IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received November 17, 2020, accepted December 5, 2020, date of publication December 11, 2020,
date of current version December 29, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3044005

Neural Knapsack: A Neural Network Based Solver for the Knapsack Problem

HAZEM A. A. NOMER[1,4], KHALID ABDULAZIZ ALNOWIBET[3], ASHRAF ELSAYED[4,5], AND ALI WAGDY MOHAMED[1,2]

[1]Wireless Intelligent Networks Center (WINC), School of Engineering and Applied Sciences, Nile University, Giza 16453, Egypt
[2]Operations Research Department, Faculty of Graduate Studies for Statistical Research, Cairo University, Giza 12613, Egypt
[3]College of Science, Department of Statistics and Operations Research, King Saud University, Riyadh 11564, Saudi Arabia
[4]Department of Mathematics and Computer Science, Faculty of Science, Alexandria University, Alexandria 21527, Egypt
[5]Faculty of Computer Science and Engineering, Al Alamein International University, El Alamein 21527, Egypt

Corresponding author: Ali Wagdy Mohamed (aliwagdy@gmail.com)

This work was supported by a Research Supporting Program at King Saud University, Riyadh, Saudi Arabia.

ABSTRACT This paper introduces a heuristic solver based on neural networks and deep learning for the knapsack problem. The solver is inspired by mechanisms and strategies used by both algorithmic solvers and humans. The neural model of the solver is based on introducing several biases in the architecture. We introduce a stored memory of vectors that holds up items representations and their relationship to the capacity of the knapsack and a module that allows the solver to access all the previous outputs it generated. The solver is trained and tested on synthetic datasets that represent a variety of instance types with different complexities. The solver neural model capabilities to generalize were tested on instances with up to 200 items, the model succeed to obtain near optimal solutions better than the greedy algorithm for instances in which there exists a correlation between items values and weights. The results also show that the capacity of the knapsack has a role in learning useful representations for each item in an instance and for the instance itself. Although the proposed solver may be not superior to other solvers, the results described here are insights for how the connection between combinatorial optimization, machine learning, and cognitive science could serve a great purpose in the operation research field. The goal of this work is not to design a state of the art solver, rather it full-fills some of the holes in the recent line of research that incorporates learning in combinatorial optimization problems.

INDEX TERMS Knapsack, neural networks, machine learning, combinatorial optimization.

I. INTRODUCTION

Under the umbrella of the No Free Lunch Theorem for Optimization [1] and in the setting of combinatorial optimization (CO), a machine learning model trained using input-output examples of a specific CO problem instance as travelling salesman problem (TSP) would perform better than another model trained using input-output examples of general integer linear programming instances. This because the learnt model would exploit both the structure and the underlying statistical distribution of the data that its trained on, and thus performs better than other heuristic algorithms on these problems.

There are two main reasons why incorporating neural networks in the CO setting could be effective. Firstly, the capability of neural networks to learn from structured data as graphs and sequences without the need of reprocessing or reformulating the problem. Secondly, neural networks specially deep ones have the capability to learn an approximate algorithm from data as shown in literature where neural networks with augmented memory modules are able to solve simple algorithmic tasks with mastery [2]–[4].

We focus on the use of neural networks, deep learning and representation learning to learn a neural model that can solve the knapsack problem (KP) from input-output examples in supervised fashion. This paper introduces a framework for tackling the KP that could be applied to other CO problems. The framework is used to design a solver using the following components:

1) Analysis of Decision Variables: as a desideratum step when using machine learning in a task, we need to understand the data distribution in hand. However since data generation of some CO problem instances is hard, it is required to analyze the distribution of solutions and corner case instances.
2) Representation Learning: The heart of our solver is a deep neural network which learns useful

The associate editor coordinating the review of this manuscript and approving it for publication was Danilo Pelusi.

representations for each item individually and for a problem. The network is based on sequence-to-sequence models, and is trained using input-output examples in supervised fashion.

3) Search/sampling Algorithm: There is no guarantee that the solver will produce feasible solutions for all instances. A search or a sampling technique should be used to constrain solutions to be feasible. As the model output is a probability distribution over decisions for each item, we use greedy decoding to generate solutions and only use search to modify infeasible ones.

The architecture is inspired by how the KP is solved by human solvers, exact and heuristic algorithms. The model is based on three ingredients: memory, attention over the memory and accessing information from solution attempts by the solver.

II. RELATED WORK

The knapsack problem has a distinguished history as a problem solved by many metaherustic algorithms [5] as monarch butterfly optimization (MBO), earthworm optimization algorithm (EWA), elephant herding optimization (EHO), moth search (MS) algorithm, and Differential Evolution (DE) [6], and [7]. Recent work uses MBO to solve both medium scale and large scale 0-1 Knapsack problem [8], [9], and [10]. The moth search algorithm was used to solve an extended version of the classical 0-1 knapsack problem (0-1 KP) called discounted knapsack [11] and for Set-Union knapsack problem [12]. Global firefly algorithm (GFA) was proposed for solving randomized time-varying knapsack problems in [13]. Combination of search techniques was used as cuckoo search algorithm with global harmony search to 0-1 KP [14]. However, the scope of this paper is the use of machine learning to design solvers for the KP problem.

The capability and computational power of neural networks have been questioned through the history of Artificial Intelligence. Many views expressed that although neural networks seems to be powerful models or the ultimate road to Artificial General Intelligence, they have failed many simple tasks. However the idea of very powerful deep models that resemble the human brain came [15]. Deep Learning has succeed in many fields as computer vision, natural language processing, and speech recognition.

Using neural networks for solving combinatorial optimization has intersection between fields as: optimization, machine learning and program induction and learning algorithmic tasks. The task of solving a combinatorial optimization problem is considered an algorithmic task to find an optimal solution. The algorithm or the program can be induced or learned by a learning model.

Incorporating neural networks in combinatorial optimization has a distinguished history, where the majority of research focuses on the traveling salesman problem [16]. One of the earliest approaches to tackle CO is the use of Hopfield networks [17] for the TSP. The network's energy function was modified to make it equivalent to the TSP objective and use Lagrange multipliers to penalize the violations of the problem's constraints. A limitation of this approach is that it is sensitive to hyperparameters and parameter initialization. Overcoming this limitation is central to the subsequent work, especially by [18] and [19].

The most prominent work is the invention of Elastic Nets as a means to solve the TSP [20], and the application of Self Organizing Map to the TSP [21], [22]. This work is an example on using deformable template models to solve the TSP.

Even though these neural networks have many appealing properties, they are still limited as a research work. When being carefully benchmarked, they have not yielded satisfying results compared to algorithmic methods [23]. Perhaps due to the negative results, this research direction is largely overlooked through centuries.

Recent approaches that tackle CO problems with neural networks either train a neural network on input-output examples of CO problem instances, or train a network with reinforcement learning where the network acts as an agent that learn a decision policy using a reward signal where there is no expert involved. In both cases, the learned model can approximate a certain decision in CO algorithm (as branching decision in branch-and-bound algorithm) or learn the algorithm from scratch usually by finding representations from problem raw inputs.

We highlight on recent research that used pointer networks [24] to solve the TSP using input-output examples in supervised fashion. In [25], the authors used neural networks and reinforcement learning to tackle combinatorial optimization problems two approaches based on policy gradients were used. The first approach, called RL pretraining, uses a training set to optimize a recurrent neural network (RNN) that parameterizes a stochastic policy over solutions, using the expected reward as objective. At test time, the policy is fixed, and one performs inference by greedy decoding or sampling. The second approach, called active search, involves no pretraining. It starts from a random policy and iteratively optimizes the RNN parameters on a single test instance, again using the expected reward objective. The two approaches were applied on both the TSP and the KP. This approach has a limitation when applied to instances with large input sizes (number of cities in TSP or number of items in KP), that it needs additional search strategies or retraining to yield feasible or optimal solutions.

In [26], the authors introduce "Divide and Conquer Network" that can discover and exploit scale invariance in discrete algorithmic tasks, and can be trained with weak supervision. The model learns how to split large inputs recursively, then learns how to solve each subproblem and finally how to merge partial solutions. The approach was applied on knapsack instances with variable capacity. When applied to knapsack instances the split module outputs a probability distribution over items and only a fraction of the capacity is filled and the remaining items are fed back to the split module and this process is repeated a number of times except for the last time where the capacity is filled completely. While this

insure that solutions are feasible, it is still requires to specify the fraction of capacity and number of times where the sack is filled.

III. THE KNAPSACK PROBLEM

Given a set of N items numbered from 1 to n each has a value v_i and weight w_i. A maximum weight capacity for the sack is denoted C. The 0-1 knapsack problem is defined as:

$$\begin{aligned} &\text{maximize} \sum_{i=1}^{N} v_i x_i \\ &\text{subject to} \sum_{i=1}^{N} w_i x_i \leq C \\ &x_i \in \{0, 1\} \end{aligned}$$

where v_i, w_i, C are all positive integers. There are other versions of the KP that do not restrict the number of copies of each item to be only one. [27], [28]. Informally, the problem is stated as: Given a set of items, each with a weight and a value, determine which items to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

The most known approach for solving the KP is dynamic programming (DP), which has a pseudo-polynomial time complexity. The time complexity of DP is $O(Cn)$. The capacity C, unlike n, is not polynomial in the length of the input to the problem. The length of C is proportional to the number of bits in C. This means that as C increases the time complexity increases.

The common heuristic approach to solve the KP is the greedy approach [29]. It begins by sorting the items in decreasing order of value per unit of weight, and then proceeds to insert them into the knapsack. The greedy algorithm is far from optimal if only one copy of each item is selected.

IV. NEURAL KNAPSACK SOLVER

A. HUMAN AND ALGORITHMIC INSPIRATIONS

In a naive way, at least two main steps are required to solve the KP: one step that encapsulates both items values and weights (as calculating values to weights ratio and sorting items based on this ratio in the greedy algorithm). The second step ensures that knapsack weight does not exceeds the capacity C as we include items.

These steps guarantee a feasible solution, but one or more additional steps are required to achieve an optimal solution. Usually this step requires from the solver to explore different solutions as in the branch and bound algorithm where the algorithm explores branches of a tree, which represent subsets of the solution. Before enumerating the candidate solutions of a branch, the branch is checked against upper and lower estimated bounds on the optimal solution, and the branch is pruned if it can not produce an optimal solution.

In case of dynamic programming, the idea is to break down a complex problem into smaller sub-problems and store the results of sub-problems, so that they are not re-computed when needed later. Dynamic programming requires additional memory during exploration of the sub-problems and the memory can grows large as the capacity C increases.

In [30], human participants were asked to solve KP instances. Human solvers strategies in this study were found to be similar to the greedy algorithm at the early stages of solution attempts combined with a strategy that is a complement but not totally error free. The results indicate that problem-solving ability was limited by biological constraints including limited working and episodic memories as well as difficulties with arithmetic. Success rates decreased with more complex instances (that requires more computational steps to find an optimal solution). Working memory was found to be a key to solve more complex instances.

B. PROBLEM REPRESENTATION

CO problems does not naturally arises as sequence transduction problems. However representing the items in the KP sequentially exploits the use of weight-sharing models as recurrent neural networks (RNNs) which helps in generalization. A problem arises with this representation, that there is no interdependence or context between items in KP as in other sequence problems as machine translation. There is no relation between the values and weights of an item i and item j, and the optimal solution does not depend on this relation.[1] There is only a relation between the decisions for items, because including or excluding an item would affect both the infeasiablity and the optimality of the solution.

C. SOLVER NEURAL ARCHITECTURE

The model is based on sequence-to-sequence models with attention [31]–[33] and memory networks [3], [4], [34], [35]. In end-to-end fashion the sequence to sequence model estimates the probability distribution:

$$p(y_1, y_2, \ldots, y_t | x_1, x_2, \ldots, x_t) = \prod_{i=1}^{t} p(y_t | \boldsymbol{h^e}, y_1 \ldots y_{t-1}; \boldsymbol{\theta}) \tag{1}$$

The parameters of the model are learnt by maximizing the conditional probabilities for the training set [24], [31].

The model is composed of an encoder network that encodes an KP instance into a single vector containing a representation for all items information of this instance, and a decoder that generates a probability distribution over decisions for each item.

Another neural network, denoted as the memory constructor M_c, is used to build a memory $\boldsymbol{M}$ that stores a vector representation for each individual item in an instance. To exploit these representations and read memory contents, the decoder uses an attention mechanism for addressing the memory only using focusing-by-content.

The decoder also depends on the representation of the entire previous outputs $[y_1, y_2, \ldots, y_{t-1}]$ obtained by a separate module called the knapsack handler, instead of the last

[1] Only when sorting items values and weights, a relation between values of items and their weights appears that may inform a solver which items to include or which items to exclude from the sack.

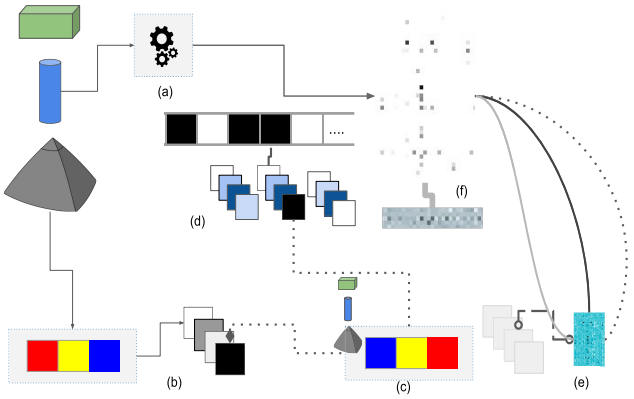


FIGURE 1. The solver architecture is composed of four modules: (a) items are first introduced to the memory constructor which produce a representation for each item, (b) the encoder receives items to produce fixed vector for the instance, (c) the decoder receives both the items and the knapsack handler representation. We introduce a new module (d) the knapsack handler that produces a representation for the state of the sack using the outputs of the decoder. The decoder produces a hidden state that is latter used to produce reading weights over memory (e), and finally dot product between the reading weights and the memory produces a read vector (f).

previous output y_{t-1}. This to provide access to the model to full-state of the sack as both human solvers and computational solvers require or do.

The knapsack handler does not allow the "backtracking" feature included in branch-and-bound algorithms, however providing a separate representation from decoder hidden state for all the previous states or outputs is indeed useful, especially when its essential for the problem structure as in case of the knapsack.

Each item is represented as vector $\boldsymbol{x_t} = (v_i, w_i, C)$ where v_i and w_i are the value and weight of the item i respectively. The vector $\boldsymbol{x_t}$ is an input fed to the encoder, memory constructor and decoder.

We choose to wire the notion of capacity directly in the model in attempt to learn better representations for the instance and its items, that would help the model to generalize to more complex KP instances.

1) MEMORY CONSTRUCTOR AND ENCODER

The memory constructor is a neural network that produces a vector representation for each item separately.

The encoder produces a representation $\boldsymbol{h_e}$ for the entire sequence of items. The encoder representation is different from memory representations, the fixed vector $\boldsymbol{h_e}$ contains information about the relations between all items. A memory representation, on the other side, encapsulates information in an individual item and weight value.

The memory is constructed dynamically and its size depends on the number of items.

$$\begin{aligned} \boldsymbol{m_t} &= M_c(\boldsymbol{x_t}; \boldsymbol{\theta_C}) \\ \boldsymbol{M} &= [\boldsymbol{m_i}, i = 1, 2, \ldots t] \end{aligned} \tag{2}$$

Each vector $\boldsymbol{m_i}$ would contain at least information related to the weights w and capacity C and the relation between the value of an item v_i and w_i.

2) KNAPSACK HANDLER

A solver for the KP problem, would require access to the state of the sack to know the solution for the next item or to backtrack and modify previous solutions. Motivated by this, we introduce a module in the model that produces a representation for the state of the sack that the decoder can use.

At each time step, the module produces an new representation of the sack $\boldsymbol{h_t^k}$ which encapsulates the decisions of the model.

The knapsack handler receives at each time step t a vector $[y_1, y_2, \ldots, y_{t-1}]$, each y_i is a probability distribution over $\{0, 1\}$ describing the likelihood of including the item in the sack.

$$\boldsymbol{h_t^k} = K([y_1, y_2, \ldots, y_{t-1}]; \boldsymbol{\theta_K}) \tag{3}$$

The knapsack handler adds a layer of complexity to the model, however it increases the amount information passed to the decoder and its up to decoder to determine the significance of this information during learning [36], [37]. The knapsack handler is a form of bias in the model inspired by the KP problem and can be applied in general to other CO problems.

After adding the representation of the sack $\boldsymbol{h^K}$, the model estimates the following probability distribution:

$$p(y_1, y_2, \ldots, y_t | x_1, x_2, \ldots, x_t) = \prod_{i=1}^{T} p(y_t | \boldsymbol{x_t}, \boldsymbol{h_e}, \boldsymbol{h_t^K}) \tag{4}$$

3) ATTENTION OVER MEMORY

The decoder receives both the representation of the sack and the item vector $\boldsymbol{x_t}$ at each time step. This makes the decoder acts separately form the encoder and does not rely on its representation completely to generate solutions.

Before generating outputs, the decoder state is compared with each memory location i. To compare the decoder state with a memory vector $\boldsymbol{m_i}$, cosine similarity measure is used. The similarity scores determine a weighting that can be used to attend to memory locations. This a form of content based addressing proposed in [3] and [4].

Instead of using a special key emitted from the decoder, the decoder state is used directly:

$$\boldsymbol{s_t} = S_c(\boldsymbol{h_t^d}, \boldsymbol{m_i}), \quad i = 1, 2, \ldots, t \tag{5}$$

where S_c is the cosine proximity between two vectors. The weights $\boldsymbol{w_t}$ over memory locations are produced by applying the softmax function over the scores $\boldsymbol{s_t}$.

$$\boldsymbol{w_t} = softmax(\boldsymbol{s_t}) \tag{6}$$

Reading happens by taking the dot product between $\boldsymbol{w_t}$ and memory contents $\boldsymbol{M}$ to produce one read vector $\boldsymbol{r_t}$.

$$\boldsymbol{r_t} = \sum_{t=1}^{n} \boldsymbol{w_t} \boldsymbol{h_t^d} \tag{7}$$

D. GENERATING FINAL SOLUTIONS

The model produces a decision for each item as a probability distribution over $\{0, 1\}$. One could sample from this probability distribution or use search algorithms to select the most likely items [25], [38], [39].

We used the greedy approach to decode the most likely items. Since the model could produce infeasible solutions, to prune these solutions the items are sorted in descending order according to their weights and the decisions produced by the model are flipped until the capacity constrain is satisfied.

This approach ensures that no feasible solution occurs for any instance. The approach could be modified to search over all possible solutions (according to the probability distribution) and select the most optimal one.

V. DATA GENERATION

It is a complex effort to generate CO problems that capture the essence of real applications [40]. We adapt previous considerations on generating CO problem instances on two sides:

1) Data is generated using random generation procedure based on careful analysis of problem instances. We generate instances of knapsack, these instances are then solved by an exact solver and the decision variables produced by the solver are the targets during learning. These decision variables distributions are analyzed carefully.
2) The capacities for all instances in both training and testing sets are varied carefully to represent a wide variety of complex knapsack problems. The capacity in an instance depends on the sum of weights in each instance.

A. GENERATION PROCEDURE

Both [25] and [26] in their generated knapsack instances considered the values and weights to be uniformly distributed over [0, 1]. The work of [25] fixed the capacity in each instance, while in [26] the capacities were uniformly distributed over $[0.2\ n, 0.3\ n]$ where n is the number of items.

We adapt the approach in [41] and [42]. The approach considered the capacities to be uniformly scattered among the possible weight sums, so that the capacity-dependent behaviour is annihilated. The capacity is chosen for an instance with n items as:

$$c = \frac{h}{\mathrm{H}+1} \sum_{j=1}^{n} w_i \tag{8}$$

where in all experiments $\mathrm{H} = 100$, meaning a variety of different capacities for each 100 instances in the set. A capacity in a set of instances generated with equation 8 will be mentioned as variable capacity through the paper. During generation we only consider instances that have feasible solutions.

VI. ANALYSIS OF DECISION VARIABLES

The analysis of decision variables constitutes a main step which is determining for a certain solver, the joint distribution of the decision variables from the independent distribution of each variable from a sample of instances. The goal of this analysis is to verify that the generation procedure described in the previous section is indeed effective in training of the model, and to understand the behaviour of a solver.

It is clear that the capacity has a role in controlling the complexity of the knapsack problem, and the solution for a knapsack problem could be too easy for a solver if the instances capacities are not designed carefully, thus the difficulty element is completely removed from these instances. As a result a training or evaluation set will not represent well designed problems that varies in difficulty. One could design a set with very hard or very easy problems that would make a neural model fails its task catastrophically.

To understand the behaviour of a solver, we visualize the decisions variables produced by this solver for different types of instances. From visualization, we can understand the roles of problem inputs as the weights and values items or their numerical ranges in solving the instance. The instances that constitute stumbles for a solver could be easily spotted through visualization.

To verify the previous assumptions, one should carefully analyze the distribution of the decision variables.

A decision variable x_i in a binary linear programming (BLP) problem is considered a random variable that follows Bernoulli distribution. If x_i is a random variable following Bernoulli distribution then:

$$P(x_i = 1) = p = 1 - P(x_i = 0) = 1 - q \tag{9}$$

where p is the probability that the random variable takes value 1 and q is the probability that the random variable takes value 0.

For simplicity, a decision variable x_i is considered independent from the other decision variables and from the value and weight of the item in the case of knapsack problem, this helps in understanding the behaviour of capacity in an instance.

We can generate a sample of CO problem instances and determine the value of p or q in the sample for each decision variable independently. Collectively, instances difficulty can be understood by averaging the values of p and q. The average value of p or q in the Bernoulli distributions of decision variables could express the shifting in variables distributions.

For example, if a solver naively by removed all items with large weights and include items only with large values and small weights, the decision variables produced by this solver will follow Bernoulli distribution where $p \ll q$ and vice versa.

VII. EXPERIMENTAL DETAILS

A. EVALUATION

We use the greedy algorithm as a baseline. The greedy approach is a common heuristic approach that is used to solve the knapsack problem. We evaluate both the Neural Knapsack solver and the greedy algorithm using two metrics: approximation ratio and optimal instances rate. We introduce a metric called infeasiablity rate for evaluation of models during training.

1) APPROXIMATION RATIO

To evaluate a heuristic solver on CO problems, approximation ratio averaged over a set of test instances is used. Averaged over J test instances, the approximation ratio [26], [38] for a solver R_s is defined as:

$$R_s = \frac{1}{J}\sum_{Y} max(\frac{Y}{C(Y_s)}, \frac{C(Y_s)}{Y}) \tag{10}$$

where Y_s is the solution produced by a solver, Y is cost the best known solution and $C(.)$ is a function that calculate the objective value for a problem.

Typically this ratio is taken in whichever direction makes it bigger than one [43], [44], that why we use the *max* function. The lower the approximation ratio, the better the solver.

2) INFEASIBILITY RATE

Instead of using the error on the validation set as a proxy for the generalization error [45], we define a new metric called infeasibility rate, defined as the percentage of the instances where the solution of the model is infeasible. The model training is stopped when the infeasibility rate is low beyond a certain threshold.

The infeasibility rate is an indicator that the model is understanding the problem. Obtaining optimal solutions indicates the quality of the solutions produced by the neural solver, while obtaining feasible solutions means that the model captured the main idea of CO problems which is: not breaking any of the problem constrains, a first step a human mathematician would do.

3) OPTIMAL INSTANCES RATE

Usually classical solvers are evaluated based on the percentage of instances solved correctly during suitable amount of time. We introduce the optimal instances rate as a metric for evaluating solvers, which is the percentage of instances where the solver produced optimal solutions. To calculate this metric, one will need the exact solutions. A solver that has high optimal instances rate is better.

B. DATASETS DESCRIPTION

The models are trained and tested on three types of knapsack instances based on the correlation between values and weights: uncorrelated (UC), strongly correlated (SC) and subset sum (SS) [42].

In UC instances, the weights and values are uniformly distributed over $[1, R]$, where R is the range of weights and values. In SC instances, the weights are uniformly distributed over $[1, R]$ but the value for item i is $v_i = w_i + \frac{R}{10}$. In SS, the weight and value of each item is equal $v_i = w_i$.

Table 1 shows the datasets names and their correlation and capacity types. Two additional data sets are the UC-KNAP-CAP and SC-KNAP-CAP, where the capacity is chosen uniformly between $[0.2n, 0.3n]$ adapted from [26].

C. SETUP

The memory constructor is a neural network with either dense layers, 1D-Convolution (CNN) layers, or gated recurrent unit (GRU) layers. The number of layer in the memory constructors of all models is one. The encoder and decoder are both implemented as GRU layers. The encoder depth is 2 and the decoder depth is 3. The knapsack handler is a stacked two GRU layers, in which the its last hidden state in the second layer is passed to the decoder as the representation of the sack at time step t.

TABLE 1. Datasets names and description.

Dataset Name	Correlation between values and weights	Capacity
UC-KNAP	Uncorrelated	Variable
SC-KNAP	Strongly correlated	Variable
SS-KNAP	Exact (values equal weights)	Variable
UC-KNAP-CAP	Uncorrelated	[0.2n,0.3n]
SC-KNAP-CAP	Strongly correlated	[0.2n,0.3n]

To obtain the targets for training, all the instances were solved using CBC an LP-based branch-and-cut library based on CLP solver which is a solver created within the Coin-OR project [46]. We used PuLP [47], a python interface to the CBC library.

We used Tensorflow library [48] with Keras [49] as an API. All GRU [50], [51] layers weights are initialized uniformly in range $[-0.08, 0, 08]$. The number of hidden in units in all layers is 32. For feed-forward and convolution layers, RELU [52], [53] is used as an activation function.

We used Adam optimizer for training with default parameters as in [54]. The learning rate was set to 0.004. The norm of the gradients was constrained to be no greater than 1. The batch size is 100 and the number of epochs was set to 100. During learning we evaluate the model after each epoch. We did not use dropout [55] or any other regularization technique.[2]

For the hardware environment, all the training and testing experiments were run on a PC with Intel-Corei5 processor and 4GB RAM, running on a Linux environment. No GPUs were used in all experiments.

D. TRAINING

We generated $200k$ instances for training and $1k$ for models evaluation from each type of instances, except for the UC-KNAP-CAP where we generated $50k$ for training. For test instances where $N \geq 50$, we generate 100 instances.

All the models, unless otherwise mentioned, was trained using naive curriculum learning strategy, more details on curriculum learning strategy can be found in [56]. We did not shuffle the data during learning and as a consequence, each 100 instances in the training set is gradually increasing in the capacity size.[3] A model would receive updated gradients using a mix of instances with different capacity ranges.

[2]We observed over fitting in models when trained on instances with N =15 and 20. We used dropout ratio = 0.2 upon training of the GRU based model and found slightly better performance.

[3]Except for the models trained on UC-KNAP-CAP dataset where capacity is random and instances are not graded in difficulty, so shuffling the training set is used.

TABLE 2. Neural Knapsack solvers performance on UC-KNAP instances with N = [5, 10, 15, 20], N = [21, 22, 23, 24] and N = [50, 100, 200]. Reported is the Optimal Instances Rate followed by the approximation ratio for each model and the baseline.

N	GREEDY	GRU	CNN	DENSE
5	64.4 - 1.0379	**80.3 - 1.0203**	76.5 - 1.022	78.1 - 1.0226
10	49.9 - 1.0311	**56.9 - 1.0268**	55.6 - 1.0313	51 - 1.0305
15	41.5 - 1.0236	44 - 1.024	41.5 - 1.0268	**46.1 - 1.0206**
20	38.9 - **1.0197**	34.3 - **1.0197**	**39.3** - 1.023	33.2 - 1.029
50	**33.0 - 1.008**	1.0 - 1.35	1.0 - 1.37	1.0-1.34
100	**24.0 - 1.004**	1.0 - 1.43	1.0 - 1.44	1.0 - 1.433
200	**9.0 - 1.002**	1.0 - 1.47	1.0 - 1.48	1.0 - 1.477

TABLE 3. Neural Knapsack solvers performance on SC-KNAP. Only the DENSE model was trained gradually up to $N = 20$, other models were trained up to $N = 15$.

N	GREEDY	GRU	GRU[Dropout = 0.2]	DENSE
5	16.5 - 1.1848	55.1 - 1.504	54.3 - 1.0515	**62 - 1.0379**
10	12.3 - 1.1005	17.8 - 1.049	**23.5 - 1.0451**	17.1 - 1.0462
15	10 - 1.0662	**12.7** - 1.0351	11.4 - **1.030**	12.7 - 1.0351
20	9.3 - 1.0502	-	-	**10.2 - 1.0298**
50	1.0 - 1.0297	1.0 - 1.0293	1.0 - 1.0273	1.0 - **1.0261**
100	1.0 - 1.0131	1.0 - 1.0135	1.0 - **1.0123**	1.0 - 1.0132
200	1.0 - 1.00686	1.0 - 1.0065	1.0 - **1.0064**	1.0 - 1.053

TABLE 4. Neural Knapsack solver performance on SS-KNAP.

N	GREEDY	GRU
5	21.4 - 1.234	**50.7 - 1.074**
10	4.2 - 1.141	**12.9 - 1.083**
15	2.2 - 1.094	**3.0 - 1.0604**
20	1.6 - 1.0672	**2.7 - 1.0493**
50	1.0 - **1.026**	**1.6** - 1.050
100	1.0 - 1.0136	1.0 - **1.0115**
200	1.0 - 1.007	1.0 - **1.005**

VIII. RESULTS

Results on different instances types and different number of items is reported in this section. Models are named after their memory constructor type, if there is any change in hyper-parameters (dropout, learning rate (lr) or number of layers),it is mentioned along with the model name.

In this section we discuss three major questions about the performance of the solvers models:

1) Can the model understand the knapsack problem as a CO problem through learning representations of problems inputs ?
2) Can the model generalize to other problem classes beyond those in the training set distribution ?
3) What are the representations that the model learn ?

A. NEURAL KNAPSACK SOLVER PERFORMANCE

Tables 2, 3, and 4 report the results on UC, SC and SS instances respectively. We only report the results of the best solvers.

Overall, the solvers performed slightly better than the greedy algorithm on all types of instances, however none of the solvers obtained optimal solutions for all test instances even for $N = 5$. This suggests that among instances of the same type, there may be instances that are hard for a neural network to obtain an optimal solution.

TABLE 5. Neural Knapsack solver performance on UC-KNAP-CAP instances.

N=5		
	Approximation Ratio	Optimal Instances Rate %
GRU	**1.02715**	**75.3**
GREEDY	1.05249	61.0
N=10		
GRU	1.039	45.9
GREEDY	**1.0341**	**46.3**
N=15		
GRU	1.0303	30.0
GREEDY	**1.0234**	**35.0**
N=20		
GRU	1.026	22.9
GREEDY	**1.020**	**29.4**

It is also worth to mention that as the model is learning by imitating the exact algorithm in a supervised fashion, it performed better than the greedy algorithm but still can not imitate exactly the branch-and-bound algorithm.

The solver variants performed better than the greedy algorithm, in obtaining better near optimal solutions (lower approximation ratio) and in obtaining optimal solutions to more instances (high optimal instances rate). Both the performance of the solver and the greedy algorithm degraded as the number of items increases. The solver was even better than the greedy algorithm with large margin on both SC and SS instances (which are considered hard knapsack instances [42]). It is suggested that the correlation between values and weights helped the neural network to learn better and hence perform better on these types of instances than the UC instances.

The solvers in which their models have GRU memory constructors were better and outperformed convolutional or dense ones, because a GRU layer produces dependent representations for items, in which each hidden representation for an item i depends on the representation of the previous item j, this would build more representative memories encoding features as order relationships, lexicographical order for pair of items or capacity relation to an item according to the weight. GRU based memory constructors were able to perform better with the increase of N, in which the curriculum learning strategy was effective.

The model outperformed the baseline on SC-KNAP-CAP instances but failed on UC-KNAP-CAP instance, again this may be due to the correlation between values and weights for items.

B. GENERALIZATION

We test our solvers after the training phase on instances with $N = [50, 100, 200]$. This was done to verify whether the solver neural model can generalize to instances with different number of items than those it already learned.

The solvers performed worse when their is no any correlation between values and weights of items, and the solvers performed better on SC instances even when number of items is 200. Both the greedy algorithm and the solver solutions are still far from optimal. In SS instances the model outperformed

TABLE 6. Neural Knapsack solver performance on SC-KNAP-CAP instances.

	Approximation Ratio	Optimal Instances Rate %
N=5		
GRU	**1.064**	**44.5**
GREEDY	1.185	19.2
N=10		
GRU	**1.046**	**18.3**
GREEDY	1.105	11.72
N=15		
GRU	**1.0455**	**7.3**
GREEDY	1.073	6.8
N=20		
GRU	**1.034**	5.0
GREEDY	1.056	**5.3**

(a) UC instances (b) SC instances (c) SS instances

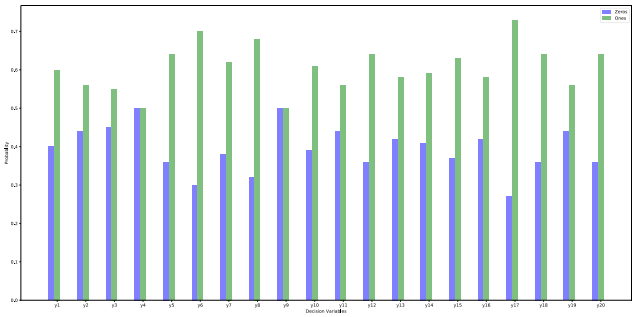
FIGURE 2. Visualization of solver decision variables for three types Knapsack instances. A black spot is a decision variable with 0 and white spot is a decision variable with 1. $N = 20$ for all instances.

the greedy algorithm even for $N = 200$ obtaining better approximation ratio and the same optimal instances rate (except for $N = 50$).

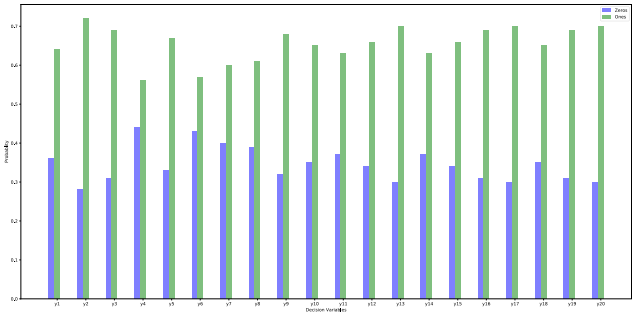
To understand the role of capacity in generalization, we generated corner instances, where the capacity is either very small or very large relative to the weights sum. With this we also could understand, whether the including of the capacity notation directly in the model is effective or not. We generate test instances with $N = [20, 25, 50, 100]$ same test instances were generated with two capacities: a small and large capacity relative to the sum of weight, and the capacity was constant in all instances of same type.

In instances with a small capacity $C = 500$, the results were worse than the greedy algorithm on all instances types, when capacity is large $C = 50000$ the solver obtained optimal solutions and approximation ratio 1 till $N = 50$, for $N = 100$ the solver performed similar to the greedy algorithm obtaining the same optimal instances rate and a lower approximation ratio.

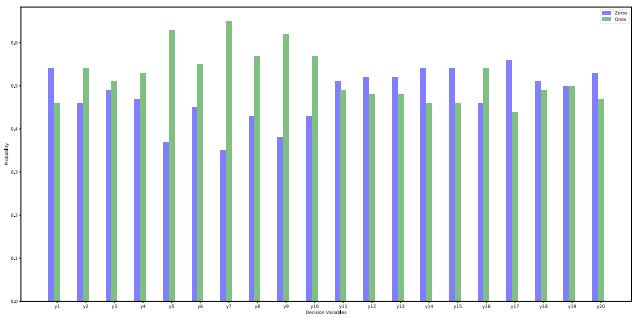
Finally, the models trained on instances with capacity directly proportional to weights sum (variable capacity) are tested on instances with capacity proportional to the number of items ($C = [0.2n, 0.3n]$) and vice-versa. The models trained on instances with capacity directly proportional to weights sum (variable capacity) performed better than model



(a) Distribution of UC instances



(b) Distribution of SC instances



(c) Distribution of SS instances

FIGURE 3. Sample distributions of each decision variable for each type of Knapsack instances. The distribution of variables is likely similar in instances of the same type except for the SS instances.

trained on instances with instances with capacity proportional to the number of items in all cases.

It is clear from results that the model performed better on instances with high correlation between values and weights (inputs), because a neural network model might learn effectively better if there is an association or correlation between inputs. In case of 2D images represented as a grid or matrix the correlation between neighbouring pixels helps the network to learn representations from these images.

C. VISUALIZATION OF DECISION VARIABLES

Figure 2 shows the decision variables distribution for samples of 100 instances for the three type of KP instances. We can see that the density of black spots representing 0 is very high at the top of each figure and then the gradually decreases, which indeed means that as the capacity increases the distribution of decision variable x_i shifts towards 0 restricting the selection of items that can be in the sack.

D. DISTRIBUTION OF DECISION VARIABLES

Figure 3 shows the distribution of each decision variable in each type of instances. In instances of the same type all variables follow similarly the Bernoulli distribution with different values for p and q.

In case of UC instances, some variables however have extreme values of p where $p > 0.7$ but none have value of p less than 0.5. Similarly is the case of SC instances but the variables have higher value of p where $p \geq 0.6$ for all variables.

The case of SS is interesting, because not all the variables follow similar distributions (i.e the value of p in Bernoulli distribution is not equally similar for all variables). However, we could notice that the distributions is shifted more towards 0 and $q > p$ in most of the variables.

IX. CONCLUSION

A neural based solver for solving a the knapsack problem is proposed. The neural architecture introduced is inspired by both computational heuristics and humans that solve the problem cognitively. The architecture has a new module is that incorporates the state of the solver decisions in the learning and inference phases. The neural models were trained end-to-end fashion using supervised targets from an exact solver. The solver models were trained and tested on different types of instances. The solver was tested on its ability to generalize to instances with large number of items (up to 200 items) and the model performed better only on instances, in which items values have a correlation with their weights. The correlation between values and weights of items appears to have a role in learning better representations from problem inputs and helps the model to generalize in some cases.

Although other meta-heuristic solvers as monarch butterfly optimization (MBO), earthworm optimization algorithm (EWA), Global firefly algorithm (GFA), and moth search (MS) algorithm may work better or have well defined research work, but it is worth to understand the consequences of incorporating learning in designing a heuristic solver for the knapsack problem, and how this solver would perform on solving different instances of knapsack problem.

A major result of this work and in the complexity analysis for the Knapsack problem is that the capacity has a role in the complexity of an instance, not the number of items as it is commonly known. Previous work that incorporates learning in knapsack problem used fixed capacity or variable capacity that is proportional to the number of items to generate instances. The solver worked better when its models were trained on instances with graded capacities than variable or fixed capacities.

There is a great body of work on heuristic and meta-heuristic algorithms for the Knapsack problem and other combinatorial optimization problems [57]. These algorithms are applied on large scale instances and benchmarks for these problems. The authors note that the machine learning community should consider and benefit from this work when incorporating learning in optimization problems.

REFERENCES

[1] D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization," *IEEE Trans. Evol. Comput.*, vol. 1, no. 1, pp. 67–82, Apr. 1997.
[2] W. Zaremba and I. Sutskever, "Learning to execute," 2014, *arXiv:1410.4615*. [Online]. Available: http://arxiv.org/abs/1410.4615
[3] A. Graves, G. Wayne, and I. Danihelka, "Neural turing machines," 2014, *arXiv:1410.5401*. [Online]. Available: http://arxiv.org/abs/1410.5401
[4] A. Graves, G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwinska, S. G. Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou, and A. P. Badia, "Hybrid computing using a neural network with dynamic external memory," *Nature*, vol. 538, no. 7626, p. 471, 2016.
[5] A. E. Ezugwu, V. Pillay, D. Hirasen, K. Sivanarain, and M. Govender, "A comparative study of meta-heuristic optimization algorithms for 0–1 knapsack problem: Some initial results," *IEEE Access*, vol. 7, pp. 43979–44001, 2019.
[6] F. Yanhong, Y. Juan, H. Yichao, and W. Gaika, "Differential evolution monarch butterfly optimization algorithm for solving discount 0-1 knapsack problem," *Acta Electronica Sinica*, vol. 46, no. 6, pp. 1343–1350, 2018.
[7] A. W. Mohamed, "A new modified binary differential evolution algorithm and its applications," *Appl. Math. Inf. Sci.*, vol. 10, no. 5, pp. 1965–1969, Sep. 2016.
[8] Y. Feng, G.-G. Wang, S. Deb, M. Lu, and X.-J. Zhao, "Solving 0–1 knapsack problem by a novel binary monarch butterfly optimization," *Neural Comput. Appl.*, vol. 28, no. 7, pp. 1619–1634, Jul. 2017.
[9] Y. Feng, G.-G. Wang, W. Li, and N. Li, "Multi-strategy monarch butterfly optimization algorithm for discounted 0-1 knapsack problem," *Neural Comput. Appl.*, vol. 30, no. 10, pp. 3019–3036, Nov. 2018.
[10] Feng, Yu, and Wang, "A novel monarch butterfly optimization with global position updating operator for large-scale 0-1 knapsack problems," *Mathematics*, vol. 7, no. 11, p. 1056, Nov. 2019.
[11] Y.-H. Feng and G.-G. Wang, "Binary moth search algorithm for discounted 0-1 knapsack problem," *IEEE Access*, vol. 6, pp. 10708–10719, 2018.
[12] Y. Feng, J.-H. Yi, and G.-G. Wang, "Enhanced moth search algorithm for the set-union knapsack problems," *IEEE Access*, vol. 7, pp. 173774–173785, 2019.
[13] Y. Feng, G.-G. Wang, and L. Wang, "Solving randomized time-varying knapsack problems by a novel global firefly algorithm," *Eng. with Comput.*, vol. 34, no. 3, pp. 621–635, Jul. 2018.
[14] Y. Feng, G.-G. Wang, and X.-Z. Gao, "A novel hybrid cuckoo search algorithm with global harmony search for 0-1 knapsack problems," *Int. J. Comput. Intell. Syst.*, vol. 9, no. 6, pp. 1174–1190, Nov. 2016.
[15] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, p. 436, 2015.
[16] K. A. Smith, "Neural networks for combinatorial optimization: A review of more than a decade of research," *INFORMS J. Comput.*, vol. 11, no. 1, pp. 15–34, Feb. 1999.
[17] J. J. Hopfield and D. W. Tank, "'Neural' computation of decisions in optimization problems," *Biol. Cybern.*, vol. 52, no. 3, pp. 141–152, 1985.
[18] S. V. B. Aiyer, M. Niranjan, and F. Fallside, "A theoretical investigation into the performance of the hopfield model," *IEEE Trans. Neural Netw.*, vol. 1, no. 2, pp. 204–215, Jun. 1990.
[19] A. H. Gee, "Problem solving with optimization networks," Ph.D. dissertation, Dept. Eng., Univ. Cambridge, Cambridge, U.K., 1993.
[20] R. Durbin and D. Willshaw, "An analogue approach to the travelling salesman problem using an elastic net method," *Nature*, vol. 326, no. 6114, p. 689, 1987.
[21] J. C. Fort, "Solving a combinatorial problem via self-organizing process: An application of the Kohonen algorithm to the traveling salesman problem," *Biol. Cybern.*, vol. 59, no. 1, pp. 33–40, Jun. 1988.
[22] B. Angéniol, G. de La Croix Vaubois, and J.-Y. Le Texier, "Self-organizing feature maps and the travelling salesman problem," *Neural Netw.*, vol. 1, no. 4, pp. 289–293, Jan. 1988.
[23] F. Sarwar and A. Aziz Bhatti, "Critical analysis of hopfield's neural network model for TSP and its comparison with heuristic algorithm for shortest path computation," in *Proc. 9th Int. Bhurban Conf. Appl. Sci. Technol. (IBCAST)*, Jan. 2012, pp. 111–114.

[24] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2015, pp. 2692–2700.
[25] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio, "Neural combinatorial optimization with reinforcement learning," 2016, *arXiv:1611.09940*. [Online]. Available: http://arxiv.org/abs/1611.09940
[26] A. Nowak-Vila, D. Folqué, and J. Bruna, "Divide and conquer networks," 2016, *arXiv:1611.02401*. [Online]. Available: http://arxiv.org/abs/1611.02401
[27] H. Kellerer, U. Pferschy, and D. Pisinger, "The unbounded knapsack problem," in *Knapsack Problems*. Berlin, Germany: Springer, 2004, pp. 211–234.
[28] V. Voinov, "A note on the intractability of partition, knapsack, subset sum and related problems," *Math. J.*, vol. 17, no. 4, pp. 13–24, 2017.
[29] G. B. Dantzig, "Discrete-variable extremum problems," *Oper. Res.*, vol. 5, no. 2, pp. 266–288, Apr. 1957.
[30] P. Bossaerts and C. Murawski, "Computational complexity and human decision-making," *Trends Cognit. Sci.*, vol. 21, no. 12, pp. 917–929, Dec. 2017.
[31] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2014, pp. 3104–3112.
[32] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," 2014, *arXiv:1409.0473*. [Online]. Available: http://arxiv.org/abs/1409.0473
[33] K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhudinov, R. Zemel, and Y. Bengio, "Show, attend and tell: Neural image caption generation with visual attention," in *Proc. Int. Conf. Mach. Learn.*, 2015, pp. 2048–2057.
[34] J. Weston, S. Chopra, and A. Bordes, "Memory networks," 2014, *arXiv:1410.3916*. [Online]. Available: http://arxiv.org/abs/1410.3916
[35] S. Sukhbaatar, J. Weston, and R. Fergus, "End-to-end memory networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2015, pp. 2440–2448.
[36] N. Tishby and N. Zaslavsky, "Deep learning and the information bottleneck principle," in *Proc. IEEE Inf. Theory Workshop (ITW)*, Apr. 2015, pp. 1–5.
[37] R. Shwartz-Ziv and N. Tishby, "Opening the black box of deep neural networks via information," 2017, *arXiv:1703.00810*. [Online]. Available: http://arxiv.org/abs/1703.00810
[38] E. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song, "Learning combinatorial optimization algorithms over graphs," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 6348–6358.
[39] M. Freitag and Y. Al-Onaizan, "Beam search strategies for neural machine translation," 2017, *arXiv:1702.01806*. [Online]. Available: http://arxiv.org/abs/1702.01806
[40] Y. Bengio, A. Lodi, and A. Prouvost, "Machine learning for combinatorial optimization: A methodological tour d'Horizon," 2018, *arXiv:1811.06128*. [Online]. Available: http://arxiv.org/abs/1811.06128
[41] D. Pisinger, "Core problems in knapsack algorithms," *Oper. Res.*, vol. 47, no. 4, pp. 570–575, Aug. 1999.
[42] D. Pisinger, "Where are the hard knapsack problems?" *Comput. Oper. Res.*, vol. 32, no. 9, pp. 2271–2284, Sep. 2005.
[43] V. Kann, "On the approximability of NP-complete optimization problems," Ph.D. dissertation, Roy. Inst. Technol., Faridabad, India, 1992.
[44] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi, *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Berlin, Germany: Springer, 2012.
[45] L. Prechelt, "Early stopping-but when?" in *Neural Networks: Tricks of the Trade*. Berlin, Germany: Springer, 1998, pp. 55–69.
[46] R. Lougee-Heimer, "The common optimization INterface for operations research: Promoting open-source software in the operations research community," *IBM J. Res. Develop.*, vol. 47, no. 1, pp. 57–66, Jan. 2003.
[47] S. Mitchell and I. Dunning, "PuLP: A linear programming toolkit for Python," Univ. Auckland, Auckland, New Zealand, Tech. Rep., 2011.
[48] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, and M. Kudlur, "Tensorflow: A system for large-scale machine learning," in *Proc. 12th Symp. Operating Syst. Design Implement. (OSDI)*, 2016, pp. 265–283.
[49] F. Chollet. (2015). *Keras*. [Online]. Available: https://keras.io
[50] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," 2014, *arXiv:1409.1259*. [Online]. Available: http://arxiv.org/abs/1409.1259
[51] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," 2014, *arXiv:1412.3555*. [Online]. Available: http://arxiv.org/abs/1412.3555
[52] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proc. 14th Int. Conf. Artif. Intell. Statist.*, 2011, pp. 315–323.
[53] V. Nair and G. E. Hinton, "Rectified linear units improve restricted Boltzmann machines," in *Proc. 27th Int. Conf. Mach. Learn. (ICML)*, 2010, pp. 807–814.
[54] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, *arXiv:1412.6980*. [Online]. Available: http://arxiv.org/abs/1412.6980
[55] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, 2014.
[56] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, "Curriculum learning," in *Proc. 26th Annu. Int. Conf. Mach. Learn.*, 2009, pp. 41–48.
[57] P. Agrawal, T. Ganesh, and A. W. Mohamed, "A novel binary gaining–sharing knowledge-based optimization algorithm for feature selection," *Neural Comput. Appl.*, pp. 1–20, Oct. 2020.

HAZEM A. A. NOMER received the B.Sc. degree in computer science and statistics from Alexandria University, and the M.Sc. degree in computer science. He is currently a Research Assistant with the Wireless Intelligent Networks Center (WINC), School of Engineering and Applied Sciences, Nile University, Giza. He is interested in deep learning, neural networks, optimization, combinatorial problems, and evolutionary computation.

KHALID ABDULAZIZ ALNOWIBET was born in Riyadh, Saudi Arabia, in 1971. He received the B.Sc. degree in operations research from the College of Sciences, King Saud University, Riyadh, Saudi Arabia, the M.Sc. degree in operations research from the School of Engineering and Applied Sciences, George Washington University, Washington DC, USA, in 1994, and the Ph.D. degree in operations research from the School of Engineering, North Carolina State University, NC, USA, in 2004. His dissertation title was: "Nonstationary Loss Queues and Network of Nonstationary Loss Queues." He has published several papers in several research areas related to operations research such as: performance evaluation using stochastic modeling, queueing networks applications, queueing theory and applications, stochastic processes theory and applications, as well as modeling communication networks. He is currently an Associate Professor with the Department of Operations Research, King Saud University, for more than 15 years. During his service in the university, he has been appointed to several key administrative positions in the university. He has also supervised graduate students and has participated in many theses defenses.

ASHRAF ELSAYED received the B.Sc. and M.Sc. degrees from Alexandria University, Alexandria, Egypt, in 1995 and 2004, respectively, both in computer science, and the Ph.D. degree in computer science from the University of Liverpool, U.K., in 2012. His is currently an Assistant Professor with the Faculty of Computer Science and Engineering, Al Alamein International University, Egypt (On leave as an Assistant Professor at the Faculty of Science, Alexandria University). His research interests include data science, big data analytics, deep learning, quantum machine learning, and medical image mining.

ALI WAGDY MOHAMED received the B.Sc., M.Sc., and Ph.D. degrees from Cairo University, in 2000, 2004, and 2010, respectively. He is an Associate Professor with the Operations Research Department, Faculty of Graduate Studies for Statistical Research, Cairo University, Egypt. He is currently an Associate Professor of Statistics with the Wireless Intelligent Networks Center (WINC), Faculty of Engineering and Applied Sciences, Nile University. He serves as a Reviewer for more than 50 internationally accredited top-tier journals and has been awarded the Publons Peer Review Awards 2018, for placing in the top 1 position worldwide in the assorted field. He is an Editor in more than 10 journals of information sciences, applied mathematics, engineering, system science, and operations research. He has presented and participated in more than five international conferences. He has participated as a member of the reviewer committee for 35 different conferences sponsored by Springer and IEEE. He has obtained Rank 3 in CEC'17 competition on single objective bound-constrained real-parameter numerical optimization in the Proceedings of the IEEE Congress on Evolutionary Computation, IEEE-CEC 2017, San Sebastián, Spain. Besides, he obtained Rank 3 and Rank 2 in CEC'18 competition on single objective bound-constrained real-parameter numerical optimization and competition on Large scale global optimization, in the Proceedings of IEEE Congress on Evolutionary Computation, IEEE-CEC 2017, Sao Paolo, Brazil. He has published more than 55 articles in reputed and high impact journals like *Information Sciences*, *Swarm and Evolutionary Computation*, *Computers & Industrial Engineering*, *Intelligent Manufacturing*, *Soft Computing*, and *International Journal of Machine Learning and Cybernetics*. He is interested in mathematical and statistical modeling, stochastic and deterministic optimization, swarm intelligence, and evolutionary computation. In addition, he is also interested in real-world problems such as industrial, transportation, manufacturing, education, and capital investment problems. He has been a supervisor of two Ph.D. students and 1 M.Sc. student.