Reinforcement Learning with Gaussian Processes

Neha Prasad
MIT Department of Computer Science
nehap@mit.edu

Soumya Ram
MIT Department of Computer Science
soumyar@mit.edu

1. Introduction

In the reinforcement learning problem, an agent acts in an unknown or incompletely known environment with the goal of maximizing an external reward signal. The problem with many existing reinforcement learning algorithms is that they are not *sample efficient*, meaning that they fail to learn anything useful from many samples and do not improve rapidly. To this end, several new algorithms have been proposed, including GP-Rmax [JS12]and C-PACE [PP13]. While these algorithms are sample efficient, they require a costly fixed computation, making them ill-suited for certain real-world agents such as aircraft. The paper [GWH14] proposes a series of PAC-MDP (defined in the next section) results for using Gaussian Processes with Reinforcement Learning. The paper demonstrates that because Gaussian Processes are sample-efficient and quantify uncertainty, they can be useful for developing robust planning models from a small number of examples.

In addition to these theoretical results, the paper provides a model-free PAC-MDP reinforcement learning algorithm, in which a Gaussian Process is used to directly model a Q-function. The authors coin the term Delayed-GPQ (DGPQ) for the model, because unlike previous attempts the model stores a GP for each possible action, and only updates a separate Q-value function when sufficient outlier data has been detected. This operation overwrites a portion of the stored value function and resets the GP confidence bounds, avoiding the slowed convergence rate of the naive model-free approach, which can take up to an exponential number of steps to reach a near-optimal policy.

In this project, we implemented the DGPQ algorithm described in the paper from scratch. Moreover, instead of using pre-existing GP libraries, we created our own. We then tested this algorithm on many different environments: a simple 2D-Square, real-world Robotics simulations in the OpenAI Gym, and a financial application. In addition, we experimented with a different choice of kernel from [CPRD14], to improve the Gaussian Process performance. All of our code for the project can be found in https://github.com/nehap25/rlwithgp.git.

2. Background and Terminology

Reinforcement learning can be modelled as a Markov Decision Process with the following variables *(S,A,R,T,γ)*.

S represents the set of possible states. This can be infinite, but is finite for the settings we explore below. A represents the set of actions. This can also be infinite, but we restrict this set to 4 in the experiments below. Taking an action in a particular state yields another state. This is modelled through the transition function, T. $T(s, a, s')$ yields the probability of ending up in s' through taking action a in state s. R is the reward function which takes in (s, a) as input and outputs a reward between $[0, R_{max}]$. A reinforcement learning environment has a goal. The reward function will give a reward of R_{max} for the goal state. Other states will receive a reward based on how close they are to the goal state.γ represents the discount factor for a reward over the course of one step.

For the purposes of planning, it is helpful to introduce a Q-function. The optimal Q-function can be represented as $Q^*(s, a) = R(s, a) + \gamma \int_{s'} T(s, a, s')V^*(s')$. $V^*(s')$ is defined as $max_a Q^*(s, a)$. Thus, $V^*(s)$ represents the optimal policy, or the best action to take in each state. Because the reward function is bounded by R_{max}, Q and V are bounded also.

There are two main approaches to reinforcement learning, model-free and model-based. Model-based approaches build models of T and R, and use these models to find Q^*. Compare to model-free approaches, model-based approaches are more sample-efficient.

Model-free approaches learn Q^* directly from experience, without explicitly generating T and R functions. As explained above, though, they need significantly more samples.

To better delineate this difference, we define a common notion of sample-complexity below.

We define the covering number $N_U(r)$of a domain U to be the cardinality of a minimum set such that for all elements x in the domain U, there exists a c_i in the minimum set such that $d(x, c_i) < r$ for some distance metric.

We define a notion of probably-approximately correct (PAC-MDP) in ϵ and δ if, with probability $1 - \delta$, it takes a

Algorithm 1 Delayed GPQ (DGPQ)

1: **Input:** GP kernel $k(\cdot,\cdot)$, Lipschitz Constant L_Q, Environment Env, Actions A, initial state s_0, discount γ, threshold σ^2_{tol}, ϵ_1
2: **for** $a \in A$ **do**
3: $\quad \hat{Q}_a = \emptyset$
4: $\quad GP_a = GP.\text{init}(\mu = \frac{R_{max}}{1-\gamma}, k(\cdot,\cdot))$
5: **for** each timestep t **do**
6: $\quad a_t = \arg\max_a \hat{Q}_a(s_t)$ by Eq 7
7: $\quad \langle r_t, s_{t+1}\rangle = Env.\text{takeAct}(a_t)$
8: $\quad q_t = r_t + \gamma \max_a \hat{Q}_a(s_{t+1})$
9: $\quad \sigma^2_1 = GP_{a_t}.\text{variance}(s_t)$
10: $\quad$ **if** $\sigma^2_1 > \sigma^2_{tol}$ **then**
11: $\quad\quad GP_{a_t}.\text{update}(s_t, q_t)$
12: $\quad \sigma^2_2 = GP_{a_t}.\text{variance}(s_t)$
13: $\quad$ **if** $\sigma^2_1 > \sigma^2_{tol} \geq \sigma^2_2$ **and** $\hat{Q}_{a_t}(s_t) - GP_{a_t}.\text{mean}(s_t) > 2\epsilon_1$ **then**
14: $\quad\quad \hat{Q}_a.\text{update}(s_t, GP_{a_t}.\text{mean}(s_t) + \epsilon_1)$
15: $\quad\quad \forall a \in A, GP_a = GP.\text{init}(\mu = \hat{Q}_a, k(\cdot,\cdot))$

Figure 1: Delayed GPQ Algorithm

polynomial (in $N_{SA}(r)$ number of steps for the value function to be within ϵ of the optimal value function, for all states.

3. Method for Model-Free RL with a GP

Model-free RL algorithms, such as Q-learning, are often used when planning is infeasible due to time constraints or the size of the state space. These algorithms do not store the state transition function, but instead try to model Q^* directly using incremental updates. Unfortunately, many of these algorithms have provably exponential sample complexity. However, the most recent Delayed Q-learning (DQL) is PAC-MDP [SLL09]. This algorithm works by initializing $Q(s,a)$ to V_{MAX} and then overwriting $Q(s,a)$ after m samples of that $\langle s,a\rangle$ pair have been seen. The authors in [GWH14], inspired by this approach, proposed a new algorithm that guarantees polynomial sample efficiency while also using the Gaussian Process.

Delayed GPQ-Learning (DGPQ: Algorithm 1) maintains two representations of the value function. The first is a set of GPs, GP_a for each action. The second representation is the value function, which stores values from previously converged GPs and is denoted by $\hat{Q}(s,a)$. Intuitively, the algorithm only updates $\hat{Q}(s,a)$ when the mean of GP_a converges to a significantly different value at s, and the variance at GP_a crosses below some σ^2_{tol}.

The algorithm chooses actions greedily based on $\hat{Q}$ (line 6) and updates the corresponding GP_a based on the observed rewards and the $\hat{Q}$ at the next state (line 8). If the GP has just crossed the convergence threshold at point s and learned a value significantly lower (measured by some threshold $(2\epsilon_1)$ than the current value of $\hat{Q}$ (line 13), the representation of $\hat{Q}$ is partially overwritten, using an operation described below. This reinitializes the variance so that the condition in line 13 is satisfied fairly infrequently, thereby avoiding the slow convergence. Determining σ^2_{tol} and ϵ_1 is described in Section 5 of the paper.

Upon initial consideration, it seems reasonable to use yet another GP to model $\hat{Q}(s,a)$. However, it is difficult to guarantee the optimism of $\hat{Q}(s,a)$ ($\hat{Q} \geq Q^* - \epsilon$) with a GP, which is a necessary property of sample-efficient algorithms. In order to address this issue, the paper describes a function approximator for $\hat{Q}$, specifically it is stored using a set of values that have been updated from the set of GPs, $(\langle s_i, a_i\rangle, \hat{\mu}_i)$ and a lipschitz constant L_Q that is used to find an optimistic upper bound for the Q function for $\langle s,a\rangle$:

$$\hat{Q}(s,a) = \min\left\{ \min_{\langle s_i,a\rangle \in BV} \mu_i + L_Q d((s,a),(s_i,a)), V_{MAX} \right\}$$

We refer to the set (s_i,a) as the basis vectors (BV). Intuitively, the basis vectors store values from the previously learned GPs. Around these points, Q^* cannot be greater than $\hat{\mu}_i + L_Q d((s,a),(s_i,a))$ by continuity. To predict optimistically over the points that are not in the BV, we search over BV for the point with the lowest prediction including the weighted distance bonus. If no point in BV is sufficiently close, then V_{MAX} is used instead. Note that the GP plays a critical in Algorithm 1 because its confidence bounds determine when $\hat{Q}$ gets updated.

In order to perform an update (partial overwrite) of $\hat{Q}$, we first an element $\langle (s_i,a_i), \mu_i\rangle$ to the basis vector set. Then, if for any j, $\mu_i + L_Q d((s_i,a),(s_j,a)) \leq \mu_j$, we delete $\langle (s_j,a), \mu_j\rangle$. The purpose of this step is to eliminate redundant basis vectors from the set.

3.1. Kernel

There are some limitations with using a Gaussian Process. Two states could have a very small Euclidean distance but could have very differing Q-values. One example is if one state is on the table and the other is not. To account for this complexity, we implemented the kernel described in *Manifold Gaussian Processes for Regression*

The paper provided a suggestion to use neural networks to transform the inputs to a more rich feature space. Thus, the Gaussian Process could be applied on this feature space, circumventing the problem above. The neural network was trained by stochastic gradient descent to minimize the negative log likelihood.

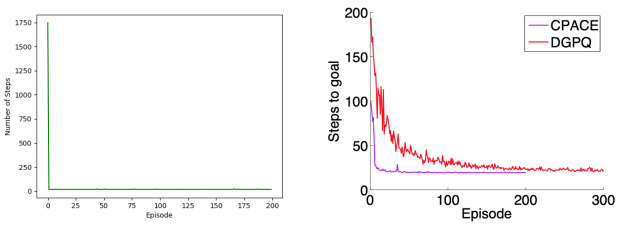

Figure 2: Number of steps to goal per episode for our implementation (left) and the paper implementation (DGPQ in right).

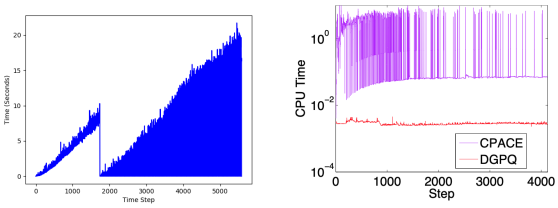

Figure 3: Amount of time per time step for our implementation (left) and the paper implementation (DGPQ in right).

4. Experimental Results

4.1. 2-D Square Environment

Our first experiment is also the first experiment in [GWH14]. Here, an agent moves in a 2-dimensional square $[0, 1]^2$. The agent starts at $[0, 0]$ with a goal of reaching within 0.15 of $[1, 1]$. Movements are 0.1 in the four compass directions with additive uniform noise of ± 0.01. We used an L_1 distance metric and a lipschitz constant $L_Q = 9$. For the GP, we used an RBF Kernel with $\theta = 0.05$ as well as a measurement noise variance of 0.1. It is important to note that at the start of each episode, we reinitialized the state to $[0, 0]$ and the episode rewards to 0. But, the Q function (represented as a dictionary in our code with key $\langle s_i, a_i \rangle$ and value $\hat{\mu_i}$) and GPs for the actions are not reinitialized, enabling the algorithm to learn the optimal set of actions in fewer timesteps in subsequent episodes.

The reward function specifications and the accuracy parameters ϵ and δ for the algorithm were not specified in the paper, and so we had to use some trial and error to determine what made the most sense. For a given state with x-coordinate s_0 and y-coordinate s_1, we defined the reward function to be $2 - ((1 - s_0)^2 + (1 - s_1)^2)^{\frac{1}{2}}$. Therefore, $R_{max} = 2$ and because we capped the number of timesteps per episode to be 200, we set $V_{max} = R_{max} \times 200 = 400$. For the discount parameter γ, as well as the accuracy parameters ϵ and δ, we chose these to maximize σ^2_{tol}, because otherwise the algorithm would require many many steps to satisfy the line 13 condition in Figure 1, and subsequently converge slower. To that end, we found that setting $\epsilon = 1$, $\gamma = 0.01$, and $\delta = 0.96$ maximized σ^2_{tol} (where we set σ^2_{tol} and ϵ_1 in the algorithm using Section 5 of [GWH14]).

In Figure 2, we compare the number of steps taken to reach the goal of 0.15 within $[1, 1]$ for both our implementation and the paper's implementation. We see that for both the number of steps converged to approximately 20 steps per episode very quickly. For ours, this happened right after the first episode, while for the paper's it was slightly more gradual, after 2 or 3 episodes. The other main difference between our implmentation and theirs is that in the first episode, ours takes about 1750 steps to converge, while theirs takes only 100. We suspect that this is in part due to a difference in the γ, ϵ, δ parameters or reward function specifications, but more importantly it could also be due to a difference in GP mean, variance, and update calcaulations as the paper relies on an online sparse GP approximation while we coded ours from scratch with no approximation. In addition, for our first episode, the number of steps it takes to converge is highly variable (ranging from 1000-5000 based on our different runs), and this changes based on the actions that get stored in the Q-function once the GP's confidence bounds are within a certain threshold.

In Figure 3, we compare time it takes per timestep for our implementation versus the paper's. Since it wasn't clear what "CPU Time" was in the paper, we instead used time.clock() from Python's time library. For DGPQ, the average amount of CPU time per timestep is $0.003s$, while for ours it was much higher and more variable. We believe that this is because the paper relied on an online Sparse GP approximation, which is presumably faster than our GP class. More interestingly, our time graph portrays how the algorithm works on a programmatic level. In particular, we see that time per timestep increases up till timestep 1750. This is because up till this timestep, the algorithm is observing more and more $\langle s_t, q_t \rangle$ samples, and as it does so the variance of the GP decreases. Once we reach timestep 1750, the timestep decreases so much so that the variance is below the threshold variance and the condition on line 13 is satisfied. From here, the most recent sample is stored in the Q-function and the GP's are reset. Now that the sample is stored in the Q-function, the algorithm learns the optimal set of states much faster, which is why we see that drop in time after timestep 1750. The algorithm can effectively rely on this Q-function to learn the optimal set of states for all subsequent episodes as well, which is why the condition on line 13 never gets satisfied after timestep 1750 - we reach the goal well before that is necessary for each episode. At the same time, the time per timestep increases because we are observing more and more samples and the GP steps take longer as a result.

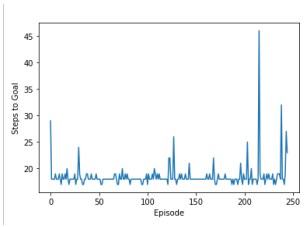

Figure 4: Number of steps to goal per episode for neural network-based kernel

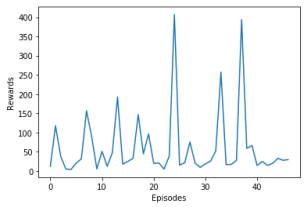

Figure 5: Rewards per episode in robotics experiment

4.2. Kernel Experiment

We applied the kernel described in 3.1 to the environment described above. The plot of episodes versus number of steps can be found in Figure 4.

We replace the standard RBF kernel with a neural-network based kernel detailed in section 3.1. The neural network was chosen to be one linear layer with 10 hidden units.

The neural network-RBF kernel is more unstable than the RBF kernel alone. However, it shows significantly better performance. The baseline converges to an average of 25 steps after 100 or so episodes. However, the neural network-based kernel is able to achieve this performance after only two iterations, where it takes 18 steps.

This is because the neural network allows the state to be elevated to a much richer feature space of dimension 10. This significantly helps with convergence.

4.3. Robotics Environments

Using pybullet, we created a robotics environment. A block was placed on the edge of a table with a robot behind it. The robot's goal was to move the block to a goal location.

4.3.1 States

The state is a length six vector consisting of the 3D position of the robot and the 3D position of the block.

4.3.2 Actions

The possible actions were the same as in the previous experiment. The robot could move a distance of 0.1 in the north,south,east, and west direction.

4.3.3 Reward

A goal position was specified at the end of the table. This position had reward 2. All other positions had a reward of 2 minus the euclidean distance to the goal.

4.3.4 Simulation Environment

The simulation environment was pybullet. Code for this environment was written for a previous undergraduate research project.

4.3.5 Method and Results

The algorithm used was the same as the one in the previous experiment. All of the algorithm's hyperparameters were the same as the previous experiment, and the kernel used was identical to the kernel described in section 4.2. The results of the experiment can be found in figure 5. The algorithm was able to achieve the goal in three different episodes, but its performance was very spiky. The likely cause of this is the neural network overfitting the small number of data points it has.

4.4. Modeling Optimal Execution of Portfolio Transactions

In Finance, Reinforcement Learning can be used for optimizing the execution of large portfolio transactions. The optimal liquidation problem is a minimization problem, i.e. we need to find the trading list that minimizes the implementation shortfall, which is the difference between the decision price and the final execution price (including commissions, taxes, etc.). In order to frame this as a Reinforcement Learning problem, we first give an overview of the states, actions, rewards, and environment. We rely on the original Almgren-Chriss paper [AC00].

4.4.1 States

At time t_k, the state vector is the following:

$$[r_{k-5}, r_{k-4}, r_{k-3}, r_{k-2}, r_{k-1}, r_k, m_k, i_k]$$

where:

- $r_k = \log(\frac{s_k}{s_{k-1}})$ is the log-return at time t_k.
- $m_k = \frac{N_k}{N}$ is the number of trades remaining at time t_k normalized by the total number of trades.

- $i_k = \frac{x_k}{X}$ is the remaining number of shares at time t_k normalized by the total number of shares.

The log-returns capture information about stock prices before t_k, which can be used to detect price trends. The number of trades and shares remaining allow the agent to learn to sell all the shares within a given time frame.

4.4.2 Actions

The action a_k is the percentage of remaining shares to sell at time t_k. Supposing there are x_k shares remaining at time t_k, then the total number of shares to sell would be $n_k = a_k \times x_k$.

4.4.3 Rewards

The utility function in the Almgren-Chriss model is

$$E(x) + \lambda V(x)$$

where $E(x)$ and $V(x)$ are the expectation and variance of the implmentation shortfall, for various values of λ, which is the trader's risk aversion. For this implementation, we set λ to be a constant. Denoting the optimal trajectory at time t as x_t^* (where x_t is the number of remaining shares), we define the reward as:

$$R_t = \frac{U_t(x_t^*) - U_{t+1}(x_{t+1}^*)}{U_t(x_t^*)}$$

4.4.4 Simulation Environment

We use a very simple simulated trading environment from Udacity's Deep-Reinforcement-Learning Finance library (https://github.com/udacity/deep-reinforcement-learning/tree/master/finance). The environment simulates stock prices that follow a discrete arithmetic random walk and that the permanent and temporary market functions are linear functions of the rate of trading, just like in the Almgren and Chriss Model. [AC00]

4.4.5 Method and Results

For this setting, our method has an $R_{MAX} = 1$ and because we are performing 60 trades per episode, we set $V_{MAX} = 60 \times R_{MAX} = 60$. We set λ, the trader's risk aversion, to 10^{-6} and the lipschitz constant $L_Q = 9$. For the GPs, we used an RBF Kernel with $\theta = 0.05$ as well as a measurement noise variance of 0.144. In this framework, the actions are considered percentages, and because we need to define a GP for each action, we represented the percentages as rounded to two decimal places. Finally, each action is coupled with a noise sampled from an Ornstein-Uhlenbeck process with $\mu = 0$, $\theta = 0.15$, $\sigma = 0.2$. In addition, we experimented with two different choices of discount γ parameters: 0.01 and 0.99. By using the same strategy as in Section 4.1, for the Reinforcement Learning model with Gaussian Processes we experiment with ϵ and δ parameters that maximize σ_{tol}^2. In particular, for $\gamma = 0.01$, we set $\epsilon = 1$ and $\delta = 0.99$; for $\gamma = 0.99$, we set $\epsilon = 99$ and $\delta = 0.99$.

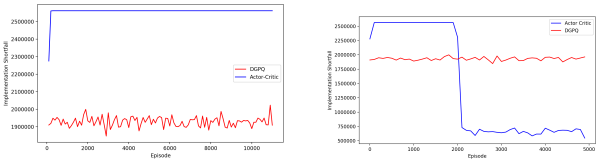

Figure 6: Average Implementation Shortfall of every 100 Episodes for DGPQ and the Actor-Critic Model with Discount $\gamma = 0.01$ (left) and Discount $\gamma = 0.99$ (right).

The Udacity's deep-reinforcement-learning finance library contains a reinforcement learning implementation with the Almgren-Chriss framework using an Actor-Critic Model, specifically with Deep Deterministic Policy Gradients (DDPG). Using their model as a baseline, we plot the average implementation shortfall every 100 episodes (with 11, 000 episodes total) for $\gamma = 0.01$ and $\gamma = 0.99$. In Figure 6, we plot the average implementation shortfall of every 100 episodes for DGPQ and the Actor-Critic Model, with the two different discount values. For $\gamma = 0.99$, the Actor-Critic Model seems to perform much better, as it minimizes the implementation shortfall more, while at $\gamma = 0.01$, the DGPQ model seems to perform better. In terms of implementation shortfall, DGPQ seems to perform roughly the same for both discounts, but for time (Figure 7), the effect of the discount parameter becomes apparent. In particular, when the discount is 0.99 (Figure 7, right), the σ_{tol}^2 is extremely tiny (on the order of 10^{-7}), which means that the condition in line 13 does not get satisfied, the GPs never get reset, the Q-function never gets updated, and the time per episode increases roughy linearly as more and more samples get observed. This differs significantly from when the discount is 0.01, because here the σ_{tol}^2 is higher and the GPs get reset every 2000 episodes or so, hence the rises and drops in time per episode. This example highlights the importance of parameter tuning for the performance and convergence of DGPQ.

5. Conclusion

We implement the DGPQ algorithm for sample-efficient reinforcement learning and are able to reproduce the baselines. We find that this algorithm generalizes well to diverse domains, including robotics and finance.

We present an improvement on the DGPQ algorithm by using neural networks to transform the inputs to a new fea-

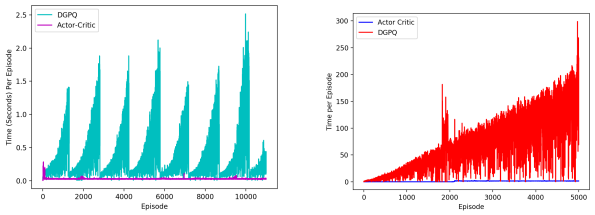

Figure 7: Time (Seconds) per Episode for DGPQ and the Actor-Critic Model with Discount $\gamma = 0.01$ (left) and Discount $\gamma = 0.99$ (right).

ture space, as detailed in another paper. We find that this improvement leads to significantly better performance and faster convergence.

5.1. Next Steps

The next step is to benchmark the neural network kernel against existing neural network based methods for reinforcement learning. We hypothesize that the neural network kernel may have better performance when only a limited number of samples is possible, as it has less parameters.

References

[AC00] Robert Almgren and Neil A. Chriss. Optimal execution of portfolio trans-actions. 2000.

[CPRD14] Roberto Calandra, Jan Peters, Carl Edward Rasmussen, and Marc Peter Deisenroth. Manifold gaussian processes for regression, 2014.

[GWH14] R.C. Grande, T.J. Walsh, and Jonathan How. Sample efficient reinforcement learning with gaussian processes. *31st International Conference on Machine Learning, ICML 2014*, 4:3136–3150, 01 2014.

[JS12] Tobias Jung and Peter Stone. Gaussian processes for sample efficient reinforcement learning with rmax-like exploration, 2012.

[PP13] Jason Pazis and Ronald. Parr. Pac optimal exploration in continuous space markov decision processes. *ro- ceedings of the AAAI Conference on Artificial Intelligence*, 2013.

[SLL09] Alexander Strehl, Lihong Li, and Michael Littman. Reinforcement learning in finite mdps: Pac analysis. *Journal of Machine Learning Research*, 10:2413–2444, 11 2009.