Reinforcement Learning with Gaussian Processes for Unmanned Aerial Vehicle Navigation

Nahush Gondhalekar

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master Of Science
in
Computer Engineering

Pratap Tokekar, Chair
Haibo Zeng
A. Lynn Abbott

June 23, 2017
Blacksburg, Virginia

Keywords: Reinforcement Learning,Gaussian Processes

Reinforcement Learning with Gaussian Processes for Unmanned Aerial Vehicle Navigation

Nahush Gondhalekar

(ABSTRACT)

Abstract Goes here ..

Grant information goes here ..

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This is a general introduction of the idea

1.1 Applications to general scenarios and Challenges

This section shall talk about the application and the shortcomings

1.1.1 Subsection 1

Any appropriate subsections.

1.2 Contributions

This section highlights contributions of our work.

1.2.1 Subsection 2

Any appropriate subsections.

1.3 Organization of the Thesis

This section should talk about how the chapters are organized.

Chapter 2

Background

2.1 Sequential Decision Making

A delivery truck trying to decide which house to go first on a tour to deliver the packages, a sudden price drop in the merchandise trying to increase the sales, a robot trying to explore unknown environments. These are all examples of sequential decision making. One of the main factors in sequential decision making is that the decisions made now can have both immediate and long term effects [**?**]. Sometimes it's effective to make a greedy choice but sometimes the decisions made depend critically on future situations. So how to approach sequential decision making?

2.1.1 Approaches to solve sequential decision making

Though in this thesis we are mainly interested in the algorithms which deal with *learning*, there exist other algorithms which are used to solve the problems related to sequential decision making. The following ways can be used in approaching sequential decision problems [**?**].

- *Programming*: For each possible event/outcome, try to specify an appropriate or optimal action *a priori*. In most of the general scenarios, this is not possible due to the massive state space of the problem or the intrinsic uncertainty of the environment or both. These solutions may work only for completely known static environments with fixed probability distribution.
- *Search and Planning*: If we know the dynamics of a system, it is easy to plan *Search and Planning* from a defined start state to a goal state. With an uncertainty element added to the environment for the outcomes of the actions, the standard algorithms

would not work. Additionally, we are looking for a general policy for all states of an environment.

- *Learning*: Imagine a robot trying to navigate an unknown maze trying to get out of it. The robot is mounted with an onboard laser sensor for obstacle detection. It's much easier and faster for the robot to interact with the environment to gather data and find the door which gets it to the exit of the maze. So why is *Learning* an effective way to solve the Sequential Decision Problems. The section 2.1.2 briefly enlists the advantages.

2.1.2 Why Learning?

- No need to perform the tedious task of trying to program all the possibilities in the design phase.
- Learning can effectively cope with uncertain environments, changing states and actions and reward oriented goal finding.
- It can successfully solve the given problem for all the states and come up with a general policy.

2.1.3 Online Vs Off-line Learning

Let's again take an example of an Unmanned Aerial Vehicle (UAV) which needs to fly close to a surface of a bridge and take pictures. It is difficult to control it manually with an erratic Global Positioning System (GPS) signal and wind disturbances near the bridge surface. What if the UAV is equipped with a knowledge of maintaining a certain distance from the bridge surface and fly near the bridge autonomously?

This demands for a situation where the UAV needs to be trained to perform the task of flying close to a surface without hitting the obstacle. Learning the controller directly on the real task *Online* is often difficult since learning a task needs a lot of data which sometimes is too time consuming. More importantly, it is not very economic and *safe*, since there is a chance of the quad-rotor colliding several times with the bridge causing financial and other setbacks. It is often desirable to train the robot in a simulator which provides much faster and *safe* training situations where the agent can explore and can afford to make mistakes. *Off-line* learning uses a simulator of the environment as a *cheap* way to gather samples in a *fast* and a *safe* way. Often times one can use the simulators to obtain a reasonable policy for a given problem and then *fine tune* it in the real world.

2.1.4 Rewards, and how to assign them ?

The important aspect of Sequential Decision Making is the fact that, if the action is going to result in *good* or *bad* outcomes, cannot be decided right away. Sometimes the first action may have a large influence in reaching the goal even though the actions between the first one and the reward obtained at the end, may be *bad*. A formal model to represent such problems would be extremely useful in analyzing and solving sequential decision making problems. In section 2.2 we would take a look at the most popular way to represent sequential decision making in an uncertain environment.

2.2 Markov Decision Processes

Markov Decision Processes are popularly used to model sequential decision making when the outcomes of the actions are uncertain. [**?**]. In fact MDPs have become the *de facto* standard formalism for learning sequential decision making. When an action is taken in a particular state, a reward is generated and a next state is attained through a particular transition probability.

Definition 1 *A Markov decision process is a tuple* (S, A, T, R) *in which* S *is a finite set of states,* A *a finite set of actions,* T *a transition function defined as* $T : S \times A \times S \to [0, 1]$ *and* R *a reward function defined as* $R : S \times A \times S \to R$.

As seen in Definition 1, MDPs consist of states, actions, transitions between states and a reward function.

- States: The set of states S is defined as the finite set $\{s^1, ..., s^N\}$ where the size of the state space is N, *i.e.*$|S| = N$. For example, for a robot moving in a $2D$ grid-world like environment, each position (x, y) may be represented as a unique state.
- Actions: The set of actions A is defined as the finite set $\{a^1, ..., a^M\}$ where the size of the action space is M, *i.e.*$|A| = M$. Actions can be used to control the system state. The set of actions that can be applied in some particular state $s \in S$, is denoted $A(s)$, where $A(s) \subseteq A$. For examples, moving the robot forward, backward or sideways can be recognized as actions.
- The Transition Function: When an action $a \in A$ is applied in a state $s \in S$, then the agent makes transition to a state $s' \in S$. This transition is based on a probability distribution over all the states. The probability of ending in state s' when an action a is taken in s, is denoted by $T(s, a, s') \leq 1$.

- The Reward Function: The reward functions determines the reward obtained by the agent for being in a state or performing some action in a state, $R(s, a, s')$ defines the reward obtained by the agent for performing action a in state s and it lands in state s'.

2.2.1 Policies

A policy for an MDP is a function which gives an action $a \in A$ as an output for each state $s \in S$. Formally, a deterministic policy π is a mapping from $\pi : S \rightarrow A$. Optimal policy π^* is computed and used by the agent to try to optimize its behavior in the environment modeled as an MDP. *Optimality* is dependent on what exactly is being optimized? What is the goal of the agent? Add Bellman equation and briefly describe value function etc.

2.2.2 Solving MDPs

As discussed in the section 2.2.1, finding an optimal policy π^* is an important task of the agent. *Solving* an MDP is nothing but finding the optimal policy π^*. When the model of the MDP is known, the algorithms may be classified under Model Based Add reference to RL section algorithms. Here, we know the transition function and the reward function of the environment. These can be used to find out the value functions and policies using Bellman Equation. Add Bellman equation and its reference

In Model-Free Add reference to RL section algorithms, agent relies on the *interaction* with the environment. *Simulations* or actual execution of the policy leads to collect *samples* of transition functions and rewards which are thereby used to determine the state-action value functions.

To directly compute the value functions for the state-action pairs when the model of the environment is not available or to determine the model by interacting with the environment in order to maximize the performance, the agent needs to posses artificial intelligence. We will discuss this branch of machine learning the next section (section 2.3).

2.3 Reinforcement Learning (RL)

Reinforcement Learning is a class of problems where the agent needs to learn the *behavior* with trial and error by interacting with a dynamic environment. The two main methods to approach Reinforcement Learning Problems are:

- To search the entire behavior space in order to find the optimal one for the given environment. Examples of these kind of algorithms are genetic programming algorithms.

- Use of statistical methods and dynamic programming to estimate the effect of actions in the states of the given environment. [**?**]

2.3.1 Model of Reinforcement Learning

A simplified Reinforcement Learning Model can be represented as shown in figure 2.1 Formally,

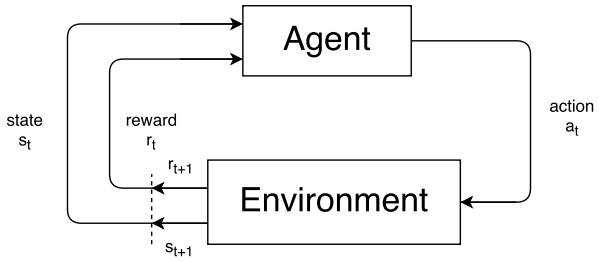


Figure 2.1: A Simplified Reinforcement Learning Model

the model consists of a set of environment states, S a set of agent actions, A; and a set of scalar reinforcement signals which are typically known as rewards. The agent's job is to find a policy π, mapping states to actions ($\pi : S \rightarrow A$) to maximize some long term measure known as *reward*. The environment is typically assumed to be stochastic *i.e.*the same action in the same state at two different occasions may result in two different next states and/or two different rewards. The main difference between Reinforcement Learning and Supervised Learning is that, in RL, the agent needs to gather experience by interacting with the environment to find out the system states, transitions, actions and rewards whereas in Supervised Learning, some information about prior input/output pairs is used to approximate the mapping function.

2.3.2 Exploration Vs. Exploitation

An RL agent needs to explicitly explore the environment. One of the classic problems in the literature known as the *k-armed bandit problem* [**?**] may best illustrate the concept of exploration Vs. exploitation. The agent is in a room with a collection of k gambling machines. The agent is permitted a fixed number of pulls, z. Any arm may be pulled on each turn. The machines do not require a deposit to play; the only cost is in wasting a pull playing a suboptimal machine. When arm i is pulled, machine i pays 1 or 0, according to some underlying probability parameter p_i, where the pays are independent events and the p_is are unknown. What should the agent's strategy be to maximize the payoffs?

The agent may believe that some arm may have a higher payoff probability. The question now is, whether to choose the same arm all the time or to explore the other arm that has

less information but seems to be better? Depending on how long the game is going to last, if the game is going to last longer, then the agent should not converge to a sub-optimal arm too early and instead explore more. In short, exploitation means to use the knowledge that the agent has found for the current state s by doing one of the actions a that maximizes the value function of the state whereas exploration means, in order to build a better estimate of the optimal value function it should select a different action from the one that it currently thinks is best.

We will see one of the ways to trade of exploration-exploitation using ϵ-greedy strategy used in Q-learning which is a form of model-free learning. The section 2.3.3 describes the two fundamental types of Reinforcement Learning.

2.3.3 Classification/Types

As we have seen previously, Reinforcement Learning agent needs to know the model of the environment; The transition function $T(s, a, s')$ and the reward function $R(s, a)$ in order to find an optimal policy. How to find an algorithm to find an optimal policy when such a model is not known in advance? There may be two approaches we can take [**?**].

- Model-free: Learn a controller without learning the model.
- Model-based: Learn the model of the environment in order to derive the controller.

Model-free

Model-free algorithms are the algorithms which do not have an explicit knowledge of the environment. The evaluation of *how good the actions are* is done through trial and error. The space complexity of such algorithms may be considered to be asymptotically less than the space required to store an MDP [**?**]. Model free algorithms tend to keep track of the value functions only unlike the model-based algorithms which tend to store the environment model information.

The limitation of model-free algorithms may be stated as; these algorithms need **extensive experience** to find an optimal policy. A few examples of model-free algorithms are; *Q-Learning* (section: 1) and SARSA (State Action Reward State Action).

Model-based

The on-line algorithms such as Q-Learning guarantee convergence if the approximations are accurate. It is expensive to run these algorithms in the real world. In model-based

algorithms, an agent tries to learn the model of the environment while obtaining on-line experience and then this model can be used to facilitate learning.

Given a state s and an action a, the probability of going to the next state s' is given by the transition function $T(s, a, a')$. The reward obtained by taking an action a in state s is given by the reward function $R(s, a)$. $T(s, a, s')$ and $R(s, a)$ can be learned and be used during the further runs in order to improve the convergence. It is expected that, during the subsequent runs of the algorithm, the model should already be sufficiently learned to speed up convergence. Extensive amount of computation is required in model-based algorithms. A few examples of model-based reinforcement learning algorithms are; $R\text{-}MAX$ [**?**] and E^3 [**?**].

2.3.4 Q-Learning

What is Q-Learning?

The basic concept behind Q-learning is that we have a representation of the environmental states $s \in S$ and actions $a \in A$, and an agent is trying to learn the value of each of these actions in each of these states. The agent maintains a table $Q[s, a]$ and at each state s tries to choose the action a which maximizes the function $Q(s, a)$. The formal definition of the Q function is given as follows:

$$Q(s, a) = r(s, a) + \gamma \max_{a'}(Q(s', a')) \tag{2.1}$$

where,
$r(s, a)$ is the immediate reward,
γ is the relative value of the delayed Vs. immediate reward, also known as *discount factor* $\gamma \in [0, 1]$,
s' is the new state after action a is taken in state s,
a, a' are the actions in state s and s' respectively.
The selected action $\pi(s) = \arg\max_a Q(s, a)$.

2.3.5 Q-Learning Algorithm

Input : S is a set of states
A is a set of action
γ is the discount factor
α is the learning rate or step size ($\alpha \in (0, 1)$)
Local data: Store the whole Q matrix $Q[S, A]$,
previous state s
previous action a,
next state s'
Initialize : Initialize $Q[S, A]$ arbitrarily/Zero
while *Termination condition not reached* **do**
  Choose an action a in the current state s based on the current Q-value estimates.
  Perform the action a in the state s and observe the current reward r and the next state s'.
  Update $Q(s, a) := Q(s, a) + \alpha[r(s, a) + \gamma \max_{a'} Q(s', a') - Q(s, a)]$:
end

Algorithm 1: Q-Learning Algorithm

The update equation given in algorithm 1 updates the Q-value of the last state-action pair (s,a) with respect to the observed next state s' and the reward r with a learning rate parameter $\alpha \in (0, 1)$. Recalling Bellman update equation from section 2.2.2, we can see that updated Q-value is the expected sum of the reward and discounted value of the next state. Unlike a general Bellman update equation, we are not marginalizing over all possible next states, since we have only observed one state here along with the reward for a particular state-action pair.

However, we can control the parameter α, which is the learning rate so as to allow the Q-value to change slightly from its old estimate to a new estimate in the direction of the observed state and reward.

The need of learning Q values.

Consider an agent that has to navigate through a grid given in the figure 2.2. The walls in the grid block the agent's path. The actions taken by the agent are noisy *i.e.* 80% of the time, the action North takes the agent North if there's no wall, 10% of the time action North takes the agent West, and 10% of the times to East. Each step taken by the agent has a small negative reward say -0.25. A reward of $+1$ or -1 is given at the states shown in figure 2.2. The goal of the agent is to maximize the sum of the rewards.

This problem can be solved using value iteration where the Bellman equations Reference the equation here characterize the optimal values. The agent has to store the Q values for

each state-action pair ($Q(s, a)$). The space required to store the Q-table is $S \times A$. In this example, the number of states are 11 and number of possible actions are 4. The efficiency of recursive Bellman update is $O(S^2)$. Imagine when the state space becomes larger and the time complexity increases with the square of number of states. A naive Q-Learning approach wouldn't be able to scale up to give faster solutions as the problem space grows. We will see that in the following scenario.

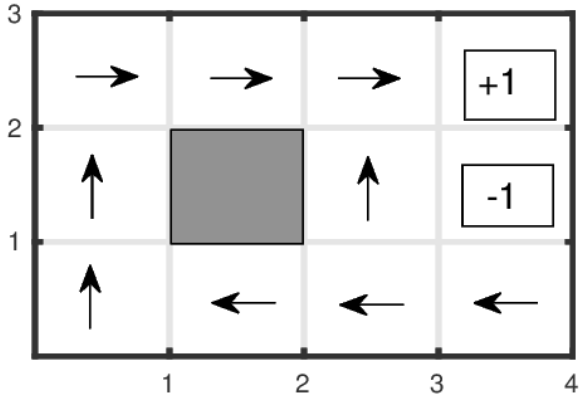


Figure 2.2: An example gridlworld with stochastic actions

Learning Laser data and space complexity!

Consider the following example. A robot (represented in green) with 6 laser sensors is trying to navigate in an environment without colliding with obstacles of different sizes which are moving with different speeds. (The white obstacles are moving at slower speeds whereas the orange obstacle moves faster). The laser sensor can output distance values in the range 0

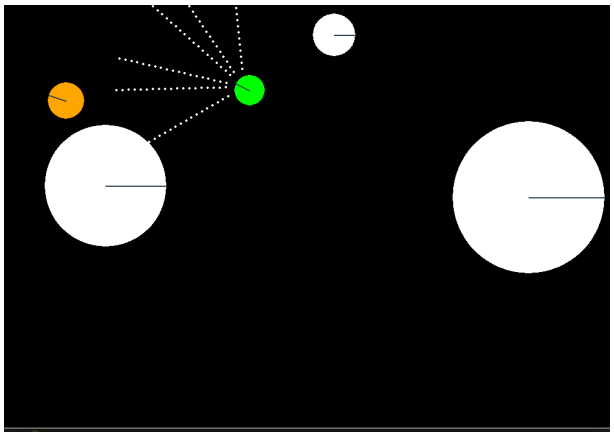


Figure 2.3: Navigation with perception in the loop

to 19. If we consider the state space to be the laser data $[l_1, l_2, l_3, l_4, l_5, l_6]$ and each of the 6 sensors can give 20 values, then we have a total of 2×10^6 possible states. In terms of space and time complexity to learn the $Q[S, A]$ and eventually an optimal policy, this state space is too large for a naive Q-Learning algorithm to achieve faster convergence. As we have seen in section 2.3.3, model-free algorithms need extensive experience in order to compute optimal policies. To experience states of the order of 10^6 is a lengthy process. What could be done to approximate and better estimate all the $Q[S, A]$ values based on the observed set of Q values?

2.4 Introduction to GPs

2.4.1 Gaussian Processes for Machine Learning

2.4.2 Use of GPs in Q learning

Chapter 3

The GP-Q Algorithm

3.1 What is "Perception in the loop"?

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam ultricies

3.1.1 Laser data and feedback

Nunc posuere quam at lectus tristique eu ultrices augue venenatis. Vestibulum

3.1.2 How is it useful?

Morbi rutrum odio eget arcu adipiscing sodales. Aenean et purus a est pulvinar pellentesque. Cras in elit neque, quis varius elit. Phasellus fringilla, nibh eu

3.2 The Simulator setup and Software

Sed ullamcorper quam eu nisl interdum at interdum enim egestas. Aliquam placerat justo sed lectus lobortis ut porta nisl porttitor. Vestibulum mi dolor, lacinia

Chapter 4

The GP-MFRL Algorithm

In this chapter,the use of GP regression to learn the transition function is described and the details about the GP-MFRL algorithm – the main work of this thesis are provided. The simulator system setup is discussed in the section

4.1 Multi-Fidelity Reinforcement Learning

We build our work upon a recently proposed Multi-Fidelity Reinforcement Learning (MFRL) algorithm by Cutler et al. [**?**]. MFRL leverages multiple simulators to minimize the number of real world (*i.e.*, highest fidelity simulator) samples. The simulators denoted by $\Sigma_0, \ldots, \Sigma_D$, have increasing levels of fidelity with respect to the real environment. For example, Σ_0 can be a simple simulator that models only the robot kinematics, Σ_1 can model the dynamics as well as kinematics, Σ_2 can additionally model the wind disturbances, and the highest fidelity simulator can be the real world (Figure 4.1).

MFRL differs from transfer learning [**?**] where transfer of parameters is allowed only in one direction. The MFRL algorithm starts in Σ_0. Once it learns an optimal policy in Σ_0, it switches to a higher fidelity simulator. If it observes that the policy learned in lower fidelity simulator is no longer optimal in the higher fidelity simulator, it either switches back to a lower fidelity simulator or stays at the same level. It was shown that the resulting algorithm has polynomial sample complexity and minimizes the number of samples required from the highest fidelity simulator.

The original MFRL algorithm uses Knows-What-It-Knows (KWIK) framework [**?**] to learn the transition and reward functions in each level. The algorithm essentially maintains a mapping from a state-action pair to the learned reward and the next state. The reward and the transition for each state-action pair is learned independently of others. While this is reasonable for general agents, in case of planning for robots we can exploit the spatial

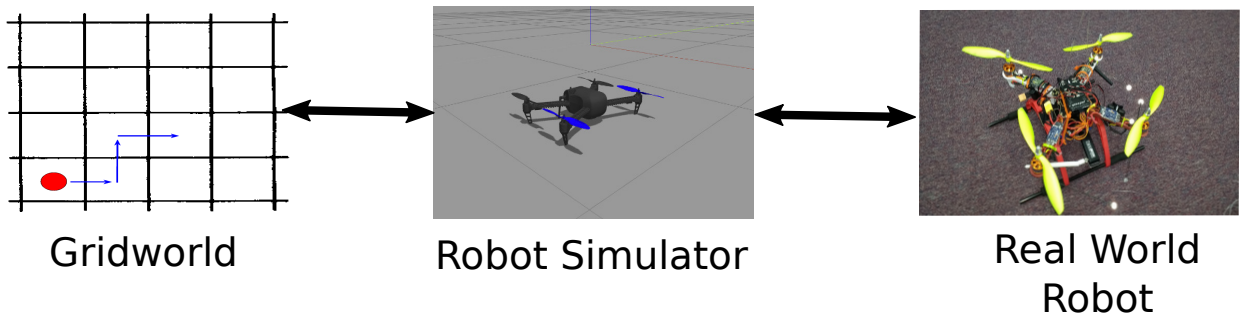


Figure 4.1: MFRL framework: First simulator captures only gridworld movements of a point robot while second simulator has more fidelity using a physics simulator. Control can switch back and forth between simulators and real environment which is essentially the third simulator in the multi-fidelity simulator chain.

correlation between neighboring state-action pairs to speed up the learning. Our main contribution in this paper is to show how to use Gaussian Process (GP) regression to learn the transition function in an MFRL framework using fewer samples.

GPs are commonly used to learn transition models for agents moving in the real world [**?**] and have been used in RL to learn the transition function [**?**], the reward function [**?**] and the value function [**?**]. GPs can predict the learned function for any query state-action pair, and not just for the discretized set of state-action pairs used when planning. In MFRL, the state space of Σ_i is a subset of the state space of Σ_j for all $j > i$. Therefore, when the MFRL algorithm switches from Σ_i to Σ_{i+1} it already has an estimate for the transition function for states in $\Sigma_{i+1} \setminus \Sigma_i$. Thus, GPs are particularly suited for MFRL which we verify through our simulation results.

4.2 Learning Transition Dynamics as a GP

A Markov Decision Processes (MDP) [**?**] is defined by a tuple: $\langle S, A, \mathcal{R}^a_{ss'}, \mathcal{P}^a_{ss'}, \gamma \rangle$. S and A are the set of states and actions respectively. $\mathcal{R}^a_{ss'}$, referred to as reward dynamics, defines the reward received by the agent in making a transition from state s to s' while taking action a and $\mathcal{P}^a_{ss'}$ is the probability of making this transition (also referred to as transition dynamics).

Generally the agent does not know the reward it will receive after making a transition nor does it know the next state it will land in. The agent learns these parameters through interactions with the environment which is subsequently used to plan an optimal policy to earn the maximum expected reward, *i.e.*, $\pi^* : S \to A$.

RL algorithms are broadly classified into *model-free* learning and *model-based* learning as seen in section 2.3. Approaches that explicitly learn transition dynamics and/or reward

dynamics of an environment are known as model-based learning [?,?]. The learned transition and reward dynamics can then be used to find the optimal policy using, for example, policy iteration or value iteration [?] which are often referred to as *planners*. In contrast, Strehl et al. [?] presented a model-free algorithm wherein the agent directly learns the value function and obtains the optimal policy. In this paper, we focus on model-based approaches and use GP regression to learn the transition dynamics.

Rasmussen and Kuss [?] showed how to use GPs for carrying out model-based RL. They assumed that the reward dynamics are known and the transition and value function was modeled as a GP. We use the same assumption for ease of exposition. However, assuming reward dynamics to be known is not a critical requirement. In fact, reward dynamics can also be easily be modeled as GPs.

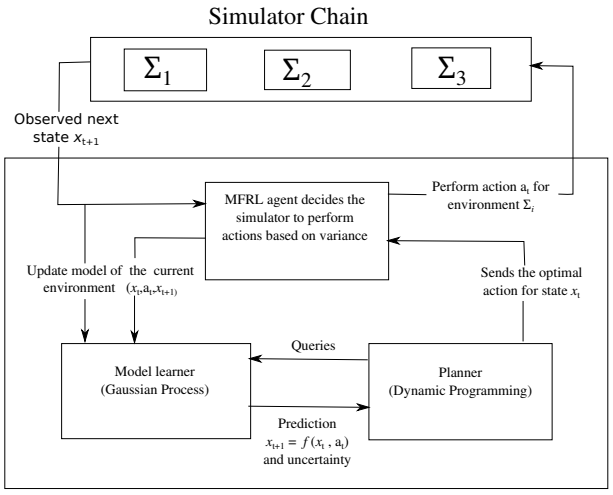


Figure 4.2: Overview of the GP-MFRL algorithm

We observe a number of transitions: $\mathcal{D} = \{(\mathbf{x}_t, a_t, \mathbf{x}_{t+1})\}$. Let $\mathbf{x}_{t+1} = f(\mathbf{x}_t, a_t)$ be the (unknown) transition function that must be learned. Our goal is to learn an estimate $\tilde{f}(\mathbf{x}, a)$ of $f(\mathbf{x}, a)$ in as few samples in $\mathcal{D}$ as possible. We can then use this estimated $\tilde{f}$ for unvisited state-action pairs (in place of f) during value iteration to learn the optimal policy. f can also be a stochastic transition function, in which case, the GP estimate gives the mean and the variance of this noisy transition function. For a given state-action pair (s, a), the estimated transition function is defined by a normal distribution with mean and variance given by:

$$\mu_{(s,a)|\mathcal{D}} = \mathcal{K}_{(s,a)\mathcal{D}} \mathcal{K}_{\mathcal{D}\mathcal{D}}^{-1} \vec{\mathcal{X}}_{\mathcal{D}} \tag{4.1}$$

$$\sigma^2_{(s,a)|\mathcal{D}} = \mathcal{K}\{(s,a),(s,a)\} - \mathcal{K}_{(s,a)\mathcal{D}} \mathcal{K}_{\mathcal{D}\mathcal{D}}^{-1} \mathcal{K}_{(s,a)\mathcal{D}} \tag{4.2}$$

where $\mathcal{K}$ is the kernel function.

GP regression requires a kernel which encodes the correlation between the values of f at two points in the state-action space. Choosing a right kernel is the most crucial step in

implementing GP regression. We choose the Radial Basis Function (RBF) kernel for our implementation since it models the spatial correlation we expect to see in an aerial robot system well. However, any appropriate kernel can be used in our algorithm depending on the environment to be modeled.

RBF has infinite dimensional feature space and satisfies the Lipschitz smoothness assumption. It can be defined as follows: for two points $\mathbf{x}$ and $\mathbf{x'}$,

$$k(\mathbf{x}, \mathbf{x'}) = \exp\left(- \frac{||\mathbf{x} - \mathbf{x'}||^2}{2\sigma^2} \right) \tag{4.3}$$

where $||\mathbf{x} - \mathbf{x'}||^2$ is the squared Euclidean distance and σ is a hyperparameter for the kernel often known as *characteristic length-scale*. Here, $\mathbf{x}$ represents a point in the joint state-action space.

Instead of using GPs to predict the next state, we use it to predict the velocity with which the robot will move when a given action a_t is applied at a state s_t. Learning the velocity vector helps in transitioning between simulators as the size of the state space itself may be different. For example, one can construct a multi-fidelity simulator where Σ_0 is a $n \times n$ grid, Σ_1 is a denser $2n \times 2n$ grid, and so on. An action in Σ_0 moves the robot one unit whereas the same action in Σ_1 moves the robot only 0.5 units. By learning the velocity instead of the next state, we can scale the learned velocity function to easily compute the transition function in any Σ_i as:

$$\vec{\mathcal{V}}\ (s_t, a_t) = \frac{s_{t+1} - s_t}{\Delta_i} \tag{4.4}$$

where Δ_i is the time scaling of a simulator. If the state spaces of all simulators are the same, then one can use GPs to predict the next state instead of the velocity vector.

We train two GP regressions, $f_x, f_y : \mathbb{R}^4 \to \mathbb{R}$, assuming independence between the two output dimensions. Let (x_i, y_i) be the current state of the agent. Actions actions are represented using a tuple (a_x, a_y) where a_x and a_y can take the values between 0 or 1.

The GP prediction is used to determine the transitions, $(x_i, y_i, a_x) \to x_{i+1}$ and $(x_i, y_i, a_y) \to y_{i+1}$ where (x_{i+1}, y_{i+1}) is the predicted next state with variance σ_x and σ_y respectively. Value of hyperparameters is estimated by gradient descent by optimizing the maximum likelihood estimate of a training data set.

4.3 GP-MFRL Algorithm

Using multiple approximations of real world environments has previously been considered in the literature [**?**, **?**]. Cutler et al. used model-based R-Max algorithm to reduce the number of samples using MFRL framework [**?**]. We use GP regression to further bring down the empirical sample complexity of MFRL framework.

Algorithm 2 gives the details of the proposed framework. As illustrated in Figure 4.2, there are two main components of GP-MFRL: (1) Model Learner; and (2) Planner. The model learner in our case is the GP-regression described in the previous subsection. We use value iteration [**?**] as our planner to calculate the optimal policy on learned dynamics of environment.

An *epoch* measures the time span between two consecutive switches in the simulators. Before executing an action, the agent checks (Step 4) if it has a sufficiently accurate estimate of the transition dynamics for the current state-action pair in the lower fidelity simulator, Σ_{d-1}. If not, it switches to Σ_{d-1} and executes the action in the potentially less expensive environment. The function ρ^{-1} checks if the current state is also a valid state in the lower fidelity simulator.

We also keep track of the variance of the $\mathcal{L}$ most recently visited state-action pairs in the current epoch. If the running sum of the variances is below a threshold (Step 8), this suggest that the robot has found a good policy in the current simulator and it must advance to the next higher fidelity simulator.

Steps 12–16 in describe the main body where the agent computes the optimal action, executes it, and records the observed transition in $\mathcal{D}$. The GP model is updated after every n_U iterations (Step 17). In the update, we recompute the hyper-parameters until they converge.

A new policy is computed every time the robot reaches the goal state (Step 21). If the robot is in the highest fidelity simulator, we also check if the policy has converged by checking if the maximum change in the value function is less than a threshold (Step 22). If so, we terminate the learner.

4.4 Simulation Results

We demonstrate the GP-MFRL algorithm in a simulator chain consisting of a virtual grid-world environment and the Gazebo robot simulator [**?**]. The setup is shown in Figure 4.3. The simple grid-world agent operates in a 21×21 grid. The agent receives a reward of $+50$ at the goal location, and -1 for all other states. If the agent hits the obstacles, it gets the reward of -20. In each time step, an agent can move in one of the four directions viz. up, down, left and right. We add a Gaussian noise of σ to the actual transition to represent stochastic environments. The Gazebo simulation setup consists of a quadrotor with PX4 autopilot running in software–in–the–loop (SITL) mode. The PX4 SITL interfaces with the Robot Operating System [**?**] via the `mavros` node.

The code is written in python and uses scikit-learn [**?**] to implement GP-regression. The code is available online at `https://github.com/raaslab/gp_gazebo`.

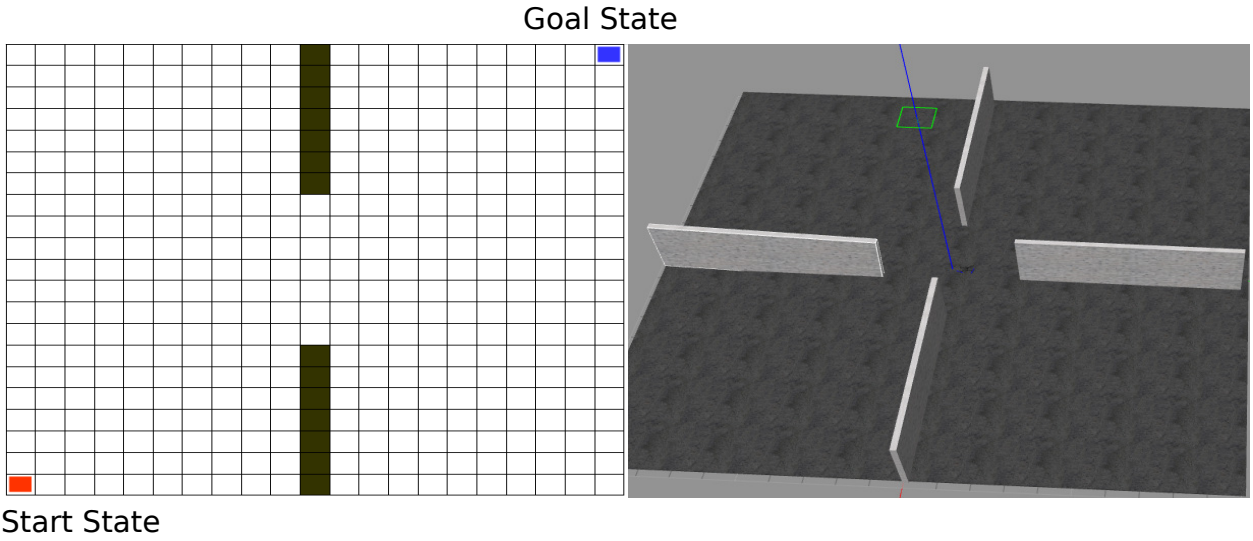


Figure 4.3: The environment setup for a multi-fidelity simulator chain. The simple gridworld environment has two wall obstacles whereas the gazebo environment has four wall obstacles as shown.

Figure 4.4 shows the switching between the simulators for one run of the GP-MFRL algorithm on the environment shown in Figure 4.3. It can be seen that the agent switches back and forth between the two simulators unlike unidirectional transfer learning algorithms. In the rest of the simulations we study the effect of the parameters used in GP-MFRL and the fidelity of the simulators on the number of samples till convergence.

Input : A simulator chain,
Confidence parameter ψ for (s, a),
History Length $\mathcal{L}$,
Confidence Ψ,
State mapping ρ,
Reward dynamics $\mathcal{R}^a_{ss'}$
Update rate n_U
Initialize: Transition dynamics $\mathcal{P}^a_{ss'}$;
$d = 1$;
$\mathcal{V}^*_d(s) \leftarrow$ `Planner` $(\mathcal{P}^a_{ss'})$

```
Learner()
```
while true **do**
  $a^*_t \leftarrow \text{argmax}_a \ \mathcal{V}^*_d(s_t)$;
  if $\sigma(\rho^{-1}(s_t, a^*_t)) \geq \psi \ \bigwedge \ d > 1$ **then** `// Return to level` $d-1$
    $d \leftarrow d - 1$;
    `epochLength` $\leftarrow 0$
  end
  if $\sum_{i=t-\mathcal{L}}^{t-1} \sigma(s_i, a^*_i) \leq \Psi \ \bigwedge$ `epochLength` $\geq \mathcal{L}$ **then**
    $d \leftarrow d + 1$ (Move up the simulator) ;
    `epochLength` $\leftarrow 0$;
  end
  $a^*_t \leftarrow \text{argmax}_a \ \mathcal{V}^*_d(s_t)$;
  Execute a^*_t and store observed s_{t+1}, r_{t+1} ;
  `epochLength` ++ ;
  $\mathcal{D}_t = \mathcal{D}_t \cup (s_t, \ a^*_t, \ s_{t+1})$;
  $s_t \leftarrow s_{t+1}$;
  if `epochLength` is multiple of n_U **then**
    $\mathcal{P}^a_{ss'} \leftarrow$ `UpdateGP`$(\mathcal{D}_t)$;
  end
  if s_t is Goal state **then**
    $\mathcal{V}_f(s) \leftarrow$ `Planner`$(\mathcal{P}^a_{ss'})$;
    if $\max_s \ \mathcal{V}_f(s) - \mathcal{V}_0(s) \leq 10\ \% \bigwedge d == D$ **then**
      break the loop ;
    end
    $\mathcal{V}_0(s) \leftarrow \mathcal{V}_f$;
  end
  $t \leftarrow t + 1$;
end

```
Planner()
```
Initialize: $\mathcal{V}(s) = 0, \ \forall (s, a)$
$\Delta = \infty$
while $\Delta > 0.1$ **do**
  for *every s:* **do**
    $temp \leftarrow \mathcal{V}(s)$;
    $\mathcal{V}(s) \leftarrow \max_a \sum_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma \mathcal{V}(s')]$;
    $\Delta \leftarrow \max(0, \ |temp - \mathcal{V}(s)|)$
  end
end
return $\mathcal{Q}(s, a)$

Algorithm 2: GP-MFRL Algorithm

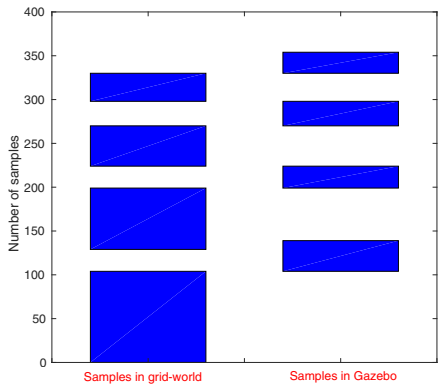
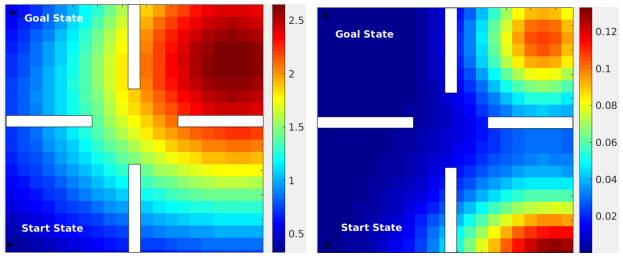


Figure 4.4: The figure represents the samples collected in each level of simulator for a 21×21 grid in a simple grid-world and Gazebo environments. Ψ and ψ were kept 0.4 and 0.1



(a) After Initial Training (b) After Convergence

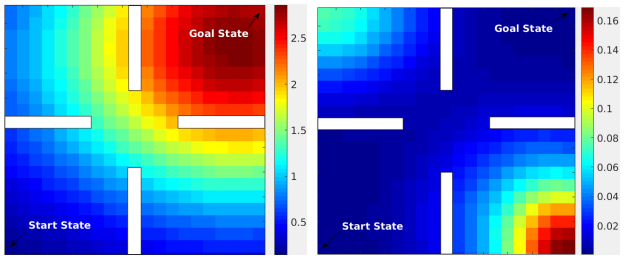
Figure 4.5: Variance plot for 21×21 multi-fidelity environment after transition dynamics initialization and after algorithm has converged

4.4.1 Representative simulations

We first present three representative scenarios to observe the qualitative performance of the GP-MFRL algorithm. Specifically, we consider three instances and show how the variance evolves over time as more samples are collected. Recall that the main advantage with using GPs is that it allows for quick generalization of observed samples to unobserved state-action pairs.

To demonstrate how variance of the predicted transition dynamics varies from the beginning of experiment to convergence, we plot "heatmaps" of the variance. The GP prediction for a state-action pair also gives the variance, σ_x and σ_y, respectively for the predicted state. The heatmap shows $\sqrt{\sigma_x^2 + \sigma_y^2}$ for the optimal action at every state as returned by the Planner.

Figures 4.5 and 4.6 show the heatmaps at the start and convergence for the same environment but with different start and goal positions. As expected, the variance along the optimal (*i.e.*, likely) path is low whereas the variance for states unlike to be on the optimal path from start to goal remains high.



(a) After Initial Training (b) After Convergence

Figure 4.6: Variance plot for 21 × 21 multi-fidelity environment after transition dynamics initialization and after algorithm has converged

A more interesting case is presented in Figure 4.7. Even though there's a path available to reach the goal from the right of wall A, the agent explores that region less than the region near the walls B, C and D (indicated in dark blue showing less variance). This is due to the fact that, the transition dynamics learned in the lower fidelity simulator is used in the higher fidelity simulator leading to lesser exploration of the regions which are not along the optimal path.

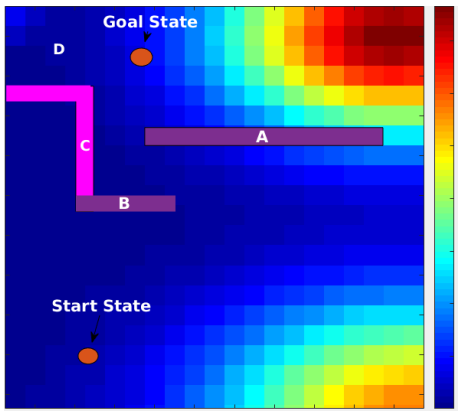


Figure 4.7: Variance plot for 21 × 21 multi-fidelity environment after the algorithm has converged. Walls A and B are only present in the grid-world simulator, whereas all four walls are present in the Gazebo simulator.

4.4.2 Effect of fidelity on the number of samples.

We first study the effect of varying fidelity on the total number of samples and the fraction of the samples collected in the higher fidelity simulator. Our hypothesis is that having learned the transition dynamics in the gridworld, the agent will need fewer samples in the higher fidelity Gazebo simulator to find the optimal policy. However, as the fidelity of the first

simulator decreases, we would need more samples in Gazebo.

In order to validate this hypothesis, we varied the noise parameter used to simulate the transitions in the gridworld. The transition model in Gazebo remains the same. The total number of samples collected increases as we increase the noise in gridworld (Figure 4.8). As we increase the noise in the first simulator, the agent learns less accurate transition dynamics leading to collection of more number of samples in the higher fidelity simulator. Not only does the agent need more samples, the ratio of the samples collected in the higher fidelity simulator to the total number of samples also increases (Figure 4.9).

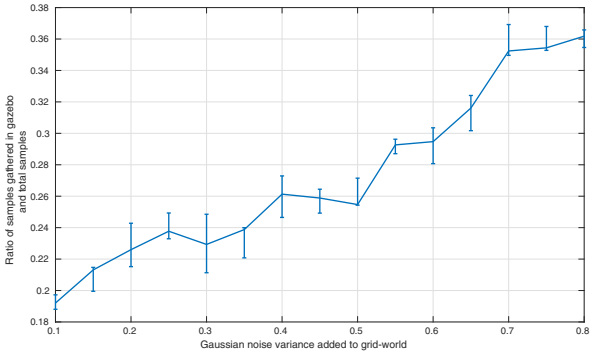


Figure 4.8: As we make first simulator more inaccurate by adding noise, the agent tends to gather more samples in second simulator

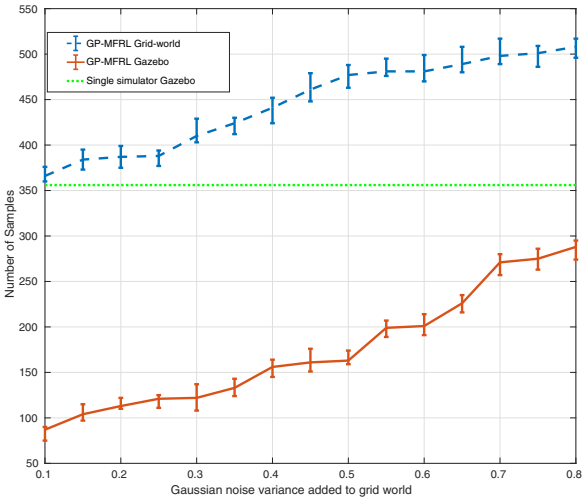


Figure 4.9: Ratio of samples gathered in the second simulator to the total samples gathered increases with inaccuracy in the first simulator. The reference line depicts the average number of samples gathered over 10 runs when only Gazebo simulator was present.

4.4.3 Effect of the confidence parameters.

The GP-MFRL algorithm uses two confidence parameters, ψ and Ψ, which are compared against the variance in the transition dynamics to switch to a lower and higher simulator, respectively. Figure 4.10 shows the effect of varying the two parameters on the ratio of number of samples gathered in the Gazebo simulator to the total number of samples. As expected, increasing ψ or decreasing Ψ leads to more samples being collected in the higher fidelity simulator.

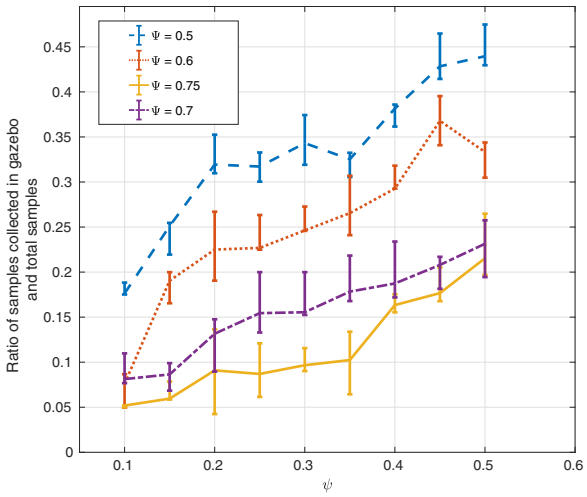


Figure 4.10: Ratio of samples gathered in second simulator vs. total samples gathered as we change the threshold or confidence parameters of the two simulators.

4.4.4 Comparison with R-max MFRL

Figure 4.11 shows the comparison between performance of GP-MFRL algorithm with the existing MFRL algorithm [**?**], GP-MFRL algorithm only in the highest fidelity simulator and Rmax algorithm running only in the highest fidelity simulator. The experiments are performed in the environment same as the one used in Figure 4.7. As expected, the GP-MFRL algorithm performs better than the existing MFRL algorithm, [**?**].

4.5 Conclusion

The GP-MFRL algorithm provides a general RL technique that is particularly suited for robotics. An extension to the existing work would be implementing the GP-MFRL algorithm on an actual quadrotor as the highest-fidelity simulator to demonstrate the utility of GP-

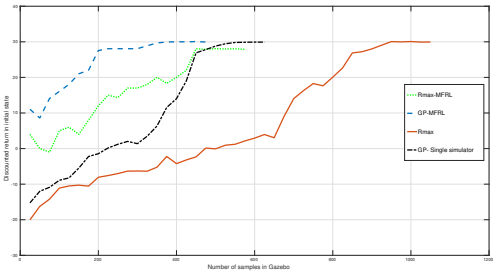


Figure 4.11: Discounted return in the start state Vs. the number of samples collected in the highest fidelity simulator.

MFRL. In this thesis, it is shown empirically that, GP-MFRL finds optimal policies using fewer samples than MFRL algorithm.

Chapter 5

Bridge Inspection