Scalable Gaussian Process Regression Using Deep Neural Networks*

Wenbing Huang[1], Deli Zhao[2], Fuchun Sun[1], Huaping Liu[1], Edward Chang[2]
[1]State Key Laboratory of Intelligent Technology and System, Tsinghua University, Beijing, China
[2]HTC Research, Beijing, China
{huangwb12@mails, sunfc@mail, hpliu@mail}.tsinghua.edu.cn
zhaodeli@gmail.com, edward_chang@htc.com

Abstract

We propose a scalable Gaussian process model for regression by applying a deep neural network as the feature-mapping function. We first pre-train the deep neural network with a stacked denoising auto-encoder in an unsupervised way. Then, we perform a Bayesian linear regression on the top layer of the pre-trained deep network. The resulting model, Deep-Neural-Network-based Gaussian Process (DNN-GP), can learn much more meaningful representation of the data by the finite-dimensional but deep-layered feature-mapping function. Unlike standard Gaussian processes, our model scales well with the size of the training set due to the avoidance of kernel matrix inversion. Moreover, we present a mixture of DNN-GPs to further improve the regression performance. For the experiments on three representative large datasets, our proposed models significantly outperform the state-of-the-art algorithms of Gaussian process regression.

1 Introduction

Gaussian Processes (GPs) are widely used in regression [Rasmussen and Williams, 2006; Seeger, 2004] and classification [Nickisch and Rasmussen, 2008; Naish-Guzman and Holden, 2007] for their flexibility and competitive learnability. A GP regression model applies a feature-mapping function to project inputs into a feature space and then performs Bayesian linear regression [Rasmussen and Williams, 2006], as shown in Figure 1 (a). A kernel function is usually defined to replace the inner product of the feature-mapping function so as to hold the condition that the feature space is infinite-dimensional. However, such a kernel method leads the training of GPs to suffer from computational issue for large datasets due to the unfavourable inversion of kernel matrices that scales cubically with the number of data points.

There are currently two kinds of methods to address the computational issue of GPs. One is to construct low-rank approximations of kernel matrices, namely Sparse Gaussian Processes (SPGPs) [Smola and Bartlett, 2000; Lawrence *et al.*, 2002; Snelson and Ghahramani, 2005]. The state-of-the-art SPGP was proposed in [Snelson and Ghahramani, 2005], which was later renamed as Fully Independent Training Conditional (FITC) model in [Quiñonero-Candela and Rasmussen, 2005]. The performance of FITC is nearly comparable to that of full GP while its training complexity only scales linearly with the number of training samples. The other one is to learn feature-mapping functions explicitly by fixing the feature space to be finite-dimensional, such as the work of [Lázaro-Gredilla and Figueiras-Vidal, 2010]. In [Lázaro-Gredilla and Figueiras-Vidal, 2010], the authors used the hidden unit transfer functions of one-hidden-layer Neural Networks (NNs) as feature-mapping functions. The resulting model, Marginalized NN mixture (mixMNN[1]), performs better than FITC at almost the same computational cost on several low-dimensional datasets.

The idea of applying the hidden unit transfer functions as the feature-mapping functions of GPs has been explored in many literatures [Neal, 1995; Williams and Barber, 1998; Rasmussen and Williams, 2006]. The famous work was performed by Neal [Neal, 1995]. He proved that a one-hidden-layer neural network equipped with infinite number of hidden units becomes a nonparametric GP model by placing independent zero-mean Gaussian priors to all the weights of the network and integrating them out. Nevertheless, this kind of GPs still suffer from the computational issue due to the assumption of the infinite hidden units. The mixMNN method alleviates this problem by fixing the number of the hidden units to be finite and only marginalizing out the output weights. The resulting model is a parametric GP, of which the kernel function is parameterized by the hidden unit transfer function.

However, in practical, the shallow architecture of the mixMNN makes it inefficient for the highly-nonlinear data, *e.g.* the image-based regression tasks [Zhou *et al.*, 2005]. In this context, we propose a more flexible model by applying Deep Neural Networks (DNNs) as the feature-mapping functions of GPs (as shown in Figure 1 (c)). We perform a Bayesian linear regression on the top layer of the

*This work was jointly supported by the National Science Funds of China for Major Instrument Project (Grant No: 61327809) and Major International Cooperation Project (Grant No: 61210013).

[1]The method is denoted as MNNmix in [Lázaro-Gredilla and Figueiras-Vidal, 2010]. We rename it as mixMNN to be consistent with the denotations of our methods.

DNN, thus resulting in a new GP model, which is named as Deep-Neural-Network-based Gaussian Process (DNN-GP). DNNs such as Stacked Denoising Auto-Encoders (SDAEs) [Vincent *et al.*, 2008; Erhan *et al.*, 2010], deep belief networks [Hinton *et al.*, 2006; Hinton and Salakhutdinov, 2006; Lee *et al.*, 2008], and deep Boltzmann machines [Salakhutdinov and Hinton, 2009; Cho *et al.*, 2013] have been widely adopted in machine learning and computer vision. They are capable of decomposing the data into regular patterns of multiple levels, in which higher level representations are the abstractions of lower level ones [Bengio, 2009]. Employing DNNs as the feature-mapping functions of GPs is expected to learn much more meaningful representation of the input although such feature map is finite-dimensional.

[Salakhutdinov and Hinton, 2007] also employed deep networks as feature extractors to improve the performance of GP regression. They applied the output of deep belief networks as the input of Gaussian kernels. which is substantially different from our algorithmic framework. Therefore, their model also needs to invert the kernel matrix, still suffering from the computational limitation as general GP models do. Our model takes advantage of deep networks by embedding them in the feature space, thus leading to the considerable reduction of computational complexity. [Snoek *et al.*, 2011] and [Damianou and Lawrence, 2012] apply GP-based building blocks as the feature extractors for highly-nonlinear regression. Unlike our DNN-based models, such GP-based building blocks are not scalable to large data unless some extra techniques are added for computational reduction.

In sum, we make the following contributions: (1) We formulate a new kind of GPs, *i.e.* DNN-GPs, by applying DNNs as the feature-mapping functions. We fix the dimensionality of the feature space to be finite and learn the feature-mapping functions explicitly. Therefore, our GPs are feasible for large datasets since the computational complexity only scales linearly with the size of the training set. (2) In DNN-GPs, we pre-train DNN-GPs with SDAEs in an unsupervised way, which not only provides better initializations to the parameters of DNNs, but also makes DNN-GPs perform competitively when the labeled data are limited. (3) We propose mixture models by combining the predictions of several DNN-GPs, dubbed mixDNN-GPs. The mixDNN-GPs algorithm significantly outperforms all compared models in our experiments.

2 Gaussian Process Regression

The most common way to interpret Gaussian processes regression is defining a kernel function as the covariance of the distribution over latent functions, which is known as the *function-space view* [Rasmussen and Williams, 2006]. In this paper, however, we start from the equivalent *weight-space view* [Rasmussen and Williams, 2006] that is more convenient to establish our model. Readers can resort to Mercer's theorem for the proof of the equivalence between the *function-space view* and the *weight-space view* in [Rasmussen and Williams, 2006]. Suppose that we have a dataset $\mathcal{D}$ of N observations $\{(\mathbf{x}_n, y_n)\}_{n=1}^N$, where the dimensionality of each input is D, *i.e.* $\mathbf{x}_n \in \mathbb{R}^D$. We denote $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$ as the input matrix and $\mathbf{y} = [y_1, \cdots, y_N]$ as the output vector. For a regression task, we assume that each output is generated from a probabilistic model

$$y_n = f(\mathbf{x}_n) + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2). \tag{1}$$

Specifying the latent function $f(\mathbf{x}_n)$ to be the linear projection $\mathbf{w}^T\mathbf{x}_n$, we obtain the Bayesian regression model, where the vector of weights $\mathbf{w}$ is sampled from a Gaussian distribution $\mathcal{N}(0, \mathbf{\Sigma}_p)$. A bias weight or offset can be derived by augmenting the input $\mathbf{x}_n$ with an additional element. We do not explicitly include it in our notation for simplicity. If we make a further extension by first projecting the inputs into a feature space with a mapping function $\boldsymbol{\phi}(\mathbf{x})$, we formulate a Gaussian process regression model, as illustrated in Figure 1 (a). It means that in a Gaussian process, the latent function is defined as

$$f(\mathbf{x}_n) = \mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}_n). \tag{2}$$

After marginalizing out the weight vector $\mathbf{w}$, we obtain the prior

$$p(\mathbf{f} \mid \mathbf{X}) = \mathcal{N}(\mathbf{f}; \mathbf{0}, \mathbf{K}), \tag{3}$$

where the latent functions $\mathbf{f} = [f(\mathbf{x}_1), \cdots, f(\mathbf{x}_N)]$ and $\mathbf{K}$ is the covariance matrix whose entries are specified by $\mathbf{K}_{ij} = \boldsymbol{\phi}(\mathbf{x}_i)^T\mathbf{\Sigma}_p\boldsymbol{\phi}(\mathbf{x}_j)$. To improve the model representation ability in standard GP models, the feature space is always assumed to be infinite-dimensional [Rasmussen and Williams, 2006], thereby leading to the application of a kernel function $k(\mathbf{x}_i, \mathbf{x}_j)$ as the entries of the covariance matrix to circumvent the difficulty of computing $\boldsymbol{\phi}(\mathbf{x}_i)^T\mathbf{\Sigma}_p\boldsymbol{\phi}(\mathbf{x}_j)$. The kernel function depends on a small number parameters θ. For example, the Automatic Relevance Determination (ARD) kernel is defined as $k(\mathbf{x}_i, \mathbf{x}_j) = \alpha \exp(-\frac{1}{2}\sum_{d=1}^D (x_i^{(d)} - x_j^{(d)})^2/\beta_d^2)$, where $\boldsymbol{\theta} = \{\alpha, \beta_1, \cdots, \beta_D\}$.

Integrating out latent functions, the negative marginal log-likelyhood takes the form

$$\begin{aligned} L &= -\log p(\mathbf{y} \mid \mathbf{X}, \boldsymbol{\theta}, \sigma) \\ &= \frac{1}{2}\mathbf{y}^T(\mathbf{K} + \sigma^2\mathbf{I})^{-1}\mathbf{y} + \frac{1}{2}\log|\mathbf{K} + \sigma^2\mathbf{I}| \\ &\quad + \frac{N}{2}\log(2\pi), \end{aligned} \tag{4}$$

which is a minimization objective to train the parameters $\boldsymbol{\theta}$ and σ. The prediction of a new input data $\mathbf{x}_*$ can be made by conditioning on observed data and parameters. The distribution of a new target y_* at the new input $\mathbf{x}_*$ is

$$\begin{aligned} &p(y_* \mid \mathbf{x}_*, \mathcal{D}, \boldsymbol{\theta}, \sigma) = \mathcal{N}(y_*; \mathbf{k}_*^T(\mathbf{K} + \sigma^2\mathbf{I})^{-1}\mathbf{y}, \\ &k_{**} - \mathbf{k}_*^T(\mathbf{K} + \sigma^2\mathbf{I})^{-1}\mathbf{k}_* + \sigma^2), \end{aligned} \tag{5}$$

where $\mathbf{k}_*$ is a vector with elements $k(\mathbf{x}_i, \mathbf{x}_*)$ and $k_{**} = k(\mathbf{x}_*, \mathbf{x}_*)$.

Equations (4-5) indicate that the key computation in GPs including the training and the prediction is to invert the covariance matrix $\mathbf{K} + \sigma^2\mathbf{I}$, which is of complexity $O(N^3)$.

3 FITC and mixMNN

To reduce the computational time of GPs from cubic scale to linear scale with respect to N, one approach resorts to

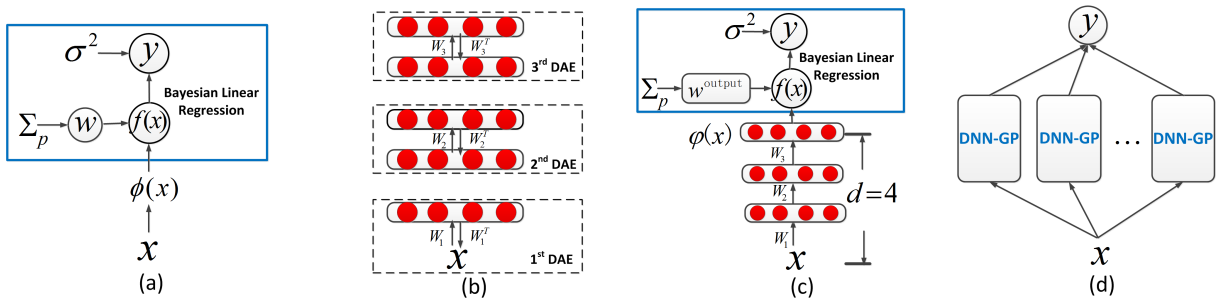

Figure 1: The illustration of related models: (a) Gaussian process regression. (b) The greedy layer-by-layer training process for a stacked denoising auto-encoder. (c) The 4-layer DNN-GP. (d) The mixture of DNN-GPs. The random variables such as $\mathbf{W}$ and y in (a) are represented by circles.

a small set of inducing points to construct a low-rank approximation to the kernel matrix $\mathbf{K}$ over the full dataset , which is known as Sparse GP [Smola and Bartlett, 2000; Lawrence *et al.*, 2002; Snelson and Ghahramani, 2005]. The most successful Sparse GP is the method proposed in [Snelson and Ghahramani, 2005], and then renamed as FITC in [Quiñonero-Candela and Rasmussen, 2005]. FITC augments the training set $\mathcal{D}$ by adding a noiseless pseudo-data set $\bar{\mathcal{D}}$ of size $M < N$: pseudo-inputs $\bar{\mathbf{X}} = \{\bar{\mathbf{x}}_n\}_{n=1}^M$ and pseudo-latent-functions $\bar{\mathbf{f}} = \{\bar{f}_n\}_{n=1}^M$. Assuming f_ns to be conditionally independent given the pseudo-data set, we attain the prior over the latent variables $\mathbf{f}$ by integrating out the pseudo-latent-functions:

$$\begin{aligned} p(\mathbf{f} \mid \mathbf{X}, \bar{\mathbf{X}}) &= \int p(\mathbf{f} \mid \mathbf{X}, \bar{\mathbf{X}}, \bar{\mathbf{f}}) p(\bar{\mathbf{f}}) d\bar{\mathbf{f}} \\ &= \int \prod_{n=1}^{N} p(f_n \mid \mathbf{x}_n, \bar{\mathbf{X}}, \bar{\mathbf{f}}) p(\bar{\mathbf{f}}) d\bar{\mathbf{f}} \\ &= \mathcal{N}(\mathbf{f}; \mathbf{0}, \mathbf{Q} + \mathrm{diag}(\mathbf{K} - \mathbf{Q})), \end{aligned} \quad (6)$$

where $\mathbf{Q} = \mathbf{K}_{\mathbf{f}\bar{\mathbf{f}}}\mathbf{K}_{\bar{\mathbf{f}}\bar{\mathbf{f}}}^{-1}\mathbf{K}_{\bar{\mathbf{f}}\mathbf{f}}$, $\mathbf{K}_{\mathbf{f}\bar{\mathbf{f}}}$, $\mathbf{K}_{\bar{\mathbf{f}}\bar{\mathbf{f}}}$ and $\mathbf{K}_{\bar{\mathbf{f}}\mathbf{f}}$ are matrices with the elements $k(\mathbf{x}_i, \bar{\mathbf{x}}_j)$, $k(\bar{\mathbf{x}}_i, \bar{\mathbf{x}}_j)$ and $k(\bar{\mathbf{x}}_i, \mathbf{x}_j)$, respectively. Compared to the original covariance matrix $\mathbf{K}$ in Equation (3), the covariance matrix here is a low-rank matrix $\mathbf{Q}$ plus a diagonal matrix $\mathrm{diag}(\mathbf{K} - \mathbf{Q})$. With the new covariance matrix, Equations (4-5) can be computed in $O(NM^2)$ time, using the Woodbury matrix identity [Rasmussen and Williams, 2006].

Another way to reduce the computational cost is to learn the feature-mapping function explicitly. Denoting $\boldsymbol{\phi}(\mathbf{X}) = [\boldsymbol{\phi}(\mathbf{x}_1), \cdots, \boldsymbol{\phi}(\mathbf{x}_N)]$, the covariance matrix $\mathbf{K}$ is equal to $\boldsymbol{\phi}(\mathbf{X})^T\boldsymbol{\Sigma}_p\boldsymbol{\phi}(\mathbf{X})$. Setting the dimensionality of $\boldsymbol{\phi}(\mathbf{X})$ to be $M < N$, the term $\boldsymbol{\phi}(\mathbf{X})^T\boldsymbol{\Sigma}_p\boldsymbol{\phi}(\mathbf{X})$ becomes a low-rank construction of $\mathbf{K}$. Explicitly substituting this term into Equations (4-5), the computational cost can be reduced in $O(NM^2)$ as well. One example is the neural-network-based models mixMNNs [Lázaro-Gredilla and Figueiras-Vidal, 2010], in which the hidden unit transfer functions of one-hidden-layer neural networks are employed as feature-mapping functions. Specifically, mixMNNs assign $\boldsymbol{\phi}(\mathbf{x}) = g(\mathbf{U}\mathbf{x} + \mathbf{u}_0)$, where $\mathbf{U}$ and $\mathbf{u}_0$ are the input weight matrix and input bias vector, respectively, and $g(\bullet)$ is the activation function.

4 Our Model: DNN-GP

Similar to mixMNNs, we also reduce the computational cost of GPs by explicitly learning the finite-dimensional feature-mapping functions. Our models can be learned in two steps: the unsupervised pre-training (Section 4.1) and the supervised fine-tuning (Section 4.2).

4.1 Model Pre-training

Stacked Denoising Auto-Encoders (SDAEs) specify deep networks by applying a Denoising Auto-Encoder (DAE) to initialize each layer of deep networks [Vincent *et al.*, 2008]. In a DAE, the original input data $\mathbf{X}$ are corrupted by some kind of stochastic noise. In this paper, we follow the corruption process used in [Vincent *et al.*, 2008]. For the input $\mathbf{X}$, a fixed proportion v of the dimensions of each sample are chosen randomly to be forced to 0, while other dimensions are left invariant, thus leading to the corrupted ones $\mathbf{X}'$. $\mathbf{X}'$ are used for the input while $\mathbf{X}$ are used for the reconstruction targets. The parameters $\mathbf{W}_i$, $\mathbf{b}_i$ and $\mathbf{c}_i$ are trained to minimize the squared error loss function

$$\|\mathbf{X} - g(\mathbf{W}_i{}^T g(\mathbf{W}_i\mathbf{X}' + \mathbf{b}_i) + \mathbf{c}_i)\|_F^2, \quad (7)$$

where $\|\cdot\|_F$ is the matrix Frobenius norm, $\mathbf{W}_i$ is a weight matrix , $\mathbf{b}_i$ and $\mathbf{c}_i$ are bias vectors, $g(\bullet)$ is a nonlinear active function, e.g. a sigmoid function.

Once an DAE is trained, its internal representation $g(\mathbf{W}_i\mathbf{X}' + \mathbf{b}_i)$ can be adopted as the input for training a second DAE. Figure 1 (b) illustrates this training procedure. After such greedy layer-by-layer training, we will obtain a top layer as the highly-nonlinear representation of the input, *i.e.* $\boldsymbol{\varphi}(\mathbf{X})$, which generally captures high-order patterns in original data. After training the whole architecture, we initialize a DNN with the parameters of the learned SDAE. SDAEs can be learned efficiently on large datasets. Training SDAEs scales linearly with the number of training samples. In addition, Equation (7) can be written as the summation of data points, so we can use a stochastic optimization procedure on the mini-batches of data points for each iteration.

4.2 Model Fine-tuning

Substituting the new feature-mapping function into Equation (2), we attain a GP with the elements of the covariance matrix $\mathbf{K}(\mathbf{x}_i, \mathbf{x}_j) = \boldsymbol{\varphi}(\mathbf{x}_i)^T\Sigma_p\boldsymbol{\varphi}(\mathbf{x}_j)$, as illustrated in Figure1 (c).

For simplicity, we consider the isotropic covariance for the weights, namely $\Sigma_p = s_p^{-2}\mathbf{I}$, where s_p^2 is the precision parameter. Thus, $\mathbf{K}(\mathbf{x}_i, \mathbf{x}_j) = s_p^{-2}\boldsymbol{\varphi}(\mathbf{x}_i)^T\boldsymbol{\varphi}(\mathbf{x}_j)$. Substituting the new covariance matrix into Equation (4) and applying the Woodbury matrix identity (A.9 and A.10 in [Rasmussen and Williams, 2006]) to the terms $(\mathbf{K} + \sigma^2\mathbf{I})^{-1}$ and $|\mathbf{K} + \sigma^2\mathbf{I}|$, we attain

$$\begin{aligned} L &= \frac{1}{2}\Big(\frac{1}{\sigma^2}\mathbf{y}^T(\mathbf{I} - \mathbf{F}^T\mathbf{A}^{-1}\mathbf{F})\mathbf{y} + \log|\mathbf{A}| - M\log(s_p^2) \\ &\quad + (N-M)\log(\sigma^2) + N\log(2\pi)\Big), \end{aligned} \tag{8}$$

where $\mathbf{F} = [\boldsymbol{\varphi}(\mathbf{x}_1), \cdots, \boldsymbol{\varphi}(\mathbf{x}_N)] \in \mathbb{R}^{M\times N}$ and $\mathbf{A} = \sigma^2 s_p^2\mathbf{I} + \mathbf{F}\mathbf{F}^T \in \mathbb{R}^{M\times M}$. M is the number of the units of the top layer of the SDAE. The computational complexity of Equation (8) is $O(NM^2)$.

Substituting the new covariance matrix into the prediction Equation (5), we have

$$\begin{aligned} &p(y_* \mid \mathbf{x}_*, \mathcal{D}, \mathbf{W}, \sigma, s_p) = \mathcal{N}(y_*; \mu_*, \sigma_*^2), \\ &\mu_* = \boldsymbol{\varphi}(\mathbf{x}_*)^T\mathbf{A}^{-1}(\mathbf{F}\mathbf{y}), \\ &\sigma_*^2 = \sigma^2\boldsymbol{\varphi}(\mathbf{x}_*)^T\mathbf{A}^{-1}\boldsymbol{\varphi}(\mathbf{x}_*) + \sigma^2. \end{aligned} \tag{9}$$

The computational complexity of the new prediction is only $O(NM)$.[2] Denoting $h(\mathbf{x}_*) = \mathbf{L}^{-1}\varphi(\mathbf{x}_*)$ as the new feature-mapping function and letting $\mathbf{s} = \mathbf{L}^{-\mathbf{1}}\mathbf{F}\mathbf{y}$, we can compactly rewrite Equation (9) as

$$\begin{aligned} &p(y_* \mid \mathbf{x}_*, \mathcal{D}, \mathbf{W}, \sigma, s_p) = \mathcal{N}(y_*; \mu_{**}, \sigma_{**}^2), \\ &\mu_{**} = \mathbf{s}^T h(\mathbf{x}_*), \\ &\sigma_{**}^2 = \sigma^2 h(\mathbf{x}_*)^T h(\mathbf{x}_*) + \sigma^2, \end{aligned} \tag{10}$$

where the new predictive mean is exactly a linear combination of the new features. It means that in terms of the prediction of the expectation, we finally derive a feature space where the output is the linear function of the input, after a highly-nonlinear transformation, *i.e.* $h(\mathbf{x}_*)$. The vector $\mathbf{s}$ depends on the training samples but is independent of the testing input. It can measure the relevances of the features to the prediction.

Now we perform the discriminative fine-tuning using back propagation. We first compute the derivatives of the objection function (8) with respect to the explicit parameters including the mapping matrix $\mathbf{F}$ and parameters σ^2 and s_p^2

$$\frac{\partial L}{\partial \mathbf{F}} = -\frac{1}{\sigma^2}\mathbf{A}^{-1}(\mathbf{F}\mathbf{y})\mathbf{y}^T + \mathbf{C}\mathbf{F}, \tag{11}$$

$$\begin{aligned} \frac{\partial L}{\partial \sigma^2} &= \frac{1}{2\sigma^4}\left((\mathbf{F}\mathbf{y})^T\mathbf{A}^{-1}(\mathbf{F}\mathbf{y}) - \mathbf{y}^T\mathbf{y}\right) \\ &\quad + \frac{s_p^2}{2}\mathbf{tr}(\mathbf{C}) + \frac{(N-M)}{2\sigma^2}, \end{aligned} \tag{12}$$

$$\frac{\partial L}{\partial s_p^2} = -\frac{M}{2s_p^2} + \frac{1}{2}\sigma^2\mathbf{tr}(\mathbf{C}), \tag{13}$$

[2]Equation (8) and Equation (9) are similar to Equation (7) and Equation (6) in [Lázaro-Gredilla and Figueiras-Vidal, 2010] respectively. The difference is that $\boldsymbol{\varphi}(\mathbf{X})$ in [Lázaro-Gredilla and Figueiras-Vidal, 2010] is only a simple hidden unit transfer function of one-hidden-layer NN, while $\boldsymbol{\varphi}(\mathbf{X})$ here is the output of the learned SDAEs.

where the matrix $\mathbf{C} = 2 \times \frac{\partial L}{\partial \mathbf{A}}$, and $\mathbf{tr}(\bullet)$ is the matrix trace. Then by the chain rule, we readily back propagate $\frac{\partial L}{\partial \mathbf{F}}$ to the implicit parameters $\mathbf{W}$ and $\mathbf{b}$. For $\mathbf{W}$, we compute the gradients as

$$\begin{aligned} &\frac{\partial L}{\partial \mathbf{W}_{d-1}} = \frac{\partial L}{\partial \mathbf{a}_d}\frac{\partial \mathbf{a}_d}{\partial \mathbf{W}_{d-1}}, \frac{\partial L}{\partial \mathbf{W}_{d-2}} = \frac{\partial L}{\partial \mathbf{a}_{d-1}}\frac{\partial \mathbf{a}_{d-1}}{\partial \mathbf{W}_{d-2}}, \\ &\cdots, \quad \frac{\partial L}{\partial \mathbf{W}_1} = \frac{\partial L}{\partial \mathbf{a}_2}\frac{\partial \mathbf{a}_2}{\partial \mathbf{W}_1}, \end{aligned} \tag{14}$$

where $\{\mathbf{a}_2, \cdots, \mathbf{a}_d\}$ are the active values of the hidden units of the corresponded layers, and d is the depth of DNN-GPs, *i.e.* the number of the layers of DNNs. It is worth noting that we do not include the output layer when counting d, as already shown in Figure 1 (c). The computation of the gradients of $\mathbf{b}$ is similar to Equation (14). One may refer to Section 5.3 in [Bishop, 2006] for details. Similar to pre-training, back-propagation in Equation (14) scales linearly with N [Hinton and Salakhutdinov, 2006]. The computational complexity of Equation (8) is $O(NM^2)$. Thus, our model is scalable to the number of training samples .

Based on Equations (11)-(14), we apply the conjugate gradient method to fine-tune our models. Because the objective function (8) cannot be decomposed into a sum over data points, it encounters the difficulty when using a stochastic optimization procedure on the mini-batches of data points. Thus, we apply the full-batch optimization for fine-tuning. Even so, fine-tuning our models on full-batch performs with the competitive speed of convergence in our experiments.

4.3 Combining predictions of several DNN-GPs

To overcome the risk of overfitting, mixMNNs average individual predictions of several one-hidden-layer NNs of which parameters are initialized randomly. In DNN-GPs, we initialize the parameters of DNNs at random and then pre-train them with SDAEs. In this way, a single DNN-GP is capable of achieving satisfactory performance. However, we find that combining predictions of several DNN-GPs can further dramatically boost the performance. Following [Lázaro-Gredilla and Figueiras-Vidal, 2010], we derive the combined mean and variance of the distribution of y_* at a new test sample $\mathbf{x}_*$

$$\mu_{final*} = \frac{1}{K}\sum\nolimits_{k=1}^{K}\mu_{*k}, \tag{15}$$

$$\sigma_{final*}^2 = \frac{1}{K}\sum\nolimits_{k=1}^{K}\mu_{*k}^2 + \sigma_{*k}^2 - \mu_{final*}^2, \tag{16}$$

where K is the number of combined GPs, μ_{*k} and σ_{*k}^2 are the predictive mean and variance of k-th GP, respectively. We denote the mixture version of DNN-GPs as mixDNN-GPs.

Dataset	Targets	#Train	#Test	#Dim
Rectangles	[2,31]	10000	2500	1024
Olivetti Faces	[-90,90]	16000	4000	1024
FGnet	[log 2,log 70]	16000	4040	1024

Table 1: Information of datasets.

5 Experiments

We compare the performance of proposed models with full GPs, FITCs [Snelson and Ghahramani, 2005] and mixMNNs

[Lázaro-Gredilla and Figueiras-Vidal, 2010] on one synthetic image dataset *Rectangles* [Larochelle *et al.*, 2007] and two real face datasets *Olivetti Faces* [Salakhutdinov and Hinton, 2007] and *FGnet* [Zhou *et al.*, 2005]. *Rectangles* is used for predicting the area of rectangles, *Olivetti Faces* is for face orientation extraction and *FGnet* is for age estimation. We first enlarge each dataset to contain more than 10000 samples, and then pre-process all the images to the same size 32×32 and normalize the values of pixels in $[0, 1]$. For reader's convenience, we list the information of datasets in Table 1.

For mixMNNs and DNN-GPs, we use the pre-processed images as the inputs and the sigmoid functions as the activation functions. For full GPs and FITCs, PCA is applied to the images for feature extraction so as to improve the performance, meaning that the first p principal components are chosen as the input. To select the optimal p, the performance of a full GP will be evaluated for various values of p and the value resulted in best performance will be chosen. The setting of training SDAEs in DNN-GPs is fixed as follows: The corrupted proportion v in each DAE is set to be 0.5. Equation (7) is optimized with the gradient descent method whose learning rate is 0.1. The dataset is divided into mini-batches of size 100, and the parameters are updated after each mini-batch iteration. Each DAE is trained for 30 passes through the entire training set. We train full GPs, FITCs, mixMNNs, and fine-tune DNN-GPs with the conjugate gradient method [Rasmussen and Williams, 2006] on the full-batch of training samples. If the objective function dose not decrease within 50 evaluations in each line search, or the epoch of the line search exceeds 100, the training process will halts.

Following [Lázaro-Gredilla and Figueiras-Vidal, 2010; Titsias and Lázaro-Gredilla, 2013], all the experiments here are measured with Normalized Mean Square Error (NMSE)

$$\text{NMSE} = \frac{\frac{1}{N_{test}} \sum_{n=1}^{N_{test}} (y_n - \mu_n)^2}{\frac{1}{N_{test}} \sum_{n=1}^{N_{test}} (y_n - \bar{y})^2}, \qquad (17)$$

where y_n and μ_n are the true label and predictive mean of the n-th testing sample, respectively, $\bar{y}$ is the mean of the labels of the training set and N_{test} is the number of testing samples.

5.1 Notations: M and d

We use a unified symbol M to denote the number of the units at the top layer of SDAEs in DNN-GPs, the number of pseudo-inputs in FITCs and the number of hidden units in mixMNNs. In all of our experiments, we fix the number of the neural units of the other hidden layers to be 500 excluding the top layer of SDAEs. Such a constraint may limit the ability of DNN-GPs. In our practice, however, we find that the DNN-GPs of this architecture are sufficient to achieve desirable performance. We denote K as the number of the individual architectures used to construct the mixture models (*i.e.* mixMNNs and mixDNN-GPs). Increasing K leads to enhanced performance and higher computational cost. We set $K = 4$ for both mixMNNs and mixDNN-GPs.

5.2 Analysis on Proposed Models

We compare the performance of DNN-GP with its several variants to evaluate the influence of the depth d, the pre-training, the fine-tuning and the mixture setting. Here, two sub-experiments are implemented.

Methods	d=2	d=3	d=4	d=5
simDNN-GP	**7.78**	1.98	**1.57**	**3.42**
DNN-GP	49.67	1.60	28.45	6.26
mixDNN-GP	31.17	**0.93**	19.27	5.02

Methods	d=2	d=3	d=4	d=5
simDNN-GP	4.44	2.88	2.28	0.69
DNN-GP	2.32	0.69	0.69	0.43
mixDNN-GP	**1.49**	**0.51**	**0.49**	**0.37**

Table 2: Results of averaged NMSEs on *Olivetti Faces*. Top: Training NMSEs ($\times 10^{-4}$). Bottom: Testing NMSEs ($\times 10^{-2}$). M is fixed to 800, and d is changed from 2 to 5.

In the first one, we define a simplified version of DNN-GPs as simDNN-GPs, in which no pre-training procedure is involved. The parameters of simDNN-GPs are initialized at random. We report both the training and testing NMSEs on *Olivetti Face* in Table 2. Here, the training NMSE and the testing NMSE mean the NMSE measured on the training set and the testing set, respectively. Table 2 shows that the testing NMSEs of all the three models will decrease when d increases. Building a deep architecture is helpful for improving the performance of GP regression. With the same d, simDNN-GPs provide lower training NMSEs but higher testing NMSEs than DNN-GPs. The unsupervised pre-training acts as a regularizer to enforce the parameters of DNN-GP to a constrained region, where a better generalization yields. The mixDNN-GPs approach provides both lower testing NMSEs and training NMSEs than DNN-GPs. Committing a mixture of DNN-GPs is able to attain an optimized solution, thereby resulting in an improved performance.

In the second one, we compare the performance of DNN-GP with its two counterparts: LBR+DNN and GP+DNN. LBR+DNNs and GP+DNNs apply the output of SDAEs as the input of Linear-Bayesian-Regression models (LBRs) and full GPs, respectively. Unlike DNN-GPs, there is no global fine-tuning involved in this two models. The average testing NMSEs are reported in Tabel 3. Clearly, DNN-GPs achieve the best accuracy. LBR+DNNs perform grossly worse than DNN-GPs, which verifies that the global fine-tuning plays the essential role in our models.

Datasets	LBR+DNN	GP+DNN	DNN-GP
Rectangles	0.0904	0.0615	**0.0091**
Olivetti Faces	0.0943	0.0569	**0.0043**
FGnet	0.7494	0.3749	**0.1833**

Table 3: Results of averaged testing NMSEs. M and d are fixed to 800 and 5, respectively.

5.3 Comparison on Full Training Sets

In this experiment, we perform the comparison between FITC+PCAs, mixMNNs, 5-layer DNN-GPs and 5-layer mixDNN-GPs. To evaluate the performance of above models

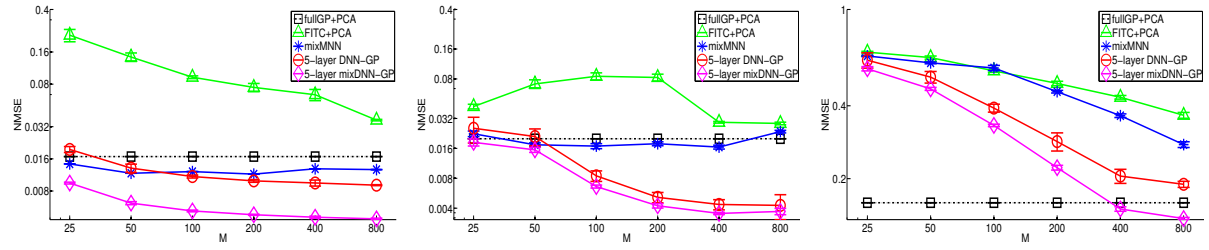

Figure 2: Testing NMSEs of fullGP+PCAs, FITC+PCAs, mixMNNs, 5-layer DNN-GPs and 5-layer mixDNN-GPs on the full training sets. From left to right, it demonstrates the results of *Rectangles*, *Olivetti Faces* and *FGnet*, respectively. M varies from 25 to 800. Both the horizontal and vertical axes are in logarithmic scale.

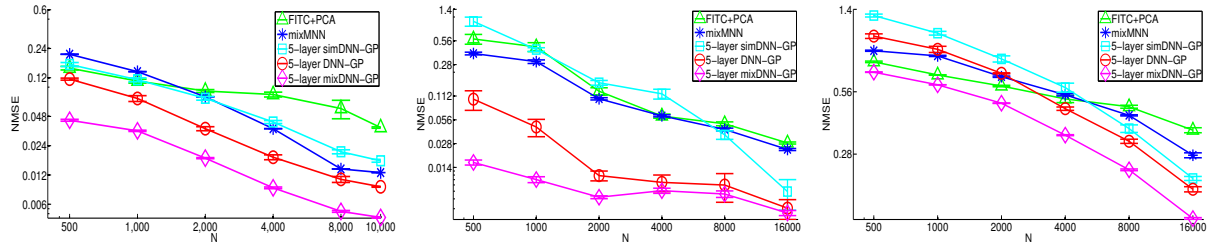

Figure 3: Testing NMSEs of fullGP+PCAs, FITC+PCAs, mixMNNs, 5-layer DNN-GPs and 5-layer mixDNN-GPs on the labeled training sets of varying size . From left to right, it demonstrates the results of *Rectangles*, *Olivetti Faces* and *FGnet*, respectively. Both the horizontal and vertical axes are in logarithmic scale.

at different computational complexities, we vary M from 25 to 800 and display the results of the testing NMSEs in Figure 2. We also implement fullGP+PCA on the full training set so as to provide a reference of the desirable performance. From Figure 2, we observe that a larger M results in a better performance for both DNN-GPs and mixDNN-GPs. DNN-GPs perform better than FITC+PCAs and mixMNNs on all three datasets when $M > 50$. Compared with fullGP+PCA, DNN-GPs obtain lower testing errors on *Rectangles* and *Olivetti Faces*, and the comparable accuracy on *FGnet*. mixDNN-GPs achieve the best performance among all the compared models when $M = 800$.

The training of DNN-GPs includes the layer-by-layer pre-training (Equation (7)), the computation of the derivatives of the explicit parameters (Equations (11) - (13)) and the back propagation of the weight parameters (Equation (14)). As we have demonstrated in Section 4, all these computations scale linearly with the number of training samples, which leads DNN-GPs to be feasible for large datasets. The computational cost for training the full GP and the DNN-GP is presented in Tabel 4. Obviously, the training of DNN-GPs is much faster than that of full GPs.

Methods	*Rectangles*	*Olivetti Faces*	*FGnet*
full GP	43,904s	142,200s	164,648s
DNN-GP	1,138s	2,387s	2,934s

Table 4: The training cost of the full GP and the DNN-GP. M and d are fixed to 800 and 5, respectively. All experiments are carried out with Matlab 8.1.0.604 (R2013a) on Intel Core i7, 2.90-GHz CPU with 8-GB RAM.

5.4 Comparison on Limited Labeled Sets

In this task, we evaluate the performance of FITC+PCAs, mixMNNs, 5-layer simDNN-GPs, 5-layer DNN-GPs and 5-layer mixDNN-GPs on limited labeled training sets. M is fixed to be 800. We construct the labeled training set of size N by randomly selecting a subset from the original training set. Training FITC+PCAs, mixMNNs and simDNN-GPs are based on the labeled training set. For DNN-GPs and mixDNN-GPs, we first pre-train these two models with the original training set and then fine-tune them with the limited labeled set. Figure 3 illustrates the performance of the compared models on the labeled sets of different sizes.

On *Olivetti Faces*, DNN-GP achieves very low error even when the number of labeled data N is very small. Particularly, DNN-GP with only 2000 labeled training samples outperforms FIIC+PCAs and mixMNN which are trained on the full dataset, *i.e.* $N = 16000$. In contrast, simDNN-GP performs worse than FIIC+PCA and mixMNN when $N < 8000$. When $N = 16000$, simDNN-GP finally outperforms FIIC+PCA and mixMNN but still is inferior to DNN-GP and mixDNN-GP. On the other two datasets *Rectangles* and *FGnet*, we still observe the fact that DNN-GPs perform better than simDNN-GPs in all cases, thus verifying the importance of the unsupervised pre-training. As expected, mixDNN-GPs outperform all the compared models in all cases.

6 Conclusion

In this paper, we applied stacked denoising auto-encoders as the feature-mapping functions to reformulate GPs,

which have been named as Deep-Neural-Network-based GPs (DNN-GPs). Our experiments verified the following conclusions: (1) DNN-GPs considerably outperform FITCs and mixMNNs, and perform better than full GPs as a whole, while the training of DNN-GPs is scalable to the size of the training set. (2) The unsupervised training of SDAEs is important to the performance of DNN-GPs, which not only provides better initializations to the parameters of DNNs, but also makes DNN-GPs perform competitively when the number of the labeled data is limited. (3) The mixture setting can significantly improve the performance of DNN-GPs. The mixDNN-GPs algorithm achieves the state-of-the-art performance in our experiments.

Overall, applying deep neural networks as the feature-mapping functions is able to effectively facilitate the performance of GPs for regression problems with a large amount of training data. For the future work, we will extend DNN-GPs to multi-output regression problems.

References

[Bengio, 2009] Y. Bengio. Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, 2(1):1–127, 2009.

[Bishop, 2006] C. M. Bishop. *Pattern recognition and machine learning*. Springer, New York, 2006.

[Cho *et al.*, 2013] Kyung Hyun Cho, Tapani Raiko, and Alexander Ilin. Gaussian-Bernoulli deep Boltzmann machine. In *International Joint Conference on Neural Networks (IJCNN)*, 2013.

[Damianou and Lawrence, 2012] Andreas C Damianou and Neil D Lawrence. Deep gaussian processes. In *Proceedings of the Sixteenth International Workshop on Artificial Intelligence and Statistics*, 2012.

[Erhan *et al.*, 2010] Dumitru Erhan, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, Pascal Vincent, and Samy Bengio. Why does unsupervised pre-training help deep learning? *Journal of Machine Learning Research (JMLR)*, 11:625–660, 2010.

[Hinton and Salakhutdinov, 2006] G E Hinton and R R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, July 2006.

[Hinton *et al.*, 2006] G. E. Hinton, S. Osindero, and Y. W. Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18:1527–1554, 2006.

[Larochelle *et al.*, 2007] Hugo Larochelle, Dumitru Erhan, Aaron Courville, James Bergstra, and Yoshua Bengio. An empirical evaluation of deep architectures on problems with many factors of variation. In *International Conference on Machine learning (ICML)*, 2007.

[Lawrence *et al.*, 2002] Neil D. Lawrence, Matthias Seeger, and Ralf Herbrich. Fast sparse Gaussian process methods: the informative vector machine. In Suzanna Becker, Sebastian Thrun, and Klaus Obermayer, editors, *Advances in Neural Information Processing Systems (NIPS)*, pages 609–616, 2002.

[Lázaro-Gredilla and Figueiras-Vidal, 2010] Miguel Lázaro-Gredilla and Aníbal R Figueiras-Vidal. Marginalized neural network mixtures for large-scale regression. *IEEE Transactions on Neural Networks*, 21(8):1345–1351, 2010.

[Lee *et al.*, 2008] Honglak Lee, Chaitanya Ekanadham, and Andrew Y Ng. Sparse deep belief net model for visual area v2. In *Advances in Neural Information Processing Systems (NIPS)*, pages 873–880, 2008.

[Naish-Guzman and Holden, 2007] Andrew Naish-Guzman and Sean B. Holden. The generalized FITC approximation. In John C. Platt, Daphne Koller, Yoram Singer, and Sam T. Roweis, editors, *Advances in Neural Information Processing Systems (NIPS)*, 2007.

[Neal, 1995] Radford M Neal. *Bayesian learning for neural networks*. PhD thesis, University of Toronto, 1995.

[Nickisch and Rasmussen, 2008] Hannes Nickisch and Carl Edward Rasmussen. Approximations for binary Gaussian process classification. *Journal of Machine Learning Research (JMLR)*, 9:2035–2078, October 2008.

[Quiñonero-Candela and Rasmussen, 2005] Joaquin Quiñonero-Candela and Carl Edward Rasmussen. A unifying view of sparse approximate Gaussian process regression. *Journal of Machine Learning Research (JMLR)*, 6:1939–1959, 2005.

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

[Salakhutdinov and Hinton, 2007] Ruslan Salakhutdinov and Geoffrey E. Hinton. Using deep belief nets to learn covariance kernels for Gaussian processes. In John C. Platt, Daphne Koller, Yoram Singer, and Sam T. Roweis, editors, *Advances in Neural Information Processing Systems (NIPS)*, 2007.

[Salakhutdinov and Hinton, 2009] Ruslan Salakhutdinov and Geoffrey E. Hinton. Deep Boltzmann machines. In David A. Van Dyk and Max Welling, editors, *International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 448–455, 2009.

[Seeger, 2004] Matthias Seeger. Gaussian processes for machine learning. *International Journal of Neural Systems.*, 14(2):69–106, 2004.

[Smola and Bartlett, 2000] Alex J. Smola and Peter L. Bartlett. Sparse greedy Gaussian process regression. In Todd K. Leen, Thomas G. Dietterich, and Volker Tresp, editors, *Advances in Neural Information Processing Systems (NIPS)*, pages 619–625, 2000.

[Snelson and Ghahramani, 2005] Edward Snelson and Zoubin Ghahramani. Sparse Gaussian processes using pseudo-inputs. In *Advances in Neural Information Processing Systems (NIPS)*, 2005.

[Snoek *et al.*, 2011] Jasper Snoek, Ryan Prescott Adams, and Hugo Larochelle. On nonparametric guidance for learning autoencoder representations. *Journal of Machine Learning Research (JMLR)*, 2011.

[Titsias and Lázaro-Gredilla, 2013] Michalis Titsias and Miguel Lázaro-Gredilla. Variational inference for mahalanobis distance metrics in Gaussian process regression. In *Advances in Neural Information Processing Systems (NIPS)*, pages 279–287, 2013.

[Vincent *et al.*, 2008] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *International Conference on Machine learning (ICML)*, 2008.

[Williams and Barber, 1998] Christopher KI Williams and David Barber. Bayesian classification with Gaussian processes. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 20(12):1342–1351, 1998.

[Zhou *et al.*, 2005] Shaohua Kevin Zhou, Bogdan Georgescu, Xiang Sean Zhou, and Dorin Comaniciu. Image based regression using boosting method. In *IEEE International Conference on Computer Vision (ICCV)*, 2005.