An explanation of the paper - "Convolutional Networks on Graphs for Learning Molecular Fingerprints"

Varun Kumar
BITS-Pilani, K.K. Birla Goa Campus

July 15, 2016

Contents

0.1 Introduction

This report explains the working of the code from the paper "Convolutional Networks on Graphs for Learning Molecular Fingerprints". This work aims towards extracting fixed length features of arbitrary sized graphs. Though the title of the paper says 'Convolution Networks', the network that thay have constructed is quite different from the traditional Covolution Networks which we come across while working with images or audio signals. Author calls it convolution network because same operation is performed over all the atoms in a molecule, this is, they say, is similar to using the same filter ot multiple places of the image. As the next layer of the convolutional network gets a larger receptive field of the image as compared to the previous layer, here also, with every new layer(or increasing radius) the receptive field of a single 'neuron' on the atoms is increased. I have considered the molecular package RDKit used in the paper as a black box. It was used to construct graph from SMILES representation of molecules and extract the handcrafted features of atoms and bonds. It was also used to create images of the molecules. They formulated the problem of extracting fixed sized features as a regression problem for predicting solubility which generates these fixed sized features in between.

0.2 Dataset

They have used the solubility dataset [1] to extract the features and predict solubility of compounds. SMILES representation of molecules were used for training with 'measured solubility in moles per litre' as target labels. A snapshot of the dataset used is shown in the Table 1. Each SMILE is converted to a graph with the nodes representing atoms and edges representing bonds. Handcrafted features are extracted from each node and edge and is called *atom_features* and *bond_features* respectively. A tensor is created of these *atom_features* for all the nodes of graphs/molecules taken in the training set/minibatch. This *input_tensor* of size (#Nodes, length(atom_features)) is used as the input to neural network. Hence each atom becomes a unique node i.e. the same carbon atom present in different molecules will be referred to as different nodes. From now onwards I will refer atom as node and molecule as graph. A dictionary is created which stores the data used in training. This dictionary is named array_rep (array representation of atoms and bonds) which contains the following key value pairs:

1. array_rep['atom_features'] - The value for this is a matrix of size (#Nodes, length(atom_features)). It stores the node features (of the dataset) sorted according to their degrees.
2. array_rep['bond_features'] - The value for this is a matrix of size (#Edges, length(bond_features)). It stores the edge features (of the dataset) sorted according to their degrees.
3. array_rep['atom_list'] - The value for this is a matrix of size (#SMILES, x); where the value of x is variable and it represents the index of nodes in each SMILE/graph/compound/molecule. Each index points to the index of atom in array_rep['atom_features']

4. array_rep['rdkit_ix'] - RDKit, after converting each SMILE to a graph, numbers each node in it (graph) to a unique number (number is unique within the same graph but can be reused among multiple graphs). The value for this dictionary key is an array which contains numerical identifier of the nodes set by RDKit which gives its position in the graph to which each of them belongs.

5. array_rep[('atom_neighbors',**degree**)] - This gives the atom/node neighbors of all the atoms/nodes in the dataset for the particular **degree**. Its size is (#Nodes of degree = **degree** , y); where the value of y is variable and it represents the neighbouring directly bonded nodes.

6. array_rep[('bond_neighbors',**degree**)] - It gives the matrix of size (#Nodes of degree = **degree** , z); where the value of z is variable and it represents the neighbouring bonds (not quite sure about z). See array_rep_from_smiles() in build_convnet.py for implementation details regarding all the above points in this section.

Table 1: Snapshot of dataset used

Smiles	**Solubility**
OCC3OC(OCC2OC(OC(C#N)c1ccccc1)C(O)C(O)C2O)C(O)C(O)C3O	-0.77
Cc1occc1C(=O)Nc2ccccc2	-3.3
CC(C)=CCCC(C)=CC(=O)	-2.06
c1ccc2c(c1)ccc3c2ccc4c5ccccc5ccc43	-7.87
c1ccsc1	-1.33

0.3 Architecture

They created the architecture to extract a fixed size fingerprint of a molecule and then used it to predict its solubility using another neural network. In place of a neural network, a linear regressor can also be used (See figure 1) for the task of prediction. They trained the network end to end thereby generating the features in between. I like to call this architecture a 2 way neural network - having a horizontal pass and a vertical pass.

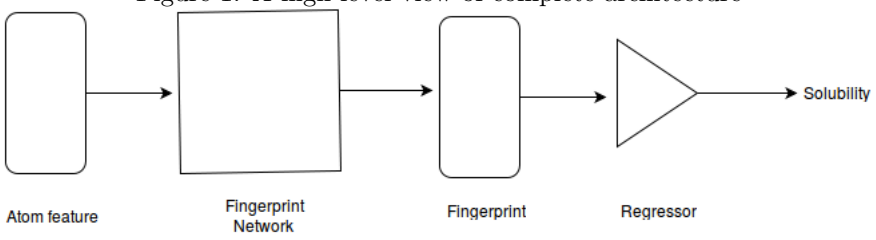
A 4 layered neural net aka fingerprint network is used for feature extraction. Layers are - [Input_Layer, Hidden_Layer1, Hidden_Layer2, Hidden_Layer3]. The layer and radius is used interchangably in this report. This radius is analogous to the radius for circular fingerprints. At radius 0, only the atom itself is considered. At level 1, atom along with its immediate bonded neighbours is considered. At level 2, atom along with its neighbours and its neighbours' neighbours are considered and so on.

Input tensor is forward propogated via these layers described above. - I call this a horizontal pass. The reason behind forward propogating the whole tensor at once is that after every layer you need to update the features of all the atom nodes in the tensor together. Because at each update, the atom features are summed up with the neighbouring atom features i.e. after every layer/radius the atom node contains a representation of its neighbouring atom nodes as well.

Therefore it is manditory that all the neighbouring nodes have the features of the same layer/radius.

Now you may wonder where is the output layer. There is no single output layer like a traditional neural network has. Instead, output is calculated at each layer (including the Input_Layer). - I call this the vertical pass.

Figure 1: A high level view of complete architecture



0.4 Methodology

Horizontal Pass:

In horizontal pass, outputs of the previous layer is updated and tranformed as inputs of the next layer. So there are 3 such transformations as per the architecture shown in the previous section. Let us look at what happens at each transformation.

1. At each layer, the atom/node features of previous layer is first transformed by some weight matrix aka *self filter* (which changes the size of the features to size of the next layer), resulting in *self activations*.

2. These *self activations* are summed up with the transformed features of its neighbouring nodes and edges. See update_layer() in build_convnet.py for actual implementation realted to point 1 and 2.

3. Now, to find the transformed features of neighbours for each node, first the neighboring nodes and the corresponding edges/bonds of each node is found. Then their feature vectors are found using RDKit. These feature vectors are then concatenated. These concatenated features are then transformed by a weight matrix aka *filter* to give *activations by degree*. Degree of an atom is defined to be its number of directly-bonded neighbors. As any atom can have maximum of 5 degrees (for organic molecules), we need 5 different *fiters*.

4. Another point to note here is that at the time of creation, the list of nodes is sorted according to their degrees. The top node in the list is the node of degree 0 or 1 and the last node is of maximum degree of all the nodes in the dataset. For each degree, the corresponding set of nodes are selected and their neighbours are found out. Then the neighbours belonging to a particular degree node is multiplied by a particular *filter* to get *activations by degree* as mentioned in the previous point. This is done

for all the degrees. The results are then appended together one after another to get the transformed neighbour activations for each node in the array_rep['atom_features'] list. See matmult_neighbors() in build_convnet.py for actual implementation realted to point 3, 4 and 5.

5. As mentioned earlier in the second point, these transformed neighbour activations are then added to the *self activations* to get the summed activations.

6. These summed activations are then passed through an activation function (relu here) to get the final activations which has to be fed to the next layer.

Vertical Pass:

1. For vertical pass, input of the current layer is multiplied by output weight matrix aka *layer_output_weights* and is passed through a softmax which results in a fixed size vector of size *FingerPrint_size*. This results in the blue tensors shown in the Figure 2.

2. After multiplyig weight matrix with the set of atom features and getting a huge matrix of size (#Atoms, FP_size) i.e. blue tensors, we then sum up the atom activations molecule-wise. That is for every molecule, sum up the activations of its nodes. This is called *sum_and_stack* operation. This results in a grey tensor of size (#Molecules, FingerPrint_size). We get different grey tensors for each layer (including the Input_Layer).

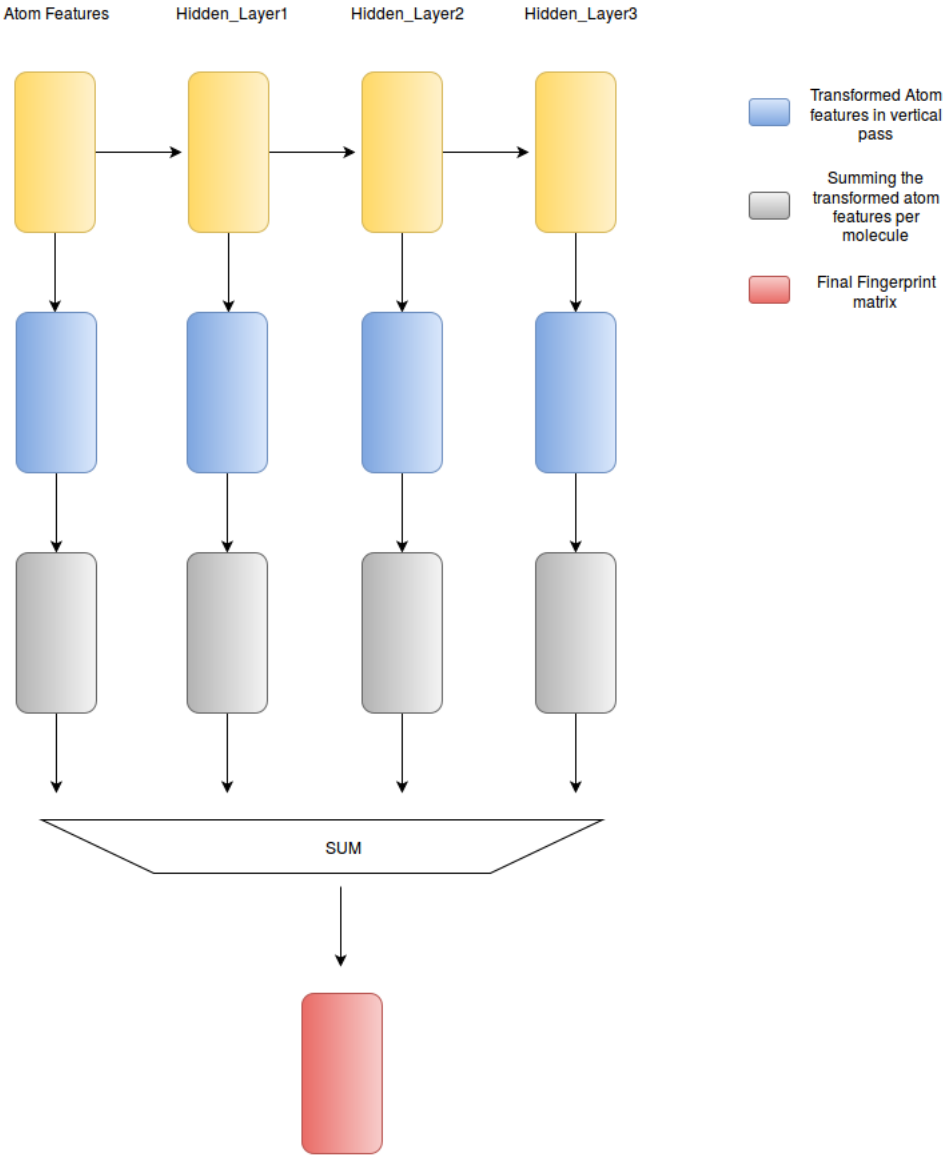
3. Next, we sum up these tensors/matrices from each layer to get a single Fingerprint matrix of size (#Molecules, FP_size) - red tensor. This is our final fingerprint output which we are going to improve later by updating the weights of the 'convolution' aka fingerprint network by gradient descent. See output_layer_fun_and_atom_activations() in build_convnet.py for actual implementation realted to point 1, 2 and 3.

So, to summarize, we can say that in total we deal with 3 types of weight matrices.

- Self filter (One for each transformation among layers)
- Filter (One for each degree)
- Output Weight matrix (One for each layer including the Input_Layer)

This fingerprint is treated as input to a linear regression module or another neural network to predict solubility. The root mean squared error in solubility is backpropogated till the start of Fingerprint network to tune the weights of the complete network along with the weights of the linear regressor. The error gradients are found out using a python based automatic differentiation package known as autograd (https://github.com/HIPS/autograd) which I have treated as a blackbox. The detailed view of fingerprint network that extracts fixed size fingerprints for each molecule can be seen in Figure 2.

Figure 2: A detailed level view of Fingerprint Network



0.5 Visualization

This section describes about the intrepretability of the features learned. To visualize which atoms/substructures contriubute the most towards solubility, they tried to find out the atoms that have the highest activations among all the layers of the fingerprint network. Before *sum_and_satck* operation they appended the blue tensors shown in the figure 2 into one long list. They then calculated, for each index of fingerprint, which atom/node for the the same index as that of fingerprint, has the highest activation. Then that node is put to the list of nodes to be highlighted. The neighbours are also put in this list based on the radius of the atom under consideration. If radius is 0, the atom/node itself is highlighted. If radius is 1, the atom/node and its neighbours are highlighted. If radius is 2, then the atom/node, its neighbours and its neighbours' neighbours are highlighted and so on. Radius refers to the layer of neural net the atom/node with the highest activation belonged to. This is done for all the fingerprint vector indexes. Instead of just the highest activation node, actually they have considered top 11 nodes and drew a seperate highlighted molecule for each of them.

I think what they mean is that to get the highest solubility, all the values of fingerprint should be high. For the values of fingerprint should be high, the corresponding atom activations should be high.

Following is the detailed procedure for selecting the relevant nodes, finding out which graph/molecule it belonged to and then plotting the molecule with the relevant atoms/nodes being highlighted.

For each index **idx** of fingerprint vector:

1. We create a list called *combined_list* that contains the tuples, each having values (activation[idx], atom_ix, radius) sorted according to activation value. The activation[idx] is the value of activation at the idx^{th} index of the transformed feature vector of the node. This combined list is actually created by appending the blue tensors shown in the figure 2 in a long list before *sum_and_satck* operation.

2. The sorted *combined_list* will give us the 3 things - the index of highest activated atom/node , its activation value and its radius.

3. Parent molecule/SMILE index is found out for that atom/node. As described earlier the neighbour nodes are found out based on the radius of the highest activated atom.

4. The index of nodes to be highlighted are the indexes of nodes in the array_rep['atom_features'] matrix. We first need to know which molecule does it belong to. This is done using dictionary value of array_rep['atom_list']. After that, it is needed to be mapped back to corresponding nodes of the parent molecule. This is done by using the dictionary array_rep['rdkit_ix'].

5. Now we know the specific molecule and the specific atoms we need to highlight. We again use the RDKit functionality to do the plotting and highlighting work. See draw_molecule_with_highlights() of visualization.py for implementation details.

Bibliography

[1] John S Delaney. Esol: estimating aqueous solubility directly from molecular structure. *Journal of chemical information and computer sciences*, 44(3):1000–1005, 2004.