Centre of Excellence in Computational
Nanoscience research,
Aalto University.

Estimation of Partial Charges on Atoms in a Molecule

Prakhar Agrawal
Computer Science and Engineering,
Indian Institute of Technology, Delhi.

Date: 15th July, 2016

Guided by:
Prof. Adam Foster
Dr. Filippo Federici Canova
Dr. Martha Arbayani Bin Zaidan

Estimation of Partial Charges on Atoms in a Molecule

August 19, 2016

Contents

1 INTRODUCTION **2**

2 The Graph Neural Network **2**
2.1 Intro 2
2.2 How it fits the problem at hand? 3

3 Dataset **4**

4 Matlab implementation **4**
4.1 Intro 4
4.2 Transition Net 4
4.3 Outnet 5
4.4 How to run the current version? 5

5 Results **6**
5.1 Limitations 7
5.2 Data representation 8

6 C/CPP Implementation **8**
6.1 How to Run? 9
6.2 Partial Results obtained on CPP implementation 9

7 Conclusions **9**

Abstract

This describes the Graph Neural Model used to machine learn a mapping between molecular structure and the partial atomic charges. Objective is to predict the partial charges on atoms given a molecule as input. This prediction is a result of the mapping learned on the training set.

1 INTRODUCTION

This report is divided into sections for easier understanding and navigation. An attempt has been made to make sections as independent of others. Section 1. is the introduction to the report and a brief description on sections to come. Section 2 describes in brief the Graph Neural Network and slight modification to it to fit our problem. Section 3 discusses the data set. Section 4 gives a brief insight into the results obtained using the GNN toolkit in matlab. This uses a rather small subset of the data set and smaller state dimensions for the reasons that will be mentioned in the section. Section 5 describes implementation of the GNN in C/CPP and its training using the Genetic Algorithm.

2 The Graph Neural Network

2.1 Intro

Many real world problems in science and engineering such as those in data mining, molecular biology, molecular physics and many other similar problems can be represented as problems on a graph data structure. These provide a perfect fit for a machine learning model that can learns and operates on graph data. We can also model problems in many other domains into a problem in the graph domain and then use the GNN approach to machine learn based on the properties of specific node and of the complete network or the graph.

Since we cannot be sure of the graph topology and number of nodes in the graph, the model has to work in a distributed way as the routers and routing algorithms on the internet. A common algorithm has to operate at each of the nodes that just takes into account the neighboring graph topology. It then repeatedly operates on the neighboring nodes and explores into graph to converge to a final state for each node. Each node based on this state and the initial input labels to nodes uses another of commonly used machine learning algorithms to predict an output at each node of the graph.

In other words, a graph neural network model uses some sort of transition function at each node to compute its state based on the neighboring nodes and edges (directed or undirected). It uses this function repeatedly until a converging state is achieved at each node. For this to occur, the choice of the transition function must be such that the Global Transition function (i.e. the stacked version of the local transition function is a Contraction map. Read Banach's fixed point theorem [1]). After achieving this convergence on a state at each node, we can then use a output function to predict the output on each node. This output is a function of the input labels on the node and the state of the node which in some way also encodes the graph structure/data.

2.2 How it fits the problem at hand?

Problems in molecular physics provide an excellent opportunity to use the graph neural network model. First, we see how we represent the molecular data as a graph network and then look the transition and the output functions for the problem in the later sections. We can represent a molecule with 'N' atoms to be a N-clique graph (since bonds are just forces of attraction which exist between all pairs) with edges representing distance between the atoms or some other similar metric like the force (inverse of distance). This ensures that we loose minimum information as for the graph topology. Each node can be represented by a set of characteristics of atom corresponding to that node in the graph. These can be atomic charge, atomic mass and many other characteristic property of the atom.

In our case, we use the atomic number to represent the node label at each atom. The edge label is the euclidean distance between the two atoms. State at each atom is a vector of 10 numbers. The output is a target partial charge that the atom posses in the molecule. Since each molecule we consider in our data set is electrically neutral, we would expect the target charges predicted by the GNN to sum up to zero. But we cannot assure the machine learning model to be 100% accurate and so it does not so occur that the sum of all charges is zero. Hence this is enforced by the following simple mechanism. Compute the sum of charge on all nodes (say 'Qnet') and then divide this Qnet by total number of nodes and subtract the final quantity from each node. This ensures that the final charges predicted by our model sum up to zero and hence predicting correctly that the molecule is electrically neutral.

3 Dataset

DataSet included about 6000 molecules of which only 300 were used for training the matlab toolkit (+ another 300 for testing). The reasons for this are discussed in the section that describes the Matlab toolkit implementation and results of GNN. Most molecules in the dataset were medium sized molecules consisting of around 20 atoms. Unlike other standard implementations of GNN that come along with the toolkit, the data set in this case could not be generated on the fly to use. Training data was preloaded into memory so as to carry out learning and then similar for the testing. The CPP implementation was also not fast enough to include all molecules from data set in training. Not more than 1200 molecules could be used for training the CPP implementation with the expensive GA. (There definitely is a lot of scope to optimize the current CPP version though)

4 Matlab implementation

4.1 Intro

The matlab implementation of the GNN used a toolkit, which provided options on using different GNN models as to linear vs non-linear versions of non-positional graph neural networks. We decided to stick to a neural (non-linear) implementation of the non-positional GNN, since the linear clearly was computationally costlier than the neural and would have turned out even slower on Matlab. The neural model used two feed forward neural networks.

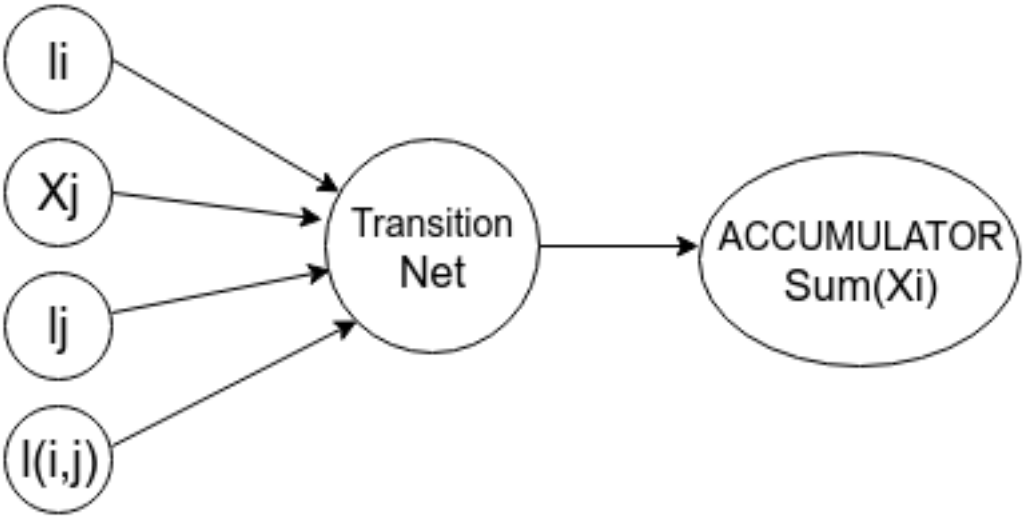
- the transition network
- the output network

These are described below.

4.2 Transition Net

Each node and each edge in the GNN model is assigned a label. We denote the labels of nodes as l_u where $u \in N$ and labels of edges as $l_{(u,n)}$ where both $u, n \in N$ and $(u, n) \in E$. Also each node is assigned a state X_u which eventually converges to a final state using a distributed graph algorithm. Based on its neighbors each node calculates the next state, until it reaches a converging state. This converging state is guaranteed to be

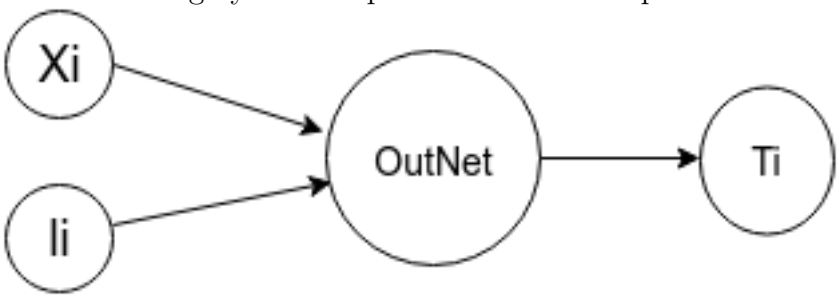
reached by out choice of the transition function (here the transition net).



The figure shown is a local transition net which computes state of node X_i. It does this for all j and accumulates the result which is then regarded as the next state of the node in the GNN. Once all nodes in the graph have reached a state of convergence, we then proceed to output (target computation step).

4.3 Outnet

Once we have computed the converging state for each node, we can now proceed to predict target based on the node label and the state. The computation of output at each node is straight forward. We input the state X_i and label li of each node in a different feed forward neural network we call the outnet and regard its output as the predicted value. The figure of the outnet shows roughly the computation that takes place at each node.



4.4 How to run the current version?

toolkit/dataset/data_new/final.DB contains about 9000 molecules which includes some repetitions. randomParse.py in the same folder parses the binary

database into text which can then be read by the matlab program. Follow the instructions below to get program running.

- cd toolkit/datasets/data_new directory and run randomParse.py to randomly distribute molecules into trainSet, validationSet and testSet files.
- Go to the main directory of the toolkit in Matlab
- addpath(genpath(pwd)) - to add the current directory and all other directory inside the current directory recursively to the Matlab search path.
- makeMoleculeNew - to load all data into the dataSet structure.
- global dataSet dyanmicSystem learning testing - to keep a watch on the learning environment
- Configure('mol.config') - to configure the GNN with different parameters such as maxForwardIter, typeOfFeedForwardNetwork - Linear vs tanh etc
- learn - to start learning on the loaded data
- plotTrainingResults - plots the various learning curves
- test - tests on the test data and pretty prints the results.

5 Results

The results obtained using the matlab implementation, though not very accurate, were encouraging to proceed with the graph neural network approach to tackle the problem at hand. Following is the learning curve (with mean squared error/2 on y-axis vs number of iterations on the x-axis) which forms a elbow at around 0.0315 value of mse/2 in about less than 200 iterations (where mse - mean squared error). After the completion of training the average error was about 0.159 per atom on train set which is not quite satisfactory if we compare it to the actual charge but is reasonable learning from the highly incorrect initial values. This encouraged me to try out larger values for state dimension in the node states which because of the limitations mentioned below could not be carried out on the Matlab implementation and that is why I switched to C/CPP.

Summary:

TrainSet error:

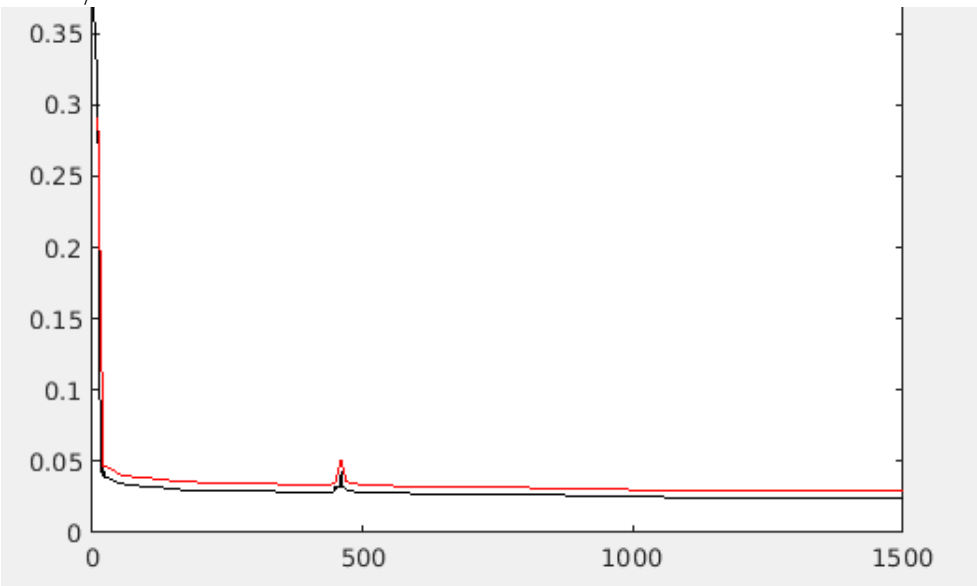
- (mse/2) = 0.03
- average error = 0.159

TestSet error:

- (mse/2) = 0.04
- average error = 0.231

Here the number of molecules considered was only around 300 for training + 300 for testing. (300 molecules consisted of about 5000 atoms). mse - mean squared error.

The figure shows the learning curve which is a plot between the training error mse/2 and number of iterations.



5.1 Limitations

Larger number of molecules could not be used because

- the Matlab toolkit implementation loaded all molecules into a single graph (of course with number of disconnected components equal to the number of molecules loaded). This created larger adjacency matrix which quickly exhausted memory and displayed results such as out of memory.

- the software in itself is a little slow though it is really handy and easy to use.

5.2 Data representation

Data was represented as a Matlab variable and divided into trainSet, validationSet and testSet. Each set possessed some number of molecule represented as completely connected graph and dis-connected from each other. Each structure possessed a vector for Atomic Numbers, positions and the target charges (for supervised learning). The edges were represented as a sparse adjacency matrix and the disconnected nodes were represented by "-1".

6 C/CPP Implementation

The Matlab version showed scope for improvement in the results obtained. So we switched to CPP. Since there wasn't enough time to come up with the full functional implementation of the graph neural network with the back propagation algorithm for learning, it was decided that Genetic Algorithm Library implemented in C by Dr. Filippo be used.

This improved matlab version of code in terms of speed and differed in the implementation to offer scalability. It avoided silly things like loading all molecules into one huge graph.

The initial CPP implementation is functional but is rather slow (due to extensive fragmented memory and vector re-allocations) which combined with the lot more expensive GA trains in about 17 hours (for 150 generations, population size of 1024 and dataSet of 1000 molecules). Also on some test trials, we found that similar atomic number atoms gave out close target results even when they were at much different distances. The extent of affect that edges had on the state was quite low. This was then increased by feeding into the transition net a larger state vector of other atom and removing the state vector of the current atom. This indeed turned out to be successful in differentiating similar atoms at different distances in similar molecules.

This implementation still lacks speed. Tweaking and trying out the large parameter set in a greedy way with such a slow training wasn't really an option. So we had to optimize it. We switched to C style code and some optimization added were (1) stacking all 2 or more dimensional arrays into a single dimensional contiguous chunk of memory. (2) Reducing unnecessary computation where ever possible. (3) extensively work with pointers instead of copying arrays or vectors each time. This new optimized implementation

is not fully functional since it still suffers from some memory error which is being looking into at the moment. We hope to soon resolve it.

6.1 How to Run?

This implementation has been provided with makefile. getting this to run is easy.

- Compile the GA library using make in the main ga folder
- cd ga/gnnTest
- use make to compile
- run using ./static.exe (can be run on multiple cores using mpirun/mpiexec)

6.2 Partial Results obtained on CPP implementation

For a transition Net and out net with 1 hidden layer and 5 hidden neurons (and stateDim -4), the best fitness is about -454.95 , for a dataSet of 1000 molecules, 200 generation and population size of 1024 genes. Here the fitness is negative of sum of squared errors. The problem of same molecules giving out similar outputs seems resolved to quite some extent.

7 Conclusions

In this report we have discussed out implementation of the graph neural network and its application in solving a problem in molecular physics. The Matlab and the C++ implementations can be found along with the report. Brief summary is given on how to get them up and running. Results have been discussed.

References

[1] Banach Fixed Point Theorem,
`https://en.wikipedia.org/wiki/Banach_fixed-point_theorem`

[2] The Graph Neural Network Model,
`https://repository.hkbu.edu.hk/cgi/viewcontent.cgi?article=1000&context=vprd_ja`