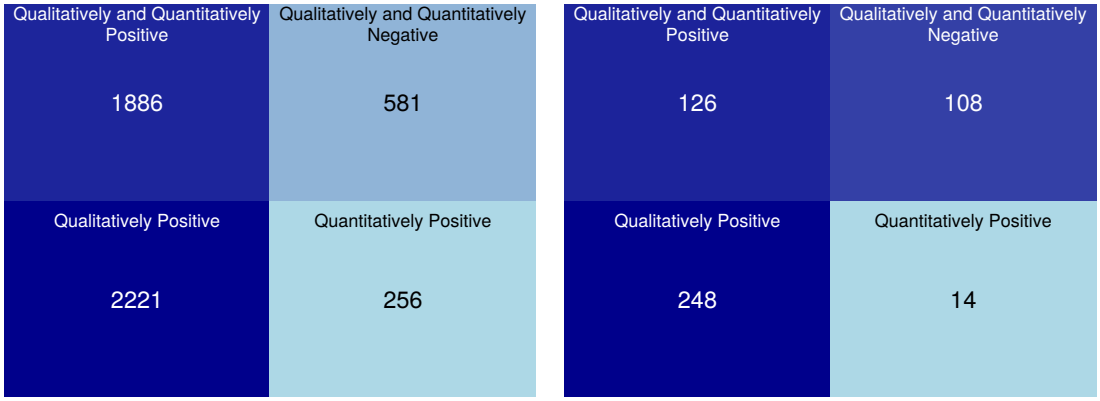


Figure 6.5: Predicted $\Delta\Delta G$ values (x-axis) compared to experimental $\Delta\Delta G$ values (y-axis) on the validation set. The red line is a linear fit on the data points. The black dashed line is the y=x line, if the model could perfectly predict the $\Delta\Delta G$ values, every point would be on this line.



(a) (b)

Figure 6.6: Confusion matrix of our trained model on the whole data set (a) and the validation data set (b). The model is capable of predicting the qualitatively right $\Delta\Delta G$ value most of the time.

The apparent outliers on Figures (6.4) and Figure (6.5) can be the outcome of many reasons. One of this reasons can be the position of the mutation. Our model and the data set creation is designed to deal with mutations on protein-protein interfaces, but if the mutation happens in the core or the interior of a chain, it wont be able to predict a usable value. Another problem which can lead to outliers is the lack of backbone optimization at the creation of the mutations. These problems can be resolved by extracting larger parts of the protein and using software for backbone optimization, but these are out of the scope of this thesis.

Diving deeper into the accuracy of this model, we can distinguish between four classes of predictions:

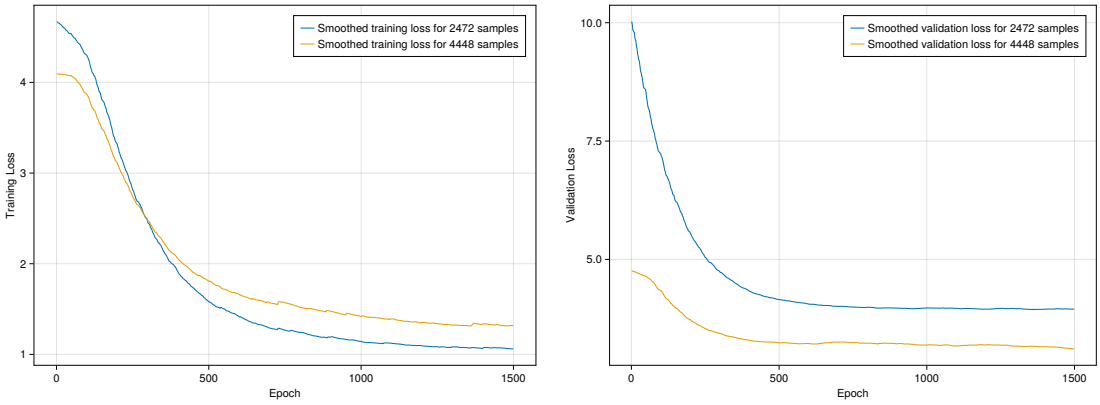
- **Qualitatively and Quantitatively Positive:** Best case, exact predictions.
- **Qualitatively Positive:** The prediction can be used to determine whether a mutation stabilizes or destabilizes a protein complex but the scale of stabilization/destabilization is wrong.
- **Quantitatively Positive:** The prediction can be used to approximately determine the absolute value of $\Delta\Delta G$ but it can't be told if the mutation stabilizes or destabilizes the protein complex.
- **Qualitatively and Quantitatively Negative:** Worst case, the prediction does not fit any experimental data.

The corresponding quantities on the training data set to these can be seen in Figure (6.6a) and on the validation data set in Figure (6.6b), depicted as a confusion matrix.

As we seen in the previously depicted confusion matrices, even though there are some inaccuracies on the validation data set, this model can be reliably used to predict qualitative changes in $\Delta\Delta G$ values.

For additional validation the model also was trained and validated on a subset (500 samples) of the data set, to show that the model is in deed learning and the correlation seen above, isn't just a lucky coincidence or memorization. The corresponding training

losses and validation losses are depicted in Figure (6.7).



(a) Training loss plotted against the epochs on a subset of the training data set.

(b) Validation loss plotted against the epochs on a subset of the training data set.

Figure 6.7: Caption

As can be seen, the training loss converges to a lower value if a subset is used, because fewer samples are easier to learn, and a validation loss converges to a higher value than previously, because the model cant generalize well enough. The outcome of the evaluation on the validation data set can be seen in Figure (6.8).

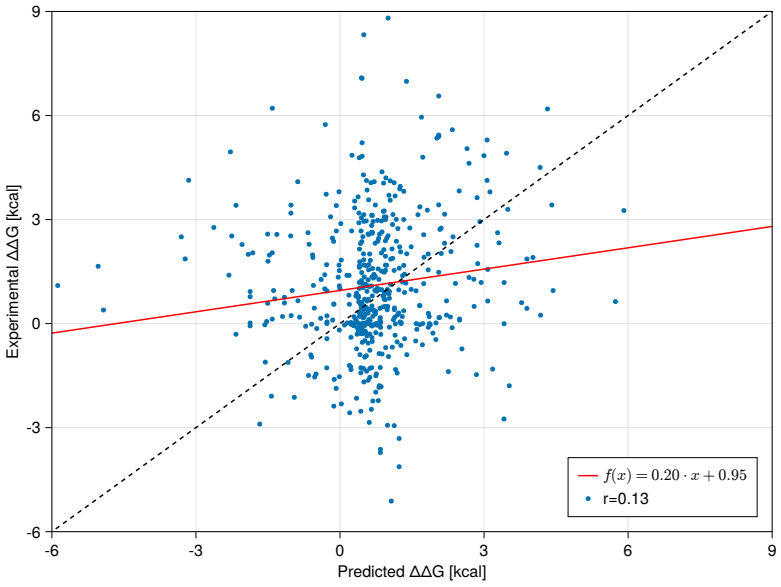


Figure 6.8: Predicted $\Delta\Delta G$ values (x-axis) compared to experimental $\Delta\Delta G$ values (y-axis) after learning on a smaller subset of the data. The red line is a linear fit on the data points. The black dashed line is the y=x line, if the model could perfectly predict the $\Delta\Delta G$ values, every point would be on this line.

6.6 Comparison to other methods

Comparing the Pearson Correlation Coefficient (PCC) of our model on the validation set (Table 6.2) with state of the art methods we can conclude that it outperforms most of the classical force-filed and statistical mechanics based methods, but there are still more sophisticated state of the art AI methods that can easily outperform this model on unseen data.

Method	PCC
Hom-ML-V2	0.857
TopNetTree	0.850
ProBindNN (our model)	0.810
Hom-ML-V1	0.792
BindProfX	0.738
Profile-score + FoldX	0.738
Profile-score	0.675
SAAMBE	0.624
FoldX	0.47
BeAtMuSic	0.272
Dcomplex	0.056

Table 6.2: Comparison of the performance of our model and other models on the SKEMPI v.2.0 data set. *Reproduced from* [1]

Conclusion

As we have seen in the discussion of our results, graph neural networks are really powerful tools that allow us to predict molecular properties only knowing fundamental quantities and structures of the molecules. We started from a simple idea, comparing two slightly different molecules extracted from a wild-type and a mutant protein, and tried to predict the corresponding $\Delta\Delta G$ value with two multi-layer GNNs and an MLP. This simple and scalable idea led us to the result, that Graph Neural Networks are in deed a powerful analysis tool with a bright future in structural biology, physics and many more application areas.

In this last section some points will be mentioned on which improvements could or should be made, regarding the procedure described in the previous sections.

Right at the creation of the data set, PyMol was used to create the mutants, later this should be replaced with a software or python package that performs additional backbone optimization. Although there are software packages like FoldX, or PyRosetta, but their python API interface is poor or does not even exist, so it is hard to use these in connection to deep learning. Crucial are also the atomic feature vectors we used. In this work they only contained atom type, electronegativity and the number of valence electrons, which was enough for qualitatively right predictions, but generally more features are required to get more accurate predictions. Another point which was also crucial, was computational time in both creating the data set, and training the model. That is, we made the assumption that only the nearest neighbors of the mutated amino acid affect the changes in the binding affinity. With more available time and computational power, the whole protein structures and interfaces could be analyzed using the network proposed previously. Finally, as can be seen in [42], the applied method in this thesis is just one of many in a large ensemble of methods, of which many could be considered as an alternative approach. Other works may use an attention mechanism [43] or add more spatial features [44] to the data they learn from.

Appendix A - Code

8.1 make_dataset.py

This section contains the function used to make the dataset based off an index file containing the mutations and ddG values. The package used to create graphs is graphein [41].

```
import os
import time

from tqdm import tqdm
from IPython.display import clear_output

import pandas as pd
import torch
from torch_geometric.data import Data
from biopandas.pdb import PandasPdb



from graphein.protein.config import ProteinGraphConfig
from graphein.protein.graphs import construct_graph
from graphein.ml.conversion import GraphFormatConvertor
from graphein.protein.edges.atomic import add_bond_order,
    ↪ add_ring_status
from graphein.protein.edges.distance import
    ↪ add_hydrogen_bond_interactions, add_ionic_interactions,
    ↪ add_peptide_bonds



from utils import *
from mutagenesis import create_mutations_pymol
from new_atomic import add_atomic_edges


```

```python
def make_dataset(index_xlsx: str,root:str, pdb_dir):

    raw_dir = os.path.join(root, "raw")
    processed_dir = os.path.join(root, "processed")

    index_df = pd.read_excel(index_xlsx, converters={"pdb_id":str.lower,
    ↪  "mut_pdb":str.strip,"mut_id":str.strip,"aff_mut":float,
    ↪  "aff_wt":float,"temp":float,  "ddg":float,})


    print("Mutations...")

    start = time.time()

    create_mutations_pymol(index_df=index_df, pdb_dir=pdb_dir,
    ↪  raw_dir=raw_dir)

    end = time.time()

    delta =(end-start)
    print("Time needed for creating the mutations: {}s".format(delta))

    params_to_change = {"granularity": "atom",
    ↪  "edge_construction_functions": [add_atomic_edges,
    ↪  add_bond_order, add_ring_status, add_hydrogen_bond_interactions,
    ↪  add_ionic_interactions, add_peptide_bonds]}

    config = ProteinGraphConfig(**params_to_change)

    format_convertor = GraphFormatConvertor('nx', 'pyg',
    ↪  verbose="default")

    for index, row in tqdm(index_df.iterrows()):
        if not
        ↪  (os.path.isfile(os.path.join(processed_dir,"mutated",str(index)+".pt"
        ↪  )) or
        ↪  os.path.isfile(os.path.join(processed_dir,"mutated",str(index)+".pt"
        ↪  ))):

            pdb_id = row["pdb_id"].split('_')[0]
            mutation = row["mut_id"]

            wildtype, chain_id, resid, mut_target = mutation[1],
            ↪  mutation[0], mutation[2:-1], mutation[-1]


            ddg = row["ddg"]
            file_mut = pdb_id + "_" + mutation

```

```
66
67
68              file_mut = pdb_id + "_" + mutation
69
70
71              print("Protein: ", pdb_id, "Mutation: ", mutation)
72
73              pdb_mutated = pdb_to_df(file_mut, raw_dir)
74              pdb_non_mutated = pdb_to_df(pdb_id, pdb_dir)
75
76              path_interface_mutated = os.path.join(raw_dir, "temp",
                ↪  str(index)+"_mutated_interface.pdb")
77              path_interface_non_mutated = os.path.join(raw_dir,"temp",
                ↪  str(index)+"_non_mutated_interface.pdb")
78
79              print("Extracting relevant residues")
80              print(chain_id, resid)
81              if not os.path.isfile(path_interface_mutated):
82
83                  interface_mutated = find_relevant(chain_id, int(resid),
                    ↪  pdb_mutated,cutout=5, cutoff=12.)
84                  interface_non_mutated = find_relevant(chain_id,
                    ↪  int(resid), pdb_non_mutated,cutout=2,cutoff = 12.)
85                  save_to_pdb(interface_mutated, path_interface_mutated)
86                  save_to_pdb(interface_non_mutated,
                    ↪  path_interface_non_mutated)
87
88
89              graph_mutated =
                ↪  construct_graph(config=config,pdb_path=path_interface_mutated)
90              graph_non_mutated =
                ↪  construct_graph(config=config,pdb_path=path_interface_non_mutated)
91
92              pyg_graph_mutated = format_convertor(graph_mutated)
93              pyg_graph_non_mutated = format_convertor(graph_non_mutated)
94
95              pyg_graph_mutated.y = ddg
96              pyg_graph_mutated.coords =
                ↪  torch.FloatTensor(pyg_graph_mutated.coords[0])
97
98              pyg_graph_non_mutated.y = ddg
99              pyg_graph_non_mutated.coords =
                ↪  torch.FloatTensor(pyg_graph_non_mutated.coords[0])
100
101
102
103             if pyg_graph_non_mutated.coords.shape[0] ==
                ↪  len(pyg_graph_non_mutated.node_id):
104
```

```
                    mut = Data(x=node_id_to_feature_matrix(graph_mutated),
                    ↪  edge_index=pyg_graph_mutated.edge_index,edge_weights=edge_weights
                    ↪  ddg=ddg)
                    non_mut =
                    ↪  Data(x=node_id_to_feature_matrix(graph_non_mutated),
                    ↪  edge_index=pyg_graph_non_mutated.edge_index,
                    ↪  edge_weights=edge_weights(graph_non_mutated))

                    torch.save(mut, os.path.join(processed_dir,"mutated",
                    ↪  str(index)+".pt"))
                    torch.save(non_mut,
                    ↪  os.path.join(processed_dir,"non_mutated",str(index)+".pt"))

                else:
                    raise ValueError

                clear_output(wait=True)

if __name__ == "__main__":
    dataset_dir = "abbind_dataset"
    if not os.path.exists(dataset_dir):
        os.mkdir(dataset_dir)
        os.mkdir(os.path.join(dataset_dir,"raw"))
        os.mkdir(os.path.join(dataset_dir, "raw/temp"))
        os.mkdir(os.path.join(dataset_dir,"processed"))
        os.mkdir(os.path.join(dataset_dir,"processed/mutated"))
        os.mkdir(os.path.join(dataset_dir, "processed/non_mutated"))
    if not os.path.exists("./index.xlsx"):
        print("Error, index.xlsx not found in ./")
    make_dataset(index_xlsx="AB_Bind/index.xlsx", root=dataset_dir,
    ↪  pdb_dir="AB_Bind/PDBs")
```

8.2 mutagenesis.py

This section contains the code to perform mutagenezis on a protein structure and/or multiple protein structures defined in an index.xlsx file using PyMol ([40]).

```
import os
import errno
import pymol
from tqdm import tqdm
from IPython.display import clear_output
import pandas as pd

from utils import AMINO_CODES, RAW_DATASET_DIR, PDB_DIR


```

```
12  def mutate_point(pdb_id: str, mutation_id: str, pdb_dir:str = PDB_DIR,
    ↪  destination_dir:str = RAW_DATASET_DIR):
13      """
14      Creates a pointmutation in a PDB Structure and saves it in the
    ↪  destionation folder.
15      pdb_id: str
16          PDB Id of the protein
17      mutation: str
18          3-digit code of the mutation
19      pdb_dir: str
20          Directory containing the PDB files.
21      destination_dir: str
22          Destionation directory in which the mutated PDB structure will
    ↪  be saved.
23      """
24
25      wildtype, chain_id, resid, mut = mutation_id[0], mutation_id[1],
        ↪  mutation_id[2:-1], mutation_id[-1]
26
27      selection = "chain "+ chain_id + " and residue " + resid
28
29      pdb_file = os.path.join(os.getcwd(), pdb_dir, pdb_id.lower() +
        ↪  ".pdb")
30      print("Selector:",selection, "---------------------------")
31
32
33      if os.path.isfile(pdb_file):
34
35          print("Starting  mutation wizard...")
36          mutant = AMINO_CODES[mut]
37
38
39
40          pymol.cmd.wizard("mutagenesis")
41          pymol.cmd.load(pdb_file)
42          pymol.cmd.refresh_wizard()
43          print("Selecting ", selection )
44          pymol.cmd.get_wizard().do_select(selection)
45          print("Mutant:", mutant)
46          pymol.cmd.get_wizard().set_mode(mutant)
47
48          pymol.cmd.get_wizard().apply()
49          pymol.cmd.set_wizard()
50
51          save_path = os.path.join(destination_dir, pdb_id + "_" +
            ↪  mutation_id + ".pdb")
52
53          pymol.cmd.save(save_path, format="pdb")
54          pymol.cmd.delete(name="all")
```

```
55          pymol.cmd.refresh_wizard()
56          pymol.cmd.refresh()
57
58
59      else:
60          raise FileNotFoundError(errno.ENOENT, os.strerror(errno.ENOENT),
            ↪  pdb_file)
61
62
63
64
65  def mutate_multiple_points(pdb_id:str, mutations:list,
    ↪  pdb_dir:str=PDB_DIR, destination_dir: str=RAW_DATASET_DIR) -> None:
66      """
67      Mutates multiple residues in a protein
68
69      pdb_id: str
70          The 4 letter PDB ID of the protein.
71      mutations: list
72          A list caontaining the IDs of the mutations.
73      pdb_dir: str
74          Directory of the raw PDB Files.
75      destination_dit: str
76          The mutated structures will be saved here.
77      """
78      pdb_file = os.path.join(os.getcwd(), pdb_dir, pdb_id.lower() +
        ↪  ".pdb")
79      save_path = os.path.join(destination_dir, pdb_id)
80
81      if os.path.isfile(pdb_file):
82
83          for mutation in mutations:
84
85              selection = mutation[0] + "/" + mutation[1:-1]  + "/"
86              mutant = AMINO_CODES[mutation[-1]]
87              pymol.finish_launching()
88              pymol.cmd.wizard("mutagenesis")
89              pymol.cmd.load(pdb_file)
90              pymol.cmd.refresh_wizard()
91              pymol.cmd.get_wizard().do_select(selection)
92              pymol.cmd.get_wizard().set_mode(mutant)
93              pymol.cmd.get_wizard().apply()
94              pymol.cmd.set_wizard()
95              pymol.cmd.deselect()
96              save_path += "_" + mutation
97
98          save_path += ".pdb"
99          pymol.cmd.save(save_path, format="pdb")
100         pymol.cmd.delete(name="all")
```

```
101
102
103     else:
104         raise FileNotFoundError(errno.ENOENT, os.strerror(errno.ENOENT),
            ↪  pdb_file)
105
106 def create_mutations_pymol(index_df:pd.DataFrame, pdb_dir, raw_dir:str):
107
108     for _, row in tqdm(index_df.iterrows()):
109
110
111         pdb_id = row["pdb_id"]
112         mutation = row["mut_id"]
113         pdb_id = row["pdb_id"].split('_')[0]
114         wildtype, chain_id, resid, mut_target = mutation[0],
            ↪  mutation[1], mutation[2:-1], mutation[-1]
115         filepath = os.path.join(raw_dir, pdb_id+ "_" + mutation +".pdb")
116
117         if  not os.path.isfile(filepath):
118             print(pdb_id, mut_target)
119             mutate_point(pdb_id, row["mut_id"],pdb_dir, raw_dir)
120         clear_output(wait=True)
```

8.3 utils.py

This section contains utilities which do not fit into any of the previous sections. These are for example constants, file I/O and download, and the data extraction algorithms used in make_dataset.py to extract the relevant part of the molecule, the feature vectors and last but not least the edge lengths.

```
1  import os
2  import re
3  import shutil
4  import torch
5  from torch.nn.functional import one_hot
6  from tqdm import tqdm
7  import pandas as pd
8  import numpy as np
9  from IPython.display import clear_output
10 from biopandas.pdb import PandasPdb
11 import errno
12 from mendeleev import element
13
14 AMINO_CODES = {
15     'A': 'Ala', 'R': 'Arg', 'N': 'Asn', 'D': 'Asp', 'C': 'Cys', 'Q':
       ↪  'Gln', 'E': 'Glu',
16     'G': 'Gly', 'H': 'His', 'I': 'Ile', 'L': 'Leu', 'K': 'Lys', 'M':
       ↪  'Met', 'F': 'Phe',
```

```
    'P': 'Pro', 'O': 'Pyl', 'S': 'Ser', 'U': 'Sec', 'T': 'Thr', 'W':
    ↪  'Trp', 'Y': 'Tyr',
    'V': 'Val', 'B': 'Asx', 'Z': 'Glx', 'X': 'Xaa', 'J': 'Xle'}

STD_AMINO_CODES = {
    'A': 'Ala', 'R': 'Arg', 'N': 'Asn', 'D': 'Asp', 'C': 'Cys', 'Q':
    ↪  'Gln', 'E': 'Glu',
    'G': 'Gly', 'H': 'His', 'I': 'Ile', 'L': 'Leu', 'K': 'Lys', 'M':
    ↪  'Met', 'F': 'Phe',
    'P': 'Pro', 'S': 'Ser', 'T': 'Thr', 'W': 'Trp', 'Y': 'Tyr',
    'V': 'Val'}

AMINO_CODES_R = dict((v.upper(), k) for k,v in AMINO_CODES.items())
COORDINATE_NAMES = ["x_coord", "y_coord", "z_coord"]

ELEMENTS = {'C':0, 'N':1, 'O':2, 'S':3, 'F':4, 'P':5, 'Cl':6, 'B':7,
↪  'H':8}
AMINO_ACIDS = dict((v.upper(), i) for i, (k, v) in
↪  enumerate(AMINO_CODES.items()))
CHAIN_IDS = {'A': 0, 'B': 1, 'C': 2, 'D': 3, 'E': 4, 'F': 5, 'G': 6,
↪  'H': 7, 'I': 8, 'J': 9, 'K': 10, 'L': 11, 'M': 12, 'N': 13, 'O': 14,
↪  'P': 15, 'Q': 16, 'R': 17, 'S': 18, 'T': 19, 'U':20, 'V':21, 'W':22,
↪  'X':23, 'Y':24, 'Z':25}

PDB_DIR = "PDBs"
RAW_DATASET_DIR="cancer_dataset/raw"
```

```
def find_relevant(chid:str, res_n:int,structure: pd.DataFrame,
↪  cutoff:float = 30., cutout:int = 5)->pd.DataFrame:
    """
    Extracts the relevant residues from a trcuture.
    chid: str
        Chain ID of the chain where the mutation occurs
    res_n: int
        Residue number of the residue which mutates
    structure: pd.DataFrame
        The DataFrame of the structure
    cutoff: float
        Maximal cutoff distance from the central mutated residue
    cutout: int
        The number of neighboring residues taken into account
    """

    base_res = structure.loc[structure["chain_id"] == chid]
    base_res = base_res.loc[base_res["residue_number"] == res_n]

    #center of mass for the mutated residue
    base_pos = np.array([sum(base_res["x_coord"].to_list()),
                         sum(base_res["y_coord"].to_list()),
```

```
                        ↪  sum(base_res["z_coord"].to_list())])/len(base_res)

    relevant = []

    #center of mass of every other residue
    for chain in structure["chain_id"].unique():
        curr_chain = structure.loc[structure["chain_id"]==chain]
        for res in curr_chain["residue_number"].unique():
            residue = curr_chain.loc[curr_chain["residue_number"]==res]

            res_pos = np.array([sum(residue["x_coord"].to_list()),
                                sum(residue["y_coord"].to_list()),

                                ↪  sum(residue["z_coord"].to_list())])/len(base_res)

            dist = np.linalg.norm(res_pos-base_pos)
            if dist<cutoff:
                relevant.append([chain, res, dist])

    dist_df = pd.DataFrame(relevant, columns=["chain_id",
    ↪  "residue_number", "distance"])

    middle = []

    for chain in dist_df.chain_id.unique():
        mid = dist_df.loc[dist_df["chain_id"]==chain]
        mid = mid.loc[mid["distance"] == mid.distance.min()]
        middle.append([mid["chain_id"].values[0],
        ↪  mid["residue_number"].values[0]])

    rel = pd.DataFrame(middle, columns=["chain_id", "residue_number"])

    parts = []
    for index, row in rel.iterrows():
        start = row["residue_number"]-cutout
        end = row["residue_number"]+cutout

        while start<0:
            start+=1
        while end>structure["residue_number"].max():
            end-=1

        for i in range(start, end+1):
            parts.append(structure.loc[structure["chain_id"] ==
            ↪  row["chain_id"]].loc[structure["residue_number"] == i])

    return pd.concat(parts)
```

```python
def node_id_to_feature_matrix(x):
    features = []
    for node in x.nodes:
        chain, res, res_num, atom = node.split(":")
        atom = element(atom[0])
        elem = x.nodes[node]
        feature = torch.concat([one_hot(torch.tensor(ELEMENTS[atom.symbol]), num_classes=9),
                                torch.tensor([atom.electronegativity()]),
                                torch.tensor([atom.nvalence()])])
        features.append(feature.unsqueeze(0))

    return torch.concat(features)


def edge_weights(x):


    nodes = {x:i for i, x in enumerate(x.nodes)}
    w = torch.zeros(len(x.edges))
    for k, (u, v, a) in enumerate(x.edges(data = True)):
        i, j = nodes[u], nodes[v]
        w[k] = a["distance"]
    return w


def pdb_to_df(pdb_id:str, root:str)->pd.DataFrame:
    """
    Opens a PDB file as a Dataframe.
    id: str
        Identificator of the pdb file without the .pdb extension
    root: str
        Root folder of the PDB File
    """
    path = os.path.join(root, pdb_id+".pdb")

    if os.path.isfile(path):
        return PandasPdb().read_pdb(path).df["ATOM"]
    else:
        raise FileNotFoundError(errno.ENOENT, os.strerror(errno.ENOENT), path)


def save_to_pdb(structure:pd.DataFrame, path:str)->None:
    """
    Save extracted segments to a pdb file
    """
    pdb_saver = PandasPdb()
```

```
    pdb_saver.df["ATOM"] = structure
    pdb_saver.to_pdb(path, records = ["ATOM"])


def fetch_pdbs(pdbids: list, pdb_dir: str, force=False)->None:
    """
    Downloades the PDB Files from the PDB.
    """
    for pdbid in tqdm(pdbids):
        if pdbid is None or pdbid=="-":
            print("Skipping empty pdbid")
            continue
        print("Downloading ", pdbid)
        destination_dir = os.path.join(pdb_dir, pdbid+".pdb")
        if not os.path.isfile(destination_dir) or force:
            pdb = PandasPdb().fetch_pdb(pdbid, source="pdb")
            pdb.to_pdb(destination_dir, records=["ATOM"])
            print("Successfully donwoaded")
            clear_output(wait=True)
        else:
            clear_output(wait=True)
```

8.4 new_atomic.py

The following code is a chained version of the add_atomic_edges of the graphein package [41]. The change creates virtual edges between seperate chains in a protein complex at the interfaces.

```


from typing import Any, Dict

import networkx as nx
import numpy as np
import pandas as pd

from graphein.protein.edges.distance import compute_distmat
from graphein.protein.resi_atoms import (
    BOND_LENGTHS,
    BOND_ORDERS,
    COVALENT_RADII,
    DEFAULT_BOND_STATE,
    RESIDUE_ATOM_BOND_STATE,
)

from graphein.protein.edges.atomic import
↪  assign_bond_states_to_dataframe, assign_covalent_radii_to_dataframe
def add_atomic_edges(G: nx.Graph, tolerance: float = 0.56) -> nx.Graph:
    """
```

```
    Computes covalent edges based on atomic distances. Covalent radii are assigned to each atom based on its bond assign_bond_states_to_dataframe
    The distance matrix is then thresholded to entries less than this distance plus some tolerance to create an adjacency matrix.
    This adjacency matrix is then parsed into an edge list and covalent edges added

    :param G: Atomic graph (nodes correspond to atoms) to populate with atomic bonds as edges
    :type G: nx.Graph
    :param tolerance: Tolerance for atomic distance. Default is ``0.56`` Angstroms. Commonly used values are: ``0.4, 0.45, 0.56``
    :type tolerance: float
    :return: Atomic graph with edges between bonded atoms added
    :rtype: nx.Graph
    """
    dist_mat = compute_distmat(G.graph["pdb_df"])

    # We assign bond states to the dataframe, and then map these to covalent radii
    G.graph["pdb_df"] = assign_bond_states_to_dataframe(G.graph["pdb_df"])
    G.graph["pdb_df"] = assign_covalent_radii_to_dataframe(G.graph["pdb_df"])

    # Create a covalent 'distance' matrix by adding the radius arrays with its transpose
    covalent_radius_distance_matrix = np.add(
        np.array(G.graph["pdb_df"]["covalent_radius"]).reshape(-1, 1),
        np.array(G.graph["pdb_df"]["covalent_radius"]).reshape(1, -1),
    )

    # Add the tolerance
    covalent_radius_distance_matrix = (
        covalent_radius_distance_matrix + tolerance
    )

    # Threshold Distance Matrix to entries where the eucl distance is less than the covalent radius plus tolerance and larger than 0.4
    dist_mat = dist_mat[dist_mat > 0.4]
    t_distmat = dist_mat[dist_mat < covalent_radius_distance_matrix]









    dist_mat_longrange = dist_mat[dist_mat > 0.4]
    t_distmat_longrange = dist_mat[dist_mat < 12.]
```

```
59
60      # Store atomic adjacency matrix in graph
61      G.graph["atomic_adj_mat"] = np.nan_to_num(t_distmat)
62
63      # Get node IDs from non NaN entries in the thresholded distance
        ↪  matrix and add the edge to the graph
64      inds = zip(*np.where(~np.isnan(t_distmat)))
65
66
67
68      for i in inds:
69          length = t_distmat[i[0]][i[1]]
70          node_1 = G.graph["pdb_df"]["node_id"][i[0]]
71          node_2 = G.graph["pdb_df"]["node_id"][i[1]]
72          chain_1 = G.graph["pdb_df"]["chain_id"][i[0]]
73          chain_2 = G.graph["pdb_df"]["chain_id"][i[1]]
74
75          # Check nodes are in graph
76          if not (G.has_node(node_1) and G.has_node(node_2)):
77              continue
78
79          # Check atoms are in the same chain
80          if not (chain_1 and chain_2):
81              continue
82
83          if G.has_edge(node_1, node_2):
84              G.edges[node_1, node_2]["kind"].add("covalent")
85              G.edges[node_1, node_2]["bond_length"] = length
86          else:
87              G.add_edge(node_1, node_2, kind={"covalent"},
                ↪  bond_length=length)
88      #Creation of virtual edges
89      inds = zip(*np.where(~np.isnan(t_distmat_longrange)))
90      for i in inds:
91          length = t_distmat[i[0]][i[1]]
92          node_1 = G.graph["pdb_df"]["node_id"][i[0]]
93          node_2 = G.graph["pdb_df"]["node_id"][i[1]]
94          chain_1 = G.graph["pdb_df"]["chain_id"][i[0]]
95          chain_2 = G.graph["pdb_df"]["chain_id"][i[1]]
96
97          # Check nodes are in graph
98          if not (G.has_node(node_1) and G.has_node(node_2)):
99              continue
100
101         # Check atoms are in the same chain
102         if chain_1 != chain_2:
103
104             if G.has_edge(node_1, node_2):
105                 G.edges[node_1, node_2]["kind"].add("long")
```

```
106                 G.edges[node_1, node_2]["bond_length"] = length
107             else:
108                 G.add_edge(node_1, node_2, kind={"long"},
                    ↪  bond_length=length)
109
110     # Todo checking degree against MAX_NEIGHBOURS
111
112     return G
```

8.5 model.py

This section contains the implementation of an MLP and of the proposed Graph Neural Network used in this thesis. They are implemented in PyTorch ([45]) and Torch Geometric ([46]).

```
import torch
import torch.nn.functional as F
import torch_geometric
from torch_geometric.nn import GatedGraphConv
from torch import nn
from torch_geometric.nn.norm import GraphNorm, LayerNorm
from torch_geometric.utils.dropout import dropout_adj
from torch_geometric.nn import global_mean_pool, global_add_pool,
↪  global_max_poo

class MLP(nn.Module):
    """
    A Multi-Layer Perceptron model.
    """
    def __init__(self, input_dim, hidden_dims, out_dim):
        super(MLP, self).__init__()

        self.mlp = nn.Sequential()
        dims = [input_dim] + hidden_dims + [out_dim]
        for i in range(len(dims)-1):

            ↪  self.mlp.add_module('lay_{}'.format(i),nn.Linear(in_features=dims[i],
            ↪  out_features=dims[i+1]))
            #self.mlp.add_module("dropout_{}".format(i),
            ↪  nn.Dropout(0.1))
            if i+2 < len(dims):
                self.mlp.add_module('act_{}'.format(i), nn.LeakyReLU())
    def reset_parameters(self):
        for i, l in enumerate(self.mlp):
            if type(l) == nn.Linear:
                nn.init.xavier_normal_(l.weight)

    def forward(self, x):
        return self.mlp(x)
```

```python
class ProBindNN(torch.nn.Module):
    """
    The neural network designed for ddG prediction.
    config: dict
        The configuration dictionary with relevant
        parameters for the underlying architecture.
        features_in: int
            Size of a feature vector
        layers: int
            Passes of the GRU in each GNN
        gnn_features_out: int
            The feature vectors will be padded to this value,
            it has to be greater than features_in
        out_dim: int
            The dimension of the values that we want to predict.
            In our case its just the ddG (1), but it can be extended for other descriptors.
        mlp_hidden_dim: List[int]
            A list of length of the layers in the MLP with does the final predictions,
            integers in the list are the number of neurons for each layer.
    """
    def __init__(self, config={"features_in":15, "layers":30, "gnn_features_out":15, "out_dim":1, "mlp_hidden_dim":[15, 15, 15]}):
        super(ProBindNN, self).__init__()

        self.GGNN_a = GatedGraphConv(config["features_in"], config["layers"])
        self.GGNN_b = GatedGraphConv(config["features_in"], config["layers"])
        self.mlp = MLP(config["gnn_features_out"], config["mlp_hidden_dim"], config["out_dim"])

    def forward(self, x, y):
        out_a = self.GGNN_a(x.x, x.edge_index, x.edge_weights)
        out_b = self.GGNN_b(y.x, y.edge_index, y.edge_weights)

        out_a = torch.nn.functional.leaky_relu(out_a)
        out_b = torch.nn.functional.leaky_relu(out_b)

        out_a = global_add_pool(out_a, x.batch)
        out_b = global_add_pool(out_b, y.batch)

        out =out_a-out_b
        out = self.mlp(out)
        return out
```

8.6 train.py

This section contains the implementation of the training alhorithm of the proposed deep learning model. For each training run all the necessary data of the process, like losses, and model parameters are saved into a tensorboard session.

```
1
2  from re import T
3  import sys
4  sys.path.append("./src")
5
6
7
8  from model import ProBindNN
9  from dataset import MutationDataset
10
11 from torch_geometric.loader import DataLoader
12 import torch
13 from torch import nn
14
15 from tqdm import tqdm
16 from IPython.display import clear_output
17 import copy
18 import os
19 import time
20 from datetime import datetime
21
22
23 from torch_geometric.loader import DataLoader
24 from torch.utils.tensorboard import SummaryWriter
25 from torch.optim.lr_scheduler import ExponentialLR
26 os.environ['KMP_DUPLICATE_LIB_OK']='True'
27
28
29
30 def train(model, loaders, optimizer, loss_fn, scheduler, n_epochs=1000):
31
32     device = "cuda" if torch.cuda.is_available() else "cpu"
33     print("Using {} device".format(device))
34
35     model.train()
36
37     t = time.time()
38     tstamp = datetime.utcfromtimestamp(t).strftime('%Y_%m_%d_%H_%M_%S')
39     path = "logs/tensorboard/RUN_{}".format(tstamp)
40     writer = SummaryWriter(path)
41
42     for epoch in tqdm(range(1, n_epochs)):
43         epoch_loss = 0.
44         best_loss = 1000.
```

```
45
46          for loader in loaders.keys():
47              if loader == "train_loader":
48                  model.train()
49
50                  for i, batch in enumerate(loaders[loader]):
51                      x, y = batch["mutated"].to(device),
                        ↪  batch["non_mutated"].to(device)
52                      ddg = x.ddg.to(device).squeeze()
53                      optimizer.zero_grad()
54                      out = model(x,y).squeeze()
55                      loss = loss_fn(out, ddg)
56
57                      loss.backward()
58                      optimizer.step()
59                      epoch_loss += loss.item()
60
61
62
63                  epoch_loss/=(len(loaders[loader]))
64
65                  writer.add_scalar("Loss/train", epoch_loss, epoch)
66                  print("Epoch: {}, Loss: {}".format(epoch, epoch_loss))
67              elif loader == "val_loader":
68
69                  model.eval()
70
71                  val_loss = 0
72
73                  for i, batch in enumerate(loaders[loader]):
74                      x, y = batch["mutated"].to(device),
                        ↪  batch["non_mutated"].to(device)
75                      ddg = x.ddg.to(device).squeeze()
76                      out = model(x,y).squeeze()
77                      loss = loss_fn(out, ddg)
78                      val_loss+=loss.item()
79
80                  val_loss /= len(loaders[loader])
81                  writer.add_scalar("Loss/val", val_loss, epoch)
82
83                  print("Validation loss:", val_loss)
84
85          if epoch_loss<best_loss:
86
87              best_model = copy.deepcopy(model)
88              t = time.time()
89              stamp =
                ↪  datetime.utcfromtimestamp(t).strftime('%Y_%m_%d_%H_%M_%S')
```

```
90              best_model_path =
                ↪  "models/aminos_model_lal{}.pt".format(tstamp, stamp)
91              torch.save(model.state_dict(), best_model_path)
92
93          else:
94
95              scheduler.step()
96      return best_model, best_model_path
97
98
99
100
101 if __name__ == "__main__":
102
103     if not os.path.exists("logs"):
104         os.mkdir("logs/nohup")
105         os.mkdir("logs/tensorboard")
106     if not os.path.exists("models"):
107         os.mkdir("models")
108
109     #Create dataset and dataloaders
110     dataset = MutationDataset(index_xlsx="index.xlsx",
        ↪  root="dataset12aa")
111     train_size = int(len(dataset)*0.9)
112     val_size = (len(dataset)-train_size)
113     train_dataset, val_dataset = torch.utils.data.random_split(dataset,
        ↪  [train_size, val_size])
114     train_loader = DataLoader(train_dataset, batch_size=32,
        ↪  shuffle=True)
115     val_loader = DataLoader(val_dataset, batch_size=128, shuffle=True)
116     loaders = {"val_loader": val_loader, "train_loader":train_loader}
117
118     device = "cuda" if torch.cuda.is_available() else "cpu"
119
120     config={"features_in":15, "layers":30, "gnn_features_out":15,
        ↪  "out_dim":1, "mlp_hidden_dim":[15, 15, 15, 15, 15]}
121
122     model = ProBindNN(config).to(device)
123     model = nn.DataParallel(model)
124
125     if os.path.isfile("pretrained_model.pt"):
126         model.load_state_dict(torch.load("pretrained_model.pt"))
127
128     print("Using {} device".format(device))
129
130     optimizer = torch.optim.Adam(model.parameters())
131     scheduler = scheduler = ExponentialLR(optimizer, gamma=0.9)
132
133     loss_fn =  nn.MSELoss()
```

```
    epochs = 3000
    d = torch.tensor(val_dataset.indices)
    torch.save(d, "val_indices_train double.pt")
    train(model, loaders, optimizer, loss_fn, scheduler, n_epochs=2000)


```

8.7 Further Remarks

All of the above code is available at https://github.com/babaid/ProBindNN .

The development of the took place on Linux in a Conda environment. To install all the dependencies run the following code in virtual environment in the terminal:

```
python -m pip install -r requirements.txt
```

With the requirements.txt provided in the repository.

The usage of the aboe code is as follows: For the creation of the dataset make_dataset.py can be used either as an import into a submodule, or directly calling

```
cd ProBindNN
nohup python src/make_dataset.py > logs/nohup/make_dataset.txt
```

The folder of the dataset will be automatically created. For training the same applies, one can either import the train method, or directly run it from the command line:

```
nohup python src/make_dataset.py > training.txt
```

The use of nohup is just an optional convenience for logging purposes. The automatically generated logs folder contains a tensorboard subfolder, where the training and validation losses are saved with a unique timestamp as an identifier. To look at the data, use the following command:

```
tensorboard --logdir logs/tensorboard
```

This will automatically launch Tensorboard and open a browser window where the training process can be observed.

Bibliography

[1] Xiang Liu et al. "Hom-Complex-Based Machine Learning (HCML) for the Prediction of Protein–Protein Binding Affinity Changes upon Mutation". In: *Journal of Chemical Information and Modeling* 62.17 (Sept. 2022). Publisher: American Chemical Society, pp. 3961–3969. ISSN: 1549-9596. DOI: 10.1021/acs.jcim.2c00580. URL: https://doi.org/10.1021/acs.jcim.2c00580 (visited on 09/13/2022).

[2] J. Schymkowitz et al. "The Foldx Web Server: An Online Force Field". In: *Nucleic Acids Res.* 33 (2005), W382.

[3] Peng Xiong et al. "BindProfX: Assessing Mutation-Induced Binding Affinity Change by Protein Interface Profiles with Pseudo-Counts". eng. In: *Journal of Molecular Biology* 429.3 (Feb. 2017), pp. 426–434. ISSN: 1089-8638. DOI: 10.1016/j.jmb.2016.11.022.

[4] Cunliang Geng et al. "iSEE: Interface structure, evolution, and energy-based machine learning predictor of binding affinity changes upon mutations". eng. In: *Proteins* 87.2 (Feb. 2019), pp. 110–119. ISSN: 1097-0134. DOI: 10.1002/prot.25630.

[5] Clément Viricel et al. "Cost function network-based design of protein–protein interactions: predicting changes in binding affinity". In: *Bioinformatics* 34.15 (Aug. 2018), pp. 2581–2589. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/bty092. URL: https://doi.org/10.1093/bioinformatics/bty092 (visited on 09/17/2022).

[6] *A physical reference state unifies the structure-derived potential of mean force for protein folding and binding - Liu - 2004 - Proteins: Structure, Function, and Bioinformatics - Wiley Online Library*. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/prot.20019 (visited on 09/17/2022).

[7] Tawny R. Gonzalez et al. "Assessment of software methods for estimating protein-protein relative binding affinities". en. In: *PLOS ONE* 15.12 (Dec. 2020). Publisher: Public Library of Science, e0240573. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0240573. URL: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0240573 (visited on 09/04/2022).

[8] *The Rosetta All-Atom Energy Function for Macromolecular Modeling and Design — Journal of Chemical Theory and Computation*. URL: https://pubs.acs.org/doi/10.1021/acs.jctc.7b00125 (visited on 09/17/2022).

[9] Thomas A. Halgren. "Merck molecular force field. I. Basis, form, scope, parameterization, and performance of MMFF94". en. In: *Journal of Computational Chemistry* 17.5-6 (1996). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/%28SICI%291096-987X%28199604%2917%3A5/6%3C490%3A%3AAID-JCC1%3E3.0.CO%3B2-P, pp. 490–519. ISSN: 1096-987X. DOI: 10.1002/(SICI)1096-987X(199604)17:5/6<490::AID-JCC1>3.0.CO;2-P. URL: https://onlinelibrary.wiley.com/doi/abs/

10.1002/%28SICI%291096-987X%28199604%2917%3A5/6%3C490%3A%3AAID-JCC1%3E3.0.CO%3B2-P (visited on 09/17/2022).

[10] Huali Cao et al. "DeepDDG: Predicting the Stability Change of Protein Point Mutations Using Neural Networks". In: *Journal of Chemical Information and Modeling* 59.4 (Apr. 2019). Publisher: American Chemical Society, pp. 1508–1514. ISSN: 1549-9596. DOI: 10.1021/acs.jcim.8b00697. URL: https://doi.org/10.1021/acs.jcim.8b00697.

[11] J. Cheng, A. Randall, and P. Baldi. "Prediction of Protein Stability Changes for Single-Site Mutations Using Support Vector Machines". In: *Proteins: Struct., Funct., Genet.* 62 (2006), p. 1125.

[12] Shuyu Wang et al. "ProS-GNN: Predicting effects of mutations on protein stability using graph neural networks". In: *bioRxiv* (Jan. 2021), p. 2021.10.25.465658. DOI: 10.1101/2021.10.25.465658. URL: http://biorxiv.org/content/early/2021/10/26/2021.10.25.465658.abstract.

[13] Lei Jia, Ramya Yarlagadda, and Charles C. Reed. "Structure Based Thermostability Prediction Models for Protein Single Point Mutations with Machine Learning Tools". en. In: *PLOS ONE* 10.9 (Sept. 2015). Ed. by Yang Zhang, e0138022. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0138022. URL: https://dx.plos.org/10.1371/journal.pone.0138022 (visited on 09/04/2022).

[14] Engelbert Buxbaum. *Fundamentals of Protein Structure and Function*. en. Cham: Springer International Publishing, 2015. ISBN: 978-3-319-19919-1 978-3-319-19920-7. DOI: 10.1007/978-3-319-19920-7. URL: http://link.springer.com/10.1007/978-3-319-19920-7 (visited on 09/10/2022).

[15] *Amino Acids – the Building Blocks of Proteins*. en. URL: http://www.technologynetworks.com/applied-sciences/articles/essential-amino-acids-chart-abbreviations-and-structure-324357 (visited on 09/10/2022).

[16] *Peptide Bonds*. en. URL: https://masteringcollegebiochemistry.wordpress.com/tag/peptide-bonds/ (visited on 09/10/2022).

[17] *PCNA - Proliferating cell nuclear antigen - Homo sapiens (Human) — UniProtKB — UniProt*. URL: https://www.uniprot.org/uniprotkb/P12004/entry (visited on 09/05/2022).

[18] *File:Protein structure (full).png - Wikipedia*. en. URL: https://commons.wikimedia.org/wiki/File:Protein_structure_(full).png (visited on 09/19/2022).

[19] M. Michael Gromiha. "Chapter 6 - Protein Stability". en. In: *Protein Bioinformatics*. Ed. by M. Michael Gromiha. Singapore: Academic Press, Jan. 2010, pp. 209–245. ISBN: 978-81-312-2297-3. DOI: 10.1016/B978-8-1312-2297-3.50006-0. URL: https://www.sciencedirect.com/science/article/pii/B9788131222973500060 (visited on 09/04/2022).

[20] Lene Clausen et al. "Protein stability and degradation in health and disease". eng. In: *Advances in Protein Chemistry and Structural Biology* 114 (2019), pp. 61–83. ISSN: 1876-1631. DOI: 10.1016/bs.apcsb.2018.09.002.

[21] Simon Haykin. *Neural Networks and Learning Machines*. Third. Hamilton, Ontario, Canada: Pearson Education, Inc., 2009. URL: http://dai.fmph.uniba.sk/courses/NN/haykin.neural-networks.3ed.2009.pdf.

[22] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". en. In: *Neural Networks* 2.5 (Jan. 1989), pp. 359–366. ISSN: 08936080. DOI: 10.1016/0893-6080(89)90020-8. URL: https://linkinghub.elsevier.com/retrieve/pii/0893608089900208 (visited on 09/10/2022).

[23] krishaitech. *Activation Functions In Deep Learning*. en-US. Feb. 2022. URL: https://krishnaik.in/2022/02/14/activation-function/ (visited on 09/06/2022).

[24] *Understanding LSTM Networks – colah's blog*. URL: https://colah.github.io/posts/2015-08-Understanding-LSTMs/ (visited on 09/02/2022).

[25] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: https://doi.org/10.1162/neco.1997.9.8.1735 (visited on 09/10/2022).

[26] Kaisheng Yao et al. "Depth-Gated Recurrent Neural Networks". In: (Aug. 2015).

[27] George Bird and Maxim E. Polivoda. *Backpropagation Through Time For Networks With Long-Term Dependencies*. arXiv:2103.15589 [cs]. Apr. 2021. DOI: 10.48550/arXiv.2103.15589. URL: http://arxiv.org/abs/2103.15589 (visited on 09/10/2022).

[28] Yao Ma and Jiliang Tang. *Deep Learning on Graphs*. Cambridge University Press, 2021. URL: https://web.njit.edu/~ym329/dlg_book/dlg_book.pdf.

[29] *An introduction to networks - Math Insight*. URL: https://mathinsight.org/network_introduction (visited on 09/13/2022).

[30] Michael M. Bronstein et al. "Geometric deep learning: going beyond Euclidean data". In: *IEEE Signal Processing Magazine* 34.4 (July 2017). arXiv:1611.08097 [cs], pp. 18–42. ISSN: 1053-5888, 1558-0792. DOI: 10.1109/MSP.2017.2693418. URL: http://arxiv.org/abs/1611.08097 (visited on 09/10/2022).

[31] Benjamin Sanchez-Lengeling et al. "A Gentle Introduction to Graph Neural Networks". en. In: *Distill* 6.9 (Sept. 2021), e33. ISSN: 2476-0757. DOI: 10.23915/distill.00033. URL: https://distill.pub/2021/gnn-intro (visited on 09/05/2022).

[32] Justin Gilmer et al. *Neural Message Passing for Quantum Chemistry*. arXiv:1704.01212 [cs]. June 2017. DOI: 10.48550/arXiv.1704.01212. URL: http://arxiv.org/abs/1704.01212 (visited on 09/13/2022).

[33] R. P. Feynman. "Forces in Molecules". In: *Physical Review* 56.4 (Aug. 1939). Publisher: American Physical Society, pp. 340–343. DOI: 10.1103/PhysRev.56.340. URL: https://link.aps.org/doi/10.1103/PhysRev.56.340 (visited on 09/05/2022).

[34] Fernando J. Pineda. "Generalization of back-propagation to recurrent neural networks". In: *Physical Review Letters* 59.19 (Nov. 1987). Publisher: American Physical Society, pp. 2229–2232. DOI: 10.1103/PhysRevLett.59.2229. URL: https://link.aps.org/doi/10.1103/PhysRevLett.59.2229 (visited on 09/05/2022).

[35] Yujia Li et al. *Gated Graph Sequence Neural Networks*. arXiv:1511.05493 [cs, stat]. Sept. 2017. DOI: 10.48550/arXiv.1511.05493. URL: http://arxiv.org/abs/1511.05493 (visited on 09/02/2022).

[36] Justina Jankauskaitė et al. "SKEMPI 2.0: an updated benchmark of changes in protein–protein binding energy, kinetics and thermodynamics upon mutation". en. In: *Bioinformatics* 35.3 (Feb. 2019). Ed. by Ioannis Xenarios, pp. 462–469. ISSN: 1367-4803, 1460-2059. DOI: 10.1093/bioinformatics/bty635. URL: https://academic.oup.com/bioinformatics/article/35/3/462/5055583 (visited on 09/14/2022).

[37] Xue Li et al. "SDNN-PPI: self-attention with deep neural network effect on protein-protein interaction prediction". In: *BMC Genomics* 23.1 (June 2022), p. 474. ISSN: 1471-2164. DOI: 10.1186/s12864-022-08687-2. URL: https://doi.org/10.1186/s12864-022-08687-2 (visited on 09/06/2022).

[38] Vladimir Gligorijević et al. "Structure-based protein function prediction using graph convolutional networks". en. In: *Nature Communications* 12.1 (May 2021). Number: 1 Publisher: Nature Publishing Group, p. 3168. ISSN: 2041-1723. DOI: 10.1038/s41467-021-23303-9. URL: https://www.nature.com/articles/s41467-021-23303-9 (visited on 09/06/2022).

[39] E. Capriotti, P. Fariselli, and R. Casadio. "A Neural-Network-Based Method for Predicting Protein Stability Changes Upon Single Point Mutations". In: *Bioinformatics* 20.Suppl 1 (2004), p. i63.

[40] *PyMOL — pymol.org*. URL: https://pymol.org/2/ (visited on 09/02/2022).

[41] Arian Jamasb. *a-r-j/graphein*. original-date: 2019-08-28T20:20:11Z. Sept. 2022. URL: https://github.com/a-r-j/graphein (visited on 09/06/2022).

[42] Jie Zhou et al. "Graph neural networks: A review of methods and applications". In: *AI Open* 1 (Jan. 2020), pp. 57–81. ISSN: 2666-6510. DOI: 10.1016/j.aiopen.2021.01.001. URL: https://www.sciencedirect.com/science/article/pii/S2666651021000012.

[43] Petar Veličković et al. *Graph Attention Networks*. arXiv:1710.10903 [cs, stat]. Feb. 2018. DOI: 10.48550/arXiv.1710.10903. URL: http://arxiv.org/abs/1710.10903 (visited on 09/12/2022).

[44] Tomasz Danel et al. *Spatial Graph Convolutional Networks*. arXiv:1909.05310 [cs, stat]. July 2020. URL: http://arxiv.org/abs/1909.05310 (visited on 09/12/2022).

[45] *PyTorch*. en. URL: https://www.pytorch.org (visited on 09/02/2022).

[46] *PyG Documentation — pytorch_geometric documentation*. URL: https://pytorch-geometric.readthedocs.io/en/latest/ (visited on 09/06/2022).