Combinatorial Optimization Over Graphs via Neural Networks

Team Funny, It Worked Last Time

Andrew King
Physics
Johns Hopkins University

Jack Armand
Economics
Johns Hopkins University

Vipul Bhat
Computer Science
Johns Hopkins University

Allison Rosen
Biomedical Engineering
Johns Hopkins University

October 7, 2020

Abstract

In this project, we will attempt to improve the generalization of the techniques used in the paper "Learning combinatorial optimization algorithms over graphs" by Hanjun Dai et al [1]. Our strategy was to adopt a supervised learning approach in which a neural network is trained to solve problems on small, pre-solved instances of the traveling salesman problem (TSP) instead of the reinforcement learning paradigm studied by Dai. We used an intuitive labeling strategy in which we labeled the next vertex in the known solution to each TSP as a target vector. After training, our strategy did not produce valuable outputs and we do not recommend this approach for use in actually solving TSPs, at least not without a better representation of the data. All of the code used in our analysis can be found at `https://github.com/agking10/Deep-Learning-Discrete-Optimization`.

1 Introduction

Many combinatorial graph problems, such as the Traveling Salesman Problem, have been popular areas of research in recent years due to their presence in NP-hard problems. An important step in reducing the time necessary to solve these combinatorial problems is developing a good heuristic. While a heuristic does not typically yield the optimal solution, a good heuristic can place a worst-bound on it. This reduces the infinitely many possible solutions to a manageable finite number of them.

With the explosion in popularity of neural networks, and their effectiveness in solving complex problems, researchers have begun to implement them in cases such as the Traveling Salesman Problem. One such instance of this is in the paper "Learning combinatorial optimization algorithms over graphs" by Hanjun Dai et al [1]. In this paper, Dai argues

that implementation of neural networks to solve graph optimization problems has failed to take into account the combinatorial structure of these problems. To address the unique challenges of developing effective heuristics for graph problems, Dai implements a neural network using a combination of reinforcement learning and graph embedding. He uses a graph embedding network to represent the graph data before running it through a greedy algorithm to select the optimal path. To create the greedy algorithm, he uses fitted Q-learning to optimize algorithm parameters.

What we look to improve upon is his implementation of this process using a reinforcement learning technique. Reinforcement learning enables an algorithm to learn from experience. Rather than training the model using the correct answer, the user will either reward or punish the model's output and the model will work to maximize its reward. In Dai's application of such methods, his algorithm looks at the 25 nearest neighbors to a single node, determines the reward for moving to each of these nodes, and moves to the node with the greatest reward. An alternative to this approach would involve feeding the algorithm the correct answer, and working to minimize the error rather than maximize the reward. In fact, Dai proposes such an alternative in his paper. While citing his reason for using this reinforcement learning approach as a lack of training labels, he states that access to such data would allow utilization of a supervised learning approach involving a softmax-layer and cross-entropy loss minimization. This is a technique from his paper "Discriminative embeddings of latent variable models for structured data" [2].

Neural networks have enabled huge advancements in complex problem solving. A well trained network can efficiently and accurately perform tasks that other forms of machine learning will do slower and less effectively. In Dai's work, the reason to use reinforcement learning over supervised learning is due to the lack of training examples for the TSP. However, software such as Gurobi or Concord can solve small instances of the TSP very quickly, so it may be possible to use these computed solutions as training data with the hope that a supervised approach will lead to a more accurate algorithm that can generalize to larger examples more easily. The more catered the design of a network is to a given problem, the better its results. If we are attempting to find the optimal TSP path, then we believe implementation of Dai's network using a supervised approach, where it is training on correct answers and not just assigned reward values, will result in an improved outcome.

2 Our Process

The architecture of our network is the same as the one created by Dai et al. including the embedding process `structure2vec` To recap, consider a graph G = (V,E). We try to represent each node $v \in$V by a p-dimensional embedding $\mu_v \in \mathbb{R}^p$. The embedding is an iterative process, so we can add a superscript so μ_v^t represents the t^{th} iteration of the process. To begin the embedding process, initialize $\mu_i^0 = 0$ for all $v \in$V. For any iteration after this, we have

$$\mu_v^{(t+1)} \leftarrow relu\left(\theta_1 x_v + \theta_2 \sum_{u \in N(v)} \mu_u^{(t)} + \theta_3 \sum_{u \in N(v)} relu(\theta_4 w(v,u))\right) \tag{1}$$

where $w(u,v)$ is the weight of the edge (u,v), $N(v)$ is the set of p nodes adjacent to v,

and x_v is a label on v. For our case, x_v is 1 if the node has been included in our partial solution and 0 if it has not. The parameters $\sigma_1, \sigma_2, \sigma_3, \sigma_4$ are linear layers in our neural network. We se by the sums that in each iteration, the embedding is pooling information from its 25 nearest neighbors. This means that on the first iteration the embedding contains information about its nearest neighbors, on the second iteration the embedding contains information about the nearest neighbors and their nearest neighbors, and so on. This means that after t iterations, the embedding of a node μ_v contains information about all the nodes that can be reached in t steps from v. We can see that if we repeat this process only a few times, we can gather a lot of information about G. In our algorithm we iterate the embedding 4 times, as was recommended in Dai's paper.

In order to turn this embedding into an algorithm, we need a way to actually select a node from the embedding. In order to do this, we can input the data from a specific node into the formula

$$\hat{Q}(h(S), v; \Theta) = \theta_5^\top relu([\theta_6 \sum_{u \in V} \mu_u^{(T)}, \theta_7 \mu_v^{(T)}]) \tag{2}$$

where $\theta_5, \theta_6, \theta_7$ are more linear feed-forward layers. This gives us our greedy algorithm policy. We can input the appropriate μ_v for a particular node, and get a scalar as an output. We can compare the output of a node to the output from all the other nodes to select the next node in our tour.

The benefit of running on only one given node at a time is that our network can be generalized to TSPs of any size. In terms of selection criteria for what is the optimal node to move to, this is established during training. Dai's model assigned reward values to each potential node and optimized hyper parameters to maximize reward. Our model works to minimize error between what the model predicts to be optimal and what the actual solved TSP output is. An explanation of our variables, along with pseudocode for our algorithm, is as follows:

For a given graph, our nodes are labeled $x_o, ..., x_{n-1}$. If we use a p-dimensional embedding then we have a $p \times n$ matrix where column i records the p nearest neighbors to node i. Call this matrix $\mathbf{N}$. We also have a $p \times n$ matrix that records the cost associated to visiting a corresponding neighbor in $\mathbf{N}$. As an example, if the entry in $\mathbf{N}$(1,1) is 5 then the weight matrix entry at (1,1) will be the weight of graph edge (5,1). Call the weight matrix $\mathbf{W}$. Finally, we have a vector $l \in \mathbb{R}^n$ of labels that will correspond to the partial solution in our algorithm. $\hat{Q}$ is a function of all these variables. If we n nodes to select from for a given step, then we have $\hat{Q} = q_1, ..., q_n$. A softmax function normalizes this vector, generating probability values for each q_i. If the optimal move were to select node 2, then we would expect an output of $\hat{Q} = 0, 1, 0, ..., 0$. The loss is computed by looking at the difference between this expected and actual output, which the network then uses to back propagate and further optimize its parameters.

The backpropagation of q occurs to optimize 7 parameters in $\hat{Q}$, $\Theta = (\theta_i)_{i=1}^7$. In these equations, v represents the node the network is looking at while u represents all of its nearest neighbors that are being considered for selection.

The key difference between our process and the one used by Dai et al. is that their algorithm optimizes their model by assigning it a reward function and compares the output of the model with an estimate of the possible reward. Our algorithm, on the other hand,

Algorithm 1

Initialize l=0,
for i=0,...,n-1:
    sum=0, count=0
    tensor q$\in \mathbb{R}^{n-i}$=0
    tensor p$\in \mathbb{R}^{n-i}$=0
    for $j \ni l_j = 0$:
        q(count)=$\hat{Q}$(W,N,l,j)
        sum+=q(count)
        if $j = x_{next} : p(count) = 1$
            count++
        end if
    end for
    q/= sum
    p(argmax(q)) = 1
    compute KL(q,p)
    backprop(q) over parameters in $\hat{Q}$
end for

optimizes our model at each iteration of the outer **for** loop in algorithm 1. We have the model estimate which node should be added to the tour next and we compare this to the node following the current node in the actual solution.

After training the network, we were left with optimized parameters and weights which can be applied to efficiently and accurately solve new instances of the traveling salesman problem. In the next section we discuss the results of our network.

3 Results and Discussion

Overall, our network performed extremely poorly. The results are shown in table 1. The first row of the table lists the opt-ratio of the raw output from our network. The ratio outputted by our network performed about as well as selecting a random permutation of the nodes as a tour. We were curious to see how our results would change if we applied a simple tour augmentation algorithm to our output. We chose to apply the 2-opt algorithm to our network output, which is the result listed in the second row. This appears to be a promising result, but when compared to other simple routing heuristics, we see that the results are about the same. The final row in our results is a control. We selected a tour entirely at random and augmented it using 2-opt, which resulted in an opt-ratio that is extremely similar to the others in the table.

On the surface, it may seem like this experiment was a failure. However, we still obtained a highly unexpected result. We used a model architecture that achieved good performance (or at least performance comparable to heuristics used in practice) in a reinforced learning paradigm. We applied a simple, but intuitive labeling to our data and attempted to train the model using supervised learning, and achieved extremely poor results. This raises an interesting question. Is it possible that the preferred way to train a graph neural network

Table 1: Opt Ratio for Different Heuristics Using Randomly Generated TSPs

Heuristic	Opt Ratio (mean)	std
Regular Output	6.70	0.44
Output w/2-opt	1.032	0.19
Farthest Insertion	1.081	0.21
Farthest w/2-opt	1.056	0.21
Random Insertion	1.076	0.25
Random w/2-opt	1.056	0.019
Random Tour w/2-opt	1.033	0.015

is using unsupervised learning?

Now, it is possible we could have chosen more effective labels for our data. We could have attempted to assign a target weight to multiple vertices instead of only the next vertex in the known solution, however the amount of time and thought this would take might eliminate the benefits of using a supervised learning approach at all.

One of the most important concepts in deep learning is that how you represent a dataset to a network can have an extraordinary effect on its output. Is it too much of an engineering hurdle to select an appropriate representation for a complex problem such as the TSP without an exceptional amount of insight? Maybe the most effective way to proceed with deep learning research in discrete optimization is using reinforcement learning trained end-to-end. In a recent paper, a team at Google Brain produced results better than Dai using an attention-based model, again with reinforcement[3]. Maybe an intuition based approach to supervised learning needs to be considered very carefully if it is to be used at all.

4 Future Work

There is always room for improvement when it comes to the performance of neural networks. Given the rapid advancements in techniques, and the seemingly infinite potential framework and hyper parameter combinations, there are countless different strategies that could be tested. More data, more computing power, and more time could get us closer to the optimal neural network for solving TSPs. Things to explore in an effort to try getting our supervised learning approach to work, just to name a few, include adding or removing various types of layers, and implementing alternative loss or activation functions.

Furthermore, we had hoped to find a way to effectively encode TSPs into a fixed length vector as a way to represent our data. This is a necessary step if you want to be able to use the same network for TSPs of varying size while still being able relay every bit of graph information to the network. As we had insufficient time to dive into this problem while concurrently tackling the creation of our neural network, we opted to use Dai's strategy to encode the 25 nearest neighbors to a given node. While this enabled us to still work with and solve TSPs of varying size, Dai cited that this was not as effective as a farthest insertion heuristic. We proposed a few different ideas to explore before choosing not to allocate our limited time resources to this problem. Could our network learn a hybrid insertion if we

encoded the nearest and furthest neighbors to any node? Were there other iterative iterative encoding procedures we could utilize that retain more relevant TSP information? Could we pre-train an auto encoder for the number of nodes in a TSP problem and feed the result into our network? We hope others will take on the task of exploring these and other potential improvements to the graph representation problem, and apply them to our network.

References

[1] "Learning combinatorial optimization algorithms over graphs", H. Dai, E. B. Khalil, Y. Zhang, B. Dilkina, L. Song. 2017. Appeared in NIPS 2017.

[2] "Discriminative embeddings of latent variable models for structured data", Dai, Hanjun, Dai, Bo, and Song, Le. 2016. Appeared in ICML 2016.

[3] "Attention Is All You Need", Ashish Vaswani and Noam Shazeer and Niki Parmar and Jakob Uszkoreit and Llion Jones and Aidan N. Gomez and Lukasz Kaiser and Illia Polosukhin. 2017.