

# 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.

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<sup>1</sup> 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).

**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

---

<sup>1</sup> 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.

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*<sup>2</sup>), 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 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

<sup>2</sup>Generalizing the convolution operator to irregular domains is typically expressed as a *neighborhood aggregation* or *message passing scheme*.

with node features, edge features and adjacency matrix, as shown in Figure 1, using RDKit (6), a collection of cheminformatics and machine learning software written in C++ and Python. The extracted node and edge features are the ones give in (11), summarized in the following Table:

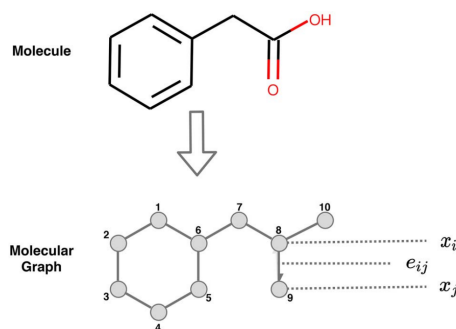
node features	edge features
atom symbol	bond type
number of covalent bonds	whether the bond is conjugated
electrical charge	whether the bond is in a ring
number of radical electrons	stereo
hybridization	
whether the atom is part of an aromatic system	
number of connected hydrogens	
whether the atom is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

For this study the evaluations are performed only on 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) strings. The SMILES is a specification in the form of a line notation for describing the structure of chemical species using short ASCII strings.

## 2.2 Architecture



**Figure 1.** Example of a molecule (top) with its corresponding molecular graph (bottom).

As shown in Figure 1 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  $h^t = h_0^t, h_1^t, \dots, h_N^t$  and the edges features  $e^t$ .

The prediction on this graph structure is performed by feeding it into the *TrimNet* architecture, depicted in Figure 4.

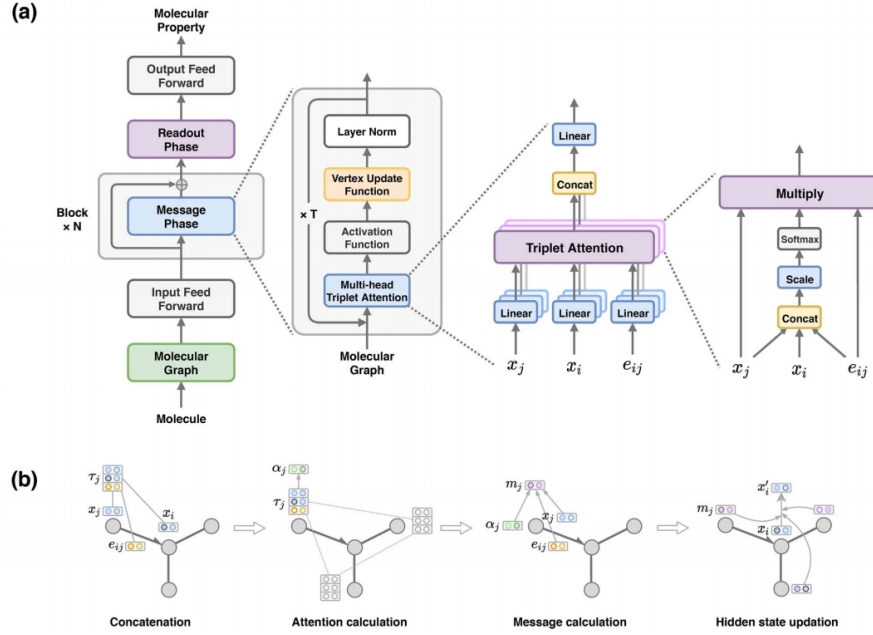


Figure 2. TrimNet architecture.

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:

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

$$h_i^{t+1} = LN(U_t(h_i^t, m_i^{t+1})),$$

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*  $R$ :

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

### 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  $h^t = h_0^t, h_1^t, \dots, h_N^t$  and the edges features  $e^t$  at each time step  $t$ , it firstly maps the node pair  $h_i, h_j$  and the edge hidden state  $e_{ij}^t$  to the same dimension  $D$ , and then it concatenates them into a triplet. This triplet is then fed into a feed-forward neural network according to:

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

where  $||$  represents concatenation,  $W_h \in R^{F1 \times D}$  is the learnable weight matrix shared across all nodes (with  $F1$  as the dimension of the initial node features),  $W_e \in R^{F2 \times D}$  is the learnable weight matrix shared across all edges (with  $F2$  as the dimension of the initial edge features),  $u \in R^{3D}$  is another learnable weight. *LeakyReLU* stands for the LeakyReLU nonlinear function.

To facilitate the comparisons of coefficients across different nodes, the coefficient themselves are normalized across all choices of neighbor  $j$  using the softmax function:

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

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.  $m_i^{t+1}$  consists of  $F$  features for each node. 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 then their features are concatenated. This results in the following output feature representation:

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

where  $||_k$  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.

The final returned output  $m_i^{t+1}$  consists now of  $K \times F$  features for each node.

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. Given a sequence of features  $m = (m^1, m^2, \dots, m^T)$ , the RNN updates its recurrent hidden state  $h^t$  by:

$$h^t = \begin{cases} 0 & \text{if } t = 0 \\ \phi(h^{t-1}, m^t) & \text{otherwise} \end{cases}\tag{6}$$

where  $\phi$  is a nonlinear function; traditionally  $h^t = g(Am^t + Bh^{t-1})$ , where  $g$  could be for example a logistic sigmoid function or a hyperbolic tangent function and  $A$  and  $B$  are weights matrices.

Training them 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.

The activation  $h_i^t$  of the GRU at time  $t$  is a liner interpolation between the previously activation  $h_i^{t-1}$  and the candidate activation  $\tilde{h}_i^t$ , given by:

$$h_i^t = (1 - z_i^t)h_i^{t-1} + z_i^t\tilde{h}_i^t. \quad (7)$$

$z_i^t$ , called *update gate*, decides how much the unit updates its activation and is computed by:

$$z_i^t = \sigma(W_z m^t + U_z h^{t-1})_i. \quad (8)$$

The candidate activation  $\tilde{h}_i^t$  is computed similarly to that of the traditional RNN:

$$\tilde{h}_i^t = \tanh(Wm^t + U(r^t \odot h^{t-1}))_i, \quad (9)$$

where  $r^t$  is a set of *reset gates*. When a reset gate is "off" (close to zero), it makes the unit act as if it was reading the first symbol of an input sequence, forgetting the previously computed state. The reset gate  $r_i^t$  is computed similarly to the update gate:

$$r_i^t = \sigma(W_r m^t + U_r h^{t-1})_i. \quad (10)$$

### Layer normalization

Finally, a layer normalization (3) stabilizes the hidden state dynamics for the recurrent network:

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

$h_i^{t+1}$  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.

The normalization of the activities also reduces the training time.

The *message phase* integrate skip connection and performs the above computations for  $T$  times recurrently to derive the final representation  $h_i^T$  for each node  $i$ .

### 2.2.2 Readout phase: the Set2Set network

The *readout phase* starts from the final updated nodes' features  $h^T = h_0^T, h_1^T, \dots, h_N^T$ . 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. When they are, the input can be read completely using an encoder, which could be a Long Short-Term Memory unit (LSTM) or a convolutional network, and the final state can be fed to a decoder LSTM whose purpose is to produce the target sequence. But when instead a natural order cannot be determined, nevertheless a non natural order which yields better performances can be found thanks to *Set2Set*.

It performs the following computations for  $T'$  times recurrently to obtain the final graph representation  $q_{T'}^*$ :

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

$q_t$  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 but takes no inputs.  $q_t^*$  is the state evolved by this LSTM, and is formed by concatenating the query  $q_t$  with the resulting attention readout  $r_t$ .

### Long Short-Term Memory Unit

The LSTM is a RNN architecture. 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.

The computation performed by *Set2Set* is accomplished by three components: a *reading block*!!!, a *process block* and a *write block*.

The *reading block* embeds each input element in a *memory vector* using a small neural network.

The *process block* is a LSTM without inputs or outputs that performs  $T'$  steps of computation over each *memory vector*  $h_i^T$ . It keeps updating its state by reading each *memory vector* repeatedly using the attention mechanism described in the previous Equations.

The *write block* is an LSTM pointer network that takes as input  $q_{T'}^*$  and points at elements of each *memory vector*, one step at a time.

**Interpretability:** Atomic attention in this *Set2Set* network can provide interpretability 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.

For the prediction of quantum mechanical property, the *Least Absolute Deviations loss function* L1 can be used as loss function:

$$L = - \sum_{i=1}^N |\hat{y}_i - y_i|. \quad (12)$$

Whereas for bioactivity and physiology prediction task, the best choice is the *weighted focal loss function*, that can deal with the data imbalance problem and is defined as:

$$L = - \sum_{i=1}^N \alpha (1 - \hat{y}_i)^\gamma \log(\hat{y}_i), \quad (13)$$

where  $\alpha$  is a weighting factor that balances out the importance of positive and negative examples (positive examples weight more !!!) and  $\gamma$  is the focusing parameters that adjusts the rate at which the easy examples are down-weighted!!!. In (1),  $\alpha = \frac{\text{No. negative samples}}{\text{No. all samples}}$  and  $\gamma = 2$ .

For CPI instead, the *cross entropy loss function* (5) is often used:

$$L = - \sum_{i=1}^N \log(\hat{y}_i). \quad (14)$$

### 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  $\theta$ , we want to minimize its value with respect to its parameters. Its realisations at subsequent time steps are called  $f_1(\theta), f_2(\theta), \dots, f_T(\theta)$ , 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  $g_t \odot g_t$ .  $\alpha$  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:  $\theta_0$ : Initial parameter vector
 $m_0 \leftarrow 0$  (Initialize 1st moment vector)
 $v_0 \leftarrow 0$  (Initialize 2nd moment vector)
 $t \leftarrow 0$  (Initialize timestep)
while  $\theta_t$  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)
     $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$  (Compute bias-corrected first moment estimate)
     $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$  (Compute bias-corrected second raw moment estimate)
     $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$  (Update parameters)
end while
return  $\theta_t$  (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 is 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.

## 2.4 The code

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**.

### 2.4.1 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 3.

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.

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:

$$CELU(x) = \max(0, x) + \min(0, \alpha(e^{\frac{x}{\alpha}} - 1)), \quad (15)$$

with  $\alpha = 1$ .

The result is given as input sequence for the GRU, which computes Equations from 7 to 10.

Finally a layer normalization (as described in the third paragraph of Section 2.2.1) is applied. The final output corresponds to Equation 11

**TrimNet Class:** This last class defines the *TrimNet* model. It includes a *forward function* that starts from the final state defined by the *Block class*, and during the training randomly zeroes some of the elements of the input tensor with probability  $p$  using samples from a Bernoulli distribution.

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.

### 2.4.2 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:

$$loss = -\alpha(1-p)^{\gamma}(target \cdot \log(p)) - (1-\alpha)p^{\gamma}((1-target)\log(1-p)), \quad (16)$$

where  $\alpha = 0.25$ ,  $\gamma = 2$ ,  $p$  is obtained applying the *Softmax function* to the input (a tensor of dimension  $batches \times 2$ ) and *target* are the true values. The mean of this value is given as output.

**One-Hot Encoding Functions:** The code contains two function for *One-Hot Encoding*, the process applied to categorical data<sup>3</sup> to convert it into a binary vector array. The first function raises an error if the input is not in the allowed set, while the second one maps it to the last element of the set.

**splitmultilabelcontainNan Function:** This function starts by returning from a comma-separated values (csv) file a two-dimensional data structure with labeled axes. Then it randomly divides this structure between training, validation<sup>4</sup> and testing samples and computes a weight that depends on the ratio between positive and negative training samples. This is done for each task.

**ScaffoldGenerator Class:** It generates *Murcko scaffolds* (or *Bemis-Murcko scaffolds*) from array-like molecules.

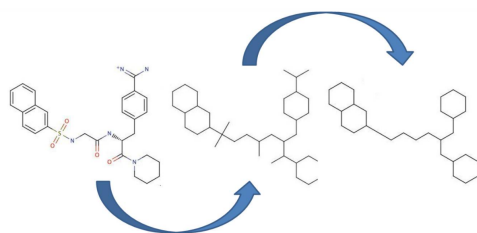


Figure 3. Bemis-Murcko framework generation.

*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 3. 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.

**generatescaffold Function:** It computes the *Bemis-Murcko* scaffold for a SMILES string.

**scaffoldrandomizedsplitting Function:** After the generation of the scaffolds, they are split for the training, validation and testing.

### 2.4.3 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 for the training set.

For each batch (including 128 molecules), this function starts by clearing old gradients from the last step. 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

<sup>3</sup>Categorical data are variables that can take on one of a limited number of possible values.

<sup>4</sup>Validation is used to select proper parameters of the system and is part of the training set.

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.

#### 2.4.4 dataset.py

This code is dedicated to the loading of the different kind of datasets.

#### 2.4.5 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 then 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:

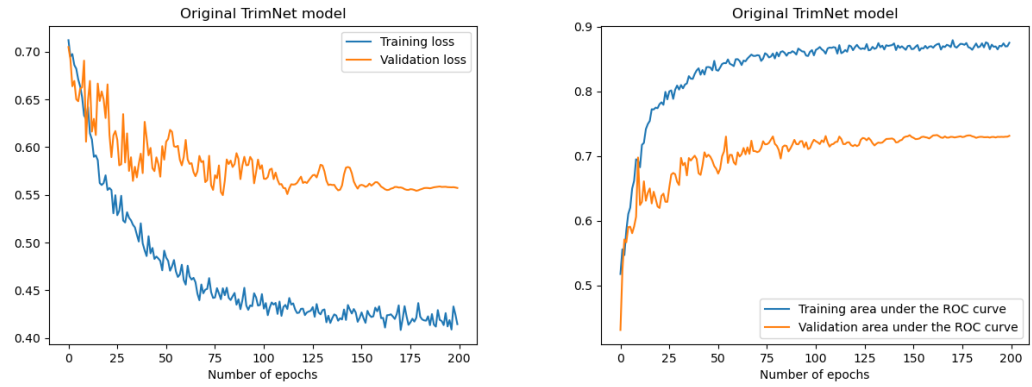
- A logbook (Notepad file) with all the parameters defining the model and its implementation. It also contains for each number of epochs the loss, the area under the ROC curve, the precision, the learning rate and the elapsed time for training, validation and testing sets.
- A Microsoft Excel table that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

## 2.5 Implementation

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.

An additional small code was 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 and the learning rate.

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.

## 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.