Reinforcement Learning for the Knapsack Problem

Pierotti, Jacopo; Kronmueller, Maximilian; Alonso-Mora, Javier; van Essen, J. Theresia; Böhmer, Wendelin

DOI
10.1007/978-3-030-86286-2_1

Publication date
2021

Document Version
Final published version

Published in
AIRO Springer Series

Reinforcement Learning for the Knapsack Problem

Jacopo Pierotti, Maximilian Kronmueller, Javier Alonso-Mora, J. Theresia van Essen, and Wendelin Böhmer

Abstract Combinatorial optimization (CO) problems are at the heart of both practical and theoretical research. Due to their complexity, many problems cannot be solved via exact methods in reasonable time; hence, we resort to heuristic solution methods. In recent years, machine learning (ML) has brought immense benefits in many research areas, including heuristic solution methods for CO problems. Among ML methods, reinforcement learning (RL) seems to be the most promising method to find good solutions for CO problems. In this work, we investigate an RL framework, whose agent is based on self-attention, to achieve solutions for the knapsack problem, which is a CO problem. Our algorithm finds close to optimal solutions for instances up to one hundred items, which leads to conjecture that RL and self-attention may be major building blocks for future state-of-the-art heuristics for other CO problems.

Keywords Reinforcement learning · Multi-task DQN · End-to-end · Knapsack problem · Transformer · Self-attention

1 Introduction

In recent years, machine learning (ML) has shown super-human capabilities in speech recognition, language translation, image classification, etc. [4, 12, 16]. Lately, more and more combinatorial optimization (CO) problems have been studied under the lens of machine learning [3]. Among these CO problems, NP-hard problems are of interest because, so far, solving them to optimality (via so-called exact methods) takes exponential time; thus, for many classes of CO problems, obtaining good solutions for large or even medium sized instances in reasonable time can only be achieved by exploiting handcrafted heuristics. Instead of creating

J. Pierotti (✉) M. Kronmueller · J. Alonso-Mora · J. T. van Essen · W. Böhmer
TU Delft, Delft, Netherlands
e-mail: J.Pierotti@tudelft.nl; M.Kronmuller@tudelft.nl; J.AlonsoMora@tudelft.nl; J.T.vanEssen@tudelft.nl; J.W.Bohmer@tudelft.nl

A. Masone et al. (eds.), *Optimization and Data Science: Trends and Applications*, AIRO Springer Series 6, https://doi.org/10.1007/978-3-030-86286-2_1

a heuristic by hand, one can also use ML to train a neural network to predict an almost optimal solution for given or randomly generated CO instances [3]. This way heuristics can be learned without expert knowledge of the problem domain, which is also called *end-to-end training*. Reinforcement learning (RL) seems to be the most promising end-to-end method to solve combinatorial problems [2]. In fact, in difference to supervised ML, RL does not need to know the solutions to given training instances to learn a good heuristic. This way one can learn a heuristic without any domain knowledge and, in principle, one could find a heuristic that works better than any a human would be able to design. RL has been used to train the neural networks used by heuristics designed to solve CO problems [9–11], including the knapsack problem (KP) [2]. The aim of this paper is to develop an RL end-to-end algorithm for the knapsack problem based on attention [16], in difference to prior work that used either recurring neural networks (RNN) or convolutional neural networks (CNN) [4, 12] (which are popular NN for end-to-end methods). By developing such an algorithm for a relatively easy CO problem (the KP) [13], we want to assess if RL with attention can be a fruitful method to tackle other, more complex, CO problems, which will be the focus of future research.

The remainder of the paper is organized as follows. The formulation of the KP, our motivations on how and why we use attention and not RNNs or CNNs, and model architecture are presented in Sect. 2. The training distributions (i.e. benchmarks of instances) used for testing and evaluating as well as the computational results are detailed in Sect. 3. Finally, in Sect. 4, we illustrate our conclusions.

2 Problem Formulation and Background Information

The knapsack problem (KP) is one of the most studied CO problems [13]. As input, we have a set of objects (denoted by set N) and a knapsack of capacity W. Each object $i \in N$ has a positive profit p_i and a positive weight w_i. The objective of the problem is to maximize the sum of the profits of the collected objects without violating the capacity constraint. Introducing binary variables x_i, which assume value one if object $i \in N$ is selected and zero otherwise, we can write the problem as follows:

$$\max \sum_{i \in N} x_i p_i \tag{1}$$

$$\sum_{i \in N} x_i w_i \leq W \tag{2}$$

$$x_i \in \{0, 1\} \qquad \forall i \in N. \tag{3}$$

The objective function (1) maximizes the total profit of the selected objects, constraint (2) acts as the capacity constraint, and constraints (3) force the variables to be binary. This integer linear program (ILP) belongs to the class of NP-hard

problems [13], which means that the computation time for obtaining optimal solutions with known exact solution methods grows exponentially with the number of objects. A simple yet very powerful heuristic is to sort the objects in non-increasing order of their ratio, i.e., $q_i = \frac{p_i}{w_i}$ for $i \in N$, and collect them in order as long as constraint (2) is respected (collecting non-consecutive objects is allowed). In the following, we refer to this as the *simple heuristic*.

2.1 Reinforcement Learning Framework

In this section, we give an overview of how we implemented our algorithm. For more details on RL, we refer the reader to [14]. Our algorithm belongs to the area of multi-task RL [17], where a task is an instance of the knapsack problem. The difference between single and multi-tasks is that: in single-task, we want to learn a policy to always solve the same (instance of a) problem; in multi-task, we want to learn a policy to solve a family of different instances of a problem (or even different problems). Moreover, while in single-task the initial state is always the same, this does not hold in multi-tasks. In order to describe a state in our case, we first define how we embed the objects into vectors. At any given time step, each object $i \in N$ is uniquely associated to a vector. Each vector is of the form $t_i = [p_i, w_i, q_i, \bar{x}_i, u]$, where $\bar{x}_i$ is a binary parameter assuming value zero when object i has already been selected or cannot be selected due to the capacity constraint (2) and one otherwise, and u is the residual capacity of the knapsack (i.e., W minus the weights of the already selected objects).

We name the selection of an object an *action*. Actions (A) are chosen based on the Q-value of each object (see Sect. 2.1.1). The algorithm that determines the Q-values is called the *agent* (see Sect. 2.2). In RL, a *state* represents the available information about the process at a given moment. We represent the observation of a state by the matrix obtained stacking all the $|N|$ object vectors together. Given a non-final state, the agent has to select an action; however, not all objects can be chosen in any state. While choosing an action, the non-selectable objects are momentarily removed, which is called masking. In our case, generic action (object) i is masked when $\bar{x}_i$ equals zero. The initial state has $u = W$ and $\bar{x}_i = 1$ for all $i \in N$, while we define a state as final if $\bar{x}_i = 0$ for all $i \in N$. Our algorithm sequentially selects objects until no additional object can be selected, in which case the algorithm terminates.

Given a state, each action leads to a new state and a reward. In our case, the reward r of choosing object $i \in N$ is the profit of the chosen object (i.e. $r = p_i$ when choosing object $i \in N$). The series of states in between an initial and a final state is called an *episode*. The final objective of an RL algorithm is to maximize the (discounted) cumulative reward observed in an episode. In general, we discount the future reward to avoid problems arising with very long or non-finishing episodes. In our case, episodes are relatively short and they always terminate (worst case

Table 1 Summary of the definitions needed for our reinforcement learning framework

Name	Definition
Task	An instance of the KP
Multi-task RL	RL algorithm to solve a family of tasks (virtually any KP problem in our case)
Action	The selection of an object
State	The available information (profits, weights, which objects have been selected and which have not,..) at a given time moment
Initial state	State where no objects have been selected yet
Final state	State where no more objects can been selected
Episode	Series of visited states from the initial state to the final state
Masking	The removal of the unselectable actions
Reward	The profit of the selected object
Transition	A sequence of an old state, a chosen action, an observed reward, and a new state
Minibatch	A set of non-consecutive[1] transitions
Training distribution	Distribution from which we draw the instances to train the algorithm

scenario, they terminate in $|N|$ steps); so, there is no need to discount the future rewards. We call the sequence of old state s, chosen action a, observed reward r, and new state s' a *transition*. A set (of fixed size, in our case) of non-consecutive[1] transitions is called a *minibatch*. The transitions in the minibatches are used to compute the loss (which is needed in order to learn) in the learning step (Sect. 2.1.2). Finally, we train and evaluate the algorithm by solving randomly drawn tasks from the so called *training distribution*. Table 1 summarizes the introduced definitions.

2.1.1 Double Q-Learning

Our RL algorithm falls under the general umbrella of Q-learning [18]. Given a state s, and a set of possible actions A, the idea of Q-learning is to estimate the expected future cumulative rewards for each possible action (called Q-values $Q(s, a), \forall a \in A$) and select one action based on an exploration/exploitation strategy. On one hand, exploration is fundamental to search the state-action space. In fact, in (non-deep) Q-learning, if one could explore for an infinite amount of time, the optimal Q-values would be retrieved. On the other hand, the agent should concentrate more on promising actions to improve convergence to an optimal policy. As exploration/exploitation strategy, we use ϵ-greedy, which greedily chooses the best action (i.e. the action with the highest Q-value) with probability 1-ϵ or a random action with probability ϵ. Often the Q-learning algorithm can be too optimistic while

[1] Transitions do not have to be consecutive, but, by chance, they could be.

estimating the Q-values. One common solution to this problem is to adopt double Q-learning [7]. In deep RL, double Q-learning is enforced by having two identically structured neural networks. The current network is used to select the best action at the next state while the other one (called the *target* network) is used to compute the Q-value of the next state. In this work, we use a similar method which helps stabilizing our results. The difference being that the Q-values are always computed via the current network and the target network is used to determine the action. Naming Q' the function to compute the Q-values associated with the target network, our revised Bellman equation becomes (see also Sect. 2.1.2):

$$Q(s, a) = r(s, a) + Q(s', \arg\max_a(Q'(s', a))). \tag{4}$$

Equation (4) is needed in the learning step (see Sect. 2.1.2), where the parameters of the Q function are tuned in such a way that the distance between $Q(s, a)$ and $r(s, a) + Q'(s', \arg\max_a(Q(s', a)))$ is minimized.

2.1.2 Learning Strategy

Our algorithm works by generating and solving new tasks of different dimensions (i.e. $|N|$ is not a constant between two different tasks). Let us assume that we train our algorithm to solve instances of k different sizes. Every time a new instance is generated and solved, all the transitions are stored in a replay buffer [20]. Our algorithm has k different fixed-size replay buffers (one for each possible dimension of $|N|$) where transitions are stored with a FIFO (first in first out) strategy. A FIFO policy guarantees that the algorithm always keeps in memory the newest generated information. Transitions of instances with the same dimensions are stored in the same buffer. When a minibatch is needed, we randomly choose one of the k replay buffers and extract a minibatch from there. Given the multiple replay buffers, each state in the minibatch has the same dimension and, thus, can be stack together, easing the computation. It is important to note that different tasks have different gradient magnitude: a task with 100 objects is likely to have a different gradient than a task with 2 objects. In fact, we are using a NN to approximate the Q-values and, reasonably, the approximation becomes more and more difficult (thus less and less accurate) with an increasing number of objects. A less accurate Q-value approximation would likely lead to greater gradient magnitudes; thus, different tasks present different gradient magnitude. However, since each time we choose the replay buffer uniformly at random, we are averaging the gradients; thus, we are not introducing any bias. When the algorithm has accumulated *enough* transitions in the replay buffers, it begins to learn. We do so by selecting, uniformly at random, from one random replay buffer, t transitions (or all the transitions if less than t transitions are present in that replay buffer). Transitions which have never been selected before

have priority over transitions that were. We call these t transitions a *minibatch*. For generic transition i (s_i, a_i, r_i, s'_i) in the minibatch, we compute the loss as:

$$loss_i = \left(Q(s_i, a_i) - \left(r_i + Q(s'_i, \arg\max_a(Q'(s'_i, a))\right)\right)^2 \quad (5)$$

Then, we backpropagate the average of the t losses. Sometimes, the loss function is so steep that blindly following its gradient would lead outside of the region where the gradient is meaningful. To prevent this, we clip the gradient [19] to a maximum length of 0.1. The parameters of the agent are updated via the RMSprop method[2] [5]. Finally, the *target* network is updated via a soft-update [6], i.e., naming p any generic parameter of the agent, p_t its corresponding one in the target network and τ (constant equal to 0.05 in our case) the soft-update parameter: $p_t \leftarrow (1-\tau)\, p_t + \tau p$.

2.2 The Agent

The agent receives the observations of the states and outputs the Q-values. It is composed by three main blocks, all using ReLU as activation function. The first and last block are composed of two fully connected linear layers of dimension 512 each. The first block enlarges the feature space of each object vector from five to 512 and the last block reduces the features to one (the Q-value). The second block is a transformer (Sect. 2.2.1). In most CO problems, there is no clear ordered object structure. Even if we introduce an arbitrary order, the problem would be permutation invariant. In the KP, a permutation of the elements would neither change the optimal solution of the problem nor its structure. For this reason, we decide to base our agent on self-attention, which is permutation invariant (unlike CNNs or RNNs). While most agents for end-to-end approaches involve CNNs and/or RNNs [11], we conjecture that, for the KP and other CO problems, the effectiveness of an algorithm does not lie within those structures. In fact, CNNs are an excellent tool to extract local features [12], but they are only useful when there is a clear ordered object structure (such as pixels in an image). RNNs sequentially embed a sequence of inputs, where each output depends also on the sequence of previous inputs. This is very useful when states are partially observable [4]; however, the KP satisfies the Markov property, i.e., the distribution of future states depends only on the current state. This memoryless property makes the problem Markovian. Thus, given the Markovian property of our problem and the absence of an underlying ordered structure, we decide to base our implementation on a variation of the transformer [16] without CNNs or RNNs. The transformer accepts as input a variable-length (d_t) tuple of objects (where all objects have the same dimension d_o) and returns a tuple of same length and dimension (d_o for each single output, d_t for the whole

[2] http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

tuple). It is composed by a series of multi-head attention mechanisms in a layer structure (see Sect. 2.2.1). Attention is a powerful mechanism that allows to look at the input and generate a context vector based on how much each part of the input is relevant for the output. Doing so, the algorithm learns to isolate from a set of features the one(s) relevant for that particular state.

2.2.1 Self-Attention, Multi-Head, and Multi-Layer Transformer

Self-attention is a powerful ML technique that takes a set of objects and returns an equally sized set of vectors. In our case, the objects taken as input are matrices, called queries Q, keys K, and values V, which are three different linear transformations of the object vectors of size d_q, d_n, and d_n, respectively. Self-attention is a function that measures the similarity of queries and keys with a dot product; then, a softmax of that similarity is used to weight the values in a linear combination. So, naming W^Q, W^K, and W^V the matrices of learnable parameters for the linear transformations and $\bar{S} \in \mathbb{R}^{n \times d_n}$ the embedded state observation (i.e., the matrix obtained by stacking the object vectors of dimension 512), we obtain:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_n}}\right)V = \text{softmax}\left(\frac{(\bar{S}W^Q)(\bar{S}W^K)^\top}{\sqrt{d_n}}\right)(\bar{S}W^V).$$

Instead of a single self-attention mechanism on vectors of dimension d_n, [16] discovered that it was beneficial to linearly project the queries, keys, and values h times (called a *head*; hence, multi-head) with different, learned linear projections on a smaller dimension of size $d_v = \frac{d_n}{h}$. The outputs are computed in parallel, concatenated, and reprojected once again (via a learnable matrix W^0). Formally, this becomes:

$$\text{MultiHead}(\bar{S}) = [head_1, \cdots, head_h]W^0,$$

where $head_i = \text{Attention}(\bar{S}W_i^Q, \bar{S}W_i^K, \bar{S}W_i^V)$ and W_i^Q, W_i^K, W_i^V are all learnable matrices for all $i \in [1, \ldots, h]$. This multi-head self-attention mechanism is repeated for L layers. Each layer is composed of two units which both produce outputs of the same dimension as their input, i.e., d_n. The first unit is indeed the multi-head self-attention mechanism, the second unit is a fully connected feed-forward network with ReLUs. Both these units adopt also a residual connection and a layer normalization [1]. The residual connection was proven to facilitate learning [8].

2.3 Model Architecture

In each instance, all objects are normalized such that the maximum profit and weight is one. The agent has a two layer fully connected neural network to expand the 5 features of a vector into 512 features. The resulting vector is fed to a transformer encoder[3] with six layers and eight heads per layer. Normalization is applied after each layer. After the transformer, another two fully connected neural network layers are used to reduce the 512 features to a single one (the Q-value associated with the action of selecting the corresponding object). The learning rate of the optimizer is set to 10^{-6} and ϵ linearly decreases with the episode number from one to 0.05. Each replay buffer can store up to a maximum of 10^5 transitions, the minibatch size is set to 512, and the soft update parameter τ is set to 0.05. The overall structure of the algorithm is given in Algorithm 1. For a total of 10^5 times, the algorithm generates and solves one instance. Its transitions are saved in the replay buffer and the algorithm takes a learning step. In order to partially fill the replay buffers, the algorithm starts to learn only after the 512th iteration. During the training, ten equally spaced greedy test evaluations over one hundred randomly generated instances are conducted in order to assess the algorithm progress.

Algorithm 1: RL algorithm overview

```
for i = 0, ··· , 10^5 do
    task ← generate new task
    transitions ← solve the task with an ε-greedy policy
    store transitions in replay buffer
    if i ≥ 512 then
        learning step
    if i mod 10^4 = 0 then
        evaluate the algorithm with a greedy policy
```

3 Computational Results

Two different training distributions are used to generate the tasks. In the first distribution, $|N|$ is chosen uniformly at random between 2 and 100 every time a new instance is generated. Moreover, the profit and weight of each object are also chosen uniformly at random in the closed interval $[10^{-6}, 1]$. A lower bound of 10^{-6} is enforced to avoid numerical errors. We call this distribution *random*.

[3] For details see https://pytorch.org/docs/stable/generated/torch.nn.TransformerEncoder.html. Many optional parameters were set to the default values, such as the feedforward dimension was set to 512 and the probability of dropout to 0.1.

The second distribution (named *Pisinger*) are some of the *small, large*, and *hard* instances taken from [13]. These Pisinger instances were generated in order to be difficult to be solved via a MILP solver. These small, large, and hard instances are further subdivided in six, six, and five groups, respectively. From these groups, we select instances with 20, 50, and 100 objects. Each pair group-number of objects contains one hundred instances, for a total of 3200 instances (because not all groups have the 20 objects instances).

We train our algorithm twice from scratch, thus obtaining two different versions of the same model. We train the first version exclusively on the random instances while we train the second one exclusively on the Pisinger instances. We evaluate the trained algorithms both on random instances and on Pisinger's. When evaluating and testing, we compare our results with the simple heuristic (see Sect. 2) which achieves, on average, 99% of the optimal solution's value (hence, it is a good measure for comparison). In Figs. 1 and 2, every result is normalized with respect to

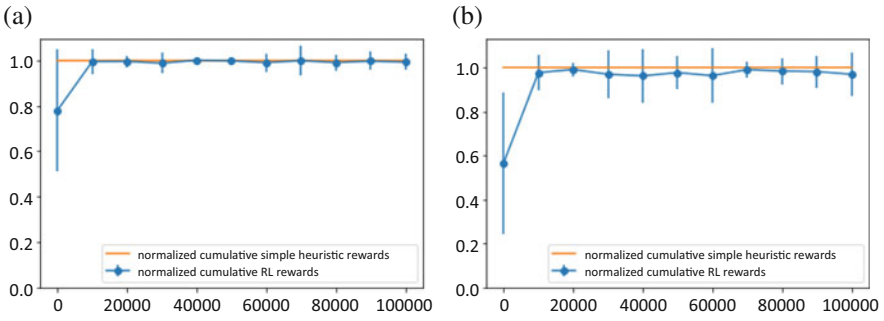

Fig. 1 During training evaluation on 100 random instances. On the x-axis, the number of iterations and on the y-axis, the averaged normalized cumulative reward are shown. The blue dots indicate the average cumulative reward, the vertical lines indicate the standard deviation. (**a**) Training on the random distribution. (**b**) Training on the Pisinger distribution

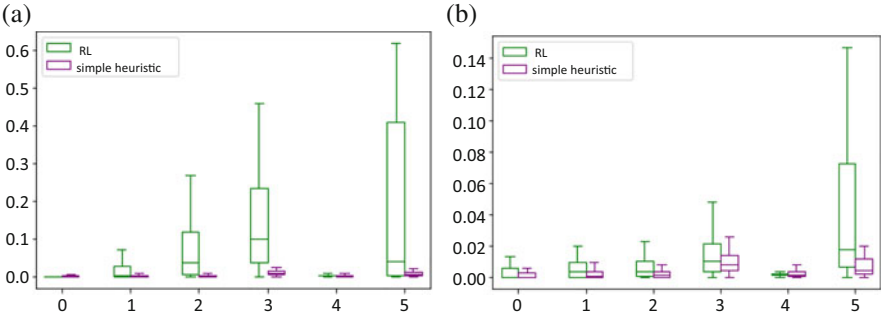

Fig. 2 Evaluation on the hard, 100 objects Pisinger instances. Green lines for the RL and purple lines for the simple heuristic. On the x-axis, different groups of instances, on the y-axis, the gaps to optimality are displayed. Please note the different scale of the y-axis. (**a**) Training on random distribution. (**b**) Training on Pisinger distribution

the optimal solutions (in the Pisinger distribution) or with respect to the heuristic solution. Figure 1 displays the evaluations of the algorithm during training on one hundred random instances. For the sake of brevity, we report only the most meaningful results, i.e., the *hard* Pisinger instances with one hundred objects. Figure 2a shows the boxplot of the gap to the optimal solution for the *hard* Pisinger instances of the algorithm trained on the random distribution. Although the results are overall satisfactory, the algorithm trained on random instances performs badly on some types of Pisinger instances. The most likely reason is that the algorithm trained on random instances has an extremely small probability of seeing some Pisinger instances (which have been handcrafted), thus it does not generalize over those particularly complex instances. On the other hand, when the algorithm is evaluated on randomly generated instances (Fig. 1a), results are very close to the heuristic solution, thus, to the optimal solution. Figure 2b displays the same gap for the algorithm trained on the Pisinger distribution. In this case, results are very satisfactory since the algorithm consistently achieves near-optimal solutions. Also while evaluating on randomly generated instances (Fig. 1b), results are very close to the heuristic solution, thus to the optimal solution; however, results are slightly worse than the results obtained by the algorithm trained on the random distribution. As expected, we conclude that training the algorithm on randomly generated instances boosts performance in the average case, but it is less effective to complex instances, while training the algorithm on the Pisinger distribution performs (slightly) worse on the average case, but is much more robust (both on the random and on the complex Pisinger instances).

4 Conclusion

In this work, we introduced a deep Q-learning framework with a transformer as the main deep architecture for the KP problem. The algorithm achieves results very close to optimality within a split second on instances up to one hundred objects. These results are promising; however, in the KP, also a simple conventional heuristic returns very solid results. Nonetheless, our results suggests that "attention is all you need" may also hold in end-to-end methods for CO problems. Future research will explore a transformer-based RL method on other CO problems where conventional heuristics fail to give good solutions in a short amount of time. Moreover, our algorithm computes the Q-values which are difficult quantities to estimate. Instead, one could aim to learn directly the policy with whom to take actions (i.e. the probability distribution of the actions for a given state). This policy could be learned by firstly using behavioural cloning [15] (to imitate the heuristic), and secondly RL, to explore more possible state-action combinations.

Acknowledgments This research was supported in part by Ahold Delhaize. All content represents the opinion of the author(s), which is not necessarily shared or endorsed by their respective employers and/or sponsors.

References

1. Ba, J.L., Kiros, J.R., Hinton, G.E.: Layer normalization. Preprint (2016). arXiv:1607.06450
2. Bello, I., Pham, H., Le, Q.V., Norouzi, M., Bengio, S.: Neural combinatorial optimization with reinforcement learning. Preprint (2016). arXiv:1611.09940
3. Bengio, Y., Lodi, A., Prouvost, A.: Machine learning for combinatorial optimization: a methodological tour d'horizon. Eur. J. Oper. Res. **290**, 405–421 (2021)
4. Bontemps, L., McDermott, J., Le-Khac, N.-A.: Collective anomaly detection based on long short-term memory recurrent neural networks. In: International Conference on Future Data and Security Engineering, pp. 141–152. Springer (2016)
5. Duchi, J., Hazan, E., Singer, Y.: Adaptive subgradient methods for online learning and stochastic optimization. J. Mach. Learn. Res. **12**(7), (2011)
6. Fox, R., Pakman, A., Tishby, N.: Taming the noise in reinforcement learning via soft updates. Preprint (2015). arXiv:1512.08562
7. Hasselt, H.: Double q-learning. Adv. Neural Inf. Process. Syst. **23**, 2613–2621 (2010)
8. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778 (2016)
9. Joshi, C.K., Cappart, Q., Rousseau, L.-M., Laurent, T., Bresson, X.: Learning TSP requires rethinking generalization. Preprint (2020). arXiv:2006.07054
10. La Maire, B.F., Mladenov, V.M.: Comparison of neural networks for solving the travelling salesman problem. In: 11th Symposium on Neural Network Applications in Electrical Engineering, pp. 21–24. IEEE (2012)
11. Nazari, M., Oroojlooy, A., Snyder, L.V., Takáč, M.: Reinforcement learning for solving the vehicle routing problem. Preprint (2018). arXiv:1802.04240
12. O'Shea, K., Nash, R.: An introduction to convolutional neural networks. Preprint (2015). arXiv:1511.08458
13. Pisinger, D.: Where are the hard knapsack problems? Comput. Oper. Res. **32**(9), 2271–2284 (2005)
14. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press (2018)
15. Torabi, F., Warnell, G., Stone, P.: Behavioral cloning from observation. Preprint (2018). arXiv:1805.01954
16. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is all you need. Preprint (2017). arXiv:1706.03762
17. Vithayathil Varghese, N., Mahmoud, Q.H.: A survey of multi-task deep reinforcement learning. Electronics **9**(9), 1363 (2020)
18. Watkins, C.J., Dayan, P.: Q-learning. Machine Learning **8**(3-4), 279–292 (1992)
19. Zhang, J., He, T., Sra, S., Jadbabaie, A.: Why gradient clipping accelerates training: A theoretical justification for adaptivity. Preprint (2019). arXiv:1905.11881
20. Zhang, S., Sutton, R.S.: A deeper look at experience replay. Preprint (2017). arXiv:1712.01275