TrimNet: a new deep learning method for learning molecular representation from triplet messages

Greta Grassmann

Machine Learning Project, Prof. Andrea Asperti
Bologna University

ABSTRACT

TrimNet is a new Deep Learning graph-based approach used for molecular representation learning. It employs a triplet message mechanism to calculate message from atom-bond-atom information and update the hidden states of the neural network. *TrimNet* outperforms other current state-of-the-art methods on molecular property predictions (including quantum properties, bioactivity and physiology) and compound-protein interaction identification. It reduces the number of parameters and its learning process is easily interpretable, since it can be seen that the attention weights focus on atoms and substructures crucial for the target properties. For this project, its application on the BACE benchmark dataset has been studied: the task is in this case the classification of different molecules as binding (1) or not (0) to the human β-secretase 1, an enzyme than when inhibited slows the Alzheimer's disease. Starting from the original *TrimNet* code, different implementation has been tested and compared. In particular, this project presents the effect of changing, compared to the original ones:

· The *loss function*.

· The number T of blocks implemented in the message phase.

· The number K of the attention mechanisms implemented in the multi-head triplet attention mechanism.

Keywords: Deep Learning, Graph structures, Message passing, Attention mechanism, Gated Recurrent Unit.

1 INTRODUCTION

AN OVERVIEW OF METHODS FOR MOLECULAR PROPERTY PREDICTION

Molecular property prediction and Compound-Protein Interaction (CPI) identification, which are of fundamental importance in fields like biomedicine (for example for drug discovery), are based on the learning of useful molecular representation.

Until recently molecular structures and properties were calculated by quantum mechanics[1] or predicted by traditional machine learning methods. The former employed tremendous computational resources and needed a high computational time, while the latter, although shortening the in silico prediction of molecular properties, required expert knowledge and were labor-intensive (since machine learning methods rely on manually extracted molecular features).

Lately the performance of molecular properties prediction has been boosted by the implementation of Deep Learning (DL) methods. DL is a class of machine learning algorithms that uses multiple layers to progressively extract higher-level features from the raw input. These methods can be divided into *sequence-based methods* and *graph-based methods*, depending on how the molecule is represented.

Sequence-based methods are used with the molecular sequence representations and usually employ Convolutional Neural Networks (CNNs).

[1] Molecular quantum mechanics, also called quantum chemistry, is a branch of chemistry focused on the application of quantum mechanics to chemical systems: electronic structure and molecular dynamics are understood using the Schrödinger equations.

CNNs: These networks are able to extract multi-scale localized spatial features and compose them to construct highly expressive representation, but they can only operate on data which have a "regular data structure" (like images, which are a grid of pixel values).

This means that when the data has non-Euclidean distance these approaches are unfit for analysis. These kind of data can be dealt with by implementing *graph-based methods*, which are used with molecular graphs and employ Graph Neural Networks (GNNs).
GNNs: GNNs are a type of neural network which directly operates on the graph structure and is based on an information diffusion mechanism. This means that compared to *sequence-based methods*, GNNs learn the representation over the molecular structure directly by capturing non-Euclidean information and performing features transformations based on graphs. In a simplified approach, each node of the graph is described by a set of features. The network uses the features from each node, as well as that of its neighbouring ones, to infer a state embedding which contains the information of the neighborhood of each node. In a more complex approach complete approach this information propagation between nodes (or *message passing*[2]), which is generally guided by the connections alone, takes into account also the features of the edges along which it happens. This happens also in the new DL method described in Section 2.

Recently two structures of GNNs have been in particular studied to learn representation from graph data: Graph Convolutional Networks (GCNs) and Message Passing Neural Networks (MPNNs).
GCNs: GCNs are a subclass of GNNs that generalize the convolutional approach of CNNs to the graph domain, by using convolutional operators to aggregate the information from the neighborhood of a node. That is, they learn to integrate node features based on labels and link structures, by generalizing the standard notion of convolution over a regular grid to convolution over a graph structure.
MPNNs: MPNN framework standardizes different message passing models, all of which share *message*, *update* and *readout functions*. Each node in the graph has a hidden state (or feature vector), and for each node the network aggregates a function of hidden states and possibly edges of all the neighbouring nodes. Then, it updates the hidden state of that node using the obtained message and the previous hidden state of that node. This message passing algorithm is repeated for a specified number of times, after which the final readout phase is reached. In this step, the newly updated hidden states are extracted, and a final feature vector describing the whole graph is created.

Attention mechanisms can be applied to further improve their performances, by computing representations based on the element importance score. Attention is a component that can be added to the network's architecture to manage and quantify the interdependence between the input and output elements (*general attention*) or within the input elements (*self-attention*). To the more important elements an higher weight is assigned.

But this approaches present some difficulties too, such as large-scale parameters (coming from the matrix in which the edges' features usually have to be mapped) and insufficient bonds information extraction (since they only aggregate the neighboring nodes' information to update the representation of the center node). Especially the latter spoils the performance, since bonds contain information about the molecular scaffolds and conformers.
Because of this, a new DL graph-based approach named *triplet message networks (TrimNet)* has been proposed in (1), which achieves the state-of-the-art performances with a significant reduction of the number of parameters and improves the extraction of the edges' information.

2 TRIMNET

TrimNet reduces significantly the number of parameters by dropping the matrix mapping of the edges' features. It employs a novel triplet message mechanism to learn molecular representation efficiently and complete multiple molecular representation learning tasks. This mechanism is employed to calculate message from atom-bond-atom information and update the hidden states of neural networks. In (1) *TrimNet* has shown to be able to outperform previous state-of-the-art method on various datasets in the

[2]Generalizing the convolution operator to irregular domains is typically expressed as a *neighborhood aggregation* or *message passing scheme.*

learning of quantum properties, bioactivity, physiology and CPI prediction. *TrimNet* could also provide a clear interpretation of the prediction task because it focuses on the atoms and the scaffolds essential to the target properties.

2.1 Dataset

As already anticipated, *TrimNet* has been evaluated for four different molecular properties: quantum properties, bioactivity, physiology and CPI. This model can be applied to many different benchmark datasets, depending on the task. For each one of them the molecules have to be preprocessed into graphs with node features, edge features and adjacency matrix, as shown in Figure 2, using RDKIT (6), a collection of cheminformatics and machine learning software written in C++ and Python.For this study the evaluations are performed only on the data resulting from the preprocessing of the BACE benchmark dataset, which can be downloaded from the site of MoleculeNet.
BACE is a classification dataset that provides quantitative (IC_{50}) and qualitative (binary label) binding results for a set of inhibitors of human $\beta-$secretase 1 (BACE-1). BACE-1 is a long sought potential therapeutic target for Alzheimer's: drugs to block this enzyme in theory would prevent the buildup of beta-amyloid and (per the Amyloid hypothesis) may help slow or stop Alzheimer's disease. The binding affinities were modeled using multiple in silico ligand based modeling approaches and statistical techniques (10).
The data are Simplified Molecular-Input Line-Entry System (SMILES[3]) strings.
The mentioned preprocessing consists in the generation of *Murcko scaffolds* (or *Bemis-Murcko scaffolds*) from the SMILES string.

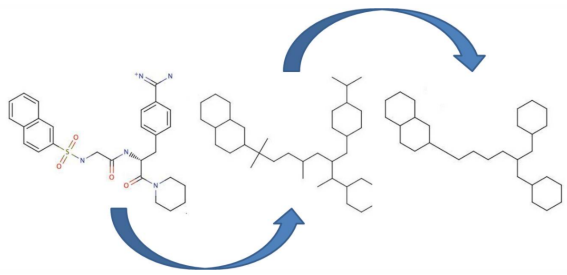

Figure 1. Bemis-Murcko framework generation.

Murcko scaffolds are essentially that part of the molecule consisting of rings and the linker atoms between them, derived by removing side chain atoms as shown in Figure 1. As already said, a molecular framework can be interpreted as a graph containing nodes and edges representing atom and bond types, respectively. Removing atom and bond labels or agglomerating nodes by chemotype yields a hierarchy of reduced graphs, or molecular equivalence classes, that represent sets of related molecules. Likewise, a framework can be further decomposed into individual rings (or the core ring assembly) using chemically intuitive rules: the rings can individually or jointly be considered as scaffolds derived from the original compound.
After the generation of the scaffolds, they are randomly split for the training, validation and testing: the dataset that is going to be studied in this project is composed by 1513 scaffolds, divided in 1216 for the training, 143 for the validation[4] and 154 for the testing.
Each atom has 39 features, while each edge has 10 of them. Some examples of the extracted node and edge features are the ones give in (11), summarized in the following Table:

[3]The SMILES is a specification in the form of a line notation for describing the structure of chemical species using short ASCII strings.
[4]Validation is used to select proper parameters of the system and is part of the training set.

node features	edge features
atom symbol	bond type
number of covalent bonds	whether the bond is conjugated
electrical charge	whether the bond is in a ring
number of radical electrons	stereo
whether the atom is part of an aromatic system	
hybridization	
chirality type	
whether the atom is a chiral center	
number of connected hydrogens	

The preprocessing ends with *One-Hot Encoding*, the process applied to categorical data[5] to convert it into a binary vector array, and with the computation of a set of weights that depend on the ratio between positive and negative training samples.

2.2 Architecture

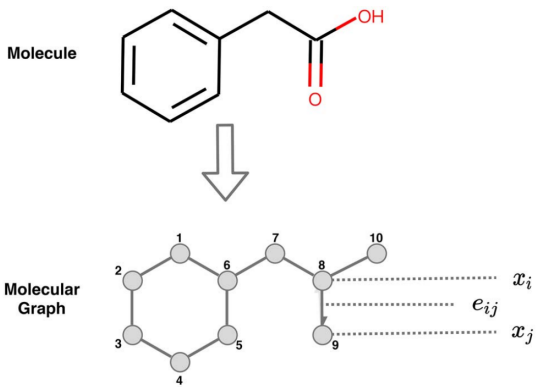

Figure 2. Example of a molecule (top) with its corresponding molecular graph (bottom).

As shown in Figure 2 a molecule can be represented by a graph structure G. A graph G is a pair $(Nodes, Edges)$, where $Nodes$ is the set of nodes and $Edges$ is the set of edges; the nodes represent objects or concepts (in our case the atoms of the molecule), and the edges the relationships among them.
G consists then of atom features h_i and bond features e_{ij}. At each time step t we are going to call the node features $h^t = h_0^t, h_1^t, ..., h_N^t$ and the edges features e^t.

The prediction on this graph structure is performed by feeding it into the *TrimNet* architecture, depicted in Figure 3.

[5] Categorical data are variables that can take on one of a limited number of possible values.

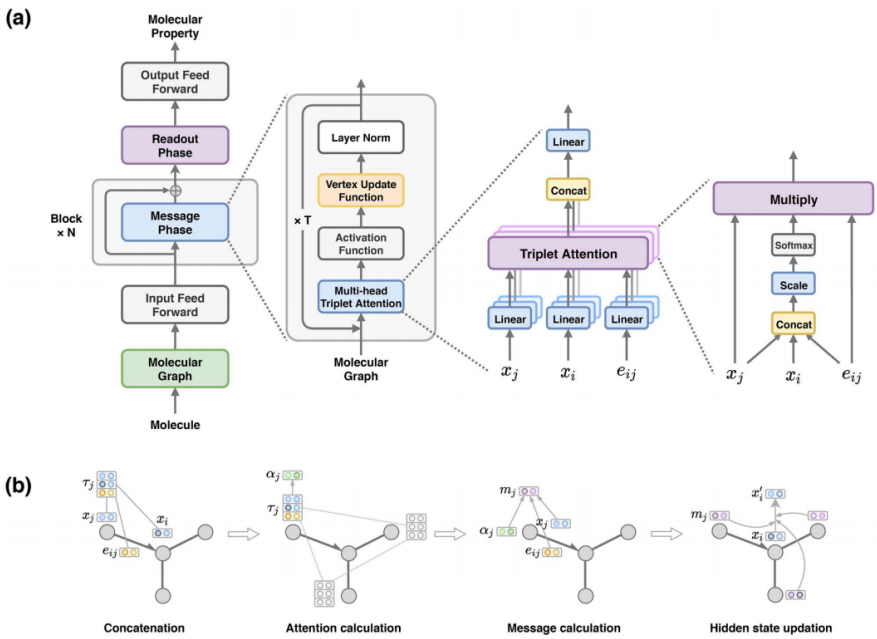

Figure 3. *TrimNet* architecture.

TrimNet operates in two phases: the *message phase* and the *readout phase*.
The *message phase* contains three components that work sequentially: a *message calculation function* M_t based on a multi-head triplet attention mechanism, a *vertex update function* U_t and a layer normalization. These components update the hidden state h_i^t at each node at each time step t, so that:

$$m_i^{t+1} = \sum_{j \in N_i} M_t(h_i^t, h_j^t, e_{ik}),$$
$$h_i^{t+1} = LN\left(U_t(h_i^t, m_i^{t+1})\right),$$

where N_i is the set of neighbors of node i, e_{ij} is the edge between the node i and j and LN is the layer normalization.
The *readout phase*, better described in Section 2.2.2, takes the final representation derived from the *message phase* and computes the feature vector for the graph by leveraging the *readout function* R:

$$\hat{y} = R(h_i^t | i \in G). \tag{1}$$

2.2.1 Message phase

Message calculation function: multi-head attention

To improve the performance and reduce the computational time, a triplet-attentive edge network is used as the *message calculation function* M_t. This triplet-attentive edge network computes the *attention score* by a multi-head triplet attention mechanism and aggregates the neighboring nodes' and edges' information according to the attention.
Given the node features $h^t = h_0^t, h_1^t, ..., h_N^t$ and the edges features e^t it concatenates them into a triplet. This triplet is then fed into a feed-forward neural network according to:

$$\tau_{i,j}^{t+1} = LeakyReLU\left(u^T [W_h h_i^t || W_e e_{ij}^t || W_h h_j^t]\right), \tag{2}$$

where $||$ represents concatenation, W_h is the learnable weight matrix shared across all nodes , $W_e$is the learnable weight matrix shared across all edges and u is another learnable weight. *LeakyReLU* stands for the LeakyReLU nonlinear function.
To facilitate the comparisons of coefficients across different nodes, the coefficient themselves are normal-

ized across all choices of neighbor j using the softmax function:

$$\begin{aligned}\alpha_{ij}^{t+1} &= softmax(\tau_{ij}^{t+1}) \\ &= \frac{e^{\tau_{ij}^{t+1}}}{\sum_{j\in N_i} e^{\tau_{ij}^{t+1}}},\end{aligned} \tag{3}$$

Next, the normalized attention coefficients, the neighboring nodes hidden states and the edges hidden states are used to derive the message m_i at each node, by performing a weighted summation operation:

$$m_i^{t+1} = \sum_{j\in N_i} \alpha_{ij}^{t+1} \odot W_h h_j^t \odot W_e e_{ij}^t, \tag{4}$$

where $\odot$ represents the element-wise multiplication.As already anticipated, this network employs multi-head attention, a module for attention mechanism which runs through an attention mechanism several times in parallel. The independent attention outputs are then concatenated and linearly transformed into the expected dimension.
This means that in order to stabilize the learning process of self-attention, K independent attention mechanism execute the transformation of Equation 4 and than their features are concatenated. This results in the following output feature representation:

$$m_i^{t+1} = ||_k^K \sum_{j\in N_i} \alpha_{ij}^{t+1,k} \odot W_h^k h_j^t \odot W_e^k e_{ij}^t, \tag{5}$$

where $||$ represents again concatenation. $\alpha^{t+1,k}$ are the normalized attention coefficients computed by the k-th attention mechanism and W_e^k is the corresponding weight matrix of input linear transformation.
By employing the triplet attention and element-wise multiplication *TrimNet* decreases effectively the parameters. Indeed the many parameters of other methods mainly come from their *message function*, in which the edge vector needs to be mapped to a $D \times D$ matrix, where D is the hidden size; *TrimNet* on the other hand does not need this matrix mapping.

Vertex update function: Gated Recurrent Unit

Next, *TrimNet* employs a Gated Recurrent Unit (GRU) (2) as the *vertex update function* U_t, which fuses the previously extracted message and the current integrated features.
GRUs are a variant of Recurrent Neural Networks (RNNs), which are an extension of conventional feedforward neural networks that can handle variable-length sequence input. RNNs handle the variable-length sequence by having a recurrent hidden state whose activation at each time step is dependent on that of the previous time. Training RNNs to capture long-term dependencies can be difficult because of the vanishing (or exploding) gradient problem, and because of the effect of long-term dependencies is hidden by the effect of short-term dependencies.
To solve this problem one can design a more sophisticated activation function, by using gating units. GRUs were proposed to make each recurrent unit to adaptively capture dependencies of different scale. Compared to traditional recurrent units, which replace the activation (or the content of a unit) with a new value computed from the current input and the previous hidden state, GRUs keep the existing content and add the new one on top of it. In this way, each unit can remember the existence of a specific feature in the input stream for a long series of steps: no important feature will be overwritten.
Other than its internal gating mechanisms, the GRU functions just like an RNN, where sequential input data is consumed by the GRU cell at each time step along with the memory, or otherwise known as the hidden state.The hidden state is then re-fed into the RNN cell together with the next input data in the sequence. This process continues like a relay system, producing the desired output.
The mechanism behind it can be broken down into three main steps (h indicates the hidden activation, while x is the input):

- Reset gate. This gate is derived and calculated using both the hidden state from the previous time step and the input data at the current time step. This is achieved by multiplying the previous hidden state and current input with their respective weights and summing them before passing the sum through a sigmoid function. The sigmoid function will transform the values to fall between 0 and

1, allowing the gate to filter between the less-important and more-important information in the subsequent steps.

$$gate_{reset} = \sigma(W_{input_{reset}} \cdot x_t + W_{hidden_{reset}} \cdot h_{t-1}) \tag{6}$$

When the entire network is trained through back-propagation, the weights in the equation will be updated such that the vector will learn to retain only the useful features.
The previous hidden state will first be multiplied by a trainable weight and will then undergo an element-wise multiplication (Hadamard product) with the reset vector. This operation will decide which information is to be kept from the previous time steps together with the new inputs. At the same time, the current input will also be multiplied by a trainable weight before being summed with the product of the reset vector and previous hidden state above. Lastly, a non-linear activation tanh function will be applied to the final result to obtain r in the equation below.

$$r = tanh(gate_{reset} \odot (W_{h_t t-1}) + W_{x_t} \cdot x_t) \tag{7}$$

- Next, we'll have to create the Update gate. Just like the Reset gate, the gate is computed using the previous hidden state and current input data.
  Both the Update and Reset gate vectors are created using the same formula, but, the weights multiplied with the input and hidden state are unique to each gate, which means that the final vectors for each gate are different. This allows the gates to serve their specific purposes.

$$gate_{update} = \sigma(W_{input_{update}} \cdot x_t + W_{hidden_{update}} \cdot h_{t-1}) \tag{8}$$

  The Update vector will then undergo element-wise multiplication with the previous hidden state to obtain u in our equation below, which will be used to compute our final output later.

$$u = gate_{update} \odot h_{t-1} \tag{9}$$

  The Update vector will also be used in another operation later when obtaining our final output. The purpose of the Update gate here is to help the model determine how much of the past information stored in the previous hidden state needs to be retained for the future.

- In the last step, we will be reusing the Update gate and obtaining the updated hidden state.
  This time, we will be taking the element-wise inverse version of the same Update vector (1 - Update gate) and doing an element-wise multiplication with our output from the Reset gate, r. The purpose of this operation is for the Update gate to determine what portion of the new information should be stored in the hidden state. Lastly, the result from the above operations will be summed with our output from the Update gate in the previous step, u. This will give us our new and updated hidden state.

$$h_t = r \odot (1 - gate_{update}) + u \tag{10}$$

  We can use this new hidden state as our output for that time step as well by passing it through a linear activation layer.

As we have seen in the mechanisms above, the Reset gate is responsible for deciding which portions of the previous hidden state are to be combined with the current input to propose a new hidden state.
And the Update gate is responsible for determining how much of the previous hidden state is to be retained and what portion of the new proposed hidden state (derived from the Reset gate) is to be added to the final hidden state. When the Update gate is first multiplied with the previous hidden state, the network is picking which parts of the previous hidden state it is going to keep in its memory while discarding the rest. Subsequently, it is patching up the missing parts of information when it uses the inverse of the Update gate to filter the proposed new hidden state from the Reset gate.

Layer normalization

At the end of each time step in which the GRU is applied, a layer normalization (3) is applied to the

output features from the last layer of the GRU. This is done to stabilizes the hidden state dynamics for the recurrent network:

$$h_i^{t+1} = LN\big(GRU(h_i^t, m_i^{t+1})\big) \tag{11}$$

h_i^{t+1} and h_i^t have the same dimension.
Layer normalization estimates the normalization statistics from the summed inputs to the units within a hidden layer on a single training case (at each time step) so that, compared for example to batch normalization, it does not introduce any new dependencies between training cases.
Layer normalization performs over the inputs the following operation:

$$y = \frac{x - E[x]}{\sqrt{Var[x] + \varepsilon}}\gamma + \beta, \tag{12}$$

where x is the input and γ and β are learnable weights and bias. The normalization of the activities also reduces the training time.

2.2.2 Readout phase: the Set2Set network

The *readout phase* starts from the final updated nodes' features $h^T = h_0^T, h_1^T, ..., h_N^T$. The *Set2Set* network (4) is used as the *readout function* to produce a graph-level embedding. In particular, *Set2Set* aggregates nodes features by different attention weights and concatenate the aggregated features with history information.
This approach has the property that the final vector would not change is the memory was randomly shuffled, which in our case means if h_i and h_j were permuted. Compared to other sequence-to-sequence methods, it is also suitable for cases in which the data are not naturally organized as a sequence, but nevertheless a non natural order which yields better performances can be found thanks indeed to *Set2Set*.
It starts by initializing two null tensors p and q^* of shape $[layers, batch\ size, input\ channels]$, $[layers, batch\ size, input\ channels]$ and $[batch\ size, output\ channels]$, where the $output\ channels$ are two times the $input channels$.
Then it performs the following computations for T' times recurrently to obtain the final graph representation $q^*_{T'}$:

$$\begin{aligned} q_t, p_t &= LSTM(q^*_{t-1}, p_{t-1}) \\ \alpha_{i,t} &= softmax(h_i^T q_t) \\ r_t &= \sum_{i=1}^{N} \alpha_{i,t} h_i^T \\ q^*_t &= q_t || r_t. \end{aligned}$$

q_t is a *query vector* which allows us to read r_t from the memories h^T and $LSTM$ is a LSTM which computes a recurrent state.
The final graph representation $q^*_{T'}$ is a tensor of shape $[batchsize, output\ channels]$.

Long Short-Term Memory Unit

The LSTM is a RNN architecture. RNNs process inputs fed as sequences.
Unlike standard feedforward neural networks, LSTM has feedback connections and can process entire sequences of data. It was developed to deal with the vanishing gradient problem that can be encountered when training traditional RNNs.
Each layer performs the following operations, for each element in the input sequence:

$$\begin{aligned} i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\ f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\ g_t &= tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\ o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\ c_t &= f_t \odot c_{t-1} + i_t \odot g - t \\ h_t &= o_t \odot tanh(c_t), \end{aligned} \tag{13}$$

where h_t is the hidden state at time t, c_t is the cell state at time t, i_t is the input gate, f_t is the forget gate, g_t is the cell gate and o_t is the output gate.

Interpretabilty: Atomic attention in this *Set2Set* network can provide interpetabilty from input molecules to output properties, or in other words it can help us to understand how *TrimNet* arrives at a successful prediction.
To do so, we can extract the atomic attention in the *Set2Set* to visualize some molecules. To be precise, the atom corresponding to the maximum weight can be detected according to the attention, and then the atom itself and its first-level neighbors (on the molecule graph drawn with RDKIT), which contribute significantly to the message passing mechanism, are highlighted.
From this visualization it can be seen that the *TrimNet* model detects the essential atomic groups, the one that determine the molecule's properties.

Finally, the derived representation q_T^* is fed into a feed-forward neural network, which gives as output the final prediction $\hat{y}$.

2.2.3 Loss function

Given this final final prediction $\hat{y}$ and the target labels y, the training objective is to minimize the *loss function*. Depending on the kind of properties that have to be predicted, different kind of *loss function* can be employed.
In (1) the chosen *loss function* for the classification task on the BACE dataset is the *Cross Entropy Loss function*, which is useful when training a classification problem with C classes (for this dataset $C = 2$).
In particular, it tries to handle the class imbalance problem, which has two important consequences. The first one is that training is inefficient as most examples are easy negatives (zero labeled), in the sense that they can be easily classified by the detector, and do not contribute to useful learning. The second one is that since easy negatives (detections with high probabilities) account for a large portion of the inputs, although they result in small loss values individually, collectively can overwhelm the loss and computed gradients and lead to degenerate models.
Given the final prediction of dimension $[batch\ size, C]$, the target labels of dimension $[batch\ size]$ and a a tensor of weights of dimension $[C]$ (given in this case by $[\frac{positive\ examples + negative\ examples}{negative\ examples}, \frac{positive\ examples + negative\ examples}{positive\ examples}]$), it computes:

$$L = \frac{\sum_{i=1}^{batch\ size} loss(i, \hat{y}[i])}{\sum_{i=1}^{batch\ size} weight[\hat{y}[i]]}, \tag{14}$$

where

$$loss(i, \hat{y}[i]) = weight[\hat{y}[i]] \left(-i[\hat{y}[i]] + log\big(\sum_j exp(i[j])\big) \right). \tag{15}$$

The idea behind *Cross Entropy Loss* is to penalize the wrong predictions more that to reward the right predictions.

2.3 Training and hyperparameters

TrimNet is trained via the standard batch gradient descent method with the error back-propagation algorithm.
To update the parameters the optimization algorithm *Adam* (8), developed to face high-dimensional parameters spaces, is used. *Adam* is an algorithm for first-order gradient-based optimization for stochastic objective functions, based on adaptive estimates of lower-order moments: it computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients.
Given an objective function $f(\theta)$, which obviously has to be differentiable with respect to the parameters θ, we want to minimize its value with respect to its parameters. Its realisations at subsequent time steps are called $f_i(\theta), f_2(\theta), ..., f_T(\theta)$, and $g_t = \nabla_\theta f_t(\theta)$ is the gradient. m_t is the estimate of the first moment (the mean) and the second moment (the variance) of the gradient, while v_t is the squared gradient; both are initialized as a vectors of zeros. g_t^2 corresponds to $g_t \odot g_t$. α is the stepsize, $\beta_1, \beta_2 \in [0, 1)$ are the

exponential decay rates for the moment estimates.
The *Adam* algorithm can be schematized as follows:

Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1^{st} moment vector)
$v_0 \leftarrow 0$ (Initialize 2^{nd} moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

All the operations on vectors are element-wise.

To prevent the potential overfitting problem two regularization techniques are employed: the dropout and the weight decay.

Dropout

Dropout is a stochastic regularization technique which reduces overfitting by combining many different neural network architectures: in theory, the training process works by temporarily removing random units from a neural network (together with their connections). This process repeats every epochs or every minibatch, and should lower generalization error rates (the overfitting) as the presence of neurons is made unreliable.
Since averaging the predictions of these many "thinned" models would be a too big computational burden, another process in executed in practice: only one neural network is used, where the weights outputs are scaled down randomly.
But why does dropout reduce overfitting? To train neural networks, given a loss-value the gradients are computed and then the optimizer (like the gradient descent) processes it into the network's weights. Computing the gradient is done with respect to the error, but also with respect to what all other units are doing: this means that certain neurons may fix the mistakes of other neurons. This could lead to a co-adaptations that may not generalize to unseen data. The dropout prevents these co-adaptations, because neurons can now not rely on other units to correct their mistakes.

Weight decay

Weight decay is another regularization technique used to reduce overfitting. After each update,it prevents the weights from growing too large and keeps under control the complexity of the model.
To do this, the squares of all the parameters are added to the loss function (adding their value would not work, since some are positive and some negative) after being multiplied by the *weight decay*. The *weight decay* is a small number which prevents the loss to become too big. Then the weights are updated with the gradient descent.

To obtain the optimal hyperparameters, a grid search procedures was applied in (1). The hyperparameters tuning process involved the learning rate, the hidden size, the dropout rate, the number of attention heads and the number of iteration for the message phase.

3 IMPLEMENTATION

The original software (9) was provided by the authors of (1). It includes five different codes written in Python: **model.py**, **utils.py**, **trainer.py**, **dataset.py** and **run.py**. As already anticipated, the only dataset that has been considered in this study is the BACE one. Because of this, the original code (9) has been "cleaned": all the lines not useful for this type of data have been deleted, to obtain a more manageable and

readable code.
Two additional small codes were written to better visualize the results of the *TrimNet* learning by plotting as a function of the number of epochs the loss, the area under the ROC curve[6] and the learning rate.
The resulting work can be found in this github repository, and is described in the following Sections.

3.1 Software structure

The model structure is summarized in the following:

```
TrimNet(
        (lin0): Linear(in_features=39, out_features=32, bias=True)
        (convs): ModuleList(
                        for i in range (0,N):
                                        (i): Block(
                                                        for i in range (0,T):
                                                                        (conv): MultiHeadTripletAttention(32, 32, heads=4)
                                                                        (gru): GRU(32, 32)
                                                                        (ln): LayerNorm((32,), eps=1e-05, elementwise_affine=True)
                                                        )
                        )
        (set2set): Set2Set(32, 64)
        (out): Sequential(
                        (0): Linear(in_features=64, out_features=512, bias=True)
                        (1): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
                        (2): ReLU(inplace=True)
                        (3): Dropout(p=0.2, inplace=False)
                        (4): Linear(in_features=512, out_features=2, bias=True)
                        )
 )
```

In the following Sections there is an in-depth description of how this strucutre is achieved.

3.1.1 model.py

This code defines three classes (or object constructors): *MultiHeadTripletAttention*, *Block* and *TrimNet*. *TrimNet*, corresponding to our studied model, contains *Block*, which in turn contains *MultiHeadTripletAttention*. These classes are defined in the following:

MultiHeadTripletAttention Class: This class defines the message passing used in *TrimNet*, which is regulated by a multi-head triplet attention mechanism. The operations here defined are the ones described in the first paragraph of Section 2.2.1. All the there mentioned weight matrices (W_h, W_e, u) are initialized with the *kaiming initialization* (7). This initialization method avoids both the vanishing gradient problem (not too small weights) and exploding gradient problem (not too big weights). The resulting tensor will have values sampled with a uniform distribution from $(bound, bound)$, where $bound = gain \times \sqrt{\frac{3}{input\ dimension}}$.
The initial call to start propagating messages is then defined.
The next function constructs messages to each node from each edge by computing the attention coefficients and concatenating them. The resulting tensor is multiplied with u, and then a *LeakyReLU* and a *softmax function* are applied. This results in Equation 13.
At the end of all the operations Equation 5 is obtained.
Finally, the function that determines the nodes embedding update is defined.

Block Class: This class defines the operations described in the second paragraph of Section 2.2.1, the one regarding the vertex update function.
Inside a loop on the time steps, it uses the preceding *MultiHeadTripletAttention class* to define the convolution for the message passing and introduces the GRU for the vertex update.
In particular, the forward function defined in the *MultiHeadTripletAttention class* is used as argument for the *CELU activation function*, defined as:

$$CELU(x) = max(0,x) + min\big(0, \alpha(e^{\frac{x}{\alpha}} - 1)\big), \tag{16}$$

with $\alpha = 1$.
The result is given as input sequence for the GRU.
Next a layer normalization (as described in the third paragraph of Section 2.2.1) is applied, so that Equation 11 is obtained.

[6] The are under the ROC curve quantifies how often the model obtained at fixed epochs predicts true positives or false positives.

TrimNet Class: This last class defines the *TrimNet* model. It includes a *forward function* that starts from a linear transformation of the tensor with all the atoms' features in the batch, that goes from the shape $[atoms\ in\ the\ batch, edge\ features]$ to $[atoms\ in\ the\ batch, node\ channels]$.
Then it applies a *CELU* function to this tensor.
Next it defines in a cycle the depth of the message phase block's structure (which includes the *Multi-head triplet attention mechanism*, the GRU and the layer normalization).
At each step the dropout is performed: during the training randomly zeroes some of the elements of the input tensor with probability p using samples from a Bernoulli distribution.
The final tensor with the atom's features is given by the sum of that output and the current integrated features (that is, the initial tensor with the atoms' features taken as input before the block structure -at the inital step- or the output of the just ended block layer).
Next it feed it to a *Set2Set network*, and the same operation of randomly zeroing some elements is repeated.
The resulting tensor is fed to a sequential container, which includes in the order: a linear transformation, a layer normalization, a ReLU transformation, another random dropout operation and a final linear transformation.

3.1.2 utils.py

This code defines some of the functions and classes used in the other codes, described in the following:

FocalLoss Class: Focal loss is extremely useful for classification when there are highly imbalanced classes. It down-weights well-classified examples and focuses on hard examples. The loss value is much high for a sample which is misclassified by the classifier as compared to the loss value corresponding to a well-classified example.
This loss is calculated as:

$$loss = -\alpha(1-p)^{\gamma}\big(target \cdot log(p)\big) - (1-\alpha)p^{\gamma}\big((1-target)log(1-p)\big), \tag{17}$$

where $\alpha = \frac{negative\ examples}{positive\ examples + negative\ examples}$, $\gamma = 2$ and $target$ are the true labels. p is obtained by applying the *Softmax function* to the input (the tensor of the predictions of the model, of dimension $[batch\ size, 2]$) and taking only the first column.
Softmax is a mathematical function that converts a vector of numbers into a vector of probabilities, where the probabilities of each value are proportional to the relative scale of each value in the vector. It works as an activation function in a neural network model: the network is configured to output C values, one for each class in the classification task, and the *Softmax function* is used to normalize the outputs, converting them from weighted sum values into probabilities that sum to one. Each value in the output is interpreted as the probability of membership for each class.
The mean of these *Focal losses* computed for all the examples in a batch is given as output.

3.1.3 trainer.py

This code starts by defining what kind of *loss function* is used. As already said, *Adam* is used as optimization algorithm and after this optimizer's update a learning rate scheduling is applied too. This is done to reduce the learning rate when a metric has stopped improving. Models often benefit from reducing the learning rate by a factor of 2-10 once learning stagnates. This scheduler reads a metrics quantity and if no improvement is seen for a 'patience' number of epochs (in our case 10), the learning rate is reduced. Indeed with a constant learning rate, the learning rate would need to be small so that weights and biases would slowly get better: a big learning rate would change weights and biases too much and training would fail, but a small learning rate made training very slow.
Then it declares the application of the model for a certain number of epochs for the training, testing and validation. In particular the following functions are defined:

trainiterations Function: This function gives as output the loss, the area under the Receiver Operating Characteristic (ROC) curve and the precision[7] for the training set.
For each batch (including 128 molecules), this function starts by clearing old gradients from the last step.

[7] By precision, we mean the area under the precision-recall curve, given by the ratio $\frac{tp}{tp+fp}$, where tp is the number of true positives and fp the number of false positives. The precision is intuitively the ability of the classifier not to label as positive a sample that is negative.

Then, for each task (in our case, there is only one task, that is the classification of the BACE dataset), it computes the prediction obtained with the model defined in **model.py**.
Starting from these predictions it computes the loss, then computes the derivative of the loss with respect to the parameters using backpropagation and causes the optimizer to take a step based on the gradients of the parameters.
The loss, the area under the Receiver Operating Characteristic (ROC) curve and the precision are saved.

validiterations Function: This function gives as output the loss, the area under the ROC curve and the precision for the validation and testing sets.
In these cases the forward behaviour of some components of the model can be switched off, and the gradient calculation can be disabled too.
The predictions are computed and then the loss, the area under the Receiver Operating Characteristic (ROC) curve and the precision are saved.

train Function: For each number of epochs this function starts by calculating the loss, the area under the ROC curve and the precision for the training, validation and testing sets (with **trainiterations.py** and **validiteration.py**).
Then the learning rate scheduler is applied.
If the area under the ROC curve or the precision for the validation set obtained with n number of epochs are bigger then the ones obtained with $n-1$ number of epochs, the corresponding model is saved.
For the final number of epochs, the final model is saved.

3.1.4 dataset.py

This code upload the preprocessed data from the BACE dataset.

3.1.5 run.py

This is the code that has to be run to implement the model.
It starts by simply defining what arguments will be required (like the learning rate, the batches dimension, the number of epochs, the data, and so on) and than figuring out how to parse those out of the list of command line arguments passed to the script, or taking the default values written in the code itself.
As a next step it takes the selected dataset for the training, the validation and the testing.
Then it implements the model on these data and opens a folder in which all the related results will be saved. This folder will in particular contain:

- A logbook (Notepad file) with all the parameters defining the model and its implementation. It also contains for each number of epochs the loss, the area under the ROC curve, the precision, the learning rate and the elapsed time for training, validation and testing sets.

- A Microsoft Excel table that records, for each number of epochs of training and validation, the loss, the area under the ROC curve and the learning rate.

- A ckpt file that stores the best model's parameters.

3.2 Comparison between the original code and new modifications

3.2.1 Original TrimNet code

Implementing this code on the values obtained from the original code with all the original parameters selected by the authors results in the following plots:

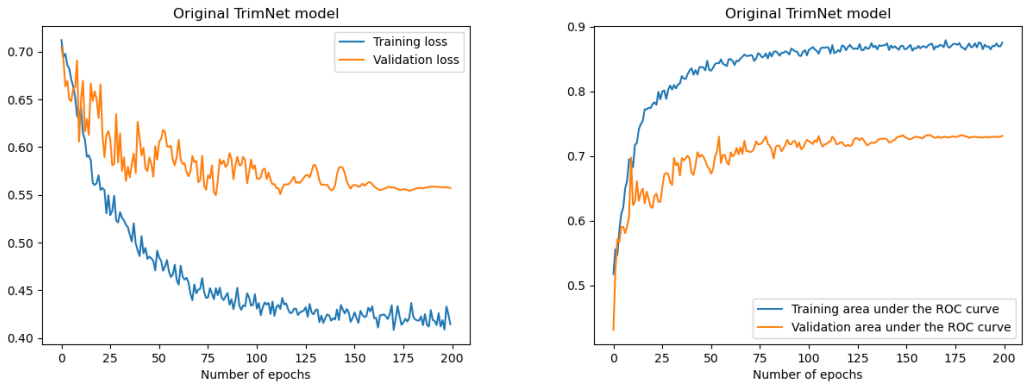

Figure 4. Left: loss for increasing number of epochs for the training and validation on the BACE dataset of the original *TrimNet* model. Right: area under the ROC curve for increasing number of epochs for the training and validation on the BACE dataset of the original *TrimNet* model.

The final best model has the following values:

Testing of the best original *TrimNet* model		
loss	area under the ROC curve	precision
0.765	0.744	0.665

3.2.2 Changing the loss function

As already anticipated, in (9) the chosen loss function is the *Cross Entropy*. But for bioactivity and physiology prediction tasks, like the one implemented in this project, a better choice could be the *weighted Focal Loss function*, that can deal with the data imbalance problem by improving *Cross Entropy*. Compared to it, *Focal Loss* reduces the contribution from easy examples and increase the importance of correcting misclassified examples and in this way can better handle the class imbalance problem: while *Cross Entropy* does a good job in differentiating positive and negative classes correctly, in can not differentiate between easy and hard examples.
Its implementation is described in Section 3.1.2. Figure 5 shows the training and validation loss values and are area under the ROC curve with $\alpha = \frac{negative\ samples + positive\ samples}{negative\ samples}$ and $\gamma = 2$.

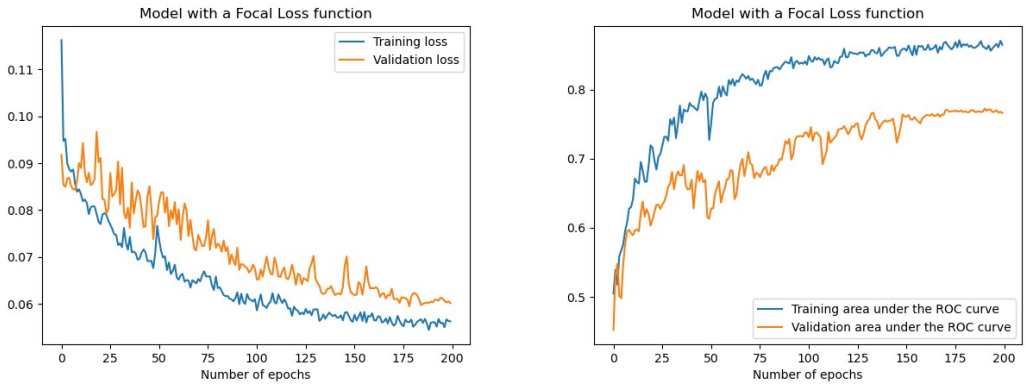

Figure 5. Left: loss for increasing number of epochs for the training and validation on the BACE dataset of the model with a Focal Loss function. Right: area under the ROC curve for increasing number of epochs for the training and validation on the BACE dataset of the model with a Focal Loss function

Whereas, Figure 6 shows a direct comparison between the results obtained with the original *TrimNet* model and the one with a Focal Loss function.

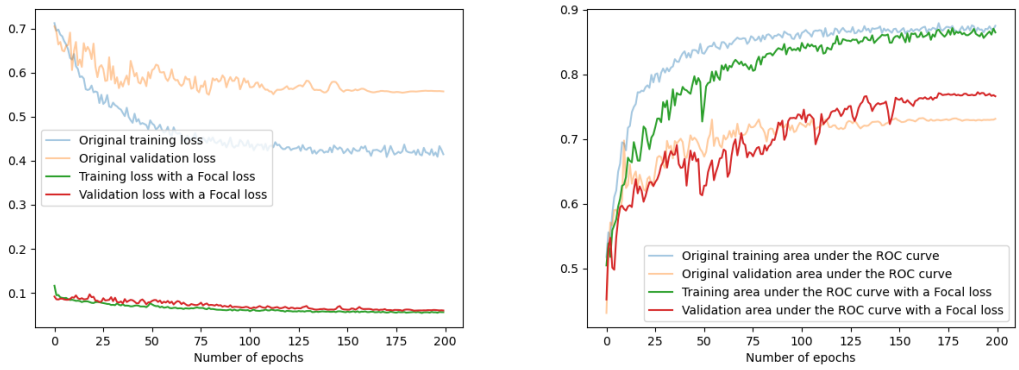

Figure 6. Comparison between original *TrimNet* model and the one with a Focal Loss function.

The following Table shows a comparison between the results of the testing for the best original model and the best model with a *Focal Loss*:

Testing of the best original *TrimNet* model		
loss	area under the ROC curve	precision
0.765	0.744	0.665
Testing of the best model with a *Focal Loss*		
loss	area under the ROC curve	precision
0.112	0.583	0.489

Since using a *Focal Loss* does not lead to better results, the *Cross Entropy* will be used from now on.

3.2.3 Changing the number N of blocks in the message phase

In (9) the chosen number of blocks in the message phase is $N = 3$. Figure 7 shows a direct comparison between the results obtained with the original *TrimNet* model and the one where instead $N = 6$ blocks are used.

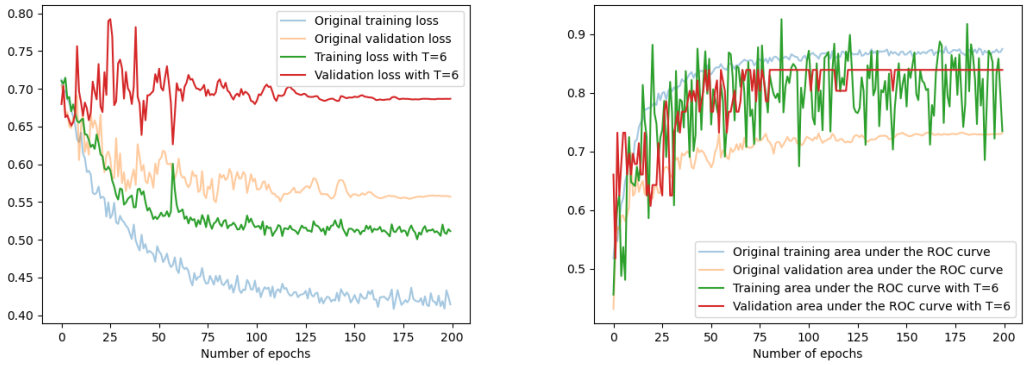

Figure 7. Comparison between original *TrimNet* model and the one where $N = 6$ blocks are used. T stands for N.

The following Table shows a comparison between the results of the testing for the best original model and the best model with $N = 1$, $N = 6$ and $N = 9$ blocks:

Testing of the best model with $N = 2$		
loss	area under the ROC curve	precision
0.856	0.601	0.531
Testing of the best original *TrimNet* model with $N = 3$		
loss	area under the ROC curve	precision
0.765	0.744	0.665
Testing of the best model with $N = 6$		
loss	area under the ROC curve	precision
0.721	0.625	0.568
Testing of the best model with $N = 9$		
loss	area under the ROC curve	precision
0.704	0.560	0.531

The following Table shows the elapsed time, for 200 epochs, of the four versions:

Elapsed time (min)	
Model with $N = 2$	30.0
Original model	44.8
Model with $N = 6$	83.2
Model with $N = 9$	113.8

Even without considering the much bigger elapsed times, it is clear that $N = 3$ gives the best results. Consequently, this is the number of blocks that will be continued to be used from now on.

3.2.4 *Changing the number K of attention mechanisms in the multi-head triplet attention mechanism*

In the original *TrimNet* model the multi-head triplet attention implemented in the message phase employs $K = 4$ attention mechanism.
Figure 8 shows the comparison between the results obtained with the original *TrimNet* model and the one with $K = 8$.

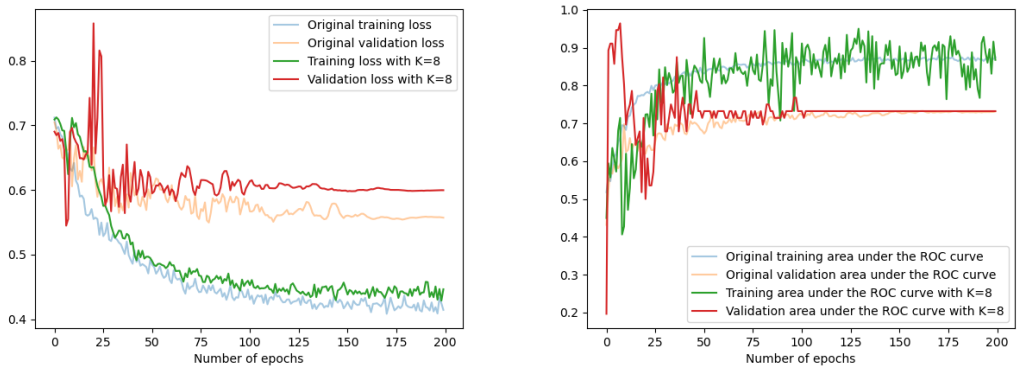

Figure 8. Comparison between original *TrimNet* model and the one where $K = 8$ attention mechanisms are used.

The following Table shows a comparison between the results of the testing for the best original model and the best models with $K = 2$ and $K = 8$ attention mechanisms:

Testing of the best model with $K = 2$		
loss	area under the ROC curve	precision
0.690	0.690	0.622

Testing of the best original *TrimNet* model with $K = 4$		
loss	area under the ROC curve	precision
0.765	0.744	0.665

Testing of the best model with $K = 8$		
loss	area under the ROC curve	precision
0.794	0.518	0.538

The following Table shows the elapsed time, for 200 epochs, of the three versions:

Elapsed time (min)	
Model with $K = 2$	25.6
Original model with $K = 4$	44.8
Model with $K = 8$	67.7

Again, it can be seen that the best result is obtained with the original *TrimNet* model.

REFERENCES

[1] Pengyong Li, Yuquan Li, Chang-Yu Hsieh, Shengyu Zhang, Xianggen Liu, Huanxiang Liu, Sen Song, Xiaojun Yao. *TrimNet: learning molecular representation from triplet messages for biomedicine.* Briefings in Bioinformatics, bbaa266

[2] Chung J, Gulcehre C, Cho KH et al. *Empirical evaluation of gated recurrent neural networks on sequence modeling.* In: arXiv preprint, arXiv:1412.3555, 2014

[3] Jimmy Lei Ba, Jamie Ryan Kiros and Geoffrey E. Hinton. *Layer Normalization.* ArXiv 1607.06450. 2016.

[4] Vinyals O, Bengio S, Kudlur M et al. *Order matters: sequence to sequence for sets.* In: International Conference on Learning Representations, San Juan, Puerto Rico, 2016. ICLR Press.

[5] Tsubaki M, Tomii K, Sese J. *Compound-protein interaction prediction with end-to-end learning of neural networks for graphs and sequences.* Bioinformatics 2019; 35(2): 309–18.

[6] RDKIT: Open-source cheminformatics. In 2006. rdkit.org

[7] K. He, X. Zhang, S. Ren and J. Sun. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification.* 2015 IEEE International Conference on Computer Vision (ICCV), Santiago, 2015, pp. 1026-1034, doi: 10.1109/ICCV.2015.123.

[8] Kingma DP, Ba JL. *Adam: a method for stochastic optimization.* In: International Conference on Learning Representations, San Diego, CA, 2015. ICLR Press

[9] https://github.com/yvquanli/TrimNet

[10] Govindan Subramanian, Bharath Ramsundar, Vijay Pande, and Rajiah Aldrin Denny. *Computational Modeling of $\beta-$Secretase 1 (BACE-1) Inhibitors Using Ligand Based Approaches.*. Chem. Inf. Model. 2016, 56, 10, 1936–1949.

[11] Xiong, Zhaoping and Wang, Dingyan and Liu, Xiaohong and Zhong, Feisheng and Wan, Xiaozhe and Li, Xutong and Li, Zhaojun and Luo, Xiaomin and Chen, Kaixian and Jiang, H. and Zheng, Mingyue. (2019). *Pushing the boundaries of molecular representation for drug discovery with graph attention mechanism.* Journal of Medicinal Chemistry. 63. 10.1021/acs.jmedchem.9b00959.