Neural Networks and Differential Dynamic Programming for Reinforcement Learning Problems

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

Abstract— **We explore a model-based approach to reinforcement learning where partially or totally unknown dynamics are learned and explicit planning is performed. We learn dynamics with neural networks, and plan behaviors with differential dynamic programming (DDP). In order to handle complicated dynamics, such as manipulating liquids (pouring), we consider temporally decomposed dynamics. We start from our recent work [1] where we used locally weighted regression (LWR) to model dynamics. The major contribution of this paper is making use of deep learning in the form of neural networks with stochastic DDP, and showing the advantages of neural networks over LWR. For this purpose, we extend neural networks for: (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, these extensions were not easy. We provide an analytic solution for these extensions using some simplifying assumptions. We verified this method in pouring simulation experiments. The learning performance with neural networks was better than that of LWR. The amount of spilled materials was reduced. We also present early results of robot experiments using a PR2. Accompanying video: `https://youtu.be/aM3hE1J5W98`**

I. INTRODUCTION

Deep learning is a powerful framework, and has been successful in image classification [2], object detection [3], pose estimation [4], and other areas. There is great interest in applying deep learning to robotic applications, especially reinforcement learning problems, i.e. learning behaviors with partially or totally unknown dynamics. There are some attempts [5], [6], but this is still an open problem.

We explore learning dynamics with neural networks, and applying differential dynamic programming (DDP) to plan behaviors. Recently we considered temporally decomposed dynamics of a complicated task like pouring [7], modeled them with locally weighted regression (LWR), and applied DDP [1]. This method was verified in simulation experiments of pouring, where the robot controlled a complicated flow to achieve pouring. The contribution of [1] was showing that learning decomposed dynamics and applying stochastic DDP is a promising solution to reinforcement learning problems, even in domains with complicated dynamics. By introducing deep neural networks into this framework, we expect: (A) better modeling accuracy, and (B) automatic feature selection.

In order to make use of neural networks with stochastic DDP, we need extensions for: (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, these extensions were not easy. In this paper we provide an analytic solution for these extensions using some assumptions for simplification.

We verified our method in simulation experiments of pouring similar to those in [1]. Our stochastic DDP with extended neural networks outperformed the LWR version. When training with many samples, there was a small amount of spilled materials in the LWR version. This was reduced with the neural networks version. In addition, we used redundant and/or non-informative state representations to investigate the feature extraction ability of (deep) neural networks. Although the state dimensionality changed from 16 to 47, the learning performance was preserved. We also conducted preliminary robot experiments using a PR2. Although the number of the experiments is not sufficient, we obtained encouraging results. From the simulator results, we think that learning dynamics with deep neural networks and planning with stochastic DDP is a promising solution to reinforcement learning problems. Furthermore, the extended neural network is a general-purpose function approximator and could be applied to learning various models used in robotics such as dynamics and kinematics.

Related Work

There are two main approaches for reinforcement learning (RL) problems: a model-free approach and a model-based approach. In recent robotics research, the model-free approach seems to be more successful, for example in a ball-in-cup task [8], flipping a pancake [9], and a crawling motion [10]. A disadvantage of model-free methods is the poor generalization ability compared to that of model-based methods. Several attempts have been made to overcome this issue [11], [12].

In a model-based method, dynamics of the system are learned, and then planning, such as differential dynamic programming (DDP) [13], is used. Although there are successful examples, such as [14], [15], this approach seems not popular in RL for robotics, since (1) we need to gather many samples to construct dynamics models, and (2) we need to solve two problems, model learning and planning.

However we think that the model-based approach has a large potential since learning dynamics would be more robust than model-free RL because it is supervised learning, and recently many good optimal control methods to solve

[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`

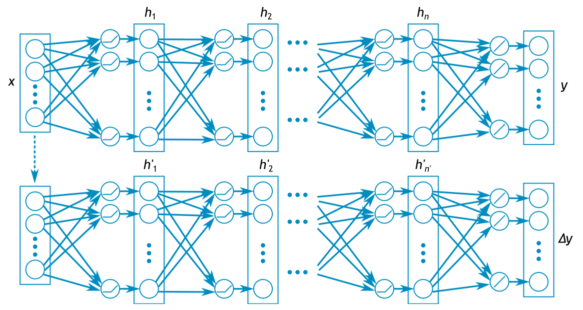


Fig. 1. Neural network architecture used in this paper. It has two networks with the same input vector. The top part estimates an output vector, and the bottom part models prediction error and output noise. Both use ReLU as activation functions.

dynamic programming have been proposed (e.g. [16], [17]). Recently Pan and Theodorou proposed a method: learning dynamics models with a Gaussian process and applying stochastic DDP [18]. We explore neural networks to model complicated dynamics.

There are several examples of reinforcement learning methods using neural networks, such as neural fitted Q iteration [19], and deep Q-learning [5]. These methods use a value function based approach to RL with neural networks. Deep Q-learning [5] makes use of the ability of deep networks, namely, automatically learning features from images, but would not be appropriate to learn behaviors with continuous control signals. This method is using a discrete set of actions like conventional RL methods [20]. Another approach is [12], [6] where a trajectory optimization method was combined with a local linear model learned from samples, and trained neural network polices to increase generalization.

Another example of using neural networks in robot control is [21] where a forward kinematics model between pneumatic actuator displacements and feature point positions on an android robot face was learned with neural networks, and its inverse problem was solved with an optimizer. Although the input and output dimensions were large (11 and 54 respectively), their method achieved an accurate inverse kinematics controller of the android face.

In Section II we propose extensions of neural networks. In Section III we describe our stochastic DDP. Section IV describes the simulation, Section V describes the robot experiments, and Section VI concludes the paper.

II. Neural Networks for Regression with Probability Distributions

We explore extensions of neural networks for: (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. Typical approaches for (2) are: (A) solving analytically, (B) approximating the neural networks with a local linear or quadratic model, and (C) computing numerically with random sampling. (A) is not easy since neural networks include nonlinear activation functions, (B) becomes inaccurate especially when learning a model with discrete changes such as a step function, and (C) has a large computation cost. We consider some assumptions to simplify (A).

A. Definitions and Assumptions

We consider a neural network with rectified linear units (ReLU; $f_{\mathrm{relu}}(x) = \max(0,x)$ for $x \in \mathbb{R}$) as activation functions. Based on our preliminary experiments with neural networks for regression problems, using ReLU as an activation function was the most stable and obtained the best results. Fig. 1 shows the neural network architecture used in this paper. For an input vector $\mathbf{x}$, the neural network models an output vector $\mathbf{y}$ with $\mathbf{y} = \mathbf{F}(\mathbf{x})$, defined as:

$$\mathbf{h}_1 = \mathbf{f}_{\mathrm{relu}}(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1), \tag{1}$$
$$\mathbf{h}_2 = \mathbf{f}_{\mathrm{relu}}(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2), \tag{2}$$
$$\dots$$
$$\mathbf{h}_n = \mathbf{f}_{\mathrm{relu}}(\mathbf{W}_n\mathbf{h}_{n-1} + \mathbf{b}_n), \tag{3}$$
$$\mathbf{y} = \mathbf{W}_{n+1}\mathbf{h}_n + \mathbf{b}_{n+1}, \tag{4}$$

where $\mathbf{h}_i$ is a vector of hidden units at the i-th layer, $\mathbf{W}_i$ and $\mathbf{b}_i$ are parameters of a linear model, and $\mathbf{f}_{\mathrm{relu}}$ is an element-wise ReLU function.

Even when an input $\mathbf{x}$ is deterministic, the output $\mathbf{y}$ might have error due to: (A) prediction error, i.e. the error between $\mathbf{F}(\mathbf{x})$ and the true function, caused by an insufficient number of training samples, insufficient training iterations, or insufficient modeling ability (e.g. small number of hidden units), and (B) noise added to the output. We do not distinguish (A) and (B). Instead, we consider additive noise $\mathbf{y} = \mathbf{F}(\mathbf{x}) + \boldsymbol{\xi}(\mathbf{x})$ where $\boldsymbol{\xi}(\mathbf{x})$ is a zero-mean normal distribution $\mathcal{N}(\mathbf{0}, \mathbf{Q}(\mathbf{x}))$. Note that since the prediction error changes with $\mathbf{x}$ and the output noise might change with $\mathbf{x}$, $\mathbf{Q}$ is a function of $\mathbf{x}$. In order to model $\mathbf{Q}(\mathbf{x})$, we use another neural network $\boldsymbol{\Delta}\mathbf{y} = \boldsymbol{\Delta}\mathbf{F}(\mathbf{x})$ whose architecture is similar to $\mathbf{F}(\mathbf{x})$, and approximate $\mathbf{Q}(\mathbf{x}) = \mathrm{diag}(\boldsymbol{\Delta}\mathbf{F}(\mathbf{x}))^2$ where diag is a diagonal function[1]. $\boldsymbol{\Delta}\mathbf{y}$ is an element-wise absolute error between the training data $\mathbf{y}$ and the prediction $\mathbf{F}(\mathbf{x})$.

When $\mathbf{x}$ is a normal distribution $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, our purpose is to approximate $\mathrm{E}[\mathbf{y}]$ and $\mathrm{cov}[\mathbf{y}]$. The difficulty is the use of the nonlinear ReLU operators $\mathbf{f}_{\mathrm{relu}}(\mathbf{p})$. We use the approximation that when $\mathbf{p}$ is a normal distribution, $\mathbf{f}_{\mathrm{relu}}(\mathbf{p})$ is also a normal distribution. We also assume that this computation is done in an element-wise manner, that is, we ignore non-diagonal elements of $\mathrm{cov}[\mathbf{p}]$ and $\mathrm{cov}[\mathbf{f}_{\mathrm{relu}}(\mathbf{p})]$. Although the covariance $\mathbf{Q}(\mathbf{x})$ depends on $\mathbf{x}$, we consider its MAP estimate: $\mathbf{Q}(\boldsymbol{\mu})$.

B. Expectation

Let us consider $\mathbf{y} = \mathbf{F}(\mathbf{x}) + \boldsymbol{\xi}(\mathbf{x})$ where $\boldsymbol{\xi}(\mathbf{x}) \sim \mathcal{N}(\mathbf{0}, \mathbf{Q}(\mathbf{x}))$. For $\mathbf{x} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$,

$$\mathrm{E}[\mathbf{y}] = \mathrm{E}[\mathbf{F}(\mathbf{x}) + \boldsymbol{\xi}(\mathbf{x})] = \mathrm{E}[\mathbf{F}(\mathbf{x})], \tag{5}$$
$$\begin{aligned}\mathrm{cov}[\mathbf{y}] &= \mathrm{cov}[\mathbf{F}(\mathbf{x}) + \boldsymbol{\xi}(\mathbf{x})] = \mathrm{cov}[\mathbf{F}(\mathbf{x})] + \mathrm{cov}[\boldsymbol{\xi}(\mathbf{x})] \\ &= \mathrm{cov}[\mathbf{F}(\mathbf{x})] + \mathbf{Q}(\boldsymbol{\mu}).\end{aligned} \tag{6}$$

[1]We use diag for two meanings: exploiting diagonal elements as a vector from a matrix, and converting a vector to a diagonal matrix.

Since $\mathbf{Q}(\boldsymbol{\mu}) = \mathrm{diag}(\boldsymbol{\Delta}\mathbf{F}(\boldsymbol{\mu}))^2$, the difficulty is computing $\mathrm{E}[\mathbf{F}(\mathbf{x})]$ and $\mathrm{cov}[\mathbf{F}(\mathbf{x})]$. We solve these step by step.

The expectation and covariance of a linear function for an input $\mathbf{p} \sim \mathcal{N}(\boldsymbol{\mu}_p, \boldsymbol{\Sigma}_p)$ are:

$$\mathrm{E}[\mathbf{W}\mathbf{p} + \mathbf{b}] = \mathbf{W}\boldsymbol{\mu}_p + \mathbf{b}, \quad (7)$$
$$\mathrm{cov}[\mathbf{W}\mathbf{p} + \mathbf{b}] = \mathbf{W}\boldsymbol{\Sigma}_p\mathbf{W}^\top. \quad (8)$$

The expectation and covariance of the ReLU function for an input $\mathbf{p} \sim \mathcal{N}(\boldsymbol{\mu}_p, \boldsymbol{\Sigma}_p)$ where $\boldsymbol{\Sigma}_p$ is a diagonal matrix are:

$$\mathrm{E}[\mathbf{f}_{\mathrm{relu}}(\mathbf{p})] = \mathcal{E}_{\mathrm{relu}}(\boldsymbol{\mu}_p, \boldsymbol{\Sigma}_p), \quad (9)$$
$$\mathrm{cov}[\mathbf{f}_{\mathrm{relu}}(\mathbf{p})] = \mathcal{V}_{\mathrm{relu}}(\boldsymbol{\mu}_p, \boldsymbol{\Sigma}_p), \quad (10)$$

where $\mathcal{E}_{\mathrm{relu}}$ is an element-wise expectation of ReLU, and $\mathcal{V}_{\mathrm{relu}}$ is a diagonal matrix of corresponding variances. For $\boldsymbol{\mu}_p = [\mu_{p1}, \mu_{p2}, \dots]^\top$ and $\boldsymbol{\Sigma}_p = \mathrm{diag}[\sigma_{p1}^2, \sigma_{p2}^2, \dots]$, each element of $\mathcal{E}_{\mathrm{relu}}(\boldsymbol{\mu}_p, \boldsymbol{\Sigma}_p)$ and $\mathcal{V}_{\mathrm{relu}}(\boldsymbol{\mu}_p, \boldsymbol{\Sigma}_p)$ is given by:

$$\begin{aligned}\mathrm{E}[f_{\mathrm{relu}}(p_i)] &= \int_{-\infty}^{\infty} \max(0, p_i) G(p_i; \mu_{pi}, \sigma_{pi}^2) dp_i \\ &= \frac{\sigma_{pi}}{\sqrt{2\pi}} X + \frac{\mu_{pi}}{2}(1 + E), \quad (11)\\ \mathrm{var}[f_{\mathrm{relu}}(p_i)] & \\ &= \int_{-\infty}^{\infty} (\max(0, p_i) - \mathrm{E}[f_{\mathrm{relu}}(p_i)])^2 G(p_i; \mu_{pi}, \sigma_{pi}^2) dp_i \\ &= \frac{1}{4}\mu_{pi}^2(1 - E^2) + \frac{\sigma_{pi}^2}{2}(1 + E - \frac{X^2}{\pi}) - \frac{\sigma_{pi}\mu_{pi}XE}{\sqrt{2\pi}}, \quad (12)\end{aligned}$$

where G is a Gaussian function, $X = \exp(-\frac{\mu_{pi}^2}{2\sigma_{pi}^2})$, $E = \mathrm{erf}(\frac{\mu_{pi}}{\sqrt{2}\sigma_{pi}})$, and erf is an error function. Derivations are described in the appendix.

The expectation and covariance of each layer's output for an input $\mathbf{p} \sim \mathcal{N}(\boldsymbol{\mu}_p, \boldsymbol{\Sigma}_p)$ are:

$$\mathrm{E}[\mathbf{f}_{\mathrm{relu}}(\mathbf{W}\mathbf{p} + \mathbf{b})] = \mathcal{E}_{\mathrm{relu}}(\mathbf{W}\boldsymbol{\mu}_p + \mathbf{b}, \mathrm{diag}(\mathbf{W}\boldsymbol{\Sigma}_p\mathbf{W}^\top)), \quad (13)$$
$$\mathrm{cov}[\mathbf{f}_{\mathrm{relu}}(\mathbf{W}\mathbf{p} + \mathbf{b})] = \mathcal{V}_{\mathrm{relu}}(\mathbf{W}\boldsymbol{\mu}_p + \mathbf{b}, \mathrm{diag}(\mathbf{W}\boldsymbol{\Sigma}_p\mathbf{W}^\top)). \quad (14)$$

Thus the probability distribution of each hidden layer $\mathbf{h}_i \sim \mathcal{N}(\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)$ is given by: $\boldsymbol{\mu}_i = \mathcal{E}_{\mathrm{relu}}(\mathbf{W}_i\boldsymbol{\mu}_{i-1} + \mathbf{b}_i, \mathrm{diag}(\mathbf{W}_i\boldsymbol{\Sigma}_{i-1}\mathbf{W}_i^\top))$, and $\boldsymbol{\Sigma}_i = \mathcal{V}_{\mathrm{relu}}(\mathbf{W}_i\boldsymbol{\mu}_{i-1} + \mathbf{b}_i, \mathrm{diag}(\mathbf{W}_i\boldsymbol{\Sigma}_{i-1}\mathbf{W}_i^\top))$, where $\boldsymbol{\mu}_0 = \boldsymbol{\mu}$ and $\boldsymbol{\Sigma}_0 = \boldsymbol{\Sigma}$ (mean and covariance of input $\mathbf{x}$). We compute from $i = 1$ to n, and obtain $\mathbf{h}_n \sim \mathcal{N}(\boldsymbol{\mu}_n, \boldsymbol{\Sigma}_n)$. Since the final activation function is linear $\mathbf{F}(\mathbf{h}_n) = \mathbf{W}_{n+1}\mathbf{h}_n + \mathbf{b}_{n+1}$, we use Eq. (7), (8). Finally assigning them into Eq. (5), (6), we obtain

$$\mathrm{E}[\mathbf{y}] = \mathbf{W}_{n+1}\boldsymbol{\mu}_n + \mathbf{b}_{n+1}, \quad (15)$$
$$\mathrm{cov}[\mathbf{y}] = \mathbf{W}_{n+1}\boldsymbol{\Sigma}_n\mathbf{W}_{n+1}^\top + \mathbf{Q}(\boldsymbol{\mu}). \quad (16)$$

C. Gradient

The gradient of element-wise ReLU $\mathbf{f}_{\mathrm{relu}}(\mathbf{p})$ is given by:

$$\frac{\partial \mathbf{f}_{\mathrm{relu}}(\mathbf{p})}{\partial \mathbf{p}} = \mathrm{diag}(\mathbf{f}_{\mathrm{step}}(\mathbf{p})), \quad (17)$$

where $\mathbf{f}_{\mathrm{step}}$ is an element-wise step function (1 if p is positive, and 0 otherwise). Thus the gradient $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ for a deterministic $\mathbf{x}$ is obtained by:

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \frac{\partial \mathbf{h}_1}{\partial \mathbf{x}} \frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1} \cdots \frac{\partial \mathbf{h}_n}{\partial \mathbf{h}_{n-1}} \frac{\partial \mathbf{y}}{\partial \mathbf{h}_n}, \quad (18)$$

where $\frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}} = \mathbf{W}_i^\top \mathrm{diag}(\mathbf{f}_{\mathrm{step}}(\mathbf{W}_i\mathbf{h}_{i-1} + \mathbf{b}_i))$, $\mathbf{h}_0 = \mathbf{x}$, and $\frac{\partial \mathbf{y}}{\partial \mathbf{h}_n} = \mathbf{W}_{n+1}^\top$.

For $\mathbf{x} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, we use $\frac{\partial \mathrm{E}[\mathbf{y}]}{\partial \boldsymbol{\mu}}$ as a gradient for the distribution of $\mathbf{x}$. This gradient takes a similar form as above. For $\mathcal{N}(\boldsymbol{\mu}_p, \boldsymbol{\Sigma}_p)$ ($\boldsymbol{\Sigma}_p$ is a diagonal matrix), the gradient of $\mathcal{E}_{\mathrm{relu}}(\boldsymbol{\mu}_p, \boldsymbol{\Sigma}_p)$ w.r.t. $\boldsymbol{\mu}_p$ is given by:

$$\frac{\partial \mathcal{E}_{\mathrm{relu}}(\boldsymbol{\mu}_p, \boldsymbol{\Sigma}_p)}{\partial \boldsymbol{\mu}_p} = \mathrm{diag}(\tilde{\mathbf{f}}_{\mathrm{step}}(\boldsymbol{\mu}_p, \mathrm{diag}(\boldsymbol{\Sigma}_p))), \quad (19)$$

where $\tilde{\mathbf{f}}_{\mathrm{step}}$ is an element-wise function similar to $\mathbf{f}_{\mathrm{step}}$. For $\boldsymbol{\mu}_p = [\mu_{p1}, \mu_{p2}, \dots]^\top$ and $\boldsymbol{\Sigma}_p = \mathrm{diag}[\sigma_{p1}^2, \sigma_{p2}^2, \dots]$, each element of $\tilde{\mathbf{f}}_{\mathrm{step}}$ is given by:

$$\tilde{f}_{\mathrm{step}}(\mu_{pi}, \sigma_{pi}) = \frac{1}{2}(1 + E), \quad (20)$$

where $E = \mathrm{erf}(\frac{\mu_{pi}}{\sqrt{2}\sigma_{pi}})$. Thus the gradient of $\mathrm{E}[\mathbf{y}]$ w.r.t. $\boldsymbol{\mu}$ is given by:

$$\frac{\partial \mathrm{E}[\mathbf{y}]}{\partial \boldsymbol{\mu}} = \frac{\partial \boldsymbol{\mu}_1}{\partial \boldsymbol{\mu}} \frac{\partial \boldsymbol{\mu}_2}{\partial \boldsymbol{\mu}_1} \cdots \frac{\partial \boldsymbol{\mu}_n}{\partial \boldsymbol{\mu}_{n-1}} \frac{\partial \mathrm{E}[\mathbf{y}]}{\partial \boldsymbol{\mu}_n}, \quad (21)$$
$$\frac{\partial \boldsymbol{\mu}_i}{\partial \boldsymbol{\mu}_{i-1}} = \mathbf{W}_i^\top \mathrm{diag}(\tilde{\mathbf{f}}_{\mathrm{step}}(\mathbf{W}_i\boldsymbol{\mu}_{i-1} + \mathbf{b}_i, \mathrm{diag}(\mathbf{W}_i\boldsymbol{\Sigma}_{i-1}\mathbf{W}_i^\top))), \quad (22)$$

where $\boldsymbol{\mu}_0 = \boldsymbol{\mu}$, $\boldsymbol{\Sigma}_0 = \boldsymbol{\Sigma}$, and $\frac{\partial \mathrm{E}[\mathbf{y}]}{\partial \boldsymbol{\mu}_n} = \mathbf{W}_{n+1}^\top$.

D. Loss Function for the Error Model

The neural networks $\mathbf{F}(\mathbf{x})$ and $\boldsymbol{\Delta}\mathbf{F}(\mathbf{x})$ are trained with a data set $\{\mathbf{x}, \mathbf{y}\}$. $\mathbf{F}(\mathbf{x})$ learns to predict from $\mathbf{x}$ to $\mathbf{y}$, and $\boldsymbol{\Delta}\mathbf{F}(\mathbf{x})$ learns to predict from $\mathbf{x}$ to $\boldsymbol{\Delta}\mathbf{y}$. First we train $\mathbf{F}(\mathbf{x})$, and generate an error data set $\{\boldsymbol{\Delta}\mathbf{y}\} = \{\mathbf{f}_{\mathrm{abs}}(\mathbf{y} - \mathbf{F}(\mathbf{x}))\}$ where $\mathbf{f}_{\mathrm{abs}}$ is an element-wise absolute value function. $\{\boldsymbol{\Delta}\mathbf{y}\}$ is a set of positive value vectors. We expect $\boldsymbol{\Delta}\mathbf{F}(\mathbf{x})$ predicts the envelope of $\{\boldsymbol{\Delta}\mathbf{y}\}$. For this purpose, using a standard mean squared error is not adequate since the obtained curve will be around the middle of the data. Thus we use a different loss function to train $\boldsymbol{\Delta}\mathbf{F}(\mathbf{x})$, given by:

$$\begin{aligned}L = \frac{1}{ND}\sum_{n=1}^{N}\sum_{d=1}^{D}(&\max(0, \Delta y_{nd} - \Delta F_d(\mathbf{x}_n))^2 \\ &+ \beta \min(0, \Delta y_{nd} - \Delta F_d(\mathbf{x}_n))^2), \quad (23)\end{aligned}$$

where N is the number of training samples, and D is the number of output dimensions. The positive error and the negative error are summed with a different weight β, $0 < \beta < 1$. A typical value of β is 0.1. With this loss function, the obtained curve shifts to a positive envelope.

E. Example

Fig. 2 shows the ReLU function, $\mathcal{E}_{\mathrm{relu}}(x, 1)$ and $\mathcal{E}_{\mathrm{relu}}(x, 1) \pm \sqrt{\mathcal{V}_{\mathrm{relu}}(x, 1)}$, and the numerically computed expectation of ReLU with the variance 1.

Fig. 3 shows an example of learning a step function with neural networks. We used two hidden layers; each hidden

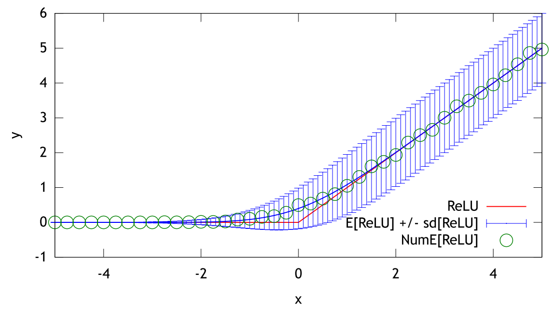


Fig. 2. Expectation of ReLU.

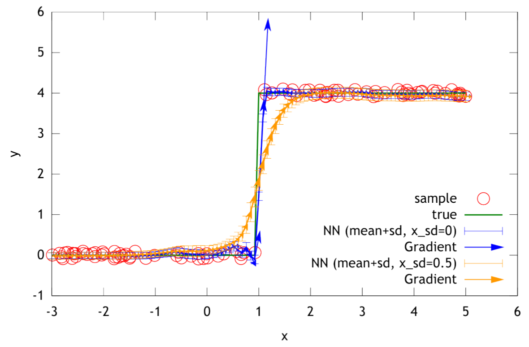


Fig. 3. Learning a step function.

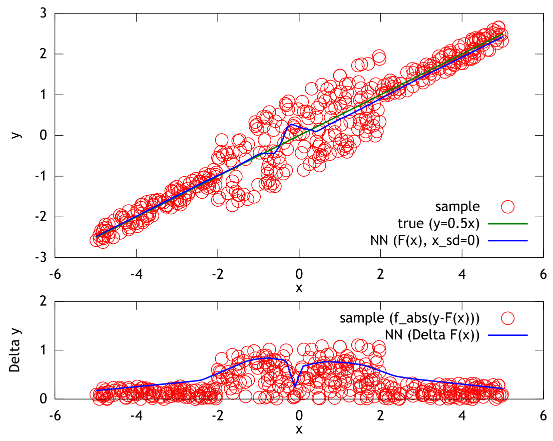


Fig. 4. Learning $y = 0.5x$ with output noise depending on x.

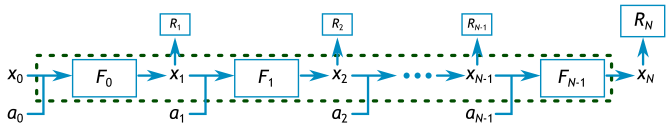


Fig. 5. Temporal decomposition considered in this paper. The dotted box denotes the whole process.

layer has 200 units. We used dropout [22] for each output of the hidden layers during training where the dropout probability is 0.01. The dropout probability for regression works better if it is smaller. In classification, typically 0.5 is used, but such a large value is problematic in regression. Srivastava *et al.* [22] analyzed the dropout in linear regression: “dropout with linear regression is equivalent, in expectation, to ridge regression”. Thus we chose 0.01. For training, we used 100 samples with adding a uniform random noise between $[-0.1, 0.1]$. In Fig. 3, we are comparing the original step function, samples for training, the neural network predictions for $\mathcal{N}(x, 0)$ (there is variance of y because of the prediction error model), and the predictions for $\mathcal{N}(x, 0.5^2)$. The gradients of each prediction are also plotted.

Fig. 4 shows an example of learning a linear function $y = 0.5x$ with neural networks. In this example, we used output noise changing with x: a uniform random noise between $[-1, 1]$ for $|x| < 2$, and a uniform random noise between $[-0.25, 0.25]$ for other x. We used two hidden layers; each hidden layer has 200 units. We used the dropout with probability 0.01 for training. The above graph of Fig. 4 shows $F(x)$ (x is a deterministic variable), and the graph below shows $\Delta F(x)$.

III. Stochastic Differential Dynamic Programming

We use the same differential dynamic programming (DDP) that we used in [1]. This DDP is stochastic; that is, we consider probability distributions of states and expectations of rewards. This design of evaluation functions is similar to recent DDP methods (e.g. [18]). On the other hand, we use a simple gradient descent method to optimize actions[2], while traditional DDP [13] and recent methods (e.g. [16], [17], [18]) use a second-order algorithm like Newton’s method. In terms of convergence speed, our DDP is inferior to second-order DDP algorithms. The quality of the solution would be the same as far as we use the same evaluation function. Our algorithm is easier to implement. Since we use learned dynamics models, there should be more local optima than when using analytically simplified models. In order to avoid poor local maxima, our DDP uses a multiple criteria gradient descent method[3].

We consider a system with N decomposed processes illustrated in Fig. 5. The input of the n-th process is a state $\mathbf{x}_n$ and an action $\mathbf{a}_n$, and its output is the next state $\mathbf{x}_{n+1}$. Note that some actions may be omitted. A reward is given to the outcome state. Let $\mathbf{F}_n$ denote the process model: $\mathbf{x}_{n+1} = \mathbf{F}_n(\mathbf{x}_n, \mathbf{a}_n)$, and R_n denote the reward model: $r_n = R_n(\mathbf{x}_n)$. Typically the most important reward is the final reward $R_N(\mathbf{x}_N)$ and $R_n(\mathbf{x}_n) = 0$ for $n < N$, but we consider a general form $R_n(\mathbf{x}_n)$ so that we can give rewards (or penalties) for intermediate states. We assume that every state is observable.

Each dynamics model $\mathbf{F}_n$ is learned from samples with the neural networks mentioned in this paper where the prediction

[2]It works with any gradient descent methods.

[3]More specifically, in our DDP, we maintain “reference states” which are optimized by ignoring dynamics constraints. These reference states are used to create reference value functions. In our multiple criteria gradient descent method, we switch the evaluation functions in these value functions and the original one. Additionally, we also use multiple gradients computed from these value functions and the original evaluation function, which are useful to avoid local optima. See [1] for more details.

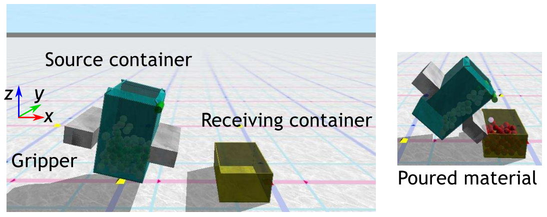


Fig. 6. Simulation environment. Right figure is after pouring material.

error is also modeled. At each step n, a state $\mathbf{x}_n$ is observed, and then an action $\mathbf{a}_n$ is planned with DDP so that an evaluation function $J_n(\mathbf{x}_n, \{\mathbf{a}_n, \dots, \mathbf{a}_{N-1}\})$ is maximized. J_n is an expected sum of future rewards, defined as follows:

$$J_n(\mathbf{x}_n, \{\mathbf{a}_n, \dots, \mathbf{a}_{N-1}\}) = \mathrm{E}[\textstyle\sum_{n'=n+1}^{N} R_{n'}(\mathbf{x}_{n'})] \\ = \textstyle\sum_{n'=n+1}^{N} \mathrm{E}[R_{n'}(\mathbf{x}_{n'})]. \quad (24)$$

In the DDP algorithm, first we generate an initial set of actions. Using this initial guess, we can predict the probability distributions of future states with the neural networks. The expected rewards are computed accordingly. Then we update the set of actions iteratively with a gradient descent method. In these calculations, we use the extensions of the neural networks described in this paper to compute the expectations, covariances, and the gradients. Thus we can replace LWR in [1] by the neural networks, and apply our DDP.

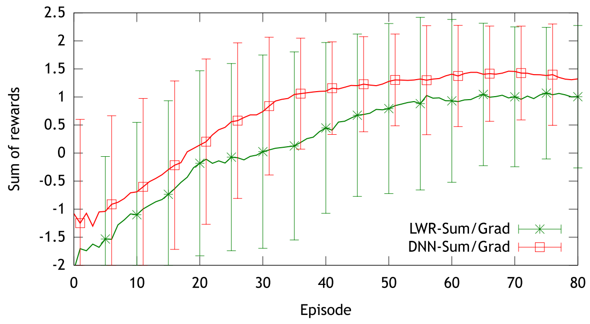
IV. Simulation Experiments

We verify our method in simulated experiments. The problem setup is the same as in [1]. In Open Dynamics Engine [23], we simulate source and receiving containers, poured material, and a robot gripper grasping the source container (Fig. 6).

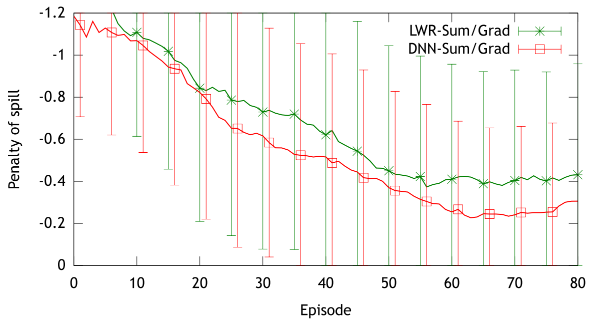
The materials are modeled with many spheres, simulating the complicated behavior of material such as tomato sauce during shaking. There are three typical failure cases: (1) pushed the receiving container, (2) materials bounced out, and (3) poured at wrong place. Note that even if we use the same parameters in the same initial condition, the result varies (we are not adding random noise); e.g. in a case where the materials spill out of the receiving container, but in another case there are no spilled materials. Thus the dynamics is complicated as in real pouring, and the simulation itself is a probabilistic process probably due to contact simulation effects and/or process communication delay (the simulator and the DDP program are connected over ROS).

We use the same state machines for pouring as those in [1], which are a simplified version of [7]. Those state machines have some parameters: a grasping height and pouring position. Those parameters are planned with DDP.

There are five separate processes. The initial or 0th state: before grasping. $\mathbf{x}_0$: x and y positions of the receiving container, $\mathbf{a}_0$: the grasping position. The 1st state: after grasping. $\mathbf{x}_1$: x and y position of the receiving container, and actual grasped position, $\mathbf{a}_1$: x and y position of the pouring



(a) Sum of rewards per episode.



(b) Penalty of spill per episode (y-axis is reversed).

Fig. 7. Learning curves of on-line learning. Moving average filter with 15 episode window is applied.

location. The 2nd state: after moving to the pouring location, and before the flow control. $\mathbf{x}_2$: x and y displacement of the receiving container, the speed of the receiving container, and the actual pouring location relative to the receiving container, $\mathbf{a}_2$: no action. The 3rd state: after the flow control. $\mathbf{x}_3$: the average flow center (x and y), the flow variance, and z-coordinate of the pouring location. $\mathbf{a}_3$: no action. The 4th state: after pouring. $\mathbf{x}_4$: the amount of materials poured in the receiving container (a_{rcv}), and the amount of spilled materials (a_{spill}). The reward consists of the amount a_{rcv}, the spilled penalty $-a_{\mathrm{spill}}$, and the penalty for the movement of the receiving container given at $\mathbf{x}_2$.

A. Comparison of Models in On-line Learning

We compare LWR and neural networks as the models of processes. Each LWR uses the same configuration as in [1]. Each neural network has three hidden layers; each hidden layer has 200 units. We use dropout [22] for each output of hidden layers during training where the dropout probability is 0.01.

We apply the learning framework of [1] in an on-line manner. After each state observation, we train the neural networks, and plan actions with the models. In the early stage of learning, we gather three samples with random actions, and after that we use DDP.

We compared a method using LWR (`LWR-Sum/Grad`) and one using neural networks (`DNN-Sum/Grad`). Fig. 7(a) shows the average learning curves of 10 trials, and Fig. 7(b) shows the penalty of spillage. `DNN-Sum/Grad` is better than `LWR-Sum/Grad`. The difference in the learning curves is

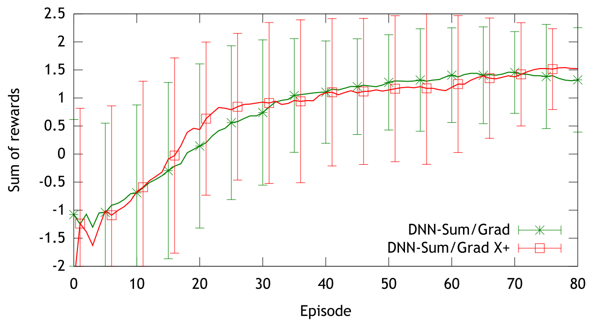


Fig. 8. Learning curves of on-line learning. Moving average filter with 15 episode window is applied.

mainly due to the penalty of spillage; the amount of spilled material is smaller in `DNN-Sum/Grad`. Since the dynamics of flow in our simulation model is very complicated, there was a small amount of spilled material even after the dynamics model is learned. This spilled amount is reduced when using neural networks, which indicates that the neural networks has a better ability to learn dynamics than LWR.

B. Using Redundant States

In the previous experiments, we used carefully selected states. Since we can expect automatic feature extraction with (deep) neural networks, we test alternative state vectors that have redundant and/or non-informative elements. Specifically, instead of the pose of the receiving container, we use four corner points on its mouth (x, y, z respectively). We also use the y-coordinate of the pouring position. Since we do not use the pose of the receiving container, the relative position of the flow center is computed with respect to the center of the four corner points. Consequently the dimensions of the state vectors $\mathbf{x}_0, \ldots, \mathbf{x}_4$ are changed from (2, 3, 5, 4, 2) to (12, 13, 16, 4, 2) respectively.

We used the same configuration as well as structure of hidden layers. Fig. 8 shows the average learning curves of 10 trials where the previous result (`DNN-Sum/Grad`) is plotted together with `DNN-Sum/Grad X+` that uses the expanded states. The learning curves of the two conditions are almost the same. This result indicates that although the total dimensionality changed from 16 to 47, the neural networks could extract the features as we expected. The two learning curves also show that the learning speeds are almost the same. The reason would be that the complexities of the dynamics are the same although the dimensionalities are different.

V. Preliminary Robot Experiments

We apply the method to a pouring task of a PR2 robot. Although the current results are preliminary, we think these are valuable to mention in this paper.

This task is a simplified version of [7]. Fig. 9 shows the setup. The robot starts pouring from an initial state shown in the figure with holding a source container. The position of the receiving container changes in each episode; a human operator places the container differently before each pouring episode. The position of the receiving container is measured by an external RGB-D sensor. First, the pose of the AR-marker is measured. Then the base cylinder of the receiving container is searched around the AR-marker position. We use a template matching method with rendered depth and surface-normal images using a ray-tracing method. The robot plans a collision-free trajectory to reach a pouring location, and executes flow control. Here we use only a shaking skill for flow control (flow control with tipping does not work in this case since the material jams inside the container). Two RGB cameras are used to measure the flow, the movement of the receiving container during the flow control, the amount of materials poured into the receiving container, and the amount of spilled materials. For the flow, we use the Lucas-Kanade method [24] to obtain optical flow. Then we compute the current flow amount, the flow center position, an average flow position, and the flow variance during flow control. The movement of the receiving container is measured by detecting colors; for this purpose, the base cylinder of the container is colored pink. The amount of materials is measured by simple color detection; for this purpose we use blue-colored materials. In order to decide that the materials are poured into the container or spilled out, we refer to the receiving container position measured by color detection. The two cameras measure these data independently. One is used as the x-axis data, and the other is used as the y-axis data. The human operator also modifies the camera positions when placing the receiving container to adjust visual measurements. After pouring a target amount, the robot moves the arm back to the initial pose.

The whole behavior is modeled with state machines. In this scenario, the robot plans a feasible pouring location, feasible trajectories, and shaking parameters ("feasible" means collision-free and IK-solvable). Although our final goal is solving those plans under a single framework, in this experiment we combine a classic method and the method proposed in this paper. DDP plans parameters that are hard to decide without learning the dynamics. The other parameters are planned using a classic method. For the pouring location, our DDP plans the z-coordinate and the distance from the receiving container's center. Other parameters (e.g. the orientation of the source container) are decided by an optimization with CMA-ES [25]. Here the evaluation function takes a better value if the source container is closer to the left side of the receiving container, and if the pose is feasible. Refer to [7] for the details of this optimization. The feasible

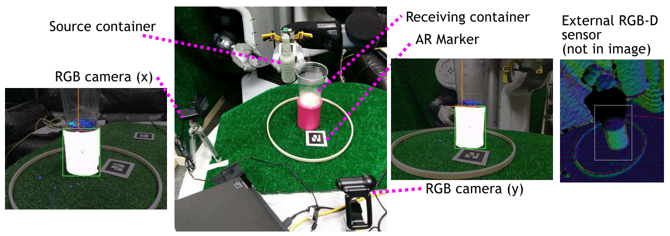


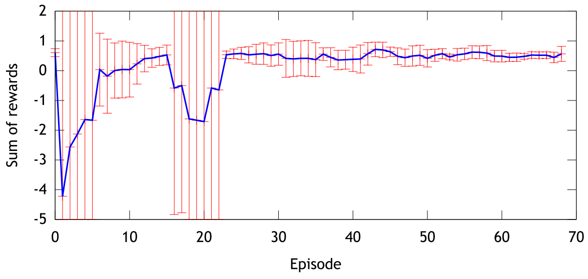
Fig. 9. Setup of the robot experiments.

trajectories are also planned with CMA-ES as mentioned in [7]. DDP also plans the amplitude of the shaking motion. We use a fixed shaking direction.

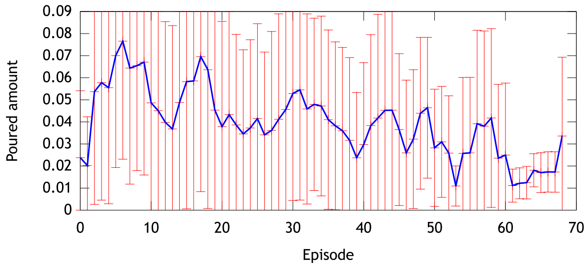
Planning with CMA-ES considers only a partial situation, while DDP plans the policy parameters considering the entire pouring process. We consider three decomposed processes for DDP. The initial or 0th state: at the initial state. $\mathbf{x}_0$: the position of the receiving container in the robot frame, $\mathbf{a}_0$: the pouring location parameters. The 1st state: after moving the source container to the pouring location and before the flow control. $\mathbf{x}_1$: the actual position of the pouring location, and the position of the receiving container in the robot frame, $\mathbf{a}_1$: the shaking amplitude. The 2nd state: at the end of the flow control. $\mathbf{x}_2$: the movement of the receiving container in the camera frame, the average flow position in the camera frame (relative to the the receiving container position), and the flow variance; the data from two cameras is used, $\mathbf{a}_2$: no action. The 3rd state: at the end of the pouring. $\mathbf{x}_3$: the amount of materials poured in the receiving container (a_{rcv}), and the amount of spilled materials (a_{spill}). The reward is mainly given to $(1-5(0.1-a_{\text{rcv}}))-100a_{\text{spill}}$ where 0.1 is the target amount. We use this small target amount in order to reduce the experiment time. A penalty is given to the movement of the receiving container at $\mathbf{x}_2$.

We conducted three runs with 38, 39, and 69 episodes respectively. For the first two runs we used dry peas colored blue, and for the third run we used blue plastic beads. Fig. 10(a), 10(b), 10(c) show the learning curves where the sum of rewards, the poured amount, and the spilled amount are plotted per episode respectively. The sum of rewards is improved with episodes. From Fig. 10(b) and 10(c) we can see a strategy of the robot. After 10 episodes, the poured amount curve is gradually decreasing in average. On the other hand, between 0th to 20th episodes, the spilled amounts are relatively large compared to the later part. Thus the robot first tried actions that produced a large amount, and then the robot adjusted the actions so that the spilled amount is reduced. This tendency would be changed by modifying the weights of poured and spilled amounts in the reward function.

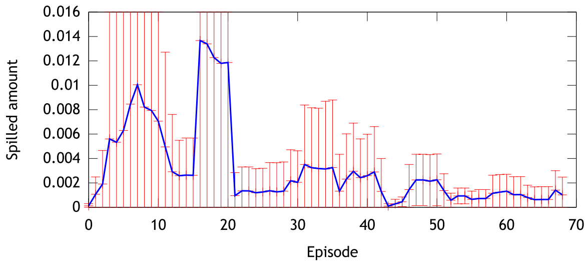
Fig. 11 shows part of the dynamics learned with neural networks. The graphs in Fig. 11 show the prediction from $\mathbf{x}_2$ to the reward for the poured and the spilled amounts. Since $\mathbf{x}_2$ has eight elements, only two elements are varied, and the median values of the samples are used for the other elements. From the 3D plot (a), we can see the samples are noisy, and there are some outliers. From the top view of the same graph (b), we can see a peak is located where the flow-y is around zero and the flow-x is around -0.1. Since the flow position is a relative value from the receiving container, this result is almost correct. On the other hand, the graph (c) is different from what we expected. The peak should be located where the flow-y is zero and the flow variance is also close to zero, but actually the peak is at a different position. Since DDP is used with the dynamics where (b) and (c) (and other not plotted elements) are unified, the problem in (c) is not a big issue now. However this might be a potential problem



(a) Sum of rewards per episode.



(b) Poured amount per episode.



(c) Spilled amount per episode.

Fig. 10. Learning curves (mean±1-SD) of the robot experiment. Moving average filter with 5 episode window is used.

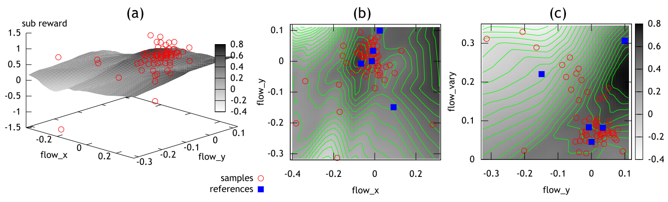


Fig. 11. Plot of the part of dynamics models. The predictions from x_2 to the reward for the poured and the spilled amounts are plotted. (a) and (b) are plots over the flow position (x and y). (c) is a plot over the flow y-position and the flow y-variance. (b) and (c) also show the contours where the reference states used in our DDP are also plotted. (zoom in using your PDF viewer)

in future.

Conducting better-controlled experiments is planned for future work. In the above experiments, the manual placement of the RGB cameras might produce noise although the stochastic extensions of neural networks could handle them. This should be improved. We also consider comparing neural networks with LWR and other function approximators in robot experiments as well as variations of neural networks such as different activation functions.

VI. Conclusion

As a model-based reinforcement learning method, we explored learning dynamics with neural networks and applying differential dynamic programming (DDP) to plan behaviors. Based on our recent work [1] where we used locally weighted regression to model temporally decomposed dynamics, we made use of neural networks with stochastic DDP. For this purpose, we extended neural networks with ReLU activation functions in terms of probability computations. We verified this method in pouring simulation experiments. The learning performance with neural networks outperformed that of locally weighted regression. The amount of spilled materials was reduced. In addition, we used redundant and/or non-informative states to investigate the feature extraction property of (deep) neural networks. Although the state dimensionality changed from 16 to 47, the learning performance did not change. The early results of the robot experiments using a PR2 were also positive. Therefore we think this framework, learning dynamics with deep neural networks and planning with stochastic DDP, is a promising solution to reinforcement learning problems.

Appendix

Here we derive $\mathrm{E}[f_{\mathrm{relu}}(x)]$ and $\mathrm{var}[f_{\mathrm{relu}}(x)]$ for $x \in \mathbb{R}$, $x \sim \mathcal{N}(\mu, \sigma^2)$, and $f_{\mathrm{relu}}(x) = \max(0, x)$. Let G denote a Gaussian function: $G(x; \mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} \exp(-\frac{(x-\mu)^2}{2\sigma^2})$. The expectation and variance are:

$$\mathrm{E}[f_{\mathrm{relu}}(x)] = \int_{-\infty}^{\infty} \max(0, x) G(x; \mu, \sigma^2) dx, \tag{25}$$

$$= \int_{0}^{\infty} \frac{x}{\sigma\sqrt{2\pi}} \exp(-\frac{(x-\mu)^2}{2\sigma^2}) dx, \tag{26}$$

$$\mathrm{var}[f_{\mathrm{relu}}(x)]$$

$$= \int_{-\infty}^{\infty} (\max(0, x) - \bar{y})^2 G(x; \mu, \sigma^2) dx, \tag{27}$$

$$= \int_{0}^{\infty} (x - \bar{y})^2 G(x; \mu, \sigma^2) dx + \int_{-\infty}^{0} \bar{y}^2 G(x; \mu, \sigma^2) dx, \tag{28}$$

where $\bar{y} = \mathrm{E}[f_{\mathrm{relu}}(x)]$. Thus we can solve these computations by finding this integral:

$$I = \int_{0}^{\infty} (ax^2 + bx + c) \exp(-\frac{(x-\mu)^2}{2\sigma^2}) dx, \tag{29}$$

where a, b, c are constants. We change the variable as $z = \frac{x-\mu}{\sqrt{2}\sigma}$, which gives:

$$I' = \int_{-\frac{-\mu}{\sqrt{2}\sigma}}^{\infty} (\alpha z^2 + \beta z + \gamma) \exp(-z^2) dz, \tag{30}$$

where $I = \sqrt{2}\sigma I$, $\alpha = 2a\sigma^2$, $\beta = \sqrt{2}\sigma(2a\mu + b)$, $\gamma = a\mu^2 + b\mu + c$. Using the Gaussian integral formulas,

$$\int \exp(-z^2) dz = \frac{\sqrt{\pi}}{2} \mathrm{erf}(z), \tag{31}$$

$$\int z \exp(-z^2) dz = -\frac{1}{2} \exp(-z^2), \tag{32}$$

$$\int z^2 \exp(-z^2) dz = \frac{\sqrt{\pi}}{4} \mathrm{erf}(z) - \frac{z}{2} \exp(-z^2), \tag{33}$$

we can compute I' and I. The result is:

$$I = \sigma^2(a\mu + b)X + \frac{\sqrt{\pi}\sigma}{\sqrt{2}}(a(\mu^2 + \sigma^2) + b\mu + c)(1 + E), \tag{34}$$

where $X = \exp(-\frac{\mu^2}{2\sigma^2})$, and $E = \mathrm{erf}(\frac{\mu}{\sqrt{2}\sigma})$. Using this result, we can compute Eq. (26) and (28), which gives Eq. (11) and (12) respectively.

References

[1] A. Yamaguchi and C. G. Atkeson, "Differential dynamic programming with temporally decomposed dynamics," in *the 15th IEEE-RAS International Conference on Humanoid Robots (Humanoids'15)*, 2015.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, 2012.

[3] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014.

[4] A. Toshev and C. Szegedy, "DeepPose: Human pose estimation via deep neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with deep reinforcement learning," in *NIPS Deep Learning Workshop 2013*, 2013.

[6] S. Levine, C. Finn, T. Darrell, and P. Abbeel, "End-to-end training of deep visuomotor policies," *ArXiv e-prints*, no. arXiv:1504.00702, 2015.

[7] 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, 2015.

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

[9] P. Kormushev, S. Calinon, and D. G. Caldwell, "Robot motor skill coordination with EM-based reinforcement learning," in *the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'10)*, 2010.

[10] A. Yamaguchi, J. Takamatsu, and T. Ogasawara, "DCOB: Action space for reinforcement learning of high dof robots," *Autonomous Robots*, vol. 34, no. 4, 2013.

[11] J. Kober, A. Wilhelm, E. Oztop, and J. Peters, "Reinforcement learning to adjust parametrized motor primitives to new situations," *Autonomous Robots*, vol. 33, 2012.

[12] S. Levine, N. Wagener, and P. Abbeel, "Learning contact-rich manipulation skills with guided policy search," in *the IEEE International Conference on Robotics and Automation (ICRA'15)*, 2015.

[13] 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, 1966.

[14] S. Schaal and C. Atkeson, "Robot juggling: implementation of memory-based learning," in *the IEEE International Conference on Robotics and Automation (ICRA'94)*, vol. 14, no. 1, 1994.

[15] 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.

[16] Y. Tassa and E. Todorov, "High-order local dynamic programming," in *Adaptive Dynamic Programming And Reinforcement Learning (ADPRL), 2011 IEEE Symposium on*, 2011.

[17] S. Levine and V. Koltun, "Variational policy search via trajectory optimization," in *Advances in Neural Information Processing Systems 26*, 2013.

[18] Y. Pan and E. Theodorou, "Probabilistic differential dynamic programming," in *Advances in Neural Information Processing Systems 27*, 2014.

[19] M. Riedmiller, "Neural fitted Q iteration – first experiences with a data efficient neural reinforcement learning method," in *In 16th European Conference on Machine Learning*, 2005.

[20] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*, 1998.

[21] 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*, 2012.

[22] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, 2014.

[23] R. Smith, *Open dynamics engine (ODE)*, http://www.ode.org/.

[24] B. D. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision," in *the 7th international joint conference on Artificial intelligence (IJCAI'81)*, 1981.

[25] N. Hansen, "The CMA evolution strategy: a comparing review," in *Towards a new evolutionary computation*, 2006.