Reinforcement Learning to solve VRP

João Castanheira 55052

1 Introduction

The last decades have seen an increasing usage of optimization packages, based on Operations Research and Mathematical Programming techniques, for the effective management of the provision of goods and services in distribution systems [3]. A large number of real-world applications have shown that the use of computerized procedures for the distribution process planning produces substantial savings (generally from 5% to 20%) in the global transportation costs [3]. The success of the usage of operations research techniques is due to the development of computer systems, and to the increasing integration of information systems into the productive and commercial processes.

1.1 Vehicle Routing Problem

The Vehicle Routing Problem (VRP) is a type of routing problem where, in its simplest form, a fleet of vehicles is responsible for delivering items to multiple customer nodes. The objective is to optimize a set of routes, all beginning and ending at a given node, called the depot, in order to attain the maximum possible reward, which is often the negative of the total vehicle traversed distance [1].

The road network is generally described through a graph, whose arcs represent the road sections and whose vertices correspond to the road junctions and the depot and customer locations [3]. Each arch is associated with a cost, which generally represents its length. Classically, approaches to solve this problem can be divided into exact methods, that guarantee finding optimal solutions, and heuristic methods, that off optimality for computational cost [5].

The VRP problem has a lot of variants, for instance:

- Capacitated VRP(CVRP), in which vehicles with limited carrying capacity need to pick up or deliver items at various locations, where each location has a deterministic demand;
- VRP with Time Windows (VRPTW) in which capacity constraints are imposed and each customer i is associated with a time interval $[a_i, b_i]$, called the time window [3];
- VRP with Pickup and Delivery where each customer i is associated with two quantities d_i and p_i representing the demand of homogeneous commodities to be delivered and picked up at customer i, respectively [3].

In this project, just the CVRP variant will be studied.

1.2 Solving VRP

Approaches to solve the VRP can be divided into exact methods, that guarantee to find optimal solutions, and heuristics, that trade optimality for computational cost.

One of the most used heuristic tool is Google Operations Research Tools (OR-Tools) [6]. OR-Tools is an open source software suitable for solving optimization problems, such as routing problems, constraint programming, flows problems, integer programming, and so on [2]. The computational time is usually very fast compared to other techniques, and its solutions are usually near-optimal when compared with exact methods [2].

More recently, reinforcement learning (RL) has been proposed as a technique to solve optimization problems.

1.3 Background

The idea to use reinforcement learning to learn heuristis was proposed by Bello et. al (2016). In their paper, they focus on traveling salesman problem (TSP), which is a specific type of routing problem. In this problem, given a set of nodes and the distances between each pair of nodes, the objective is to find the shortest possible route that visits each node and returns to the depot. Like VRP, TSP is a NP-hard problem in combinatorial optimization, and the difference between them is that in TSP there is just a single route, and in VRP there are multiple routes, each of them assigned to a specific vehicle.

Bello et. al (2016) state that TSP solver relies on handcrafted heuristics that guide their search procedures to find competitive tours efficiently. Even though these heuristics work well on TSP, once the problem formulation changes slightly, they need to be revised. Using classical heuristics, the entire distance matrix must be recalculated and the system must be re-optimized from scratch, which is often impractical, especially if the problem size is large [1]. In contrast, machine learning methods have the potential to be applicable across many optimization tasks by automatically discovering their own heuristics based on the training data, thus requiring less hand-engineering than solvers optimized for one task only [4].

Vinyals et al. (2015) introduce the Pointer Network (PN) as a model that uses attention to output a permutation of the input and train this model offline to solve the (Euclidean) TSP, supervised by example solutions [4]. The PN architecture, introduced by [7], comprises two recurrent neural network (RNN) modules, encoder and decoder, both of which consist of Long Short-Term Memory (LSTM) cells [4].

Based on these approaches, Bello et. al (2017) proposed Neural Combinatorial Optimization, a framework to tackle combinatorial optimization problems using RL and neural networks. Bello et. al (2016) use RL to train a PN, that given a set of node coordinates, predicts a distribution over different city permutations. They used negative tour length as the reward signal and optimized the parameters of the PN using a policy gradient method.

Although the framework proposed by Bello et al. (2016) works well on problems such as the knapsack problem and TSP, it is not applicable to more complicated combinatorial optimization problems in which the system representation varies over time, such as VRP [1]. Nazari et. al (2018) generalized the framework proposed by [4] in order to include a wider range of combinatorial optimization problems, such as the VRP.

Kool et. al (2018) proposed a framework that uses RL and a graph attention network to tackle many combinatorial optimization problems: TSP; Prize Collecting TSP (PCTSP); Stochastic PCTSP; CVRP; Split Delivery VRP and the Orienteering Problem. They proposed to use a model based on attention and train it using REINFORCE algorithm with a simple greedy rollout baseline.

2 Objectives

In the project proposal phase, I proposed to compare the results of CVRP by solving it using three different approaches: the OR-Tools; the deep RL framework proposed by Nazari et. al (2018), and the approach proposed by Kool et. al (2018). During the project development, I found that it wouldn't be possible to reproduce the solution proposed by Nazari et. al (2018) because I don't have access to the required computational resources to train the model.

Otherwise, the framework proposed by Kool et. al (2018) contains pretrained models for problems of size 10, 20, 50, and 100. The pretrained models could be found in github paper's repository [1]. Since I don't have access to the required computational resources to train these deep RL models, for this study I will use just the pretrained models provided by Kool et. al (2018), and compare the results obtained with OR-Tools results.

3 Notations

Kool et. al (2018) define a problem instance s, as a graph with n nodes, where node $i \in \{1, \ldots, n\}$ is represented by features $\mathbf{x}_i$, with $\mathbf{x}_i$ being the coordinates of node i. Each node i has a demand d_i to be satisfied. A vehicle, that always starts at and ends at the depot, can serve a set of customers as long as the total customer demand does not exceed the capacity of the vehicle D [9].

[1] https://github.com/wouterkool/attention-learn-to-route

4 Attention Model

Kool et. al (2018) defined their model as a Graph Attention Network (Velickovic et al., 2018), and as an attention based encoder-decoder. The encoder produces embeddings of all input nodes [5], and the decoder produces the sequence $\boldsymbol{\pi}$ of input nodes.

Attention mechanisms have become almost a standard in many sequence-based tasks [8]. Based on the recent work in attention mechanisms, Velickovic et. al (2018) introduce an attention-based architecture to perform node classification of graph-structured data. The idea is to compute the hidden representation of each node in the graph, by attending over its neighbors, following a self-attention strategy [8].

Kool et. al (2018) define a solution $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_n)$ as a permutation of the nodes, so $\pi_t \in \{1, \ldots n\}$. The attention based encoder-decoder model defines a stochastic policy $p(\pi|s)$ for selection a solution π given a problem instance s. It is factorized and parameterized by θ as

$$p_\theta(\pi|s) = \prod_{t=1}^{n} p_{\boldsymbol{\theta}}\left(\pi_t | s, \pi_{1:t-1}\right)$$

4.1 Encoder

From the input features, the encoder computes initial d_{h} -dimensional node embeddings $\mathbf{h}_i^{(0)}$ ($d_{\mathrm{h}} = 128$) through a learned linear projection with parameters W^{x} and b^{x} $\mathbf{h}_i^{(0)} = W^{\mathrm{x}}\mathbf{x}_i + \mathbf{b}^{\mathrm{x}}$. The embeddings are updated using N attention layers, each consisting of two sublayers. With $h_i^{(\ell)}$ being the node produced by layer $\ell \in \{1, .., N\}$ [5].

In order to allow the attention model to distinguish the depot node from the regular nodes, Kool et. al (2018) use separate parameters W_0^{x} and $\mathrm{b}_0^{\mathrm{x}}$ to compute the initial embedding $\mathrm{h}_0^{(0)}$ of the depot node. Additionally, Kool et. al (2018) provides the normalized demand d_i as input feature (and adjust the size of parameter W^{x} accordingly).

$$\mathbf{h}_i^{(0)} = \begin{cases} W_0^{\mathrm{x}} \mathrm{x}_i + \mathrm{b}_0^{\mathrm{x}} & i = 0 \\ W^{\mathrm{x}} \left[\mathrm{x}_i, \hat{d}_i\right] + \mathrm{b}^{\mathrm{x}} & i = 1, \ldots, n \end{cases}$$

Attention layer Each attention layer consists of two sublayers: a multi-head attention layer that executes message passing between nodes and a node-wise fully connected feed forward layer [5]. The multi-head attention mechanism can be seen as a message passing algorithm that allows nodes to communicate relevant information over different channels, such that the node embeddings from the encoder can learn to include valuable information about the node *in the context of the graph* [5].

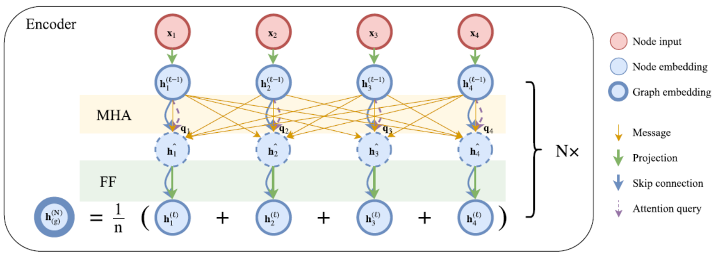


Fig. 1: Attention based encoder. Input nodes are embedded and processed by N sequential layers, each consisting of a multi-head attention (MHA) and node-wise feed-forward (FF) sublayer. The graph embedding is computed as the mean of node embeddings [5]. Image from [5]

4.2 Decoder

The decoder receives as input the encoder embeddings, both the node embeddings $\mathrm{h}_i^{(N)}$ and the graph embedding $\overline{\mathrm{h}}^{(N)}$, and a problem specific mask and context. The decoder observes a mask to know which nodes have been visited [5]. The context for the decoder for the VRP at time t is the current/last location π_{t-1} and the remaining capacity D_t [5]:

$$\mathbf{h}_{(c)}^{(N)} = \begin{cases} \left[\overline{\mathbf{h}}^{(N)}, \mathbf{h}_{\pi_{t-1}}^{(N)}, \hat{D}_t\right] & t > 1 \\ \left[\overline{\mathbf{h}}^{(N)}, \mathbf{h}_0^{(N)}, \hat{D}_t\right] & t = 1 \end{cases}$$

Decoding happens sequentially, and at timestep $t \in \{1, \ldots n\}$, the decoder outputs the node π_t based on the embeddings from the encoder and the outputs $\pi_{t'}$ generated at time $t' < t$. The final probabilities are computed using a single-head attention mechanism [5].

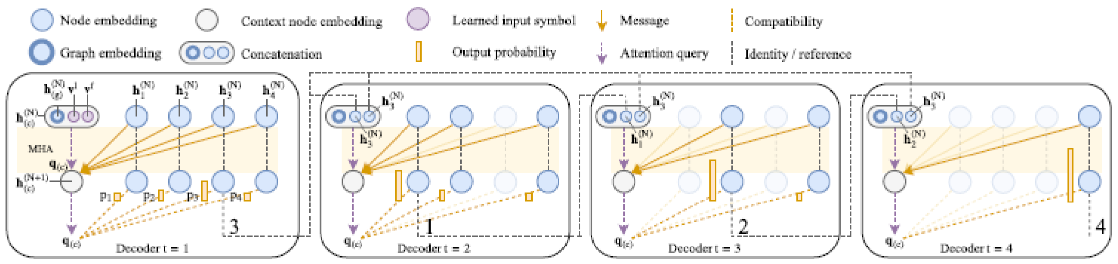


Fig. 2: Attention based decoder. The decoder takes as input the node and graph embeddings. Nodes that cannot be visited (since they are already visited) are masked. The example shows how a tour $\pi = (3, 1, 2, 4)$ is constructed [5]. Image from [5].

5 REINFORCE with greedy rollout baseline

In the previous section the attention model was defined, from which given an instance s defines a probability distribution $p_{\boldsymbol{\theta}}(\pi|s)$. In order to train this model, Kool et. al (2018) uses Monte Carlo Policy Gradient algorithm (REINFORCE), which updates the attention model parameters θ by stochastic gradient descent, using the policy gradient theorem.

Kool et. al (2018) defines the loss function as the expectation of the cost $L(\pi)$ (tour length): $\mathcal{L}(\boldsymbol{\theta}|s) = \mathbb{E}_{p_{\boldsymbol{\theta}}(\boldsymbol{\pi}|s)}[L(\pi)]$. Then, they optimize L by gradient descent using the REINFORCE (Williams, 1992) gradient estimator with baseline $b(s)$:

$$\nabla\mathcal{L}(\boldsymbol{\theta}|s) = \mathbb{E}_{p_{\boldsymbol{\theta}}(\boldsymbol{\pi}|s)}\left[(L(\boldsymbol{\pi}) - b(s))\nabla \log p_{\boldsymbol{\theta}}(\pi|s)\right]$$

The baseline $b(s)$ is a rollout algorithm. Rollout algorithms are decision-time planning algorithms based on Monte Carlo control applied to simulated trajectories [11]. They estimate action values for a given policy by averaging the returns of many simulated trajectories that start with each possible action and then follow the given policy. The goal of a rollout algorithm is not to estimate a complete optimal action-value function, q_pi for a given policy $\boldsymbol{\pi}$. Instead, they produce Monte Carlo estimates of action value only for each current state for a given policy usually called the rollout policy [11]. The aim of a rollout algorithm is to improve upon the rollout policy, not to find an optimal policy [11].

With the greedy rollout as baseline $b(s)$, the function $L(\pi) - b(s)$ is negative if the sampled solution $\boldsymbol{\pi}$ is better than the greedy rollout, causing actions to be reinforced, and vice versa. This way the model is trained to improve over its (greedy) self [5]. The pseudo code could be found in the Appendix A.

6 Results

In order to compare the OR-Tools with the RL framework, i would follow the approach defined in [5], and I will use the RL pretrained models. I generated random instance problems for $n = 10$, 20, 50 and 100. The depot location as well as n node location are sampled uniformly at random in the unit square. The demands are normalized according to D, and defined as $d_i = \frac{d_i}{D^n}$, with $D^{10} = 20$, $D^{20} = 30$, $D^{50} = 40$ and $D^{100} = 50$. An example of a random generated problem, and the solution provided by both the RL and OR-Tools, is in Appendix B.

For each of the problem sizes defined above, I generated 100 random problem instances as defined above, and then I count the number of times the RL wins to the OR-Tools, i.e when the solution length provided by the RL framework is smaller than the solution length provided by the OR-Tools.

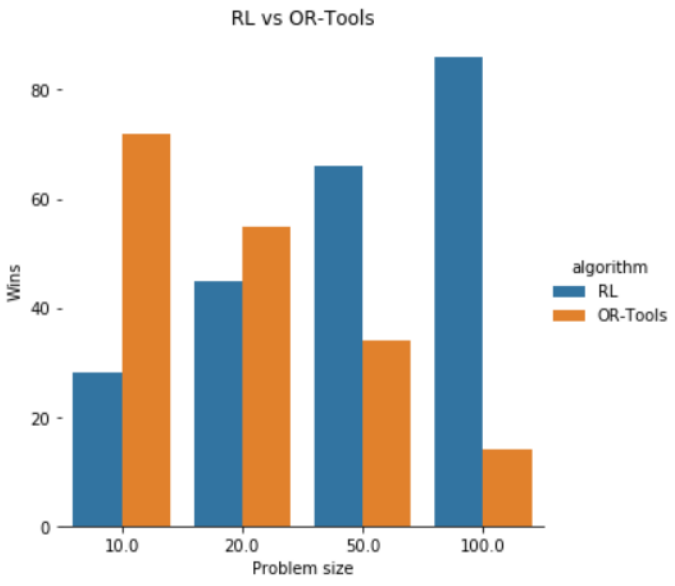


Fig. 3: RL vs OR-Tools

We can see that as the problem size increases the RL outperforms the OR-Tools solutions.

Another interesting experiment made was testing if a model trained with 100 nodes could still get good results by applying it to smaller or higher problem sizes. To do this test, I tested the pretained model with 100 nodes with problem sizes of 70, 80, 90, 110, 120, and 130 nodes. For each problem size, 100 random problem instances were sampled and for each one was applied to the RL algorithm and the OR-Tools.

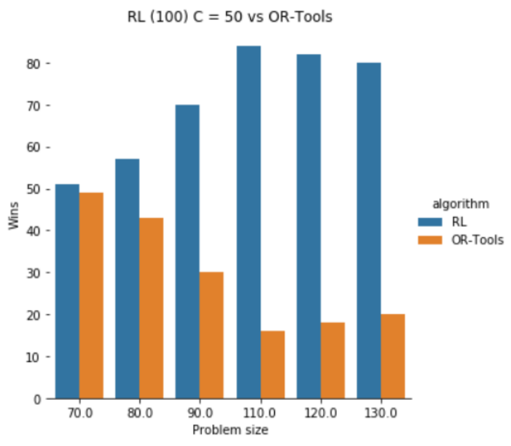


Fig. 4: RL vs OR-Tools

We can see that the model generalizes well. A model trained with 100 nodes still gets better results, in general, with problems of size 130 when compared to the OR-Tools. As the problem size decreases, the OR-Tools start to get closer results to the RL. This was expected because we saw above that the OR-Tools outperforms the RL framework in smaller problem sizes.

7 Final comments

In their study Kool et. al (2018) introduced a model and training method which contributes to improve the results on learned heuristics for routing problems. They have shown that the idea to use RL for optimization problems is very promising. They presented a framework that could be a starting point for learning heuristics for other combinatorial optimization problems defined on graphs.

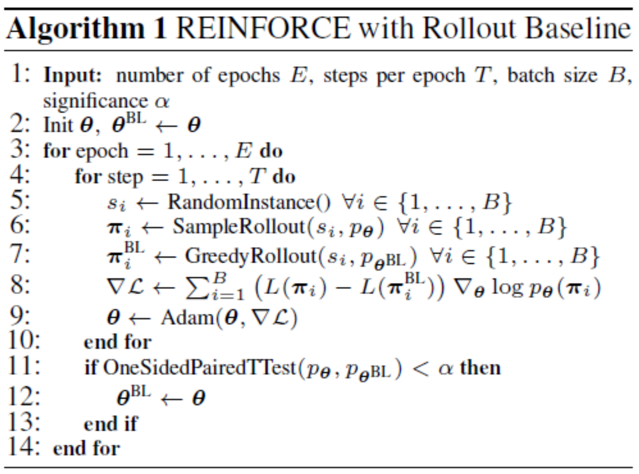
Their proposed model outperforms the OR-Tools for larger problem sizes. In order to improve these RL frameworks, a lot of research should be done to include constraints to the problem, such as time window constraints or pickup and delivery constraints. Research should be done to scale this approach to larger problem sizes. In order to adopt these RL solutions to real-world problems, I think that more research should be done to include the distances between the nodes by road.

References

1. Nazari, M., Oroojlooy, A., Snyder, L.V., & Takác, M. (2018). Deep Reinforcement Learning for Solving the Vehicle Routing Problem. ArXiv, abs/1802.04240.
2. Ibrahim, Abdullahi & Abdulaziz, Rabiat & Ishaya, Jeremiah. (2019). CAPACITATED VEHICLE ROUTING PROBLEM. International Journal of Research - GRANTHAALAYAH. 7. 310 - 327. 10.5281/zenodo.2636820.
3. Toth, P., & Vigo, D. (Eds.). (2002). The vehicle routing problem. Society for Industrial and Applied Mathematics.
4. Bello, I., Pham, H., Le, Q. V., Norouzi, M., & Bengio, S. (2016). Neural combinatorial optimization with reinforcement learning. arXiv preprint arXiv:1611.09940.
5. Kool, W., Van Hoof, H., & Welling, M. (2018). Attention, learn to solve routing problems!. arXiv preprint arXiv:1803.08475.
6. OR-Tools 7.2. Laurent Perron and Vincent Furnon. https://developers.google.com/optimization/.
7. Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In Advances in Neural Information Processing Systems, pp. 2692–2700, 2015.
8. Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. In International Conference on Learning Representations (ICLR), 2018.
9. Hao Lu, Xingwen Zhang,& Shuang Yang (2020). A Learning-based Iterative Method for Solving Vehicle Routing Problems. In International Conference on Learning Representations.
10. Ronald JWilliams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. Machine learning, 8(3-4):229–256, 1992.
11. Richard S. Sutton and Andrew G. Barto. 2018. Reinforcement Learning: An Introduction. A Bradford Book, Cambridge, MA, USA.
12. David Silver, University College London Reinforcement Learning course. https://www.davidsilver.uk/teaching/.

8 Appendix

8.1 A - REINFORCE pseudo code for attention



Algorithm 1 REINFORCE with Rollout Baseline

```
1:  Input: number of epochs $E$, steps per epoch $T$, batch size $B$,
    significance $\alpha$
2:  Init $\boldsymbol{\theta}$, $\boldsymbol{\theta}^{\mathrm{BL}} \leftarrow \boldsymbol{\theta}$
3:  for epoch $= 1, \ldots, E$ do
4:      for step $= 1, \ldots, T$ do
5:          $s_i \leftarrow$ RandomInstance() $\forall i \in \{1, \ldots, B\}$
6:          $\boldsymbol{\pi}_i \leftarrow$ SampleRollout$(s_i, p_{\boldsymbol{\theta}})$ $\forall i \in \{1, \ldots, B\}$
7:          $\boldsymbol{\pi}_i^{\mathrm{BL}} \leftarrow$ GreedyRollout$(s_i, p_{\boldsymbol{\theta}^{\mathrm{BL}}})$ $\forall i \in \{1, \ldots, B\}$
8:          $\nabla \mathcal{L} \leftarrow \sum_{i=1}^{B} \left( L(\boldsymbol{\pi}_i) - L(\boldsymbol{\pi}_i^{\mathrm{BL}}) \right) \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\boldsymbol{\pi}_i)$
9:          $\boldsymbol{\theta} \leftarrow$ Adam$(\boldsymbol{\theta}, \nabla \mathcal{L})$
10:     end for
11:     if OneSidedPairedTTest$(p_{\boldsymbol{\theta}}, p_{\boldsymbol{\theta}^{\mathrm{BL}}}) < \alpha$ then
12:         $\boldsymbol{\theta}^{\mathrm{BL}} \leftarrow \boldsymbol{\theta}$
13:     end if
14: end for
```

Fig. 5: REINFORCE with Rollout Baseline. Image from [5].

8.2 B - RL vs OR-Tools

In the figure bellow there is the CVRP solution provided by both the RL framework and the OR-Tools. We can see that RL outperforms the OR-Tools, with a total route length of 15.89 vs 16.12 respectively.

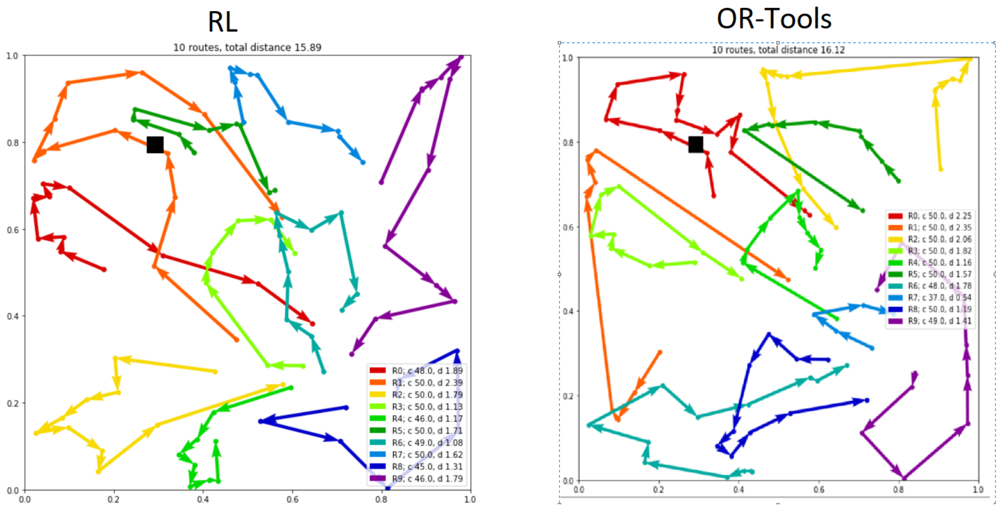


Fig. 6: RL vs OR-Tools for the same problem