Differential Dynamic Programming for Graph-Structured Dynamical Systems: Generalization of Pouring Behavior with Different Skills

Akihiko Yamaguchi[1] and Christopher G. Atkeson[1]

***Abstract*— We explore differential dynamic programming for dynamical systems that form a directed graph structure. This planning method is applicable to complicated tasks where subtasks are sequentially connected and different skills are selected according to the situation. A pouring task is an example: it involves grasping and moving a container, and selection of skills, e.g. tipping and shaking. Our method can handle these situations; we plan the continuous parameters of each subtask and skill, as well as select skills. Our method is based on stochastic differential dynamic programming. We use stochastic neural networks to learn dynamical systems when they are unknown. Our method is a form of reinforcement learning. On the other hand, we use ideas from artificial intelligence, such as graph-structured dynamical systems, and frame-and-slots to represent a large state-action vector. This work is a partial unification of these different fields. We demonstrate our method in a simulated pouring task, where we show that our method generalizes over material property and container shape. Accompanying video: `https://youtu.be/_ECmnG2BLE8`**

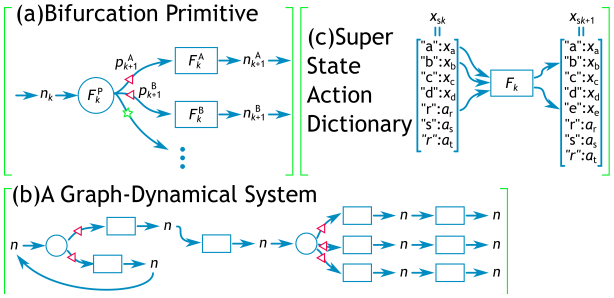

Fig. 1. (a) A primitive of graph-structured dynamical systems. We omit a bifurcation model $\mathbf{F}_k^{\mathrm{P}}$ when the number of branches is one. (b) An example of a graph-structured dynamical system. Circles denote bifurcations, boxes denote component dynamical systems, and n denotes a node. The marks on edges after bifurcation models of (a) and (b) denote *groups*. Only a single edge in a group can actually happen. (c) An example of mapping from a super-state-action dictionary (SSA) to another SSA.

I. Introduction

In order to create robots that can handle complicated tasks such as manipulation of liquids (e.g. pouring), we explore a form of policy optimization for directed-graph-structured dynamical systems that involve continuous parameter adjustments and selections of discrete strategies. For example, a pouring behavior model can be graph structured with transitions such as: grasping a source container, moving it to the location of a receiving container, and pouring with a skill such as tipping, shaking, and squeezing. We also consider learning dynamical models when we do not have good analytical models, as in liquid flow. A practical benefit of our method is that in a case study of pouring [1], the behavior could generalize more.

More concretely we consider a dynamical system that has a graph structure (Fig. 1 (a) and (b)). Each node has a state vector $\mathbf{x}$, and each edge has a dynamical system that takes $\mathbf{x}$ and a continuous action vector $\mathbf{a}$ as an input and outputs a state vector $\mathbf{x}'$ of the next node. The structure may involve bifurcations; an actual next node is decided by $\mathbf{x}$, $\mathbf{a}$, and a selection variable s. We consider a bifurcation as a discrete probability distribution model. A graph-structured dynamical system has a start node, and one or more terminal nodes that give rewards. When a dynamical system is partially or totally unknown, we use learning methods to construct dynamical models and bifurcation models. Our problem is to find all actions $\{\mathbf{a}\}$ and selections $\{s\}$ used in the graph-structured dynamical system so that the expected sum of rewards is maximized. As far as we know, such an optimization method has not been proposed yet. Note that this problem formulation includes dynamic programming (DP), differential dynamic programming (DDP), model predictive control (MPC), and so on as subclasses.

In order to solve this problem, we first transform the graph structure into a tree structure; i.e. if the graph structure involves loops, they are unrolled. For the optimization of continuous action vectors, we reformulate a stochastic version of DDP [2]. DDP is a gradient-based optimization algorithm. DDP is applicable since we can propagate the state vectors along the tree dynamical structure and get the rewards, and then we calculate gradients with respect to state and action vectors by propagating backward through the tree from terminal nodes. For the optimization of discrete selections, we combine DDP with a multi-start local search. A multi-start local search is also useful to avoid poor local maxima. Since we use learned dynamical models, there might be many local maxima.

We also introduce a *super-state-action* dictionary (SSA) that is our specific version of *frames and slots* used in traditional artificial intelligence [3]. An SSA contains various types of information for each node, such as container positions, material properties, current and target amounts, robot configuration, and actions such as target joint angles. These variables are stored with their labels. Each edge dynamical system maps a part of an SSA to a part of a new SSA where new slots may be added (Fig. 1 (c)). The benefits of SSA in our research are: (1) We can reduce modeling or learning costs since edge dynamical systems usually use a small part of whole SSA. (2) We can naturally represent discontinuous changes in dynamics (e.g. flow appears after pouring starts). (3) The reusability of component dynamical systems might be increased.

We explore our method in simulated pouring experiments. The simulator creates variations of the source container

[1]A. Yamaguchi and C. G. Atkeson are with The Robotics Institute, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh PA 15213, United States `info@akihikoy.net`

shape and the poured material property. In this simulator, using different pouring skills such as tipping and shaking is necessary. With these experiments, we show that our method can select skills as well as optimizing continuous parameters for given situations. Our method achieved a generalization of pouring over container shapes and material properties.

Related Work

The features of the proposed methods are: (1) Differential dynamic programming (DDP) for graph-structured dynamical systems. (2) Model-based reinforcement learning for hierarchical dynamic systems. (3) Super-state-action dictionary (SSA) to represent complicated systems. Thus we mention the related work in the fields of DDP, reinforcement learning, and artificial intelligence.

There is a great deal of work on DDP ([2]). Some of it is stochastic (e.g. [4]) similar to ours. Usually DDP methods use a second-order gradient method (e.g. [4], [5]), while we use a first-order algorithm [6]. Our approach is simpler to implement. Previous DDP methods consider linear-structured dynamical systems (including a single loop structure) while we consider graph-structured dynamical systems.

In general our problem is a reinforcement learning (RL) problem. Although a current popular approach of RL in robot learning is a model-free approach (cf. [7]), especially direct policy search (e.g. [8], [9]), there are many reasons to use model-based RL. In model-based RL, we learn a dynamical model of the system, and apply dynamic programming such as DDP (e.g. [10], [11], [12]). The advantages of the model-based approach are: (A) Generalization ability: in some cases, a model-based approach generalizes well (e.g. [13]). (B) Reusability: learned models can be commonly used in different tasks. (C) Robustness to reward changes: even when we modify the reward function, we can plan a new policy with learned models without new physical practice. A major issue of the model-based approach is so-called *simulation biases* [7]; the modeling error accumulates rapidly during time integrals. In our approach, we learn (sub)task-level dynamic models according to the idea of task-level robot learning [14], [15]. We learn the relation between input and output states of a subtask which often does not require time integrals in forward estimation. In addition, we use probabilistic representations for dealing with modeling errors. Our strategies are described in detail in [16]. This paper is following these strategies, and extending them to graph-structured dynamical systems.

Recently deep learning neural networks (DNN) have become popular. Many researchers are investigating their application to reinforcement learning (e.g. [12], [17], [18]). Similar to our approach, DeepMPC uses neural networks to learn models [12]. A notable difference of our approach is that we learn task-level dynamical systems that are robust to the simulation-bias issue. In addition, in our previous work, we introduced a probabilistic computation into neural networks [18]. In this paper we use this extension to learn dynamical models when they are unknown.

The idea of SSA is found in traditional artificial intelligence (AI) literature (e.g. [3]), known as *frames and slots*. Similarly, the idea of graph-structured dynamical systems is also known in the AI field [19]. Our contribution to this field is the introduction of the numerical approaches (DDP, RL, DNN) to the AI methods, and its application to a practical robot learning task (pouring).

II. Differential Dynamic Programming for Graph-Structured Dynamical Systems

For a given graph structure of a dynamical system, models of edge dynamical systems and bifurcations, and the current states, we plan continuous action vectors and discrete selections used in the entire dynamical system. The planning algorithm is based on stochastic differential dynamic programming (DDP; [2]). DDP assumes a linear structure of dynamical system (including a single loop structure); we extend it to a graph structure. We refer to our algorithm as Graph-DDP. Graph-DDP optimizes a value function which is an expected sum of rewards with respect to continuous action vectors and discrete selections. We use the ideas of the original DDP, a gradient-based optimization, to plan the continuous action vectors. We combine DDP with a multi-start gradient-based optimization in order to (1) plan discrete selections, and (2) avoid poor local optima that often exist especially when we use learned dynamical models (e.g. using neural networks).

The Graph-DDP method consists of a graph structure analysis, forward and backward computation to obtain the value function and its gradient with respect to the continuous action vectors, and multi-start gradient-based optimization.

A. Problem Formulation

For the convenience of calculation, we represent a graph structure of a dynamical system as a set of *bifurcation primitives*. As illustrated in Fig. 1 (a), each bifurcation primitive has an input node n_k and multiple output nodes $\{n^b_{k+1}\}$ where b is a label of each branch. There is an edge dynamical system $\mathbf{F}^b_k$ between each pair of n_k and n^b_{k+1}. The probability p^b_{k+1} of actually going through a branch b is modeled by $\mathbf{F}^{\mathrm{P}b}_k$.

Although we refer to $\mathbf{F}^b_k$ as a model of an edge dynamical system, we can use a kinematic model or a reward model instead of a dynamical model. The following calculations do not change for these types of models. When an output node n^b_{k+1} does not have a succeeding bifurcation or dynamical system, we refer to it as a *terminal node*. We assume that terminal nodes have a reward in their states. That is, when n^b_{k+1} is a terminal node, the corresponding $\mathbf{F}^b_k$ is a reward model.

In order to increase the representation flexibility, we consider *groups* of branches (Fig. 1(a)(b)). Only a single branch per group can actually happen; while, branches of different groups may actually happen simultaneously. This idea is useful when defining a reward bifurcation (one branch is for a reward calculation, and another branch is for succeeding processes). In branches of the same group, the sum of transition probabilities must be one. We have to consider this constraint when we learn the bifurcation models, but the planning calculation may omit that. Thus in this section on planning, we do not denote the groups explicitly.

Each node has a super-state-action dictionary (SSA). Although SSA dramatically increases the computation and learning efficiency, the planning calculation becomes complicated. In this section, we consider a serialized vector $\mathbf{x}_k$ for simplicity; $\mathbf{x}_k$ contains all values (states, actions, and

selections) in an SSA. Since we apply a stochastic DDP, we consider a normal distribution of $\mathbf{x}_k$; $\mathbf{x}_k \sim \mathcal{N}(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$ where $\boldsymbol{\mu}_k$ is a mean vector and $\boldsymbol{\Sigma}_k$ is a covariance matrix. We give special treatment to selections in $\mathbf{x}_k$ as they are discrete values: we consider they are non-probabilistic (deterministic) values, and their gradients are zero (i.e. DDP does not update them). In the following, for simplicity, we use a deterministic notation for $\mathbf{x}_k$, but its extension to the probabilistic form $\mathcal{N}(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$ is straightforward.

The objective of Graph-DDP is maximizing an expected sum of rewards with respect to the actions and selections used in the dynamical system; i.e. we solve

$$J_0(\mathbf{x}_0) = \mathrm{E}[\textstyle\sum_{n \in terminal\ nodes} r_n] \tag{1}$$

$$\mathrm{w.r.t.}\mathbf{A}, \mathbf{S} \tag{2}$$

where $\mathbf{A}$ and $\mathbf{S}$ are concatenated vectors of all actions and selections, i.e. $\mathbf{A} = [\mathbf{a}_0, \mathbf{a}_1, \dots]$, $\mathbf{S} = [s_0, s_1, \dots]$. Here we assume that $\mathbf{x}_0$ contains all of $\mathbf{A}$ and $\mathbf{S}$. This notation might not be a common style of DDP, but it makes the planning calculation simpler, and the computational efficiency is improved by using the SSA calculus. We assume that there is no hidden state in $\mathbf{x}_0$.

We assume that the model of an edge dynamical system $\mathbf{F}_k^b$ gives a prediction of the next SSA as a probabilistic distribution, and an expected derivative with respect to the input SSA mean. Similarly, $\mathbf{F}_k^{\mathrm{P}}$ gives expected probabilities of branches, and an expected derivative with respect to the input SSA mean.

B. Graph-Structure Analysis

The analysis of a graph structure involves two steps: (1) unrolling the loops (i.e. transforming the structure to a tree structure), and (2) deciding the backward computation order. Note that the obtained tree and the backward computation order is kept during the DDP iterations.

For unrolling the loops, we define a maximum number of visitations per node. We use a breadth-first search to traverse the graph structure until the number of visitations reaches the limit or all nodes are visited at least once. Consequently we obtain a tree structure.

The backward computation order is used to compute gradients in the backward direction from terminal nodes. Similar to breadth-first search, we traverse the tree backwards. However we keep the condition that when computing about a node n_k, all its succeeding branches $\{n_{k+1}^b\}$ must be computed in advance.

C. Forward and Backward Computation

Since DDP is an iterative algorithm, we have current (tentative) values of $\mathbf{A}$ and $\mathbf{S}$. The forward and the backward computations are done with these current values. Starting from a start node, we propagate SSA to the terminal nodes by traversing the tree, and we compute the gradients of the value function with respect to SSA in the backward computation order. In the following, we assume that only the bifurcation models $\mathbf{F}_k^{\mathrm{P}b}$ have probabilistic distributions, and treat $\mathbf{x}_k$ as deterministic variables instead of $\mathcal{N}(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$. For the extension from $\mathbf{x}_k$ to $\mathcal{N}(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$, refer to stochastic DDP papers (e.g. [20]).

We use a breadth-first search to traverse the tree. In every visitation of a non-terminal node n_k, we compute the bifurcation model and the dynamical models of branches. Let $\mathbf{x}_k$ denote the SSA at n_k. For each branch b, we compute the output SSA $\mathbf{x}_{k+1}^b$ and the transition probability p_{k+1}^b: $\mathbf{x}_{k+1}^b = \mathbf{F}_k^b(\mathbf{x}_k)$, $p_{k+1}^b = \mathbf{F}_k^{\mathrm{P}b}(\mathbf{x}_k)$. We also compute their gradients with respect to $\mathbf{x}_k$: $\frac{\partial \mathbf{F}_k^b}{\partial \mathbf{x}_k}$, $\frac{\partial \mathbf{F}_k^{\mathrm{P}b}}{\partial \mathbf{x}_k}$. This forward propagation is calculated from a start node to terminal nodes.

We use the backward computation order to traverse the tree from terminal nodes. We assume that each node has a value function $J_k(\mathbf{x}_k)$ that estimates an expected sum of succeeding rewards. Such a value function is back-propagated from terminal nodes, and eventually we have a value function $J_0(\mathbf{x}_0)$ for the start node. The back-propagation at a bifurcation is given by:

$$J_k(\mathbf{x}_k) = \mathrm{E}\Big[\sum_b J_{k+1}^b(\mathbf{x}_{k+1}^b)\Big] = \sum_b \mathbf{F}_k^{\mathrm{P}b}(\mathbf{x}_k) J_{k+1}^b(\mathbf{x}_{k+1}^b) \tag{3}$$

$$= \sum_b \mathbf{F}_k^{\mathrm{P}b}(\mathbf{x}_k) J_{k+1}^b(\mathbf{F}_k^b(\mathbf{x}_k)). \tag{4}$$

When n_{k+1} is a terminal node, $\mathbf{F}_k^b$ is a reward model and $\mathbf{x}_{k+1}^b$ has a reward. In that case, we define J_{k+1}^b as the expectation of the reward.

In DDP, we use a gradient of J_0 with respect to $\mathbf{x}_0$, which is calculated with a chain rule of derivatives. The chain rule back-propagates the gradients from the terminal node. The back-propagation at a bifurcation is given by:

$$\frac{\partial J_k}{\partial \mathbf{x}_k} = \sum_b \frac{\partial}{\partial \mathbf{x}_k}\Big[\mathbf{F}_k^{\mathrm{P}b}(\mathbf{x}_k) J_{k+1}^b(\mathbf{F}_k^b(\mathbf{x}_k))\Big] \tag{5}$$

$$= \sum_b \Big[\frac{\partial \mathbf{F}_k^{\mathrm{P}b}}{\partial \mathbf{x}_k} J_{k+1}^b + \mathbf{F}_k^{\mathrm{P}b} \frac{\partial \mathbf{F}_k^b}{\partial \mathbf{x}_k} \frac{\partial J_{k+1}^b}{\partial \mathbf{x}_{k+1}^b}\Big]. \tag{6}$$

When n_{k+1} is a terminal node, $\frac{\partial J_{k+1}^b}{\partial \mathbf{x}_{k+1}^b}$ is 1. Finally we obtain $\frac{\partial J_0}{\partial \mathbf{x}_0}$. This contains gradients of J_0 with respect to $\mathbf{A}$ (a serialized vector of all continuous actions) which is used in the gradient-based optimization to update $\mathbf{A}$.

D. Multi-start Gradient-based Optimization

We describe the whole optimization process of Graph-DDP. First we analyze the graph structure, and obtain a corresponding tree structure of the graph-structured dynamical system and the backward computation order. Second, we apply a multi-start gradient-based optimization to optimize the continuous action vectors and the discrete selections. This is an iterative process: in the initialization, we generate multiple starting points (initial guess) including different values of discrete selections. In each iteration for each starting point, we compute the forward and backward propagations, and update the continuous action vectors with a gradient-based optimization method. Specifically we use ADADELTA [6].

In the initial guess, we generate starting points in two ways: (1) randomly choosing from a database that is storing all samples used in the past, and (2) randomly generating continuous action vectors from a uniform distribution or a Gaussian distribution, and discrete selections from a uniform distribution. The obtained points are stored in a start-point-queue.

We use multi-process programming in the iteration process. Each process has a different starting point popped from the start-point-queue, and updates the continuous action vectors independently from the other processes. The discrete

selections are not updated in the iterations. The iterations of each process stops when (1) a convergence criterion is satisfied, (2) the number of iterations exceeds a limit, or (3) an oscillation is detected. In case (1), the converged point is appended into a finished-point-list. In cases (2) and (3), the point is appended into the finished-point-list only when its value is greater than the best value. In any cases, the point is pushed onto the start-point-queue. The whole multi-process optimization is terminated when (a) the number of points in the finished-point-list satisfies a termination criterion, (b) the start-point-queue is empty, or (c) the total number of iterations reaches a limit.

III. Super-State-Action Dictionary (SSA)

In an implementation, a super-state-action dictionary (SSA) would be represented by an associative array; e.g. a map container in C++, a dictionary in Python, and a hash in Perl. The keys of SSA are *labels* that distinguish the types of elements in SSA; e.g. "source container position", "current joint angles", and "target joint angles".

The Graph-DDP described in the previous section considers a serialized vector for each SSA. Using the dictionary form of SSA, we can obtain the benefits as mentioned in the introduction section, including computational efficiency.

In order to make use of SSA in Graph-DDP, we modify the forward and backward computations of each bifurcation primitive. We consider a bifurcation primitive illustrated in Fig. 1 (a) where the input node is n_k. As illustrated in Fig. 1 (c), each edge dynamical system maps a part of an input SSA to a part of an output SSA. The other elements of the input SSA are kept in the output SSA. Then although a gradient of each edge dynamical system is a huge matrix, many of the diagonal elements are one, and many of non-diagonal elements are zero. This sparsity leads to computational efficiency.

Let $l_{k,i} \in L_k$ denote an i-th label of SSA at a node n_k ($i = 1, 2, \dots$), and $l^b_{k+1,i} \in L^b_{k+1}$ denote an i-th label of SSA at a node n^b_{k+1} (an output node of a branch b), where L_k and L^b_{k+1} denote a set of labels. $\boldsymbol{\xi}_k$, $\boldsymbol{\xi}^b_{k+1}$: SSA at a node n_k and n^b_{k+1} respectively. $\boldsymbol{\xi}[l]$: a value of a label l in an SSA $\boldsymbol{\xi}$. We consider a probabilistic distribution of SSA $\boldsymbol{\xi}$ as that each element $\boldsymbol{\xi}[l]$ has an independent normal distribution (we omit the calculation of the probabilistic distribution for the readability).

An edge dynamical system $\mathbf{F}^b_k(\mathbf{x}_k)$ is modified to: $\boldsymbol{\xi}^b_{k+1} = \mathbf{F}^b_k(\boldsymbol{\xi}_k)$. $\mathbf{F}^b_k$ uses only $In^b_k \subset L_k$ labels of $\boldsymbol{\xi}_k$ as the input, and modifies only $Out^b_k \subset L^b_{k+1}$ labels[1]. Formally, $L^b_{k+1} = L_k \cup Out^b_k$. The forward computation is: $\boldsymbol{\xi}^b_{k+1}[l] = \mathbf{F}^b_k(\boldsymbol{\xi}_k)[l]$ for $l \in Out^b_k$, and $\boldsymbol{\xi}^b_{k+1}[l] = \boldsymbol{\xi}_k[l]$ for ($l \in L_k$ and $l \notin Out^b_k$). Similarly, a bifurcation probability model $\mathbf{F}^{\mathrm{P}b}_k(\mathbf{x}_k)$ is modified to: $p^b_{k+1} = \mathbf{F}^{\mathrm{P}b}_k(\boldsymbol{\xi}_k)$. $\mathbf{F}^{\mathrm{P}b}_k$ uses only $In^{\mathrm{P}b}_k \subset L_k$ labels of $\boldsymbol{\xi}_k$ as the input.

Let us denote the label-wise gradients of $\mathbf{F}^b_k$ and $\mathbf{F}^{\mathrm{P}b}_k$ with respect to the input SSA as follows: $\partial\mathbf{F}^b_k[l_2][l_1] = \frac{\partial \mathbf{F}^b_k(\boldsymbol{\xi}_k)[l_2]}{\partial \boldsymbol{\xi}_k[l_1]}$, $\partial\mathbf{F}^{\mathrm{P}b}_k[l_1] = \frac{\partial \mathbf{F}^{\mathrm{P}b}_k(\boldsymbol{\xi}_k)}{\partial \boldsymbol{\xi}_k[l_1]}$. For all $l_1 \in In^b_k$ and $l_2 \in Out^b_k$, the corresponding elements $\partial\mathbf{F}^b_k[l_2][l_1]$ are obtained by differentiating $\mathbf{F}^b_k$. Otherwise, the elements are calculated as follows: $\partial\mathbf{F}^b_k[l_2][l_1] = \mathbf{1}$ for ($l_1 \in L_k$ and $l_1 \notin Out^b_k$ and $l_2 = l_1$), and $\partial\mathbf{F}^b_k[l_2][l_1] = \mathbf{0}$ otherwise, where $\mathbf{0}$ and $\mathbf{1}$ denotes a zero and an identity matrix. Similarly, for all $l_1 \in In^{\mathrm{P}b}_k$, the corresponding elements $\partial\mathbf{F}^{\mathrm{P}b}_k[l_1]$ are obtained by differentiating $\mathbf{F}^{\mathrm{P}b}_k$. Otherwise, the elements are $\mathbf{0}$.

[1] Out^b_k may include new labels that do not appear in the labels L_k.

The backward computation is also done in a label-wise fashion. The back-propagation of a value function J is obtained easily from Eq. (4) just replacing $\mathbf{x}_k$ by $\boldsymbol{\xi}_k$. In terms of the gradients of J, we assume that succeeding gradients are back-propagated as follows: $\partial J^b_{k+1}[l] = \frac{\partial J^b_{k+1}(\boldsymbol{\xi}^b_{k+1})}{\partial \boldsymbol{\xi}^b_{k+1}[l]}$ for all $l \in L^b_{k+1}$. By substituting these and the forward computation results into Eq. (6), we obtain: for each $l \in L_k$,

$$\partial J_k[l] = \frac{\partial J_k(\boldsymbol{\xi}_k)}{\partial \boldsymbol{\xi}_k[l]} = \sum_b \Big[\partial\mathbf{F}^{\mathrm{P}b}_k[l] J^b_{k+1} + \mathbf{F}^{\mathrm{P}b}_k \sum_{l' \in L^b_{k+1}} \partial\mathbf{F}^b_k[l'][l] \partial J^b_{k+1}[l']\Big]. \quad (7)$$

IV. Toy Example (1)

We demonstrate how Graph-DDP works in a simple toy example inspired by [14]. Fig. 2(left) illustrates this example. We consider a 2D world with two cannons at $\mathbf{p}_1 = [0, 0]^\top$ and $\mathbf{p}_2 = [0, 0.8]^\top$, and a static target at $\mathbf{p}_e$. The task is shooting a bullet to hit the target with one of the cannons. We can decide the launch angle $\theta \in [-\pi/2, \pi/2]$, while the initial speed is a constant $v_0 = 3$. There is gravity $\mathbf{g} = [0, -g]^\top = [0, -9.8]^\top$. The cannon-selection criterion is that the bullet must reach the target, and a smaller flight time is better.

In addition to $\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{p}_e$, θ, and v_0, the variables (i.e. SSA elements) are: $s \in \{1, 2\}$: a selection of cannons each of which corresponds to $\mathbf{p}_1$ and $\mathbf{p}_2$, t_h: the time when the bullet passes the x position of $\mathbf{p}_e$ (i.e. reaching time), p_{hy}: the height (y position) of bullet at t_h.

Fig. 2(right) shows the graph-structured dynamical system used for planning. The bifurcation model $\mathbf{F}^{\mathrm{P}}_{\mathrm{s}}$ models the bifurcation probabilities p_1, p_2 based on the selection s. Specifically, $\mathbf{F}^{\mathrm{P}}_{\mathrm{s}}(s) = [\delta(s, 1), \delta(s, 2)]^\top$ where $\delta(s, s')$ takes 1 if $s = s'$, otherwise takes 0. We consider the gradient of $\mathbf{F}^{\mathrm{P}}_{\mathrm{s}}$ with respect to s to be zero. The edge dynamical systems $\mathbf{F}_1$ and $\mathbf{F}_2$ model the result of shooting, t_h and p_{hy} (i.e. $[t_h, p_{hy}]^\top = \mathbf{F}_0(\mathbf{p}_0, \mathbf{p}_e, v_0, \theta)$ where $\mathbf{F}_0$ is one of $\mathbf{F}_1$ and $\mathbf{F}_2$, and $\mathbf{p}_0$ is one of $\mathbf{p}_1$ and $\mathbf{p}_2$). In this case we use an analytical model: $t_h = \beta/(v_0 \cos\theta))$, $p_{hy} = p_{0y} + \beta \sin\theta/\cos\theta - \alpha/(\cos\theta)^2$, where $\mathbf{p}_0 = [p_{0x}, p_{0y}]^\top$, $\mathbf{p}_e = [p_{ex}, p_{ey}]^\top$, $\beta = p_{ex} - p_{0x}$, and $\alpha = g\beta^2/(2v_0^2(\cos\theta)^2)$. The gradient of $\mathbf{F}_0$ with respect to the input variables is a 6×2 matrix. Since the optimized action is only θ, we compute only the partial gradients with respect to θ, and assign zero to other elements of the matrix; $\partial t_h/\partial\theta = \beta \sin\theta/(v_0(\cos\theta)^2))$, $\partial p_{hy}/\partial\theta = \beta/(\cos\theta)^2 - 2\alpha \sin\theta/(\cos\theta)^3$. The reward function R is given as follows: $R(t_h, p_{hy}, p_{ey}) = -(p_{hy} - p_{ey})^2 - 10^{-3} t_h^2$; the reason of the small weight on t_h^2 is because the main task purpose is making the shot reach the target.

We varied p_{ex} from 0 to 1.5, and set $p_{ey} = 0.3$. Fig. 3 shows the result of planning θ and s with Graph-DDP at each p_{ex}. Solving the problem analytically with respect to θ, we find two possible solutions for each cannon when the target is in the reachable range. For each cannon, we chose a better θ (i.e. t_h is smaller), and plot in Fig. 3 as the analytical

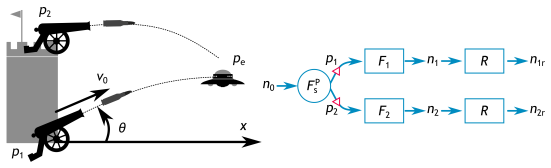

Fig. 2. Shoot-UFO-by-cannon task. Left: illustration of the task, right: graph-structured dynamical system of the task.

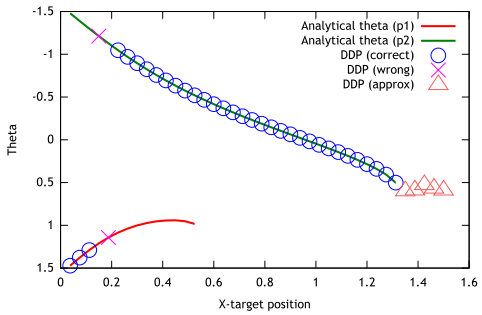

Fig. 3. Result of the shoot-UFO-by-cannon task. Analytically obtained θ for each cannon is also plotted. There are three marker types: circle shows the selection is correct, cross shows the selection is wrong, and triangle shows an approximation (there is no analytical solution).

results. Note that there is no solution of θ for the cannon $\mathbf{p}_1$ and $p_{ex} > 0.54$, and for the cannon $\mathbf{p}_2$ and $p_{ex} > 1.32$. The marker shapes denote whether the cannon selection is correct or wrong. There were two mistakes. Since they were around the border where the optimal cannon changes, the t_h values of local optima were close. As a consequence, suboptimal solutions were chosen. When $p_{ex} > 1.32$, although there is no solution, Graph-DDP gives θ that maximizes R.

V. Learning Dynamics

When solving practical and complicated tasks such as manipulation of liquids, dynamical models are sometimes partially or totally unknown. For example in pouring, the flow dynamics are complicated to construct models analytically. In such situations, we learn models from samples obtained from practice. Specifically we use neural networks extended to be usable with stochastic DDP [18]. The extended neural networks are capable of: (1) modeling prediction error and output noise, (2) computing an output probability distribution for a given input distribution, and (3) computing gradients of output expectation with respect to an input. Since neural networks have nonlinear activation functions (in our case, rectified linear units, ReLU), these extensions were not trivial. In [18] we gave an analytic solution for them with some simplifications.

Fig. 4 shows the neural network architecture used in this paper. We consider a mean model and an error model. The error model estimates output noise and prediction error. For a given set of samples of input $\{\mathbf{x}\}$ and output $\{\mathbf{y}\}$, we train the mean model with a back propagation technique. Then we generate prediction errors (including output noise) $\{\mathbf{\Delta y}\}$ for training the error model. A special loss function is used. When a normal distribution $\mathbf{x} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ is given as an input, our method computes $\mathrm{E}[\mathbf{y}]$, $\mathrm{cov}[\mathbf{y}]$, and the gradient $\frac{\partial \mathrm{E}[\mathbf{y}]}{\partial \boldsymbol{\mu}}$ analytically. Thus we can use the neural networks in

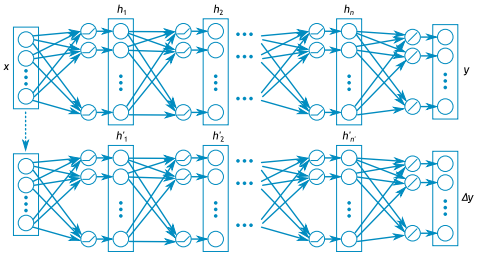

Fig. 4. Neural network architecture used in this paper. It has two networks with the same input vector. The top part estimates an output vector, and the bottom part models prediction error and output noise. Both use ReLU as activation functions.

the forward and the backward propagations of the stochastic DDP. Refer to [18] for more details including comparisons with locally weighted regression.

VI. Toy Example (2)

We apply Graph-DDP to a simplified pushing task where a target object position has a mixture-of-Gaussian uncertainty. This task was inspired by [21]. The task setup is illustrated in Fig. 5(left) where we consider a pushing task on a 2D plane. The target object is located at $\mathbf{p}_o$, whose observation has a mixture-of-Gaussian distribution. An example physical situation is that a template matching algorithm detects multiple local optima of the matching function. Here we consider only a mixture of two Gaussian distributions, $\mathcal{N}(\mathbf{p}_1, \sigma_1)$, $\mathcal{N}(\mathbf{p}_2, \sigma_2)$, with fixed weights 0.5, 0.5 respectively. The pushing motion is predefined and parameterized with $\mathbf{p}_m$, θ, and m; the gripper is put at $\mathbf{p}_m$ with the orientation θ, and moves forward with distance m. There are four possible cases of the result of the pushing motion as depicted in (a)…(d) of Fig. 5(center). Only in cases (c) and (d), the target object moves, and only in (d) there is success (the object is pushed by the center of the gripper).

Fig. 5(right) shows a graph-structured dynamical system to plan the pushing parameters. We use a bifurcation to represent multimodality (not a selection of actions); i.e. each branch corresponds with a Gaussian distribution, and $\mathbf{F}_{\mathrm{s}}^{\mathrm{P}}$ gives the mixture weights: $\mathbf{F}_{\mathrm{s}}^{\mathrm{P}}() = [0.5, 0.5]^\top$ (this is a constant). The edge dynamical systems $\mathbf{F}_{\mathrm{grasp1}}$ and $\mathbf{F}_{\mathrm{grasp2}}$ model the result of the pushing motion. We assume that these functions return $\Delta\bar{\mathbf{p}}'_1$ and $\Delta\bar{\mathbf{p}}'_2$ that denote the object position after the movement in the gripper coordinate system. Deriving these equations is trivial, but they have a nonlinearity; there are discrete changes in the dynamics as shown in (a)…(d) of Fig. 5(center). We use a Taylor series expansion to obtain gradients of the dynamical systems, and compute the propagation of Gaussian distributions with local linear models. The reward is defined as: $R = -1000\|\Delta\bar{\mathbf{p}}'_i\|^2 - 0.001m^2$ where i indicates 1 or 2. This reward means that we add a big penalty for $\|\Delta\bar{\mathbf{p}}'_i\|$ (the cases other than (d) of Fig. 5(center) are penalized), and a small penalty for gripper movement.

We compare three cases: (1) using an analytical model, (2) using an analytical model without considering the Gaussian distributions (i.e. let σ_1 and σ_2 be zero), and (3) using neural network models for $\mathbf{F}_{\mathrm{grasp1}}$ and $\mathbf{F}_{\mathrm{grasp2}}$. (2) is for verifying if the planned actions (1) consider the Gaussian distributions. Since the dynamical system contains discrete

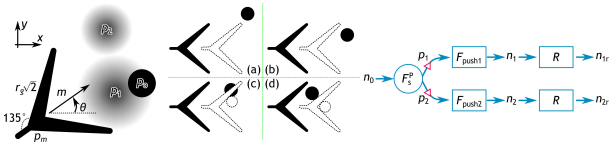

Fig. 5. Push-under-uncertainty task. Left: illustration of the task, center: different results of pushing, right: graph-structured dynamical system of the task.

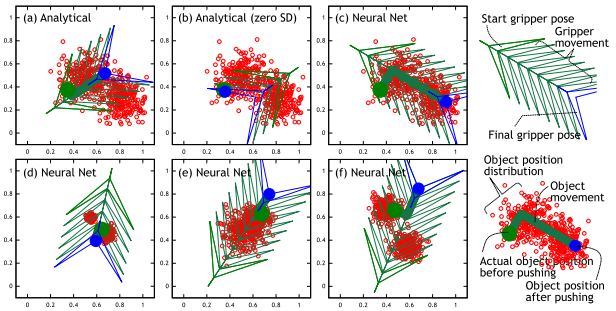

Fig. 6. Result of the push-under-uncertainty task.

changes, the local linear model becomes inaccurate around the discontinuities, affecting DDP and the propagation of Gaussian distributions. We use our neural networks to see if these problems are reduced. As reported in [18], our stochastic neural networks give better predictions of output expectations and gradients especially when the approximated function includes discrete changes such as a step function.

We randomly changed $\mathcal{N}(\mathbf{p}_1, \sigma_1)$ and $\mathcal{N}(\mathbf{p}_2, \sigma_2)$. The success rate of 100 trials are: (1) Analytical: 83, (2) Analytical (zero SD): 42, (3) Neural Networks: 96. Considering Gaussian distributions increases the success rate. Using neural network models also improves the success rate. Some examples are shown in Fig. 6. (a), (b), and (c) compare the three cases in the same situation. In (b), the gripper starts from the position where it touches both centers of Gaussian distributions $\mathbf{p}_1$, $\mathbf{p}_2$. The neural network model (c) gives intuitively correct plan compared to (a). (d), (e), and (f) are the results of the neural network model in different situations. These results would also be intuitively correct. Note the reason why the gripper movement is not on the line $\mathbf{p}_1$-$\mathbf{p}_2$ in (d) and (f) would be due to the movement penalty.

VII. Simulation Experiments of Pouring

We explore the usefulness of Graph-DDP in a more complicated task, pouring. We extend the pouring simulator from our previous work [20], [18], to simulate a different viscosity of a material, and different mouth sizes of a source container. The purpose of the extension is to create a pouring task where different skills are necessary to generalize, similar to a real pouring task. Refer to the accompanying video.

In Open Dynamics Engine (`http://www.ode.org/`), we simulate source and receiving containers, poured material, and a robot gripper grasping the source container as shown in Fig. 7(a). The gripper is modeled as fixed blocks around the source container. We can change the grasping position, but it does not affect the grasp quality. This gripper possibly pushes the receiving container during pouring.

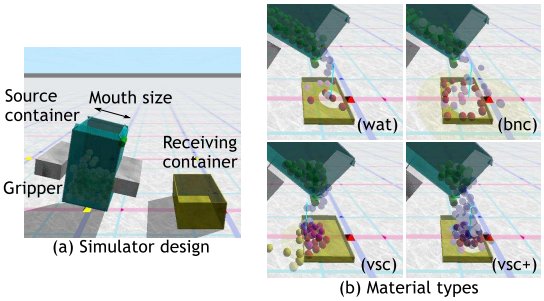

Fig. 7. (a) Pouring simulation setup. (b) Types of poured materials.

We simulate the poured material with many (100) spheres. Although each of the spheres is a rigid object, the entire group behaves like a liquid (no surface tension or adhesive effects). For simulating different types of materials, (1) we model a viscosity, and (2) we modify some contact model parameters, such as bouncing parameters. The viscosity is modeled by applying gravitational forces between spheres virtually. We use four types of materials: (wat): water-like viscosity (non-viscous), (bnc): water-like viscosity (non-viscous) and high bouncing parameters, (vsc): large viscosity, and (vsc+): very large viscosity. The examples of these flows are shown in Fig. 7(b). If the mouth size of the source container is small enough, the flow of (vsc) and (vsc+) stop completely as the material is jammed around the mouth. Shaking the container can solve jamming. The diameter of each sphere is 0.05 (length unit in simulator; the gravity is 1.0), and the size of source mouth varies from 0.11 to 0.25.

The behavior of pouring is designed with a state machine as illustrated in Fig. 8(top). The robot grasps the source container, moves it to somewhere close to the receiving container, moves it to the pouring location, and produces a flow with a selected skill. If the flow is not observed in a specific time, the robot moves the source container back and restarts from `MoveToPour`. This trial-and-error loop is repeated until the target amount is achieved or a specific duration passes.

Fig. 8(bottom) models the dynamical system. The first two bifurcations are for penalizing the corresponding actions. Each flow control skill (tipping and shaking) is decomposed into two dynamical models (F_{flowc_*} and F_{amount}). This is because the decomposition makes the modeling easier as discussed in [20], and we can share a common model F_{amount}. F_{flowc_*} estimates how the flow happens by the skill, and F_{amount} estimates how the flow affects the poured and spilled amounts. Thus F_{flowc_*} depends on each skill but F_{amount} can be common. We can train F_{amount} with the samples of both tipping and shaking. The planning is done at the beginning and at the node n_3 where the trial-and-error loop starts. Note that the dynamical system does not represent the trial-and-error loop because the planning assumes the pouring is completed with a single skill selection.

We consider 19 types of state variables including container position, target amount, material property, mouth size, and flow position. The total dimensionality of the state vectors is 49. There are 6 types of action parameters including grasping, pouring position, and parameters for each skill. The total dimensionality of the action parameters is 8. Not

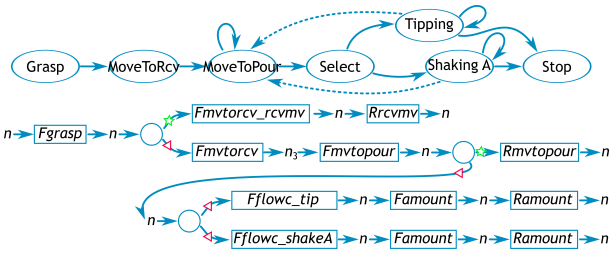

Fig. 8. State machine of pouring behavior (top) and its graph-structured dynamical system (bottom). F_* denotes an edge dynamical system, and R_* denotes a reward model.

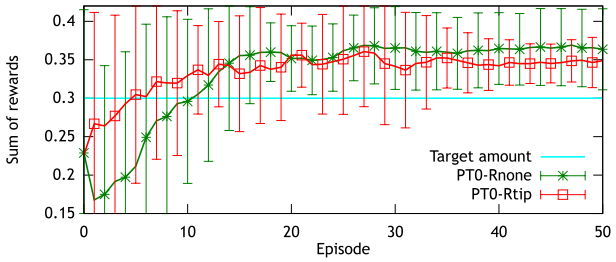

Fig. 9. Poured amount per episode in SETUP1. A moving average filter with 5 episode window is applied.

all of the state vectors and action parameters are used in each dynamical system. The state vectors that do not affect a dynamical system are omitted. Consequently the dimensionalities of the edge dynamical systems vary up to 18. These are represented as a super-state-action dictionary.

We use a reward function defined as follows:

$$R = -100(\min(0, a_{\mathrm{rcv}} - a_{\mathrm{trg}}))^2 - 10(\max(0, a_{\mathrm{spill}}))^2 - (\max(0, a_{\mathrm{rcv}} - a_{\mathrm{trg}}))^2 \quad (8)$$

where a_{rcv} is an amount poured into the receiving container, a_{trg} is a target amount, and a_{spill} is a spilled amount. This reward function means that a big penalty is given for $a_{\mathrm{rcv}} < a_{\mathrm{trg}}$ since it is the first priority, a medium penalty is given for $a_{\mathrm{spill}} > 0$ since spillage should be avoided, and otherwise a_{rcv} being close to a_{trg} is better. $a_{\mathrm{spill}} \geq$ is always satisfied in observation data, but the learned model might estimate a_{spill} as a slight negative value. Thus we are using $\max(0, a_{\mathrm{spill}})$.

A. On-line Learning and Learning Scheduling

We use on-line learning, i.e. the robot plans actions with the latest models, and executes the actions and updates the models according to the results. If no model is available, the robot takes random actions. Each episode is defined as an entire pouring sequence including the trial-and-error loops.

Through the preliminary experiments, we noticed that scheduling learning is useful to avoid local optima. We consider two types of scheduling. (1) Reward shaping: modifying the reward functions according to the number of episodes. (2) Setup scheduling: controlling the experimental setups. Note that an alternative approach to avoid local optima is increasing the exploration randomness, where these supervisory signals would be removed. Since we will use our method with real robots, here we explore efficient solutions.

B. Experiments and Results

First we consider a setup referred to as SETUP1: water-like viscosity and high bouncing parameters (bnc), and wide mouth size of the source container (0.23). Note that tipping is the best skill in this setup. This is the same as our previous work [20], [18], but the difference is that these experiments have multiple skills. Thus the robot has to learn to select a skill as well as to tune skill parameters.

We use pre-trained dynamical models other than F_{flowc_*} and F_{amount} as these dynamical models are similar to those in [18] and can be learned easily. We compare two conditions: PT0-Rnone: no reward shaping, and PT0-Rtip: tipping-biased reward shaping. The reward shaping of PT0-Rtip is guiding the correct skill (tipping). More specifically, from 1st to 10th episode, we use $R+10$ if tipping is selected otherwise R, and after 11th episode, we use R.

Fig. 9 shows the results of 10 runs. Each curve shows an average of poured amount per episode. The target amount is 0.3. The poured amount of PT0-Rtip is closer to the target than PT0-Rnone. In some runs of PT0-Rnone, shaking was chosen after learning. This was because shaking was good for early adaptation to the first objective, achieving $a_{\mathrm{rcv}} \geq a_{\mathrm{trg}}$, as defined in the reward Eq. (8). The excess amount is penalized, so tipping is better than shaking. But since that penalization is the third priority, there were a few runs where the robot tended to use shaking and dynamical models of tipping were not trained enough. The tipping-biased reward shaping was useful to handle this issue. It gained the tendency to select tipping in the early stage of learning. As the result, the average poured amount of PT0-Rtip is smaller than that of PT0-Rnone.

Next we investigate a setup referred to as SETUP2: large viscosity (vsc), and narrow mouth size of the source container (0.13). Shaking is the best skill in this setup. We compare three conditions: PT0-Rnone: Pre-trained dynamical models other than F_{flowc_*} and F_{amount}, no reward shaping. PT1-Rnone: Pre-trained with SETUP1, no reward shaping. PT1-Rshake: Pre-trained with SETUP1, shaking-biased reward shaping. The shaking-biased reward shaping is that from 1st to 10th episode, we use $R+10$ if shaking is selected otherwise R, and after 11th episode, we use R. PT1-Rnone and PT1-Rshake are examples of the setup scheduling.

Fig. 10 shows the learning curves, i.e. sum of rewards per episode, that are averages of 10 runs. PT1-Rnone and PT1-Rshake had more prior knowledge than PT0-Rnone. Especially F_{amount} was learned in SETUP1, which would generalize in SETUP2. What PT1-Rnone and PT1-Rshake did not have are the training samples of F_{flowc_*} in SETUP2. Thus these converged faster than PT0-Rnone. The reward shaping was also useful in this setup. However since shaking is only the adequate solution in this setup (tipping does not work), all runs of PT1-Rnone (no reward shaping) did not converge to poor local optima.

The variance of PT1-Rshake was increasing after 10th episode. This was because the reward-shaping changed. The reason why the variance of PT1-Rshake is greater than that of PT1-Rnone around the 10th episode is that tipping in this setup is not well-trained with PT1-Rshake in the 0th to 9th episodes because of the shaking-biased reward shaping. Thus in some runs, the robot tried tipping after the 10th episode.

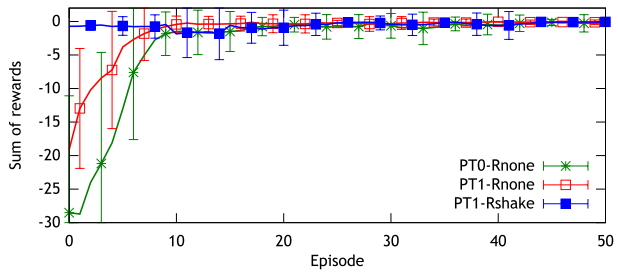

Fig. 10. Learning curves (sum of rewards per episode) of SETUP2. A moving average filter with 5 episode window is applied.

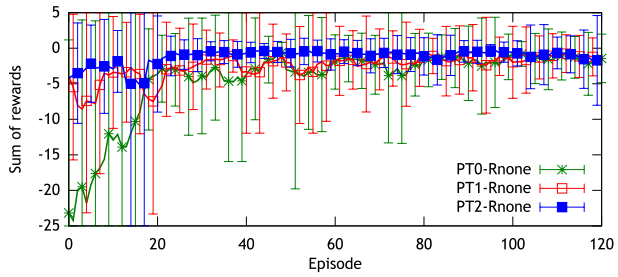

Fig. 11. Learning curves (sum of rewards per episode) of the general pouring setup. A moving average filter with 5 episode window is applied.

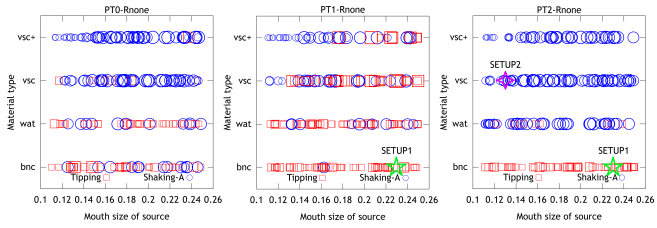

Fig. 12. Skill selections in different situations (mouth size of source container, material type). The shape of the points show the skill (tipping, shaking). Setups for pre-training (SETUP1, SETUP2) are also shown. The size of point shows an error of poured amount.

Finally we explore the generalization ability of our method in a general pouring setup: four material types (wat, bnc, vsc, vsc+), and various mouth sizes of the source container (0.11 to 0.25). We consider three conditions including the setup scheduling. PT0-Rnone: Pre-trained dynamical models other than F_{flowc_*} and F_{amount}, no reward shaping. PT1-Rnone: Pre-trained with SETUP1, no reward shaping. PT2-Rnone: Pre-trained with SETUP1 and SETUP2, no reward shaping.

Fig. 11 shows the learning curves, i.e. sum of rewards per episode, which are averages of 10 runs. PT1-Rnone had more prior knowledge than PT0-Rnone, and PT2-Rnone had more than PT1-Rnone. The speed of the convergence seems to be proportional to the prior knowledge. Especially PT2-Rnone performs well in the early stages although it pre-trained only in specific situations (SETUP1, SETUP2). The variance of the learning curves is greater than that of Fig. 10. This is because in Fig. 11, the rewards from different setups are shown together.

Although the three conditions converged to similar performance level, their details were slightly different. The graphs in Fig. 12 show which skill was selected in a situation. Each graph plots 20 samples of all runs after convergence of learning curve. Each condition had a different tendency. In PT1-Rnone and PT2-Rnone, the skill selection was biased by the pre-training setups. In the viscous materials (vsc, vsc+) with a narrower mouth size, shaking was dominant regardless of the pre-training. In the highly bouncing material (bnc) case, tipping was dominant regardless of the pre-training. Since the learning curves in Fig. 11 converged to close values, there appears to be no convergence to poor local optima.

VIII. Conclusion

We proposed a stochastic differential dynamic programming for graph-structured dynamical systems. We introduced the idea of frame-and-slots based on traditional artificial intelligence researches to represent a large state-action vector. The proposed method was applied to a simulated pouring task and showed that our method generalized over the material property (viscosity) and the source container shape (mouth size). Future work includes verifying our method in real pouring tasks by robots.

References

[1] A. Yamaguchi, C. G. Atkeson, and T. Ogasawara, "Pouring skills with planning and learning modeled from human demonstrations," *International Journal of Humanoid Robotics*, vol. 12, no. 3, p. 1550030, 2015.

[2] D. Mayne, "A second-order gradient method for determining optimal trajectories of non-linear discrete-time systems," *International Journal of Control*, vol. 3, no. 1, pp. 85–95, 1966.

[3] P. H. Winston, *Artificial Intelligence (3rd Ed.)*. Addison-Wesley Longman Publishing Co., Inc., 1992.

[4] Y. Pan and E. Theodorou, "Probabilistic differential dynamic programming," in *Advances in Neural Information Processing Systems 27*. Curran Associates, Inc., 2014, pp. 1907–1915.

[5] S. Levine and V. Koltun, "Variational policy search via trajectory optimization," in *Advances in Neural Information Processing Systems 26*. Curran Associates, Inc., 2013, pp. 207–215.

[6] M. D. Zeiler, "ADADELTA: an adaptive learning rate method," *ArXiv e-prints*, no. arXiv:1212.5701, 2012.

[7] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *International Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274, 2013.

[8] E. Theodorou, J. Buchli, and S. Schaal, "Reinforcement learning of motor skills in high dimensions: A path integral approach," in *ICRA'10*, may 2010, pp. 2397–2403.

[9] J. Kober and J. Peters, "Policy search for motor primitives in robotics," *Machine Learning*, vol. 84, no. 1-2, pp. 171–203, 2011.

[10] S. Schaal and C. Atkeson, "Robot juggling: implementation of memory-based learning," in *ICRA'94*, 1994, pp. 57–71.

[11] J. Morimoto, G. Zeglin, and C. Atkeson, "Minimax differential dynamic programming: Application to a biped walking robot," in *the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'03)*, vol. 2, 2003, pp. 1927–1932.

[12] I. Lenz, R. Knepper, and A. Saxena, "DeepMPC: Learning deep latent features for model predictive control," in *Robotics: Science and Systems (RSS'15)*, 2015.

[13] E. Magtanong, A. Yamaguchi, K. Takemura, J. Takamatsu, and T. Ogasawara, "Inverse kinematics solver for android faces with elastic skin," in *Latest Advances in Robot Kinematics*, Innsbruck, Austria, 2012, pp. 181–188.

[14] E. W. Aboaf, C. G. Atkeson, and D. J. Reinkensmeyer, "Task-level robot learning," in *IEEE International Conference on Robotics and Automation*, 1988, pp. 1309–1310.

[15] E. W. Aboaf, S. M. Drucker, and C. G. Atkeson, "Task-level robot learning: juggling a tennis ball more accurately," in *the IEEE International Conference on Robotics and Automation*, 1989, pp. 1290–1295.

[16] A. Yamaguchi and C. G. Atkeson, "Model-based reinforcement learning with neural networks on hierarchical dynamic system," in *the Workshop on Deep Reinforcement Learning: Frontiers and Challenges in IJCAI'16*, 2016.

[17] V. Mnih, K. Kavukcuoglu, D. Silver, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[18] A. Yamaguchi and C. G. Atkeson, "Neural networks and differential dynamic programming for reinforcement learning problems," in *ICRA'16*, 2016.

[19] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Inc., 1995.

[20] A. Yamaguchi and C. G. Atkeson, "Differential dynamic programming with temporally decomposed dynamics," in *Humanoids'15*, 2015.

[21] M. C. Koval, N. S. Pollard, and S. S. Srinivasa, "Pre- and post-contact policy decomposition for planar contact manipulation under uncertainty," *The International Journal of Robotics Research*, vol. 35, no. 1-3, pp. 244–264, 2016.