Last layer Bayesian recurrent neural network and application in reinforcement learning

Yan Song*

MSc. Computational Statistics and Machine Learning

Supervisor: Prof. David Barber

Submission date: September 2020

Abstract

Neural network has become a powerful tool in the field of information engineering in recent years, such as in computer vision, natural language processing, bioinformatics, etc. At the same time, as the demands for probabilistic interpretation in deep learning arise greatly, various works have been done to incorporate uncertainty estimate.

Combining Bayesian modelling with deep learning methods has always been a research area with great potential. In this thesis, we present a last-layer Bayesian recurrent neural network which is a simplified version of fully-Bayesian RNN experimented in many literature [28, 8, 24]. It places a distribution on the last-layer parameters of the recurrent unit and offer estimate of two types of uncertainty: *aleatoric* and *epistemic* uncertainty. We have shown that such a partially-Bayesian dynamic model is sufficient to give descent prediction on a simple noise-free control environment and capture the uncertainty when the dynamic becomes stochastic.

The presented model is applied in model-based reinforcement learning (MBRL) in which we have shown that the incorporation of epistemic uncertainty provides stable and robust policy learning on both noise-free and noisy tasks. Furthermore, the model has also been shown to outperform a probabilistic dynamic model with no epistemic uncertainty estimate.

Acknowledgements

I would like to thank my academic supervisor, Prof. David Barber, for his valuable advice and insights on the topics. Our talks inspired me greatly which helps guiding the direction of the project.

Also I would like to thanks two of his PhD students, Peter Hayes and Mingtian Zhang, for their continuous supports and really helpful suggestions on experiment running.

Furthermore, my great appreciation goes to Marco Carobene, who is also a MSc. student exploring on these topics and a brilliant man to work with.

2020 has been such a tough year for all of us. Finally, my great thanks goes to my family and friends who offer constant encouragement to keep my spirits up during home-working period along in UK. Also, I would like to give my enthusiastic applause to all the medical staff for their selflessness. Stay safe everyone.

Contents

Chapter 1

Introduction

In this section we give a brief introduction on the problems discussed in the thesis and the motivation behind it. Meanwhile we will list the objective we aim to pursue and provide a brief overview of the content, outlining how the thesis is formatted.

1.1 Uncertainty estimate in deep learning

Neural network has become a powerful tool in the field of information engineering in recent years, such as in computer vision, natural language processing, bioinformatics, etc. At the same time, as the demands for probabilistic interpretation in deep learning arise greatly, various works have been done to incorporate uncertainty estimate in deep learning models. One particular way to introduce model uncertainty is by using *Bayesian neural network* which places distributions over the weights parameter [48, 11]. As the last layer of a neural network is usually a linear function, a simplified way of this is to estimate uncertainty by placing the weight only in this last layer, a model called *last-layer Bayes*(LLB) [73]. Compared to a fully-bayesian neural network, a LLB model is more model-agnostic meaning that it can be easily applied on many other deep learning models [73], such as recurrent neural network with different recurrent cell; and the model inference is much more simplified and has less computational burden.

In this work, we apply the idea of LLB to recurrent neural networks and present a variational approximation way to implement model learning. The goal is to evaluate the learning performance and we have carried out experiments on both deterministic and stochastic data in order to demonstrate that such a simple way of uncertainty estimate is sufficient to handle data in simple task probabilistically. The future extension of this work is therefore testing the model on more complex data such as with high dimension.

1.2 Uncertainty estimate in model-based planning

Besides model learning, the aim of the thesis is to examine how a LLB recurrent neural network leverage uncertainty estimate in *model-based reinforcement learning*(MBRL). An advantage of MBRL is data efficiency as the simulated model serves as a substitute of the real dynamic. However, the problems of *model bias* [17] hinder it from being wide-used in the field. Probabilistic

dynamic models are shown to alleviate the issue and provide great improvement in control task [17, 29]. Recently, the use of *recurrent state-space model*(RSSM) with *latent planning* applied in place have achieve state-of-the-art performance in various environment setting.

The objective of the rest of the thesis after model learning has been evaluated is therefore to show how the presented probabilistic dynamic model behaves in MBRL. In particularly, we compare LLB model with RSSM on both deterministic and stochastic environment and since LLB estimates model uncertainty whereas a RSSM does not, we look at whether it affects planning performance in both control task.

In summary, the objective of the thesis are listed as:

- Review various probabilistic dynamic models implemented in model-based reinforcement learning, including ensemble models [52], Gaussian process [17], Bayesian neural network [29, 11], etc.
- Present LLB recurrent neural network and variational approximation for training.
- Implement the presented model and evaluate the model learning performance on control tasks.
- Apply the model in gradient-based planning and evaluate the performance comparing to other baselines.

1.3 Chapter summary

Chapter 2 gives an overview of the backgrounds and related works in both probabilistic dynamics modelling and model-based reinforcement learning.

Chapter 3 will be mainly concentrating on the inference of a last-layer Bayesian recurrent neural network. This includes the description of model structure (Sec 3.1); the training objective function (Sec 3.2); the experiment setup (Sec 3.3) and the details of model learning experiments (Sec 3.4-3.6).

The rest part (Chapter 4-5) considers the experiments of applying LLB model in gradient-based planning on simple control task. This includes experiments done on a standard noise-free cart-pole balancing task (Sec 5.2) in which we also examine the situation when data is limited (Sec 5.3). Additionally, we also carry out experiment on a noisy version of cart-pole balancing (Sec 5.4) after which the thesis ends with summary of conclusion and potential future works (Chapter 6).

The code for experiments is available at `https://github.com/YanSong97`.

Chapter 2

Background and Related works

In this chapter, we will first introduce *Artificial Neural network*, a powerful and widely-used model in the field of deep learning. Two variants of it will be discussed, *Feed-forward Neural Network* and *Recurrent Neural network*. Next, we talk about the problem of lack of uncertainty quantification in these deterministic models, which brings us to Section 2.3 where Bayesian methods is applied, providing probabilistic interpretation to the deterministic model. In Section 2.4 we provide a summary of various implementations of probabilistic models in the field of *Model-based Reinforcement Learning*, as well as different *Planning* methods shown up in previous study.

2.1 Deterministic neural network

In a supervised learning task, given a set of input featured vectors $\mathcal{X} = \{\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_T\}, \mathbf{x}_t \in \mathcal{R}^n$ and the corresponding labels $\mathcal{Y} = \{\mathbf{y}_1, \mathbf{y}_2, \cdots, \mathbf{y}_T\}, \mathbf{y}_t \in \mathcal{R}^m$, the objective of a machine learning algorithm is to infer a function $\mathbf{f} : \mathcal{X} \to \mathcal{Y}$ from the training data pair $(\mathcal{X}, \mathcal{Y})$, which can be used to map new unseen samples to predicted labels. A classical and fairly simple example is *linear regression*[30]. In which a linear function $\mathbf{f}_{LR}$ is constructed as:

$$\hat{\mathbf{y}}_t = \mathbf{f}_{LR}(\mathbf{x}_t) = \mathbf{W} \cdot \mathbf{x}_t + \mathbf{b} \ , \ \ \mathbf{W} \in \mathcal{R}^{m \times n} \ , \ \ \mathbf{b} \in \mathcal{R}^m$$

where W is some m by n matrix and $\mathbf{b}$ is some real vector with m elements. The value of $\mathbf{W}, \mathbf{b}$ represents different linear transformation of input sample $\mathbf{x}_t$. The best value can be found by, for instance, minimising the mean square error loss denoted as:

$$\mathcal{L}_{LR} = \frac{1}{T} \sum_{t=1}^{T} \|\mathbf{y}_t - \mathbf{f}(\mathbf{x}_t)\|^2$$

In some complex tasks where $\mathbf{x}$ and $\mathbf{y}$ are not linear, a simple linear regression might not suffice. Apart from its non-linear variants [5], a neural network model [60] introduces non-linearity by applying an element-wise non-linear *activation function* $\boldsymbol{\sigma}(\cdot)$ (such as Sigmoid, TanH and Rectified linear unit (ReLU)). Then the output is linearly transformed again by another *weight matrix* $\mathbf{W}_2$.

The formula is represented as:

$$\hat{\mathbf{y}}_t = \mathbf{W}_2 \cdot \boldsymbol{\sigma}(\mathbf{W}_1 \cdot \mathbf{x}_t + \mathbf{b}_1) + \mathbf{b}_2 \ , \ \mathbf{W}_1 \in \mathcal{R}^{d \times n} \ , \ \mathbf{W}_2 \in \mathcal{R}^{m \times d} \ , \ \mathbf{b} \in \mathcal{R}^d \ , \ \mathbf{b}_2 \in \mathcal{R}^m \tag{2.1}$$

where d is the dimension of intermediate state. One can see that the input vectors $\mathbf{x}_t \in \mathcal{R}^n$ is first projected onto a higher (or lower, depending on the value of d) dimensional space, where it goes through a non-linear function before being projected onto the output space $\mathcal{R}^m$.

2.1.1 Feed-forward neural network

In a feed-forward neural network setting, the input vector $\mathbf{x}_t$ in equation 2.1 is referred to as a *input layer*, the first linear transformation $\mathbf{W}_1 \cdot \mathbf{x}_t + \mathbf{b}_1$ is the *hidden layer* and the second one $(\mathbf{W}_2, \mathbf{b}_2)$ is the *output layer*. Similar to linear regression, the best value of the model parameters $\boldsymbol{\theta} = \{\mathbf{W}_1, \mathbf{W}_2, \mathbf{b}_1, \mathbf{b}_1\}$ are found by minimising some objective function, such as the mean square error loss function between $\hat{\mathbf{y}}$ and $\mathbf{y}$.

In practice, the feed-forward neural network is flexible. By concatenating multiple neural network layers one can design highly complex models with richer hidden representation. This is often call a *Multi-layer perceptron* (MLP) and it has been long-established in various classification tasks [40, 49, 76].

2.1.2 Recurrent neural network (RNN)

One variant of feed-forward neural network is *Recurrent neural network* [60, 74], which aims at handling sequential data. As illustrated in Fig 2.1, a recurrent neural network model can be seen as a single feed-forward neural network but taking the input $\{\mathbf{x}_0, \cdots, \mathbf{x}_t\}$ sequentially. The model parameters in each RNN unit (A in the graph) are shared across time steps, therefore enabling the model to process and generate sequences with variable length. These make RNN a dominant choice of model design in varying *Natural Language processing* task. [33, 67, 2]. The mathematical formulation of RNN is:

$$\begin{aligned} \mathbf{h}_t &= \tanh\big(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{hx}\mathbf{x}_t + \mathbf{b}_h\big) \\ \mathbf{y}_t &= \mathbf{W}_{yh} \cdot \mathbf{h}_t + \mathbf{b}_{yh} \\ \mathbf{x}_t &\in \mathcal{R}^n \ , \ \mathbf{y}_t \in \mathcal{R}^m \ , \ W \in \mathbf{R}^d \\ \mathbf{W}_{hx} &\in \mathcal{R}^{d \times n} \ , \ \mathbf{W}_{hh} \in \mathcal{R}^{d \times d} \ , \ W_{yh} \in \mathcal{R}^{m \times d} \end{aligned} \tag{2.2}$$

where $(\mathbf{x}_{1:T}, \mathbf{y}_{1:T})$ is the training data sequence pair and $\mathbf{h}_{1:T}$ are the hidden states (A in graph) of RNN.

Due to the lack of long-term prediction and the vanishing gradient gradient problem [9] in RNN, *Long Short Term Memory* (LSTM) [38] and *Gated Recurrent Unit* (GRU) [13] were proposed which give promising performance. In our experiment, we are mainly focusing on implementing LSTM.

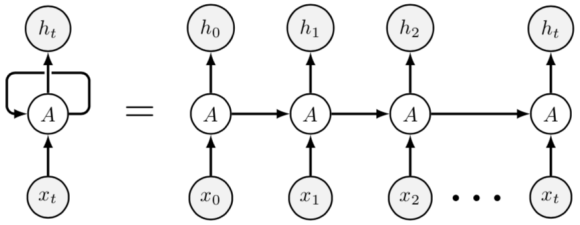


Figure 2.1: Schematic diagram of a simple Recurrent neural network seen as a sequence of a neural network.

2.2 Uncertainty quantification

2.2.1 Aleatoric and epistemic uncertainty

In machine learning field, uncertainty is usually categorised into two types: *Aleatoric* and *Epistemic* uncertainty [26]. Aleatoric uncertainty is also referred to as *data uncertainty*, which generally exists in real experiment since the measurement is unlikely to be perfectly accurate. While epistemic uncertainty is also referred to as *model uncertainty*, this includes being uncertain about the model parameters as well as the model structure [26]. Note that epistemic/model uncertainty is an essential topic in this report and will be mentioned repeatedly.

As for a neural network in classification task, a soft-max layer outputting a distribution on predicted labels is often claimed as modelling epistemic uncertainty. A soft-max function is denoted as:

$$softmax\ (\mathbf{x})_i = \frac{exp(\mathbf{x}_i)}{\sum_{j=1}^{n} exp(\mathbf{x}_j)}$$

which generates a discrete probability distribution over labels and one can sample from it to get prediction. However, it is argued that this is not showing model confidence [26]. The reason for this will be given in Section 2.3.2 which states that the optimised weight only gives point estimate.

One more natural way to model epistemic uncertainty in a neural network is by placing a probability distribution on the parameters of the model (weight and bias term in neural network) [48]. This is also known as *Bayesian neural network* and will be further discussed in next section.

2.2.2 Application

Demand for uncertainty quantification arises in diverse area of science. For example in medical decision making, a decisive result coming from, says a deterministic neural network, does not provide enough trust in the model nor in the predicted decision [7]. Especially in a case when the decision is critical, such as patient diagnosis. In deep learning, neural network model suffers from *over-fitting* [69]. This implies that the model has great performance at a particular set of training data but terrible at generalising it on unseen data. Regularisation methods preventing over-fitting include early-stopping, *cross-validation* [66], *spectral filtering* [16] and also probabilistic schemes such as *dropout* [27] and Bayesian treatment.[11, 32]. In *Reinforcement Learning*, uncertainty quantification helps solving model-bias problem [17] and also leads to *Exploitation-exploration*

trade-off [58, 11] during planning. Section 2.4.1 reviews the role of uncertainty more explicitly in MBRL.

2.3 Bayesian neural network (BNN)

Different from a standard neural network which gives point estimate, a Bayesian neural network places a distribution on model parameter and applies posterior inference to carry out training. In BNN, a prediction is made by model averaging, meaning a decision is made after considering all possible value of model parameter. So that a BNN tends to have richer model representation and therefore offer better robustness [11].

Training a Bayesian neural network relies on *Bayesian inference* [12]. We first give a short introduction to it and then discuss how it is applied on a neural network. Next we consider applying uncertainty estimation only on last layer of a neural network before moving on to Bayesian recurrent neural network.

2.3.1 Bayesian inference

In a classification task, assume we have data pairs $(\mathcal{X}, \mathcal{Y})$, $\mathcal{X} = \{\mathbf{x}_1, \cdots, \mathbf{x}_T\}$, $\mathcal{Y} = \{y_1, \cdots, y_T\}$ and some function $\mathbf{f}_\theta$ with model parameters denoted as θ. Then according to *Bayes' theorem* [12], placing a prior distribution on θ, denoted as $p(\theta)$, leads to posterior inference as:

$$p(\theta|(\mathcal{X}, \mathcal{Y})) = \frac{p(\mathcal{Y}|\mathcal{X}, \theta)p(\theta)}{p(\mathcal{Y}|\mathcal{X})} = \frac{p(\mathcal{Y}|\mathcal{X}, \theta)p(\theta)}{\int p(\mathcal{Y}|\mathcal{X}, \theta)p(\theta)\, d\theta}$$

where $p(\mathcal{Y}|\mathcal{X}, \theta)$ is the *likelihood probability*, understood as the probability of obtaining model output $\mathbf{f}_\theta(\mathcal{X}) = \mathcal{Y}$ given model parameter θ. The denominator $p(\mathcal{Y}|\mathcal{X})$ is a normaliser also known as *model evidence* and it can be computed by marginalising the likelihood probability over prior distribution $p(\theta)$. Different from prior $p(\theta)$, the posterior distribution $p(\theta|\mathcal{X}, \mathcal{Y})$ quantifies the belief on model parameters under the light of data pairs $(\mathcal{X}, \mathcal{Y})$.

Indication of model averaging (or marginalising) occurs in model prediction. Given an unseen testing data $\mathbf{x}^*$, the model generates a prediction probability by:

$$p(\mathbf{y}^*|\mathcal{X}, \mathcal{Y}, \mathbf{x}^*) = \int p(\mathbf{y}^*|\theta, \mathbf{x}^*)\, p(\theta|\mathcal{X}, \mathcal{Y})\, d\theta$$

The integration on posterior distribution can be seen intuitively as taking full consideration of all possible values of model parameter θ and weight the prediction accordingly. Thus, the output gains robustness under the variation in the model parameter space, but the amount and quality of the variation is learnt from the training data $(\mathcal{X}, \mathcal{Y})$, or to be more specific, from the variability of the data [11].

Next, we apply this Bayesian idea on a neural network model.

2.3.2 Applying Bayesian method on neural network

We first state the difference between neural network as a probabilistic model and Bayesian neural network.

Neural network as a probabilistic model

As we discussed in Section 2.2 that applying a soft-max layer on the output of the neural network is claimed to demonstrate model confidence. However, the weight in such a probabilistic neural network is still a point estimate. For example, denote the model as $\mathbf{f}_\theta$ with weight parameter θ and training data pairs as $\mathcal{D} = (\mathcal{X}, \mathcal{Y})$. The output of the model can either follow a categorical distribution:

$$\hat{y}^* | \mathbf{x}^* \sim \text{Cate}(p_1, \cdots, p_T) \ , \ \ (p_1, \cdots, p_T) = \mathbf{f}_{\theta|\mathcal{D}}(\mathbf{x}^*)$$

or a Gaussian distribution:

$$\hat{y}^* | \mathbf{x}^* \sim \mathcal{N}(\mu, \sigma^2) \ , \ \ (\mu, \sigma^2) = \mathbf{f}_{\theta|\mathcal{D}}(\mathbf{x}^*)$$

The parameter θ can be learnt by either *maximum likelihood estimator* (MLE) or *maximum a posteriori* (MAP) [11, 71] with weight regularisation applied:

$$\begin{aligned} \theta_{MLE} &= \arg\max_\theta \ p(\mathcal{X}, \mathcal{Y} | \theta) \\ &= \arg\max_\theta \sum_i \log \ p(y_i | \mathbf{x}_i, \theta) \end{aligned} \tag{2.3}$$

$$\begin{aligned} \theta_{MAP} &= \arg\max_\theta \ \log \ p(\mathcal{X}, \mathcal{Y} | \theta) + \log \ p(\theta) \\ &= \arg\max_\theta \ \sum_i \log \ p(y_i | \mathbf{x}_i, \theta) + \log \ p(\theta) \end{aligned}$$

with $p(\theta)$ a prior distribution on θ as a regulariser preventing θ being too far away from its prior belief. These optimisation of model parameters is still giving a point estimate, not showing the sign of epistemic uncertainty estimate, while a Bayesian neural network does it differently.

Variational inference on model weights

In Bayesian NN setting, a prior is placed on all (or parts of) the model weight $\mathbf{W}$ and the objective is to find out the posterior distribution $p(\mathbf{W} | \mathcal{X}, \mathcal{Y})$ conditioned on training data. According to Sec. 2.3.1, this is done by:

$$p(\mathbf{W} | \mathcal{X}, \mathcal{Y}) = \frac{p(\mathcal{Y} | \mathcal{X}, \mathbf{W}) p(\mathbf{W})}{\int p(\mathcal{Y} | \mathcal{X}, \mathbf{W}) p(\mathbf{W}) \ d\mathbf{W}} \ , \quad p(\mathcal{Y} | \mathcal{X}, \mathbf{W}) = p(\mathcal{Y} | \mathbf{f}_\mathbf{W}(\mathcal{X}))$$

However in practice, the non-linearity in function $\mathbf{f}$ (due to the non-linear activation function) makes the integral in the denominator analytically intractable. Variational approximation [3] is used to directly approximate the posterior $p(\mathbf{W} | \mathcal{X}, \mathcal{Y})$ with some variational distribution $q(\mathbf{W} | \theta)$ in previous studies [32, 11]. This is achieved by minimising the *Kullback-Leibler* (KL) divergence

[46] between $q(\mathbf{W}|\theta)$ and the true posterior distribution $p(\mathbf{W}|\mathcal{X}, \mathcal{Y})$. Here KL divergence serves as a measure of discrepancy between distributions [3] where zero implies identity and higher the value is, more different these distributions are.

One can also expand the expression for KL and derive the *variational free energy* objective function [50, 11]:

$$\begin{aligned}
\theta^* &= \arg\min_{\theta} \ \mathbf{KL}\Big[q(\mathbf{W}|\theta)\Big\|p(\mathbf{W}|\mathcal{X}, \mathcal{Y})\Big] \\
&= \arg\min_{\theta} \ \int q(\mathbf{W}|\theta) \log \frac{q(\mathbf{W}|\theta)}{p(\mathbf{W}|\mathcal{X}, \mathcal{Y})} \ d\mathbf{W} \\
&= \arg\min_{\theta} \int q(\mathbf{W}|\theta) \log \frac{q(\mathbf{W}|\theta)}{p(\mathcal{Y}|\mathcal{X}, \mathbf{W})p(\mathbf{W})} \ d\mathbf{W} \\
&= \arg\min_{\theta} \int q(\mathbf{W}|\theta) \log \frac{q(\mathbf{W}|\theta)}{p(\mathbf{W})} \ d\mathbf{W} - \int q(\mathbf{W}|\theta) \log p(\mathcal{Y}|\mathcal{X}, \mathbf{W}) \ d\mathbf{W} \\
&= \arg\min_{\theta} \ \mathbf{KL}\Big[q(\mathbf{W}|\theta)\Big\|p(\mathbf{W})\Big] - \mathbb{E}_{q(\mathbf{W}|\theta)}\Big[\log p(\mathcal{Y}|\mathcal{X}, \mathbf{W})\Big]
\end{aligned} \tag{2.4}$$

This is also called the *Evidence lower bound* (ELBO) in some of the Bayesian literature [10, 77].

Therefore, the parameter θ of variational distribution $q(\mathbf{W}|\theta)$ can be found by optimising the free energy function:

$$\mathcal{F}(\mathcal{X}, \mathcal{Y}, \theta) = \mathbf{KL}\Big[q(\mathbf{W}|\theta)\Big\|p(\mathbf{W})\Big] - \mathbb{E}_{q(\mathbf{W}|\theta)}\Big[\log p(\mathcal{Y}|\mathcal{X}, \mathbf{W})\Big] \tag{2.5}$$

Given with sufficient statistics of both prior $p(\mathbf{W})$ and variational distribution $q(\mathbf{W}|\theta)$ (or parametrised by parameter θ), the KL divergence term can be written in a closed-form. (An uni-Gaussian example is given in Appendix A.1). However, the issue is with the expectation likelihood function on the right side of eq.2.5 due to the non-linearity of the neural network. A typical method to approximate the expectation is *Monte-Carlo samping* [12]:

$$\mathbb{E}_{q(\mathbf{W}|\theta)} \approx \frac{1}{N} \sum_{i=1}^{N} \log p(\mathcal{Y}|\mathcal{X}, \mathbf{W}^{(i)}) \ , \quad \mathbf{W}^i \sim q(\mathbf{W}|\theta) \ , \quad i = 1, \cdots, N$$

But often we are more interested in finding the derivative of the expectation with respect the the distribution parameter θ. Here we discuss two widely-used methods.

a. REINFORCE trick [75]

Also known as *likelihood ratio gradient* [52]. With tricky equation $\frac{\partial p_\theta(x)}{\partial \theta} = p_\theta(x)\frac{\partial \log\, p_\theta(x)}{\partial x}$, the derivative of expectation w.r.t θ is: $\big($set $f(\mathbf{W}) = \log\, p(\mathcal{Y}|\mathcal{X}, \mathbf{W})\big)$

$$\begin{aligned}\nabla_\theta \mathbb{E}_{q(\mathbf{W}|\theta)}\Big[f(\mathbf{W})\Big] &= \int \nabla_\theta q(\mathbf{W}|\theta)\; f(\mathbf{W})\; d\mathbf{W} \\ &= \int q(\mathbf{W}|\theta)\frac{\nabla_\theta q(\mathbf{W}|\theta)}{q(\mathbf{W}|\theta)}\; f(\mathbf{W})\; d\mathbf{W} \\ &= \int q(\mathbf{W}|\theta)\; \nabla_\theta \log\, q(\mathbf{W}|\theta)\; f(\mathbf{W})\; d\mathbf{W} \\ &= \mathbb{E}_{q(\mathbf{W}|\theta)}\Big[\nabla_\theta \log\, q(\mathbf{W}|\theta)\; f(\mathbf{W})\Big]\end{aligned} \tag{2.6}$$

Then $\Big(\nabla_\theta \log\, q(\mathbf{W}^{'}|\theta)\;\; f(\mathbf{W}^{'})\Big)$ with $\mathbf{W}^{'} \sim q(\mathbf{W}|\theta)$ is an unbiased estimator of the derivative of expectation $\nabla_\theta \mathbb{E}_{q(\mathbf{W}|\theta)}\Big[f(\mathbf{W})\Big]$ and it can be easily computed by sampling. Note that the reason we do not take the gradient of $f(\mathbf{W})$ with respect to θ is that only distribution $q(\mathbf{W})$ is parametrised by θ but not the variable $\mathbf{W}$.

This method avoids taking the gradient of f and therefore it can be applied even in the case where f is non-differentiable, which distinguishes itself from the next method. But this method suffers from high variance at the same time [52].

b. Reparametrisation trick [43]

Reparametrisation trick make it possible to take gradient directly with respect to the distribution parameter through samples. Taking a Gaussian distribution $p_\theta(x) = \mathcal{N}(\mu, \sigma^2)$ as an example, one can reparametrise a sample as a function $g(\cdot)$ of $[\mu, \sigma]$ and a sample from a standard normal denoted as $p(\epsilon)$, formulated as:

$$x^{(i)} = g(\theta) = \mu + \sigma \cdot \epsilon^{(i)}\;,\;\; \epsilon^{(i)} \sim p(\epsilon) = \mathcal{N}(0, 1)$$

where $x^{(i)}$ can be seen as a sample from a parameter-free unit Gaussian transformed by mean and standard deviation (μ, σ) and so that the gradient with respect to the sample $x^{(i)}$ can directly pass through parameter of distribution using chain rule:

$$\frac{\partial x^{(i)}}{\partial \mu} = 1\;,\;\; \frac{\partial x^{(i)}}{\partial \sigma} = \epsilon^{(i)}$$

and the derivative of expectation log likelihood in e.q 2.5 can be computed as: $\big($assume $q(\mathbf{W}) \sim \mathcal{N}(\mu_W, \sigma_W^2)$, $f(\mathbf{W}) = \log\, p(\mathcal{Y}|\mathcal{X}, \mathbf{W})\;\big)$

$$\begin{aligned}\nabla_\theta \mathbb{E}_{q(\mathbf{W}|\theta)}\Big[f(\mathbf{W})\Big] &= \nabla_\theta \mathbb{E}_{p(\epsilon)}\Big[f\big(\mu_W, \sigma_W, \epsilon\big)\Big] \\ &= \mathbb{E}_{p(\epsilon)}\Big[\nabla_\theta f\big(\mu_W, \sigma_W, \epsilon\big)\Big]\end{aligned} \tag{2.7}$$

where it is required that function f is differentiable.

Thus far, we have the objective function and the tools for optimisation. In next chapter we will provide detailed derivation for learning a Bayesian neural network.

2.3.3 Last-layer-only Bayesian neural network

In practice use, a neural network model often has large number and size of hidden layers. Thus, it can be computational expensive and extremely hard to train a Bayesian neural network with distribution on all model parameters. An intermediate is to apply distribution only at the last layer of the network and use the rest part of the model as a feature extractor preparing input for the last-layer Bayesian linear model. This idea is often referred to as *Bayesian last layer model*[73] and has already been experimented in a bandit setting [73, 58]. Advantages of estimating uncertainty only in the last layer includes simplicity and scalability to larger models.

2.3.4 Bayesian recurrent neural network

In sequential modelling task, Bayesian scheme has also been explored on a recurrent neural network model, such as dropout [78, 28]. Dropout is often used in feed-forward neural network to avoid over-fitting. However, for a recurrent neural network model, it was discovered that a naive dropout method can only bring uncertainty to encoding and decoding layer [78] otherwise adding noise to recurrent layers can deteriorate model performance for long-term sequences. A new variant of dropout was proposed [27] to overcome this by sharing the dropout mask for encoding, decoding and recurrent parameters.

Standard back-propagation has also been experimented on a fully Bayesian recurrent neural network [8]. In which a weight parameter $\mathbf{W} \in \mathbb{R}^{m \times n}$ is represented by a mean and a diagonal covariance parameter $[\mathbf{W}^{\mu}, \mathbf{W}^{\Sigma}] \in \mathbb{R}^{m \times n}$ such that for a recurrent transition function:

$$\mathbf{h}_t = f_\theta(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t)$$

the output at each time step is represented by mean and variance:

$$\text{MEAN}_{h,x,t} = \mathbf{W}_h^{\mu} \mathbf{h}_{t-1}^{\mu} + \mathbf{W}_x^{\mu} \mathbf{x}_t$$

$$\text{VAR}_{h,x,t} = \mathbf{W}_h^{\Sigma} \mathbf{h}_{t-1}^{\Sigma} + \mathbf{W}_x^{\Sigma} \mathbf{x}_t$$

and the gradient of marginal likelihood at each time step is back-propagate to update the mean and variance parameters.

A more rigorously bayesian way to train a BRNN is applying variational approximation [24]. Similar to Bayesian neural network where we approximate the posterior distribution on model parameter $q(\mathbf{W})$ by maximising the free energy function (eq.2.5), a Bayesian RNN is trained by maximising the free energy as well:

$$\mathcal{F}(\mathbf{x}_{1:T}, \mathbf{y}_{1:T}, \theta) = \mathbf{KL}\Big[q(\mathbf{W}|\theta)\Big\|p(\mathbf{W})\Big] - \mathbb{E}_{q(\mathbf{W}|\theta)}\Big[\log p(\mathbf{y}_{1:T}|\mathbf{x}_{1:T}, \mathbf{W})\Big]$$

where $(\mathbf{x}_{1:T}, \mathbf{y}_{1:T})$ are sequential input-output pairs.

In this report, we are proposing a Bayesian recurrent neural network but only with distribution on the weight of last layer of recurrent unit. We call this *Last-layer-Bayes* (LLB) recurrent neural network. Compared to a fully Bayesian neural network in which all parameters in recurrent layer are stochastic, placing uncertainty only in the last layer is expected to simplify the inference and lessen the computational burden while still maintaining probabilistic interpretation. Meanwhile, a Bayesian last-layer is model-agnostic and can be easily applied in different deep learning models, for instance a recurrent neural network with different recurrent cell. The detailed inference of the LLB model can be found in Chapter 3.

Next section discusses model-based reinforcement learning where our model is tested on.

2.4 Model-based Reinforcement learning

Reinforcement learning (RL) is an area of study where an agent aims at learning a control task by interacting with the environment [68]. Examples can be self-driving cars and robot navigation. A reinforcement learning task is mathematically formalised as a *markov decision process* (MDP) problem. A trajectory of MDP can be denoted as:

$$s_0 \to a_0 \to (r_1, s_1) \to a_1 \to (r_2, s_2) \to \cdots \quad , \quad s \in \mathcal{S}; a \in \mathcal{A}$$

where s_t represents environment's state at time step t, a_t represents an action selected by the agent at state s_t and tuple (r_{t+1}, s_{t+1}) represents the next state and feedback from the environment after taking action a_t. A MDP follows *Markov property* which states that the transition to next state s_{t+1} only depends on last state s_t and action taken a_t, formulated as: for some transition function $T(s_{t+1}|s_t, a_t)$ we have

$$T(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \cdots, a_0, s_0) = T(s_{t+1}|s_t, a_t)$$

Then the goal is to learn an optimal policy $\pi : \mathcal{S} \to \mathcal{A}$ maximising some reward function $R(\cdot)$ defined specifically for the problem.

RL can be further categorised into *model-free* and *model-based* depending on whether to model the environment dynamic. In a model-free setting, it is common to learn the *Q-value* instead which estimate rewards of different actions using models such as a neural network [51]. The optimal policy might be obtained by selecting the action with highest Q-value. This makes the learning fast and easy for implementation but require massive interaction with the real environment. While a model-based method learns a model of dynamic first and infers a policy from the learnt simulated environment. This often provide more computational burden but is more data-efficient than a model-free method, so that it is applicable in a scarce data cases [1, 62].

2.4.1 Role of uncertainty in model-based RL

However, learning dynamic model from scratch suffer from *model bias* in a RL setting [63]. As we have discussed in Sec.2.2 that a deterministic model shows lack of generalisation by placing full confidence in a single prediction result. This is essential in reinforcement learning area especially when the number of available data is limited. In this situation a deterministic model can easily be over-confident and gives unreliable prediction [17]. One way to alleviate this is to use a probabilistic model estimating uncertainty such as *PILCO* [17], a model-based policy search method using *Gaussian process* dynamic model to propagate state uncertainty. The uncertainty is also incorporated in planning by integrating out the state distribution when computing the expected rewards. This is shown to be beneficial in small-sample case in control task.

Besides reducing model-bias, uncertainty in model-based RL can also help balancing explore-exploit trade-off [18]. Exploration is when the agent want to develop novel strategies (or actions) potentially more beneficial and exploitation is when it follows the current optimal policy in order to maximise gained rewards. One example of the conventional $\epsilon-$*greedy* policy [68] where an agent selects the current best action with probability $1-\epsilon$ and explores others randomly with probability ϵ. This makes sure that the agent is always sceptical of the current policy and open to new actions. For a parametric policy function $\pi_\phi(\cdot)$, this can be referred to as using a Bernoulli controller in discrete action case or adding a Gaussian noise to the generated actions in a continuous case [23]:

$$a_t = \pi_\phi(s_t) + \mathcal{N}(0, \sigma^2 I) \;\text{ or }\; a_t \sim \mathcal{N}(\pi_\phi(s_t), \sigma^2 I) \tag{2.8}$$

However, such random exploration suffers from low efficiency in many environment due to its state-independence [23], which means that each time visiting the same state s in a model rollout, the agent will most likely generate different actions. A way to include state-dependent exploration is by injecting noise in policy parameter space instead [23, 54]:

$$a_t = \pi_\phi(x_t)\ ,\quad \phi := \mu + \Sigma \odot \epsilon\ ,\quad \epsilon \sim \mathcal{N}(0, I)$$

where (μ, Σ) are learnable parameters and $\odot$ represents element-wise multiplication. Therefore, for each model rollout if one samples a policy parameter $\theta^{'}$ at the beginning and keeps it fixed throughout the whole trajectory, the state-dependent exploration is enforced. Fig.2.2 shows how two types of exploration differ.

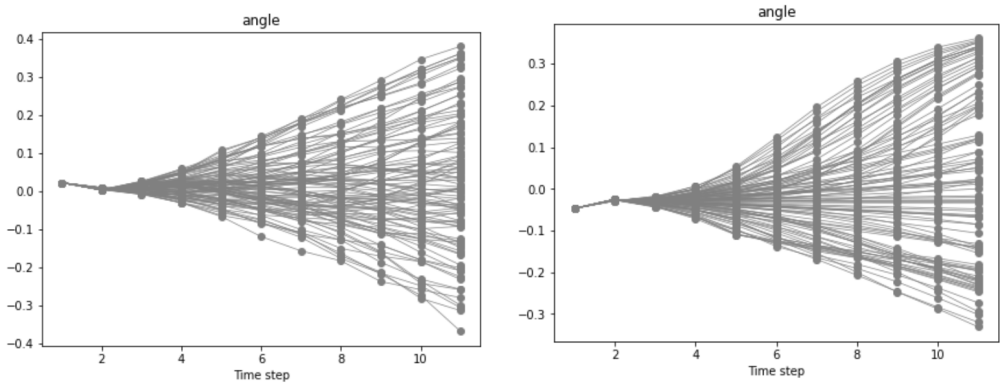


Figure 2.2: State exploration in angle of a single model rollout during planning under continuous Cart-pole environment. Given an initial state, 100 particles are created and passed through the model. **Left**: at each time step an action is re-sampled from a Gaussian distribution (like eq.2.8), this leads to the interweave of particle trajectories seen in the plot ; **Right**: before the rollout, values of policy parameters ϕ are sampled and fixed throughout the entire rollout, this ensures the consistence in perturbation over time step in the same model rollout and therefore smooths out the exploration plot.

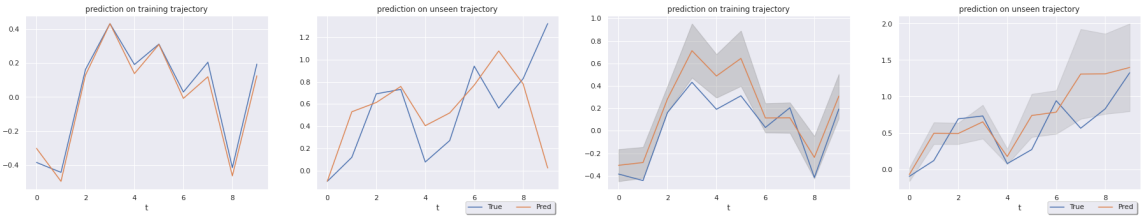
In our experiment we focus mainly on incorporating uncertainty in the model as opposed to in policy. In next section we provide a summarisation of probabilistic models ever used in model-based reinforcement learning.

2.4.2 Summary of probabilistic models used in MBRL

This section considers the recent application of probabilistic models in model-based reinforcement learning. Compared to a model-free agent such as *Q-learning* [68], a MBRL agent can more efficiently extract valuable information from data, such as dynamic of the environment, based on which a policy can be updated accordingly without real-time interaction. However, this inherently requires a reliable model of the real environment since the agent assumes the learnt model mimics a perfect environmental setting which is often impossible in practice [61]. This is known as the problem of *model-bias* and the issue gets worse when the number of available data is limited or the environment has larger stochasticity.

Probabilistic dynamic model has been shown to alleviate the problem in various works [17, 29, 20]. Fig 2.3 illustrates how a deterministic and a probabilistic models cope with model-bias under a noisy environment. Deterministic LSTM (Fig 2.3(a)) gives a almost perfect match on training data which implies over-fitting to the noise, as expected it shows great discrepancy on testing data. On stochastic environment a deterministic model predicts with full confidence which might bias the policy searching to a large extent. Bayesian LSTM (Fig 2.3(b)) on the other hand, maintains a distribution over model weights and its prediction as well. This prevents the model from severe over-fitting to the noise and therefore offers better generalisation on unseen data. Next we summarise different types of probabilistic models implemented in MBRL.

Gaussian process(GP) [56] is one method to model the dynamic probabilistically. *PILCO* [17]



(a) Deterministic LSTM (b) Bayesian LSTM

Figure 2.3: Shown above are training and testing performance on a noisy version Cart-pole balancing task at convergence. The trajectories are change of pole angle through time and x axis represents time steps. (a) Deterministic LSTM can learn perfectly from training data (**left**) but generalise badly on unseen trajectory (**right**). (b) A bayesian LSTM with probabilistic interpretation place a distribution over its prediction which offers robustness. The shaded area represent ± 1 std.

applies GP to model the environment dynamic and incorporates state uncertainty in planning as well. The dynamic is stated as:

$$\boldsymbol{x}_t = f(\boldsymbol{x}_{t-1}, \boldsymbol{u}_{t-1}) \quad , \quad \boldsymbol{x} \in \mathbb{R}^D, \boldsymbol{u} \in \mathbb{R}^F$$

Given input and target pairs $\mathcal{D} = \Big((\mathcal{X}, \mathcal{A}), \mathcal{Y}\Big)$, the difference between consecutive states $\Delta_t = \boldsymbol{x}_t - \boldsymbol{x}_{t-1}$ is modelled as a Gaussian process and we have the predictive distribution:

$$p(\Delta^{'}|\mathcal{D}, \boldsymbol{x}^{'}) = \mathcal{N}\Big(m_f(\boldsymbol{x}^{'}), \Sigma(\boldsymbol{x}^{'})\Big)$$

for some testing state $\boldsymbol{x}^{'}$, mean function $m_f(\cdot)$ and covariance function $\Sigma(\cdot)$. From which one can compute the marginal distribution $p(\Delta_t)$:

$$p(\Delta_t) = \int p(\Delta_t|\boldsymbol{x}_t)p(\mathbf{x}_t)\ d\mathbf{x}_t$$

This is not usually tractable and PILCO approximate it with moment matching. The computation procedure given the current state distribution $p(\boldsymbol{x}_{t-1})$ is:

$$p(\boldsymbol{x}_{t-1}) \underset{\pi}{\rightarrow} p(\boldsymbol{u}_{t-1}) \longrightarrow p(\boldsymbol{x}_{t-1}, \boldsymbol{u}_{t-1}) \longrightarrow p(\Delta_t) \longrightarrow p(\boldsymbol{x}_t)$$

Knowing the marginal distribution of states $p(\boldsymbol{x})$, the expected return $J^\pi = \sum_{t=1}^{T} \mathbb{E}_{\boldsymbol{x}_t}\big[c(\boldsymbol{x}_t)\big]$ can be computed as:

$$\mathbb{E}_{\boldsymbol{x}_t}\big[c(\boldsymbol{x}_t)\big] = \int c(\boldsymbol{x}_t)p(\boldsymbol{x}_t|\pi)\ d\boldsymbol{x}_t \quad , \quad p(\boldsymbol{x}_t|\pi) = \mathcal{N}(\boldsymbol{x}_t|\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t, \pi)$$

therefore PILCO considers the state uncertainty during planning.

Combining latent variables models and recurrent neural networks is one way to incorporate uncertainty in deterministic recurrent neural network and has been widely-investigated. The main

difference among these works is how the latent variable interacts with deterministic recurrent layer. *STORN* [6] (as shown in Fig 2.4(a)) introduces an independent latent variable, denoted as $\boldsymbol{z}_t$, for each time step. The next recurrent hidden $\mathbf{h}_t$ takes previous hidden state $\mathbf{h}_t$, action a_{t-1}, observations $\mathbf{x}_{t-1}$ and a sample from posterior $q(\mathbf{z}_t|\mathbf{x}_{\leq t-1})$ as input. An extra forward recurrent layer $\mathbf{h}^f$ is used to encode historical observation sequentially such that at time step $t-1$, the distribution of latent variable $q(\mathbf{z}_t|\mathbf{h}^f_{t-1})$ conditioned on hidden state depicts the variational posterior distribution $q(\mathbf{z}_t|\mathbf{x}_{\leq t-1})$ (Markov property). The model is trained by maximising the ELBO loss with reparametrisation trick:

$$q(\mathbf{z}_t|\mathbf{h}^f_{t-1}) := \boldsymbol{\mu}_{z,t} + \boldsymbol{\Sigma}_{z,t} \cdot \boldsymbol{\epsilon}\ , \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, I)$$

VRNN [15] (variational RNN) argues that adding past information to the prior on latent variable $p(\mathbf{z})$ can help improving representational ability of the model. As shown in Fig 2.4(b) at time step $t-1$, the prior distribution on $\mathbf{z}_{t-1}$ has a form of:

$$\mathbf{z}^{prior}_{t-1} \sim \mathcal{N}(\boldsymbol{\mu}^{prior}_{z,t-1}, \boldsymbol{\Sigma}^{prior}_{z,t-1})\ , \quad [\boldsymbol{\mu}^{prior}_{z,t-1}, \boldsymbol{\Sigma}^{prior}_{z,t-1}] = \boldsymbol{\psi}^{prior}(\mathbf{h}_{t-1})$$

where $\boldsymbol{\psi}(\cdot)$ is a neural network. And the inference model on $\mathbf{z}_{t-1}$ is a variational Gaussian distribution with sufficient statistics parametrised by current observations $\mathbf{x}_{t-1}$ and deterministic hidden state $\mathbf{h}_{t-1}$, denoted as:

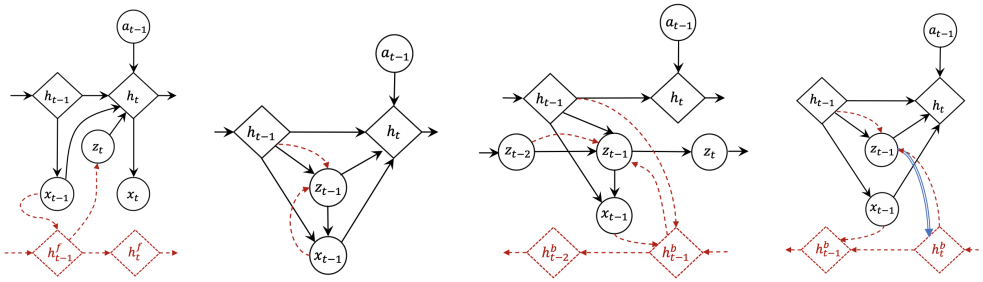
$$\mathbf{z}^{post}_{t-1} \sim q(\mathbf{z}_{t-1}|\mathbf{h}_{t-1}, \mathbf{x}_{t-1}) = \mathcal{N}(\boldsymbol{\mu}^{post}_{z,t-1}, \boldsymbol{\Sigma}^{post}_{z,t-1})\ , \quad [\boldsymbol{\mu}^{post}_{z,t-1}, \boldsymbol{\Sigma}^{post}_{z,t-1}] = \boldsymbol{\psi}^{post}(\boldsymbol{x}_{t-1}, \mathbf{h}_{t-1})$$

which is an approximation of the true filtering posterior distribution $p(\mathbf{z}_{t-1}|\mathbf{x}_{\leq t-1}, \mathbf{z}_{\leq t-2})$. Compared to STORN, the recurrent transition function from $\mathbf{h}_{t-1}$ to $\mathbf{h}_t$ additionally takes observation $\mathbf{x}_{t-1}$ into consideration. *SRNN* [25] further improves the model by separating the deterministic and probabilistic states and taking future context into consideration. This can be seen as an integration of a *state space model* (e.g. *Hidden Markov model* (HMM) [70], *Linear Gaussian SSM* [45]) and a recurrent neural network (as shown in Fig 2.4(c)), where SSM provides stochasticity and RNN models long-term dependencies. The prior distribution on latent variable $p(\mathbf{z}_{t-1})$ is parametrised by $\mathbf{h}_{t-1}$ and also previous latent variable $\mathbf{z}_{t-2}$. In the inference model at time $t-1$, VRNN uses a backward recurrent layer $\boldsymbol{f}^{back}$ encoding sequentially the future information (hidden states $\mathbf{h}_{\geq t-1}$ and observations $\mathbf{x}_{\geq t-1}$) into a hidden representation $\mathbf{h}^b_{t-1}$, which together with $\mathbf{z}_{t-2}$ constitutes the condition set for variational posterior distribution $q(\mathbf{z}_{t-1}|\mathbf{z}_{t-2}, \mathbf{h}_{\geq t-1}, \mathbf{x}_{\geq t-1})$, namely:

$$q(\mathbf{z}_{t-1}|\mathbf{z}_{t-2}, \mathbf{h}_{\geq t-1}, \mathbf{x}_{\geq t-1}) := q(\mathbf{z}_{t-1}|\mathbf{z}_{t-2}, \mathbf{h}^b_{t-1})$$

$$\mathbf{h}^b_{t-1} = \boldsymbol{f}^{back}(\mathbf{h}^b_t, \mathbf{x}_{t-1}, \mathbf{h}_{t-1})$$

Compared to VRNN, the separation of probabilistic and deterministic states improve posterior approximation and speed up the training procedure. *Z-forcing* [31] applies the idea of backward recurrent layer encoder on VRNN model but disconnects latent variable $\mathbf{z}_{t-1}$ from prediction $\mathbf{x}_{t-1}$, instead it bridges the gap by adding an auxiliary generating model $p(\mathbf{h}^b_t|\mathbf{z}_{t-1})$ (blue double line in the plot). It is later used in model-based planning which gives good long-term prediction [42].



(a) STORN (b) VRNN (c) SRNN (d) Z-forcing

Figure 2.4: Computational graph of recurrent latent variable model: (a): STORN [6]. (b): VRNN [15]. (c): SRNN [25]. (d): Z-forcing [31]. Diamond and circle stand for deterministic and probabilistic states respectively. Red dashed line represents the inference model. Blue double line represents the auxiliary prediction proposed by the author. $\mathbf{z}_t$ is the latent variable; $\mathbf{x}_t, a_t$ are observation and action; $\mathbf{h}_t$ is the recurrent hidden state for generating model; $\mathbf{h}_t^f$ is the forward recurrent layer for inference and $\mathbf{h}_t^b$ is the backward recurrent layer.

Recently, *PlaNet* [36] implements a recurrent state space model (RSSM) on image data which is akin to VRNN but with observation excluded from the transition. The computational graph is shown in Fig 2.5. During model training, PlaNet uses a modified KL divergence which achieves better training on multi-step predictions. Recently with the same model, *Dreamer* [35] designs *latent planning* achieving state-of-the-art planning performance. In the experiment we compare our LLB model with RSSM, the difference between these two is that LLB models epistemic uncertainty while RSSM does not. The detailed model inference and training information can be found in Chapter 5.

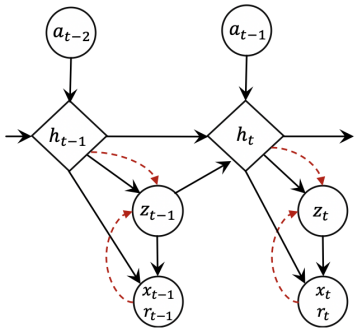


Figure 2.5: Computational graph of recurrent state-space model (RSSM) in PlaNet. Different from VRNN, the model also predicts values of rewards at each time step and the observations are no longer fed into the transition layer.

Probabilistic ensemble neural network is another class of uncertainty-aware models that is frequently-

used in MBRL. A deterministic ensemble model is a set of deterministic models with different initialisation trained on the same data (*randomisation type*) or trained on different subsets of data in the case of bootstrapping [51]. And a prediction is made by averaging over multiple models, which capture the epistemic (or model) uncertainty. Furthermore, using probabilistic model instead (such as a soft-max NN) which outputs parametrised distribution in ensemble learning can also capture aleatoric (or data) uncertainty [47]. This is often named probabilistic ensemble models. Different to prior latent variable neural network, a probabilistic ensemble NN can effectively isolate the model and data uncertainty [14]. A typical application in MBRL is *PETS* (probabilistic ensemble trajectory sampling) [14, 72, 41], which has achieved similar asymptotic performances as model-free method, while being more data-efficient and faster in convergence.

There has also been an increasing interests in applying Bayesian neural network in MBRL. A BNN places a distribution on the weight parameters and this can be seen as an ensemble of networks where their weights are sampled from a shared and learnable distribution except that BNN is comparatively easier to train [11]. In a contextual bandits setting, weight uncertainty in BNN together with *Thompson sampling* can drive efficient exploration [11], in which the model is trained with Bayes by Backprop. *Deep PILCO* [29] improves PILCO by replacing Gaussian process with BNN, making it scalable to high dimensional space.

Limited amount of works apply Bayesian recurrent neural network in MBRL [20]. Under a Safe RL scenario, model uncertainty estimate of a fully Bayesian recurrent neural network leads to safer behaviour [8]. However, in his work the BRNN is not trained in a rigorously bayesian way. In this report, we apply variational method to train a partially-Bayesian RNN and apply it on classic control task. The goal is to investigate how the uncertainty estimate affect both model learning and planning performance.

2.4.3 Model-based planning

This section describe how planning is generally performed after a probabilistic model is trained.

Planning is originally defined as the computational process that accepts a model of the environment as input and creates or updates a policy so that the agent behaves optimally in the real environment. [68]. In the report, we refer to learning the policy with models as model-based planning and without models as model-free planning. (As shown in Fig 2.6). A model-free agent constantly interacts with real environment and learns policies from sampled trajectories. Usually learning a policy is easier than learning a dynamic model of the environment therefore a model-free planning is more frequently used than model-based method [19]. A model-based agent normally experiment on simulated models. It requires less real-time interaction and with a learnt dynamic model the agent can explore unseen states easier. This is essential in the scenario when experiments in real environment is difficult or expensive to carry out. In this report the main focus is on model-based planning.

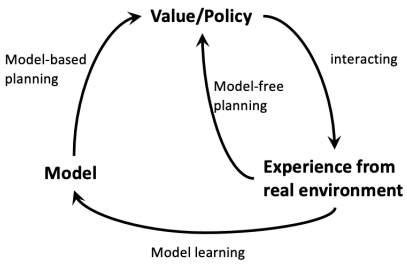


Figure 2.6: Flow chart of RL. In this report, learning the policy directly from experience without the help of additional models is referred to as model-free planning and model-based planning is learning the policy from a simulated models.

One way to categorise model-based planning methods is to determine where a parametric policy function is used. An example of planning without a parametric function is *MPC* (model predictive control) [57]. In a discrete action case, given with a reward function or reward models $R(observation)$, a MPC agent first estimates the reward returned from each possible next action: (assume at time t)

$$R(\mathbf{o}_{t+1}|a_t)\ ,\quad o_t \sim P(\mathbf{o}_{t+1}|a_t, \mathbf{o}_t)\ ,\quad a_t \in \mathcal{A}$$

or cumulative rewards of action sequences:

$$R(\mathbf{o}_{t+1}, \mathbf{o}_{t+2}, \cdots, \mathbf{o}_{t+k}|a_t, a_{t+1}, \cdots, a_{t+k-1})$$

$$\mathbf{o}_i \sim p(\mathbf{o}_{i+1}|\mathbf{o}_i, a_i)\ ,\quad a_i \in \mathcal{A}\ ,\quad i = t : t+k-1$$

then the best next action a_t^* is selected as:

$$a_t^* = \arg\max_{a_t} R(\mathbf{o}_{t+1}|a_t) \quad \text{(one-step MPC)}$$

or

$$a_t^* = \arg\max_{a_t} R(\mathbf{o}_{t+1}, \cdots, \mathbf{o}_{t+k}|a_t, \cdots, a_{t+k-1}) \quad \text{(multi-step MPC)}$$

where multi-step MPC is able to consider long-term rewards and act in a far-sighted way. Another example is *CEM* (cross entropy method) [59] which is in a similar vein to MPC method. CEM is normally used in the continuous action case and occurs in various state-of-the-art RL frameworks for its robustness [36, 14, 72]. The algorithm can be described roughly as:

1. Sample N sequences of action from an action sequence distribution: $a_{t:t+k-1}^{(i)} \sim p_0(a_{t:t+k-1})\ ,\quad i = 1:N$
2. Compute the resulting trajectory for each sampled action sequences from the model: $\mathbf{o}_{t+1:t+k}^{(i)} \sim p(\mathbf{o}_{t+1:t+k}^{(i)}|\mathbf{o}_t, a_{t:t+k-1}^{(i)})$
3. Compute the cumulative rewards for each trajectory $R^{(i)} = \sum_{h=t+1}^{t+k} R(\mathbf{o}_h^{(i)})$
4. Pick M best action sequences $\{a^{*(1)}_{t:t+k-1}, \cdots, a^{*(M)}_{t:t+k-1}\}$ with highest rewards

5. Update the sufficient statistics of action distribution (assumed Gaussian) by:

$$\boldsymbol{\mu}_{t:t+k-1} = \frac{1}{M}\sum_{m=1}^{M} a^{*(m)}_{t:t+k-1} \quad , \quad \boldsymbol{\sigma}^2_{t:t+k-1} = \frac{1}{M}\sum_{m=1}^{M} \|a^{*(m)}_{t:t+k-1} - \boldsymbol{\mu}_{t:t+k-1}\|^2$$

6. Go back to step 1 and repeat.

One can also train a parametric policy function in a supervised learning manner after collecting a set of valuable state-action pairs. This is known as *behaviour cloning* [55].

One disadvantage of planning without a parametric policy is its relatively slow convergence. Using a parametric policy function is expected to speed it up with gradient update [19]. This is also known as *Policy Gradient* method. The objective is to maximise the cumulative expected reward:

$$J_\pi(\phi) = \sum_{t=0}^{T} \mathbb{E}_{\mathbf{o}_t}\big[r(\mathbf{o}_t)\big]$$

with respect to policy parameter $\boldsymbol{\phi}$. One can either update analytically or using approximate methods. Analytic policy gradient requires both differentiable reward function $r(\cdot)$ and model transition. PILCO approximates the intractable state distribution using moment matching such that the expectation term:

$$\mathbb{E}_{\mathbf{o}_t}\big[c(\mathbf{o}_t)\big] = \int c(\mathbf{o}_t) p(\mathbf{o}_t|\boldsymbol{\pi}_\phi)\, d\mathbf{o}_t$$

is tractable and one can take the gradient $\frac{\partial J_\pi(\phi)}{\partial \phi}$ directly.

However, it is computationally difficult to apply moment matching in neural network models [52]. Instead, one could use particle sampling method to get a Monte-Carlo estimate of the expected rewards. Reparametrisation trick (RP) is often implemented to back-propagate through sampling procedure [20] and one can also apply likelihood ratio (LR) method also known as REINFORCE trick when the reward/cost function is non-differentiable. Deep PILCO [29] use particle method along with *Gaussian re-sampling* to estimate marginal state distribution from $p(\mathbf{o}_0)$ to $p(\mathbf{o}_t)$. *PIPPS* proposes a way to combine LR and RP tricks to solve the inaccurate RP problem. In our work, we mainly focus on applying RP and passing the gradient through a known cost function $c(\cdot)$.

Chapter 4 gives a more explicit description on policy gradient method in a probabilistic dynamic model. In next chapter, we start introducing our proposed model and the way to train it. Following that is the analysis on experimental results.

Chapter 3

Last layer Bayes recurrent neural network

In previous chapter, we gave a brief introduction on Bayesian neural network and other probabilistic dynamic models. We talked about their application of uncertainty estimate in model-base reinforcement learning. In this chapter, we will propose a partially-Bayesian recurrent neural network which place distribution only on the weight parameter of the last layer in a recurrent cell. We call it *Last-layer Bayesian* (LLB) RNN. This is a simplified version of a fully-Bayesian RNN and is comparatively easier to implement and scalable to different model structure.

The chapter starts by describing in detail the network setup and also the variational objective function for conducting model fitting. Following this is the model training experiment done on classic control data (cart-pole balancing) in which we explore how different model structure affect the learning performance.

3.1 Network structure

Fig 3.1 depicts the network structure of a Last-layer Bayesian RNN. Similar to a standard RNN, a LLB-RNN has recurrent hidden state $\mathbf{h}_t$ which propagates past information forward. At each time step t, it also takes input (such as action a_t in control task) and outputs decoded state $\mathbf{x}_t$. What distinguishes it from a normal recurrent neural network is the distribution on recurrent layer parameter $p(\mathbf{W}_{LLB})$ and the decoded state $p(\mathbf{x}_t|\mathbf{h}_t)$. So that with uncertain weight value $\mathbf{W}_{LLB}$, the recurrent hidden states $[\mathbf{h}_1, \cdots, \mathbf{h}_T]$ are varying even with same input data. In reality, this could stand for the systematic noise that exists in a physical model. In the face of uncertainty estimate, the distribution on weight parameters model epistemic uncertainty. On the other hand, the distribution on generated state $p(\mathbf{x}_t|\mathbf{h}_t)$ conditioned on hidden state $\mathbf{h}_t$ models aleatoric uncertainty. In reality this can be referred to as measurement noise.

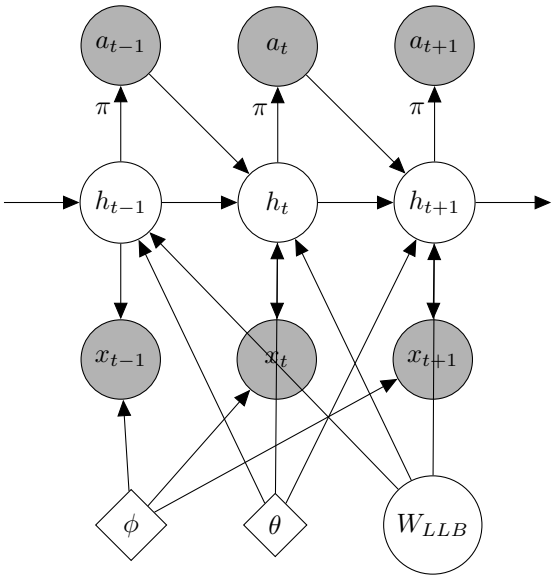


Figure 3.1: Schematic plot of Last layer Bayes Recurrent neural network. $\mathbf{h}_t$: hidden states of recurrent unit; $\mathbf{a}_t$: action generated from policy π; $\mathbf{x}_t$: observation state; θ: parameter of the recurrent layer; ϕ: parameter of the decoder layer; $\mathbf{W}_{LLB}$: a probabilistic weight parameter applied at the last layer of recurrent unit.*

3.1.1 Distribution on last-layer weight $\mathbf{W}_{LLB}$

The distribution on weight parameter in our model is assumed to be Gaussian. Let $\mathbf{W}_{LLB}$ denotes the last-layer weight, the distribution of it is denoted as:

$$p(\mathbf{W}_{LLB}) = \mathcal{N}(\boldsymbol{\mu}_W, \boldsymbol{\Sigma}_W) \ , \ \ \mathbf{W}_{LLB}, \boldsymbol{\mu}_W \in \mathbb{R}^m \ , \ \ \boldsymbol{\Sigma}_W \in \mathbb{R}^{m \times m}$$

or as a *matrix normal distribution* [4] if $\mathbf{W}_{LLB}$ is two-dimensional. However, in our experiment we assume a diagonal covariance matrix so that each entry w_{ij} of the weight matrix follows a uni-variate Gaussian distribution as:

$$w_{ij} \sim \mathcal{N}(\mu_{ij}, \sigma_{ij}^2)$$

and the distribution on $\mathbf{W}_{LLB}$ is re-written as:

$$p(\mathbf{W}_{LLB}) = \prod_{i=1}^{m} \prod_{j=1}^{n} \mathcal{N}\big(w_{ij}; \mu_{ij}, \sigma_{ij}^2\big) \ , \ \ \mathbf{W}_{LLB} \in \mathcal{R}^{m \times n}$$

where w_{ij} is a trainable parameter in the model learning experiment.

One thing worth noticing is that the value of last-layer weight parameter is kept fixed in a single roll-out, as suggested by the model graph Fig 3.1. This ensure the long-term dependencies on each data sequence. Next we review the mathematical formula for different recurrent cell: RNN, LSTM ,GRU and show how last-layer distribution applies respectively.

*Note that in the model graph the policy function π can either be conditioned on hidden state $\mathbf{h}_t$ or observations $\mathbf{x}_t$. The former setting needs to learn an encoder for each time step and is known as *latent planning* [36, 35]. In our experiment we condition on observations.

Mathematical formulation of RNN, LSTM and GRU with last-layer Bayesian concept are listed below as:

- RNN:

$$\mathbf{h}_t = \mathbf{W}_{LLB} \tanh(\mathbf{W}_{ih} a_{t-1} + \mathbf{W}_{hh} \mathbf{h}_{t-1} + \mathbf{b})$$

- LSTM:

$$\begin{aligned}
i_t &= \sigma(a_{t-1}\mathbf{U}^i + \mathbf{h}_{t-1}\mathbf{W}^i + \mathbf{b}^i) \\
f_t &= \sigma(a_{t-1}\mathbf{U}^f + \mathbf{h}_{t-1}\mathbf{W}^f + \mathbf{b}^f) \\
o_t &= \sigma(a_{t-1}\mathbf{U}^o + \mathbf{h}_{t-1}\mathbf{W}^o + \mathbf{b}^o) \\
\tilde{C}_t &= \tanh(a_t\mathbf{U}^g + \mathbf{h}_{t-1}\mathbf{W}^g) \\
C_t &= f_t * C_{t-1} + i_t * \tilde{C}_t \\
\mathbf{h}_t &= \mathbf{W}_{LLB} \left(\tanh(C_t) * o_t \right)
\end{aligned}$$

- GRU*:

$$\begin{aligned}
z_t &= \sigma(a_{t-1}\mathbf{W}_z + \mathbf{h}_{t-1}\mathbf{U}^z + \mathbf{b}^z) \\
r_t &= \sigma(a_{t-1}\mathbf{W}^r + \mathbf{h}_{t-1}\mathbf{U}^r + \mathbf{b}^r) \\
\tilde{h}_t &= \tanh\Big(a_{t-1}\mathbf{W}^h + r_t * (\mathbf{h}_{t-1}\mathbf{U}^h) + \mathbf{b}^h \Big) \\
\mathbf{h}_t &= \mathbf{W}_{LLB} \left(z_t * \mathbf{h}_{t-1} + (1 - z_t) * \tilde{h}_t \right)
\end{aligned}$$

where all parameters except $\mathbf{W}_{LLB}$ are deterministic. So all the operations happen inside a recurrent cell are same as usual and the output is multiplied by a probabilistic weight $\mathbf{W}_{LLB}$. The result coming out is fed into the next recurrent step and so on. This makes LLB flexible and easy to implement on other deep learning models.

3.1.2 Distribution on generated states $\mathbf{x}_t$

The predicted states $\mathbf{x}_t$ also follows a (multi-variate) Gaussian distribution, denoted as:

$$p(\mathbf{x}_t|\mathbf{h}_t) = \mathcal{N}(\mathbf{x}_t; \boldsymbol{\mu}_{x,t}, \boldsymbol{\Sigma}_{x,t}) \ , \ \ \mathbf{x}_t, \boldsymbol{\mu}_{x,t} \in \mathbb{R}^K, \boldsymbol{\Sigma}_{x,t} \in \mathbb{R}^{K \times K}$$

the conditioning on hidden state implies a state-dependent data noise as opposed to a fixed Gaussian perturbed noise: $\mathbf{x}_t = g(\mathbf{h}_t) + \boldsymbol{\epsilon} \ , \ \ \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma})$. This is essential as it let the model to learn the noise from the data and adjust it based on the state it is currently in. By doing so, the model gains more flexibility and richer representation.

It is also interesting to see that a LLB RNN is able to separate model and data uncertainty. For example, once a model has been trained, one can either choose to generate prediction using same fixed weight parameter for each trial or vary the value of weight and using mean prediction on $\mathbf{x}_t$. This enable the analysis of different uncertainty which will be seen in our planning experiment. Also, it was claim that the isolation of epistemic uncertainty is beneficial for *direct exploration* [14] which can be an appealing research area. Meanwhile, as we will see in the next section where we apply variational method to approximate the posterior distribution of $\mathbf{W}_{LLB}$, such Bayesian treatment on the model parameter can also be leveraged for curiosity-driven exploration [39] which is another broad research area worth exploring. For more potential future work see Chapter 6.

*In our experiment we add a *tanh* layer to $\mathbf{h}_t$ to avoid numerical error for long trajectory learning.

3.2 Model learning objective function

Now we apply variational inference to train a LLB-RNN model. The core is the approximation of posterior distribution on last-layer weight parameter $q(\mathbf{W}_{LLB})$.

3.2.1 Free energy function

Given training data pairs $(\mathbf{x}_{1:T}, a_{1:T-1})$ and initial hidden state $\mathbf{h}_1$, we denote the recurrent transition function as $\mathbf{f}_\theta$ with parameter θ, and the data generating probability as $p_\phi(\mathbf{x}_t|\mathbf{h}_t)$ with generating function parameter ϕ. The model transition can be written as:

$$\mathbf{h}_t = \mathbf{W}_{LLB}\mathbf{f}_\theta(\mathbf{h}_{t-1}, a_{t-1}) \tag{3.1}$$

Then the marginal likelihood function of data sequence $\mathbf{x}_{1:T}$ can be written as (according to Markov property):

$$\begin{aligned} p(\mathbf{x}_{1:T}|a_{1:T-1}) &= \int \prod_{t=1}^{T} p_\phi(\mathbf{x}_t|\mathbf{h}_t)\; p_\theta(\mathbf{h}_t|\mathbf{h}_{t-1}, a_{t-1})\; d\mathbf{h}_{1:T} \\ &= \iint p(\mathbf{W}_{LLB}) \prod_{t=1}^{T} p_\phi(\mathbf{x}_t|\mathbf{h}_t)\; p_\theta(\mathbf{h}_t|\mathbf{h}_{t-1}, a_{t-1}, \mathbf{W}_{LLB})\; d\mathbf{h}_{1:T}\; d\mathbf{W}_{LLB} \\ &= \int p(\mathbf{W}_{LLB}) \prod_{t=1}^{T} p_\phi(\mathbf{x}_t|\mathbf{h}_t)\Big|_{h_t = \mathbf{W}_{LLB} f_\theta(\mathbf{h}_{t-1}, a_{t-1})} d\mathbf{W}_{LLB} \quad \text{(Dirac delta function)} \end{aligned} \tag{3.2}$$

where $p(\mathbf{W}_{LLB})$ is a prior distribution on $\mathbf{W}_{LLB}$.

It is noticeable that once the value of $\mathbf{W}_{LLB}$ is sampled and fixed, the hidden states are computable through:

$$\mathbf{h}_2 = \mathbf{W}_{LLB} f_\theta(\mathbf{h}_1, a_1)\;,\quad \mathbf{h}_3 = \mathbf{W}_{LLB} f_\theta(\mathbf{h}_2, a_2)\;, \cdots, \mathbf{h}_T = \mathbf{W}_{LLB} f_\theta(\mathbf{h}_{t-1}, a_{t-1})$$

which implies that the generating probability only depends on $\mathbf{W}_{LLB}, a_{1:T-1}$ and θ:

$$p_\phi(\mathbf{x}_t|\mathbf{h}_t) = p_\phi(\mathbf{x}_t|\mathbf{W}_{LLB}, \mathbf{f}_\theta, a_{1:T-1}) \tag{3.3}$$

Next, with e.q 3.2 and *Jensen's inequality* we can derive our objective function for model learning:

$$\begin{aligned} \log p(\mathbf{x}_{1:T}|a_{1:T-1}) &= \log \mathbb{E}_{q(\mathbf{W}_{LLB})}\left[\frac{p(\mathbf{W}_{LLB})}{q(\mathbf{W}_{LLB})} \prod_{t=1}^{T} p_\phi(\mathbf{x}_t|\mathbf{W}_{LLB}, \mathbf{f}_\theta, a_{1:T-1})\right] \\ &\geq \mathbb{E}_{q(\mathbf{W}_{LLB})}\left[\log\frac{p(\mathbf{W}_{LLB})}{q(\mathbf{W}_{LLB})}\right] + \mathbb{E}_{q(\mathbf{W}_{LLB})}\left[\sum_{t=1}^{T} \log\, p_\phi(\mathbf{x}_t|\mathbf{W}_{LLB}, \mathbf{f}_\theta, a_{1:T-1})\right] \\ &= -\mathbf{KL}\Big[q(\mathbf{W}_{LLB})\Big\|p(\mathbf{W}_{LLB})\Big] + \left\langle \sum_{t=1}^{T} \log\, p_\phi(\mathbf{x}_t|\mathbf{W}_{LLB}, \mathbf{f}_\theta, a_{1:T-1})\right\rangle_{q(\mathbf{W}_{LLB})} \\ &:= \mathcal{F}(\theta, \phi, \mathbf{W}_{LLB}) \end{aligned} \tag{3.4}$$

where variational distribution $q(\mathbf{W}_{LLB})$, according to Sec.2.3.2, is an approximation to the true posterior distribution $p(\mathbf{W}_{LLB}|\mathbf{x}_{1:T}, a_{1:T-1})$, the second term is a sum of conditional log-likelihood function under the expectation of $q(\mathbf{W}_{LLB})$ and the KL term acts as a regulariser preventing $q(\mathbf{W}_{LLB})$ being too far away from the prior. The KL term is also named *complexity cost* [11] indicating the complexity of encoded $\mathbf{W}_{LLB}$.

3.2.2 Monte-Carlo approximation

The objective of model learning is to maximise the free energy function $\mathcal{F}$ (eq 3.4) with respect to model parameter $\{\theta, \phi\}$ and also variational distribution $q(\mathbf{W}_{LLB})$, that is to maximising the expected sum of conditional log-likelihood function while regularising the model complexity. However, as we have discussed in Sec 2.3.2, the derivative of expectation term in $\mathcal{F}$ is intractable and therefore we apply Monte-Carlo sampling method along with reparametrisation trick. For example, let $[\boldsymbol{\mu}_W, \log \boldsymbol{\sigma}_W]$ denote the mean and log standard deviation of distribution $q(\mathbf{W}_{LLB})$ (assume diagonal covariance), a sample from the posterior can be represented as:

$$\mathbf{W}_{LLB} = \boldsymbol{\mu}_W + \boldsymbol{\sigma}_W \odot \boldsymbol{\epsilon}\ , \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$$

where $\odot$ stands for element-wise multiplication.

By sampling N number of weight values $\mathbf{W}_{LLB}^{(i)}\ , i = 1, \cdots, N$, one can approximate the expectation term as:

$$\frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T} \log\, p_\phi(\mathbf{x}_t|\mathbf{W}_{LLB}^{(i)}, \mathbf{f}_\theta, a_{1:T-1})$$

the gradient of which can be back-propagate using SGD to both the model parameters θ, ϕ and $[\boldsymbol{\mu}_W, \boldsymbol{\sigma}_W]$ as well. This is also known as Bayes by Back-prop. An example procedure is provided in Algorithm 1.

Now that we have a theoretical foundation of how to train a LLB-RNN model using variational method. Next section discuss how in practice we setup the model using *Pytorch* [53].

3.3 Model learning experiment setting

We describe how we setup the network architecture explicitly and how the training procedure in previous section is carried out practically. Also we introduce how we implement particle sampling method to obtain model prediction.

3.3.1 LLB model layers

The model contains four components: an initial encoder, a recurrent layer, a last layer weight network and an emission network.(Illustration can be found in Fig 3.2)

- **Initial state encoder**: a two-layer perceptron (MLP) with ReLU activation function, converting the starting observations $\mathbf{o}_0$* from the environment into the initial hidden state $\mathbf{h}_0$ of the **recurrent layer**.

*from now on we denote the states/observations as symbol $\mathbf{o}$

Algorithm 1: MC sampling with reparametrisation trick and SGD (Bayes by Backprop)

1 **Given** learnable variational mean $\boldsymbol{\mu}_W \in \mathcal{R}^{m\times n}$ and standard deviation $\log \boldsymbol{\sigma}_W \in \mathcal{R}^{m\times n}$

2 **Sample** $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I}), \boldsymbol{\epsilon} \in \mathcal{R}^{m\times n}$ and we obtain a weight sample as

$$\mathbf{W}^{'}_{LLB} = \boldsymbol{\mu}_W + exp(\log \boldsymbol{\sigma}_W) \circ \boldsymbol{\epsilon}$$

3 Carry out forward propagation and compute the objective function $\mathcal{F}(\theta, \phi, \mathbf{W}^{'}_{LLB})$ (eq. 2.5) with sample $\mathbf{W}^{'}_{LLB}$ and a single or a mini-batch of training data $(\mathbf{x}_{1:T}, a_{1:T-1})$

4 Compute the derivative with respect to variational mean $(\boldsymbol{\mu}_W)$ and log std $(\log \boldsymbol{\sigma}_W)$:

$$\frac{\partial \mathcal{F}(\mathbf{W}^{'}_{LLB})}{\partial \boldsymbol{\mu}_W} = \frac{\partial \mathcal{F}}{\partial \mathbf{W}^{'}_{LLB}} \frac{\partial \mathbf{W}^{'}_{LLB}}{\partial \boldsymbol{\mu}_W} + \frac{\partial \mathcal{F}}{\partial \boldsymbol{\mu}_W} = \frac{\partial \mathcal{F}}{\partial \mathbf{W}^{'}_{LLB}} + \frac{\partial \mathcal{F}}{\partial \boldsymbol{\mu}_W}$$

$$\frac{\partial \mathcal{F}(\mathbf{W}^{'}_{LLB})}{\partial \log \boldsymbol{\sigma}_W} = \frac{\partial \mathcal{F}}{\partial \mathbf{W}^{'}_{LLB}} \cdot exp(\log \boldsymbol{\sigma}_W) \circ \boldsymbol{\epsilon} + \frac{\partial \mathcal{F}}{\partial \log \boldsymbol{\sigma}_W}$$

5 Update the variational parameter:

$$\boldsymbol{\mu}_W = \boldsymbol{\mu}_W - \alpha \frac{\partial \mathcal{F}}{\partial \boldsymbol{\mu}_W}$$

$$\log \boldsymbol{\sigma}_W = \log \boldsymbol{\sigma}_W - \alpha \frac{\partial \mathcal{F}}{\partial \log \boldsymbol{\sigma}_W}$$

- **Recurrent layer**: this a standard recurrent cell such as RNN, LSTM or GRU. In the experiment we compare the model performance with different recurrent layers. Detailed mathematical formula can be found in Sec 3.1.

- **Last layer weight Bayesian layer**: this contains two learnable weight matrix for both mean $\mathbf{W}^{\mu}_{LLB}$ and diagonal standard deviation $\mathbf{W}^{\Sigma}_{LLB}$. And the value of $\mathbf{W}_{LLB}$ for doing model roll-outs is determined though reparametrisation:

$$\mathbf{W}^{(i)}_{LLB} = \mathbf{W}^{\mu}_{LLB} + \mathbf{W}^{\Sigma}_{LLB} \odot \boldsymbol{\epsilon}^{(i)} \ , \ \boldsymbol{\epsilon}^{(i)} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$$

  and kept fixed throughout the whole trajectory. In Pytorch, the setup of the weight parameter is written as: `nn.Parameter()`.

- **Emission/generating network**: this is the network taking recurrent hidden state $\mathbf{h}_t$ as input at each time step and generates predicted observation $\hat{\mathbf{o}}_t$. It starts with a single-layer MLP and followed by a mean and a log std layer, denoted as $\boldsymbol{\mu}_{x,t}$, $\log \boldsymbol{\sigma}_{x,t}$ respectively in Fig.3.2. In model training both the mean and std are required to calculate likelihood function and in model prediction phase, sampling by reparametrisation is applied and the output prediction is denoted as:

$$\hat{\mathbf{o}}_t = \boldsymbol{\mu}_{x,t} + \exp\big(\log \boldsymbol{\sigma}_{x,t}\big) \odot \boldsymbol{\epsilon} \ , \ \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$$

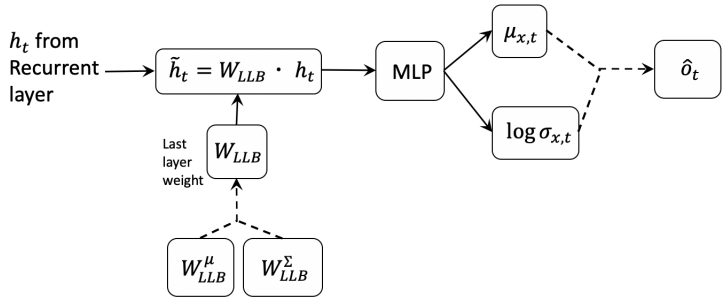


Figure 3.2: LLB model layers flow chart. The **dashed** line represents sampling using reparametrisation trick. Note that the value of sampled $\mathbf{W}_{LLB}$ is kept fixed for the entire trajectory.

3.3.2 Model learning and prediction

We perform model learning in a SGD manner. For a batch of input data sequence $(\mathbf{o}^{(b)}_{1:T}, a^{(b)}_{1:T-1})^B_{b=1}$, we first sample B number of last-layer weight value

$$\mathbf{W}^{(i)}_{LLB} \sim q(\mathbf{W}_{LLB}), i = 1, \cdots, B$$

then for each sample, forward propagation gives deterministic recurrent hidden states $\{\mathbf{h}^{(i)}_1, \cdots, \mathbf{h}^{(i)}_T\}$ through:

$$\mathbf{h}^{(i)}_1 = \mathbf{W}^{(i)}_{LLB}\mathbf{f}_\theta(\mathbf{h}^{(i)}_0, a_0)\ ,\ \ \mathbf{h}^{(i)}_2 = \mathbf{W}^{(i)}_{LLB}\mathbf{f}_\theta(\mathbf{h}^{(i)}_1, a_1)\ ,\ \cdots\ ,\ \mathbf{h}^{(i)}_T = \mathbf{W}^{(i)}_{LLB}\mathbf{f}_\theta(\mathbf{h}^{(i)}_{T-1}, a_{T-1})$$

then with access to all hidden state, we are able to compute the mean and log std of predicted observation at each time step through generating function $g_\theta(\cdot)$:

$$[\boldsymbol{\mu}^{(i)}_{x,t}\ ,\ \log\ \boldsymbol{\sigma}^{(i)}_{x,t}] = g_\theta(\mathbf{h}^{(i)}_t)$$

Now one can compute the objective function (eq 3.4) and back-propagate the gradient to update the model parameter θ and variational distribution parameters $[\mathbf{W}^{\mu}_{LLB}, \mathbf{W}^{\Sigma}_{LLB}]$.

Model prediction applies particle filtering [21], also known as *sequential Monte-Carlo* method [65]. It is similar to the Monte-Carlo sampling in model learning except that we are sampling from observation distribution as well. During prediction phase, given an initial states $\mathbf{o}_0$ and a sequence of actions $a_0, \cdots, a_{T-1}$, N number of particles(samples) are created from variational distribution of last layer weight:

$$\mathbf{W}^{(i)}_{LLB} \sim q(\mathbf{W}_{LLB})\ ,\ \ i = 1, \cdots, N$$

for each of these particles, we can compute the recurrent hidden states $\{\mathbf{h}_1, \cdots, \mathbf{h}_T\}$ and sample predicted observation by:

$$\hat{\mathbf{o}}^{(i)}_t \sim p(\mathbf{o}_t | g_\theta(\mathbf{h}^{(i)}_t))\ ,\ \ i = 1, \cdots, N$$

Therefore we obtain N sets of particles $\{\hat{\mathbf{o}}^{(i)}_{1:T}\}^N_{i=1}$ from which the predicted mean and variance at

each time step are calculated as:

$$\mathbb{E}[\hat{\mathbf{o}}_t] = \frac{1}{N}\sum_{i=1}^{N}\hat{\mathbf{o}}_t^{(i)} \quad , \quad \mathrm{Var}[\hat{\mathbf{o}}_t] = \frac{1}{N}\sum_{i=1}^{N}\left(\hat{\mathbf{o}}_t^{(i)} - \mathbb{E}[\hat{\mathbf{o}}_t]\right)^2$$

There are also other ways to carry out prediction using sampling. For example, one can create multiple samples with same value of $\mathbf{W}_{LLB}$ which might reduce the variance and provide stability. In our experiment, since the environment is simple and the prediction horizon is short, we found no obvious improvement for doing that. But it would be interesting to see a variance analysis of different sampling schemes on more complex tasks and we leave it for future work.

Next section gives a brief summarisation of model learning experiments. Particularly we look at how different model structure affect model learning.

3.4 Effects of model structure on training and uncertainty estimate

In this section we assess the training performance of a LLB model. We experiment with different recurrent unit and various number of hidden dimension under a deterministic cart-pole environment.

In particular, we look at three recurrent units: RNN, LSTM, GRU and three values of hidden dimension: $16, 32, 64$. Given the same available data, we will mainly focusing on comparing the accuracy of mean prediction and degree of uncertainty captured in their prediction (which is represented by the standard deviation in prediction).

3.4.1 Environment setup

We first take a look at two simulated control environments: deterministic and stochastic cart-pole.

In a standard cart-pole balancing task, a pole with length 1 is attached to a cart which can only move horizontally along a track. A constant force generated by an actuator is applied on the cart moving it towards right left or right. The objective is to control the actuator so that the pole can be held upright while not running off the track. The observation(state) at each time step t contains four quantities: position of the cart x_t; velocity of the cart $\dot{x}_t$; angle between pole and upright position θ_t and angle velocity $\dot{\theta}_t$. In computer simulation, we first reset the initial state $\mathbf{o}_0 = [x_0, \dot{x}_0, \theta_0, \dot{\theta}_0]$ randomly, then by inputting an action value a consecutively the environment return back a state trajectory $[\mathbf{o}_1, \cdots, \mathbf{o}_T]$. So the input-output data pairs can be denoted as:

$$\textbf{Input} : \mathbf{o}_0, a_0, a_1, \cdots, a_{T-1}$$
$$\textbf{Output} : \mathbf{o}_1, \mathbf{o}_2, \cdots, \mathbf{o}_T$$

Termination happens when the position of the cart is outside the range $[-2.4, 2.4]$ or the angle is outside the range $[-0.21 \text{ rad}, \ 0.21 \text{ rad}]$. The reward returned from the environment is the total

number of time step the system lasts. The episode also ends when the task is consider solved, which is when the cumulative reward reaches 200.

In a stochastic cart-pole environment. We add Gaussian noise to both the actuator and the sensor, meaning that the force applied on the cart is not accurate and so is the measurement of the observations. In a mathematical form this is denoted as:

$$\begin{aligned} force &= force_0 \times (1 + \epsilon_f) \ , \quad \epsilon_f \sim \mathcal{N}(0, \sigma_f^2) \\ \tilde{\mathbf{o}}_t &= \mathbf{o}_t \odot (1 + \boldsymbol{\epsilon}_o) \ , \quad \boldsymbol{\epsilon} \sim \mathcal{N}\Big(\mathbf{0}, \mathrm{diag}(\sigma_x^2, \sigma_{\dot{x}}^2, \sigma_\theta^2, \sigma_{\dot{\theta}}^2)\Big) \end{aligned} \tag{3.5}$$

Note that once ϵ_f is sampled, the value of perturbed force is kept fixed in the current trial until termination. The termination criteria and the returned rewards are the same as in a deterministic cart-pole task.

3.4.2 Different recurrent layers

We now look at how different recurrent layers affect training and uncertainty estimate in standard cart-pole setting. We have tested LLB model with RNN, LSTM and GRU recurrent units. We discover that a GRU unit can easily cause numerical error when handling trajectory with length larger than 20. This is due to the error accumulation in recurrent hidden state. Thus, a Tanh layer is appended to the hidden state of the GRU unit such that the equation becomes (original one listed in Sec 3.1):

$$\mathbf{h}_t = \tanh\Big(\mathbf{W}_{LLB}\big(z_t * \mathbf{h}_{t-1} + (1 - z_t) * \tilde{h}_t\big)\Big)$$

this regularises the hidden state and therefore solves the value explosion problem.

For different recurrent layer, we train a LLB model with hidden dimension 32 on 10 trajectories with length 20 for 5,000 iterations. The models are trained with Adam optimiser [44] with learning rate 10^{-3}. The model training loss plots are shown in Fig 3.4. It suggests that a LSTM recurrent unit can speed up the training and is comparatively more stable than others. The prediction performance is shown in Fig 3.5. From which we can see that both RNN and GRU units are still maintaining a wide distribution on their prediction while LLB-LSTM predicts more deterministically. On the same plots the mean prediction on data is shown where we use the mean of observation distribution instead of sampling:

$$\hat{\mathbf{o}} = \mathbb{E}[p(\mathbf{o}_t|\mathbf{h}_t)]$$

so that the stochasticity in prediction only stem from the distribution on weight parameter which is a way to demonstrate the amount of model uncertainty a model estimates. However, under a deterministic environment, high degree of model uncertainty such as RNN or GRU unit, is more of an indication of poor convergence property. When we implement LLB model with different recurrent unit, we found that such high model uncertainty indeed leads to bad model prediction and hence deteriorates the planning performance.

A quantitative analysis of accuracy and prediction uncertainty is given in Fig 3.7 which suggests that a LSTM recurrent unit can speed up the convergence to a more certain and accurate prediction

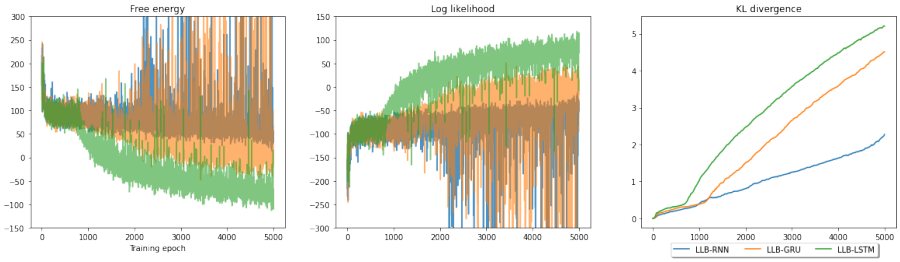


Figure 3.4: Loss plot of LLB-RNN(**blue**), LLB-GRU(**orange**) and LLB-LSTM(**green**) training on 10 trajectory with length 20 under deterministic cart-pole. The **left** panel shows the free energy $\mathcal{F}$; the **middle** shows the log-likelihood of the data, also known as reconstruction loss; the **right** panel shows the KL divergence between variational distribution $q(\mathbf{W}_{LLB})$ and prior $p(\mathbf{W}_{LLB})$.

under a deterministic environment.

Interestingly, in Fig 3.5 we have already seen that the model uncertainty of both RNN and GRU unit is much larger than LSTM after training for only $5,000$ iterations. This looks contradicting to Fig 3.3 where it is shown that the the LLB-LSTM model has similar value of diagonal variance of $q(\mathbf{W}_{LLB})$ where the model uncertainty are coming from. This is likely due to the output gate in the LSTM cell. According to Sec 3.1, the formula for LSTM is written as:

$$\mathbf{h}_t = \mathbf{W}_{LLB}\Big(\tanh(C_t) * o_t\Big)$$

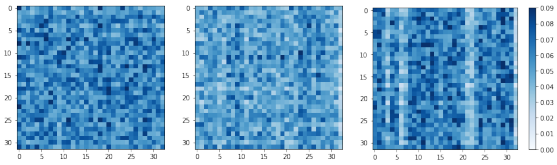
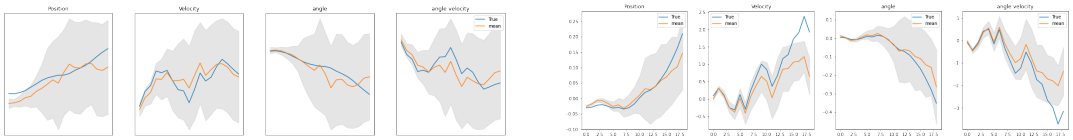


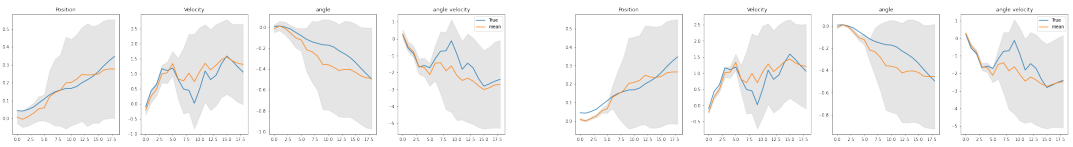
Figure 3.3: Matrix plot of diagonal covariance matrix of last-layer weight distribution $q(\mathbf{W}_{LLB})$ from models: LLB-RNN(**Left**); LLB-GRU(**Mid**) and LLb-LSTM(**Right**).

since the range of tanh function is between -1 and 1, o_t can act as a down-scaling factor shrinking the uncertainty in last-layer weight $\mathbf{W}_{LLB}$. Fig 3.6 shows that this is indeed happening where the LSTM cell is trained to reduce the value of output gate o_t.

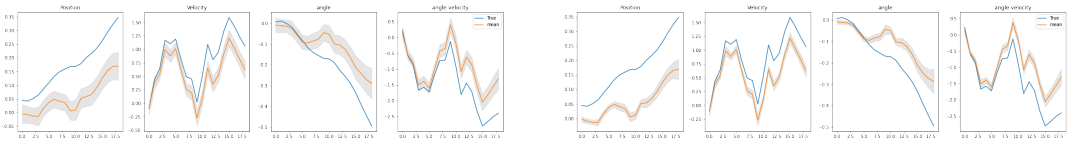
Hence, we pick LSTM as the recurrent layer for later experiment. In the next section we deliver a test on different number of hidden dimension.



(a) LLB-RNN



(b) LLB-GRU



(c) LLB-LSTM

Figure 3.5: Prediction on unseen trajectory after training on noise-free cart-pole. From top to bottom the prediction is made by **(a)** LLB-RNN; **(b)** LLB-GRU and **(c)** LLB-LSTM. For each model, **left** panel represents prediction by sampling from $q(\mathbf{W}_{LLB})$ and $p(\mathbf{o}_t)$; **right** panel represents prediction by sampling from $q(\mathbf{W}_{LLB})$ only (mean prediction on observations) which demonstrates the induced uncertainty from last-layer weight only. The **blue** curve represents the true trajectory, the **orange** represents the mean prediction and the shaded area represents ± 1 standard deviation at each time step.

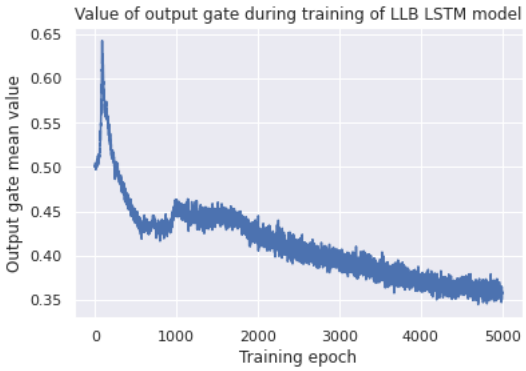
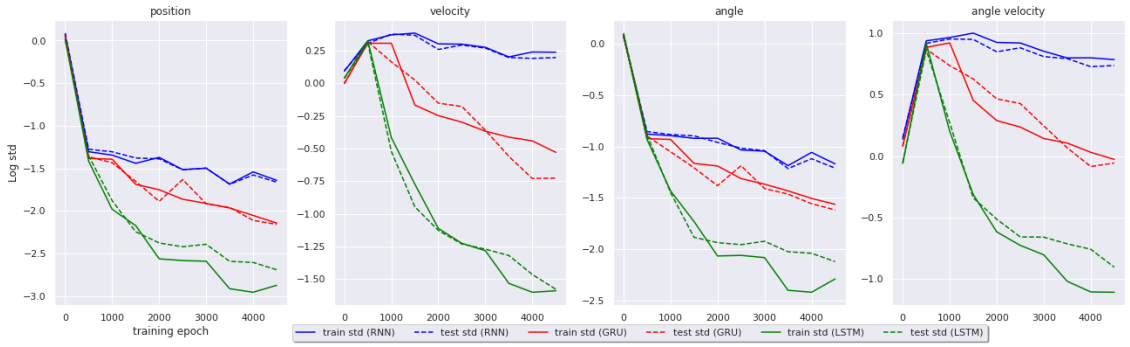


Figure 3.6: Shown above is the change of output gate value during LLB-LSTM model training. The value is averaged over number of time step and training data as well. Being at a deterministic environment, the model learns to minimise the effect of probabilistic weight parameter, which explains why the weight variance is still kept at a high level even though the prediction looks already certain.



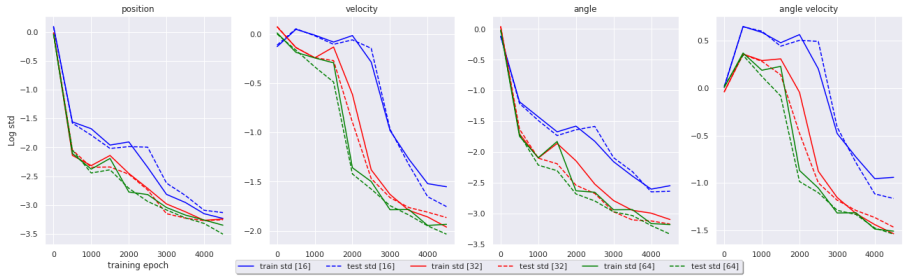
(a) change of average prediction standard deviation on both training and testing data during model training.

(b) change of average accuracy between mean prediction and true trajectory on both training and testing data during model training

Figure 3.7: Analysis of model predictive uncertainty and accuracy under deterministic cart-pole. For both plots, the solid curve represents the value on training data and the dashed curve represents the value on testing data. Color blue, red and green represents LLB-RNN, LLB-GRU and LLB-LSTM respectively. **(a)** Change of log std during model training. The value is averaged over time steps and also the number of testing data. **(b)** This illustrate the change in accuracy between mean prediction (orange curve in Fig 3.5) and true trajectory (blue curve in Fig 3.5).The values are also averaged over time steps and number of testing data.

3.4.3 Different hidden dimension

Next we test how different hidden dimension affect model learning. We train a LLB-LSTM model with three different hidden dimension: [16], [32] and [64]. Again, we evaluate the change in standard deviation on model prediction and the averaged accuracy between mean prediction and the true trajectory. Fig 3.8 shows that hidden dimension 32 and 64 leads to faster reduction in prediction uncertainty and therefore accelerates convergence on deterministic cart-pole data. The accuracy on testing data (illustrated as dashed line in the plot), however, does not differ much. Thus, for the purpose of computational complexity, we choose hidden dimension 32 in later experiment.



(a) Change in prediction standard deviation on both training and testing data

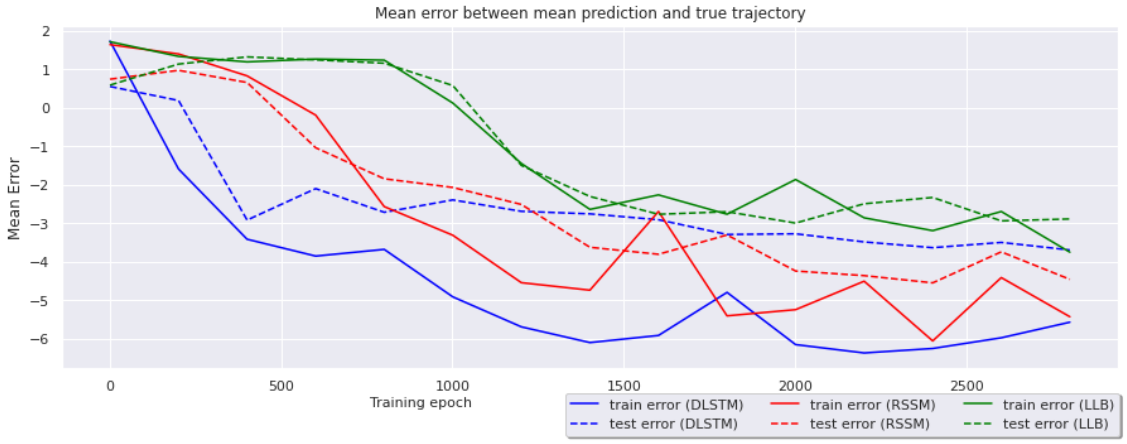
(b) Change in error between mean prediction and true trajectory on both training and testing data

Figure 3.8: Same as previous setting, shown above are the change of prediction log std and error on different choice of hidden dimension.

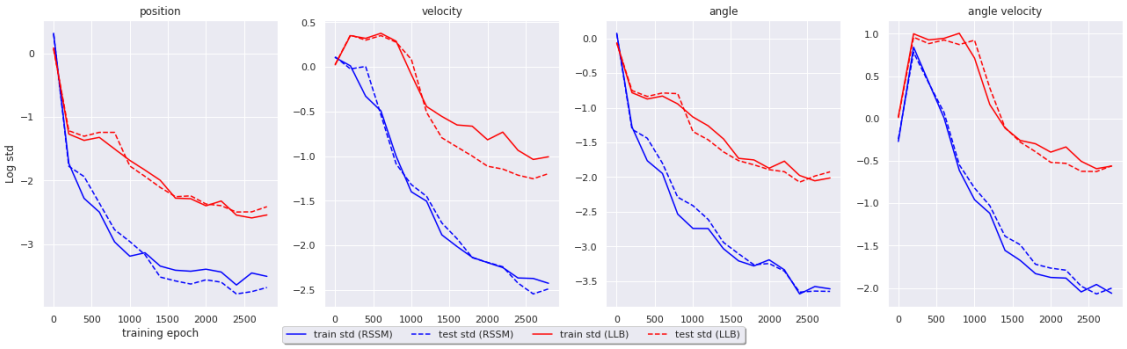
3.5 Comparing to baseline models on noise-free cart-pole

On deterministic cart-pole balancing task, we compare the learning performance with other baseline models: 1) Deterministic LSTM (DLSTM) and 2) Recurrent state-space model with LSTM layer (RSSM-LSTM). As the name suggested, a deterministic LSTM has no uncertainty included in the model and it is indeed a very powerful for learning noise-free data and making accurate prediction. A RSSM-LSTM integrates a latent variable model and a recurrent neural network which therefore only provides estimation of data uncertainty. A detail description will be given in Sec 5.1. Here we just treat RSSM as an intermediate which only models data uncertainty.

Fig 3.9(a) compares the log error between the prediction and the true trajectory during model



(a) Mean error

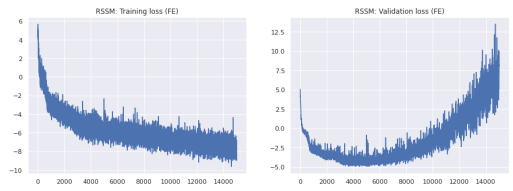


(b) Mean std

Figure 3.9: Shown above are the comparison of **(a)** prediction error and **(b)** prediction uncertainty for different models. **(a)**: Being trained on same set of data, all three models have been tested on the same unseen data iteratively and the curve represents the change of error in prediction as model training proceeds. The **solid** curve represents error on training data and the **dashed** line represents error on testing data. **(b)** For RSSM-LSTM and LLB-LSTM, we compare the change in log standard deviation in their prediction on both training (**solid** line) and testing data (**dashed** line). The values are averaged over time steps and number of testing data.

learning. Note that for probabilistic model such as LLB and RSSM, the error is computed between the mean prediction and the true trajectory. Fig 3.9(b) compares the standard deviation in probabilistic model prediction between RSSM-LSTM and LLB-LSTM. These plots suggest that while a deterministic model has the faster convergence and good accuracy on noise-free data, RSSM reduce its prediction uncertainty relatively faster than LLB. This is due to the model uncertainty in a LLB model which slow down the training. However, as we will see later in the planning experiment, such "drawback" is actually beneficial for policy learning. In next section we move on to a noisy cart-pole environment where probabilistic dynamic models come into play.

(a) DLSTM loss



(b) RSSM-LSTM loss (FE)

(c) LLB-LSTM loss (FE)

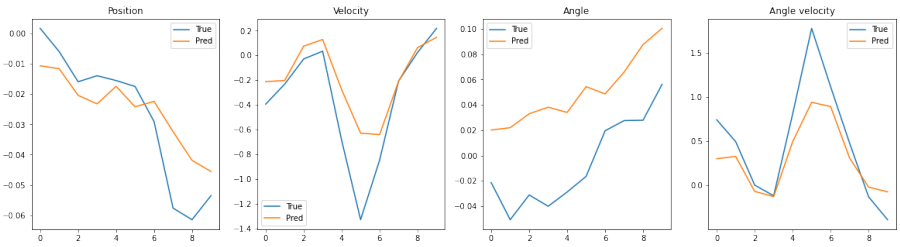
Figure 3.10: value of loss function on both training and testing data during model training. For each sub-graph, **left** panel represents loss on training set and **right** represents on testing set.

3.6 Comparing to baseline models on noisy cart-pole

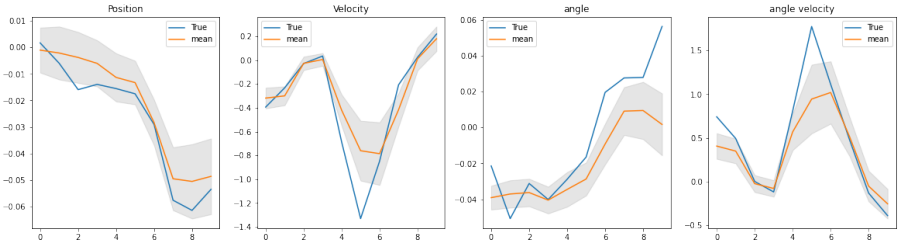
We first test LLB models on noisy cart-pole data with different recurrent units. However, we found no obvious difference in model prediction at *early-stopping*, except that both GRU and RNN unit are more computational expansive than a LSTM unit. We further compare the LLB model with baseline models. We train all models on 50 noisy trajectories with length 10. With early-stopping applied, DLSTM is trained approximately for 8,000 iterations on mean square loss; RSSM-LSTM and LLB-LSTM are trained for around 5,000, and 10,000 iterations respectively on variational free energy loss. Fig 3.10 compare the loss of three models on the same training and validation set. Compared to LLB model, a RSSM-LSTM tends to be easier to over-fit to the noisy data. This is likely due to the lack of model uncertainty estimate such that the model fail to capture the systematic error in the dynamic.

Additionally, Fig 3.11 gives an example of model prediction on the same unseen trajectory at early-stopping. It is easy to see that without uncertainty estimate, deterministic LSTM (DLSTM) shows larger discrepancy in its prediction, while a probabilistic model such as RSSM or LLB, holds a margin of error which is believed to have better generalisation ability on unseen data. Also the prediction of LLB tends to have slightly better match than RSSM. In Section 5.4 we will be comparing these two model in more detailed.

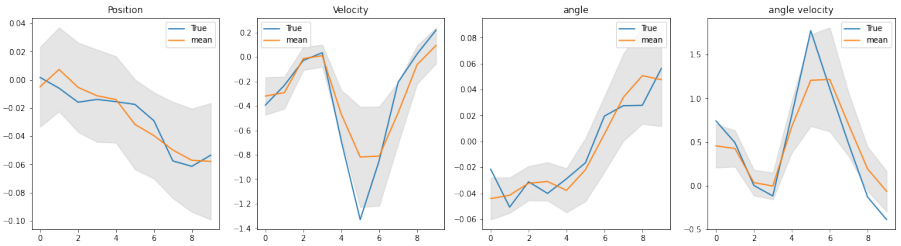
To sum up, we have tested the model learning performance of a last-layer Bayesian recurrent neural network with different model structure. Also we have shown that it is able to give decent prediction on both deterministic and stochastic environment by adjusting the uncertainty estimate. This demonstrates that at least in simple control task, adding only a Bayesian weight parameter to the last layer of the recurrent cell does not deteriorate the model learning performance on both noise-free and noisy data. And particularly on noisy cart-pole where the underlying dynamics suffer from model uncertainty, a LLB model gives more robust training than a RSSM which is lack of model uncertainty estimate.



(a) DLSTM prediction after 8,000 training iterations



(b) RSSM-LSTM prediction after 5,000 training iterations



(c) LLB-LSTM prediction after 10,000 training iterations

Figure 3.11: Shown above are the model prediction of **(a)**: DLSTM ; **(b)**: RSSM-LSTM and **(c)**: LLB-LSTM at early-stopping. The orange curve represent the prediction or the mean prediction for a probabilistic model. The blue curve represents the true trajectory.

For the rest of the report, we will apply LLB-LSTM model in model-based reinforcement learning where we investigate how it leverage uncertainty to improve planning performance compared to other deterministic or probabilistic baseline models.

Chapter 4

Planning with learnt probabilistic models

In this chapter, we consider how a trained Last layer Bayesian recurrent neural network is applied in model-based reinforcement learning. In particular, we discuss gradient-based planning in which we take gradient of some policy learning objective function with respect to the policy parameter. Also due to the non-linearity in neural network, a particle filtering approximation is used solve intractability.

4.1 Policy gradient with particle filtering approximation

Assume we have access to a known differentiable reward $r(\cdot)$ or cost $c(\cdot)$ function of the environment. For example, in the experiment of Cart-pole balancing, the state $\mathbf{o}$ is determined by $[\mathbf{x}\ ,\ \dot{\mathbf{x}}\ ,\ \boldsymbol{\theta}\ ,\ \dot{\boldsymbol{\theta}}]$ where $\mathbf{x}$ is the position of the cart; $\dot{\mathbf{x}}$ is its velocity; $\boldsymbol{\theta}$ is the angle of the pole and $\dot{\boldsymbol{\theta}}$ is its angle velocity. Then our choice of cost function is denoted as:

$$c(\mathbf{o}_t) = \mathbf{x}_t^2 + \boldsymbol{\theta}_t^2$$

such that minimising the cost function refers to minimising the absolute value of cart position and pole angle so that the cart-pole system is able to balance while not running off the track. Then for some trajectory $\boldsymbol{\tau} = \mathbf{o}_1, \cdots, \mathbf{o}_T$, the cumulative cost and the expected cumulative cost are written as:

$$c^{\text{total}}(\boldsymbol{\tau}) = \sum_{t=1}^{T} c(\mathbf{o}_t)$$

$$\mathbb{E}_{p(\boldsymbol{\tau})}[c(\boldsymbol{\tau})] = \mathbb{E}\Big[\sum_{t=1}^{T} c(\mathbf{o}_t)\Big] = \sum_{t=1}^{T} \mathbb{E}_{p(\mathbf{o}_t)}[c(\mathbf{o}_t)]$$

Denote a parametric policy function as $\boldsymbol{\pi}_{\boldsymbol{\phi}}(a_t|\mathbf{o}_t)$ with parameters denoted as $\boldsymbol{\phi}$. The policy can either be deterministic or stochastic $p_{\boldsymbol{\phi}}(a_t|\mathbf{o}_t)$. In our experiment we use a deterministic policy as we are focusing the uncertainty from the model only and the task is relatively easy. The trajectory

generated by following $\boldsymbol{\pi}_\phi$ is parametrised by policy parameter ϕ, written as: $\boldsymbol{\tau}(\boldsymbol{\phi}) = \{\mathbf{o}_1, \cdots, \mathbf{o}_T\}$. The objective of planning becomes minimising the expected cumulative cost with respect to the policy parameter $\boldsymbol{\phi}$. Namely, the optimal parameter $\boldsymbol{\phi}^*$ is found as:

$$\boldsymbol{\phi}^* = \arg\min_{\boldsymbol{\phi}} \sum_{t=1}^{T} \mathbb{E}_{p(\mathbf{o_t}|\boldsymbol{\phi})}[c(\mathbf{o}_t|\boldsymbol{\phi})] \tag{4.1}$$

The difficulty is to infer the marginal state distribution $p(\mathbf{o}_t|\boldsymbol{\phi})$ and make it differentiable w.r.t $\boldsymbol{\phi}$. A closed-form derivative $\frac{\partial \mathbb{E}[c(\mathbf{o}_t)]}{\partial \boldsymbol{\phi}}$ is given by PILCO which use moment matching to deliver a differentiable approximation of $p(\mathbf{o}_t)$. However, this is inapplicable to neural network for its computational complexity [52]. Instead, we use Monte-Carlo estimate of the expected cumulative cost and reparametrisation trick so that we can back-propagate through sampling.

First, we sample N trajectories from the learnt probabilistic model with length T. This can be done by ancestral sampling [10], or particle filtering introduced in Sec 3.3.2 . For a Last layer Bayesian recurrent neural network, we sample N last layer weight values $\mathbf{W}_{LLB}^{(i)} \sim p(\mathbf{W}_{LLB})$, $i = 1, \cdots, N$ and for each value of weight we perform one model roll-out, resulting in N sequences of deterministic recurrent hidden states $\{\mathbf{h}_1^{(i)}, \mathbf{h}_2^{(i)}, \cdots, \mathbf{h}_T^{(i)}\}$, $i = 1, \cdots, N$, from which we can sample a predictive observation using the generating function g_θ : $\hat{\mathbf{o}}_t^{(i)} \sim p(\mathbf{o_t}|g_\theta(\mathbf{h}_t^{(i)}))$. Then a set of trajectories $\{\mathbf{o}_1^{(i)}, \cdots, \mathbf{o}_T^{(i)}\}$ is obtained which is parametrised by policy parameter $\boldsymbol{\phi}$ and therefore can be easily differentiated with reparametriation applied.

With the sampled trajectories, the expectation term in equation 4.1 can be approximated as:

$$\boldsymbol{\phi}^* = \arg\min_{\boldsymbol{\phi}} \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} c(\mathbf{o}_t^{(i)}|\boldsymbol{\phi})$$

and the derivative of cost of individual particle w.r.t $\boldsymbol{\phi}$ is (set $c_t^{(i)} = c(\mathbf{o}_t^{(i)}|\boldsymbol{\phi})$):

$$\frac{\partial c_t^{(i)}}{\partial \boldsymbol{\phi}} = \frac{\partial c_t^{(i)}}{\partial \mathbf{o}_t^{(i)}} \frac{\partial \boldsymbol{\mu}_t^{(i)}}{\partial \boldsymbol{\phi}} + \frac{\partial c_t^{(i)}}{\partial \mathbf{o}_t^{(i)}} \frac{\partial \boldsymbol{\Sigma}_t^{(i)}}{\partial \boldsymbol{\phi}} \cdot \boldsymbol{\epsilon}$$

$$\text{Since } \mathbf{o}_t^{(i)} = \boldsymbol{\mu}_t^{(i)} + \boldsymbol{\Sigma}_t^{(i)} \cdot \boldsymbol{\epsilon} \ , \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$$

$$[\boldsymbol{\mu}_t^{(i)}, \boldsymbol{\Sigma}_t^{(i)}] = g_\theta(\mathbf{h}_t^{(i)}|\boldsymbol{\phi})$$

according to which the policy parameter is updated as:

$$\boldsymbol{\phi} \longleftarrow \boldsymbol{\phi} - \alpha \Big[\sum_{i=1}^{N} \sum_{t=1}^{T} \frac{\partial c(\mathbf{o}_t^{(i)}|\boldsymbol{\phi})}{\partial \boldsymbol{\phi}} \Big]$$

Now that we know how policy gradient works, next section will discuss how it is integrated with model learning to give the planning algorithm we implement in the experiment.

4.2 Learning-Planning iteration

One common planning algorithm is to iterate between model learning and policy learning [17, 36, 29]. For example, Algorithm 2 summarises planning in PILCO. This planning framework relies on the principle that planning using the current simulated model can provide better policy in the real environment to collect more valuable data, which in turns improve the current model by training with new data collected, as suggested in Fig 4.1.

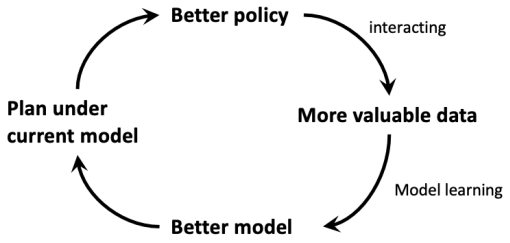


Figure 4.1: Learning-Planning iteration

In a cart-pole balancing task, the trained policy is able to generate longer trajectories in the environment, which improve long-term prediction of the models and therefore the quality of policy as well.

Algorithm 2: PILCO

```
Initialise model and policy function
for iteration = 1, ··· , N_i do
    Interacting with environment and record the data
    Model learning on current data pool
    Do model roll-out predicting trajectory p(o_0), ··· , p(o_T)
    Evaluate total expected cost
    Minimise the expected cost with respect to policy parameter and update the policy
```

Similar to PILCO, the algorithm used in our experiment is listed in Algorithm 3. At the very beginning, we start with N_{init} number of episodes collected using random policy, then we train the model and perform policy learning. At the end of each iteration i, we collect new episode through interacting with the environment with the updated policy. Note that it is common to add exploration noise $\epsilon \sim p(\epsilon)$ to the action generated from the controller [36, 35]. In our experiment we did not include this as the task is relatively simple.

In next chapter we implement Algorithm 3 on LLB and other baseline model on both deterministic and stochastic cart-pole balancing task.

Algorithm 3: Model-based policy gradient with particle sampling

Set constant $N_{init}, N_{iter}, N_{train}, N_{plan}, N_{test}, B$(training batch size), R(particle number)
Initialise training data pool $\mathcal{D}$
Initialise LLB model parameters θ and policy/controller parameter ϕ ;

// Initial data collecting //
for $n = 1, \cdots, N_{init}$ **do**
    `Environment reset:` $\mathbf{o}_1 \longleftarrow env.reset()$
    while *task does not fail*, $t = 1, \cdots, T_n$ **do**
        $a_t \longleftarrow$ random policy$(\mathbf{o}_t)$
        Observe next state: $\mathbf{o}_{t+1} \longleftarrow env.step(a_t)$
    `Record the collected data:` $\mathcal{D} \longleftarrow \mathcal{D} \cup \{(\mathbf{o}_t, a_t)_{t=1}^{T_n}\}$
for *iteration* $i = 1, \cdots, N_{iter}$ **do**

    // Model fitting // (Sec 3.3.2)
    for *training step* $j = 1, \cdots, N_{train}$ **do**
        From training data pool $\mathcal{D}$ sample trajectory batches $\{(\mathbf{o}_t, a_t)_{t=1}^{T_b}\}_{b=1}^{B}$
        Compute variational free energy function $\mathcal{F}$ (Eq 3.4) and take gradient w.r.t. θ
        Update model parameter θ

    // Policy learning // (Sec 4.1)
    for *planning step* $k = 1, \cdots, N_{plan}$ **do**
        Reset an initial observations $\mathbf{o}_1$
        Initialise R number of particle samples of $\mathbf{W}_{LLB}$: $\mathbf{W}_{LLB} \sim q(\mathbf{W}_{LLB})$
        Forward propagate on each sample for horizon H, obtaining trajectories $\{\mathbf{o}_{1:H}^{(m)}\}_{m=1}^{R}$
        Compute the average expected cost on trajectories $\{\mathbf{o}_{1:H}^{(m)}\}_{m=1}^{R}$ and take gradient w.r.t. ϕ
        Update policy parameter ϕ

    // Policy testing //
    for *testing step* $l = 1, \cdots, N_{test}$ **do**
        Reset an initial observation $\mathbf{o}_1$, and reward $r_l = 0$
        while *task does not fail* **do**
            $a \longleftarrow controller_{\phi}(\mathbf{o})$
            $\mathbf{o} \longleftarrow env.step(a)$
            $r_l \longleftarrow r_l + 1$
    Compute average testing reward: $r_i^{avg} = 1/N_{test} \sum_{l=1}^{N_{test}} r_l$

    // Data collecting //
    `Environment reset:` $\mathbf{o}_1 \longleftarrow env.reset()$
    while *task does not fail*, $t = 1, \cdots, T_n$ **do**
        $a_t \longleftarrow controller_{\phi}(\mathbf{o}_t)$
        Observe next state: $\mathbf{o}_{t+1} \longleftarrow env.step(a_t)$
    `Record the collected data:` $\mathcal{D} \longleftarrow \mathcal{D} \cup \{(\mathbf{o}_t, a_t)_{t=1}^{T}\}$

Chapter 5

RL Experiment

After a model has been trained, the focus is now on planning. We will be comparing our last-layer Bayesian recurrent neural network to other baseline model on Algorithm 3. We also evaluate how different uncertainty estimate affect the performance of planning under both a deterministic and a stochastic environment.

Particularly, we are comparing LLB-LSTM model with 1) deterministic LSTM (DLSTM) and 2) recurrent state-space LSTM (RSSM LSTM). The planning is carried out with policy gradient using reparametrisation trick (as discussed in Chapter 4). The experiment is firstly done on noise-free cart-pole balancing. Next we test the LLB-LSTM model on limited number of training data with different uncertainty estimate schemes. We finish the chapter with experiments on noisy cart-pole and a comparison between LLB-LSTM and RSSM-LSTM.

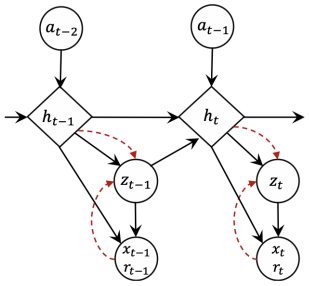
5.1 Baseline model

Here we introduce two baseline models implemented in the experiments. As for RSSM-LSTM, we also describe explicitly how it is trained and how prediction is made through different pathway of the model.

Deterministic LSTM (DLSTM)

In order to better understand how probabilistic models are beneficial for planning, we implement a deterministic LSTM network where no uncertainty is included. Different from a LLB model, the DLSTM model is trained with mean square error loss.

Recurrent state-space model (RSSM LSTM)



In Sec 2.4.2 we mentioned that a RSSM is very similar to a VRNN [15] model which is an integration of latent variable models and recurrent neural networks. It is different from a LLB model in uncertainty estimation. A LLB model has stochastic recurrent hidden states which are expected to model systematic error exists in real world. While for a RSSM, the recurrent layer is deterministic and it is the latent variables $\mathbf{z}_t$ at each time step that are uncertain. One way to understand the difference is to see the latent variable $\mathbf{z}_t$ as a probabilistic summary of historical information $(\mathbf{x}_0, \cdots, \mathbf{x}_{t-1})$ and also the information from current state $\mathbf{x}_t$. Then such an uncertain summary together with an action are input to the recurrent layer, propagating uncertain message to the next time step through a deterministic transition function $\mathbf{f}_\theta$.

In fact, during model training of a RSSM, this historical message is encoded in the posterior distribution of latent variables $q(\mathbf{z})$. As shown in Fig 5.1(a), the distribution $q(\mathbf{z}_2)$ takes current hidden state and observation as input and according to Markov property we have:

$$q(\mathbf{z}_2|\mathbf{h}_2, \mathbf{o}_2) = q(\mathbf{z}_2|\mathbf{o}_0, \mathbf{o}_1, \mathbf{o}_2)$$

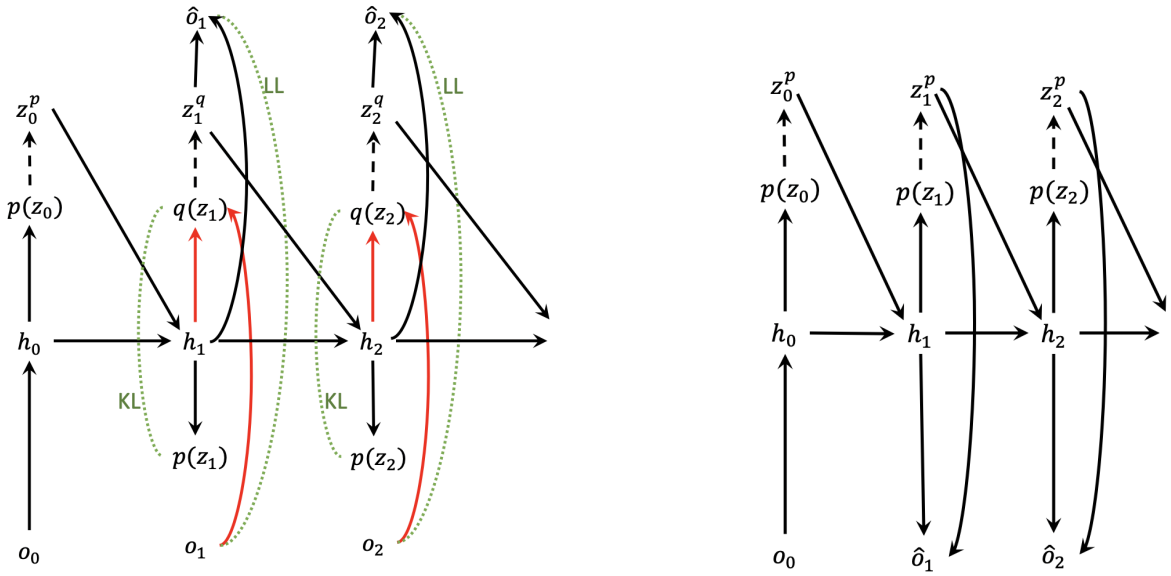
so $q(\mathbf{z}_t)$ can indeed be trained to approximate the posterior on $\mathbf{z}_t$ in a Kalman filtering manner*. Meanwhile, a network for prior distribution on $\mathbf{z}_t$ is conditioned on current hidden state $\mathbf{h}_t$ only, denoted as $p(\mathbf{z}_t|\mathbf{h}_t)$. Note that all networks for modelling distribution apply reparametrisation trick, or *amortised variational inference*, where the network outputs both mean and variance as output. This is to enable gradient back-propagation on distribution parameters.

The training of a RSSM is to optimise the variational free energy objective function written as:

$$\sum_{t=1}^{T} \mathbb{E}_{q(\mathbf{z}_t|\mathbf{h}_t,\mathbf{o}_t)}\Big[\log p\big(\mathbf{o}_t|\mathbf{h}_t, \mathbf{z}_t^q \sim q(\mathbf{z}_t|\mathbf{h}_t, \mathbf{o}_t)\big)\Big] - \mathbb{E}_{q(\mathbf{z_{t-1}}|\mathbf{h}_{t-1},\mathbf{o}_{t-1})}\Big[\mathbf{KL}[q(\mathbf{z}_t|\mathbf{h}_t, \mathbf{o}_t)\|p(\mathbf{z}_t|\mathbf{h}_t)]\Big]$$

with respect to model parameters. This includes parameters of posterior, prior networks, generating function and also transition function. It is noticeable that the KL term in objective function is minimised so at each time step the prior $p(\mathbf{z}_t|\mathbf{h}_t)$ is trained to be closed to posterior distribution $q(\mathbf{z}_t|\mathbf{h}_t)$. This makes it possible to discard the posterior network and only use prior during prediction phase. As shown in Fig 5.1(b), we use samples from the prior $\mathbf{z}_{t-1}^p$ to generate predicted observation $\hat{\mathbf{o}}_t$ and the sample is also fed to the hidden state h_t, which were supposed to be the works of a posterior sample in training.

*This means the posterior is conditioned on past information, but one may also use smoothing posterior which conditions on both past and future information [34]



(a) Training

(b) Predicting

Figure 5.1: (a) Model graph of RSSM during model training. Black dashed arrow represents sampling from distribution; red arrow represent the inference model (filtering posterior on latent state at each time step); green dotted curve shows the computation of objective function which is LL-KL. (b) Model graph during prediction phase. Samples are taken from prior instead and used for both transition and prediction.

The next section gives a brief summary of experiments done on deterministic cart-pole balancing task with LLB and baseline models.

5.2 On deterministic cart-pole balancing task

PILCO suggests using probabilistic dynamic model even when the environment is deterministic. We first conduct the experiment on a deterministic cart-pole balancing task and compare the performance of different models.

We implement Algorithm 3 on DLSTM, RSSM-LSTM and LLB-LSTM models. For all models, the hidden dimension is set as 32. The initial encoder is a two-layer MLP $[32, 32]$ with ReLU and Tanh activation functions. The emission/generating model is a also a two-layer MLP $[32, 32]$ while for probabilistic models (RSSM, LLB), an extra data variance layer is appended.

During learning-planning iteration, the number of initial collected data is 10 and the length of trajectories is around 20 under a random policy. The number of iterations N_{init} is set as 50; the number of training steps N_{train} is 1000 with batch size $B = 10$. As for planning, the number of planning steps N_{plan} is 50 ; the number of particles is 1,000 and the planning horizon H is 10. This implies that during each iteration, we provide 50 random initial observations based on which the agent performs model roll-out using particle sampling and try to optimise the policy on simulated trajectories. To evaluate the policy, at the end of each iterations, we evaluate the

updated policy by performing 20 independent trial with the real environment and compute the averaged testing rewards (Line 28 in Algorithm 3).

Fig 5.3 shows the change in averaged testing rewards as the algorithm proceeds. The experiment is done on 6 random seeds for each model and the plots show that both probabilistic models give averaged reward value closed to 200 (where task is consider solved) after 20 iterations. This means the agent almost solve the task after seeing $10 + 20$ trajectories, which reveals data efficiency of MBRL. While a DLSTM model requires more iterations to converge and the variance over multiple experiment repetitions is the highest. Additionally, when comparing to LLB-LSTM, a RSSM-LSTM model experiences slightly higher variance at early-stage of the experiment but it manages to reduce it and keep it at the similar level as LLB.

There are two possible reasons for the discrepancy in planning performance. One is that the model prediction accuracy is deviated by the incoming new collected data so that it is the problem with model learning and prediction which deteriorate the planning performance. Fig 5.2 shows the results of a diagnostic experiment on this in which we tested how the model prediction error changes when they are re-trained with new data collected. The error is calculated between the mean prediction and the true trajectory and is averaged over time step. The plots show that all models learn to reduce the error when new data is collected and DLSTM even has the lowest error value. Hence we can see the first reason is not giving a convincing explanation.

Figure 5.2: Change in log mean error on testing set when new data is added.

Another potential reason is due to the uncertainty estimate in a probabilistic model that helps policy learning. To verify this, we examine the quality of policy learning on the learnt models.

Fig 5.4 provide a diagnostic plot of policy learning during one particular experiment repetition which might gives some insights. The aim is to test how a learnt policy generalises on unseen initial state under the model it is trained on. Note that this is different from testing on the real environment as it gives a sense of the quality of policy learning with no regard to the quality of model learning. Thus, during each learning-planning iteration, after policy learning ends, we give 20 random initial observations to the agent and use the current trained model and policy function to roll-out simulated trajectories for both length 10 and 100 respectively. Then we compute the expected cumulative cost from these generated trajectories. Note that for probabilistic dynamic models the model roll-out is again carried out by particle sampling and the cumulative rewards are averaged over number of particles.

Fig 5.4 shows the mean and std of the expected cost over those 20 initial observations. From the plot we can see that on trajectory length 10 which is also the horizon the agent plans on, both DLSTM and RSSM model experience high variance during early iterations while LLB-LSTM model shows a consistent low variance on unseen initial state. Also the mean in LLB-LSTM plot is showing clearly a decreasing trend. On trajectory length 100, LLB-LSTM manage to keep the expected cost at a very low level even after one iteration, while DLSTM and RSSM require a

Figure 5.3: **Top**: The curve represents the averaged rewards over 20 independent trials of models DLSTM (blue), RSSM-LSTM (orange) and LLB-LSTM (green). The shaded area represents the standard deviation over 6 random seeds. **Bottom**: A more clear plot showing the standard deviation of averaged rewards over 6 experiment repetitions with different random seed on model initialisation.

few more iterations and DLSTM has the largest variance on long-term trajectories. This suggests that planning on the a probabilistic models enables the agent to generalises on new simulated trajectories more efficiently and also with greater stability. As the models are kept improving to accommodate new training data, such an advantage leads to better performance on the real environment. Moreover, the stability of policy learning is provided by the uncertainty in model prediction. For example, Fig 5.6 compare the state exploration between different models with particle sampling applied. Since a deterministic LSTM has no uncertainty included, given with an initial observation the model roll-out is just one single trajectory as suggested by Fig 5.6(a). Whereas for RSSM-LSTM and LLB-LSTM, multiple trajectories are generated. This enables their agents to consider more potential states along the path so the updated policy is comparatively less over-confident than a DLSTM model. This explains the early high variance of DLSTM during policy testing on long-term trajectories.

As for the reason why LLB-LSTM is slightly more stable than RSSM-LSTM, Fig 5.6 suggests that a LLB-LSTM model maintains higher degree of stochasticity in its prediction than RSSM when new data is collected. It is still unclear whether it is the amount of prediction uncertainty or the quality of epistemic/model uncertainty that provides better robustness for LLB-LSTM under a deterministic environment. In either way, however, the inclusion of epistemic/model uncertainty in the model is shown to be beneficial for model planning and a potential future work might be to conduct the experiment on a large scale or a more complex environment.

Figure 5.4: Diagnostic plots of policy learning quality on the learnt models in one particular experiment repetition. During each iteration, we test the updated policy on the trained model to see the cost on trajectories with length 10 and 100. From **top** to **bottom**, the plots show the results from DLSTM, RSSM and LLB models respectively. For each model, **left** panel shows the expected cost on simulated trajectories with length 10; **right** panel shows the expected cost with length 100. The shaded area represents the standard deviation over 20 random initial observations.

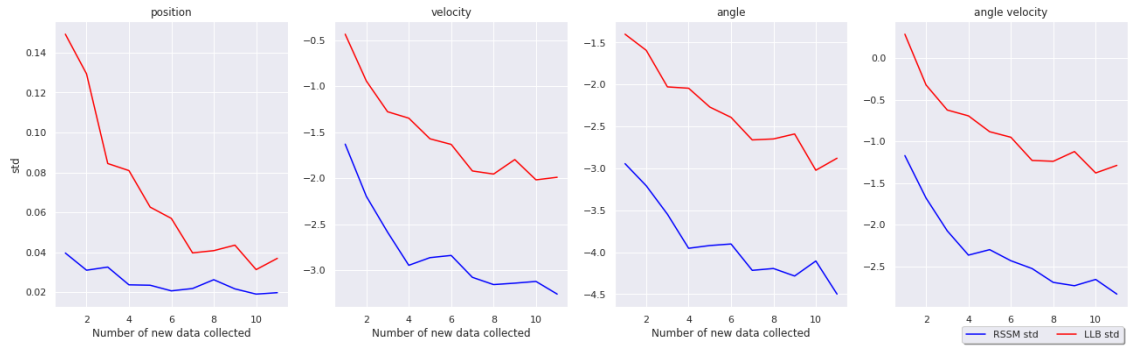
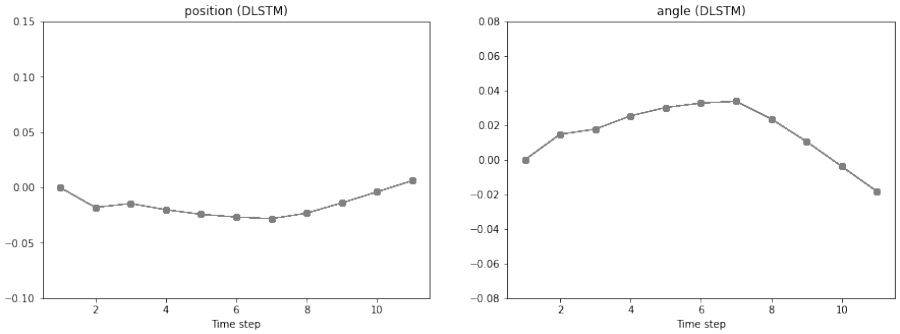
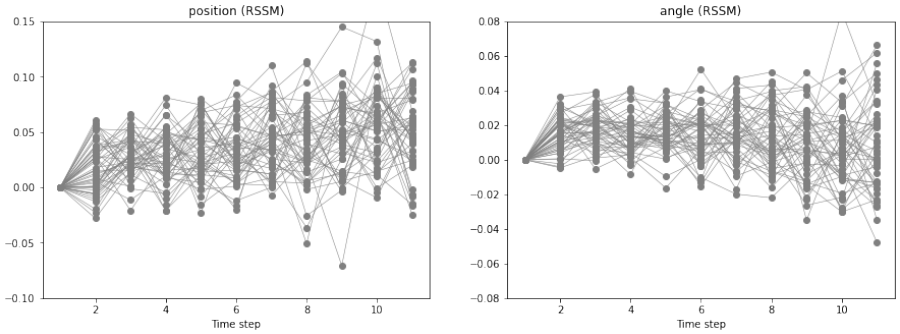


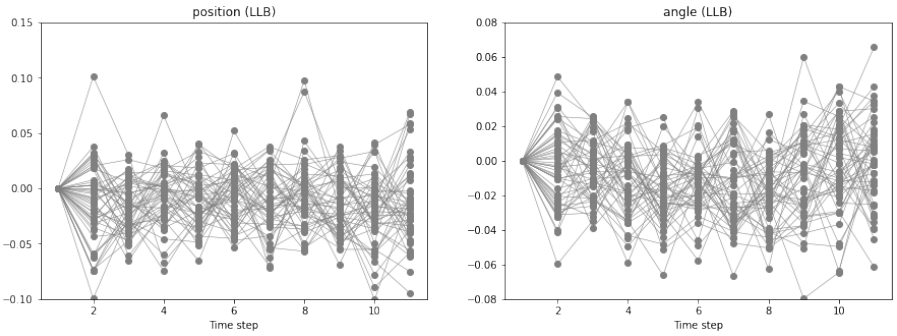
Figure 5.5: Shown above are the changes of std in model prediction as new data is collected. When new data is added, the model is re-trained on the data set for 1,000 iterations which follows the setting of Algorithm 3. The plots show that LLB-LSTM model maintains more uncertainty in its prediction which could potentially provide robustness for policy learning.



(a) DLSTM state exploration



(b) RSSM state exploration



(c) LLB state exploration

Figure 5.6: Shown above are the visited states (position x and angle θ in cart-pole) when the agent performs model roll-out given an initial observation. The connecting line represents two states are on the same trajectory path. DLSTM **(a)** gives deterministic prediction so only one trajectory is generated. Probabilistic models **(b)&(c)** generate multiple trajectories using particle sampling.

5.3 On deterministic cart-pole with limited data available

In this experiment we explore how different uncertainty estimation affect planning performance of a LLB-LSTM model when the size of training data is limited. Particularly, we will be looking at four different estimation schemes when doing planning:

1. **No noise included**: this means we use mean prediction for planning, including mean prediction on last-layer weight distribution $q(\mathbf{W}_{LLB})$:

   $$\text{Sampling prediction: } \mathbf{W}_{LLB} \sim q(\mathbf{W}_{LLB}) \;\; ; \;\; \text{Mean prediction: } \mathbf{W}_{LLB} = \mathbb{E}[q(\mathbf{W}_{LLB})]$$

   and mean prediction on observation $\hat{\mathbf{o}}_t$:

   $$\text{Sampling prediction: } \hat{\mathbf{o}} \sim p(\mathbf{o}_t|g_\theta(\mathbf{h}_t)) \;\; ; \;\; \text{Mean prediction: } \hat{\mathbf{o}}_t = \mathbb{E}\big[p(\mathbf{o}_t|g_\theta(\mathbf{h}_t))\big]$$

   such that the agent plans in the same deterministic way as a DLSTM model does.

2. **Only data noise included:** this means we use mean prediction on last-layer weight but sampling from observation distribution.

3. **Only weight noise included:** this means we sample weight values from its distribution but use mean prediction on observation.

4. **Both noise included:** this is the same setting as we conducted the previous experiment. Both weight value and observation prediction are generated through sampling.

We carried out planning on 6 sets of training data generated by random actions. The data size of each training set is 5 and the trajectory length is fixed at 10. The experiment procedure is summaries in Algorithm 4. Early-stopping is applied on model training which means training terminates when loss function on validation set converges or starts to show the sign of over-fitting. We set the number of planning iteration N_{plan} as 100 and number of particles R as 1,000. The planning horizon H is still 10.

Fig 5.7 illustrates the change of testing rewards on 6 different data sets. One can observe that the agent find it relatively harder to plan when no uncertainty (**blue** curve) or only data uncertainty is included (**orange** curve). Whereas sampling from last-layer weight distribution $q(\mathbf{W}_{LLB})$ (**green** curve) shows obvious improvement in planning performance. This can be explained by diagnostic plots Fig 5.8 where we evaluate the generalisation ability of the trained policy, just like what we did in previous experiment (Fig 5.4). It turns out that even though planning deterministically (with no noise included) can achieve the lowest testing cost on trajectory horizon 10, the cost on long-term trajectory is devastating. This echoes with the conclusion in previous experiment where we discovered that inclusion of uncertainty can offer robustness in planning. It is also interesting to see that the inclusion of data noise only also suffer from terrible generalisation, whereas planning with weight noise included is much more effective.

One possible explanation of this is suggested by Fig 5.9 where the state exploration with different uncertainty schemes is plotted. State exploration by weight uncertainty only is shown to have more smooth trajectories and less randomness involved. This can help policy learning. Since the model parameters are fixed during the same model roll-out with no data noise, each time the

Algorithm 4: Experiment of LLB-LSTM on small data size setting

Set constant N_{plan}, N_{test}, R
Initialise 6 training data sets $\{\mathcal{D}_1, \cdots, \mathcal{D}_6\}$ and 6 validation set $\{\mathcal{V}_1, \cdots, \mathcal{V}_6\}$
for $i = 1, \cdots, 6$ **do**

  // Model fitting //
  Initialise a LLB-LSTM model
  while *not early-stopping* **do**
    Train the LLB-LSTM model on data set $\mathcal{D}_i$ (Sec 3.3.2)

  for *each uncertainty estimate scheme* **do**
    for *planning step* $j = 1, \cdots, N_{plan}$ **do**

      // Policy learning // (Sec 4.1)
      Reset an initial observations $\mathbf{o}_1$
      Initialise R number of particle samples $\mathbf{W}_{LLB}$: $\mathbf{W}_{LLB} \sim q(\mathbf{W}_{LLB})$ or use mean prediction $\mathbf{W}_{LLB} = \mathbb{E}[q(\mathbf{W}_{LLB})]$
      Forward propagation on each sample for horizon H, obtaining trajectories $\{\mathbf{o}_{1:H}^{(m)}\}_{m=1}^{R}$ through sampling or mean prediction.
      Compute the average expected cost and take gradient w.r.t policy parameter ϕ
      Update policy parameter ϕ

      // Policy testing //
      for *testing number* $k = 1, \cdots, N_{test}$ **do**
        Reset an initial observation $\mathbf{o}_1$, $r_k = 0$ **while** *task does not fail* **do**
          $a \longleftarrow controller_{\phi}(\mathbf{o})$
          $\mathbf{o} \longleftarrow env.step(a)$
          $r_k \longleftarrow r_k + 1$
        Record the total reward obtained r_k
      Compute the average reward $R_{mean} = 1/N_{test} \sum_{1}^{N_{test}} r_k$

agent re-visit a state, the deterministic controller and the model will return a consistent action and prediction for the next state, so the gradient back-propagation is relatively more efficient and suffer less from the random perturbation from data noise. Also with the help of weight noise, planning with both noise (**red** curve) can gain better performance as opposed to planning without (**orange**).

Figure 5.7: Planning performance with different uncertainty estimate schemes on 6 different data set with size 5 and trajectory length 10. The **blue** curve represents planning with no uncertainty included; **orange** represents only data uncertainty (in emission layer) is considered; **green** represents only last-layer weight uncertainty is considered and **red** curve represents planning with both uncertainty included.

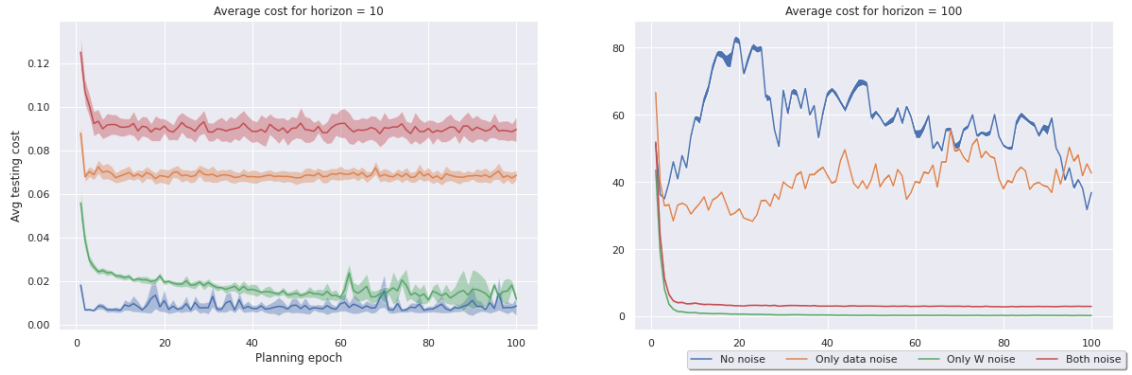


Figure 5.8: Diagnostic plot of LLB model on different uncertainty schemes on data set 2. At the end of each planning iteration, the agent is given 20 random initial observations $\mathbf{o}_0$ and the mean cost is computed on the learnt LLB model for horizon 10 (**Left**) and 100 (**Right**). Being trained to minimise cost with horizon 10, agent without any uncertainty included (green curve) is able to give the lowest testing cost on unseen initial observation. However, the testing cost on horizon 100 trajectory takes much more epochs to converge. Agent with only data noise included (yellow curve) find it hard to

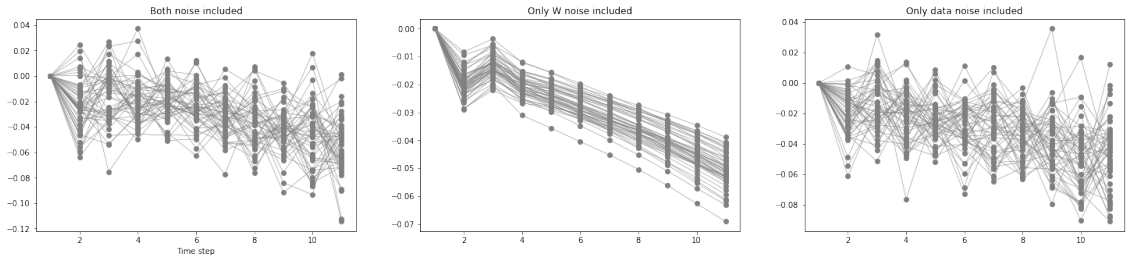


Figure 5.9: State exploration during planning by imagination of a LLB-LSTM model with various uncertainty estimation scheme. **Left**: Both $q(\mathbf{W}_{LLB})$ and data noise are considered; (b) only $q(\mathbf{W}_{LLB})$ is considered and (c) only data noise is included. The trajectories generated by weight uncertainty only tends to more smooth and less chaotic than other schemes. This is because the value of $\mathbf{W}_{LLB}$ is kept fixed for all time step after it has been sampled and the prediction is not perturbed randomly by data noise.

5.3.1 Learning-Planning iterations

We also tested different uncertainty schemes under a normal learning-planning iterations (Algorithm 3). Different from previous setting, we train independently three LLB-LSTM model with all noise included, but plan differently. We also reduce the number of sampled particles to only 100. This is to highlight the quality of each single particle. The results are shown in Fig 5.10. Model plans with weight uncertainty only gives faster convergence at the early stage but also experience slightly higher variance across experiment repetitions. This might be due to the low scale of weight noise as suggested by Fig 5.9.

Meanwhile, even though previous experiment suggests that a LLB-LSTM model finds it hard to plan with only data noise included when the number of training data is very limited. It still manage to solve the task under a learning-planning setting within 20 iterations.

In summary, we have shown that in a limited data setting, model uncertainty can perform more efficient policy search compared to data noise under a noise-free environment. And during normal Learning-Planning iterations, weight uncertainty can speed up the convergence. However, it is true that a more rigorous conclusion of this demands more large-scale analysis in various environment and we leave it for future work.

In next section we apply model on a stochastic cart-pole environment which is more difficult to solve than a standard cart-pole.

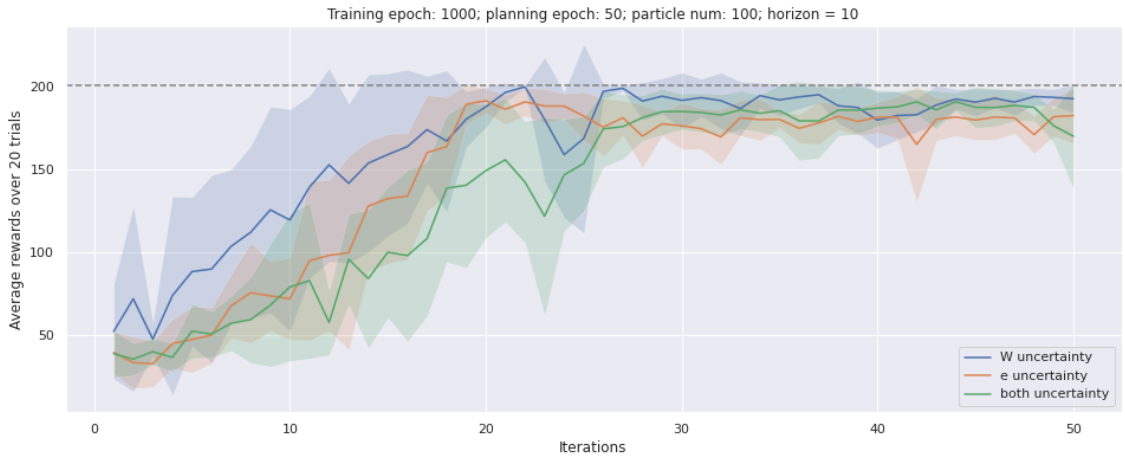


Figure 5.10: Shown above are the average testing rewards of three independent LLB-LSTM models with different uncertainty estimate schemes during planning: only weight noise included (**blue**); only data noise included (**orange**) and both noise included (**green**). The shaded area is the standard deviation over 6 experiment repititions.

5.4 On stochastic cart-pole balancing task

In a deterministic cart-pole balancing task we discover that a probabilistic dynamic model has more robust and stable policy learning. To further test the robustness of the models, we experimented with stochastic cart-pole balancing task where we add a Gaussian noise to the underlying physical model parameters and the observations as well.

There are multiple ways to setup a stochastic cart-pole environment, such as varying the length of a pole [22] or introduce actions delay [37]. In our experiment setting, as described by equation 3.5, a Gaussian noise with $\sigma = 0.1$ is applied on the force such that it varies across different episodes. This is in order to emphasise the significance of model uncertainty which two baseline models do not possess. Also a Gaussian noise $\sigma = 0.1$ is applied on the observation returned from the simulation at each time step.

The experiment also follows the learning-planning iteration (Algorithm 3) and the parameters of the algorithm are the same as before. Fig 5.11 shows the change in average testing rewards for three models. Among which LLB-LSTM is able to converge within 20 iterations and after that the variance of testing rewards across multiple attempts is kept at the lowest level, while it takes more iteration for a RSSM-LSTM to converge and the variance is comparatively higher. A DLSTM model gives the worse performance and the variance grows as it tries to achieve higher testing rewards. This implies that a LLB model can better accommodate the model and observation noise in the environment.

One potential reason for this might be related to the the ability to maintain stochasticity in the model prediction in order to accommodate the noise exist in the dynamics. For example, Fig 5.12 shows that during the learning-planning iterations the average standard deviation of prediction on unseen data in a LLB-LSTM model is larger than a RSSM-LSTM, especially in the position $\mathbf{x}$ and angle θ where the cumulative cost is computed from.

To verify this, we first deliver a comparison experiment on model learning performance between LLB-LSTM and RSSM-LSTM.

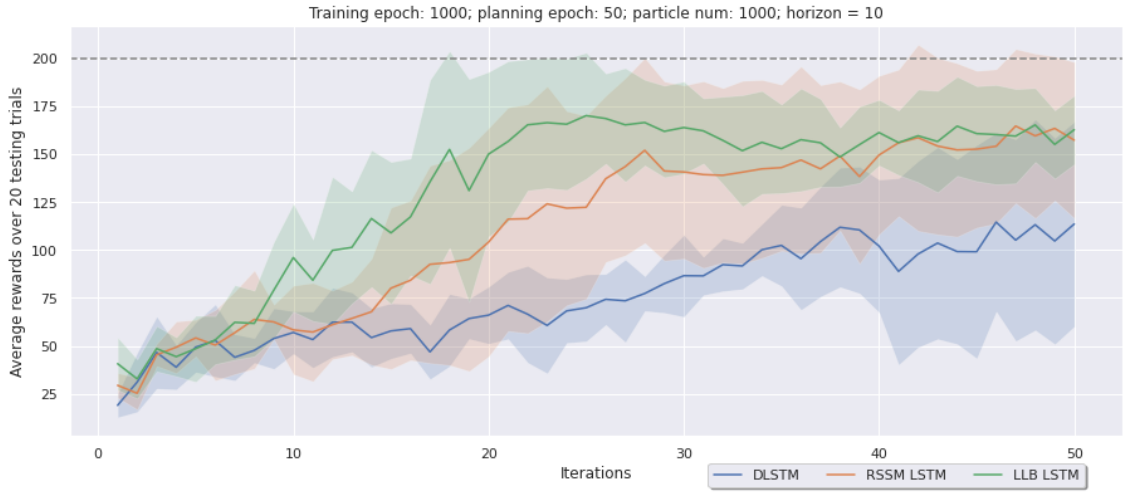


Figure 5.11: The curve represent the mean of average testing rewards over 20 trails and the shaded area represents the standard deviation across 6 independent experiment repetitions. The experiment is done under a stochastic cart-pole balancing environment in a Learning-Planning iteration setting.

Figure 5.12: The change of average standard deviation (over 20 testing data) of prediction during learning-planning iterations between RSSM-LSTM (**blue**) and LLB-LSTM (**red**) models. We particularly focus on position and angle plots since the expected cumulative cost is calculated only with these two quantities.

5.4.1 Model learning comparison between LLB and RSSM

We first initialise both model using the same random seed. After which the models are trained on same set of noisy cart-pole data with size 10 and trajectory length 10. Fig 5.13 shows the average log standard deviation of the prediction on both training (solid curve) and testing data (dashed curve). We can observe that RSSM-LSTM model tends to more rapidly reduce the stochsticity in its prediction as the model learning proceeds. A more clear illustration of uncertainty estimate in prediction can be found in Fig 5.14, which plots the prediction on a certain testing trajectory

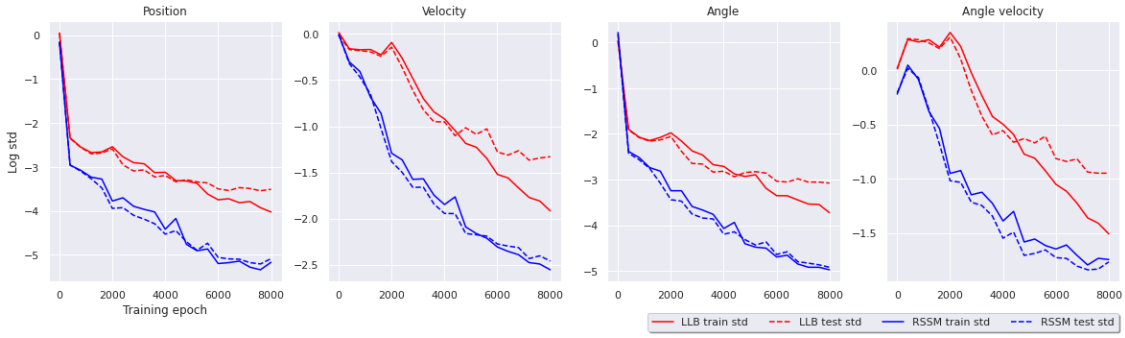


Figure 5.13: Shown above are the change of average log std (over trajectory time step) of the model prediction on both training and testing data during model learning iterations. **Solid line** represent the average std on training trajectories and **dashed line** represents on testing data. Color **red** and **blue** represents LLB-LSTM and RSSM-LSTM models respectively.

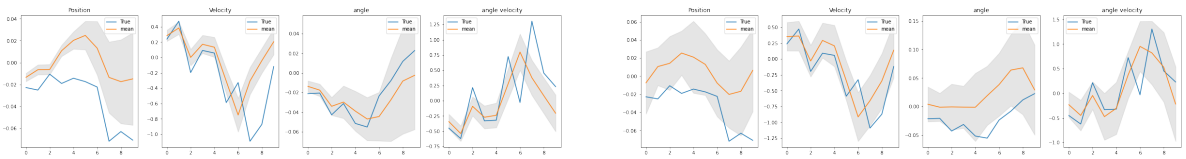
iteratively during model learning. This may implies that a RSSM fails to capture the model noise in the environment and becomes easily over-fitting.

The consequence of such a quick drop in prediction uncertainty is the high variance during policy learning and will be discussed in next section.

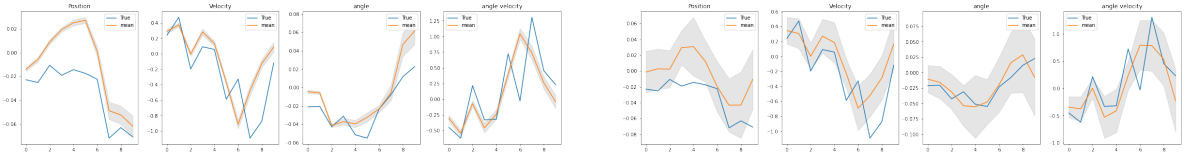
5.4.2 Policy learning comparison between LLB and RSSM

In previous experiment on noise-free environment we have already discovered that maintaining certainty amount of stochasticity can gain robustness in policy learning. Fig 5.15 suggests that this is also the case when the environment is stochastic. We train different copies of RSSM-LSTM and LLB-LSTM models for different number of iterations. Then we fix the model and perform policy learning along with policy testing. The plots demonstrate how policy learning generalises on new trajectories under the current model. As the training iteration number increases, the prediction from a RSSM becomes more deterministic (according to Fig 5.14) and therefore the policy learning conducts on current model has bad generalisation on long-term simulated trajectories. Whereas a LLB model still manage to give stable and robust policy training.

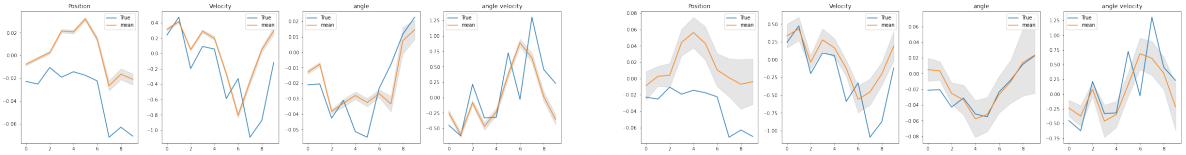
In summary, we discover that a LLB-LSTM model which estimates model uncertainty, compared to a model that does not, is able to provide more stable model learning on a stochastic environment where the underlying physical model parameters and the measurement of the observations are varying. This consequently offers robustness to policy learning which in turns results in better planning performance.



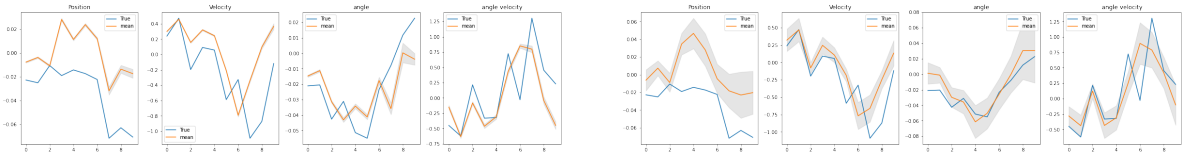
(a) 2000 training epoch



(b) 4000 training epoch

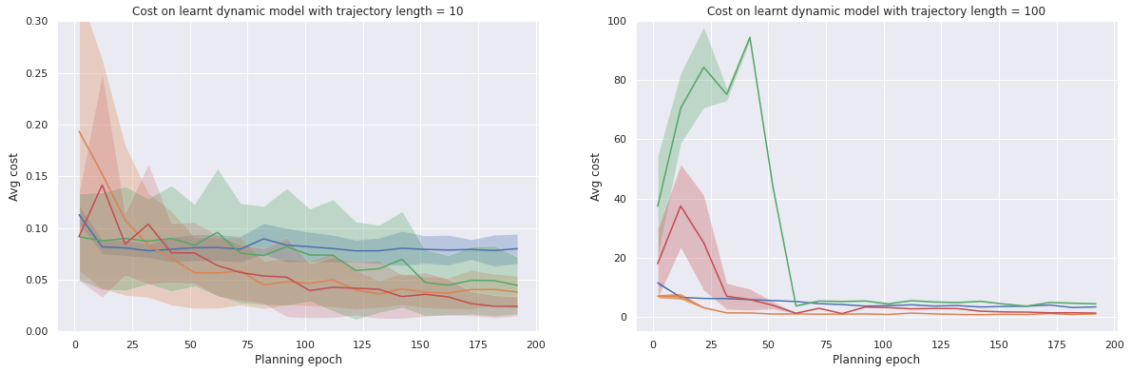


(c) 6000 training epoch

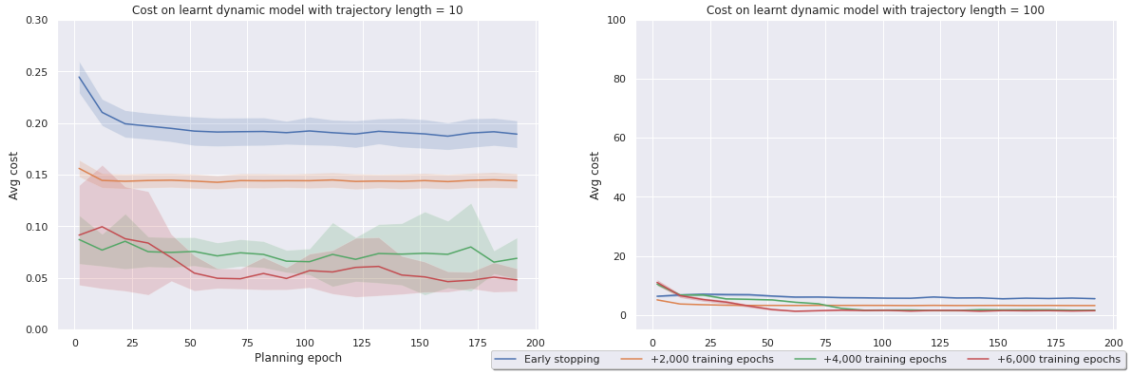


(d) 8000 training epoch

Figure 5.14: Shown above are the prediction on unseen trajectory under a noisy cart-pole environment. The **left** panel shows the prediction of a RSSM-LSTM model and the **right** panel shows the prediction of a LLB-LSTM model. As the model training proceed, the prediction from RSSM-LSTM becomes quickly deterministic. Whereas LLB-LSTM is able to maintain an error margin in its prediction and the mean prediction gives comparatively a better match.



(a) RSSM-LSTM



(b) LLB-LSTM

Figure 5.15: Shown above are the testing of the current policy on current learnt model with unseen initial observations at each iteration of policy learning. For each model: RSSM-LSTM (a) and LLB-LSTM (b), we train independently four copies of it according to different stopping criteria. The **blue** curve represents that we train the model with early-stopping which means the training terminate when it converge or start to show signs of over-fitting on validation set. The rest of color represents keeping on training for $+2,000$ (**orange**), $+4,000$ (**green**) and $+6,000$ (**red**) more iterations. The early-stopping happens around 2,300 iterations on RSSM-LSTM and 2,500 on LLB-LSTM. **(a)**: keep training the RSSM-LSTM model leads to large variance and terrible generalisation of the policy on new trajectories under the current learnt model. This is due to the over-fitting of RSSM. **(b)**: LLB-LSTM's policy learning shows better robustness and stability.

5.5 Computational consideration

The experiments are done on a Tesla V100 GPU and the running time for model learning is listed as below:

	Hidden dimension=32	**Hidden dimension**=64	**Hidden dimension**=256
DLSTM	38 s	39 s	40 s
RSSM-LSTM	256 s	260 s	264 s
LLB-LSTM	170 s	177 s	176 s

Table 5.1: Computation time on 50 trajectories with length 10 and batch size 10 for 5,000 training iterations

the running time for Learning-Planning iterations with the parameter setting same as described in the report is listed as:

	Learning-Planning iterations
DLSTM	1440 s
RSSM-LSTM	2983
LLB-LSTM	2934 s

Table 5.2: Computation time on Learning-Planning for 50 iterations

The model training of RSSM require more computation time since KL divergence is computed at each time step whereas in LLB model the KL is only computed once for each trajectory. However, the planning of LLB require more computational resources since the sampling procedure happens on both weight and data distribution.

Chapter 6

Conclusion and future work

In this thesis, a last-layer Bayesian recurrent neural network is proposed which places distribution on the last layer of the recurrent cell only. We successfully trained the model using variational approximation and tested it on various choice of model structure. We have shown that adding a Bayesian last layer does not deteriorate the model learning much on deterministic control data and a LLB recurrent network is able to capture the noise when the environment becomes stochastic. In the face of uncertainty, when the dynamic environment suffers from both systematic and measurement noise, a LLB recurrent network which models both aleatoric(data) and epistemic(model) uncertainty is shown to have more stable learning and better uncertain estimate in model prediction when compared to baseline models that estimate no epistemic uncertainty (such as RSSM).

In the field of gradient-based planning, we verify that a probabilistic dynamic model is good even for a deterministic control environment. Particularly, having stochasticity in the model provides robustness and stability for planning. In addition, we discover that with the inclusion of model uncertainty, such robustness is enhanced. In a limited data setting where the model has access to only small amount of training data, model uncertainty in a LLB model tends to provide more efficient policy search than random perturbed noise in the observation space.

We also setup a simple stochastic control environment with both systematic and measurement noisy included. We have shown that a LLB model outperforms a recurrent state-space model due to better uncertainty estimate provided by distribution on model parameters.

In conclusion, we implemented a partially-Bayeisan recurrent neural and have shown that the incorporation of model uncertainty is beneficial to gradient-based planning on simple control task for its robustness and more efficient exploration.

The potential extensions of this work are multiple. From the model side, for example, adding a Bayesian last layer weight to a recurrent neural network is a flexible way to introduce probabilistic interpretation to a deep learning model. However, we have only tested the model in a low-scale task and it it interesting to see how such model behaves when the task is more complex and high-dimensional.

From the planning side, it is obviously clear that our experiment needs to be done on a more complex task. Another research area of great potential is the curiosity-driven exploration, which means a probabilistic model tries to explore new states by leveraging the model uncertainty. For instance, previous works have applied Bayesian neural network (BNN) in a sparse-rewards control

task [39, 64]. A sparse reward implies that one cannot take direct gradient of it with respect to the policy parameter. One way to get around it is by using REINFORCE trick as we discussed in Section 2.3.2. Another method is by introducing an extra term in the reward function which is named *intrinsic reward* [39]. It has the form of a KL divergence between the current posterior distribution on model parameter $p(\mathbf{W}|\mathbf{o}_{1:t}, a_{1:t-1})$ and the modified posterior distribution after taking action a_t and observing next observation $\mathbf{o}_{t+1}$, denoted as:

$$r^{'} = r^{sparse} + \mathbf{KL}\Big[p(\mathbf{W}|\mathbf{o}_{1:t}, a_{1:t-1}, a_t, \mathbf{o}_{t+1})\Big\|p(\mathbf{W}|\mathbf{o}_{1:t}, a_{1:t-1})\Big]$$

then the objective is to take action a_t that maximise the new reward function $r^{'}$. The KL term is also interpreted as *information gain* and therefore the taken action is suppose to lead to a state that is maximally informative about the model. This method can naturally be applied on a Bayesian recurrent neural network with proper approximation method applied in place. Furthermore, in our experiment the model uncertainty is leveraged to explore in state space whereas curiosity-driven exploration is one way to largely utilise the model uncertainty in action space.

Another method to leverage model uncertainty is conditioning the policy on hidden state instead of on observations. This is inspired by *Dreamer* [35] (with RSSM as dynamic model) which is the current state-of-the-art in numerous control tasks. By conditioning on the hidden state the action is perturbed only by the uncertainty in weight parameter and this might have great potential benefits as we have shown that the weight uncertainty provide more sufficient exploration.

Bibliography

[1] C. G. Atkeson and J. C. Santamaria. A comparison of direct and model-based reinforcement learning. In *Proceedings of international conference on robotics and automation*, volume 4, pages 3557–3564. IEEE, 1997.

[2] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[3] D. Barber. *Bayesian reasoning and machine learning*. Cambridge University Press, 2012.

[4] S. Barratt. A matrix gaussian distribution, 2018.

[5] D. M. Bates and D. G. Watts. *Nonlinear regression analysis and its applications*, volume 2. Wiley New York, 1988.

[6] J. Bayer and C. Osendorfer. Learning stochastic recurrent networks. *arXiv preprint arXiv:1411.7610*, 2014.

[7] E. Begoli, T. Bhattacharya, and D. Kusnezov. The need for uncertainty quantification in machine-assisted medical decision making. *Nature Machine Intelligence*, 1(1):20–23, 2019.

[8] M. Benatan and E. O. Pyzer-Knapp. Fully bayesian recurrent neural networks for safe reinforcement learning. *arXiv preprint arXiv:1911.03308*, 2019.

[9] Y. Bengio, P. Frasconi, and P. Simard. The problem of learning long-term dependencies in recurrent networks. In *IEEE international conference on neural networks*, pages 1183–1188. IEEE, 1993.

[10] C. M. Bishop. *Pattern recognition and machine learning*. springer, 2006.

[11] C. Blundell, J. Cornebise, K. Kavukcuoglu, and D. Wierstra. Weight uncertainty in neural networks. *arXiv preprint arXiv:1505.05424*, 2015.

[12] W. M. Bolstad and J. M. Curran. *Introduction to Bayesian statistics*. John Wiley & Sons, 2016.

[13] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[14] K. Chua, R. Calandra, R. McAllister, and S. Levine. Deep reinforcement learning in a handful of trials using probabilistic dynamics models. In *Advances in Neural Information Processing Systems*, pages 4754–4765, 2018.

[15] J. Chung, K. Kastner, L. Dinh, K. Goel, A. C. Courville, and Y. Bengio. A recurrent latent variable model for sequential data. In *Advances in neural information processing systems*, pages 2980–2988, 2015.

[16] M. Defferrard, X. Bresson, and P. Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in neural information processing systems*, pages 3844–3852, 2016.

[17] M. Deisenroth and C. E. Rasmussen. Pilco: A model-based and data-efficient approach to policy search. In *Proceedings of the 28th International Conference on machine learning (ICML-11)*, pages 465–472, 2011.

[18] M. P. Deisenroth, D. Fox, and C. E. Rasmussen. Gaussian processes for data-efficient learning in robotics and control. *IEEE transactions on pattern analysis and machine intelligence*, 37(2):408–423, 2013.

[19] M. P. Deisenroth, G. Neumann, and J. Peters. *A survey on policy search for robotics*. now publishers, 2013.

[20] S. Depeweg, J. M. Hernández-Lobato, F. Doshi-Velez, and S. Udluft. Learning and policy search in stochastic dynamical systems with bayesian neural networks. *arXiv preprint arXiv:1605.07127*, 2016.

[21] P. M. Djuric, J. H. Kotecha, J. Zhang, Y. Huang, T. Ghirmai, M. F. Bugallo, and J. Miguez. Particle filtering. *IEEE signal processing magazine*, 20(5):19–38, 2003.

[22] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel. Benchmarking deep reinforcement learning for continuous control. In *International Conference on Machine Learning*, pages 1329–1338, 2016.

[23] M. Fortunato, M. G. Azar, B. Piot, J. Menick, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin, et al. Noisy networks for exploration. *arXiv preprint arXiv:1706.10295*, 2017.

[24] M. Fortunato, C. Blundell, and O. Vinyals. Bayesian recurrent neural networks. *arXiv preprint arXiv:1704.02798*, 2017.

[25] M. Fraccaro, S. K. Sønderby, U. Paquet, and O. Winther. Sequential neural models with stochastic layers. In *Advances in neural information processing systems*, pages 2199–2207, 2016.

[26] Y. Gal. Uncertainty in deep learning. *University of Cambridge*, 1(3), 2016.

[27] Y. Gal and Z. Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*, pages 1050–1059, 2016.

[28] Y. Gal and Z. Ghahramani. A theoretically grounded application of dropout in recurrent neural networks. In *Advances in neural information processing systems*, pages 1019–1027, 2016.

[29] Y. Gal, R. McAllister, and C. E. Rasmussen. Improving pilco with bayesian neural network dynamics models. In *Data-Efficient Machine Learning workshop, ICML*, volume 4, page 34, 2016.

[30] C. F. Gauss. *Theoria motus corporum coelestium in sectionibus conicis solem ambientium*, volume 7. Perthes et Besser, 1809.

[31] A. G. A. P. Goyal, A. Sordoni, M.-A. Côté, N. R. Ke, and Y. Bengio. Z-forcing: Training stochastic recurrent networks. In *Advances in neural information processing systems*, pages 6713–6723, 2017.

[32] A. Graves. Practical variational inference for neural networks. In *Advances in neural information processing systems*, pages 2348–2356, 2011.

[33] A. Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

[34] K. Gregor, G. Papamakarios, F. Besse, L. Buesing, and T. Weber. Temporal difference variational auto-encoder. *arXiv preprint arXiv:1806.03107*, 2018.

[35] D. Hafner, T. Lillicrap, J. Ba, and M. Norouzi. Dream to control: Learning behaviors by latent imagination. *arXiv preprint arXiv:1912.01603*, 2019.

[36] D. Hafner, T. Lillicrap, I. Fischer, R. Villegas, D. Ha, H. Lee, and J. Davidson. Learning latent dynamics for planning from pixels. In *International Conference on Machine Learning*, pages 2555–2565, 2019.

[37] T. Hester and P. Stone. The open-source texplore code release for reinforcement learning on robots. In *Robot Soccer World Cup*, pages 536–543. Springer, 2013.

[38] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[39] R. Houthooft, X. Chen, Y. Duan, J. Schulman, F. De Turck, and P. Abbeel. Vime: Variational information maximizing exploration. In *Advances in Neural Information Processing Systems*, pages 1109–1117, 2016.

[40] Y. H. Hu, W. J. Tompkins, J. L. Urrusti, and V. X. Afonso. Applications of artificial neural networks for ecg signal detection and classification. *Journal of electrocardiology*, 26:66–73, 1993.

[41] M. Janner, J. Fu, M. Zhang, and S. Levine. When to trust your model: Model-based policy optimization. In *Advances in Neural Information Processing Systems*, pages 12519–12530, 2019.

[42] N. R. Ke, A. Singh, A. Touati, A. Goyal, Y. Bengio, D. Parikh, and D. Batra. Learning dynamics model in reinforcement learning by incorporating the long term future. *arXiv preprint arXiv:1903.01599*, 2019.

[43] D. P. Kingma. Stochastic gradient vb and the variational auto-encoder.

[44] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[45] G. Kitagawa and W. Gersch. Linear gaussian state space modeling. In *Smoothness priors analysis of time series*, pages 55–65. Springer, 1996.

[46] S. Kullback. *Information theory and statistics*. Courier Corporation, 1997.

[47] B. Lakshminarayanan, A. Pritzel, and C. Blundell. Simple and scalable predictive uncertainty estimation using deep ensembles. In *Advances in neural information processing systems*, pages 6402–6413, 2017.

[48] D. J. MacKay. A practical bayesian framework for backpropagation networks. *Neural computation*, 4(3):448–472, 1992.

[49] F. Murtagh. Multilayer perceptrons for classification and regression. *Neurocomputing*, 2(5-6):183–197, 1991.

[50] R. M. Neal and G. E. Hinton. A view of the em algorithm that justifies incremental, sparse, and other variants. In *Learning in graphical models*, pages 355–368. Springer, 1998.

[51] I. Osband, C. Blundell, A. Pritzel, and B. Van Roy. Deep exploration via bootstrapped dqn. In *Advances in neural information processing systems*, pages 4026–4034, 2016.

[52] P. Parmas, C. E. Rasmussen, J. Peters, and K. Doya. Pipps: Flexible model-based policy search robust to the curse of chaos. In *International Conference on Machine Learning*, pages 4065–4074, 2018.

[53] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.

[54] M. Plappert, R. Houthooft, P. Dhariwal, S. Sidor, R. Y. Chen, X. Chen, T. Asfour, P. Abbeel, and M. Andrychowicz. Parameter space noise for exploration. *arXiv preprint arXiv:1706.01905*, 2017.

[55] D. A. Pomerleau. Alvinn: An autonomous land vehicle in a neural network. In *Advances in neural information processing systems*, pages 305–313, 1989.

[56] C. E. Rasmussen. Gaussian processes in machine learning. In *Summer School on Machine Learning*, pages 63–71. Springer, 2003.

[57] A. G. Richards. *Robust constrained model predictive control*. PhD thesis, Massachusetts Institute of Technology, 2005.

[58] C. Riquelme, G. Tucker, and J. Snoek. Deep bayesian bandits showdown: An empirical comparison of bayesian deep networks for thompson sampling. *arXiv preprint arXiv:1802.09127*, 2018.

[59] R. Y. Rubinstein. Optimization of computer simulation models with rare events. *European Journal of Operational Research*, 99(1):89–112, 1997.

[60] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[61] S. Schaal. Learning from demonstration. In *Advances in neural information processing systems*, pages 1040–1046, 1997.

[62] A. M. Schäfer. Reinforcement learning with recurrent neural networks. 2008.

[63] J. G. Schneider. Exploiting model uncertainty estimates for safe dynamic control learning. In *Advances in neural information processing systems*, pages 1047–1053, 1997.

[64] P. Shyam, W. Jaśkowski, and F. Gomez. Model-based active exploration. In *International Conference on Machine Learning*, pages 5779–5788, 2019.

[65] A. Smith. *Sequential Monte Carlo methods in practice*. Springer Science & Business Media, 2013.

[66] M. Stone. Cross-validatory choice and assessment of statistical predictions. *Journal of the Royal Statistical Society: Series B (Methodological)*, 36(2):111–133, 1974.

[67] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[68] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[69] I. V. Tetko, D. J. Livingstone, and A. I. Luik. Neural network studies. 1. comparison of overfitting and overtraining. *Journal of chemical information and computer sciences*, 35(5):826–833, 1995.

[70] A. Varga and R. Moore. Hidden markov model decomposition of speech and noise. In *International Conference on Acoustics, Speech, and Signal Processing*, pages 845–848. IEEE, 1990.

[71] M. Vladimirova, J. Verbeek, P. Mesejo, and J. Arbel. Understanding priors in bayesian neural networks at the unit level. In *International Conference on Machine Learning*, pages 6458–6467, 2019.

[72] T. Wang and J. Ba. Exploring model-based planning with policy networks. *arXiv preprint arXiv:1906.08649*, 2019.

[73] N. Weber, J. Starc, A. Mittal, R. Blanco, and L. Màrquez. Optimizing over a bayesian last layer. In *NeurIPS workshop on Bayesian Deep Learning*, 2018.

[74] P. J. Werbos. Generalization of backpropagation with application to a recurrent gas market model. *Neural networks*, 1(4):339–356, 1988.

[75] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[76] H. Yan, Y. Jiang, J. Zheng, C. Peng, and Q. Li. A multilayer perceptron-based medical decision support system for heart disease diagnosis. *Expert Systems with Applications*, 30(2):272–281, 2006.

[77] X. Yang. Understanding the variational lower bound, 2017.

[78] W. Zaremba, I. Sutskever, and O. Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.

Appendix A

Appendix

A.1 KL divergence between two uni-Gaussian distribution

Assume for random variable W we have two Gaussian distribution $q(W), p(W)$ with sufficient statistics denoted respectively as $[\mu_q, \sigma_q^2], [\mu_p, \sigma_p^2]$, then the KL divergence between $q(W)$ and $p(W)$ is derived as below:

$$\mathbf{KL}\Big(q(W)\Big\|p(W)\Big) = \int q\ log\ q\ dW\ -\ \int q\ log\ p\ dW$$

$$\begin{aligned}
\int q\ log\ q\ dW &= \int \mathcal{N}(\mu_q, \sigma_q^2)\Big[-\frac{1}{2}log(2\pi\sigma_q^2) - \frac{1}{2\sigma_q^2}\big(W-\mu_q\big)^2\Big]\ dW \\
&= -\frac{1}{2}log(2\pi\sigma_q^2) - \frac{1}{2\sigma_q^2}\int q(W)\big[W^2 - 2W\mu_q + \mu_q^2\big]\ dW \\
&= -\frac{1}{2}log(2\pi\sigma_q^2) - \frac{1}{2\sigma_q^2}\Big[\big\langle W^2\big\rangle_{q(W)} - 2\mu_q\big\langle W\big\rangle_{q(W)} + \mu_q^2\Big] \\
&= -\frac{1}{2}log(2\pi\sigma_q^2) - \frac{1}{2\sigma_q^2}\Big[\sigma_q^2 + \mu_q^2 - 2\mu_q^2 + \mu_q^2\Big] \\
&= -\frac{1}{2}log(2\pi\sigma_q^2) - \frac{1}{2}
\end{aligned}$$

$$\begin{aligned}
\int q\ log\ p\ dW &= \int q(W)\Big[-\frac{1}{2}log(2\pi\sigma_p^2) - \frac{1}{2\sigma_p^2}\big(W-\mu_p\big)^2\Big]\ dW \\
&= -\frac{1}{2}log(2\pi\sigma_p^2) - \frac{1}{2\sigma_p^2}\Big[\big\langle \mathbf{h}^2\big\rangle_{q(W)} - 2\mu_p\big\langle \mathbf{h}\big\rangle_{q(W)} + \mu_p^2\Big]\ dW \\
&= -\frac{1}{2}log(2\pi\sigma_p^2) - \frac{1}{2\sigma_p^2}\Big[\sigma_q^2 + \mu_q^2 - 2\mu_p\mu_q + \mu_p^2\Big] \\
&= -\frac{1}{2}log(2\pi\sigma_p^2) - \frac{1}{2\sigma_p^2}\Big[\sigma_q^2 + (\mu_q - \mu_p)^2\Big]
\end{aligned}$$

$$\mathbf{KL}\Big(q(W)\Big\|p(W)\Big) = \frac{1}{2}log\left(\frac{\sigma_p^2}{\sigma_q^2}\right) + \frac{1}{2\sigma_p^2}\Big[\sigma_q^2 + (\mu_q - \mu_p)^2\Big] - \frac{1}{2}$$