ISSN 2281-4299

DIPARTIMENTO DI INGEGNERIA INFORMATICA AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Heuristics for the Traveling Salesperson Problem based on Reinforcement Learning

Corrado Coppola
Giorgio Grani
Marta Monaci
Laura Palagi

Technical Report n. 04, 2021

Heuristics for the Traveling Salesperson Problem based on Reinforcement Learning

Corrado Coppola[1], Giorgio Grani[2], Marta Monaci[1], and Laura Palagi[1]

[1]Department of Computer, Control and Management Engineering, Sapienza University of Rome, Rome, Italy
[2]SINTEF Digital, Forskningveien 1, Oslo, Norway

Abstract

We use Reinforcement Learning, in particular a deep Q-Learning algorithm and an adaptation of two actor-critic algorithms (Proximal Policy Optimization and Phasic Policy Gradient), originally proposed for robotic control, to solve the metric Traveling Salesperson Problem (TSP). We introduce a convolutional model to approximate action-value and state-value functions, centered on the idea of considering a weighted incidence matrix as the agent's graph representation at a given instant. Our computational experience shows that Q-Learning does not seem to be adequate to solve the TSP, but nevertheless we find that both PPO and PPG can achieve the same solution of standard optimization algorithms with a smaller computational effort; we also find that our trained models are able to orient themselves through new unseen graphs and with different costs distributions.

Keywords: traveling salesperson problem; reinforcement learning, heuristic

1 Introduction

Firstly formulated more than 90 years ago by Karl Menger in [13], the Traveling Salesperson problem (TSP) is still one of the most challenging combinatorial problems in optimization. While being initially thought as the problem of finding the shortest route to visit each city in a given list, the TSP includes several logistical, industrial, and technological applications, whose solution can play a significant role in determining operational and economical performances (see [9] for a more detailed description).
The metric TSP, a sub-class of TSP characterized by costs on arcs satisfying the triangle inequality, is proved to be NP-complete by Papadimitriou in [18] and, as proved by Bellman in [3], any dynamic programming algorithm capable of achieving the exact optimal solution would have $\Theta(n2^n)$ complexity. Although exact algorithms with lower complexity have been proposed, such as the branch-and-search of Xiao et al. in [23], worst-case computational time grows exponentially with the number of nodes and thus most of the research activities have been oriented towards the development of new and more effective heuristics, see [22] for a detailed analysis on different algorithms. Concerning exact solution methods, Padberg and Rinaldi have studied the TSP from a mathematical programming perspective, providing some of the most relevant results in this field. In[16], they achieved an exact solution of a 532-nodes symmetric TSP instance, whereas in [17], employing a cutting-plane procedure within a Branch-and-Bound framework, they generated a subset of all the inequalities defining the convex hull of the incidence vectors of all the Hamiltonian cycles on a given graph, being able to solve to optimality TSP instances up to 3000 nodes. More recently, McMenemy et al. proposed in [12] a decision tree model to identify the best performing algorithm according to the instance features. In spite of the notable achieved progress, traditional heuristics build a feasible solution by incrementally adding single elements, making thus algorithm choices deterministic and irreversible. In 1987 Filar et al. proposed in [11] a mathematical embedding of the TSP into a Markov decision process and then introduced for the first time the idea of considering the TSP as a stochastic control problem, where the traveling salesperson (the agent) pursues the goal of finding the cheapest route to visit all the nodes in a graph by moving from a node to another according to a given probability distribution (i.e taking actions).
Recently, machine learning-based heuristics to solve hard combinatorial optimization problems have gained an increasing interest in the community. Among the hundreds of different techniques applicable, Reinforcement Learning (RL) is emerging as a natural counterpart to traditional methods. From the optimization perspective, RL can be seen as an approximate version of dynamic programming (see [6]), whereas from the machine learning point of view, it is acknowledged to be the third learning paradigm alongside Supervised and Unsupervised Learning (see [10]). Reinforcement Learning (RL) has been increasingly used as a flexible and effective framework to solve hard combinatorial problems; a recent example is given by Barrett et al. in [1], where RL-based heuristics are adopted to solve the MaxCut problem. In particular, Deep RL algorithms use neural networks to approximate functions, whose values are needed to train the agent and to make it learn a stochastic policy (see [21] for extended description). Concerning the use of deep networks for TSP, one of the first experiments is provided by Vinyals et al. in [15]. They address the general problem of estimating the conditioned probability of an output sequence (a sequence of arcs, in the case of TSP), whose elements are discrete tokens correlated with the input sequence positions; for the purpose of learning this probability, they introduce

a new neural attention mechanism, called Pointer Network (Ptr-Net). Their computational experience on TSP instances with up to 20 nodes shows that Ptr-Net provides almost the same Christofides solution. Differently from the latter, Bello et al. developed in [4] a policy gradient algorithm, adopting Recurrent Neural Networks (RNNs) to approximate state-value and action-value functions. Beyond specific research for TSP, Mnih et al. in [14] proposed a general deep Q-Learning algorithm, while Schulman et al. developed in [19] a significantly new actor-critic algorithm, the Trust Region Policy Optimization (TRPO), which is proved to achieve monotone policy improvements by iteratively solving a constrained optimization problem.
This paper aims at assessing and comparing performances of the TSP solution of three state-of-art RL algorithms. We describe the TSP as a Markov's Decision Process, where the state of the graph at each step is described by a dense matrix. We decided to deal with such input data adopting convolutional neural networks (CNNs) as estimators. In fact, CNNs are characterized by the use of a discrete convolution operator and parameters sharing trough a grid-type structure, we direct the reader to [5] for a detailed description.
The paper is organized as follows: after a brief introduction to TSP in section (2), and to Reinforcement Learning in section (3), we describe an adaptation of deep Q-Learning applied to TSP in section (4) and in section (5) the adaption of two actor-critic algorithms, that had been originally developed for robotic control by Cobbe et al. (PPG in [8] and PPO in [20]). In section (6), we explain in details our convolutional models and finally in section (7), we present and discuss the computational experience.

2 The Traveling Salesperson Problem as a Markov Decision Process

Given a connected graph $G(N, E)$, the TSP is a NP-hard combinatorial optimization problem tasked with finding the cheapest route to visit all nodes in N.
In the following of the paper, we will assume that the graph $G(N, E)$ is directed and complete, i.e. $\forall\ v, w \in N : v \neq w\ \exists\ vw, wv \in E$. The arc costs are assumed to be non-negative and to satisfy the triangle inequality, i.e. $\forall\ \{u, v, w\} \subseteq N$ it holds $c_{uw} \leq c_{uv} + c_{vw}$.
According to what stated above, the problem is thus solved when the cheapest Hamiltonian cycle on $G(N, E)$ is found. Completeness and triangle inequality do not affect model generality: in fact, given a connected yet non-complete graph, it is possible to build up a new complete graph setting the cost of each arc (u, v) to be equal to the value of the shortest path from u to v.

An MDP is a stochastic control process characterized by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, p)$, where $\mathcal{S}$ is the set of all possible states, $\mathcal{A}$ is the set of all possible actions, $\mathcal{R}$ is the set of rewards and p is the transition probability from state s_t to s_{t+1} defined as $p(s_{t+1}, r_t) = P(s_{t+1}, r_t | S_t = s_t, \mathcal{A}_t = a_t)$, varying at time t.
In a finite-horizon and discrete-time MDP, the agent selects at each time-step $t = 1, \ldots, T$ feasible actions from the finite set $\mathcal{A}$ in order to maximize the cumulative reward returned by the system. TSP can be cast into a finite-horizon and discrete-time Markov Decision Process (MDP), where the traveling salesperson is our learning agent, which can take an action at each time step t, deciding where to go from its current position and then obtaining a reward r_t representing the paid price to move from a node to another one.

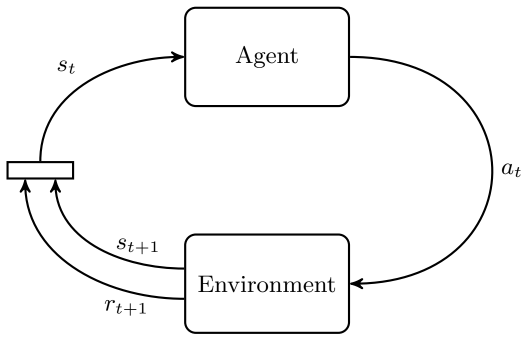


Figure 1: Basic agent-environment interaction

For the TSP, given the graph $G(N, E)$, the state s_t is the weighted incidence matrix of the subgraph $H(V_t, E_t)$ obtained removing from $G(N, E)$ the already visited nodes and corresponding arcs at time step t. Since the graph is complete, the set of action can be set as $\mathcal{A} = N$, namely the node that can be visited, and at every time step t we have $\mathcal{A}_t = V_t \setminus \{w\}$, being $w \in N$ the agent position at t. The agent pursues the maximization of the reward, that has to represent the price paid by the traveling salesperson to move from node u to node v. Hence, the reward is set to the opposite of the cost of the arc uv; hence, the set of all possible rewards is $\mathcal{R} = \{-c_{uv}, \forall\ uv \in E\}$. The transition function $p(s_{t+1}, r_t)$ is deterministic, meaning that given a state s_t, the current position $w \in N$ of the agent and the chosen action $a_t = u$, the system returns the reward $r_t = -c_{wu}$ and the new state s_{t+1} with probability 1.

3 Basics of Reinforcement Learning

Reinforcement Learning (RL) is a machine learning paradigm concerned with training an agent to take actions in a defined environment in order to maximize the expected average reward. As a form of approximated dynamic programming (see [6] and [7]), RL environments can be easily considered as Markov's Decision Processes (MDPs), where an agent, which in our case is the traveling salesperson, interacts with a surrounding environment (the graph) by taking actions (move from a node to another one) and receiving consequent rewards (the arc cost). Figure 1 provides a schematic representation of this interaction.
In finite-horizon and time-discrete problems, such as TSP, the final goal of the agent is to maximize the total reward accumulated trough the process, depicting an optimal strategy. To do so, throughout a determined number of discrete time steps $t = 1, 2, \ldots, N$, the agent interacts with the environment by taking actions and collecting their effects, eventually learning from this experience to improve its strategy. More formally, the agent objective is to maximize the following:

$$J = \mathbb{E}\left[\sum_{k=0}^{N-t} \gamma^k r_{t+k}\right] \tag{1}$$

where $\gamma \in (0, 1)$ is a discount factor.
As J is stochastic and depending on the agent strategy, (1) cannot be maximized using standard optimization algorithms. Hence, the problem can be reformulated in terms of learning a prof-

itable policy π, that is a discrete probability distribution $\pi(a_t|s_t) = P(\mathcal{A}_t = a_t|\mathcal{S}_t = s_t)$ which returns high probability values for the actions that are more likely to produce a better long-run reward. However, to pursue its goal, the agent should know which is the expected average reward associated to a specific policy at a given time step t, that is to know the consequences of the taken actions in terms of reward. For this purpose, given a policy π the following functions are defined.

- The state-value function

$$V_\pi(s_t) = \mathbb{E}_\pi \left[\sum_{k=0}^{N-t} \gamma^k r_{t+k} | \mathcal{S}_t = s_t \right] \qquad \forall s_t \in \mathcal{S} \tag{2}$$

- The action-value function

$$Q_\pi(s_t, a_t) = \mathbb{E}_\pi \left[\sum_{k=0}^{N-t} \gamma^k r_{t+k} | \mathcal{S}_t = s_t, \mathcal{A}_t = a_t \right] \qquad \forall\ (s_t, a_t) \in (\mathcal{S} \times \mathcal{A}) \tag{3}$$

- The advantage estimation function

$$A(s_t, a_t) = r_t + V_\pi(s_{t+1}) - V_\pi(s_t) \tag{4}$$

As highlighted by the notation, all the functions above are expected values computed according to a specific stochastic trajectory given by the policy π. In RL algorithms the current policy $\pi(\cdot|s_t)$ changes at each time-step and this makes computationally too heavy to evaluate the exact statistical values, which are instead replaced by some approximations.
Different RL algorithms provide different approximated strategies to make the agent learn a policy. While using the action-value function to look ahead for a defined number of time steps and to take a decision is typical of deep Q-Learning strategies, Actor-Critic algorithms perform parallel training of a parametrized policy (the actor) $\pi(\cdot|a; \theta)$ and of an approximated state-value function (the critic) $\widehat{V_\pi}(s_t)$.
Deep RL algorithms use neural networks to approximate both the policy and the functions (3) and (2). We will further use the notation θ to indicate the parameters vector of our neural networks. In particular we will refer to the actor parameters and critic parameters respectively with θ_p and θ_v.
In the computational experience, the traveling salesperson training has been performed firstly using Q-Learning, then two different Actor-Critic algorithms, Proximal Policy Optimization (PPO) and Phasic Policy Gradient (PPG).

4 Deep Q-Learning algorithm

The underlying idea of Q-Learning is to use the action-value function to select an action at a given time step. Q-Learning algorithms determine the optimal action a_t^* according to the rule $a_t^* = \arg\max_{a_t \in \mathcal{A}_t} q(s_t, a_t; \theta)$, where $q(s_t, a_t; \theta)$ is an estimator of the optimal state-value function $q^*(s_t, a_t)$.

The training process is performed using a loss function based on the following identity, known as Bellman equation (see [2]).

$$q^*(s_t, a_t) = \mathbb{E}_\pi \left[r_{t+1} + \gamma \max_{a_{t+1} \in \mathcal{A}_{t+1}} q^*(s_{t+1}, a_{t+1}) \right]$$

The loss function is, indeed, the MSE between the approximation of the left member of Bellman equation returned by the network and the right member sampled from past iterations. To facilitate exploration, a ϵ-greedy strategy is usually employed, meaning that an action is chosen randomly with probability ϵ and according to the Bellman equation otherwise. The structure of the method is reported in Algorithm (1).

Algorithm 1: Q-Learning for TSP

```
Initialize $\theta_0$, $\epsilon_0$, $\eta \in (0,1)$, $\bar{\epsilon} > 0$ and empty buffer $B$ with capacity $C_B$
for episode i=0, ..., N do
    Initialize $s_0$
    for time-step t=0, ..., T do
        Select action $a_t = \arg\max_{a_t} q^*(s_t, a_t; \theta_i)$ with probability $1 - \epsilon_i$ or a random action
          with probability $\epsilon_i$
        Execute action $a_t$ and get the reward $r_t$ and the new state $s_{t+1}$
        Store $(s_t, a_t, r_t, s_{t+1})$ inside $B$
        Sample a random tuple $(s_j, a_j, r_j, s_{j+1})$ from B
        Set $y_j = r_j$ if $s_{j+1}$ is terminal, $y_j = r_j + \gamma \max_a q(s_{j+1}, a; \theta)$ otherwise
        Perform a gradient descend step on $MSE(y_j, q(s_j, a_j; \theta))$ w.r.t. $\theta$
    end
    Set $\epsilon_{i+1} = \max(\eta\epsilon_i, \bar{\epsilon})$
end
```

5 Actor-Critic algorithms

Differently from Q-learning, which takes advantage of the Bellman equation to optimize, Actor-Critic algorithms are a class of methods operating on the policy (called actor) both directly and through the influence of a value network (called critic). In Deep RL actor and critic are two deep neural networks with parameters vectors θ_p and θ_v; as shown in figure 2, they respectively return the stochastic policy $\pi(\cdot|s)$ and an approximation of the state-value function $\widehat{V}_\pi(s)$. The underlying idea is to use the approximate gradient of (1) to update the policy parameters θ_p. In particular, it is well known that (1) can be rewritten as

$$\nabla_\theta J(\theta) \propto \sum_{a_t} \nabla_\theta \log \pi(a_t|s_t, \theta)(r_t + V(s_{t+1}) - V(s_t)) \tag{5}$$

where $\propto$ denotes linear dependence. As the effect of any proportionality constant can be absorbed by an appropriate choice of the step-size α, we can define the following update rule:

$$\theta_{new} = \theta_{old} + \alpha \widehat{\mathbb{E}}_t \left[\frac{\nabla_\theta \pi(a_t|s_t, \theta)}{\pi(a_t|s_t, \theta)} \widehat{A}_t(s_t, a_t) \right] \tag{6}$$

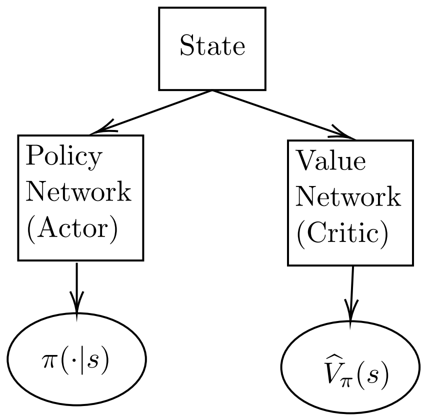


Figure 2: Actor-Critic scheme

Where $\widehat{A}_t(s_t, a_t)$ is the estimate of the advantage function, that is directly computed from the critic output, i.e. using the definition (4) and replacing $V_\pi(\cdot)$ with the approximation provided by the value network described in Section 6.
However, computational experiments carried out by Schulman et al. in [20] suggest that the gradient estimator

$$\hat{g} = \widehat{\mathbb{E}}_t \left[\frac{\nabla_\theta \pi(a_t|s_t, \theta)}{\pi(a_t|s_t, \theta)} \widehat{A}_t(s_t, a_t) \right]$$

leads to excessively wide policy updates and it is thus not directly usable to maximize the (1). In fact, as suggested by the notation $\widehat{\mathbb{E}}_t[\cdot]$, it is an empirical average over a finite number of time steps and, although the policy is assumed to be ergodic as a consequence of Markov's property, the approximation error could nevertheless remain unacceptable for too small time horizons. To solve this problem Schulman et al. proposed in [19] the Trust Region Policy Optimization (TRPO), a method that performs constrained maximization of the objective function within a trust region, blocking policy updates above a certain threshold. Update size was there controlled by the average Kullback-Leibler divergence between the current policy and the new one.
A schematic representation of policy and value networks roles in the following algorithms is given in figure 2.
We will further use the notation θ_p and θ_v to refer respectively to policy network and value network parameters.

5.1 Proximal Policy Optimization

The Proximal Policy Optimization (PPO) attempts to solve the problem behind the TRPO approach by dealing directly with the unconstrained formulation, which uses the Kullback-Leibler divergence as a penalty term in the objective function. Cobbe et al. in [20] find that the best results are obtained when feasible policy updates are explicitly clamped in a defined interval.

Being $r_t(\theta) = \dfrac{\pi(a_t|s_t;\theta)}{\pi(a_t|s_t;\theta_{old})}$, the new proposed objective function is the following:

$$L_{CLIP}(\theta_p) = \widehat{\mathbb{E}}_t \left[\min(r_t(\theta_p)\widehat{A}_t, \texttt{clip}(r_t(\theta_p), 1+\epsilon, 1-\epsilon)\widehat{A}_t\right] \tag{7}$$

Where it holds:

$$\text{clip}(r_t(\theta_p), 1+\epsilon, 1-\epsilon) = \begin{cases} r_t(\theta_p) & \text{if } r_t(\theta_p) \in [1-\epsilon, 1+\epsilon] \\ 1-\epsilon & \text{if } r_t(\theta_p) < 1-\epsilon \\ 1+\epsilon & \text{if } r_t(\theta_p) > 1+\epsilon \end{cases} \tag{8}$$

The probability ratio $r_t(\theta)$, which measures the dimension of current policy update, is here blocked in the interval $[1-\epsilon; 1+\epsilon]$.
While policy network training is directly performed on (7), the value network is trained with a MSE loss, using the target values obtained during policy roll-outs as follows.

$$L_{value}(\theta_v) = \frac{1}{2}\widehat{\mathbb{E}}_t \left(V_{\theta_v}(s_t) - \widehat{V}_t^{target}\right)^2 \tag{9}$$

The PPO algorithm is presented in Algorithm 2.

Algorithm 2: Proximal Policy Optimization

```
Initialize $\theta_v, \theta_p$ and empty buffer $B$
for iteration i=1, ..., N do
    for rollout j = 1, ..., K do
        Complete the episode by picking a actions $a_t \sim \pi^i_{\theta_p}$
        Compute $\hat{A}_t$ for all time steps in the instance
        Store $V_{\theta_v}(s_t)$, the rewards $r_t$ and $\hat{A}_t$ in the buffer $B$
    end
    for epoch=1, ..., E do
        Optimize $L_{CLIP}(\theta_p)$ wrt $\theta_p$ in mini-batches on dataset $B$
        Optimize $L_{value}(\theta_v)$ wrt $\theta_v$ in mini-batches on dataset $B$
    end
    Store policy performances
    Update current policy
    Clear buffer $B$
end
```

5.2 Phasic Policy Gradient

Phasic Policy Gradient (PPG) is an extension of PPO performing the training in two phases at each step, called *policy phase* and *auxiliary phase* respectively. In the first part, the agent is trained according to the PPO as in Algorithm 2, while in the second, the value and the policy networks are updated using two auxiliary loss functions, with the aim of reducing variance and to speed up the convergence rate.

As in the PPO, the value network is trained according to (9), whereas the formula in (7) is adjusted by adding an entropy term.

$$L_{policy} = L_{CLIP} + \beta S[\pi] \tag{10}$$

Being $S[\pi]$ the cross entropy of the stored policy π_{old} with respect to the current one and $\beta > 0$. The auxiliary training is performed using the formula in (9) for the value network, and the following loss function for the policy network:

$$L_{joint} = L_{value} + \beta_C \widehat{\mathbb{E}}_t \left[KL[\pi(.|s_t, \theta_{p,old}), \pi(.|s_t, \theta_p)] \right] \tag{11}$$

The structure of PPG is reported in Algorithm 3.

Algorithm 3: Phasic Policy Gradient for TSP

```
Initialize $\theta_v, \theta_p$ and empty buffer $B$
for phase = 1, ..., $N_{ph}$ do
    for policy iteration i=1, ..., $N_\pi$ do
        for rollout j = 1, ..., $K$ do
            Complete the episode by picking a actions $a_t \sim \pi^i_{\theta_p}$
            Compute $\hat{A}_t$ for all time steps in the instance
            Store $V_{\theta_v}(s_t)$, the rewards $r_t$ and $\hat{A}_t$ in the buffer $B$
        end
        for policy epoch=1, ..., $E_p$ do
            Optimize $L_{policy}(\theta_p)$ wrt $\theta_p$ in mini-batches on dataset $B$
        end
        for value epoch=1, ..., $E_v$ do
            Optimize $L_{value}(\theta_v)$ wrt $\theta_v$ in mini-batches on dataset $B$
        end
        Store policy performances
    end
    for auxiliary epoch=1, ..., $E_{aux}$ do
        Optimize $L_{joint}(\theta_p)$ wrt $\theta_p$ in mini-batches on dataset $B$
        Optimize $L_{value}(\theta_v)$ wrt $\theta_v$ in mini-batches on dataset $B$
    end
    Update current policy
    Clear buffer B
end
```

6 The convolutional models

Approximation of action-value $\widehat{A}$ and state-value $\widehat{V}$ functions has been performed with two different models involving convolutional deep neural networks (CNNs), see [5] for a detailed definition.

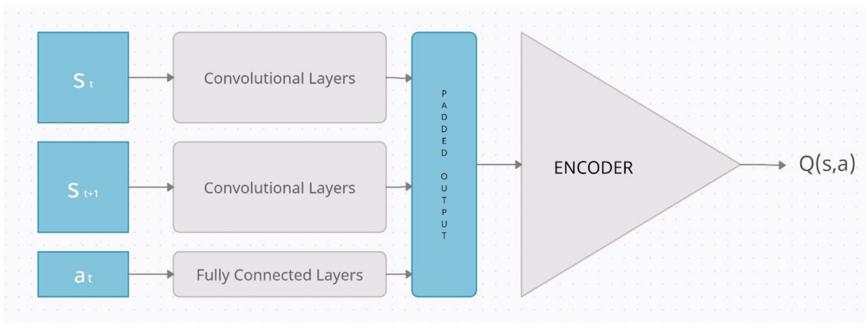


Figure 3: Convolutional Model for Q-Learining

In the proposed TSP representation as a Markov's decision process, the state is a highly dense matrix of dimension $n \in [2, d]$, being d the instance dimension. Since our goal is to build up models able to generalize to a certain extent, the flexibility to the input size is a key factor when choosing the network structure. For their ability to process matrix of different sizes, CNNs appeared to be the best choice compared to other architectures. Once fixed the number of channels, CNNs employ a discrete convolution operator on the input features by looking at a fixed neighborhood on a grid data structure. Despite other frameworks have been proposed in literature on sparse graph structures, like graph convolutional neural networks (see [24]), the density of the state matrices makes ineffective any attempt to embed costs data in neighborhood statistics. In fact, according to the completeness assumption, the neighborhood of each node is the entire graph.
Preliminary experiments showed that the weighted incidence matrix could be insufficient to identify a state, since the information concerning the current position is missing. This can be solved in two ways, either by defining a moving label that points at the state-matrix row of the current node, or by reshaping the state-matrix such that the current node row is always fixed in the first index position. Our computational experience suggests that both solutions speed up the training process, with the second one requiring less memory allocation.

6.1 Q-Learning model

In the Q-Learning algorithm, actions are taken according to their value at a given time step. In the TSP environment, given a state s_t and taken an action a_t then r_t and s_{t+1} are deterministically determined. Indeed, being $u \in N$ the current agent position and having chosen the arc $uv \in E$, the traveling salesperson will certainly receive a reward $-c_{uv}$ and end up in node v. The first source of experience is thus the sequence state-action-next state, which is the triple input of the model in figure 3.
As shown in the figure, while state matrices go through convolutional layers of two different networks, the one-hot encoded action is sent into a fully connected network with two hidden layers and ReLU activation functions. The output of the three networks is pooled and padded with zeros to reach the fixed maximum dimension of the model. The padded result goes eventually through a multi-layer encoder, which returns the approximation of the action-value function $\hat{Q}_\pi(s_t, a_t)$. According to the current policy, it is then taken the action $a_t^* = \arg\max_{a_t \in \mathcal{A}_t} \hat{Q}_\pi(s_t, a_t)$.

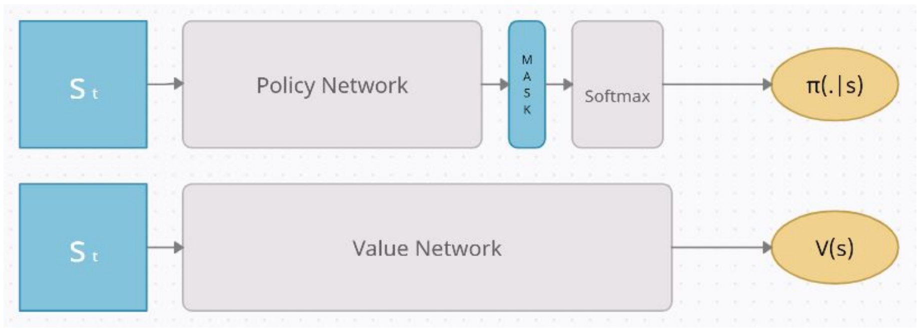


Figure 4: Convolutional Model for PPO and PPG

6.2 AC Model

As stated before in section 5, Actor-Critic algorithms rely on two different networks, one to estimate of $V_\pi(s_t)$ and the other for $\pi(\cdot|s_t)$.
In the model proposed for algorithms 2 and 3, policy and value networks are made of two convolutional layers and two fully connected layers with ReLU activation functions. While the value network directly returns the numerical approximation of $V_\pi(s_t)$, policy network output is filtered with a feasibility mask at time step t (i.e. the set $\mathcal{A}_t$ of all available actions at t) and then with a SoftMax layer.
A schematic representation of the model is provided in figure 4.

7 Computational experience

Experiments on random TSP instances have been carried out to evaluate the training performances and generalization capability of the models. The benchmark used for model evaluation is the open-source suite for optimization *OR-TOOLS* (`https://developers.google.com/optimization`).
The training datasets have been created exploiting the triangle inequality assumption according to the following procedure. Given a maximum instance dimension D, exactly D 2-dimensional vectors are sampled from uniform distribution $\mathcal{U}_{[0,100]\times[0,100]}$. The nodes coordinates set $\{x_1, \ldots, x_D\}$ is hence used to build the weighted incidence matrix by fixing $c_{ij} = c_{ji} = ||x_i - x_j||$, i.e. the euclidean distance between x_i and x_j. Uniformly distributed positions are to represent an environment with a constant node concentration, which does not necessarily fit with most TSP applications. However, training on this type of graphs highlights the real capability of the model to learn from the whole state-matrix more than from local structures recurring in single instances. We have tested the generalization properties of the learned models in two ways:

1. changing the test distribution
2. increasing the size of the test instances compared to the training set.

Training an agent with a RL algorithm leads to the well-known trade-off between exploration of new policies and exploitation of the knowledge achieved so far. It is thus necessary to introduce a new hyper-parameter (called $\epsilon > 0$) to prevent the salesperson from getting stuck in unprofitable

policies. We name ϵ the probability for the agent to take a random action, that is the probability of not following the current policy at a given time step. In the Q-Learning algorithm, we update ϵ at each iteration according to an exponentially decaying function with a fixed threshold (between 0.1 and 0.01) as a lower bound. In AC algorithms 2 and 3, ϵ is updated according to a step function, and pushed to zero for the final part of the training.
For each trained model with different hyper-parameters setting, tests have been carried out on 100 TSP instances with six different nodes configurations:

C1 Node positions uniformly distributed as in the training set, i.e. $x_i \sim \mathcal{U}_{[0,100]\times[0,100]}$, see figure 5a

C2 Node positions uniformly distributed with different parameters to assess model capability to detect and to exploit a more concentrated structure. It is set $x_i \sim \mathcal{U}_{[a,b]\times[c,d]}$, being $[a,b]\times[c,d] \subset [0,100]\times[0,100]$

C3 Node positions sampled from a normal distribution $\mathcal{N}_{(\mu,\sigma)}$ to create extremely concentrated configurations (see figure 5b) with some outliers and to check, whether the trained agent is able to optimize his visits to this type of nodes

C4 Node positions distributed alongside a narrow curve with an additive Gaussian error, i.e. $x_i = p_i + \xi_i$, being p_i sampled from $\mathcal{U}_{[a_i,b_i]\times[c_i,d_i]}$ and $\xi_i \sim \mathcal{N}_{(\mu=0,\sigma^2=10)}$, where it holds $[a_i,b_i]\times[c_i,d_i] \subset [0,100]\times[0,100]$ and parameters a_i, b_i, c_i, d_i are chosen according to the selected curve. This configuration, reported in figure 5d, is to verify model capability to route the traveling salesperson following the natural nodes disposition

C5 Node positions asymmetrically distributed with the presence of two main clusters in the graph as in figure 5c. It is set $x_i = \delta_i u_{1i} + (1-\delta_i)u_{2i} + \xi_i$, being $u_{1i} \sim \mathcal{U}_{[0,20]\times[0,20]}$ and $u_{2i} \sim \mathcal{U}_{[80,100]\times[80,100]}$ and $\delta_i \sim \mathcal{U}_{\{0,1\}}$ and $\xi_i \sim \mathcal{N}_{(\mu=0,\sigma^2=10)}$

C6 Node positions uniformly distributed as in the training set but different graph size

For all experiments the following metrics have been used to evaluate computational time and reward performance:

- The average normalized reward over the 100 test TSP instances

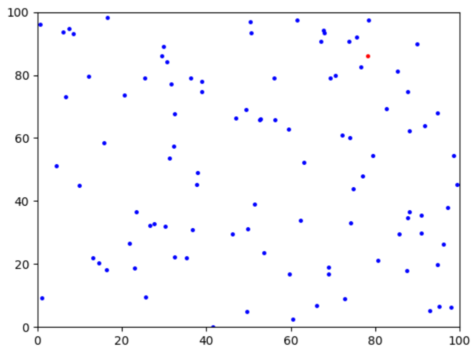
$$\bar{r} = \frac{1}{100}\sum_{i=1}^{100}\frac{r_i^*}{r_i} \tag{12}$$

being r_i and r_i^* respectively the average normalized reward on 10 roll-outs returned by the model and the one returned by *ORTOOLS* solver.

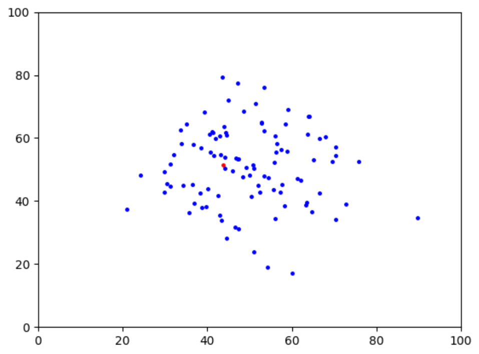
- The average normalized time over the 100 test TSP instances

$$\bar{t} = \frac{1}{100}\sum_{i=1}^{100}\frac{t_i^*}{t_i} \tag{13}$$

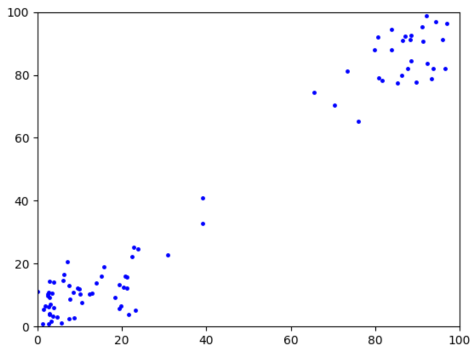
being t_i and t_i^* respectively the average computational time needed by the model and by *ORTOOLS* solver.



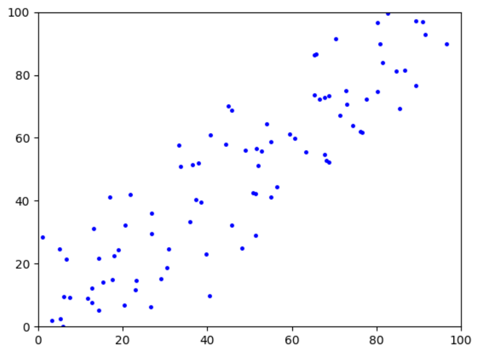
(a) Uniform configuration



(b) Normal configuration



(c) Cluster configuration



(d) Curve configuration

Figure 5: Different types of nodes configuration.

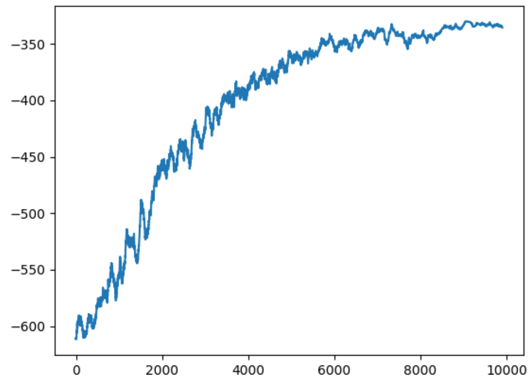


Figure 6: Successful training

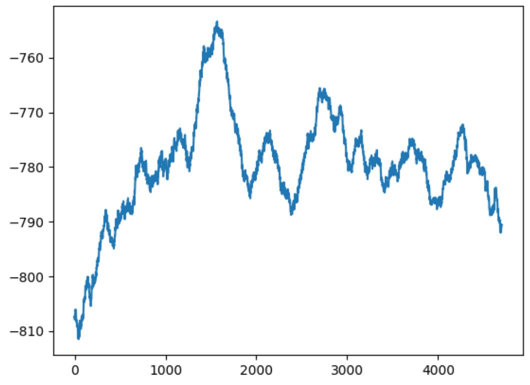


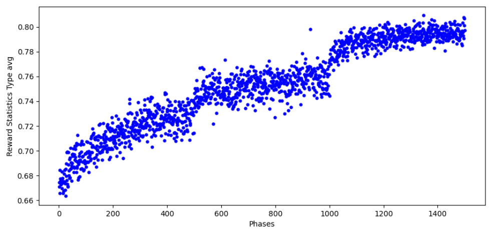
Figure 7: Unsuccessful training

Configuration	**Avg norm. reward**	**Avg norm. time**
C1	0.43	0.12
C3	0.31	0.13
C4	0.36	0.12
C5	0.28	0.14

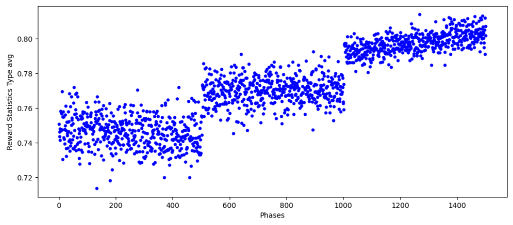
Table 1: Test performance of Q-Learning algorithm

Algorithm 1 turned to be generally ineffective even in the training process and too sensitive to hyper-parameters setting, in particular to the choice of η and of the probability threshold $\bar{\epsilon}$. We provide below two examples of average reward evolution during the train on the same data-set with minimally different hyper-parameters settings. While in the first case the score was regularly increasing (see figure 6), in the second one, after not significant improvement, it started decreasing (see figure 7), as the traveling salesperson was learning from his past wrong actions.
For the sake of completeness, we also report test performances in table 1, where the average normalized reward on 100 instances with different node configurations is shown.

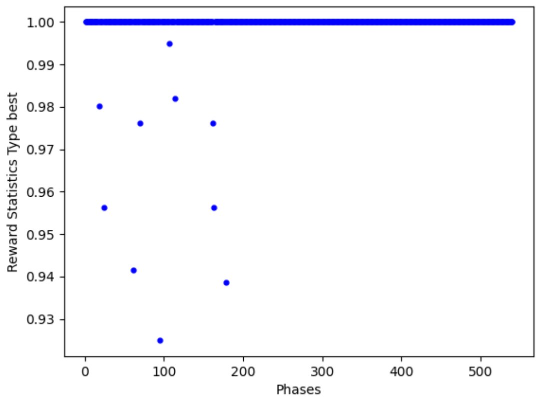
PPO and PPG turned to be effective both in the training and in the test processes. Due to the presence of auxiliary training, PPG requires generally more computational effort and is more sensitive to the hyper-parameters setting. Furthermore, PPG enables a high-level control over the training process by multiplying the number of hyper-parameters to be set, making the setting of a satisfying configuration more complicated. PPG was also more affected by the way we managed the ϵ, as shown in figure 8b. The average normalized reward was improving following a step function like trajectory, whose increases were determined by progressive updates of ϵ at 500th and 1000th phases out of 1500. PPO training was in general faster and more regular, as shown in figure 8b.
In figures 8c and 8d, the best normalized reward per phase is plotted and we find that both algorithms can find the same optimal solution of *ORTOOLS* solver for each graph in the training data-set.



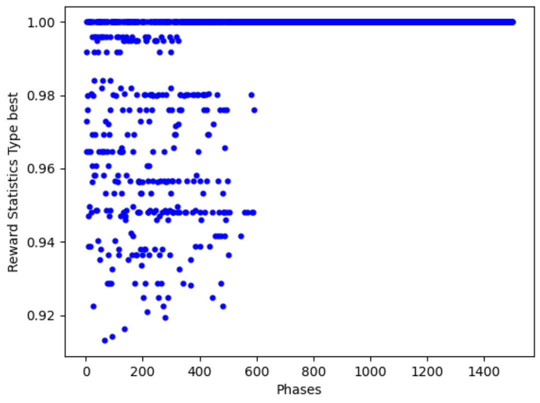
(a) PPO: average reward



(b) PPG: average reward



(c) PPO: Best reward



(d) PPG: best reward

Figure 8: AC algorithms training performances

Instance dim.	**N**	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	$\bar{t}$
10 nodes	1000	0.839	0.699	0.712	0.802	0.611	N/A	1.212
15 nodes	1000	0.812	0.693	0.749	0.729	0.683	0.769	1.041
25 nodes	5000	0.534	0.447	0.572	0.575	0.343	0.344	1.511
30 nodes	5000	0.589	0.551	0.559	0.602	0.411	0.488	2.012
50 nodes	6000	0.687	0.568	0.518	0.499	0.398	0.509	1.997
10 to 40 nodes	9000	0.489	0.488	0.501	0.412	0.339	0.500	1.443

Table 2: Test average performances of PPO over 100 test instances with different configurations and 10 roll-outs per test instance

Instance dim.	N_{ph}	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	$\bar{t}$
10 nodes	1000	0.846	0.612	0.692	0.704	0.601	N/A	0.996
15 nodes	3000	0.733	0.714	0.723	0.699	0.512	0.678	1.791
20 nodes	7000	0.788	0.611	0.776	0.791	0.449	0.588	1.886
10 to 25 nodes	12.000	0.709	0.677	0.699	0.512	0.477	0.555	1.253

Table 3: Test average performances of PPG over 100 test instances with different configurations and 10 roll-outs per test instance

Test results in terms of (12) and (13) are presented in the following tables 2 and 3.
Increasing the instance dimension, PPG, due to its complex structure, requires a training time that is more than proportional to PPO; for this reason, PPG has been trained only on graphs up to 25 nodes.
Figure 9 shows the behavior of test normalized reward with an increasing number of roll-outs. After approximately 10 roll-outs, that is the number set for all the tests, the best-case curve appears to be stable in the proximity of the optimal value.

8 Conclusions

In this paper we compared the performances of three different RL algorithms and we proposed the use of Convolutional Neural Networks to approximate the policy, the state-value and the action-value functions. We find that, while Q-Learning algorithm 1 did not produce satisfying results, experiments carried out on PPO and PPG lead to the two following considerations:

- Two AC algorithms, which had been originally developed for robotic control problems, turned to be highly effective in learning TSP solutions on graphs with a uniform node concentration. Furthermore, both the algorithms are able to generalize within a certain extent to other distributions and bigger graphs, proving that the agent is capable of detecting local structures such as clusters and curves with no prior knowledge.
- Computational experiments have shown that Convolutional Neural Networks are an efficient and effective tool to process the information stored in a weighted incidence matrix

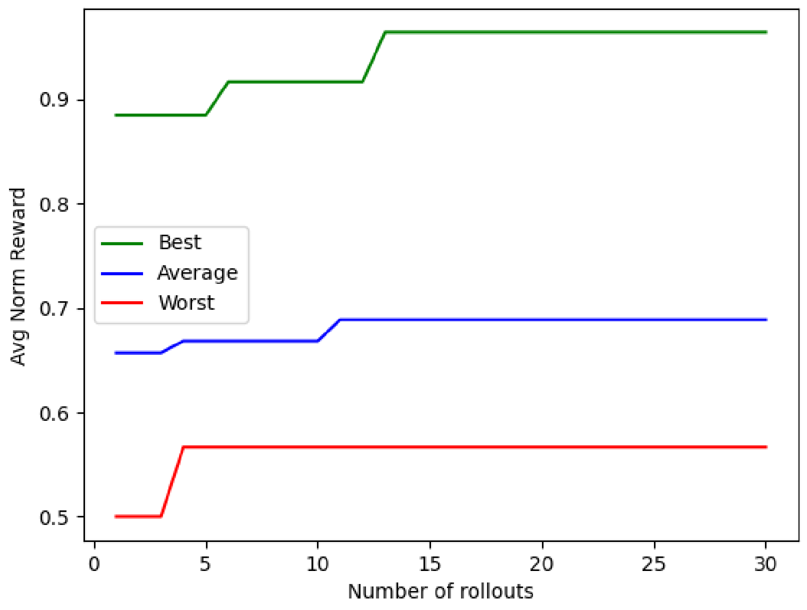


Figure 9: Normalized reward on 100 test instances: worst, average and best result with increasing number of roll-outs

and, in particular, corroborate our initial idea to consider the graph as an image, whose visual representation is given by the state-matrix.

Eventually, the experimental observations suggest that by increasing the instance size, the trained agent is more than proportionally faster than the *ORTOOLS* solver in achieving a solution.

References

[1] Thomas Barrett, William Clements, Jakob Foerster, and Alex Lvovsky. Exploratory combinatorial optimization with reinforcement learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(04):3243–3250, Apr. 2020.

[2] Richard Bellman. On the theory of dynamic programming. *Proceedings of the National Academy of Sciences of the United States of America*, 38(8):716, 1952.

[3] Richard Bellman. Combinatorial processes and dynamic programming. Technical report, RAND CORP SANTA MONICA CA, 1958.

[4] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

[5] Y. Bengio and Yann Lecun. Convolutional networks for images, speech, and time-series. 11 1997.

[6] Dimitri P Bertsekas. *Reinforcement learning and optimal control.* Athena Scientific Belmont, MA, 2019.

[7] Dimitri P Bertsekas et al. *Dynamic programming and optimal control: Vol. 1.* Athena scientific Belmont, 2000.

[8] Karl Cobbe, Jacob Hilton, Oleg Klimov, and John Schulman. Phasic policy gradient. *arXiv:2009.04416v1*, 2020.

[9] Donald Davendra. *Traveling Salesman Problem: Theory and Applications.* BoD Books on Demand, 2010.

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning.* MIT press, 2016.

[11] Dmitry Krass Jerzy A. Filar. The embedding of the traveling salesman problem in a markov decision process. *Proceedings 01 the 26th Conference on Declslon and Control, Los Angeles*, 1987.

[12] Paul McMenemy, Nadarajen Veerapen, Jason Adair, and Gabriela Ochoa. Rigorous performance analysis of state-of-the-art tsp heuristic solvers. In *European Conference on Evolutionary Computation in Combinatorial Optimization (Part of EvoStar)*, pages 99–114. Springer, 2019.

[13] Karl Menger. *Ergebnisse eines Mathematischen Kolloquiums.* 1930.

[14] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[15] Navdeep Jaitly Oriol Vinyals, Meire Fortunato. Pointer network. *arXiv:1506.03134v2*, 2017.

[16] Manfred Padberg and Giovanni Rinaldi. Optimization of a 532-city symmetric traveling salesman problem by branch and cut. *Operations Research Letters*, 6(1):1–7, 1987.

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

[18] Christos H. Papadimitriou. The euclidean travelling salesaman problem is np-complete. *Theoretical Computer Science 4, pp. 237-244*, 1977.

[19] John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. *arXiv:1502.05477v5*, 2017.

[20] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy gradient. *arXiv:1707.06347v2*, 2017.

[21] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[22] Ola Svensson. Overview of new approaches for approximating tsp. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 5–11. Springer, 2013.

[23] Mingyu Xiao and Hiroshi Nagamochi. An exact algorithm for tsp in degree-3 graphs via circuit procedure and amortization on connectivity structure. *Algorithmica*, 74(2):713–741, 2016.

[24] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications, 2021.

A Q-Learning hyper-parameters setting

ϵ_0	0.999
$\bar{\epsilon}$	0.008
η	0.998
N	10.000
Optimizer	Adam
Learning Rate	1×10^{-4}
E	1
K	10
γ	1

B PPO hyper-parameters setting

ϵ	0.2
N	See Table 2
Actor Optimizer	Adam
Actor Learning Rate	5×10^{-7}
Critic Optimizer	Adam
Critic Learning Rate	5×10^{-4}
E	1
K	10
γ	1
Mini-batch sizes	Set to the instance dimension

C PPG hyper-parameters setting

ϵ	0.2
N_{ph}	See Table 3
N_{π}	3
Actor Optimizer	Adam
Actor Learning Rate	1×10^{-6}
Critic Optimizer	Adam
Critic Learning Rate	5×10^{-4}
E_p	1
E_v	1
E_{aux}	1
K	15
γ	1
Mini-batch sizes	Set to the instance dimension