Journal of Global Optimization
https://doi.org/10.1007/s10898-024-01364-6

A K-means Supported Reinforcement Learning Framework to Multi-dimensional Knapsack

Sabah Bushaj[1] · **İ. Esra Büyüktahtakın**[2]

Received: 26 September 2022 / Accepted: 8 January 2024

Abstract
In this paper, we address the difficulty of solving large-scale multi-dimensional knapsack instances (MKP), presenting a novel deep reinforcement learning (DRL) framework. In this DRL framework, we train different agents compatible with a discrete action space for sequential decision-making while still satisfying any resource constraint of the MKP. This novel framework incorporates the decision variable values in the 2D DRL where the agent is responsible for assigning a value of 1 or 0 to each of the variables. To the best of our knowledge, this is the first DRL model of its kind in which a 2D environment is formulated, and an element of the DRL solution matrix represents an item of the MKP. Our framework is configured to solve MKP instances of different dimensions and distributions. We propose a K-means approach to obtain an initial feasible solution that is used to train the DRL agent. We train four different agents in our framework and present the results comparing each of them with the CPLEX commercial solver. The results show that our agents can learn and generalize over instances with different sizes and distributions. Our DRL framework shows that it can solve medium-sized instances at least 45 times faster in CPU solution time and at least 10 times faster for large instances, with a maximum solution gap of 0.28% compared to the performance of CPLEX. Furthermore, at least 95% of the items are predicted in line with the CPLEX solution. Computations with DRL also provide a better optimality gap with respect to state-of-the-art approaches.

Keywords Multi-dimensional Knapsack problem · K-means · Deep reinforcement learning · Combinatorial optimization · Mixed integer programming · Heuristics

Abbreviations

A2C:	Advantage actor-critic
A3C:	Asynchronous advantage actor-critic
ACKTR:	Actor-critic with kronecker-factored trust region
BIP:	Binary integer programming

✉ İ. Esra Büyüktahtakın
esratoy@vt.edu

[1] Department of Management Information Systems and Analytics, SUNY Plattsburgh, New York, USA

[2] Grado Department of Industrial and Systems Engineering, Virginia Tech, Blacksburg, USA

Published online: 15 February 2024

COP:	Combinatorial optimization problem
DDQN:	Double deep Q-network
DQN:	Deep Q-network
DRL:	Deep reinforcement learning
DP:	Dynamic programming
FMSTS:	Fitting for minimizing SubTree size
KP:	Knapsack problem
MIP:	Mixed integer programming
MKP:	Multi-dimensional Knapsack problem
PPO:	Proximal policy optimization
TSP:	Traveling salesman problem
UBQO:	Unconstrained binary quadratic problem

Algorithm Notations

$\mathcal{C}$:	Set of constraints where the respective right-hand side value is appended to each constraint
$\mathcal{A}$:	Set of possible actions
$\mathcal{V}$:	Set of violated constraints
Ψ:	Set of item worth values
Υ:	Set of clusters
υ:	Index for a cluster where $\upsilon \in \Upsilon$
$\dot{\upsilon}$:	Index for a centroid of a cluster υ
ι:	Number of K-means iterations
θ:	DRL testing steps
τ:	DRL training steps
$\mathcal{Q}$:	DRL trained model

Problem Notations

$\mathcal{J}$	Set of all items, $\mathcal{J} = \{1, 2, \ldots, n\}$
$\mathcal{I}$	Set of knapsack constraints, $\mathcal{I} = \{1, 2, \ldots, m\}$
$\mathcal{B}$	Set of knapsack limits, $\mathcal{B} = \{b_1, \ldots, b_m\}$
$\mathcal{W}$	Set of weights for each item of set $\mathcal{J}$
$\mathcal{P}$	Set of all P problem instances
j	Index for an item where $j \in \mathcal{J}$
i	Index for the knapsack constraint where $i \in \mathcal{I}$
b_i	Index for the knapsack limit where $b_i \in \mathcal{B}$
a_{ij}	Index for the weight of the knapsack where $a_{ij} \in \mathcal{W}$

1 Introduction

The multidimensional knapsack problem (MKP) is an intriguing, strongly NP-hard problem [38] with multiple knapsack constraints. MKP can also be considered as a special case of integer programming, restricting the decision variables to 0 or 1. MKP first got attention as a capital budgeting problem [47]. MKP is a core resource allocation problem that lies as a sub-problem in many other problems having resource allocation constraints. Thus, contributions to solving MKP can affect a wide range of applications in various businesses, logistics, and computer networks.

Despite many research efforts worldwide and multiple solution approaches to solve MKP, there is still a large space for improvement, especially for efficiently solving large-scale

instances. The recent empowerment of machine learning methods to address optimization problems presents a wide area to explore. In particular, reinforcement learning is a promising candidate to outperform current approaches for large MKP instances. The literature suggests that reinforcement and deep reinforcement learning (DRL) approaches can learn solution strategies to solve combinatorial optimization problems [5, 6, 49].

In our paper, we propose a deep reinforcement learning approach combined with a heuristic and a K-Means algorithm to enforce the DRL in a framework to solve large MKP instances. The heuristic reduces the MKP to a more compact representation by evaluating all items and assigning a value of worthiness for each. Specifically, we create an RL environment that arranges a feasible solution according to the worthiness of items suggested by the heuristic. We use this environment as a training ground for an agent where different MKP instances are generated. Then, we propose an iterative K-means clustering algorithm to get a reasonable initial feasible solution that is used to train the DRL algorithm. We construct the DRL framework as a sequential decision-making process, where at each step of the algorithm, the agent decides whether the value of a specific item is predicted 1 or 0 until the problem becomes infeasible. So, an episode of the DRL algorithm is made of sequential decisions. Our approach is flexible to train and test using different state-of-the-art DRL models as a sub-method, such as deep Q-learning, policy gradient, or trust region gradient-based methods, in our DRL framework. In our framework, we account for solving MKP instances of various sizes. Our results suggest that we can train RL agents using distinct instances and then generalize the prediction to instances of different sizes and distributions. We improve CPLEX performance in terms of solution time with only an average of 0.28% additional solution gap over CPLEX. Furthermore, we offer the option of partially using the predicted MKP solutions to find better solutions than those provided by CPLEX.

Binary Decision Variables

$$x_j = \begin{cases} 1 & \text{if item } j \text{ is selected} \\ 0 & \text{otherwise.} \end{cases}$$

$$\hat{x}_j = \begin{cases} 1 & \text{if item } j \text{ is in reversed item vector is selected} \\ 0 & \text{otherwise.} \end{cases}$$

2 Related work

Early work on MKP treats it as a budget planning problem [47, 74, 75]. Applications include, but are not limited to, computer science [28, 66], epidemic disease control [83, 84], retail business organization [79], and modeling invasive species [9, 10]. There exist different approaches to solving MKP, such as exact algorithms [51, 70], approximation schema [55], heuristics and metaheuristics [16, 26, 29, 33, 69]. The main approaches used to obtain an exact solution of MKP are based on branch-and-bound and branch-and-cut. Different approaches have been suggested over the years for branch and bound for the MKP (see, e.g., Thesen [67], Gavish and Pirkul [27], Vasquez and Vimont [70], and Mansini and Speranza [51]). Several studies have also used dynamic programming (DP) to obtain an exact solution. Among recent DP attempts, there are methods proposed by Pisinger [57], Bertsimas and Demir [7], and Balev et al. [4]. Others, such as Büyüktahtakın [12, 13], present theoretical contributions to aid solving large stochastic multi-dimensional knapsack problems. Finding an optimal solution is computationally very expensive, which motivates researchers to investigate approximation algorithms. Different contributions to polynomial-time approximation solutions are due to

Frieze and Clarke [25] and Caprara et al. [14]. Again, the computational difficulty of MKP inspired many heuristic algorithms to compute a feasible solution in a reasonable time. We can categorize such heuristics as greedy [20, 48, 62, 78], relaxation-based [3, 7, 34, 50], and advanced [24, 42, 56]. The greedy approach is based on a relatively simple idea of considering items one by one until the problem becomes infeasible [23]. Even more detailed greedy approaches do not fall far from that.

Lately, deep-reinforcement learning (DRL) has gained much attention from researchers working on combinatorial optimization problems (COP). Despite many heuristics and exact algorithms proposed to solve COPs, solving large instances is still impossible by the best exact algorithms, such as branch-and-bound algorithms [76]. Despite excellent performance in small and medium-sized models, these methods are still not efficient in handling large-scale COPs. Other studies propose different approximation algorithms to deal with the time complexity of these instances [71]. Although approximation algorithms improve on the solution time, they frequently involve specific properties that require change each time problem settings are altered.

Recent studies use DRL in traveling salesmen problem (TSP) [6, 40, 54], job scheduling [15, 43], bin packing [35, 72], logistic problems [19, 58] and game playing [53, 64]. These developments have shown that the DRL formulation is suitable for solving sequential decision-making problems. Although many of these applications aim to solve TSP and vehicle routing, it is not difficult to extend these applications to the sequence-to-sequence concept, which is then applicable to solve instances of the knapsack problem (KP) Bello et al. [6]. Most of these studies intend to use the power of deep learning towards tackling the curse of dimensionality. Some studies use a Pointer Network architecture [6, 31, 40, 73]. Others come up with image or matrix formulations that can represent different classes of COP problems, such as maximum cut, minimum cover of the vertex, and knapsack [1, 5, 17, 36]. Some more recent studies use DRL to learn from state-of-the-art algorithms and improve them further [21, 45, 65, 80, 85]. Bushaj et al. [11] present a simulation deep reinforcement learning framework that avoids using COP for epidemic control optimization, instead using a reinforcement learning algorithm as a decision maker.

Vinyals et al. [73] introduce a Pointer Network architecture, in which the output layer of the deep neural network used in the pointer networks is a function of the input. Bello et al. [6] use the pointer with reinforcement learning to solve the TSP and knapsack problem. They use a policy gradient with an Advantage Actor-Critic (A3C) algorithm to train their deep neural network. Although initially designed to tackle TSP problems, they report optimal solutions for instances with up to 200 items for the knapsack problem. Gu et al. [31] present a deep learning algorithm to learn sequential decisions in an unconstrained binary quadratic programming problem (UBQP) since many of the COPs can be generalized into a UBQP. Kool et al. [40] propose a model based on attention layers with benefits incorporated into the pointer network. Using the REINFORCE algorithm, they claim to obtain a close-to-optimal solution for two variants of TSP problems with instances up to 100 nodes of the TSP network.

Dai et al. [18] introduce a new neural network framework for graph-based combinatorial optimization problems. They refer to *structure2vec*, introduced in Dai et al. [17], to derive an embedding of the graph vertices. They claim that their approach learns effective algorithms for TSP, Maximum Cut, and Minimum Vertex Cover problems. Barrett et al. [5] use DRL in a different direction as they present exploratory combinatorial optimization, in which they aim to improve agent learning even during test time continuously. In doing so, they claim they can achieve the state-of-the-art performance, although further improvements can be made by developing a better starting point. Afshar et al. [1] propose a DRL approach to solve the knapsack problem. They use state aggregation to extract features and construct states.

They compare the results with the implementations of the pointer network in Bello et al. [6] and Gu et al. [31] and report that their state-aggregated approach outperforms them. Hubbs et al. [36] develop a library of reinforcement learning environments consisting of multiple classic optimization problems. Even though they show that DRL is capable of picking up a policy for every problem, it was not able to outperform the heuristic models related to off-line knapsack problems. Kong et al. [39] follow a slightly different approach to the use of RL. They investigate whether a theoretically optimal algorithm can be found for online optimization problems. They claim that their results are consistent with the behaviors of optimal algorithms for problems such as AdWords problem, online knapsack, and secretary.

Another exciting ML-based approach to solving combinatorial optimization problems is presented by Yilmaz and Büyüktahtakın [82]. In their study, the authors develop a bidirectional long-short-term memory (LSTM) framework that can process information forward and backward in time to learn optimal solutions to sequential combinatorial problems with a focus on the capacitated lot-sizing problem (CLSP). Along these lines of research, Yilmaz and Büyüktahtakın [81] have innovatively adapted a well-known local attention-based encoder-decoder network that is initially designed for neural machine translation to solve multi-period combinatorial optimization problems. Different from previous work, the authors present an iterative algorithm to determine the optimal prediction level and eliminate any infeasible solutions, and demonstrate their framework on the multi-item CLSP and multi-item knapsack problems.

In Tang et al. [65]'s study focuses on building a configuration or classical representation of the original models, they propose using DRL to learn state-of-the-art cutting plane methods. They state that their model outperforms human-designed heuristics, and that their model can also benefit from the branch-and-cut algorithm. Similarly, Liao et al. [45] involve DRL in improving global vehicle routing algorithms. Based on their results, they claim that they can outperform the A* algorithm, which is a benchmark on a global search. Etheve et al. [21] show the DRL's strength as they use it to optimize the branching strategy of a Mixed Integer Program (MIP). The authors present Fitting for Minimizing the SubTree Size (FMSTS), a model that learns the branching strategy from scratch, and they compare it with commercial solvers, such as CPLEX.

To our knowledge, our DRL framework is unique with respect to the way the learning environment is defined to represent the multi-dimensional knapsack problem. Despite the increased complexity due to the multiple items and constraints considered in the multi-dimensional KP, our DRL method could be generalized to solving other Binary Integer Programming (BIP) problems.

2.1 Key contributions

In this study, we present a new DRL algorithm to tackle the computational difficulty of solving one of the most difficult classes of problems, MKP. Our approach joins the forces of reinforcement learning, K-means, and heuristic approaches and integrates them all into a DRL framework to solve a COP, such as the MKP. Because MKP forms the root of many practical problems, Chu and Beasley [16] state that MKP can be regarded as a general zero–one integer programming with non-negative coefficients. Therefore, improvements in solution methods for this class of problems can be extended to any integer programming problem of zero. Furthermore, due to the large domain of applications, this hard NP problem is frequently used as a benchmark problem to compare general-purpose methods in combinatorial optimization [32].

In an attempt to simplify the problem at hand by finding a reasonably starting solution, we propose an unsupervised learning algorithm using K-means to cluster the constraints with a distance-based similarity matrix and relax the problem by only considering a subset of the constraints determined by this K-means algorithm. This approach enables us to reduce the complexity of the problem at hand and get an acceptable feasible solution even for the largest instances to train our DRL algorithm. We also present a heuristic that helps us estimate each item's worth. This is a contribution to the heuristic solution approaches, combining data analysis and a greedy strategy that can be used alone as an algorithm itself. To harness the power of reinforcement learning, we initially formulate our problem in a 1D environment and then extend it to a 2D environment where an agent is trained to select and deselect items. Together with the above K-means algorithm and heuristic, a powerful DRL framework is presented to solve large MKP instances. To our knowledge, this is the first DRL model of its kind where a 2D environment is formulated, and an element of the matrix represents an item of the MKP. In addition to the framework, we also create a helper generator for the MKP instances that are randomly generated. We use this generator to produce MKP instances with a different number of items, constraints, and varying distributions of weight and cost parameters.

Among the powerful properties of the DRL framework is its generalization. The environment is set to extract information from different MKP instances and learn general patterns. The K-means algorithm plays an important role in generalization as well. For every instance, K-means and heuristic are executed, enabling a similar pattern to the DRL environment despite the different distributions of the instances. With their help, the solution is concentrated in an isolated area of the 2D environment. Specifically, when we create the DRL environment, we use sorting according to the heuristic, and for all instances, the agent learns "focused" and "advantageous" areas in the environment and where it focuses on searching for an optimal solution. Once trained using the initial K-means solution, the DRL model used with the same training can solve distinct instances. We present results to show that the DRL framework solves instances of different sizes and distributions faster than CPLEX, with a small gap where the DRL agent is only trained once. We also benchmark our approach against state-of-the-art heuristic approaches and show that our method provides better solutions in a slightly longer computational time.

3 Multi-dimensional Knapsack problem formulation

Here we present the mathematical formulation of the multi-dimensional knapsack problem. Without loss of generality, we assume that all parameters are non-negative. The multi-dimensional knapsack problem (MKP) is formulated as a binary integer program (BIP) (1) as follows:

$$P \quad \min \sum_{j=1}^{n} c_j x_j \tag{1a}$$

$$\text{s.t.} \sum_{j=1}^{n} a_{ij} x_j \geq b_i \qquad \forall i = 1, 2, \ldots, m. \tag{1b}$$

$$x_j \in \{0, 1\} \qquad \forall j = 1, 2, \ldots, n. \tag{1c}$$

The objective function (1a) minimizes the sum of knapsack (investment) costs over all items $j \in \{1, 2, \ldots, n\}$. Constraints (1b) ensure that the total return value of the invested items must exceed a given lower limit b_i defined for each constraint $i \in \mathcal{I}$, representing an investment type i. Finally, the constraints (1c) represent binary integer restrictions on the x_j variables.

4 Deep reinforcement learning Knapsack model

To solve our multi-dimensional knapsack problem, we present a DRL framework to derive a sequential selection policy minimizing the cost without violating any of the original constraints. Originally, our problem P consists of a set of items $\mathcal{J}$, a set of constraints $\mathcal{I}$ with a capacity b_i for each $i \in \mathcal{I}$, a set of weights $a_{ij} \in \mathcal{W}$ for each item $j \in \mathcal{J}$, and constraint $i \in \mathcal{I}$. To solve our problem, we incorporate four different state-of-the-art DRL algorithms, as described in Sect. 4.2.

To facilitate the multi-dimensional knapsack problem for the usage in the DRL algorithm, we present a heuristic that evaluates the items and sorts them based on their importance, considering their contribution to the objective function and the feasibility of the constraints. The details of this heuristic are presented in Appendix A.

4.1 K-means algorithm and initial solution

We develop a K-means algorithm [46] to provide a feasible good starting solution to DRL and a benchmark of what a solution should look like as input into the DRL training algorithm. In our K-means algorithm, the constraints of the knapsack instance are divided into multiple clusters, and an initial solution is generated with the help of a Cplex recursive procedure.

There are other heuristics that we can use to get a feasible solution faster and closer to optimality than the K-means approach. But our experiments show that having a closer initial solution to optimality does not necessarily imply that its easier to learn for the agent. While there is a general agreement that a good initial solution can often accelerate the agent's learning process in reinforcement learning (RL), there are situations where an overly good initial solution can actually hinder the capabilities of the RL agent. This phenomenon is known as "premature convergence" and has been widely studied in the literature [22, 41, 63]. RL agents must balance exploration (trying new actions to discover better policies) with exploitation (choosing actions that are known to be good based on current knowledge) [44]. Potential reasons for the premature convergence that arises with an excessively proficient initial solution is that the agent may become overly confident in this initial policy and thus may be less inclined to explore alternative strategies or actions. Furthermore, if the RL agent is too focused on exploitation, it may get stuck in a suboptimal policy because it does not explore enough to find better solutions.

During preliminary experiments, in addition to the K-means, we delve into a myriad of heuristic algorithms, such as the Primal Effective Capacity Heuristic (PECH) [2] and Adaptive Fixing (AF) [7], alongside various heuristics grounded in item ratios to find a good initial solution to feed the RL. What became evident from our extensive exploration was that these heuristic methods, more often than not, outperformed nested K-means in terms of speed and solution quality. However, this efficiency came at a price - it severely restricted the RL agent's capacity to explore and adapt, undermining the very essence of reinforcement learning. Moreover, it is worth noting that many of these heuristics pose the risk of becoming ensnared in local minima, further challenging the RL agent's ability to learn and adapt effectively.

We provide all constraints (weights and right-hand side) as input to the K-means algorithm. We calculate the similarity of two constraints using an $n+1$-dimensional Euclidean distance between two $n+1-$-dimensional vectors. Each of the vectors contains n items and the right-hand side value of the corresponding constraint. Minkowski and Jaccard's similarity measures [37] are tested as well, but observed to perform worse than Euclidean's.

Algorithm 1 K-means Constraint Clustering

Procedure: Generate Clusters
Input: $\iota, \Upsilon, \mathcal{C} = \mathcal{I} \cup \mathcal{B}$ {*Number of iterations, set of clusters, set of constraint vectors for problem P.*}
$\dot{\upsilon}_1 = \zeta_1, \dot{\upsilon}_2 = \zeta_2, \ldots, \dot{\upsilon}_{|\Upsilon|} = \zeta_{|\Upsilon|}$ {*Assign first constraints as centroids for each cluster* $\upsilon \in \Upsilon$*, where* $\dot{\upsilon}$ *is a centroid of a cluster* υ *and* $\zeta \in \mathcal{C}$}
for each ι **do**
  for $\zeta \in \mathcal{C}$ **do**
    for $\dot{\upsilon} \in \Upsilon$ **do**
      $D_{\zeta,\dot{\upsilon}} = \sqrt{\sum_{j=1}^{n+1} (\zeta - \dot{\upsilon})^2}$
    end for
  end for
  for $\zeta \in \mathcal{C}$ **do**
    Assign ζ to the closest cluster $\upsilon \in \Upsilon$ {*Reassign constraints to the closest cluster* υ *with centroid* $\dot{\upsilon}$.}
  end for
  for $\upsilon \in \Upsilon$ **do** {for each cluster}
    for $\zeta \in \upsilon$ **do**
      $\dot{\upsilon} = \left[\frac{\sum \zeta_1}{|\upsilon|}, \ldots, \frac{\sum \zeta_{|\upsilon|}}{|\upsilon|}\right]$ {*Recalculate the mean distance for each cluster dimension and assign it as the new centroid.*}
    end for
  end for
end for
Output: $\upsilon_1, \upsilon_2, \ldots, \upsilon_{|\Upsilon|}$ {*Output* $|\Upsilon|$ *clusters.*}

Procedure: Cplex Recursive Selection
Input: $\upsilon_1, \upsilon_2, \ldots, \upsilon_{|\Upsilon|}$
Define: $\bar{P}$ {$\bar{P}$ *is the unconstrained original problem.*}
Define: $\mathcal{V}$ {*Set of violated constraints.*}
for $\upsilon \in \Upsilon$ **do**
  Append the farthest pair of vectors of υ to $\bar{P}$
end for
Solve $\bar{P} \longmapsto \mathcal{V}$ {*Solve* $\bar{P}$ *and determine the set of the violated constraints* $\mathcal{V}$.}
while $\mathcal{V}$ is not $\emptyset$ **do**
  if $|\mathcal{V}| > 10$ **then**
    Append five most violated constraints to $\bar{P}$
  else
    Append all the remaining constraints to $\bar{P}$
  end if
  Solve $\bar{P} \longmapsto \mathcal{V}$ {*Recalculate violation set* $\mathcal{V}$.}
end while
Output: $\bar{X}$, $\bar{Z}$ {*Feasible solution and objective value of* $\bar{P}$.}

Considering two vectors $d_f = \left[a_{f1}, \ldots, a_{fn}, b_f\right]$ and $d_k = [a_{k1}, \ldots, a_{kn}, b_k]$ where f and $k \in \mathcal{I}$, we calculate each distance between d_f and d_k, D_{d_f,d_k} as:

$$D_{d_f,d_k}{}^2 = \sum_{j=1}^{n+1} \left(d_{fj} - d_{kj}\right)^2, \tag{2}$$

where d_{ij} is the j^{th} element of the constraint vector d_i for $i \in \mathcal{I}$.

At the starting point, we assign a centroid for each cluster by randomly selecting one of the constraints for each cluster. A distance map is populated by calculating the distance

between each vector d_i for $i \in \mathcal{I}$ and the centroid vectors $\upsilon_k \in \Upsilon$, using Eq. (2). Using this distance map, we reassign the vector d_i to the closest centroid or cluster. With the new assignment of the cluster, we recalculate the means of the cluster and the new centroids. We repeat this procedure for a pre-set number of iterations. We limit the number of iterations since we do not want to spend too much time on the K-means algorithm and only a good enough feasible solution is needed to design our RL environment. A good enough solution is a feasible solution obtained by using the least number of the constraints from the original problem. For each cluster, we select two of the farthest constraints from the centroid to serve as initial constraints in the following recursive Cplex procedure. If not feasible from the start, we calculate the highest violations in the constraint set and add them to resolve the problem.

4.2 DRL model

In recent years, different algorithmic approaches using neural network approximators for RL have been proposed to tackle large problems, especially COPs. Among the most popular ones are deep Q-learning, policy gradient methods, and trust region gradient methods. For each algorithm, we serve the state and possible actions as input, and we get back an action as an output. We perform training and testing using four state-of-the-art algorithms. For example, we train the model and test it using the Advantage Actor-Critic (A2C) method introduced in Mnih et al. [52]. The authors propose a DRL framework that uses asynchronous gradient descent to optimize deep neural network controllers. They use parallel actor-learners to update a shared model instead of experience replay used in Deep Q Network (DQN) to achieve a stable learning process. We also train and test our knapsack framework using a double DQN with a replay of the experience presented in Schaul et al. [60]. Although using too much memory and computational power, experience replay can correlate episode updates. Among the trust region policy optimization algorithms, we also tested our results in Actor-Critic using the Kronecker-Factored Trust Region (ACKTR) developed by Wu et al. [77]. ACKTR is a scalable trust region optimization algorithm for actor-critic methods. The authors use a Kronecker-factored approximation to the natural policy gradient, allowing the covariance matrix of the gradient to be inverted efficiently. Their paper is among the first to try to combine the benefits of different groups of algorithms (trust region policy optimization and the policy gradient). Another such algorithm is proposed by Schulman et al. [61] to achieve data efficiency and reliable performance of trust region policy optimization while only using the first-order optimization. Raffin et al. [59] implement reliable learning algorithms that make them reusable for the average developer. The selected algorithms (ACKTR, DQN, PPO, A2C) work well in discrete learning environments; hence, we tune each one and train our agents to identify the best learning agent.

4.3 1D Knapsack environment

In this section, we describe a one-dimensional vector representation of potential solutions of the multi-dimensional knapsack problem. We represent the states of the DRL algorithm as the combination of all possible binary selections in the vector where each element of the vector represents an item. Figure 1 shows the environment used to describe the one-dimensional formulation achieved using our heuristic in Algorithm 3. Our heuristic provides the order of the items in a vector based on their importance, and when sorting it, the agents will learn to concentrate on searching for the solution in a slightly isolated area. For example, when we sort the items according to the item's worth, the first items are more likely to be selected, and

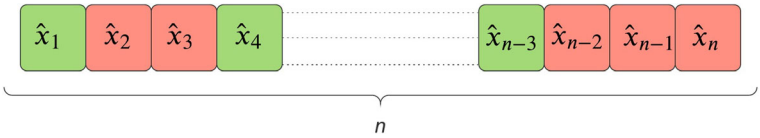


Fig. 1 One-dimensional vector representation of our problem

the last items are more likely to be ignored. This approach creates a focus area of decisions. Our aim is to train the agent to learn "swimming" in that area. Below, we describe episodes, state, action, and a reward function in our deep reinforcement learning algorithm using this 1D knapsack environment.

Episode: We define an episode as the steps taken from a current state until we find an infeasible solution or the maximum number of steps per episode is reached. In each episode, we aim to maximize the average reward.

State $s(P)$: Each state describes the current selection of an item $j \in \mathcal{J}$, as either 0 or 1. The heuristic defined in Algorithm 3 simplifies our states from the collective combination of items with their costs and weights and knapsack constraints to a simple state space of 2^n where n represents the number of items, and each item can take a value of 0 or 1. Thus, the state space is only defined by the selection of items free of the structure of constraints. We denote a state of the problem by $s(P)$.

Actions: We allow the agent to select / de-select an item at a step of each episode based on the current state. Therefore, we have n potential actions where n is the number of items at each step of the algorithm. We denote each action as A_j where j denotes a specific item. For each step, our algorithms are fed a certain state that describes whether an item j is selected or deselected, and an action is taken in that state. For example, if action A_j is taken in state $s(P)$, then if item j was selected, we deselect it, or if an item was not selected, we select it. This will form a new state $s'(P)$. At each step, we take only one action A_j for a particular item j.

Reward Function: Our reward function is guided by the original problem. Since we minimize the objective function, we aim to reduce the objective value without violating any constraints. Therefore, we give rewards based on four different situations. First, if the action reduces the objective value and leads to another feasible state, the agent is given a positive reward. Second, if the objective is increased and the problem remains feasible, then a small negative reward is given. Third, if an action leads to infeasibility, then a high negative reward is given. Lastly, if an action leads to a better solution than the starting solution, we give a higher positive reward, and if the solution is still feasible, we continue to the next step. Let Z_s be the objective value of a certain state s and j be the item currently selected in a step. We can then formulate the reward function as follows:

$$r(s(P), A_j) = \begin{cases} +Z_s & \text{if a better solution than the starting solution is found} \\ +c_j & \text{if the objective is reduced and feasibility is maintained} \\ -c_j & \text{if the objective is increased and feasibility is maintained} \\ -Z_s & \text{otherwise} \end{cases} \tag{3}$$

Based on our results, there are some disadvantages in using the 1D representation of the model. First, as instances' size increases, the number of states and actions increases, thus considerably affecting training time. Therefore, the model training time and the time the agent needs to learn will also increase. We reformulate our model using a two-dimensional

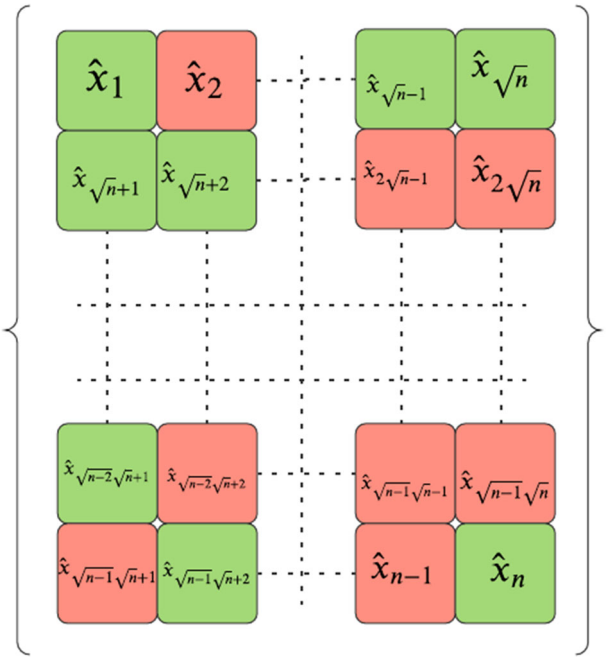


Fig. 2 Two-dimensional matrix representation of the state space

representation to overcome these weak points, gaining more advantage of the heuristic and lower action space, as presented in the next section.

4.4 2D Knapsack environment

Here, we develop a novel two-dimensional knapsack environment and formulation to overcome the weaknesses of the one-dimensional representation described in Sect. 4.3. The heuristic is used in this representation as well, but in addition, we reshape a vector of n items into a square two-dimensional matrix of $\sqrt{n}$, as shown in Fig. 2.

The reshape of the 1D representation is done after using the sorting heuristic. Here, the items are sorted according to their worthiness, starting from the top-left first cell to the bottom-right last cell. Similar to the isolated area of the solution on the 1D representation, sorting and locating the items based on their worthiness in the 2D matrix will help the agent learn faster to select or deselect items.

As an extension of the 1D formulation, some of the design properties are inherited. Changing the shape of the environment does not change our state space. Instead, it reduces the action space and also provides a path-like movement in selecting and deselecting items.

Episode: Similar to the 1D formulation, the episode starts with a feasible solution and moves on the 2D matrix until the solution (state) becomes infeasible or the maximum number of steps for each episode is reached.

States s: Changing the dimensions of the representation does not change our state space. For example, consider a 1D representation with 100 items. In our 2D representation, the solutions will be represented as a 10x10 matrix, where each element of the matrix represents an item's location. Again, state space would amount to 2^n.

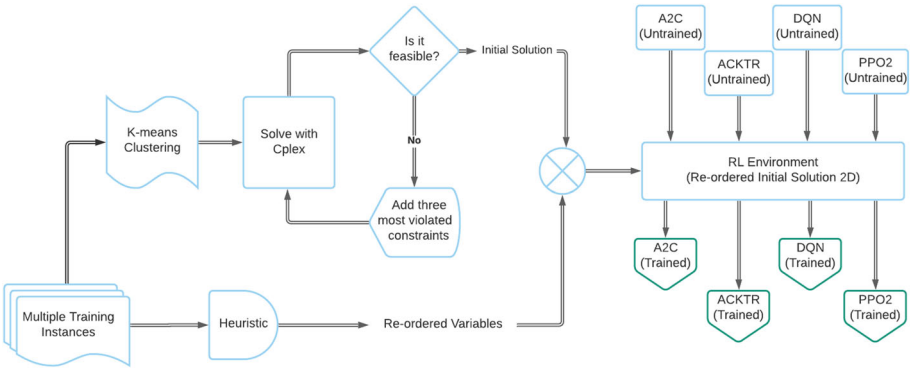


Fig. 3 Training DRL Flowchart

Actions: In this formulation, the primary benefit is seen in the action space. From having a representation where the action space increases with the size of the problem, we move to a formulation where the action space is the same for different-sized instances. We model the actions as movements in the matrix. We have four (4) discrete actions at any step. Up, Down, Right, and Left. Whenever the agent moves from one cell to another at each step, it reverts an item's selection or deselection decision and moves to another item to make a decision for the next step.

Reward Function: The reward structure also does not change with our structure. So, for the 2D formulation, we still use the reward function shown in Eq. (3).

4.5 Main DRL algorithm

To train and test both the 1D and 2D knapsack environments, we use a similar algorithm. The environments differ internally, but the general steps of the algorithm are the same. Combining the above structures, we build our training framework, as shown in Fig. 3. We use the same testing framework as in the training framework shown in Fig. 3. Using ten knapsack training instances of different sizes, we process each one in our K-means cluster to get a close-to-optimal feasible solution and also feed these instances to our heuristic to create a 1D ratio representation for each of the instances. Both of these results are used to prepare the DRL environment. Converting the 1D representation of ratios to a 2D matrix and organizing the K-means solution according to the ratios make up our initial DRL environment. We reset the training environment for every new knapsack instance. Each of the MKP instances for training is generated randomly and has different sizes. Independent of the sizes, the learning is controlled by a set number of steps that we define in the algorithm. Each of the instances is used to train the DRL model until the set number of steps is done. When all training is finished, the model is stored to be used for testing.

For testing, a similar flow to the training algorithm is followed, using a different set of instances for testing. Different from training, we perform three tests. For each of the tests, we generate ten instances for each size considered: small, medium, and large. Therefore, in total, 30 instances are tested based on the same loaded model from the training.

We demonstrate the steps of the training DRL framework given in Algorithm 2 in Fig. 3. The steps of the Testing DRL framework given in Algorithm 2 are also similar to the steps of the training procedure. We use K-means Constraint Clustering (Algorithm 1) to obtain a

close-to-optimal and feasible solution and a suboptimal objective value. The variable indices for each instance are re-ordered based on a list of item-worth values, using the heuristic shown in Algorithm 3. Using the initial solution combined with the item worthiness values, we can create the 1D and 2D environments with re-ordered items. After creating the environment, we train our agent for a pre-set number of steps and use one of the training DRL algorithms, A2C, ACKTR, DQN, and PPO2. Internally, 1D and 2D environments have their differences and similarities. When it comes to state space, reward strategy, and episode concept, they are similar. But the action space changes. In a 2D environment, we have reduced the number of possible actions in each step to four (4) of n possible actions in a 1D environment, where n is the number of items in the knapsack.

Algorithm 2 describes the training and testing procedures of the DRL framework. Both training and testing procedures make use of Algorithms 3 and 1 to prepare the ground for training DRL agents. The training loop is configured to run a set number of steps for each training instance. Although having the same number of steps, larger instances contribute mainly to the training time. In addition, the testing loop runs for a set number of steps for each MKP instance. The number of steps for testing is calculated based on the MKP instance size because of the huge range of the instance sizes we solve. During a set number of steps, multiple episodes can occur. Due to selection and deselection decisions, we keep track of the best solution achieved throughout all episodes.

4.6 Generalization to larger instances

Another essential property of every machine learning model is knowledge transfer. Knowledge/Learning transfer is a machine learning technique in which a model trained on one task is repurposed on a second related task [30]. With the current methods, similar instances can be easily implemented and the agent learns fast in a stable environment. In our model, we do not aim to solve only instances of the same size. We model our environment in a 30x30 matrix, despite the size of the instances. Our initial environment has all cells (items) assigned as -1. With this environment, we aim to solve instances as large as 900 items and 900 constraints.

For large instances, the values of -1 are replaced by assigning the item values of 1 or 0 in all cells. For smaller instances, we put the formulation of the square matrix in the top left corner of the environment as 0 and 1 s of an initial state obtained from K-means while leaving the other cells at -1 as initially assigned. Figures 4(a), (b), and (c) show how the environment is filled in different instance sizes. We expect the agent to learn a path that leads to the best solution. In the case of (a) and (b), we expect the agent to learn not to move around cells assigned with -1. This representation is used to generalize the framework to instances with different sizes, as it orders items in the matrix, and the RL agent learns to focus on a certain search area that is more advantageous in terms of finding the best set of solutions. In addition to the preprocessing of the instances before creating the RL environment, this formulation also reduces our action space, as mentioned in Sect. 4.4.

5 Experiments

5.1 Instance generation and implementation

To evaluate the computational performance of the DRL algorithm, we generate three types of instances with different sizes. We classify instances as small, medium, and large, based on

Algorithm 2 DRL Framework Algorithm

Procedure: Training DRL Framework
Input: τ {*DRL training steps.*}
Input: $\mathcal{P}$ {*Set of problems P for training.*}
for $p \in \mathcal{P}$ **do**
  $\hat{X} \longleftarrow Heuristic$ {*Get 2D ratio representation using heuristic Algorithm 3.*}
  $\bar{X},\ \bar{Z}_{\mathcal{P}} \longleftarrow K$-means {*Solution and objective obtained using K-means Algorithm 1.*}
  Env $\longleftarrow \bar{X} + \hat{X}$ {*Using heuristic result and K-means initial solution build DRL environment (Env).*}
  for each τ **do**
    Predict action $a \in \mathcal{A}$ {*For each step we only perform one action.*}
    Perform action $a \rightarrow obs, rew, done$ {*After action get the new state (obs), reward (rew), and episode end flag (done).*}
  end for
end for
Output: $\mathcal{Q}$ {*DRL trained model.*}

Procedure: Testing DRL Framework
Input: θ {*DRL testing steps.*}
Input: $\mathcal{P}_{\mathcal{T}}, \mathcal{Q}$ {*Set of test problems P_T, DRL trained model.*}
for $p \in \mathcal{P}_{\mathcal{T}}$ **do**
  $\hat{X} \longleftarrow Heuristic$ {*Get 2D ratio representation using Algorithm 3.*}
  $\bar{X},\ \bar{Z}_{\mathcal{P}_{\mathcal{T}}} \longleftarrow K$-means {*Solution and objective obtained using Algorithm 1.*}
  Env $\longleftarrow \bar{X} + \hat{X}$ {*Using heuristic result and K-means initial solution build DRL environment.*}
  for each θ **do**
    Predict action $a \in \mathcal{A}$ {*For each step we only perform one action.*}
    Perform action $a \rightarrow obs, rew, done$ {*After action get the new state (obs), reward (rew), and episode end flag (done).*}
    Store $\dot{X}^*$ and $\dot{Z}_{\mathcal{P}_{\mathcal{T}}}$ {*Keep the solution and objective if it is the best found.*}
  end for
end for
$\longleftarrow \dot{X}^*, \dot{Z}_{\mathcal{P}_{\mathcal{T}}}$ {*Return best solution and objective value from DRL.*}
Output: $\dot{X}^*,\ \dot{Z}_{\mathcal{P}_{\mathcal{T}}}$ {*Solution and objective value at the end of the testing procedure.*}

their sizes. A small instance is made up of 100 items and 100 constraints. A medium instance has 400 items and 400 constraints. Finally, a large instance is made up of 900 items and 900 constraints. For each size of the instance, we generate ten testing instances. The training set of 9 instances is made up of instances of different sizes, with three instances of each size. Using instances of different sizes has a significant impact on the results in terms of scalability and generalization. By training our methodology in varying instances, we can evaluate its ability to handle different practical problems. In addition, it helps us assess whether our DRL framework can generalize and adapt to different sizes and complexities of problems.

We generate each of the test instances for the MKP with the following distributions:

The parameters c_j and a_{ij} are independent and identically distributed (i.i.d.) random variables sampled from the uniform distribution over $\{1, \ldots, 10\}$, e.g. $U\,[1, R]$, where $R = 10$. We set $b_i = \frac{3}{4}\left(\sum_{j=1}^{n} a_{ij}\right)$.

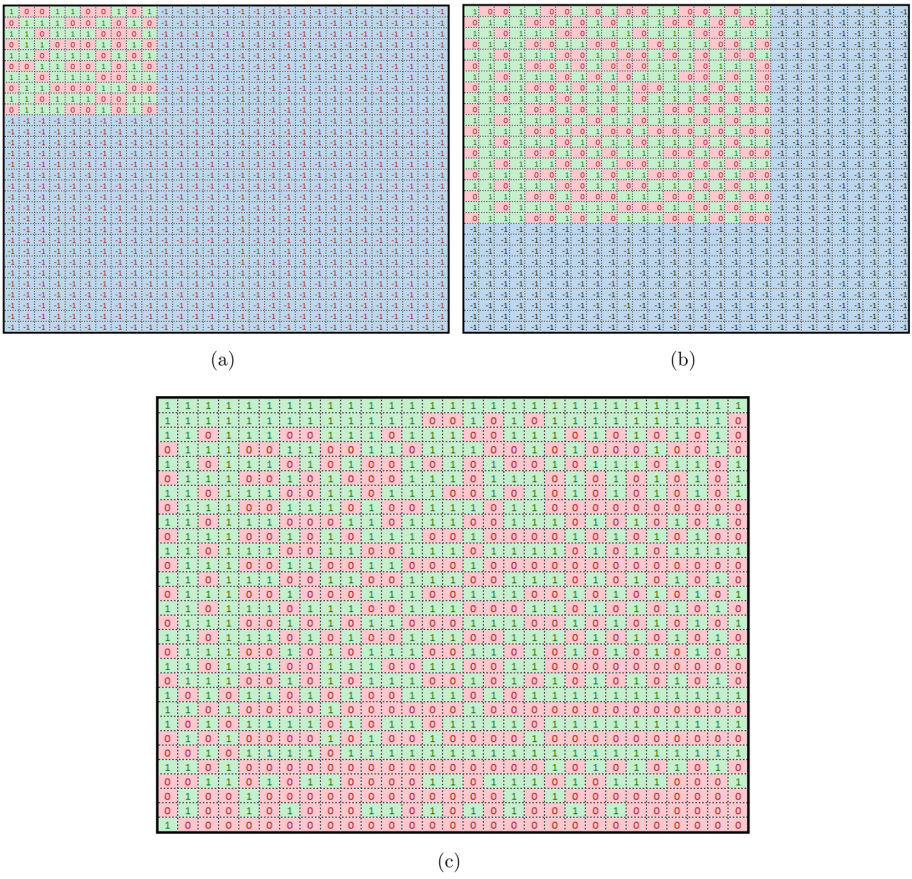


Fig. 4 2D DRL Environment for **a** 100 items and 100 constraints (small) **b** 400 items and 400 constraints (medium), and **c** 900 items and 900 constraints (large)

5.2 Implementation details

We implement our DRL algorithm using Python v3.7 and CPLEX v12.71 on a machine running Windows 10 with an Intel i7 CPU, 64 GB of RAM, and NVIDIA GeForce GTX 1070 GPU. CPLEX v12.71 was the state-of-the-art solver at the time of the implementation and provided exact solutions for most of the MKP problems tested. Thus, we use CPLEX as a benchmark and further investigate the quality of the solution with respect to the CPLEX solution. We first evaluate the performance of the DRL algorithm in terms of the objective function value, where we calculate % items predicted that are in line with the CPLEX solution, then we investigate further to understand how different our solutions are in terms of the selected elements.

Our DRL models were trained on a Windows 10 machine equipped with an Intel i7 CPU, 64 GB of RAM, and an NVIDIA GeForce GTX 1070 GPU. The software environment utilized during the training comprised industry standard deep learning frameworks and libraries, including TensorFlow 2.0 and Python 3.7. We used GPU acceleration to optimize the training process, leveraging the capabilities of the NVIDIA GeForce GTX 1070 GPU. During training,

Table 1 DRL, K-means, and Heuristic Parameters

Parameter	Description	Value
n	Number of items	100;400;900
m	Number of constraints	100;400;900
ι	Number of iterations for the K-means algorithm	30
Υ	Set of clusters for each instance size	n/25
τ	Number of DRL steps for training	100,000
θ	Number of DRL steps for testing	100
Gap^1	CPLEX optimality gap preset to solve the original problem	0.001 %
Gap^2	CPLEX optimality gap preset to solve the K-means reduced problem	0.01 %

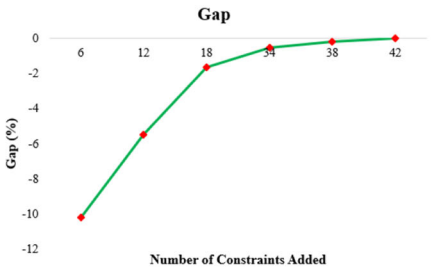
we feed different-sized instances in a random order and perform iterative training cycles. We used a stopping criterion based on a predefined threshold level to ensure that our models achieved convergence while avoiding overfitting.

Table 1 shows the parameter values used in the heuristic, K-means, and DRL algorithms with their symbol, description, and experimental value. We run our K-means algorithm for 30 iterations. The number of iterations is not set in a way that provides us with a very good initial solution; rather, we want it to be time effective and obtain a good enough initial solution by using only a subset of constraints, hence reducing complexity. We define a different number of clusters in the K-means for each instance size. As the size of the instance increases, the number of clusters in the K-means algorithm also increases.

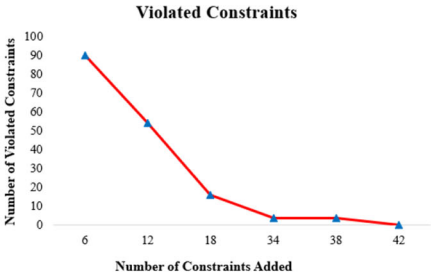
5.3 Results

To report computational results for each considered approach, we describe the following abbreviations:

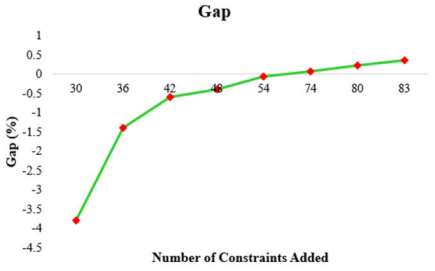
- **cpx:** solving the original problem (1a)-(1c) using CLPEX
- **ppo:** training and testing our DRL algorithm using PPO
- **acktr:** training and testing our DRL algorithm using ACKTR
- **a2c:** training and testing our DRL algorithm using A2C
- **dqn:** training and testing our DRL algorithm using DQN
- **obj:** objective function value based on the best solution found for all instances for its specific size averaged over 10 instances
- **ttime:** training time in CPU hours
- **soltime:** solution time in CPU seconds averaged over 10 instances
- **ipred (%):** percentage of item values correctly predicted with respect to the optimal solution averaged over 10 instances
- **gapdiff (%):** the average percentage change in the objective value compared with CPLEX objective $(rl\_obj - cpx\_obj)/cpx\_obj * 100$ where rl_obj and cpx_obj represent the objective found by DRL agent and CPLEX, respectively, averaged over 10 instances. The **gapdiff** value for **cpx** refers to the CPLEX MIP gap.



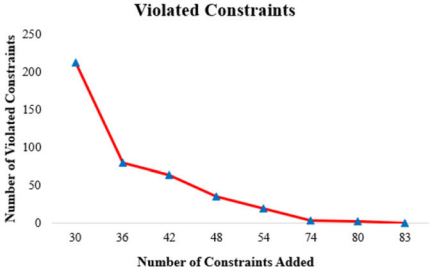
(a) Percent gap for small instances



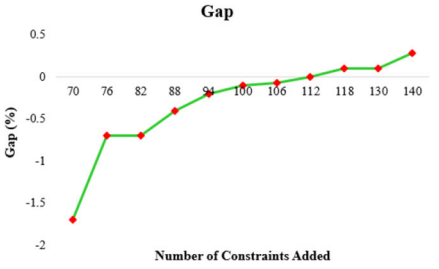
(b) Violated constraints for small instances



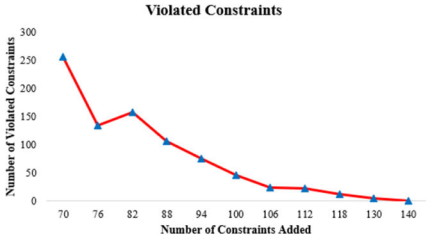
(c) Percent gap for medium instances



(d) Violated constraints for medium instances



(e) Percent gap for large instances



(f) Violated constraints for large instances

Fig. 5 Percent gap and the number of violated constraints for small, medium, and large instances using the iterative K-means Algorithm 1

5.3.1 K-means algorithm evaluation

To evaluate the performance of our K-means algorithm presented in Sect. 1, we investigate the rate of improvement and the progress after each CPLEX loop. Starting with clustered constraints, we solve our relaxation, namely the reduced K-means problem, check for violated constraints and calculate the gap between the objectives of the original and the reduced problem, as described in Algorithm 1. Since we start with a low number of constraints included from the clusters, the first solution found from the relaxed problem is often infeasible for the original problem. Figure 5 shows how the percent gap between the original and the K-means reduced problem's objectives, and the number of violated constraints changes after each loop to the point where a subset of constraints is reached to result in feasibility in the

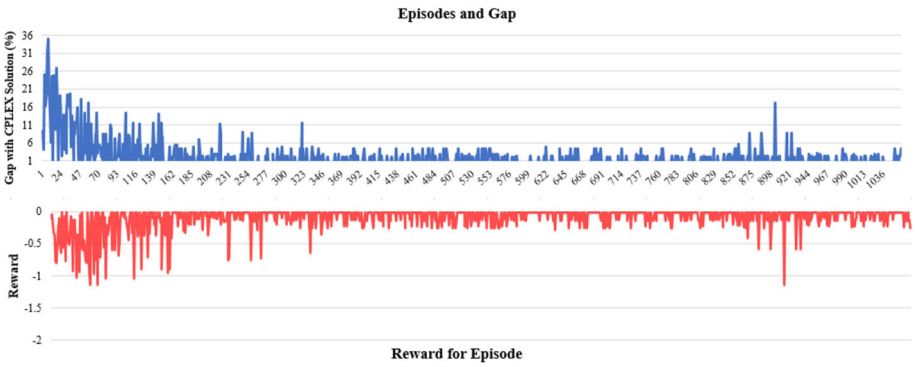


Fig. 6 Training gap and rewards for each episode where a negative reward represents a positive contribution to reducing the minimization objective function

original problem. For each instance size, we can see that this solution helps us identify a feasible solution (with a small gap on large instances) in a timely manner.

Figures 5(a),5(c), and 5(e) show how the gap changes between the objectives of the original and K-means reduced problems after each iteration for small, medium, and large instances, respectively. For small instances, an optimal solution is reached in most cases, but for medium and large instances, when a feasible solution is found, its objective value has a positive gap of around 0.5% compared to the best objective value of the original problem found by CPLEX.

Figures 5(b),5(d), and 5(f) also show how the number of violated constraints changes over each iteration of the K-means algorithm for small, medium, and large instances, respectively. Initially, the K-means algorithm starts by violating a big subset of the constraints, but very fast, it converges to a feasible solution. We will use the solution found by K-means to train the DRL agent. The performance of the agent's learning with respect to the initial K-means solution is discussed in the next section.

5.3.2 Instance learning and reward performance

To evaluate the learning process of an agent and the performance of our reward function, we plot the reward results of the training process against the respective values of the gap between DRL's updated objective value and the objective value of the initial solution provided by the K-means algorithm using CPLEX. We use the initial solution and the initial objective value found by the K-means approach given in Algorithm 1 to guide the DRL agents in the learning process. Figure 6 shows how the reward function reacts to the percent gap between the DRL solution and the K-means solution. The top y-axis presents the percent gap between the DRL and initial K-means objectives, while the down y-axis shows the reward for each respective episode, where a negative reward represents a positive contribution to reducing the objective function. Notice in Fig. 6 that as the gap increases, the rewards given to the agent decrease. However, the improvement is not completely smooth during the training episodes due to training using different-sized instances. We can still see that the model is learning to get higher rewards and, consequently, getting close to or better than the initial objective value found by the K-means algorithm.

As with general applications of the DRL, the learning or training process is the most time-consuming. To evaluate the results regarding generalization, we only train one model for each of the DRL algorithms (**a2c**, **acktr**, **dqn**, and **ppo**) where the parameters of the MKP

model are sampled using $U[1, R]$, where $R = 10$. Then, each of these DRL models will be used to predict MKP instances of different sizes and distributions. The training time (**ttime**) for **a2c** and **dqn** is 39 CPU hours, while for **ppo** and **acktr**, training takes 38 and 33 CPU hours, respectively.

5.3.3 Comparing Four DRL algorithms and CPLEX performances

In this section, we compare the performance of each DRL algorithm with that of the CPLEX commercial solver. Each row of Table 2 shows results that are averaged over 10 test instances with a specific instance size.

The agent is trained only once for each DLR algorithm (a2c, acktr, dqn, and ppo), and then we predict small, medium, and large instances with the same respective agent. The percentage difference (**gapdiff**) is calculated with respect to the best solution found by CPLEX. For the results in Table 2, the gap (**gapdiff**) and solution time (**soltime**) depend on the number of iterations used in the K-means algorithm ι and the number of DRL steps used for testing θ.

Our primary aim in this study is to benefit in solving harder instances faster, but we present results for small instances to show a wide range of applicability of our approach. For small instances, the DRL agents are able to find an optimal solution, although each of the agents takes longer to solve than **cpx**. Also, the percentage of item values correctly predicted with respect to the optimal solution is consistently above 97%. For medium instances, we notice that we have small gaps with respect to **cpx**, but they do come with a significant improvement in the solution time. In terms of solution time and gap with respect to **cpx**, **acktr** is a clear winner for medium instances. Each DRL solution requires at least 45 times less CPU solution time in CPU seconds and only has a max **gapdiff** of 0.28 %. Despite this, **acktr** is closely followed by the other algorithms with a slightly larger gap and solution time. With respect to the number of item values correctly predicted, **a2c** is slightly ahead with a small percentage.

For large instances, trained DRL agents still show some gaps, but in the worst-case scenario, we improve ten times over **cpx** in terms of solution time. **dqn** seems a clear winner over other DRL algorithms with respect to the solution time, the gap, and the percentage of item values correctly predicted. The **dqn** provides a solution that predicts 97. 8% of the same items as **cpx** using at least 14 times less time and only having a **gapdiff** of 0.22%.

In this section, we address the generalization of the proposed DRL algorithms by training each of the agents with a mix of varying-size instances and then using that agent to predict instances with different sizes. In the next section, we show the generalization of our DRL approach to solving instances with not only different sizes, but also different distributions.

5.3.4 Generalization to different distributions

To further expand the generalization, we want to use the same trained agents to predict instances generated from different distributions. For this set of generalization experiments, we generate groups of 10 small, medium, and large instances with a uniform distribution $U[1, R]$, where $R = 25$ and $R = 100$, to test the trained DRL models.

Each row of Table 3 shows the test results for an average of 10 instances generated with $U[1, R]$, where $R = 25$, using the DRL models that are formerly trained using instances with $U[1 : R]$, where $R = 10$. Thus, for these generalization experiments, we do not perform any additional training. Our results show that for small instances, the DRL framework finds the optimal solution using each of the four RL algorithms, and a high percentage of items are predicted to be the same as the solution from **cpx**. In medium and large instances, we

Table 2 Comparison of DRL algorithms and CPLEX performances

Instance	Algorithm	obj	soltime (CPU sec.)	gapdiff %	ipred %
Small	**cpx**	359.9	4	0	–
	a2c	359.9	7	0	98.2
	acktr	359.9	8	0	97.8
	dqn	359.9	5	0	97.0
	ppo2	359.9	7	0	97.2
Medium	**cpx**	1355.1	7206	0.26	-
	a2c	1358.8	159	0.27	97.6
	acktr	1358.7	135	0.26	96.8
	dqn	1398.9	136	0.28	96.6
	ppo2	1358.8	147	0.27	96.8
Large	**cpx**	2994.2	7209	0.25	-
	a2c	3002.9	715	0.29	97.4
	acktr	3002.8	607	0.28	97.3
	dqn	3002.6	494	0.22	97.8
	ppo2	3002.9	680	0.29	97.2

notice a slight increase in the **cpx** optimality gap and the solution time compared to similar results shown in Table 2, implying that the testing instances with $R = 25$ are harder than the training instances with $R = 10$. Despite the change in distribution (R = 10 to R = 25), the gap **gapdiff** and the percentage of items' values correctly predicted **ipred** still perform well. Specifically, the **gapdiff (%)** values provided by DRL algorithms are improved in Table 3 compared to Table 2. This shows that the solution found by our DRL algorithm is closer to the CPLEX solution for those more difficult instances when evaluating the **gapdiff**. Thus, using a distribution for the test instances than those used in training instances does not impact the good solution performance of our DRL approach.

Table 4 shows further results for our generalization to different distributions. Again, using models trained with instances with $U[1, R]$, where $R = 10$, we predict ten instances for instances generated with $U[1, R]$, where $R = 100$ and varying sizes. Once again, our results show that the DRL framework can scale to different distributions without any significant loss in **soltime** or **gapdiff**. Although we notice an increase in **time of absence** for small instances, the percentage of item values correctly predicted **pred** is better compared to those in Tables 2 and 3. In the case of medium and large instances, despite a small increase in **soltime**, the significant improvement with respect to **cpx** is still maintained. For large instances, while **soltime** increases, a 7-fold improvement by the DRL algorithms is preserved. Hence, our generalization experiments prove that the DRL framework retains a good performance even if the distribution of the test instances is varied.

5.3.5 RL partial prediction

To search for better solutions in large instances, we propose a partial prediction that can be used to guide CPLEX and offer a better and more time-efficient solution. By utilizing the DRL framework properties, we can decide on what items the framework selects and deselects with high certainty. On the basis of this information, we can tailor a partial prediction guided

Table 3 Comparison of DRL algorithms with CPLEX for test instances with $R = 25$

Instance	Algorithm	obj	soltime (CPU sec.)	gapdiff %	ipred %
Small	**cpx**	818.5	4	0	-
	a2c	818.5	7	0	97.2
	acktr	818.5	8	0	98.6
	dqn	818.5	5	0	98.6
	ppo2	818.5	6	0	98.6
Medium	**cpx**	3157.6	7206	0.43	-
	a2c	3166.5	203	0.28	95.5
	acktr	3164.8	176	0.22	96.0
	dqn	3163.5	177	0.18	96.0
	ppo2	3163.5	173	0.18	96.3
Large	**cpx**	6811.1	7209	0.33	-
	a2c	6826.0	727	0.22	96.9
	acktr	6824.7	637	0.20	97.1
	dqn	6822.7	533	0.17	97.7
	ppo2	6828.5	721	0.25	97.4

Table 4 Comparison of DRL algorithms with CPLEX for test instances with $R = 100$

Instance	Algorithm	obj	soltime (CPU sec.)	gapdiff %	ipred %
Small	**cpx**	3179.3	6	0	-
	a2c	3179.3	11	0	100
	acktr	3179.3	12	0	100
	dqn	3179.3	7	0	100
	ppo2	3179.3	10	0	96.7
Medium	**cpx**	12209.1	7201	0.36	-
	a2c	12236.6	215	0.22	96.0
	acktr	12699.0	205	0.27	96.0
	dqn	12324.9	192	0.24	96.0
	ppo2	12691.6	201	0.21	95.0
Large	**cpx**	26864.3	7209	0.35	-
	a2c	26937.3	1012	0.27	97.5
	acktr	26945.0	978	0.30	97.6
	dqn	26930.3	882	0.24	97.1
	ppo2	26930.6	961	0.24	97.3

by a threshold level or solely based on the DRL certainty on selection. We show results for three tested partial predictions: *default*, *85%*, and *95%*. The default partial prediction only fixes items that DRL agents predict with high confidence. This is done by ordering the solution of the DRL agent according to the worthiness of items defined by the Knapsack Transformation Heuristic given in Algorithm 3. Such a solution would have the form $1, 1, 1, 1, 1, \ldots, 0, 1, 1, 0, 0, 1, \ldots, 0, 0, 0$. The *default* prediction is defined by selecting items starting from the left with the most rated until a deselected item is found and starting

from the right with the least rated item until a selected item is found. Our goal is to have the DRL agent predict only a subset of all items and to let CPLEX work on the uncertainty region of the solution. Once the values of the predicted items are fixed in the optimization model (1), we solve it by CPLEX in its default settings. Due to this, for different instance sizes, the prediction percentage changes for the *default* method (see, e.g., Table 5, *70%*, *78%* and *72%*). To predict at the *85%* and *95%* levels, we still fix a set of items from the right and left of the solution vector and then use CPLEX to solve for the remaining items. For *85%* partial prediction, we use *75%* of the predictions starting from the left of the solution vector and *10%* of the predictions starting from the right of the solution vector, while for the *95%* partial prediction, we fix *80%* of the items starting from the left and *15%* starting from the right leaving only *5%* of the total item values for CPLEX to decide.

Table 5 shows the results for three partial predictions for each instance size and each algorithm used. **PredPerc** is the percentage of items that are predicted using the DRL solution. For small instances that are solved in a few seconds, **cpx** has a time advantage, sometimes even two-fold. However, all DRL agents are able to find the optimal solution. Notice that **soltime** for all partial predictions does not increase compared to **soltime** reported in Table 2, despite solving an additional model where the partial predictions are fixed. This is because the additional time to solve each of the models with partial predictions using **cpx** is quite small, on average 0.01 CPU seconds. For medium instances, in the case of prediction at 95%, the solution time is insignificant. Therefore, no additional time is needed in addition to the solution time of the DRL algorithm. In the case of the prediction at 85%, a short period of time is required for **cpx** to solve it further, but a great improvement is seen with respect to **gapdiff**. Even on the default case, which takes the longest time among the partial predictions, an average gap of 0.001% with **cpx** is achieved in considerably lower **soltime**. For large instances, the default partial prediction not only achieves an average of 0.0005% **gapdiff** with **cpx** but also provides a slightly better solution in the case of **a2c**. Larger instances take more time to solve compared to medium instances using all methods. Among them, 85% consumes more **cpx** time, while 95% partial prediction model with fixed variables uses, on average, around 10 CPU seconds. Based on our results in Table 5, a manager would not benefit much from partial predictions at 95% but would gain at least a two-fold decrease in **gapdiff** if they decided to use partial prediction at 85%. Meanwhile, certain decision-makers who need a close-to-optimal solution and would allow more time to get a better solution might find the *default* partial prediction most useful.

Partial prediction is a useful feature that serves as a trade-off between reductions in computational time and solution gap. In situations requiring a fast solution of large problems, a partial prediction with a high percentage of fixed items can be used, while in situations demanding close-to-optimal solutions, a default partial prediction can be the preferable approach.

5.3.6 Comparison to other heuristics

We compare our DRL framework with other state-of-the-art heuristics, such as the Primal Effective Capacity Heuristic (PECH) [2] and Adaptive Fixing (AF) [7]. The algorithms considered were benchmarked against other state-of-the-art heuristics such as [16, 48, 50, 62, 68].

Both PECH and AF algorithms consider a minimization formulation; therefore, we adapt them accordingly to our maximization formulation. We do so by referring to the original papers and only mentioning the necessary changes in the algorithms. In the case of PECH, we redefine $\lfloor a \rfloor$ as the smallest integer that exceeds a real number a. In turn, this changes the way effective capacity, denoted as $\bar{y}_j$, is calculated for each item j,

Table 5 Three levels of partial prediction for four DRL algorithms and CPLEX

Instance	Algorithm	PredPerc%	obj	soltime (CPU sec.)*	gapdiff %	ipred%
		85%	359.9	8	0	97.0
	acktr	95%	359.9	8	0	97.0
		72%	359.9	5	0	99.0
		85%	359.9	5	0	98.0
Small	dqn	95%	359.9	5	0	98.0
		70%	359.9	7	0	97.6
		85%	359.9	7	0	97.6
	ppo2	95%	359.9	7	0	97.6
	cpx	-	1355.1	7206	0.26	-
		78%	1355.5	2284	0.02	97.0
		85%	1356.4	416	0.09	96.0
	a2c	95%	1358.8	159	0.27	97.0
		81%	1355.9	657	0.05	97.4
		85%	1357.9	145	0.2	96.5
	acktr	95%	1358.7	135	0.26	96.8
		75%	1355.2	2112	0.007	97.2
		85%	1356.4	373	0.11	96.6
Medium	dqn	95%	1358.9	136	0.28	97.0
		73%	1355.4	1235	0.02	96.7
		85%	1357.2	389	0.15	96.0
	ppo2	95%	1358.8	147	0.27	96.7
	cpx	-	2994.2	7209	0.25	-
		72%	2994.1	5570	−0.003	98.1
		85%	2995.2	2583	0.03	98.0
	a2c	95%	2997.1	716.2	0.09	97.4
		81%	2994.6	2823	0.01	97.6
		85%	2997.1	1730	0.09	97.3
	acktr	95%	3002.8	615	0.28	97.3
		64%	2994.3	2710	0.003	98.1
		85%	2997	1617	0.09	97.6
Large	dqn	95%	3002.6	504	0.22	97.6
		72%	2995.2	2896	0.03	97.5
		85%	2996.5	1803	0.07	97.2
	ppo2	95%	3002.9	688	0.29	97.6

* Solution time is calculated by summing up the time needed to find a solution with a DRL algorithm and the model solution with fixed predictions using CPLEX

where $j \in E = \left\{ j \,|x_j = 0,\ \forall j \right\}$. Then we select the item with the lowest cost, j^*, as $j^* = \arg\min_{j \in E} \left\{ c_j \times \bar{y}_j \right\}$, where c_j represents the cost coefficient of an item j. After j^* is computed, the algorithm update process continues, where if the set of effective capacities is empty ($E = \emptyset$), we end the heuristic; else we go back to calculating the effective capacities again.

Table 6 Comparison of DRL algorithms with CPLEX and other heuristics for test instances with $R = 10$

Instance	Algorithm	obj	soltime (CPU sec.)	gapdiff %	ipred %
Small	**cpx**	359.9	4.0	0.0	-
	dqn	359.9	5.0	0.0	97.0
	pech	376.1	0.3	4.5	86.0
	$\mathbf{af}_1$	372.4	0.1	3.5	93.0
	$\mathbf{af}_2$	365.6	0.1	1.5	93.0
Medium	**cpx**	1355.1	7206.0	0.26	-
	dqn	1358.9	136.0	0.28	96.6
	pech	1464.5	14.4	7.5	90.0
	$\mathbf{af}_1$	1368.1	2.6	0.96	97.0
	$\mathbf{af}_2$	1363.9	2.7	0.66	97.0
Large	**cpx**	2994.2	7209.0	0.25	-
	dqn	3002.6	494.0	0.22	97.8
	pech	3222.0	171.3	7.7	79.0
	$\mathbf{af}_1$	3012.2	5.5	0.60	97.0
	$\mathbf{af}_2$	3009.8	6.5	0.51	97.0

In the case of AF, we develop two different approaches, one the same as described in Bertsimas and Demir [7] ($\mathbf{af}_1$), and another where we consider a maximization formulation ($\mathbf{af}_2$). In $\mathbf{af}_2$, we use the same notation as in the original algorithm, where X_0 and X_1 denote the set of variables that are fixed at 0 and 1, respectively, and γ represents the user-specified threshold parameter. The difference between $\mathbf{af}_1$ and $\mathbf{af}_2$ is that in $\mathbf{af}_2$ we use $\gamma = 0.75$, which is decided as an analog value to $\gamma = 0.75$ in the $\mathbf{af}_1$ within the range 0 to 1 and modify the iteration process. Specifically, we update the equations to calculate X_0 and X_1 as $X_0 \longleftarrow X_0 \cup \left\{ j | x_j^{LP^C} = 0 \right\}$ and $X_1 \longleftarrow X_1 \cup \left\{ j | \gamma < x_j^{LP^C} \leq 1 \right\}$, where $x_j^{LP^C}$ is the optimal solution obtained by solving the linear-programming relaxation of the problem, LP^C (Step 3 in Figure 2 of Bertsimas and Demir [7]). In addition, we also modify the iteration process in which we update the *argmin* equation to $j^* = \arg\max \left\{ x_j^{LP^C} \right\}$ and $X_1 \longleftarrow X_1 \cup j^*$. We show the solution results for PECH and AF in Tables 6, 7, and 8 with the acronym **pech**, $\mathbf{af}_1$, and $\mathbf{af}_2$, respectively.

We solve the same set of instances with **cpx**, **dqn**, **pech**, $\mathbf{af}_1$, and $\mathbf{af}_2$. In Tables 6, 7, and 8 we show results for small, medium, and large problems averaged over ten instances for distributions $R = 10$, $R = 25$, and $R = 100$, respectively. We compare the developed heuristics with **dqn** and **cpx**, as presented in Table 2. For **cpx** we present the solution gap after a two-hour time limit, and for the other algorithms, we present the gap of their best objective from the best **cpx** solution.

As shown in Table 6, **pech**, $\mathbf{af}_1$, and $\mathbf{af}_2$ are quite fast due to their simplistic structure. For all algorithms, in terms of percentage gap difference **gapdiff**, **dqn** dominates for all different instance sizes. We notice that in small instances, **dqn** has the advantage over the others that provide optimal solutions. Interestingly, $\mathbf{af}_1$ and $\mathbf{af}_2$ seem to perform slightly better on medium and large instances, rather than on small ones. Unlike $\mathbf{af}_1$ and $\mathbf{af}_2$, **gappdif** of **pech** highly increases as we move from small to medium and large instances.

Table 7 Comparison of DRL algorithms with CPLEX and other heuristics for test instances with $R = 25$

Instance	Algorithm	obj	soltime (CPU sec.)	gapdiff %	ipred %
Small	**cpx**	818.5	4.0	0.0	-
	dqn	818.5	5.0	0.0	98.6
	pech	898.5	0.3	8.9	84.0
	$\mathbf{af}_1$	841.1	0.1	2.68	92.0
	$\mathbf{af}_2$	833.9	0.1	1.84	94.0
Medium	**cpx**	3157.6	7206.0	0.43	-
	dqn	3166.5	177.0	0.18	96.0
	pech	4040.3	14.9	21.8	84.0
	$\mathbf{af}_1$	3195.3	2.6	1.18	96.0
	$\mathbf{af}_2$	3182.3	2.7	0.77	96.0
Large	**cpx**	6811.1	7209.0	0.33	-
	dqn	6822.7	533	0.17	97.7
	pech	8326.8	182.2	18.2	88.0
	$\mathbf{af}_1$	6875.1	7.2	0.93	98.0
	$\mathbf{af}_2$	6859.2	8.7	0.70	98.0

In Table 7, we compare the experimental results with instances that have a distribution $R = 25$. As stated previously, **dqn**, even though trained from a mix of instances having different sizes and only of distribution $R = 10$, performs in the same way in terms of the solution gap **gapdiff**, preserving the closeness to **cpx** objective. In the case of **pech** we notice that the effect of shifting distributions strongly affects the solution gap **gapdiff** and slightly the solution time **soltime** as well. Differently, for $\mathbf{af}_1$ and $\mathbf{af}_2$, solving instances of different distributions does not significantly affect solution gaps **gapdiff** in any of the sizes of instances.

In Table 8, we also compare the results for instances generated with $R = 100$. Although **dqn** needs more time than the previous distributions, it still preserves the proximity to the best solution provided by **cpx**. The **pech** still shows a similar trend. Especially, it shows a huge **gapdiff** for large instances. In this distribution, $\mathbf{af}_1$ and $\mathbf{af}_2$ only show a slight increase in terms of **soltime** and **gapdiff**.

Overall, **pech**, $\mathbf{af}_1$, and $\mathbf{af}_2$ are faster than **dqn**; however, **dqn** performs the best in terms of percent deviation from the best **cpx** solution. $\mathbf{af}_1$ and $\mathbf{af}_2$ perform better as the instances get larger, while **dqn** performance does not change much with respect to the size of the instance or its distribution.

6 Discussion and future work

We present a DRL framework to solve MKP instances of different sizes and distributions. The framework consists of a heuristic to analyze and generalize MKP properties to estimate item worthiness and an unsupervised clustering algorithm based on K-means to reduce the problem size and obtain an initial feasible solution. Four different state-of-the-art RL algorithms are used to learn patterns in a two-dimensional environment where items can be selected or

Table 8 Comparison of DRL algorithms with CPLEX and other heuristics for test instances with $R = 100$

Instance	Algorithm	obj	soltime (CPU sec.)	gapdiff %	ipred %
Small	**cpx**	3179.3	6.0	0.0	-
	dqn	3179.3	7.0	0.0	100
	pech	3373.0	0.3	5.7	80.0
	$\mathbf{af}_1$	3268.7	0.1	2.73	92.0
	$\mathbf{af}_2$	3235.5	0.1	1.73	92.0
Medium	**cpx**	12209.1	7201.0	0.36	-
	dqn	12324.9	192.0	0.24	96.0
	pech	13531.7	22.1	9.7	86.0
	$\mathbf{af}_1$	12329.1	3.1	0.97	96.0
	$\mathbf{af}_2$	12293.1	3.5	0.68	96.0
Large	**cpx**	26864.3	7209.0	0.35	-
	dqn	26930.3	882.0	0.24	97.1
	pech	41922.5	194.8	35.91	79.0
	$\mathbf{af}_1$	27102.5	8.9	0.88	97.0
	$\mathbf{af}_2$	27025.2	10.1	0.60	98.0

deselected. Our design is based on a lot of testing while identifying fast and efficient ways to feed the main RL learning process.

Our results show that COPs can highly benefit from the power of deep learning methodologies. Specifically, reformulating hard problems into convenient general deep learning environments allows one to generalize over the solution of a broad class of problems, such as MKP. Based on our experiments, DRL agents can learn and generalize solution strategies for the MKP. Furthermore, commercial solvers can benefit from the computational power of deep learning and form hybrid frameworks that reflect the best side of each methodology.

As a general framework, future work can include improvements anywhere in the framework. For example, another way can be found to gain an initial solution and an objective value faster. A more accurate representation of the MKP, rather than using the heuristic, could lead to an improvement in time and accuracy. Future improvements could be introduced to the RL parts. The reward function is a key point in training; therefore, a multi-objective reward function could be designed to look at different aspects of a solution, such as a gap, feasibility, or small violations. A new DRL algorithm designed specifically for the MKP environment can improve sequential decision making and also contribute to faster and more accurate prediction. Lastly, the developed MKP environment can be further extended to incorporate stochasticity in sequential decision making.

Acknowledgements We gratefully acknowledge the support of the National Science Foundation CAREER Award co-funded by the CBET/ENG Environmental Sustainability program and the Division of Mathematical Science in MPS/NSF under Grand No. CBET-1554018.

Data Availability The data and the codes are publicly available on this GitHub website: RL for MKP Framework Repository (https://github.com/sbushaj93/rl-mkp-framework).

Appendix A: Heuristic Transformation

To facilitate the multidimensional knapsack problem for usage in the DRL algorithm, we present a heuristic that evaluates the items and sorts them based on their importance, considering their contribution to the objective function and the feasibility of the constraints.

We build our analysis by a worthiness formulation considering the effect of cost values, weights of each item, and the sizes of each knapsack. Despite many different heuristic approaches in the literature used to solve knapsack and multi-dimensional knapsack problems [8, 56], we develop a new heuristic used before the DRL algorithm to improve its performance. We perform experiments with multiple heuristics, but eventually our results show that the normalization procedure sets a ground truth that the DRL agent picks up during training. In our heuristic, we consider every component of the problem that affects the decision regarding a certain item, such as the item cost, the item weight for each constraint, the respective right-hand-side value, and normalize all parameters used. We only aim to transform our multidimensional knapsack into a one- or two-dimensional vector representation. We do not consider any duality properties or multipliers, but only consider proportions between the problem components.

Algorithm 3 shows the step-by-step procedures of the heuristic algorithm. Initially, we calculate the item worth for each item in the knapsack and store them in a vector. Then in a second procedure, we sort them in ascending order, but we maintain their original position. We use some utility methods to convert from the sorted items to the original problem.

Algorithm 3 Knapsack Transformation Heuristic

Procedure: Calculate Item Worth
Input: c_j, a_{ij}, b_i {*Item cost, item weight, and constraint i right-hand side.*}
Output: Ψ {*The item worth set.*}
for $j \in \mathcal{J}$ **do** {for each item}
  for $i \in \mathcal{I}$ **do** {for each knapsack}
    $r_j = +\dfrac{c_j}{a_{ij}/b_i}$ {*Calculate worthiness ratio.*}
  end for
  append $\dfrac{r_j}{|\mathcal{I}|}$ **to** Ψ
end for

Procedure: Sort Elements According to Item Worth
Input: Ψ {*Item worth list.*}
Output: S, S' {*S - sorted values, S' keeps re-ordered indices for items according to worthiness ratio.*}
$S, S' \longleftarrow$ sortAscending(Ψ)

References

1. Afshar, R.R., Zhang, Y., Firat, M., Kaymak, U.: A state aggregation approach for solving knapsack problem with deep reinforcement learning. In: Asian Conference on Machine Learning, pp. 81–96. PMLR (2020)
2. Akçay, Y., Li, H., Xu, S.H.: Greedy algorithm for the general multidimensional knapsack problem. Ann. Oper. Res. **150**(1), 17–29 (2007)

3. Balas, E., Martin, C.H.: Pivot and complement-a heuristic for 0–1 programming. Manag. Sci. **26**(1), 86–96 (1980)
4. Balev, S., Yanev, N., Fréville, A., Andonov, R.: A dynamic programming based reduction procedure for the multidimensional 0–1 knapsack problem. Eur. J. Oper. Res. **186**(1), 63–76 (2008)
5. Barrett, T., Clements, W., Foerster, J., Lvovsky, A.: Exploratory combinatorial optimization with reinforcement learning. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 34(04), pp. 3243–3250 (2020)
6. Bello, I., Pham, H., Le, Q.V., Norouzi, M., Bengio, S.: Neural combinatorial optimization with reinforcement learning. CoRR arXiv:1611.09940 (2016)
7. Bertsimas, D., Demir, R.: An approximate dynamic programming approach to multidimensional knapsack problems. Manag. Sci. **48**(4), 550–565 (2002)
8. Boyer, V., Elkihel, M., El Baz, D.: Heuristics for the 0–1 multidimensional knapsack problem. Eur. J. Oper. Res. **199**(3), 658–664 (2009)
9. Bushaj, S., Büyüktahtakın, İE., Haight, R.G.: Risk-averse multi-stage stochastic optimization for surveillance and operations planning of a forest insect infestation. Eur. J. Oper. Res. **299**(3), 1094–1110 (2022)
10. Bushaj, S., Büyüktahtakın, İE., Yemshanov, D., Haight, R.G.: Optimizing surveillance and management of emerald ash borer in urban environments. Nat. Resour. Model. **34**(1), e12267 (2020)
11. Bushaj, S., Yin, X., Beqiri, A., Andrews, D., Büyüktahtakın, İE.: A simulation-deep reinforcement learning (sirl) approach for epidemic control optimization. Ann. Oper. Res. **328**(1), 245–277 (2023)
12. Büyüktahtakın, İE.: Stage-t scenario dominance for risk-averse multi-stage stochastic mixed-integer programs. Ann. Oper. Res. **309**(1), 1–35 (2022)
13. Büyüktahtakın, İE.: Scenario-dominance to multi-stage stochastic lot-sizing and knapsack problems. Comput. Oper. Res. **153**, 106149 (2023)
14. Caprara, A., Kellerer, H., Pferschy, U., Pisinger, D.: Approximation algorithms for knapsack problems with cardinality constraints. Eur. J. Oper. Res. **123**(2), 333–345 (2000)
15. Chen, W., Xu, Y., Wu, X.: Deep reinforcement learning for multi-resource multi-machine job scheduling. arXiv preprint arXiv:1711.07440 (2017)
16. Chu, P.C., Beasley, J.E.: A genetic algorithm for the multidimensional knapsack problem. J. Heurist. **4**(1), 63–86 (1998)
17. Dai, H., Dai, B., Song, L.: Discriminative embeddings of latent variable models for structured data. CoRR arXiv:1603.05629 (2016)
18. Dai, H., Khalil, E.B., Zhang, Y., Dilkina, B., Song, L.: Learning combinatorial optimization algorithms over graphs. CoRR arXiv:1704.01665 (2017)
19. Delarue, A., Anderson, R., Tjandraatmadja, C.: Reinforcement learning with combinatorial actions: an application to vehicle routing. arXiv preprint arXiv:2010.12001 (2020)
20. Dobson, G.: Worst-case analysis of greedy heuristics for integer programming with nonnegative data. Math. Oper. Res. **7**(4), 515–531 (1982)
21. Etheve, M., Alès, Z., Bissuel, C., Juan, O., Kedad-Sidhoum, S.: Reinforcement learning for variable selection in a branch and bound algorithm. In: International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research, pp. 176–185. Springer (2020)
22. Eysenbach, B., Gupta, A., Ibarz, J., Levine, S.: Diversity is all you need: Learning skills without a reward function. arXiv preprint arXiv:1802.06070 (2018)
23. Fox, G.E., Scudder, G.D.: A heuristic with tie breaking for certain 0–1 integer programming models. Nav. Res. Logist. Q. **32**(4), 613–623 (1985)
24. Fréville, A., Plateau, G.: An exact search for the solution of the surrogate dual of the 0–1 bidimensional knapsack problem. Eur. J. Oper. Res. **68**(3), 413–421 (1993)
25. Frieze, A., Clarke, M.: Approximation algorithms for the m-dimensional 0–1 knapsack problem: Worst-case and probabilistic analyses. Eur. J. Oper. Res. **15**(1), 100–109 (1984)
26. Gaspar, D., Lu, Y., Song, M.S., Vasko, F.J.: Simple population-based metaheuristics for the multiple demand multiple-choice multidimensional knapsack problem. Int. J. Metaheurist. **7**(4), 330–351 (2020)
27. Gavish, B., Pirkul, H.: Efficient algorithms for solving multiconstraint zero-one knapsack problems to optimality. Math. Program. **31**(1), 78–105 (1985)
28. Gavish, B., Pirkul, H.: Computer and database location in distributed computer systems. IEEE Trans. Comput. **35**(7), 583–590 (1986)
29. Glover, F., Kochenberger, G.A.: Critical event Tabu search for multidimensional knapsack problems. In: Meta-heuristics, pp. 407–427. Springer (1996)
30. Goodfellow, I., Bengio, Y., Courville, A., Bengio, Y.: Deep Learning, vol. 1. MIT Press, Cambridge (2016)
31. Gu, S., Hao, T., Yao, H.: A pointer network based deep learning algorithm for unconstrained binary quadratic programming problem. Neurocomputing **390**, 1–11 (2020)

32. Hanafi, S., Freville, A.: An efficient tabu search approach for the 0–1 multidimensional knapsack problem. Eur. J. Oper. Res. **106**(2–3), 659–675 (1998)
33. Haul, C., Voss, S.: Using surrogate constraints in genetic algorithms for solving multidimensional knapsack problems. In: Advances in Computational and Stochastic Optimization, Logic Programming, and Heuristic Search, pp. 235–251. Springer (1998)
34. Hillier, F.S.: Efficient heuristic procedures for integer linear programming with an interior. Oper. Res. **17**(4), 600–637 (1969)
35. Hu, H., Zhang, X., Yan, X., Wang, L., Xu, Y.: Solving a new 3d bin packing problem with deep reinforcement learning method. arXiv preprint arXiv:1708.05930 (2017)
36. Hubbs, C.D., Perez, H.D., Sarwar, O., Sahinidis, N.V., Grossmann, I.E., Wassick, J.M.: Or-gym: A reinforcement learning library for operations research problem. arXiv preprint arXiv:2008.06319 (2020)
37. Jaccard, P.: The distribution of the flora in the alpine zone. 1. New Phytol. **11**(2), 37–50 (1912)
38. Kellerer, H., Pferschy, U., Pisinger, D.: Multidimensional knapsack problems. In: Knapsack Problems, pp. 235–283. Springer (2004)
39. Kong, W., Liaw, C., Mehta, A., Sivakumar, D.: A new dog learns old tricks: Rl finds classic optimization algorithms. In: Proceedings of International Conference on Learning Representations, pp. 1–25 (2019)
40. Kool, W., Van Hoof, H., Welling, M.: Attention, learn to solve routing problems! Proceedings of International Conference on Learning Representations **3499**, 3508 (2019)
41. Kwon, Y.-D., Choo, J., Kim, B., Yoon, I., Gwon, Y., Min, S.: Pomo: Policy optimization with multiple optima for reinforcement learning. Adv. Neural. Inf. Process. Syst. **33**, 21188–21198 (2020)
42. Lee, J.S., Guignard, M.: Note-an approximate algorithm for multidimensional zero-one knapsack problems-a parametric approach. Manag. Sci. **34**(3), 402–410 (1988)
43. Li, F., Hu, B.: Deepjs: Job scheduling based on deep reinforcement learning in cloud data center. In: Proceedings of the 2019 4th International Conference on Big Data and Computing, pp. 48–53 (2019)
44. Li, Y.: Deep reinforcement learning: an overview. arXiv preprint arXiv:1701.07274 (2017)
45. Liao, H., Zhang, W., Dong, X., Poczos, B., Shimada, K., Burak Kara, L.: A deep reinforcement learning approach for global routing. J. Mech. Des. 142(6) (2020)
46. Lloyd, S.: Least squares quantization in pcm. IEEE Trans. Inf. Theory **28**(2), 129–137 (1982)
47. Lorie, J.H., Savage, L.J.: Three problems in rationing capital. J. Bus. **28**, 229–229 (1955)
48. Loulou, R., Michaelides, E.: New greedy-like heuristics for the multidimensional 0–1 knapsack problem. Oper. Res. **27**(6), 1101–1114 (1979)
49. Ma, Q., Ge, S. He, D., Thaker, D., Drori, I.: Combinatorial optimization by graph pointer networks and hierarchical reinforcement learning. arXiv preprint arXiv:1911.04936 (2019)
50. Magazine, M., Oguz, O.: A heuristic algorithm for the multidimensional zero-one knapsack problem. Eur. J. Oper. Res. **16**(3), 319–326 (1984)
51. Mansini, R., Speranza, M.G.: Coral: An exact algorithm for the multidimensional knapsack problem. INFORMS J. Comput. **24**(3), 399–415 (2012)
52. Mnih, V., Badia, A.P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., Kavukcuoglu, K.: Asynchronous methods for deep reinforcement learning. In: International Conference on Machine Learning, pp. 1928–1937. PMLR (2016)
53. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.: Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602 (2013)
54. Nazari, M., Oroojlooy, A., Snyder, L., Takác, M.: Reinforcement learning for solving the vehicle routing problem. In: Advances in Neural Information Processing Systems, pp. 9839–9849 (2018)
55. Nomer, H.A., Alnowibet, K.A., Elsayed, A., Mohamed, A.W.: Neural knapsack: a neural network based solver for the knapsack problem. IEEE Access **8**, 224200–224210 (2020)
56. Pirkul, H.: A heuristic solution procedure for the multiconstraint zero-one knapsack problem. Nav. Res. Logist. **34**(2), 161–172 (1987)
57. Pisinger, D.: A minimal algorithm for the 0–1 knapsack problem. Oper. Res. **45**(5), 758–767 (1997)
58. Pontrandolfo, P., Gosavi, A., Okogbaa, O.G., Das, T.K.: Global supply chain management: a reinforcement learning approach. Int. J. Prod. Res. **40**(6), 1299–1317 (2002)
59. Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., Dormann, N.: Stable-baselines3: Reliable reinforcement learning implementations. J. Mach. Learn. Res. **22**(268), 1–8 (2021)
60. Schaul, T., Quan, J., Antonoglou, I., Silver, D.: Prioritized experience replay. arXiv preprint arXiv:1511.05952 (2015)
61. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347 (2017)
62. Senju, S., Toyoda, Y.: An approach to linear programming with 0-1 variables. Manag. Sci. B196–B207 (1968)

63. Shehab, M., Khader, A.T., Alia, M.A.: Enhancing cuckoo search algorithm by using reinforcement learning for constrained engineering optimization problems. In 2019 IEEE Jordan international joint conference on electrical engineering and information technology (JEEIT), pp. 812–816. IEEE (2019)
64. Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al.: A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. Science **362**(6419), 1140–1144 (2018)
65. Tang, Y., Agrawal, S., Faenza, Y.: Reinforcement learning for integer programming: Learning to cut. In International Conference on Machine Learning, pp. 9367–9376. PMLR (2020)
66. Thesen, A.: Scheduling of computer programs in a multiprogramming environment (1974)
67. Thesen, A.: A recursive branch and bound algorithm for the multidimensional knapsack problem. Nav. Res. Logist. Q. **22**(2), 341–353 (1975)
68. Toyoda, Y.: A simplified algorithm for obtaining approximate solutions to zero-one programming problems. Manag. Sci. **21**(12), 1417–1427 (1975)
69. Vasquez, M., Hao, J.-K.: A hybrid approach for the 0-1 multidimensional knapsack problem. In: IJCAI, pp. 328–333 (2001)
70. Vasquez, M., Vimont, Y.: Improved results on the 0–1 multidimensional knapsack problem. Eur. J. Oper. Res. **165**(1), 70–81 (2005)
71. Vazirani, V.V.: Approximation Algorithms. Springer, Berlin (2013)
72. Verma, R., Singhal, A., Khadilkar, H., Basumatary, A., Nayak, S., Singh, H.V., Kumar, S., Sinha, R.: A generalized reinforcement learning algorithm for online 3d bin-packing. arXiv preprint arXiv:2007.00463 (2020)
73. Vinyals, O., Fortunato, M., Jaitly, N.: Pointer networks. arXiv preprint arXiv:1506.03134 (2015)
74. Weingartner, H.M.: Capital budgeting of interrelated projects: survey and synthesis. Manag. Sci. **12**(7), 485–516 (1966)
75. Weingartner, H.M., Ness, D.N.: Methods for the solution of the multidimensional 0/1 knapsack problem. Oper. Res. **15**(1), 83–103 (1967)
76. Woeginger, G.J.: Exact algorithms for np-hard problems: a survey. In: Combinatorial Optimization-Eureka, You Shrink!, pp. 185–207. Springer (2003)
77. Wu, Y., Mansimov, E., Grosse, R.B., Liao, S., Ba, J.: Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. Adv. Neural. Inf. Process. Syst. **30**, 5279–5288 (2017)
78. Yang, Yan, Shengjian Liu, Y.Z.: Greedy binary lion swarm optimization algorithm for solving multidimensional knapsack problem. J. Comput. Appl. **40**(5), 1291–1294 (2020)
79. Yang, M.-H.: An efficient algorithm to allocate shelf space. Eur. J. Oper. Res. **131**(1), 107–118 (2001)
80. Yang, Y., Rajgopal, J.: Learning combined set covering and traveling salesman problem. arXiv preprint arXiv:2007.03203 (2020)
81. Yilmaz, D., Büyüktahtakın, İ.E.: An expandable learning-optimization framework for sequentially dependent decision-making. Eur. J. Oper. Res. **314**(1), 280–296 (2024). https://doi.org/10.1016/j.ejor.2023.10.045
82. Yilmaz, D., Büyüktahtakın, İ.E.: Learning optimal solutions via an LSTM-optimization framework. Oper. Res. Forum **4**(2), 28 (2023)
83. Yin, X., Büyüktahtakın, İE.: Risk-averse multi-stage stochastic programming to optimizing vaccine allocation and treatment logistics for effective epidemic response. IISE Trans. Healthc. Syst. Eng. **12**(1), 52–74 (2022)
84. Yin, X., Büyüktahtakın, İE., Patel, B.: COVID-19: Data-driven optimal allocation of ventilator supply under uncertainty and risk. Eur. J. Oper. Res. **304**(1), 255–275 (2023)
85. Yilmaz, Dogacan and Büyüktahtakın, İEsra.: A deep reinforcement learning framework for solving two-stage stochastic programs. Optimization Letters, 1–28 (2023)