Neural Simulated Annealing

Alvaro H.C. Correia[⋆†] **Daniel E. Worrall**[⋆†] **Roberto Bondesan**[†]
Eindhoven University of Technology

Abstract

Simulated annealing (SA) is a stochastic global optimisation metaheuristic applicable to a wide range of discrete and continuous variable problems. Despite its simplicity, SA hinges on carefully handpicked components, viz. proposal distribution and annealing schedule, that often have to be fine tuned to individual problem instances. In this work, we seek to make SA more effective and easier to use by framing its proposal distribution as a reinforcement learning policy that can be optimised for higher solution quality given a computational budget. The result is Neural SA, a competitive and general machine learning method for combinatorial optimisation that is efficient, and easy to design and train. We show Neural SA with such a learnt proposal distribution, parametrised by small equivariant neural networks, outperforms SA baselines on several problems: Rosenbrock's function and the Knapsack, Bin Packing and Travelling Salesperson problems. We also show Neural SA scales well to large problems (generalising to much larger instances than those seen during training) while getting comparable performance to popular off-the-shelf solvers and machine learning methods in terms of solution quality and wall-clock time.

1 INTRODUCTION

There are many different kinds of combinatorial optimisation (CO) problem, including bin packing, routing, assignment, scheduling, constraint satisfaction, and more. Solving these problems while sidestepping their inherent computational intractability has great importance and impact for the real world, where suboptimal bin packing or routing lead to wasted profit or excess greenhouse emissions (Salimifard et al., 2012). General solving frameworks or *metaheuristics* for all these problems are desirable, due to their conceptual simplicity and ease-of-deployment, but require manual tailoring to each individual problem.

One such metaheuristic is *Simulated Annealing* (SA) (Kirkpatrick et al., 1987), a simple and very popular, iterative global optimisation technique for numerically approximating the global minimum of both continuous- and discrete-variable problems. While SA has wide applicability, this is also its Achilles' Heel, leaving many design choices to the user. Namely, a user has to design 1) neighbourhood proposal distributions, which define the space of possible transitions from a solution $\mathbf{x}_k$ at time k to solutions $\mathbf{x}_{k+1}$ at time $k+1$, and 2) a temperature schedule, which controls the balance of exploration to exploitation. In this work, we mitigate the need for extensive fine-tuning of the hyperparameters in SA by designing a learnable proposal distribution, which we show improves convergence speed while adding little computational overhead (limited to $\mathcal{O}(N)$ per step for problem size N).

In recent years, research on approximate optimisation methods has been inundated by works in machine learning for CO (ML4CO) (Bengio et al., 2021). A lot of the focus has been on end-to-end neural architectures (Bello et al., 2016; Bresson and Laurent, 2021; Khalil et al., 2017; Emami and Ranka, 2018; Kool et al., 2018; Vinyals et al., 2015). Other works aimed at learning good parameters for classic algorithms, whether they be parameters of the original algorithm (Bonami et al., 2018; Kruber et al., 2017) or extra neural parameters introduced into the computational graph of classic algorithms (Chen and Tian, 2019; da Costa et al., 2020; Fu et al., 2021; Gasse et al., 2019; Gupta et al., 2020; Kool et al., 2022; Li et al., 2018; Wu et al., 2019). Our method, *neural simulated annealing* (Neural SA) can be viewed as sitting firmly within this last category.

SA is an *improvement heuristic*; it navigates the search space of feasible solutions by iteratively applying (small) perturbations to previously found solutions. Figure 1 illustrates this for the Travelling Salesperson Problem (TSP), perhaps the most classic of NP-hard problems. In this work, we pose this as a Reinforcement Learning (RL) agent

[⋆]Both authors contributed equally.

[†]This work was done while the authors worked at Qualcomm Technologies, Inc.

Proceedings of the 26[th] International Conference on Artificial Intelligence and Statistics (AISTATS) 2023, Valencia, Spain. PMLR: Volume 206.

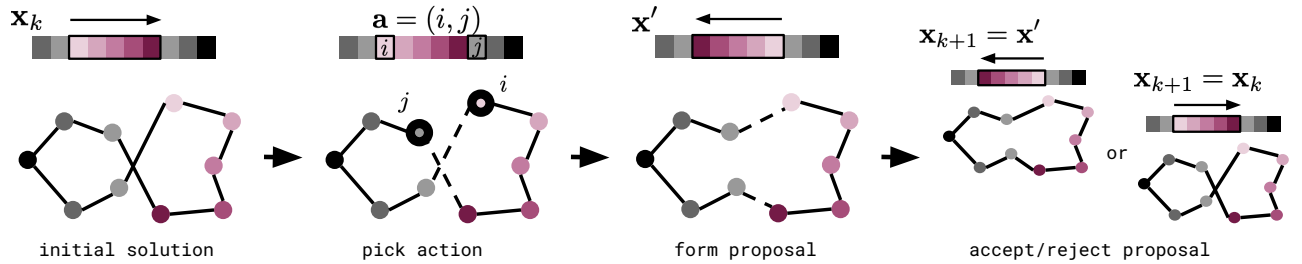


Figure 1: Neural SA pipeline for the TSP. Starting with a solution (tour) $\mathbf{x}_k$, we sample an action $\mathbf{a}{=}(i, j)$ from our learnable proposal distribution, defining start i and end j points of a 2-opt move (replacing two old with two new edges). Each pane shows both the linear and graph-based representations of a tour. From $\mathbf{x}_k$ and $\mathbf{a}$ we form a proposal $\mathbf{x}'$ which is either accepted or rejected in a MH step. Accepted moves assign $\mathbf{x}_{k+1}{=}\mathbf{x}'$; whereas, rejected moves assign $\mathbf{x}_{k+1}{=}\mathbf{x}_k$.

navigating an environment in search of better solutions. In this light the proposal distribution is an optimisable quantity that can learn about an entire class of problem instances and achieve faster convergence and better solution quality under a fixed computation budget. Our contributions are:

- We pose SA as a Markov decision process, bringing it into the realm of RL. This allows us to optimise the proposal distribution in a principled manner, while preserving the convergence guarantees of vanilla SA.
- We introduce *Neural SA*, a general machine learning method for combinatorial optimisation that is efficient and easy to design and train for new problems. Still, as we show in the experiments, Neural SA is competitive to off-the-shelf CO tools and other ML4CO methods on the TSP, Knapsack and Bin Packing problems, in terms of solution quality and wall-clock time.
- We show Neural SA transfers to problems of different sizes and also performs well on problem instances up to $40\times$ larger than the ones used for training.

2 RELATED WORK

Here we outline the basic simulated annealing algorithm and its main components. Then we provide an overview of prior works in the machine learning literature which have sought to learn parts of the algorithm or where SA has found uses in machine learning.

2.1 Combinatorial optimisation

A combinatorial optimisation problem is defined by a triple $(\mathbf{\Psi}, \mathcal{X}, E)$ where $\boldsymbol{\psi} \in \mathbf{\Psi}$ are problem instances (city locations in the TSP), $\mathcal{X}$ is the set of feasible solutions given $\boldsymbol{\psi}$ (Hamiltonian cycles in the TSP) and $E : \mathcal{X} \times \mathbf{\Psi} \to \mathbb{R}$ is an *energy function* (tour length in the TSP). Without loss of generality, the task is to minimise the energy $\min_{\mathbf{x}\in\mathcal{X}} E(\mathbf{x}; \boldsymbol{\psi})$. CO problems are in general NP-hard, meaning that there is no known algorithm to solve them in time polynomial in the number of bits that represents a problem instance. Like other ML4CO methods, Neural SA uses a finite collection of (unsolved) training instances to learn a policy that hopefully performs well on an entire class of problem instances $\mathbf{\Psi}$. At test time, we can deploy the learnt policy without having to fine tune any hyperparameters to individual instances.

2.2 Simulated Annealing

Simulated annealing (Kirkpatrick et al., 1987) is a metaheuristic for CO problems. It builds an inhomogeneous Markov chain $\mathbf{x}_0{\to}\mathbf{x}_1{\to}\mathbf{x}_2{\to}\cdots$ for $\mathbf{x}_k{\in}\mathcal{X}$, asymptotically converging to a minimiser of E. The stochastic transitions $\mathbf{x}_k{\to}\mathbf{x}_{k+1}$ depend on two quantities: 1) a proposal distribution, and 2) a temperature schedule.

The proposal distribution $\pi : \mathcal{X} \to \mathbb{P}(\mathcal{X})$, for $\mathbb{P}(\mathcal{X})$ the space of probability distributions on $\mathcal{X}$, suggests new states in the chain. It perturbs current solutions to new ones, potentially leading to lower energies immediately or later on. After perturbing a solution $\mathbf{x}_k{\to}\mathbf{x}'$, a Metropolis–Hastings (MH) step (Metropolis et al., 1953; Hastings, 1970) is executed. This either accepts the perturbation ($\mathbf{x}_{k+1}{=}\mathbf{x}'$) or rejects it ($\mathbf{x}_{k+1}{=}\mathbf{x}_k$); see Algorithm 1 for details. The target distribution of the Metropolis–Hastings step has form $p(\mathbf{x}|T_k) \propto \exp\{-E(\mathbf{x})/T_k\}$, where T_k is the *temperature* at time k. In the limit $T_k{\to}0$, this distribution tends to a sum of Dirac deltas on the minimisers of the energy. The temperature is annealed according to the temperature schedule, $T_1, T_2, \ldots$, from high to low, to steer the target distribution smoothly from broad to peaked around the global optima. The algorithm is outlined in Algorithm 1.

Under certain regularity conditions and provided the chain is long enough, it will visit the minimisers almost surely (Geman and Geman, 1984). More concretely, $\lim_{k\to\infty} P\left(\mathbf{x}_k \in \arg\min_{\mathbf{x}\in\mathcal{X}} E(\mathbf{x}; \boldsymbol{\psi})\right) = 1$. Despite this guarantee, practical convergence speed is determined by π and the temperature schedule, which are hard to fine-tune. There exist problem-specific heuristics for setting these (Pereira and Fernandes, 2004; Cicirello, 2007), but in this paper we propose to learn the proposal distribution.

2.3 Simulated annealing and machine learning

A natural way to combine machine learning and SA is to design local improvement heuristics that feed off each other. Cai et al. (2019) and Vashisht et al. (2020) use RL to find good initialisations for standard SA. Kosanoglu et al. (2022) go a step further and alternate between a few RL steps and complete SA runs, i.e. they run SA every few steps to explore the region around the current RL solution. These approaches are fundamentally different to ours, as we augment SA with RL-optimisable components, instead of simply using them as standalone algorithms that only interact via shared solutions. In fact, our method is fully compatible with theirs and any other SA application.

Other works sought to optimise components of SA or other optimisation algorithms with RL (Beloborodov et al., 2020; Khairy et al., 2020; Mills et al., 2020; Wauters et al., 2020; Biedenkapp et al., 2020; Shala et al., 2020) or statistical machine learning techniques (Blum et al., 2021). In contrast to these methods that optimise hyperparameters, we frame SA itself as an RL problem, which allows us to define and train the proposal distribution as a policy.

More closely to our method, other approaches improve the proposal distribution of SA or similar local search heuristics (da Costa et al., 2020; Yolcu and Póczos, 2019). In Adaptive Simulated Annealing (ASA) (Ingber, 1996) the proposal distribution is not fixed but evolves throughout the annealing process as a function of the variance of the quality of visited solutions. ASA improves the convergence of vanilla SA but is not learnable like Neural SA and dedicated to continuous optimisation problems. To the best of our knowledge, Marcos Alvarez et al. (2012) are the only ones to have proposed learning the proposal distribution for SA as we do, but they relied on supervised learning instead of RL, and thus their method requires high quality solutions or good search strategies to imitate; both expensive to compute. Conversely, Neural SA is easier to train and extend to new CO tasks, since it only requires a simple reward function that is trivially derived from the energy function.

Simulated annealing is also akin to Metropolis-Hastings, a popular choice for Markov Chain Monte Carlo (MCMC) sampling. Noé et al. (2019), Albergo et al. (2019) and de Haan et al. (2021) recently studied how to learn a proposal distribution of an MCMC chain for sampling the Boltzmann distribution of a physical system. While their results serve as motivation for our methods, we investigate a completely different context and set of applications.

Finally, our work falls under bi-level optimisation methods, where an outer optimisation loop finds the best parameters of an inner optimisation. This encompasses learning the parameters (Rere et al., 2015) or hyperparameters of a neural network optimiser (Maclaurin et al., 2015; Andrychowicz et al., 2016), learning to optimise (Li and Malik, 2017), and meta-learning (Finn et al., 2017). However, most recent approaches assume differentiable losses on continuous state spaces (Likhosherstov et al., 2021; Ji et al., 2021; Vicol et al., 2021), while we focus on the more challenging CO setting. We note, however, methods in (Vicol et al., 2021) are based on Evolution Strategies (ES) (Salimans et al., 2017), a technique that is applicable to the discrete setting and that we also consider in our experiments.

2.4 Markov Decision Processes

Simulated annealing naturally fits into the Markov Decision Process (MDP) framework as we explain below. An MDP $\mathcal{M} = (\mathcal{S}, \mathcal{A}, R, P, \gamma)$ consists of *states* $s \in \mathcal{S}$, *actions* $a \in \mathcal{A}$, an *immediate reward function* $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$, a *transition kernel* $P : \mathcal{S} \times \mathcal{A} \to \mathbb{P}(\mathcal{S})$, and a *discount factor* $\gamma \in [0, 1]$. On top of this MDP we add a stochastic *policy* $\pi : \mathcal{S} \to \mathbb{P}(\mathcal{A})$. The policy and transition kernel together define a length-K trajectory $\tau = (s_0, a_0, s_1, a_1, ..., s_K)$, which is a sample from the distribution $P(\tau|\pi) = \rho_0(s_0) \prod_{k=0}^{K-1} P(s_{k+1}|s_k, a_k)\pi(a_k|s_k)$ and where $s_0 \sim \rho_0$ is sampled from the *start-state distribution* ρ_0. One can then define the *discounted return* $R(\tau) = \sum_{k=0}^{K-1} \gamma^t r_k$ over a trajectory, where $r_k = R(s_k, a_k, s_{k+1})$. We say that we have solved an MDP if we have found a policy that maximises the expected return $\mathbb{E}_{\tau\sim P(\tau|\pi)}[R(\tau)]$.

3 METHOD

Here we outline our approach to learn the proposal distribution. First we define an MDP corresponding to SA. We then show how the proposal distribution can be optimised and provide a justification as to why this does not affect convergence guarantees of the classic algorithm.

3.1 MDP Formulation

We formalise SA as an MDP, with states $\mathbf{s} = (\mathbf{x}, \boldsymbol{\psi}, T) \in \mathcal{S}$ for $\boldsymbol{\psi}$ a parametric description of the problem instance as in Section 2, and T the instantaneous temperature. Examples are in Section 4. Our actions $\mathbf{a} \in \mathcal{A}$ perturb $(\mathbf{x}, \boldsymbol{\psi}, T) \mapsto (\mathbf{x}', \boldsymbol{\psi}, T)$, where $\mathbf{x}' \in \mathcal{N}(\mathbf{x})$ is a solution in the neighbourhood of $\mathbf{x}$. It is common to define small neighbourhoods, to limit energy variation from one state to the next. This heuristic discards exceptionally good and exceptionally bad moves, but since the latter are more common than the former, it tends to improve convergence.

We view the MH step in SA as a stochastic transition kernel, governed by the temperature of the system, with transition probabilities following a Gibbs distribution and dynamics

$$\mathbf{x}_{k+1} = \begin{cases} \mathbf{x}', & \text{with probability } p \\ \mathbf{x}_k, & \text{with probability } 1-p, \end{cases}$$
$$p = \min\left\{1, e^{-\frac{1}{T_k}(E(\mathbf{x}';\boldsymbol{\psi}) - E(\mathbf{x}_k;\boldsymbol{\psi}))}\right\}. \quad (1)$$

Algorithm 1 Neural simulated annealing. To get back to vanilla SA, replace the learnable proposal distribution π_θ with a uniform distribution π over neighbourhoods $\mathcal{N}(\bullet)$.

Require: Initial state $\mathbf{s}_0 = (\mathbf{x}_0, \boldsymbol{\psi}, T_0)$, proposal distribution π, transition function P, temperature schedule $T_1 \geq T_2 \geq T_3 \geq ...$, energy function $E(\bullet; \boldsymbol{\psi})$
  for $k = 1 : K$ **do**
    $\mathbf{a} \sim \pi_\theta(\mathbf{s}_k)$ {Sample action}
    $u \sim \text{Uniform}(u; 0, 1)$ {Metropolis–Hastings step}
    if $u < \exp\{-(E(\mathbf{x}'; \boldsymbol{\psi}) - E(\mathbf{x}_k; \boldsymbol{\psi}))/T_k\}$ **then**
      $\mathbf{s}_{k+1} \leftarrow (\mathbf{x}', \boldsymbol{\psi}, T_{k+1})$ {Accept}
    else
      $\mathbf{s}_{k+1} \leftarrow (\mathbf{x}_k, \boldsymbol{\psi}, T_{k+1})$ {Reject}
    end if
  end for

This defines a transition kernel $P(\mathbf{s}_{k+1}|\mathbf{s}_k, \mathbf{a})$, where $\mathbf{s}_{k+1}{=}(\mathbf{x}_{k+1}, \boldsymbol{\psi}, T)$. For rewards, we use either the immediate gain $r_k = E(\mathbf{x}_k; \boldsymbol{\psi}){-}E(\mathbf{x}_{k+1}; \boldsymbol{\psi})$ or the primal reward $r_k = \min_{\mathbf{x}\in\mathbf{x}_{1:k+1}} E(\mathbf{x}; \boldsymbol{\psi})$. We can learn π_θ with any policy optimisation method, but in this work we experimented with Proximal Policy Optimisation (PPO) (Schulman et al., 2017) and Evolution Strategies (ES) (Salimans et al., 2017). The immediate gain works best with PPO, where at each iteration of the rollout, the immediate gain gives fine-grained feedback on whether the previous action helped or not. The primal reward works best with ES because it is non-local, returning the minimum along an entire rollout τ at the very end. We explored using the acceptance count but found that this sometimes led to pathological behaviours. Similarly, we tried the primal integral (Berthold, 2013), which encourages finding a good solution fast, but found we could not get training dynamics to converge.

Arguably, this is a simple RL framework that does not incorporate much information about previous solutions in its state representation or reward function. Yet, as we see in the TSP experiments (Table 3), when combined with SA, this simple approach is on par with more complex RL frameworks like that of da Costa et al. (2020).

3.2 Policy Network Architecture

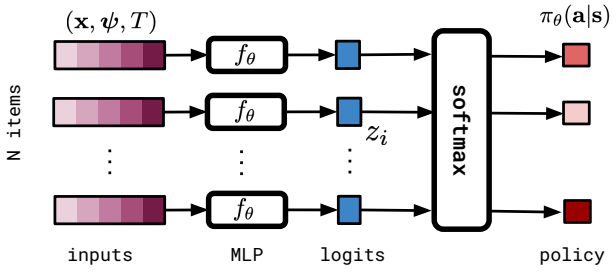


Figure 2: Policy network used in all experiments. The same MLP is applied to all inputs pointwise.

SA chains are long. That is why we need as lightweight a policy architecture as possible. Furthermore, this architecture should have the capacity to scale to varying numbers of inputs, so that we can transfer experience across problems of different size N. We opt for a very simple network, shown in Figure 2. We map the state $(\mathbf{x}, \boldsymbol{\psi}, T)$ to a set of N feature vectors, including global and relative information among dimensions. For all problems we tried, there is a natural way to do this. An MLP handles the feature vector of each dimension independently, embedding into a logit space, and a softmax maps all N logits to probabilities. The complexity of this architecture scales linearly with N and the computation is embarrassingly parallel, which is important since we plan to evaluate it many times.

A notable property of this architecture is that it is *permutation equivariant* (Zaheer et al., 2017). That is a key requirement for the CO problems we consider since they are all permutation *invariant*, e.g. the solution to the TSP is the same regardless of the order in which the cities are presented. We would like our model to preserve this symmetry, so we encode it directly into the architecture because learning it from scratch would be inefficient, requiring lots of data. In our case, the output of the model is a categorical distribution over actions $\mathbf{a}$ (the policy), not a scalar, so we actually want the architecture to be *equivariant*, i.e., $\pi_\theta(\mathbf{a}|\mathbf{s}) = \pi_\theta(\sigma \cdot \mathbf{a}|\sigma \cdot \mathbf{s})$ for σ a permutation of N objects.

3.3 Convergence

Convergence of SA to the optimum in the infinite time limit requires the Markov chain of the proposal distribution to be irreducible (Faigle and Kern, 1991; van Laarhoven and Aarts, 1987), meaning that for any temperature, any two states are reachable through a sequence of transitions with positive conditional probability under π. Our neural network policy satisfies this condition as long as the softmax layer does not assign zero probability to any state, a condition which is met in practice. Thus, Neural SA inherits convergence guarantees from standard simulated annealing.

4 EXPERIMENTS

We evaluate our method on 4 tasks: Rosenbrock's function, and the Knapsack, Bin Packing and TSP problems. We use *the same architecture and hyperparameters of Neural SA for all tasks*. This shows the wide applicability and ease of use of our method. For each task (except for Rosenbrock's function) we test Neural SA on problems of different size N, but *only train on instances of the smallest size*. Similarly, we consider rollouts of different lengths, but *only use the shortest ones during training*. This accelerates training and demonstrates Neural SA's generalisation capabilities. This type of transfer learning is one of the challenges in ML4CO (Joshi et al., 2019b), and is a merit of our lightweight, equivariant architecture.

In all experiments, we start from trivial or random solutions and adopt an exponential multiplicative cooling schedule as originally proposed by (Kirkpatrick et al., 1987), with $T_k = \alpha^k T_0$. In practice, we define the temperature schedule by fixing T_0 and T_K, and computing α according to the desired number of steps K. That allows us to vary the rollout length while maintaining the same range of temperatures for every run. We provide further experimental details in the appendix.

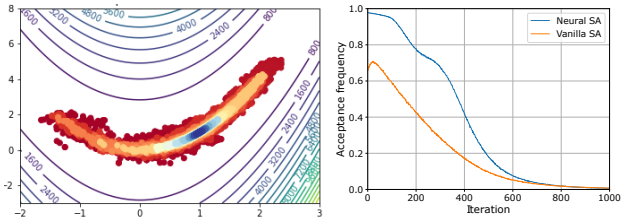
4.1 The Rosenbrock function

The Rosenbrock function is a common benchmark for optimisation algorithms. It is a non-convex function over Euclidean space given by

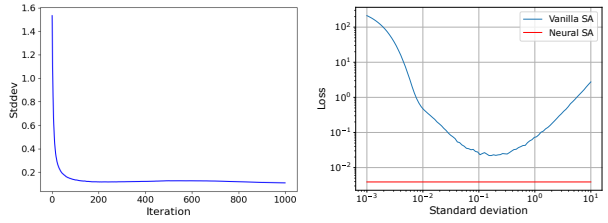
$$E(x_0, x_1; a, b) = (a - x_0)^2 + b(x_1 - x_0^2)^2, \qquad (2)$$

and with global minimum at $\mathbf{x}{=}(a, a^2)$. Gradient-based optimisers are naturally better suited to this problem, but this is still a useful toy example to showcase the properties of Neural SA. Our policy is an axis-aligned Gaussian $\pi_\theta(\mathbf{a}|\mathbf{s}){=}\mathcal{N}(\mathbf{a}; \mathbf{0}, \sigma_\theta^2(\mathbf{s}_k))$, where the variance σ_θ^2 is given by an MLP of shape $2{\to}16{\to}2$ with a ReLU in the middle. Proposals are of the form $\mathbf{x}'{=}\mathbf{x}{+}\mathbf{a}$, and states defined as $\mathbf{s}_k{=}(\mathbf{x}_k, a, b, T_k)$. Figure 3a illustrates an example rollout.

We contrast Neural SA against vanilla SA with fixed proposal distribution, i.e. $\sigma(\mathbf{s}_i){=}\sigma$, for different σ averaged over 2^{17} problem instances. Figure 3d shows that no constant variance policy can outperform an adaptive policy on this problem. Plots of acceptance ratio in Figure 3b show Neural SA has higher acceptance probability early in the rollout, a trend we observed in all experiments, suggesting its proposals are skewed towards lower energy solutions than standard SA. Figure 3c shows the variance network $\sigma_\theta^2(\mathbf{s}_i)$ as a function of time. It has learnt to take large steps until hitting the basin, whereupon large moves are rejected with high probability, and thus variance must be reduced.



(a) 2D rollout (b) Acceptance ratios



(c) σ by iteration (d) Baseline sweep

Figure 3: Results on Rosenbrock's function: (a) Example trajectory, going from red to blue, showing convergence to the minimiser at (1,1); (b) Neural SA has higher acceptance ratio than the baseline, a trend observed in all experiments; (c) Standard deviation of the learnt policy per iteration. Large initial steps offer great gains followed by small exploitative steps; (d) A non-adaptive SA baseline cannot match an adaptive one, no matter the standard deviation.

4.2 Knapsack Problem

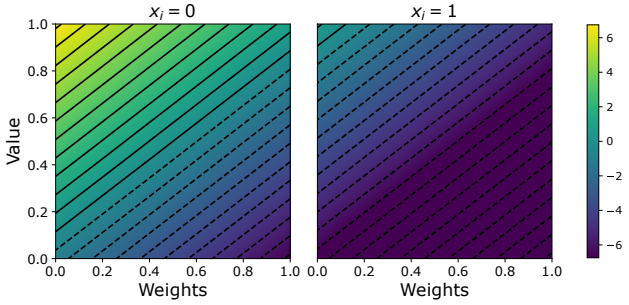


Figure 4: Knapsack logits: Policy logits for $x_i = 0$ and $x_i = 1$ are shown in each pane. Light, valuable objects are favoured to insert. Once inserted the policy down-weights an object's probably of flipping state again.

The Knapsack problem is a classic CO problem in resource allocation. Given a set of N items, each of a different value $v_i{>}0$ and weight $w_i{>}0$, the goal is to find a subset of items that maximises the sum of values while respecting a maximum total weight of W. This is the 0-1 Knapsack Problem, which is weakly NP-complete and has a search space of size 2^N and corresponding integer linear program

$$\begin{aligned} &\text{minimise } E(\mathbf{x}; \boldsymbol{\psi}) = -\sum_{i=0}^{N-1} v_i x_i, \\ &\text{subject to } \sum_{i=0}^{N-1} w_i x_i \leq W, \quad x_i \in \{0, 1\}. \end{aligned} \qquad (3)$$

Solutions are represented as a binary vector $\mathbf{x}$, with $x_i{=}0$ for 'out of the bin' and $x_i{=}1$ for 'in the bin'. Our proposal distribution flips individual bits, one at a time, with the constraint that we cannot flip $0 \mapsto 1$ if the bin capacity will be exceeded. The neighbourhood of $\mathbf{x}_k$ is thus all feasible solutions at a Hamming distance of 1 from $\mathbf{x}_k$. We use the proposal distribution described in Section 3.2 and illustrated in Figure 2, consisting of a pointwise embedding of each item—its weight, value, occupancy bit, the knapsack's overall capacity, and global temperature—into a logit-space, followed by a softmax. Mathematically the

Table 1: Average solutions for the Knapsack Problem across five random seeds and, in parentheses, optimality gap to best solution found among solvers. Bigger is better. *Values as reported by (Bello et al., 2016).

	Random Search	Bello RL	Bello AS	SA	Ours (PPO)	Ours (ES)	Greedy	OR-Tools
KNAP50	17.91*	19.86*	20.07*	18.43 (8.40%)	19.69 (2.23%)	19.95 (0.84%)	19.94 (0.89%)	**20.12** (0.00%)
KNAP100	33.23*	40.27*	40.50*	36.81 (8.91%)	39.54 (2.15%)	39.90 (1.26%)	40.17 (0.59%)	**40.41** (0.00%)
KNAP200	35.95*	57.10*	57.45*	50.89 (11.73%)	55.03 (4.54%)	55.58 (3.59%)	57.30 (0.61%)	**57.65** (0.00%)
KNAP500		-	-	126.92 (11.95%)	138.14 (4.16%)	141.01 (2.17%)	143.77 (0.25%)	**144.14** (0.00%)
KNAP1K	-	-	-	254.45 (11.96%)	277.41 (4.01%)	282.46 (2.26%)	288.64 (0.13%)	**289.01** (0.00%)
KNAP2K	-	-	-	507.72 (12.03%)	554.32(3.97%)	563.75(2.34%)	576.89 (0.06%)	**577.28** (0.00%)

policy and state–action to proposal mapping are

$$\pi_\theta(i|\mathbf{s}) = \text{softmax}\,(\mathbf{z})_i\,, \quad z_i = f_\theta([x_i, w_i, v_i, W, T]),$$
$$\mathbf{x}' = \mathbf{x} + \text{onehot}(i) \pmod 2. \tag{4}$$

where f_θ is a small two-layer neural network $5{\to}16{\to}1$ with ReLU activations, comprising only 112 parameters. Actions are sampled from the categorical distribution induced by the softmax and cast to one-hot vectors onehot(i).

Neural networks have been used to solve the Knapsack Problem in (Vinyals et al., 2015), (Nomer et al., 2020), and (Bello et al., 2016). We follow the setup of (Bello et al., 2016), honing in on 3 self-generated datasets: KNAP50, KNAP100 and KNAP200. KNAPN consists of N items with weights and values generated uniformly at random in $(0, 1]$ and capacities $C_{50}{=}12.5, C_{100}{=}25$, and $C_{200}{=}25$, with C_N the capacity in KNAPN. We use OR-Tools (Perron and Furnon, 2019) to compute ground-truth solutions.

Results in Table 1 show that Neural SA improves over vanilla SA by up to 10% optimality gap, and heuristic methods (Random Search) by much more. Neural SA falls slightly behind two methods by (Bello et al., 2016) that use (1) a large attention-based pointer network with several orders of magnitude more parameters in Bello RL, and (2) this coupled with 5000 iterations of their Active Search method. Neural SA also falls behind a greedy heuristic for packing a knapsack based on the value-to-weight ratio. In Figure 4 we analyse the policy network and a typical rollout. It has learnt a mostly greedy policy to fill its knapsack with light, valuable objects, only ejecting them when full. This is in line with the value-to-weight greedy heuristic. Despite not coming top among methods, Neural SA is within 1-3% of the minimum energy, although its architecture was not designed for this problem in particular.

4.3 Bin Packing Problem

The Bin Packing problem is similar to the Knapsack problem in nature. Here, one wants to pack *all* of N items into the smallest number of bins possible, where each item $i \in \{1, \cdots, N\}$ has weight w_i, and we assume, without loss of generality, N bins of equal capacity $W \geq \max_i(w_i)$; there would be no valid solution otherwise. This problem is NP-hard and has a search space of size equal to the N^{th} Bell number. If $x_{ij} \in \{0, 1\}$ denotes item i occupying bin j, then the problem can be written as:

$$\text{min.}\;\; E(\mathbf{x}; \boldsymbol{\psi}) = \sum_{j=0}^{N-1} y_j, \;\; \text{sbj. to} \underbrace{\sum_{i=0}^{N-1} w_i x_{ij} \leq W}_{\text{bin capacity constraint}},$$
$$\underbrace{\sum_{j=0}^{N-1} x_{ij} = 1}_{\text{1 bin per item}}, \quad \underbrace{y_j = \min\left(1, \sum_{i=0}^{N-1} x_{ij}\right)}_{\text{bin occupancy indicator}}, \tag{5}$$

where the constraints apply for all i and j. We define the policy in two steps: we first pick an item i, and then select a bin j to place it into. We can then write the policy as $\pi_{\theta,\phi}(\mathbf{a}{=}(i,j)|\mathbf{s}){=}\pi_\phi(i|\mathbf{s})\pi_\theta(j|\mathbf{s}, i)$, with

$$\pi_\theta(i|\mathbf{s}){=}\text{softmax}\left(\mathbf{z}^{\text{item}}\right)_i, \;\; z_i^{\text{item}}{=}f_\theta([w_i, c_{b(i)}, T]),$$
$$\pi_\phi(j|\mathbf{s}, i){=}\text{softmax}\left(\mathbf{z}^{\text{bin}}\right)_j, \;\; z_j^{\text{bin}}{=}f_\phi([w_i, c_j, T]), \tag{6}$$

where $b(i)$ is the bin item i is in before the action (in terms of x_{ij}, we have $x_{ib(i)}{=}1$), c_j is the free capacity of bin j ($c_j = W - \sum_{i=1}^{N} w_i x_{ij}$), and both f_θ and f_ϕ are lightweight architectures $3{\to}16{\to}1$ with a ReLU nonlinearity between the two layers. We sample from the policy ancestrally, sampling first an item from $\pi_\theta(i|\mathbf{s})$, followed by a bin from $\pi_\phi(j|\mathbf{s}, i)$. Results in Table 2 show that our lightweight model is able to find a solution to about 1% higher energy than the minimum found by FFD (Johnson, 1973), a very strong heuristic for this problem (Rieck,

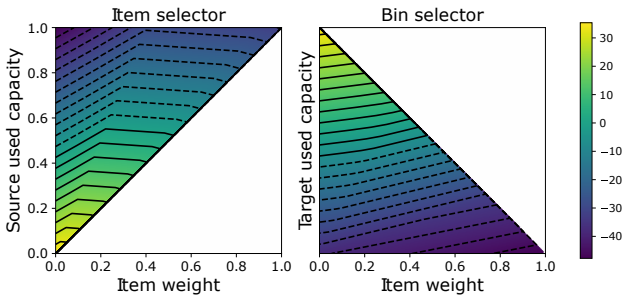


Figure 5: Policy for the Bin Packing Problem: The learnt policy is very sensible. The item selector looks for a light item in an under-full bin. The bin selector then places this in an almost-full bin. We mask bins with insufficient free capacity, hence the triangular logit-spaces.

Table 2: Average solutions for Bin Packing across five random seeds and, in parentheses, optimality gap to best solution found among solvers. Lower is better. We set a time out for Or-Tools (2 min. per instance for BIN2000 and 1 min. for the others); *only the trivial solution found before time-out.

	SA	Ours (PPO)	Ours (ES)	OR-Tools (SCIP)	FFD
BIN50	30.38 (13.74%)	27.32 (2.28%)	27.24 (1.98%)	**26.71** (0.00%)	27.10 (1.46%)
BIN100	60.66 (14.65%)	53.53 (1.17%)	53.38 (0.88%)	53.91 (1.89%)	**52.91** (0.00%)
BIN200	121.27 (16.32%)	105.63 (1.32%)	105.43 (1.13%)	109.19 (4.74%)	**104.25** (0.00%)
BIN500	302.84 (17.82%)	259.08 (0.80%)	259.09 (0.80%)	267.63 (4.13%)	**257.02** (0.00%)
BIN1000	605.23 (18.79%)	512.66 (0.63%)	512.66 (0.63%)	1000*	**509.46** (0.00%)
BIN2000	1209.72 (18.84%)	**1017.88** (0.00%)	**1017.88** (0.00%)	2000*	1028.67 (1.06%)

2021). We even see that we very often beat the SCIP (Gamrath et al., 2020) optimiser in OR-Tools, which timed out on most problems. Moreover, the learnt policy shown in Figure 5 converges much faster than the vanilla version (see Figure 6 in the appendix). Once more, we see that our method, albeit simple, is competitive with hand-designed alternatives, while vanilla SA is not.

4.4 Travelling Salesperson Problem

Imagine you will make a round road-trip through N cities and want to plan the shortest route visiting each city once; this is the Travelling Salesperson Problem (TSP) (Applegate et al., 2006). The TSP has been a long time favourite of computer scientists due to its easy description and NP-hardness (the base search space has size equal to the factorial of the number of cities). Here we use it as an example of a difficult CO problem. Given cities $i \in \{0, 1, ..., N-1\}$ with spatial coordinates $\mathbf{c}_i \in [0, 1]^2$, we wish to find a linear ordering of the cities, called a *tour*, denoted by the permutation vector $\mathbf{x} = (x_0, x_1, ..., x_{N-1})$ for $x_i \in \{0, 1, ..., N-1\}$ to optimise the following objective:

$$\begin{aligned} &\text{minimise } E(\mathbf{x}; \boldsymbol{\psi}) = \sum_{i=0}^{N-1} \|\mathbf{c}_{x_{i+1}} - \mathbf{c}_{x_i}\|_2 \\ &\text{subject to } x_i \neq x_j \text{ for all } i \neq j, \end{aligned} \tag{7}$$

where we have defined $x_N = x_0$ for convenience of notation. Our action space consists of so-called *2-opt* moves (Croes, 1958), which reverse contiguous segments of a tour. An example of a 2-opt move is shown in Figure 1. We have a two-stage architecture, like in Bin Packing, which selects the start and end cities of the segment to reverse. Denoting i as the start and j as the end cities, we have $\pi_{\theta,\phi}(\mathbf{a}{=}(i, j)|\mathbf{s}) = \pi_\phi(i|\mathbf{s})\pi_\theta(j|\mathbf{s}, i)$, parametrised as

$$\begin{aligned} \pi_\theta(i|\mathbf{s}) &= \text{softmax}\,(\mathbf{z})_i\,, \quad z_i = f_\theta([\mathbf{c}_{\mathbf{x}_{[i-1:i+1]}}, T]), \\ \pi_\phi(j|\mathbf{s}, i) &= \text{softmax}\,(\mathbf{z})_j\,, \\ z_j &= f_\phi([\mathbf{c}_{\mathbf{x}_{[i-1:i+1]}}, \mathbf{c}_{\mathbf{x}_{[j-1:j+1]}}, T]), \end{aligned} \tag{8}$$

where $\mathbf{x}_{[i-1:i+1]}$ are the indices of city i and its tour neighbours $i-1$ and $i+1$. Like in the other problems we consider, this is a natural and straightforward state representation, in sharp contrast to other machine learning approaches to the TSP (Kool et al., 2018; da Costa et al., 2020).

Again, we use simple MLPs: f_θ has architecture $7{\to}16{\to}1$ and f_ϕ, $13{\to}16{\to}1$. We test on publicly available TSP20/50/100 with 10K problems each (Kool et al., 2018) and generate TSP200/500 with 1K tours each. Results, in Table 12, show Neural SA improves on vanilla SA. We also compared Neural SA to the TSP-specific Adaptive SA algorithm of Geng et al. (2011). We used the same hyperparameters reported by the authors but skipped the greedy search step they proposed, since it is too costly and applicable to any method, including ours. Even with less compute time, Neural SA (ES) matched their results, while Neural SA (PPO) largely surpassed them. Albeit not outperforming the model by Fu et al. (2021), Neural SA is neck-to-neck with other neural improvement heuristics methods, GAT-T$\{1000\}$ (Wu et al., 2019) and Costa$\{500\}$ (da Costa et al., 2020). Since Neural SA is not custom designed for the TSP as the competing methods, we view this as surprisingly good. A more complete comparison, including other neural approaches, is given in the appendix, Table 12.

5 DISCUSSION

Neural SA is a general ML4CO method that achieves competitive results, despite relying only on a simple RL framework and compact neural architectures with straightforward input features. This makes it easy to design and train Neural SA for new problems, requiring only a training set of problem instances (no solutions needed). In this section, we discuss some of the main features of Neural SA.

Computational Efficiency In its current form, with a compact equivariant architecture, Neural SA requires little computational resources; the cost of each step scales linearly in the problem size, since the architectures are embarrassingly parallel. In terms of running times, Neural SA is on par with and often faster than other TSP solvers (see Tables 3 and 12 in the appendix). For the Knapsack and Bin Packing problems, we compare running times against OR-Tools in Table 4, where we see Neural SA lags behind in the Knapsack problem, for which a fast branch-and-bound

Table 3: Comparison of Neural SA against competing methods with similar running times on TSP. We report running times for solving all problems of each class (10K instances for TSP20/50/100; 1K instances for TSP200/500). Lower is better. *Values as reported in respective works (Wu et al., 2019; da Costa et al., 2020; Fu et al., 2021).

	TSP20			TSP50			TSP100			TSP200			TSP500		
	Cost	Gap	Time	Cost	Gap	Time	Cost	Gap	Time	Cost	Gap	Time	Cost	Gap	Time
CONCORDE	3.836	0.00%	48s	5.696	0.00%	2m	7.764	0.00%	7m	10.70	0.00%	38m	16.54	0.00%	7h58m
LKH-3	3.836	0.00%	1m	5.696	0.00%	14m	7.764	0.00%	1h	10.70	0.00%	21m	16.54	0.00%	1h15m
SA	3.881	1.17%	5s	5.943	4.34%	37s	8.343	7.45%	3m	11.98	11.87%	9m	20.22	22.25%	56m
GENG ET AL.	3.844	0.21%	46m	5.814	2.07%	68m	8.156	5.06%	131m	11.74	9.72%	33m	20.00	20.93%	3h29m
OURS (PPO)	3.838	0.05%	9s	5.734	0.67%	1m	7.874	1.42%	9m	11.00	2.80%	16m	17.64	6.65%	2h16m
OURS (ES)	3.840	0.10%	9s	5.828	2.32%	1m	8.191	5.50%	9m	11.74	9.72%	16m	20.27	22.55%	2h16m
OR-TOOLS*	3.86	0.85%	1m	5.85	2.87%	5m	8.06	3.86%	23m	-	-	-	-	-	-
GAT-T{1000}*	3.84	0.03%	12m	5.75	0.83%	16m	8.01	3.24%	25m	-	-	-	-	-	-
Costa {500}*	3.84	0.01%	5m	5.72	0.36%	7m	7.91	1.84%	10m	-	-	-	-	-	-
Fu et al.*	3.84	0.00%	1m	5.70	0.01%	8m	7.76	0.04%	15m	-	-	-	-	-	-

solver is known. Yet, for the Bin Packing problem, Neural SA is faster than the available Mixed-Integer Programming solver, which only found trivial solutions for $N \geq 1000$. Finally, Neural SA is fast to train; only a few minutes with PPO and a few hours with ES. This can be attributed to its low number of parameters and to its generalisation ability; in all experiments, we could get away with training *only on the smallest instances with very short rollouts*.

Table 4: Comparison of running times (at test time) for Neural SA (PPO/ES) against OR-tools for the Knapsack and Bin Packing Problems. We report the average time to evaluate one instance for different problem sizes.

	Knapsack		Bin Packing	
	Ours	OR-Tools	Ours	OR-Tools
50N	$< 1s$	$< 1s$	$< 1s$	$54s$
100N	$1s$	$< 1s$	$1s$	$56s$
200N	$2s$	$< 1s$	$3s$	$\geq 1m$
500N	$6s$	$1s$	$10s$	$\geq 1m$
1000N	$18s$	$2s$	$29s$	$\geq 1m$
2000N	$1m5s$	$8s$	$1m43s$	$\geq 2m$

PPO vs ES Neural SA can be trained with any policy optimisation method making it highly extendable. We found no winner between PPO and ES, apart from on the TSP, where PPO excelled and generalised better to larger instances. We also noted PPO converged $\sim 10\times$ faster than ES, but ES policies were more robust, still performing well when we switched to greedy sampling, for example. Curiously, the acceptance rate over trajectories was problem dependent and always higher in Neural SA (PPO and ES) than in vanilla SA, contradicting conventional wisdom that it should be always close to 0.44 (Lam and Delosme, 1988).

Generalisation Our experiments show Neural SA generalises to different problem sizes and rollout lengths; a remarkable feat for such a simple pipeline, since transfer learning is difficult in RL and CO. Many ML4CO methods do handle problems of different sizes but underperform when tested on larger instances than the ones seen in training (Kool et al., 2018; Joshi et al., 2019b) (see appendix, Table 9). Fu et al. (2021) did achieve better generalisation for the TSP but had to resort to a suite of techniques to allow a small supervised model to be applied to larger problems. These are not easy to implement, TSP-specific, and consist only the first step in a complex pipeline that still relies on a tailored Monte-Carlo tree search algorithm.

Solution Quality In all problems we considered, Neural SA, with little to no fine-tuning of its hyperparameters, outperformed vanilla SA and could get within a few percentage points or less of global minima. Conversely, state-of-the-art SA variants are designed by searching a large space of different hyperparameters (Franzin and Stützle, 2019), a costly process that Neural SA helps us mitigate. In fact, on the TSP it outperforms the manually designed SA algorithm of Geng et al. (2011). Neural SA did not achieve state-of-the-art results, but that was not to be expected nor our main goal. Instead, we envision Neural SA as a general purpose solver, allowing researchers and practitioners to get a strong baseline quickly without the need to fine-tune classic CO algorithms or design and train complex neural networks. Given the good performance, small computational resources, and fast training across a diverse set of CO problems, we believe Neural SA is a promising solver that can strike the right balance among solution quality, computing costs and development time.

6 CONCLUSION

We presented *Neural SA*, neurally augmented simulated annealing, where the SA chain is a trajectory from an MDP. In this light, the proposal distribution could be interpreted and optimised as a policy. This has numerous benefits: 1) accelerated convergence of the chain, 2) ability to condition the proposal distribution on side-information 3) no need of ground truth data to learn the proposal distribution, 4) lightweight architectures that can be run on CPU unlike many contemporary ML4CO methods, 5) scalabil-

ity to large problems due to its lightweight computational overhead, 6) generalisation across different problem sizes. These contributions show augmenting classic, time-tested (meta)heuristics with learnable components is a promising direction for future ML4CO research. In contrast to costly end-to-end methods, this could be a shorter path to models capable of solving a wide range of CO problems. As we show in this paper, this approach can yield solid results for diverse problems and retain theoretical guarantees of existing CO algorithms, while requiring only simple neural architectures that can be easily trained on small problems.

The ease of use and flexibility of Neural SA do come with drawbacks. In all experiments we were not able to achieve the minimum energy, although we could usually get within a percentage point. Also, the model has no built-in termination condition, neither can it provide a certificate on the quality of solutions found. There is still also the question of how to tune the temperature schedule, which we did not attempt in this work. These shortcomings are all points to be addressed in upcoming research. We are also interested in extending the framework to multiple trajectories, such as in parallel tempering (Swendsen and Wang, 1986) or genetic algorithms (Holland, 1992). For these, we would maintain a population of chains, which could exchange information.

References

Albergo, M., Kanwar, G., and Shanahan, P. (2019). Flow-based generative models for markov chain monte carlo in lattice field theory. *Physical Review D*, 100(3):034515.

Andrychowicz, M., Denil, M., Gomez, S., Hoffman, M. W., Pfau, D., Schaul, T., Shillingford, B., and De Freitas, N. (2016). Learning to learn by gradient descent by gradient descent. In *Advances in neural information processing systems*, pages 3981–3989.

Applegate, D. L., Bixby, R. E., Chvátal, V., and Cook, W. J. (2006). *The Traveling Salesman Problem: A Computational Study*. Princeton University Press.

Bello, I., Pham, H., Le, Q. V., Norouzi, M., and Bengio, S. (2016). Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*.

Beloborodov, D., Ulanov, A. E., Foerster, J. N., Whiteson, S., and Lvovsky, A. (2020). Reinforcement learning enhanced quantum-inspired algorithm for combinatorial optimization. *Machine Learning: Science and Technology*, 2(2):025009.

Bengio, Y., Lodi, A., and Prouvost, A. (2021). Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421.

Berthold, T. (2013). Measuring the impact of primal heuristics. *Operations Research Letters*, 41(6):611–614.

Biedenkapp, A., Bozkurt, H. F., Eimer, T., Hutter, F., and Lindauer, M. (2020). Dynamic algorithm configuration: foundation of a new meta-algorithmic framework. In *ECAI 2020*, pages 427–434. IOS Press.

Blum, A., Dan, C., and Seddighin, S. (2021). Learning complexity of simulated annealing. In *AISTATS*, pages 1540–1548. PMLR.

Bonami, P., Lodi, A., and Zarpellon, G. (2018). Learning a classification of mixed-integer quadratic programming problems. In van Hoeve, W.-J., editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 595–604, Cham. Springer International Publishing.

Bresson, X. and Laurent, T. (2021). The transformer network for the traveling salesman problem. *arXiv preprint arXiv:2103.03012*.

Cai, Q., Hang, W., Mirhoseini, A., Tucker, G., Wang, J., and Wei, W. (2019). Reinforcement learning driven heuristic optimization. *arXiv preprint arXiv:1906.06639*.

Chen, X. and Tian, Y. (2019). Learning to perform local rewriting for combinatorial optimization. *Advances in Neural Information Processing Systems*, 32:6281–6292.

Cicirello, V. A. (2007). On the design of an adaptive simulated annealing algorithm. In *Proceedings of the international conference on principles and practice of constraint programming first workshop on autonomous search*.

Croes, G. A. (1958). A method for solving traveling-salesman problems. *Operations Research*, 6:791–812.

da Costa, P. R., Rhuggenaath, J., Zhang, Y., and Akcay, A. (2020). Learning 2-opt heuristics for the traveling salesman problem via deep reinforcement learning. In *Asian Conference on Machine Learning*, pages 465–480. PMLR.

de Haan, P., Rainone, C., Cheng, M. C., and Bondesan, R. (2021). Scaling up machine learning for quantum field theory with equivariant continuous flows. *arXiv preprint arXiv:2110.02673*.

Emami, P. and Ranka, S. (2018). Learning permutations with sinkhorn policy gradient. *arXiv preprint arXiv:1805.07010*.

Faigle, U. and Kern, W. (1991). Note on the convergence of simulated annealing algorithms. *SIAM journal on control and optimization*, 29(1):153–159.

Finn, C., Abbeel, P., and Levine, S. (2017). Model-agnostic meta-learning for fast adaptation of deep networks. In *ICML*, pages 1126–1135. PMLR.

Franzin, A. and Stützle, T. (2019). Revisiting simulated annealing: A component-based analysis. *Computers & operations research*, 104:191–206.

Fu, Z.-H., Qiu, K.-B., and Zha, H. (2021). Generalize a small pre-trained model to arbitrarily large tsp instances.

Proceedings of the AAAI Conference on Artificial Intelligence, 35(8):7474–7482.

Gamrath, G., Anderson, D., Bestuzheva, K., Chen, W.-K., Eifler, L., Gasse, M., Gemander, P., Gleixner, A., Gottwald, L., Halbig, K., Hendel, G., Hojny, C., Koch, T., Le Bodic, P., Maher, S. J., Matter, F., Miltenberger, M., Mühmer, E., Müller, B., Pfetsch, M. E., Schlösser, F., Serrano, F., Shinano, Y., Tawfik, C., Vigerske, S., Wegscheider, F., Weninger, D., and Witzig, J. (2020). The SCIP Optimization Suite 7.0. Technical report, Optimization Online.

Gasse, M., Chételat, D., Ferroni, N., Charlin, L., and Lodi, A. (2019). Exact combinatorial optimization with graph convolutional neural networks. *Advances in neural information processing systems*, 32.

Geman, S. and Geman, D. (1984). Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-6(6):721–741.

Geng, X., Chen, Z., Yang, W., Shi, D., and Zhao, K. (2011). Solving the traveling salesman problem based on an adaptive simulated annealing algorithm with greedy search. *Applied Soft Computing*, 11(4):3680–3689.

Gupta, P., Gasse, M., Khalil, E., Mudigonda, P., Lodi, A., and Bengio, Y. (2020). Hybrid models for learning to branch. *Advances in neural information processing systems*, 33:18087–18097.

Hastings, W. K. (1970). Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97–109.

Helsgaun, K. (2000). An effective implementation of the lin–kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130.

Holland, J. H. (1992). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA.

Ingber, L. (1996). Adaptive simulated annealing (asa): lessons learned. *Control and Cybernetics*, 25(1).

Ji, K., Yang, J., and Liang, Y. (2021). Bilevel optimization: Convergence analysis and enhanced design. In *ICML*, pages 4882–4892. PMLR.

Johnson, D. S. (1973). *Near-optimal bin packing algorithms*. PhD thesis, MIT.

Joshi, C. K., Laurent, T., and Bresson, X. (2019a). An efficient graph convolutional network technique for the travelling salesman problem. *arXiv preprint arXiv:1906.01227*.

Joshi, C. K., Laurent, T., and Bresson, X. (2019b). On learning paradigms for the travelling salesman problem. *arXiv preprint arXiv:1910.07210*.

Khairy, S., Shaydulin, R., Cincio, L., Alexeev, Y., and Balaprakash, P. (2020). Learning to optimize variational quantum circuits to solve combinatorial problems. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(03):2367–2375.

Khalil, E., Dai, H., Zhang, Y., Dilkina, B., and Song, L. (2017). Learning combinatorial optimization algorithms over graphs. *Advances in neural information processing systems*, 30.

Kingma, D. P. and Ba, J. (2015). Adam: A method for stochastic optimization. In Bengio, Y. and LeCun, Y., editors, *International Conference on Learning Representations (ICLR)*.

Kirkpatrick, S., Gelatt Jr, C. D., and Vecchi, M. P. (1987). Optimization by simulated annealing. In *Readings in Computer Vision*, pages 606–615. Elsevier.

Kool, W., van Hoof, H., Gromicho, J., and Welling, M. (2022). Deep policy dynamic programming for vehicle routing problems. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 19th International Conference, CPAIOR 2022, Los Angeles, CA, USA, June 20-23, 2022, Proceedings*, pages 190–213. Springer.

Kool, W., van Hoof, H., and Welling, M. (2018). Attention, learn to solve routing problems! In *International Conference on Learning Representations (ICLR)*.

Kosanoglu, F., Atmis, M., and Turan, H. H. (2022). A deep reinforcement learning assisted simulated annealing algorithm for a maintenance planning problem. *Annals of Operations Research*, pages 1–32.

Kruber, M., Lübbecke, M., and Parmentier, A. (2017). Learning when to use a decomposition. In *CPAIOR*, pages 202–210.

Lam, J. and Delosme, J.-M. (1988). Performance of a new annealing schedule. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pages 306–311.

Li, K. and Malik, J. (2017). Learning to optimize. In *International Conference on Learning Representations (ICLR)*.

Li, Z., Chen, Q., and Koltun, V. (2018). Combinatorial optimization with graph convolutional networks and guided tree search. *Advances in neural information processing systems*, 31.

Likhosherstov, V., Song, X., Choromanski, K., Davis, J. Q., and Weller, A. (2021). Debiasing a first-order heuristic for approximate bi-level optimization. In *ICML*, pages 6621–6630. PMLR.

Maclaurin, D., Duvenaud, D., and Adams, R. (2015). Gradient-based hyperparameter optimization through reversible learning. In *ICML*, pages 2113–2122. PMLR.

Marcos Alvarez, A., Maes, F., and Wehenkel, L. (2012). Supervised learning to tune simulated annealing for in silico protein structure prediction. In *ESANN 2012 proceedings, 20th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*, pages 49–54. Ciaco.

Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., and Teller, E. (1953). Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092.

Mills, K., Ronagh, P., and Tamblyn, I. (2020). Finding the ground state of spin hamiltonians with reinforcement learning. *Nature Machine Intelligence*, 2(9):509–517.

Noé, F., Olsson, S., Köhler, J., and Wu, H. (2019). Boltzmann generators: Sampling equilibrium states of many-body systems with deep learning. *Science*, 365(6457).

Nomer, H. A., Alnowibet, K. A., Elsayed, A., and Mohamed, A. W. (2020). Neural knapsack: A neural network based solver for the knapsack problem. *IEEE Access*, 8:224200–224210.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R., editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

Pereira, A. I. and Fernandes, E. M. G. P. (2004). A study of simulated annealing variants. In *Proceedings of XXVIII Congreso de Estadística e Investigación Operativa*.

Perron, L. and Furnon, V. (2019). Or-tools.

Qi, C. R., Su, H., Mo, K., and Guibas, L. J. (2017). Pointnet: Deep learning on point sets for 3d classification and segmentation. In *CVPR*, pages 652–660.

Rere, L. R., Fanany, M. I., and Arymurthy, A. M. (2015). Simulated annealing algorithm for deep learning. *Procedia Computer Science*, 72:137–144. The Third Information Systems International Conference 2015.

Rieck, B. (2021). Basic analysis of bin-packing heuristics. *arXiv preprint arXiv:2104.12235*.

Salimans, T., Ho, J., Chen, X., Sidor, S., and Sutskever, I. (2017). Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*.

Salimifard, K., Shahbandarzadeh, H., and Raeesi, R. (2012). Green transportation and the role of operations research. In *2012 International Conference on Traffic and Transportation Engineering (ICTTE 2012)*, pages 74–79.

Schulman, J., Moritz, P., Levine, S., Jordan, M., and Abbeel, P. (2016). High-dimensional continuous control using generalized advantage estimation. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Shala, G., Biedenkapp, A., Awad, N., Adriaensen, S., Lindauer, M., and Hutter, F. (2020). Learning step-size adaptation in cma-es. In *International Conference on Parallel Problem Solving from Nature*, pages 691–706. Springer.

Swendsen, R. H. and Wang, J.-S. (1986). Replica monte carlo simulation of spin-glasses. *Phys. Rev. Lett.*, 57:2607–2609.

van Laarhoven, P. and Aarts, E. (1987). *Simulated Annealing: Theory and Applications*, chapter 3, Thm. 6. Mathematics and Its Applications. Springer Netherlands.

Vashisht, D., Rampal, H., Liao, H., Lu, Y., Shanbhag, D., Fallon, E., and Kara, L. B. (2020). Placement in integrated circuits using cyclic reinforcement learning and simulated annealing. *arXiv preprint arXiv:2011.07577*.

Vicol, P., Metz, L., and Sohl-Dickstein, J. (2021). Unbiased gradient estimation in unrolled computation graphs with persistent evolution strategies. In *ICML*, pages 10553–10563. PMLR.

Vinyals, O., Fortunato, M., and Jaitly, N. (2015). Pointer networks. *Advances in neural information processing systems*, 28.

Wagstaff, E., Fuchs, F., Engelcke, M., Posner, I., and Osborne, M. A. (2019). On the limitations of representing functions on sets. In *ICML*, pages 6487–6494. PMLR.

Wauters, M. M., Panizon, E., Mbeng, G. B., and Santoro, G. E. (2020). Reinforcement-learning-assisted quantum optimization. *Physical Review Research*, 2(3).

Wu, Y., Song, W., Cao, Z., Zhang, J., and Lim, A. (2019). Learning improvement heuristics for solving the travelling salesman problem. *arXiv preprint arXiv:1912.05784*.

Yolcu, E. and Póczos, B. (2019). Learning local search heuristics for boolean satisfiability. *Advances in Neural Information Processing Systems*, 32.

Zaheer, M., Kottur, S., Ravanbakhsh, S., Poczos, B., Salakhutdinov, R. R., and Smola, A. J. (2017). Deep sets. *Advances in Neural Information Processing Systems*, 30.

Neural Simulated Annealing
Supplementary Material

A GENERAL INFORMATION

Implementation Our code was implemented in Pytorch 1.9 (Paszke et al., 2019) and run in a standard machine with a single GPU RTX2080. The code will be made publicly available upon publication.

Architectures In all experiments, the proposal distribution is parametrised by a two-layer neural network, with ReLU activation and 16 neurons in the hidden layer: input_size $\to 16 \to 1$, where the size of the input is problem specific. More precisely, our architectures had 384 parameters for the TSP, 160 for the Bin Packing problem, and 112 for the Knapsack problem. When using PPO, we also need a critic network to estimate the state-value function so that we can compute advantages using Generalised Advantage Estimator (GAE) (Schulman et al., 2016). The critic network does not share any parameters with the proposal distribution (actor) but has the exact same architecture. The only difference is that the actor outputs logits of the proposal distribution, while the critic outputs action values from which we compute state values.

Our architecture defines a permutation equivariant set-to-set mapping. Similar architectures have been proposed (Zaheer et al., 2017) and studied (Wagstaff et al., 2019) for set data. These models rely on functions of the form $f(\{\mathbf{x}_1, \dots, \mathbf{x}_n\}) = g(h(\mathbf{x}_1), \dots, h(\mathbf{x}_n))$ with h an arbitrary neural network applied to each coordinate $\mathbf{x}_i$ individually, and g a symmetric function (e.g. sum or max operation). It is common to aggregate different functions like f to construct global set representations, e.g. PointNets (Qi et al., 2017), but in our model we found input features encoding global information (e.g. Knapsack capacity) and relative node information (e.g. distance to neighbouring cities in a TSP tour) to be sufficient, and thus we opt for a simple architecture with h an MLP, and g a softmax operation. That maintains the computational complexity of our model linear in the number of items (nodes), allowing us to scale to large problems despite the typically long Markov chains in simulated annealing.

Training We train Neural SA using both Proximal Policy Optimisation (PPO) (Schulman et al., 2017) and Evolution Strategies (ES) (Salimans et al., 2017). We did little to no fine-tuning of hyperparameters and, in fact, kept most of the hyperparameters of both methods the same across all experiments, as detailed below.

- **PPO**: We optimise both actor and critic networks using Adam (Kingma and Ba, 2015) with learning rate of $2\mathrm{e}{-4}$, weight decay of $1\mathrm{e}{-2}$ and $\beta = (0.9, 0.999)$. For PPO, we set the discount factor and clipping threshold to $\gamma = 0.9$ and $\epsilon = 0.25$, respectively, and compute advantages using GAE (Schulman et al., 2016) with trace decay $\lambda = 0.9$.
- **ES**: We use a population of 16 perturbations sampled from a Gaussian of standard deviation 0.05. Updates are fed into an SGD optimizer with learning rate $1\mathrm{e}{-3}$ and momentum 0.9.

Testing The randomly generated datasets used for testing can be recreated by setting the seed of Pytorch's random number generator to 0. Similarly, we evaluate each configuration (problem size, number of steps) 5 times and report the average as well as the standard deviation across the different runs. For reproducibility, we also seed each of these runs (seeds 1, 2, 3, 4 and 5).

B KNAPSACK PROBLEM

Data We consider different problem sizes. We use KNAPN to denote a knapsack problem with N items, each with a weight w_i and value v_i sampled from a uniform distribution, $w_i, v_i \sim \mathcal{U}_{(0:1)}$. Each problem has also an associated capacity C_N, that is, the maximum weight the knapsack in KNAPN can comport. Here we follow Bello et al. (2016) and set C_{50}=12.5, C_{100}=25 and C_{200}=25. However, for larger problems we set $C_N = N/8$.

Initial Solution We start with a feasible initial solution corresponding to an empty knapsack, that is, $\mathbf{x} = \mathbf{0}$. That is the trivial (and worst) feasible solution, so our models do not require any form of initialisation, pre-processing or heuristic.

Training We train *only on* KNAP50 with short rollouts of length $K = 100$ steps. The model is trained for 1000 epochs each of which is run on 256 random problems generated on the fly as described in the previous section. We set initial and final temperatures to $T_0 = 1$ and $T_K = 0.1$, and compute the temperature decay as $\alpha = (T_K/T_0)^{\frac{1}{K}}$.

Testing We evaluate Neural SA on a test set of 1000 randomly generated Knapsack problems, while varying the length of the rollout. For each problem size N, we consider rollouts of length $K = N$, $K = 2N$, $K = 5N$ and $K = 10N$. The initial and final temperatures are kept fixed to $T_0 = 1$ and $T_K = 0.1$, respectively, and the temperature decay varies as function of K, with $\alpha = (T_K/T_0)^{\frac{1}{K}}$.

We compare our methods against one of the dedicated solvers for the knapsack problem in OR-Tools (Perron and Furnon, 2019) (Knapsack Multidimension Branch and Bound Solver). We also compare sampled and greedy variants of Neural SA. The former samples actions from the proposal distribution while the latter always selects the most likely action.

Table 5: ES results on the Knapsack benchmark. Bigger is better. Comparison among rollouts of different lengths: 1, 2, 5 or 10 times the dimension of the problem.

	Greedy	Sampled				OR-Tools
	$\times 1$	$\times 1$	$\times 2$	$\times 5$	$\times 10$	
KNAP50	$16.59 \pm .00$	$19.45 \pm .01$	$19.70 \pm .00$	$19.86 \pm .00$	$19.95 \pm .00$	**20.12**
KNAP100	$31.15 \pm .00$	$39.07 \pm .01$	$39.49 \pm .01$	$39.76 \pm .01$	$39.90 \pm .01$	**40.41**
KNAP200	$55.96 \pm .00$	$53.72 \pm .02$	$55.21 \pm .02$	$56.22 \pm .02$	$56.58 \pm .01$	**57.65**
KNAP500	$135.92 \pm .00$	$134.20 \pm .05$	$137.89 \pm .03$	$140.20 \pm .02$	$141.01 \pm .03$	**144.14**
KNAP1K	$259.20 \pm .00$	$269.21 \pm .04$	$276.48 \pm .05$	$280.94 \pm .02$	$282.46 \pm .03$	**289.01**
KNAP2K	$489.02 \pm .00$	$537.53 \pm .08$	$551.92 \pm .07$	$560.75 \pm .07$	$563.75 \pm .02$	**577.28**

Table 6: PPO results on the Knapsack benchmark. Bigger is better. Comparison among rollouts of different lengths: 1, 2, 5 or 10 times the dimension of the problem.

	Greedy	Sampled				OR-Tools
	$\times 1$	$\times 1$	$\times 2$	$\times 5$	$\times 10$	
KNAP50	$19.52 \pm .00$	$19.37 \pm .01$	$19.42 \pm .01$	$19.55 \pm .01$	$19.69 \pm .01$	**20.12**
KNAP100	$38.97 \pm .00$	$38.64 \pm .01$	$38.81 \pm .01$	$39.20 \pm .01$	$39.54 \pm .01$	**40.41**
KNAP200	$48.58 \pm .00$	$48.99 \pm .06$	$51.00 \pm .04$	$53.57 \pm .03$	$55.03 \pm .01$	**57.65**
KNAP500	$119.38 \pm .00$	$122.40 \pm .03$	$128.46 \pm .05$	$134.95 \pm .03$	$138.14 \pm .04$	**144.14**
KNAP1K	$238.18 \pm .00$	$246.54 \pm .09$	$259.15 \pm .08$	$271.68 \pm .05$	$277.41 \pm .03$	**289.01**
KNAP2K	$472.67 \pm .00$	$493.47 \pm .07$	$519.27 \pm .08$	$543.72 \pm .11$	$554.32 \pm .04$	**577.28**

C BIN PACKING PROBLEM

Data We consider problems of different sizes, with BINN consisting of N items, each with a weight (size) sampled from a uniform distribution, $w_i \sim \mathcal{U}_{(0:1)}$. Without loss of generality, we also assume N bins, all with unitary capacity. Each dataset BINN in Tables 7 and 8 contains 1000 such random Bin Packing problems that are not seen during training and are used to evaluate the methods at test time.

Initial Solution We start from the solution where each item is assigned to a different bin, e.g. $x_{ij} = i$.

Training We train *only on* BIN50 with short rollouts of length $K = 100$ steps. The model is trained for 1000 epochs each of which is ran on 256 random problems generated on the fly as described in the previous section. We keep the same temperature decay with $\alpha = (T_K/T_0)^{\frac{1}{K}}$, but use different initial and final temperatures for PPO and ES. For PPO, we set $T_0 = 1$ and $T_K = 0.1$, whereas for ES we set $T_0 = 0.1$ and $T_K = 1e-4$.

Testing We evaluate Neural SA on a test set of 1000 randomly generated Bin Packing problems, while varying the length of the rollout. For each problem size N, we consider rollouts of length $K = N$, $K = 2N$, $K = 5N$ and $K = 10N$. The initial and final temperatures are kept the same as in training, and the temperature decay parameter varies as function of K, with $\alpha = (T_K/T_0)^{\frac{1}{K}}$.

We compare Neural SA against First-Fit-Decreasing (FFD) (Johnson, 1973), a powerful heuristic for the Bin Packing problem, and against OR-Tools (Perron and Furnon, 2019) MIP solver powered by SCIP (Gamrath et al., 2020). The OR-Tools solver can be quite slow on Bin Packing so we set a time out of 1 minute per problem for BIN50-1000 and of 2 minutes for BIN2000 to match Neural SA running times (see Table 4).

We also compare sampled and greedy variants of Neural SA. The former naturally samples actions from the proposal distribution while the latter always selects the most likely action.

Table 7: ES results on the Bin Packing benchmark. Lower is better.

	Greedy	Sampled				OR-Tools	FFD
	×1	×1	×2	×5	×10		
BIN50	27.62±.00	27.43±.01	27.36±.01	27.29±.00	27.24±.01	**26.71**	27.10
BIN100	53.80±.00	53.63±.00	53.54±.01	53.44±.01	53.38±.01	53.91	**52.91**
BIN200	105.63±.00	105.78±.02	105.64±.01	105.51±.01	105.43±.01	109.19	**104.25**
BIN500	259.09±.00	260.86±.03	260.65±.01	260.42±.02	260.27±.02	267.63	**257.02**
BIN1K	512.66±.00	517.87±.02	517.46±.02	517.08±.02	516.84±.01	1000*	**509.46**
BIN2K	**1017.88** ± .00	1030.66±.01	1029.89±.01	1029.11±.02	1028.67±.02	2000*	1028.67

Table 8: PPO results on the Bin Packing benchmark. Lower is better.

	Greedy	Sampled				OR-Tools	FFD
	×1	×1	×2	×5	×10		
BIN50	27.62 ± .00	27.95 ± .01	27.71 ± .01	27.45 ± .01	27.32 ± .01	**26.71**	27.10
BIN100	53.80 ± .00	54.88 ± .02	54.27 ± .02	53.75 ± .01	53.53 ± .01	53.91	**52.91**
BIN200	105.63 ± .00	108.51 ± .01	107.20 ± .01	106.21 ± .01	105.86 ± .01	109.19	**104.25**
BIN500	259.08 ± .00	268.42 ± .02	264.79 ± .01	262.66 ± .02	261.98 ± .01	267.63	**257.02**
BIN1K	512.66 ± .00	533.97 ± .04	526.23 ± .02	522.30 ± .03	521.22 ± .02	1000*	**509.46**
BIN2K	**1017.88** ± .00	1064.74 ± .11	1048.80 ± .06	1041.02 ± .01	1039.09 ± .04	2000*	1028.67

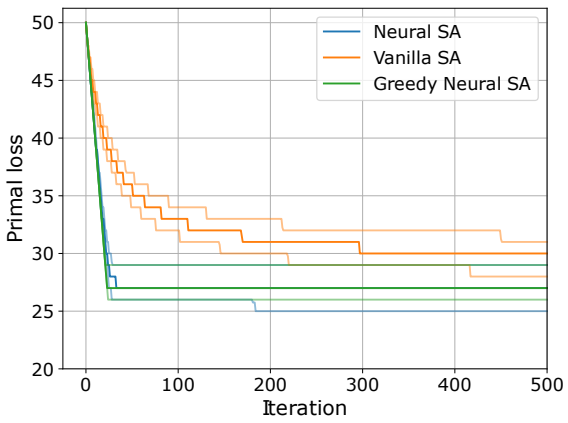


Figure 6: Plot of the BIN50 primal objective comparing convergence speed of Neural SA with vanilla SA and a third option, Greedy Neural SA, which uses argmax samples from the policy; 25th, 50th, and 75th percentiles shown.

D TRAVELLING SALESPERSON PROBLEM (TSP)

Data We generate random instances for 2D Euclidean TSP by sampling coordinates uniformly in a unit square, as done in previous research (Kool et al., 2018; Chen and Tian, 2019; da Costa et al., 2020). We assume complete graphs (fully-connected TSP), which means every pair of cities is connected by a valid route (an edge).

Initial Solution We start with a random tour, which is simply a random permutation of the city indices. This is likely to be a poor initial solution, as it ignores any information about the problem, namely the coordinates of each city. Nevertheless, Neural SA achieves competitive results in spite of this, and it is reasonable to expect an improvement in its performance (at least in running time) when using better initialisation methods, like in LKH-3 (Helsgaun, 2000) for instance.

Training We train *only on* TSP20 with very short rollouts of length $K = 40$. Just like in the other problems we consider, we train using 256 random problems generated on the fly for each epoch. We also maintain the same initial temperature and cooling schedule with $T_0 = 1$ and $\alpha = (T_K/T_0)^{\frac{1}{K}}$, but use lower final temperatures for the TSP. We set $T_K = 1\mathrm{e}{-2}$ for PPO and $T_K = 1\mathrm{e}{-4}$ for ES, which we empirically found to work best with the training dynamics of each of these methods. We also use different number of epochs for each training method, 1000 for PPO and 10000 for ES, as the latter has slower convergence.

Testing We evaluate Neural SA on TSP20, TSP50 and TSP100 using the 10K problem instances made available in (Kool et al., 2018). This allows us to directly compare our methods to previous research on the TSP. We also consider larger problem sizes, namely TSP200 and TSP500 to showcase the scalability of Neural SA. For each of these, we randomly generate 1000 instances by uniformly sampling coordinates in a 2D unit square. For each problem size N, we consider rollouts of length $K = N^2$, $K = 2N^2$, $K = 5N^2$ and $K = 10N^2$. That is different from the other CO problems we study since the complexity in the TSP is related to the number of edges N^2 rather than the number of cities N. We also compare sampled and greedy variants of Neural SA. The former naturally samples actions from the proposal distribution while the latter always selects the most likely action.

We compare Neural SA against standard solvers LKH-3 (Helsgaun, 2000) and Concorde (Applegate et al., 2006), which we have run ourselves. We also compare against the self-reported results of other Deep Learning models that have targeted TSP and relied on the test data provided by (Kool et al., 2018): GCN (Joshi et al., 2019b), GAT (Kool et al., 2018), GAT-T (Wu et al., 2019), and the works of da Costa et al. (2020) and Fu et al. (2021).

Note that Fu et al. (2021) also provide results for TSP200 and TSP500, but given that we do not know the exact test instances they used, it is hard to make a direct comparison to our results, especially regarding running times; they use a dataset of 128 instances, while we use 1000. For that reason, we omitted these results from Table 3 in the main text, but for the sake of completeness, presented them in Table 12.

Generalisation We always train Neural SA only on the smallest of problem sizes we consider. In Table 9, we compare Neural SA with other models in the literature that have been evaluated the same way: trained only on TSP20 and tested on TSP20, 50 and 100. While not outperforming the model by Fu et al. (2021), Neural SA, especially with PPO, does generalise better than previous end-to-end methods (Kool et al., 2018).

Table 9: Optimality gap for models trained on TSP20 and evaluated on the test instances provided by Kool et al. (2018) for TSP20/50/100; *Values taken from respective papers.

	TSP20	TSP50	TSP100
(Kool et al., 2018)*	0.34%	$\sim 5.0\%$	$> 14.0\%$
(Fu et al., 2021)*	0.00%	0.01%	0.04%
SA	1.17%	4.34%	7.43%
Neural SA (PPO)	0.42%	1.16%	1.85%
Neural SA (ES)	0.10%	2.32%	5.50%

Table 10: ES results on the TSP benchmark. Lower is better

	Greedy	Sampled				LKH-3	Concorde
	×1	×1	×2	×5	×10		
TSP20	$3.868 \pm .000$	$3.868 \pm .001$	$3.854 \pm .000$	$3.844 \pm .000$	$3.840 \pm .000$	**3.836**	**3.836**
TSP50	$6.020 \pm .002$	$6.022 \pm .002$	$5.947 \pm .001$	$5.871 \pm .000$	$5.828 \pm .001$	**5.696**	**5.696**
TSP100	$8.659 \pm .003$	$8.660 \pm .002$	$8.477 \pm .001$	$8.298 \pm .002$	$8.191 \pm .002$	**7.764**	**7.764**

Table 11: PPO results on the TSP benchmark. Lower is better

	Greedy	Sampled				LKH-3	Concorde
	×1	×1	×2	×5	×10		
TSP20	$3.864 \pm .000$	$3.865 \pm .000$	$3.850 \pm .000$	$3.841 \pm .000$	$3.838 \pm .000$	**3.836**	**3.836**
TSP50	$5.828 \pm .001$	$5.828 \pm .000$	$5.786 \pm .000$	$5.752 \pm .001$	$5.734 \pm .001$	**5.696**	**5.696**
TSP100	$8.074 \pm .001$	$8.073 \pm .001$	$7.986 \pm .001$	$7.912 \pm .001$	$7.874 \pm .000$	**7.764**	**7.764**
TSP200	$11.41 \pm .00$	$11.41 \pm .00$	$11.23 \pm .00$	$11.09 \pm .00$	$11.00 \pm .00$	**10.70**	**10.70**
TSP500	$18.44 \pm .002$	$18.43 \pm .001$	$18.07 \pm .003$	$17.79 \pm .006$	$17.64 \pm .003$	**16.54**	**16.54**

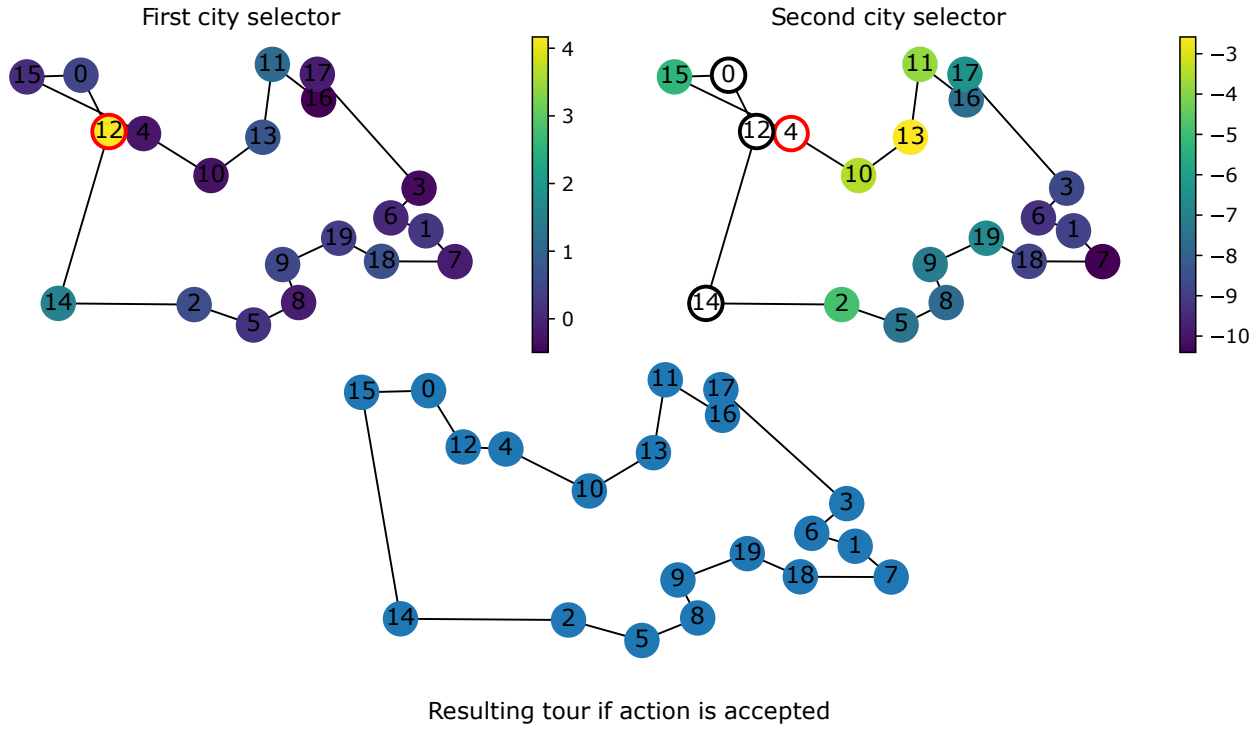


Figure 7: Policy for the Travelling Salesperson Problem. At each step, an action consists of selecting a pair of cities (i, j), one after the other. The figure depicts a TSP problem layed out in the 2D plane, with the learnt proposal distribution over the first city i in the left, and in the right, the distribution over the second city j, given $i = 12$. We mask out and exclude the neighbours of i (0 and 14) as candidates for j because selecting those would lead to no changes in the tour. It is clear the model has a strong preference towards a few cities, but otherwise the probability mass is spread almost uniformly among the other nodes. However, once i is fixed, Neural SA strongly favours nodes j that are close to i. That is a desirable behaviour that even features in popular algorithms like LKH-3 (Helsgaun, 2000). That is because a 2-opt move (i, j) actually adds edge (i, j) to the tour, so leaning towards pairs of cities that are close to each other is more likely to lead to shorter tours.

Table 12: Comparison of different TSP solvers on the 10K instances for TSP20/50/100 provided in (Kool et al., 2018), and 1K random instances for TSP200/500. We report the average solution cost, optimality gap and running time (to solve all instances) for each problem size. We split competing neural methods in two groups: construction heuristics (Kool et al., 2018; Joshi et al., 2019a) and improvement heuristics like Neural SA (Wu et al., 2019; da Costa et al., 2020; Fu et al., 2021). *Values as reported in the corresponding paper. †Different test data.

	TSP20			TSP50			TSP100			TSP200			TSP500		
	Cost	Gap	Time	Cost	Gap	Time	Cost	Gap	Time	Cost	Gap	Time	Cost	Gap	Time
CONCORDE (Applegate et al., 2006)	3.836	0.00%	48s	5.696	0.00%	2m	7.764	0.00%	7m	10.70	0.00%	38m	16.54	0.00%	7h58m
LKH-3 (Helsgaun, 2000)	3.836	0.00%	1m	5.696	0.00%	14m	7.764	0.00%	1h	10.70	0.00%	21m	16.54	0.00%	1h15m
OR-TOOLS (Perron and Furnon, 2019)	3.86	0.85%	1m	5.85	2.87%	5m	8.06	3.86%	23m	-	-	-	-	-	-
SA	3.881	1.17%	10s	5.943	4.34%	37s	8.343	7.45%	3m	11.98	11.87%	9m	20.22	22.25%	56m
GENG ET AL. (Geng et al., 2011)	3.844	0.21%	46m	5.814	2.07%	68m	8.156	5.06%	131m	11.74	9.72%	33m	20.00	20.93%	3h29m
NEURAL SA PPO	3.837	0.02%	17s	5.727	0.54%	1m	7.856	1.18%	9m	10.96	2.50%	15m	17.64	6.65%	2h16m
NEURAL SA ES	3.840	0.10%	10s	5.828	2.32%	1m	8.191	5.50%	9m	11.74	9.72%	15m	20.27	22.55%	2h16m
GCN Greedy (Joshi et al., 2019a)*	3.86	0.60%	6s	5.87	3.10%	55s	8.41	8.38%	6m	-	-	-	-	-	-
GCN Beam Search (Joshi et al., 2019a)*	3.84	0.01%	12m	5.70	0.01%	18m	7.87	1.39%	40m	-	-	-	-	-	-
GAT Greedy (Kool et al., 2018)*	3.85	0.34%	0s	5.80	1.76%	2s	8.12	4.53%	6s	-	-	-	-	-	-
GAT Sampling (Kool et al., 2018)*	3.84	0.08%	5 m	5.73	0.52%	24m	7.94	2.26%	1 h	-	-	-	-	-	-
GAT-T {1000} (Wu et al., 2019)*	3.84	0.03%	12m	5.75	0.83%	16m	8.01	3.24%	25m	-	-	-	-	-	-
GAT-T {3000} (Wu et al., 2019)*	3.84	0.00%	39m	5.72	0.34%	45 m	7.91	1.85%	1 h	-	-	-	-	-	-
GAT-T {5000} (Wu et al., 2019)*	3.84	0.00%	1 h	5.71	0.20%	1 h	7.87	1.42%	2 h	-	-	-	-	-	-
(da Costa et al., 2020) {500}*	3.84	0.01%	5m	5.72	0.36%	7m	7.91	1.84%	10m	-	-	-	-	-	-
(da Costa et al., 2020) {1000}*	3.84	0.00%	10m	5.71	0.21%	13m	7.86	1.26%	21 m	-	-	-	-	-	-
(da Costa et al., 2020) {2000}*	3.84	0.00%	15m	5.70	0.12%	29m	7.83	0.87%	41m	-	-	-	-	-	-
Att-GCRN+MCTS (Fu et al., 2021)*	3.84	0.00%	2m	5.69	0.01%	9m	7.76	0.03%	15m	10.81†	0.88%†	3m†	16.96†	2.96%†	6m†