Unsupervised Learning of Probabilistic Models for Robot Navigation*

Sven Koenig   Reid G. Simmons
School of Computer Science, Carnegie Mellon University
Pittsburgh, PA 15213-3891

Abstract

Navigation methods for office delivery robots need to take various sources of uncertainty into account in order to get robust performance. In previous work, we developed a reliable navigation technique that uses partially observable Markov models to represent metric, actuator, and sensor uncertainties. This paper describes an algorithm that adjusts the probabilities of the initial Markov model by passively observing the robot's interactions with its environment. The learned probabilities more accurately reflect the actual uncertainties in the environment, which ultimately leads to improved navigation performance. The algorithm, an extension of the Baum-Welch algorithm, learns without a teacher and addresses the issues of limited memory and the cost of collecting training data. Empirical results show that the algorithm learns good Markov models with a small amount of training data.

1 Introduction

Navigation methods for office delivery robots need to take various sources of uncertainty into account in order to get robust performance. We have developed a navigation technique for the Xavier mobile robot (Figure 1) that explicitly represents uncertain metric information (e.g. "corridor X is between 2 and 9 meters long"), actuator (dead-reckoning) uncertainty, and sensor uncertainty [13]. The technique uses partially observable Markov decision process (POMDP) models to estimate the position of the robot in the form of probability distributions. Experience with the technique has shown it to produce reliable navigation. The results are better, however, when the probabilities of the POMDP more closely reflect the actual environment of the robot. In this paper, we describe a technique that learns more accurate probabilities, thus reducing uncertainty about the lengths of corridors and improving the accuracy of the actuator and sensor models. Our approach is fairly general, and is applicable to other robot navigation systems that use POMDPs (e.g., [9]).

We desire that the learning be *unsupervised* and *passive*. Unsupervised learning means that the robot gets no information from a teacher, such as where it really is or what it really observed. This is desirable because then the learning can be done autonomously. Passive learning means that the robot learns while it is performing other tasks – the learning algorithm does not need to control the robot at any time. This is desirable, since the robot does not need a separate training phase – learning takes place while the robot is performing its delivery tasks. Consequently, training data is obtained whenever the robot moves and it reflects the environment in which the robot has to perform.

Unsupervised and passive learning is difficult, however: there is no "ground truth" to compare against, and the robot cannot execute strategies that would likely maximize its information. For example, the robot's positional uncertainty prevents it from simply learning corridor lengths by first moving to the beginning of a corridor and then to its end while measuring the distance traveled, and the robot cannot reduce its uncertainty by asking a teacher or by executing localization actions. In addition, robot learning is hard because it must run within the CPU and memory constraints of the robot's computers, and must deal with the fact that collecting data is time consuming.

Our POMDP learning algorithm addresses all these concerns. It is an unsupervised, passive method based on the Baum-Welch algorithm [11], a simple expectation-maximization algorithm for learning POMDPs from observations. To enable the algorithm to run on-board the robot, we have extended the Baum-Welch algorithm to use a floating window of training data. This decreases its memory requirements, while producing comparable results to the traditional Baum-Welch algorithm and maintaining its efficiency (run-times of seconds to minutes). Since the algorithm merely updates the probabilities associated with the POMDP, rather than changing its actual structure, learning is transparent to all other components of the robot architecture and the learned POMDP can be used immediately

*Thanks to Lonnie Chrisman, Richard Goodwin, Geoffrey Gordon, Joseph O'Sullivan, and Gregory Whelan for helpful discussions. This research was supported in part by NASA under contract NAGW-1175, and by the Wright Laboratory and ARPA under grant number F33615-93-1-1330. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations or the United States government.

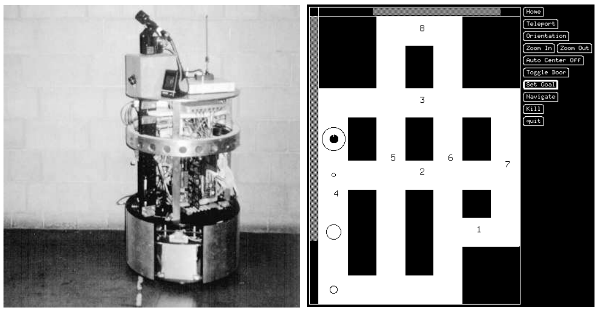

Figure 1: Xavier and its simulator

to improve the navigation performance of the robot. While all three kinds of uncertainty (metric uncertainty, noisy actuator models, and noisy sensor models) can be adapted simultaneously, if desired, any subset of the probabilities can be held constant in order to reduce the amount of training data needed. To further reduce the need for training data, the extended Baum-Welch algorithm can utilize additional constraints, such as that the left and right sensors are identical or that two corridors have the same (although unknown) length.

The next two sections describe the inputs needed by our learning algorithm – an execution trace and an initial POMDP describing the environment. The standard Baum-Welch algorithm is described in Section 4, and Section 5 describes our extensions to deal with limited memory and limited training data. Section 6 presents our empirical results, which demonstrate that our learning method acquires good models when being supplied with a small amount of training data.

2 Input: The Execution Trace

The learning algorithm tries to find the POMDP that best fits the data collected as the robot moves around. This data, called the *execution trace*, is generated automatically by the navigation algorithm. A trace begins with the initial probability distribution of the robot's position and orientation. As the robot moves, it issues a sequence of action and sensor reports. Let A be a set of actions and VS be a set of (virtual) sensors. An action report consists of the action $a \in A$ executed. Before each action report, the execution trace contains one report for each sensor $vs \in VS$. That report consists of a probability distribution r_{vs} over all features $f \in F(vs)$ that the sensor can detect. $r_{vs}(f)$ is the degree with which sensor vs believes that it has detected feature f.

For Xavier, we instantiate this framework as follows [13]: We discretize its location (with a resolution of 1 meter) and orientation (into the four compass directions). Consequently, we distinguish three different actions: turning right 90 degrees (r), turning left 90 degrees (l), and going forward one meter (f). Thus, every time Xavier travels a meter or turns, it adds an action report to its execution trace along with the associated sensor reports. Currently, three virtual sensors are modeled: a front sensor reports whether there is a wall directly in front of Xavier; a left sensor and a right sensor report walls and openings of various sizes (small, medium, and large) to the immediate left and right, respectively, of Xavier. Alternatively, a sensor can report the feature `dont-know` if it is uncertain about how to interpret the raw sensor data.

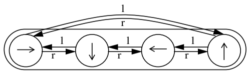

Figure 2: Four states representing one location

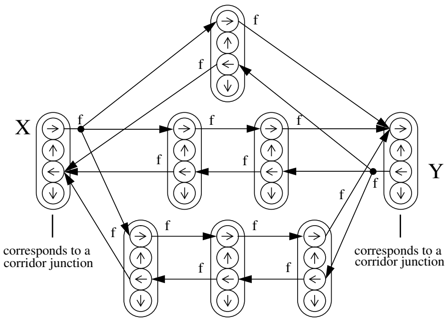

Figure 3: Corridor of length 2 to 4 meters

3 Input: The Markov Model

The initial POMDP is automatically compiled from actuator and sensor models and a topological map that is augmented with uncertain metric information [13]. The POMDP is specified as (1) a finite set of states S, (2) a set of actions A that can be executed in each state, (3) transition probabilities $p(s'|s,a)$ for all $s, s' \in S$ and $a \in A$ (the probability that the successor state is s' if the robot executes action a in state s), and (4) sensor probabilities $p_{vs}(f|s)$ for all $vs \in VS$, $f \in F(vs)$, and $s \in S$ (the probability that sensor vs reports feature f when the robot is in state s). The transition probabilities are obtained from the actuator models and uncertain metric information, and the sensor probabilities are obtained from the sensor models.

For Xavier, we instantiate this framework as follows: Each state encodes both the orientation and location of the robot. Right and left turn actions are defined for every state (Figure 2). These actions are nearly deterministic, but there is a small chance that the robot ends up in any of the three unintended orientations at the same location (not shown in the figure). Forward actions transition from location to location.[1] We model corridors (or, to be precise, the corridor parts between two adjacent junctions) as sets of parallel chains that share their first and last states (Figure 3). Each chain corresponds to one of the possible lengths for that stretch of corridor, reflecting both metric uncertainty and dead-reckoning error of the robot. From each junction, forward actions have probabilistic outcomes, indicating how likely the corridor is to be perceived of that length (initially, we assume a uniform distribution). Each forward transition after that is deterministic.

4 The Baum-Welch Algorithm

The Baum-Welch algorithm is a simple expectation-maximization algorithm for learning POMDPs from observations [11]. To describe it, we assume that at time $t = 1 \dots T$, just before executing action a_t ($t \neq T$), the robot is in state s_t and each sensor vs reports a probability distribution $r_{vs,t}$ over its features (T is the length of the execution trace). We call the aggregate of all sensor reports at time t the observation o_t and define $p(o_t|s) := \prod_{vs \in VS} \sum_{f \in F(vs)} [r_{vs,t}(f) p_{vs}(f|s)]$ for all $s \in S$.

Given a POMDP and an execution trace, the Baum-Welch algorithm estimates a POMDP that better fits the trace, in the sense that the probability $p(o_{1\dots T}|a_{1\dots T-1})$ with which the POMDP explains (or, synonymously, generates) the observations is increased. This bootstrapping process is then repeated with the same execution trace and the improved POMDP. The resulting hill-climbing process converges eventually to a POMDP which locally, but not necessarily globally, best fits the trace.

The Baum-Welch algorithm estimates an improved POMDP in three steps: First, it calculates the "scaling factors" $scale_t$, "alpha values" $\alpha_t(s)$, and "beta values" $\beta_t(s)$ for all $s \in S$ and $t = 1 \dots T$. Then, it uses these values to calculate the "gamma values" $\gamma_t(s, s')$ for all $t = 1 \dots T-1$ and $s, s' \in S$, and $\gamma_t(s)$ for all $t = 1 \dots T$ and $s \in S$. Finally, it uses the gamma values to determine the improved transition and sensor probabilities.

$$
\begin{aligned}
scale_t &:= p(o_t|o_{1\dots t-1}, a_{1\dots t-1}) \\
\alpha_t(s) &:= p(s_t = s|o_{1\dots t}, a_{1\dots t-1}) = \frac{p(o_{1\dots t}, s_t = s|a_{1\dots t-1})}{\prod_{t'=1\dots t} scale_{t'}} \\
\beta_t(s) &:= \frac{p(o_{t+1\dots T}|s_t = s, a_{t\dots T-1})}{p(o_{t\dots T}|o_{1\dots t-1}, a_{1\dots T-1})} = \frac{p(o_{t+1\dots T}|s_t = s, a_{t\dots T-1})}{\prod_{t'=t\dots T} scale_{t'}} \\
\gamma_t(s, s') &:= p(s_t = s, s_{t+1} = s'|o_{1\dots T}, a_{1\dots T-1}) \\
\gamma_t(s) &:= p(s_t = s|o_{1\dots T}, a_{1\dots T-1})
\end{aligned}
$$

The *alpha values* α_t are essentially what our navigation system uses to estimate the state of the robot. They can be calculated on the fly, because they are only conditioned on the part of the execution trace that is available at the current time. The *gamma values* γ_t, on the other hand, are more precise estimates of the same state distribution, because they also utilize information that became available after time t. For example, going forward three meters and seeing a wall ahead at time t provides strong evidence that at time $t-3$ the robot was three meters away from the end of the corridor. The *scaling factors* are used to prevent the numerators of the alpha and beta values from underflowing. They are also convenient for calculating how well the POMDP fits the execution trace, since $p(o_{1\dots T}|a_{1\dots T-1}) = \prod_{t=1}^{T} scale_t$.

The Baum-Welch algorithm uses the following dynamic programming approach ("forward-backward algorithm"), that applies Bayes' rule repeatedly, to calculate the scaling factors, alpha values, and beta values efficiently [3]:

A1. Set $scale_1 := \sum_{s \in S} [p(o_1|s) P(s_1 = s)]$.
A2. Set $\alpha_1(s) := p(o_1|s) P(s_1 = s) / scale_1$ for all $s \in S$.
A3. For $t := 1$ to $T-1$ *("forward propagation")*
  (a) Let $temp_t(s) = \sum_{s' \in S} [p(s|s', a_t) \alpha_t(s')]$ for all $s \in S$.
  (b) Set $scale_{t+1} := \sum_{s \in S} [p(o_{t+1}|s) temp_t(s)]$.
  (c) Set $\alpha_{t+1}(s) := p(o_{t+1}|s) temp_t(s) / scale_{t+1}$ for all $s \in S$.
A4. Set $\beta_T(s) := 1/scale_T$ for all $s \in S$.
A5. For $t := T-1$ downto 1 *("backward propagation")*
  (a) Set $\beta_t(s) := \sum_{s' \in S} [p(s'|s, a_t) p(o_{t+1}|s') \beta_{t+1}(s')] / scale_t$ for all $s \in S$.

Next, the Baum-Welch algorithm calculates the gamma values as follows:

A6. Set $\gamma_t(s, s') := \alpha_t(s) p(s'|s, a_t) p(o_{t+1}|s') \beta_{t+1}(s')$ for all $t = 1 \dots T-1$ and $s, s' \in S$.
A7. Set $\gamma_t(s) := scale_t \alpha_t(s) \beta_t(s)$ for all $t = 1 \dots T$ and $s \in S$.

Finally, the algorithm uses the following frequency-counting re-estimation formulas to calculate the improved initial state probabilities, transition probabilities, and observation probabilities (the overlined symbols represent the probabilities that constitute the improved POMDP):

A8. Set $\bar{p}(s_1 = s) := \gamma_1(s)$.
A9. Set $\bar{p}(s'|s, a) := \sum_{t=1\dots T-1|a_t=a} \gamma_t(s, s') / \sum_{t=1\dots T-1|a_t=a} \gamma_t(s)$ for all $s, s' \in S$ and $a \in A$.
A10. Set $\bar{p}(o|s) := \sum_{t=1\dots T|o_t=o} \gamma_t(s) / \sum_{t=1\dots T} \gamma_t(s)$ for all $s \in S$ and all observations o.

When implementing the Baum-Welch algorithm as part of our learning method, we made the following design decisions:

Updating the Sensor Probabilities: We are not really interested in updating the observation probabilities $p(o|s)$

[1] To be precise: forward actions can only be executed in states that do not face walls, and a successful execution of a forward action conveys information. This is not a problem for the learning algorithms discussed in this paper, but would complicate our notation for their derivation. We therefore ask the reader to assume that forward actions are defined for all states (resulting in self-transitions for states that face walls) and that the information normally provided by them is instead provided by a special virtual sensor that is part of *VS*.

as is done by the Baum-Welch algorithm – we want to update the sensor probabilities $p_{vs}(f|s)$ instead. Our learning method therefore uses the following re-estimation formula:

A10'. Set $\bar{p}_{vs}(f|s) := \sum_{t=1...T}[r_{vs,t}(f)\gamma_t(s)] / \sum_{t=1...T}\gamma_t(s)$ for all $vs \in VS, f \in F(vs)$, and $s \in S$.

Influence of the Initial POMDP: Because the Baum-Welch algorithm converges to a local optimum, the final POMDP can, in theory, depend on the initial POMDP. We found that the Baum-Welch algorithm is very robust towards variations of the initial probabilities. Our learning method therefore applies the Baum-Welch algorithm only to the initially given POMDP (after having added a small amount of noise).

Extreme Transition Probabilities: The re-estimation formulas do not change transition probabilities that are zero or one. This means that transitions that were impossible (or certain) in the original model remain so in the updated model. Our learning method uses this property to save computation time by calculating $\gamma_t(s, s')$ and $\bar{p}(s'|s, a)$ only if $p(s'|s, a) \neq 0$.

5 Problems and Solutions

Despite its theoretical elegance, the Baum-Welch algorithm has two deficiencies that make it impractical for real robots: its memory and training data requirements. We address these problems by extending the Baum-Welch algorithm.

5.1 Memory Requirements

Standard implementations of the Baum-Welch algorithm need arrays of floating point numbers whose sizes are on the order of the product of the number of states and the length of the execution trace. Even our smallest POMDPs have thousands of states, and we need execution traces with hundreds of action executions to get sufficient data. Since many other processes are run on the same on-board computer, the memory requirements of a learning method should be rather small and relatively constant. The extended Baum-Welch algorithm still calculates the alpha values precisely, since they can be calculated incrementally with only two arrays the size of the number of states: one for the current alpha values and another one for the alpha values of the next time step. The beta values, however, are approximated using a sliding "time window" on the execution trace. The gamma values are then calculated using the alpha and beta values, as before. Approximating the gamma values this way is reasonable because floors of buildings are usually constructed in a way that allows one to obtain sufficient clues about the current location from past experience and the local environment only. Otherwise not only robots, but also people, would easily get confused.

The values $\beta_t(s)$ are approximated with $p(o_{t+1...t'}|s_t = s, a_{t...t'-1}) / \prod_{t''=t...t'} scale_{t''}$. This results in $\gamma_t(s, s')$ being approximated with $p(s_t = s, s_{t+1} = s'|o_{1...t'}, a_{1...t'-1})$ and $\gamma_t(s)$ being approximated with $p(s_t = s|o_{1...t'}, a_{1...t'-1})$. The extended Baum-Welch algorithm guarantees the "effective look-ahead" $t' - t$ of the beta values to be at least la_{min}, where the "minimal look-ahead" $la_{min} \geq 0$ is chosen so that the gamma values are approximated closely. For the calculations, the algorithm uses a time window of size $x \geq la_{min}+2$, where x is independent of the length of the execution trace. A time window that starts at t_{start} can store the scaling, alpha, and beta values for all $t = t_{start} \dots t_{start} + x - 1$. It only needs arrays of floating point numbers whose sizes are on the order of x times the number of states and thus its memory requirements do no longer depend on the length of the execution trace.

The window-based Baum-Welch algorithm operates as follows (for simplicity, we do not show how the initial state distribution is updated or $\gamma_T(s)$ is calculated):

B1. Set $\bar{p}'(s'|s, a) := 0$ for all $s, s' \in S$ and $a \in A$.

B2. Set $\bar{p}'_{vs}(f|s) := 0$ for all $vs \in VS, f \in F(vs)$, and $s \in S$.

B3. Initialize $scale_1$ using A1 and α_1 using A2.

B4. Set $t_{start} := 1$ and $t_{end} := x$.

B5. While $t_{start} < T$:
   - (a) If $t_{end} > T$, then set $t_{end} = T$.
   - (b) Calculate $scale_t$ and α_t for $t = t_{start} \dots t_{end}$ (if they have not been calculated already), working forward from $scale_{t_{start}}$ and $\alpha_{t_{start}}$ using A3(a), A3(b), and A3(c).
   - (c) Approximate the beta values β_t for $t = t_{start} \dots t_{end}$, initializing $\beta_{t_{end}}(s) := 1/scale_{t_{end}}$ for all $s \in S$ and working backward using A5(a). (Previously calculated beta values cannot be re-used.)
   - (d) If $t_{end} = T$, then set $t_{newstart} := T$ else set $t_{newstart} := t_{end} - la_{min}$.
   - (e) For all $t = t_{start} \dots t_{newstart} - 1$:
     - i. Calculate $\gamma_t(s, s')$ for all $s, s' \in S$, using α_t and the approximation of β_{t+1} in A6.
     - ii. Calculate $\gamma_t(s)$ for all $s \in S$, using α_t and the approximation of β_t in A7.
     - iii. Set $\bar{p}'(s'|s, a_t) := \bar{p}'(s'|s, a_t) + \gamma_t(s, s')$ for all $s, s' \in S$ (these values will be normalized in B6).
     - iv. Set $\bar{p}'_{vs}(f|s) := \bar{p}'_{vs}(f|s) + r_{vs,t}(f)\gamma_t(s)$ for all $vs \in VS, f \in F(vs)$, and $s \in S$ (these values will be normalized in B7).
   - (f) Forget all $scale_t$ and α_t for $t = t_{start} \dots t_{newstart} - 1$, and all β_t for $t = t_{start} \dots t_{end}$.
   - (g) Set $t_{start} := t_{newstart}$ and $t_{end} := t_{newstart} + x - 1$ (i.e. move the time window).

B6. Set $\bar{p}(s'|s, a) := \bar{p}'(s'|s, a) / \sum_{s' \in S} \bar{p}'(s'|s, a)$ for all $s, s' \in S$ and $a \in A$ (i.e. normalize the transition probabilities).

B7. Set $\bar{p}_{vs}(f|s) := \bar{p}'_{vs}(f|s) / \sum_{f \in F(vs)} \bar{p}'_{vs}(f|s)$ for all $vs \in VS$, $f \in F(vs)$, and $s \in S$ (i.e. normalize the sensor probabilities).

With the extended Baum-Welch algorithm, there is a tradeoff between memory requirements, run time, and precision of the improved POMDP. Run-time overhead is incurred mostly for calculating the beta values repeatedly. While the traditional Baum-Welch algorithm calculates every state distribution β_t once, the extended Baum-Welch algorithm calculates it on average $x/(x - la_{min} - 1)$ times for long execution traces. Its precision increases with its effective look-ahead, which is, on average, $(la_{min} + x)/2$ for long execution traces. Thus, the precision can be increased by increasing the minimal look-ahead la_{min} or the window size x. Increasing the minimal look-ahead produces a small amount of run-time overhead, but leaves the memory requirements unchanged; increasing the window size decreases the overhead, but increases the amount of memory needed. We therefore suggest to make the window size as large as possible and to set the minimal look-ahead based on the average lengths of the corridors, because the most useful sensor reports are obtained when the robot traverses junctions.

5.2 Training Data Requirements

The traditional Baum-Welch algorithm requires a large amount of training data: as the degrees of freedom (settable parameters) increase, so does the need for training data, to decrease the likelihood of overfitting the model. Given the relatively slow speed at which mobile robots can move, we want our learning method to learn good POMDPs with as few corridor traversals as possible. Thus, we use several methods to decrease the number of model parameters that must be learned. The initial structure of the POMDP already reduces the model parameters considerably by disallowing transitions that are clearly impossible (such as teleporting to distant locations). We employ two additional techniques to reduce the degrees of freedom further:

Leaving Probabilities Unchanged: Our learning method does not adjust parameters that we believe to be approximately correct. Actuator and sensor models, for example, are often similar in different environments and consequently need only be learned once.

Imposing Equality Constraints: Our learning method constrains some probabilities to be identical. This has the advantage that the Baum-Welch algorithm can now update a probability using all the information that applies to any probability in its class. Consider the following examples:

- **Metric Uncertainty:** We constrain the transition probabilities of the forward actions for "junction" states that lead into the same corridor to be identical (e.g., states X and Y in Figure 3). This forces the length estimates for a corridor to be the same in both directions. In general, we group all junction states that are known to lead into equally long corridor segments (such as those intersected orthogonally by two parallel corridors – see Figure 4).
- **Actuator Models:** We assume that the models for the left and right turn actions are the same for all states. We further constrain the left and right turn probabilities to be symmetrical.
- **Sensor Models:** Instead of learning separate sensor models for each state, we learn them for classes of states (wall, "near wall", junction opening, open-door, closed-door). These classes reflect our prior knowledge about how the sensors are supposed to operate – they are currently predefined and not learned. For example, all states that have a wall on their left are construed to have the same left sensor model. Our learning method also assumes that the left and right sensors behave identically, so their models are constrained to have the same probabilities.

These techniques enable the Baum-Welch algorithm to operate with a smaller amount of training data. However, frequency-based estimates are not very reliable if the sample size is small. To understand why, consider the following analogy: If a fair coin were flipped once and came up heads, the frequency-based estimate would set $p(heads) = 1$. If this model were used to predict future coin flips, one would be very surprised if the coin came up tails next time – this would be inconsistent with the learned model. To avoid this problem, we change the re-estimation formulas A9 and A10' to use Bayes' rule (Dirichlet distributions) instead of frequencies (both methods produce asymptotically the same results for long execution traces). For example, the re-estimation formula for the transition probabilities becomes (probability classes are not shown):

A9'. Set $\bar{p}(s'|s,a) := (k \times p(s'|s,a) + \sum_{t=1 \ldots T-1 | a_t = a} \gamma_t(s,s'))/(k + \sum_{t=1 \ldots T-1 | a_t = a} \gamma_t(s))$ for all $s, s' \in S$ and $a \in A$.

In this formula, $p(s'|s,a)$ are the transition probabilities before learning and $k > 0$ is a constant whose magnitude indicates the confidence that one has in the initial probabilities (we use $k = 1$). Note that the original re-estimation formula A9 is a special case of A9' where $k = 0$. Similarly, leaving the transition probabilities unchanged is a special case of A9' where k is large.

6 Experimental Evaluation

The world cannot be expected to satisfy the independence properties that POMDPs and the Baum-Welch algorithm assume. In the end, one has to test empirically whether they are satisfied well enough for our learning method to yield plausible models. We therefore performed several experiments with the Xavier simulator, a highly realistic simulation of Xavier. Figure 1 shows a snapshot of the simulator operating on the prototypical corridor environment that we used in our experiments. (The circles show the probability mass with which the robot believes itself to be in a certain junction or corridor.) The initial

left and right sensors	original human estimates			learned probabilities		
	reality			reality		
sensor report	wall	near-wall	opening	wall	near-wall	opening
wall	0.90	0.54	0.05	0.784432	0.669705	0.010528
small-opening	0.03	0.01	0.20	0.000010	0.000010	0.001762
medium-opening	0.02	0.15	0.40	0.000649	0.000010	0.146208
large-opening	0.01	0.25	0.30	0.001779	0.000010	0.291644
dont-know	0.05	0.05	0.05	0.213130	0.330265	0.549859

front sensor	original human estimates			learned probabilities		
	reality			reality		
sensor report	wall	near-wall	opening	wall	near-wall	opening
wall	0.98	0.50	0.02	0.034271	0.057838	0.000010
dont-know	0.02	0.50	0.98	0.965729	0.942162	0.999990

Table 1: Original and learned sensor probabilities

fit (closer to zero is better)

$1/T \times \ln p(o_{1\ldots T}|a_{1\ldots T-1})$

"how well a model explains an execution trace"

	actuator and sensor models	
metric model	original	learned
original	-3.188194	-1.917637
learned	-2.917535	-1.684340

entropy (closer to zero is better)

$1/(T \ln |S|) \times \sum_{t=1\ldots T} \sum_{s \in S} [\alpha_t(s) \ln(\alpha_t(s))]$

"how certain the robot is about its position"

	actuator and sensor models	
metric model	original	learned
original	-0.101616	-0.050821
learned	-0.049178	-0.044137

Table 2: Quality of the learned models

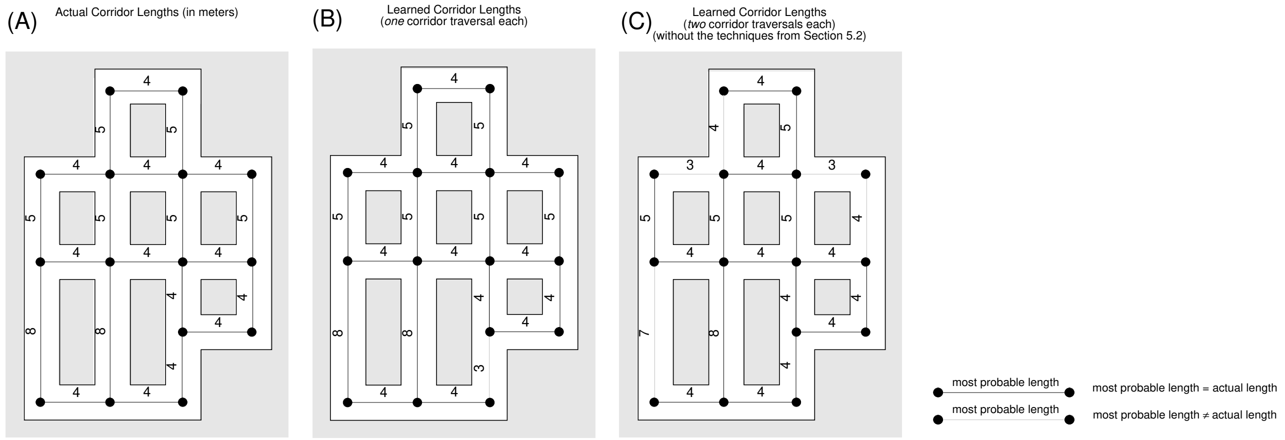

Figure 4: Actual and learned metric models

metric uncertainty of each corridor part was characterized by a uniform distribution over the lengths from 2 to 9 meters, yielding a POMDP with 2472 states. Unless stated otherwise, we used the extended Baum-Welch algorithm with a minimal look-ahead of 5 time steps and a window size of 20 time steps.

In the first experiment, we simultaneously learned the metric model, actuator models, and sensor models using an execution trace that consisted of ten smaller execution traces with various start locations of the robot. In total, every corridor of our environment was traversed five times. The purpose of this experiment was to learn good actuator and sensor models for use in the subsequent experiments.

Table 1 lists the learned sensor model together with the human estimates that the navigation system used originally (open and closed doors are omitted, since they do not appear in the test environment). Note that the sensors are reasonably accurate when they issue a report. The relatively high probability of `dont-know` observations is partly due to the virtual sensors being quite conservative: they don't report features until they have collected sufficient evidence. It is also partly due to the fact that the sensor and action reports are generated asynchronously, and the execution trace fills in a `dont-know` whenever a sensor has not issued a report in time.

The second experiment fixed the previously learned actuator and sensor models and learned the metric model only, this time using an execution trace that was generated by traversing every corridor only once. The extended Baum-Welch algorithm needed 15 iterations to converge and produced the metric model shown in Figure 4(B). Even with only one traversal per corridor, the learned metric model is very accurate. In general, it is our experience that the Baum-Welch algorithm can learn good models with about one to three corridor traversals, depending on how confusing the corridor environment is. It is unrealistic to expect the Baum-Welch algorithm to learn the distances perfectly: the estimates are actually the *perceived* corridor lengths, and the perception can be distorted by dead-reckoning uncertainty, the discretization granularity used, and how sharply the robot turns at junctions (which may change the distance traveled by up to a meter).

difference between approximated and correct gamma values
$1/T \times \sum_{t=1 \ldots T} \sum_{s \in S} [p(s_t = s | o_{1 \ldots t+la}, a_{1 \ldots t+la-1}) - \gamma_t(s)]^2$

look-ahead la	difference	look-ahead la	difference
1	4.0×10^{-03}	7	2.1×10^{-09}
2	4.5×10^{-05}	8	4.9×10^{-10}
3	6.5×10^{-05}	9	3.8×10^{-10}
4	4.7×10^{-05}	10	1.4×10^{-10}
5	1.1×10^{-05}	...	...
6	1.4×10^{-06}	50	1.4×10^{-19}

Table 3: Approximation errors due to time windows

To determine how good the learned models are, we calculated how well they fit a (different) long "evaluation" execution trace. We used a transformation of the fit to make it independent of the length of the execution trace. Learning improves the models if the value of this transformation gets closer to zero. Table 2 shows that this is indeed the case both for the learned actuator and sensor models alone, for the learned metric model alone, and for their combination. To determine how much the learned models improve Xavier's on-line position estimation capabilities (and thus it navigation performance), we calculated the average entropy of the alpha values (the state distributions used during navigation) after every action execution of the "evaluation" execution trace. The entropy is a measure for how certain the robot knows its current state, ranging from 0 (absolute certainty at every point in time) to -1 (absolute ignorance, a uniform state distribution). If learning improves the models, we expect the entropy to get closer to zero. Table 2 shows that this is indeed the case.

To determine how well the extended Baum-Welch algorithm does, we compared the true gamma values and the approximate gamma values for various (strict) effective look-aheads. Table 3 shows that the errors start out small to begin with and then quickly become negligible for increasing look-aheads. This means that the gamma values can indeed be closely approximated with small window sizes. Not surprisingly, then, our experiments confirm that the window-based and windowless Baum-Welch algorithm learn the same models.

To test the power of the equality constraints, we repeated the second experiment without the techniques introduced in Section 5.2 except that we required each corridor to be equally long in both directions. Now, there were six corridors where the most probable length was not the correct one. Even if we used an execution trace that was twice as long (starting with the original execution trace), the result did not improve dramatically, see Figure 4(C). This is also reflected in the statistics: when using this learned model, the fit of our "evaluation" execution trace was -1.797074 and its entropy was -0.054793. Table 2 shows that the same values are only -1.684340 and -0.044137 when using the learned model from the second experiment instead. Thus, the former model is somewhat inferior, despite the larger amount of training data. We conclude that the techniques from Section 5.2 are an effective means for reducing the amount of training data required to learn good models.

The corridor environment used in this example was relatively small and thus one could have used distance learning methods with exponential run-time (such as methods that match the routes probabilistically against the topological map). The extended Baum-Welch algorithm, however, has been used successfully to learn in more complex environments than the one used here. Consider, for example, the slightly more difficult environment shown in Figure 5. We let the robot traverse parts of the corridor seven times such that it passed each landmark (doors and corridor junctions) a total of about three times. However, we did not inform the robot of its start locations, the status of the doors (all of which were open), or any distance constraints. The initial metric uncertainty from one landmark to the next was given as a uniform distribution over the lengths from 1 to 10 meters. Note that each traveled route can be matched in numerous ways against the topological map. Furthermore, doors are hard to detect: the robot cannot detect closed doors, misses open doors 22 percent of the time, and can confuse them with corridor openings. Nevertheless, the extended Baum-Welch algorithm (using a minimal look-ahead of 20 time steps) learned all distances but two correctly (it placed the corridor marked X one meter to the left of its correct position).

7 Related Approaches

Our method learns metric information passively, together with actuator and sensor models, using prior topological (and other) knowledge of its environment. In contrast, most other approaches in the literature use active exploration to learn either metric or topological maps from scratch (sometimes assuming perfect actuators or sensors) with the goal either to map the environment completely or to reach a given goal location. Approaches whose properties have been analyzed formally, for example, include [5], [10], [12], [7], and [15]. Approaches that have been demonstrated experimentally include [6], [8], and [14]. The approaches of [1] and [2] learn Markov models of the environment, as we do, but they use active exploration, while our approach is passive. The learning approach of [4] does use a passive learning approach, but it learns a topological map only. These approaches also differ from our learning method in that they learn their models from scratch.

8 Conclusion

This paper has presented a method that can simultaneously learn more accurate metric, actuator, and sensor mod-

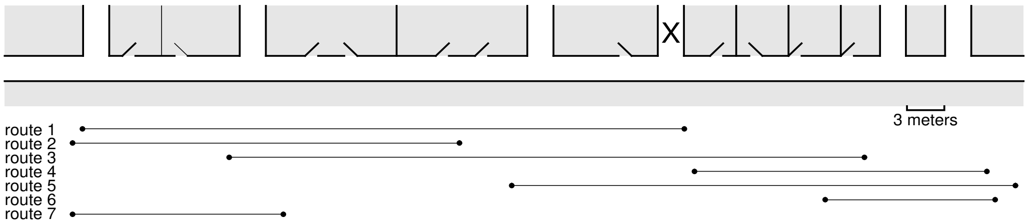

Figure 5: A long corridor with doors

els for robot navigation It is unsupervised (does not require a teacher) and passive (does not need to control the robot at any time). The training data are generated in the course of normal robot navigation. The learning method can update the models selectively, efficiently, and incrementally. The quality of the models improves as the amount of available training data increases.

Our learning method is an extension of the Baum-Welch algorithm, an expectation-maximization algorithm for learning partially observable Markov models from observations. To reduce the amount of training data needed, we augmented the Baum-Welch algorithm to take advantage of prior knowledge, such as symmetry in the map and of the sensors. This reduces the number of parameters that must be adjusted. To decrease the amount of memory required, our learning method uses a sliding "time window" on the execution trace. Time windows add a small overhead to the run time and cause a small loss in precision of the improved models, but allow the memory requirements to be dynamically scaled to the available memory.

Our experimental results confirm that the method produces good metric, actuator, and sensor models with a small amount of training data. The resulting models help the robot to determine its position more accurately, and to navigate more efficiently and reliably.

References

[1] K. Basye, T. Dean, and J.S. Vitter. Coping with uncertainty in map learning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-89)*, pages 663–668, 1989.

[2] T. Dean, D. Angluin, K. Basye, S. Engelson, L. Kaelbling, E. Kokkevis, and O. Maron. Inferring finite automata with stochastic output functions and an application to map learning. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-92)*, pages 208–214, 1992.

[3] P.A. Devijver. Baum's forward backward algorithm revisited. *Pattern Recognition Letters*, 3:369–373, 1985.

[4] S.P. Engelson and D.V. McDermott. Error correction in mobile robot map learning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA-92)*, pages 2555 – 2560, 1992.

[5] S.S. Iyengar, C.C. Jorgensen, S.V.N. Rao, and C.R. Weisbin. Robot navigation algorithms using learned spatial graphs. *Robotica*, 4:93–100, 1986.

[6] B.J. Kuipers and Y.-T. Byun. A robust, qualitative method for robot spatial learning. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-88)*, pages 774–779, 1988.

[7] V.J. Lumelsky, S. Mukhopadhyay, and K. Sun. Dynamic path planning in sensor-based terrain acquisition. *IEEE Transactions on Robotics and Automation*, 6(4):462–472, 8 1990.

[8] M.J. Mataric. Environment learning using a distributed representation. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA-90)*, pages 402–406, 1990.

[9] I. Nourbakhsh, R. Powers, and S. Birchfield. Dervish: An office-navigating robot. *AI Magazine*, 16(2):53–60, 1995.

[10] J.B. Oommen, N.S.V. Rao S.S. Iyengar, and R.L. Kashyap. Robot navigation in unknown terrains using learned visibility graphs. Part I: The disjoint convex obstacle case. *IEEE Journal of Robotics and Automation*, 3:672–681, 1987.

[11] L.R. Rabiner. An introduction to hidden Markov models. *IEEE ASSP Magazine*, pages 4–16, 1 1986.

[12] N.S.V. Rao, N. Stoltzfus, and S.S. Iyengar. A "retraction" method for learned navigation in unknown terrains for a circular robot. *IEEE Transactions on Robotics and Automation*, 7(5):699–707, 10 1991.

[13] R. Simmons and S. Koenig. Probabilistic robot navigation in partially observable environments. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 1080–1087, 1995.

[14] S.B. Thrun. Exploration and model building in mobile robot domains. In *Proceedings of the IEEE International Conference on Neural Networks (ICNN)*, pages 175–80, 1993.

[15] A. Zelinsky. A mobile robot exploration algorithm. *IEEE Transactions on Robotics and Automation*, 8(6):707–717, 12 1992.