Reinforcement Learning for Combinatorial Optimization

Saiyue Lyu
School of Computer Science
University of Waterloo
s6lyu@uwaterloo.ca

Abstract

Combinatorial optimization is a topic that aims at finding optimal solutions and designing efficient algorithms for optimization problems over discrete structures. Typical problems are the traveling salesman problem (TSP), the Minimum Vertex Cover problem (MVC), Maximum Cut problem (MAXCUT) and the knapsack problem. Researches have built up powerful frameworks to leverage Reinforcement Learning (RL) to automate the process of designing good heuristics and approximation algorithms. This survey will give a review of recent breakthrough.

1 Introduction

Many real-world problems such as transportation, scheduling and social networks can be reduced to combinatorial optimization on a discrete structure. Combinatorial optimization problems are mostly well-studied theoretically and many exact or approximate algorithms have been proposed. There are several canonical combinatorial optimization problems :

Traveling Salesman Problem (TSP) : (Euclidean Space) Given a set of nodes in 2-dimensional space, find a tour of minimum total weight, where the corresponding graph G is fully connected with edge weights corresponding to distances between points.

Minimum Vertex Cover Problem (MVC) : Given a graph G, find a subset of nodes $S \subseteq V$ such that every edge is covered, i.e. $(u, v) \in E \Leftrightarrow u \in S$ or $v \in S$ and $|S|$ is minimized.

Maximum Cut Problem (MAXCUT) : Given a graph G, find a subset of nodes $S \subseteq V$ such that the weight of the cut set $\sum_{(u,v)\in C} w_{u,v}$ is maximized, where the cut set $C \subseteq E$ is the set of edges with one end in S and the other end in $V \setminus S$.

The KnapSack Problem (KnapSack) : (0-1 Knapsack) Given a set of n items each with weight w_i and value v_i and a maximum weight capacity of W, maximize the sum of the values of items present in the knapsack such that the sum of the weight is less than or equal to the knapsack capacity W.

Over the past several decades, there have been remarkable advances in the development of approximation algorithms of these problems. For example, TSP, first introduced by the Irish mathematician W.R. Hamilton in the 1800s, has been well-studied with both exact and approximation algorithms for Euclidean and non-Euclidean graphs. After years, Beardwood, Halton and Hammersley (1959) present a practical solution by deriving an asymptotic formula to determine the length of shortest route for fixed number of cities. Karp (1972) showed that the Hamilton cycle problem is NP-complete, which implies the NP-hardness of TSP. Christofids (1976) finds an algorithm for minimum spanning tree (MST) to return an approximate solution for TSP in polynomial time. After years of great progress, Cook (1990) develop the program Concorde to tackle TSP and record recent solutions. Later Cook (2006) computed an optimal route given 85,900 cities, which is currently the

Submitted to Reinforcement Learning (CS885, Spring 2020)

largest solved TSPLIB instance. To generalize the method, the difficulty in applying search algorithms to new instances of similar problem is No Free Lunch theorem (Wolpert & Macready, 1997) due to the fact that all search heuristics have the same performance when averaged over all problems, one must appropriately rely on a prior over problems when selecting a search algorithm to guarantee performance.

The design of good heuristics of combinatorial optimization problems is often sensitive to constraints, and even a slight change can require more significant specialized knowledge. So one of the greatest challenge is applying existing search heuristics to newly encountered problems, though researchers used "hyper-heuristics" to generalize the system, more or less, human created heuristic is needed. Another concern is that these problems are often NP-hard and analytically intractable. Approximation algorithms guarantee a worst-case solution, but sufficiently strong bounds may not exist, even if they do, these algorithms can have limited scalability (Williamson & Shmoys, 2011).

Given above challenges, the application of neural networks to combinatorial optimization has been quite a hot topic. Hopfield and Tank (1985) use Hopfield network and modify TSP objective to get the equivalent network energy function, which can be solved using Lagrange multipliers. But Wilson and Pawley (1988) point out the limitation that it is sensitive to hyperparameters and parameter initialization. Motivated by the advancements in sequence-to-sequence learning, Bello *et al.*(2016) design the framework using RL to tackle this problem, they used policy gradients to train pointer networks (Vinyals *et al.*, 2015), a recurrent architecture that produced a softmax attention mechanism (a "pointer") to select a member of the input sequence as an output.

2 Background

Policy Gradient

RL wants to find an optimal behavior strategy for the agent to obtain optimal rewards. The policy is modeled with a parameterized function respect to θ and the value of the reward function depends on this policy :

$$J(\theta) = \sum_{s \in \mathcal{S}} d^{\pi}(s) \sum_{a \in \mathcal{A}} \pi_{\theta}(a|s) Q^{\pi}(s,a)$$

where $d^{\pi}(s)$ is the stationary distribution of Markov chain for π_{θ}. The policy gradient methods aims to model and optimize the policy directly, so various algorithms can be applied. But computing the gradient $\nabla_{\theta} J(\theta)$ is tricky since it depends on both the action selection and the stationary distribution of states following the target selection behavior. Sutton and Barto (2017) proves a powerful theorem, the policy gradient theorem :

$$\begin{aligned}
\nabla_{\theta} J(\theta) &= \nabla_{\theta} \sum_{s \in \mathcal{S}} d^{\pi}(s) \sum_{a \in \mathcal{A}} Q^{\pi}(s,a) \pi_{\theta}(a|s) \\
&\propto \sum_{s \in \mathcal{S}} d^{\pi}(s) \sum_{a \in \mathcal{A}} Q^{\pi}(s,a) \nabla_{\theta} \pi_{\theta}(a|s) \\
&= \mathbb{E}_{s \sim d_{\pi}, a \sim \pi_{\theta}} \left[Q^{\pi}(s,a) \nabla_{\theta} \ln \pi_{\theta}(a|s) \right]
\end{aligned}$$

The policy gradient theorem lays the theoretical foundation for various algorithms, it provides a nice reformation of the derivative of the objective function to not involve the derivative of the state distribution d^{π} and simplify the gradient computation very much.

REINFORCE

REINFORCE algorithm (Williams, 1992), a Monte-Carlo policy gradient, relies on an estimated return by Monte-Carlo methods using episode samples to update the policy parameter θ, the process is as follows :

1. Initialize the policy parameter θ at random.
2. Generate one trajectory on policy $\pi_{\theta} : \mathcal{S}_1, \mathcal{A}_2, \cdots, \mathcal{S}_T$
3. For $t = 1, \cdots, T$:

   Estimate the return G_t and update policy parameters :$\theta \leftarrow \theta + \alpha \gamma^t G_t \nabla_{\theta} \ln \pi_{\theta}(\mathcal{A}_t|\mathcal{S}_t)$.

Note that $Q^\pi(\mathcal{S}_t, \mathcal{A}_t) = \mathbb{E}_{s\sim d_\pi, a\sim\pi_\theta}\big[G_t \nabla_\theta \ln \pi_\theta(\mathcal{A}_t|\mathcal{S}_t)\big]$, and the expectation of the sample gradient equals the actual gradient, so REINFORCE works.A widely used variation of REINFORCE is to subtract a baseline value from the return G_t to reduce the variance of gradient estimation while keeping the bias unchanged.

Actor-Critic

Tow main components in policy gradient are the policy model ad the value function. Actor-Critic helps to understand the value function, which assist the policy update. Actor-Critic algorithm consists of two models, Critic and Actor. Critic updates the value function parameter w and it could be action=value $Q_w(a|s)$ or state-value $V_w(s)$ depending on the algorithm. Actor updates the policy parameter θ for $\pi_\theta(a|s)$ in the direction suggested by the critic. Before training, two learning rate α_θ and α_w are predefined for policy and value function parameter updates respectively. A simple action-value actor-critic algorithm is as follows :

1. Initialize s, θ, w at random; sample $a \sim \pi_\theta(a|s)$.
2. For $t = 1, \cdots, T$:

   Sample reward $\gamma_t \sim R(s, a)$ and next states $s' \sim P(s'|s, a)$;

   Sample the next action $a' \sim \pi_\theta(a'|s')$;

   Update the policy parameters $\theta \leftarrow \theta + \alpha_\theta Q_w(s, a)\nabla_\theta \ln \pi_\theta(a|s)$;

   Compute the correction for action-value at time t : $\delta_t = \gamma_t + \gamma Q_w(s', a') - Q_w(s, a)$;

   Update the parameters of action-value function : $w \leftarrow w + \alpha_w \delta_t \nabla_w Q_w(s, a)$;

   Update $a \leftarrow a'$ and $s \leftarrow s'$.

Q-learning

Let $\mathcal{S}$ be the set of states, $\mathcal{A}$ be the set of actions, $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ be the transition function, $\mathcal{R} : \mathcal{S} \to \mathbb{R}$ be the reward function and $\gamma \in [0, 1]$ be the discount factor, $\pi : \mathcal{S} \to [0, 1]$ be a policy mapping a state to a probability distribution over actions. Consider a Markov Decision Process (MDP) defined by $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma)$, the Q-value of a given pair (s, a) is given by :

$$Q^\pi(s, a) = \mathbb{E}\big[\sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t) \,|\, s_0 = s, a_0 = a, \pi\big]$$

Mnih (2015) proposed a deep Q-network (DQN) with parameters θ. The Q-values of each state-action pair is approximated by $Q^*(s, a) = \max_\pi Q^\pi(s, a)$, which is trained by the θ parameterized network. And an approximated optimal policy is given by acting greedily with respect to the predicted Q-values : $\pi(s; \theta) = \arg\max_a Q(s, a; \theta)$.

Sequence-to-Sequence Model

Given a training pair, $(\mathcal{P}, \mathcal{C}^\mathcal{P})$, where $\mathcal{P} = \{P_1, \cdots, P_n\}$ is a sequence of n vectors and $\mathcal{C}^\mathcal{P} = \{C_1, \cdots, C_{m(\mathcal{P})}\}$ is a sequence of $m(\mathcal{P})$ indices, each between 1 and n, the sequence-to-sequence model uses an recurrent neural network (RNN) with parameters θ to estimate the conditional probability :

$$p(\mathcal{C}^\mathcal{P}|\mathcal{P}; \theta)) = \prod_{i=1}^{m(\mathcal{P})} p_\theta(C_i|C_1, \cdots, C_{i-1}, \mathcal{P}; \theta)$$

where the learnt parameters θ^* is given by :

$$\theta^* = \arg\max_\theta \sum_{\mathcal{P}, \mathcal{C}^\mathcal{P}} \log p(\mathcal{C}^\mathcal{P}|\mathcal{P}; \theta))$$

The model makes no statistical independence assumptions, so we consider two seperate RNN models, encoder and decoder, both of them use Long Short Term Memory (LSTM) cells (Hochreiter & Schmidhuber, 1997), which is a mechanism helping the model to keep track of long-distance dependencies more easily. The encoder steps through an input string one element at a time and transform the sequence into an output that it gives to the decoder. The decoder then unpacks the encoder's output one element at a time, creating the model's output sting, which can be illustrated as :

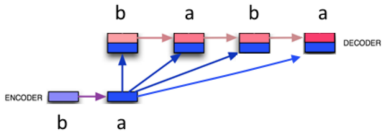

Figure 1: A Sequence-to-Sequence Model (Nag, 2016)

3 Reinforcement Learning Approaches for Typical Problems

3.1 Pointer Network

First we take a close look at TSP for 2-dimensional Euclidean, which is formally formulated as : given a graph G with a sequence of n nodes $\{x_i\}_{i=1}^n$ in a 2-dimensional spaces, find a permutation of the points π, i.e. a tour, that visits each city once and has the minimum total length, where the length of a tour is defined with respect to l_2 norm :

$$L(\pi|s) = \|x_{\pi(n)} - x_{\pi(1)}\|_2 + \sum_{i=1}^{n-1} \|x_{\pi(i)} - x_{\pi(i+1)}\|_2$$

The main task is to assigns high probabilities to short tours and low probabilities to long tours naturally. To achieve this, we need to learn the parameters of a stochastic policy $p(\pi|s)$ via the neural network architecture, which uses the chain rule to factorize the probability (Sutskever, 2014) of a tour as :

$$p(\pi|s) = \prod_{i=1}^{n} p\big(\pi(i)|\pi(< i), s\big)$$

Naturally one can use a vanilla sequence-to-sequence model to tackle TSP when the output vocabulary is $[n] = \{1, \cdots, n\}$. But the trained network can not be generalized for more than n nodes of inputs. Another issue is that in order to optimizing the parameters with conditional log-likelihood, access to ground-truth output permutations are needed. So Bello *et al.*(2016) used policy gradients to train Pointer Network (Vinyals *et al.*, 2015), a recurrent architecture that produces a softmax attention mechanism to select a member of the input sequence as an output.

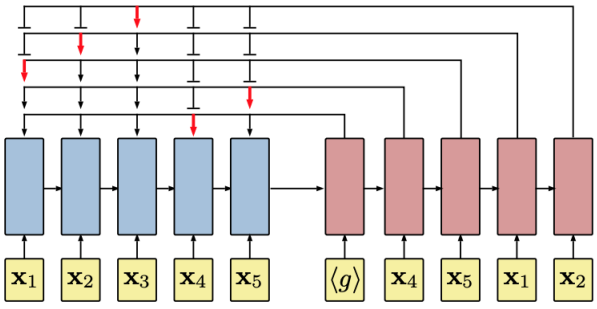

Figure 2: A Pointer Network Architecture (Vinyals et al., 2015)

The TSP problem is encoded as a sequence with encoder RNN θ_{enc}, i.e. the encoder takes the input sequence s, one node at one time and transforms it into a sequence of latent memory states $\{enc_i\}_{i=1}^n \subseteq \mathbb{R}^d$. And the solution is decoded as a sequence with decoder RNN θ_{dec}, i.e. the decoder also obtain its latent memory states $\{dec_i\}_{i=1}^n \subseteq \mathbb{R}^d$ and uses a pointing mechanism to return a distribution of the next visit city at each step. Then they optimized the parameters using policy gradients, where the training objective is the expected tour length, given an input graph s :

$$J(\theta|s) = \mathbb{E}_{\pi \sim p_\theta(\cdot|s)} L(\pi|s)$$

They uses the REINFORCE trick (Williams, 1992) to express gradient as an expectation and approximate gradient with Monte Carlo sampling given that $\ln p_\theta$ is differentiable :

$$\nabla_\theta J(\theta|s) = \mathbb{E}_{\pi\sim p_\theta(\cdot|s)}\Big[\big(L(\pi|s) - b(s)\big)\nabla_\theta \ln p_\theta(\pi|s)\Big]$$

$$\nabla_\theta J(\theta) \approx \frac{1}{B}\sum_{i=1}^{B}\Big(L(\pi_i|s_i) - b(s_i)\Big)\nabla_\theta \ln p_\theta(\pi_i|s_i)$$

But the problem is that this gradient has lots of variance since most of the time the cost will be very large and it can be very slow to train, so in some cases they use a more recent algorithm, Actor-Critic training, which introduces an auxiliary network parameterized by θ_v, a Critic, to encode $b_{\theta_v}(s)$ and evaluate the expected tour length under the parameter. Instead of weighting by the extra cost, the additional objective subtracts a bias which is also an estimate of how the cost will be, i.e.

$$\mathcal{L}(\theta_v) = \frac{1}{B}\sum_{i=1}^{B}\|b_{\theta_v}(s) - L(\pi_i|s_i)\|_2^2$$

where $b(s)$ is the expected tour length $\mathbb{E}_{\pi\sim p_\theta(\cdot|s)[L(\pi|s)]}$. It is trained with Mean Squared Error (MSE) gradient descent to reduce variance of gradient approximation. So it will gives a faster result.

For the inference, note that the optimal inference is NP-hard due to the chain factorization, there are many possible alternative ways to get an approximate inference :

Greedy Decoding : Look at the max of the first city and take it as the input as the next step. At each step in the decoder, select the next city with highest probability.

Beam Search : Instead of just looking at the max, look at the B max. Greedily maintain B candidate solutions based on the probability of the subsequences so far.

Sampling : Sample many solutions from $p_\theta(\cdot|s)$ and keep track of the best one.

Active Search : Train parameter θ with REINFORCE on single input problem and keep track of best solution found during training.

With experiments on Euclidean TSP with 20, 50 and 100 cities in the unit square, despite the computational expense, this structure achieves close to the optimal results. This can also be applied to the KnapSack, which will obtain optimal solutions for instances with up to 200 items.

3.2 Structure2Vec Deep Q-learning (S2V-DQN)

Bello *et al.* (2016) used policy gradients to train Pointer Network, however, this architecture did not reflect the structure of problems defined over a graph as their model only works on the Euclidean TSP problem,where each node is represented by its 2-dimensional coordinates, and it is not designed for problems with graph structure. And it often require a huge number of instances to learn to generalize to new ones. What's more, the policy gradient is not particularly sample-efficient. To handle these issues, Khalil *et al.*(2017) addressed S2V-DQN, a general RL based network for combinatorial optimization problems that uses a combined graph embedding network and deep Q-network. The particular problem statement they addressed is that : Given a graph optimization problem G and a distribution $\mathbb{D}$ of the problem instances, can we learn better heuristics that generalize to unseen instances from $\mathbb{D}$? For example, given the integer programming formulation of MVC :

$$\min_{x_i\in\{0,1\}} \sum_{i\in V} x_i$$
$$s.t.\ x_i + x_j \geq 1, \forall (i,j) \in E$$

We consider a constant reward $\gamma^t = -1$ and the Q-function takes a current partial solution S in their candidate vertex as input and returns additional number of vertices required, i.e. the action value function is $\hat{Q}(S, v)$, where state S is the current partial solution. And the greedy policy selects the vertex with largest score at each step, i.e. $v^* = \arg\max_v \hat{Q}(S, v)$. But how to represent the action value function $\hat{Q}(S_t, v; \theta)$?

To represent the action value function, one can use feature engineering, but it is hard to construct and generalize. Dai *et al.*(2017) used *structure2vec*, a graph embedding network for better graph representation, which they introduced in 2016. The *structure2vec* computes each node embedding as a nonlinear function of neighborhood embedding iteratively. In the last layer, it predicts the Q-function on each node. This procedure can be illustrated as follows :

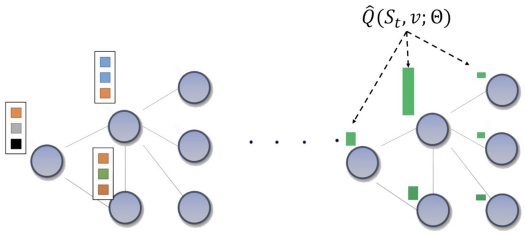

Figure 3: Structure2Vec (Dai et al., 2016)

This graph embedding network computes a $p-$dimensional feature embedding μ_v for each node $v \in V$ given the current partial solution S. The node-specific tags or features x_v are aggregated recursively according to G's graph topology. It begins with initialization of the embedding $\mu_v^{(0)}$ at each node as 0 and for all $v \in V$, it update the embeddings synchronously at each iteration as :

$$\mu_v^{(t+1)} \leftarrow F\Big(x_v, \{\mu_u^{(t)}\}_{u \in \mathcal{N}(v)}, \{w(v,u)\}_{u \in \mathcal{N}(v)}; \theta\Big)$$

where $\mathcal{N}$ is the set of neighbors of node v in graph G and F is a generic nonlinear mapping such as a neural network or kernel function.

There is one more step we need to do, that is, to formulate these combinatorial problems using a common way for greedy algorithm. Consider a problem instance G is sampled from a distribution $\mathbb{D}$, a partial solution is represented as an ordered list $S = (v_1, \cdots, v_{|S|})$, $v_i \in V$ and $\bar{S} = V \setminus S$. Define a indicator variable x such that $x_v = 1$ for all $v \in S$ and 0 otherwise. They assume a given helper function $h(S)$, which maps S to a combinatorial structure satisfying the specific constraints of a problem. For example, for MVC, the helper function does not need to do any work while for MAXCUT, the helper function divides V into two sets S and its complement $\bar{S}$, and maintains a cut-set $C = \{e = (u,v) \in E \,|\, u \in S,\, v \in \bar{S}\}$. And for TSP, the helper function maintain a tour according to the order of the nodes in S .The quality of a partial solution S is given by $c\big(h(S), G\big)$, an objective function based on h. Then a greedy algorithm selects a node v to add next such that v is a maximizer of an evaluation function $Q\big(h(S), v\big) \in \mathbb{R}$, which depends on $h(S)$. The partial solution S will be extended as $S \leftarrow (S, v^*)$ by appending v^*, where $v^* = \arg\max_{v \in \bar{S}} Q\big(h(S), v\big)$. They also assume the termination criterion $t\big(h(S)\big)$ and the cost function c are given. For example, for MVC, the cost function $c\big(h(S), G\big) = -|S|$ and the termination criterion checks whether all edges are covered.

Then we can discuss how to parameterize $\hat{Q}\big(h(S), v; \theta\big)$ using *structure2vec*. Dai *et al.*(2017) defined the nonlinear update rule F to behave as :

$$\mu_v^{(t+1)} \leftarrow \text{Relu}\Big(\theta_1 x_v + \theta_2 \sum_{u \in \mathcal{N}(v)} \mu_u^{(t)} + \theta_3 \sum_{u \in \mathcal{N}(v)} \text{Relu}\big(\theta_4 w(v,u)\big)\Big)$$

where $\theta_1 \in \mathbb{R}^p$, $\theta_2, \theta_3 \in \mathbb{R}^{p \times p}$ and $\theta_4 \in \mathbb{R}^p$ are the model parameters. After T iterations, we can define the $\hat{Q}\big(h(S), v; \theta\big)$ to be :

$$\hat{Q}\big(h(S), v; \theta\big) = \theta_5^\top \text{Relu}\Big(\big[\theta_6 \sum_{u \in V} \mu_u^{(T)}\,,\, \theta_7 \mu_v^{(T)}\big]\Big)$$

where $\theta_5 \in \mathcal{R}^{2p}$, $\theta_6, \theta_7 \in \mathcal{R}^{p \times p}$. So our parameters $\theta = \{\theta_i\}_{i=1}^7$. The embedding representation allows the policy to discriminate among nodes and generalizes to problem instances of different sizes, which fully exploit graph structure unlike previous sequence-to-sequence mapping.

Dai *et al.*(2017) used a DQN algorithm to learn a greedy parameterized policy, where a states S is a sequence of actions (nodes) on a graph G, an action v is a node of G that is not part of the current state S, the reward function $r(S, v)$ at state S is defined to be the change on the cost function, i.e. $r(S, v) = c\big(h(S'), G\big) - c\big(h(S), G\big)$, where $S' = (S, v)$ is the new state, and a deterministic greedy policy $\pi(v|S) = \arg\max_{v' \in \bar{S}} \hat{Q}\big(h(S), v'\big)$. The main DQN algorithm is a combination of $n-$step Q-learning and fitted Q-iteration. At the step t, v_t is updated to a random node $v \in \bar{S}_t$ with probability ϵ and to $\arg\max_{v \in \bar{S}_t} \hat{Q}\big(h(S_t), v; \theta\big)$ with probability $1 - \epsilon$. The partial solution is updated by appending the updated v_t. The overall framework can be illustrated as :

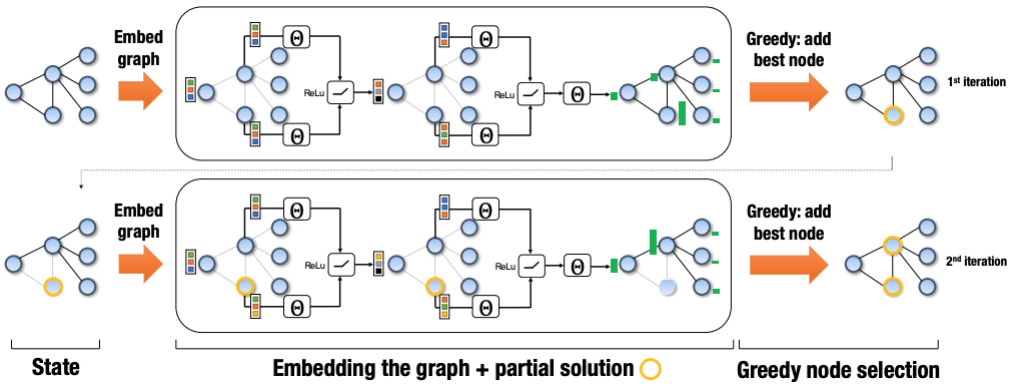

Figure 4: S2V-DQN applied to an instance of MVC (Dai et al., 2017)

It is efficient since it can be executed in polynomial time. Dai *et al.*(2017) also demonstrated the learned model's ability to generalize to much larger problem instances than it was trained on. Their model generalized well even to instances of 1200 nodes and could produce in 12 second solutions that were sometimes better than what a commercial solver could find in one hour.

3.3 Exploratory Combinatorial Optimization Deep Q-learning (ECO-DQN)

Approaches following S2V-DQN's framework incrementally construct solutions one element at a time, which reduce the problem to predict the value of adding one vertex to the current solution subset. However, due to the inherent complexity of different combinatorial problems, learning a policy that directly produces a single optimal solution is often not practical. As vertices can only be added to the solution set, policies derived from the learned Q-values, such as a typical greedy policy, will likely be sub-optimal.

Following S2V-DQN, Barrett *et al.*(2020) proposed an approach of exploratory combinatorial optimization (ECO-DQN), which seeks to continuously improve the solution by learning to explore at test time. Focusing on MAXCUT, each vertex $v \in V$ is represented with an n-dimensional embedding $\mu_v^{(t)}$ similarly like in S2C-DQN. The embeddings are repeatedly updated based on neighborhood vertices :

$$m_v^{(t+1)} \leftarrow M_t\big(\mu_v^{(t)}, \{\mu_u^{(t)}\}_{u \in \mathcal{N}(v)}, \{w(u, v)\}_{u \in \mathcal{N}(v)}\big)$$
$$\mu_v^{(t+1)} \leftarrow U_t\big(\mu_v^{(t)}, m_v^{(t+1)}\big)$$

where M_t and U_t are message function and update function. After T iterations, some readout function will produce a prediction, which is the set of Q-values of actions corresponding to "flipping" each vertex, i.e. adding or removing it from the current partial solution, i.e. $\{Q_v\}_{v \in V} = R\big(\{\mu_u^{(T)}\}_{u \in V}\big)$. The Q-value of the modified vertex is continually re-evaluated among the episode's history, to achieve this with more freedom for improved performance, Barrett *et al.*(2020) shaped the reward by setting the the reward at state $s_t \in S$ to be $R(s_t) = \max\big(C(s_t) - C(s^*), 0\big)|V|$, where $s^* \in S$ is the state corresponding to the highest cut value previously seen within the episode and the reward is normalized by $|V|$, to mitigate the impact of different reward scales across different graph sizes. Barrett *et al.*(2020) used 7 observations to calculate Q-value for each flipping vertex, which are : vertex state (if whether v is in the current partial solution set S), immediate cut change if vertex

state is changed, steps since the vertex state was last changed, difference of current cut-value from the best observed, distance of current partial solution set from the best observed, number of available actions that immediately increase the cut-value, steps remaining in the episode. By comparing ECO-DQN to S2V-DQN, Barrett *et al.*(2020) demonstrate that their approach improves the state-of-the-art for applying RL to MAXCUT. And it can start from from any arbitrary configuration, thus can be combined with other search methods to further improve performance. Moreover, ECO-DQN also generalizes well to graph from unseen distribution.

4 Discussion

The above work all showed their significance, here are some possible further direction from my point of view.

For Pointer Network, it is designed for 2-dimensional problems, how can it be generalized to larger dimensions? And the Pointer Network is sensitive to the order of the input, Bello *et al.* (2016) took the obvious order flattening them all from from input to output, but what happened if we rule out that order? Will it give a better or a worse result? For S2V-DQN, a helper function h is assumed to be given to describe the graph structure and aid the neural network find better solutions, which is human-designed and problem-specific, which is what we would like to avoid. And S2V-DQN solves the decision versions of combinatorial problems while in the real world, they are often encountered as budget constrained versions.

And the real-world task is to train large deep learning models where model parallelism is important. But getting good performance given multiple computing devices is non-trivial and non-obvious. How to make choices of these devices? Can we actually automate this procedure again? A model could have multiple layers, multiple attention layers and multiple output layers as well as in some point it could have Softmax. It can be put on a single device if it is not big, but if it is, we have to put is on multiple devices, for example :

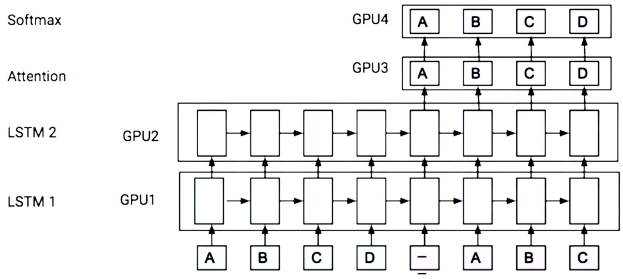

Figure 5: A Task on Multiple Devices (Bengio et al., 2017)

But this way of assigning devices may not be optimal. Is there any better way? This could be considered as a problem in reinforcement learning for higher performance models.

Overall, finding structure in problems with vast search spaces is quite practical direction for RL. Many algorithms have been proposed to tackle games and control problems, but these breakthrough in combinatorial problems via RL showed the power of RL can also be leveraged to graph problems to better satisfy real-world needs.

References

[1] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *CoRR*, abs/1611.09940, 2016. URL http://arxiv.org/abs/1611.09940.

[2] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in neural information processing systems*, pages 2692–2700, 2015.

[3] Thomas D Barrett, William R Clements, Jakob N Foerster, and Alex I Lvovsky. Exploratory combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1909.04063*, 2019.

[4] Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, pages 6348–6358, 2017.

[5] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[6] Wouter Kool, Herke Van Hoof, and Max Welling. Attention, learn to solve routing problems! *arXiv preprint arXiv:1803.08475*, 2018.

[7] Akash Mittal, Anuj Dhawan, Sahil Manchanda, Sourav Medya, Sayan Ranu, and Ambuj Singh. Learning heuristics over large graphs via deep reinforcement learning. *arXiv preprint arXiv:1903.03332*, 2019.

[8] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[9] David P Williamson and David B Shmoys. *The design of approximation algorithms*. Cambridge university press, 2011.

[10] David Applegate, Robert Bixby, Vašek Chvátal, and William Cook. Implementing the dantzig-fulkerson-johnson algorithm for large traveling salesman problems. *Mathematical programming*, 97(1-2):91–153, 2003.

[11] David L Applegate, Robert E Bixby, Vasek Chvatal, and William J Cook. *The traveling salesman problem: a computational study*. Princeton university press, 2006.

[12] Laura I Burke. Neural methods for the traveling salesman problem: insights from operations research. *Neural Networks*, 7(4):681–690, 1994.

[13] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zhang. Tensorflow: A system for large-scale machine learning. *CoRR*, abs/1605.08695, 2016. URL http://arxiv.org/abs/1605.08695.

[14] S. V. B. Aiyer, M. Niranjan, and F. Fallside. A theoretical investigation into the performance of the hopfield model. *IEEE Transactions on Neural Networks*, 1(2):204–215, 1990.

[15] X. Xu, Z. Jia, J. Ma, and J. Wang. A self-organizing map algorithm for the traveling salesman problem. In *2008 Fourth International Conference on Natural Computation*, volume 3, pages 431–435, 2008.

[16] George Dantzig, Ray Fulkerson, and Selmer Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the operations research society of America*, 2(4):393–410, 1954.

[17] Favio Favata and Richard Walker. A study of the application of kohonen-type neural networks to the travelling salesman problem. *Biological Cybernetics*, 64(6):463–468, 1991.

[18] Shen Lin and Brian W Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations research*, 21(2):498–516, 1973.

[19] Manfred Padberg and Giovanni Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM review*, 33(1):60–100, 1991.

[20] Christos H. Papadimitriou and PAPADIMITRIOU CH. The euclidean traveling salesman problem is np-complete. 1977.