A Review of Social Hierarchical Learning

George Wildridge
Social Robotics Lab
Yale University
51 Prospect St, New Haven, Connecticut 06520, USA
gwildridge17@choate.edu

Abstract— **As robots begin to transition into our everyday lives, a great potential opens for robots to collaboratively aid humans in the completion of everyday tasks. Social Hierarchical Learning (SHL) is an area of active research within Machine Learning (ML) that aims to enable effective human robot collaboration. This is accomplished through extending the capabilities of Hierarchical Learning (HL). This review paper will discuss the present state of SHL, beginning with the underlying concepts and progressing to the areas of active research.**

Keywords—**Markov Decision Processes; Reinforcement Learning; Hierarchical Learning; Learning from Demonstration; Deep Learning.**

I. INTRODUCTION

As of right now, robots most often execute tasks in rigidly defined state spaces with very small degrees of unpredictability. Further, they do this from behind barriers that serve to protect human workers. SHL aims to bring robots into more collaborative environments; however, doing so requires a robot to have the ability to interact with an unpredictably changing environment and have a high level of self-awareness. In order for a robot to collaborate effectively, it should be able to reduce the physical or cognitive workload of fellow teammates. It sets about these tasks through use of various methods within ML including: Markov Decision Processes, Reinforcement Learning, Hierarchical Learning, Learning from Demonstration and, potentially, Deep Learning.

II. BACKGROUND

A. History of Machine Learning, Artificial Intelligence and Data Mining

The field of ML and Artificial Intelligence (AI) have their roots in the middle of the 20th century with Alan Turing's 1950's creation of the "Turing Test", which determines a computer's intelligence. The test stipulated that for a computer to have real intelligence it must be capable of fooling a human being into believing that it is also human. Two years later, in one of the first application of ML and AI, Arthur Samuel wrote a program to play checkers that would improve at the game the more it played. Over the next forty years researchers took a knowledge based approach to ML and AI. This approach makes the assumption that learning is made up of symbolic systems, or physical patterns, that can be manipulated to achieve many aspects of intelligence.

Submitted 5/31/16

Concurrently, a much smaller number of researchers were taking a data based approach; focusing on pattern recognition and information retrieval. As the symbolic approach evolved into the field of AI known today, the data based approach became the field of ML. In 1957 the first attempt at this area was made by Frank Rosenblatt with the perceptron, also known as a neural network. Although initially too computationally costly, with the reinvention of the backpropagation algorithm in 1986 by Hopfield, Rumelhart and Hinton, neural networks have become much more widespread as they require much less computational power and, in recent years, have had incredible results.

Over this same time period, the field of data mining also rose to prominence. Like ML, it too was based on large coalescences of data. What differentiates it from ML and AI; however, is that data mining focuses on the discovery of unknown properties within data. However, even with the key differences between these three fields, they frequently go hand in hand.

B. Markov Decision Processes

Markov Decision Processes (MDPs) are defined by a 4 tuple (S, A, R, T). Having a set of states $(s \in S)$, a set of actions $(a \in A)$, a transition model $T(s, a, s')$ which defines the probability of moving to the next state s' from the current state s taking action a, and a reward model $R(s, a, s')$ which defines the reward achieved through transition between state s and the next state s' taking action a. The goal of most MDPs is to find an optimal policy, π^*, that defines the best possible action to take in every next possible state with the objective of maximizing the expected sum of rewards. As MDPs are grounded in the Markov Property (the future is independent of the past given the present), only future rewards are maximized and past rewards are treated as irrelevant. As some element of chance may be associated with whether an action decided upon is undertaken, a tuple of $Q(s, a)$, called a q-state, must also be introduced as an intermediary between states as seen in Fig. 1.

Within MDPs there is a desire to have stationary preferences which mitigates the necessity for multiple policies to be defined through limiting the way future rewards are defined to either an additive utility:

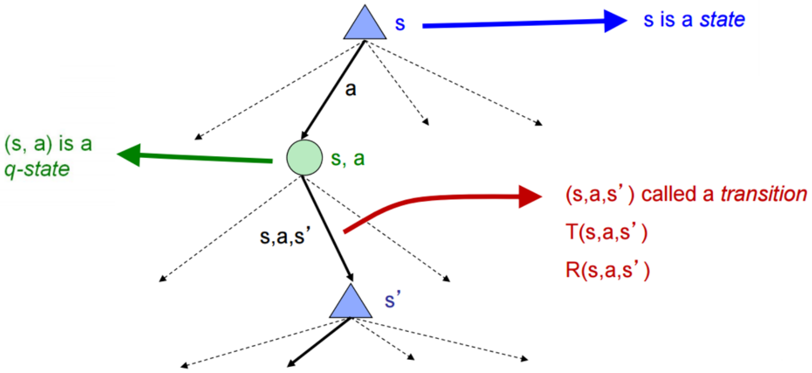

Fig. 1. Model of a Markov Decision Process. From [15]

$$U([r_0, r_1, r_2, ...]) = r_0 + r_1 + r_2 + ... \tag{1}$$

Or a discounted utility:

$$U([r_0, r_1, r_2, ...]) = r_0 + \gamma r_1 + \gamma^2 r_2 + ... \tag{2}$$

In the latter equation γ functions as a discount factor that enables the programmer to weight near future rewards over long distance rewards. The discount utility is necessary for two reasons: it usually weights near future rewards over long term rewards with a value between 0 and 1 to account for uncertainty within the model, and it also sets a finite horizon. A finite horizon is important in case the model lasts forever as the behaving agent would be forced to continuously be adding future rewards. Using $0 < \gamma < 1$ a finite horizon can be set using this equation:

$$U([r_0, ...r_\infty]) = \sum_{t=0}^{\infty} \gamma^t r_t \leq R_{max}/(1-\gamma) \tag{3}$$

Alternatively, the MDP could stop counting rewards after a fixed number of states. However, this approach runs into issues because it creates a nonstationary policy that is dependent on the time left. Additional problems arise if the behaving agent needs to look far into the future.

To help solve for the optimal policy, various optimal utilities must be introduced including a value utility of a state s and a value utility of a q-state. The value utility of state s, $V^*(s)$, provides the expected utility starting in s and acting optimally while the value utility of a q-state, $Q^*(s, a)$ defines the expected utility from starting out having taken action a from state s and thereafter acting optimally. These quantities can be defined recursively:

$$V^*(s) = \max_a Q^*(s, a) \tag{4}$$

With $V^*(s)$ being equal to the best available s over all actions.

$$Q^*(s, a) = \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V^*(s')] \tag{5}$$

If one assumes you are landing in only one s' the expected reward of $R(s, a, s')$ and the future expected rewards $\gamma V^*(s')$ must be weighted by $T(s, a, s')$ or the probability of actually landing in state s'. As the agent is not always landing in only one next state s', all the possible next states s' are summed. These two equations can be combined to form an equation known as the Bellman equation:

$$V^*(s) = \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V^*(s')] \tag{6}$$

As MDPs often use discounting, the very deep parts of the tree eventually do not matter. As this is the case an idea called time-limited values can be introduced to help solve this equation. In which the quantity $V_k(s)$ is the optimal value of s if the game ends in k more time steps. Furthermore, a technique called value iteration can be used in which the computer propagates from the bottom of the MDP to the top. If it is assumed some vector $V_k(s)$ is given then $V_{k+1}(s)$ can be computed:

$$V_{k+1}(s) \rightarrow \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V_k(s')] \tag{7}$$

This recursively defines $V_{k+1}(s)$ as a function of $V_k(s)$ and the model that is given for the MDP. Although this would ideally be repeated until infinity, if started with some arbitrary initial function V_0, $V_{k+1}(s)$ is found to converge if k is high enough, effectively yielding the unique optimal values, $V^*(s)$ that were initially sought after. An issue arises with this approach as the policy often tends to converge long before the values do. However, it is still necessary to extract a policy from these values.

A policy is more complicated than just moving in the direction of the state with the highest values as oftentimes transition probabilities must also be consulted. If $V^*(s)$ is given, then a one-step look ahead with expectimax looks like:

$$\pi^*(s) = arg \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V^*(s')] \tag{8}$$

This equation gets the policy implied by the values, hence it is called policy extraction. The difference between this and the bellman equation is that it returns the optimal action instead of the optimal value. A new concept called policy iteration can now be introduced as it might work faster than value iteration for certain problems as it specifically addresses the issue that the actions converge prior to the values. The first step of policy iteration is policy evaluation in which the utilities are calculated for some fixed policy until convergence. This could look like this:

$$V_{k+1}^{\pi_i}(s) \rightarrow \max_a \sum_{s'} T(s, \pi_i(s), s')[R(s, a, s') + \gamma V_k^{\pi_i}(s')] \quad (9)$$

Which looks a lot like value iteration, with the exception that there is no max function over actions as the behaving agent is following a fixed policy and thus has a fixed action. The second step is to improve the policy through updating it using

$$\pi_{i+1}(s) = arg \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V^{\pi_i}(s')] \quad (10)$$

with resulting converged utilities as future values.

Value iteration and policy iteration represent the two extremes in seeking the optimal policy. Value iteration essentially finds the optimal action with (10) and then updates the value whereas policy iteration keeps updating the values until you have the value of that policy and then switches to updating the actions. However, both have the same goal of computing the optimal policy.

C. Semi Markov Decision Processes

A Semi Markov Decision Process (SMDP) differentiates itself from a regular MDP in that the amount of time that passes between decision stages is relevant. In an SMDP, the amount of time between decisions is a random variable, either integer- or real- valued. The integer-valued variable is especially relevant to HL as decisions can only be made at positive integer multiples of an underlying time step which allows instantaneous transitions to the next state at the end of the random waiting time. A fact that will become of importance in section 2F. The Bellman equations for V^* and Q^* become

$$V^*(s) = \max_{a \in A_s} [R(S, a) + \sum_{s', \tau} \gamma^\tau P(s', \tau | s, a)(s, a, s') V^*(s')] \quad (11)$$

for all $s \in S$, and

$$Q^*(s, a) = R(S, a) + \sum_{s', \tau} \gamma^\tau P(s', \tau | s, a)(s, a, s') \max_{a \in A_s} Q^*(s', a') \quad (12)$$

for all $s \in S$ and $a \in A$ within an SMDP. Where the random variable τ denotes the waiting time for state s when action a is executed. The joint probability $P(s', \tau | s, a)$ that a transition from state s to state s' occurs after τ time steps when action a is executed as given by the generalization of the transition function. The immediate rewards $R(s, a)$ give the amount of discounted reward expected to accumulate over the waiting time in s given action a.

D. Partially Observable Markov Decision Processes

Partially Observable Markov Decision Processes (POMDPs) are useful to model real world situations in which situational awareness is not perfect (like working with a human co-worker). POMDPs can be built on top of previously designed task networks, giving them some intuition of likely next states given certain actions. However, the precise state of system within a POMDP is not known and can only be reasoned about probabilistically given a dataset of examples. Formally POMDPs are represented by the 6-tuple (S, A, R, T, O, Ω) where O represents a set of observations (output from state S), and Ω describes the conditional probability of an observation given a state and an action.

E. Reinforcement Learning

Reinforcement Learning (RL) is based off the idea that an agent can learn from positive and negative reinforcement of its actions. MDPs provide a framework for reinforcement learning where an agent still has a set of states $(s \in S)$, a set of actions $a \in A$ and it still is searching for an optimal policy π. What differentiates RL from MDPs is the lack of either a transition model $T(s, a, s')$ or a reward function $R(s, a, s')$ or, as most often is the case, both. Thus the learning aspect of RL can be introduced as the agent must try actions and visit states. Within an MDP, offline learning can take place where, prior to taking an action, a plan can be created based on knowledge of the state space, a reward function and a transition model to find the best path through the processes. Within RL, where there is no knowledge of future rewards or the transition model, learning must take place through choosing random actions and seeing what happens.

1) Model-Based Learning: Model based learning approximates a model based on experience before solving for values as if the learned model were correct. To learn an empirical MDP model, count the outcomes s' for each s and a and then normalize to give an estimate of $T(s, a, s')$. After which R(s,a,s') can be discovered through experiencing (s, a, s'). Finally, methods like value iteration or policy iteration can be used to find the optimal policy.

F. Hierarchical Learning

HL is an extension of RL that introduces various forms of abstraction into problem solving and planning systems. Abstraction often comes in the form of a macro-operator, or a macro, which is a sequence of operators or actions that can be invoked as if it were essentially a single action. As macros can include other macros in their definitions, macros form the basis of hierarchical specification of operator or action sequences. Hierarchical approaches to RL generalize the macro idea to closed-loop policies, or more precisely, closed loop partial policies because they are generally defined for a subset of state s.

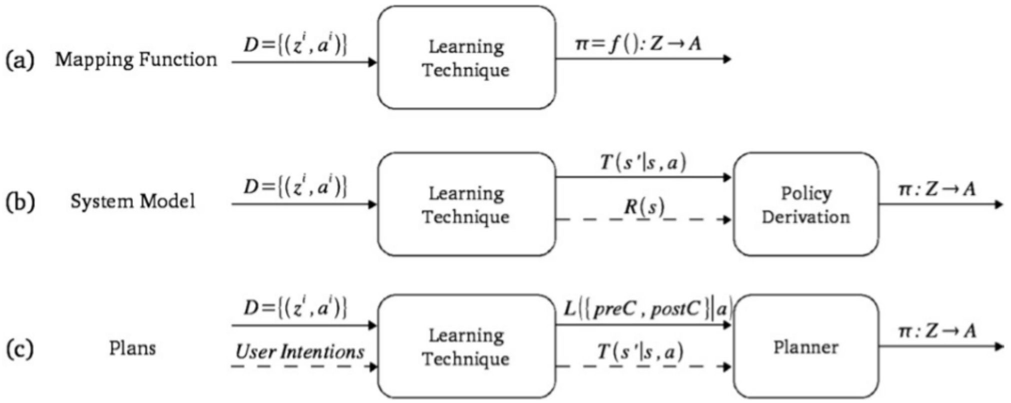

Fig. 2. Policy learning techniques within LfD. Adapted from [3]

1) Options Approach: Building on top of a finite MDP, called the core MDP, the simplest type of option consists of a stationary, stochastic policy $\pi : S \times \cup_{s\in S} A_s \to [0, 1]$ that indicates how the robot should act when following the option, a termination condition $\beta : S \to [0, 1]$ that gives the probability of terminating the option per each state in the option, and an input set $I \subseteq S$. Formally, an option is defined as a 3-tuple $(I, \pi, \beta.)$ In simpler words an option is the coalescence of low level actions with an input, a termination state and a policy to dictate which actions to take. Once an option is selected, all actions in its sequence are followed until termination, upon which a new option can be selected. As an options is treated as a single action, option policies have the ability to call other options prior to termination. This provides a framework for HL as related low level actions can be grouped together and easily called from anywhere in the MDP, giving the agent the ability to reason at a higher cognitive level.

An options policy sets action probabilities in manner that is independent of the past but dependent on the present, thus fulfilling the Markov Property. Giving it another, more specific name, a Markov option. More flexibility is needed, however, to allow an options policy to set action probabilities based on the history of states, actions and rewards since the option was initiated. To allow for this, semi-Markov options are often included, and with them all the theory and algorithms applicable to SMDPs can be adjusted for decision making with options as the policy of each action is semi-Markov. A high level policy over options, μ, selects options upon termination with probability $\mu(s, o)$. μ also determines a conventional policy π at the core MDP called the flat policy. This policy π often is not Markov either as it is following Semi-Markov options.

G. Learning From Demonstration

Learning from Demonstration(LfD) provides an intuitive method to impart knowledge into a robot, allowing experts and novices alike to program robot behaviors. Using a dataset of examples provided by a teacher, LfD algorithms attempt to define a policy, π, to reproduce the examples. This differentiates it from RL where a π is derived from data consequent of exploration. Imparting knowledge to a robot can be done in a multitude of different ways, including teleoperation, kinesthetic teaching, or guided by human speech. Policy derivation is most often done using three different methods, as seen in Fig. 2, defined as mapping functions, system model and a planning approach.

1) Mapping Function: Within a mapping function the underlying function mapping from a robots state observations to actions is approximated directly. The goal of the mapping function is to reproduce the unknown underlying teacher policy and to generalize over the set of available training examples such that valid solutions are also acquired for similar stats that may not have been encountered during demonstration. This is done through the calculation of a function that approximates the state to action mapping $f() : Z \to A$.

2) System Model: Within a system model, demonstration data is used to determine a model of the world dynamics with a state transition model, $T(s, a, s')$, and possibly a reward function $R(s)$ from which a policy is derived. This approach often utilizes RL.

3) Planning Approach: Finally, within a planning approach, the policy is represented as a sequence of actions from the initial state to the terminal state. Through use of demonstration data and, often, additional information from the teacher in the form of annotations or intentions, a set of rules are defined to associate a set of pre-conditions (the state that must be established before the action can be performed) and post-conditions (the state resulting from action executions) with each action, $L(preC, postC|a)$. It is also possible to incorporate a transition function, $T(s, a, s')$, before planning a sequence of actions.

H. Deep Learning

The driving force behind deep learning is to recognize and learn feature hierarchies within data. This is most helpful in fields like computer vision, where a neural network is tasked of learning features from images, however, it is also used in cases of RL, as an efficient method to store a policy.

Although, the impact of deep learning on SHL is relatively limited, it has great potential for future incorporation.

To understand what deep learning, or neural nets, is doing, a great place to start is with a linear classifier. Which in its most basic form can be represented as

$$f = Wx \tag{13}$$

where x is some input, whether it be an image or a world state, and W is a matrix of arbitrary numbers called weights with as many rows as input dimensions and as many columns as classes. The result of this matrix multiplication is a set of scores. Ideally the optimal action or class would correspond with the highest score value. However, especially in initial stages, this often not the case. To account for this, a classifier like a support vector machine or the softmax classifier are introduced to quantify how poorly the linear classifier is working. The value of these classifiers can then be used to gradually shift the weights using gradient descent so as to improve the scores in another pass. After many iterations over the dataset, optimal values for the weights can be found.

The difference between a linear classifier and a neural network are a couple of layers of complexity. Neural networks embed linear classifiers within each other. To understand this, it is easiest to look at the equation of a three-layer neural network:

$$f = W_3 \max(0, W_2 \max(0, W_1 x)) \tag{14}$$

and then a two-layer neural network:

$$f = W_2 \max(0, W_1 x) \tag{15}$$

and, again, a linear classifier:

$$f = Wx \tag{16}$$

As more layers are added to the network, so too are more sets of weights. This gives the ability for more detail to be stored about the features within an world state or an image, and to an extent leads to greater precision in classification. Optimizing the weights of neural networks is still equally as important as optimizing the weights of the linear classifier, though it too becomes more complex. Through the recursive application of chain rule, the agent is capable of backpropagating through the neural network in an effort to determine which weights have the most effect on the output. After, these weights can be incremented using gradient descent to help improve the accuracy of the neural network on the next pass. Through repeated iteration, neural networks can become extraordinarily good at classifying certain tasks. For more information of the specifics of Neural Networks see ().

III. Present Work

SHL consists of three phases: primitives acquisition, plan decomposition and cooperative execution.[4]

Primitives acquisition, in simpler terms, means learning and acquiring component sub-skills. Skills are acquired through kinesthetic LfD and learned through the development of an SMDP for each behavior. Concurrently a POMDP is developed to model human movement intentions. The SMDP and the POMDP are then mapped together so as to tie the behavior of the robot with the lead worker. Skills can be refined using a short term method and a long term method that are dependent on the length of time since the robot was trained. In the short term, the teacher can kinesthetically retrain the robot using a similar SMDP development: instead of overwriting the old data, the new data is added to the underlying SMDP, this in conjunction with a change in the reward function allows the robot to act more flexibly in the future. Over longer time periods reinforcement training can help refine the robots skills. Within SHL specific assistive tasks are focused on such as materials stabilization and retrieval, collaborative object manipulation, enhancing awareness and task progression guidance.

The second phase, plan decomposition, involves the agent learning to sequence primitive actions into a complex hierarchical task representation using HL. This phase specifically addresses the question of when to apply the skills and actions. Skills and actions are chosen using previously gathered information on the duration to complete actions to inform a reward function which chooses the best possible action through policy iteration. Within SHL the effectiveness of the robot is gauged on its ability to affect the time it takes to complete an action. With this in mind, there is a desire to optimize the policies of the POMDP and the SMDP to decrease the time spent in any given state s. Cooperative execution is the need for a robot to work collaboratively in an effective and safe way within a live, unpredictable environment using RL. When executing behaviors, instead of using an SMDP to directly plan motion, the SMDP is used to inform a standard motion planner through the input of valid parameters like position, pose, orientation, etc. [6]

Still, many questions are left unanswered within SHL. As the scope of the task is quite broad, current and future research surrounds four questions:

A. How can co-robots enable and facilitate bi-directional intent recognition?

Another way to phrase this question is how can a robot understand a humans intentions and convey its own? Answering this question requires communication between the robot and the human, either verbally or nonverbally. Examples of nonverbal communication include gestures and facial expressions. The answer is important in cases like object transfer where two agents require coordination. It is also vital for maintaining a safe space for a human collaborator as a robot must be able to effectively convey its intentions, especially when its participating in potentially dangerous tasks like using a drill or a knife. Further, the robot must be able to anticipate where objects will be placed or moved while in use as the availability of objects is highly influential as to what actions are possible. [5]

1) Social Force: Social force is defined as a projection of ones intention through movement. Using a model of the agent along with the agents motion history, social force quantifies the likely future positions of an agent at any given point in the state space. Although it may not be readily apparent; how social force is interpreted makes large strides in the realm of human robot collaboration as it allows a robot to fluidly transition between an instructor, peer and learner.

When social force is treated as an attractive, positive force within the action planning system, the robot will function as a learner. When the likelihood of an agents future position is higher than some arbitrary threshold value, the robot will treat the agents action as if it is a gesture inviting the robot to execute some action in relation to that point. The order of actions the agent instructs the robot to follow further helps to optimize the task-level preferred option orderings.

If social force is treated as a negative, repulsive force within the action planning system the robot will act as a peer. The fundamental idea behind this approach is the robot will choose actions that enable it to work in parallel with the other agent and minimize possible collisions.

Finally, social force can be used as a trigger once it passes a particular threshold, causing the robot to act as an instructor. As an instructor the robot will perform keyframes of skills it desires an agent to learn, progressively completing more and more keyframes within a skill until either the human understands the robots intentions or the robot demonstrates the skill. [7]

B. How does a co-robot know what roles to take when working with human teammates?

For a robot to act as an effective member of a team, it must be able to solve problems like how to generate role divisions within a task taking into account both social and practical task demands, how to communicate these divisions with a co-worker, recognize implicit communication and anticipate the needs or actions of co-workers, be able to accommodate the fact that roles may change during the execution of a task or that its partner might not fulfill its commitments. Further, it is necessary to identify what roles a teammate has chosen to undertake in a task for the robot to go about non-conflicting actions. [5]

1) Discovering Task Constraints through Observation and Active Learning: The first step towards a solution to this overarching question is figuring out how to differentiate tasks and roles in the first place. This entails having a high level understanding of what skills, or low level actions, are or are not related. A solution to this can be found utilizing the robots observation of the task and active learning. Active learning is the ability for a learner to generate or select examples to be labeled by an oracle. A concept that is particularly useful in ML as a robot can direct an agent to help mitigate areas of learning that are unclear which helps to make its policies more robust in the presence of a lack of diverse data. As more and more observations of successful skill execution sequences are presented to the learner, the connections, or lack thereof, become greater defined. Allowing a robot to traversal task networks more effectively while simultaneously aiding in a robots understanding of the task. In examining the effects active learning has on overall understanding of task constraints, three separate query strategies were tested: a baseline random query method and two feature based query methods (distance based and connectivity based). In practice, the feature based methods were found to significantly outperform the random method. A second experiment tested the effect the number of instructors have on a robots understanding of the task constraints. Results pointed to using multiple instructors over a single instructor. For more insight on these tests see [8].

C. How does a co-robot know when to trade off task execution optimality for co-worker preferences?

Within human robot teams it is inevitable for one agent to be more suited for some tasks then others. For instance, tasks where a similar action is repeated over and over like squeezing oranges is more suitable for a robot, while more dexterous tasks like shirt buttoning are more appropriate for humans. The robot must be able to recognize its and its partners strengths and weaknesses in order to complete the task efficiently. In addition, a robot must consider the psychological costs of asking a human to do tasks against their preferences and how that will effect the collaborative dynamic. Further, for a robot to determine which trade-offs are most important, the robot needs some metric of quantifying its partners performance versus its own on a given task.[5]

1) Effect of Empathy on SHL: In the case that a robot desires a human to trade-off their preferences within a task for the purpose of optimization, the robot must have built up a dynamic between itself and human in which it is capable of getting what it wants. In an effort to gauge human response to robot requests, the effects of empathy generating robot dialogue upon humans was examined. Particularly, the empathy found in humans if the robot is asking for help in its own interest or out of interest for someone else.

This was done through tasking a human to compete against a robot to count the number of items in a bin. Three conditions were given to the robot in which the dialogue was varied. In the control condition, the robot made neutral comments on the game. In the self-directed condition, the robot implored the human to slow down for it to have a chance to catch up (measuring the empathy associated with the robot). Finally, in an externally directed condition where the robot asked the human to slow down for the programmers benefit (measuring the empathy associated with the robot if it is working on behalf of a human). Through looking at the human response to all three it was determined if the human feels empathy for a robot attempting to help itself or for a robot attempting to help another human. It was found that the human was more likely to be empathetic to an externally directed robot then to either an internally directed robot or the robot in the control condition. [9]

D. How does a co-robot self-evaluate during live collaboration?

A collaborative robot must be able to gauge the quality of its contributions to a task. This requires a shared mental model of a task with its co-worker, knowledge of a coworkers roles and responsibilities and accurate estimates of expected overall task progress at given times. Further, a robot must be able to understand whether it has properly completed a task.[5]

1) A Transfer Learning Approach for Predicting Skill Durations: As duration is an indication of ability, a method of self-evaluation is to compare the length of time it has taken to complete a task in the past to the current amount of time it took to complete a task. With this in mind, the ability for a robot to predict skill durations becomes essential whether it is evaluating its own ability, or comparing itself to others. Giving a robot the ability to predict skill durations can be done through building models to estimate the duration of sub-tasks that can be obtained from the decompositions of complex tasks as accomplished through transfer learning and LfD. Transfer learning is a well-studied area within robotics and psychology that leverages the idea that one can generalize across topics as well as within them. Using annotated video data, an agents skill performance was predictively modeled using four types of features: expected skill duration, a skills experience curve, an agents tool proficiencies, and an agents motor skill proficiencies. The model was evaluated based on its performance adapting to new tasks and extrapolating on known data. The results of the general model in the context of estimating the duration of a novel task were a 63.8% error; however, training the model with demonstrations of tasks sharing some of the same tool requirements and motor skills increased performance. When trained on one task the models error fell to 36.8%, and when fully trained on two tasks the models error was 23.6%. The model was also tested on its ability to estimate the future performance of an agent and yielded a result of 20% error without the use of transfer learning and 13% error with it. For more information see [H].

IV. CONCLUSIONS

In recent years, Social Hierarchical Learning has yielded promising results towards effective human robot collaboration. Still; however, many questions are left unanswered and it is still very much an active area of research.

ACKNOWLEDGMENT

The author would like to thank Dr. Bradley Hayes, Dr. Alessandro Roncone, Dr. Olivier Manguin, Dr. Christopher Hogue and the Science Research Program for their support in the research process.

REFERENCES

[1] A. Barto and S. Mahadevan, Recent advances in hierarchical reinforcement learning, Discrete Event Dynamic Systems, vol. 13, no. 4, pp. 341379, 2003.

[2] B. Argall, S. Chernova, M. Veloso, and B. Browning, A survey of robot learning from demonstration, Robotics and Autonomous Systems, vol. 57, no. 5, pp. 469483, 2009.

[3] C. Atkeson and S. Schaal, Robot learning from demonstration, in International Conference on Machine Learning, 1997, pp. 1173.

[4] Hayes, B., and Scassellati, B. (2013). Social Hierarchical Learning. In: Proceedings of the "HRI Pioneers" Workshop at HRI 2013. Tokyo, Japan, March 3.

[5] Hayes, B., and Scassellati, B. 2013. Challenges in shared environment human-robot collaboration. In Proceedings of the 8th ACM/IEEE International Conference on Human Robot Interaction (HRI 2013) Workshop on Collaborative Manipulation.

[6] Hayes, B., and Scassellati, B. 2014b. Online development of assistive robot behaviors for collaborative manipulation and human-robot teamwork. In Proceedings of the Machine Learning for Interactive Systems (MLIS) Workshop at AAAI 2014.

[7] Hayes, B., and Scassellati, B. 2013b. Improving implicit communication in mixed human-robot teams with social force detection. In 2013 IEEE Third Joint International Conference on Development and Learning and Epigenetic Robotics. IEEE.

[8] Hayes, B., and Scassellati, B. 2014a. Discovering task constraints through observation and active learning. In IEEE/RSJ International Conference on Intelligent Robots and Systems.

[9] Hayes, B., Ullman D., Alexander E., Bank C., and Scassellati, B. 2014. People Help Robots Who Help Others, Not Robots Who Help Themselves. In Proceedings of the 23rd IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN 2014). Edinburgh, Scotland, August 25-29.

[10] Hayes, B., and Scassellati, B. (2014). Developing Effective Robot Teammates for Human-Robot Collaboration. In Proceedings of the "Artificial Intelligence and Human-Robot Interaction" (AI-HRI) Fall Symposium. Arlington, Virginia USA, November 13-15.

[11] Hayes, B., Grigore, E., Litoiu, A., Ramachandran, A., Scassellati, B. (2014). A Developmentally Inspired Transfer Learning Approach for Predicting Skill Durations. In Proceedings of the 4th joint IEEE International Conference on Development and Learning and Epigenetic Robotics (ICDL 2014). Genoa, Italy, October 13-16.

[12] Hayes, B. (2015). Social Hierarchical Learning (Extended Abstract). In Proceedings of the 20th AAAI/SIGAI Doctoral Consortium. Austin, Texas, USA, January 26-27.

[13] Wikipedia.org (Date Accessed: May 31st, 2015) `https://en.wikipedia.org/wiki/Machine_learning`

[14] Marr, B. (2016) A Short History of Machine Learning Every Manager Should Read. In Forbes. `http://tinyurl.com/gslvr6k`

[15] Abbeel, P. (2013) Lecture 8: Markov Decision Processes (MDPs). `https://www.youtube.com/watch?v=i0o-ui1N35U`