Function Autoencoder: A Neural Network Approach to Gaussian Processes

Xiangyu Zhao
Trinity College
xz398@cam.ac.uk

Abstract

Gaussian processes (GPs) are data-efficient and flexible probabilistic methods that learn distributions of functions based on given priors. However, GPs suffer from unscalability as they become very computationally expensive on large datasets, and choosing the appropriate priors for GPs can be nontrivial. In this project, I investigated a neural network (NN) alternative to GPs, and introduced the *function autoencoders* that preserve GPs' own advantages and avoid their weaknesses with NNs' benefits. I tested the performance of the various function autoencoders on a 1-dimensional function regression task, and compared and analysed their results. The trained function autoencoder models indeed have the ability to learn distributions over random functions, and performed decently on the selected task. Moreover, the function autoencoders demonstrate a great potential for further improvements.

1 Introduction

Function approximation sits at the core of most machine learning problems, and in some cases modelling the uncertainties of the approximations, in addition to predicting the values of the functions, can also be highly desirable. This requires the suitable machine learning methods to be able to not only learn the function instances, but also model the distributions over random functions. A popular approach for this task is to perform inference on a stochastic process, in which Gaussian processes (GPs) are the most common instantiation [1]. GPs are data efficient as it can carry out Bayesian inference about underlying ground truth function conditioned on only a handful of observations, and is very flexible at test time. In addition, GPs can also represent infinitely many different functions at the locations that has not been observed, which allows them to capture the uncertainty of the predictions. However, GPs are computationally expensive and scale cubically with respect to the number of datapoints, and it can be very difficult to design appropriate priors in practice.

On the other hand, a neural network (NN) can learn to parameterise a single function without the need of being taught with an appropriate prior, but requires a large number of training data and cannot model distributions over functions. Knowing how to combine the benefits of both GPs and NNs therefore becomes very desirable. Previous works on this task include Conditional Neural Processes (CNPs) [2] and Neural Processes (NPs) [3], with the latter improving the former by adding a latent variable to the original deterministic representation. In this project, I shall build upon the NPs and CNPs and design a neural network approach to GPs, in an autoencoder style, called the *function autoencoders*.

2 Method

2.1 Stochastic process interpretation

A stochastic process is a collection of random variables indexed by some parameters. When defining a stochastic process, a standard approach is to define its finite-dimensional marginal distributions. In

this specific case, the stochastic process is a random function $F : \mathcal{X} \to \mathcal{Y}$, where F is sampled from a distribution $\mathcal{F}$. For each finite sequence $\boldsymbol{x}_{[1:n]} = (x_1, \cdots, x_n) \in \mathcal{X}^n$, the marginal joint distributions over the function values can be defined as $\boldsymbol{Y}_{[1:n]} = (Y_1, \cdots, Y_n) \sim (F(x_1), \cdots, F(x_n))$. For example, in the case of Gaussian processes (GPs), the joint distributions are multivariate Gaussians parameterised by a mean and covariance function. According to the Kolmogorov extension theorem [4], these joint distributions need to have the following properties:

- *Exchangeability*: the joint distributions need to be permutation invariant. More precisely, for all permutations π on $\{1, \cdots, n\}$, where $n \in \mathbb{N}$,

$$\Pr_Y(y_1, \cdots, y_n | x_1, \cdots, x_n) = \Pr_Y(y_{\pi(1)}, \cdots, y_{\pi(n)} | x_{\pi(1)}, \cdots, x_{\pi(n)}) \tag{1}$$

- *Consistency*: if a part of the sequence is marginalised, the resulting marginal distribution needs to be the same as the original sequence. In mathematical terms, if $1 \leq m \leq n$, then

$$\Pr_Y(\boldsymbol{y}_{[1:m]} | \boldsymbol{x}_{[1:m]}) = \int_{\boldsymbol{Y}_{[m+1:n]}} \Pr_Y(\boldsymbol{y}_{[1:n]} | \boldsymbol{x}_{[1:n]}) d\boldsymbol{Y}_{[m+1:n]} \tag{2}$$

Therefore, given a particular instantiation of the function F_i, the joint distribution is defined as:

$$\Pr_{Y_i}(\boldsymbol{y}_i | \boldsymbol{x}_i) = \int_F \Pr_F(F_i) \Pr_{Y_i}(\boldsymbol{y}_i | F_i, \boldsymbol{x}_i) dF \tag{3}$$

If we assume a Gaussian noise for the observations, i.e., $Y \sim \mathcal{N}(F(x), \sigma^2)$, then

$$\Pr_{Y_i}(\boldsymbol{y}_i | F_i, \boldsymbol{x}_i) = \prod_{j=1}^{n_i} \mathcal{N}(y_{ij} | F_i(x_{ij}), \sigma^2) \tag{4}$$

Inserting this into Equation (3), we can specify the joint distribution as

$$\Pr_{Y_i}(\boldsymbol{y}_i | \boldsymbol{x}_i) = \int_F \Pr_F(F_i) \prod_{j=1}^{n_i} \mathcal{N}(y_{ij} | F_i(x_{ij}), \sigma^2) dF \tag{5}$$

This means that the observations $Y_1, \cdots, Y_n$ are conditionally independent given the function F. Therefore, as long as we can approximate F with an NN, we will be able to represent such a stochastic process using NNs and find the marginal joint distributions.

2.2 Function autoencoders

Assume that the random function F can be written as $F(x) = f_\theta(x, Z)$, where f_θ is a deterministic, learnable function parametrised by θ, and Z is a latent variable (i.e., the randomness of F is provided by Z). Then, following the idea of variational autoencoders, we can assume Z to be a multivariate normal, and train an NN to fit f_θ in an autoencoder style. However, a typical NN trained on a dataset can only learn a single function, rather than a distribution over random functions. To learn such a distribution over random functions instead of a single function instance, it is essential to train the model using multiple datasets concurrently:

$$\mathcal{D} = \{(\boldsymbol{x}_1, \boldsymbol{y}_1), \cdots, (\boldsymbol{x}_k, \boldsymbol{y}_k)\}$$

where each dataset $(\boldsymbol{x}_i, \boldsymbol{y}_i)$ is a sequence of observed inputs $\boldsymbol{x}_i = [x_{i1} \ \cdots \ x_{in_i}]$ and outputs $\boldsymbol{y}_i = [y_{i1} \ \cdots \ y_{in_i}] = [F_i(x_{i1}) \ \cdots \ F_i(x_{in_i})]$ of an instance of the random functions $F_i \sim \mathcal{F}$. In this way, we can learn the variability of the random function from the variability of the datasets.

From what we know about a variational autoencoder, we can write down the log likelihood of a single dataset $(\boldsymbol{x}_i, \boldsymbol{y}_i)$ as a function of θ:

$$\begin{aligned}
\mathcal{L}_i(\theta) &= \log \Pr_{Y_i}(\boldsymbol{y}_i|\boldsymbol{x}_i;\theta) \\
&= \sum_j \log \Pr_{Y_i}(y_{ij}|x_{ij};\theta) && ((x_{ij}, y_{ij}) \text{ are independent samples}) \\
&= \sum_j \log \int_{z_i} \Pr_{Y_i}(y_{ij}|z_i, x_{ij};\theta) \Pr_{Z_i}(z_i) dz_i && \text{(law of total probability)} \\
&= \sum_j \log \mathbb{E}_{z_i \sim Z_i} \left[\Pr_{Y_i}(y_{ij}|z_i, x_{ij};\theta)\right] && \text{(writing integral as expectation)} \\
&= \sum_j \log \mathbb{E}_{z_i \sim \tilde{Z}_i} \left[\Pr_{Y_i}(y_{ij}|z_i, x_{ij};\theta) \frac{\Pr_{Z_i}(z_i)}{\Pr_{\tilde{Z}_i}(z_i)}\right] && \text{(importance sampling using } \tilde{Z}) \\
& && (6)
\end{aligned}$$

If we train an encoder $g_\phi(\boldsymbol{x}_i, \boldsymbol{y}_i)$ to estimate the latent variable $\tilde{Z}_i$, then the log likelihood can be written as:

$$\begin{aligned}
\mathcal{L}_i(\theta) &= \sum_j \log \mathbb{E}_{z_i \sim \tilde{Z}_i^{(\phi)}} \left[\Pr_{Y_i}(y_{ij}|z_i, x_{ij};\theta) \frac{\Pr_{Z_i}(z_i)}{\Pr_{\tilde{Z}_i^{(\phi)}}(z_i|\boldsymbol{x}_i, \boldsymbol{y}_i)}\right] \\
&\geq \sum_j \mathbb{E}_{z_i \sim \tilde{Z}_i^{(\phi)}} \left[\log \left(\Pr_{Y_i}(y_{ij}|z_i, x_{ij};\theta) \frac{\Pr_{Z_i}(z_i)}{\Pr_{\tilde{Z}_i^{(\phi)}}(z_i|\boldsymbol{x}_i, \boldsymbol{y}_i)}\right)\right] && \text{(Jensen's inequality)} \\
&= \mathcal{L}_{\text{lb}i}(\theta, \phi) && (7)
\end{aligned}$$

We can then use $\mathcal{L}_{\text{lb}i}(\theta, \phi)$ to define the lower bound log likelihood of the particular dataset $(\boldsymbol{x}_i, \boldsymbol{y}_i)$. Moreover, the log likelihood of the entire collection of the datasets $\mathcal{D}$ can be derived:

$$\begin{aligned}
\mathcal{L}(\theta) &= \log \Pr_{\boldsymbol{Y}}(\boldsymbol{y}_i, \cdots, \boldsymbol{y}_k|\boldsymbol{x}_i, \cdots, \boldsymbol{x}_k;\theta) \\
&= \sum_i \log \Pr_{Y_i}(\boldsymbol{y}_i|\boldsymbol{x}_i;\theta) = \sum_i \mathcal{L}_i(\theta) && ((\boldsymbol{x}_i, \boldsymbol{y}_i) \text{ are independent datasets}) \\
&= \sum_i \sum_j \log \mathbb{E}_{z_i \sim \tilde{Z}_i^{(\phi)}} \left[\Pr_{Y_i}(y_{ij}|z_i, x_{ij};\theta) \frac{\Pr_{Z_i}(z_i)}{\Pr_{\tilde{Z}_i^{(\phi)}}(z_i|\boldsymbol{x}_i, \boldsymbol{y}_i)}\right] \\
&\geq \sum_i \sum_j \mathbb{E}_{z_i \sim \tilde{Z}_i^{(\phi)}} \left[\log \left(\Pr_{Y_i}(y_{ij}|z_i, x_{ij};\theta) \frac{\Pr_{Z_i}(z_i)}{\Pr_{\tilde{Z}_i^{(\phi)}}(z_i|\boldsymbol{x}_i, \boldsymbol{y}_i)}\right)\right] && \text{(Jensen's inequality)} \\
&= \sum_i \mathcal{L}_{\text{lb}i}(\theta, \phi) = \mathcal{L}_{\text{lb}}(\theta, \phi) && (8)
\end{aligned}$$

Therefore the training goal of a function autoencoder becomes finding the optimal $\hat{\theta}$ and $\hat{\phi}$ that jointly maximises $\mathcal{L}_{\text{lb}}(\theta, \phi)$, the lower bound log likelihood of the entire collection of the datasets $\mathcal{D}$. Besides, at training time, $\mathcal{L}_{\text{lb}}(\theta, \phi)$ is computed using the Kullback-Leibler divergence:

$$\mathcal{L}_{\text{lb}i}(\theta, \phi) = \sum_j \left\{\mathbb{E}_{z_i \sim \tilde{Z}_i^{(\phi)}} \left[\log \Pr_{Y_i}(y_{ij}|z_i, x_{ij};\theta)\right] - \text{KL}\left(\Pr_{\tilde{Z}_i^{(\phi)}|\boldsymbol{x}_i, \boldsymbol{y}_i} || \Pr_{Z_i}\right)\right\} \tag{9}$$

$$\mathcal{L}_{\text{lb}}(\theta, \phi) = \sum_i \sum_j \left\{\mathbb{E}_{z_i \sim \tilde{Z}_i^{(\phi)}} \left[\log \Pr_{Y_i}(y_{ij}|z_i, x_{ij};\theta)\right] - \text{KL}\left(\Pr_{\tilde{Z}_i^{(\phi)}|\boldsymbol{x}_i, \boldsymbol{y}_i} || \Pr_{Z_i}\right)\right\} \tag{10}$$

which enables us to approximate $\mathcal{L}_{\text{lb}}(\theta, \phi)$ using Monte Carlo sampling, and makes it differentiable for the sake of gradient descent.

At test time, each dataset $(\boldsymbol{x}_{i[1:n_i]}, \boldsymbol{y}_{i[1:n_i]})$ is split into a training dataset containing the *context* points $(\boldsymbol{x}_{i[1:m_i]}, \boldsymbol{y}_{i[1:m_i]})$, and a holdout dataset containing the *target* points $(\boldsymbol{x}_{i[m_i+1:n_i]}, \boldsymbol{y}_{i[m_i+1:n_i]})$, for some m such that $1 \leq m \leq n$. The functional autoencoder is then required to model the conditional joint distributions of the target points, given the context points. Therefore, we can revise the training

goal to a conditional log likelihood of the holdout dataset given the training dataset, which reflects better of the model's desired behaviour at test time:

$$\begin{aligned}
\mathcal{L}_i(\theta) &= \log \Pr_{Y_i}(\boldsymbol{y}_{i[m_i+1:n_i]} | \boldsymbol{x}_{i[1:n_i]}, \boldsymbol{y}_{i[1:m_i]}; \theta) \\
&\geq \sum_{j=m_i+1}^{n_i} \mathbb{E}_{z_i \sim \tilde{Z}_i^{(\phi)}} \left[\log \left(\Pr_{Y_i}(y_{ij} | z_i, x_{ij}; \theta) \frac{\Pr_{Z_i}(z_i | \boldsymbol{x}_{i[1:m_i]}, \boldsymbol{y}_{i[1:m_i]})}{\Pr_{\tilde{Z}_i^{(\phi)}}(z_i | \boldsymbol{x}_{i[1:n_i]}, \boldsymbol{y}_{i[1:n_i]})} \right) \right] \\
&= \mathcal{L}_{\text{lb}i}(\theta, \phi) \qquad (11) \\
\mathcal{L}(\theta) &= \sum_{i=1}^{k} \log \Pr_{Y_i}(\boldsymbol{y}_{i[m_i+1:n_i]} | \boldsymbol{x}_{i[1:n_i]}, \boldsymbol{y}_{i[1:m_i]}; \theta) \\
&\geq \sum_{i=1}^{k} \sum_{j=m_i+1}^{n_i} \mathbb{E}_{z_i \sim \tilde{Z}_i^{(\phi)}} \left[\log \left(\Pr_{Y_i}(y_{ij} | z_i, x_{ij}; \theta) \frac{\Pr_{Z_i}(z_i | \boldsymbol{x}_{i[1:m_i]}, \boldsymbol{y}_{i[1:m_i]})}{\Pr_{\tilde{Z}_i^{(\phi)}}(z_i | \boldsymbol{x}_{i[1:n_i]}, \boldsymbol{y}_{i[1:n_i]})} \right) \right] \\
&= \mathcal{L}_{\text{lb}}(\theta, \phi) \qquad (12)
\end{aligned}$$

Since the perfect sampling distribution $\Pr_{Z_i}(z_i | \boldsymbol{x}_{i[1:m_i]}, \boldsymbol{y}_{i[1:m_i]})$ is intractable, we can approximate it using $\Pr_{\tilde{Z}_i^{(\phi)}}(z_i | \boldsymbol{x}_{i[1:m_i]}, \boldsymbol{y}_{i[1:m_i]})$:

$$\mathcal{L}_{\text{lb}i}(\theta, \phi) \approx \sum_{j=m_i+1}^{n_i} \mathbb{E}_{z_i \sim \tilde{Z}_i^{(\phi)}} \left[\log \left(\Pr_{Y_i}(y_{ij} | z_i, x_{ij}; \theta) \frac{\Pr_{\tilde{Z}_i^{(\phi)}}(z_i | \boldsymbol{x}_{i[1:m_i]}, \boldsymbol{y}_{i[1:m_i]})}{\Pr_{\tilde{Z}_i^{(\phi)}}(z_i | \boldsymbol{x}_{i[1:n_i]}, \boldsymbol{y}_{i[1:n_i]})} \right) \right] \qquad (13)$$

$$\mathcal{L}_{\text{lb}}(\theta, \phi) \approx \sum_{i=1}^{k} \sum_{j=m_i+1}^{n_i} \mathbb{E}_{z_i \sim \tilde{Z}_i^{(\phi)}} \left[\log \left(\Pr_{Y_i}(y_{ij} | z_i, x_{ij}; \theta) \frac{\Pr_{\tilde{Z}_i^{(\phi)}}(z_i | \boldsymbol{x}_{i[1:m_i]}, \boldsymbol{y}_{i[1:m_i]})}{\Pr_{\tilde{Z}_i^{(\phi)}}(z_i | \boldsymbol{x}_{i[1:n_i]}, \boldsymbol{y}_{i[1:n_i]})} \right) \right] \qquad (14)$$

In order to be able to predict accurately across the entire collection of datasets, the model needs to learn a distribution that covers all the functions observed during training, as well as being able to take into account the context data at test time.

2.3 Model structures

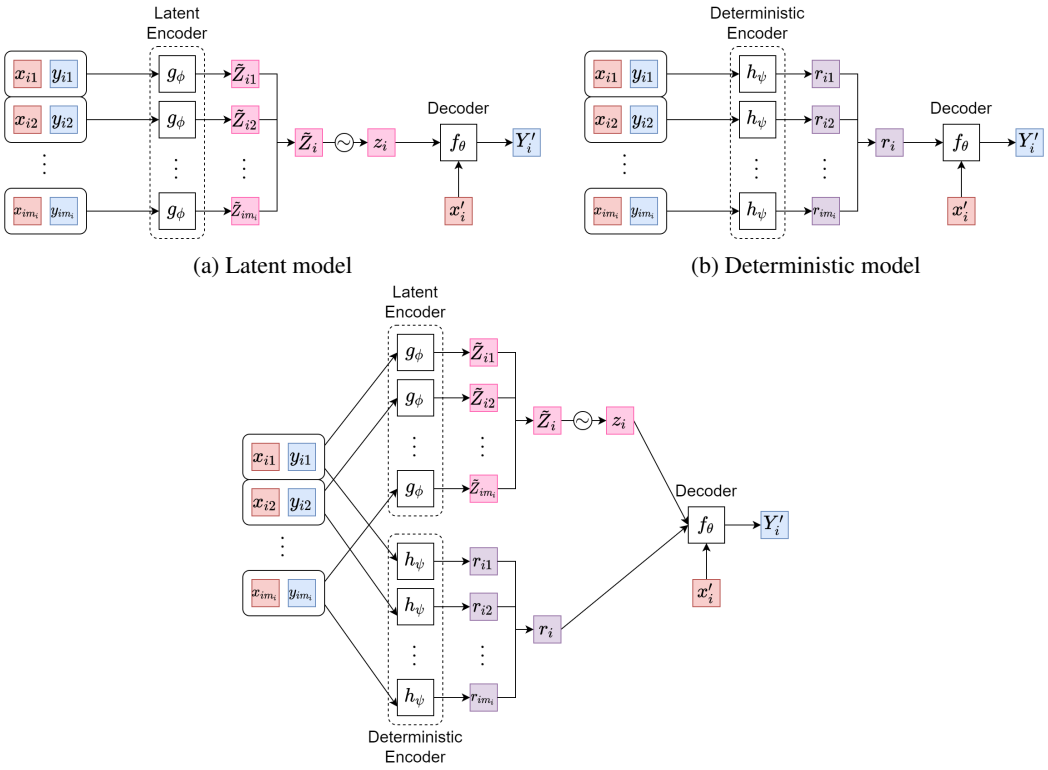
A basic function autoencoder implemented in this project consists of three core components:

- An NN *latent encoder* g_ϕ that, for every instance F_i of random functions $F_1, \cdots, F_k$, takes in one context point (x_{ij}, y_{ij}) at each time, and produces the parameters of a sampling distribution $\tilde{Z}_{ij}$. Since, in this project's case, the latent distribution Z is assumed to be a multivariate Gaussian, the parameters that $g_\phi(x_{ij}, y_{ij})$ should produce would be the mean μ_{ij} and standard deviation σ_{ij} of the multivariate Gaussian distribution $\tilde{Z}_{ij} \sim \text{MVN}(\mu_{ij}, \sigma_{ij}^2 I)$.
- An *aggregator* a that summarises the encoder's outputs, as we are interested in obtaining a single permutation-invariant global latent variable $\tilde{Z}_i \sim \text{MVN}(\mu_i, \sigma_i^2 I)$ for each random function instance F_i. The simplest operation that ensures permutation invariance and works well in practice is the mean function:

  $$\mu_i = a(\boldsymbol{\mu}_{i[1:n_i]}) = \frac{1}{n_i} \sum_{i=1}^{n_i} \mu_i \quad \sigma_i = a(\boldsymbol{\sigma}_{i[1:n_i]}) = \frac{1}{n_i} \sum_{i=1}^{n_i} \sigma_i \qquad (15)$$

  Moreover, the aggregator reduces the runtime to $O(n)$ where n is the total number of datapoints, which is crucial in improving the model's computational efficiency.
- An NN *decoder* f_θ that takes in a sampled global latent variable $z_i \sim \tilde{Z}_i$ and a new target input $x_{ij'}$, and outputs the mean $\hat{\mu}_{ij'}$ and standard deviation $\hat{\sigma}_{ij'}$ of the predicted Gaussian distribution of the output $\hat{Y}_{ij'} \sim \mathcal{N}(\hat{\mu}_{ij'}, \hat{\sigma}_{ij'}^2)$, for the corresponding evaluation of the function $Y_{ij'} \sim F_i(x_{ij'})$.

Figure 1a shows the structure of such a basic function autoencoder, which is named a *latent model*. However, experimental results shows that such a function autoencoder that relies only on a global



(a) Latent model

(b) Deterministic model

(c) Latent-deterministic model

Figure 1: Structures of the function autoencoder models.

latent variable suffers from underfitting, giving inaccurate predictions at the target inputs, conditioning on the context observations. Therefore, I introduced an additional component, which is an NN *deterministic encoder* h_ψ: for each input context point (x_{ij}, y_{ij}) instead of producing a distribution of random variable Z, it produces a deterministic representation r_{ij}. The vector of representations $\boldsymbol{r}_i$ also needs to be passed through an aggregator, to obtain the global representation $r_i = a(\boldsymbol{r}_i)$.

I then improved the latent model by including both the latent encoder and the deterministic encoder into the function autoencoder, and built the *latent-deterministic model*, as shown in Figure 1c. The lower bound log likelihood of the datasets for this model therefore becomes

$$\mathcal{L}_{\text{lb}i}(\theta, \phi, \psi) = \sum_j \mathbb{E}_{z_i \sim \tilde{Z}_i^{(\phi)}} \left[\log \left(\Pr_{Y_i}(y_{ij} | z_i, r_i, x_{ij}; \theta, \psi) \frac{\Pr_{Z_i}(z_i)}{\Pr_{\tilde{Z}_i^{(\phi)}}(z_i | \boldsymbol{x}_i, \boldsymbol{y}_i)} \right) \right] \tag{16}$$

$$\mathcal{L}_{\text{lb}}(\theta, \phi, \psi) = \sum_i \sum_j \mathbb{E}_{z_i \sim \tilde{Z}_i^{(\phi)}} \left[\log \left(\Pr_{Y_i}(y_{ij} | z_i, r_i, x_{ij}; \theta, \psi) \frac{\Pr_{Z_i}(z_i)}{\Pr_{\tilde{Z}_i^{(\phi)}}(z_i | \boldsymbol{x}_i, \boldsymbol{y}_i)} \right) \right] \tag{17}$$

I have also built a function autoencoder variant by using only the deterministic encoder, called a *deterministic model*, as shown in Figure 1b. The deterministic model serves as a reference model to be compared with the other two models that involve a latent variable. Since the entire model is deterministic, this model is essentially not a variational autoencoder, and can not model distributions over random functions (i.e. it can only fit functions from observing the context points, but is unable to produce different function samples for the same context data). The log likelihood of the datasets for this model can also be computed deterministically, without the need of variational inference:

$$\mathcal{L}_i(\theta, \psi) = \sum_j \log \Pr_{Y_i}(y_{ij} | r_i, x_{ij}; \theta, \psi) \tag{18}$$

$$\mathcal{L}(\theta, \psi) = \sum_i \sum_j \log \Pr_{Y_i}(y_{ij} | r_i, x_{ij}; \theta, \psi) \tag{19}$$

In summary, the pipeline of the three aforementioned models can be formulated as follows: for $i = 1, \cdots, k$ and $j = 1, \cdots, n_i$:

for the latent and latent-deterministic models:

$$\tilde{Z}_{ij} = g_\phi(x_{ij}, y_{ij}) \tag{20}$$

$$\tilde{Z}_i = a(\tilde{\boldsymbol{Z}}_{i[1:n_i]}) \tag{21}$$

$$z_i \sim \tilde{Z}_i \tag{22}$$

for the latent-deterministic and deterministic models:

$$r_{ij} = h_\psi(x_{ij}, y_{ij}) \tag{23}$$

$$r_i = a(\boldsymbol{r}_{i[1:n_i]}) \tag{24}$$

decoding:

$$\hat{y}_{ij} = f_\theta(\langle z_i \rangle, \langle r_i \rangle, x_{ij})^1 \tag{25}$$

3 Experiments

3.1 Experimental setup

In order to test the abilities of my function autoencoders to learn to model the distributions over random functions, I applied them to a 1-dimensional function regression task. For this experiment, the functions were generated using a GP with varying kernel parameters for each function. When generating functions, I first sampled a set of parameters for the Gaussian kernel for the GP, and then used those to sample a function instance F_i. For each function instance F_i, a random number m_i of observations $(\boldsymbol{x}_{i[1:m_i]}, \boldsymbol{y}_{i[1:m_i]})$, limited by a preset maximum number of context points m_{max}, were used as context points for training (i.e., both $\boldsymbol{x}_{i[1:m_i]}$ and $\boldsymbol{y}_{i[1:m_i]}$ were visible to the model), and a further unobserved $n_i - m_i$ datapoints $(\boldsymbol{x}_{i[m_i+1:n_i]}, \boldsymbol{y}_{i[m_i+1:n_i]})$ were used as target points for evaluation (i.e., only $\boldsymbol{x}_{i[m_i+1:n_i]}$ were fed into the model, while the ground truth values $\boldsymbol{y}_{i[m_i+1:n_i]}$ were kept for comparison against the model's predictions $\hat{\boldsymbol{y}}_{i[m_i+1:n_i]}$). For the same function instances, I experimented on training the function autoencoders using a maximum of 10, 50, and 100 context points respectively.

During training, every sampled function instance F_i were trained for a number of iterations, before being discarded and resampling of new function instances. Finding the right number of iterations to train on each function instance can greatly affect the training performance. If each function instance is trained for an insufficient number of iterations, the model cannot learn enough information from each function instance, resulting in a meaningless training overall, and is a waste of the overwhelming data. On the contrary, if each function instance is trained for too many iterations, the model would overfit to that particular function, and cannot learn the big picture of the overall distribution. The number of iterations for training each function instance varies for different models, based on the different characteristics of the models. Since the latent encoder is prone to underfitting, the number of iterations for training each function instance required by a model containing a latent encoder tend to be larger. In practice, after several trials, I eventually set the number of iterations for every function instance to be trained before resampling to be 5,000 for a latent model, and this number became 500 for a latent-deterministic model, and 100 for a deterministic model.

The hidden NN structure for each encoder used in this project was a multi-layer perceptron (MLP) of 4 layers, with each layer containing 128 perceptrons. Each decoder was structured as a 2-layer MLP plus an output layer of dimension 2 (one each for the predicted mean and standard deviation), also with 128 perceptrons in each hidden layer. The dimensions for both the latent variable and the deterministic representation were set to 128. During training, the function instances were trained in batches of size 16. All three types of models, each with all three choices of maximum numbers of context points, were trained for 100,000 epochs, using Adam with learning rate of 10^{-4} as the optimiser. Since this project is more a exploratory project, I did not perform hyperparameter tuning on those hyperparameters, which means that the training outcomes using the above setup may not be the full potential of this method. The code for this project can be found in my GitHub repository.

[1] $\langle \cdot \rangle$ represents an optional argument, but z_i and r_i cannot both be absent.

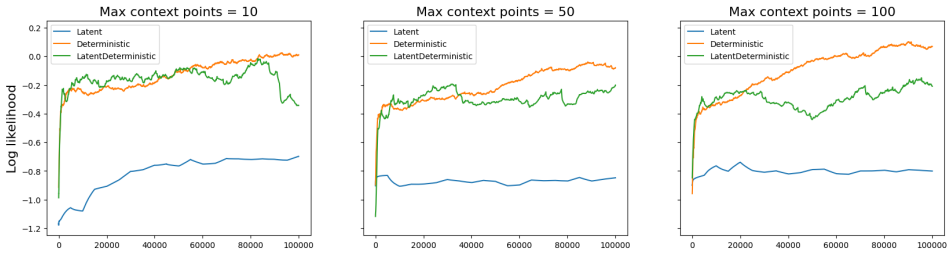


Figure 2: The training log likelihood[2]curves of the function autoencoders with respect to iterations.

Table 1: Best holdout log likelihoods[2] of the trained function autoencoders for 1-d function regression.

Model	Max. num. of context points		
	10	50	100
Latent	-0.6656	-0.7303	-0.6251
Latnet-deterministic	**-0.4708**	-0.5109	-0.3368
Deterministic	-0.4963	**-0.4871**	**0.1013**

3.2 Results

The training log likelihood (lower bounds for the models involving a latent variable) curves of the trained function autoencoders with different limitations on the maximum number of context points are plotted in Figure 2. Table 1 reports the models' training results using different numbers of context points, in terms of the best holdout log likelihoods during training. The results clearly show that although the latent model suffers from underfitting, its performance is significantly boosted when combined with a deterministic encoder. Although admittedly, there is still a tiny gap between the latent-deterministic model and the deterministic model, the latent-deterministic model is able to capture the distribution of the function while the deterministic model can only fit function instances, which makes such a tradeoff totally acceptable, not to mention that there is still a great potential in the latent-deterministic model yet to be exploited. Surprisingly, the number of context points does not affect the model's performances too much, which is likely to be due to underfitting as well.

Figure 3 shows the regression results (blue ribbon plots) of the trained models on the same holdout curve (grey line), using different numbers of context points (black 'plus' marks). The plots show that the latent and latent-deterministic models indeed fits the curve more poorly than the deterministic model, though the latent-deterministic model has regressed much better than the latent model. However, it is still delightful to see that for both the latent and latent-deterministic models, the predicted variance (light blue shaded area) at the locations of the curve where there are less context points are, although only vaguely, higher than those where there are more context points. This suggests that both the latent and latent-deterministic models can indeed learn that as the number of context points increases, the uncertainty at that location is reduced, which is similar to the manner of a GP. In addition, given that the function autoencoder is an NN approximation, it is possible for the predicted curves to miss the observed context points, as opposed to always going through them like a GP. However, on the other hand, the function autoencoder scales much better than a GP as the data size becomes larger, and once the function autoencoder is trained, it can fit more than just one dataset, as it is able to produce sensible predictions for curves generated using any kernel parameters observed during training.

The underfitting issue of the latent and latent-deterministic models can be due to the following reasons:

- no hyperparameter tuning was performed in this experiment, meaning that the current NN setup is highly likely to be sub-optimal;
- the random sampling of the latent variables introduces instability to the system, resulting the model to diverge at some unfavourable datapoints, or some unfortunate sampled values.

[2]lower bounds for the latent and latent-deterministic models.

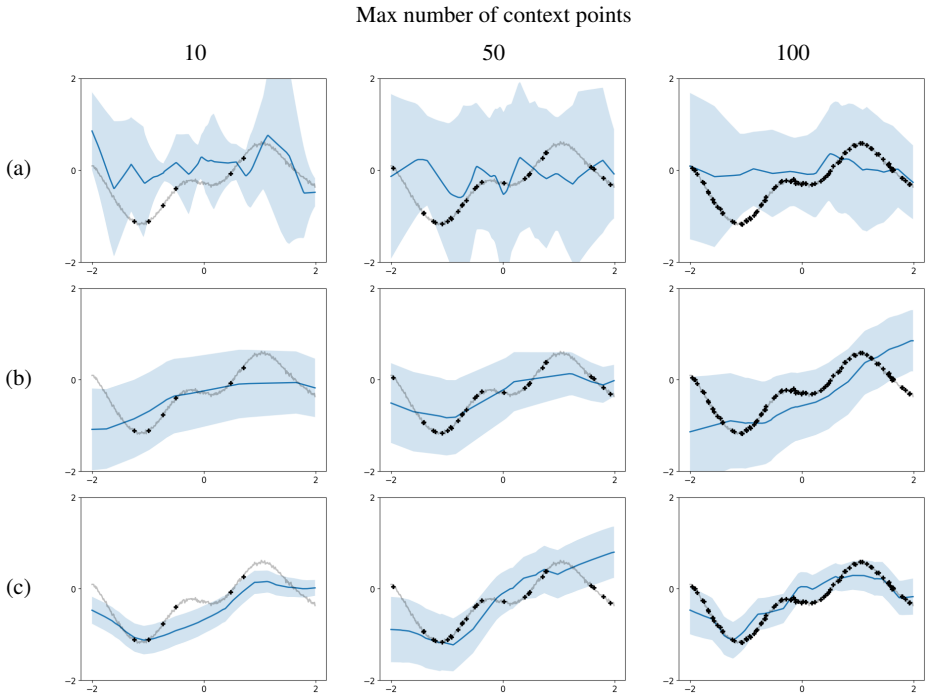


Figure 3: Regression results of the trained (a) latent model (b) latent-deterministic model (c) deterministic model on a holdout 1-dimensional curve generated by a GP (grey line), using different numbers of context points (black 'plus' marks). The blue ribbon plots shows the models' predicted means $f_\theta(x_{ij}, z_i)$ and standard deviations σ of the curve, within the range $f_\theta(x_{ij}, z_i) \pm 1.96\sigma$.

This can also explain the sudden drops on the lower bound training log likelihood curves of the latent-deterministic model, provided that its training outcomes were even higher than those of the deterministic model before the dropping took place;

- there might exists other aggregation functions that performs better than taking the mean;
- the mean-aggregation step in the aggregator can be a bottleneck for the model: since taking the mean across the context latent distributions gives the same weight to each context point, it can be difficult for the decoder to learn which context points provide more relevant information for a given set of target points.

4 Conclusion

In this project, I investigated a neural network approximation to Gaussian processes, and introduced the function autoencoders that can learn to model distributions over random functions. I investigated the suitable maximum log likelihood estimation for the function autoencoders, and built three different variants of it, one using only a global latent variable, another using only a global deterministic representation, and one that combines them both. I trained them on a 1-dimensional function regression task, using random functions generated by a GP with varying kernel parameters. The training results show that although a naïve latent model suffers from underfitting, it can be significantly improved under the assistance of a global deterministic representation, and shows a great potential for further improvements, with the following possible future extensions:

- perform hyperparameter tuning on the hidden NN structures to increase their expressiveness;
- explore more aggregator choices to better combine the outputs of the encoders;
- introduce a variable-weight mechanism for the context points that enables the function autoencoder to up-weight more relevant context points for a given set of target points, or to model relevance within the context points. One promising approach for this would be to introduce attention to the model, as suggested by Attentive Neural Processes (ANPs) [5].

References

[1] C. E. Rasmussen and C. K. I. Williams, *Gaussian processes for machine learning*, ser. Adaptive computation and machine learning. MIT Press, 2006.

[2] M. Garnelo, D. Rosenbaum, C. Maddison, T. Ramalho, D. Saxton, M. Shanahan, Y. W. Teh, D. J. Rezende, and S. M. A. Eslami, Conditional neural processes, in *Proceedings of the 35th International Conference on Machine Learning*, vol. 80. PMLR, 2018, pp. 1704–1713.

[3] M. Garnelo, J. Schwarz, D. Rosenbaum, F. Viola, D. J. Rezende, S. M. A. Eslami, and Y. W. Teh, Neural processes, *arXiv preprint arXiv:1807.01622*, 2018.

[4] B. Øksendal, *Stochastic differential equations: an introduction with applications*, 6th ed. Springer, 2003, p. 11.

[5] H. Kim, A. Mnih, J. Schwarz, M. Garnelo, S. M. A. Eslami, D. Rosenbaum, O. Vinyals, and Y. W. Teh, Attentive neural processes, in *7th International Conference on Learning Representations*. OpenReview.net, 2019.