NEURAL COMBINATORIAL OPTIMIZATION

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

ABSTRACT

This paper presents 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 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 network using a policy gradient method. We compare learning the network parameters on a set of training graphs against learning them on individual test graphs. The best results are obtained when the network is first optimized on a training set and then refined on individual test graphs. Without any supervision and with minimal engineering, Neural Combinatorial Optimization achieves close to optimal results on 2D Euclidean graphs with up to 100 nodes.

1 INTRODUCTION

Combinatorial optimization is a fundamental problem in computer science. A canonical example is the *traveling salesman problem (TSP)*, where given a graph, one needs to search the space of permutations to find an optimal sequence of nodes with minimal total edge weights (tour length). The TSP and its variants have myriad applications in planning, manufacturing, genetics, *etc.* (see Applegate et al. (2011) for an overview).

Finding the optimal TSP solution is NP-hard, even in the two-dimensional Euclidean case (Papadimitriou, 1977), where the nodes are 2D points and edge weights are Euclidean distances between pairs of points. In practice, TSP solvers rely on handcrafted heuristics that guide their search procedures to find competitive tours. Even though these heuristics work well on TSP, once the problem statement changes slightly, they need to be revised. In this paper, we advocate the use of machine learning algorithms with minimal hand engineering to address the TSP and its variants, and more generally combinatorial optimization. The end goal is to develop a generic framework which is applicable across many optimization tasks by automatically discovering new heuristics based on the training data.

While most successful machine learning techniques fall into the family of supervised learning, where a mapping from training inputs to outputs is learned, supervised learning is not applicable to most combinatorial optimization problems because one does not have access to optimal labels. However, one can compare the quality of a set of solutions using a verifier, and provide some reward feedbacks to a learning algorithm. Hence, we follow the reinforcement learning (RL) paradigm to tackle combinatorial optimization. We empirically demonstrate that, even when using optimal solutions as labeled data to optimize a supervised mapping, the generalization is rather poor compared to an RL agent that explores different tours and observes their corresponding rewards.

We propose Neural Combinatorial Optimization, a framework to tackle combinatorial optimization problems using reinforcement learning and neural networks. We consider two approaches based on policy gradients (Williams, 1992). The first approach, called *RL pretraining*, uses a training set to optimize a recurrent neural network (RNN) that parameterizes a stochastic policy over solutions, using the expected reward as the objective. At test time, the policy is fixed, and one performs inference by greedy decoding or sampling. The second approach, called *active search*, involves no pretraining. It starts from a random policy and iteratively optimizes the RNN parameters on a single test instance, again using the expected reward objective, while keeping track of the best solution

*Equal contributions. Members of the Google Brain Residency program (g.co/brainresidency).

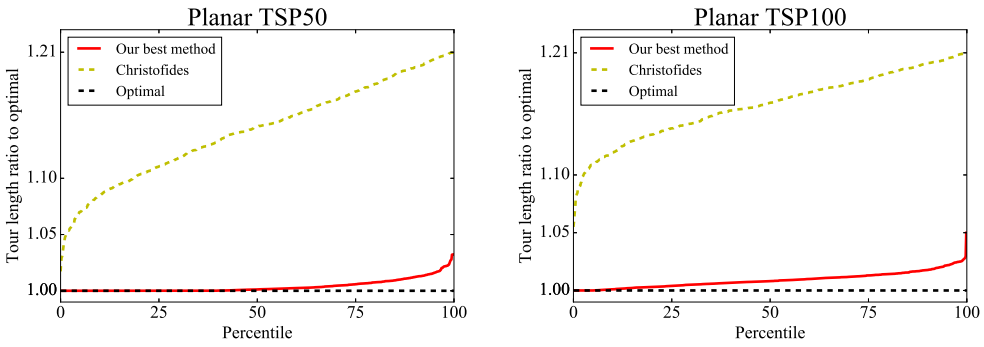

Figure 1: Tour length ratios of our two methods against optimality. The first method is (Christofides, 1976) and the second method is RL pretraining followed by active search.

sampled during the search. We find that combining RL pretraining and active search works best in practice.

On 2D Euclidean graphs with up to 100 nodes, without supervision, Neural Combinatorial Optimization significantly outperforms the supervised learning approach to the TSP (Vinyals et al., 2015b) and obtains close to optimal results (see Figure 1). Furthermore, we empirically demonstrate that the same framework can solve to optimality the *KnapSack problem* (Kellerer et al., 2004), another NP-hard problem, on instances with up to 200 items. These surprising results shed light on 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 Previous Work

The Traveling Salesman Problem is a well studied combinatorial optimization problem. Many exact or approximate algorithms have been proposed for both Euclidean and non-Euclidean graphs. Christofides (1976) propose a heuristic algorithm involves computing a minimum-spanning tree and a minimum-weight perfect matching. The algorithm has polynomial running time and returns solutions that are guaranteed to be within a factor of $1.5\times$ to optimality in the metric instance of the TSP. For the 2D Euclidean TSP, Arora (1998) proposed an $O\left(n(\log n)^{O(c)}\right)$ algorithm with a guaranteed $O(1+1/c)$ approximation ratio. This method recursively partitions the plane into rectangular regions and then employs dynamic programming to find a tour that crosses their boundaries no more than $O(c)$ times.

The best known exact dynamic programming algorithm for TSP has a complexity of $\Theta(2^n n^2)$, making it infeasible to scale up to large instances, say with 40 points. Nevertheless, state of the art TSP solvers, thanks to carefully handcrafted heuristics that describe how to navigate the space of feasible solutions in an efficient manner, can *provably solve to optimality* symmetric TSP instances with thousands of nodes. Concorde (Applegate et al., 2006), widely accepted as one of the best exact TSP solvers, makes use of cutting plane algorithms (Dantzig et al., 1954; Padberg & Rinaldi, 1990; Applegate et al., 2003), iteratively solving linear programming relaxations of the TSP, in conjunction with a branch-and-bound approach that prunes parts of the search space that provably will not contain an optimal solution.

More generic solvers, such as vehicle routing problem solvers (Google, 2016) that tackle a superset of the TSP, rely on a combination of local search algorithms and metaheuristics. Local search algorithms apply a specified set of local move operators on candidate solutions, based on hand-engineered heuristics such as the Lin–Kernighan heuristic (S. Lin, 1973), to navigate from solution to solution in the search space. A metaheuristic is then applied to propose uphill moves and escape local optima. A popular choice of metaheuristic for the TSP and its variants is guided local search (Voudouris & Tsang, 1999), which moves out of a local minimum by penalizing particular solution features that it considers should not occur in a good solution.

The application of neural networks to combinatorial optimization has a distinguished history, where the majority of research focuses on the Traveling Salesman Problem (Smith, 1999). One of the earliest proposals is the use of Hopfield networks (Hopfield & Tank, 1985). The authors modify the network's energy function to make it equivalent to the objective of TSP and then use Lagrange multipliers to penalize the violations of the problem's constraints. A limitation of this approach is that it is sensitive to hyperparameters and parameter initialization as analyzed by (Wilson & Pawley, 1988). Overcoming this limitation is central to the subsequent work in the field, especially by (Aiyer et al., 1990; Gee, 1993).

Parallel to the development of Hopfield networks is the work on using deformable template models to solve TSP. Perhaps most prominent is the invention of Elastic Nets as a means to solve TSP (Durbin, 1987), and application of Self Organizing Map to TSP (Fort, 1988; Angeniol et al., 1988; Kohonen, 1990). Addressing the limitations of deformable template models is central to the following work in this area (Burke, 1994; Favata & Walker, 1991; Vakhutinsky & Golden, 1995).

Even though these neural networks have many appealing properties, they are still limited as research work. When being carefully benchmarked, they have not yielded satisfying results compared to algorithmic methods (Sarwar & Bhatti, 2012; La Maire & Mladenov, 2012). Perhaps due to the negative results, this research direction is largely overlooked since the turn of the century.

Thanks to advances in sequence-to-sequence learning (Sutskever et al., 2014) with attention (Bahdanau et al., 2015), neural networks are again the subject of study for TSP. In particular, they are revisited in the work of Pointer Networks (Vinyals et al., 2015b), where a recurrent network with non-parametric softmaxes is trained in a supervised manner to predict the sequence of visited cities. Despite architecural improvements, their models were trained using supervised signals given by an approximate solver.

3 Neural Network Architecture for TSP

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 aim to learn the parameters of a stochastic policy $p(\pi \mid s)$ that given an input set of points s, assigns high probabilities to short tours and low probabilities to long tours. Our neural network architecture uses the chain rule to factorize the probability of a tour as

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

and then uses individual softmax modules to represent each term on the RHS of (2).

We are inspired by previous work (Sutskever et al., 2014) that makes use of the same factorization based on the chain rule to address sequence to sequence problems like machine translation. One can use a vanilla sequence to sequence model to address the TSP where the output vocabulary is $\{1, 2, \ldots, n\}$. However, there are two major issues with this approach: (1) networks trained in this fashion cannot generalize to inputs with more than n cities. (2) one needs to have access to ground-truth output permutations to optimize the parameters with conditional log-likelihood. We address both isssues in this paper.

For generalization beyond a pre-specified graph size, we follow the approach of (Vinyals et al., 2015b), which makes use of a set of non-parameteric softmax modules, resembling the attention mechanism from (Bahdanau et al., 2015). This approach, named *pointer network*, allows the model to effectively point to a specific position in the input sequence rather than predicting an index value from a fixed-size vocabulary. We employ the pointer network architecture, depicted in Figure 2, as our policy model to parameterize $p(\pi \mid s)$.

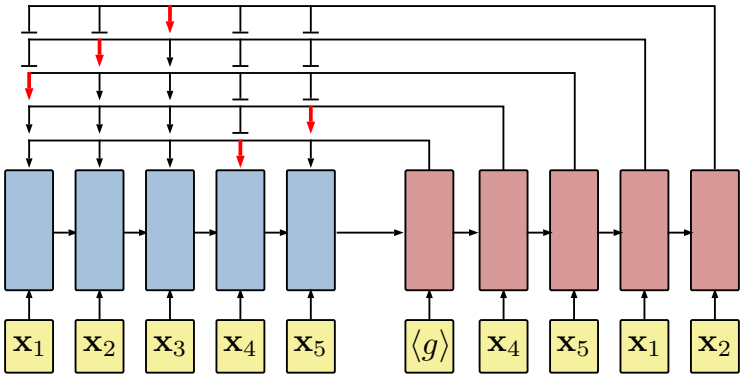

Figure 2: A pointer network architecture introduced by (Vinyals et al., 2015b).

3.1 Architecture Details

Our pointer network comprises two recurrent neural network (RNN) modules, encoder and decoder, both of which consist of Long Short-Term Memory (LSTM) cells (Hochreiter & Schmidhuber, 1997). The encoder network reads the input sequence s, one city at a time, and transforms it into a sequence of latent memory states $\{enc_i\}_{i=1}^n$ where $enc_i \in \mathbb{R}^d$. The input to the encoder network at time step i is a d-dimensional embedding of a 2D point $\mathbf{x}_i$, which is obtained via a linear transformation of $\mathbf{x}_i$ shared across all input steps. The decoder network also maintains its latent memory states $\{dec_i\}_{i=1}^n$ where $dec_i \in \mathbb{R}^d$ and, at each step i, uses a pointing mechanism 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. The input of the first decoder step (denoted by $\langle g \rangle$ in Figure 2) is a d-dimensional vector treated as a trainable parameter of our neural network.

Our attention function, formally defined in Appendix A.1, takes as input a query vector $q = dec_i \in \mathbb{R}^d$ and a set of reference vectors $ref = \{enc_1, \ldots, enc_k\}$ where $enc_i \in \mathbb{R}^d$, and predicts a distribution $A(ref, q)$ over the set of k references. This probability distribution represents the degree to which the model is pointing to a reference r_i upon seeing a query q.

Vinyals et al. (2015a) also suggest including some additional computation steps, named *glimpses*, to aggregate the contributions of different parts of the input sequence, very much like (Bahdanau et al., 2015). We discuss this approach in details in Appendix A.1. In our experiments, we find that utilizing one glimpse in the pointing mechanism yields performance gains at an insignificant cost latency.

4 Optimization with Policy Gradients

Vinyals et al. (2015b) propose training a pointer network using a supervised loss function comprising conditional log-likelihood, which factors into a cross entropy objective between the network's output probabilities and the targets provided by a TSP solver. Learning from examples in such a way is undesirable for NP-hard problems because (1) the performance of the model is tied to the quality of the supervised labels, (2) getting high-quality labeled data is expensive and may be infeasible for new problem statements, (3) one cares more about finding a competitive solution more than replicating the results of another algorithm.

By 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 model-free policy-based Reinforcement Learning to optimize the parameters of a pointer network denoted $\boldsymbol{\theta}$. Our training objective is the expected tour length which, given an input graph s, is defined as

$$J(\boldsymbol{\theta} \mid s) = \mathbb{E}_{\pi \sim p_\theta(.|s)} \, L(\pi \mid s) \, . \tag{3}$$

During training, our graphs are drawn from a distribution $\mathcal{S}$, and the total training objective involves sampling from the distribution of graphs, *i.e.* $J(\boldsymbol{\theta}) = \mathbb{E}_{s \sim \mathcal{S}} \, J(\boldsymbol{\theta} \mid s)$.

Algorithm 1 Actor-critic training

procedure TRAIN(training set S, number of training steps T, batch size B)
  Initialize pointer network params θ
  Initialize critic network params θ_v
  for $t = 1$ to T **do**
    $s_i \sim \text{SAMPLEINPUT}(S)$ for $i \in [|1, B|]$
    $\pi_i \sim \text{SAMPLESOLUTION}(p_\theta(.|s_i))$ for $i \in [|1, B|]$
    $b_i \leftarrow b_{\theta_v}(s_i)$
    $\nabla_\theta \leftarrow \frac{1}{B} \sum_{i=1}^{B} (L(\pi_i|s_i) - b_i) \nabla_\theta \log p_\theta(\pi_i|s_i)$
    $\mathcal{L}_v \leftarrow \frac{1}{B} \sum_{i=1}^{B} \|b_i - L(\pi_i)\|_2^2$
    $\theta \leftarrow \text{ADAM}(\theta, \nabla_\theta)$
    $\theta_v \leftarrow \text{ADAM}(\theta_v, \nabla_{\theta_v} \mathcal{L}_v)$
  end for
  return θ
end procedure

We resort to policy gradient methods and stochastic gradient descent to optimize the parameters. The gradient of (3) is formulated using the well-known REINFORCE algorithm (Williams, 1992):

$$\nabla_\theta J(\theta \mid s) = \mathbb{E}_{\pi \sim p_\theta(.|s)} \Big[\big(L(\pi \mid s) - b(s) \big) \nabla_\theta \log p_\theta(\pi \mid s) \Big] , \tag{4}$$

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.

By drawing B *i.i.d.* sample graphs $s_1, s_2, \ldots, s_B \sim \mathcal{S}$ and sampling a single tour per graph, *i.e.* $\pi_i \sim p_\theta(. \mid s_i)$, the gradient in (4) is approximated with Monte Carlo sampling as follows:

$$\nabla_\theta J(\theta) \approx \frac{1}{B} \sum_{i=1}^{B} \Big(L(\pi_i|s_i) - b(s_i) \Big) \nabla_\theta \log p_\theta(\pi_i \mid s_i) . \tag{5}$$

A simple and popular choice of the baseline $b(s)$ is an exponential moving average of the rewards obtained by the network over time to account for the fact that the policy improves with training. While this choice of baseline proved sufficient to improve upon the Christofides algorithm, it suffers from not being able to differentiate between different input graphs. In particular, the optimal tour π^* for a difficult graph s may be still discouraged if $L(\pi^*|s) > b$ because b is shared across all instances in the batch.

Using a parametric baseline to estimate the expected tour length $\mathbb{E}_{\pi \sim p_\theta(.|s)} L(\pi \mid s)$ typically improves learning. Therefore, 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 critic is trained with stochastic gradient descent on a mean squared error objective between its predictions $b_{\theta_v}(s)$ and the actual tour lengths sampled by the most recent policy. The additional objective for optimizing the baseline parameters denoted θ_v is formulated as

$$\mathcal{L}(\theta_v) = \frac{1}{B} \sum_{i=1}^{B} \big\| \, b_{\theta_v}(s_i) - L(\pi_i \mid s_i) \, \big\|_2^2 . \tag{6}$$

Our training algorithm, described in Algorithm 1, is closely related to the asynchronous advantage actor-critic (A3C) proposed in (Mnih et al., 2016), as the difference between the sampled tour lengths and the critic's predictions is an unbiased estimate of the advantage function. We perform our updates asynchronously across multiple workers, but each worker also handles a mini-batch of graphs for better gradient estimates.

4.1 SEARCH STRATEGIES

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. This inference process resembles how solvers search over a large set of feasible solutions. In this paper, we consider two search strategies detailed below, which we refer to as *sampling* and *active search*.

Algorithm 2 Active Search

procedure ACTIVESEARCH(input s, θ, number of candidates K, B, α)
  $\pi \leftarrow$ RANDOMSOLUTION()
  $L_\pi \leftarrow L(\pi \mid s)$
  $n \leftarrow \lceil \frac{K}{B} \rceil$
  for $t = 1 \dots n$ **do**
    $\pi_i \sim$ SAMPLESOLUTION$(p_\theta(. \mid s))$ for $i \in [|1, B|]$
    $j \leftarrow$ ARGMIN$(L(\pi_1 \mid s) \dots L(\pi_B \mid s))$
    $L_j \leftarrow L(\pi_j \mid s)$
    if $L_j < L_\pi$ **then**
      $\pi \leftarrow \pi_j$
      $L_\pi \leftarrow L_j$
    end if
    $\nabla_\theta \leftarrow \frac{1}{B} \sum_{i=1}^{B} (L(\pi_i \mid s) - b) \nabla_\theta \log p_\theta(\pi_i \mid s_i)$
    $\theta \leftarrow$ ADAM(θ, ∇_θ)
    $b \leftarrow \alpha * b + (1 - \alpha) * (\frac{1}{B} \sum_{i=1}^{B} b_i)$
  end for
  return π
end procedure

Sampling. Our first approach is simply to sample many candidate tours from our stochastic policy $p_\theta(.|s)$ and select the shortest one. In contrast to heuristic solvers, we do not enforce our model to sample different tours during the search. However, we can control the diversity of the sampled tours with a temperature hyperparameter when sampling from our non-parametric softmax (see Appendix A.2). This sampling process yields significant improvements over greedy decoding, which always selects the index with the largest probability. We also considered perturbing the pointing mechanism with random noise and greedily decoding from the obtained modified policy, similarly to Cho (2016), but this proved less effective than sampling in our experiments.

Active Search. Rather than sampling with a fixed model and ignoring the reward information obtained from the sampled solutions, one can refine the parameters of the stochastic policy p_θ during inference to minimize $\mathbb{E}_{\pi \sim p_\theta(.|s)} L(\pi \mid s)$ on a *single test input s*. This approach proves especially competitive when starting from a trained model. But more remarkably, it also produces satisfying solutions when starting from an untrained model. We will refer to these two approaches as *RL pretraining - Active Search* and *Active Search* because the model actively updates its parameters while searching candidate solutions on a single test instance.

Active Search applies policy gradients similarly to Algorithm 1 but draws Monte Carlo samples over candidate solutions $\pi_1 \dots \pi_B \sim p_\theta(.|s)$ for a single test input. It also resorts to an exponential moving average baseline, rather than a critic, as there is no need to differentiate between inputs. Our Active Search training algorithm is presented in Algorithm 2. We note that while RL training does not require supervision, it still requires training data and hence generalization depends on the training data distribution. In contrast, Active Search is distribution independent. Finally, since we encode a set of cities as a sequence, we randomly shuffle the input sequence before feeding it to our pointer network. This increases the stochasticity of the sampling procedure and leads to large improvements in Active Search.

5 EXPERIMENTS

We conduct experiments to investigate the behavior of the proposed Neural Combinatorial Optimization methods. We consider three benchmark tasks, Euclidean TSP20, 50 and 100, for which we generate a test set of $1,000$ graphs. Points are drawn uniformly at random in the 2D unit square $[0, 1]^2$. Table 1 summarizes the configurations and different search strategies used in the experiments. Our experimental procedure and results are as follows.

5.1 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

Table 1: Different learning configurations.

Configuration	Learn on training data	Sampling on test set	Refining on test set
RL pretraining-Greedy	Yes	No	No
Active Search (AS)	No	Yes	Yes
RL pretraining-Sampling	Yes	Yes	No
RL pretraining-Active Search	Yes	Yes	Yes

the Adam optimizer (Kingma & Ba, 2014) and use an initial learning rate of 10^{-3} for TSP20 and TSP50 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 and observed that using more glimpses did not improve the results. 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. Our model and training code will be made availabe soon. Experiments were conducted using Tensorflow (Abadi et al., 2016).

Supervised Learning. In addition to the described baselines, we implement and train a pointer network with supervised learning, similarly to (Vinyals et al., 2015b). While our supervised data consists of one million optimal tours, we find that our supervised learning results are not as good as those reported in by (Vinyals et al., 2015b). We suspect that learning from optimal tours is harder for supervised pointer networks due to subtle features that the model cannot figure out only by looking at given supervised targets. We thus refer to the results in (Vinyals et al., 2015b) for TSP20 and TSP50 and report our results on TSP100, all of which are suboptimal compared to other approaches.

RL pretraining. For the RL experiments, we generate training mini-batches of inputs on the fly and update the model parameters with the Actor Critic Algorithm 1. We use a validation set of $10,000$ randomly generated instances for hyper-parameter tuning. Our critic consists of an encoder network having the same architecture as the policy network's encoder, but followed by 3 processing steps as described in (Vinyals et al., 2015a) and 2 fully connected layers with respectively 128 and 1 unit(s). We find that clipping the logits to $[-10, 10]$ with a tanh activation function, as described in Appendix A.2, helps with exploration and yields marginal performance gains. The simplest search strategy using an RL pretrained model is greedy decoding, *i.e.* selecting the city with the largest probability at each decoding step.

RL pretraining Sampling. For each test instance, we sample $1,000$ batches of solutions from a pretrained model, after which we do not see significant improvement, and keep track of the shortest tour. A grid search over the temperature hyperparameter found respective temperatures of 3.0, 2.0 and 1.0 to yield the best results for TSP20, TSP50 and TSP100.

RL pretraining Active Search. For each test instance, we initialize the model parameters from a pretrained RL model and run Active Search for up to $10,000$ training steps. We set the learning rate to a hundredth of the initial learning rate the TSP agent was trained on (i.e. 10^{-5} for TSP20/TSP50 and 10^{-6} for TSP100).

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 TSP20,50 and $200,000$ training steps on TSP100.

5.2 Results and Analyses

We compare our methods against Christofides (1976) as well as optimal solutions, which we get from Concorde (Applegate et al., 2006). We report the average tour lengths of our approaches on our three test sets in Table 2. All our methods comfortably outperform Christofides'. Notably, results demonstrate that training with RL significantly improves over supervised learning (Vinyals et al., 2015b). We present a more detailed comparison of methods in Figure 3, where we sort the ratios to optimality of our different learning configurations.

Searching at inference time proves crucial to approach optimality. Most importantly, RL pretraining Active Search is the best among all Neural Combinatorial Optimization methods and successfully re-

Table 2: Average tour length (lower is better). Results marked $^{(\dagger)}$ are from (Vinyals et al., 2015b).

Task	Supervised Learning	RL pretraining			AS	Christofides	Optimal
		greedy	sampling	AS			
TSP20	$3.88^{(\dagger)}$	3.89	3.82	3.82	3.96	–	3.82
TSP50	$6.09^{(\dagger)}$	5.95	5.71	5.70	5.87	6.62	5.68
TSP100	10.81	8.30	7.92	7.83	8.19	9.18	7.77

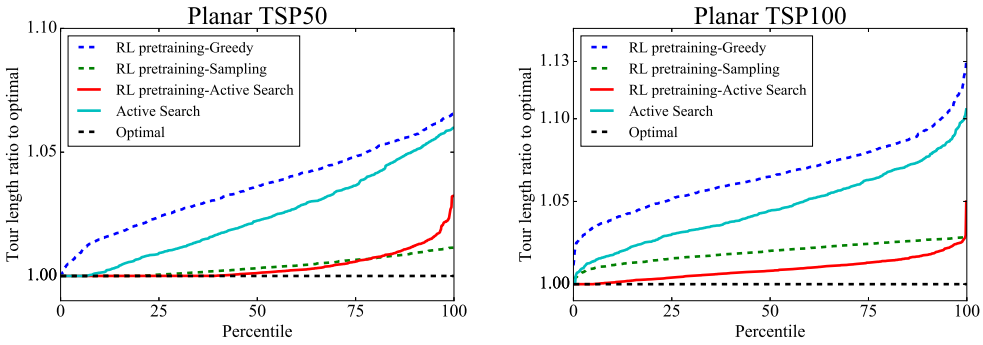

Figure 3: Sorted tour length ratios to optimality

covers optimal tours in a significant numbers of test cases. Remarkably, Active Search also achieves satisfying results and outperforms the greedy policy obtained with RL pretraining.

In Table 3, we show the average performance of our search strategies on TSP50 and TSP100 as they consider more batches of candidate solutions. We find that RL pretraining Sampling and RL pretraining Active Search get most of their improvements over the greedy policy within a small number of candidate batches but still benefit from longer search procedures. On the contrary, Active Search, which starts from randomly initialized parameters, requires many more training steps before producing competitive tours. Finally, we show randomly picked example tours found by our methods in Figure 4 in Appendix A.3.

Table 3: Average tour lengths of all methods as they sample more mini-batches of tours. Each batch consists of 128 sequences

	# Batches	RL pretraining			AS
		Sampling $T = 1$	Sampling $T = T^*$	AS	
TSP50	0	5.97	6.41	5.97	23.84
	50	5.80	5.80	5.74	17.83
	100	5.79	5.79	5.74	16.29
	1,000	5.73	5.71	5.71	12.47
	10,000	–	–	5.70	10.76
	100,000	–	–	–	5.87
TSP100	0	8.59	8.59	8.59	49.12
	50	8.13	8.13	7.97	39.15
	100	8.10	8.10	7.95	37.07
	1,000	7.92	7.92	7.88	25.74
	10,000	–	–	7.83	25.64
	200,000	–	–	–	8.19

6 Generalization to other problems

In this section, we discuss how to apply Neural Combinatorial Optimization to other problems than the TSP. In Neural Combinatorial Optimization, the model architecture is tied to the given combinatorial optimization problem. Examples of useful networks include the pointer network, when the output is a permutation or a truncated permutation or a subset of the input, and the classical seq2seq

model for other kinds of structured outputs. For combinatorial problems that require to assign labels to elements of the input, such as graph coloring, it is also possible to combine a pointer module and a softmax module to simultaneously point and assign at decoding time. Constraints on the solutions are added to the decoder to ensure the feasiliby of the solutions, similarly to how we enforce the model to not point at the same city twice in our pointing mechanism (see Appendix A.1). Given a model that outputs feasible solutions, the training procedures described in Section 4 can then be applied by adapting the reward function depending on the optimization problem.

As an example of the flexibility of Neural Combinatorial Optimization, we consider the KnapSack problem, another intensively studied problem in computer science. Given a set of n items $i = 1...n$, each with weight w_i and value v_i and a maximum weight capacity of W, the 0-1 KnapSack problem consists in maximizing the sum of the values of items present in the knapsack so that the sum of the weights is less than or equal to the knapsack capacity:

$$\begin{aligned} \max_{S \subseteq \{1,2,\dots,n\}} \quad & \sum_{i \in S} v_i \\ \text{subject to} \quad & \sum_{i \in S} w_i \leq W \end{aligned} \tag{7}$$

With w_i, v_i and W taking real values, the problem is NP-hard (Kellerer et al., 2004). Two simple heuristics are ExpKnap, which employs brand-and-bound with Linear Programming bounds (Pisinger, 1995), and MinKnap, which employs dynamic programming with enumerative bounds (Pisinger, 1997). Exact solutions can also be optained by quantizing the weights to high precisions and then performing dynamic programming with a pseudo-polynomial complexity (Bertsimas & Demir, 2002).

We apply the pointer network and encode each knapsack instance as a sequence of 2D vectors (w_i, v_i). At decoding time, the pointer network points to items to include in the knapsack and stops when the total weight of the items collected so far exceeds the weight capacity.

We generate three datasets, KNAP50, KNAP100 and KNAP200, of a thousand instances with items' weights and values drawn uniformly at random in $[0, 1]$. Without loss of generality (since we can scale the items' weights), we set the capacities to 12.5 for KNAP50 and 25 for KNAP100 and KNAP200. We present the performances of RL pretraining-Greedy and Active Search (which we run for $5,000$ training steps) in Table 4. RL pretraining-Greedy yields solutions that, in average, are just 1% less than optimal and Active Search solves all instances to optimality.

Table 4: Performance of RL pretraining-Greedy and Active Search on KnapSack.

	RL pretraining greedy	Active Search	Optimal	ExpKnap and MinKnap
KNAP50	19.86	20.07	20.07	20.07
KNAP100	40.27	40.50	40.50	40.50
KNAP200	57.10	57.45	57.45	57.45

7 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.

ACKNOWLEDGMENTS

The authors would like to thank Vincent Furnon, Oriol Vinyals, Barret Zoph, Lukasz Kaiser, Mustafa Ispir and the Google Brain team for insightful comments and discussion.

REFERENCES

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. *arXiv preprint arXiv:1605.08695*, 2016.

Sreeram V. B. Aiyer, Mahesan Niranjan, and Frank Fallside. A theoretical investigation into the performance of the Hopfield model. *IEEE Transactions on Neural Networks*, 1(2):204–215, 1990.

Bernard Angeniol, Gael De La Croix Vaubois, and Jean-Yves Le Texier. Self-organizing feature maps and the Travelling Salesman Problem. *Neural Networks*, 1(4):289–293, 1988.

David Applegate, Robert Bixby, Vašek Chvátal, and William Cook. Implementing the dantzig-fulkerson-johnson algorithm for large traveling salesman problems. *Mathematical programming*, 2003.

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

David L Applegate, Robert E Bixby, Vasek Chvatal, and William J Cook. *The traveling salesman problem: a computational study*. Princeton university press, 2011.

Sanjeev Arora. Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems. pp. 753–782, 1998.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *ICLR*, 2015.

Dimitris Bertsimas and Ramazan Demir. An approximate dynamic programming approach to multidimensional knapsack problems. *Management Science*, 48(4):550–565, 2002.

Laura I. Burke. Neural methods for the Traveling Salesman Problem: insights from operations research. *Neural Networks*, 7(4):681–690, 1994.

Kyunghyun Cho. Noisy parallel approximate decoding for conditional recurrent language model. *arXiv preprint arXiv:1605.03835*, 2016.

Nicos Christofides. Worst-case analysis of a new heuristic for the Travelling Salesman Problem. In *Report 388*. Graduate School of Industrial Administration, CMU, 1976.

George Dantzig, Ray Fulkerson, and Selmer Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the operations research society of America*, 1954.

Richard Durbin. An analogue approach to the Travelling Salesman. *Nature*, 326:16, 1987.

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.

J. C. Fort. Solving a combinatorial problem via self-organizing process: an application of the Kohonen algorithm to the traveling salesman problem. *Biological Cybernetics*, 59(1):33–40, 1988.

Andrew Howard Gee. *Problem solving with optimization networks*. PhD thesis, Citeseer, 1993.

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

Sepp Hochreiter and Jurgen Schmidhuber. Long short-term memory. 1997.

John J. Hopfield and David W. Tank. "Neural" computation of decisions in optimization problems. *Biological Cybernetics*, 52(3):141–152, 1985.

Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack Problems*. Springer-Verlag Berlin Heidelberg, 2004.

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

Teuvo Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78(9):1464–1480, 1990.

Bert F. J. La Maire and Valeri M. Mladenov. Comparison of neural networks for solving the Travelling Salesman Problem. In *NEUREL*, pp. 21–24. IEEE, 2012.

Volodymyr Mnih, Adri Puigdomnech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *arXiv preprint arXiv:1605.03835*, 2016.

Manfred Padberg and Giovanni Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *Society for Industrial and Applied Mathematics*, 33:60–100, 1990.

Christos H. Papadimitriou. The Euclidean Travelling Salesman Problem is NP-complete. *Theoretical Computer Science*, 4(3):237–244, 1977.

David Pisinger. An expanding-core algorithm for the exact 0-1 knapsack problem european journal of operational research. *European Journal of Operational Research*, pp. 175–187, 1995.

David Pisinger. A minimal algorithm for the 0-1 knapsack problem. *Operations Research*, pp. 758–767, 1997.

B. W. Kernighan S. Lin. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, 21(2):498–516, 1973.

Farah Sarwar and Abdul Aziz Bhatti. Critical analysis of Hopfield's neural network model for TSP and its comparison with heuristic algorithm for shortest path computation. In *IBCAST*, 2012.

Kate A. Smith. Neural networks for combinatorial optimization: a review of more than a decade of research. *INFORMS Journal on Computing*, 1999.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, pp. 3104–3112, 2014.

Andrew I. Vakhutinsky and Bruce L. Golden. A hierarchical strategy for solving traveling salesman problems using elastic nets. *Journal of Heuristics*, 1(1):67–76, 1995.

Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. Order matters: Sequence to sequence for sets. *arXiv preprint arXiv:1511.06391*, 2015a.

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

Christos Voudouris and Edward Tsang. Guided local search and its application to the traveling salesman problem. *European journal of operational research*, 1999.

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

G. V. Wilson and G. S. Pawley. On the stability of the travelling salesman problem algorithm of hopfield and tank. *Biological Cybernetics*, 58(1):63–70, 1988.

A APPENDIX

A.1 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:

$$u_i = \begin{cases} v^\top \cdot \tanh\left(W_{ref} \cdot r_i + W_q \cdot q\right) & \text{if } i \neq \pi(j) \text{ for all } j < i \\ -\infty & \text{otherwise} \end{cases} \text{ for } i = 1, 2, ..., k \tag{8}$$

$$A(ref, q; W_{ref}, W_q, v) \stackrel{\text{def}}{=} softmax(u). \tag{9}$$

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

$$p(\pi(j)|\pi(< j), s) \stackrel{\text{def}}{=} A(enc_{1:n}, dec_j). \tag{10}$$

Setting the logits of cities that already appeared in the tour to $-\infty$, as shown in Equation 8, 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:

$$p = A(ref, q; W^g_{ref}, W^g_q, v^g) \tag{11}$$

$$G(ref, q; W^g_{ref}, W^g_q, v^g) \stackrel{\text{def}}{=} \sum_{i=1}^{k} r_i p_i. \tag{12}$$

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:

$$g_0 \stackrel{\text{def}}{=} q \tag{13}$$

$$g_l \stackrel{\text{def}}{=} G(ref, g_{l-1}; W^g_{ref}, W^g_q, v^g) \tag{14}$$

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.

A.2 IMPROVING EXPLORATION

Softmax temperature: We modify Equation 9 as follows:

$$A(ref, q, T; W_{ref}, W_q, v) \stackrel{\text{def}}{=} softmax(u/T), \tag{15}$$

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 9 as follows:

$$A(ref, q; W_{ref}, W_q, v) \stackrel{\text{def}}{=} softmax(C \tanh(u)), \tag{16}$$

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

A.3 SAMPLE TOURS

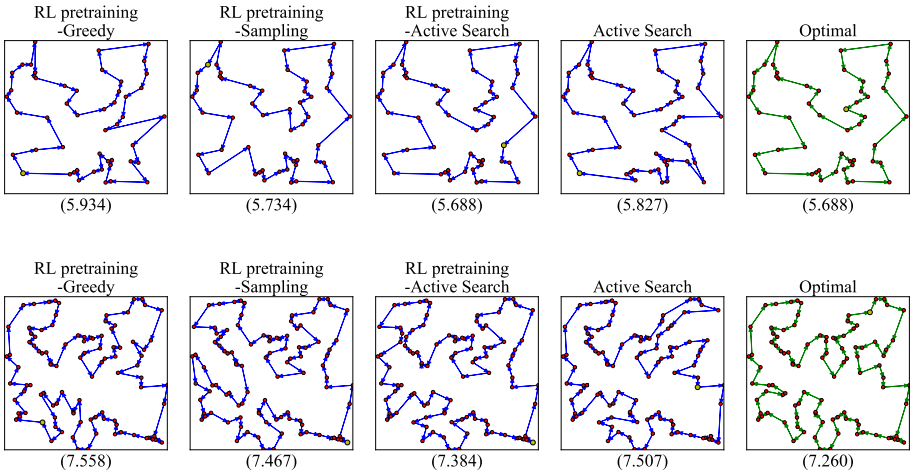

Figure 4: Sample tours. Top: TSP50; Bottom: TSP100.