SAM-RL: Sensing-Aware Model-Based Reinforcement Learning via Differentiable Physics-Based Simulation and Rendering

Jun Lv¹, Yunhai Feng², Cheng Zhang³, Shuang Zhao³, Lin Shao⁴* and Cewu Lu⁵*

Abstract—Model-based reinforcement learning (MBRL) is recognized with the potential to be significantly more sample efficient than model-free RL. How an accurate model can be developed automatically and efficiently from raw sensory inputs (such as images), especially for complex environments and tasks, is a challenging problem that hinders the broad application of MBRL in the real world. In this work, we propose a sensing-aware model-based reinforcement learning system called SAM-RL. Leveraging the differentiable physics-based simulation and rendering, SAM-RL automatically updates the model by comparing rendered images with real raw images and produces the policy efficiently. With the sensing-aware learning pipeline, SAM-RL allows a robot to select an informative viewpoint to monitor the task process. We apply our framework to real world experiments for accomplishing three manipulation tasks: robotic assembly, tool manipulation, and deformable object manipulation. We demonstrate the effectiveness of SAM-RL via extensive experiments. Videos are available on our project webpage at https://sites.google.com/view/rss-sam-rl.

I. INTRODUCTION

Over the past decade, deep reinforcement learning (RL) has resulted in impressive successes, including mastering Atari games [1], winning the games of Go [2], and solving Rubik’s cube with a human-like robot hand [3]. However, deep RL algorithms adopt the paradigm of model-free RL and require vast amounts of training data, significantly limiting their practicality for real-world robotic tasks. Model-based reinforcement learning (MBRL) is recognized with the potential to be significantly more sample efficient than model-free RL [4].

How to automatically and efficiently develop an accurate model from raw sensory inputs, especially for complex environments and tasks, is a challenging problem that hinders the wide application of MBRL in the physical world.

One line of works [5, 6, 7, 8] adopt representation learning approaches to learn the model from raw input data. They aim to learn low-dimensional latent states and action representations from high-dimensional input data like images. But the learned deep network might not satisfy the physical dynamics, and its quality may also significantly degenerate beyond the training data distribution when testing in the wild. Recent developments in differentiable physics-based simulation [9, 10, 11, 12, 13, 14, 15, 16] and rendering [17, 18, 19, 20] provide an alternative direction to model the environment [21, 22]. Lv et al. [23] use differentiable physics-based simulation as the backbone of the model and train robots to perform articulated object manipulation in the real world. Their pipeline produces a file of Unified Robot Description Format (URDF) [24] of the environment, which is loaded into the differentiable simulation from raw point clouds gathered by an RGB-D camera mounted on its wrist. However, a sequence of camera poses is needed to scan the 3D environment every time step, and these camera poses are manually predefined, which is time-consuming and

*Equal advising.

¹Jun Lv is with the Department of Electronic Engineering, Shanghai Jiao Tong University, China. [ljun@sjtu.edu.cn]
²Yunhai Feng is with the Department of Computer Science and Engineering, University of California San Diego, USA. [yuf020@ucsd.edu]
³Cheng Zhang and Shuang Zhao are with the Department of Computer Science, University of California Irvine, USA. [chengz20@uci.edu, shz@ics.uci.edu]
⁴Lin Shao is with the Department of Computer Science, National University of Singapore, Singapore. [linshao@nus.edu.sg]
⁵Cewu Lu is the corresponding author, the member of Qing Yuan Research Institute and MoE Key Lab of Artificial Intelligence, AI Institute, Shanghai Jiao Tong University, China and Shanghai Qi Zhi Institute. [lucewu@sjtu.edu.cn]
difficult to adapt to various tasks. Object colors and geometric details are not included in the model, limiting its representation capability [23].

By integrating differentiable physics-based simulation and rendering, we propose a sensing-aware model-based reinforcement learning system called \textit{SAM-RL}. As shown in Fig. [1] we apply \textit{SAM-RL} on a robot system with two 7-DoF robotic arms (Flexiv Rizon [25] and Franka Emika Panda), where the former mounts an RGB-D camera, and the latter handles manipulation tasks. Our framework is sensing-aware, which allows the robot to automatically select an informative camera view to effectively monitor the manipulation process, providing the following benefits. First, the system no longer requires obtaining a sequence of camera poses at each step, which is extremely time-consuming. Second, compared with using a fixed view, \textit{SAM-RL} leverages varying camera views with potentially fewer occlusions and offers better estimations of environment states and object status (especially for deformable bodies). The improved quality in object status estimation contributes more effective robotic actions to complete various tasks. Third, by comparing rendered and measured (i.e., real-world) images, discrepancies between the simulation and the reality are better revealed and then reduced automatically using gradient-based optimization and differentiable rendering.

In practice, we train the robot to learn three challenging manipulation skills: Peg-\textit{Insertion}, Spatula-\textit{Flipping}, and Needle-\textit{Threading}. Our experiments indicate that \textit{SAM-RL} can significantly reduce training time and improve success rate by large margins compared to common model-free and model-based deep reinforcement learning algorithms.

Our primary contributions include:

\begin{itemize}
\item proposing an active-sensing framework named \textit{SAM-RL} that enables robots to select informative views for various manipulation tasks;
\item introducing a model-based reinforcement learning algorithm to produce efficient policies;
\item conducting extensive quantitative and qualitative evaluations to demonstrate the effectiveness of our approach;
\item applying our framework to robotic assembly, tool manipulation, and deformable object manipulation tasks both in simulation and real world experiments.
\end{itemize}

II. RELATED WORK

We review related literature on key components in our approach, including model-based reinforcement learning, next best view, integration of differentiable physics-based simulation and rendering, and robotic manipulation. We describe how we are different from previous work.

A. Model-based Reinforcement Learning

MBRL is considered to be potentially more sample efficient than model-free RL [4]. However, automatically and efficiently developing an accurate \textit{model} from raw sensory data is a challenging problem, which retards MBRL from being widely applied in the real world. For a broader review of the field on MBRL, we refer to [26]. One line of works [5] 6 7 8 use representation learning methods to learn low-dimensional latent state and action representations from high-dimensional input data. But the learned models might not satisfy the physical dynamics, and the quality may also significantly drop beyond the training data distribution. Recently, Lv et al. [23] leveraged the differentiable physics simulation and developed a system to produce a URDF file to model the surrounding environment based on an RGB-D camera. However, the RGB-D camera poses used in [23] are predefined and can not adjust to different tasks. Our approach allows the robot to select the most informative camera view to monitor the manipulation process and update the environment \textit{model} automatically.

B. Next Best View in Active Sensing

Next Best View (NBV) has been one of the core problems in active sensing. It studies the problem of how to obtain a series of sensor poses to increase the information gain. The information gain is explicitly defined to reflect the improved perception for 3D reconstruction [27] 28 29 30, object recognition [31] 32 33 34, 3D model completion [35], and 3D exploration [36] 37. Unlike perception-related tasks, we explore the NBV over a wide range of robotic manipulation tasks. Information gain in the robotic manipulation tasks is difficult to define explicitly and is implicitly related to task performance. In our system, the environment changes accordingly after the robot’s interaction. We integrate the information gain into the $Q$ function to reflect the informative viewpoint for manipulation.

C. Integration of Differentiable Physics-Based Simulation and Rendering

Recently, great progresses have been made in the field of differentiable physics-based simulation and rendering. For a broader review, please refer to [9] 10 11 12 13 14 15 16 and [38] 39. With the development of these techniques, Jatavallabhula et al. [21] first proposed a pipeline to leverage differentiable simulation and rendering for system identification and visuomotor control. Ma et al. [22] introduced a rendering-invariant state predictor network that maps images into states that are agnostic to rendering parameters. By comparing the state predictions obtained using rendered and ground-truth images, the pipeline can backpropagate the gradient to update system parameters and actions. Sundaresan et al. [40] proposed a real-to-sim parameter estimation approach from point clouds for deformable objects. Different from these works, we use the differentiable simulation and rendering to find the next best view for various manipulation tasks and update the object status in the model by comparing rendered and captured images.

D. Manipulation

Our framework can be adopted to improve the performance of a range of manipulation tasks. We review the related work in these domains. 1) Peg-insertion. Peg insertion is a classic robotic assembly task with rich literature [41] 42 43. For a
A. Model with Differentiable Simulation and Rendering

In this work, we combine the differentiable physics-based simulation and rendering. The resulting differentiable system plays the backbone of the \( \mathcal{M} \), which we denote as \( \mathcal{M} \), for the model-based reinforcement learning. The \( \mathcal{M} \) can load robots, cameras, and objects denoted as \( \{\mathcal{O}_j\} \) along with their visual/geometric (e.g., shape, pose, and texture) and physical attributes (e.g., mass and inertial). We denote the attributes of all objects loaded in the simulation as one type of \( \mathcal{M} \)'s parameters \( \psi_{\mathcal{M}} \) as follows:

\[
\psi_{\mathcal{M}} = \sum_j \mathcal{O}^{\text{sim}}_j.
\]

The \( \mathcal{M} \) can render an image \( \mathcal{I}^{\text{sim}} \) under the camera pose \( \mathcal{P}^{c} \) and model parameter \( \psi_{\mathcal{M}} \) through:

\[
\mathcal{I}^{\text{sim}} = \mathcal{M}(\psi_{\mathcal{M}}, \mathcal{P}^{c}; \text{render})
\]

We can get the gradients \( \partial \mathcal{I}^{\text{sim}} / \partial \mathcal{P}^{c} \) and \( \partial \mathcal{I}^{\text{sim}} / \partial \mathcal{O}^{\text{sim}}_j \) using differentiable rendering. Note that \( \partial \mathcal{I}^{\text{sim}} / \partial \mathcal{O}^{\text{sim}}_j \) contains only the gradient with respect to object visual and geometric attributes.

Additionally, given the state \( s_t^{\text{sim}} \) including the object attributes \( \psi_{\mathcal{M}} \) and the robots’ status, when an action denoted as \( a_t^{\text{sim}} \) is executed (for example, an external force is exerted on the object), the \( \mathcal{M} \) simulates the next state via

\[
s_{t+1}^{\text{sim}} = \mathcal{M}(s_t^{\text{sim}}, a_t^{\text{sim}}; \text{forward})
\]

in a differentiable fashion \cite{10}, providing the gradients \( \partial s_{t+1}^{\text{sim}} / \partial s_t^{\text{sim}} \) and \( \partial s_{t+1}^{\text{sim}} / \partial a_t^{\text{sim}} \).
B. Real2Sim: Developing Model from the Real World

1) Build the Initial Model: For model-based reinforcement learning, the first step is to build an initial model of the environment. The initial model does not need to be accurate and can be created using current 3D object reconstruction methods with RGB-D cameras like BundleFusion [50] and KinectFuction [51]. In our setting, as shown in Fig. 2, a calibrated RGB-D camera is mounted on the robot’s wrist. Therefore the robot system takes a set of images with corresponding accurate camera poses. The initial model can also be built directly from a CAD model [52] or following the pipeline described in SAGCI [23] to produce the URDF. After the initialization, the model M contains the robots, objects and an RGB-D camera.

2) Update the Model with Differentiable Simulation and Rendering: After having an initial model, we then describe how to update the model by directly comparing the raw visual observation in simulation and the real world, leveraging the differentiable simulation [10] and rendering [18]. In this work, we only care about one object and assume we can get accurate object segmentation of real world images. With common techniques such as Mask R-CNN [53], it’s feasible to get a fine object-level instance segmentation. At the beginning, we update the camera and robot pose in simulation to match the corresponding camera and robot pose in the real world. Then we get the rendered RGB-D image from M and corresponding real RGB-D image with associated segmentation. Based on the camera parameters, we transform the depth image to point cloud and segment the point cloud. We denote the RGB image, associated object segmentation, and segmented point cloud (I_{sim,rgb}, G_{sim}, X_{sim}) in the simulation and (I_{real,rgb}, G_{real}, X_{real}) in the real world. For simplicity, we use I_{sim} to represent the (I_{sim,rgb}, G_{sim}, X_{sim}) and I_{real} is defined accordingly.

We define the loss functions as follows.

\[ L_1 = \| G_{real} \otimes I_{real,rgb} - G_{sim} \otimes I_{sim,rgb} \|_1 \]
\[ L_2 = EMD(X_{sim}, X_{real}) \]
\[ L = \lambda_1 L_1 + \lambda_2 L_2 \]

where \otimes represents the pixel-wise product operator and EMD represents the Earth Mover Distance [54] to measure the distance between two 3D point clouds. Note that (I_{sim,rgb}, G_{sim}, X_{sim}) = M(ψ_M, P^c; render), as explained in Sec. III-A is differentiable. With the gradient ∂L/∂ψ_M, we can update the model M’s parameters ψ_M including object mesh vertices, colors, and poses so that the loss L is reduced.

\[ ψ_M \leftarrow ψ_M - \lambda_M \frac{∂L}{∂ψ_M} \]

During the manipulation, at each time step t, the model parameters like the poses would be changed by the robot action a_t. We denote the object poses as P^O_t. We will first update the object pose via the forward simulation by

\[ P^O_{t+1} \leftarrow M(P^O_t, a_t; forward) \]

However, the sim-to-real gap may make the model pose in simulation inaccurate. So we still need to use the method described in 7 to further update the model pose.

Next, we describe how to update objects’ physical attributes. Starting from poses P^O_0 at timestep t = 0, with a sequence of actions \{a_t\}_{t=0}^T applied, the simulation calculates the object poses at timestep t = T denoted as P^O_T based on objects’ physical attributes by simulating Eqn. 8 for T steps.

\[ P^O_T \leftarrow M(P^O_0, \{a_t\}_{t=0}^T; forward) \]

The object poses from the real-world at timestep t = T are gathered with differentiable rendering denoted as P^O_T real. We then calculate the distance between the two poses.

\[ L_p = \| P^O_T - P^O_T \|_1 \]

Then object physical attributes ψ_M^phy (mass and inertial) are updated to minimize the loss L_p.

\[ ψ_M^phy \leftarrow ψ_M^phy - \lambda_M \frac{∂L_p}{∂ψ_M^phy} \]

Through this, we can keep reducing the discrepancy between the simulation and the real world, making the model more and more accurate. Up to now, we have introduced how to build and update the model.

C. Sim2Real: Learning Residual Policy in the Real World

We delay the discussion of how to learn the policy to complete the task T with the model in the simulation to the next subsection III-D. Here we assume that we have the policy π_{sim} in simulation which takes rendered images I_{sim} as inputs and outputs actions denoted as a_{sim} = π_{sim}(I_{sim}).

We describe how to apply the learned policy from simulation to the real world by learning a residual policy π_{res} to reduce the sim-to-real gap in this subsection.

We set the same camera pose denoted as P^c both in the simulation and the real world and get images denoted as I_{sim,0} and I_{real}. We first update the model’s parameter ψ_M by minimizing the loss with Eqn. 6 as described in Sec. III-B2 and then get the new images I_{sim} with the updated ψ_M.

With the I_{sim} and I_{real}, we get an action from simulation a_{sim}. Instead of directly applying the action a_{sim} in the real world, we train a residual policy that takes in the real image I_{real} and the a_{sim}, outputs the residual action δa_{real} = π_{res}(I_{real}, a_{sim})

Then actor model in the real world gets the action as follows.

\[ a_{real} = π_{real}(I_{real}, a_{sim}) = a_{sim} + δa_{real} \]

We follow the training process of residual policy described in [55]. The residual policy should never make a good initial policy worse. We therefore initialize the residual policy so that π_{res}(I_{real}, a_{sim}) = 0.

However, the sim-to-real gap may make the model pose in simulation inaccurate. So we still need to use the method described in 7 to further update the model pose.

Next, we describe how to update objects’ physical attributes. Starting from poses P^O_0 at timestep t = 0, with a sequence of actions \{a_t\}_{t=0}^T applied, the simulation calculates the object poses at timestep t = T denoted as P^O_T based on objects’ physical attributes by simulating Eqn. 8 for T steps.

\[ P^O_T \leftarrow M(P^O_0, \{a_t\}_{t=0}^T; forward) \]

The object poses from the real-world at timestep t = T are gathered with differentiable rendering denoted as P^O_T real. We then calculate the distance between the two poses.

\[ L_p = \| P^O_T - P^O_T \|_1 \]

Then object physical attributes ψ_M^phy (mass and inertial) are updated to minimize the loss L_p.

\[ ψ_M^phy \leftarrow ψ_M^phy - \lambda_M \frac{∂L_p}{∂ψ_M^phy} \]

Through this, we can keep reducing the discrepancy between the simulation and the real world, making the model more and more accurate. Up to now, we have introduced how to build and update the model.

C. Sim2Real: Learning Residual Policy in the Real World

We delay the discussion of how to learn the policy to complete the task T with the model in the simulation to the next subsection III-D. Here we assume that we have the policy π_{sim} in simulation which takes rendered images I_{sim} as inputs and outputs actions denoted as a_{sim} = π_{sim}(I_{sim}).

We describe how to apply the learned policy from simulation to the real world by learning a residual policy π_{res} to reduce the sim-to-real gap in this subsection.

We set the same camera pose denoted as P^c both in the simulation and the real world and get images denoted as I_{sim,0} and I_{real}. We first update the model’s parameter ψ_M by minimizing the loss with Eqn. 6 as described in Sec. III-B2 and then get the new images I_{sim} with the updated ψ_M.

With the I_{sim} and I_{real}, we get an action from simulation a_{sim}. Instead of directly applying the action a_{sim} in the real world, we train a residual policy that takes in the real image I_{real} and the a_{sim}, outputs the residual action δa_{real} = π_{res}(I_{real}, a_{sim})

Then actor model in the real world gets the action as follows.

\[ a_{real} = π_{real}(I_{real}, a_{sim}) = a_{sim} + δa_{real} \]

We follow the training process of residual policy described in [55]. The residual policy should never make a good initial policy worse. We therefore initialize the residual policy so that π_{res}(I_{real}, a_{sim}) = 0.
before training. We do this by initializing the last layer of the network to be zero. Once the real world action \( a_{\text{real}} \) is executed, we receive the next image \( I_{\text{real}}' \) and the reward

\[
r_{\text{real}} = \begin{cases} 1 & \text{succeed} \\ 0 & \text{otherwise} \end{cases}
\]  

(16)

The task will be considered done when it succeeds or exceeds the max action step number. To train the residual policy, we store the \((I_{\text{real}}, a_{\text{real}}, a_{\text{sim}}, r_{\text{real}}, done, I_{\text{real}}')\) to the reply buffer to update the \( \pi_{\text{res}} \) following a common deep reinforcement learning procedure TD3 [56], an actor-critic deep RL method.

D. Learn@Sim: Learning Sensing-aware Action in Simulation

With the model, we discuss how to learn the informative camera pose \( P^c \) associated with the rendered image \( I_{\text{sim}} \) and action \( a_{\text{sim}} \) to complete the given task \( T \) in the simulation.

1) Sensing-aware Q-function: We adopt the Q function \( Q^\text{sim}(I_{\text{sim}}, a_{\text{sim}}) \) to reflect the informative viewpoint. We can calculate the gradient of \( Q^\text{sim} \) with respect to the camera pose \( P^c \) as follows.

\[
\frac{\partial Q^\text{sim}(I_{\text{sim}}, a_{\text{sim}})}{\partial P^c} = \frac{\partial Q^\text{sim}(I_{\text{sim}}, a_{\text{sim}})}{\partial I_{\text{sim}}} \frac{\partial I_{\text{sim}}}{\partial P^c}
\]  

(17)

Here the first term \( \partial Q^\text{sim} / \partial I_{\text{sim}} \) is available through the backward propagation of the neural network. The second term \( \partial I_{\text{sim}} / \partial P^c \) can be obtained from the differentiable renderer [18]. The gradient \( \partial Q^\text{sim} / \partial P^c \) provides information on how to find a more informative viewpoint, which makes the pipeline sensing-aware. Under the more informative viewpoint, the actor has a better sense of the state to generate an action associated with a higher Q value. We verify our hypothesis by visualizing the learned Q function in Sec [IV-B2].

2) Learning Actor in Simulation to Reflect the Sensing Quality in the Real World: In this part, we discuss how we learn an actor \( \pi^\text{sim} \) in simulation, which takes rendered image \( I_{\text{sim}} \) and outputs the action \( a_{\text{sim}} \). There are multiple ways to learn the \( \pi_{\text{sim}} \) in the simulation via reinforcement learning or imitation learning. In this work, we choose the imitation learning approach to learn the \( \pi_{\text{sim}} \), which we find effective and efficient. With the model \( M \) built with the differentiable physics simulation [10], our pipeline generates trajectories completing the tasks \( \{(s_{t_{\text{sim}}}^\text{sim}, a_{t_{\text{sim}}}^\text{sim}|T)\}_{t=1}^T \) to train the actor in simulation \( \pi_{\text{sim}} \). To effectively generate the trajectories inside the differentiable physics simulation [10], we follow the method mentioned in [23]. Given task \( T \), we optimize the trajectory via the following equations.

\[
\mathcal{L}(T) = \sum_{t=0}^{T-1} l(s_{t+1}^\text{sim}, a_t^\text{sim}; T) \quad \text{s.t. } s_0^\text{sim} = s_{\text{init}} \quad \text{forward}
\]  

(19)

\[
s_0^\text{sim} = s_{\text{init}}
\]  

(20)

Here we explain the loss function in Eqn. [18]

\[
l(s_t^\text{sim}, a_t^\text{sim}; T) = \begin{cases} \|a_t^\text{sim}\|^2 & t < T_1 - 1 \\ \|s_{t_{\text{sim}}}^\text{sim} - \mathcal{G}(T)\|_1^2 & t = T_1 - 1 \\ \|a_t^\text{sim}\|^2 & T_{n-1} - 1 < t < T_n - 1 \\ \|s_{t_{\text{sim}}}^\text{sim} - \mathcal{G}(T)\|_2^2 & t = T_n - 1 \end{cases}
\]  

(22)

\( \mathcal{G}(T) \) is the goal of the task. To make it easier to generate the trajectories, we define \( n \) sub-goals for each task. In practice, instead of directly optimizing the whole trajectory, the pipeline will optimize the trajectory to achieve each sub-goal in sequence. Once a goal is achieved, the pipeline will move on to the next goal.

After gathering these trajectories \( \{(s_t^\text{sim}, a_t^\text{sim}|T)\}_{t=1}^T \) in the simulation, we rendered multiple images \( \{I_{t}^\text{sim}(P^c,i)\} \) for each state \( s_t^\text{sim} \) under different camera poses \( \{P^c,i\} \). We train an actor \( \pi^\text{sim}(I_t(s_{t_{\text{sim}}}^\text{sim}, P^c,i)) \) to predict the action \( a_t^\text{sim} \) to imitate the corresponding action \( a_{t_{\text{sim}}}^\text{sim} \). In the process, the prediction error is defined as follows.

\[
\mathcal{L}^\text{sim} = \|a_t^\text{sim} - a_{t_{\text{sim}}}^\text{sim}\|_2
\]  

(23)

During the learning process, the prediction error also depends on the sensing quality. Our hypothesis is the actor might not learn the effective ground-truth action if there is insufficient state information in the rendered image. For example, if the camera is always looking into the sky and the object does not appear in the rendered image, it will be difficult for the actor to learn the correct action. We visualize the prediction error in Sec [IV-B2]. Note that the learning process also generates success and failure trajectories, which are used to learn the Q function.

3) Learning Sensing-aware Q Function: Here we describe how to learn the sensing-aware Q function denoted as \( Q^\text{sim}(I_{\text{sim}}, a_{\text{sim}}) \). There are also multiple ways to learn the Q function. In this work, we formulate it as a supervised learning problem. We roll out the trajectories in the simulation to generate the training data. Given a trajectory \( \{(I_t^\text{sim}(P^c,i), a_t^\text{sim})\}_{t=1}^T \), we calculate the return \( R_t = \sum_{i=t}^{T} \gamma^{T-i} r_i \) associated with each image and action pair, where \( r_i \) is the reward with same definition as Eqn. [16] and \( \gamma \) is the discount factor. We then train a deep network that takes \( I_{t_{\text{sim}}}^\text{sim} \) and \( a_t^\text{sim} \) as inputs and outputs the Q value by minimizing the following loss.

\[
\mathcal{L}^Q = \|Q^\text{sim}(I_{t_{\text{sim}}}^\text{sim}, a_t^\text{sim}) - R_t\|_2
\]  

(24)

4) Selecting Perception and Action Leveraging the Actor and Q-function: Starting from an initial viewpoint \( P^c \), the simulation rendered an image \( I_{\text{sim}} \). We feed the image into the actor model to get an action \( a_{\text{sim}} = \pi_{\text{sim}}(I_{\text{sim}}) \), and then send the image and action to the Q function to get the value \( Q^\text{sim}(I_{\text{sim}}, a_{\text{sim}}) \). We get the gradient with respect to the \( P^c \) and update the camera pose as follows.

\[
P^c' = P^c + \lambda \frac{\partial Q^\text{sim}(I_{\text{sim}}, a_{\text{sim}})}{\partial P^c}
\]  

(25)
Algorithm 1: Overall algorithm

Input: the model $M$ with its model parameters $\psi_M$, camera pose $P$, take the real image using real camera $\text{Cam}(P)$

for $t$ in each iteration do

$T^{\text{sim}} \leftarrow M(\psi_M, P; \text{render})$; $T^{\text{real}} \leftarrow \text{Cam}(P)$;

Update $\psi_M$ by reducing $\|T^{\text{sim}} - T^{\text{real}}\|$;

$T^{\text{sim}} \leftarrow M(\psi_M, P; \text{render}), a^{\text{sim}} = \pi^{\text{sim}}(T^{\text{sim}})$;

while True do

$P' = P + \lambda_P \nabla Q(T^{\text{sim}}, a^{\text{sim}})$;

$T^{\text{sim}} \leftarrow M(\psi_M, P'; \text{render})$; $T^{\text{real}}' \leftarrow \text{Cam}(P')$;

Update $\psi_M$ by reducing $\|T^{\text{sim}} - T^{\text{real}}'\|$;

$T^{\text{sim}} \leftarrow M(\psi_M, P'; \text{render}), a^{\text{sim}} = \pi^{\text{sim}}(T^{\text{sim}})$;

if $Q^{\text{sim}}(T^{\text{sim}}, a^{\text{sim}}) \leq Q^{\text{sim}}(T^{\text{sim}}', a^{\text{sim}}')$ then

$T^{\text{real}} \leftarrow T^{\text{real}}'$; $a^{\text{real}} \leftarrow a^{\text{sim}}'$;

break;

end

$T^{\text{sim}} \leftarrow T^{\text{sim}}$; $P \leftarrow P'$, $a^{\text{sim}} \leftarrow a^{\text{sim}}'$;

end

Add the updated Model with $\psi_M$ to data buffer to generate trajectories:

Update the actor and Q function:

$a^{\text{real}} = \pi^{\text{real}}(T^{\text{real}}, a^{\text{sim}})$;

Execute the action $a^{\text{real}}$ both in the real world and in simulation;

Add the transition data into the replay buffer to $\text{sim2real}$;

Update the residual policy

end

With the new camera pose $P'$, we gather the new rendered image $T^{\text{sim}}'$, action $a^{\text{sim}}' = \pi^{\text{sim}}(T^{\text{sim}}')$, and the associated $Q^{\text{sim}}(T^{\text{sim}}', a^{\text{sim}}')$. We accept the new camera pose $P'$ and new action $a^{\text{sim}}'$ if $Q(T^{\text{sim}}, a^{\text{sim}}) \geq Q(T^{\text{sim}}, a^{\text{sim}}')$.

E. Overall Learning Process

We summarize the overall test stage pipeline in the following Alg. 1. Given an initial model, we leverage the differentiable rendering and simulation to update the model parameter $\psi_M$ and select the camera pose (Line 1-15).

Our algorithm add the update model $M$ into our data buffer. Then our framework starts to generate the trajectories to accomplish the according task starting from the state described by the model $M$ associated with the parameter $M$. (Line 16)

We calculate the return value and expert actions to train the actor network and Q network in simulation. (Line 17)

Then we train the residual policy to deal with the $\text{sim2real}$ gap. (Line 18-21)

IV. EXPERIMENTS

In this section, we introduce our experimental setup and conduct quantitative and qualitative evaluations to demonstrate the effectiveness of our approach. Our experiments focus on evaluating the following questions:

• How effective is each component in our proposed SAM-RL algorithm?
• How effective is the model update process leveraging the differentiable simulation and rendering?

A. Experimental Setup

1) Real World: We set up the real world experiment with two 7 DoF robotic arms as shown in Fig. 1. One robot is the Franka Panda performing the manipulation task. The other robot is Flexiv Rizon [25] holding the RGB-D RealSense camera. We calibrate the camera intrinsic matrix and the hand-eye transformation between the camera and the Flexiv’s end-effector. We also measure the relative transformation between two robots’ bases in the world coordinate.

With the new camera pose $P'$, we gather the new rendered image $T^{\text{sim}}'$, action $a^{\text{sim}}' = \pi^{\text{sim}}(T^{\text{sim}}')$, and the associated $Q^{\text{sim}}(T^{\text{sim}}', a^{\text{sim}}')$. We accept the new camera pose $P'$ and new action $a^{\text{sim}}'$ if $Q(T^{\text{sim}}, a^{\text{sim}}) \geq Q(T^{\text{sim}}, a^{\text{sim}}')$.

2) Simulation: In the simulation, we set the camera intrinsic matrix and relative transformations of the camera to Flexiv and Flexiv to Franka based on the real-world calibration results. We use PyBullet [57] to simulate the real world, which is different from our differentiable physics simulation for quantitative experiments. Objects are initialized with a random pose within manually defined bounds as described following.

3) Differentiable Physics Simulation and Rendering: We combine the differentiable physics-based simulation and rendering to model the real world, update objects attributes, calculate expert trajectories, and optimize camera pose. We use NimblePhysics [10] as differentiable simulation and Redner [18] as differentiable renderer.

4) Manipulation Tasks: We design three different manipulation tasks as shown in Fig. 3 and the goal of each task as shown in Fig. 3.

Peg-Insertion, which is inserting a peg into a hole. We assume the robot holds the peg tightly, so the state of the task is the translation of the peg, which is 3-dimension and could be calculated via robot forward kinematic. And the hole is initialized at a random location by $10cm \times 10cm$ both in simulation and real world. The task is considered successful if the peg is inserted into the hole. Our pipeline adopt an automatic termination function in real world by comparing
the distance between the current state and goal state of the peg. The max number of action steps is 100.

**Spatula-Flipping**, which is flipping a pancake with a spatula. The state of the task is the translation of the pancake, which is 3-dimension. The pancake is initialized at a random location by $2cm \times 2cm$ both in simulation and real world. The Spatula-Flipping is successful if the pancake is lifted up by the spatula and then flipped. We track the position of the pancake via the method mentioned in Sec. III-B2. Our pipeline also adopt an automatic termination function for Spatula-Flipping in real world by comparing the distance between the current state and goal state. The max number of action steps is 200.

**Needle-Threading**, which is threading a needle. In simulation, the thread is simulated using 10 links. There are two revolute joints (See Fig. 5) between the links next to each other. The state of the task is the position of each link of the thread, which is 30-dimension in total. In the real world, the thread is manually randomly bent to initialize. While in simulation, for each revolute joint described in Fig. 5, the state of the joint is initialized ranging from $-10^\circ$ to $10^\circ$. We manually decide success for Needle-Threading, because it is hard to detect whether the thread is through the needle hole automatically with high precision. The max number of action steps is 100.

![Fig. 5. There are two revolute joints between links, rotate along the x and y axis (blue and red).](image)

5) **Training Details:** The size of the input images is $128 \times 128 \times 6$ (RGB-XYZ), while the size of the action space is 3, we only enable the translation of the Franka’s end-effector and disable the rotation. For both the actor and critic (Q function) of $\pi_{sim}$ and $\pi_{res}$, the network contains a CNN feature extractor and a MLP head. The feature extractor has 5 layers to extract a 128-dimensional vector from images, while the MLP head has 3 layers takes in the extracted vector, outputs the action or $Q$ value. Note that other inputs, like the predicted action for the critic network or the base action for residual actor, will be concatenated with the images and input to the feature extractor.

In real world experiments, we train the residual policy network for 100 episodes for Needle-Threading and 10 episodes for Peg-Insertion and Spatula-Flipping. To better reduce the sim-to-real gap, we also augment the training image $I_{sim}$ by adding noise to the RGB and XYZ value of each pixel to imitate the sensor noise.

### B. Evaluating the Sensing-aware Function

1) **Quantitative Results:** To evaluate the learned sensing-aware $Q$ function during Learn@Sim stage, we set up the following experiments inside the differentiable physics simulation. We use the same actor model $\pi_{sim}$ trained with rendered images under multiple camera views, and evaluate the actor with and without leveraging the sensing-aware $Q$ function to optimize the camera pose during manipulation processes, denoted as Ours and Ours(w/o pose-opt), respectively. We also train the same actor with rendered images under a fixed camera view, and evaluate the trained actor under the same camera view, denoted as Fixed-View. To make relatively fair compassion, we train the actor in Fixed-View to have the same prediction error in the training process as in Ours. We report the task success rate based on 100 experiments under these three settings. As shown in Tab. I, our pipeline can find the informative view to benefit robot manipulation with the sensing-aware $Q$ function.

<table>
<thead>
<tr>
<th>Task</th>
<th>Ours</th>
<th>Ours(w/o pose-opt)</th>
<th>Fixed View</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peg Insertion</td>
<td>0.82</td>
<td>0.64</td>
<td>0.71</td>
</tr>
<tr>
<td>Spatula Flipping</td>
<td>0.65</td>
<td>0.57</td>
<td>0.55</td>
</tr>
<tr>
<td>Needle Threading</td>
<td>0.70</td>
<td>0.46</td>
<td>0.66</td>
</tr>
</tbody>
</table>

2) **Qualitative Visualization:** We visualize the learned $Q$ function (Fig. 6(a)) and the prediction error of the learned imitation learning policy (Fig. 6(b)) for the Needle-Threading task. We render images $I_{sim}(P_{c,i})$ from multiple view points and send them to the actor $\pi_{sim}$ to get associated actions $\pi_{sim}(I_{sim}(P_{i}))$, and get the $Q$ values $Q_{sim}(I_{sim}(P_{c,i}), \pi_{sim}(I_{sim}(P_{c,i})))$. It indicates that our $Q$ function learns a reasonable policy to select the camera viewpoint.

3) **Salient Map of Actor Network:** In order to have a better understanding about the actor network, we generate the salient map of the Needle-Threading task as shown in Fig. 7. Given an input image denoted as $I_{sim}$, we calculate the prediction error denoted as

$$L_s = \| \hat{a}_{sim} - a_{sim} \|_2$$  \hspace{1cm} (26)

where $\hat{a}_{sim}$ and $a_{sim}$ are the predicted action and ground-truth action, respectively. We visualize the gradient $\partial L_s/\partial I_{sim}$ as

![Image](image)
(a) Visualization of the learned Q function.

(b) Visualization of the prediction error of the learned imitation learning policy.

Fig. 6. Qualitative visualization of sensing-aware Q function for the Needle Threading. Each pixel corresponds to a yaw and pitch value of the camera view.

the salient map. Fig. 7 indicates that the network pay more attentions to the thread’s region, including the curved thread region and contact region between the thread and the needle.

(a) Input Image

(b) Salient Map

Fig. 7. The input image of the actor network and the corresponding salient map from the actor network

C. Comparing the SAM-RL with Model-free and Model-based Deep RL

We compare our SAM-RL with common model-free deep RL algorithm TD3 [56], SAC [58] and model-based deep RL algorithm Dreamer [6]. In this experiment, we adopt pybullet as the “real world”. For TD3, SAC, and Dreamer, we directly train the robot in pybullet. The observation including an RGB-XYZ image and the position of the robot’s end-effector. The action space is the translation of the robot’s end-effector, which is 3 dimensions. The reward function of Peg-Insertion is the distance between the peg and the hole. The reward function of Needle-Threading is the distance between the thread and needle hole. The reward function of Spatula-Flipping is the position of the pancake among z-axis which is perpendicular to the ground. For SAM-RL, we develop the model via the differentiable physics-based simulation and rendering and use the model to complete tasks in pybullet (“real world”). Every time the environment is initialized or reset, the position and shape of the thread are randomly set within a certain range as described in Sec. IV-A. Fig. 8(a), Fig. 8(b), and Fig. 8(c) show the average success rate of the SAM-RL, SAC, TD3, and Dreamer during the training stage. Take Needle-Threading as an example, after training 100k steps, the average success rate of SAC models and TD3 models are about 50%, while Dreamer is about 60%. However, our pipeline achieves a success rate of around 80% after 8k training steps. It indicates that our proposed SAM-RL is significantly sample-efficient compared to model-free deep reinforcement learning approaches and existing model-based method.

D. Ablation: Evaluating Components of SAM-RL

We apply SAM-RL in the real robot system and conduct two ablation study experiments.

- We remove the component of updating the model \( \mathcal{M} \) by comparing \( T_{\text{real}} \) and \( T_{\text{sim}} \) and denote the experiment as SAM-RL (w/o update modeling).
- We remove the residual policy used to address the sim2real gap and directly execute the predicted action \( a_{\text{sim}} \) in the real world denoted as SAM-RL (w/o residual policy).

Note that SAM-RL (w/o residual policy) directly apply \( \pi_{\text{sim}} \) in real world, have no need to train, so it has a constant success rate. We post the result of Needle Threading in Fig. 8(d). It shows that these components play important roles in our pipeline, and removing these components results in significant performance decreases.
E. Evaluating the Model Updating via Differentiable Simulation and Rendering

1) Pose and Texture: With differentiable simulation and rendering, we can update the texture of the model $M$. And also, we can update the pose of the model $M$ during the manipulation. For Peg-Insertion, the model’s pose is the translation of the hole on the table, which is 2-dimension ($x$ and $y$). We only need to update this at the beginning of each episode because the location of the hole is randomly initialized and fixed during manipulation. For Spatula-Flipping, The model’s pose is 5-dimension, include the translation of the pancake, which is 3-dimension ($x$, $y$, and $z$), and rotation of the pancake, which is 2-dimension (rotate among $x$-axis and $y$-axis) as the pancake is symmetry among the $z$-axis. For Needle-Threading, the thread is simulated using 10 links as described in Sec. IV-A. There are two revolute joints (See Fig. 5) between the links next to each other. So the model’s pose is the states of the joints, which is 20-dimension in total.

As shown in Fig. 9, we demonstrate the mean $L_1$ distance between the estimated and ground-truth position of the thread’s links for Needle-Threading, and the $L_1$ distance between the estimated and ground-truth position of the pancake. To get the ground-truth position of the thread and pancake, we use the image rendered by Pybullet rendering, which is different from the differential rendering, as the ground-truth image. It indicate that our pipeline can reduce the distance between the estimated and the ground-truth state and texture by model updating via differentiable rendering. The first row shows the experiment where the thread (in the left most image) has a different position and different color from the ground truth image (as shown in the right most). The second row reflects the experiment where the thread has the same color but different position from the ground truth image. The last row shows the experiment where the pose of the pancake is different from the ground truth image.

Note that to better evaluate the performance of the model updating, we make the distance between the initialization and ground-truth larger. So it takes 200 iterations for Needle-Threading and 100 iterations for Spatula-Flipping to optimize. In practice, one action step can only cause a little change to the thread and pancake, so we only take 20 iterations for the thread updating and 10 iterations for the pancake updating at each action step.

2) Physical Attributes: Next we describe how to update objects’ physical attributes (mass and inertial) in detail. We still use Pybullet to simulate the real world. We first use the spatula to push the pancake for $T_1$ time steps, then we stop moving the spatula and the pancake keeps sliding until time step $T_2$ (See Fig. 10). We use the robot to manipulate the spatula in torque control mode. So with the same robot action, the impulse provided by the robot is constant, and the pancake pose at time step $T_2$ is influenced by the mass value of the pancake. We update the object’s physical attributes (mass and inertial) via the differentiable physics simulation. As shown in the learning curve, the relative error of the pancake’s mass value can be reduced from 50% to around 20% after the optimization.

3) Real World: We also demonstrate the model updating via differentiable rendering in the real world. (See Fig. 11)

V. CONCLUSION

We propose a sensing-aware model-based reinforcement learning called SAM-RL leveraging the differentiable physics-based simulation and rendering. SAM-RL automatically updates the model by comparing the raw observations between simulation and the real world, and produces the policy efficiently. It also allows robots to select an informative viewpoint to better monitor the task process. We apply the system to
robotic assembly, tool manipulation, and deformable object manipulation tasks. Extensive experiments in the simulation and the real world demonstrate the effectiveness of our proposed learning framework.

VI. ACKNOWLEDGEMENT

This work was in part supported by the National Key R&D Program of China (2021ZD0110700), Shanghai Municipal Science and Technology Major Project (2021SHZDZX0102), Shanghai Qi Zhi Institute, Shanghai Science and Technology Major Project (2021SHZDZX0102), and a startup grant from National University of Singapore.

REFERENCES


