UNIVERSITY OF CALIFORNIA,
IRVINE

Deep Learning in Chemoinformatics using Tensor Flow

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Computer Science

by

Akshay Jain

Thesis Committee:
Professor Pierre Baldi, Chair
Professor Cristina Videira Lopes
Professor Eric Mjolsness

2017

DEDICATION

To my family and friends.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank my thesis advisor Prof. Pierre Baldi of the University of California Irvine for providing me with this opportunity and guidance throughout my thesis. I am deeply grateful to Prof. Eric Mjolsnes and Prof. Cristina Videira Lopes being being on the committee and carefully reviewing my work.

Also, I would like to thank Gregor Urban, a graduate student at the University of California, Irvine for providing me valuable feedbacks and the help during the implementation part of the project.

ABSTRACT OF THE THESIS

Deep Learning in Chemoinformatics using Tensor Flow

By

Akshay Jain

Master of Science in Computer Science

University of California, Irvine, 2017

Professor Pierre Baldi, Chair

One of the widely discussed problems in the field of chemoinformatics is the prediction of molecular properties. These properties can range from physical, chemical, or biological properties of molecules to the behaviour or molecules under certain chemical conditions.Traditionally, these properties were calculated using chemical experiments. But with the increase in computational capabilities, various machine learning methods like neural networks and kernel methods have also been tried. These approaches have been successful to a certain extent. But with recent development in data, deep machine learning techniques have been developed, which matches or exceed the performance of state-of-the-art techniques. One such approach is to consider the molecular graphs of the molecules and use them for prediction. Here, I discuss this approach in great details and provide its application on two problems: predicting aqueous solubility and melting point.

Chapter 1

Introduction

One of the widely discussed problems in the field of chemoinformatics is the prediction of molecular properties. These properties can range from physical, chemical, or biological properties of molecules to the behaviour or molecules under certain chemical conditions. Traditionally, these properties were calculated using chemical experiments. However, with the increase in computational capabilities, various machine learning methods like neural networks and kernel methods have also been tried [4–8]. These approaches have been successful to a certain extent. But with recent development in data(increase in the number of datasets and bigger datasets), deep machine learning techniques have been developed, which matches or exceed the performance of state-of-the-art techniques. Here, I discuss one such technique proposed by Lusci et al. [1] and its implementation in Tensor Flow. To access the applicability of the method, we use the model to predict two different molecular properties: Aqueous Solubility and Melting Point.

Aqueous Solubility

Aqueous solubility is the concentration of molecules in the aqueous phase, when aqueous phase is in equilibrium with its original state, at specified temperature and pressure. The fact that 80% of the human body is made of water makes this property very important in drugs and other medicines. The ability to accurately determine the aqueous solubility can reduce the cost in drug discoveries and avert failures[9].

Melting Point

Melting Point is a very important characteristic property of a chemical compound. It is the temperature at which the solid phase of a compound is in equilibrium with its liquid phase at atmospheric pressure. Recently, interest in the prediction of melting points has been fostered by the growing body of work on ionic liquids.

1.1 QSAR Prediction Methods

The limitations of experimental determination of chemical properties of molecules have led to the development of predictive methods. Various in silico prediction techniques have been suggested, most of which employs the method of QSAR (Quantitative Structure-Activity Relationship) models [10]. QSARs are based on the assumption that activity of molecule is directly related to its structure, i.e compounds with similar structure exhibit similar properties. QSAR models are designed to find relationships between molecular structure(or structure related properties) and target activity of the chemical compound. These are regression models, which relate a set of "predictor" variables to the potency of the response variable. The predictors generally consist of physio-chemical properties or theoretical molecular descriptors of the molecule, and response variable is the activity of the molecule. The general

form of QSAR model is:

$$Activity = F(structure) = M(E(structure)) \tag{1.1}$$

QSAR models(function F()) are generally factorised into two functions: the encoding function E and the mapping function M. The encoding function performs the required transformation of molecule's structural properties to a fixed length vector. This step is necessary because the naturally represented graph structure of molecules are not understood by standard regression tools. The fixed length representations obtain after E step can now be used to perform prediction using the mapping function M. In general, Neural Networks and Support Vector Machines are used to learn mapping function F from the training examples.

Molecular features are at the core of QSAR modelling and various such feature has been proposed in theory so far. These feature range from the 1-D molecular formula to the 2-D structural formula[25] to 3-D conformation-dependent and even higher levels orientation of the molecules.

Over the year through various experiments, several other molecular features were found to correlate with aqueous solubility, including polar surface area[13], octanol-water partition coefficient,[14–16] melting point[17], hydrogen bond count[18], and various molecular connectivity indexes[19–21]. Many prediction methods[24] uses some combination of these features to predict aqueous solubility(similarly for melting point).

The accuracy of these models have been limited due to the inability of finding a perfect set of molecular feature that captures all the properties of the chemical compound [12], and also due to the errors in the descriptor values, whether measured or calculated [27, 28]. Another reason behind the current limitations of prediction methods is the small size of the available training sets and inaccuracy in their experimental values.

1.2 Deep Learning

Over the past few years, deep learning systems have matched and improved the state-of-the-art in many fields from speech recognition, to computer vision, to artificial intelligence [30–37]. Hence, it is natural to extend the idea of deep learning to chemoinformatics. Since the most common and natural form of representation of small molecules is through graphs, it is useful to develop deep architecture, that are able to understand the properties of molecules directly through its graphical structure. This can be achieved using the recursive approach described in [45]. However, the standard recursive approach relies on data represented by directed acyclic graphs (DAGs), whereas molecules are usually represented by undirected graphs (UGs). A novel approach of was suggested in Lusci et al. [1], which tries to extend the idea of DAG, and which I will be discussing in further details. In the next chapter, we review the neural networks and the basic terminologies used in the general recursive approach for building deep learning architectures from DAGs and then show how the approach is adapted to molecular UGs.

Chapter 2

Artificial Neural Networks(ANN)

Artificial neural networks(neural networks from here on) are a computational approach, which tries to mimic the modelling of a human brain in order to learn to perform tasks as humans do. Though neural networks try to mimic the human brain, they are far less complex for the real brain. In fact, NN lacks a lot of details and are quite simple in nature. The most basic structural and functional unit of the neural network is a neuron (Figure 2.2). These neurons are generally stacked together in the layered structure connected via weights as shown in Figure 2.1. These systems are self-learning and trained through back propagation. Neural Networks are the fundamental blocks on which deep architecture are built.

2.1 Artificial Neuron

Artificial neurons (Figure 2.2) are the most basic unit of the neural networks. It receives one or more inputs and sums them to produce outputs. The sum is usually a weighted sum of the inputs and passed through activation function or transfer function. Neurons are

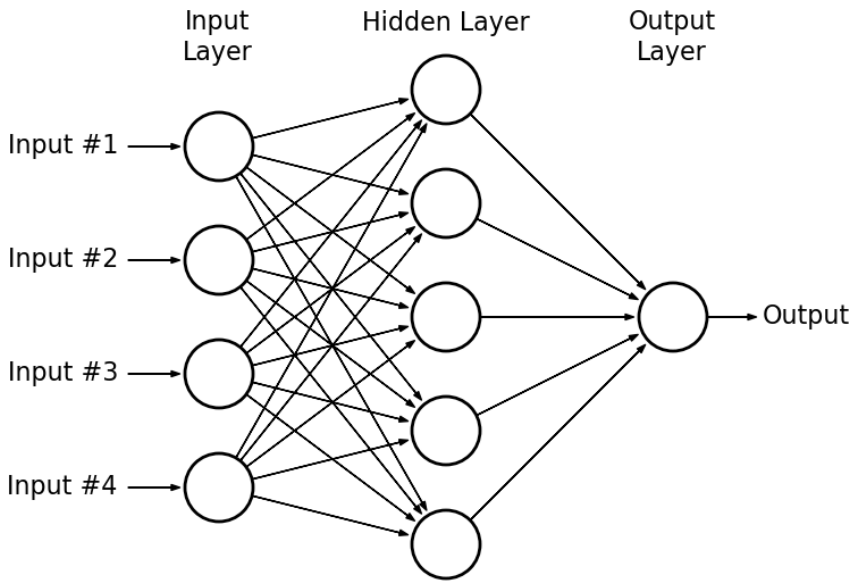

Figure 2.1: Neural Network
A neural network with one hidden layer. The inputs are passed via weighted connection to hidden layer. At each neural unit of the hidden layer, the summation function combines all the incoming weighted inputs, which is finally passed to the output layer.

mathematically described as operation:

$$y = \sigma(\sum w_i x_i + b) \tag{2.1}$$

where $x_i \in R$ is one of the input, $w_i \in R$ is the weight corresponding to the input node, $b \in R$ is bias and $\sigma : R \to R$ is called activation function.

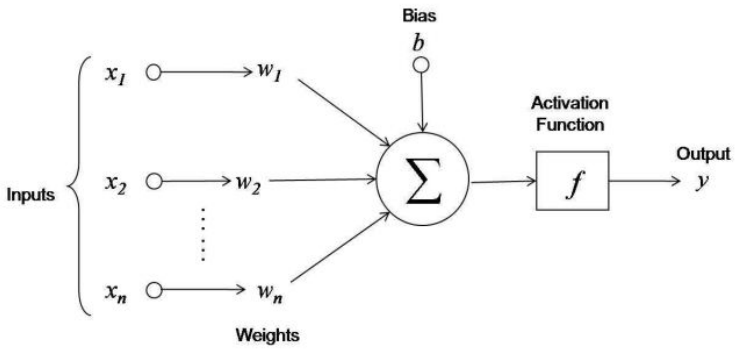

Figure 2.2: Artificial Neuron

2.2 Activation Function

The activation function of the neuron is chosen to have a number of properties. Activation functions are continuous function and piecewise differential, which allows the network to be trained during backpropagation. The problem with the linear transfer function is that any multilayer neural network can effectively be converted into an equivalent single-layer network and does not support non-linear function. Hence, a non-linear activation function is necessary to gain the advantages of a multi-layer network network. Various activation function have been suggested and experimented with, but the most common and widely used are sigmoid function (2.2), hyperbolic tangent function (2.3) and rectifier linear unit function (2.4)

Sigmoid(Logistic) Function:

$$y = \frac{1}{1 + e^{-x}} \tag{2.2}$$

Hyperbolic Tangent Function:

$$y = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.3}$$

Rectifier Linear(ReLu):

$$X(\omega) = \begin{cases} 0, & x \leq 0 \\ x & x > 0 \end{cases} \tag{2.4}$$

2.3 Loss function

The loss function is used for parameter estimation and estimates the accuracy of the neural network. It tries to capture the difference between the predictor value and the true value. The parameters of the neural network are optimised, so as to minimise the loss function. This allows use to treat the parameter update as the optimisation problem. The type of the loss function strongly depends on the type of the problem, though it is advisable to choose a loss function that is differentiable, in order to satisfy optimisation constraints. The most common loss function for regression problems are:

Root Mean Square Error (RMSE):

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(t_i - p_i)^2} \tag{2.5}$$

Average Absolute Error (AAE):

$$AAE = \frac{1}{n}\sum_{i=1}^{n}|t_i - p_i| \tag{2.6}$$

where n is the total number of data points, p_i and t_i is the predicted and the target value respectively for i^{th} data point.

2.4 Optimization

In order to minimise the loss function, we need to update the model parameters. The way loss function is defined, we can differentiate it and perform gradient descent to find the loss function optimum. Also, choosing a differentiable activation function allows us to

calculate the gradients at all layers using back propagation. Though gradient descent is the most common approach, it often gets stuck in local optima. There are various different optimizers, which tries to solve this problem either by considering second order derivatives or some heuristics. The most common of these are Conjugate Gradient Descent, Adam Optimizer, Momentum gradient decent.

Chapter 3

Deep Recursive Architectures

ANN generally has one or two layers between the input and output unit, which works fine in case of approximating simple functions. But more complex tasks or function requires multiple levels of learning. In deep architectures, there are multiple layers between the input and output, allowing the model to use multiple processing layers, composed of multiple linear and non-linear transformations. These deep layers can be the direct result of the multiple layers of neurons or the recursive nature of the architecture. Here, we discuss various kind of recursive deep architecture, upon which the deep learning architecture for molecules is built.

3.1 Recurrent Neural Networks (RNN)

In many cases, inputs are in form of structured data with variable length, for example, a sequence of words of any length. Neural Networks in their native form are unable to capture this ordering. It's mainly because traditional networks assumes that all inputs(and outputs) are independent of each other. A recurrent neural network (RNN) on the other hands, handle this by making connections between units to form a directed cycle. This creates an internal

state of the network which allows it to store information for previous inputs and can process sequences of inputs with arbitrary length.

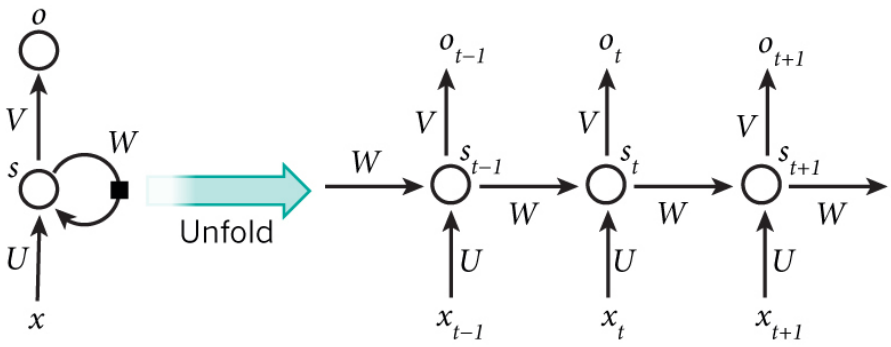

Figure 3.1: Recurrent Neural Network

3.2 Recursive Neural Networks

A recursive neural network is a deep neural network created by applying the same weights over structured inputs. This recursive approach produces a prediction over a variable length input by traversing the given structure. Recursive neural networks are the more generalized form of recurrent neural network. RNNs works only with the specific type of structure i.e a linear chain, whereas recursive neural network works with any type of hierarchical structure, combining child representations into parent representations (Figure 3.2).

3.3 Directed Acyclic Graph Recursive Neural Networks (DAG-RNN)

DAG RNN is useful when the data can be represented as DAG. The edges typically denote some kind of causal or temporal relationship between the nodes or variable. DAG-RNN

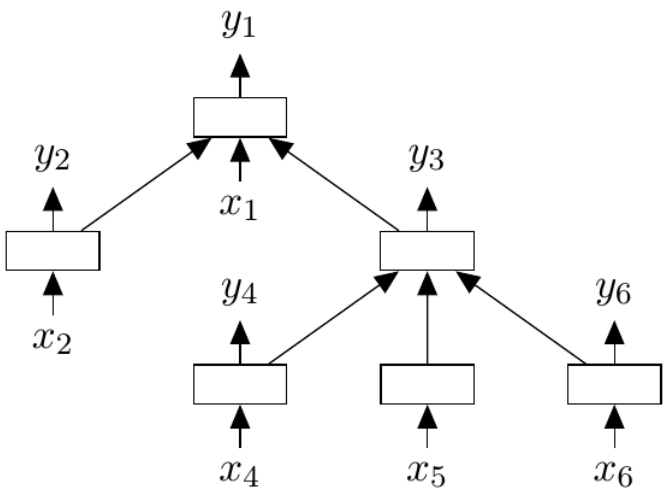

Figure 3.2: Recursive Neural Network

associates a vector with each node of the graphs and used the directed edges to propagate information. It places a neural network at each edge to capture the relationship between the corresponding node. In theory, a different neural network can be applied to each edge, it's often useful to share the weights of the neural networks among similar edges. One advantage of weight sharing is that it reduces the number of parameters in the architectures, which allows it to train faster and lesser data.

For instance, Figure 3.3 is the graphical model representation of a first-order Hidden Markov Model (HMM) for sequence data. The HMM is a finite set of hidden states, where each states transitions to the another states among the set,governed by the transition probabilities (horizontal edges). At every states, an observation can be generated, according to the associated probability distribution called emission probabilities (vertical edges). The DAG associated with a hidden Markov model of the data can be converted to a deep neural network by using two basic neural network, as shown in Figure 3.4. One neural network for the transition probabilities and one neural network for the emission probabilities. When the architecture is unfolded in time or space, it yields a deep neural network with many layers and shared weights which can be trained by gradient descent (back propagation) and other

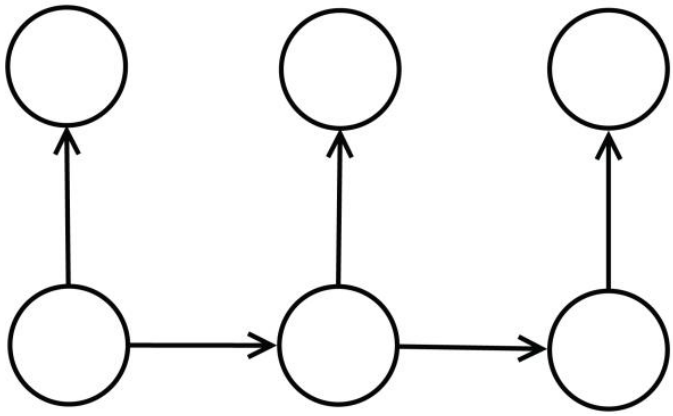

Figure 3.3: Directed acyclic graph (DAG) of a Hidden Markov Model

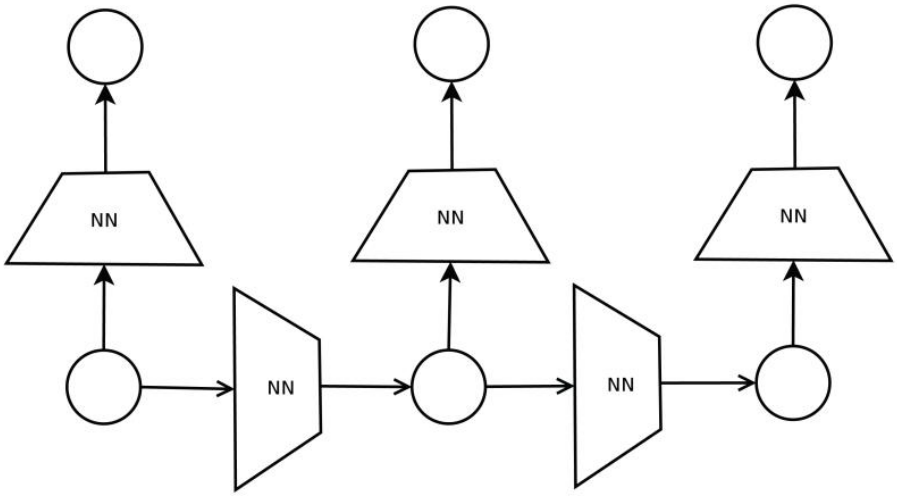

Figure 3.4: DAG RNN

algorithms.

Chapter 4

UG-RNN for small molecules

All the approaches discussed so far works fine with data that can be represented as a sequence or a DAG [44–46]. But, molecules are generally represented as undirected graphs (UG) and may contain cycles, which raise an obvious question of how to extend the approach of DAG-RNN to UG. One possible solution is to consider to convert the UG into a DAG in some orderly fashion. Since there can be multiple DAG possible for a UG, the resulting DAG is going to be quite arbitrary among all the possible DAG's and might not capture all the information. Lusci et al. [1] suggests an approach which tries to overcome this limitation by considering all possible DAG of the molecule and using them as an ensemble. This is possible because small molecules have a small number of nodes and few cycles, which make the computation feasible.

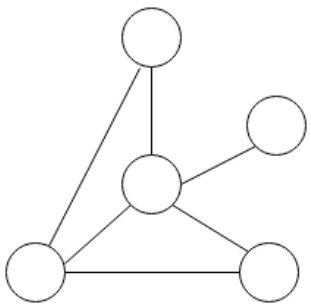

Figure 4.1: Undirected Graph

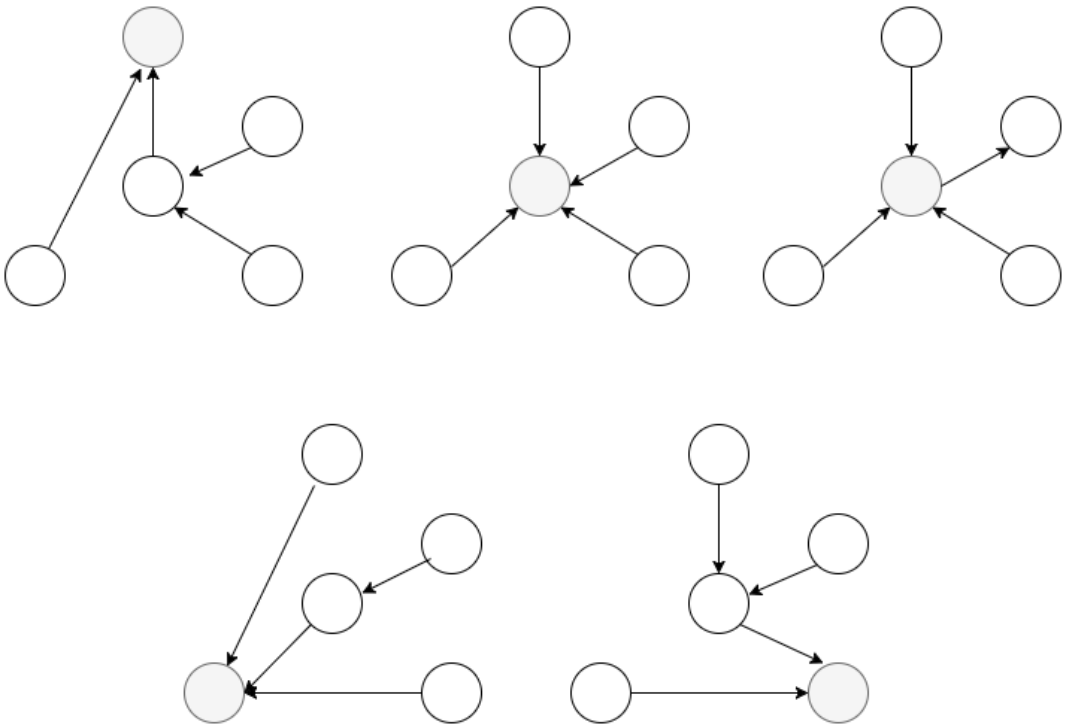

Figure 4.2: Directed acyclic graphs.

We consider all possible acyclic orientations of the undirected graph and use them as an ensemble. Given an undirected graph, UG-RNN iterates over all the vertices and generates a DAG using the selected vertex as the root. The DAG has the same number of vertices as in the UG and all its edges are pointed towards the root vertex along the shortest possible path. The process is schematically illustrated in Figures 4.1 and 4.2. There may be certain cases, where the original undirected can be oriented in any direction and more than on one shortest path exists, resulting in multiple DAGs for one root node. Though we can consider all these possible DAGs, it will lead to a high number of DAG's specially in case of multiple cycles and would require lot more computations. We solve this problem by considering one random orientation, which allows having fixed number of DAGs. For instance, if a UG has N vertices, the UG-RNN approach yield N DAGs, i.e one DAG per root vertex. Once DAGs are generated, we can then apply the DAG-RNN approach discussed in 3.3. The output of these DAGs are combined to obtain ensemble and make the final prediction.

4.1 DAG Generation

Consider a molecular undirected graph with N nodes $v_1, ..., v_N$. In order to generate the DAG(for instance for vertex v_1), we first generate shortest path from v_1 to all the other nodes $v_2, ..., v_N$ and since graph is unweighted graph, we can use Dijkstra's shortest path algorithm. Let the paths be $p_2, ..., p_N$, where p_i denotes shortest path to node v_2. Once we have all the paths, create a new graph with N nodes $v_1, ..., v_N$ and edges in paths $p_2, ..., p_N$. In the new graph all the edges are pointing away from v_1, so simply reverse all the edge to get the DAG D_1(D_k represents the DAG generated with v_k as root node). The parent of node v in D_k are represented by ${pa^1}_{[v,k]}, ..., {pa^n}_{[v,k]}$.

4.2 Local Information Vector

The local information vector $i_{[v,k]} \in R^l$ is the vector associated with the properties of vertex v in the DAG D_k. UG-RNN allows the flexibility to choose the nature of information, one want to associate with each node. This can include atom type, properties of the molecule like aromaticity, it bonds with neighbours, topological indices, information about local paths(like part of the cycle). Lusci et al. [1] suggests of using just atom type and bond properties. The underlying hypothesis behind this it that UG-RNN is able to extract these properties automatically by crawling. The atom type is encoded as one-hot vector,for instance in case of three atom types, the atom type is encoded as C = (1,0,0), N = (0,1,0), and O =(0,0,1). For bonds, we only consider the the bonds between the node and its parents ${pa^1}_{[v,k]}, ..., {pa^n}_{[v,k]}$.

4.3 Contextual Vectors

Each node v in the DAG D_k has an contextual vector $G_{v,k}$ associated with it. It tries to capture the local properties of the node and the properties associated with its parent ${pa^1}_{[v,k]}, ..., {pa^n}_{[v,k]}$. It can be seen as function (equation ...) of local information vector $i_{v,k}$ and of the contextual vectors of its parent nodes, $G_{{pa^1}_{[v,k]}}, ..., G_{{pa^n}_{[v,k]}}$. The function can be represented as:

$$G_{v,k} = M^G(i_{[v,k]}, G_{{pa^1}_{[v,k]}}, ..., G_{{pa^n}_{[v,k]}}) \tag{4.1}$$

The function M^G is recursive function, as contextual vectors are computed as the function other contextual vectors, and is implemented by using a parameterized neural network. Since, neural network has fixed size of input vector, a node cannot have arbitrary large number of parents. We need to have upper bound n to the number of parents a node can have. In case of small molecules, n=4 works. In case, a node has m parent nodes with $m < n$, blank vectors (all zeroes) are passed to the function M^G as its last n-m arguments. If, a node has m parent nodes $m > n$, we choose any random n parents.

As all the edges in the DAG are pointing towards the root node, the above process produces a final contextual vector $G_{r_k,k}$ at the root node. This vector receives some information directly or indirectly from all the other nodes of the DAG and can be seen as the summary of the DAG from the view of the root node.

4.4 Activity Prediction

At the end of crawling process, the N different views are combined to get the overall description of the molecule $G_{structure}$. There are various ways of combining the final contextual

vectors, but here to keep things simple, we just add them. Hence $G_{structure}$ can be defined as:

$$G_{structure} = \sum_{k=1}^{N} G_{r_k,k} \tag{4.2}$$

This $G_{structure}$ can be treated as the vectorial representation of the molecule and is used to make predictions about the molecule. The final prediction is produced by the output function M^O

$$Activity = M^O(G_{structure}) \tag{4.3}$$

The output function M^O can be implemented by a feed-forward neural network. Though other types of parameterized function can be used like SVM, by using neural networks for M^G and M^O, the overall architecture is deep feed-forward neural network, which can be trained by gradient decent [49–52] to minimise the error between predicted and true values. Given a training data set of decent size, the model can be trained in fully automatic and task-specific manner. Once the parameters of the M^G and M^O networks are trained successfully, $G_{structure}$ provides an optimal encoding for prediction.

4.5 UG-RNN With Contracted Rings (UG-RNN-CR)

One of the common problems with deep architectures is that it can run into the problem of vanishing gradient or exploding gradients during backpropagation [53, 54].In back propagation, gradients are calculated using the chain rule, i.e. gradient of "front" layers are calculated by multiplying the gradients of deeper layers. If the gradients are small, the gradient decreases exponentially and the front layers are never trained. This is called gradient vanishing. And if the gradients are large, this can lead to gradient explosion. The problem

gets severe as the depth of the deep architecture increases. One possible solution to avoiding vanishing gradient or gradient explosion is by reducing the depth of the architecture. Since the actual size of networks M^G and M^O are quite small, we essentially need to reduce the depth of the recursive architectures. We achieve this by contracting the rings in the original to a single point, which results in smaller UGs. It is very common in molecules to have one or more cycles. An example of a molecule with a single ring is Aspirin (Figure 4.3) and an example of a polycyclic molecule is Cirpo (Figure 4.4). In order to reduce the rings, we

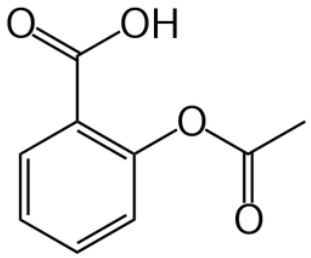

Figure 4.3: Aspirin.

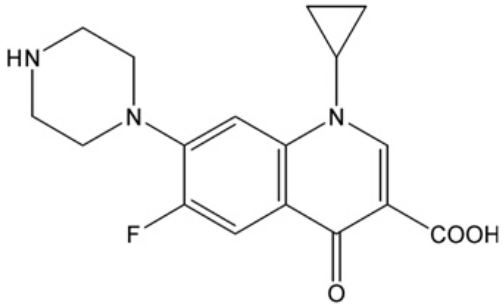

Figure 4.4: Cirpo.

fist compute the smallest set of smallest rings [56, 57] of the molecular graph G. Now, each ring R is contracted to a single node with a new label R_n where n is the length of the ring. All the edges connecting the non-ring nodes to the ring nodes are connected to the new node. In the case of polycyclic molecules, a new node associated with a ring R can also be connected to other newly created nodes with edge label as for single-bond edges. Applying this procedure to the graphs representing Aspirin and Cirpo yields the graphs in Figures 4.5 and 4.6, respectively.

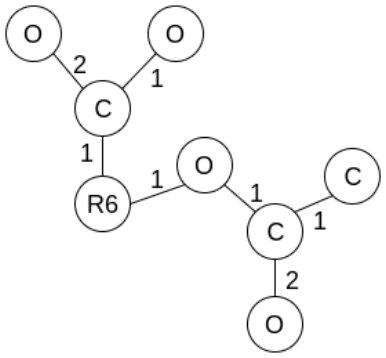

Figure 4.5: Contracted graph of Aspirin with bond types.

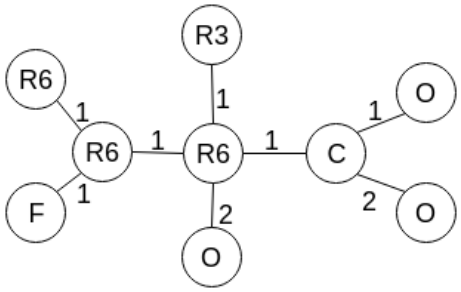

Figure 4.6: Contracted graph of Cirpo with bond types.

4.6 Example: UG-RNN Model of Propionic Acid

Propionic acid (Figures 4.7) is one of the naturally occurring carboxylic acids, which we use here as an example to demonstrate the UG-RNN approach

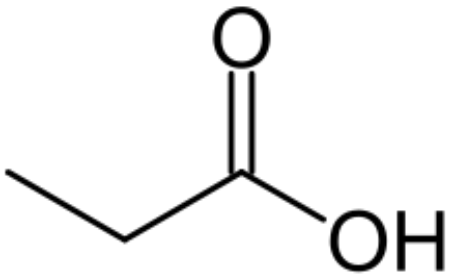

Figure 4.7: Propionic acid.

The UG of Propionic acid has 5 nodes(implicit Hydrogen atoms are ignored) as shown in Figure 4.8. The first step is to convert the UG into 5 DAG by the procedure discuss in 4.1 (Figure 4.9, root atoms highlighted).

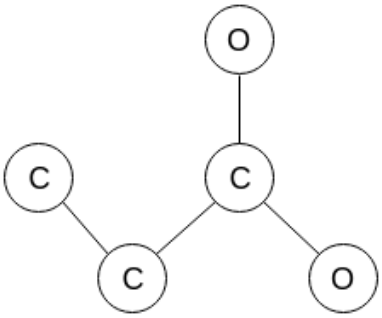

Figure 4.8: Propionic acid undirected graph.

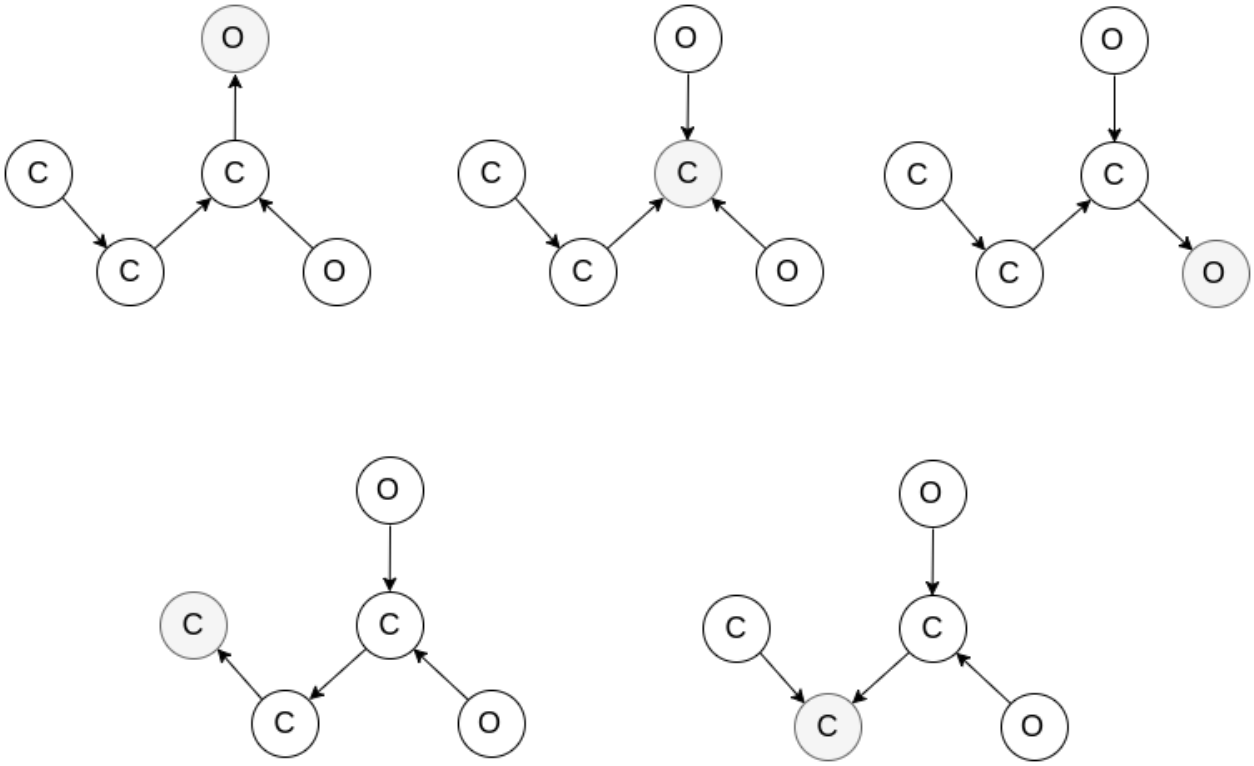

Figure 4.9: Propionic acid DAGs

The second step is to initialize the contextual vector of each source nodes in all DAGs and crawl the DAGs along the edges using the neural network M^G, to calculate the contextual vector for all the internal nodes, up to the root node. The contextual vectors for source node are set to 0. For instance, in the top DAG in Figure 4.9 with root node v_5, the contextual vector of the parents two source nodes: v_1 (carbon atom) and v_3(oxygen atom) are set to 0 and only the input vector associated with the nodes are used. Information is then propagated along the DAG structure using Equation 4.1 to compute the contextual vector

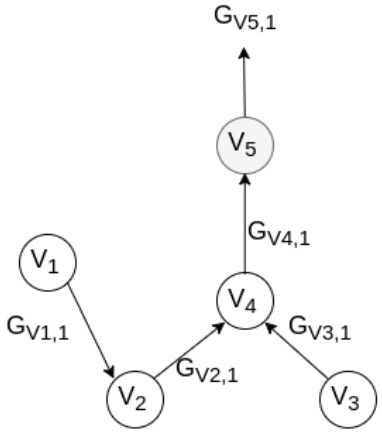

Figure 4.10: Application of M^G to the first DAG.

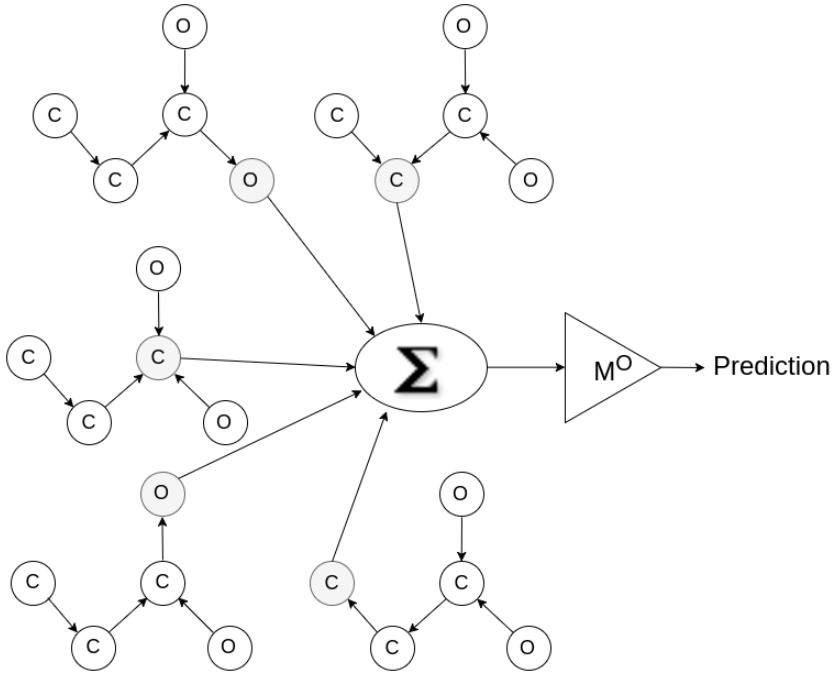

Figure 4.11: UG-RNN approach on Propionic Acid
Sum of five G vectors to produce the vector $G_{structure}$. The output function M^O produces the final prediction.

of each internal node, and ultimately of the root v_5, resulting in the vector $G_{v5,1}$ (Figure 4.10). The same procedure is applied to the other four DAGs, producing five vectors, one for each DAG. These five contextual vector associated with the roots of the DAGs are then

summed together to generate the vector $G_{structure}$, describing the whole molecular graph (Figure 4.10). In the final step consists, this vector $G_{structure}$ is mapped to the property of interest using the neural network corresponding the output function M^O. The parameters of M^G and M^O are then adjusted by training the architecture on dataset, reducing the error between the predicted and true value.

Chapter 5

Implementation

One of the biggest problems with deep architecture is the amount of time and computation required for training the models. Hence, its implementation is really crucial. In recent years, various frameworks have been developed, which tries to increase the performance of neural networks. TensorFlow is one of the latest addition to these frameworks, developed by Google. One of the biggest motivation behind using TF is the ability to run on 64-bit Linux or Mac OS systems, with multiple CPUs and GPUs.

We use a standard three-layer neural network(with one hidden layer) for both the encoding function M^G and the output mapping function M^O. In order to train and test the models, the data is randomly divided training and test set in 90/10 proportion. Models are trained using the training set and the test set is used for testing and performance evaluation. Furthermore, the training set is randomly split into proper training set and validation set in 80/20 proportion. The validation set is used to fit the hyperparameters of the models

The parameters of the model are optimised in order to get the lowest RMSE. (root mean square error) using gradient descent. We experimented with various learning rates η, varying it from 10^1 to 10^4 and the most optimal value found is 10^2. We train the models for 300

epochs.

In order to reduce the residual generalisation error [68], we use an ensemble of 20 models with a different number of hidden units and outputs units, as described in Table 5.1. These 20 models are stacked together and their outputs are combined together to create an ensemble. Though there are various methods of stacking models together, we use three: random forest regression, best 10 models and greedy, and one that gives the least root mean square error (RMSE) on the validation set is used to make predictions on the test set.

Neural Network	M^G Hidden Units	M^G Output Units	M^O Hidden Units
Model 1	7	3	5
Model 2	7	4	5
Model 3	7	5	5
Model 4	7	6	5
Model 5	7	7	5
Model 6	7	8	5
Model 7	7	9	5
Model 8	7	10	5
Model 9	7	11	5
Model 10	7	12	5
Model 11	3	3	5
Model 12	4	3	5
Model 13	5	3	5
Model 14	6	3	5
Model 15	7	3	5
Model 16	8	3	5
Model 17	9	3	5
Model 18	10	3	5
Model 19	11	3	5
Model 20	12	3	5

Table 5.1: Architecture of 20 encoding neural networks M^G and output neural networks M^O

Chapter 6

Data & Results

In order to compare the performance of UG-RNN with other methods, we use three metrics: Root Mean Square Error (RMSE 2.3), Absolute Average Error (AAE 2.3) and Pearson correlation coefficient (R 6.1)

$$R = \frac{\sum_{i=1}^{n}(t_i - \bar{t})(p_i - \bar{p})}{\sqrt[2]{\sum_{i=1}^{n}(t_i - \bar{t})^2 \sum_{i=1}^{n}(p_i - \bar{p})^2}} \tag{6.1}$$

where $\bar{p}$ and $\bar{t}$ are the average prediction and target values respectively. In some places, we calculate R^2 in order to compare it with other published results.

6.1 Aqueous Solubility Prediction

In order to train and test the UGRNN approach, we use 2 publicly available datasets, which has been widely used by others for benchmarking solubility prediction methods.

Through various experiments, it has been found that using octanol-water partition coefficient leads to more accurate aqueous solubility predictions. Hence, we also assess the performances

of both the UG-RNN and UG-RNN-CR models, where the input to M^Gconsists of $G_{structure}$ and the log $P_{octanol}$. This allows us to access the generalization capability of the UG-RNN and UG-RNN-CR models and better understand the kind of information contained in the vector $G_{structure}$.

Small Delaney Dataset

The Delaney dataset [16] contains 2874 molecules together with their measured aqueous solubility (logmol/L at 25 C). We use this data set to compare UG-RNN with the GSE method [24]. The GSE was obtained on the smaller set of the molecules from the original dataset, commonly known as "Small" Delaney Dataset. This dataset is also used by various kernel methods [58, 80] for benchmarking, with yields better results than GSE.

Models	R^2	RMSE	AAE
UG-RNN	0.90	0.61	0.45
UG-RNN-CR	0.86	0.75	0.52
UG-RNN+LogP	0.90	0.61	0.44
UG-RNN-CR+LogP	0.92	0.59	0.44
GSE [24]	-	-	0.47
2D Kernel (param d=2)[80]	0.91	0.61	0.44

Table 6.1: Solubility prediction performances on the Small Delaney Dataset (1144 molecules)

Results obtained the solubility prediction on Small Delaney Dataset are shown in Table 6.1. The UG-RNN+logP gives the best result among all the 4 models, though it is very close to the UG-RNN approach. The UG-RNN-CR doesn't perform as well as the UG-RNN approach. The addition of logP information leads to a significant improvement in UG-RNN-CR approach. The UG-RNN model matches or surpass the performance of the other published results, although the differences are very small.

Huuskonen Data Set

This dataset contains 1297 organic molecules selected by Huuskonen [59], listed together with their aqueous solubility values. We benchmark various UG-RNN predictor against the current state-of-the-art 3D Kernel Methods [80].

Models	R^2	RMSE	AAE
UG-RNN	0.86	0.72	0.50
UG-RNN-CR	0.77	0.95	0.66
UG-RNN+LogP	0.90	0.59	0.43
UG-RNN-CR+LogP	0.90	0.62	0.45
RBF Kernel[62]	0.90	-	-
3D Kernel [80]	0.91	0.15	0.11

Table 6.2: Solubility prediction performances on the Huuskonen Dataset (1026 molecules)

Results obtained for solubility prediction the Huuskonen Dataset are shown in Table 6.2.The results are similar to ones observed on Delaney dataset. The UG-RNN+logP model achieves the best results across all the models. Adding the logP information lead to significant improvement in both UG-RNN-CR approach, but not as much in UG-RNN approach. The result obtained by UG-RNN models are similar to other published methods.

6.2 Melting Point Prediction

We use the Karthekeyan Melting Point Data Set reported by Karthikeyan et al. [78] to access the performance of UGRNN models for melting point predictions.

Models	R^2	RMSE	AAE
UG-RNN	0.53	44.40°C	34.57°C
UG-RNN-CR	0.47	46.26°C	37.28°C
Karthikeyan et al. [78] (ANN)	0.42	52.0°C	41.3°C
2D Kernel(d=10) [80]	0.56	42.71°C	32.58°C

Table 6.3: Melting Point prediction performances on Karthekeyan Melting Point Data Set.

This dataset contains 4173 diverse compounds, with melting points in the range from 14°C to 392.5°C.

Results obtained for melting point prediction on the Karthekeyan Dataset are shown in 6.3. The result of UG-RNN method is very similar to the state-of-the-art 2D-Kernel methods. Just like the result on solubility datasets, UG-RNN-CR works worse than the UG-RNN approach.

Chapter 7

Conclusions

UG-RNN are a general class of machine learning models that maps the undirected graphs to desired property. The successful application of UG-RNN approach on aqueous solubility and melting points shows us that it's ability to automatically extract the information from the molecular representations of the graph. This has a huge advantage over the other approaches, as it completely removes the need of finding the optimal feature set for every property. For properties like aqueous solubility, where the feature sets are not known, this can lead to a significant improvement, saving time and costs.

In all the experiments, we have found that the UG-RNN-CR models have weaker predictive capabilities when compared to the UG-RNN. This can be attributed to the loss of information due to the ring contraction. This also suggests that the decrease in depth does not lead to significant improvements. The addition of logP information significantly increases the efficiency of UG-RNN-CR models but doesn't help much in the case of UG-RNN models, hence providing evidence that UG-RNN models are able to extract this information implicitly.

In future, the approach of UG-RNN can be applied to any problem in molecular biology, as long as the inputs can be represented as simple molecular graphs. The approach can be

further improved to consider the 3D information along with it 2D molecular graph.

Bibliography

[1] A. Lusci, G. Pollastri, and P. Baldi. Deep architectures and deep learning in chemoinformatics: The prediction of aqueous solubility for drug-like molecules. *J. Chem. Inf. Model.*, 53:1563–1575, 2013.

[2] B. Scholkopf and A. J. Smola. Learning with Kernels. MIT Press: Cambridge, MA, 2012.

[3] L. Ralaivola, S. J. Swamidass, H. Saigo, and P. Baldi. Neural networks. *Special Issue on Neural Networks and Kernel Methods for Structured Domains*, 18:1093–1110, 2005.

[4] C. Azencott, A. Ksikes, S. J. Swamidass, J. Chen, L. Ralaivola, and P. Baldi. J. chem. inf. model. 47:965–974, 2007.

[5] A. Ceroni, F. Costa, and P Frasconi. Bioinformatics. 23:2038–2045, 2007.

[6] P. Mah and J-P Vert. Machine learning. 75:3–35, 2009.

[7] M. Kayala, C. Azencott, J. Chen, and P. Baldi. *J. Chem. Inf. Model*, 51:2209–2222, 2011.

[8] M. Kayala and P. Baldi. *J. Chem. Inf. Model*, 52:25262540, 2012.

[9] H. V. D. Waterbeemd and E Gifford. *Nature Reviews*, 2:192–204, 2003.

[10] A. Starita, A. Micheli, and A. Sperduti. *J. Chem. Inf. Comput. Sci*, 1:202–218, 2000.

[11] H Fhner. *Ber. Dtsch. Chem. Ges*, 57B:510–515, 1924.

[12] M. Hewitt, M. T. D. Cronin, S. J. Enoch, J. C. Madden, and J. C. Roberts, D. W.and Dearden. *J. Chem. Inf. Model*, 49:2572–2587, 2009.

[13] J. Reynolds, D. B. Gilbert, and C Tanford. *Proc. Natl. Acad. Sci. U.S.A*, 71:2925–2927, 1974.

[14] C. Hansch, J. E. Quinlan, and G. Lawrence. *L. J. Org. Chem*, 33:347–350, 1968.

[15] B. Faller and P Ertl. *Adv. Drug Delivery Rev*, 59:533–545, 2007.

[16] J. S. Delaney. *J. Chem. Inf. Comput. Sci*, 44:1000–1005, 2003.

[17] S. H. Yalkowsky and S. C. Valvani. *J. Pharm. Sci.*, 69:912–922, 1980.

[18] M.J. Kamlet, R. M. Doherty, J-L. M. Abboud, M. H. Abraham, and R. W. Taft. *J. Pharm. Sci*, 75:338–348, 1986.

[19] M. Randic. *J. Am. Chem. Soc.*, 97:6609–6615, 1975.

[20] L. B. Kier and L. H Hall. *Molecular Connectivity in Chemistry and Drug Design*. Academic Press: New York, 1976.

[21] L. B. Kier and L. H Hall. *Molecular Connectivity in Structure Activity Analysis*. John Wiley & Sons: New York, 1986.

[22] A. Leo, C. Hansch, and D. Elkins. *Chem. Rev.*, 71:525–616, 1971.

[23] A. Leo. *Chem. Rev.*, pages 1281–1306, 1993.

[24] N. Jain and S. Yalkowsky. *J. Pharm. Sci.*, 90:234–252, 2001.

[25] H. Timmerman, R. Todeschini, V. Consonni, R. Mannhold, and H. Kubiny. *Handbook of Molecular Descriptors*. Wiley-VCH: Weinheim, Germany, 2002.

[26] B. Louis, V. K. Agrawal, and P. V. Khadikar. *Eur. J. Med. Chem.*, 45:4018–4025, 2010.

[27] J. Dearden. *Expert Opinion in Drug Discovery*, 1:31–52, 2006.

[28] R. M. Dannenfelser, M. Paric, M. White, and S Yalkowsky. *Chemosphere*, 23(2):141–165, 1991.

[29] W. Jorgensen and E. Duffy. *Adv. Drug Delivery Rev.*, 54(30):355–366, 2002.

[30] G. Hinton, S. Osindero, and Y. Teh. *Neural Comput.*, 18:1527–1554, 2006.

[31] Bengio Y and LeCun Y. *Scaling learning algorithms towards AI*. 2007.

[32] Lee H, Grosse R, Ranganath R, and Ng A. *Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations*. 2009.

[33] Lee H, Pham P, Largman Y, Ng A., Bengio Y, Schuurmans D, Lafferty J, Williams CKI, and Culotta A. *Advances in Neural Information Processing Systems*, 22:1096–1104, 2009.

[34] G. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. 2012.

[35] P. Di Lena, K. Nagata, and P Baldi. Improving neural networks by preventing co-adaptation of feature detectors. *Bioinformatics*, 28:2449–2457, 2012.

[36] A. Krizhevsky, I. Sutskever, and G. Hinton. *Advances in NeuralInformation Processing Systems*. 2012.

[37] R. Socher, J. Pennington, E. H. Huang, A. Y. Ng, and C.D Manning. *Semi-Supervised Recursive Autoencoders for Predicting Sentiment Distributions.* 2011.

[38] G. Hinton and R. Salakhutdinov. *Science*, 312:504, 2006.

[39] Y. Bengio, P. Lamblin, D. Popovici, H. Larochelle, and U. Montreal. *In Advances in Neural Information Processing Systems*, 19:153, 2007.

[40] D. Erhan, Y. Bengio, A. Courville, P.-A. Manzagol, P. Vincent, and S. Bengio. *J. Mach. Learn. Res.*, 11:625–660, 2010.

[41] P Baldi. *Designs, Codes, Cryptogr*, 65:383–403, 2012.

[42] Y. LeCun, O. Matan, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, L. D. Jackel, and H. S Baird. Handwritten zip code recognition with multilayer networks. *Proc. IEEE*, 2:35–40, 1900.

[43] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. *Proc. IEEE*, 86:2278–2324, 1998.

[44] P. Baldi, S. Brunak, P. Frasconi, G. Pollastri, and G. Soda. *Bioinformatics*, 15:937–946, 1991.

[45] P. Baldi and G. Pollastri. *J. Mach. Learn. Res*, 4:575–602, 2003.

[46] L. Wu and P. Baldi. *Neural Networks*, 21:1392–1400, 2008.

[47] D. Koller and N. Friedman. *Probabilistic Graphical Models:Principles and Techniques*. MIT Press: Cambridge, MA, 2009.

[48] P. Baldi and S. Brunak. *Bioinformatics: The Machine Learning Approach, 2nd ed.* MIT Press: Cambridge, MA, 2001.

[49] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Nature*, 323:533–536, 1986.

[50] P. Baldi. Neural networks. *IEEE Trans.*, 6:182–195, 1995.

[51] G. Pollastri and P. Baldi. *Bioinformatics*, 18:62–70, 2002.

[52] P. Baldi and G. Pollastri. *J. Mach. Learn. Res.*, 4:575–602, 2003.

[53] Y. Bengio, P. Simard, and P. Frasconi. Neural networks. *IEEE Trans*, 5(2):157–166, 1994.

[54] H. Larochelle, Y. Bengio, J. Louradour, and P. Lamblin. *J. Mach.Learn. Res*, 10:1–40, 2009.

[55] J. March. *Advanced Organic Chemistry: Reactions, Mechanisms,and Structure, 3rd ed.* New York: Wiley, 1985.

[56] A. J. Zamora. *Chem. Inf. Comput. Sci.*, 16:40–43, 1976.

[57] B. T. Fan, A. Panaye, J. P. Doucet, and A. Barbu. *J. Chem. Inf.Comput. Sci*, 33: 657–662, 1993.

[58] C. A. Azencott, A. Ksikes, S. J. Swamidass, J. H. Chen, L. Ralaivola, and P. Baldi. *J. Chem. Inf. Comput. Sci*, 47:965–974, 2007.

[59] J. Huuskonen. *J. Chem. Inf. Comput. Sci.*, 40:773–777, 2000.

[60] S. H. Yalkowsky and D. R. M. *The Arizona Database of AqueousSolubility and College of Pharmacy*. University of Arizona: Tucson, AZ, 1990.

[61] *Physical/Chemical Property Database(PHYSOPROP)*. SRC Environmental Science Center: Syracuse, NY, 1994.

[62] H. Fr hlich, J. K. Wegner, and A Zell. *QSAR Comb. Sci.*, 23:311–318, 2004.

[63] C. Bergstroem, M. Strafford, L. Lazorova, A. Avdeef, K. Luthman, and P. Artursson. *J. Med. Chem.*, 46:558–570, 2003.

[64] A.; Bergstrom C.; Zamora I.; Artursson P. Wassvik, C.; Holmen. *Eur. J. Pharm. Sci.*, 29:294–305, 2006.

[65] P. Faller, B.; Ertl. *Adv. Drug Delivery Rev.*, 59:533–545, 2007.

[66] J.; Dressman J. Glomme, A.; Maerz. *J. Pharm. Sci.*, 94:1–16, 2005.

[67] A. Linas, R. Glen, and J. Goodman. *J. Chem. Inf. Model.*, 48:1289–1303, 2008.

[68] L. Hanses, L.; Salamon. *IEEE Trans*, 12:993–1001, 1990.

[69] Marvin Beans. Chemaxon. http://chemaxon.com (accessed July 1, 2013).

[70] *Dragon Professional Software for Windows*. Milano Chemo-metrics and QSAR Research Group.

[71] M. J ONeil. *The Merck Index*. 13th ed.; Merck & Co. Inc.: Whitehouse Station, NJ, 2001.

[72] T. S. Schr eter, A. Schwaighofer, S. Mika, A. T. Laak, D. Suelze, U. Ganzer, N. Heinrich, and K.-R. M ller. *Estimating the Domain ofApplicability for Machine Learning Qsar Models: A Study on Acqueous Solubility of Drug Discovery Molecules*. Springer Science+Business Media B.V.: Dordrecht, The Netherlands, 2007.

[73] T. I. Netzeva and et al. *ATLA, Altern. Lab. Anim.*, 33 (2):1–19, 2005.

[74] I. V. Tetko, P. Bruneau, D. C. Mewes, H.-W.and Rohrer, and G. I. Poda. *Drug Discovery Today*, 11 (15/16):700–707, 2006.

[75] Tropsha A. *Variable Selection QSAR Modeling, Model Validation, and Virtual Screening.* In Annual Reports in Computational Chemistry; Spellmeyer, D. C., Ed.; Elsevier: Amsterdam, The Netherlands; Volume 2, Chapter 7, 2006.

[76] N. R. Bruneau, P.; McElroy. *J. Chem. Inf. Model.*, 46:1379–1387, 2006.

[77] A. N. Tegge, Z. Wang, J. Eickholt, and J. Cheng. *Nucleic Acids Res.*, 37:515–518, 2009.

[78] M. Karthikeyan, R. C. Glen, and A Bender. General melting point prediction based on a diverse compound data set and artificial neural networks. *J. Chem. Inf. Model.*, 45: 581–590, 2005.

[79] C. A. S. Bergstrom, U. Norinder, K. Luthman, and P. Artursson. Molecular descriptors influencing melting point and their role in classification of solid drugs. *J. Chem. Inf. Model.*, 43:1177–1185, 2003.

[80] C-A Azencott, A. Ksikes, S. J. Swamidass, J. H. Chen, L. Ralaivola, and P. Baldi. One- to four-dimensional kernels for virtual screening and the prediction of physical, chemical, and biological properties. *J. Chem. Inf. Model.*, 47(3):965–974, 2007.

[81] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `http://tensorflow.org/`. Software available from tensorflow.org.

Appendix A

Source Code

A.1 UGRNN

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import os
import logging

import matplotlib.pyplot as plt
import tensorflow as tf

import numpy as np

from ugrnn import config
from ugrnn.molecule import Molecule
from ugrnn import nn_utils
from ugrnn.utils import get_metric
```

```
logger = logging.getLogger(__name__)


class UGRNN(object):
    def __init__(self, model_name, encoding_nn_hidden_size,
        encoding_nn_output_size,
                 output_nn_hidden_size, batch_size=1, learning_rate=0.001,
                     add_logp=False):
        """Build the ugrnn model up to where it may be used for inference."""

        # logger.info("Creating the UGRNN")
        # logger.info('Initial learning rate: {:}'.format(learning_rate))

        self.model_name = model_name
        self.batch_size = batch_size

        """Create placeholders"""
        self.local_input_pls = [tf.placeholder(tf.float32, shape=[None, None,
            Molecule.num_of_features()]) for i in
                                xrange(self.batch_size)]
        self.path_pls = [tf.placeholder(tf.int32, shape=[None, None, 3]) for i
             in xrange(self.batch_size)]
        self.target_pls = [tf.placeholder(tf.float32) for i in xrange(self.
            batch_size)]
        self.logp_pls = [tf.placeholder(tf.float32) for i in xrange(self.
            batch_size)]
        self.sequence_len_pls = [tf.placeholder(tf.int32) for i in xrange(self
            .batch_size)]
        self.global_step = tf.Variable(0, name='global_step', trainable=False)
        self.global_step_update_op = tf.assign(self.global_step, tf.add(self.
            global_step, tf.constant(1)))

        """Set the hyperparameters for the model"""
```

```
self.learning_rate = learning_rate * tf.pow(config.
    learning_rate_decay_factor, tf.to_float(self.global_step),
                                                name=None)
self.encoding_nn_output_size = encoding_nn_output_size
self.encoding_nn_hidden_size = encoding_nn_hidden_size
self.encoding_nn_input_size = 4 * encoding_nn_output_size + Molecule.
    num_of_features()
self.add_logp = add_logp

if self.add_logp:
    self.output_nn_input_size = self.encoding_nn_output_size + 1
else:
    self.output_nn_input_size = self.encoding_nn_output_size
self.output_nn_hidden_size = output_nn_hidden_size

self.initializer_fun = nn_utils.get_initializer(config.initializer)

self.flattened_idx_offsets = [(tf.range(0, self.sequence_len_pls[i]) *
    config.max_seq_len * 4) for i in
                               xrange(0, self.batch_size)]

self.trainable_variables = []
self.create_UGRNN_variable()

prediction_op = self.add_prediction_op(self.local_input_pls[0],
                                       self.path_pls[0],
                                       self.logp_pls[0],
                                       self.sequence_len_pls[0],
                                       self.flattened_idx_offsets[0])
self.prediction_ops = [prediction_op]
for i in xrange(1, self.batch_size):
    with tf.control_dependencies([self.prediction_ops[i - 1]]):
        prediction_op = self.add_prediction_op(self.local_input_pls[i
```

```
                ],
                                                    self.path_pls[i],
                                                    self.logp_pls[i],
                                                    self.sequence_len_pls[i
                                                        ],
                                                    self.
                                                        flattened_idx_offsets
                                                        [i])
            self.prediction_ops.append(prediction_op)

    self.loss_op = self.add_loss_op()
    self.train_op = self.add_training_ops()

def create_UGRNN_variable(self):
    with tf.variable_scope("EncodingNN") as scope:
        contextual_features = tf.get_variable("contextual_features",
                                              [config.max_seq_len * config
                                                  .max_seq_len * 4,
                                               self.
                                                   encoding_nn_output_size
                                                   ],
                                              dtype=tf.float32,
                                              initializer=tf.
                                                  constant_initializer(0),
                                              trainable=False)

        with tf.variable_scope('hidden1') as scope:
            weights = nn_utils.weight_variable([self.
                encoding_nn_input_size,
                                                self.encoding_nn_hidden_size
                                                   ],
                                               initializer=self.
                                                   initializer_fun)
```

```
            biases = nn_utils.bias_variable([self.encoding_nn_hidden_size
                ])
            self.trainable_variables.append(weights)
            self.trainable_variables.append(biases)

        with tf.variable_scope('output') as scope:
            weights = nn_utils.weight_variable(
                [self.encoding_nn_hidden_size,
                 self.encoding_nn_output_size],
                initializer=self.initializer_fun)

            biases = nn_utils.bias_variable([self.encoding_nn_output_size
                ])
            self.trainable_variables.append(weights)
            self.trainable_variables.append(biases)

    with tf.variable_scope("OutputNN") as scope:
        with tf.variable_scope('hidden1') as scope:
            weights = nn_utils.weight_variable(
                [self.output_nn_input_size, self.output_nn_hidden_size],
                self.initializer_fun, 'weights_decay')

            biases = nn_utils.bias_variable([self.output_nn_hidden_size])
            self.trainable_variables.append(weights)
            self.trainable_variables.append(biases)

        with tf.variable_scope('output') as scope:
            weights = nn_utils.weight_variable(
                [self.output_nn_hidden_size, 1],
                self.initializer_fun, 'weights_decay')
            self.trainable_variables.append(weights)

def add_prediction_op(self, feature_pl, path_pl, logp_pl, sequence_len,
```

```
flattened_idx_offset):
 with tf.variable_scope("EncodingNN", reuse=True) as scope:
     step = tf.constant(0)
     contextual_features = tf.get_variable("contextual_features")
     contextual_features = contextual_features.assign(
         tf.zeros([config.max_seq_len * config.max_seq_len * 4,
                   self.encoding_nn_output_size],
                  dtype=tf.float32))

     _, step, _, _, _, contextual_features, _ = tf.while_loop(
         UGRNN.cond, UGRNN.body,
         [sequence_len, step, feature_pl,
          path_pl,
          flattened_idx_offset, contextual_features,
          self.encoding_nn_output_size],
         back_prop=True,
         swap_memory=False, name=None)

     # use flattened indices1
     step_contextual_features = UGRNN.get_contextual_feature(
         contextual_features=contextual_features,
         index=0,
         flattened_idx_offset=flattened_idx_offset,
         encoding_nn_output_size=self.encoding_nn_output_size)

     indices = tf.pack([tf.range(0, sequence_len), tf.range(0,
        sequence_len)], axis=1)
     step_feature = tf.gather_nd(feature_pl, indices)

     inputs = tf.concat(1, [step_contextual_features, step_feature])
     encodings = UGRNN.apply_EncodingNN(inputs, config.activation_type)

     molecule_encoding = tf.expand_dims(tf.reduce_sum(encodings, 0), 0)
```

```
            x = tf.expand_dims(logp_pl, 0)
            x = tf.expand_dims(x, 1)
            if self.add_logp:
                outputNN_input = tf.concat(1, [x, molecule_encoding])
            else:
                outputNN_input = molecule_encoding

        with tf.variable_scope("OutputNN", reuse=True) as scope:
            prediction_op = UGRNN.apply_OutputNN(outputNN_input,
                                                 config.activation_type)

        return prediction_op

    @staticmethod
    def cond(sequence_len, step, feature_pl, path_pl, flattened_idx_offset,
             contextual_features, encoding_nn_output_size):
        return tf.less(step, sequence_len - 1)

    @staticmethod
    def body(sequence_len, step, feature_pl, path_pl, flattened_idx_offset,
             contextual_features, encoding_nn_output_size):
        zero = tf.constant(0)
        one = tf.constant(1)
        input_begin = tf.pack([zero, step, zero])

        input_idx = tf.slice(path_pl, input_begin, [-1, 1, 1])
        input_idx = tf.reshape(input_idx, [-1])

        indices = tf.pack([tf.range(0, sequence_len), input_idx], axis=1)
        step_feature = tf.gather_nd(feature_pl, indices)

        output_begin = tf.pack([zero, step, one])
        tf.get_variable_scope().reuse_variables()
```

```
        contextual_features = tf.get_variable("contextual_features")

        step_contextual_features = UGRNN.get_contextual_feature(
            contextual_features=contextual_features,
            index=input_idx,
            flattened_idx_offset=flattened_idx_offset,
            encoding_nn_output_size=encoding_nn_output_size)

        nn_inputs = tf.concat(1, [step_contextual_features, step_feature])
        updated_contextual_vectors = UGRNN.apply_EncodingNN(nn_inputs,
                                                            config.
                                                                activation_type
                                                                )
        output_idx = tf.squeeze(tf.slice(path_pl, output_begin, [-1, 1, 2]))

        contextual_features = UGRNN.update_contextual_features(
            contextual_features=contextual_features,
            indices=output_idx,
            updates=updated_contextual_vectors,
            flattened_idx_offset=flattened_idx_offset)

        with tf.control_dependencies([contextual_features]):
            return (sequence_len,
                    step + 1,
                    feature_pl,
                    path_pl,
                    flattened_idx_offset,
                    contextual_features,
                    encoding_nn_output_size)

    def add_training_ops(self):
        def apply_gradient_clipping(gradient):
```

```
            if gradient is not None:
                return tf.mul(tf.clip_by_value(tf.abs(grad), 0.1, 1.),
                              tf.sign(grad))
            else:
                return None

        # optimizer = tf.train.GradientDescentOptimizer(learning_rate=self.
            learning_rate)
        optimizer = tf.train.AdamOptimizer(learning_rate=self.learning_rate,
                                           beta1=0.9, beta2=0.999,
                                           epsilon=1e-08,
                                           use_locking=False, name='Adam')

        loss_op = self.loss_op + config.weight_decay_factor * tf.add_n(
            [tf.nn.l2_loss(v) for v in tf.get_collection('weights_decay')])

        gvs = optimizer.compute_gradients(loss_op)

        if config.clip_gradient:
            gvs = [(apply_gradient_clipping(grad), var) for grad, var in gvs]

        train_op = optimizer.apply_gradients(gvs)

        return train_op

    def add_loss_op(self):
        loss_op = [
            tf.square(tf.sub(self.prediction_ops[i], self.target_pls[i])) for
                i
            in xrange(0, self.batch_size)]
        loss_op = tf.add_n(loss_op, name=None) / 2
        return loss_op
```

```
def train(self, sess, epochs, train_dataset, validation_dataset):

    train_metric = self.evaluate(sess, train_dataset)
    validation_metric = self.evaluate(sess, validation_dataset)

    plt.subplot(2, 1, 1)
    plt.title('Training data set')
    plt.axis([0, epochs, 0, train_metric[0]])

    plt.subplot(2, 1, 2)
    plt.title('Vaidation data set')
    plt.axis([0, epochs, 0, validation_metric[0]])

    plt.ion()
    logger.info('Start Training')

    steps_in_epoch = train_dataset.num_examples // self.batch_size

    for epoch in xrange(0, epochs):
        for i in xrange(0, steps_in_epoch):
            feed_dict = self.fill_feed_dict(train_dataset, self.batch_size
                )
            _ = sess.run([self.train_op], feed_dict=feed_dict)

        train_dataset.reset_epoch(permute=True)

        sess.run([self.global_step_update_op])

        if epoch % 10 == 0:
            train_metric = self.evaluate(sess, train_dataset)
            validation_metric = self.evaluate(sess, validation_dataset)
            plt.subplot(2, 1, 1)
            plt.scatter(epoch, train_metric[0], color='red', marker=".")
```

```
            plt.scatter(epoch, train_metric[1], color='blue', marker=".")

            plt.subplot(2, 1, 2)
            plt.scatter(epoch, validation_metric[0], color='red', marker="
                .")
            plt.scatter(epoch, validation_metric[1], color='blue', marker=
                ".")
            learning_rate = self.get_learning_rate(sess)
            plt.pause(0.05)
            logger.info(
                "Epoch: {:}, Learning rate {:.8f}  Train RMSE: {:.4f}, 
                    Train AAE: {:.4f} Validation RMSE {:.4f}, Validation 
                    AAE {:.4f}".
                     format(epoch, learning_rate[0], train_metric[0],
                            train_metric[1], validation_metric[0],
                            validation_metric[1],
                            precision=8))

    logger.info('Training Finished')

def evaluate(self, sess,  dataset):
    predictions = self.predict(sess, dataset)
    targets = dataset.labels
    return get_metric(predictions, targets)

def predict(self, sess, dataset):
    dataset.reset_epoch()
    predictions = np.empty(dataset.num_examples)
    for i in xrange(0, dataset.num_examples):
        feed_dict = self.fill_feed_dict(dataset, 1)
        prediction_value = sess.run([self.prediction_ops[0]], feed_dict=
            feed_dict)
        predictions[i] = np.mean(prediction_value)
```

```
        return predictions

    def fill_feed_dict(self, dataset, batch_size):
        assert batch_size <= self.batch_size
        molecules_feeds, targets_feeds = dataset.next_batch(batch_size)
        feed_dict = {}
        for i in xrange(batch_size):
            feed_dict[self.local_input_pls[i]] = molecules_feeds[i].
                local_input_vector
            feed_dict[self.path_pls[i]] = molecules_feeds[i].directed_graphs
            feed_dict[self.target_pls[i]] = targets_feeds[i]
            feed_dict[self.sequence_len_pls[i]] = molecules_feeds[i].
                local_input_vector.shape[1]

            if self.add_logp:
                feed_dict[self.logp_pls[i]] = molecules_feeds[i].logp

        return feed_dict

    def get_learning_rate(self, sess):
        return sess.run([self.learning_rate])

    def save_model(self, sess, checkpoint_dir, step):
        logging.info("Saving model {:}".format(self.model_name))
        saver = tf.train.Saver(self.trainable_variables, max_to_keep=1)
        checkpoint_file = os.path.join(checkpoint_dir, 'model.ckpt')
        saver.save(sess, save_path=checkpoint_file)

    def restore_model(self, sess, checkpoint_dir):
        # logging.info("Restoring model {:}".format(self.model_name))
        saver = tf.train.Saver(self.trainable_variables)
        saver.restore(sess, tf.train.latest_checkpoint(checkpoint_dir))
```

```
@staticmethod
def get_contextual_feature(contextual_features, index,
                           flattened_idx_offset, encoding_nn_output_size):
    """
    Contextual vector is flatted array
        index is 1D index with
    """
    indices = index + flattened_idx_offset
    values = [indices, indices, indices, indices]
    indices = tf.pack(values, axis=1, name='pack')
    indices = indices + tf.constant([0, 1, 2, 3])
    indices = tf.reshape(indices, [-1])
    contextual_vector = tf.gather(contextual_features, indices)
    contextual_vector = tf.reshape(contextual_vector,
                                   [-1, 4 * encoding_nn_output_size])
    return contextual_vector

@staticmethod
def update_contextual_features(contextual_features, indices, updates,
                               flattened_idx_offset):
    first_indices, second_indices = tf.split(1, 2, indices)
    indices = tf.squeeze(first_indices + second_indices)
    indices = indices + flattened_idx_offset
    contextual_features = tf.scatter_add(contextual_features, indices,
                                         updates, use_locking=None)
    return contextual_features

@staticmethod
def apply_EncodingNN(inputs, activation_type):
    activation_fun = nn_utils.get_activation_fun(activation_type)
    with tf.variable_scope('hidden1') as scope:
        weights = tf.get_variable("weights")
        biases = tf.get_variable("biases")
```

```
            hidden1 = activation_fun(tf.matmul(inputs, weights) + biases)

        with tf.variable_scope('output') as scope:
            weights = tf.get_variable("weights")
            biases = tf.get_variable("biases")
            return activation_fun(tf.matmul(hidden1, weights) + biases)

    @staticmethod
    def apply_OutputNN(inputs, activation_type):
        activation_fun = nn_utils.get_activation_fun(activation_type)
        with tf.variable_scope('hidden1') as scope:
            weights = tf.get_variable("weights")
            biases = tf.get_variable("biases")
            hidden1 = activation_fun(tf.matmul(inputs, weights) + biases)

        with tf.variable_scope('output') as scope:
            weights = tf.get_variable("weights")
            return tf.matmul(hidden1, weights)
```

A.2 Training

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import argparse
import logging
import os

import numpy as np
```

```
from ugrnn.input_data import DataSet
from ugrnn.ugrnn import UGRNN
from ugrnn.utils import model_params
np.set_printoptions(threshold=np.inf, precision=4)

import tensorflow as tf

FLAGS = None


def main(_):
    model_dir = os.path.join(FLAGS.output_dir, FLAGS.model_name)

    if tf.gfile.Exists(model_dir):
        tf.gfile.DeleteRecursively(model_dir)
    tf.gfile.MakeDirs(model_dir)

    with tf.Graph().as_default():
        # Create a session for running Ops on the Graph.
        sess = tf.Session()

        logp_col_name = FLAGS.logp_col if FLAGS.add_logp else None

        logger.info('Loading Training dataset from {:}'.format(FLAGS.
            training_file))
        train_dataset = DataSet(csv_file_path=FLAGS.training_file,
            smile_col_name=FLAGS.smile_col,
                                target_col_name=FLAGS.target_col,
                                    logp_col_name=logp_col_name,
                                contract_rings=FLAGS.contract_rings)

        logger.info('Loading validation dataset from {:}'.format(FLAGS.
            validation_file))
```

```
        validation_dataset = DataSet(csv_file_path=FLAGS.validation_file,
            smile_col_name=FLAGS.smile_col,
                                     target_col_name=FLAGS.target_col,
                                         logp_col_name=logp_col_name,
                                     contract_rings=FLAGS.contract_rings)

        logger.info("Creating Graph.")

        ugrnn_model = UGRNN(FLAGS.model_name, encoding_nn_hidden_size=FLAGS.
            model_params[0],
                            encoding_nn_output_size=FLAGS.model_params[1],
                                output_nn_hidden_size=FLAGS.model_params[2],
                            batch_size=FLAGS.batch_size, learning_rate=0.001,
                                add_logp=FLAGS.add_logp)

        logger.info("Succesfully created graph.")

        init = tf.global_variables_initializer()
        sess.run(init)
        logger.info('Run the Op to initialize the variables')
        ugrnn_model.train(sess, FLAGS.max_epochs, train_dataset,
            validation_dataset)

        ugrnn_model.save_model(sess, model_dir, FLAGS.max_epochs)


if __name__ == '__main__':
    log_format = '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
    logging.basicConfig(level=logging.INFO, format=log_format)
    logger = logging.getLogger(__name__)

    parser = argparse.ArgumentParser()
```

```
parser.add_argument('--model_name', type=str, default='default_model',
                    help='Name of the model')

parser.add_argument('--max_epochs', type=int, default=300,
                    help='Number of epochs to run trainer.')

parser.add_argument('--batch_size', type=int, default=10,
                    help='Batch size.')

parser.add_argument('--model_params', help="Model Parameters", dest="
    model_params", type=model_params)

parser.add_argument('--learning_rate', type=float, default=0.001,
                    help='Initial learning rate')

parser.add_argument('--output_dir', type=str, default='train',
                    help='Directory for storing the trained models')

parser.add_argument('--training_file', type=str, default='ugrnn/data/
    delaney/train_delaney.csv',
                    help='Path to the csv file containing training data
                        set')

parser.add_argument('--validation_file', type=str, default='ugrnn/data/
    delaney/validate_delaney.csv',
                    help='Path to the csv file containing validation data
                        set')

parser.add_argument('--smile_col', type=str, default='smiles')

parser.add_argument('--logp_col', type=str, default='logp')

parser.add_argument('--target_col', type=str, default='solubility')
```

```
    parser.add_argument('--contract_rings', dest='contract_rings',
                        action='store_true')
    parser.set_defaults(contract_rings=False)

    parser.add_argument('--add_logp', dest='add_logp',
                        action='store_true')
    parser.set_defaults(add_logp=False)

    parser.add_argument('--clip_gradient', dest='clip_gradient',
                        action='store_true')
    parser.set_defaults(clip_gradient=False)

    FLAGS = parser.parse_args()

    tf.app.run(main=main)
```

A.3 Prediction

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

from sklearn import linear_model
from sklearn.ensemble import RandomForestRegressor

import argparse
import logging
import os

import numpy as np
```

```
from ugrnn.input_data import DataSet
from ugrnn.ugrnn import UGRNN
from ugrnn.utils import model_params, get_metric

np.set_printoptions(threshold=np.inf)

import tensorflow as tf

FLAGS = None


def save_results(file_path, targets, predictions):
    data = np.array([targets, predictions])
    data = data.T
    f = open(file_path, 'w+')
    np.savetxt(f, data, delimiter=',', fmt=['%.4f', '%.4f'], header="Target, 
        Prediction", comments="")
    f.close()


def get_prediction_from_model(model_name, encoding_nn_hidden_size,
    encoding_nn_output_size,
                                  output_nn_hidden_size, test_dataset,
                                      validation_dataset):
    model_dir = os.path.join(FLAGS.output_dir, model_name)

    if not tf.gfile.Exists(model_dir):
        raise Exception("Invalid path or the model paramter doesnot exist")

    with tf.Graph().as_default():
        # Create a session for running Ops on the Graph.
        sess = tf.Session()
```

```
        # logger.info("Creating Graph.")

        ugrnn_model = UGRNN(model_name, encoding_nn_hidden_size=
            encoding_nn_hidden_size,
                            encoding_nn_output_size=encoding_nn_output_size,
                            output_nn_hidden_size=output_nn_hidden_size,
                            add_logp=FLAGS.add_logp)

        # logger.info("Succesfully created graph.")

        init = tf.global_variables_initializer()
        sess.run(init)
        # logger.info('Run the Op to initialize the variables')

        # logger.info('Restoring model parameters')
        ugrnn_model.restore_model(sess, model_dir)

        prediction_validate = ugrnn_model.predict(sess, validation_dataset)
        prediction_test = ugrnn_model.predict(sess, test_dataset)

    test_results_file_path = os.path.join(model_dir, "test_result.csv")
    validation_results_file_path = os.path.join(model_dir, "validation_result.
        csv")

    save_results(test_results_file_path, test_dataset.labels, prediction_test)
    save_results(validation_results_file_path, validation_dataset.labels,
        prediction_validate)

    return prediction_validate, prediction_test


def ensemble_prediction_linear_regression(validation_dataset,
    all_validation_predictions, all_test_predictions):
```

```
    lr = linear_model.LinearRegression(fit_intercept=False)
    lr.fit(all_validation_predictions.T, validation_dataset.labels)
    emsemble_preditions = lr.predict(all_test_predictions.T)
    # print("Liner Regression Weights: ", lr.coef_)
    return emsemble_preditions


def ensemble_prediction_rf_regression(validation_dataset,
    all_validation_predictions, all_test_predictions):
    rfr = RandomForestRegressor(n_estimators=1000)
    rfr.fit(all_validation_predictions.T, validation_dataset.labels)
    emsemble_preditions = rfr.predict(all_test_predictions.T)
    return emsemble_preditions


def ensemble_prediction_average(validation_dataset, all_validation_predictions
    , all_test_predictions):
    emsemble_preditions = np.mean(all_test_predictions, axis=0)
    return emsemble_preditions


def ensemble_prediction_top_k(validation_dataset, all_validation_predictions,
    all_test_predictions, k=10):
    no_of_models = len(all_validation_predictions)
    errors = []
    for i in xrange(0, no_of_models):
        metric = get_metric(all_validation_predictions[i], validation_dataset.
            labels)
        errors.append(metric[0])

    errors = np.array(errors)
    index_of_best_networks = errors.argsort()[:k]
    # logging.info("Top {:} models: {:}".format(k, index_of_best_networks))
```

```
    emsemble_preditions = np.mean(all_test_predictions[index_of_best_networks
        ], axis=0)
    return emsemble_preditions

def ensemble_prediction_greedy(validation_dataset, all_validation_predictions,
    all_test_predictions):
    current_prediction = np.zeros(len(all_validation_predictions[0]))
    index = 0
    index_of_best_networks = []

    index_of_next_best = get_next_best_model(index, current_prediction,
        all_validation_predictions,validation_dataset.labels)

    while index_of_next_best != -1:
        index_of_best_networks.append(index_of_next_best)
        current_prediction = (index * current_prediction +
            all_validation_predictions[index_of_next_best]) / (index + 1)
        index+=1
        index_of_next_best = get_next_best_model(index, current_prediction,
            all_validation_predictions,
                                                 validation_dataset.labels)

    logging.info("Best models: {:}".format(index_of_best_networks))
    emsemble_preditions = np.mean(all_test_predictions[index_of_best_networks
        ], axis=0)
    return emsemble_preditions


def get_next_best_model(index, current_prediction, all_predictions, targets):
    no_of_models = len(all_predictions)

    current_error = (get_metric(current_prediction, targets))[0]
    next_best_model_index = -1
```

```
    for i in xrange(0, no_of_models):
        temp_prediction = (index * current_prediction + all_predictions[i]) /
            (index + 1)
        metric = get_metric(temp_prediction, targets)
        if metric[0] < current_error:
            next_best_model_index = i
            current_error = metric[0]

    return next_best_model_index


def main(_):
    logger.info('Loading Models From {:}'.format(FLAGS.output_dir))

    logp_col_name = FLAGS.logp_col if FLAGS.add_logp else None
    test_dataset = DataSet(csv_file_path=FLAGS.test_file, smile_col_name=FLAGS
        .smile_col,
                           target_col_name=FLAGS.target_col, logp_col_name=
                               logp_col_name,
                           contract_rings=FLAGS.contract_rings)


    validation_dataset = DataSet(csv_file_path=FLAGS.validation_file,
        smile_col_name=FLAGS.smile_col,
                                 target_col_name=FLAGS.target_col,
                                     logp_col_name=logp_col_name,
                                 contract_rings=FLAGS.contract_rings)

    validation_predictions = np.empty((len(FLAGS.model_names),
        validation_dataset.num_examples))
    test_predictions_ = np.empty((len(FLAGS.model_names), test_dataset.
        num_examples))
```

```
for i in xrange(0, len(FLAGS.model_names)):
    predictions = get_prediction_from_model(FLAGS.model_names[i], FLAGS.
        model_params[i][0],
                                            FLAGS.model_params[i][1],
                                                FLAGS.model_params[i][2],
                                            test_dataset,
                                                validation_dataset)

    validation_predictions[i, :] = predictions[0]
    test_predictions_[i, :] = predictions[1]
ensemble_predictor = [ensemble_prediction_rf_regression,
    ensemble_prediction_top_k, ensemble_prediction_greedy]
predictor_names = [ "Random forest regression", "Top 10", "Greedy"]

for fun, name in zip(ensemble_predictor, predictor_names):
    emsemble_preditions = fun(validation_dataset, validation_predictions,
         test_predictions_)
    prediction_metric = get_metric(emsemble_preditions, test_dataset.
        labels)
    logger.info("Method {:} RMSE: {:}, AAE: {:}, R: {:}".format(name,
        prediction_metric[0], prediction_metric[1],
                                                                prediction_metric
                                                                    [2]))


final_prediction_path = os.path.join(FLAGS.output_dir, "
    ensemble_test_prediction.csv")
save_results(final_prediction_path, test_dataset.labels,
    emsemble_preditions)
logging.info("-------------------------------DONE
    ---------------------------------")
logging.info("")
logging.info("")
```

```
if __name__ == '__main__':
    log_format = '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
    logging.basicConfig(level=logging.INFO, format=log_format)
    logger = logging.getLogger(__name__)

    parser = argparse.ArgumentParser()

    parser.add_argument('--model_names', nargs='+', type=str,
                        help='Name of the models used for prediction')

    parser.add_argument('--model_params', help="Model Parameters", dest="
        model_params", type=model_params, nargs='+')

    parser.add_argument('--output_dir', type=str, default='train',
                        help='Root Directory where the  model parameters are
                            stored')

    parser.add_argument('--test_file', type=str, default='ugrnn/data/delaney/
        validate_delaney.csv',
                        help='Path to the csv file containing test data set')

    parser.add_argument('--validation_file', type=str, default='ugrnn/data/
        delaney/test_delaney.csv',
                        help='Path to the csv file containing validation data
                            set')

    parser.add_argument('--smile_col', type=str, default='smiles')

    parser.add_argument('--logp_col', type=str, default='logp')

    parser.add_argument('--target_col', type=str, default='solubility')
```

```
parser.add_argument('--contract_rings', dest='contract_rings',
                    action='store_true')
parser.set_defaults(contract_rings=False)

parser.add_argument('--add_logp', dest='add_logp',
                    action='store_true')
parser.set_defaults(add_logp=False)

parser.add_argument('--optimize_ensemble', dest='optimize_ensemble',
                    action='store_true')
parser.set_defaults(optimize_ensemble=False)

FLAGS = parser.parse_args()
assert len(FLAGS.model_params) == len(FLAGS.model_names)

tf.app.run(main=main)
```