Lifelong Learning: A case study of punting balls

Ravikiran Janardhana
Department of Computer Science
University of North Carolina at Chapel Hill
Email: ravikirn@cs.unc.edu

***Abstract*—Designing robots that learn by themselves to perform complex real-world tasks is a still-open challenge for the field of Robotics and Artificial Intelligence. In this project, I use the case study of punting balls as a lifelong learning problem. The robot is fed with demonstrations of punting a variety of balls and records the target distanced acheived and updates the physics model of the punt with each demonstration. The learning process is continuous and not stagnant and as a result of this, the internal model needs to be bounded by a finite memory and we need to represent model/data in an efficient format. After the initial demonstrations, the robot is able to predict the launch velocity and the angle of elevation given any new ball and a target distance to reach. The efficient data representation of the model is achieved using the combination of Gaussian Mixture Models (GMM) and Gaussian Mixture Regression (GMR) and I show that the memory required to store the such an internal model compared to storing each of the individual demonstrations is substantially small. Using this physics model, the robot can predict the configuration to punt a new ball to reach a target distance with high accuracy.**

I. Introduction

Throughout the last decades, the field of robotics has produced a large variety of approaches to automate and perform a variety of tasks. Despite significant progress in virtually all aspects of robotics science, most of today's robots are specialized to perform a narrow set of tasks in a very particular kind of environment. Most robots employ specialized controllers that have been carefully designed by hand, using extensive knowledge of the robot, its environment and its task. If one is interested in building autonomous multi-purpose robots, such approaches face some serious bottlenecks such as knowledge bottleneck (not aware of all the dynamics at design time), engineering bottleneck (most models are explictly hand coded) and tractability bottleneck (high computational complexity).

Machine learning aims to overcome these limitations, by enabling a robot to collect its knowledge on-the-fly, through realworld experimentation. If a robot is placed in a novel, unknown environment, or faced with a novel tak for which no a priori solution is known, a robot that learns can collect new experiences, acquire new skills, and eventually perform new tasks all by itself. For example, in [1] a robot manipulator is described which learns to insert a peg in a hole without prior knowledge regarding the manipulator or the hole. Maes and Brooks [2] successfully applied learning techniques to coordinating leg motion for an insect-like robot. Their approach, too, operates in the absence of a model of the dynamics of the system. Learning techniques have frequently come to bear in situations where the physical world is extremely hard to model by hand (e.g., the characteristics of noisy sensors). For example, Pomerleau describes a computer system that learns to steer a vehicle driving at 55mph on public highways, based on input sensor data from a video camera [3]. Learning techniques have also successfully been applied to speed-up robot control, by observing the statistical regularities of "typical" situations (like typical robot and environment configurations), and compiling more compact controllers for the frequently encountered. For example, Mitchell [4] describes an approach in which a mobile robot becomes increasingly reactive, by using observations to compile fast rules out of a database of domain knowledge.

However, there is a principle shortcoming in most of to date's rigorous learning approaches. Most of the robot control learning approaches focus on learning to achieve single, isolated performance tasks. If one is interested in learning with a minimum amount of initial knowledge, as is often the case in approaches to robot learning, such approaches have a critical limiting factor the number of training examples required for successful generalization. The more complex the task at hand and the lesser is known about the problem beforehand, the more training data is necessary to achieve the task. In many robotics domains the collection of training data is an expensive undertaking due to the slowness of robotics hardware. Hence, it does not surprise that the time required for real-world experimentation has frequently been found to be the limiting factor that prevents rigorous machine learning techniques from being truly successful in robotics.

The task of learning from scratch can be significantly simplified by considering robots that face whole collections of control learning problems over their entire lifetime. In such a lifelong robot learning scenario [5], [6], learning tasks are related in that they all play in the same environment, and that they involve the the same robot hardware. Lifelong learning scenarios open the opportunity for the transfer of knowledge across tasks. Complex tasks, which might require huge amounts of training data when faced in isolation, can conceivably be achieved much faster if a robot manages to exploit previously learned knowledge. For example, a lifelong learning robot might acquire general-purpose knowledge about itself and its environment, or acquire generally useful skills that can be applied in the context of multiple tasks. Such functions, once learned, can be applied to speed up learning in new tasks.

In order to transfer knowledge across various tasks, first we need to have an efficient physical model representation

of any task at hand rather than storing each of the individual demonstration as is. Also, it is expected that the robot continues to learn via new demonstrations and this places a limit on the amount of data it can store. In this project, I have used the example of punting a ball and analyzed the physical model required to replicate the punt trajectories. I present a novel method to store the punt physical model extracted using Gaussian Mixture Models (GMM) in an efficient matrix structure which can be later used to retrieve trajectories using Gaussian Mixture Regresstion given any new configuration. By doing so, I show that the memory required to store the model using my method is substantially smaller when compared to storing all of the trajectory data. I also show that the retrieved trajectory from this model is highly accurate without losing too much of information.

II. Method

A. Physics model of a punt

Equations for hang time and horizontal distance can be derived from the projectile equations of motion

$$y(t) = v_y t - \frac{1}{2} a_y t^2 \tag{1}$$

$$x(t) = v_x t \tag{2}$$

where y and x are the height and horizontal displacements respectively, v_y and v_x are its vertical and horizontal velocities, a_y is its vertical acceleration, and t is its time in flight. An expression for hang time, T, results from solving Eqn 1 for t when y = 0 and a = 9.8m/s^2 ,

$$T = \frac{2 v_0 sin(\theta_0)}{9.8} \tag{3}$$

where v_0 and θ_0 are initial velocity and launch angle respectively.

However, the above equations neglect air resistance which can lead to a substantial error when dealing in a practical environment. Air resistance, or drag, is a viscous force in a direction opposing the velocity of the projectile. Air drag, W, is given by the relation,

$$W = \frac{1}{2} \rho C_D A v^2 \tag{4}$$

where ρ is the air density, C_D is the drag coefficient, A is the cross-sectional area of the projectile normal to the trajectory and v is the speed of the projectile relative to the air.

In order to include the drag into the physics model, the acceleration in x and y directions have to be suitably modified as below,

$$a_x = -C_D v v_x \tag{5}$$

$$a_y = -C_D v v_y - g \tag{6}$$

where g = 9.8m/s^2 and the negative sign indicates that these forces are acting against the launch velocity.

Consider a scenario where the robot needs to predict the launch velocity and angle so that a football reaches a target distance of 100 meters assuming the Drag Coefficient C_D in air is 0.006. Table I shows the variation of maximum horizontal distance (max_x) and total time taken (t_{total}) by the projectile with and without taking drag forces into account. Using the learned model, the robot now predicts the correct launch velocity and angle to reach the target distance using a greedy optimizer. Figure 1 shows the pictorial representation of the scenario described by Table I.

TABLE I
Variation of Model parameters

Model	v_0	θ_0	max_x	t_{total}
Without Drag	31.3 m/s	45°	100.01 m	4.52 secs
With Drag	31.3 m/s	45°	63.21 m	3.6 secs
With Drag Corrected	43.31 m/s	44°	100.01 m	4.7 secs

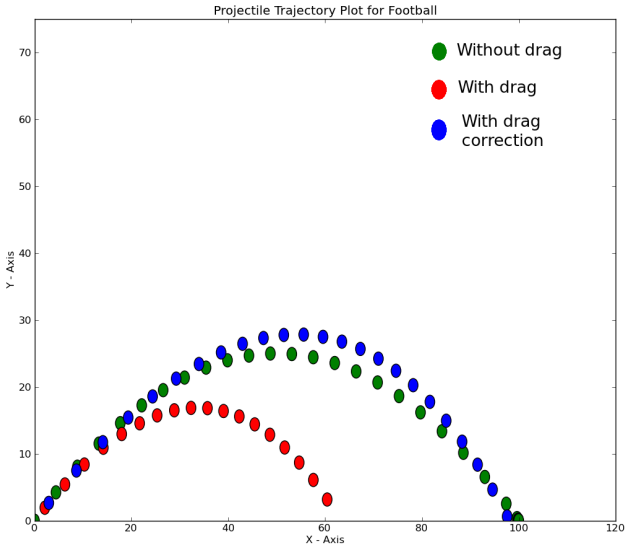


Fig. 1. Projectile with and without drag

By varying the drag coefficient C_D, we can introduce noise in the trajectory distribution and obtain various trajectories which we can use for training the robot. Figure 2 shows different noisy trajectories obtained for different values of C_D.

B. Data Representation and Retrieval

The model is composed of the following processes:

- *Probabilistic data encoding:* The data is encoded in a two stage process. First, I determine the latent space of the trajectory motion by estimating the optimal Gaussian Mixture Model (GMM) to encode the motions. Second, we encode the dynamics of the motions (i.e. the transition across the states of the GMM), using Hidden Markov Model(HMM). This is the same approach as followed in [7].
- *Optimal trajectory generation:* I compute the optimal trajectory for a new configuration via Gaussian Mixture Regression (GMR) and Lagrange optimizer if any constraints are placed.

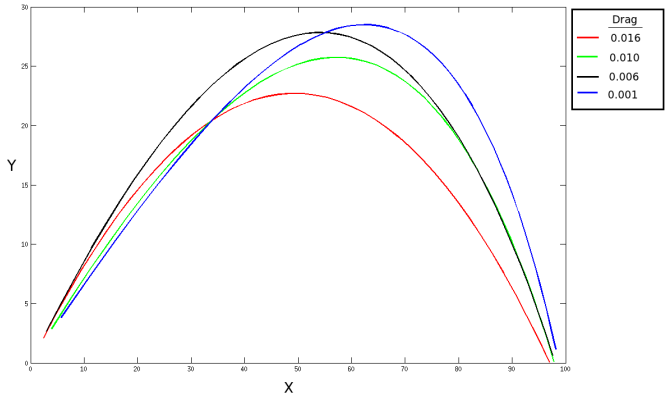


Fig. 2. Figure shows different Trajectories used for training. These are obtained by varying the drag coefficient C_D

1) Probabilistic data encoding: To avoid making too many assumptions on the spatio-temporal variability of the dataset, I use a HMM with the most general architecture, such as a fully connected continuous HMM, with full covariance matrix, describing the output variables distribution. However, using such a model requires the estimation of a large set of parameters, which can be achieved only with a large dataset. However, to program efciently a robot by demonstration, the demonstrator should not have to perform more than a few (5 to 10) demonstrations. This means that the set of parameters to learn is quite large, compared to the amount of training data.

The standard Expectation-Maximization (EM) algorithm used to estimate the HMM parameters starts from initial estimates, and converges to the nearest local maximum of the likelihood function. Thus, initialization highly affects the model performance. To better estimate the state distribution of the HMM, I perform first a rough clustering of the data using k-means. Next, I estimate a Gaussian Mixture Model (GMM) by EM, using the k-means clusters at initialization. Finally, the dynamics, i.e. transitions across states, are encoded in a HMM, with the GMM state variable distribution.

a) **Gaussian Mixture Model (GMM):** A dataset of N data of dimensionality D, X = $\{x(\vec{t}_1), x(\vec{t}_1), \ldots, x(\vec{t}_N)\}$ with $x(\vec{t}_n) \in R^D$ is modelled by a multivariate Gaussian mixture of K-components

$$p(x(\vec{t}_n)) = \sum_{k=1}^{K} \pi_k N(x(\vec{t}_n); \vec{\mu_k}, \Sigma_k) \tag{7}$$

where π_k is the prior probability on the Gaussian component k, and $N(x(\vec{t}_n); \vec{\mu_k}, \Sigma_k)$ is the D-Dimensional Gaussian density of component k. $\vec{\mu_k}$ and Σ_k are the mean and covariance matrix of the multivariate Gaussian k. $\{\pi_k, \vec{\mu_k}, \Sigma_k\}$ are estimated using the Expectation-Maximization (EM) algorithm.

b) **Hidden Markov Model (HMM):** Similarly to Gaussian Mixture Models, Hidden Markov Models use a mixture of multivariate Gaussians to describe the distribution of the data. The difference is that HMM also encapsulate the transitions probabilities between the Gaussians. It offers, thus, a way of describing probabilistically the temporal variations of the data.
Let $\{\Pi, A, B\}$ be, respectively, the initial state distribution, the transition probabilities between the states, and the multivariate output data distribution. In my experiments, I compute only $\{\Pi, A\}$ by Baum-Welch algorithm, and set $B = \vec{\mu_k}, \Sigma_k{}_{k=1}^{K}$, which are the state distributions learned by the GMM.
In order to measure the similarity between a new trajectory and the ones encoded in the model, I run the forward-algorithm, an iterative procedure to estimate the likelihood that the observed data could have been generated by the model.

c) **Gaussian Mixture Regression (GMR):** To reconstruct the trajectory from the GMM/HMM encoding, after training and generalization over the demonstrations, I apply a Gaussian Mixture Regression (GMR). For a D-Dimensional variable $\vec{x} \in R^D$, the means and covariance matrices given by the GMM/HMM representation for component k are given by $\vec{\mu}^H_{kX}$ and Σ^H_{kX}. The regression is done along the time index. We compute the means and covariance matrices of the set of observations $\{t, \vec{x}(t)\}$ with dimension (D+1). Note that the sole time-indexed covariances matrices and means are estimated, since the rest of the means and covariance matrices $\{\vec{\mu}^H_{kX}, \Sigma^H_{kX}\}$ have already been estimated:

$$\vec{\mu}^R_k = \{\vec{\mu}^R_{kt}, \vec{\mu}^H_{kx_1}, \vec{\mu}^H_{kx_2}, \ldots, \vec{\mu}^H_{kx_D}\} \tag{8}$$

$$\Sigma^R_k = \begin{pmatrix} \Sigma^R_{kt} & \Sigma^R_{ktX} \\ \Sigma^R_{kXt} & \Sigma^H_{kX} \end{pmatrix} \tag{9}$$

The Gaussian Mixture Regression estimates:

$$\vec{x}^d(t) = \sum_{k=1}^{K} \beta_k(t) \vec{x}^d_k(t) \tag{10}$$

$$\beta_k(t) = \frac{\pi_k N(t; \mu^R_{kt}, \Sigma^R_{kt})}{\sum_{i=1}^{K} \pi_i N(t; \mu^R_{it}, \Sigma^R_{it})} \tag{11}$$

$$\vec{x}^d_k(t) = \vec{\mu}_{kX} + \Sigma^R_{kXt} {\Sigma^R_{kt}}^{-1} (t - \mu_{kt}) \tag{12}$$

$\vec{x}^d_k(t)$ are the regression output for each associated Gaussian component k, $\beta_k(t)$ the corresponding weight, that measures the relative influence of component k, and π_k the prior probability. $\vec{x}^d(t)$ is the desired trajectory, a generalized form of the motion learned during training. The time dependent inverse covariance of $\vec{x}(t)$ is estimated by:

$$W^x(t) = \left(\sum_{k=1}^{K} \beta_k(t) \Sigma^R_k\right)^{-1} \tag{13}$$

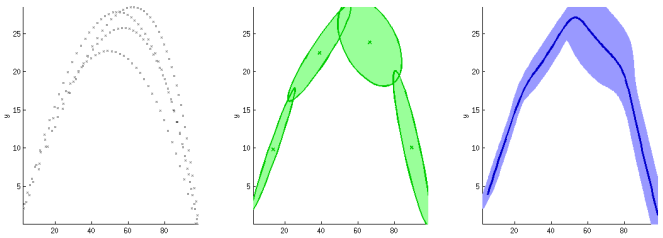


Fig. 3. The first subfigure shows the different demonstration trajectories, the second subfigure shows the GMM encoding (no. of states = 4) of the trajectories and the third subfigure shows the retrieved generalized trajectory via GMR.

Figure 3 shows the results of GMM encoding of the data and the generalized trajectory retrieved via GMR.

C. Continuous Learning and Data Storage

In lifelong learning, the robot is expected to learn continuously over time and update its internal model to accurately reproduce trajectory. In doing so, there needs to be a bound on the memory when the no. of input trajectories explode. I have researched on this problem and using the proposed data representation and encoding, I show that the memory required to store such a model is substantially small when compared to storing each of the individual demonstrations. The following section explains my approach in detail.

To demonstrate my method, I use the example of the projectile trajectories. In the normal case, one needs to store all the trajectories in order to efficiently represent the physics model. However, using the data representation approach detailed above, we only need to store the priors, means and covariance matrices of k components describing the demonstrations.

In lifelong learning, if a new trajectory is required to be processed by the system, depending on how much variance we allow within a single class of trajectories, a decision needs to be made as to whether we include it in one of the existing trajectory classes or split it into a new class. There is a fundamental assumption of an already existing minimal model for this approach to work. Hence, the major advantage in my approach is that I store trajectory classes instead of the actual trajectories. Another advantage is instead of storing all the data points within a trajectory, I replace them with the GMM encoded priors, means and covariances matrices of the components describing the trajectory. The math equations of my method are described below.

Let,

K = total no. of trajectories.
N = total no. of trajectory splits.
n = no. of states in the gaussian mixture model.
m = no. of gaussian mixture components.
ϵ = threshold value of trajectory class variation.
$traj_{points}$ = no. of data points per trajectory.
σ = SD of the components in the trajectory class.
μ = mean of the components in the trajectory class.
σ' = SD of the components in the updated trajectory class.
μ' = mean of the components in updated trajectory class.
Δ_{encode} = change of variation in updated trajectory class.
GMM_{encode} = variation of the trajectory class (0 - 1).
GMM_{memory} = used memory with GMM encoding.
$Regular_{memory}$ = used memory without GMM.
$bytes_{double}$ = no. of bytes to store a double value

Assume that there is an existing model, when a new trajectory is fed into the robot to process, a decision needs to be made whether the new trajectory can be put in one of the existing trajectory classes or a new split needs to be made. To make this decision, I compute the variation of all existing trajectory classes Δ_{encode} up on including the new trajectory as below,

$$\Delta_{encode} = \frac{\sum_{i=1}^{m}\sum_{j=1}^{n}\sum_{k=1}^{n}|\sigma_{ijk} - \sigma'_{ijk}| + \sum_{i=1}^{m}\sum_{j=1}^{n}|\mu_{ij} - \mu'_{ij}|}{\epsilon}. \tag{14}$$

The GMM_{encode} is a scaled version of Δ_{encode} which varies between 0 to 1 and is computed as below,

$$GMM_{encode} = \begin{cases} 1, & \text{if } \Delta_{encode} \geq 1 \\ \Delta_{encode}, & \text{otherwise} \end{cases}$$

If the GMM_{encode} value is more towards 0, it indicates that the trajectory class has little or no variation and hence can accomodate newer trajectories. As and when, the GMM_{encode} nears 1, it indicates that a split is required as the variation limit has already been reached in the existing class.

The memory usage with and without GMM encoding can be computed as below,

$$GMM_{memory} = N * bytes_{double} * (nm + n^2m + n). \tag{15}$$

$$Regular_{memory} = K * bytes_{double} * m * traj_{points}. \tag{16}$$

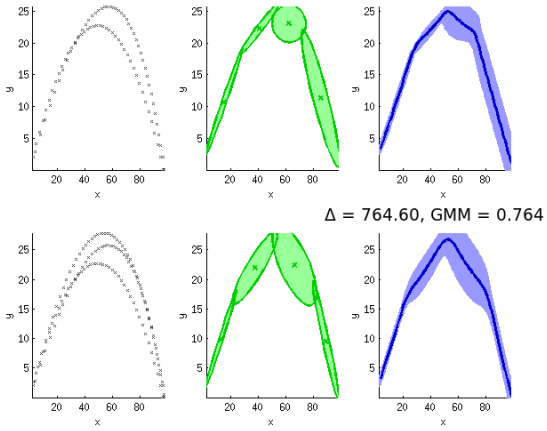
III. Experimental Results

For the experiments, I produced 100 demonstration trajectories using the Physics model outlined in Section IIA. Each of the trajectory has 50 data points and captures 3 components namely horizontal position *x*, vertical position *y* and the time *t*. These trajectories are used to validate my method in the following sections.

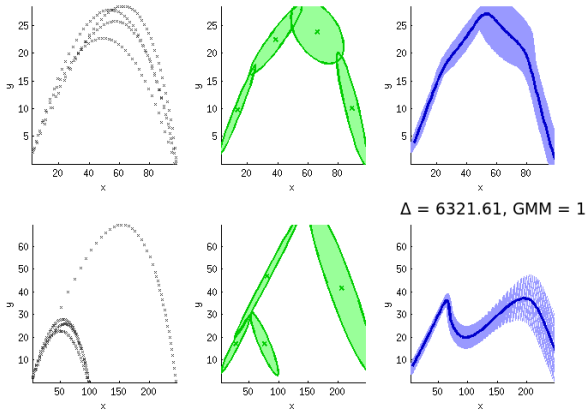
A. Expanding/Splitting trajectory classes

Figure 4 shows the result of expanding and splitting of existing trajectory classes while processing new trajectories given that the variation threshold of the trajectory class is $\epsilon = 1000$. In Figure 4, consider Case I, the existing model consists of 3 trajectories which are very similar having a target distance of 100±5 m. A fourth trajectory is encountered which has a target distance of 97m and upon including this trajectory in the class, the Δ_{encode} computed is 764.60 and the GMM_{encode} is 0.764 which is less than 1 and hence, it means that the new trajectory can be included in the class.

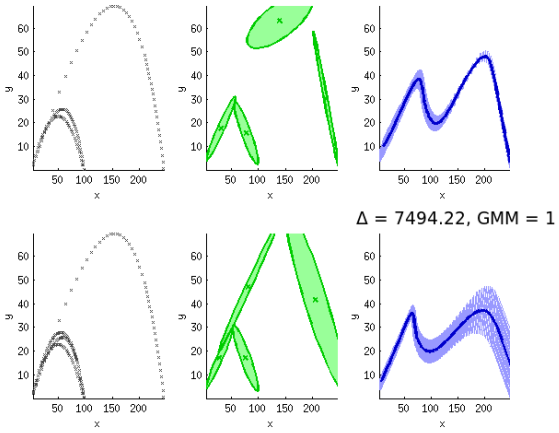
Consider Case II, the existing model consists of similar trajectories with target distance of 100±5m. The new trajectory is having the target distance of 250m and upon including this trajectory in the class, the Δ_{encode} computed is 6321.61 and



(a) Case I



(b) Case II



(c) Case III

Fig. 4. In each of the figures, the three subfigures (Trajectory points, GMM Encoded Trajectory, GMR retrieved trajectory) in the upper half correspond to the existing trajectory class and three subfigures in the lower half correspond to the updated trajectory class. (a) Case I - The figure shows 3 existing trajectories captured in the model and on processing a new trajectory, the variation captured is $\Delta_{encode} = 764.60, GMM_{encode} = 0.764, \epsilon = 1000$, (b) Case II - The variation captured is $\Delta_{encode} = 6321.61, GMM_{encode} = 1.00, \epsilon = 1000$, (c) Case III - The variation captured is $\Delta_{encode} = 7494.22, GMM_{encode} = 1.0, \epsilon = 1000$.

the GMM_{encode} is 1.0 which means that the new trajectory cannot be included in the class as it will exceed the variation of the class greater than the threshold and will hurt the quality of the retrieved trajectory significantly.

Consider Case III, the existing model is faulty and includes more variation than intended and on trying to include a new trajectory which is again different from existing trajectories, the Δ_{encode} computed reads 7494.22 and the GMM_{encode} is 1.0. This result shows that the Δ_{encode} captures the change to high accuracy and indicates the worsening of system when successive bad trajectories are added to a class, indicating that a trajectory class split is highly necessary.

Thus, Δ_{encode} and GMM_{encode} are indicators which can be used to track and maintain trajectory classes instead of individual trajectories. This also gives a logic of when to split trajectory classes which has not been addressed in previous research.

B. Memory Usage

I encoded the 100 demonstration trajectories each of 50 data points ($traj_{points} = 50$) using GMM-encoding multiple times with a threshold variation of $\epsilon = 1000$ and found out that the no. of trajectory class splits were 10-25 (N = 10-25) always even with random velocities and launch angle. I used n=6 states in the GMM encoding and m = 3 (no. of components tracked - x, y, t). Let the number of bytes required to store a double be 8 ($bytes_{double} = 8$) Given, the above configuration, the number of bytes required to store the encoded demonstrations with and without GMM can be computed using Equations 15 and 16 as below,

GMM_{memory} = (10 to 25) * 8 * (6*3 + 6^2*3 + 6) = 10,560 to 26,400 bytes.

$Regular_{memory}$ = 100 * 8 * 3 * 50 = 120,000 bytes.

As seen from above values, the reduction is by a factor of 10

TABLE II
RETRIEVED TRAJECTORIES

No.	$dist_{expected}(m)$	$dist_{retrieved}(m)$	$\lvert\%dist_{error}\rvert$
1	100.00	99.95	0.05
2	150.00	148.02	1.32
3	200.00	202.36	1.18
4	1000.00	992.14	0.786
5	2000.00	2020.01	1.005

which is highly significant when you consider a large dataset over a lifetime.

C. Accuracy of the Retrieved Trajectories

Once the data has been modeled as a GMM, the retrieval of trajectories for any new configuration are done via the GMR. The accuracy of the retrieved trajectories in terms of the horizontal distance covered are as shown in Table II which shows the expected distance ($dist_{expected}$), retrieved distance from the model ($dist_{retrieved}$) and the absolute percentage error between the two ($|\%dist_{error}|$). From the experiments I conducted, the average absolute percentage error between the retrieved and the expected distance was found to be $2.03\%\pm1.02\%$, which shows that the model is highly accurate in terms of the horizontal distance covered. However, this does not show the quality of the trajectories i.e., how accurate the trajectory is at each retrieved data point when compared to the ground truth which is beyond the scope of this project.

IV. CONCLUSION

Lifelong learning helps robots to tackle variety of tasks and enables sharing of knowledge across tasks rather than re-learning everytime. To enable this to happen, we first needs to establish an efficient storage mechanism for the task's physical model and accurate retrieval of data for new configurations. In this project I have tried to solve this problem by using the projectile motion of punting a ball as an example. I encode the physical model using GMM encoding and as a result I show that the data can be compressed substantially (factor of 10) when compared to storing all data points, the logic for splitting the trajectory classes is based on the values of Δ_{encode} and GMM_{encode} and highly accurate trajectories are retrieved using GMR.

V. FUTURE WORK

The results obtained for the method described are highly promising (See Section III). However, validation of the trajectory retrieved at each data point is something to be done in the future work. The proposed also needs to be tested on various scenarios which can be thought of analogous to punting the ball example. Below are some of the example tasks which can be used to validate the method proposed so that it can be accepted as a generalized framework,

- Putting a golf ball
- Throwing darts
- Throwing a basketball into a hoop
- Knocking out a cricket stump with a ball, etc

ACKNOWLEDGMENT

I would like to thank Dr. Ron Alterovitz for his continuous feedback throughout this project which helped me a great deal in overcoming many of the obstacles encountered in this project. I would also like acknowledge Dr. Sylvain Calinon as his implementation of the GMM and GMR formed the basis of my project on which I extended my method.

REFERENCES

[1] Vijaykumar Gullapalli, Judy A. Franklin, and Hamid Benbrahim. *Acquiring robot skills via reinforcement learning. IEEE Control Systems*, i72(1708): 13-24, February 1994.

[2] Pattie Maes and Rodney A. Brooks. *Learning to coordinate behaviors.* In Proceedings Eighth National Conferenceon Artificial Intelligence, pages 796-802, Cambridge, MA, 1990. AAAI, The MIT Press.

[3] D. A. Pomerleau. *ALVINN: an autonomous land vehicle in a neural network.* Technical Report CMU-CS-89- 107, Computer Science Dept. Camegie Mellon University, Pittsburgh PA, 1989.

[4] Tom M. Mitchell. *Becoming increasingly reactive.* In Proceedings of1990 AAAl Conference, Menlo Park, CA, August 1990. AAAI, AAAl Press I The MIT Press.

[5] Sebastian B. Thrun and Tom M. Mitchell. *Lifelong robot learning. Robotics and Autonomous Systems, 1993.* Also appeared as Technical Report IAI-TR-93-7. University of Bonn, Dept. of Computer Science 111.

[6] Thrun, S. B. 1994. *A Lifelong Learning Perspective for Mobile Robot Control.* In Proceedings of the IEEE/RSJ/GI International Conference on Intelligent Robots and Systems, 23-30. Washington, D.C.: IEEE Computer Society.

[7] Calinon S, Guenter F, Billard A (2006). *On Learning the Statistical Representation of a Task and Generalizing it to Various Contexts.* Proc IEEE/ICRA 2006.