Solving TSP with Deep Reinforcement Learning

20180424 윤태영

Abstract

TSP(Traveling Salesman Problem) is an NP-hard problem in combinatorial optimization. I present a stochastic optimization method to successfully find an optimal solution by using reinforcement learning and graph neural network. I apply my method to "pr107" problem and "rl11849" problem. Furthermore, I propose faster RL algorithm that can solve many various TSP problems effiectively.

1. Introduction

The goal of the TSP problem is to find a shortest possible path that visits each city exactly once and returns to the origin city. Since time complexity of a greedy approach to find an optimal solution is O(n!), which is impossible to compute when the number of cities increases, many heuristic algorithms are developed.

Recent advances in deep learning networks and reinforcement learning can be a breakthrough to find an optimal solution. In reinforcement learning, agent explore the environment and get reward corresponding to its action. By doing so, the agent is trained to maximize a reward(or minimize a penalty), resulting in a good performance what we wanted. Nowadays reinforcement learning solves a complex problems with a help of deep learning networks. Deep learning networks such as convolutional neural network[1] and graph neural network[2] make agent get optimal policies from complex RL environments.

In this report, I demonstrate a graph neural network can help RL agent find an optimal control polices tov solve a TSP problem. The network get graph that extracts information from TSP problem as an input and return updated graph, whose nodes are communicated with each other through edges. I use a DQN agent[3] to solve the problem to alleviate correlated data and non-stationary problems.

I apply my RL algorithm to three TSP problems, "pr107", "rl1304", and "rl11849". "pr107" problem has 107 cities to visit, and rl11849 has 11849 cities to visit. "pr107" and "rl1304" is a quite smaller task, and "r111849" is large problem that greedy algorithm will never reach to optimal solution. My goal is to create a graph neural network DQN agent that is able to successfully learn to solve many TSP problems from trivial to large-scale. The agent was not provided with any information about the problem, it learned from nothing but the graph that I constructed, the reward and terminal signals, and the set of possible actions—just as a salesman who has to visit each city would.

Preceding RL method is effective, but constructing a graph is memory-wasting, especially when trying to solve bigger TSP problems, such as "rl11849". Therefore, I suppose a new RL algorithm with much faster speed and effectiveness. In this time, RL agent is given a random path, and choose 2 cities to reverse subpath where starting point and end point are chosen cities in the given path. I use DDPG framework to implement my algorithm.

2. Background

I consider TSP problem as an environment $\boldsymbol{E}$ where an agent interacts with, in a sequence of actions(where to visit next step), observations(state from graph neural network), and rewards(distance from source to destination). At each time step, the agent select an action a_t from the set of possible action space $\boldsymbol{A}$**.** Then the agent moves to next city that directed by a_t and modify a graph information and get a reward r_t, distance between previous city and current city.

The goal of the agent is to interact with the environment by selecting actions in a way that maximizes future rewards. I make the standard assumption that used in many RL algorithm that future rewards are discounted by a factor of γ per time-step, and define the future discounted return at time t as $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$, where T is the time-step at which the salesman returns to the original city. I define the optimal action-value function $Q^*(s,a)$ as the maximum expected return achievable by following any strategy, after seeing some state s and then taking some action a, $Q^*(s,a) = \max_{\pi} E[R_t | s_t = s, a_t = a, \pi]$, where π is a policy mapping states to actions.

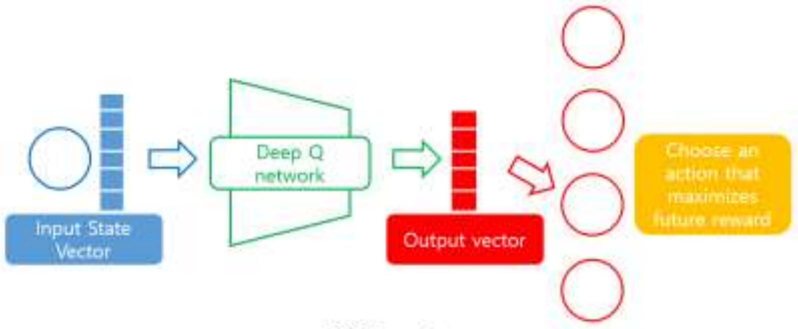
It is common to use a function approximator to estimate the action-value function, $Q(s,a;\theta) \approx Q^*(s,a)$. I use a neural network function approximator with weights θ as a Q-network. A Q-network can be trained by minimizing a sequence of loss functions $L_i(\theta_i)$ that changes at each iteration i,

$$\boldsymbol{L_i(\theta_i) = E_{s,a\sim p}[(y_i - Q(s,a;\theta_i))^2]},$$

where $y_i = E_{s'}[r + \gamma \max_{a'} Q(s',a';\theta_{i-1})]$is the target for iteration i and $\rho(s,a)$ is a probability distribution over states s and actions. The parameters from the previous iteration θ_{i-1} are held fixed when optimizing the loss function $L_i(\theta_i)$. Differentiating the loss function with respect to the weights I arrive at the following gradient,

$$\boldsymbol{\nabla_{\theta_i} L_i(\theta_i) = E_{s,a\sim\rho,s'}[(r + \gamma \max_{a} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i))\ \nabla_{\theta_i} Q(s,a;\theta_i)]}$$

It is also off-policy learning method: it learns about the greedy strategy $\mathrm{a} = \operatorname{argmax}_{a} Q(s,a;\theta)$, while following a behavior distribution that ensures adequate exploration of the state space. In practice, the behavior distribution is often selected by an ϵ-greedy strategy that follows the greedy strategy with probability $1-\epsilon$ and selects a random action with probability ϵ.[3] Below figure summarizes how DQN agent learn policy.



<DQN agent>

I apply a graph neural network so that the agent can choose an optimal action with the information of not only it can get from its position, but also information propagated from other cities so that it can look further.

Now I explain my graph neural network. For each time step in the environment, the agent receives an observation $s = (x_1, \dots, x_n)$, where n is a number of nodes. Then the observation vector goes through an input network to obtain a fixed-size state vector as follows:

$$\boldsymbol{h_u^0 = F_{in}(x_u)},$$

where the subscript and superscript denote the node index and propagation step respectively. I use MLP as an input network, F_{in}.

At propagation step, for every edge $(u, v) \in E$, node *u* computes a message vector as below,

$$\boldsymbol{m_{(u,v)}^t = M_{(u,v)}(h_u^t)},$$

where $M_{(u,v)}$ is the message function. I just copy edge feature as message vector.

Once every node finishes computing messages, I aggregate messages sent from all in-coming adjacent nodes. In particular, I perform the following aggregation.

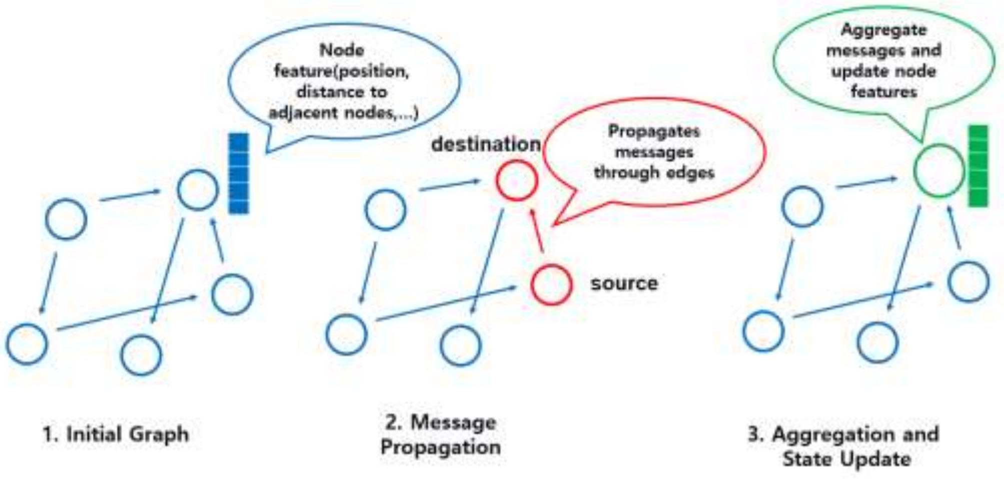
$$\boldsymbol{m_u^t = A(\{h_u^t | v \in E_{in}(u)\})},$$

where *A* is the aggregation function, which may be a summation, average, or max-pooling function. I use summation as an aggregation function.

Finally we update every node's state vector based on both the aggregated message and its current state vector. In particular, we perform the following update,

$$\boldsymbol{h_u^{t+1} = U(h_u^t, m_u^t)}$$

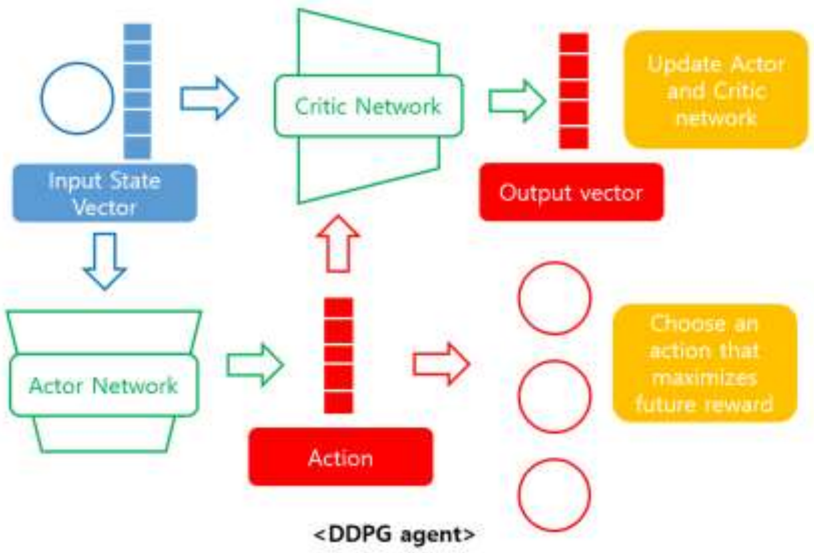
where *U* is the update function. I use MLP as an update function approximator. The following figure summarizes how my graph neural network model works.[2]


Node feature(position, distance to adjacent nodes,...)
destination
Propagates messages through edges
source
Aggregate messages and update node features
1. Initial Graph
2. Message Propagation
3. Aggregation and State Update

For second RL agent who choose two cities in path and inverse the subpath, I benchmark DDPG framework. Therefore I use actor-critic approach based on DPG algorithm. The DPG algorithm estimate actor as parametrized actor function $\mu(s|\theta^{\mu})$ which specifies the policy by deterministically mapping states to a specific action in a continuous space. The critic $Q(s,a)$ is learned using Bellman equation as in DQN. I use MLP to implement actor and critic network. The actor is updated by policy gradient, given by

$$\nabla_{\theta^{\mu}} J = E_{s_t \sim \rho^{\beta}}[\nabla_a Q(s,a|\theta^{Q})|_{s=s_t,a=a_t} \nabla_{\theta^{\mu}} \mu(s|\theta^{\mu})|_{s=s_t}]$$

Following figure summarizes how DDPG algorithm works.[4]


Input State Vector
Critic Network
Output vector
Update Actor and Critic network
Actor Network
Action
Choose an action that maximizes future reward
<DDPG agent>

3. Algorithm

I utilize off-policy learning strategy by experience replay, where I store the agent's experiences at each time step and update deep Q network using sample from the memory. I consider cities as a node, and edge is a path that connects cities. The full algorithm is presented in Algorithm 1.

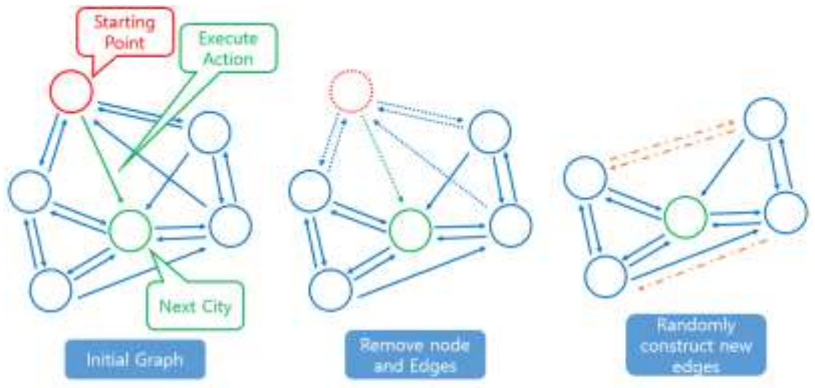
Algorithm 1 Deep Q-Learning with Graph Neural Network

```
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights
Initialize graph neural network $G = (V, E)$ with random weights

For node = $1,\dots,|V|$ do
    Calculate distances between the node and adjacent node to get distribution of distances $d$
    Initialize state $s^{node} = \{x, y, d_1, \dots, d_k\}$
End for

For episode = $1,\dots,M$ do
    Randomly choose a start point $i_0$
    Set $s_1 = s^{node}$ and preprocessed state $G_1 = G(s_1)$
    For t = $1,\dots, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \mathrm{argmax}_a Q^*(G(s_t), a, i_t; \theta)$
        Execute action $a_t$ and observe reward $r_t$, next state $s_{t+1}$ and next
```

```
            city $i_{t+1}$
            Modify graph $\boldsymbol{G}$ to $\boldsymbol{G'}$ by remove node $i_t$ and related edges
            Set $\boldsymbol{s_{t+1}} = \boldsymbol{s_t}$ and preprocess $\boldsymbol{G_{t+1}} = \boldsymbol{G'}(\boldsymbol{s_{t+1}})$
            Store transition $(\boldsymbol{G_t}, \boldsymbol{i_t}, \boldsymbol{a_t}, \boldsymbol{r_t}, \boldsymbol{G_{t+1}}, \boldsymbol{i_{t+1}})$ in $\boldsymbol{D}$
            Sample random minibatch of transitions $(\boldsymbol{G_j}, \boldsymbol{i_j}, \boldsymbol{a_j}, \boldsymbol{r_j}, \boldsymbol{G_{j+1}}, \boldsymbol{i_{j+1}})$ from $\boldsymbol{D}$
            Set $\boldsymbol{y_j} = \begin{cases} r_j & \text{for terminal } G_{j+1} \\ r_j + \gamma \max_{a'} Q(G_{j+1}, a', i_{j+1}; \theta) & \text{for non-terminal } G_{j+1} \end{cases}$
            Perform a gradient descent step on $\left(\boldsymbol{y_j} - \boldsymbol{Q}(\boldsymbol{G_j}, \boldsymbol{a_j}, \boldsymbol{i_j}; \boldsymbol{\theta})\right)^2$
        End for
    End for
```

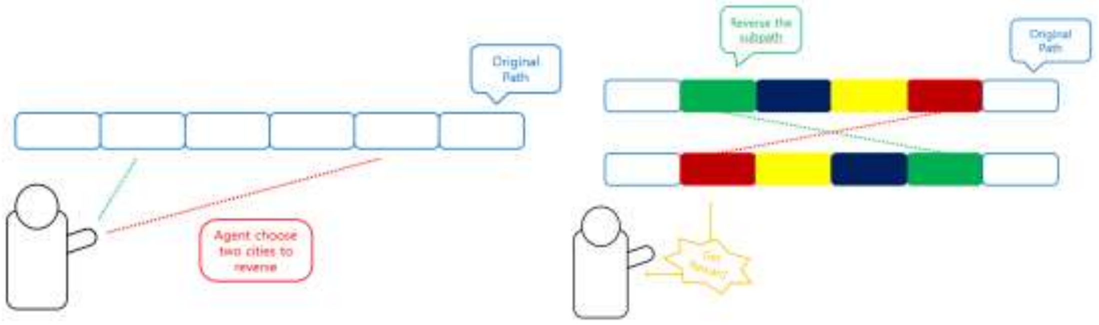
Now I'll talk about details of my algorithm. First, I set the initial state vector contains information about current node's position, the number of adjacent nodes which are classified by distance from the current node so that agent can learn current city is in rural, suburban area, or an urban core. Edge is constructed randomly but sorted by distance. In particular, each node is connected to 12 randomly chosen adjacent nodes since connect all nodes each other is too complex and it becomes a hard task since the agent has a lot of choice alternatives as the number of nodes increases. I give an agent a penalty reward after action executed, by distance from current city to next city chosen by agent's action.

At each time step, agent execute an action and move to next city. Then I modify a graph by removing previous node and edges connected to the node. To preserve the number of edges so that agent have always same number of candidates, I randomly connect nodes who lost one of their edges and sort their edges by distance so that agent's policy becomes stable. The following figure is an abbreviated version of my algorithm.



Graph Modification

However, when I try to solve "rl11849" problem, the scale of graph is too large so that it is impossible to construct a graph with my laptop. Even I use desktop in KAIST, it is too slow algorithm because of constructing graph. Therefore, I try to find faster algorithm, but still maintain RL framework. So I consider an agent which is given a random path to travel all cities, and choose two cities to reverse subpath where starting point and end point are chosen cities in the given path. The agent is trained by a reward, which is given by difference of path length between the original path and the path after reverse. Following figure describes agent’s action execution and how to get a reward from the environment.



I use DDPG framework to implement the agent which choose two cities since DDPG algorithm can train agent to learn continuous action. Even though our situation has discrete action space, since there are lots of cities, choosing a continuous action space might be reasonable rather than a discrete action space with dimension more than 10,000 in “rl11849” case. I give a reward by difference of path length between the original path and the path after swapping two cities chosen by an agent. For exploration, I use ε-greedy policy instead of generating Ornstein-Uhlenbeck process since Ornstein-Uhlenbeck process generates temporally correlated variables, which is not our case. The full algorithm is presented in Algorithm 2

Algorithm 2 Reverse Subpath Algorithm using Deep Deterministic Policy Gradient

```
Initialize critic network $\boldsymbol{Q(s,a|\theta^{\mu})}$ and actor $\boldsymbol{\mu(s|\theta^{\mu})}$ with random weights
  $\boldsymbol{\theta^{Q}}$ and $\boldsymbol{\theta^{\mu}}$
Initialize replay buffer $\boldsymbol{R}$
Initialize random path $l_1$
For episode = 1,…,M do
    Receive initial observation state $s_1 = l_1$
    For t = 1,…, T do
        With probability $\boldsymbol{\epsilon}$ select a random action $\boldsymbol{a_t}$
        otherwise select $\boldsymbol{a_t = \mu(s_t|\theta^{\mu})}$ according to the current policy
        Execute action $\boldsymbol{a_t}$ and observe reward $\boldsymbol{r_t}$, next state $\boldsymbol{s_{t+1}}$
        Store transition $\boldsymbol{(s_t, a_t, r_t, s_{t+1})}$ in $\boldsymbol{D}$
        Sample random minibatch of transitions $\boldsymbol{(s_j, a_j, r_j, s_{j+1})}$ from $\boldsymbol{D}$
        Set $\boldsymbol{y_j = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})}$
        Update critic by minimizing loss: $\boldsymbol{L = \frac{1}{N}\sum_i \left(y_i - Q(s_i, a_i|\theta^{Q})\right)^2}$
        Update the actor policy using the sampled policy gradient:
            $\boldsymbol{\nabla_{\theta^{\mu}} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^{Q})|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^{\mu}} \mu(s|\theta^{\mu})|_{s_i}}$
        Update the target networks:
                        $\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'}$
                        $\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'}$

    End for
    Set $l_1$ be the shortest path in the episode
End for
```

4. Experiments

I performed experiments on two TSP problems – “pr107”, “rl1304”, “rl11849”. I use the same network architecture, but clipping the reward since distribution of distances between cities varies greatly from problem to problem. For each game, I find maximum distance between two cities, and at each execution of action, I give a reward that scaled by maximum distance to be 1. Figure 1 shows how total reward and performance evolves during training DQN agent in “pr107” problem. Since it takes too long time to construct a graph in “rl1304”, and “rl11849” problems, I omit those experiments.

Figure 2 shows performance of DDPG agent which exchanges two cities each time step in “pr107”, “rl1304”, “rl11849”. I train each problem with 500 episodes, and each episode performs 200 reverse operations. Best performance of the DDPG agent for “rl11849” problem is 58,289.66(best: 44303), 5,018,484.64(best: 252948), 77,761,109.62(best: 920847). From the form of performance graph of “pr107” problem. Better solution can be achieved by training the agent with more episodes.

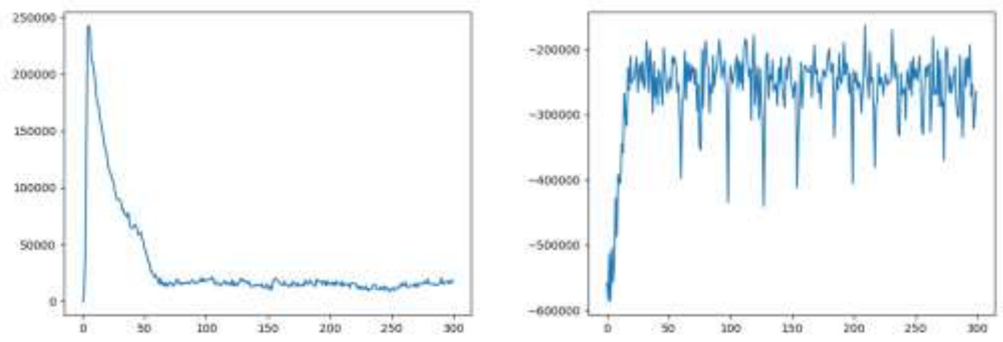


Figure 1: Loss(Left) and Reward(Right) of DQN agent training by “pr107” problem. Reward is minus value of total traveling time.

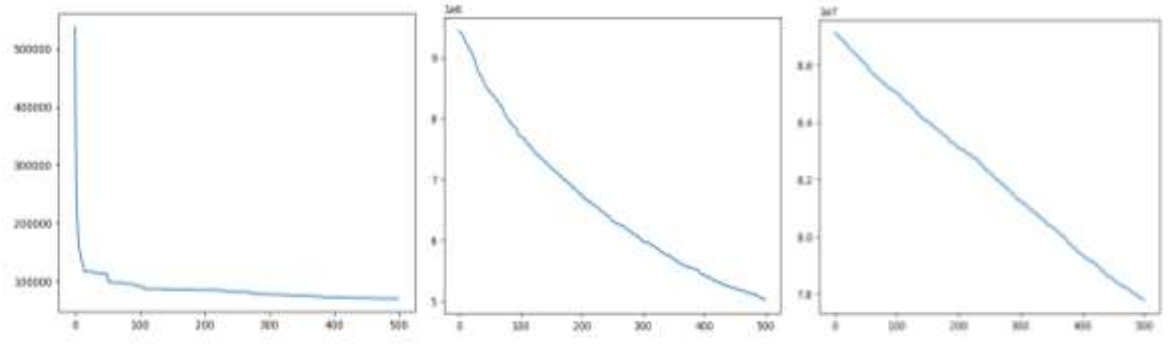


Figure 2: Performance of DDPG agent training by “pr107”(left), “rl1304”(middle), “rl11849”(right) respectively.

5. Conclusion

I try to solve TSP problem using representative RL agent, with graph neural network so that it can get information of structure of city network. My approach can be developed better as I improve my graph network can represent the structure of city network more clearly and memory-efficient(maybe I need a good computer). I think my second approach is quite creative, and by improving state information and training agent with more episodes, I think it can achieve better performance.

6. **References**

[1] A.Krizhevsky, I Sutskever, GE Hinton. Imagenet classification with deep convolutional neural networks

[2] T Wang, R Liao. NerveNet: Learning Structured Policy with Graph Neural Networks

[3] V Mnih. Playing Atari with Deep Reinforcement Learning

[4] TP Lilicrap. Continuous control with Deep Reinforcement Learning