Machine Learning Based Cognitive Radar Resource Management

Mahdi Shaghaghi and Raviraj S. Adve
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario, Canada
Email: mahdi.shaghaghi@utoronto.ca, rsadve@ece.utoronto.ca

***Abstract*—A modern radar may be designed to perform multiple functions, such as surveillance, tracking, and fire control. Each function requires the radar to execute a number of transmit-receive tasks. This raises the problem of assigning radar resources, such as the time, frequency and energy budget, to different tasks. Specifically, a radar resource management (RRM) module makes decisions on parameter selection, prioritization, and scheduling of such tasks. RRM becomes especially challenging in overload situations, where some tasks may need to be delayed or even dropped. Such task scheduling is an NP-hard problem. Furthermore, with multichannel, e.g., multi-frequency, radars becoming increasingly viable, multiple tasks can be executed simultaneously. While this greatly enhances the ability to execute tasks, it also complicates task scheduling, now on multiple timelines. In previous work, we had developed the branch-and-bound (B&B) method to solve the NP-hard problem, an approach with exponential computational complexity. In this work, we use the results of the B&B method to train a machine-learning based scheduler. Essentially, we propose to speed up the B&B method by estimating the value of the nodes of the search tree using a neural network. Our results show that the use of neural networks in conjunction with the B&B method results in a close-to-optimal solution while significantly reducing the computational complexity.**

I. Introduction

A cognitive radar (CR) is defined as a radar performing intelligent signal processing, which builds on learning through interactions of the radar with the surrounding environment [1]. The key element of a CR is acquisition of knowledge through observation (training) and also using that knowledge in future decision making. In this paper, we consider a cognitive approach for radar resource management (RRM) in a multi-function radar (MFR).

A MFR handles various functions, such as wide-area surveillance and tracking multiple targets, by executing a number of transmit-receive tasks. In this case, the available radar resources, specifically time, frequency, and energy, need to be assigned to different tasks in an efficient manner. A RRM module is designed to make decisions on parameter selection, prioritization, and scheduling of the tasks [2]. Often, the number of tasks exceeds the capabilities of the MFR; RRM becomes especially challenging in these overload situations, where some tasks may need to be delayed or even dropped.

Various techniques have been developed in the literature for resource management for multifunction radars (see for example [3]–[5] and references therein). Since RRM, specifically task scheduling, is, in general, an NP-hard problem, a common theme considered in the literature is to split RRM into two steps [2]. The first step is to determine task parameters such as the priority, dwell time, and revisit interval by using rule-based methods for each task individually [6] or by joint optimization, across all tasks, of an overall utility function, while accounting for resource constraints [3], [7].

Once the parameters are determined, the second step is task selection and scheduling. In the selection phase, a subset of the tasks are chosen based on resource constraints, and a figure of merit, and the rest of the tasks are dropped. In the scheduling phase, time slots are assigned to the tasks. The above steps can also be repeated iteratively to improve performance.

On a separate track, multichannel, e.g., multi-frequency, radars are becoming increasingly viable. Multichannel radars bring the possibility of executing multiple tasks simultaneously on different channels (timelines) [8]. However, this additional capability also complicates the already difficult RRM problem. Tasks must now be assigned to channels in addition to being scheduled. In this paper, we consider the problem of joint resource management for a multichannel multifunction radar, i.e., one that is able to concurrently execute multiple tasks.

In considering RRM for multichannel radars, in previous work, we developed the optimal branch-and-bound (B&B) technique [9]. Compared to this optimal solution, we showed that heuristic methods in the literature have relatively poor performance. However, as any solution to an NP-hard problem must, the B&B algorithm suffers from exponential computational complexity and, for anything beyond a very short task list, cannot be expected to be executed in anything approaching real time.

Motivated by the need to balance performance with complexity, in this paper, we introduce machine learning (ML) techniques to the RRM problem in a multichannel MFR. Our ML methods *learn from* the previous executions of the B&B algorithm (in a *training phase*). The acquired knowledge is then used in future scheduling problems. Specifically, we use neural networks to estimate the value associated with a node in the B&B method, thereby converging to a close-to-optimal solution while significantly reducing the computational burden. The value is the least cost of a complete solution which can be obtained starting from the given node and, if it is too far from the optimal solution, the node and its sub-branches are eliminated from the tree.

The neural networks are trained using supervised learning. The training datasets (labeled data) are obtained by running the B&B algorithm offline for relatively small problems. The resulting training data allows us to train a convolutional neural network. Our simulation results show significant gains in computation load for a low loss in performance.

This paper is organized as follows. The problem formulation is provided in Section II. We present the enhanced B&B method in Section III, and compare its performance with the heuristic and B&B methods using numerical simulations in Section IV. Finally, Section V concludes the paper.

II. Problem Formulation

The problem formulation is as follows [9]: we consider N tasks to be executed on K identical channels (timelines) within a time window T. Tasks can be performed concurrently on different channels, but cannot overlap on a given timeline. Furthermore, we consider a non-preemptive scheduling scenario, where a running task is not stopped until completion. For each task, there is a *starting time* r_n after which the task can be executed. There is also a *deadline* d_n after which the task is dropped (with an associated *dropping cost* D_n). The time to execute (dwell time) the n-th $(1 \leq n \leq N)$ task is denoted by ℓ_n.

As an example, consider a tracking task. The starting time of the task is determined by the required tracking accuracy and the time when the last measurement was made. The deadline depends on the beamwidth of the radar and the estimated trajectory of the target. The task will not be executed after the target is assumed to have moved out of the radar beam, and it is therefore dropped with a cost. Consequently, further actions may be required to compensate for the dropping.

A scheduled, but delayed, task suffers a *tardiness cost* which is, here, modeled as linearly proportional to the delay; let e_n be the time when task n begins execution; the tardiness cost is given by $w_n(e_n - r_n)$, where w_n is the weight which scales the delay. Let the binary variable x_{nk} indicate if task n is scheduled $(= 1)$ on the $k-$th timeline or not. Task n is dropped if $x_{nk} = 0 \; \forall k$. Then, the cost associated with the n-th task is given by $\sum_{k=1}^{K} x_{nk} w_n (e_n - r_n) + (1 - \sum_{k=1}^{K} x_{nk}) D_n$. Our joint task selection and scheduling problem is a minimization of the total cost C, given by

$$C = \sum_{n=1}^{N} \sum_{k=1}^{K} x_{nk} w_n (e_n - r_n) + (\frac{1}{K} - x_{nk}) D_n. \quad (1)$$

Here, the optimization variables are x_{nk} and e_n. If a task is scheduled, i.e., $x_{nk} = 1$ for some k, the execution time e_n needs to be determined as well. Our optimization problem is, therefore,

$$\begin{aligned} \{x^*_{nk}, e^*_n\} \; &= \min_{x_{nk}, e_n} \textstyle\sum_{n=1}^{N} \sum_{k=1}^{K} x_{nk} w_n (e_n - r_n) \\ &\qquad\qquad + (1/K - x_{nk}) D_n \\ &\text{s.t. } x_{nk} \in \{0, 1\} \\ &\quad \textstyle\sum_{k=1}^{K} x_{nk} \leq 1 \\ &\quad r_n \leq e_n \leq d_n \\ &\quad n = 1, \ldots, N \\ &\quad k = 1, \ldots, K \\ &\quad \text{and no tasks overlap in time.} \end{aligned} \quad (2)$$

The second constraint, $\sum_{k=1}^{K} x_{nk} \leq 1$, ensures that, if scheduled, the task is assigned to a single timeline only.

The scheduling problem without deadlines (and therefore without dropped tasks) is NP-hard [10]. It can be shown that the joint task selection and scheduling is also NP-hard.

III. Proposed Method

The B&B procedure can be employed to find the optimal solution of the problem in (2) [9]. The main drawback with the B&B method is the execution time, which could be exponential in the number of tasks. The execution time depends on the task parameters and, in our experience, is heavy-tailed, i.e., while many executions are possible in a reasonable amount of time, a few cases require an inordinate execution time. In order to reduce the complexity of the B&B algorithm, we propose an approximate method which uses a neural network to speed up the search.

The B&B method implicitly enumerates all possible solutions on a search tree. The root node of the tree represents the whole solution space. The rest of the nodes are associated with a subset of the solution space. The branching operation splits the space of the parent node into the subspaces of the resulting children nodes. Each subspace represents a partial solution. During the search, bounds and dominance rules are used to eliminate nodes from the tree. Along with these rules, we propose to use a neural network to facilitate the elimination of the unpromising nodes from the search tree. Once the entire tree has been explored, the best solution found in the search is returned.

Regarding the problem in (2), each node of the tree represents a partial schedule which includes a subset of the tasks. It is shown in [11] that instead of searching for the optimal execution times of the tasks, we can search for the optimal permutation of the tasks. We construct the search tree in the following way. The node at the root is an empty sequence. A branch represents choosing a task which has not been scheduled in the parent node and its deadline has not passed yet. The selected task is appended to the sequence of the parent node to obtain the sequence of the resulting child node. A schedule is obtained using the "sequence to schedule mapping" [9], which iterates on the elements of the sequence, and places each task on the earliest available timeline. Using this technique, we search for the best sequence.

Dominance rules are problem specific and mathematically proven rules used to eliminate nodes from the search tree. If it can be shown that the performance of all the solutions in a given node s_1 is worse than the performance of a solution of another node s_2, then s_1 is dominated by s_2, and we can therefore eliminate s_1 from the tree.

Lower and upper bounds can also be used for the purpose of pruning the nodes of the search tree. If the lower bound on the cost of all the solutions of a given node s is larger than the upper bound, UB, on the cost of the optimal solution, it can be concluded that s does not include the optimal solution. Therefore, s can be eliminated from the tree. The upper bound can be initialized using a heuristic method, and it can be updated using the best-solution-found-so-far during the search.

In the partial schedule associated with a node, a set of tasks are scheduled on certain timelines with known execution times and delay costs. Furthermore, the tasks whose deadlines have already passed on all timelines are dropped (with certain dropping costs). The rest of the tasks are not scheduled or dropped yet. A *state*, s, is defined as a representation of a partial schedule (node). It includes the initial parameters of the tasks as well as their status (scheduled, dropped, or not scheduled) in the corresponding partial schedule.

Suppose there is a value function $v^*(s)$ which determines the least achievable cost of a complete schedule starting from a given state s. Note that here the value function is the cost of the best solution in the subspace of s. The depth of the search may be reduced by truncating the tree at state s and replacing the cost of the best schedule in the subtree below s by the value function evaluated at s, $v^*(s)$. We use a neural network (referred to as the *value network* [12]) to produce an approximation for the value function $v_\theta(s) \approx v^*(s)$, where θ denotes the weights of the value network. The network has a state as its input and outputs an approximation of the optimal value of the given state. Such a network can be trained using recorded solutions (labeled data) obtained by the offline execution of the B&B algorithm on sample problems.

The prediction made by the value network at a given node s is compared with UB, and if the predicted cost is large enough, node s can be eliminated from the search tree. In this way, the B&B method is made faster by removing the nodes that are less likely to end up with the optimal solution.

The steps of the proposed method are listed in Table I. The lines marked with "$*$" are used for data recording and are not required parts of the algorithm. The search tree is implemented using a stack data structure (see Algorithm 1 in [11]). A depth-first search strategy is used to traverse the nodes of the tree. Here, we have modified the B&B method of [11] to include task dropping. Each element (node) of the stack is a tuple which consists of a sequence of tasks $\boldsymbol{T}$ (representing a partial schedule), a set of tasks that can be scheduled right after $\boldsymbol{T}$ (denoted by the possible-first $\boldsymbol{PF}$ set), a set of not-scheduled tasks, $\boldsymbol{NS}$, and a set of tasks which have been dropped, $\boldsymbol{DR}$.

At a given node s, branching is performed by choosing the task a from $\boldsymbol{PF}$ which has the smallest starting time. Task a is then removed from $\boldsymbol{PF}$ and is added to $\boldsymbol{NS}$. Note that a is scheduled for the current branch and regarding the rest of the branches of s, it has not been scheduled yet. Therefore, it is added to the $\boldsymbol{NS}$ set. In this way, when we add a new branch at node s, we merge the tasks of the $\boldsymbol{NS}$ set with the elements of the $\boldsymbol{PF}$ set to form the possible-first set of the new child node.

At the root node, the possible-first set is initialized to include all the tasks, and $\boldsymbol{NS}$ and $\boldsymbol{DR}$ are set to be empty. An upper bound UB holds the cost of the best complete solution obtained during the search. UB is initially set to infinity. The root node is pushed on top of the stack. Then, as long as the stack is not empty, the following procedure is performed: at each iteration, the node on top of the stack is checked. If the possible-first set $\boldsymbol{PF}$ is empty, the node will be removed from the stack. Before removing the node, we check if there is any task in the not-scheduled set $\boldsymbol{NS}$. If $\boldsymbol{NS}$ is empty, the node is terminal and represents a complete solution. In this case,the overall cost C is compared with UB, and if there is an improvement, UB is updated accordingly.

In the scenario that the possible-first set of the node on top of the stack is not empty, a new node is generated using a task of $\boldsymbol{PF}$. Specifically, let a be the task in $\boldsymbol{PF}$ with the smallest starting time. We remove a from $\boldsymbol{PF}$ and append it at the end of $\boldsymbol{T}$ to obtain the sequence of the new node (denoted by $\boldsymbol{T}'$). The possible-first set of the new node, $\boldsymbol{PF}'$, is set as the union of $\boldsymbol{PF}$ and $\boldsymbol{NS}$. The dropped tasks, $\boldsymbol{DR}'$, are inherited from $\boldsymbol{DR}$, and the not-scheduled set of the new node, $\boldsymbol{NS}'$, is set as empty. Then, task a is added to $\boldsymbol{NS}$.

After a new node is generated, we determine whether it should be added to the stack for further investigation in the next iteration or it can be ignored and therefore pruned off. To do so, we first check the *start-times dominance rule* on $\boldsymbol{T}'$. This rule states that the sequence of execution times of tasks in $\boldsymbol{T}'$ should be non-decreasing; otherwise, $\boldsymbol{T}'$ is dominated and cannot result in an optimal solution [13].

We next check the tasks in $\boldsymbol{PF}'$; any task whose deadline is before the earliest available time of the channels is dropped. At this point, the sum of the tardiness and dropping costs of the new partial solution can be computed, providing a lower bound on any complete solution derived from this node (denoted by C'). If this cost is not lower than the cost of the best-solution-found-so-far, i.e., UB, the new node can be ignored. Furthermore, if $\boldsymbol{T}'$ does not map to an *active schedule*, it can be discarded. An active schedule is such that no task can be completed earlier without delaying another task [14]. In order to determine whether $\boldsymbol{T}'$ is active, the tasks in $\boldsymbol{PF}'$ are checked to see if any of them can be scheduled before the last task in $\boldsymbol{T}'$ (on the same timeline) without imposing a delay.

Next, we check if $\boldsymbol{T}'$ is a *LOWS-active* (LOcally Well Sorted active) schedule [11]. A LOWS-active schedule is such that no exchange of two adjacent tasks can improve the schedule. The adjacent tasks of task a are defined as the set of tasks scheduled on the final slot of each timeline just before scheduling task a. The conditions for checking if a schedule is LOWS-active are the same as given in [11, Section 4] with the exception that we also need to consider task dropping. The conditions need to be modified such that we consider the event when a swapping of two tasks requires one of the tasks to be dropped. In this case, the availability time of the timeline is used instead of the completion time, and the dropping cost replaces the tardiness cost (since the task is not scheduled). If the new node is LOWS-active, we check if it is also a complete solution (terminal node). In this case, we update the statistics of the parent node for the purpose of data recording. This will be explained in more detail in the following section.

TABLE I. BRANCH-AND-BOUND METHOD AUGMENTED WITH A VALUE NETWORK

```
Initialization
  $UB \leftarrow \infty$
  Let $\boldsymbol{T}$ be an empty sequence
  $\boldsymbol{PF} \leftarrow \{$all tasks$\}$
  $\boldsymbol{NS} \leftarrow \{\}$
  $\boldsymbol{DR} \leftarrow \{\}$
  Push $(\boldsymbol{T}, \boldsymbol{PF}, \boldsymbol{NS}, \boldsymbol{DR})$ on stack.
while STACK is not empty
  Let $s = (\boldsymbol{T}, \boldsymbol{PF}, \boldsymbol{NS}, \boldsymbol{DR})$
  be the node on top of STACK.
  if $\boldsymbol{PF} \neq \{\}$
    Let $a \in \boldsymbol{PF}$
    $\boldsymbol{PF} \leftarrow \boldsymbol{PF} \setminus a$
    $\boldsymbol{T}' \leftarrow \boldsymbol{T}|a$
    $\boldsymbol{PF}' \leftarrow \boldsymbol{PF} \cup \boldsymbol{NS}$
    $\boldsymbol{NS}' \leftarrow \{\}$
    $\boldsymbol{DR}' \leftarrow \boldsymbol{DR}$
    $\boldsymbol{NS} \leftarrow \boldsymbol{NS} \cup a$
    if $\boldsymbol{T}'$ follows the start-times dominance rule
      Move any task whose deadline has passed on all
      timelines from $\boldsymbol{PF}'$ to $\boldsymbol{DR}'$.
      $C' \leftarrow$ TardinessCost($\boldsymbol{T}'$) + DroppingCost($\boldsymbol{DR}'$)
      if ($\boldsymbol{T}'$ is active) and ($\boldsymbol{T}'$ is LOWS-active)
        * if ($\boldsymbol{PF}' = \{\}$) (i.e., if the new node is terminal)
        *   Update the statistics of node $s$.
        if ($C' < UB$) and ($v_\theta(s) < \alpha \times UB$)
          Push $s' = (\boldsymbol{T}', \boldsymbol{PF}', \boldsymbol{NS}', \boldsymbol{DR}')$ on stack.
  else
    $C \leftarrow$ TardinessCost($\boldsymbol{T}$) + DroppingCost($\boldsymbol{DR}$)
    if ($\boldsymbol{NS} = \{\}$) and ($C < UB$)
      $UB \leftarrow C$
    * if $s$ has termination
    *   Update the statistics of the parent node of $s$.
    *   Record the statistics of $s$.
    Remove $(\boldsymbol{T}, \boldsymbol{PF}, \boldsymbol{NS}, \boldsymbol{DR})$ from stack.
```

Besides checking if C' is smaller than UB, we also use the value network to obtain the predicted least final cost that can be achieved from the new node, and we compare it with UB. If the estimated value is larger than UB multiplied by a scaling coefficient α (i.e. the estimated value $\geq \alpha \times UB$), the node is eliminated from the search. The scaling coefficient $\alpha \geq 1$ is introduced to make the algorithm robust to estimation errors. Better performance may be achieved with larger values of α, but that would mean that less nodes are eliminated from the search tree.

In the case that the bounds and dominance rules are passed, the new node is pushed on top of the stack. After this step is complete, the search goes on to the next iteration. Finally, once the entire tree is explored, the best solution found is returned.

A. Data Recording

We record the statistics of the nodes during the search and later use them as labeled data for supervised learning of the value neural network. For a given node s, there is a flag, $\mathbb{I}_s$, which indicates whether we have reached at least one complete solution (terminal node) in the descendant nodes of s during the search. In the case that node s has termination, i.e., $\mathbb{I}_s$ is true, the cost of its best terminal solution is recorded.

As explained in the previous section, in the step of branching, we check whether the new child node, s', is LOWS-active. If true, next we check whether s' is a complete solution. If the possible-first set of s' is empty, we have reached a terminal node, i.e., a complete solution. In this case, the statistics of the parent node, s, are updated. If $\mathbb{I}_s$ is false or if it is true and the cost of s' is smaller than the best-terminal-cost of s, $\mathbb{I}_s$ will be set to true (if not already true), and the best-terminal-cost of s will be set to the cost of s', respectively.

In the B&B procedure, before the last step of removing a given node s form the stack, we check whether it has termination, i.e., if $\mathbb{I}_s$ is true. In this case, the statistics of the parent node of s are updated (i.e., the best-terminal-cost of the parent of s is updated according to s). Furthermore, node s along with all of its statistics is recorded. Note that the recorded best-terminal-cost of the given node s is indeed its value, i.e., $v^*(s)$. The recorded labeled data, i.e., $(s, v^*(s))$ pairs, are then used in supervised learning of the value network as explained in Section III-B.

B. Value Network Architecture

We have implemented the value network using a convolutional neural network with 6 layers. The first three layers are convolutional and the last three layers are fully connected. At each convolutional layer, the input is convolved with a filter to produce the output features. The coefficients of the filter are obtained using supervised learning.

The output of each layer goes through a non-linear function (the rectifier function) before being passed to the next layer. The final output of the network is a scalar number representing the estimated least overall cost of the input partial schedule. The input to the network (a partial schedule) is a matrix with each column representing a task and each row representing a feature of the corresponding task.

In this paper, we have considered 14 features for each task as given in Table II. It is assumed that there are 4 channels available for task scheduling. One input feature is dedicated for each timeline (channel), and we use one-hot encoding to represent the channel on which the task is scheduled. For each task, the input feature corresponding to the channel on which the task is scheduled is set to 1, and the rest of input timeline features are set to 0. As an example, assuming that there are 4 timelines (channels) available for task scheduling, each task will have 4 features to represent the assigned timeline (only one input corresponding to the assigned timeline is set to 1, and the rest are 0).

For the convolutional layers, filters with a width of 7 (looking at the features of 7 consecutive tasks at each stride) are used. At each layer 64 filters are used (the output of each convolutional layer has 64 features).

We have considered 512 hidden units for the first fully connected layer. The second fully connected layer has 128 hidden units, and the last layer has one scalar output.

Regularization methods are generally used in the training phase to avoid the problem of overfitting the neural networks

TABLE II. FEATURES OF THE TASKS AS USED FOR THE INPUT OF THE VALUE NETWORK

Feature	Description
$1, 2, 3$	status (1 0 0: scheduled, 0 1 0: dropped, 0 0 1: unscheduled)
4	start time
5	end time
6	task length
7	execution time
8	tardiness coefficient
9	dropping cost coefficient
10	tardiness cost or drop cost (if scheduled or dropped)
$11, 12, 13, 14$	assigned timeline (one-hot encoded)

to the training samples. These methods help the network to learn the general patterns in the training samples instead of memorizing them. We use the dropout technique, one of the most common methods used for regularization [15]. During the training stage, a fixed percentage (50 percent in our case) of the units of each layer is randomly dropped (multiplied by zero). This results in training subnetworks of the base network. Dropout regularizes each unit to be not merely a good feature but a feature that is good in many contexts.

Batch normalization is also used to improve the optimization process during the training phase [16]. Let $\boldsymbol{H}$ be a mini-batch of activations of a given layer. To normalize $\boldsymbol{H}$, we replace it with

$$\boldsymbol{H}' = \frac{\boldsymbol{H} - \boldsymbol{\mu}}{\boldsymbol{\sigma}} \tag{3}$$

where $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ represent the mean and standard deviation of the units, respectively. For the convolutional layers, these moments are obtained over the first three dimensions of $\boldsymbol{H}$, and therefore, they have a length of 64 (the depth of the feature maps). For the fully connected layers, the moments are computed over the rows of $\boldsymbol{H}$, and therefore, their length is equal to the number of hidden units. The subtraction and division operations in (3) are performed by broadcasting.

During the training, $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are obtained using the sample mean and variance of the current batch [17, Section 8.7.1]. During the inference, $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ can be replaced by the running averages that were collected during training time.

Normalizing the activations of the layers may reduce the expressive power of the network. Therefore, the normalized batch $\boldsymbol{H}'$ is replaced with $\gamma\boldsymbol{H}' + \boldsymbol{\beta}$ where γ and $\boldsymbol{\beta}$ are variables learned during the training phase.

The weights of the network can be found by regression on the training state-outcome pairs $(s, v^*(s))$ using gradient descent to minimize the mean squared error (MSE) between the estimated value, $v_\theta(s)$, and the corresponding true outcome $v^*(s)$

$$\Delta\theta \propto \frac{\partial v_\theta(s)}{\partial\theta}\left(v^*(s) - v_\theta(s)\right). \tag{4}$$

The training data are obtained by the offline execution of the B&B method as explained in Section III-A. The network is trained using 90000 samples. The weights are obtained by minimizing the L2-loss using the Adaptive Moment Estimation (Adam) optimization method [18]. We use 100000 steps of the Random Reshuffling (RR) method [19] with mini-batches of size 100.

IV. SIMULATION RESULTS

We now present the results of simulations illustrating the performance of the proposed method. We consider a multichannel radar with $K = 4$ identical channels. There are $N = 35$ tasks distributed over a timeline window of 100 ms. The objective is to schedule the tasks such that the overall cost of dropping and delaying the tasks, the objective in (2), is minimized. In our simulations, the starting time of the tasks is uniformly distributed on the 100 ms time window, i.e. $r_n \sim \mathcal{U}(0, 100)$, where $x \sim \mathcal{U}(a, b)$ represents a random variable uniformly distributed between a and b.

For each task, the interval between the starting time and the deadline, i.e., $d_n - r_n$, is sampled from $\mathcal{U}(2, 12)$. The task length, ℓ_n, is distributed according to $\mathcal{U}(2, 11)$. Furthermore, the dropping costs, D_n, and the tardiness weights, w_n, have distributions $\mathcal{U}(100, 500)$ and $\mathcal{U}(1, 5)$, respectively. In the simulations, we use the Monte Carlo method with 100 trials to obtain the average performance of each method. We emphasize that these models for the parameters is for simulations only; in practice, the parameters would be determined by mission requirements.

The average cost of the heuristic methods, the B&B algorithm, and the proposed method with different scaling coefficients are given in Table III. The average number of visited nodes in the search tree for the B&B and also the enhanced B&B methods are given in Table IV. Note that the running time of the B&B and the proposed methods are proportional to the number of visited nodes. As expected, the B&B algorithm has the best performance and highest complexity. The B&B method has an average visited nodes of 13134. In the case that the value network with $\alpha = 1$ is used, the average number of visited nodes drops to only 342 nodes which is about 38 times less than the number of visited nodes of the B&B method without the value network. However, this complexity reduction results in performance degradation. Better performance can be achieved by increasing the scaling coefficient. With $\alpha = 2$, an average cost of 49.3 is obtained which is significantly lower than the average cost of the heuristic methods. The average number of visited nodes, however, is increased to 962, but it is still about 14 times less than the average visited nodes in the B&B algorithm.The performance can be further improved with $\alpha = 3$ with a slight increase in the computational burden.

TABLE III. AVERAGE COST OF DIFFERENT METHODS. EST AND ED REFER TO EARLIEST STARTING TIME FIRST AND EARLIEST DEADLINE FIRST METHODS, RESPECTIVELY [9]. SW REFERS TO THE TASK SWAPPING TECHNIQUE [9]. THE LAST THREE COLUMNS ARE THE RESULTS OF THE B&B METHOD WHEN ENHANCED BY THE VALUE NETWORK WITH DIFFERENT SCALING COEFFICIENTS.

	EST	EST+SW	ED	ED+SW	B&B	$\alpha = 1$	$\alpha = 2$	$\alpha = 3$
Average cost	93.2	68.3	115.8	101.5	38.6	69.5	49.3	46.6

TABLE IV. AVERAGE NUMBER OF VISITED NODES IN THE SEARCH TREE OF THE B&B METHOD AS WELL AS THE B&B METHOD WHEN ENHANCED BY THE VALUE NETWORK WITH DIFFERENT SCALING COEFFICIENTS.

	B&B	$\alpha = 1$	$\alpha = 2$	$\alpha = 3$
Average number of visited nodes	13134	342	962	1158

V. CONCLUSION

In this paper, we consider the problem of radar resource management for a multichannel multifunction radar. We introduce how machine learning techniques can be used to acquire knowledge during the operation of the radar in a way that such knowledge can be used in future decision making.

The branch-and-bound algorithm finds the optimal solution for the task scheduling problem but with high computational burden. We show how the complexity of the branch-and-bound method can be reduced using machine learning methods. A value network is used at each node of the search tree to predict the least final cost that can be achieved from the given partial schedule of the node. While this significantly reduces the number of visited nodes of the search tree, the performance does not suffer from much degradation. The value network is trained using supervised learning with data samples obtained from the offline execution of the optimal branch-and-bound method.

ACKNOWLEDGMENT

This work is supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada and Defence Research and Development Canada (DRDC).

REFERENCES

[1] S. Haykin, “Cognitive radar: a way of the future,” *IEEE Sig. Process. Mag.*, vol. 23, no. 1, pp. 30–40, Jan. 2006.

[2] P. W. Moo and Z. Ding, *Adaptive Radar Resource Management*. Academic Press, 2015.

[3] A. Charlish, K. Woodbridge, and H. Griffiths, “Phased array radar resource management using continuous double auction,” *IEEE Trans. Aerosp. Electron. Syst.*, vol. 51, no. 3, pp. 2212–2224, Jul. 2015.

[4] S. L. C. Miranda, C. J. Baker, K. Woodbridge, and H. D. Griffiths, “Comparison of scheduling algorithms for multifunction radar,” *IET Radar Sonar Navig.*, vol. 1, no. 6, pp. 414–424, Dec. 2007.

[5] A. M. Ponsford, L. Sevgi, and H. C. Chan, “An integrated maritime surveillance system based on high-frequency surface-wave radars, Part 2: Operational status and system performance,” *IEEE Antennas Propag. Mag.*, vol. 43, no. 5, pp. 52–63, Oct. 2001.

[6] S. P. Noyes, “Calculation of next time for track update in the MESAR phased array radar,” in *IEE Colloquium on target tracking and data fusion (digest no. 1998/282)*, Jun. 1998, pp. 2/1–2/7.

[7] A. Sinha, Z. J. Ding, T. Kirubarajan, and M. Farooq, “Track quality based multitarget tracking algorithm,” in *Proc. SPIE 6236, Signal and Data Processing of Small Targets 2006, 623609*, May 2006.

[8] J. Yan, H. Liu, B. Jiu, B. Chen, Z. Liu, and Z. Bao, “Simultaneous multibeam resource allocation scheme for multiple target tracking,” *IEEE Trans. on Sig. Proc.*, vol. 63, no. 12, pp. 3110–3122, June 2015.

[9] M. Shaghaghi and R. S. Adve, “Task selection and scheduling in multifunction multichannel radars,” in *2017 IEEE Radar Conference (RadarConf)*, May 2017, pp. 969–974.

[10] J. K. Lenstra, A. R. Kan, and P. Brucker, “Complexity of machine scheduling problems,” *Annals of Discrete Mathematics*, vol. 1, pp. 343–362, 1977.

[11] A. Jouglet and D. Savourey, “Dominance rules for the parallel machine total weighted tardiness scheduling problem with release dates,” *Comput. Oper. Res.*, vol. 38, no. 9, pp. 1259 – 1266, 2011.

[12] D. Silver *et al.*, “Mastering the game of Go with deep neural networks and tree search,” *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016.

[13] F. Yalaoui and C. Chu, “New exact method to solve the P$m/r_j/\sum C_j$ schedule problem,” *International Journal of Production Economics*, vol. 100, no. 1, pp. 168 – 179, 2006.

[14] A. Jouglet, P. Baptiste, and J. Carlier, “Branch-and-bound algorithms for total weighted tardiness,” in *Handbook of scheduling: Algorithms, models, and performance analysis*. CRC Press, 2004.

[15] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: a simple way to prevent neural networks from overfitting,” *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

[16] S. Ioffe and C. Szegedy, “Batch normalization: accelerating deep network training by reducing internal covariate shift,” in *Proceedings of the 32nd International Conference on Machine Learning*, vol. 37. Lille, France: PMLR, Jul. 2015, pp. 448–456.

[17] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[18] D. Kingma and J. L. Ba, “Adam: a method for stochastic optimization,” *arXiv:1412.6980*, 2014.

[19] M. Gürbüzbalaban, A. Ozdaglar, and P. Parrilo, “Why random reshuffling beats stochastic gradient descent,” *arXiv:1510.08560*, 2015.