Policy Gradient Methods with Gaussian Process Modelling Acceleration

Dong Li*†, Dongbin Zhao*†, Qichao Zhang*†, Chaomin Luo‡
*The State Key Laboratory of Management and Control for Complex Systems
Institute of Automation, Chinese Academy of Sciences. Beijing, China
†University of Chinese Academy of Sciences, Beijing, China
‡Department of Electrical and Computer Engineering, University of Detroit Mercy, Detroit, USA
Email: dongleecsu@gmail.com, dongbin.zhao@ia.ac.cn, zhangqichao2013@163.com, luoch@udmercy.edu

***Abstract*—Policy gradient algorithm is often used to deal with the continuous control problems. But as a model-free algorithm, it suffers from the low data efficiency and long learning phase. In this paper, a policy gradient with Gaussian process modelling (PGGPM) algorithm is proposed to accelerate learning process. The system model is approximated by Gaussian process in an incremental way, which is used to explore state action space virtually by generating imaginary samples. Both the real and imaginary samples are used to train the actor and critic networks. Finally, we apply our algorithm to two experiments to verify that Gaussian process can accurately fit system model and the supplementary imaginary samples can speed up the learning phase.**

I. Introduction

Reinforcement learning (RL) [1] has been successfully applied to many challenging fileds like robot control [2] and optimal control [3] [4]. Recently, deep learning achieves state-of-the-art in many fields such as image classification [5], speech recognition [6] and natural language processing [7]. It utilizes deep structure to abstract different level features so as to obtain high-level feature representation capability. Although deep learning has a strong sensing ability, it lacks the decision making capability, which is reinforcement learning's strength. So combining reinforcement learning and deep learning has became a new trend and many achievements have been obtained, such as human-level video game play [8] [9] and attention networks for vehicle classification [10]. Like the classical reinforcement learning, deep reinforcement learning can also be divided into model-free methods and model-based methods. Utilizing deep neural network represented policies makes it possible to deal with complex tasks with minimal human-made features. However, the sample complexity for model-free methods is very high, especially when the neural network structure is deep. In other words, tremendous samples are needed if we want to use high dimensional deep neural network as a function approximator. In contrast, model-based reinforcement learning can improve data efficiency, but sometimes learning a good system dynamics itself is a difficult task. Moreover, once we have fitted a system model, the model-based reinforcement learning would limit the performance to that model, which means that if the learned model itself is not good, then the model-based policy is not good as well. Hence, it would be desirable if we can combine the strength of both model-free and model-based reinforcement learning. This can be realized by employing Gaussian process.

This work is supported by National Natural Science Foundation of China (NSFC) under Grants No. 61573353, No. 61533017 and No. 61603382.

Gaussian process [11], which is a generalization of the Gaussian distribution is a stochastic process that can deal with functions. One remarkable advantage of Gaussian process over other non-Bayesian models is that it explicitly expresses the model from a probabilistic perspective, which means it can not only represent the mean value of the target output, but can describe the uncertainty of the estimation. This property makes Gaussian process a good approximator for reinforcement learning. Xia et al. [12] propose an online Bayesian SARSA algorithm in which Gaussian process is used to approximate the value function. Engel et al present a Gaussian process SARSA algorithm in [13]. Besides the application on reinforcement learning field, there are also many modelling works by using Gaussian process. Engel [14] et al. propose a recursive implementation based on kernel recursive least squares (KRLS) algorithm, which attempts to estimate the target output by updating their hyper-parameters when a new data is collected, typically by minimizing a least square cost function. However, the growth of memory and computational complexities increase greatly when more and more samples are collected, so a data sparsification mechanism is needed. To this end, sparse kernel methods are proposed [14], by which we can approximately represent the target using only a subset of bases. Moreover, in order to track the system dynamics' slow change, several Gaussian process tracking methods are proposed for non-stationary system such as back-to-the-prior (B2P) forgetting and uncertainty-injection (UI) forgetting in [15], and exponentially weighted KRLS (EW-KRLS) in [16]. Therefore, system model can be fitted by employing Gaussian process, which will encode the uncertainty of the fitted model and catch the non-stationary property of the true model.

In this paper, we attempt to combine the advantages of model-free RL and model-based RL to tackle the continuous control problems. To this end, we first fit the system model by using Gaussian process. Then, the fitted model is used to generate more supplementary off-policy samples to enrich the on-policy data pool. Finally, the policy network is updated by employing the samples from combined data pool.

The rest of this paper is mainly organized as follows. Section II introduces the background of reinforcement learning. The Gaussian process modelling part which includes system dynamics modelling, update and sparsification mechanism is detailed in section III. In section IV, a policy gradient with Gaussian process acceleration algorithm is proposed and two simulations are given in section V to support the proposed algorithm. The conclusion is given in section VI.

II. Background

A. Preliminaries

In the reinforcement learning settings, the agent takes the action a_t at time t following a policy $\pi(a_t|s_t)$, which is based on the state s_t. Then, conditioning on s_t and a_t, the agent will be transformed into a new state s_{t+1} at the next time stamp $t+1$. It will also receive a reward signal r_t. The objective is to maximize the cumulative rewards from the initial state s_1. We can model this problem as a Markov decision process (MDP), which consists a state space $\mathcal{S}$, an action space $\mathcal{A}$, a transition dynamics distribution of state s_{t+1} conditioned on the last state action pair $P(s_{t+1}|s_t, a_t)$, a reward function $r : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ and an initial state distribution $p(s_1)$. The policy $\pi : \mathcal{S} \rightarrow \mathbb{R}$ is used to pick the suitable action $a_t = \pi(s_t)$ so as to maximize the objective. In general, the policy is a stochastic distribution: $\pi(a_t|s_t)$, which means the probability of choosing action a_t at state s_t is $\pi(a_t|s_t)$. The agent will follow the policy to interact with the environment which gives the trajectory: $\tau = s_1, a_1, r_1, ..., s_T, a_T, r_T$ where T is the terminal time stamp. The cumulative rewards from time stamp t can be formulated as $R_t = \sum_{k=t}^{\infty} \gamma^{k-t} r(s_k, a_k)$ where $\gamma \in (0, 1)$ is discount rate. Two kinds of value functions that are the expected cumulative discount rewards following policy π can be defined as

$$V^{\pi}(s) = \mathbb{E}_{\pi}[R_1|s_1 = s] \tag{1}$$

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi}[R_1|s_1 = s, a_1 = a] \tag{2}$$

where $\mathbb{E}_{\pi}[\cdot]$ denotes the expectation with respect to policy π. The value function can be updated in an incremental style following the Bellman equation:

$$Q^{\pi}_{k+1}(s_t, a_t) = (1-\alpha)Q^{\pi}_k(s_t, a_t) + \alpha(r_t + \gamma Q^{\pi}_k(s_{t+1}, a_{t+1})) \tag{3}$$

where $\alpha \in (0, 1)$ is the learning rate and k is the update index. So the optimal policy which maximizes the Q function can be obtained by

$$\pi^* = \arg\max_{\pi} Q^{\pi}(s, a) \tag{4}$$

B. Stochastic Policy Gradient

Although the optimal policy can be obtained by (4), this kind of representation has its intrinsic shortcomings. First, the maximization operation means we must sweep all the actions at a given state point to figure out which action can bring us maximal immediate reward. It is obviously not a good choice when action dimension is very large or even continuous. Second, a small change of the Q value may cause an action to be selected or not, which will cause the policy chatter and hard to converge. Therefore, directly parameterizing policy π with parameters θ will cast as a good alternative. Moreover, the objective function that the policy tries to maximize can be formulated as $J(\pi_\theta) = \mathbb{E}[R_1|\pi_\theta]$. If we define the discounted state distribution starting from initial state s_1 and then following policy π as $\rho^{\pi}(s) = \int_{\mathcal{S}} \sum_{t=1}^{\infty} \gamma^{t-1} P(s_1) P(s_t = s|s_1, \pi_\theta) \mathrm{d}s$. Then the objective function can be expressed as

$$\begin{aligned} J(\pi_\theta) &= \mathbb{E}_{s\sim\rho^{\pi}, a\sim\pi_\theta}[r(s, a)] \\ &= \int_{\mathcal{S}} \rho^{\pi}(s) \int_{\mathcal{A}} \pi_\theta(a|s) r(s, a) \mathrm{d}a \mathrm{d}s \end{aligned} \tag{5}$$

Hence the policy parameters θ can be adjusted in the direction of objective gradient $\nabla_\theta J(\pi_\theta)$.

Policy gradient theorem [17] gives a fundamental way to compute this gradient as

$$\begin{aligned} \nabla_\theta J(\pi_\theta) &= \mathbb{E}_{s\sim\rho^{\pi}, a\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) Q^{\pi}(s, a)] \\ &= \int_{\mathcal{S}} \rho^{\pi}(s) \int_{\mathcal{A}} \pi_\theta(a|s) \nabla_\theta \log \pi_\theta(a|s) Q^{\pi}(s, a) \mathrm{d}a \mathrm{d}s \end{aligned} \tag{6}$$

Recently, Silver [18] et al. propose a deterministic policy gradient algorithm. The deterministic policy gradient can be formulated as

$$\begin{aligned} \nabla_\theta J(\mu_\theta) &= \mathbb{E}_{s\sim\rho^{\mu}}[\nabla_\theta \mu_\theta(s) \nabla_a Q^{\mu}(s, a)|_{a=\mu_\theta(s)}] \\ &= \int_{\mathcal{S}} \rho^{\mu}(s) \nabla_\theta \mu_\theta(s) \nabla_a Q^{\mu}(s, a)|_{a=\mu_\theta(s)} \mathrm{d}s \end{aligned} \tag{7}$$

Here $\mu_\theta(s)$ is the deterministic policy parameterized by θ.

One notable property of deterministic policy gradient over stochastic policy gradient is that we need not to compute expectation over action since the probability of choosing an action at a given state is one. Hence the deterministic policy is more efficient than the stochastic one.

Note the action-value function $Q^{\mu}(s, a)$ in (7) is unknown, but there are many options about how to estimate this term. We can use the discounted cumulative reward R_T to estimate it, which will be a variant of the REINFORCE algorithm [19]. Some other alternatives can be temporal error or advantage function and so on.

Since policy gradient methods do not conduct maximization operation and represent policy with parameters, they have been the most popular algorithms used to tackle high dimension or continuous action reinforcement learning problems. But as mentioned above, as model-free methods, these algorithms needs tremendous trajectory samples to learn a workable policy. This makes them not suitable for some practical control tasks, like robot control or vehicle control, in which tons of samples will cause high economic costs. Therefore, a sample efficient algorithm is urgently needed. To this end, we combine model-free algorithm with model-based reinforcement learning to speed up learning.

III. Gaussian Process Modelling

In reinforcement learning settings, the agent interacts with environment in a trial-and-error manner and attempts to learn a good control policy. Along interacting with environment, we can collect trajectories data set following a policy π, say $D_t = \{(s_i, a_i, r_i, s_{i+1})\}_{i=1}^t$. We will use D_t to fit a system model by employing Gaussian process.

A. System Dynamics Modelling

Fitting a system model, i.e. the system transition dynamics distribution $P(s_{t+1}|s_t, a_t)$ means we need to compute a latent function $f(s_t, a_t)$ that approximates the true probability distribution above. Now we regard the state action pairs $\{(s_i, a_i)\}_{i=1}^t$ as observations and rewrite $f_t = f(s_t, a_t)$, then latent functions vector can be expressed as $\mathbf{f}_t = \{[f_1, ..., f_t]^{\mathrm{T}} | f_i = s_{i+1}\}_{i=1}^t$.

Gaussian process provides an elegant way to compute latent function in a probabilistic perspective. It can be viewed as a collection of random variables with respect to inputs x, any finite subsets of which have joint Gaussian distribution. In this case, latent functions are the random variables, namely, $f(x) \sim \mathcal{GP}(m(x), k(x, x'))$, where the mean function $m(x) = \mathbb{E}[f(x)]$ is taken to be f_0 and $k(x, x') = \mathbb{E}[(f(x) - m(x))(f(x') - m(x))]$ is a positive semi-definite kernel function. We regard the observations tuple as input $x_i = [s_i^{\mathrm{T}}, a_i^{\mathrm{T}}]^{\mathrm{T}}$,and also define the bias vector as $\mathbf{x}_t = [x_1, x_2, ..., x_t]^{\mathrm{T}}$. The kernel matrix $\mathbf{K}_t$ with element $[\mathbf{K}_t]_{i,j} = k(x_i, x_j)$ is the covariance matrix of the latent functions.

When the agent takes action a_t at state s_t and is transited to the next state s_{t+1}, a new data sample x' is collected. The predictive distribution of $f' = f(x')$ can be calculated as a conditional distribution given trajectories data set

$$p(f'|\mathcal{D}_t) = \int p(f'|\mathbf{f}_t) p(\mathbf{f}_t|\mathcal{D}_t) \mathrm{d}\mathbf{f}_t \tag{8}$$

According to Gaussian process, the predictive distribution latent function f' is also a Gaussian distribution with mean $\mu(x')$ and variance $\sigma^2(x')$ as

$$\mu(x') = f_0 + k(\mathbf{x}_t, x')^{\mathsf{T}} \mathbf{K}_t^{-1} (\mathbf{f}_t - f_0) \tag{9a}$$

$$\sigma^2(x') = k(x', x') - k(\mathbf{x}_t, x')^{\mathsf{T}} \mathbf{K}_t^{-1} k(\mathbf{x}_t, x') \tag{9b}$$

where $k(\mathbf{x}_t, x')$ is the kernel function between bias vector $\mathbf{x}_t$ and input x'.

In reinforcement learning settings, due to the environment's noise, the next observation is the noisy version of the true next state. Hence, using the re-parameterization trick, the next observation can be formulated as the sum of latent function $f(\cdot)$ on input and independent zero-mean Gaussian noise $\epsilon \sim \mathcal{N}(0, \sigma_n^2)$, namely,

$$y_{t+1} = s_{t+1} + \epsilon_{t+1} \tag{10}$$

Till now, we have related the input and true next state in (8), of which the mean and variance can be obtained following per (9). Moreover, the observed next state given by environment can be computed through (10).

Algorithm 1 GP_Model_Fitting(B)

```
1: Initialize bias vector $\mathbf{x}_t \leftarrow \emptyset$ and state noise variance $\sigma_n^2$
2: while $B \neq \emptyset$ do
3:     Sample a transition $(s_t, a_t, r_t, s_{t+1})$ from $B$
4:     $x_t = [s_t, a_t]$ and $k_{t,t} = k(x_t, x_t)$
5:     if $\mathbf{x}_t = \emptyset$ then
6:         $\mathbf{K}_1 = [k_{t,t}], \mathbf{K}_1^{-1} = [1/k_{t,t}], \mu_1 = s_{t+1}, \mathbf{\Sigma}_1 = [k_{t,t}]$ and $\mathbf{x}_1 = [x_1]$
7:     else
8:         $\mathbf{k}_t = k(\mathbf{x}_t, x_t)$
9:         $\mathbf{q}_{t+1} = \mathbf{K}_t^{-1} \mathbf{k}_t$
10:        Compute ALD criterion: $\delta^2 = k_{t,t} - \mathbf{q}_{t+1}^{\mathrm{T}} \mathbf{k}_t$
11:        if $\delta^2 < \nu$ then
12:            Remove transition $(s_t, a_t, r_t, s_{t+1})$ from $B$
13:            Continue
14:        end if
15:        $\hat{s}_{t+1} = f_0 + \mathbf{q}_{t+1}^{\mathrm{T}} (\mathbf{f}_t - \mathbf{f}_0)$
16:        $\hat{\sigma}_{s_{t+1}}^2 = k_{t,t} - \mathbf{q}_{t+1}^{\mathrm{T}} \mathbf{k}_t$
17:        $\hat{y}_{t+1} = f_0 + \mathbf{q}_{t+1}^{\mathrm{T}} (\boldsymbol{\mu}_t - \mathbf{f}_0)$
18:        $\hat{\sigma}_{y_{t+1}}^2 = \hat{\sigma}_{s_{t+1}}^2 + \mathbf{q}_{t+1}^{\mathrm{T}} \Sigma_t \mathbf{q}_{t+1} + \sigma_n^2$
19:        $\mathbf{h}_{t+1} = \mathbf{\Sigma}_t \mathbf{q}_{t+1}$
20:        Update $\boldsymbol{\mu}_{t+1}$ and $\mathbf{\Sigma}_{t+1}$ as (17b),(17c)
21:        Update $\mathbf{K}_{t+1}^{-1}$ as (18)
22:        $\mathbf{x}_{t+1} = \mathbf{x}_t \cup x_t$
23:    end if
24:    Remove transition $(s_t, a_t, r_t, s_{t+1})$ from $B$
25: end while
```

B. Online Updating

We have derived how to obtain latent function in (8) above, here we are interested in updating the system model $p(\mathbf{f}_t|\mathcal{D}_t)$ online. When a new input pair is collected, instead of recomputing the prior every time, we can recursively update it as follows to reduce computation complexity

$$\begin{aligned} p(\mathbf{f}_{t+1}|\mathcal{D}_{t+1}) &= p(s_{t+2}, \mathbf{f}_t | \mathcal{D}_t, y_{t+2}) \\ &= \frac{p(y_{t+2}|s_{t+2}, \mathbf{f}_t, \mathcal{D}_t) p(s_{t+2}|\mathbf{f}_t, \mathcal{D}_t) p(\mathbf{f}_t|\mathcal{D}_t)}{p(y_{t+2}|\mathcal{D}_t)} \\ &= \frac{p(y_{t+2}|s_{t+2}) p(s_{t+2}|\mathbf{f}_t) p(\mathbf{f}_t|\mathcal{D}_t)}{p(y_{t+2}|\mathcal{D}_t)}. \end{aligned} \tag{11}$$

where the last step of (11) follows the fact that observed noisy output y_{t+2} is independent of $\mathbf{f}_t, \mathcal{D}_t$ following (10) and s_{t+2} is independent of $\mathcal{D}_t$.

We can see from (11) that the system model posterior distribution is updated in a recursive manner, which means that the posterior at time stamp t would serve as prior when a new sample is collected at $t + 1$. Since the four distributions at last step in (11) are Gaussian distributions, the posterior distribution is also a Gaussian distribution. Now we derive four distributions so as to update posterior in a closed form. First, we assume the prior $p(\mathbf{f}_t|\mathcal{D}_t)$ is a known Gaussian distribution

$$p(\mathbf{f}_t|\mathcal{D}_t) = \mathcal{N}(\boldsymbol{\mu}_t, \Sigma_t) \tag{12}$$

When a new sample x_{t+1} is collected, the joint distribution of the true next state s_{t+1} and previous latent functions vector $\mathbf{f}_t$ is still a Gaussian distribution

$$\begin{bmatrix}\mathbf{f}_t\\ s_{t+1}\end{bmatrix} \sim \mathcal{N}(\begin{bmatrix}\mathbf{f}_0\\ f_0\end{bmatrix}, \begin{bmatrix}\mathbf{K}_t & k(\mathbf{x}_t, x_{t+1})\\ k(\mathbf{x}_t, x_{t+1})^\mathsf{T} & k(x_{t+1}, x_{t+1})\end{bmatrix}) \tag{13}$$

According to the relationship of Gaussian distribution, the conditional distribution of s_{t+1} given the latent functions vector $\mathbf{f}_t$ can be described as

$$p(s_{t+1}|\mathbf{f}_t) = \mathcal{N}(\hat{s}_{t+1}, \hat{\sigma}^2_{s_{t+1}}) \tag{14}$$

where mean $\hat{s}_{t+1} = f_0 + \mathbf{q}^\mathrm{T}_{t+1}(\mathbf{f}_t - \mathbf{f}_0)$, $\mathbf{q}_{t+1} = \mathbf{K}_t^{-1}k(\mathbf{x}_t, x_{t+1})$, and variance $\hat{\sigma}^2_{s_{t+1}} = k(x_{t+1}, x_{t+1}) - \mathbf{q}^\mathrm{T}_{t+1}k(\mathbf{x}_t, x_{t+1})$.

Following (10), the noisy observed output y_{t+1} given the true next state s_{t+1} is

$$p(y_{t+1}|s_{t+1}) = \mathcal{N}(s_{t+1}, \sigma_n^2) \tag{15}$$

Given the past data set, the conditional distribution of noisy observed output can be computed as

$$\begin{aligned} p(y_{t+1}|\mathcal{D}_t) &= \int p(y_{t+1}|s_{t+1})p(s_{t+1}|\mathbf{f}_t)p(\mathbf{f}_t|\mathcal{D}_t)\mathrm{d}\mathbf{f}_t\mathrm{d}s_{t+1}\\ &= N(\hat{y}_{t+1}, \hat{\sigma}^2_{y_{t+1}}) \end{aligned} \tag{16}$$

where the mean $\hat{y}_{t+1} = f_0 + \mathbf{q}^\mathrm{T}_{t+1}(\boldsymbol{\mu_t} - \mathbf{f}_0)$, and the variance $\hat{\sigma}^2_{y_{t+1}} = \hat{\sigma}^2_{s_{t+1}} + \mathbf{q}^\mathrm{T}_{t+1}\Sigma_t\mathbf{q}_{t+1} + \sigma_n^2$.

According to the relationship of Gaussian distributions, the posterior can be updated as follows when incorporating the new sample to bias vector $\mathbf{x}_{t+1} = \mathbf{x}_t \cup x_{t+1}$

$$p(\mathbf{f}_{t+1}|\mathcal{D}_{t+1}) = \mathcal{N}(\boldsymbol{\mu}_{t+1}, \boldsymbol{\Sigma}_{t+1}) \tag{17a}$$

$$\boldsymbol{\mu}_{t+1} = \begin{bmatrix}\boldsymbol{\mu}_t\\ \hat{s}_{t+1}\end{bmatrix} + \frac{y_{t+1} - \hat{y}_{t+1}}{\hat{\sigma}^2_{y_{t+1}}}\begin{bmatrix}\mathbf{h}_{t+1}\\ \hat{\sigma}^2_{s_{t+1}}\end{bmatrix} \tag{17b}$$

$$\boldsymbol{\Sigma}_{t+1} = \begin{bmatrix}\boldsymbol{\Sigma}_t & \mathbf{h}_{t+1}\\ \mathbf{h}^\mathsf{T}_{t+1} & \hat{\sigma}^2_{s_{t+1}}\end{bmatrix} - \frac{1}{\hat{\sigma}^2_{y_{t+1}}}\begin{bmatrix}\mathbf{h}_{t+1}\\ \hat{\sigma}^2_{s_{t+1}}\end{bmatrix}\begin{bmatrix}\mathbf{h}_{t+1}\\ \hat{\sigma}^2_{s_{t+1}}\end{bmatrix}^\mathsf{T} \tag{17c}$$

where $\mathbf{h}_{t+1} = \Sigma_t\mathbf{q}_{t+1}$.

The inverse of the kernel matrix $\mathbf{K}^{-1}$ including a new sample is updated as

$$\mathbf{K}^{-1}_{t+1} = \begin{bmatrix}\mathbf{K}^{-1}_t & \mathbf{0}\\ \mathbf{0}^\mathsf{T} & 0\end{bmatrix} + \frac{1}{\hat{\sigma}^2_{s_{t+1}}}\begin{bmatrix}\mathbf{q}_{t+1}\\ -1\end{bmatrix}\begin{bmatrix}\mathbf{q}_{t+1}\\ -1\end{bmatrix}^\mathsf{T}. \tag{18}$$

Till now, we have derived the way to update system dynamics posterior distribution when a new input sample is collected every time. This update consists of two parts: the value update part and the uncertainty casting part. For example, the first term of the right hand side of (17b) represents the normal mean vector updating while the second term provides the uncertainty information of Gaussian process.

C. Sparsification

The updating in (17) has a drawback. Every time a new input sample is collected, it would be added to bias vector $\mathbf{x}_t$ and update the inverse covariance matrix as (18). This means the bias vector length would become infinity when $t \to \infty$ and both the memory overhead and computational complexity would blow up. Therefore, we need to do sparsification to tackle this issue.

One efficient way to do this is to transform data into a high dimensional reproducing kernel Hilbert space and measure correlation strength among data points. The correlation strength between $\mathbf{x}_t$ and x' is described by approximately linear dependent (ALD) criterion [14]

$$\delta^2 = k(x', x') - k(\mathbf{x}_t, x')^\mathsf{T}\mathbf{K}_t^{-1}k(\mathbf{x}_t, x') \tag{19}$$

We can set a threshold ν to draw the line. Every time a new sample x' is collected, we will measure the strength between x' and bias vector $\mathbf{x}_t$ by ALD criterion. If $\delta^2 > \nu$, namely the correlation is low and this new sample will bring new information, then $\mathbf{x}_{t+1} = \mathbf{x}_t \cup x'$. Otherwise, it would be discarded.

Up to now, we can fit the system model by using online Gaussian process. The algorithm for this is expressed in Algorithm 1.

IV. Policy Gradient with Gaussian Process Acceleration

Policy gradient algorithms are the most popular algorithms among which are used to deal with continuous action space reinforcement learning problems. Here we use neural network as the policy and action value function approximator, which is used to compute deterministic policy gradient in (7). Recently, Mnih et al [8] take deep neural network to approximate action value function. Their Deep Q-network algorithm can play Atari video games from pixels. So we adopt their key ideas: replay buffer and separate target network. Thus by using actor-critic approach, the loss function used to train critic network is

$$L = \frac{1}{N}\sum_i(y_i - Q(s_i, a_i|\theta^Q))^2 \tag{20}$$

The target is defined as

$$y_i = r_i + \gamma Q'(s_{i+1}, \pi'(s_{i+1}|\theta^{\pi'})|\theta^{Q'}) \tag{21}$$

where θ^Q is critic network weights, $\theta^{\pi'}$ and $\theta^{Q'}$ is target policy and target action value function network weights respectively.

The fitted system model is also employed to produce additional data samples. First, we follow the policy π to obtain the real trajectories. Then these trajectories are further used to fit a system model. Having the fitted model, we pick up samples from real trajectories and use them as initial points to run imaginary rollouts. Due to the model fitting bias, we only run the imaginary rollouts in a shallow manner instead of starting from a initial point and generating samples until the

Algorithm 2 Policy Gradient with GP Acceleration

```
1: Randomly initialize critic network weights $\theta^Q$ and actor network weights $\theta^\pi$
2: Initialize target network weights $\theta^{Q'} \leftarrow \theta^Q$ and $\theta^{\pi'} \leftarrow \theta^\pi$
3: Initialize replay buffer $R \leftarrow \emptyset$ and GP buffer $R_{gp} \leftarrow \emptyset$
4: Initialize fitted model $\mathcal{M} \leftarrow \emptyset$ and model buffer $B \leftarrow \emptyset$
5: for episode = 1,M do
6:     Initialize a random process $\mathcal{N}$ for action exploration
7:     Receive initial observation state $s_1$
8:     for t = 1,T do
9:         Select action $a_t = \pi(s_t|\theta^\pi) + \mathcal{N}_t$
10:        Execute action $a_t$ and observe $r_t$ and $s_{t+1}$
11:        Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$ and $B$
12:        if mod$(t, N) = 0$ and $\mathcal{M} \neq \emptyset$ then
13:            Sample a minibatch of $N$ from $R$
14:            Use $\mathcal{M}$ to simulate $k$ steps from each samples
15:            Store all transitions in $R_{gp}$
16:        end if
17:        Sample a random minibatch of $N$ transitions from $R$ and $R_{gp}$
18:        Set $y_i = r_i + \gamma Q'(s_{i+1}, \pi'(s_{i+1}|\theta^{\pi'})|\theta^{Q'})$
19:        Update $\theta^Q$ by minimizing the loss:
           $L = \frac{1}{2N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
20:        Update $\theta^\pi$ using the gradient (7)
21:        Update the target networks:
           $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
           $\theta^{\pi'} \leftarrow \tau\theta^\pi + (1-\tau)\theta^{\pi'}$
22:    end for
23:    if $B$ is full then
24:        $\mathcal{M} \leftarrow GP\_Model\_Fitting(B)$
25:        $B \leftarrow \emptyset$
26:    end if
27: end for
```

terminal state is met. The reason for this is that we would deviate from the true system model further and further if the fitted system model is inaccurate. Both these imaginary samples and real trajectory samples are used to train the critic and actor networks. Since we are using the learned model to perform rollouts from real trajectory samples, the data set would be largely augmented. This is equivalent to explore the state and action space in some sense. Algorithm 2 shows the details of this process.

V. Simulation

In this section, we want to answer two questions: first, whether Gaussian process can approximate the system model accurately or not; second, whether the generated imaginary samples can speed up learning phase. In order to answer the first question, we utilize Gaussian process to fit a pendulum's transition model and reward model. Then we use these fitted pendulum models to generate imaginary samples to figure out whether Gaussian process modelling can be used to accelerate learning, namely, answering the second question.

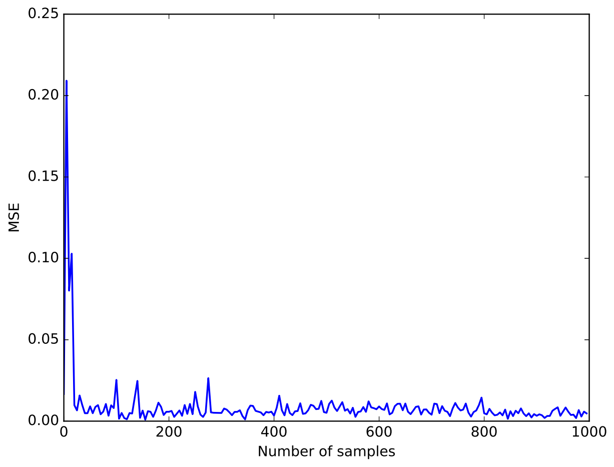

Fig. 1. The mean squared error of fitted system state transition model.

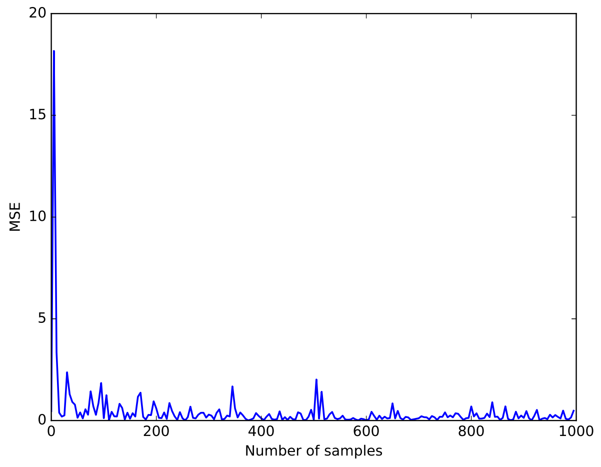

Fig. 2. The mean squared error of fitted system reward model.

A. System Model Fitting

In the first experiment, we want to figure out whether Gaussian process can model system model accurately. To this end, an agent is built via OpenAi gym [20] to control a pendulum. Two models, transition dynamics model and reward model, are learned separately. The gym platform defines these two models, which we do not know and can only learn them from raw data. Gaussian kernel function is selected as the covariance function

$$k(x, x') = \alpha \exp(-\frac{\|x - x'\|^2}{2l^2}). \tag{22}$$

The Gaussian kernel hyperparameters are selected as $\alpha = 1$, $l = 4$, bias vector size equals to 800 for reward model and $\alpha = 1$, $l = 7$, bias vector size equals to 1000 for transition dynamics model. Note that all the hyper-parameters are selected empirically. The fitting mean-squared-error (MSE)

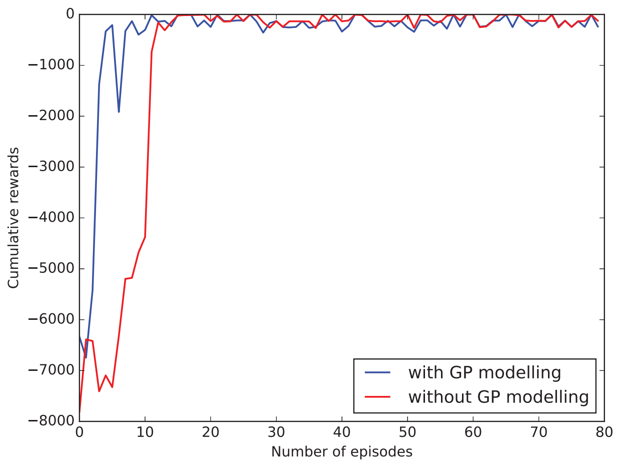

Fig. 3. The training process of the algorithms with and without Gaussian process acceleration.

of transition dynamics model and system reward model are shown in Fig. 1 and Fig. 2 respectively. The MSE is averaged over five independent training process. The small chatter in both figures is due to the intrinsic system noise which is used to describe real world noise. Both Figure 1 and Figure 2 show that Gaussian process can approximate the true system model accurately and quickly. Therefore, we can first fit the system models by using Gaussian process and then use them to generate samples to explore state and action space in an imaginary sense.

B. Gaussian Process Acceleration

In this experiment, we would like to answer the second question: whether the generated imaginary samples can speed up learning phase. Since we have obtained good system models, we now follow the Algorithm 2 to train a reinforcement learning agent. All the hyperparameters remain the same as the first experiment. We compare two algorithms: the deterministic policy gradient with and without Gaussian process modelling.

The training performance is shown in Fig. 3. We can tell from this figure that the proposed algorithm converges faster than the algorithm without Gaussian process acceleration. The proposed algorithm can achieve good control performance less than 10 episodes. However, the policy gradient algorithm only gets good performance after about 15 episodes. Therefore, Gaussian process modelling can indeed accelerate learning.

VI. Conclusion

In this paper, a policy gradient algorithm with Gaussian process acceleration algorithm is proposed to combine the advantages of model-free method and model-based method. A Gaussian process is used to fit system model, which is used to generate imaginary samples so as to enrich the training data pool later. Compared with the policy gradient algorithm without modelling, this proposed algorithm can take sufficient exploration in an imaginary state action space which is important at some circumstances like autonomous vehicle and robot control because some states are too dangerous to reach. Two experiments are implemented to verify the accuracy of Gaussian process modelling. Moreover, compared with the policy gradient algorithm, the proposed algorithm can speed up the learning phase obviously. Future work involves how to efficiently explore the imaginary state action space.

References

[1] R. Sutton and A. Barto, "Reinforcement learning: an introduction." MIT Press, 1998.

[2] A. Y. Ng, A. Coates, M. Diel, V. Ganapathi, J. Schulte, B. Tse, E. Berger, and E. Liang, "Autonomous inverted helicopter flight via reinforcement learning," in *Experimental Robotics IX*. Springer, 2006, pp. 363–372.

[3] Q. Zhang, D. Zhao, and Y. Zhu, "Event-triggered h-infinity control for continuous-time nonlinear system via concurrent learning," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, doi: 10.1109/TSMC.2016.2531680, 2016.

[4] D. Zhao, Q. Zhang, D. Wang, and Y. Zhu, "Experience replay for optimal control of nonzero-sum game systems with unknown dynamics." *IEEE Transactions on Cybernetics*, vol. 46, no. 3, pp. 854–865, 2016.

[5] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *arXiv preprint arXiv:1512.03385*, 2015.

[6] G. E. Hinton, L. Deng, D. Yu, G. E. Dahl, A. Mohamed, N. Jaitly, A. W. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath *et al.*, "Deep neural networks for acoustic modeling in speech recognition," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[7] S. R. Bowman, G. Angeli, C. Potts, and C. D. Manning, "A large annotated corpus for learning natural language inference," *arXiv preprint arXiv:1508.05326*, 2015.

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[9] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[10] D. Zhao, Y. Chen, and L. Lv, "Deep reinforcement learning with visual attention for vehicle classification," *IEEE Transactions on Cognitive and Developmental Systems*, doi:10.1109/TCDS.2016.2614675, 2016.

[11] C. E. Rasmussen, "Gaussian processes for machine learning." MIT Press, 2006.

[12] Z. Xia and D. Zhao, "Online reinforcement learning control by Bayesian inference," *IET Control Theory and Applications*, vol. 10, no. 12, pp. 1331–1338, 2016.

[13] Y. Engel, S. Mannor, and R. Meir, "Reinforcement learning with Gaussian processes," *Proceedings of the 22nd International Conference on Machine Learning*, pp. 201–208, 2005.

[14] ——, "The kernel recursive least-squares algorithm," *IEEE Transactions on signal processing*, vol. 52, no. 8, pp. 2275–2285, 2004.

[15] S. Van Vaerenbergh, M. Lázaro-Gredilla, and I. Santamaría, "Kernel recursive least-squares tracker for time-varying regression," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 23, no. 8, pp. 1313–1326, 2012.

[16] W. Liu, I. Park, Y. Wang, and J. C. Príncipe, "Extended kernel recursive least squares algorithm," *IEEE Transactions on Signal Processing*, vol. 57, no. 10, pp. 3801–3814, 2009.

[17] R. S. Sutton, D. Mcallester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," *Advances in Neural Information Processing Systems*, pp. 1057–1063, 2000.

[18] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," *Proceedings of the 31st International Conference on Machine Learning*, pp. 387–395, 2014.

[19] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine Learning*, vol. 8, no. 3-4, pp. 229–256, 1992.

[20] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.