Learning Executable Agent Behaviors from Observation

Andrew Guillory, Hai Nguyen, Tucker Balch, Charles Lee Isbell, Jr.
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{guillory, haidai, tucker, isbell}@cc.gatech.edu

ABSTRACT

We present a method for learning a human understandable, executable model of an agent's behavior using observations of its interaction with the environment. By *executable* we mean that the model is suitable for direct execution by an agent. Traditional models of behavior used for recognition tasks (*e.g.*, Hidden Markov Models) are insuffcent because they cannot respond to input from the environment. We train an Input/Output Hidden Markov Model where the output distributions are mixtures of learned low level actions and the transition distributions are conditional on features calculated from the agent's sensors. We show that we are able to recover both the behavior and human-understandable structure of a simulated model inspired by animal behavior studies. We also present a novel training method that combines multiple EM trials through discrete optimization.

1. INTRODUCTION

An ethologist studying the behavior of an animal such as an ant or bee typically starts by defining a set of low-level actions the animal performs. For example, in social insect foraging, these actions include searching for a food source and exploiting a food source [16]. Having defined such low-level actions, the scientist then determines high-level structure; that is, she models what causes the animal to switch between low-level actions. Similarly, a roboticist or agent developer might program the behavior of a robot by coding low-level control modes corresponding to primitive actions and combining these low-level control modes together through some kind of high-level action selection mechanism.

In both ethology and robotics it is common to represent the high-level behavior as a discrete state model like a Finite State Machine or Markov Chain where each state is assigned to a low-level action. The transitions between states in the model may be random or more generally conditional on the agents perceptions. In this paper we present a method for learning both low-level actions and a high-level switching model. The result is human-understandable, but also *executable* in the sense that it can be used as a control program in a simulation or as part of an agent, making the method of broad interest. For example, ethologists could use it to learn models of social insect behavior from tracking data, examining the models in an attempt to better understand the underlying behavior and executing the models in simulation to predict behavior in different situations. We more formally define the problem as:

- **Given:**
  1. A model of the agent's perception;
  2. A set of perceptions that may cause the agent to switch behaviors; and
  3. Example trajectories of the agent interacting in its environment (tracking data), some of which has been labeled by a human expert, designating the low-level action the agent is performing at a given time step.
- **Assume:** The agent acts according to a Markov Process with inputs
- **Compute:** An executable model of the agents behavior composed of controllers for low-level actions and rules for switching between them

We divide the task of learning an executable model into the separate problems of 1) learning controllers for low-level actions from labeled data and 2) learning high-level switching from unlabeled data using our model of low-level actions. Thus, it is not necessary that the human expert label any data where the agent exhibits switching behavior–the expert can simply label portions of the data where the agent is obviously performing a particular action.

Our approach is to train an Input/Output Hidden Markov Model where the output distributions are mixtures of learned low-level actions and the transitions are conditioned on perceptions of the agents. We first learn from labeled data a set of low-level actions that may be assigned to states in the model. Having learned these low-level actions, the problem is then to learn from unlabeled data a mapping between states and low-level actions as well as the transition rules between states. It is necessary to learn the mapping because we do not wish to assume a one-to-one correspondence between states and low-level actions (there may be, for example, more than one state within the model corresponding to the same low-level action).

After reviewing related work, we present a simulated model of social foraging as an example problem domain. Next we formally define an *Input/Output Hidden Markov Model* and show how to use it to build executable models, first assuming we already know the low-level actions, then discussing how to also learn the low-level actions from data. Finally, we present results applying our approach to the simulated foraging domain, and introduce a new training method that improves performance on our problem.

2. RELATED WORK

Our work is in contrast to *activity recognition.* Although models of behavior learned in activity recognition are often generative, they are not usually suitable for re-creation. Hidden Markov Models, for example, are generative models but are usually severely limited in their ability to recreate real activities because they lack the notion of responding to input from the environment. In other words, Hidden Markov Models can only represent open-loop control policies, where we believe for a model to be truly executable it should also be able to represent closed-loop control policies. In activity recognition, domain knowledge is typically represented as high-level knowledge of the structure of the behavior, for example in the form of topology restrictions on the model or high-level grammars. We cannot assume prior knowledge of any high-level structure, because recovering human-understandable structure is one of our goals.

Our work is more similar in goal to *imitation learning.* The notion of primitives in [7] is similar to our notion of low-level actions. Imitation learning for robots was applied to a foraging problem in [11]. Such techniques generally differ from our work by usually focusing more on learning and detecting low-level actions (*i.e.* mappings from senses to actions) while constrained by the kinematics of particular robots. These approaches usually use simpler high-level methods for selecting among low-level actions. Further, they typically do not seek human understandability.

Our work is most similar to that of Delmotte and Egerstedt [9] who modeled both re-creation and high-level human-understandability. They approach the problem from the persepective of control theory and recover from unlabeled data both low-level control modes and a high-level control program in the form of a motion description string. We instead approach the problem from the perspective of machine learning and graphical models: we use labeled data to recover the low-level control modes but recover a more expressive kind of high-level model in that it allows for loops in control flow as well as stochastic behavior.

3. FORAGING MODEL

We programmed by hand a model of foraging inspired by the behavior of social insects, using the TeamBots simulation platform and motor schema-based control [1]. Our experimental method was to generate data from this known model, then see if we are able to automatically recover the structure and behavior from the data. Figure 1 shows the state diagram for our model and simulation screenshots. The agents perform four low-level actions:

- **Loiter:** with gripper open, move slowly, randomly around the center of the screen (the base)
- **Explore:** with gripper open, move forward with random variation, turn away from walls and other agents when bumped into them
- **Move to target:** with gripper on trigger mode, move towards the closest target
- **Move to base:** with gripper closed, move towards the center of the screen.

There are four binary transition triggers. The agent is:

- **BUMPED:** very close to the wall or another agent
- **SEE TARGET:** can see a target within its cone of vision
- **TARGET IN GRIPPER:** holding a target
- **AT BASE:** within a certain radius of the center

If an agent bumps into a loitering agent, that agent is recruited and begins to explore. This produces behavior where agents often leave the base in small groups. Some transitions are also triggered randomly (*e.g.*, agents eventually return to the base if they do not find a target). From the simulation we created two data sets, one labeled and one unlabeled. For both data sets we ran the model in simulation with 12 agents at 33 frames per second, waiting for all of the targets to be carried to base, recording at each frame the position and orientation of all agents as well as the position of all targets. In the first data set we also recorded the low-level action being performed by each agent at each time. In the second data set, we move the targets to a different location and did not use knowledge of which low-level action the agent was performing. Our goal is then to, from the labeled data set, learn the low-level actions of the agent and, from the unlabeled data set, learn the switching behavior.

4. IOHMMS

Hidden Markov Models (HMMs) [15, 2] represent the probability of an observation sequence $P(\mathbf{y}_1^T)$ where $\mathbf{y}_1^T = \mathbf{y}_1 ... \mathbf{y}_T$ is a sequence of observation vectors. Input/Output Hidden Markov Models (IOHMMs) [4] are a generalization of this model that represent the conditional probability of an observation sequence given an input sequence $P(\mathbf{y}_1^T | \mathbf{u}_1^T)$ where $\mathbf{u}_t$ is the input vector for time t. Both HMMs and IOHMMs are discrete state models that marginalize their distributions over a sequence of discrete states x_1^T, factoring the distribution according to the dependency assumptions in Figure 2. Applying these assumptions, for each state $i = 1, ..., n$ in an HMM there is a transition distribution $P(x_t | x_{t-1} = i)$ and an output distribution $P(\mathbf{y}_t | x_t = i)$. In addition there is an initial state distribution $P(x_1)$. In each state in an IOHMM there is a conditional transition distribution $P(x_t | x_{t-1} = i, \mathbf{u}_t)$ and a conditional output distribution $P(\mathbf{y}_t | x_t = i, \mathbf{u}_t)$. Normally the initial state distribution is still $P(x_1)$, although it also makes sense to use

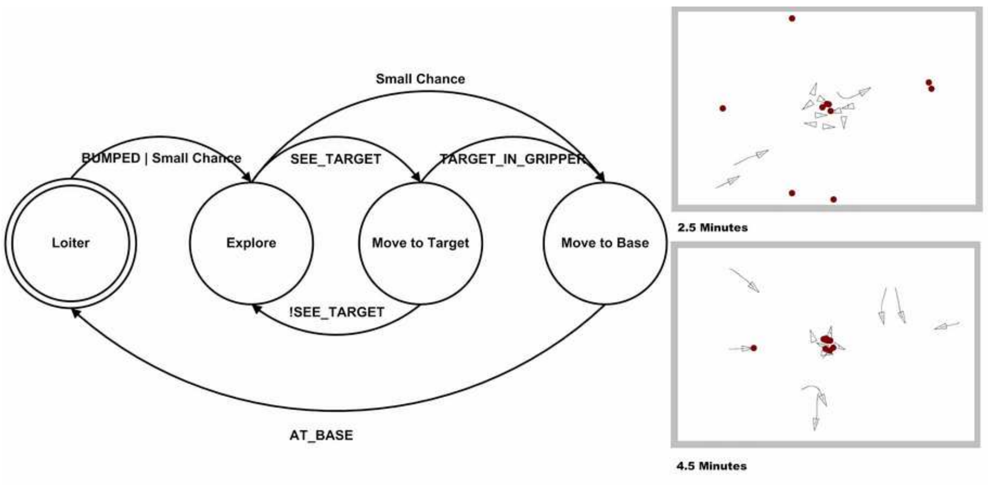


Figure 1: Our foraging model, including screenshots from the TeamBots simulation.

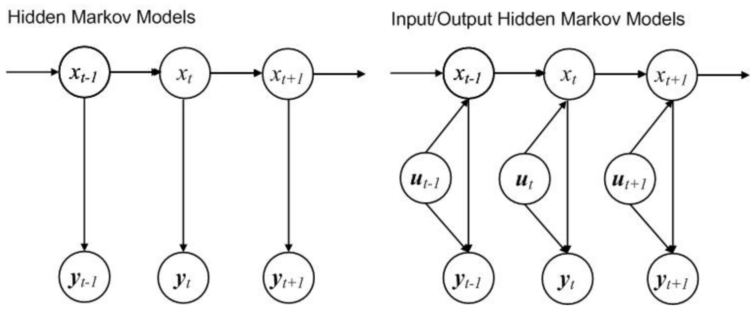


Figure 2: Bayesian networks for IOHMMs and HMMs (adapted from [4])

$P(x_1|\mathbf{u}_1)$. The standard algorithm for training IOHMMs is an Expectation Maximization (EM) algorithm that is a straightforward extension of Baum-Welch for HMMs. After training, the model can be interactively executed on an input sequence in a manner similar to a finite state machine.

IOHMMs were motivated by work in analyzing temporal credit diffusion in recurrent neural networks and traditional HMMs [6, 3]. IOHMMs are typically used in a manner similar to recurrent neural networks for sequence regression or classification [10, 5, 13, 14]. The output vectors $\mathbf{y}_t$ are the target values for classification or regression. After training, new sequences are processed online by filtering through the input vectors, calculating the most likely output at each step. This is distinct from HMMs. where the states themselves typically correspond to labels and new sequences are labeled by calculating the most likely state sequence.

5. IOHMMS AS EXECUTABLE MODELS

In our case, the input $\mathbf{u}_t$ is the tracking data at time t, including the position of the agent. The corresponding output $\mathbf{y}_t$ is the next position of the agent after moving. We think of the inputs and the outputs as the senses and actions of the agent, except expressed in terms of observables. By using the model interactively in a simulator environment we can recreate the global behavior of the agent.

It should be noted that our use of variable names x, y, and u as well as the terms *input* and *output* are different than their use in dynamic systems literature (where input often refers to input into the environment as opposed to the agent). We have chosen the notation and terminology to be consistent with previous papers on IOHMMs.

5.1 Output Distributions

We model output distributions as mixtures over learned low-level actions. Mixture distributions just marginalize a distribution over a discrete variable, in this case a_t, the low-level action the agent is performing at time t.

$$P(\mathbf{y}_t|x_t = i, \mathbf{u}_t) = \sum_j P(\mathbf{y}_t|a_t = j, \mathbf{u}_t)P(a_t = j|x_t = i, \mathbf{u}_t) \tag{1}$$

In this equation, the first term in the summation is the mixing component (the low-level action), and the second term is a mixing weight, the probability of selecting a mixing component. In the first term, we assume that $\mathbf{y}_t$ is conditionally independent of x_t given a_t, assuming the low-level actions are already learned and are fixed and the same for every state. Because the mixing weights are conditioned on $\mathbf{u}_t$ as well as x_t, the low-level action of an agent at a given time step depends on not only on the state of the agent but also on the current input. For our application, input-conditional mixing weights are undesirable because they make it hard to interpret the meaning of a state–there is not a simple mapping between states and low-level actions. One way to remove the dependency on $\mathbf{u}_t$ is to assume that a_t is conditionaly independent of $\mathbf{u}_t$ given x_t:

$$P(\mathbf{y}_t|x_t = i, \mathbf{u}_t) = \sum_j P(\mathbf{y}_t|a_t = j, \mathbf{u}_t) m_{i,j} \qquad (2)$$

Here $m_{i,j}$ is a standard mixing weight $P(a_t = j|x_t = i)$; however, this model does not make use of information from $\mathbf{u}_t$ when determing which low-level action the agent is performing. The input vectors can be valuable in determining the mixing weights when typical sensor values are different in the different low-level actions. For example, in our foraging problem the only action which is performed when the agent is holding something is Move to Nest. Our approach is to use the model from Equation 2 when executing the behavior, but for the purposes of parameter estimation, model the output distributions as generating both $\mathbf{u}_t$ and $\mathbf{y}_t$ (*i.e.* represent the joint probability of $\mathbf{u}_t$ and $\mathbf{y}_t$). This allows us to represent information about $\mathbf{u}_t$ without introducing input-conditional mixing weights. More specifically we use

$$P(\mathbf{y}_t, \mathbf{u}_t|x_t = i) = \sum_j P(\mathbf{y}_t|a_t = j, \mathbf{u}_t) P(\mathbf{u}_t|a_t = j) m_{i,j} \qquad (3)$$

Here we have assumed both $\mathbf{y}_t$ and $\mathbf{u}_t$ are conditionally independent of x_t given a_t. This equation, as compared to Equation 2, introduces a new term into the mixing component, $P(\mathbf{u}_t|a_t = j)$, the prior probability of observing an input given that the agent is performing a particular action–something like a confidence in our estimate of $P(\mathbf{y}_t|a_t = j, \mathbf{u}_t)$. We refer to this term as the input prior term. Using this joint output distribution, the overall model is something like an HMM/IOHMM hybrid: the transition distributions are conditional while the output distributions are not.

With this model, assuming we have calculated $P(\mathbf{y}_t|a_t = j, \mathbf{u}_t)$ and $P(\mathbf{u}_t|a_t = j)$ for each time step and low-level action—a task that only needs to be done once—only the mixing weights need to be re-estimated during EM training. In this way our variation of IOHMMs is less discriminant than standard models as the actual input-output mappings are fixed. The mixing weights are estimated with a variation of the standard formulas for mixing weights in a HMM where our summation term over low level actions replaces the standard summation term [8]. Once estimated, the mixing weights provide a soft mapping between the states of the model and the learned actions. The mapping is not necessarily one-to-one, admitting models with several states corresponding to the same low level action and with states that correspond to a mixture of multiple low-level actions.

5.2 Transition Distributions

We must also represent our transition distributions. This is done in a variety of ways in the literature, including neural networks. In our experiments we avoid some complexity and maintain human understandability by conditioning the transition distributions on a set of binary sensory features calculated from $\mathbf{u}_t$, just as we have done in our simulated domain. We can then represent the transition distributions as look up tables, using standard formulas for re-estimation [4].

We experimented with ranking our binary features. We then took as our input the most important triggered feature. For example, using the ranking in Figure 3 for our foraging problem, if the agent has bumped into something, the input value is 1 regardless of the state of the other features. This ranking method greatly reduces the number of parameters and constrains the model to resemble the sorts of models typically created by hand where only a single binary feature triggers a transition. Unfortunately, choosing a feature ranking requires domain knowledge. We also experimented with a standard binary encoding, allowing for a different transition rule for every possible combination of binary feature values.

When estimating probabilities in discrete probability distributions represented as look up tables, it is standard to add a small constant to the counts for each entry to allow some probability mass everywhere. This is sometimes called a Laplace correction and corresponds to placing a uniform Dirichlet prior on the probabilities. In our case, it is useful to add a small number to only the entries in the table corresponding to self-transitions. In other words, given little or no evidence, an agent should remain in its current state. This prevents, for example, the model from learning an extraneous transition for when the agent is at the nest in the Move to Target state, a situation that never happens.

6. LEARNING LOW-LEVEL ACTIONS

To estimate the mixing weights during EM from unlabeled data we need to estimate values for $P(\mathbf{y}_t|a_t, \mathbf{u}_t)$ for each time step and low-level action. To execute the resulting model, we also need to be able to sample from this distribution. Our approach to this problem is to learn a controller for each low-level action from labeled data. When combined with a simulation environment, the controller allows us to sample from $P(\mathbf{y}_t|a_t, \mathbf{u}_t)$ and, using sample-based approximation techniques, estimate values for unlabled data.

¿From labeled data, we can calculate sensory features and corresponding motor commands for each low-level action (example inputs and outputs for each controller). We solve the function approximation problem by using a modified version of k-nearest neighbor (KNN), randomly choosing from the k nearest neighbors with the probability of each weighted using a kernel function by the point's distance from the query point. In terms of senses and actions, we find the k vectors of sensory features closest to the input vector and randomly chose between the corresponding actions with probability proportional to the sensory feature vector distances. By randomly choosing in this way we can model random actions like the Explore or Loiter actions.

Given the controller and a simulation environment, we can sample from $P(\mathbf{y}_t|a_t, \mathbf{u}_t)$ (*i.e.* simulate a single time step of the agent's motion). From this we can also evaluate the distribution on unlabeled data using sample-based approximation techniques. Specifically we estimate the probability values using kernel density estimation [12]. For a particular

$\mathbf{u}_t$, we sample from $P(\mathbf{y}_t|a_t, \mathbf{u}_t)$ and take the average of a kernel function applied to $\mathbf{y}_t$ and each sample. In practical terms, for each time step and action we run a simulation many times to produce distributions of points predicting the next position of the agent. We then compare these point distributions to the actual next position to decide which action the agent is performing. This method assumes nothing concerning the representation of the low-level actions, so long as they can be executed in simulation, and fully captures interactions with the environment to the extent that the simulator can recreate them.

For the model from Equation 3, we also learn a model for the input prior $P(\mathbf{u}_t|a_t = j)$ from labeled input-output pairs. We are not interested in sampling from this distribution as our model does not generate $\mathbf{u}_t$, but we need to be able to estimate values for it in order to calculate the mixing weights during EM. In our experiments, we again reduce $\mathbf{u}_t$ to a set of binary features, using the same binary features and encoding as we did for the transition distributions.

7. RESULTS

7.1 Low-Level Action Learning

We used k-nearest neighbors on the agent's sense-action pairs to learn the low level controllers. To derive the sense-action pairs we ran the labeled data set back through the TeamBots simulation environment, and recorded the outputs of perceptual schemas corresponding to salient features in the environment. For the sensory features we used distance and angle to the closest obstacle, closest target, closest agent, and the base as well as whether or not the agent is currently carrying a target. For the motor outputs we calculated the distance and angle the agent moved at each time step (*i.e.* approximating the first order derivative by taking the difference between the position at t+1 and t-1) as well as whether the agent was carrying an object or not. We then tested the learned controllers in simulation using the original forage switching behavior. Using the switching behavior, our learned low-level controllers were able to recreate the original foraging behavior even on foraging arenas where the position of food items are moved from their original position as well as differently sized arenas with more attractors and more obstacles.

Even though the agents were able to recreate the original foraging behavior by successfully gathering all the targets, there were still flaws. The agents would frequently miss targets while moving towards them in the Move to Target controller, especially targets in in corners. This slowed down the foraging some, but because the agents would then continue exploring until they succesfully picked up a target, it didn't have a large effect on the system behavior. Another class of flaws includes our agent sometimes becoming stuck in corners with other groups of agents and they bump into each other. We believe that the second class of flaws were probably caused by the derived perceptual features only taking into account the closest obstacle.

We ran the detection method on the unlabeled data, using 9000 frames of data per track. To simulate motion for each of the actions we re-used the TeamBots simulation environment. Table 1 shows the confusion matrix for detection using only $P(\mathbf{y}_t|a_t, \mathbf{u}_t)$. Table 2 shows the confusion matrix for detection also using the input prior term, $P(\mathbf{u}_t|a_t)$. As seen in Table 3, the input prior term greatly increases detection accuracy for the Move to Target and Move to Base low level actions. These two actions have distinctly different characteristic inputs. In particular, Move to Target is only performed when the agent sees a target and is not holding a target, while Move to Base is performed when the agent is either holding a target or does not see a target. This also suggests why detecting these actions without the input prior term does so poorly. Given an input unlike any previously observed inputs, it is hard to predict what a learned controller will produce, and it may very well by chance produce an output similar to that of another controller.

7.2 IOHMM Learning

We trained 50 IOHMMs using EM on the same unlabeled data, for both ranked and unranked input encodings in the transition distributions, without using the input prior term in the output distributions. Of these models, only 1 ranked input (2%) and no unranked input models were able to recover the structure of the behavior in that they correctly learned the one-to-one correspondence between states and low-level actions. The one ranked input model could not recreate the behavior in simulation when combined with the learned low-level controllers, because the mixing weights in its Move to Target and Move to Base states were too noisy. The unsuccesful models tended to have two Explore or two Loiter states, presumably because of the proportionaly large amount of Explore and Loiter data.

With the input prior term, results were better. We trained 150 models with EM for both ranked and unranked input types using the input prior term, and 40 ranked and 32 unranked input models learned the correct structure. The likelihood scores for the models that learned the correct structure, as calculated using the standard method [4], were all higher than the scores for the models that did not. Their likelihood scores were also greater than the original model's. Within these models, there were several clusters of models whose likelihood scores were identical within range of our convergence criteria. We tested models from each of these clusters. Figure 3 shows models from the clusters with the highest likelihood scores for both ranked and unranked types. 14 ranked and 21 unranked models were within this maximum likelihood cluster.

All of the models trained with input prior terms that learned the correct structure were also able to recreate the forage behavior in simulation when combined with the learned controllers. However, they also all had flaws that affected the behavior to varying degrees. The models usually had some small nonzero mixing weights that caused states to sometimes flicker to a different low-level action. This would cause the agent to miss targets more when the flicker was in Move to Target. In some models agents would also drop the target when bumped into. This seems to be caused by detection inaccuracies when the agent is bumped into things. Another flaw we found is that many models did not know what to do when they missed a target or another agent reached a target before them. Some would even remain stuck in Move to Target in this situation, in the worst case then becoming stuck against a wall. Finally, some models would incorrectly transition from Move to Nest to Loiter when not carrying a

	Loiter	Explore	Move to Target	Move to Base
Loiter	84477	0	1	0
Explore	420	14355	34	90
Move to Target	0	60	333	87
Move to Base	1192	10	3744	3197

Table 1: Confusion matrix for detecting the low-level actions, without using an input prior term

	Loiter	Explore	Move to Target	Move to Base
Loiter	84478	0	0	0
Explore	0	14802	2	95
Move to Target	0	6	474	0
Move to Base	55	0	0	8088

Table 2: Confusion matrix for detecting low-level actions, using the input prior and an unranked encoding

target, sometimes transitioning early, sometimes late.

Subjectively the models with the maximum likelihood scores seemed to recreate the behavior the best in simulation. These models had only a small amount of flicker in the Explore state that did not have a large visible effect on the behavior. The ranked but not the binary models would drop the target when bumped. The binary but not the ranked models would stay in Move to Target on a miss, but return to Explore as soon as they bumped into something. The binary model also only learned a random transition from Move to Target to Loiter when returning without a target, while the ranked model did learn to transition when reaching the nest, but would also sometimes transition early.

8. EXTENDING EM WITH DISCRETE OPTIMIZATION

Although training with EM was succesful in that it could learn the correct model, the rate for doing so was low. One could simply run many more EM trials on randomly initialized models. Most of the computational effort lies in detecting low-level actions (done only once): even with over 100000 data points, EM only took an average of 2-5 minutes to complete. However, we expect more complicated models will have even more local minima; it is desirable to have a technique for combining information from multiple EM runs, so that we can avoid learning the same incorrect models over and over. Our approach is to use discrete optmization to learn the mapping between states and actions. The intuition behind is that the local minima found by EM tended to be only 1 or 2 discrete steps away from the correct model. For a model with two Explore states, for example, only one of the states needs to be switched.

First, we train a model with EM. We then extract a discrete mapping by setting the largest mixing weight in each state to 1 and all others to 0. To calculate the likelihood of this discrete mapping we randomize the transition distributions again and rerun EM with the mixing weights fixed. On successive iterations we randomly change the discrete mapping by changing the low-level action of a randomly selected state to a different, random action. We then evaluate the change by randomizing the transition distributions again and rerunning EM, keeping the mixing weights fixed. If the resulting model has a higher likelihood we accept the new mapping, else we revert to the last discrete mapping. Essentially we are performing randomized hill climbing over the space of state-action mappings allowing us to move across local optima in the original distribution space. Finally, after the discrete optimizaiton has completed, we perform one more final EM run, again randomizing the transition distributions but this time also allowing the mixing weights to change and replacing the 1s and 0s in the mixing weights with numbers close to but not equal to 1 or 0. The last step allows our model to use nonzero mixing weights to represent noise in the detection of low-level actions. With the mixing weights fixed and 1 or 0, models are forced to instead represent this noise through extraneous transitions.

8.1 Results with Discrete Optimization

On top of 50 of the 150 trials using the input prior term, we ran 10 steps of our proposed discrete optimization procedure. With the discrete optimization, all but 1 of the ranked and 3 of the unranked models found the correct structure. All but 3 of the ranked and 6 of the unranked models also had likelihood scores that fell in the maximum likelihood cluster of scores. Results are summarized in Table 4.

Because the discrete optimization runs on top of an initial, standard EM run, it doesn't seem fair to compare it to single runs of EM. Instead, we compare it to the expected results for running x repeated iterations of EM and keeping the best model, where x is chosen such that the expected training time is the same as the average training time for the discrete optimization method. As seen in Table 4, EM with discrete optimization gives signficantly better results than the expected results for running EM multiple times. Part of the reason why is that, despite the fact that we ran 10 steps of discrete optimization, EM with discrete optimization was still only 3-4 times slower than a single run of standard EM. This seemed to be because the single EM runs would sometimes take many iteartions to converge, while the discrete optimization steps, because the mixing weights were fixed, converged consistently and quickly. Even the final EM run when the mixing weights were allowed to change tended to converge faster than the standard EM runs, because of the strong initialization of the mixing weights.

Our original convergence criteria for EM was to stop when the average likelihood score per data track changed by less than .1. To make sure this was not too strict of a convergence criteria—which could artificially increase the train-

Action	Percent of Data	Accuracy w/o Input Prior	Accuracy w/ Input Prior
Loiter	78.22	100.0	100.0
Explore	13.80	96.35	99.35
Move to Target	0.44	69.38	98.75
Move to Base	7.54	39.26	99.32

Table 3: Percents of the total data and detection percent accuracies for the low-level actions

ing times for EM—we tried running 50 additional trials for ranked and unranked input with the discrete optimization, this time stopping when the change was less than 1. Doing this did speed up EM some compared to EM with discrete optimization, but we found that on these trials very few single EM runs were successful. As seen in Table 5, repeated EM trials compared even worse to EM with discrete optimization using this looser convergence criteria. In fact, even arbitrarily placing the training times from Table 5 into Table 4, EM with discrete optimization still gives significantly better results than repeated EM runs, suggesting these results are not a by product of a particular convergence criteria.

9. CONCLUSIONS AND FUTURE WORK

We have presented a method for learning an executable model of behavior from observations. Our method consists of learning an Input/Output Hidden Markov Model (IOHMM) where the output distributions are mixtures of learned low-level actions and the transition distributions are conditioned on a set of binary sensory features. We tested our method on simulated data generated from a hand-constructed model inspired by insect behavior. The learned models recreated the behavior of the original model and exhibited the same basic structure. We also introduced a novel extension to EM for our particular type of IOHMM that compares favorably to standard EM runs.

We hope to next apply these techniques to actual insect tracking data. We would also like to perform more testing with simulated models, in particular with models that contain repeated low-level actions with noisier tracking and harder detection.

10. ACKNOWLEDGMENTS

Thank you to Sridhar Mahadevan, Zia Khan, and Adam Feldman for helpful discussions. This work was supported in part by NSF.

11. REFERENCES

[1] R. C. Arkin and T. R. Balch. Aura: Principles and practice in review. *Journal of Experimental and Theoretical Artificial Intelligence(JETAI)*, 9(2-3):175–188, April 1997.

[2] Y. Bengio. Markovian models for sequential data. *Neural Computing Surveys*, 2:129–162, 1999.

[3] Y. Bengio and P. Frasconi. Diffusion of context and credit information in markovian models. *Journal of Artificial Intelligence Research*, 3:249–270, 1995.

[4] Y. Bengio and P. Frasconi. Input-output HMM's for sequence processing. *IEEE Transactions on Neural Networks*, 7(5):1231–1249, September 1996.

[5] Y. Bengio, V.-P. Lauzon, and R. Ducharme. Experiments on the application of iohmms to model financial returns series. *IEEE Transaction on Neural Networks*, 12:113–123, January 2001.

[6] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, March 1994.

[7] D. C. Bentivegna, A. Ude, C. G. Atkeson, and G. Cheng. Humanoid robot learning and game playing using pc-based vision. In *2002 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS02)*, volume 3, pages 2449–2454, 2002.

[8] J. Bilmes. A gentle tutorial on the em algorithm and its application to parameter estimation for gaussian mixture and hidden markov models, 1997.

[9] F. Delmotte and M. Egerstedt. Reconstruction of low-complexity control programs from data. In *43rd IEEE Conference on Decision and Control*, volume 2, pages 1460–1465, December 2004.

[10] P. Frasconi and Y. Bengio. An em approach to grammatical inference: Input/output hmms. In *12th IAPR International Conference on Pattern Recognition*, volume 2, pages 289–294, 1994.

[11] Y. Gatsoulis, G. Maistros, Y. Marom, and G. Hayes. Learning to forage through imitation. In *Second IASTED International Conference on Artificial Intelligence and Applications (AIA2002)*, pages 485–491, September 2002.

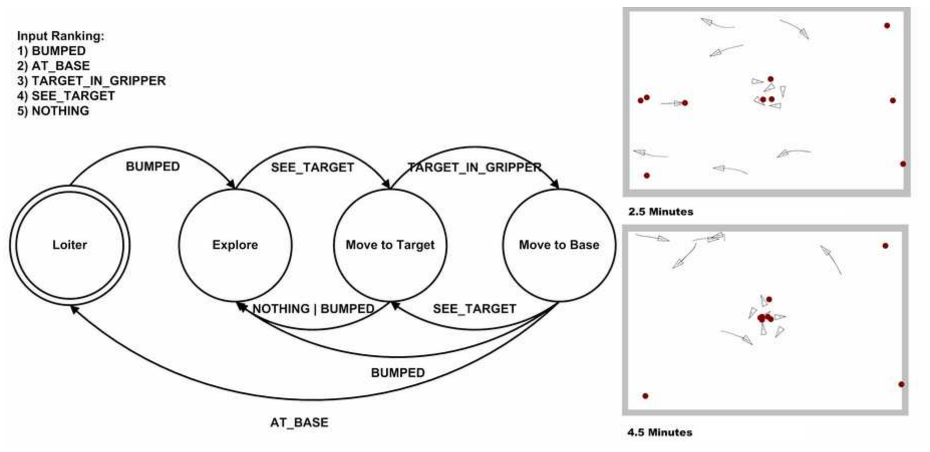
[12] A. Gray and A. Moore. Rapid evaluation of multiple density models. In *Artificial Iintelligence and Statistics*, 2003.

[13] S. Marcel, O. Bernier, J.-E. Viallet, and D. Collobert. Hand gesture recognition using input-output hidden markov models. In *4th IEEE International Conference on Automatic Face and Gesture Recognition*, pages 456–461, March 2000.

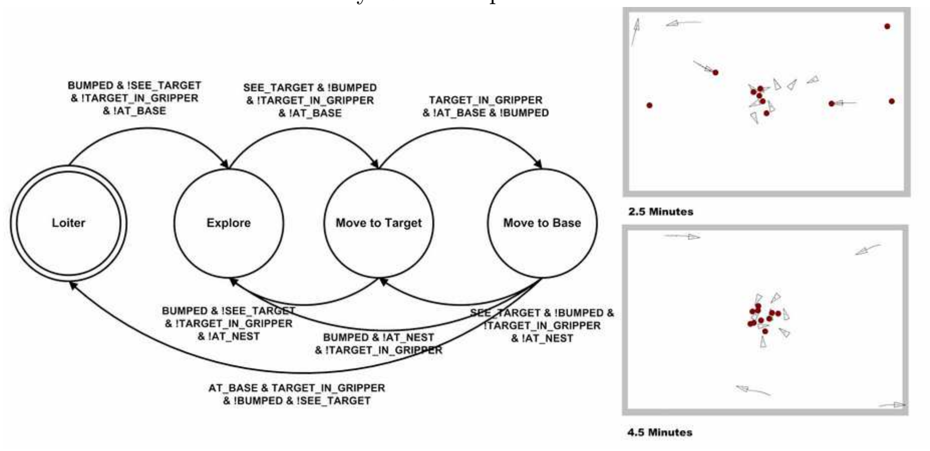
[14] N. Mukherjee. Speaker recognition using least squares iohmms. In *2002 IEEE Workshop on Multimedia Signal Processing*, pages 276–279, December 2002.

[15] L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77:257–286, February 1989.

[16] D. Sumpter and S. Pratt. A modelling framework for understanding social insect foraging. *Behavioral Ecology and Sociobiology*, 53(3):131–144, February 2003.



Most Likely Ranked Input Learned Model


BUMPED & !SEE_TARGET & !TARGET_IN_GRIPPER & !AT_BASE
SEE_TARGET & !BUMPED & !TARGET_IN_GRIPPER & !AT_BASE
TARGET_IN_GRIPPER & !AT_BASE & !BUMPED
Loiter
Explore
Move to Target
Move to Base
BUMPED & !SEE_TARGET & !TARGET_IN_GRIPPER & !AT_NEST
BUMPED & !AT_NEST & !TARGET_IN_GRIPPER
SEE_TARGET & !BUMPED & !TARGET_IN_GRIPPER & !AT_NEST
AT_BASE & TARGET_IN_GRIPPER & !BUMPED & !SEE_TARGET
2.5 Minutes
4.5 Minutes

Most Likely Unranked Input Learned Model

Figure 3: Diagrams and screenshots of our learned models. The learned model diagrams show the most likely transitions for every possible input, excluding self transitions.

Model / Algorithm	Learn Structure (%)	Learn Max Likelihood (%)	$\frac{\text{Training Time}}{\text{EM Training Time}}$
Ranked IOHMM / EM	26.67	9.33	1.00
Ranked IOHMM / EM+D	98.00	94.00	3.21
Ranked IOHMM / EM*3.21	63.06	29.00	3.21
Unranked IOHMM / EM	21.33	14.00	1.00
Unranked IOHMM / EM+D	94.00	88.00	3.83
Unranked IOHMM / EM*3.83	60.10	43.88	3.83

Table 4: Percentage of trials finding the correct structure, percentage of trials finding a model with a likelihood score about equal to the maximum likelihood score for all trials, and training times as compared to a single run of EM. EM+D stands for EM with discrete optimization. EM*x is the expected results for running x iterations of EM and keeping the best model.

Model / Algorithm	Learn Structure (%)	Learn Max Likelihood (%)	$\frac{\text{Training Time}}{\text{EM Training Time}}$
Ranked IOHMM / EM	6.00	6.00	1.00
Ranked IOHMM / EM+D	92.00	78.00	4.56
Ranked IOHMM / EM*4.56	24.58	24.58	4.56
Unranked IOHMM / EM	2.00	2.00	1.00
Unranked IOHMM / EM+D	86.00	76.00	5.10
Unranked IOHMM / EM*5.10	9.79	9.79	5.10

Table 5: The same as Table 4 but using a looser convergence criteria when running EM