CO 759: Deep Learning in Computational Discrete Optimization-Course Project

Sushant Agarwal Zhi-Yuan Wang

November 6, 2019

Contents

1 Introduction

1.1 Overview

Previously in Dai et al [4] a framework that combined reinforcement learning and graph embedding was used to learn heuristic algorithms for combinatorial optimization problems on graphs. What was particularly attractive about their framework was that it can be applied to a diverse range of optimization problems over graphs. They demonstrated that it learns effective algorithms for the Minimum Vertex Cover, Maximum Cut and Traveling Salesman problems.

Local search heuristics can often be a practical way of obtaining relatively good solutions to NP-hard combinatorial optimization problems in a reasonable time frame. Local search works by starting from a candidate solution and repeatedly improving the solution by a set of allowed operations. This will usually result in reaching a solution(local optimum) which cannot be further improved by those operations.

In this study we look at the 2-opt method, a local search heuristic for the Traveling Salesman Problem. We use a similar framework as [4] that combines reinforcement learning and graph embedding to learn good moves that aim to reach a local optimum in the shortest number of 2-opt swaps.

We also look at the network simplex method, a standard algorithm for solving the minimum cost flow problem. A basic feasible solution is represented by a spanning tree and the pivot operation involves adding an edge to the tree then deleting one of the edges in the unique cycle formed as a result. Choosing a good edge to add in the pivot step of the simplex method is a challenge. We use a similar framework as the one for 2-opt to learn good pivot moves that aim to reach the optimum flow in the shortest number of steps.

2 Proposed method for TSP

Let $G(V, E, w)$ denote a weighted graph, where V is the set of nodes, E the set of edges and $w : E \to \mathbb{R}^+$ the edge weight function. The cost of a tour S is denoted by $c(S)$.

We start from an initial random tour S. We then choose a swap which aims to minimize the evaluation function, $Q(S, (e_1, e_2))$. We add the pair of edges (e_1, e_2) and remove the corresponding pair of edges $(e_1{}', e_2{}')$. This step is repeated until the local optimum is reached. Intuitively, we want the Q function to tell us the minimum number of swaps away the current tour is from a local optimum. Q will be learned using a collection of problem instances.

2.1 Representation: Graph Embedding

We need to make some modifications to Dai's [4] framework in order to adapt to the problem we are considering. We want the graph embedding to capture the current tour S. We use a vector of binary decision variables x, with each dimension x_e corresponding to a edge $e \in E$, $x_e = 1$ if $e \in S$ and 0 otherwise. We will use a deep learning architecture over graphs, `structure2vec` of [4], to parameterize $\widehat{Q}(S, (e_1, e_2); \Theta)$.

2.1.1 Structure2Vec

`Structure2vec` will compute a p-dimensional feature embedding μ_e for each edge $e \in E$, given the current tour S.

We initialize the embedding $\mu_e^{(0)}$ at each edge as 0, and for all $e \in E$ update the embeddings synchronously at each iteration as

$$\mu_e^{(t+1)} \leftarrow F\big(x_e, \{\mu_u^{(t)}\}_{u \in N(e)}, \big\{w(u)\big\}_{u \in N(e)}; \Theta\big), \tag{1}$$

where $N(e) \subseteq E$ is the set of neighbors of edge e, and F is a generic nonlinear mapping.

2.1.2 Parameterizing $\widehat{Q}(S, (e_1, e_2); \Theta)$

F updates p-dimensional embedding μ_e as:

$$\mu_e^{(t+1)} \leftarrow \mathrm{relu}\big(\theta_1 x_e + \theta_2 \sum\nolimits_{u \in N(e)} \mu_u^{(t)} + \theta_3 \sum\nolimits_{u \in N(e)} \mathrm{relu}(\theta_4\, w(u))\big), \tag{2}$$

where $\theta_1 \in R^p$, $\theta_2, \theta_3 \in R^{p \times p}$ and $\theta_4 \in R^p$ are the model parameters, and relu is the rectified linear unit ($\mathrm{relu}(z) = \max(0, z)$) applied elementwise to its input.

Once the embedding for each edge is computed after T iterations, we will use these embeddings to define the $\widehat{Q}(h(S), (e_1, e_2); \Theta)$ function. We use the embedding $(\mu_{e1}^{(T)} + \mu_{e2}^{(T)})$ and $\sum_{e \in S} \mu_e^{(T)}$, as the surrogates for (e_1, e_2) and S, respectively, and:

$$\widehat{Q}(S, (e_1, e_2); \Theta)) = \theta_5^\top \mathrm{relu}([\theta_6 \sum_{e \in S} \mu_e^{(T)}, \theta_7 \, (\mu_{e_1}^{(T)} + \mu_{e_2}^{(T)})]) \tag{3}$$

where $\theta_5 \in R^{2p}$, $\theta_6, \theta_7 \in R^{p \times p}$ and $[\cdot, \cdot]$ is the concatenation operator.

2.2 Training with Reinforcement Learning

We define the states, actions and rewards in the reinforcement learning framework as follows:

1. *States*: a state S is a sequence of edges on a graph G represented as a vector in p-dimensional space, $\sum_{e \in S} \mu_e$.
2. *Actions*: an action v is adding the two edges e_1 and e_2 into S and removing the corresponding swap edges $e_1{}'$ and $e_2{}'$.
3. *Transition*: transition corresponds to tagging the two edges e_1 and e_2 that were selected as the last action with feature $x_{e_1} = 1$ and $x_{e_2} = 1$. We also set the corresponding removed edges $x_{e_1{}'}$ and $x_{e_2{}'}$ to 0.
4. *Rewards*: the reward function $r(S, (e_1, e_2)) = 1$ for every pair S and (e_1, e_2) The *cumulative reward* R of a terminal tour $\widehat{S}$ coincides exactly with the number of swaps required to reach it from the initial tour;
5. *Policy*: based on $\widehat{Q}$, a deterministic greedy policy

$$\pi((e_1, e_2)|S) := \underset{(e_1{}', e_2{}') \in S}{\mathrm{argmin}} \, \widehat{Q}(S, (e_1, e_2))$$

will be used.

2.2.1 Learning Algorithm

We let Q^* denote the *optimal* Q-function. $\widehat{Q}(S, (e_1, e_2); \Theta)$ will then be an approximation for it. We would like to learn $\widehat{Q}$ across a set of m graphs with potentially different sizes. We perform end-to-end learning of the parameters in $\widehat{Q}(S, (e_1, e_2); \Theta)$.

We use the term *episode* to refer to a complete sequence of edge swaps, starting from the intital tour upto termination(local optimum); a *step* within an episode is a single action (edge swap).

Standard (1-step) Q-learning updates the function approximator's parameters *at each step* of an episode by performing a gradient step to minimize the squared loss:

$$(y - \widehat{Q}(S_t, (e_{1_t}, e_{2_t}); \Theta))^2, \tag{4}$$

where $y = \gamma \min_{(e_{1_{t+1}}{}', e_{2_{t+1}}{}') \in S} \widehat{Q}(S_{t+1}, (e_{1_{t+1}}, e_{2_{t+1}})) + r(S_t, (e_{1_t}, e_{2_t}))$ for a non-terminal state S_t.

Say a particular episode takes k steps to reach local optimum from a state s. We wait until the end of the episode before updating the approximator's parameters. Formally, the update is over the same squared loss, but with a different target:

$$y = \sum_{i=0}^{k-1} r(S_i, (e_{1_i}, e_{2_i})) = k \tag{5}$$

Instead of updating the Q-function sample-by-sample, we update it with a batch of samples from a dataset F. The dataset F is populated during previous episodes. At the end of an episode of length k, the tuples $(S_i, (e_1, e_2)_i, R_{i,k}, S_k)$ are added to F for $i \in \{0, ..., k-1\}$, with $R_{t,t+n} = \sum_{i=0}^{n-1} r(S_{t+i}, (e_{1_{t+i}}, e_{2_{t+i}}))$. Stochastic gradient descent updates are performed on a random sample of tuples drawn from F.

2.2.2 Pseudo-Code

1. Initialize experience replay memory M to capacity N
2. **for** episode $e = 1$ **to** m **do**
3. Draw graph G from distribution D
4. Initialize the state to a tour $S_0, t = 0$
5. **while** there exists a benefitial 2-opt move **do**
6. Let $E_S \subseteq E \times E$ be the set of possible 2-opt moves
7. $(e_1, e_2)_t = \begin{cases} \text{random pair } (e_1, e_2) \in E_S, \text{ w.p. } \epsilon \\ \text{argmin}_{(e_1,e_2) \in E_S} \hat{Q}(S_t, (e_1, e_2); \Theta) \end{cases}$
8. Create new tour S_{t+1}: Add (e_{1_t}, e_{2_t}), remove $(e_{1_t}{}', e_{2_t}{}')$
9. $t = t + 1$
10. Add the tuples $(S_{t-i}, (e_1, e_2)_{t-i}, R_{t-i,t}, S_t)$ to M for $i \in \{t, ..., 1\}$
11. Sample random batch from $B \sim M$.
12. Update Θ by SGD by B

3 Experiments for TSP

3.1 Data generation

Our experiment for 2-opt is on 2D euclidean TSP instances. We uniformly sample n points in the square $(0, 1) \times (0, 1)$ and this specifies our TSP instance. We create a random tour by a random permutation of the vertices as initialization for each of the graphs.

For the experiment we set $n = 20$. We generate 1000 graphs for training, 100 for validation and 100 for testing.

3.2 Training procedure

We use a minibatch size of 64 and the Adam optimizer [6] with learning rate 0.0001. We decay the learning rate by multiplying it by 0.95 every 50 iterations.

The loop (as in the pseudocode) is run for 10000 iterations or until it seems to have converged. The iteration we report for the performance on the test set is the one with the best performance on the validation set (we use the validation set to select the "best" version of the model to evaluate for the test set performance).

For the computation graph of the neural network we use the current tour and the edges in the swap to be evaluated. For node features we have the coordinates of the nodes and for edge features we have the distances between the two nodes. We use the original node embedding model of Dai et al [4] due to lack of time and deep learning framework expertise to create the edge embedding model originally proposed. We use an embedding dimension of 64 and 4 node embedding updates for the graph neural network.

3.3 Results

The average number of swaps the neural network takes on the test set is 34.53 and the average for the random swap selection is 35.82. Due to the small difference with random selection, this performance may be within error margins. A more rigorous statistical analysis may be required to tell if this result is meaningful.

4 Proposed method for Network Simplex

Let $G(V, E, c, d, w)$ denote a directed graph, where V is the set of nodes, E the set of edges, $w : E \to \mathbb{R}^+$ the edge capacity function, $c : E \to \mathbb{R}^+$ the edge cost function and $d : V \to \mathbb{R}$ the node demand function. The cost of flow in current network S is denoted by $c(S)$.

We start from an initial spanning tree S. The algorithm selects an edge e to add as the pivot operation which aims to minimize the evaluation function, $Q(S, e)$. A cycle is now formed and a unique edge $e^{'}$ is removed that restores the subgraph to a tree. This pivot step is repeated until the optimum flow is reached. Intuitively, we want the Q function to tell us the minimum number of pivots away the current tree is from the optimum flow. Q will be learned using a collection of problem instances.

4.1 Representation: Graph Embedding

We want the graph embedding to capture the current tree S. We use a vector of binary decision variables x, with each dimension x_e corresponding to a edge $e \in E$, $x_e = 1$ if $e \in S$ and 0 otherwise. We will use `structure2vec` to parameterize $\widehat{Q}(S, e; \Theta)$.

4.1.1 Structure2Vec

This graph embedding network will compute a p-dimensional feature embedding μ_e for each edge $e \in E$, given the current tree S. We initialize the embedding $\mu_e^{(0)}$ at each edge as 0, and for all $e \in E$ update the embeddings synchronously at each iteration as

$$\mu_e^{(t+1)} \leftarrow F\big(x_e, \{\mu_u^{(t)}\}_{u \in N(e)}, \big\{w(u)\big\}_{u \in N(e)}, \big\{c(u)\big\}_{u \in N(e)}, \big\{d(u)\big\}_{u \in N_v(e)}; \Theta\big), \tag{6}$$

where $N(e) \subseteq E$ is the set of neighbors of edge e in graph G, $N_v(e) \subseteq V$ is the set of node neighbors of edge e in graph G and F is a generic nonlinear mapping.

4.1.2 Parameterizing $\widehat{Q}(S, e; \Theta)$

F updates the p-dimensional embedding μ_e as:

$\mu_e^{(t+1)} \leftarrow \text{relu}\big(\theta_1 x_e + \theta_2 \sum_{u \in N(e)} \mu_u^{(t)} + \theta_3 \sum_{u \in N(e)} \text{relu}(\theta_4\, w(u)) +$

$\theta_5 \sum_{u \in N(e)} \text{relu}(\theta_6\, c(u)) + \theta_7 \sum_{u \in N_v(e)} \text{relu}(\theta_8\, d(u))\big)$

where $\theta_1, \theta_4, \theta_6, \theta_8 \in R^p$ and $\theta_2, \theta_3, \theta_5, \theta_7 \in R^{p \times p}$.

Once the embedding for each edge is computed after T iterations, we will use these embeddings to define the $\widehat{Q}(S, e; \Theta)$ function. We will use $\sum_{e \in S} \mu_e^{(T)}$, as the surrogate for S, i.e.

$$\widehat{Q}(S, e; \Theta)) = \theta_9^\top \text{ relu}([\theta_{10} \sum_{e \in S} \mu_e^{(T)}, \theta_{11}\, \mu_e^{(T)}]) \tag{7}$$

where $\theta_9 \in R^{2p}$, $\theta_{10}, \theta_{11} \in R^{p \times p}$ and $[\cdot, \cdot]$ is the concatenation operator.

4.2 Training with Reinforcement Learning

We define the states, actions and rewards in the reinforcement learning framework as follows:

1. *States*: a state S is a sequence of edges on a graph G represented as a vector in p-dimensional space, $\sum_{e \in S} \mu_e$.
2. *Actions*: an action v is adding edge e into S and removing the corresponding edge $e^{'}$.
3. *Transition*: transition corresponds to tagging the edge e that was selected as the last action with feature $x_e = 1$. We also set $x_{e'} = 0$ the corresponding removed edge $e^{'}$.
4. *Rewards*: the reward function $r(S, e) = 1$ for every pair S and e. The *cumulative reward* R of a terminal tree $\widehat{S}$ coincides exactly with the number of pivots required to reach it from the initial tree;
5. *Policy*: based on $\widehat{Q}$, a deterministic greedy policy

$$\pi(e|S) := \underset{e \in \overline{S}}{\text{argmin}}\, \widehat{Q}(S, e)$$

will be used.

4.2.1 Learning algorithm

Let Q^* denote the *optimal* Q-function. $\widehat{Q}(S, e; \Theta)$ will then be an approximation for it. We would like to learn $\widehat{Q}$ across a set of m graphs with potentially different sizes. We perform end-to-end learning of the parameters in $\widehat{Q}(S, e; \Theta)$. We use the term *episode* to refer to a complete sequence of pivots, starting from the intital tree upto termination(optimum); a *step* within an episode is a single pivot.

Standard (1-step) Q-learning updates the function approximator's parameters *at each step* of an episode by performing a gradient step to minimize the squared loss:

$$(y - \widehat{Q}(S_t, e_t; \Theta))^2, \tag{8}$$

where $y = \gamma \min_{e' \in S} \widehat{Q}(S_{t+1}, e) + r(S_t, e_t)$ for a non-terminal state S_t.

Say a particular episode takes k steps to reach the optimum. We wait for k steps before updating the approximator's parameters. The update is over the same squared loss, but with a different target:

$$y = \sum_{i=0}^{k-1} r(S_i, e_i) = k \tag{9}$$

Instead of updating the Q-function sample-by-sample, we update it with a batch of samples from dataset F. The dataset F is populated during previous episodes. At the end of an episode of length k, the tuples $(S_i, e_i, R_{i,k}, S_k)$ are added to F for $i \in \{0, ..., k-1\}$ with $R_{t,t+n} = \sum_{i=0}^{n-1} r(S_{t+i}, e_{t+i})$. Stochastic gradient descent updates are performed on a random sample of tuples drawn from F.

4.2.2 Pseudo-Code

1. Initialize experience replay memory M to capacity N
2. **for** episode $e = 1$ **to** m **do**
3. Draw graph G from distribution D
4. Initialize the state to a tree $S_0, t = 0$
5. **while** there exists a benefitial pivot move **do**
6. Let $E_S \subseteq E$ be the set of possible edges to add
7. $e_t = \begin{cases} \text{random edge } e \in E_S, \text{ w.p. } \epsilon \\ \text{argmin}_{e \in E_S} \hat{Q}(S_t, e; \Theta) \end{cases}$
8. Create new tree S_{t+1}: add e_t, remove $e_t{}'$
9. $t = t + 1$
10. Add the tuples $(S_{t-i}, e_{t-i}, R_{t-i,t}, S_t)$ to M for $i \in \{t, ..., 1\}$
11. Sample random batch from $B \sim M$.
12. Update Θ by SGD by B

5 Experiments for Network Simplex

5.1 Data generation

The network simplex instances were generated by the following procedure. First a graph with N nodes is generated by the Erdos-Renyi model where each edge between any two nodes appears with independent probability p. Of this one possible edge between any two nodes its orientation is decided by coin toss. Then for each edge its weight is sampled uniformly between the integers 1 and W, its capacity between the integers 1 and C. An array A of $N/2$ numbers between 1 and D are generated. To guarantee supplies equal demands in the instance $N/2$ nodes are assigned demands according to A while the remaining $N/2$ nodes are assigned supplies according to A. This completely specifies a min cost flow instance. This instance is checked for feasibility by running the network simplex method.

For the experiment we set $n = 20$, $p = 0.2$, $W = C = D = 10$. We generate 1000 graphs for training, 100 for validation and 100 for testing.

5.2 Training procedure

We use a minibatch size of 64 and the Adam optimizer [6] with learning rate 0.0001. We decay the learning rate by multiplying it by 0.95 every 50 iterations.

The loop (as in the pseudocode) is run for 10000 iterations or until it seems to have converged. The iteration we report for the performance on the test set is the one with the best performance on the validation set (we use the validation set to select the "best" version of the model to evaluate for the test set performance).

For the computation graph of the neural network we use the current spanning tree flow and the cycle formed by the entering edge. For node features we have an indicator function for whether they are in the cycle formed or the entering edge and for edge features we have the weights, capacities and flows on the edges for the current solution. We use the original node embedding model of Dai et al [4] due to lack of time and deep learning framework expertise to create the edge embedding model originally proposed. We use an embedding dimension of 64 and 4 node embedding updates for the graph neural network.

5.3 Results

The average number of pivots the neural network takes on the test set is 55.96, the average for the random pivot selection is 65.19 and the average for a combination of Dantzig and Bland's rule (according to its implementation in the NetworkX library in python in version 2.1 [5]) pivot heuristic is 42.96. The neural network pivot selection beats random pivot selection but not the heuristic included in the NetworkX library. The heuristic groups the edges into blocks of size B and selects the entering edge which has the smallest reduced cost within a block and returns it if it is viable, else cycling to the next block to find the entering edge repeating the same computation of the smallest reduced cost edge and so on. The next time an entering edge is to be selected the procedure continues off from the block it exited from. This selection from blocks can force a more varied set of edges to be selected since if a viable edge is within all blocks for a significant portion of pivots then each block will contribute an edge where if we always selected the edge which had the smallest reduced cost then the edges selected may be only a few. So providing a way for the neural network to know its previous pivots may allow it to understand the space of sequences of pivots (making them more varied in some way if that would be better) which might help it choose better pivots.

The difference in sequence length is significant from random pivot selection. We are not sure whether this is due to chosen pivots at the start or the end of the sequence which results in smaller sequence length. Future work with more exact experiments and analysis of the neural network would give a better understanding of what the neural network has learned. This would allow its combination with classical heuristics which may lead to improvements in the state of the art performance in pivot selection.

6 Conclusion and Future Work

Due to time contraints, we didn't use the exact graph embedding we proposed in the write-up in our experiments. We think that using our proposed embedding will capture the tour/tree better and will result in a more powerful neural net, yielding better results. Also we would like to incorporate sequential information into the computation (the swaps or entering edges for the previous k steps). To best incorporate this information, this would require using a neural network model that had sequential computation like a recurrent neural network. A hybrid model between a graph neural network and recurrent neural network might suit this task more and should be investigated.

Our method for the TSP focussed on reaching **some** local optimum in the shortest

number of steps. Using a similar framework as in the write-up, we could also look at learning swaps that result in a relatively large decrease in cost in one step. Applying this to more modern techniques like LKH this may be able to boost their performance if integrated properly. Another thing to look at could be learning good kick methods for local search, or finding a **good** local optima.

We would like to thank Professor William Cook for his help.
We list the references we used below: [5], [6], [4], [3], [2], [1]

References

[1] Min-cost flow problems and network simplex algorithm.

[2] David L. Applegate, Robert E. Bixby, Vasek Chvatal, and William J. Cook. *The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics)*. Princeton University Press, Princeton, NJ, USA, 2007.

[3] William J. Cook, William H. Cunningham, William R. Pulleyblank, and Alexander Schrijver. *Combinatorial Optimization*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[4] Hanjun Dai, Elias B. Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. *CoRR*, abs/1704.01665, 2017.

[5] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[6] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.