Report

Annabelle Cloutier

February 27, 2023

1 Paper

1.1 Network Structure

All of the below is based on the work in [1] The network structure is incredibly generic most of the way through, but notes are written here to avoid having to parse through the equations that describe the network structure in the paper. Note that at any step where learned weights are used, the ReLU function is used on the resulting vector. The only exception is for the last set of learned weights in the readout layer.

1.1.1 Node Embedding

The MPNN (Message Passing Neural Network) structure used converts each vertex into a set of m observations and encodes them into a n-dimensional embedding using learned weights. The paper states that they use 64 dimensional embeddings and they do in code, but this could be any size.

1.1.2 Edge Embedding

The edges for each vertex are also encoded into a separate n-dimensional embedding, same size as for each vertex, once again learned. The input for this step is the set of m observations of the neighboring vertices catenated with the weight on the connecting edge, creating an $m + 1$ dimensional vector for each neighboring vertex. All of these vectors are then summed and passed through a learned layer, creating an $n - 1$ dimensional vector. At this stage, the resulting $n - 1$ dimensional vector is divided by the number of neighbors, catenated with the number of neighbors, and passed through another learned layer, resulting in an n-dimensional embedding representing the edges for a vertex.

The end result of these two steps are an n-dimensional embedding representing a vertex and another representing it's neighbors, all created using the same learned weights for every vertex. Meaning there will be $2|V|$, n-dimensional embeddings.

1.1.3 Message Passing

Here there is a message pass layer and an update layer. The message pass per vertex is summing the products of the connected vertices and the weight, divided by the number of neighbors, catenated with the edge embedding for that vertex and then passed through learned weights, resulting in a n-dimensional vector.

The next is the update layer, which is the embedding for that vertex, catenated with the message and passed through another set of learned weights, into an n-dimensional vector, representing the "new" embedding for that vertex.

The message then update is performed K times. Mentioned in the paper and corroborated in the code, this is done 3 times, but can be done however many times necessary.

1.1.4 Readout

The readout layer goes through each vertex, summing the embeddings for the neighbors of that vertex, dividing the result by the number of vertices in the entire graph and then passing it through learned weights, resulting in an n-dimensional vector.

The embedding for the vertex itself is then catenated to the resulting vector and passed through another set of learned weights (without applying ReLU), resulting in a single output value. This value represents the Q-value for that node, which is used by the algorithm to determine which vertex will be added/removed from the solution set on that step. The node associated to the maximum Q-value is added to the solution set if it doesn't yet belong to it, or removed from the solution set if it belongs to it.

1.2 Generic Sections

Most of this structure is generic for any graph. The entire input structure (node and edge embeddings) are completely generic and can be used to embed graph information irrespective of topology. The graph input is even flexible to changes in problem statement as the number of observations as well as their values can be modified to fit a different problem, or modify the observations of the same problem, which was the Maximum Cut Problem in the paper. Because the entire structure within the neural network doesn't have anything specific to the Maximum Cut Problem and is purpose built to simply pass the information of other vertices from vertex to vertex, none of the inner structure necessarily needs to change to fit a different problem.

To apply this structure to a different problem, one only needs to change the number of observations as needed or change the observations themselves, the rest of the network simply passes this information between graph vertices and produces an output for each vertex, going through trained layers to interpret the information when needed.

1.3 Non-Generic Sections

The main issue comes with the output and it's interpretation. Each vertex is represented by a single value as the output, and that value is interpreted as adding or removing it from the solution set, in the context where the solution is a subset of the vertices in the graph. More specifically for the Max Cut problem, the vertex associated with the maximum Q-value (output value) is taken and then either added or removed from the solution set, depending on whether it already belongs to it or not.

This approach works fine for a problem like Maximum Cut, Minimum Vertex Cover or any other combinatorial optimization problem where the solution is to split the vertices of the graph into two sets. However, this does not work for a problem like the Traveling Salesman Problem where the solution is an ordered list of vertices, or the Minimum K-Cut Problem which can have an arbitrary number partitions of the graph. Modifications to either the output or its interpretation would have to be done in order to use this structure or a similar one for one of these different problems or a similar one. This is especially important if the output is to be as generic as possible.

1.4 Benchmarks

The paper displays the performance of the graph in reference to S2V-DQN, a similar paper where the algorithm does not allow for reversing actions as well as a greedy algorithm. It also compares it's performance against modifications of itself, namely where some observations are restricted, intermediate rewards are not given for reaching locally optimal solutions, as well as keeping it from reversing its actions.

They use GSet graphs G1-10 and G22-32 for calculating the approximation ratio, asl well as the Physics dataset.

1.5 Graph Generation

They train and test on Erdos-Renyi [2] and Barabasi-Albert [3] graphs.

2 Code

The code holds the capability of running all of the above tests, but they're not explicitly written and must be generated via self-written code.

2.1 Running Code

They very generously provide a README file that specifies the exact commands to run in order to train, test and validate networks. However, there is no code for reproducing their specific tests. These all need to be hand-coded. The only results I've reproduced so far are training and testing.

2.2 Graph Generation

Unsure of the exact implementation, it seems they use NetworkX's ability to generate random graphs in order to do this. It's completely unnecessary though seeing as they have many test, validation and benchmark graphs pre-built within their code that can be used. If absolutely necessary, creating random graphs and storing them within a pickle file as they do would likely lower the workload on inputs as the code for parsing through these types of files is already prebuilt.

2.3 Input Conversion

Completely in the dark here. The code for converting graphs into observations seems to exist within src/envs/spinsystem.py but further code inspection is needed to see exactly what's happening. The idea behind it is incredibly straightforwards, it's merely a matter of finding where this is done so modifications can be made, as our observations will have to be different if we're not doing the Maximum Cut.

2.4 Benchmarks

They generously provide testing, validation and benchmark graphs in Pickle files, which is a special type of object file for Python, as well as their solutions, which are also stored in Pickle files, but are simply a boolean (technically floats, just 1.0 and 0.0) list for determining whether a node is in the solution set or not.

References

[1] T. D. Barrett, W. R. Clements, J. N. Foerster, and A. I. Lvovsky, "Exploratory combinatorial optimization with reinforcement learning," in *Proceedings of the Thirty-fourth AAAI conference on Artificial Intelligence*, arXiv, 2019.

[2] P. Erdős, A. Rényi, *et al.*, "On the evolution of random graphs," *Publ. Math. Inst. Hung. Acad. Sci*, vol. 5, no. 1, pp. 17–60, 1960.

[3] R. Albert and A.-L. Barabási, "Statistical mechanics of complex networks," *Reviews of Modern Physics*, vol. 74, pp. 47–97, jan 2002.