NEURAL COMBINATORIAL OPTIMIZATION WITH REINFORCEMENT LEARNING

Irwan Bello[*], **Hieu Pham**[*], **Quoc V. Le, Mohammad Norouzi, Samy Bengio**
Google Brain
{ibello,hyhieu,qvl,mnorouzi,bengio}@google.com

ABSTRACT

We present a framework to tackle combinatorial optimization problems using neural networks and reinforcement learning. We focus on the traveling salesman problem (TSP) and train a recurrent neural network that, given a set of city coordinates, predicts a distribution over different city permutations. Using negative tour length as the reward signal, we optimize the parameters of the recurrent neural network using a policy gradient method. Without much engineering and heuristic designing, Neural Combinatorial Optimization achieves close to optimal results on 2D Euclidean graphs with up to 100 nodes. These results, albeit still quite far from state-of-the-art, give insights into how neural networks can be used as a general tool for tackling combinatorial optimization problems.

1 INTRODUCTION

Combinatorial optimization problems, such as the *Traveling Salesman Problem (TSP)*, are fundamental to computer science. Many solutions to those problems rely on handcrafted heuristics that guide their search procedures to find competitive (and in many cases optimal) solutions efficiently. Each combinatorial optimization problem has its own heuristics that needs to be revised once the problem statement changes slightly.

In contrast, machine learning methods are applicable across many tasks and can discover their own heuristics, thus requiring less hand-engineering than solvers that are optimized for one task only. In this work, We propose Neural Combinatorial Optimization (NCO), a framework to tackle combinatorial optimization problems using reinforcement learning and neural networks. We apply NCO to the 2D Euclidean TSP, a well-studied NP-hard problem with with many proposed algorithms (Applegate et al., 2006; Helsgaun, 2000; Google, 2016) . Our results, while still inferior to the state-of-the-art in many dimensions (such as speed, scale and performance), give insights into how neural networks can be used as a general tool for tackling combinatorial optimization problems, especially those that are difficult to design heuristics for.

2 NEURAL NETWORK ARCHITECTURE

We focus on the 2D Euclidean TSP in this paper. Given an input graph, represented as a sequence of n cities in a two dimensional space $s = \{\mathbf{x}_i\}_{i=1}^n$ where each $\mathbf{x}_i \in \mathbb{R}^2$, we are concerned with finding a permutation of the points π, termed a tour, that visits each city once and has the minimum total length. We define the length of a tour defined by a permutation π as

$$L(\pi \mid s) = \left\|\mathbf{x}_{\pi(n)} - \mathbf{x}_{\pi(1)}\right\|_2 + \sum_{i=1}^{n-1} \left\|\mathbf{x}_{\pi(i)} - \mathbf{x}_{\pi(i+1)}\right\|_2 , \tag{1}$$

where $\|\cdot\|_2$ denotes ℓ_2 norm. We use the chain rule to factorize the probability of a tour as

$$p(\pi \mid s) = \prod_{i=1}^{n} p\left(\pi(i) \mid \pi(< i), s\right) \ , \tag{2}$$

[*]Equal contributions. Work done when the authors were in the Google Brain Residency program.

and then use individual softmax modules to represent each term on the RHS of (2). We employ the pointer network architecture (Vinyals et al., 2015) as our policy model to parameterize $p(\pi \mid s)$.

Our pointer network consists of two RNN modules, both of which use LSTM (Hochreiter & Schmidhuber, 1997). The encoder network reads the input sequence s, one point $\mathbf{x}_i$ at a time, and transforms it into a sequence of latent memory $\{enc_i\}_{i=1}^n$. The decoder network maintains its latent memory $\{dec_i\}_{i=1}^n$, and at each step i, uses a pointing mechanism, described in Appendix 6.2, to produce a distribution over the next city to visit in the tour. Once the next city is selected, it is passed as the input to the next decoder step.

3 Optimization with policy gradients

Vinyals et al. (2015) proposes training a pointer network using a supervised cross-entropy loss function against the outputs produced by a solver. Such method is undesirable for NP-hard problems because (1) the performance of the model is tied to the quality of the supervised labels, and (2) getting high-quality labeled data is expensive and may be infeasible for new problem statements.

In contrast, we believe Reinforcement Learning (RL) provides an appropriate paradigm for training neural networks for combinatorial optimization, especially because these problems have relatively simple reward mechanisms that could be even used at test time. We hence propose to use policy-based RL to optimize our pointer network's parameters, denoted $\boldsymbol{\theta}$. Our training objective is the expected tour length given an input graph s, formally $J(\boldsymbol{\theta} \mid s) = \mathbb{E}_{\pi \sim p_\theta(.|s)}\, L(\pi \mid s)$. The policy gradient of the objective is formulated using REINFORCE algorithm (Williams, 1992):

$$\nabla_\theta J(\theta \mid s) = \mathbb{E}_{\pi \sim p_\theta(.|s)} \Big[\big(L(\pi \mid s) - b(s) \big) \nabla_\theta \log p_\theta(\pi \mid s) \Big] , \tag{3}$$

where $b(s)$ denotes a baseline function that does not depend on π and estimates the expected tour length to reduce the variance of the gradients.

A simple and popular choice of the baseline $b(s)$ is an exponential moving average of the rewards obtained during training. While this choice of baseline proved sufficient to improve upon the Christofides algorithm, we find that using a parametric baseline to estimate the expected tour length $\mathbb{E}_{\pi \sim p_\theta(.|s)} L(\pi \mid s)$ typically improves learning.

We introduce an auxiliary network, called a *critic* and parameterized by θ_v, to learn the expected tour length found by our current policy p_θ given an input sequence s. The architecture of our critic network is an RNN with LSTM, where predictions were made based on the final state. The critic is trained with stochastic gradient descent on a mean squared error objective

$$\mathcal{L}(\theta_v) = \frac{1}{B} \sum_{i=1}^{B} \left\| \, b_{\theta_v}(s_i) - L(\pi_i \mid s_i) \, \right\|_2^2 . \tag{4}$$

4 Experiments

We consider three benchmark tasks, Euclidean TSP{20,50,100}, for each we generate a test set of $1,000$ graphs. Points are drawn uniformly at random in the unit square $[0,1]^2$. Additional details about our experiments can be found in Appendix 6.1. We employ the following strageties

RL pretraining: For the RL experiments, we generate training mini-batches of inputs on the fly. We find that clipping the logits to $[-10, 10]$ with a $\tanh(\cdot)$ activation function, as in Appendix 6.3, yields marginal gains. At test time, we simply take the city with highest probability at each decoding step. We also consider ensembling 16 models and taking their best results. We refer to those approaches as *RL pretraining-greedy* and *RL pretraining-greedy@16*.

Searching at inference: As evaluating a tour length is inexpensive, our TSP agent can easily simulate a search procedure at inference time by considering multiple candidate solutions per graph and selecting the best. We employ the following strategies:

- **RL pretraining-Sampling:** For each test instance, we sample $1,280,000$ candidate solutions from a pretrained model and keep track of the shortest tour. A grid search over the temperature hyperparameter found respective temperatures of 2.0, 2.2 and 1.5 to yield the best results for TSP{20,50,100} respectively.

- **RL pretraining-Active Search:** For each test instance, we initialize the model parameters from a pretrained RL model and update them for up to $10,000$ steps, each with a batch size of 128, therefore seeing a total of $1,280,000$ candidate solutions. We set the learning rate to a hundredth of the initial learning rate the TSP agent was trained on.
- **Active Search:** We allow the model to train much longer to account for the fact that it starts from scratch. For each test graph, we run Active Search for $100,000$ training steps on TSP{20,50} and $200,000$ training steps on TSP100.

We compare our methods against several baselines.

- **Supervised Learning:** We implement and train a pointer network with supervised learning (Vinyals et al., 2015). We find that our supervised learning results are not as good as theirs. We suspect that learning from optimal tours is harder for supervised pointer networks. We thus refer to the results in Vinyals et al. (2015) for TSP{20,50} and report our results on TSP100.
- **Christofides Algorithm:** In polynomial time, the algorithm finds solutions that are guaranteed to be within a $1.5\times$ of optimality.
- **Generic Solvers:** We use the vehicle routing solver from (Google, 2016), which improves over Christofides' solutions with simple local search operators and metaheuristics.
- **State-of-the-art opensource TSP-specific solvers:** We use Concorde (Applegate et al., 2006) and LK-H's local search (Helsgaun, 2000; 2012). While only Concorde provably solves instances to optimality, we empirically find that LK-H also achieves *optimal solutions* on all of our test sets after 50 trials per graph (which is the default parameter setting).

Table 1: Average tour lengths (lower is better). Results marked $^{(\dagger)}$ are from (Vinyals et al., 2015). Concorde's solutions are provably optimal, and LK-H finds the same solutions.

Task	Supervised Learning	RL pretraining				AS	Christo -fides	OR Tools' local search	Concorde/ LK-H
		greedy	greedy@16	sampling	AS				
TSP20	$3.88^{(\dagger)}$	3.89	–	3.82	3.82	3.96	4.30	3.85	3.82
TSP50	$6.09^{(\dagger)}$	5.95	5.80	5.70	5.70	5.87	6.62	5.80	5.68
TSP100	10.81	8.30	7.97	7.88	7.83	8.19	9.18	7.99	7.77

Solution Quality: We report the average tour lengths of our approaches on TSP{20,50,100} in Table 1. We observe that training with RL significantly improves over supervised learning (Vinyals et al., 2015). All our methods comfortably surpass Christofides' heuristic.

Speed: Searching at inference time proves crucial to get closer to optimality but comes at the expense of longer inference times. Fortunately, our search strategies can be stopped early with a small performance tradeoff in terms of the final objective. More details can be found in Table 2 in Appendix 6.5. Many of our methods outperform Google's OR-Tools, including RL pretraining-Greedy@16 which runs similarly fast. However, compared to TSP-specific state-of-the-art solvers, such as Concorde and LK-H, our methods are at least five orders of magnitude worse than our methods while running on CPUs.

5 Conclusion

This paper presents Neural Combinatorial Optimization, a framework to tackle combinatorial optimization with reinforcement learning and neural networks. We focus on the traveling salesman problem (TSP) and present a set of results for each variation of the framework. Experiments demonstrate that Neural Combinatorial Optimization achieves close to optimal results on 2D Euclidean graphs with up to 100 nodes. Our results, while still far from the strongest solvers (especially those which are optimized for one problem), provide an interesting research avenue for using neural networks as a general tool for tackling combinatorial optimization problems.

REFERENCES

David L Applegate, Robert E Bixby, Vasek Chvatal, and William J Cook. Concorde tsp solver, 2006. URL www.math.uwaterloo.ca/tsp/concorde.

Google. Or-tools, google optimization tools, 2016. URL https://developers.google.com/optimization/routing.

Keld Helsgaun. An effective implementation of the Lin-Kernighan traveling salesman. *European Journal of Operational Research*, 126:106–130, 2000.

Keld Helsgaun. LK-H, 2012. URL http://akira.ruc.dk/~keld/research/LKH/.

Sepp Hochreiter and Jurgen Schmidhuber. Long short-term memory. *Neural Computations*, 1997.

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR*, 2014.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems*, pp. 2692–2700, 2015.

Ronald Williams. Simple statistical gradient following algorithms for connectionnist reinforcement learning. In *Machine Learning*, 1992.

6 APPENDIX

6.1 EXPERIMENTAL DETAILS

Across all experiments, we use mini-batches of 128 sequences, LSTM cells with 128 hidden units, and embed the two coordinates of each point in a 128-dimensional space. We train our models with the Adam optimizer (Kingma & Ba, 2014) and use an initial learning rate of 10^{-3} for TSP{20,50} and 10^{-4} for TSP100 that we decay every 5000 steps by a factor of 0.96. We initialize our parameters uniformly at random within $[-0.08, 0.08]$ and clip the $L2$ norm of our gradients to 1.0. We use up to one attention glimpse. When searching, the mini-batches either consist of replications of the test sequence or its permutations. The baseline decay is set to $\alpha = 0.99$ in Active Search.

6.2 POINTING AND ATTENDING

Pointing mechanism: Its computations are parameterized by two attention matrices $W_{ref}, W_q \in \mathbb{R}^{d\times d}$ and an attention vector $v \in \mathbb{R}^d$ as follows:

$$u_i = \begin{cases} v^\top \cdot \tanh\left(W_{ref} \cdot r_i + W_q \cdot q\right) & \text{if } i \neq \pi(j), \forall j < i \\ -\infty & \text{otherwise} \end{cases} \tag{5}$$

$$A(ref, q; W_{ref}, W_q, v) \stackrel{\text{def}}{=} softmax(u) \tag{6}$$

Our pointer network, at decoder step j, then assigns the probability of visiting the next point $\pi(j)$ of the tour as follows:

$$p(\pi(j)|\pi(<j), s) \stackrel{\text{def}}{=} A(enc_{1:n}, dec_j). \tag{7}$$

Setting the logits of cities that already appeared in the tour to $-\infty$, as shown in Equation 5, ensures that our model only points at cities that have yet to be visited and hence outputs valid TSP tours.

Attending mechanism: Specifically, our glimpse function $G(ref, q)$ takes the same inputs as the attention function A and is parameterized by $W^g_{ref}, W^g_q \in \mathbb{R}^{d\times d}$ and $v^g \in \mathbb{R}^d$. It performs the following computations:

$$p = A(ref, q; W^g_{ref}, W^g_q, v^g) \tag{8}$$

$$G(ref, q; W^g_{ref}, W^g_q, v^g) \stackrel{\text{def}}{=} \sum_{i=1}^{k} r_i p_i. \tag{9}$$

The glimpse function G essentially computes a linear combination of the reference vectors weighted by the attention probabilities. It can also be applied multiple times on the same reference set ref:

$$g_0 \stackrel{\text{def}}{=} q \tag{10}$$

$$g_l \stackrel{\text{def}}{=} G(ref, g_{l-1}; W^g_{ref}, W^g_q, v^g) \tag{11}$$

Finally, the ultimate g_l vector is passed to the attention function $A(ref, g_l; W_{ref}, W_q, v)$ to produce the probabilities of the pointing mechanism. We observed empirically that glimpsing more than once with the same parameters made the model less likely to learn and barely improved the results.

6.3 Improving exploration

Softmax temperature: We modify Equation 6 as follows:

$$A(ref, q, T; W_{ref}, W_q, v) \stackrel{\text{def}}{=} softmax(u/T), \tag{12}$$

where T is a *temperature* hyperparameter set to $T = 1$ during training. When $T > 1$, the distribution represented by $A(ref, q)$ becomes less steep, hence preventing the model from being overconfident.

Logit clipping: We modify Equation 6 as follows:

$$A(ref, q; W_{ref}, W_q, v) \stackrel{\text{def}}{=} softmax(C \tanh(u)), \tag{13}$$

where C is a hyperparameter that controls the range of the logits and hence the entropy of $A(ref, q)$.

6.4 Sample tours

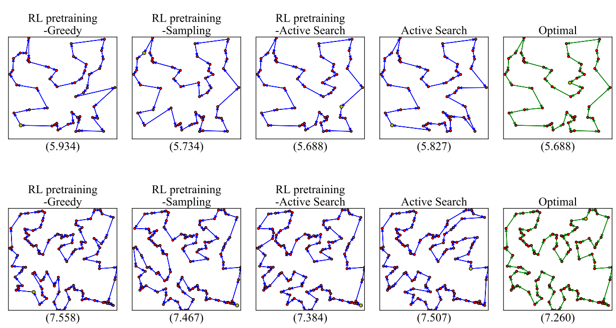

Figure 1: Sample tours. Top: TSP50; Bottom: TSP100.

6.5 OR Tool's Metaheuristics Baselines for TSP

Table 2: Average tour lengths of RL pretraining-Sampling and RL pretraining-Active Search as they sample more solutions. Corresponding running times on a single Tesla K80 GPU are in parantheses.

Task	# Solutions	RL pretraining			Simulated Annealing	Tabu Search	Guided Local Search
		Sampling $T = 1$	Sampling $T = T^*$	Active Search			
TSP50	128	5.80 (3.4s)	5.80 (3.4s)	5.80 (0.5s)	5.81 (0.24s)	5.79 (3.4s)	5.76 (0.5s)
	1,280	5.77 (3.4s)	5.75 (3.4s)	5.76 (5s)	5.81 (4.2s)	5.73 (36s)	5.69 (5s)
	12,800	5.75 (13.8s)	5.73 (13.8s)	5.74 (50s)	5.81 (44s)	5.69 (330s)	5.68 (48s)
	128,000	5.73 (110s)	5.71 (110s)	5.72 (500s)	5.81 (460s)	5.68 (3200s)	5.68 (450s)
	1,280,000	5.72 (1080s)	5.70 (1080s)	5.70 (5000s)	5.81 (3960s)	5.68 (29650s)	5.68 (4530s)
TSP100	128	8.05 (10.3s)	8.09 (10.3s)	8.04 (1.2s)	8.00 (0.67s)	7.99 (15.3s)	7.94 (1.44s)
	1,280	8.00 (10.3s)	8.00 (10.3s)	7.98 (12s)	7.99 (15.7s)	7.93 (255s)	7.84 (18.4s)
	12,800	7.95 (31s)	7.95 (31s)	7.92 (120s)	7.99 (166s)	7.84 (2460s)	7.77 (182s)
	128,000	7.92 (265s)	7.91 (265s)	7.87 (1200s)	7.99 (1650s)	7.79 (22740s)	7.77 (1740s)
	1,280,000	7.89 (2640s)	7.88 (2640s)	7.83 (12000s)	7.99 (15810s)	7.78 (208230s)	7.77 (16150s)