Robot Task Interruption by Learning to Switch Among Multiple Models

Anahita Mohseni-Kabir and **Manuela Veloso**
School of Computer Science, Carnegie Mellon University
{anahitam, mmv}@cs.cmu.edu

Abstract

While mobile robots reliably perform each service task by accurately localizing and safely navigating avoiding obstacles, they do not respond in any other way to their surroundings. We can make the robots more responsive to their environment by equipping them with models of multiple tasks and a way to interrupt a specific task and switch to another task based on observations. However the challenges of a multiple task model approach include selecting a task model to execute based on observations and having a potentially large set of observations associated with the set of all individual task models. We present a novel two-step solution. First, our approach leverages the tasks' policies and an abstract representation of their states, and learns which task should be executed at each given world state. Secondly, the algorithm uses the learned tasks and identifies the observation stimuli that trigger the interruption of one task and the switch to another task. We show that our solution using the switching stimuli compares favorably to the naive approach of learning a combined model for all the tasks. Moreover, leveraging the stimuli significantly decreases the amount of sensory input processing during the execution of tasks.

1 Introduction

Our service robots can successfully perform user-requested tasks by executing a single planned task at a time [Veloso *et al.*, 2015]. Our goal is to increase the robots' responsiveness to their environment by enabling them to interrupt their task execution and switch to another task when appropriate. For example, if the robot is scheduled to deliver an object to an office, and on the way to the office it sees a person lying on the ground, asking for help, we would want the robot to first assist the person and then finish its delivery. There are ways to hard-code the switching response for a specific problem. However, we target service robot domains in which the tasks have hundreds of state variables (*e.g.*, human's distance and speed, human gaze, distance to obstacles) and hard-coding the switching response is not feasible.

One way to address the task-switching behavior is to encode all the details of the tasks' structure and environment in one combined model and use a reinforcement learning (RL) [Sutton and Barto, 1998] algorithm to find an optimal policy. This approach has a large number of state variables with the potential of high computational complexity. Another drawback of the combined model approach is that the robot should process all the sensory state variables for all the task models during the execution of the combined model. In most robotics applications, there is a cost associated with processing the sensory state variables, *e.g.*, cost for human pose estimation and speech recognition, and it is often not feasible to process all sensory inputs.

We use a multiple task model representation and learn when to switch between task models to focus the sensory computations. We propose a two-step solution. In the first step, *learning*, the robot learns a task selection policy that specifies which task should be executed at each world state. We formulate the task-switching problem as a Markov Decision Process (MDP) and leverage a Dueling Deep Q-Network architecture to solve it [Wang *et al.*, 2015]. In the second step, *identification*, we speed up the execution of the task models by identifying "stimuli" that trigger the task-switching. Leveraging the stimuli enables the robot to focus on only one task at a time. When a stimulus is triggered, the robot computes all sensory inputs and decides if a switch is more rewarding than continuing with the current task. We identify these stimuli by identifying the sensory inputs that have higher impact on the task switching.

In summary, our work makes the following contributions: 1) a novel approach that learns a mapping from world states to task models (*learning* step), 2) a novel algorithm that identifies the stimuli that trigger the switch between task models (*identification* step), 3) our approach enables a robot to be more responsive to its surrounding environment, and 4) our two-step algorithm significantly decreases the amount of sensory input processing during the execution of the tasks.

2 Related Work

Hierarchical Reinforcement Learning (HRL) approaches are capable of learning both the internal policies and the termination conditions of options (or tasks), in tandem with the policy over options [Barto and Mahadevan, 2003; Konidaris, 2016; Kulkarni *et al.*, 2016; Bacon *et al.*, 2017]. However, these approaches do not learn when it is more rewarding for the robot to interrupt the execution and switch to another option. Another approach shows that greedy interruption of options is better than not interrupting and treating them as indivisible units [Sutton *et al.*, 1999]. This approach processes all the state variables to decide if a switch to another option is better. Differently, our approach learns the switching policy and only processes all the variables when the stimuli are triggered.

Goal management is another area of research that has some similarities to our work [Vattam *et al.*, 2013]. Some work permits arbitration between current goals based on priority values that are dynamically computed from predefined conditions and rules [Choi, 2011]. Another work evaluates all possible goals for the agent using a set of predefined fitness functions and selects the goal with the best combined score [Muñoz-Avila *et al.*, 2015]. However, in our service robot domain with hundreds of state variables, it is not feasible to hard-code the conditions or the fitness functions. In some other work, Q-learning is used to learn a goal selection policy over all the state variables. This approach computes the same amount of state variables as the combined model in contrast to our approach that enables the robot to focus on only one task at a time [Jaidee *et al.*, 2012].

Behavior-based control systems (BBC) provide algorithms to select and activate the appropriate behaviors given the robot's observations [Pirjanian, 1999; Matarić and Michaud, 2008]. In these approaches, the behaviors are selected and executed concurrently to collectively achieve the desired system-level behavior [Nicolescu *et al.*, 2006]. Different from these approaches that process all the state variables, our approach decreases the amount of sensory computations by pursuing only one task at a time. In our approach, the robot is committed to perform one task, might temporarily switch to other tasks, but eventually returns to execute the initial task. Some BBC approaches predefine the conditions and learn a switching policy over them [Martinson *et al.*, 2001; Raïevsky and Michaud, 2008]. Differently, we introduce an approach that learns the switching policy, identifies the stimuli, and then uses both the policy and stimuli to switch between multiple task models.

3 Approach

In this section, we explain how we formalize the task-switching problem as an MDP. We then discuss how we identify the stimuli that trigger the task-switching behavior.

We consider a scenario in which the robot is executing a user-requested task, *e.g.*, object delivery, and is also alert for other observations like humans and objects (*e.g.*, trash) around it. These observations may lead the robot to interrupt the current task execution if the switch to a human interaction or trash cleaning task results in a higher future utility.

3.1 Learning Task Selection Policy

Our task selection algorithm is provided with a set of task models; each task model in this set achieves one goal. The output of the algorithm is a policy, denoted by $\pi_{select-task}$, that specifies which task model should be executed given an observation of the environment. Each task model, denoted by T_i for the i^{th} task model, is represented as an MDP [Kober *et al.*, 2013]. Solving the i^{th} task model's MDP by a value-function based approach provides a policy π_i and a value function V_i. The policy π_i specifies the best action that should be taken in each of the task model's states. The value function V_i specifies how good each state is from the perspective of the i^{th} task model. The ultimate goal of the task selection algorithm is to use π_i's and V_i's, and learn when to switch between the task models.

Problem formulation:

We formalize the task-switching problem as an MDP, and we refer to it as "switching MDP". The MDP's representation[1] is as follows:

- ***States***: In a domain with n task models, the switching MDP's state is $[V_1(S_1), V_2(S_2), \ldots, V_n(S_n)]$. State S_i represents the state of the i^{th} task model, and $V_i(S_i)$ represents the expected reward when starting in S_i for the i^{th} task model.
- ***Actions***: In a domain with n task models, the actions are *execute* π_1, *execute* π_2, ..., and *execute* π_n. Each call to *execute* π_i executes an action based on policy π_i of task model T_i.
- ***Reward function***: The reward is produced by the environment. This reward function specifies the multi-task behavior of the robot.

Learning $\pi_{select-task}$:

Our MDP's state space is continuous, therefore we use a neural network to approximate the Q function. Some work provides a variant of Q-learning called Deep Q-Network (DQN) that stabilizes RL by utilizing an experience replay mechanism and a second fixed target network [Mnih *et al.*, 2015]. In addition to utilizing these two improvements of DQN approach, we leverage the dueling architecture which compared to DQN provides a more robust estimate of the Q function and has shown better performance in domains with many similar-valued actions [Wang *et al.*, 2015]. Particularly, since the task models can have common state variables and actions in our domain, the optimal policy for different task models might be the same in some parts of the state space. Hence, the dueling method is more appropriate for our domain.

We use our MDP formulation of the task-switching problem and apply the deep Q-network method to approximate the task-switching policy. To learn the task-switching policy for n task models, our deep model gets $[V_1(S_1), V_2(S_2), .., V_n(S_n)]$ as an input. The output of our network specifies which π_i should be executed given the input to the network. Fig. 1 shows what happens in one step of learning if our domain consists of n task models. If $n = 3$ the robot observes the environment, updates S_1, S_2, and S_3, and computes the corresponding $V_1(S_1)$, $V_2(S_2)$, and $V_3(S_3)$. The robot picks one of the task models (*e.g.*, T_1) and executes one step of its policy (*e.g.*, if $a_3 = \pi_1(S_1)$, the robot executes action a_3). The robot observes the new state and the reward of

[1]We assume a discount factor of 0.99 in all our experiments.

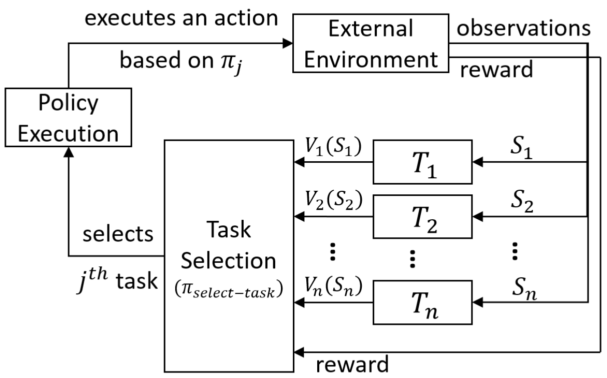

Figure 1: Overview of the task selection module.

the action execution, and updates the parameters of the network. The robot keeps updating the parameters by applying the DQN approach until the loss converges to 0.

Executing $\pi_{select-task}$:
We explain how the robot uses the trained network at execution time to perform the following task: the robot is scheduled to deliver an object to a location (main task), while interacting with the people around it. The robot starts going to the goal location to deliver the object. At each step during execution, the robot computes an array of V_i's and passes it to the network. The network then selects a task, and the robot executes an action from the task's policy. In our example scenario, the robot might see multiple people in the scene. Interacting with each person is a different task. The robot might choose to interact with a person, and then decide to interact with another person or return to the object delivery task.

The Q-values and V-values have been used in other work to address the action (model) selection problem [Karlsson, 1997; Humphrys, 1996; Sutton *et al.*, 1999]. Their algorithms use heuristic values calculated from Q-values and V-values to select actions from different modules. Similar to their approach, our tasks are learned with different reward functions, but our approach trains a model based on the global reward function that specifies the relative utility of the tasks to each other. The robot learns a behavior that specifies how it should switch between its multiple tasks to gain the highest utility. Notice that just getting the maximum of V-values does not work for the following reasons: 1) each task is learned separately with a different reward function, and 2) the robot might greedily select a task that is closer and miss a dense group of rewarding tasks further away.

Our task-switching formulation has the following characteristics:

- The structure of our task selection algorithm is independent of the number of state variables and actions in each task model's MDP, and the size of the input to the deep network increases linearly as the number of task models increases. This is because each task model is learned separately using its own reward function, and the high-level task selection module only learns how to switch between the tasks to get the highest utility.
- Our formulation decouples the task model representation from the task selection module. Thus, MDPs can be replaced by more complex representations, *e.g.*, partially observable MDPs (POMDPs), or simpler representations, *e.g.*, a simple shortest path policy to goal, as long as they provide a value function and a policy.
- The task selection module might not learn the combined model's optimal solution in some cases since it only uses the individual task models' policy and value function. However, training the task selection module is faster than solving the combined model. The task selection algorithm favors computational feasibility over optimality.
- Although the task selection algorithm enables the robot to switch between multiple task models, the robot is still processing the same amount of sensory input as the combined model during the execution phase. More specifically, the robot should first update all the state variables and then use them to calculate V_i's. That is to say, in a real scenario, while the robot is executing a navigation task, it should also process all the state variables of a Human-Robot Interaction task to decide if a switch to a different task is appropriate. We introduce the notion of task-switching stimuli in the next section to decrease the amount of sensory computations at execution time.

3.2 Identifying Task-Switching Stimuli

On one end of the spectrum, we have approaches like our task selection algorithm that process all the sensory information for decision making. These approaches work if the total time of processing the sensory information is less than the desired response time of the robot. However, as we increase the number of sensors on robots and the tasks become more complex, these approaches become infeasible. On the other end of the spectrum, we have approaches that only focus on one task at a time. These approaches are not responsive to their external environment which makes them less effective in real-world applications. We are interested in a method that trades-off between the amount of sensory input computations and responsiveness of robots. We introduce another algorithm that leverages stimuli to address this trade-off.

A "stimulus" (plural stimuli) is a detectable change in the internal or external environment of a robot that causes a reaction in the robot. These stimuli, when triggered, interrupt the robot to assess if switching to another task model is beneficial. The information that a robot requires to execute actions and achieve a task model's goal is already provided in the task model's state variables (features). We introduce an algorithm that takes as input the state variables of each task model and selects the most informative state variable as the stimulus that triggers the reevaluations of the task selection policy $\pi_{select-task}$. For each task model, our algorithm computes a sorted list of the task model's features with their associated importance percentage.

Algorithm 1 presents the stimuli identification algorithm. The stimuli identification algorithm takes as an input a list of target task models U, the task selection policy $\pi_{select-task}$, and the main robot task c_{main}, and computes U's features' importances. We run N simulations (line 1) with different goal locations and random initial values for the features (line 2) and evaluate the task selection policy at each step of the simulation (line 5). To find the features that have the most impact on the task-switching policy, the algorithm gathers examples of observations where the robot decides to switch to another task or continue with the current task. For each target task model, the algorithm considers an empty positive and negative example sets. At each simulation step, if we switch from task model A to task model B, we add the current state of the task models $U - B$ to their negative example set (line 7) and the current state of task model B to its positive example set (line 6); otherwise, we add the current state of task model A to its positive example set, and the current state of the other task models $U - A$ to their negative example set. Notice that if the robot does not switch from A to another task model, we still consider the state as a positive example for A, since A is preferred to the other task models in that state. We keep updating the positive and negative example sets for the tasks until all the simulations terminate.

Algorithm 1 Task-Switching Stimuli Identification. The algorithm takes the task selection policy $\pi_{select-task}$, the number of simulations N, the main robot task c_{main}, and a list of target task models U as input.

```
 1: for $1$ to $N$ do
 2:     Randomly reset all task models.
 3:     while $c_{main}$ not done do
 4:         $c_{state} \leftarrow compute\_taskmodel\_values()$
                ▷ computes state of the switching MDP ($V_i$'s)
 5:         $c_{task} \leftarrow \pi_{select-task}(c_{state})$
                ▷ selects a task model
 6:         Add the state of $c_{task}$ to its positive set
 7:         Add the state of $U - c_{task}$ to their negative sets
 8:         $execute(c_{task})$
                ▷ executes one step of $c_{task}$
 9: for $task\ in\ U$ do
10:     $data, label \leftarrow get\_sensory\_data(task)$
11:     $clf \leftarrow fit\_classifier(data, label)$
12:     $task.importances \leftarrow feature\_importances(clf)$
```

We formalize the identification problem as a classification problem. The stimuli identification algorithm first filters out all nonsensory features since they cannot be used as a stimulus to interrupt the task execution (line 10). The sensory feature vectors with their labels are then provided to the classification algorithm (line 11), and the feature importances are computed (line 12). The feature with the highest importance is selected as the stimulus for the target task.

Algorithm 2 describes how the *learning* and *identification* steps of our approach are integrated and executed by the robot. The robot starts with the main robot task, *e.g.*, object delivery task, and keeps executing it (line 4) until a stimulus is triggered, *e.g.*, the robot sees a person, or the current task is done (line 5). The robot then computes all the sensory variables and selects a task model to execute (lines 6-7). Notice that while executing the current task c_{task}, the robot only processes the sensory variables of c_{task} and the stimuli.

Algorithm 2 Task-switching behavior. The algorithm takes the task selection policy $\pi_{select-task}$ and the main robot task c_{main} as input.

```
1: $c_{task} \leftarrow c_{main}$
2: while $c_{main}$ not done do
3:     repeat
4:         $execute(c_{task})$
5:     until stimuli not triggered & $c_{task}$ not done
6:     $c_{state} \leftarrow compute\_taskmodel\_values()$
7:     $c_{task} \leftarrow \pi_{select-task}(c_{state})$
```

4 Experiments

In this section, we discuss the results of our task selection and stimuli identification algorithms in a scenario with 1 to 6 tasks (*e.g.*, the object delivery and human interaction tasks) that our service robot encounters everyday in our building.

Neural Network Structure

The network gets as input an array with size equal to the number of task models. This is followed by 3 hidden layers, each with 60 neurons and ReLU activation functions. The output layer has size equal to the number of task models. We sample uniformly a batch of size 32 from the replay memory of size $50,000$ to perform each update. We use a linear decay epsilon greedy policy with maximum value 1 and minimum value 0.1 and the Adam stochastic gradient descent method as the optimizer with learning rate 0.001 [Kingma and Ba, 2014]. We use the same network structure and parameters in all our experiments. Instead of applying a hard update on the network, we use a soft update method with smoothing parameter $\alpha = e^{-2}$ to update the model. The parameters of the DQN approach, *e.g.*, minibatch and replay memory size, are equivalent to the ones used by other works in deep RL [Mnih *et al.*, 2015].

Feature Importance Computation

In order to compute the feature importances, we apply the Extra-trees algorithm on the positive and negative example sets[2] [Geurts *et al.*, 2006]. We used the extra-trees algorithm with 1000 estimators, *i.e.*, 1000 trees in the ensemble, gini criterion and maximum depth of 4. Features with higher ranks, *i.e.*, at the top of the tree, contribute more to the final classification decision of a larger fraction of the examples. The expected fraction of the examples that each feature contributes to is used as an estimate of the relative importance of the feature. Averaging the relative importances over several randomized trees produces the feature importances for each target task model.

Simulation Setup

We tested our task-switching behavior in an 11×11-grid environment (Fig. 2) with three types of tasks: an object delivery task with 3 features and 3 actions, a trash cleaning task with 4 features and 5 actions, and a Human-Robot Interaction task (HRI) with 7 features and 7 actions. Except for the *x*

[2]We used the scikit-learn implementation of the algorithm [Pedregosa *et al.*, 2011]

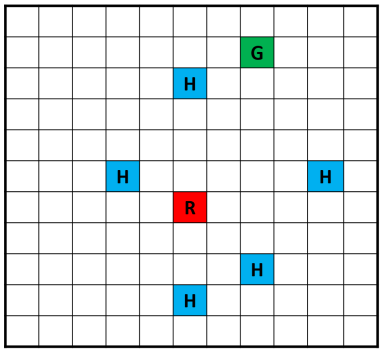

Figure 2: 11×11-grid environment with the robot (R), the navigation goal (G), and 5 humans goals (H's).

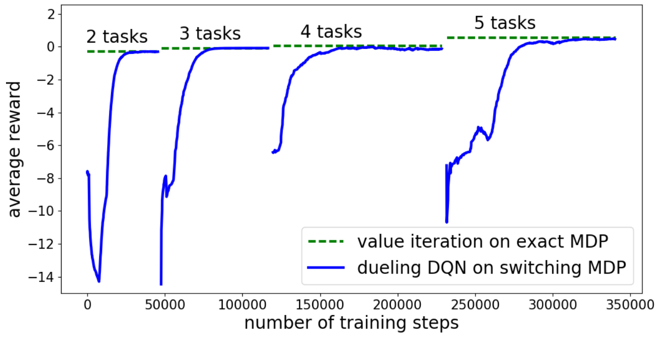

Figure 3: Performance of the switching MDP during the training phase for 1 delivery task and 1 to 4 trash cleaning tasks.

and *y* position of the robot, all other features are binary. In all experiments, the robot is performing an object delivery task while interacting with 0 to n people or executing 0 to n trash cleaning tasks. Thus, the number of tasks ranges from 1 to $n+1$.

We build a huge MDP with the $n+1$ tasks, and we call it "exact MDP" since it computes the exact solution to our problem. The exact MDP would have $5n+3$ state variables, $121 \times 2^{5n+1}$ states, and 8 actions if we use the HRI task and $2n+3$ state variables, $121 \times 2^{2n+1}$ states, and 6 actions if we use the trash cleaning task. We compare our switching MDP's solution to the solution of the value-iteration algorithm on the exact MDP. In both MDPs, the robot gets -0.1 reward for each action execution. The reward of interacting with a human goal, cleaning trash, and delivering an object is 1, 1, and 0 respectively. The episode terminates when the robot achieves the delivery task regardless of whether it interacts with the humans or performs the trash cleaning tasks. We consider the same setup for our switching MDP. With $n+1$ tasks, our switching MDP has $n+1$ state variables and $n+1$ actions.

4.1 Results of Task-Switching Behavior

We evaluate our task-switching behavior in different setups and show its benefits over the exact MDP. Here, we assume that the switching stimulus is detected by the stimuli identification algorithm, and we provide the results of the identification step in the next section. We report the final results of our task-switching approach in three evaluations:

1. In the first evaluation, we used one object delivery task and 1 to 4 different trash cleaning tasks and compared the switching MDP and the exact MDP's performance during training. While executing the object delivery task, the robot observes 1 to 4 different trash goals, and it should decide if it should switch to another task. Fig. 3 shows the performance of our switching MDP compared to the exact MDP (dashed lines). To compute the performance, we average the final reward of running 120 simulations with random initial values for the state variables. The final reward of a simulation is the total reward of its episode until it terminates or the robot reaches the maximum number of steps, which is set to 150. Fig. 3 shows our dueling Q-network performance is very close to the exact solution at the end of the training process.

As the number of tasks increases, the neural network requires more training steps to converge to the optimal solution. For two, three, four, and five tasks, the network required $25,000$, $30,000$, $60,000$, and $70,000$ training steps respectively. Due to space constraints, here we briefly highlight the advantages of our approach, in terms of time complexity of the learning phase, compared to the naive approach. To better illustrate the computational complexity of the combined model when the state space is just slightly bigger, we introduce a new task HRI-13 that has the same 7 variables as the HRI task, and we add 6 more binary variables to it.

The value-iteration algorithm computes the optimal solutions of the object delivery and HRI-13 tasks in 0.08 and 66.04 seconds (s) respectively. For 2 tasks, object delivery and HRI-13, the naive approach takes 137.01 s to solve the MDP, and our approach takes 183.03 s to learn the switching policy. The time complexity of our approach, in total 249.15 s ($183.03 + 0.08 + 66.04$), is almost twice as much as the complexity of the naive approach. If we add one more task with 13 variables (total 25 variables), value-iteration was not able to compute the solution even after hours of waiting, but our approach in total took 336.98 s to compute the solution.

2. In the second evaluation, we compared the performance of our switching MDP with the exact MDP when the robot executes the final learned network. In addition to the cost of each action execution, the robot receives a negative reward for updating each sensory variable, we call this cost "observation cost" (oc). For example, the cost of detecting a person in the scene is 0.01 ($oc = 0.01$). In the exact MDP, all the task models' state variables are being computed. However, in the switching MDP, only the state variables of the current task model and one stimulus for each one of the other task models are being computed. We used the same setup as before and included the observation cost. Fig. 4 shows how the difference between the performance, in terms of average reward, of the exact MDP, denoted by e_r, and the switching MDP, denoted by s_r, increases as the number of task models increases. Notice the difference in performance of the MDPs is 0 when the robot is only scheduled to execute the delivery task. Although adding each trash cleaning task to the task models only increases the number of state variables by 2, with 6 task models the exact MDP on average processes 89 more state variables than the switching MDP.

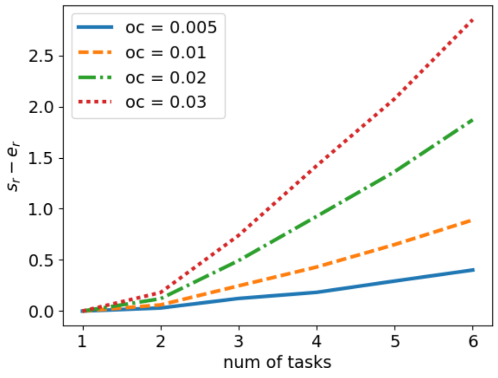

Figure 4: Difference between the performance of the exact MDP (e_r) and the switching MDP (s_r) when there is an observation cost (oc) for processing each sensory variable.

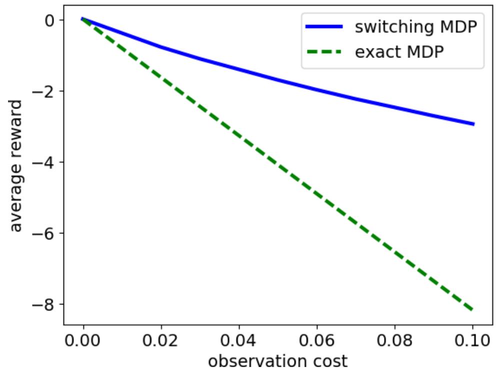

Figure 5: Average reward that the robot gains for 3 tasks as we increase the observation cost by 0.01.

3. In the third evaluation, we incorporated the observation cost into the training phase. In addition to the positive reward for achieving the goals and negative reward for each action execution, the robot gets a negative reward equal to $\#\ sensory\ variables \times oc$ in each state. This ensures that the robot considers the cost of observations during training, and if the observation and execution cost of achieving another goal exceeds its reward, the robot will not switch to the other task. Fig. 5 shows the results of the task-switching behavior with different observations costs. The performance is computed as before for 3 tasks, 1 object delivery task and 2 trash cleaning tasks. As the observation cost increases by 0.01, the exact MDP's average reward decreases by almost 0.8, but the switching MDP's average reward only decreases by 0.3.

Discussion We tested different versions of neural networks with a different number of layers and neurons, but all other network structures degraded our results or didn't improve them. Whether a deep network is needed or just a linear network suffices depends on the target domain. We attempted to use the linear combination of basis functions [Sutton and Barto, 1998; Bethke *et al.*, 2008], but this approach did not achieve satisfactory results.

4.2 Results of Identifying Task-Switching Stimuli

To evaluate our stimuli identification algorithm, we ran multiple simulations of the task selection policy for 2 tasks (a delivery task and a trash cleaning, or a delivery task and an HRI task) with random initial values for the state variables. The sensory state variables (features) of the trash cleaning and HRI tasks are shown in Table 1 and Fig. 6. The algorithm randomly selects the value of the *willing*, *looking*, and *stopped* variables. The value of the *present* variable is initially set to 0, and it becomes 1 when the robot is 5 steps away from a human goal or a trash goal, *i.e.*, we assume that the sensor range is 5. We applied our proposed algorithm on our data and calculated the feature importances. Table 1 and Fig. 6 show the feature importances for the trash cleaning and HRI tasks respectively for 40 simulation runs. Feature importances for each task model sum to 100. We evaluated the performance of the extra-trees classifier by 5-fold cross-validation technique. The classifier's accuracy is 89% on the HRI and 76% on the trash cleaning task.

feature	*present*	*x*	*y*
mean ± std %	69 ± 31	27 ± 29	4 ± 8

Table 1: Feature importances for the trash cleaning task.

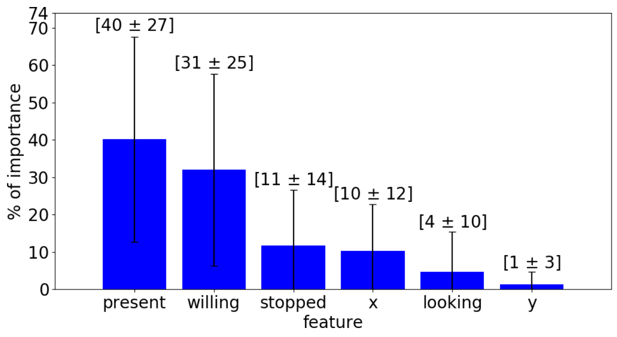

Figure 6: Feature importances for the HRI task.

Table 1 shows that the *present* feature is more important than the *x* and *y* features, so the algorithm selects it as the stimulus for the trash cleaning task. The importance of the *present* feature is not close to 100% since the robot will only switch to another task if the switch is beneficial. We observed similar results for the HRI task. The *present* and *willing* features are the most important features, and their sum of importances (71%) is almost the same as the importance of the *present* feature in the trash cleaning task (69%). Although the *looking* and *stopped* features are involved in the termination conditions of the HRI task, the robot can execute actions to change their value, so they do not affect the task-switching behavior. The *present* feature is more important than the other features, so it is selected as the stimulus for the HRI task.

We decreased the sensor range from 5 to 3 and observed that the importance of the *present* and *willing* features is 46% and 24% respectively. We increased the sensor range from 5 to 8, and we observed that the importance of the *present* and *willing* features is 26% and 43% respectively. As the sensor range increases, the *present* feature becomes less important since the robot can see the person from most places, and the *present* feature does not significantly affect the task-switching behavior. However, if we decrease the sensor range, the *present* feature becomes more important for the task-switching behavior.

Discussion We tested different ensemble approaches (averaging and boosting) on our problem. In summary, averaging methods, which use a set of strong classifiers, such as extratrees and random forest performed quite well on our dataset. However, boosting methods, which use a set of weak classifiers, such as gradient boosting and adaptive boosting (AdaBoost) performed poorly. In future work, we plan to apply other feature selection methods. Our current approach is not efficient for a continuous stimulus if its value changes constantly, *e.g.*, constant changes in battery level.

5 Conclusion

We contribute a novel approach for switching among multiple task models. We explain how we leverage stimuli to interrupt the robot's task execution and reevaluate the task selection policy. We evaluate our approach in a scenario with 1 to 6 tasks and show that it requires less sensory computations compared to the combined model. In future work, we will address the following issues: 1) how to scale up our approach for robots with more tasks and sensors, 2) how to augment the existing task planners with our task switching behavior, 3) how to factorize a huge MDP into smaller MDPs and learn the task models and the task-switching policy simultaneously, and 4) how to transfer and refine a policy that is learned in simulation to apply it in the real-world. We believe this transfer is possible since our approach uses high-level state variables (not low-level control inputs), such as position.

Acknowledgements

This research is partially sponsored by DARPA under agreements FA87501620042 and FA87501720152, and NSF under grant IIS1637927. The views and conclusions contained in this document are those of the authors only.

References

[Bacon *et al.*, 2017] Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. In *AAAI*, pages 1726–1734, 2017.

[Barto and Mahadevan, 2003] Andrew G. Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, pages 341–379, 2003.

[Bethke *et al.*, 2008] Brett Bethke, Jonathan P. How, and Asuman Ozdaglar. Approximate dynamic programming using support vector regression. In *CDC*, pages 3811–3816, 2008.

[Choi, 2011] Dongkyu Choi. Reactive goal management in a cognitive architecture. *Cognitive Systems Research*, pages 293–308, 2011.

[Geurts *et al.*, 2006] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine learning*, pages 3–42, 2006.

[Humphrys, 1996] Mark Humphrys. Action selection methods using reinforcement learning. *From Animals to Animats*, pages 135–144, 1996.

[Jaidee *et al.*, 2012] Ulit Jaidee, Héctor Muñoz-Avila, and David W. Aha. Learning and reusing goal-specific policies for goal-driven autonomy. In *ICCBR*, pages 182–195, 2012.

[Karlsson, 1997] Jonas Karlsson. *Learning to solve multiple goals*. PhD thesis, University of Rochester, 1997.

[Kingma and Ba, 2014] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[Kober *et al.*, 2013] Jens Kober, J. Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, pages 1238–1274, 2013.

[Konidaris, 2016] George Konidaris. Constructing abstraction hierarchies using a skill-symbol loop. In *IJCAI*, page 1648, 2016.

[Kulkarni *et al.*, 2016] Tejas D. Kulkarni, Karthik Narasimhan, Ardavan Saeedi, and Josh Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. In *NIPS*, pages 3675–3683, 2016.

[Martinson *et al.*, 2001] Eric Martinson, Alexander Stoytchev, and Ronald C. Arkin. Robot behavioral selection using q-learning. Technical report, Georgia Institute of Technology, 2001.

[Matarić and Michaud, 2008] Maja J. Matarić and François Michaud. Behavior-based systems. In *Springer Handbook of Robotics*, pages 891–909. 2008.

[Mnih *et al.*, 2015] Volodymyr Mnih, Koray Kavukcuoglu, and David Silver et al. Human-level control through deep reinforcement learning. *Nature*, pages 529–533, 2015.

[Muñoz-Avila *et al.*, 2015] Héctor Muñoz-Avila, Mark A. Wilson, and David W. Aha. Guiding the ass with goal motivation weights. In *Goal Reasoning: Papers from the ACS Workshop*, pages 133–145, 2015.

[Nicolescu *et al.*, 2006] Monica Nicolescu, Odest C. Jenkins, and Adam Olenderski. Learning behavior fusion estimation from demonstration. In *ROMAN*, pages 340–345, 2006.

[Pedregosa *et al.*, 2011] Fabian Pedregosa, Gaël Varoquaux, and Alexandre Gramfort et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, pages 2825–2830, 2011.

[Pirjanian, 1999] Paolo Pirjanian. Behavior coordination mechanisms-state-of-the-art. Technical report, University of Southern California, 1999.

[Raïevsky and Michaud, 2008] Clément Raïevsky and François Michaud. Improving situated agents adaptability using interruption theory of emotions. In *SAB*, pages 301–310, 2008.

[Sutton and Barto, 1998] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: An introduction*. Cambridge: MIT press, 1998.

[Sutton *et al.*, 1999] Richard S. Sutton, Satinder P. Singh, Doina Precup, and Balaraman Ravindran. Improved switching among temporally abstract actions. In *NIPS*, pages 1066–1072, 1999.

[Vattam *et al.*, 2013] Swaroop Vattam, Matthew Klenk, Matthew Molineaux, and David W. Aha. Breadth of approaches to goal reasoning: A research survey. Technical report, Naval Research Lab Washington DC, 2013.

[Veloso *et al.*, 2015] Manuela M. Veloso, Joydeep Biswas, Brian Coltin, and Stephanie Rosenthal. Cobots: Robust symbiotic autonomous mobile service robots. In *IJCAI*, page 4423, 2015.

[Wang *et al.*, 2015] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015.