Improved Uncertainty Estimates for Geophysical Parameter Retrieval

by

Zuoyu Tao

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2010

Author .....
Department of Electrical Engineering and Computer Science
May 18, 2010

Certified by .....
William J. Blackwell
Senior Staff
Thesis Supervisor

Certified by .....
David H. Staelin
Professor
Thesis Supervisor

Accepted by .....
Dr. Christopher J. Terman
Chairman, Department Committee on Graduate Theses

Improved Uncertainty Estimates for Geophysical Parameter Retrieval

by

Zuoyu Tao

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2010, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science

Abstract

Algorithms for retrieval of geophysical parameters from radiances measured by instruments onboard satellites play a large role in helping scientists monitor the state of the planet. Current retrieval algorithms based on neural networks are superior in accuracy and speed compared to physics-based algorithms like iterated minimum variance (IMV). However, they do not have any form of error estimation, unlike IMV. This thesis examines the suitability of several different approaches to adding in confidence intervals and other methods of error estimation to the retrieval algorithm, as well as alternative machine learning methods that can both retrieve the parameters desired and assign error bars. Test datasets included both current generation operational instruments like AIRS/AMSU, as well as a hypothetical future hyperspectral microwave sounder. Mixture density networks (MDN) and Sparse Pseudo Input Gaussian processes (SPGP) were found to be the most accurate at variance prediction. Both of these are novel methods in the field of remote sensing. MDNs also had similar training and testing time to neural networks, while SPGPs often took three times as long to train in typical cases. As a baseline, neural networks trained to estimate variance were also tested, but found to be lacking in accuracy and reliability compared to the other methods.

Thesis Supervisor: William J. Blackwell
Title: Senior Staff

Thesis Supervisor: David H. Staelin
Title: Professor

Acknowledgments

I want to thank my supervisor, Bill Blackwell, for introducing me to the field of remote sensing, for his support and encouragement throughout my work on the thesis, and for his thoughtful comments and suggestions on the thesis itself.

I would also like to thank my faculty supervisor, Prof. David Staelin, for the help he gave me in finding a good thesis question, for answering all my questions, and for his guidance and advice while writing the thesis.

The people at Lincoln Lab were extremely helpful while I was working on the thesis. I'd especially like to thank Mike Pieper for his help in obtaining the datasets I used and for the many interesting discussions we had.

A special thanks for Ed Snelson, who kindly made available his implementation of SPGP, as well as answering the questions I had about SPGP.

Last but not least, I want to thank my parents for their support and advice both on the thesis and in all matters beyond.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As the sensor capabilities of weather satellites improve, there is need for faster algorithms to convert the raw data from the satellites into useful measurements like temperature and humidity [6]. This is known as the retrieval problem. Better algorithms that solve this retrieval problem could lead to more accurate weather forecasts, and could help us better understand the earth's climate as a whole [2].

For the most part, retrieval algorithms are still based on models of the underlying physics [24]. However, there has always been interest in statistical techniques like simple linear regression [26], because of the speed advantage such statistical methods have (after training is completed). Recently researchers have successfully used neural networks, a certain kind of nonlinear regression, in many retrieval problems of interest [1] [6]. Neural networks were found to be considerably faster than physical techniques and had comparable, or better, accuracy in most cases.

Unfortunately, current neural network retrievals are hampered by the inability to judge how accurate their predictions are, unlike the older retrieval algorithms based on physical models. Because it is useful for numerical weather prediction models to understand how reliable an estimate is, predictions that include a probability distribution, confidence intervals or variance predictions could greatly improve the utility of retrievals in that setting [17]. Overall, having estimates of the prediction error would also lead to greater acceptance and use of statistical retrieval methods [1], providing a way to quantify their stability and to make sure the retrieved quantities

are physically possible.

In this thesis, I will investigate several statistical retrieval methods that can be used to assign uncertainty estimates to predictions and then test their suitability to some representative retrieval problems.

1.1 Thesis outline

Chapter 2 covers the remote sensing problem in slightly more detail. It also introduces baseline statistical retrieval methods–linear and quadratic regression, and neural networks. The retrieval problems and the datasets are then presented, and the performance of the retrieval methods on those datasets evaluated.

Chapter 3 describes Bayesian neural networks, a method previously used with some success on retrieval problems like the type covered. Its suitability and performance on the datasets is discussed. Unfortunately, the presence of heteroscedastic noise in retrieval problems degrades the performance of Bayesian neural networks, and motivates the search for other techniques to assign error bars.

Chapter 4 reviews two methods, Mixture Density Networks (MDNs) and Sparse pseudo-input Gaussian processes (SPGP), that have not been used before in the retrieval problems. Their performance is evaluated on the datasets in terms of both retrieval accuracy and the accuracy of the predicted error bars.

Finally, Chapter 5 concludes the thesis and includes a few recommendations for future work.

Chapter 2

Statistical Retrieval Methods

In this chapter, I will briefly review the basic principles of remote sensing before reviewing a couple of statistical retrieval algorithms. I will then introduce three representative remote sensing datasets and discuss the performance of the statistical methods on those retrieval problems.

2.1 Remote Sensing

Modern weather satellites measure spectral radiance–the power flux at a particular frequency. From this, retrieval algorithms are expected to produce the more useful geophysical parameters. These include temperature throughout the levels of the atmosphere (known as a "profile"), humidity profile, precipitable water, surface temperature, cloud liquid water content, and so on [2]. This is possible because each frequency is sensitive to those quantities at different altitudes.

Although the so-called "inverse problem" of retrieving these geophysical parameters from the measured radiances is hard, the reverse—predicting the measured radiances given the geophysical parameters—is much easier. The physics are well-known, and there a multitude of packages, such as the Standalone AIRS Radiative Transfer Algorithm (SARTA), that can quickly simulate infrared radiances based on extensively validated models (AIRS is the Atmospheric Infrared Sensor, an instrument). Similar packages exist for simulating the observed radiances of microwave instruments

as well [22]. The relative ease of the reverse problem is the basis of many physics based retrieval methods, such as iterated minimum-variance, or IMV [18].

At each iteration, IMV essentially guesses at the geophysical parameters, simulates the resulting radiances, and adjusts the guess based on the differences between the simulated and the actual radiances. One of the practical advantages of this approach is that errors due to instrument noise or uncertainty in the model can be accounted for. Unfortunately, the algorithm takes many, time-consuming, iterations until convergence, even with a good first guess of the parameters.

The slowness of the physical retrieval methods motivates the search for alternative approaches to retrieval of geophysical parameters. As mentioned before, the reverse problem is relatively easy, allowing us to build up large datasets of simulated data. Geophysical parameters from the thousands of available radiosondes (instruments mounted on weather balloons), or from numerical weather prediction models, can also be synced with satellite observations to create datasets. With the easy availability of large datasets, it is natural to look at statistical retrievals as an alternative to physics-based inversion. This idea is not new—linear regression has been used to retrieve parameters of interest from the radiances from a microwave instrument since at least the 1970s [26]. As long as the training dataset is comprehensive enough, statistical retrievals should give good accuracy while taking much less time to perform retrievals on new test cases. Recently though, there has been an burgeoning interest in neural networks in the context of remote sensing, primarily because in many applications they are much superior in speed and at least equal in accuracy to operational physics-based algorithms such as IMV [7].

In the following, I will review a couple of common statistical retrieval methods, linear regression and neural networks. I will then discuss the datasets I used to evaluate their performance. The neural network's performance will serve as a good baseline for comparison with other statistical retrieval methods in later chapters, and the results from linear and quadratic regression should give a good indication as to how nonlinear the retrieval problem is.

The convention used throughout the thesis is that D will represent the training

dataset of n cases, split into the set T of training targets t_i (or $\mathbf{t}_i$ if there are multiple outputs, so that the targets are a vector and not a scalar) and a set X of training inputs $\mathbf{x}_i$, where $i = [1, n]$. The function and the outputs estimated by the regression model will be denoted $\mathbf{y}$.

2.2 Linear Regression

Linear least squares regression is a commonplace statistical retrieval method, guaranteed to be optimal for linear problems with Gaussian noise. The underlying function is assumed to be of the form

$$\mathbf{t} = \beta\mathbf{X} + \epsilon; \tag{2.1}$$

Where ϵ is the Gaussian noise, and $\mathbf{t}$ is a vector consisting of the training targets t_i. The model itself is of the form

$$\mathbf{y} = \beta\mathbf{X}; \tag{2.2}$$

Where $\mathbf{y}$ is either a $1 \times n$ vector (one output) or a $c \times n$ matrix (c outputs). $\mathbf{X}$ is a $d \times n$ matrix of inputs, with d rows for each dimension of the input. β is a $c \times n$ matrix of inputs. If the columns of $\mathbf{X}$ are not zero-mean, a bias term can be included by appending an extra row of ones to $\mathbf{X}$.

The best linear fit can be determined by taking the derivative of β with respect to the error function (this is simply the sum of the squared residuals):

$$E = \sum_{\mathbf{x}_i, t_i \in D} (t_i - \beta\mathbf{x}_i).^2 = (\mathbf{t} - \beta\mathbf{X})(\mathbf{t} - \beta\mathbf{X})^T \tag{2.3}$$

and setting the derivative of E with respect to β to zero. Then, after applying some common matrix derivative identities:

$$\mathbf{0} = \left.\frac{dE}{d\beta}\right|_{\beta=\hat{\beta}} = \mathbf{t}\mathbf{X}^T - \beta\mathbf{X}\mathbf{X}^T \tag{2.4}$$

$$\beta = \mathbf{t}\mathbf{X}^T(\mathbf{X}\mathbf{X}^T)^{-1} \tag{2.5}$$

2.2.1 Quadratic Regression

Quadratic regression is much the same as linear regression, except that the matrix of inputs now has additional rows representing the squared inputs, so that it is now $[\mathbf{x}^2, \mathbf{x}]$, where $\mathbf{x}^2$ is the matrix of the squared inputs. Obviously, this can be generalized to even higher order terms if desired, or other indeed any other function of the inputs. However, for extremely nonlinear problems, a neural network may be preferable due to the fact that it assumes much less about the shape of the function.

2.3 Neural Networks

Neural networks are a powerful nonparametric regression technique well suited for nonlinear problems. A neural network consists of "nodes". Each node is connected to other nodes—the "strength" of such a connection is determined by a variable called the weight. These weights are adjusted during the training phase. In figure 2-1, w_{ij} and w_{jk} are two weights. The nodes themselves take in a scalar input and applies a so-called activation function before passing on the outputs to the next node it is connected to. The output is multiplied by the weight of that connection before being used as the input of the next node.

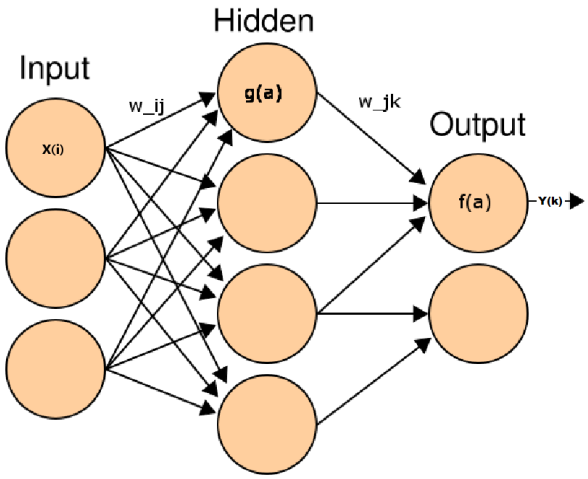


Figure 2-1: A simple neural network

Multiple nodes are arranged into layers, with all nodes in a particular layer (except

for the final output layer) having connections to nodes in the next layer closer to the output layer. All layers besides the input and output layers are usually referred to as hidden layers, and the nodes in them as hidden nodes. All nodes in a layer have the same form of activation function. In figure 2-1, $f(a)$ and $g(a)$ are two different activation functions.

Although the current neural network uses only one hidden layer, the neural network can have an arbitrary number of layers. However, increasing numbers of layers leads to increasing numbers of local minima when training. It has also been shown that a neural network with a single hidden layer can approximate any real-valued continuous function to arbitrary accuracy given enough hidden nodes [5]. From previous work on problems similar to the ones discussed here, there is also evidence that having a single hidden layer with enough hidden nodes does not significantly change performance on retrieval problems compared to having multiple hidden layers [6] [1]. Thus, for the rest of the thesis, all the network structures have one hidden layer.

Once the structure of the neural network has been selected, the network must be trained to fit the data. It is useful to first introduce some notation. Let the input layer I have nodes with activation function $f(a)$, and let hidden layer H have activation function $g(a)$ (for regression problems like ours, $g(a)$ is often of the form $\frac{1}{1+e^{(-a)}}$, and $f(a)$ is linear). Then, given an d-dimensional input $\mathbf{x} = [x_1, ..., x_i, ...x_d]$, the c-dimensional output $\mathbf{y} = [y_1, ..., y_k, ..., y_c]$ of a one hidden layer neural network can then be expressed as

$$y_k = \sum_{j \in H} f(w_{jk} a_j) \tag{2.6}$$

$$a_j = \sum_{i \in I} g(w_{ij} x_i) \tag{2.7}$$

where w_{jk} is the weight between hidden node j and output node k, and w_{ij} is the weight between input node i and hidden node j. Often, $\mathbf{y}$ is written as $\mathbf{y}(\mathbf{x})$ to emphasize that $\mathbf{y}$ represent the neural network function of $\mathbf{x}$, although here this is omitted for brevity.

During conventional neural network training, the goal is to minimize a cost func-

tion over a training dataset D. The usual cost function is the sum-squared error

$$E(\vec{w}) = \frac{1}{2}\sum_{i=1}^{n}(\mathbf{t}_i - \mathbf{y}_i)^2 \tag{2.8}$$

where $\vec{w}$ is a vector of all the weights in the neural network, and $\mathbf{t}_i$ is the target vector for the input vector $\mathbf{x}_i$. Sometimes, a regularization term $\frac{\alpha}{2}\|\vec{w}\|^2$ is added to prevent weights from becoming too extreme. Although I did not choose to do this for network training, weight regularization will be a factor later on in Bayesian neural networks.

It is easy to find the gradients $\frac{\partial E(\vec{w})}{\partial \vec{w}}$ (see Appendix A.1 for details). Conventional training methods are then mostly variations of gradient ascent, or some other function optimization method, to minimize the cost function over the space of the weight vector $\vec{w}$. Common training algorithms used in this thesis are Levenberg-Marquardt and scaled conjugate gradient, although those are certainly not the only possibilities.

2.3.1 Overfitting

Because there are so many more parameters in neural networks than in linear regression, and the function is not nearly as constrained, there exists the possible problem of overfitting. This is the problem where a function models the training data extremely well due to learning features of the data that are only present in the training dataset (random noise, for example), but does much poorer on test datasets. As a crude example, if there are d parameters in a linear regression (d dimensions), and only d data points, the linear regression can model the data perfectly, but is very unlikely to extrapolate well since it has not really learned anything beyond the training data.

To prevent overfitting in the neural network, early stopping was used. The RMSE (or the relevant performance metric) of a separate validation set of roughly the same size as the test set was evaluated at each iteration of the optimization process, and if the performance metric increased instead of decreased for too many iterations in a row, training was stopped. Although there have been no strong theoretical justification for this in the literature, there has been plenty of empirical support for the efficacy of early stopping, and it is ubiquitous in many practical applications [6] [5] [20]. It

can been seen as a form of regularization, leading to "smoother" functions [20].

Because early stopping proved so successful at preventing overfitting, I used early stopping on all the later methods discussed in this thesis as well.

2.3.2 Neural Network parameters

The number of nodes in the hidden layer was determined fairly arbitrarily. Limited testing with different numbers of nodes was done to ensure that the number of nodes was not too small as to cause the network to underfit, but it is quite possible that the number of nodes is more than is actually needed for a good fit to the data (early stopping should lessen the possibility of overfitting though). Based on that and earlier work on using neural networks on similar datasets, I chose to use 20 hidden nodes.

The neural network weights were initialized using the Nguyen-Widrow method. This initialization has been found to speed up convergence during training on many problems [15].

2.4 Results and Description of Datasets

In the following, I use as a performance metric the root mean square error (RMSE) of the test cases. If the set of cases D are enumerated as input-target pairs $(\mathbf{x}_i, t_i)$ and the predictions of the method as $y(x_i)$, where $i \in D$, then:

$$\mathrm{RMSE}(D) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y(\mathbf{x}_i) - t_i)^2} \tag{2.9}$$

If the mean of the residuals $y(\mathbf{x}_i) - t_i = 0$, then the RMSE is exactly equal to the standard deviation of the residuals.

2.4.1 ECMWF/Aqua dataset description

The ECMWF(European Center for Medium Ranged Weather Forecasts)/Aqua dataset consists of 33198 training cases, 4149 validation, and 4149 test cases (the test cases

were obtained by taking every fifth case from the training set, then removing those cases from the training set; the validation cases were obtained in a similar manner). The inputs are the 25 most significant principal components [1] of the 588 stochastically cloud cleared radiances [2] measured by the operational AIRS (**A**tmospheric **I**nfra**R**ed **S**ensor) and AMSU (**A**dvanced **M**icrowave **S**ounding **U**nit) sensors aboard the NASA Aqua satellite.

The targets are temperature and water vapor concentration. The European Center for Medium Ranged Weather Forecasts (ECMWF) provides the parameters used as the truth. Both the inputs and the truth are available for 60 distinct pressure levels, ranging from just above the surface (about 1013 mb) to the stratosphere (0.1 mb).

The dataset was subsampled from a much larger set, so only near-nadir measurements (the satellite was directly overhead) taken over "ocean" (which includes large inland seas and lakes as well) between 60N and -60S latitude were included. Ice-covered samples were deliberately excluded, due to the different radiances obtained.

Furthermore, because this dataset contains actual radiances from the satellite, no additional simulated instrument noise was added to the outputs.

The *a priori* standard deviation of temperature is shown on figure 2-2. In a sense, this also represents the worst possible RMSE for a statistical retrieval method, since the standard deviation would be the RMSE achieved if the temperature estimation for every case was simply the mean.

Water vapor was expressed as mass mixing ratio (a dimensionless unit), but normalized by the standard deviation of the water vapor at each pressure level, so that the *a priori* standard deviation is unity throughout the atmosphere.

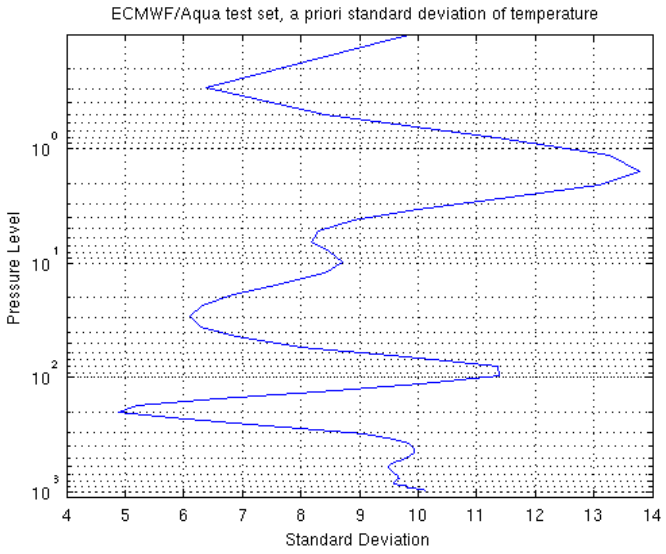


Figure 2-2: This is the *a priori* standard deviation of temperature on the ECMWF/Aqua dataset. The x-axis is in degrees kelvin, while the y-axis is in millibars.

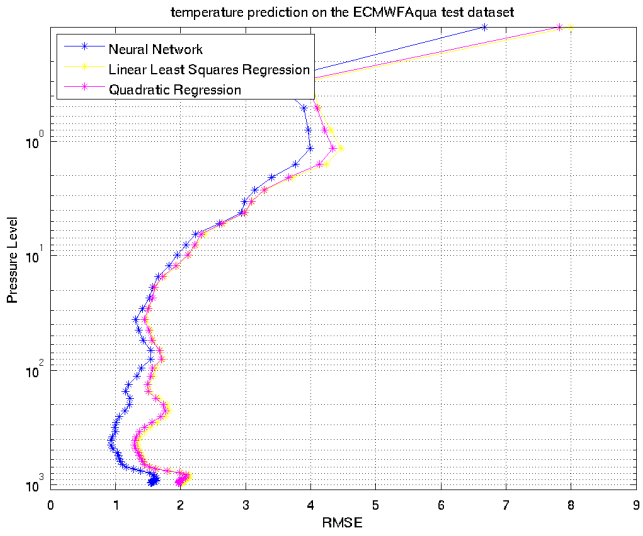


Figure 2-3: This figure shows the RMSE profile of several methods estimating temperature on the ECMWF/Aqua dataset. The RMSE is in degrees kelvin. The pressure is in millibars, with the surface at the bottom of the chart.

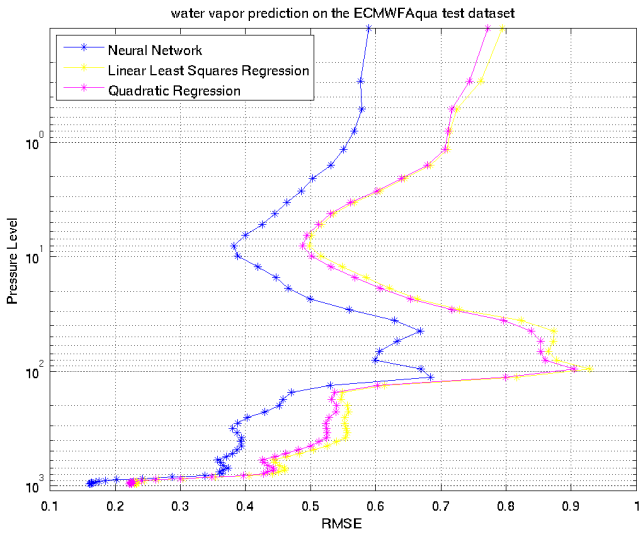


Figure 2-4: This figure shows the RMSE profile of several methods estimating water vapor on the ECMWF/Aqua dataset. The RMSE is in normalized mass mixing ratio. The pressure is in millibars, with the surface at the bottom of the chart.

2.4.2 ECMWF/Aqua results

Temperature estimation on the ECMWF/Aqua dataset is a fairly linear problem, with the neural network showing only a small increase in performance over either linear or quadratic regression (see figure 2-3). Surprisingly, quadratic regression performs much the same as linear regression on this problem (both on the test set and on the training set) perhaps indicating the non-linear aspects of the problem involve higher-order terms than just 2nd order.

Water vapor estimation on the ECMWF/Aqua dataset is much more nonlinear than temperature estimation (see figure 2-4). The neural network does much better than either linear or quadratic regression. Overall then, water vapor is a more challenging problem for statistical retrieval methods.

[1]The principal components are a linear transformation and dimensional reduction that best preserves the original data. The principal components are ranked by the amount they contribute to the variance in the original data, as judged by the eigenvalues of the covariance matrix of the input dimensions.

[2]The details of the stochastic cloud clearing algorithm can be found in the 2006 Cho and Staelin paper [9].

2.4.3 HyMAS dataset description

The HyMAS (the name comes from the Hyperspectral Microwave Atmospheric Sounder [8]) dataset consists of 30000 training cases, 5000 validation cases, and two different test datasets. The first, which I will call simply the test set, consists of 5000 cases and is obtained by taking every fifth sample from the training cases, much like in the case of the ECMWF/Aqua dataset. The second, which I will call the "golden days" set, consists of 40000 cases that are taken from several years not represented in the training set at all. It is known as the "golden days" dataset due to its usage as a benchmark of sorts by the AIRS science team. This "golden days" set should be a much more independent test set.

The inputs are the 25 most significant principal components of 88 simulated radiances from the hypothetical HyMAS instrument, simulated from ECMWF ground truth. It is again only over ocean between -60S and 60N in latitude, and is simulated with only nadir soundings. There are 96 distinct pressure levels, from near the surface (1013 mb) to the stratosphere (roughly 0.0384 mb). However, I only examined every fifth pressure level for temperature. For water vapor, I examined every fourth pressure level, but only starting from roughly 18 mb to the surface.

Since this was a simulated dataset, simulated instrument noise (random samples from a Gaussian with mean 0 and standard deviation equal to the predicted instrument noise) was added to the "clean" simulated radiances to create the training, testing, and golden days datasets ultimately used to train and test the various methods. For the neural network methods, different samples were added to the "clean" radiances from the training set at every iteration during training. Such a procedure was found to improve neural network accuracy and prevent overfitting [7].

The *a priori* standard deviation of temperature is shown in figure 2-5.

As before, water vapor is normalized mass mixing ratio, so *a priori* standard deviation is unity throughout the atmosphere.

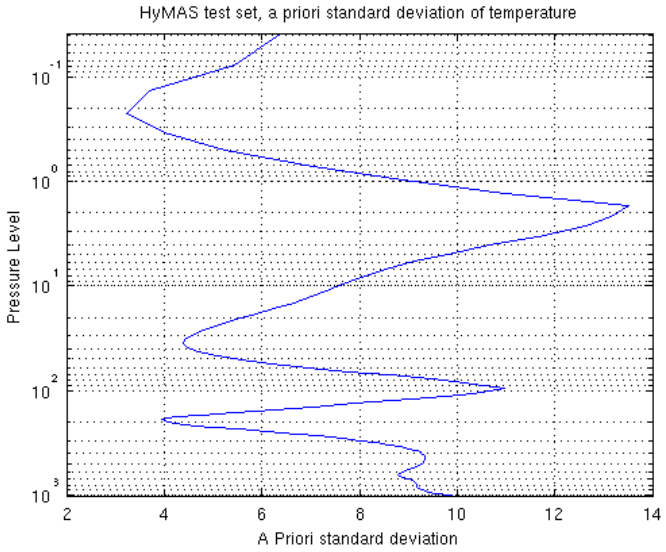


Figure 2-5: The *a priori* standard deviation of temperature on the HyMAS dataset. x-axis is in degrees kelvin, y-axis is millibars.

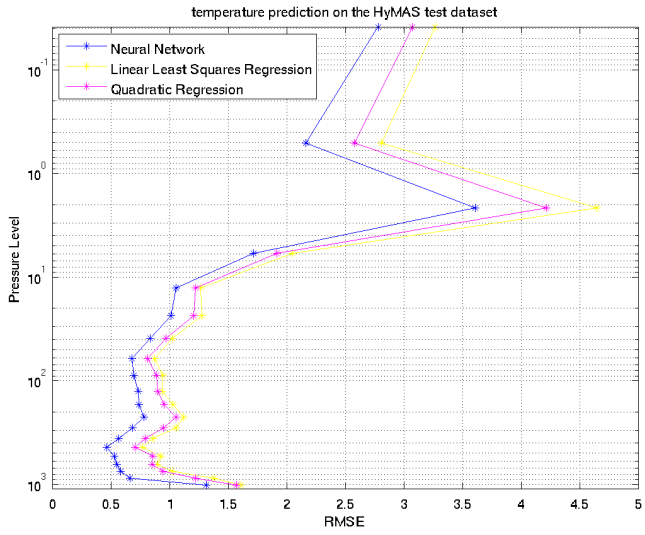


Figure 2-6: This figure shows the RMSE profile of several methods estimating temperature on the HyMAS dataset. The RMSE is in degrees kelvin. The pressure is in millibars with the surface at the bottom of the chart.

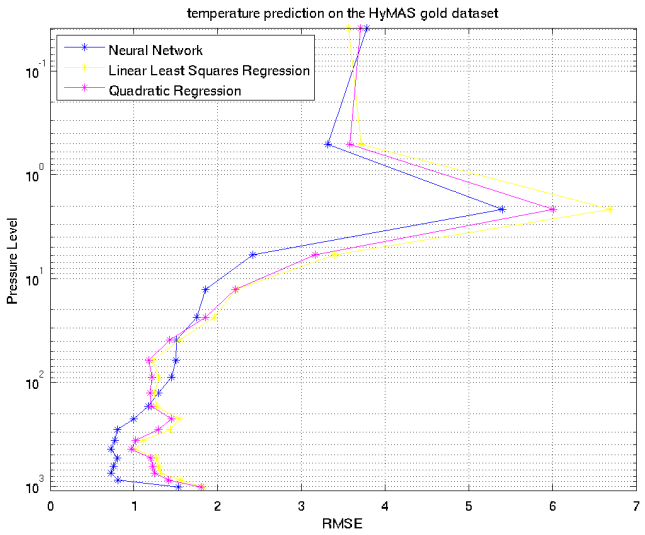


Figure 2-7: This figure shows the RMSE profile of several methods estimating temperature on the HyMAS "golden days" dataset. The RMSE is in degrees kelvin. The pressure is in millibars with the surface at the bottom of the chart.

2.4.4 HyMAS dataset results

The temperature estimation performance on the training and the test dataset (see figure 2-6) is reminiscent of the results on the ECMWF/Aqua dataset. The neural network does slightly better than either linear or quadratic regression. It seems having a hyperspectral microwave sensor as opposed to a combination of infrared and microwave radiances does not appreciably change the linearity of the problem. The real surprise comes from the performance on the golden days test set (see figure 2-7). There are several pressure levels between 23 mb and 170 mb where quadratic and linear regression outperform neural networks (note that all methods perform worse in terms of RMSE on the golden days dataset). This suggests that the neural network is overfitting to some features that were present in the training set but not the golden days dataset, perhaps due to the training set not being comprehensive enough. Overall, the golden days set should also provide an interesting test for variance estimation methods; good estimation methods should assign high uncertainty to cases which are not similar to those in the training set.

The relative performance of the methods on estimating water vapor is very similar

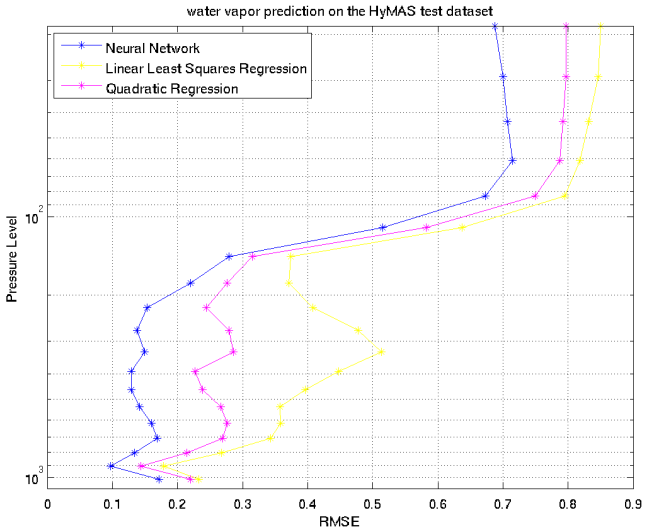


Figure 2-8: This figure shows the RMSE profile of several methods estimating water vapor on the HyMAS dataset. The RMSE is in units of normalized mass mixing ratio. The pressure is in millibars with the surface at the bottom of the chart.

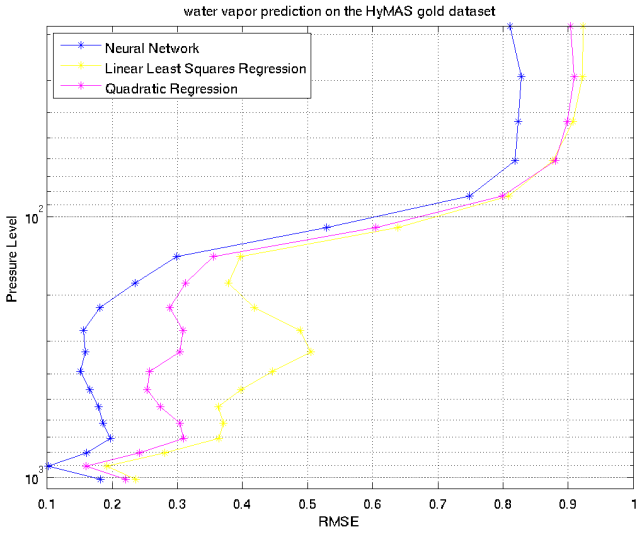


Figure 2-9: This figure shows the RMSE profile of several methods estimating water vapor on the HyMAS "golden days" dataset. The RMSE is in units of normalized mass mixing ratio. The pressure is in millibars with the surface at the bottom of the chart.

on the golden days and the test dataset (see figures 2-8 and 2-9). In both, the neural network significantly outperforms quadratic regression, and quadratic regression significantly outperforms linear regression. This is expected, as water vapor estimation is a very nonlinear problem. The lack of any significant difference in relative performance of the methods between the golden days and the regular test set suggests that the water vapor problem is sufficiently complicated that overfitting is not as large a concern here as underfitting.

2.4.5 Precipitation dataset description

The precipitation dataset is subsampled from the Pennsylvania State University - National Center for Atmospheric Research Mesoscale Model (MM5), a database of 106 storms. The inputs are 8 simulated AMSU-A and 5 AMSU-B microwave sounder radiances. The targets are the surface precipitation in millimeters per hour. There are a total of 38266 cases, of which 29762 were used as training cases, and the remaining cases split evenly between test and validation sets.

2.4.6 Precipitation dataset results

Rain rate retrieval is a nonlinear and fairly hard problem. Competitive algorithms often have multiple stages, based on factors such as terrain type, latitude, and temperature radiances of specific channels [23]. Because I was primarily interested in using this dataset to illustrate the differences between the methods for estimating variance, rather than accurate rain rate retrieval, I did not use any pre or post-processing except for a principal components transform of the inputs. Thus, the performance of the following retrievals can certainly be much improved.

As in the Surussavadee and Staelin paper [23], the performance of the methods is shown by the RMSE binned by the actual precipitation values (see table 2.1). The total RMSE in precipitation retrieval is often heavily biased by cases with the highest rainrates, and the majority of cases have no precipitation. So although linear regression and quadratic regression are only slightly worse than neural networks when

MM5 Range (mm/hr)	Number of Cases	Neural Network RMSE	Linear Regression	Quadratic Regression
[0 0.125)	2875	0.4025	0.9712	0.6223
[0.125 0.25)	212	0.9885	1.59	1.43
[0.25 0.5)	226	2.004	1.81	2.00
[0.5 1)	301	1.192	1.50	1.19
[1 2)	279	1.792	1.76	1.82
[2 4)	195	2.666	2.35	2.49
[4 8)	100	5.037	3.81	4.98
[8 16)	40	5.964	7.04	7.16
[16 32)	17	12.85	14.68	16.71
[32 75)	7	25.29	31.45	20.77
All	4252	**1.911**	**2.27**	**2.02**

Table 2.1: Precipitation retrieval performance of neural networks, linear regression, and quadratic regression.

all cases are considered, the RMSE is in general much higher than neural networks in cases with relatively low rainfall, which also happens to make up the majority of the cases. A more principled approach to "patch" this flaw might be to introduce a classifier to separate out precipitating from non-precipitating cases. However, I chose not to both because it would add extra complexity that would not directly contribute to determining the best method for variance estimation later, and because this introduces nonlinearities that might serve as an interesting test for the later methods in testing retrieval accuracy.

2.5 Conclusion

The results from these datasets show that neural networks generally offer substantial improvements over linear and quadratic regression, making neural network performance a good baseline for comparison. The temperature estimation problems are for the most part much more linear than the water vapor estimation, which is useful to keep in mind as the performance of various statistical retrieval techniques are evaluated. Precipitation retrieval is also a highly nonlinear problem, although part of the

reason is due to the large number of non-precipitating cases.

An important theme underlying the results is the tradeoff between the power of the model and the danger of overfitting. Techniques such as early stopping and weight regularization, as well as cruder methods like reducing the number of inputs or hidden units (in the case of neural networks) can force a smoother, more regularized output function that is less prone to overfitting. However, too much regularization can also lead to worse performance, as shown by the performance of the much simpler linear regression versus the neural network on complicated problems like water vapor estimation.

Chapter 3

Variance Estimation Neural Networks and Bayesian Neural Networks

Although neural networks are very successful at estimating the geophysical parameters, there is no guarantee of the quality of any individual prediction, such as an estimate of the full probability distribution of the outputs, or an estimate of the variance (equally as informative if we assume that the output distribution is Gaussian). Such an estimate could be used in many ways. One important application is based on the operation of the current operational AIRS algorithm, which divides its "data products" (the retrieved geophysical parameters) into three quality categories, varying in their estimated reliability [10]. Even this relatively coarse division into only three categories allows the AIRS data to be used with more confidence, so that it is much more likely to be incorporated into numerical weather prediction models and help make forecasts more accurate [17]. A good variance estimation scheme can conceivably improve on this, allowing users to divide the retrieved geophysical parameters into as many quality categories as they wished.

Contributors to increased variance and error in remote sensing retrieval can come from many sources. There is the inherent noisiness of the measurements, the non-uniqueness of possible outputs due to retrievals being an inverse problem, and other

physical sources of noise like dust or volcanic ash interfering with the radiance measurements. There is a good chance that the influence of some of these sources of noise are dependent on the input radiances measured. For example, some radiances are known to be sensitive to dust, and noise due to dust would then be dependent on the inputs having that "dust" characteristic. A function with such input-dependent variance is known as a heteroscedastic function.

Another possible source of variance in statistical retrieval methods could be due to uncertainty in the parameters of the model. For example, lack of training data could lead to an ill-posed problem. A good variance estimation method should assign higher variances for outputs where the lack of training data might affect performance.

Aires has demonstrated Bayesian neural network techniques to give confidence intervals, but the main focus of the paper was on the uncertainty in the neural network, rather than the total uncertainty in the retrieval [1]. Although the reliability of the retrieval method is certainly very useful to know, there are many other factors that can affect the quality of a retrieval that are not accounted for.

Thus, an ideal variance estimation method should account for both heteroscedastic variance and model uncertainty. In this chapter, I will introduce first a baseline technique for estimating variance, the variance estimation neural network, and then describe Bayesian neural networks, a method which has been used with some success before to estimate variance in the remote sensing context.

3.1 Variance Estimation Neural Networks

One simple way to estimate the variance is to treat the variance as an target, and try to estimate it on the basis of the inputs, essentially turning variance estimation into another regression problem. This regression problem can then be solved by another neural network, which I termed a "variance estimation neural network". The variance estimation neural network takes the same inputs as the original neural network, but the targets are now the square of the residuals of the first neural network (on the training cases). Given enough training cases, the square of the residuals should

approach the true variance of the data.

The advantage of this approach is that it can easily take into account heteroscedasticity. However, it would have no conception of model uncertainty, since it would be solving an almost unrelated regression problem. It is possible that if there is a lack of training data only in certain areas of the input space, the extra uncertainty can be picked up by the variance estimation neural network. For example, if data in the polar region are lacking (which will cause large residuals in the first neural network), the variance estimation neural network can account for that noise.

In the problems, I used a variance estimation neural network with 10 hidden nodes, based on testing on a few sample problems. The lower number of nodes was chosen to discourage overfitting to the squares of the residuals, and also because a higher number of nodes did not improve performance on the sample problems tested. Intuitively, it seems plausible that the variance should be a far smoother function of the inputs than the mean (the mean in this case is the geophysical parameter). The weights were initialized using the Nguyen Widrow algorithm, the same way as the regular neural network.

Overall, the simplicity and ease of implementation make the variance estimation neural network a good benchmark to judge the performance of more complicated variance estimation methods. Another approach that is perhaps more theoretically justified, is made to account for parameter uncertainty, and has been used with success in the past [1] is Bayesian neural networks.

3.2 Introduction to Bayesian Neural Networks

As stated before, during conventional neural network training, the goal is to minimize a cost function of the weights $\vec{w}$ over a set of training examples D. If we think of the possible weights as a distribution over a multi-dimensional "weight space", the resulting weights $\vec{w}$ that are found by conventional training are the most likely weights given the training data D.

3.2.1 Bayesian Methods

In Bayesian training, the goal is to find the entire distribution of $p(\vec{w}|D)$, instead of just the most likely weights, the $\arg\max p(\vec{w}|D)$. Because Bayesian training does not explicitly model the distribution of the inputs $\mathbf{x}$, it is convenient to separate D into T, the set of targets, and X, the set of inputs $\mathbf{x}$. All the following probabilities are implied to be conditional on X. We can now write

$$p(\vec{w}|T) = \frac{p(T|\vec{w})p(\vec{w})}{p(T)} \tag{3.1}$$

where $p(\vec{w})$ is a prior distribution that reflects our belief on $\vec{w}$ before we see the data. Usually, this prior distribution is taken to be a Gaussian to simplify calculations, although it can be any probability distribution. $p(T) = \int_{\vec{w}} p(T|\vec{w})p(\vec{w})d\vec{w}$ is just a normalization factor, so the focus will be on the terms in the numerator.

Once we have evaluated equation (3.1), we can use the distribution $p(\vec{w}|T)$ to compute the predictive distribution of an output t given an input $\mathbf{x}$.

$$p(t|\mathbf{x}, T) = \int p(\mathbf{t}|\mathbf{x}, \vec{w})p(\vec{w}|T)d\vec{w} \tag{3.2}$$

$p(t|\mathbf{x}, \vec{w})$ is a measure of the intrinsic noise in the data, and is reviewed in the next section.

3.2.2 Intrinsic Noise

$p(T|\vec{w}) = p(T|\vec{w}, X)$ can be rewritten as $p(t_1, ...t_j..., t_n|\vec{w}, X)$. Since the training samples are independent, this can be written as

$$p(T|\vec{w}) = \prod_{j=1}^{n} p(t_j|\mathbf{x}_j, \vec{w}) \tag{3.3}$$

The individual probabilities $p(t_j|\mathbf{x}_j, \vec{w})$ were seen earlier in equation (3.2), and represent the variability of t_j based on intrinsic error or noise, for fixed values of $\vec{w}$. Essentially, we assume that the target observations t are noisy, and even if we model

the underlying function $h(\mathbf{x})$ perfectly with the neural network, the actual targets $t = h(\mathbf{x}) + \epsilon$, where ϵ is the intrinsic noise. Then the probability of seeing the target observation t given $\mathbf{x}$ is simply $p(\epsilon)$. In other words

$$p(t|\mathbf{x}, \vec{w}) = p(\epsilon) \tag{3.4}$$

In the literature, it is assumed that the intrinsic error is simply a zero-mean Gaussian noise, so ϵ comes from the distribution

$$p(\epsilon) = \frac{1}{Z} \exp(-\beta \epsilon^2) \tag{3.5}$$

Z is a normalization constant, and β is the inverse variance. Then, if the neural network is a good fit for $h(\mathbf{x})$ (so that we can replace $h(\mathbf{x})$ with $y(\mathbf{x})$, the neural network function), we can write $\epsilon = t - h(\mathbf{x})$, and consequently

$$p(t_j|\mathbf{x}_j, \vec{w}) = p(\epsilon) = p(y_j(\mathbf{x}_j, \vec{w}) - \vec{t_j}) = \frac{1}{Z} \exp(-\beta (y_j(\mathbf{x}_j, \vec{w}) - \vec{t_j})^2) \tag{3.6}$$

Substituting in equation (3.6) into equation (3.3), we get that

$$p(T|\vec{w}) = \prod_{j \in D} p(t_j|\mathbf{x}_j, \vec{w}) = \frac{1}{Z_D} \exp(-\beta \sum_{j \in D} (y_j(\mathbf{x}_j, \vec{w}) - \vec{t_j})^2) = \frac{1}{Z_D} \exp(-\beta E(\vec{w})) \tag{3.7}$$

where $Z_D = Z^n$, and the mean error term

$$E(\vec{w}) = \sum_{j \in D} (y_j(\mathbf{x}_j, \vec{w}) - \vec{t_j})^2 \tag{3.8}$$

Note that $E(\vec{w})$ is simply the cost function in conventional training (equation (2.8)).

3.2.3 Priors

The prior $p(\vec{w})$ represents a guess at the distribution of weights before seeing the training data. Because such a guess is often difficult, the most popular priors $p(\vec{w})$

are either the uniform distribution, or a simple Gaussian prior:

$$p(\vec{w}) = \frac{1}{Z_R} \exp(\frac{-\alpha}{2} \|\vec{w}\|^2) \tag{3.9}$$

where Z_R is the appropriate normalization constant. α is a scalar constant to be discussed later. A Gaussian prior prevents any weight from growing too large and is equivalent to regularization in conventional training.

If we assume that the prior is Gaussian, we can use equations (3.7) and (3.9) to write equation 3.1 as

$$p(\vec{w}|T) = \frac{1}{Z_S} \exp(-\beta \sum_{j \in P} (y_j(\mathbf{x}_j, \vec{w}) - \vec{t_j})^2 - \frac{\alpha}{2} \|\vec{w}\|^2) = \frac{1}{Z_S} \exp(S(\vec{w})) \tag{3.10}$$

where Z_S is the appropriate normalization constant (had we used a uniform prior, we would simply remove the term $\frac{-\alpha}{2} \|\vec{w}\|^2$ from equation (3.10)). It is convenient to write the term in the exponential as simply $S(\vec{w})$, so that

$$S(\vec{w}) = -\beta E(\vec{w}) - \frac{\alpha}{2} \|\vec{w}\|^2 \tag{3.11}$$

3.2.4 Evaluating the Output Distribution

Substituting in equations (3.7) and (3.10) into equation 3.2, we get

$$p(t|\mathbf{x}) = \frac{1}{Z_S} \int p(-\beta(y(\mathbf{x}, \vec{w}) - t)^2) p(S(\vec{w})) d\vec{w} \tag{3.12}$$

Unfortunately, this integral is sufficiently complicated that analytic integration is impossible. In the literature, there are two major approaches to dealing with this problem. One is to approximate $\exp(S(\vec{w}))$ as a Gaussian by replacing $S(\vec{w})$ with its second-order Taylor expansion. Afterwards, equation (3.12) can be evaluated analytically. This approach, pioneered by Mackay, has been used in various "real world" applications [14] [25] [1]. Indeed, this is the approach used by Aires to train a neural network for a remote sensing task similar to the problems studied here [1].

This was also the approach I ultimately decided to use on the various datasets.

An alternative is to numerically integrate equation (3.12) using Markov Chain Monte Carlo (MCMC) techniques. Radford Neal tried this using hybrid Monte Carlo (HMC) and it has been expanded upon by others [12] [11]. However, it is invariably slower than the approximation method. As far as I know, this method has not been tried in any large scale problem, due to the computational cost of using MCMC methods on a probability distribution in a high-dimensional space. Because the computational cost was simply too high compared to regular neural networks (on a small test problem of 1000 training samples, the HMC approach took nearly 20 times as long as Gaussian approximation approach to get values for the hyperparameters), I did not pursue MCMC methods any further.

3.2.5 Gaussian Approximation Method

To approximate $\exp(S(\vec{w}))$ as a Gaussian, we replace $S(\vec{w})$ with its second-order Taylor expansion around a minima (this minima is usually found by conventional neural network training methods). If we let $\vec{w}_{min}$ denote the $\arg\min S(\vec{w})$, we can then write

$$S(\vec{w}) = S(\vec{w}_{min}) + \nabla S(\vec{w})\Delta\vec{w} + \Delta\vec{w}^T(\nabla\nabla S(\vec{w}))\Delta\vec{w} \tag{3.13}$$

where $\Delta\vec{w} = (\vec{w} - \vec{w}_{min})$. Note that at a minima, the gradient $\nabla S(\vec{w}) = 0$ by definition. The Hessian $\nabla\nabla S(\vec{w}_{min})$ is composed of the Hessian of the error $E(\vec{w}_{min})$ (see equation (3.8)) and a regularization term resulting from the Gaussian weight prior.

$$\nabla\nabla S(\vec{w}_{min}) = \beta\nabla\nabla E(\vec{w}_{min}) + \alpha\mathbf{I} \tag{3.14}$$

Next, $y(\mathbf{x}, \vec{w})$ is linearly approximated as $y(\mathbf{x}, \vec{w}_{min}) + \mathbf{g}\Delta\vec{w}$, where the gradient $\mathbf{g} \equiv \nabla_{\vec{w}} y$. Straightforward conventional methods exist to find the gradient $\mathbf{g}$ [5].

With those two approximations, the integral (3.12) can be analytically evaluated to give yet another Gaussian of the form

$$p(t|\mathbf{x}, T) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(t - y(\mathbf{x}, \vec{w}_{min}))^2}{2\sigma^2}\right) = N(y(\mathbf{x}, \vec{w}_{min}), \sigma^2) \tag{3.15}$$

Where the variance is:

$$\sigma^2 = \beta^{-1} + \mathbf{g}^T(\nabla\nabla S(\vec{w}))^{-1}\mathbf{g} \tag{3.16}$$

The standard deviation of the Gaussian, σ, can be used now as a confidence interval for the prediction $y(\mathbf{x}, \vec{w}_{min})$, which is the mean of the Gaussian. It is worthwhile to note that the function $y(\mathbf{x}, \vec{w}_{min})$ has weights $\vec{w}_{min}$ that are exactly the minima found by gradient descent during training. Thus, the Bayesian neural network prediction should be the same as any regular neural network with the same sum of squares error metric and weight regularization. But whereas a regular neural network gives only one number, the most probable output, the Bayesian neural network gives the entire distribution.

3.2.6 Hyperparameter Estimation

In the preceding sections, α and β have been treated as fixed constants, but in reality they are rarely known in advance and it would be wise to adjust them to best fit the data. In the literature, they are known as hyperparameters, so called because they control the distribution of the parameters (weights) [5]. In the experiments, I used Mackay's "evidence" approach to estimating the hyperparameters, although there are other possible approaches. The evidence approach assumes that the distribution $p(\alpha, \beta|T)$ is sharply peaked enough to be approximated by a delta function centered around the most probable hyperparameter values α_{MP}, β_{MP}. Then:

$$p(\vec{w}|T) = \int\int p(\vec{w}|\alpha, \beta, T)p(\alpha, \beta|T)\, d\alpha d\beta \tag{3.17}$$

$$\approx p(\vec{w}|\alpha_{MP}, \beta_{MP}, T) \tag{3.18}$$

which implies that the hyperparameters have no effect on the probability distribution of the weights, and thus no effect on the optimal weights chosen during training. Essentially, with the approximation of $p(\alpha, \beta|T)$ by a delta function, we can then find the optimal hyperparameters independently of finding the optimal weights.

To find the optimal hyperparameters, we start with an application of Bayes rule:

$$p(\alpha, \beta|T) = \frac{p(T|\alpha, \beta)p(\alpha, \beta)}{p(T)} \tag{3.19}$$

The denominator $p(T)$ is not dependent on α or β, and if we assume a uniform, improper prior $p(\alpha, \beta)$, then to maximize $p(\alpha, \beta|T)$ we only have to maximize the "evidence" term $p(T|\alpha, \beta)$. This is equivalent to the likelihood; it's somewhat intuitive that the best hyperparameters should be ones that maximize the likelihood of the training data.

Since $p(t_i|\mathbf{x}_i) = p(t_i|\alpha, \beta) = N(y_i, \sigma_i^2)$ (see equation (3.15)), we can write the likelihood as

$$p(T|\alpha, \beta) = \prod_{i=1}^{n} N(y_i, \sigma_i^2) \tag{3.20}$$

Unfortunately equation (3.20) is not easy to evaluate to maximize α and β. Instead, rewrite $p(T|\alpha, \beta)$ with explicit dependence on $\vec{w}$ to allow us to substitute terms that were previously evaluated:

$$\begin{aligned}
p(T|\alpha, \beta) &= \int p(T|\vec{w}, \beta)p(\vec{w}|\alpha)\, d\vec{w} && (3.21)\\
&= \int \frac{1}{Z_R} \exp(\frac{-\alpha}{2} \|\vec{w}\|^2) \frac{1}{Z_D} \exp(-\beta E(\vec{w}))\, d\vec{w} && (3.22)\\
&= \frac{1}{Z_R Z_D} \int \exp(\frac{-\alpha}{2} \|\vec{w}\|^2 - \beta E(\vec{w}))\, d\vec{w} && (3.23)\\
&= \frac{1}{Z_R Z_D} \int \exp(S(\vec{w}))\, d\vec{w} && (3.24)
\end{aligned}$$

where in the second step we substitute in the equations (3.7) and (3.9), and in the third step substitute in equation (3.11). Z_R and Z_D are defined in equations (3.9) and (3.7), respectively. If we continue to accept the Gaussian approximation of $\exp(S(\vec{w}))$, then $\int \exp(S(\vec{w}))\, d\vec{w}$ is the normalization term of a Gaussian. Then we can rewrite $p(T|\alpha, \beta)$ as:

$$p(T|\alpha, \beta) = \left(\frac{\alpha}{2\pi}\right)^{\frac{W}{2}} \left(\frac{\beta}{2\pi}\right)^{\frac{n}{2}} (2\pi)^{\frac{W}{2}} |\nabla\nabla S(\vec{w}_{min})|^{-\frac{1}{2}} \exp(-S(\vec{w}_{min})) \tag{3.25}$$

where W is the number of elements (weights) in the vector $\vec{w}$ and n is the number of cases in T. We can then take the derivative of this with respect to α and β to maximize $p(T|\alpha,\beta)$. In practice, it is easier to maximize $\ln(p(T|\alpha,\beta))$

$$\begin{aligned}\ln(p(T|\alpha,\beta)) = -\alpha(\|\vec{w}_{min}\|^2) - \beta E(\vec{w}_{min}) - \frac{1}{2}\ln|\nabla\nabla S(\vec{w}_{min})| \\ + \frac{W}{2}\ln(\alpha) + \frac{n}{2}\beta - \frac{n}{2}\ln(2\pi)\end{aligned} \tag{3.26}$$

If we assume that there is no dependence of the eigenvalues of $\nabla\nabla S(\vec{w}_{min})$ on α, the derivative with respect to α is

$$2\alpha\|\vec{w}_{min}\|^2 = W - \sum_{i=1}^{W}\frac{\alpha}{\lambda_i+\alpha} = \gamma \tag{3.27}$$

where λ_i is the i^{th} eigenvalue of $\nabla\nabla S(\vec{w}_{min})$ and

$$\gamma \equiv W - \sum_{i=1}^{W}\frac{\alpha}{\lambda_i+\alpha} = \sum_{i=1}^{W}\frac{\lambda_i}{\lambda_i+\alpha} \tag{3.28}$$

γ has an interpretation as the number of well determined weights, which are weights that are determined more by the training data than the weight prior.

The derivative with respect β is

$$2\beta E_{\vec{w}} = n - \gamma \tag{3.29}$$

Since we know the derivatives must be zero at the maximum, we can optimize the hyperparameters by periodically re-estimating them using the equations

$$\alpha = \frac{\gamma}{2\|\vec{w}_{min}\|^2} \tag{3.30}$$

$$\beta = \frac{n-\gamma}{2E_{\vec{w}_{min}}} \tag{3.31}$$

This re-estimation is usually done between iterations of our favorite neural network training algorithm (for these problems scaled conjugate gradients was used).

Finally, note that approximating $\exp(S(\vec{w}_{min}))$ with a Gaussian distribution is clearly not accurate, since there are multiple local minima. However, we can think of each local minima leading to different, but equally valid, interpretations of the data [14]. The hyperparameters chosen will then be specific to that particular local minima.

3.3 Results

To quantify the performance of the various methods, I sorted the cases by the variance predicted by a particular method. Then, I graphed the RMSE of the n percent of cases with the lowest predicted variance, so that it is easy to see at a glance the RMSE of the "best" n percent predicted by that method. This particular presentation was chosen partly based on ease of comparison to the way the operational AIRS algorithm divides the cases into quality categories. Because of the discrete categories, it is natural to throw out all cases with poor quality and calculate the RMSE of the leftover cases, something which is also easy to see in figures presented in this manner. I called this presentation style "graphed by cumulative RMSE".

Note that this presentation does not show the actual variance predicted, only the relative quality of the predictions. Therefore, I also used another presentation of the figures, where I sorted the cases into 20 bins based on their predicted variance, much as before. However, I then graphed the RMSE of each bin individually against the average predicted standard deviation of the cases in each bin. This style I called "graphed by RMSE by bin". The optimal variance estimation method should give a straight line such that the RMSE is equal to standard deviation. In a sense, the previous presentation style, showing the cumulative RMSE, is the "integral" of this presentation.

3.3.1 Variance as a function of latitude

For some of the problems, there is sometimes a high correlation of the variance with some function of the latitude. I thought it was interesting enough to include as a

baseline. In the figures, the function of latitude is given in the legend. The cases are then sorted by that function of the latitude, divided into bins, and the RMSE of the cases in those bins are plotted. Note that the latitude is *not* an input to any of the retrieval methods or variance estimation methods tested, although it is possible that it may improve performance were it an additional input.

3.3.2 Geophysical Parameter Prediction accuracy

The Bayesian neural network gives comparable accuracy to a regular neural network when estimating geophysical parameters like temperature and water vapor on the ECMWF/Aqua and the HyMAS datasets. Indeed, on the ECMWF/Aqua dataset the Bayesian neural network actually performs slightly better (see figure 3-1 and figure 3-2), suggesting that the weight regularization parameters, α, help to avoid overfitting (the HyMAS dataset includes added noise, but the ECMWF/Aqua dataset does not). Nevertheless, the overall difference is slight, as expected given that the early stopping should also help mitigate overfitting.

Another advantage of Bayesian neural networks is that the performance metric and the training is exactly the same as regular neural networks, save for some additional calculation required to re-estimate hyperparameters.

3.3.3 Variance prediction performance

I show the performance of the methods on one pressure level at a time. Each pressure level can be thought of as a separate problem, since one instance of each method is trained for each level. Although multitask learning could conceivably be applied in the case of the variance estimation neural network, that is a task for future work.

I used two main types of charts in this thesis. The first shows the cumulative RMSE of the cases, where the cases are sorted by their predicted variance. The y-values are the RMSE, while the x-values are the fraction of profiles used to calculate that RMSE. For example, the y-value at $x = .5$ corresponds to the RMSE of the profiles with a predicted variance less than the median predicted variance. The y-

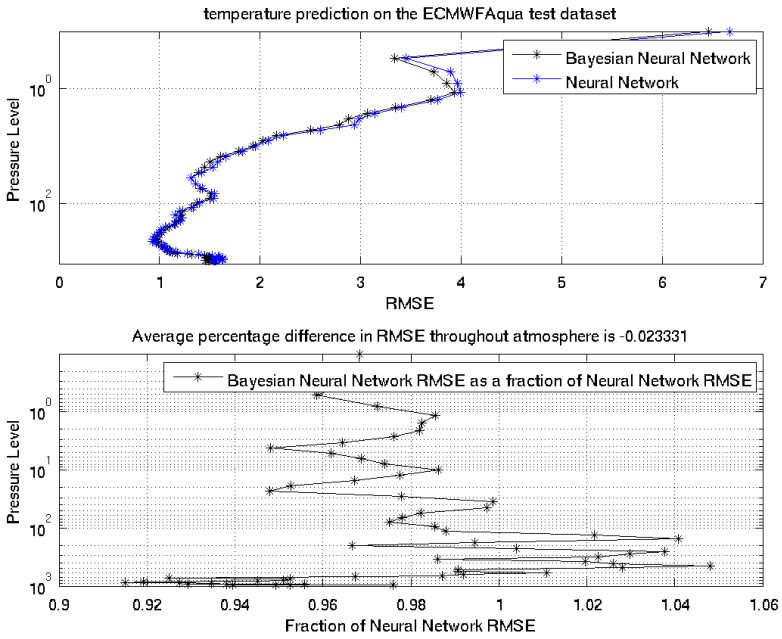


Figure 3-1: This figure compares the RMSE performance of Bayesian neural networks versus neural networks while predicting temperature on the ECMWF/Aqua dataset. RMSE is in degrees kelvin, and pressure is in millibars.

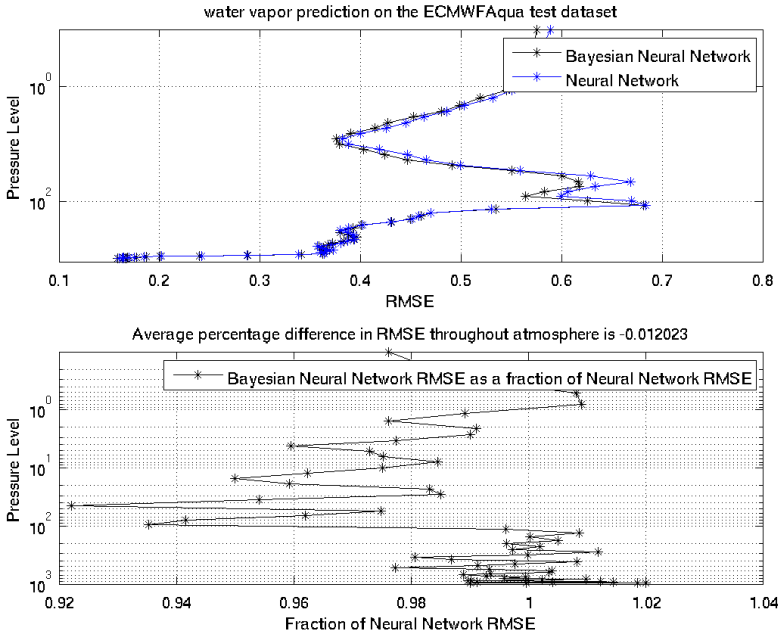


Figure 3-2: This figure compares the RMSE performance of Bayesian neural networks versus neural networks while predicting water vapor on the ECMWF/Aqua dataset. RMSE is in units of normalized mass mixing ratio, and pressure is in millibars.

value at $x = 1$ is simply the RMSE of all profiles. The advantage of this is that it is easy to see how removing cases with the highest variances affects the overall RMSE, suggesting how best to set up the quality categories like in the operational AIRS algorithm. The primary disadvantage is that cases with low variance have a disproportionate influence on the shape of the graph, due to the cumulative RMSE metric.

If the predicted variance is accurate, then it is expected that the RMSE of profiles with lower predicted variance should be lower than the RMSE of the profiles with higher predicted variance, so that the slope should be positive. For a given problem (or pressure level), the quality of a method at predicting variance can be judged by how steep the slope is. This represents how well the method can separate problematic, high-error cases from easy cases.

The second shows the RMSE of groups of cases with a particular predicted standard deviation, instead of the cumulative RMSE. More precisely, I sorted the cases by predicted variance and divided up the cases into 20 groups, each containing 5 percent of the total cases. The predicted standard deviation of each group was then graphed against the actual RMSE of cases in that group. The x-values are now the predicted standard deviations of that group of cases, while the y-values are the actual RMSE of those cases. If the groups of cases are large enough, the ideal method would have the predicted standard deviation equal to the actual RMSE, so as to form a straight line $y = x$. The advantage of this graph is that if a method is particularly bad at estimating the variance of cases with low RMSE, it does not skew the graph like with the cumulative RMSE graph. This graph also shows the actual predicted variance. On the other hand, this type of graph is much more prone to noise due to the smaller sizes of the groups of cases, and it is sometimes harder to tell what method is better because of that.

I will primarily judge performance based on how well the Bayesian neural network can separate out the cases with the lowest RMSE from the cases with the highest RMSE based on the predicted variance of the Bayesian neural network. By that standard, the variance prediction performance of Bayesian neural networks is

inconsistent on both the ECMWF/Aqua and the HyMAS datasets. For example, on the HyMAS water vapor estimation problem, on some levels, such as pressure level 459 mb, Bayesian neural networks show skill in separating out the higher RMSE cases from the easier ones (see figure 3-3). In fact, there are levels where the Bayesian

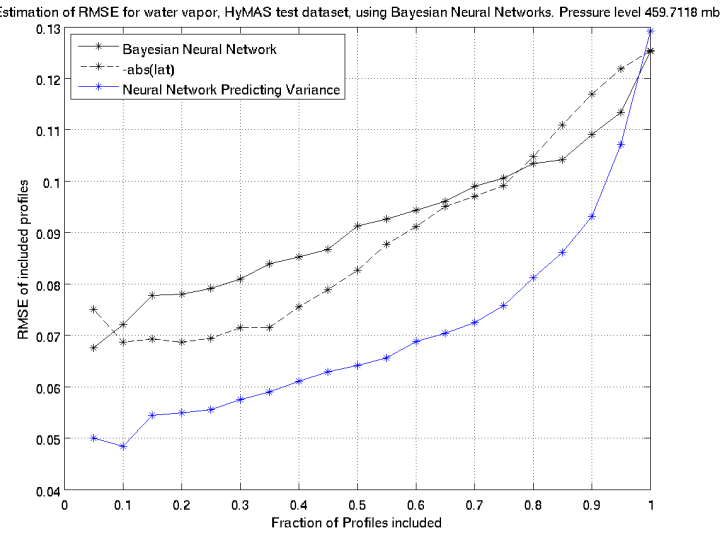


Figure 3-3: This figure shows the performance of the methods on estimating water vapor on the HyMAS dataset on pressure level 460 mb. RMSE is in normalized mass mixing ratio.

neural network's performance is as good or better than that of a variance estimation neural network (see figure 3-4). On other levels, the performance is mostly good except for a few cases that have high RMSE but relatively low predicted variance. A good example of that is on pressure level 706 mb (see figure 3-5). There, if the cases with the lowest 10 percent of the predicted variance (it is perhaps easier to see this on figure 3-6, which shows the RMSE of the cases instead of the cumulative RMSE) are ignored, the variances of the rest of the cases are better estimated; cases with higher variance prediction also have higher RMSE. As discussed later, this may be due to poorly estimated hyperparameters. On still others, such as pressure level 535 mb (see figure 3-7), the variance predicted by Bayesian neural networks show little correlation to the RMSE. Although these examples only showed HyMAS water vapor results, the results are typical across both temperature and water vapor prediction on both the HyMAS and the ECMWF/Aqua datasets.

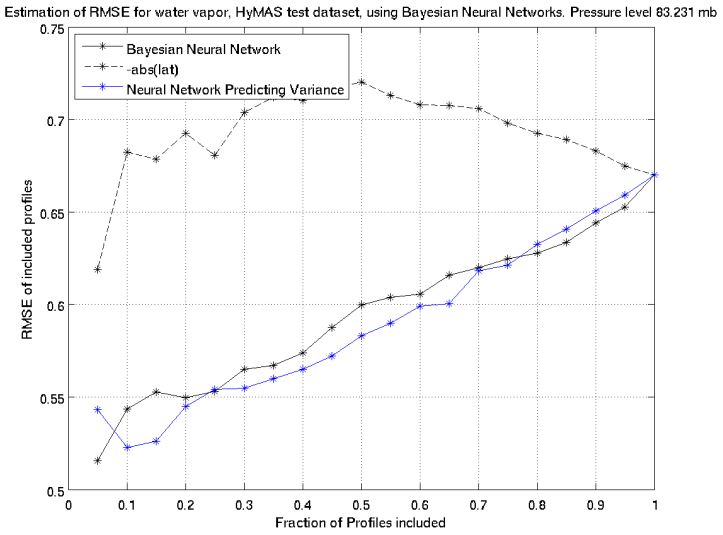


Figure 3-4: This figure shows the performance of the methods on estimating water vapor on the HyMAS dataset on pressure level 83 mb. RMSE is in normalized mass mixing ratio. See section 3.3.1 for an explanation of the black line (the function of latitude).

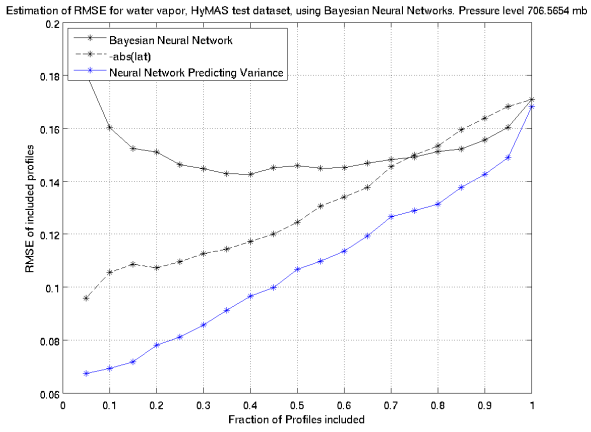


Figure 3-5: This figure shows the performance of the methods on estimating water vapor on the HyMAS dataset on pressure level 706 mb. RMSE is in normalized mass mixing ratio.

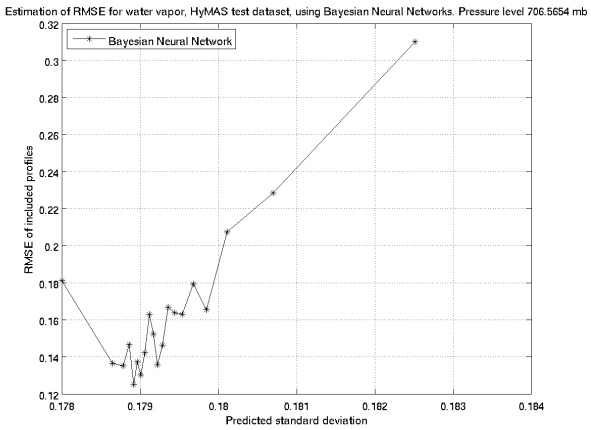


Figure 3-6: This is the same problem as depicted in figure 3-5, except that the RMSE of cases in each 5 percent bin is shown, instead of the cumulative RMSE of the cases in all previous bins. Predicted standard deviation is now also in units of normalized mass mixing ratio.

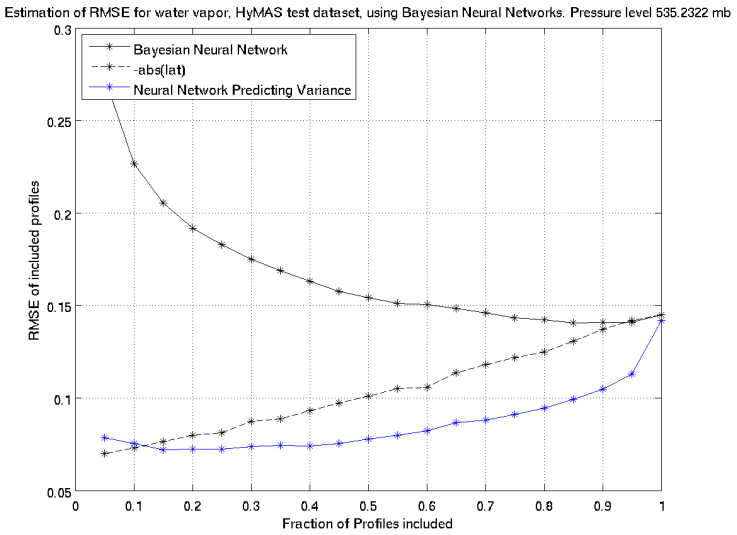


Figure 3-7: This figure shows the performance of the methods on estimating water vapor on the HyMAS dataset on pressure level 535 mb. RMSE is in normalized mass mixing ratio.

One important thing to note is that even when the Bayesian neural net shows skill at separating out harder from easier cases based on predicted variance, the predicted standard deviation itself is always in a very narrow range compared to the actual range of RMSE of those cases. A typical case is on pressure level 459 mb for the HyMAS water vapor prediction. There the predicted standard deviations ranges from roughly 0.132 to 0.141, but the RMSE spans a range from 0.07 to 0.27. Now, the RMSE for all cases on that particular levels is 0.125, which is close to the range predicted (in fact, β, the inverse of the variance, is equal to $\frac{1}{.122^2}$), but the Bayesian neural network clearly does not give accurate variance predictions for the cases, which can be a problem for applications where we are primarily interested in the variance, and not just in weeding out the worst performing cases or creating quality categories.

This small range of predicted variance is a common problem amongst all the datasets. Recall that the equation for variance was:

$$\sigma^2 = \beta^{-1} + \mathbf{g}^T(\nabla\nabla S(\vec{w}))^{-1}\mathbf{g} \tag{3.32}$$

Since β is constant across the different cases, the only thing that varies is the second

term $\mathbf{g}^T(\nabla\nabla S(\vec{w}))^{-1}\mathbf{g}$, which is only dependent on the uncertainty of the model parameters, $\vec{w}$, at that input point. Given how small the range of variances are, the model uncertainty must also be relatively constant throughout the input space, implying that the density of points in the 25 dimensional input space is also fairly constant (recall that the density, or sparsity, of inputs is a partial contributor to the model uncertainty [5]).

In a few cases, the Bayesian neural network even gives negative variance predictions. Normally, this would be impossible if a minima is found during network training. But because we use early stopping, we may not stop at a minima. Although these negative variance predictions are few, their utility is questionable since they represent a failure in the approximations, so it's not obvious whether the actual variance of that case is high or low. With the variance estimation neural network, a negative variance prediction unambiguously indicates the actual variance of the case is thought to be extremely low. Luckily, negative variance predictions by the Bayesian neural network are extremely few, numbering no more than 0.1 percent of cases.

Given the problems that the Bayesian neural network has on some levels in modeling variance, or even just separating out the cases, it is apparent that model uncertainty alone is not enough to account for the variance. Heteroscedasticity must be considered as well, which the Bayesian neural network cannot model, but which the variance estimation neural network can. Moreover, the various approximations made to simplify calculations must also degrade the final performance of the Bayesian neural network.

Still, despite all that, the Bayesian neural network is clearly very successful on some pressure levels of the problems, as mentioned above (see figure 3-4), if not for all the pressure levels. Its performance is even stable when confronted with data not originally in its training set, as evidenced by its performance on the HyMAS "Golden Days" dataset (an example level is shown in figure 3-8). On that example, the Bayesian neural network performance on the golden days set is similar to the performance on the test set (see figure 3-3). This suggests that whatever correlation it has discovered between the inputs and the variance was not just due to artifacts in

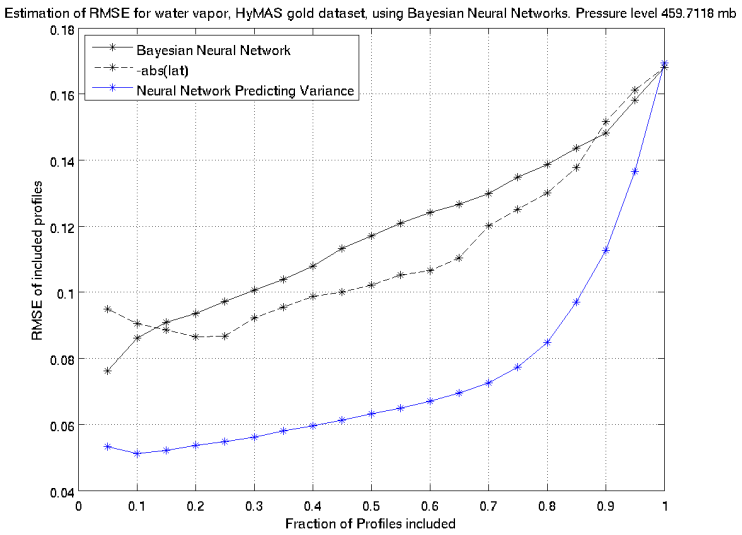


Figure 3-8: This is the same problem as shown in figure 3-3, except the Bayesian neural networks is tested on the HyMAS "golden days" test set. RMSE is in units of normalized mass mixing ratio.

the training data.

To explain why the Bayesian neural net still sometimes does well at separating out the cases despite being unable to predict the actual variance well, recall that the hyperparameters are ultimately optimized using a maximum likelihood metric (the likelihood is given in equation (3.20)). $p(T|\alpha, \beta)$ will be maximized by having the predicted variances σ^2 as close as possible to the true variances of the cases. Given enough data, the true variance in a certain region of the input space can be approximated as the RMSE of cases in that region, so to maximize the likelihood, σ^2 should be approximately be that RMSE.

Now, in the problems where Bayesian neural nets do well, there must be still be some correlation between the density of the input data in some region (of course, how dense the data needs to be is dependent on how complicated the function of the inputs is) and the RMSE of the cases in that region. The hyperparameters would then be optimized to take advantage of that correlation.

However, as mentioned before, in all problems the predicted variance σ_n^2 was dominated by a constant term $\frac{1}{\beta}$, so that the predicted variance was relatively constant compared to the actual RMSE, which is certainly not optimal in a maximum likelihood

sense. The key here is that the input data density must simply too uniform to allow for large differences in model uncertainty and thus allow for the predicted variance to accurately match the RMSE. At best, the Bayesian neural network can assign slightly higher variance to cases with high RMSE, which is still an improvement in the maximum likelihood over assigning constant variance to everything. In a sense, the Bayesian neural network is doing the best it can while constrained by the lack of a heteroscedastic term in the variance prediction.

3.3.4 Consistency and Hyperparameters

A possible concern, given the very small range of predicted variances, is whether the Bayesian neural network can consistently be trained to discover that correlation. If we suppose that all cases are separated into quality categories, a minor change in the predicted variance of a case might lead to a large change in the designation of quality of that case. Thus, it is important to ask whether the variance predictions are generally stable across different trials, and whether the ordering of the cases by predicted variance is consistent as well across trials.

A major cause might be the weights being in a non-optimal local minima during training. This should be no more likely than it is for a regular neural network, since they share roughly the same training algorithm. Another factor could be unoptimized or poorly estimated hyperparameters, which could arise from deficiencies in the evidence approximation used. Of course, if the correlation between data density and the variance is strong enough, incorrectly estimated hyperparameters should only cause the predicted variance to be either too low or too high, but should not have a major effect on the ordering of the cases by predicted variance.

Empirically it appears that on levels where the Bayesian neural network shows no skill, repeated trials do not help. An example is shown for water vapor prediction on the ECMWF/Aqua dataset, where in all the trials the actual RMSE of the various cases has little to do with the predicted variance (see figure 3-9). In that figure, although the variance predicted by each Bayesian neural network is slightly different, none of them show any particular skill at estimating variance (note the y-axis scale

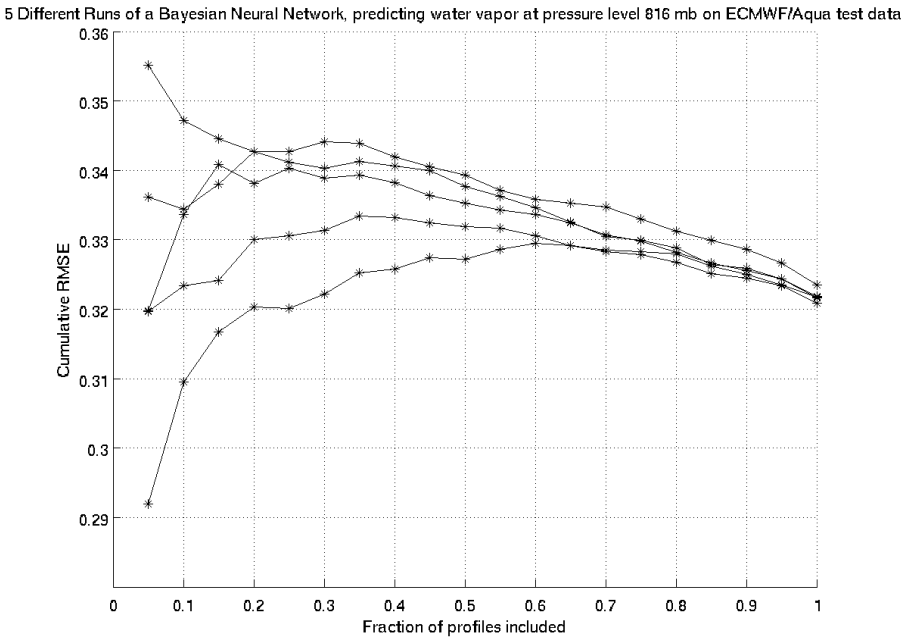


Figure 3-9: This figure shows repeated trials of training a Bayesian neural network. RMSE is in units of normalized mass mixing ratio.

in figure 3-9).

However, on levels where the Bayesian neural network perform well, there are occasional trials where the hyperparameters are sometimes less than optimal. In the problem shown in figure 3-10 (estimating temperature on the ECMWF/Aqua dataset at 958 mb), this leads to a situation where the cases with the lowest predicted actually have relatively high RMSE (high only relative to other cases with low predicted variance; the RMSE is still slightly lower than the average RMSE of all cases). In this particular case, setting the hyperparameter ratio lower brings the variance estimation performance more in line with the other Bayesian neural networks. The differences in the results are likely due to finding different local minima in the weight space. Recall that in Mackay's approach to hyperparameter estimation, each local minima should have its own set of optimal (with respect to maximizing likelihood) hyperparameters, and it seems that some local minima have optimal hyperparameters that do not separate out the cases well. Following is a more detailed discussion of why the "optimal" hyperparameters are chosen and how perturbing them affects the variance

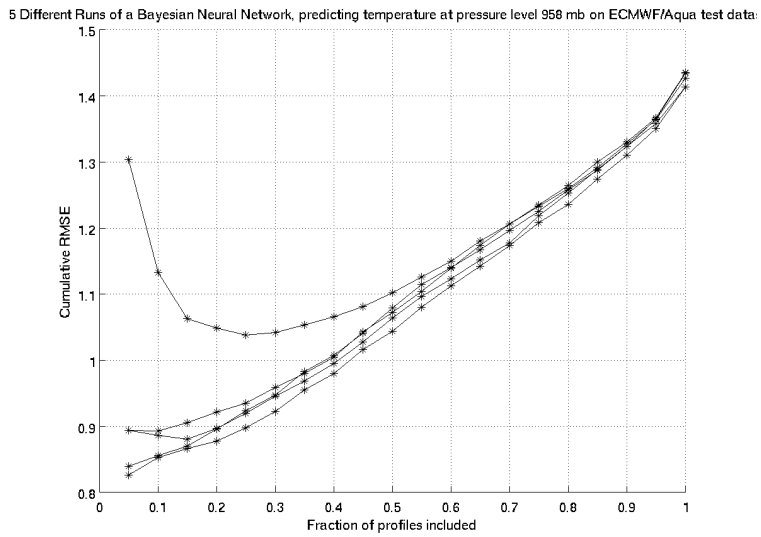


Figure 3-10: This figure shows repeated trials of training a Bayesian neural network. The anomaly is due to the RMSE of the cases with the 5 percent lowest predicted variance, which is much higher than predicted. RMSE is in degrees kelvin.

predictions.

This leads to the question of how sensitive the variance predictions are to changing the hyperparameters of a Bayesian neural network. If we subscribe to the approximation that there is no dependence of the minima $\vec{w}_{min}$ on α, then changing the hyperparameters after training should be the same as starting with those hyperparameters before training. Of course, this approximation is not true except for the most basic linear neural networks [5], but it simplifies the following analysis quite a bit.

Since the term $\frac{1}{\beta}$ has a constant effect on all variance predictions, the only term we need to look at is $\mathbf{g}^T(\nabla\nabla S(\vec{w}))^{-1}\mathbf{g}$. The gradients $\mathbf{g}$ with respect to the weights $\vec{w}$ are not dependent on the hyperparameters, but the Hessian $\nabla\nabla S(\vec{w})$ can be written as $\beta(\nabla\nabla E + \frac{\alpha}{\beta}\mathbf{I})$, where E is the error function (given in equation (3.14)). Since we are more interested in the predicted variance relative to the other cases and not the accuracy of the actual number itself, the main factor affecting the Hessian (and thus the variance) is the ratio of the hyperparameters $\frac{\alpha}{\beta}$. Increasing this ratio will "regularize" the Hessian more, by increasing the contribution of $\mathbf{I}$. Figure 3-11 shows the results of varying this ratio on a Bayesian neural network predicting water vapor

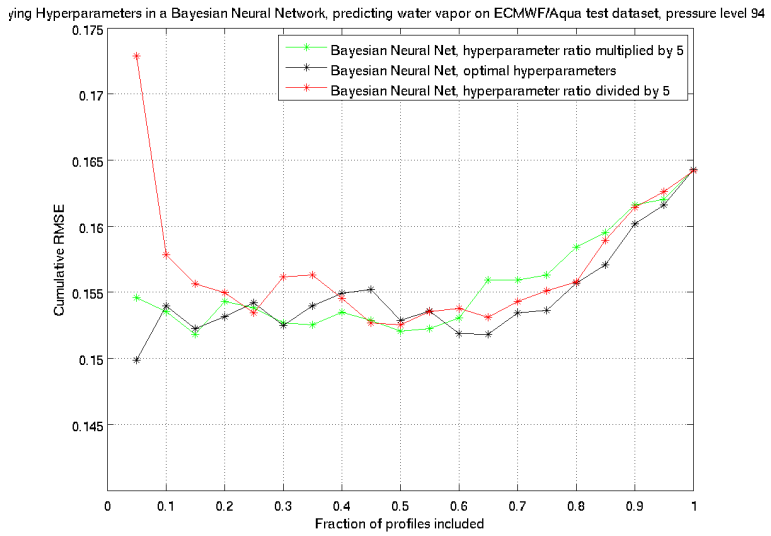


Figure 3-11: This figure shows the effects of scaling the hyperparameter ratio after the Bayesian neural network has been trained. RMSE is in units of mass mixing ratio.

on the ECMWF/Aqua dataset on pressure level 940 mb. The "flatness" of that particular graph implies the Bayesian neural network assigns the same variance to 60 percent of the cases.

One notable consequence of raising the hyperparameter ratio is that the RMSE of the cases with the lowest predicted variances increases dramatically when the hyperparameter ratio is raised slightly from the optimal setting. One factor is that as the hyperparameter ratio increases, the increasingly dominant $\mathbf{I}$ term in the Hessian will lead to the variance being dominated by the magnitude of the output gradient $\mathbf{g}^T\mathbf{g}$. If the magnitude of the gradient is not positively correlated with the RMSE, then it would certainly be expected that the variance prediction performance would be degraded as the hyperparameter ratio increased, and that the lowest predicted variances would be the first to be changed by the increasingly dominant magnitude term.

However, this also occurs even when the magnitude of the gradient happens to be correlated with RMSE, such as on the problem of predicting temperature on ECMWF/Aqua dataset at 958mb (see figure 3-12). On that problem, the variance prediction improves when the hyperparameter ratio increases, especially when it is

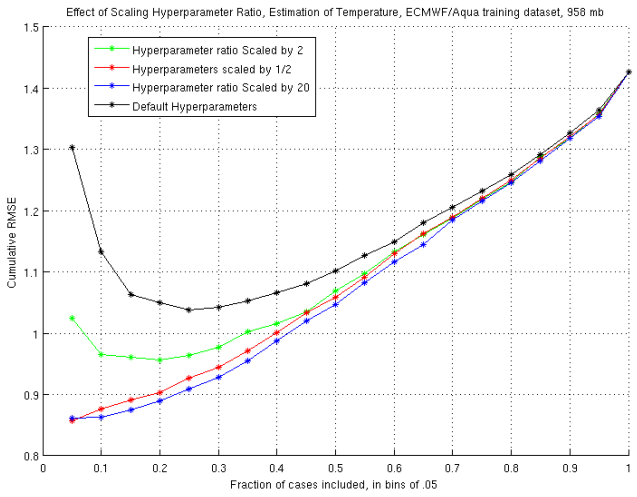


Figure 3-12: This figure is based on the same concept as 3-11, except in this case the default hyperparameters appear to be sub-optimal in terms of allowing the Bayesian neural network to estimate the difficulty of the cases. RMSE is in degrees kelvin.

increased so much that the magnitude of the gradient starts to completely dominate the variance prediction. However, when the hyperparameter ratio is decreased, the variance prediction performance on the first few bins is improved as well. Thus, had we started with the hyperparameter ratio at $\frac{1}{2}$ its actual value, increasing the hyperparameter ratio (to its actual value) would have lead to worse variance estimation, even though the gradient magnitude term would theoretically be more dominant than before. So a small increase (or indeed, any perturbation at all) in the hyperparameter ratio could potentially lead to worse variance estimation regardless of whether or not the gradients are correlated with RMSE.

This is mostly because the effect on the Hessian from a small change in the hyperparameters is primarily dependent on the non-diagonal entries, and those are unfortunately not predictable or consistent from problem to problem. This is exacerbated by the small range of variances predicted, which means that even the tiniest shift in the Hessian can potentially lead to a major reshuffling of the cases when ordered by the magnitude of the predicted variance.

Overall, these examples suggests that there are "good" values of the hyperparameters, good in the sense that they lead to variance predictions which can accurately

sort out the cases with the lowest RMSE from those with higher RMSE. In some problems, these good hyperparameters are NOT the hyperparameters found by the Bayesian neural network, which is maximizing the likelihood.

A natural question is why the Bayesian neural network did not find the good hyperparameters. One factor could be simply that the approximations made (both the approximation that $p(\alpha, \beta|T)$ is sharply peaked, and that the eigenvalues of H have no dependence on α) are not valid enough here. However, even if the approximations were valid, it is possible that the "good" value of the hyperparameters actually leads to a lower likelihood of the data.

To see why, pretend to adjust the hyperparameters from the results. β is hard to adjust since it represents the average RMSE well. However, α can be increased so that the the hyperparameter ratio increases (thus letting the Hessian approach $\mathbf{I}$ and letting magnitude of the gradients dominate the variance). Unfortunately, this also regularizes the Hessian and squeeze the variance range even smaller, so that the likelihood $P(T|\alpha, \beta)$ might be lower than otherwise. The opposite (decreasing α) may not occur due to similar scaling issues. We can write the predicted variance as:

$$\sigma^2 = \frac{1}{\beta} + \frac{1}{\beta}\mathbf{g}(\nabla\nabla E_D + \frac{\alpha}{\beta}\mathbf{I})^{-1}\mathbf{g}^T \tag{3.33}$$

If α, or more precisely the hyperparameter ratio $\frac{\alpha}{\beta}$ decreases, the diagonal entries of the inverse Hessian should increase, most likely causing the term $\mathbf{g}(\nabla\nabla E_D + \frac{\alpha}{\beta}\mathbf{I})^{-1}\mathbf{g}^T$ to increase. This in turn would throw off the actual variance predicted, since even if β was decreased (of course assuming $\frac{\alpha}{\beta}$ was kept at the same decreased level) to try and compensate for $\mathbf{g}(\nabla\nabla E_D + \frac{\alpha}{\beta}\mathbf{I})^{-1}\mathbf{g}^T$ increasing, it would be impossible to adjust the $\frac{1}{\beta}$ and the $\frac{1}{\beta}\mathbf{g}(\nabla\nabla E_D + \frac{\alpha}{\beta}\mathbf{I})^{-1}\mathbf{g}^T$ term independently of each other. Again, the variance predicted for cases will be unpredictably different, and so the maximum likelihood under the new setting of hyperparameters might be lower than before.

Thus, it is certainly possible that in the presence of heteroscedastic noise, a setting of the hyperparameters that leads to worse variance estimation performance (as judged by the ability of the predicted variance to sort the cases in order of increasing

RMSE) actually has higher maximum likelihood. Ultimately then, it seems that for Bayesian networks the maximum likelihood metric might be less suitable when the goal is to sort cases by their predicted variance instead of comparing the predicted variance to the actual variance. The central assumptions of Bayesian neural networks is that β should account for any intrinsic noise in the outputs, and that α, which controls the prior of the weights, should be less relevant as the amount of data increases. Both of these assumptions appear to be violated in these problems.

3.3.5 Discussion of Results

Overall, the results suggests that a Bayesian neural network is not well-suited to estimating variance on geophysical parameter retrieval problems. The nature of the problem suggests a strong heteroscedastic element to the variance, and the results obtained by the Bayesian neural network confirms this. Even when the Bayesian neural network does well at estimating the relative difficulty of a case (as judged by the residual of that case), the estimation is often unstable, as shown both by the effects of perturbing the hyperparameters and by the results of multiple trials. This suggests that any correlation between model uncertainty and the RMSE is not very strong, as otherwise it should be able to withstand small perturbations in the hyperparameters. Finally, the predicted standard deviation is often limited to a small range, and does not correspond well to the actual RMSE of the cases.

Most of the problems with Bayesian neural networks stems from its assumption that the noise is constant throughout, and the only source of non-constant variance should be from model uncertainty caused by lack of data. Although this assumption may be true for some problems, the datasets examined here have enough data to make the main strength of Bayesian neural networks less useful. This motivates the need for methods that can take into account heteroscedasticity.

Chapter 4

Additional Confidence Estimation Methods

As previously mentioned, Bayesian neural network's main weakness is its inability to adapt to heteroscedasticity. In remote sensing the reliability of, say, a temperature retrieval can be much lower in the presence of heavy clouds or heavy dust cover, both conditions which also lead to detectable changes in the radiance inputs. Methods that are to accurately estimate confidence must therefore take this heteroscedasticity into account.

In this chapter, I will review two such confidence estimation methods, Mixture Density Networks and Sparse Pseudo Input Gaussian Process Regression (SPGP). As far as I am aware, these methods have not been tried before in remote sensing. I will discuss the theoretical strengths and weaknesses of each approach, and then present the results on the datasets previously discussed.

4.1 Mixture Density Networks

Mixture density networks (MDNs) are a variation on the classic neural network. Instead of estimating the targets directly, mixture density networks attempt to estimate the parameters of a Gaussian mixture model (GMM) which describes the target distribution. In theory, the GMM is general enough that a MDN can model any sort

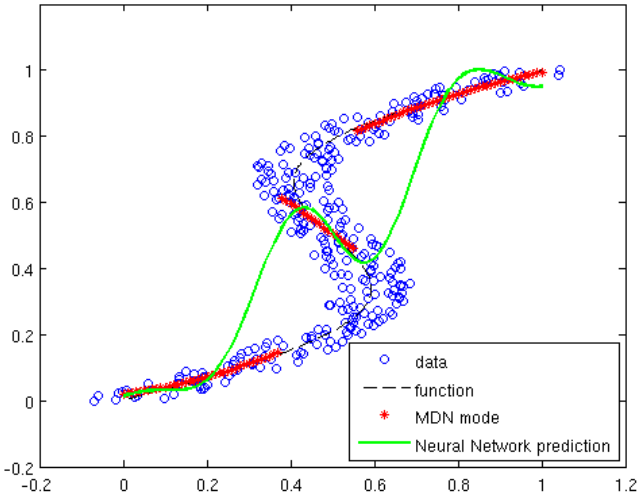


Figure 4-1: This is an example of the type of data that could benefit from being modeled by an MDN. This is a simple synthetic toy dataset, and so the x-axis and y-axis units are not important here.

of output distribution, even inverse problems involving a one-to-many mapping. A simple example is shown in figure 4-1, and the contour plot of the actual predicted output distribution of the MDN is shown in figure 4-2. There, we can see that the arcsine-like function is impossible to model correctly with a regular neural network, because the center portion has multiple y-values for any single x-value. However, theoretically an MDN can model the distribution of data in the center using a triple peaked Gaussian mixture model.

4.1.1 Gaussian Mixture Models

Gaussian mixture models are a probabilistic model for density estimation using Gaussian distributions as the mixture components. As a trivial example, if the distribution we wanted to model was actually Gaussian with mean μ and standard deviation σ, the ideal (in terms of Bayesian likelihood) GMM would consist of one mixture component: a Gaussian with mean μ and standard deviation σ. A more complicated, non-Gaussian probability distribution would be modeled by the sum of several Gaussian components. For example, if a probability distribution is double-peaked, there

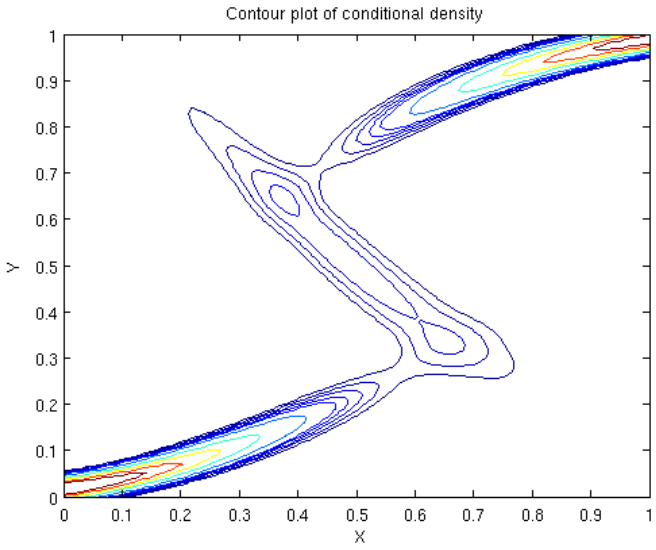


Figure 4-2: A contour plot of the output distribution of a MDN with three Gaussian components modeling the same dataset as in figure 4-1. This plot shows the distribution $p(y, x)$, from which it is easy to obtain $p(y|x)$ for any x by simply multiplying by a normalization factor $p(x)$.

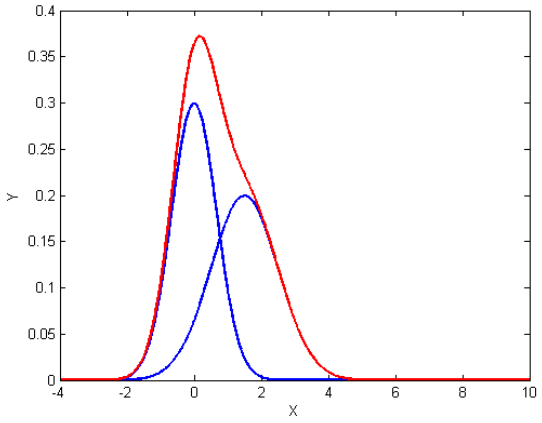


Figure 4-3: The red non-Gaussian distribution is revealed to be a weighted sum of two Gaussian ones

could be two Gaussian components in the GMM, one for each peak. Another interesting case is shown in figure 4-3, where the complicated non-Gaussian distribution is modeled by the sum of two Gaussian distributions.

For more flexibility, a GMM is usually a weighted sum of the Gaussian components, and the weights themselves are also parameters to be determined. Thus, we can fully parametrize a Gaussian mixture model as the means and standard deviations of

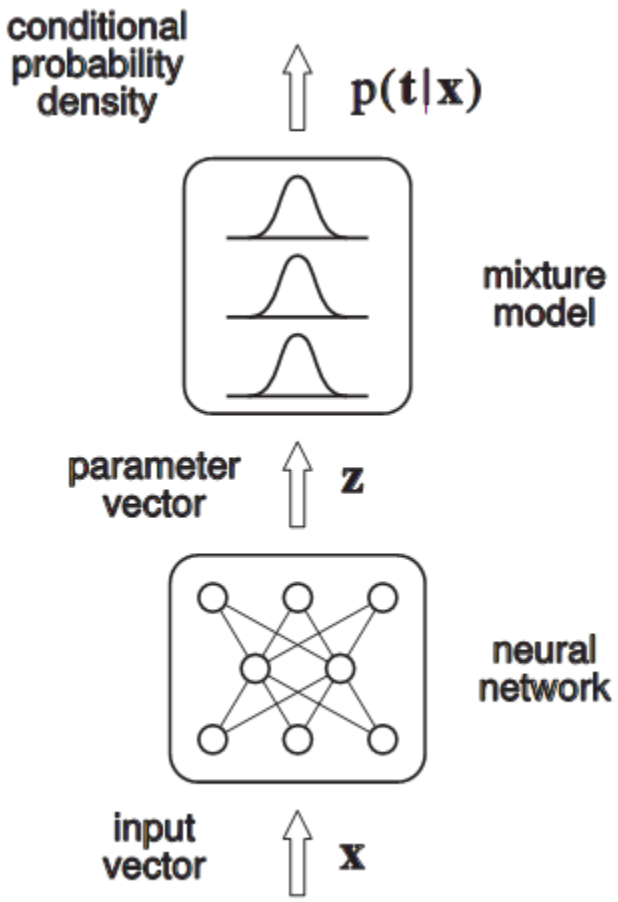


Figure 4-4: This shows the structures of a mixture density network. The parameter vector $\mathbf{x}$ refers to the weights o_i, the means μ_i and the standard deviations σ_i. This figure was taken from the MDN paper by Bishop [4]

the g Gaussian distributions, as well as the g weights in the weighted sum (these are not the same as the neural network weights). Mathematically, the distribution can be written as

$$p(t) = \sum_{i=1}^{g} o_i N(\mu_i, \sigma_i) = \sum_{i=1}^{g} o_i \frac{1}{\sqrt{2\pi\sigma_i}} \exp\left(-\frac{||t-\mu_i||^2}{2\sigma_i^2}\right) \tag{4.1}$$

4.1.2 MDN Structure

Structurally, an MDN (shown in figure 4-4) is very similar to a regular neural network. The primary difference is that instead of one output, the MDN has three groups of outputs, which correspond to the weights, means, and standard deviations that

parametrize a Gaussian mixture model. The actual number of outputs depends on the number g of Gaussian components, which is specified beforehand. Since there are g weights, g means, and g standard deviations, there are $3g$ outputs, as opposed to one output for a typical neural network. Call these outputs z, with the outputs controlling the weights being z^o, the outputs controlling the standard deviations as z^σ and the means as z^μ.

Because the outputs are parameters of a Gaussian mixture model, they must obey certain rules. To this end, the actual neural network outputs z are transformed into the parameters of the Gaussian mixture model.

The weights, o_i, must lie within [0, 1] and sum to unity. To enforce this, the neural network outputs z^o are transformed to o via a softmax function:

$$o_i = \frac{\exp(z_i^o)}{\sum_j \exp(z_j^o)} \tag{4.2}$$

The variances σ must be positive, so we exponentiate the outputs z^σ:

$$\sigma_i = \exp(z_i^\sigma) \tag{4.3}$$

This has the additional important benefit of making it harder for standard deviations to go to zero, which can potentially be a large problem when maximizing the likelihood later on.

Finally, the means μ_i are just directly the outputs, so that $\mu_i = z_i^\mu$.

To train this neural network, we minimize the log-likelihood of the training targets T, which is:

$$E_T = \sum_{t \in T} -\log(p(t|\mathbf{x})) \tag{4.4}$$

where $p(t|\mathbf{x})$ is the same as equation (4.1), except that the dependence of $\sigma_i(\mathbf{x})$, $w_i(\mathbf{x})$, and $\mu_i(\mathbf{x})$ on the inputs $\mathbf{x}$ is made explicit. The derivatives of the outputs z with respect to this error function E_T can be obtained in a manner similar to regular neural networks, and the MDN can then be trained by similar gradient ascent methods by maximizing likelihood (see appendix A.2 for details). Note that due to the lack of

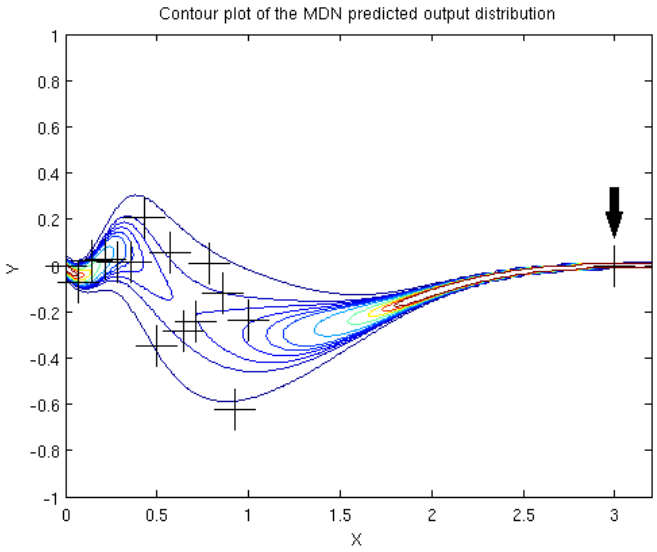


Figure 4-5: This figure shows a toy dataset, as well as the contour plot of the output distribution of a MDN with one Gaussian component. The MDN assigns the lone data point indicated by the arrow very low variance.

any model complexity penalties, given enough Gaussians components in the GMM, the highest likelihood would be achieved with 0 standard deviation Gaussians exactly centered around the input data. The earlier transformation $\sigma_i = \exp(z_i^\sigma)$ helps to combat this by making it impossible for standard deviations to reach zero.

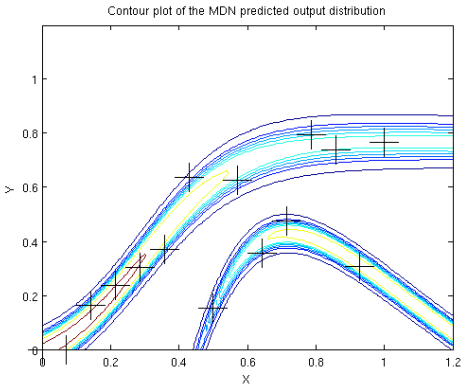
MDNs should be ideal for modeling the full distributions of the target outputs provided there is enough training data. Unfortunately, like a variance estimation neural network, MDN variance predictions on inputs that are different enough from the training inputs are not trustworthy. Furthermore, if input data are lacking (for example, there is only one training point in some area of the input space), the MDN can "pinch in" on the training point, estimating very low variance, due to the fact that the maximum likelihood Gaussian output distribution at that point would essentially be a delta function (see figure 4-5). This can be a problem depending on the characteristics of the dataset.

4.1.3 MDN training

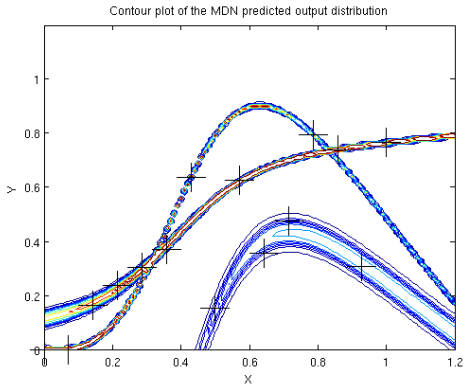
In the problems that I used MDNs on, I found that having more than one mixture component leads to unnecessary complications. Inverse problems where a single x is mapped to multiple y-values often have multiple local minima if we optimize the likelihood, especially if there is not enough data. With multiple Gaussian components and not enough data density, the MDN often tended towards multi-valued solutions, whether warranted or not. For example, there were many instances of the MDN predicting a multi-modal distributions, but with at least one of the Gaussian's predicted variance being extremely small. Figure 4-6 shows three different trials of a MDN with three Gaussian components on a toy dataset that exhibits heteroscedastic behavior. In all three trials, the MDN predicts multi-modal output distributions which seem unnecessarily complex. Moreover, the three trials give wildly different results, suggesting local minima is a large problem when optimizing MDNs that have multiple Gaussian components. The reason this pathological behavior did not occur on the sinusoidal dataset (figure 4-1) was because the data density is extremely high there, which is not the case in the toy dataset here, and more importantly is not the case on the actual datasets. The aforementioned lack of a complexity penalty term to penalize complex output distributions hurts the MDN performance here.

We have already seen that the MDN with multiple Gaussian components tends towards multi-modal output distributions. However, the predicted mean with one mixture component and a multitude of mixture components should be one and the same, and having multiple components does not give more useful outputs unless the target function is indeed strongly multi-valued. In the remote sensing problems that I studied, such multi-valued functions seemed rare, if they existed at all. One factor could be the noisiness of the data obscuring any such distribution. It could also be the case that there is not enough data to unambiguously make the case for a multi-valued function rather than simply a very noisy or complex one-to-one function, such as what seemed to happen for the dataset in figure 4-6.

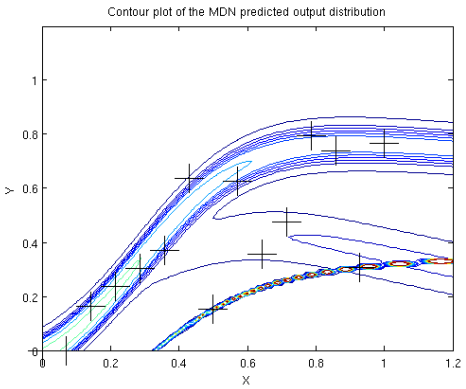
Finally, having multiple mixture components imposes the problem of interpreting



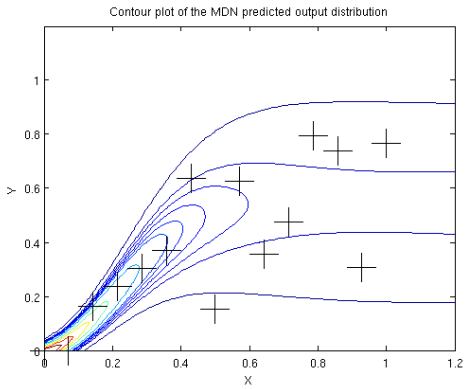
(a) A contour plot of the predicted output distribution of a trained MDN with three Gaussian components



(b) A contour plot of the predicted output distribution of a trained MDN with three Gaussian components, starting with different initial weights



(c) A contour plot of the predicted output distribution of a trained MDN with three Gaussian components, starting with different intial weights



(d) A contour plot of the predicted output distribution of a trained MDN with one Gaussian component

Figure 4-6: This figure shows three different runs of a MDN with three Gaussian components (figures 4-6(a),4-6(b), and 4-6(c)), as well as an MDN with only a single Gaussian component (figure 4-6(d)) on the same toy dataset. The data points are indicated with the crosses.

the predicted standard deviation for any particular input. Since we don't know the actual shape of the output distribution short of actually graphing and examining it, the predicted standard deviation may not be of much use in terms of summarizing the distribution.

Overall, I found that having only a single mixture component was enough for my purposes, while simultaneously also reducing the training time and increasing the robustness of the final result to local minima. The obvious downside is that it cannot accurately model any non-Gaussian noise, which may be a problem for some applications.

MDN consistency

Once there is only a single Gaussian component, the performance of MDN is fairly stable across retraining the hyperparameters on the same problem. Although there are indeed many local minima, they seem to give roughly the same performance in variance estimation. As an example, I ran five trials of MDN on the HyMAS water vapor dataset, shown in figure 4-7.

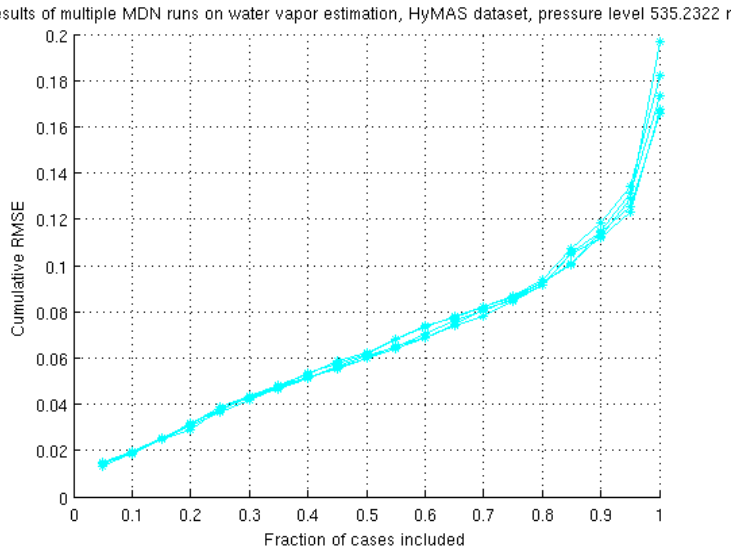


Figure 4-7: This figure shows five repeated trials of training MDN to estimate water vapor on the HyMAS dataset, pressure level 535 mb. RMSE is in units of normalized mass mixing ratio.

4.1.4 MDN "hyperparameters"

Recall from section 2.3 that all the networks I used have only one hidden layer.

I chose the number of hidden nodes (20) of the MDN to be the same as that of any other neural network, and initialized the weights similarly also. The intuition for the latter is that the MDN I used (with only one Gaussian component) is simply a neural network with a different error metric, so that the optimal settings of the weights should still be roughly the same as before, at least for estimating the mean of the Gaussian. The weights that optimize the variance of the Gaussian with respect to the likelihood of the data may be different, but after MDN training the magnitude of the weights was similar to conventional neural networks, indicating that the initializations were sound.

The number of hidden nodes was chosen fairly arbitrarily. The main motivator was that 20 hidden nodes was a good choice for regular neural networks, and the performance of the neural network seemed fairly insensitive to the number of hidden nodes past a certain number. However, there could certainly be room for improvement here in terms of reducing the number of hidden nodes, if training time is of paramount concern.

4.1.5 Network weights initialization

For the most part, the network weights (including the biases) were initialized by taking random samples from a normal distribution, with the standard deviation set to the square root of the number of nodes in the layer (so that the sum of the weights in a particular level would be a normal with unity standard deviation). However, the bias terms of the output layer was set to be the appropriate initial parameters of the Gaussian "mixture" model. In this case, since there was only one Gaussian, the mean and the standard deviation were set to the mean and standard deviation of the targets.

4.2 Sparse Pseudo-Input Gaussian Process regression

Overall then, MDNs should be a powerful method for modeling variance. However, the fact that MDNs cannot take into account the model uncertainty (the uncertainty due to lack of data) was the motivation for looking into other methods, like SPGP.

Sparse Pseudo-input Gaussian Process regression (SPGP for short) is a relatively recent regression technique that scales the highly successful Gaussian Process regression to be tractable on large data sets ($>$ 10000 training profiles). To understand SPGP, it is helpful to first discuss Gaussian processes and how they can be used for regression.

4.2.1 Gaussian Process Regression

Gaussian Process regression, or GPR, is a way to do Bayesian inference using Gaussian processes. Gaussian processes themselves can be thought of as infinite dimensional Gaussian distributions. The Gaussian process defines a probability distribution over functions $y(\mathbf{x})$, in much the same way that a multivariate Gaussian distribution is defined over vectors $\mathbf{x}$ [16]. It is perhaps intuitive to think of functions as an infinite dimensional vector, such as the coefficients of the Fourier transform or the Taylor series.

The (infinite) functions defined by the Gaussian processes (the prior) are then conditioned on the training data, so that only functions which match the training data are left (the posterior). This (also infinite) number of functions is then used for predictions. The mean of these functions is the predicted function, and the standard deviation is derived similarly. A simple example can be seen in figure 4-8. A few sample functions from the prior are shown, along with the implied standard deviation (the gray area). The rightmost figure shows the posterior function and the implied standard deviation. As expected, the standard deviation is zero at the training data, and grows larger away from the data.

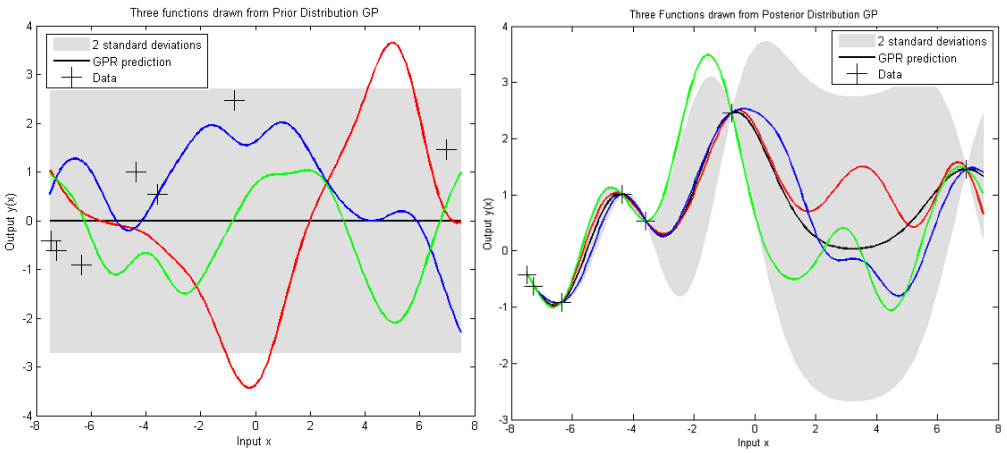


Figure 4-8: A simple example of GPR. On the left, a few sample functions from the prior are shown, along with the implied standard deviation (the gray area). The rightmost figure shows sample functions drawn from the posterior after conditioning on the data marked by the crosses.

Of course, it would be inconvenient to write out infinite dimensional vectors and infinite numbers of functions, so in the context of GPR, Gaussian processes are often written as the output distribution at a particular point we are interested in, like so:

$$\mathrm{GP}(\mathbf{y}(\mathbf{x})) = N(\mu, K(\mathbf{x}, \mathbf{x}')) \tag{4.5}$$

where μ is a mean, and K is the covariance, also known as a "kernel" in this context. The covariance matrix consists of $n \times n$ entries $k(x, x')$, where k is the kernel function and x are the individual input values contained in the vector $\mathbf{x}$. The previous equation focused on the values of the function $\mathbf{y}(\mathbf{x})$ on only the finite vector of input values $\mathbf{x}$, which had a multivariate Gaussian distribution with mean μ and covariance K.

Since the Gaussian process is a probability distribution, it can then be used as a prior over possible functions to model the data. Often, priors over the functions are

$$p(\mathbf{y}) = N(0, K) \tag{4.6}$$

with mean 0 and covariance K (μ is often taken to be 0 for simplicity [21]; we can always subtract the mean from the targets so that the mean of the transformed

targets is 0). Now, assume that the noise in the training targets $\mathbf{t}$ is distributed normally with variance σ^2. In this case, the notation $\mathbf{t}$ is a vector consisting of all the targets t_i (essentially, $\mathbf{t}$ is the vector form of all the training targets T). Then we have that

$$p(\mathbf{t}|\mathbf{y}) = N(\mathbf{y}, \sigma^2\mathbf{I}). \tag{4.7}$$

Finally,

$$\int p(\mathbf{t}|\mathbf{y})p(\mathbf{y})d\mathbf{y} = p(\mathbf{t}) = N(\mathbf{0}, K + \sigma^2\mathbf{I}) \tag{4.8}$$

This quantity is also known as the marginal likelihood, much like the similar quantity in Bayesian neural nets (note that there is also a similar variance term). It is the probability that we see the training targets, given the possible underlying functions.

To perform predictions $\mathbf{t}_t$ on a test set (although the notation $\mathbf{t}_t$ implies that these are the test targets, this is somewhat unfortunate since these are actually the predictions), we form the joint marginal likelihood $p(\mathbf{t}, \mathbf{t}_t)$ of the test set and training sets (which is Gaussian), and then find the conditional probability $p(\mathbf{t}_t|\mathbf{t})$. Here is where the use of Gaussian processes comes in handy, because

$$p(\mathbf{t}_t|\mathbf{t}) = \frac{p(\mathbf{t}, \mathbf{t}_t)}{p(\mathbf{t}_t)} \tag{4.9}$$

is also a Gaussian process (this is a property of Gaussian distributions). Thus, we can write

$$p(\mathbf{t}_t|\mathbf{t}) = N(\mu, \Sigma) \tag{4.10}$$

To find the parameters μ and Σ, we first examine the joint prior probability $p(\mathbf{y}, \mathbf{y}_t)$ (the notation $(\mathbf{y}, \mathbf{y}_t)$ means it is a single, concatenated vector)

$$p(\mathbf{y}_t, \mathbf{y}) = N(0, K_{n+t}) \tag{4.11}$$

where K_{n+t} is a block matrix

$$K_{n+t} = \begin{bmatrix} K & K_{tn} \\ K_{nt} & K_t \end{bmatrix}. \tag{4.12}$$

K is the previous square covariance matrix consisting of the training data. K_t is similar, but consisting of the test data only. K_{tn} and its transpose K_{nt} are rectangular matrices that are the covariances of the training with the test data, with entries $k(\mathbf{x}, \mathbf{x}_t)$, where $\mathbf{x}$ are the inputs of the training set and $\mathbf{x}_t$ are the inputs of the test set.

Analogous to before (equation (4.8)), the joint marginal likelihood is then:

$$p(\mathbf{t}, \mathbf{t}_t) = N(\mathbf{0}, K_{n+t} + \sigma^2\mathbf{I}) \tag{4.13}$$

Using the previously mentioned theorem on Gaussian distributions, the conditional probability can be extracted from the joint:

$$p(\mathbf{t}_t|\mathbf{t}) = N(\mu, \Sigma) \tag{4.14}$$

$$\mu = K_{nt}(K + \sigma^2\mathbf{I})^{-1}\mathbf{t} \tag{4.15}$$

$$\Sigma = K_t - K_{tn}(K + \sigma^2\mathbf{I})^{-1}K_{nt} + \sigma^2\mathbf{I} \tag{4.16}$$

Thus, for any particular input vector $\mathbf{x}_t$, we get the full probability distribution $p(\mathbf{t}_t)$ of possible outputs, which is a multivariate Gaussian distribution. The most likely $\mathbf{t}_t$ is the mean $K_{nt}(K + \sigma^2\mathbf{I})^{-1}\mathbf{t}$. The predicted variance is $\mathrm{diag}(K_t - K_{tn}(K + \sigma^2\mathbf{I})^{-1}K_{nt} + \sigma^2\mathbf{I})$, since we usually do not care about correlations amongst the various output cases.

Kernels

The covariance matrices, or the kernels, completely parametrize the Gaussian process function priors. In the literature, the squared exponential (or Gaussian) kernel is

often used, having the form:

$$K(\mathbf{x}, \mathbf{x}') = a^2 \exp\left(-\frac{||\mathbf{x} - \mathbf{x}'||^2}{2\lambda^2}\right) \tag{4.17}$$

where $\mathbf{x}$ and $\mathbf{x}'$ are vectors. a and λ are hyperparameters which are the same for all elements in K, and that are assumed to be fixed for now. Selection of suitable hyperparameters will be discussed later.

One desirable property of this kernel is that points that are close together (in x) are highly correlated, which is intuitively appealing. The speed of the dropoff in the correlation is controlled by λ.

Of course, there are many other possible kernels that could be used as well, but the squared exponential kernel has been successful in a wide variety of nonlinear regression and classification tasks [21] [19] [3], and is the one used in this thesis.

Hyperparameter Optimization

One attractive property of Gaussian process regression is that optimizing hyperparameters can be done through gradient descent instead of through more time consuming methods like cross-validation. The quantity to be optimized (maximized) is usually the log of the marginal likelihood expressed in equation (4.8) (this is of course the same as maximizing the likelihood, but is more numerically stable and easier to simplify). We rewrite it as a function of the hyperparameters, θ:

$$L = \log p(\mathbf{t}|\theta) = \log N(\mathbf{0}, K + \sigma^2\mathbf{I}) \tag{4.18}$$

$$L = -\frac{1}{2}(\mathbf{t}^T(K + \sigma^2\mathbf{I})^{-1}\mathbf{t} + \log|K + \sigma^2\mathbf{I}| + \frac{n}{2}\log 2\pi) \tag{4.19}$$

The hyperparameters θ are included in the kernel matrix K.

The gradients of L with respect to each individual hyperparameter θ_i in θ is then:

$$\frac{\partial L}{\partial \theta_i} = \frac{1}{2}(\mathbf{t}^T K^{-1}\frac{\partial K}{\partial \theta_i}K^{-1}\mathbf{t} - \mathrm{tr}(K^{-1}\frac{\partial K}{\partial \theta_i})) \tag{4.20}$$

where $\frac{\partial K}{\partial \theta_i}$ is a matrix of derivatives that is dependent on the choice of the kernel. We can then use any gradient based optimization method, like the popular LBFGS (Low memory Broyden-Fletcher-Goldfarb-Shanno algorithm) [13], to optimize the likelihood and find the "best" hyperparameters. Note that the marginal likelihood (equation (4.19)) is composed of two terms that ultimately end up in the derivative: $\mathbf{t}^T(K + \sigma^2\mathbf{I})^{-1}\mathbf{t}$ and $\log|K + \sigma^2\mathbf{I}|$ (the third term $\frac{n}{2}\log 2\pi$) is just a normalization term and has no dependence on any hyperparameters). The first term increases as the model better fits the data. The latter term $\log|K + \sigma^2\mathbf{I}|$ is independent of the data and is effectively a complexity penalty that favors less "extreme" settings of the hyperparameters [16]. In this sense, it is much like the weight regularization term in Bayesian neural networks.

4.2.2 SPGP Predictions and Derivatives

The primary weakness of Gaussian process regression as presented is its inability to scale to large data sets. Setting aside the time it would take to optimize the hyperparameters, the prediction of new data requires the inversion of $(K + \sigma^2\mathbf{I})$ (see equation (4.14)), which is a $n \times n$ sized matrix, where n is the number of training cases. Since matrix inversion requires $O(n^3)$ computation, this does not scale well past 10000 entries or so on modern computers. Moreover, a 20000 by 20000 matrix of double precision floats would be 3 gigabytes, which is difficult to fit into memory. Yet in remote sensing, databases of millions of simulated cases are not uncommon.

Even though full Gaussian process regression would not be tractable on such datasets, approximations to Gaussian process regression might be. In the literature, such approximations often take the form of replacing the kernel matrix K with a lower rank matrix Q. The simplest is perhaps just discarding parts of the dataset, an approach known as subset of data (SD) [16]. Perhaps a more sophisticated approach is to use only part of the data, say m cases, to approximate K, so that Q becomes

$$Q = Q_n = K_{nm}K_m^{-1}K_{mn} \tag{4.21}$$

This is known as the Nystrom construction [21] [16], and is quite common in many approximation schemes [27]. The trick is selecting the best m cases, because the approximation will decrease in accuracy away from those m inputs. Another way to think about this is to regard the function predicted by Gaussian process regression as function of the input cases:

$$\mathbf{t}_t(\mathbf{x}_t) = K_{nt}(K + \sigma^2\mathbf{I})^{-1}\mathbf{t} = \sum_{i=1}^{n} \alpha_i k(\mathbf{x}_i, \mathbf{x}_t) \tag{4.22}$$

Then, the approximation is to simply replace the n terms $\alpha_i k(\mathbf{x}_i, \mathbf{x}_t)$ with m terms, creating a simpler approximation to the original function.

In SPGP, the m vectors are not necessarily inputs, but are instead known as *pseudo-inputs*. Pseudo-inputs, denoted as $\bar{\mathbf{x}}_i$, where $i \in [1, m]$, can be thought of as hyperparameters to be optimized later, and are not necessarily part of the original training data. The pseudo-inputs (combined with appropriate pseudo-outputs) make up an alternative, smaller dataset that best summarizes the features of the original data, in a sense.

To start, it is assumed that there is no noise on the pseudo-outputs $\bar{\mathbf{y}}$, since they are not real data. Now, if we apply Gaussian process regression to the pseudo-dataset, the predicted probability of an output target $\mathbf{t}_t$ for a test input $\mathbf{x}_t$ would be the following:

$$p(\mathbf{t}_t(\mathbf{x}_t)|\bar{\mathbf{y}}) = N(K_{mt}(K_m)^{-1}\bar{\mathbf{y}}, K_t - K_{tm}(K_m)^{-1}K_{mt} + \sigma^2) \tag{4.23}$$

This is exactly the same as the prediction of equation (4.14), except with the noise term $\sigma^2\mathbf{I}$ removed from the kernel. If the test inputs $\mathbf{x}_t$ are now taken to be the actual training data $\mathbf{x}_i$, the full likelihood of the training targets is

$$p(\mathbf{t}(\mathbf{x})|\bar{\mathbf{y}}) = \prod_{i=1}^{n} p(t_i(\mathbf{x}_i)) = N(K_{mn}(K_m)^{-1}\bar{\mathbf{y}}, \mathrm{diag}(K_n - Q_n) + \sigma^2\mathbf{I}) \tag{4.24}$$

Q_n is given in equation (4.21). It turns out that the pseudo-outputs $\bar{\mathbf{y}}$ can be easily

integrated out if a Gaussian prior is placed upon them

$$p(\bar{\mathbf{y}}) = N(\mathbf{0}, \mathbf{K}_m) \tag{4.25}$$

This is effectively the same prior placed upon the actual outputs in regular Gaussian process regression (see equation (4.6)). Then, we can integrate out the pseudo-outputs to get the marginal likelihood $p(\mathbf{t})$, in much the same way that we did in equation (4.8):

$$p(\mathbf{t}) = \int p(\mathbf{t}(\mathbf{x})|\bar{\mathbf{y}})p(\bar{\mathbf{y}})d\bar{\mathbf{y}} = N(\mathbf{0}, Q_n + \text{diag}(K_n - Q_n) + \sigma^2\mathbf{I}) \tag{4.26}$$

The derivation of the predicted distributions again closely follows that of the Gaussian process regression. We form the joint marginal likelihood after considering the joint prior, much like in equation (4.13):

$$p(\mathbf{t}, \mathbf{t}_t) = N(\mathbf{0}, Q_{n+t} + \text{diag}(K_{n+t} - Q_{n+t}) + \sigma^2\mathbf{I}) \tag{4.27}$$

By the same theorem as before (see equation (4.10)), SPGP gives a Gaussian process for the test data, with predicted mean and variance

$$\begin{aligned} p(\mathbf{t}_t) &= N(\mu, c^2) && (4.28)\\ \mu &= Q_{tn}[Q + \text{diag}(K - Q) + \sigma^2\mathbf{I}]^{-1}\mathbf{t} && (4.29)\\ c^2 &= K_{tn} - Q_{tn}[Q + \text{diag}(K - Q) + \sigma^2\mathbf{I}]^{-1}Q_{nt} + \sigma^2 && (4.30) \end{aligned}$$

where c^2 is the variance of the distribution.

So far, the hyperparameters (mostly, the pseudo-inputs) have been assumed to be fixed, but we can again find the hyperparameters that maximizes the marginal likelihood (equation (4.26)) by using gradient ascent just as in Gaussian process regression (see appendix A.3 for details).

4.2.3 SPGP with Dimensionality Reduction and Heteroscedasticity

The speed of SPGP can be further improved with dimensionality reduction (DR). Since the number of hyperparameters to optimize scales linearly with the size of the input dimension, high input dimensions can lead to very slow performance as the optimizer is forced to work over an increasingly high dimensional manifold. Moreover, the possibility of being trapped in an undesirable local minima increases as well.

The idea of dimensionality reduction is to reduce the number of input dimensions, which also reduces the dimensions of the pseudo-inputs (they have the same dimensions as the real inputs), leading to a reduction in the number of hyperparameters to optimize. Dimensionality reduction is accomplished by a linear transformation of the inputs, much like PCA. The components of the linear transformation matrix $\mathbf{P}$ are then additional hyperparameters to be optimized. Note that the pseudo-inputs need not be transformed, since we can just make them have the required number of reduced dimensions. Thus, the kernel matrix K_m is unchanged from this addition. On the other hand, the kernel matrix K_{nm} is now composed of terms:

$$K_{nm}(i,j) = K(\mathbf{x}_i, \bar{\mathbf{x}}_j) = a^2 \exp(-\frac{1}{2}(\mathbf{P}\mathbf{x}_i - \bar{\mathbf{x}}_j)^2) \tag{4.31}$$

Where P is the linear transformation matrix. P should be of size $r \times d$, where d is the original number of input dimensions (for example, there are 25 input dimensions in the ECMWF/Aqua dataset), and r is the number of reduced dimensions. Nothing else need change from before. As a side effect, the projection matrix $\mathbf{P}$ can scale the input dimensions, so that the lengthscale parameter in the square exponential kernel is subsumed into $\mathbf{P}$.

Empirical results show that DR is effective in reducing training and prediction times without compromising accuracy [21]. Moreover, if the entries in the projection matrix are then optimized by gradient descent on the likelihood function, it turns out that using $\mathbf{P}$ is markedly superior to PCA for reducing the number of input dimensions [21].

Another extension of SPGP is to deal with heteroscedasticity. So far, Gaussian process regression is much like Bayesian neural networks in that it primarily accounts only for variance from model uncertainty. Now, SPGP can partially account for heteroscedasticity by the positioning of the pseudo-inputs–since the variance increases further away from the pseudo-inputs, theoretically the SPGP can best optimize the likelihood by placing pseudo-inputs in places with low variance and moving pseudo-inputs away from places with high variance. Unfortunately, since the SPGP function tends to the uninformative prior (ie. the constant zero function) further away from the pseudo-inputs, moving pseudo-inputs away from high variance areas could also potentially lead to underfitting in those areas.

An addition of a term h_i to each diagonal entry of the kernel matrix K_m can allow SPGP to somewhat compensate for this. There will be m such terms, one for each pseudo-input. As h_i increases, the row and column that contain the kernels of the i^{th} pseudo-input in the inverse kernel K_m^{-1} tends to 0. Thus, the contribution from that pseudo-input to the approximate kernel Q disappears. At $h_i = 0$, the i^{th} pseudo-input acts the same way as before. This leads to a gradation of a pseudo-input's contribution to the prediction and to the predicted variance, allowing for heteroscedasticity to be modeled while at the same time allowing for the function to be accurately modeled in that region as well.

This affects the kernel only, and we again can treat h_i as hyperparameters that can be optimized by gradient ascent.

4.2.4 SPGP training

From the preceding sections, it's clear that most new hyperparameters introduced by SPGP can be automatically optimized. However, the number of pseudo-inputs and the number of reduced dimensions must be chosen manually, akin to selecting the number of hidden nodes in a neural network. For the various problems, I have chosen numbers through limited testing on particular pressure levels on the ECMWF/Aqua dataset, but they are not guaranteed to be optimal throughout the atmosphere, and certainly not optimal for different datasets.

Hyperparameter Initialization

One important consideration in SPGP training is the initialization of hyperparameters. The gradient descent portion of SPGP, like many other schemes that rely on maximizing likelihood, is theoretically vulnerable to being trapped in undesirable local minima [21]. Even if the manifold is accommodating enough that local minima are not a problem (unlikely in our high-dimensional problems with large training sets), it is still in our interest to initialize the SPGP hyperparameters so that gradient descent can finish as soon as possible, since SPGP is more time consuming than neural network training.

I initialized the hyperparameters in the following manner. For the projection matrix $\mathbf{P}$, I chose to initialize it using the PCA matrix, reasoning that the best linear dimensionality reduction technique should provide a good baseline to improve upon. Of course, there is much room for improvement here. For example, I do not take into account the SPGP-DR's use of the P as a substitute for the length-factor hyperparameters in regular SPGP without DR. A better initialization might scale P so that the points in the projected space are closer together in more "informative" dimensions.

For the hyperparameters $\mathbf{h} = [h_1, ..., h_i, ..., h_m]$ that control the heteroscedasticity of SPGP, I initialized by choosing a group of 100 points near each pseudo-input and computing their standard deviation of the targets. Intuitively, this would seem to generate good $\mathbf{h}$ values because $\mathbf{h}$ should be large for those pseudo-inputs that are in regions of with a fair amount of noise, since we might expect the variance of cases in that region to be higher. Although the initial $\mathbf{h}$ values may then not be scaled correctly, I compromised by simply dividing all the initial $\mathbf{h}$ by the mean. In case the standard deviation of the targets are extremely high or extremely low, this normalization of $\mathbf{h}$ will ensure that the mean of the $\mathbf{h}$ values is not affected, hopefully allowing for a more consistent and probable starting $\mathbf{h}$ and thus faster convergence during training.

Empirically, the initialization of hyperparameters mostly affects the training time

of SPGP. For example, randomly initializing the projection matrix (instead of using PCA to initialize) increases the training time (time until validation failure), but does not change the final log likelihood a significant amount (not more than what changes between separate trials). However, empirically there was a twofold increase in training time, so it is still important to have a good starting point for the hyperparameters.

Dimension and Pseudo-Input selection

Unlike the other hyperparameters, the number of dimensions and the number of pseudo-inputs must be fixed beforehand and cannot be found via gradient descent. Thus, the natural way to select good numbers is by exhaustively testing different settings on a validation set. Unfortunately, given the time-consuming nature of training, it was impractical to do this for each pressure level, just like it would be impractical to change the neural network structure at each level. Therefore, the parameters were chosen by limited testing of different hyperparameters on a chosen pressure level on the ECMWF/Aqua test set only. From that experience, the number of dimensions and number of pseudo-inputs only had very limited impact on the actual RMSE and variance prediction past a certain setting of the hyperparameters. Instead, the main tradeoff here is between accuracy of the results (both the parameter and variance predictions) and training and testing time (with larger values of the hyperparameters leading to slightly more accurate results). Clearly, this result is problem dependent, and it is essential to do more hyperparameter testing on any new remote sensing problem with more input parameters or a fundamentally different output.

For all the SPGPs I used 8 reduced dimensions, and 100 pseudo-inputs [1].

SPGP Training by Iteration

It is helpful to understand how the mean prediction and variance estimate of SPGP evolves as the hyperparameters are optimized to increase the likelihood. There are

[1]Since this was optimized for the ECMWF/Aqua dataset, with $d = 25$ inputs, this may be more than the optimal number of reduced dimensions for the precipitation dataset, which only has $d = 13$ inputs. However, there was no sign of overfitting on that dataset when comparing performance of the SPGP on the training and test set.

two distinct stages to SPGP training. First, the root mean square error (RMSE) of the targets compared to the predictions is reduced by optimizing the projection matrix P and the location of the pseudo-inputs. The predicted variance is generally constant and the same for all profiles. Then, the RMSE stays relatively constant while the variance estimates become more refined.

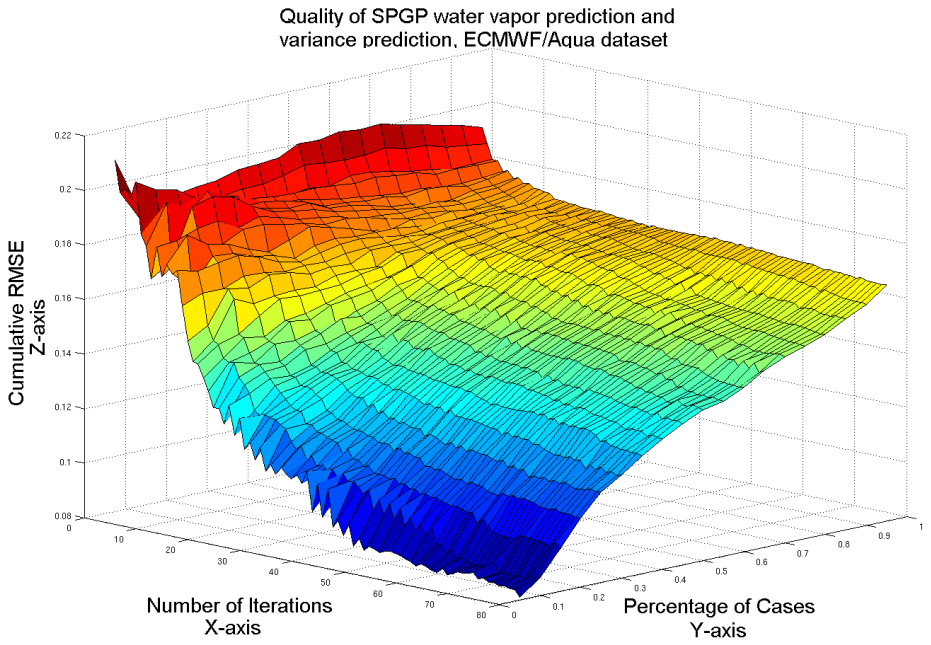


Figure 4-9: This figure shows how the RMSE (in normalized mass mixing ratio) and variance estimate of SPGP evolve over 80 iterations of gradient descent. The x-axis represents the training time. The 2D y-z plane shows the RMSE as a function of the predicted variance. As training time increases, the slope in the y-z plane becomes steeper, representing better variance estimation.

As an example, figure 4-9 shows how the RMSE and variance estimates evolve on the particular problem of estimating water vapor near the surface. The figure is set up so that the 2D slice at each iteration represents a graph of the cumulative RMSE. The numbers represent the percentage of profiles used to calculate the RMSE; the profiles chosen have the lowest estimated variance. For example, we can see that after 80 iterations, the 50 percent of profiles with the lowest estimated variance have an RMSE of around 0.12.

The general trend evident is that the RMSE (the right side of the figure) decreases

rapidly in the first ten or so iterations. However, past that RMSE remains relatively constant. By contrast, the quality of the variance (as measured by the steepness of the surface from left to right) is low in the first 10 iterations, but improves thereafter. The quantity that SPGP is optimizing, the log-likelihood of the data, decreases the most rapidly during the phase when RMSE is improving, and much more slowly thereafter (see figure 4-9).

This has some implications for SPGP training. Unlike the RMSE metric used by neural networks, the raw log-likelihood is not directly related to the quality of the retrieval. It is possible for the log-likelihood to be lower for some particular SPGP, but the variance prediction of that same SPGP could be poorer than another SPGP with a higher log-likelihood (although the RMSE would likely be better). It is also important to keep in mind during SPGP training that although the log-likelihood does not seem to be improving much per iteration later on, the actual variance predictions could still be improving quite dramatically.

SPGP consistency

The performance of SPGP is fairly stable across retraining the hyperparameters on the same problem, probably due to hyperparameter initializations biasing the optimization method toward certain local minima. Although there are indeed many local minima, they seem to give roughly the same performance in both variance estimation and parameter estimation. As an example, I ran five trials of SPGP on the HyMAS water vapor dataset, shown in figure 4-10. The SPGP is extremely consistent on that example.

4.3 Results on the Datasets

The dataset used in the subsequent section are the ECMWF/Aqua dataset and the HyMAS dataset, which were described in detail in chapter 2. Both datasets consist of detected radiances as inputs and geophysical parameters as targets.

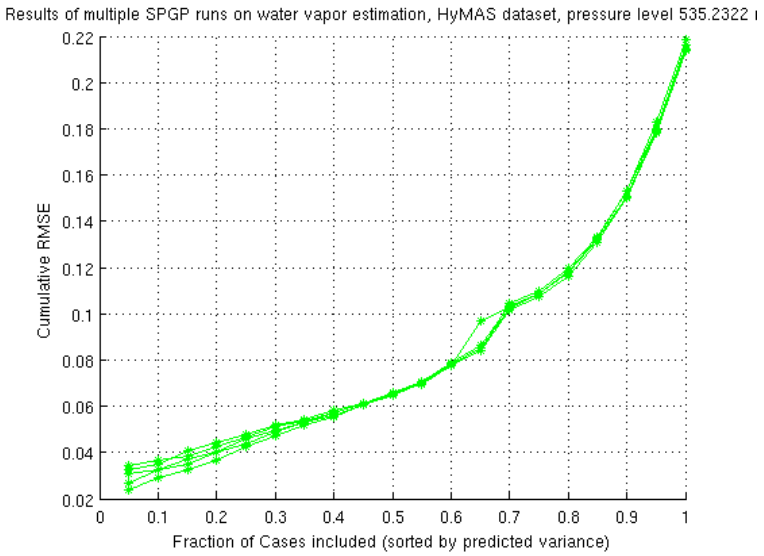


Figure 4-10: This figure shows 5 repeated trials of training SPGP to estimate water vapor on the HyMAS dataset, pressure level 535 mb. RMSE is in units of normalized mass mixing ratio.

4.3.1 Metric used

I used the same two presentation methods for the figures described in section 3.3. I also again used functions of latitude as a baseline for comparison (see section 3.3.1).

However, to compare between variations of the same general method (SPGP using different hyperparameters, for example), it can be advantageous to use the negative log-predictive density (NLPD) as an alternative metric to RMSE. The NLPD is defined for n test cases as:

$$\text{NLPD} = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{1}{2}\log\left((2\pi\sigma_i^2) + \frac{(t_i - y_i)^2}{2\sigma_i^2}\right)\right) \tag{4.32}$$

where σ_i, t_i, y_i are the variance prediction, the truth, and the mean prediction, respectively of the i^{th} test case. The NLPD is simply the likelihood that the set of cases are generated by the Gaussians with the predicted parameters. Empirically, the NLPD does very well at summarizing the performance of SPGP variants with a single number (see figure 4-11), which is to be expected given that SPGP picks parameters that maximizes the likelihood that the data is generated by a Gaussian process. Based

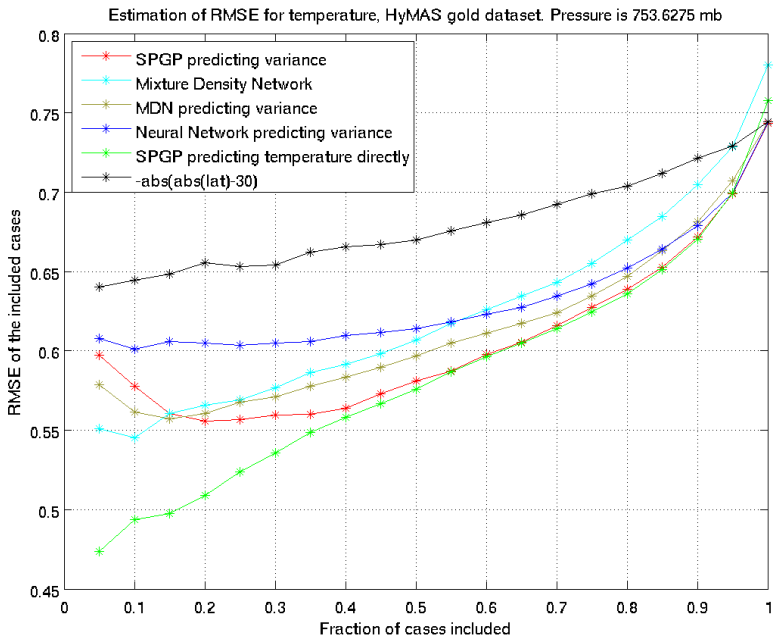


Figure 4-11: This figure shows various variance estimation methods on the HyMAS golden days dataset, pressure level 753 mb. RMSE is in degrees kelvin. Note that the SPGP estimating temperature directly (green) has a steeper slope than SPGP estimating the temperature residuals (red), but the green SPGP has a slightly higher overall RMSE (the RMSE at x = 1) than the red SPGP. However, NLPD of the green SPGP is 1.07, lower than the 1.17 for the red SPGP, which is consistent with our intuition that the green SPGP is "better" overall because of its superior performance at predicting variance.

on the same principle, it is also useful to compare between different MDNs with only one Gaussian component (MDNs with multiple Gaussian components do not predict Gaussian output distributions).

However, it is important to keep in mind that the two metrics (the plots and the NLPD) are not completely interchangeable. If the predicted variance is scaled by some factor, the graph of RMSE vs predicted variance will not change, but NLPD will. Because of this, NLPD is unfortunately not as useful in comparing between the neural networks and SPGP, at least not in the context of our problem. The variance predicted by neural networks on test sets is often too low, which causes the NLPD to be extremely large.

The most useful characteristic of NLPD is comparing between two methods that

give fairly accurate estimates of the variance, such as MDNs and SPGPs. If a problem demands an accurate estimate of the variance and any noise is assumed to be Gaussian, then NLPD can conceivably replace the other metrics used. Still, to give fair consideration to variance estimation neural networks, I did not use NLPD as the main metric for comparison.

4.3.2 Residual Estimation SPGP and MDNs

The most straightforward way to use SPGP or MDN is to apply it directly to the problem at hand, as a substitute for neural networks or linear regression. The inputs and the targets are the same as those given to a neural network. Although simple, this approach leads to some mixed results. In general, both the SPGP and the MDN are weaker in terms of estimating total RMSE than a neural network. For example, figure 4-20 compares the performance of SPGP and MDNs versus neural networks in estimating water vapor on the ECMWF/Aqua dataset.

However, another possibility for using the two methods described is as a substitute for the variance estimation neural network, so that the targets for the SPGP or MDN are now the residuals of the parameter estimation neural network. The SPGP (or MDN) thus becomes a post-processing stage, with the inputs the same as for the neural network, and the target function being the residuals of the parameter estimation neural network (see figure 4-12). Assuming that the parameter estimation neural network is unbiased in its estimation of the data, the residuals will have a mean of zero. But the SPGP or MDN should still be able to model the variance of the residuals, which should reflect any heteroscedasticity. Moreover, the SPGP or MDN variant can also still model any uncertainty caused by lack of data, an advantage over a variance estimation neural network.

A minor advantage of this approach is that the SPGP or MDN may be able to improve upon the performance of the parameter estimation neural network if it can model any pattern in the residuals. This is only possible if the neural network underfit. However, in the problems that I tested, subtracting the *predictions* of the residual estimation methods from the neural network prediction did not improve the

RMSE significantly (or at all, in some cases). Thus, in the following I only looked at the variance predictions of the residual estimation methods.

The most compelling reason for this approach over the direct approach is that we do not need to discard any previous neural networks, especially when they have been working well at estimating parameters (temperature, water vapor), and estimating them more accurately than the SPGP and MDN. There can be a significant difference, especially for difficult problems like the HyMAS water vapor estimation.

Discussion of residual estimation SPGP

For SPGP, an additional slight advantage is that the time required to optimize hyperparameters for SPGP is often reduced. This is related to how SPGP training proceeds, where precipitous decreases in negative log likelihood first occur due to increasingly more accurate parameter estimation, and only later does the log likelihood decrease due to optimizing the predicted variance. In the case of residual estimation SPGP, the first part of the training (learning to estimate the parameters correctly) is effectively skipped, since there should be theoretically be no more improvement possible in parameter estimation. However, in a few problems, adding the mean predicted by SPGP does improve the overall RMSE by a very small amount, indicating that the neural network was stuck in a suboptimal local minima.

One potential objection to this method in the case of SPGP is that the model uncertainty (uncertainty from lack of data) is dependent on the complexity of the function at that point. If a function is varying rapidly, the intuitive expectation is that the uncertainty increases very quickly when extrapolating. On the other hand, if a function is mostly constant, there should be a higher degree of confidence in the extrapolation. For example, imagine being asked to predict the trajectory of a fighter jet engaged in a dogfight versus that of car driving along a straight stretch of highway. The residual estimation SPGP uses the residuals as the targets, and the mean of the residuals (the "car trajectory" we are estimating) is on average constant and zero. On the other hand, the actual geophysical parameter function is not constant and is changing rapidly as a function of radiance. Thus, it is possible that the predicted

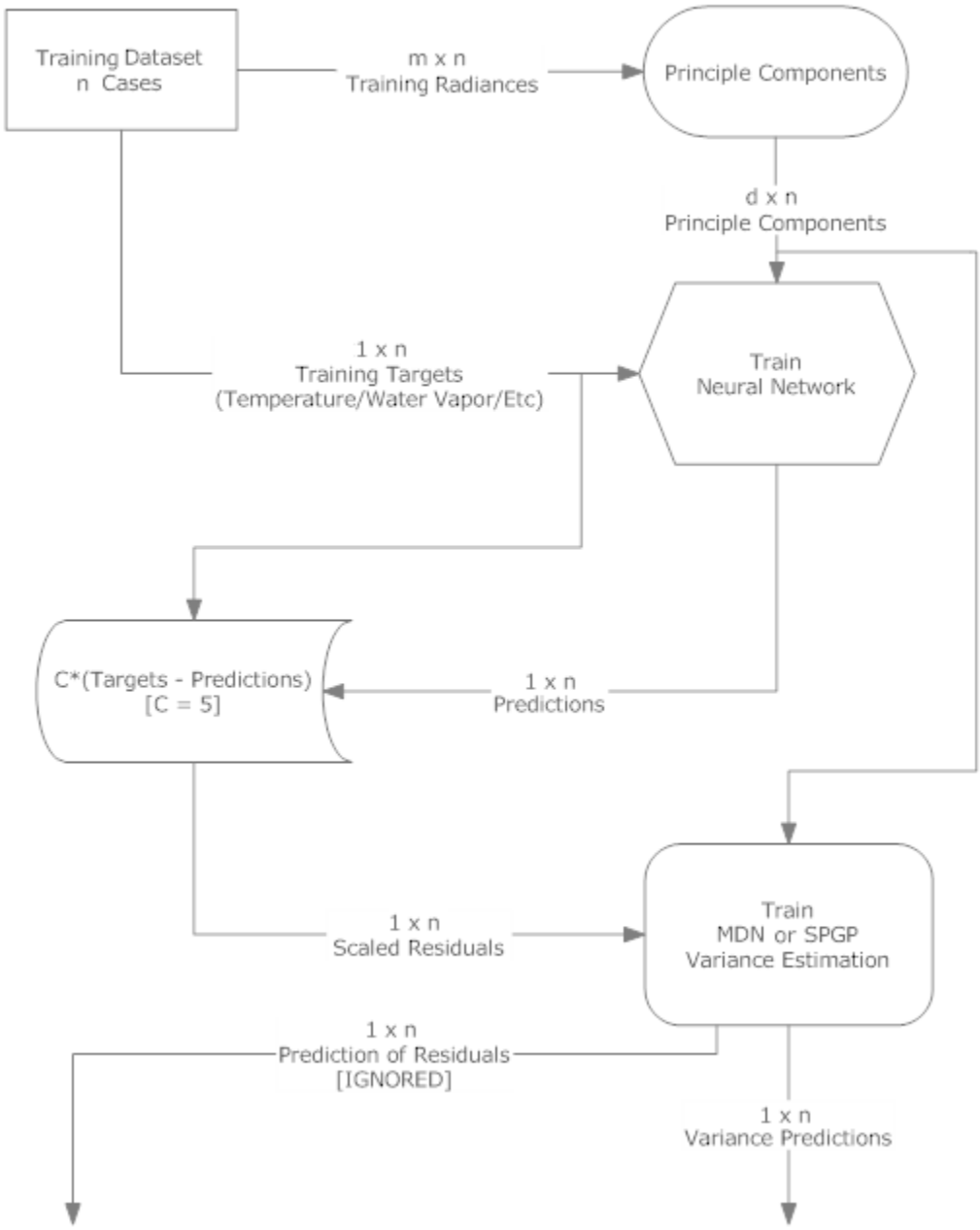
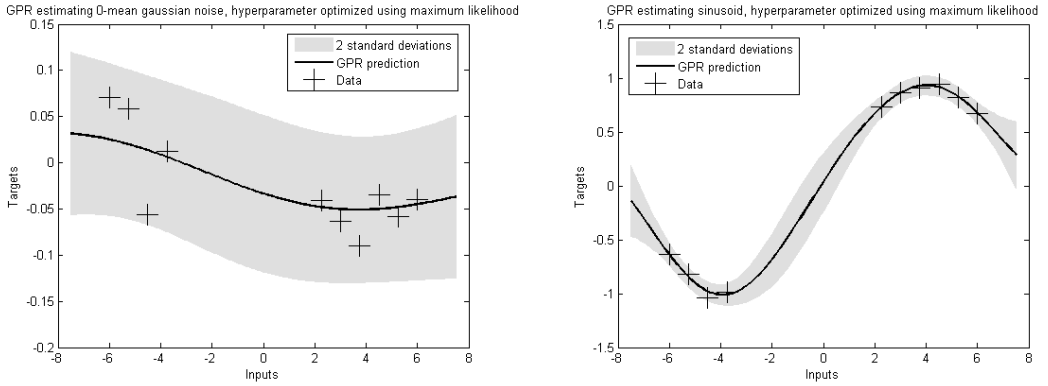


Figure 4-12: A block diagram showing how MDNs and SPGPs can be used to estimate the variance, so as to take advantage of the neural network's superior parameter estimation. The targets are scaled up by 5 because I found that doing so helped prevent the residual estimation SPGP from being trapped in local minima, possibly due to the initializations of the hyperparameters that I used. Of course, later the predicted mean and standard deviation are scaled down by 5 to compensate.



(a) This shows the GPR estimate of an approximately constant function with additional Gaussian noise. The hyperparameters are learned by optimizing the likelihood. The predicted standard deviation is fairly constant.

(b) The inputs are the same as in figure 4-13(a), but the target is now a sinusoid. The predicted standard deviations are clearly different from 4-13(a) even though the added noise is the same.

Figure 4-13: The y-data is generated from two different functions of x, but the additional noise is the same in both.

variance is lower than it should be in areas where we need to extrapolate, since the SPGP is estimating the uncertainty in the trajectory of the car, when it should be estimating uncertainty in the trajectory of the fighter jet.

Of course, it is true that a standard Gaussian process regression should predict the same standard deviation regardless of the target values of the training samples, since the target-values y only appear in the equation for the posterior mean (see equation (4.30)), and not in the equation for the standard deviation. However, this result depends on the hyperparameters being the same. Optimizing the hyperparameters using the likelihood as the metric can lead to very different hyperparameters if there are different target values, as intuition should suggest. An extreme example is shown, comparing a flat, constant function (figure 4-13(a)) to a sinusoid (figure 4-13(b)). The predicted standard deviations of figure 4-13(a) are clearly different from figure 4-13(b) even though the added noise is the same in both. This shows that optimizing the hyperparameters using the likelihood as the metric can lead to very different hyperparameters, and thus different variance predictions, if there are different target values, as intuition should suggest.

Unfortunately, because the hyperparameters are optimized via gradient descent,

it is not possible to say exactly how the variance estimate will be changed by using the SPGP as a post-processing stage as opposed to using it to estimate parameters directly. Any problems can only be identified empirically. One notable problem was the initially poor performance of residual estimation SPGP on some water vapor estimation problems due to being trapped in local minima, which was fixed by scaling all the target residuals by 5. This seems to be mostly due to the initializations I used for the hyperparameters–presumably using different initializations would have had the same desired result of avoiding local minima.

Discussion of residual estimation MDNs

For MDNs, there is no potential pitfall of underestimating variance due to the smoother target function, since it would not take into account model uncertainty in the first place. However, both residual estimation MDNs and residual estimation SPGP would still be vulnerable to another problem, that of the neural network overfitting. If the neural network overfits, it is very likely that the variance predictions from MDN or SPGP will also be compromised, since low residuals on the training data could simply be due to overfitting and not due to low variance in that area (see figure 4-14 for a crude example). In that example, the MDN or SPGP that is employed to estimate the variance of the neural network will predict low variance, whereas an MDN or SPGP predicting the data directly may have predicted much higher variance. The MDN or SPGP would have no way to detect whether the low residuals of the neural network was due to overfitting or due to low variance if their targets were neural network residuals rather than the actual geophysical parameters themselves.

These potential weaknesses are useful to keep in mind when deciding how to apply MDNs and SPGPs to future problems.

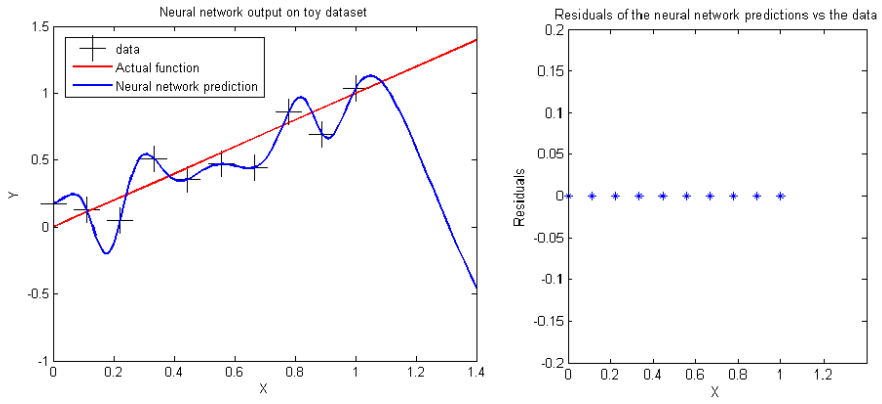


Figure 4-14: The figure on the left shows the toy dataset created by taking a function (red) and adding some random noise, as well as showing the function predicted by the neural network (blue). The neural network parameters were deliberately chosen to allow overfitting. The figure on the right shows the residuals on the data points, which are all near zero. From the figure on the right, the variance (the added noise) looks as if it would be zero, if it were to be predicted by any residual estimation method.

4.3.3 ECMWF/Aqua dataset results, temperature

Parameter estimation performance

Figure 4-15 shows the RMSE profile when estimating temperature on this dataset. The SPGP slightly trails neural networks by an average of 1.4 percent, although there are a few levels in which the SPGP does better than the neural network. The MDN trails by roughly 1 percent. Overall, there does not seem to be a large advantage for using a neural network in this case, especially considering that different runs of the neural network can change the RMSE by more than 2 percent.

Variance prediction performance

For a quick comparison between the two most successful methods of MDN and SPGP, see figure 4-16, which shows the NLPD of both methods for all pressure levels. All methods perform roughly equally, although the residual estimation methods may have a slight advantage in the lower atmosphere due to their more accurate temperature estimation.

Generally, it appears that SPGP and MDN can better predict the low-variance

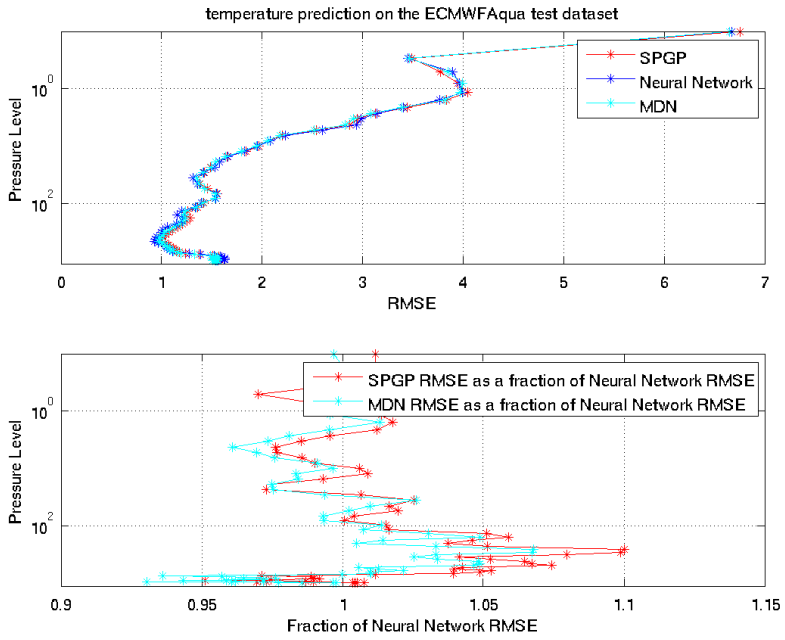


Figure 4-15: This figure shows the RMSE profile in the ECMWF/Aqua dataset when estimating temperature. RMSE is in kelvins and the pressure level is in millibars.

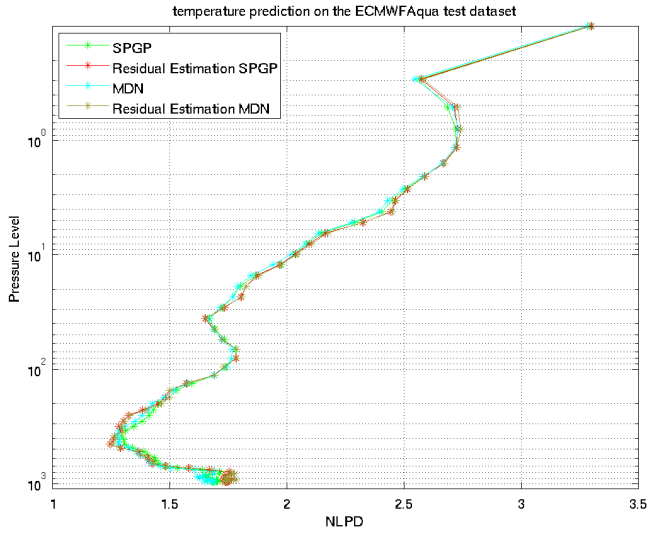


Figure 4-16: This figure shows the NLPD profile of MDNs and SPGPs on the problem of temperature estimation. NLPD is in kelvins and pressure level is in millibars

cases. If the predicted variance of a group of profiles is low, the actual RMSE of that group is usually also low. In contrast, on many levels the variance estimation neural network shows very little skill below a certain variance; all the profiles with predicted variances below that have roughly similar RMSE.

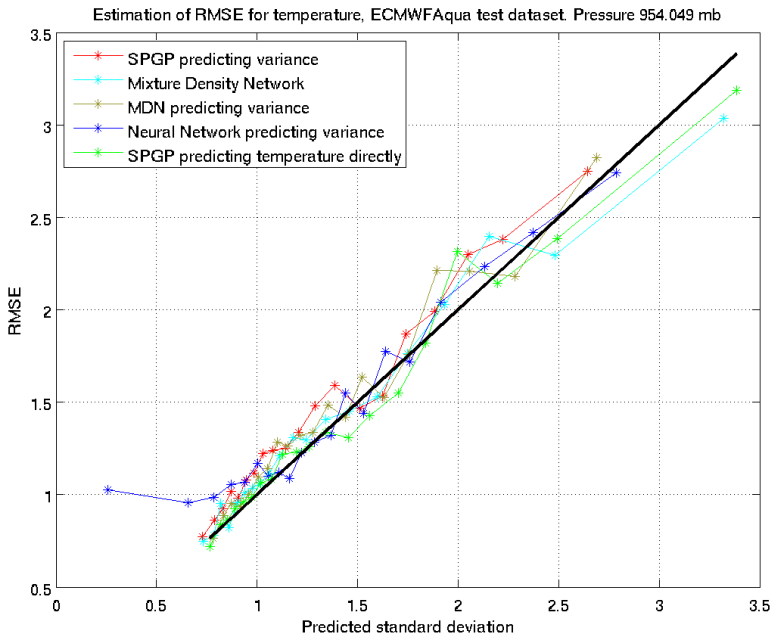


Figure 4-17: This figure shows the performance of the methods on the problem of estimating temperature on the ECMWF/Aqua dataset at pressure level 954 mb. The RMSE and predicted standard deviations are in degrees kelvin.

For example, on pressure level 954 mb (see figure 4-17), the RMSE of the 50 percent of the samples with the lowest predicted variance by the neural network is actually the same in all bins, meaning that the neural network shows no skill in estimating the difficulty of 50 percent of the cases. The SPGP variants both have superior variance estimation on that problem.

In fact, SPGPs and MDNs consistently show skill in predicting variance for all cases. The primary result in temperature is that the neural network can identify the most troublesome or noisy cases, but lumps all the easier cases together without distinction. SPGP and MDN can separate those easier cases out more finely.

The two different approaches to using SPGPs or MDNs, the residual estimation method and using the method to predict the parameters directly, exhibit varying

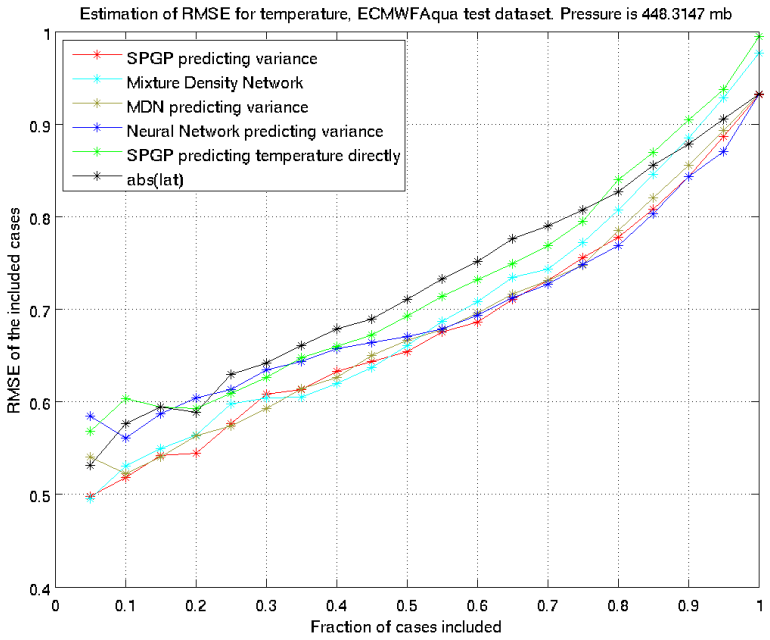


Figure 4-18: This figure compares the performance of various techniques to estimate variance on the problem of temperature retrieval on the ECMWF/Aqua dataset at pressure level 448 mb. The RMSE is in degrees kelvin.

performance throughout the atmosphere. In general, the residual estimation method is equal or slightly inferior in variance prediction to that of the method predicting the temperature directly, if allowance is made for the fact that direct temperature estimation is sometimes worse than that of a comparable neural network. Still, at some pressure levels in the upper atmosphere (see figure 4-18), the higher RMSE of MDNs or SPGPs (compared to neural networks) becomes a major concern. There, the residual estimation method may be a better option.

On the other hand, on some pressure levels, notably from 730 to 940 mb, SPGP and MDN actually gives superior RMSE to neural networks in estimating temperature, so that the major advantage of residual estimation method is no longer applicable (see figure 4-15). Finally, from 940 mb to the surface, SPGPs and MDNs simply gives much better results at estimating variance than the residual estimation methods, by virtue of being able to accurately identify the least troublesome cases with the lowest RMSE. This trend shows up in the training dataset (albeit less strongly) as well as the test dataset, so it is not completely a problem of overfitting on the part

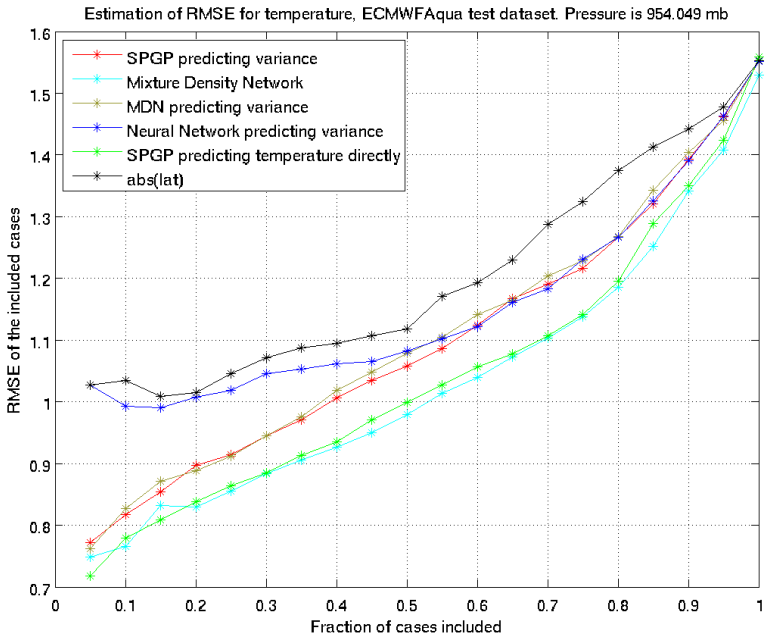


Figure 4-19: This figure shows the performance of the methods on the problem of estimating temperature on the ECMWF/Aqua dataset at pressure level 954 mb. The RMSE is in degrees kelvin. This is the same problem as in figure 4-17, except graphed by cumulative RMSE instead of RMSE by bins.

of the residual estimation SPGP.

Here, it appears that the problem is that of the neural network overfitting, because the residual estimation MDN also exhibits poorer variance estimation than the MDN estimating temperature directly. As mentioned before, neural network overfitting can lead to residual estimation MDN or SPGP predicting low variance based on overfitted training data. The symptom of that is poorer variance estimation for cases with lower variance, which is exactly what happens on the pressure levels 940mb to the surface. An example of that is shown in figure 4-19, which is the same problem as figure 4-17, except graphed by cumulative RMSE to better show the poorer variance estimation of the residual estimation methods.

The only major difference between the performance of SPGP and MDN in the case of temperature estimation is the relative performance of the methods on the training and the test set. MDN tends to do slightly better on the training set, perhaps an indication of overfitting, whereas the performance of the SPGP in both the training

and test sets are roughly equal.

Overall, the residual estimation methods do not offer a huge advantage over simply using a SPGPs or MDNs directly for estimating temperature, mostly because the SPGP or MDN RMSE is quite competitive with that of a neural network. However, for variance estimation, both SPGP and MDN are clearly better than a neural network. For a fairly linear problem like temperature estimation, either SPGPs or MDNs are a good choice to characterize uncertainty.

4.3.4 ECMWF/Aqua results, water vapor

Water vapor was normalized, so *a priori* standard deviation is unity throughout the atmosphere.

This was also the dataset where I tested different settings of the SPGP hyperparameters. In particular, I used the problem of estimating water vapor content near the surface. I chose the hyperparameters (number of reduced dimensions, number of pseudo-inputs) mainly to optimize RMSE while keeping the training time as short as possible.

Parameter Estimation Performance

Estimating water vapor is considered a harder problem than that of estimating temperature. In the water vapor estimation problem, using the same ECMWF/Aqua dataset, the SPGP does worse than the neural net by a slightly larger margin, averaging 4.5 percent worse throughout the atmosphere, and 5.7 percent worse at pressures below 200 millibars (see figure 4-20). The MDN also lags slightly behind neural networks, doing on average 3.82 percent worse throughout the atmosphere and 4.9 percent worse at pressures below 200 millibars, but overall this is still better than SPGP.

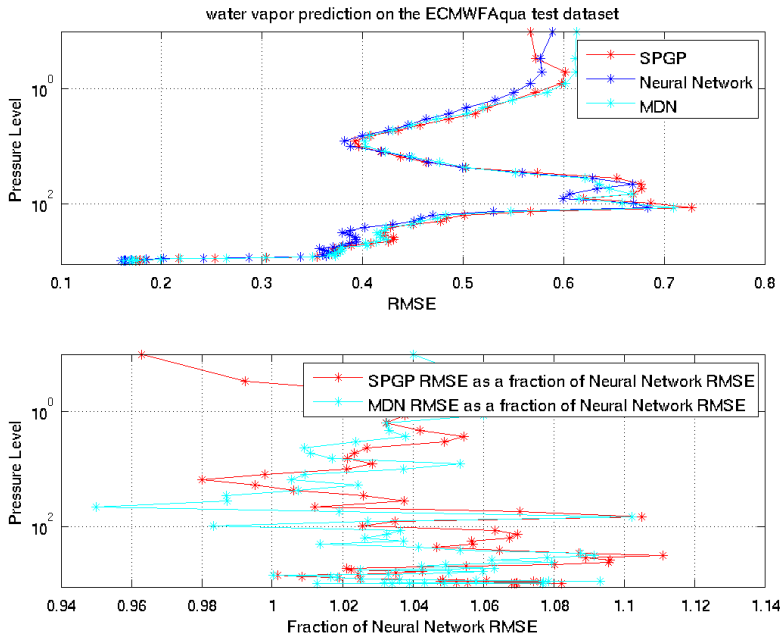


Figure 4-20: This figure shows the RMSE profile in the ECMWF/Aqua dataset when estimating water vapor, using MDNs, SPGPs, and neural networks. RMSE is in units of mass mixing ratio, and pressure is in millibars.

Variance estimation performance

For a quick comparison between the two most successful methods of MDN and SPGP, see figure 4-21, which shows the NLPD of both methods for all pressure levels. Here, the MDN is generally the best method according to this metric.

In terms of comparisons between SPGP and variance estimation neural networks, an interesting case occurs on a few levels: the SPGP (including residual estimation SPGP) shows great skill at predicting what the easiest cases are, but has difficulty in separating out the high variance cases (at least, it does worse at this than a variance estimation neural network). Visually, the lines representing cumulative RMSE of the SPGP and the neural network often intersect at some point before 100 percent of the cases are included. Afterwards, the slope of the SPGP line becomes flatter as compared to that of the neural network. A good example is at pressure level 132.5 mb when estimating water vapor (see figure 4-22), where the SPGP line and the neural network line cross over at $x = 0.55$. In those cases, the neural network is better able

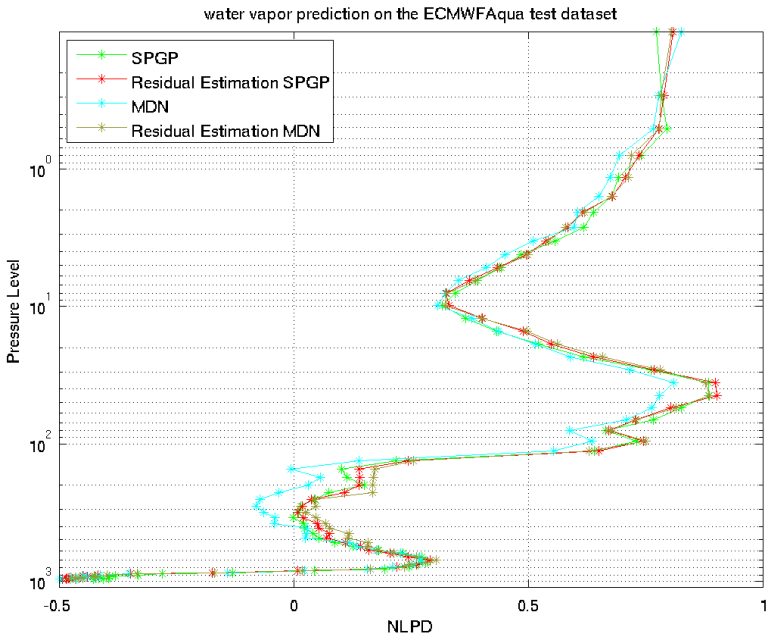


Figure 4-21: This figure shows the NLPD profile of MDNs and SPGPs on the problem of water vapor estimation. NLPD is in units of mass mixing ratio.

to estimate variance of cases with high RMSE, but does worse at estimating variance of cases with low RMSE. A plot of the actual predicted variance versus the RMSE confirms this (see figure 4-23).

Note that both SPGP variants predict high, but similar, variance for the 30 percent hardest cases, whereas the variance estimation neural network separates out these harder cases more finely. It is possible that the SPGP predicts a smoother function of the variance as a function of the inputs (so that all cases in a region of the input space are predicted to have a similar high variance), while the variance estimation neural network attempts to model a much more complicated function (so that for the same region the neural network predicts a much broader range of variances), possibly due to the extra emphasis the RMSE metric would place on modeling high residuals well. A contributing factor to this may be that SPGP is simply not be able to model more complicated function of the variance due to the restrictions of having only 100 pseudo-inputs to work with, although upping the number of pseudo-inputs to 200 did not appreciably change anything on this level.

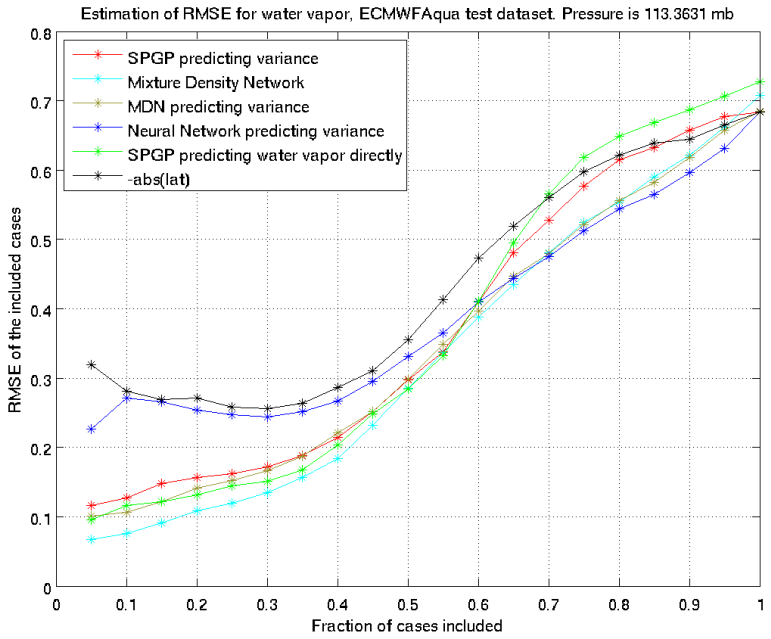


Figure 4-22: This figure shows the performance of various variance estimation methods on estimating water vapor on the ECMWF/Aqua dataset at pressure level 113 mb. RMSE is in units of normalized mass mixing ratio.

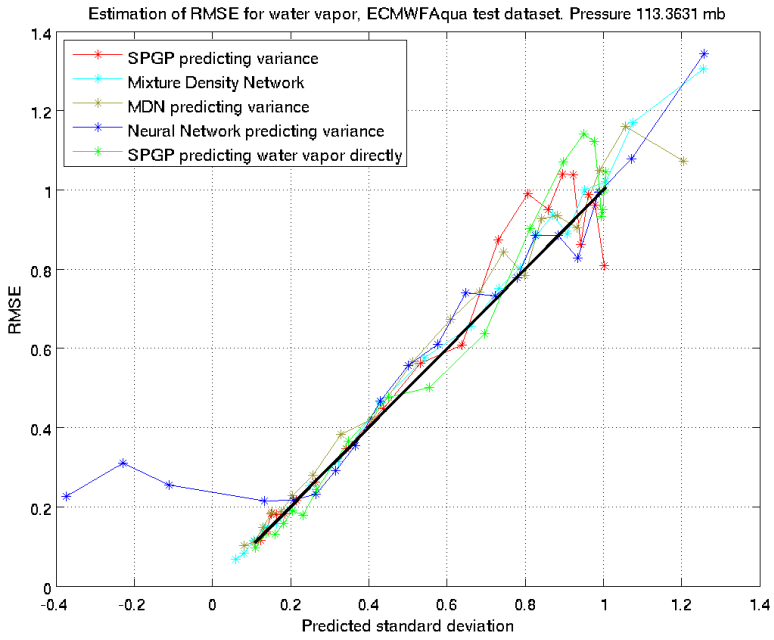


Figure 4-23: This is the same problem as depicted in figure 4-22, except that presented with the RMSE of each group of cases instead of the cumulative RMSE of the cases. RMSE is in units of normalized mass mixing ratio.

Finally, note that SPGP predicts high variance in two distinct cases: one where the heteroscedastic nature of the problem dominates, and one where uncertainty due to lack of data dominates. In the latter case, the actual RMSE from the test set data may actually be low, even though we do not have as much confidence in the prediction. On the other hand, the variance estimation neural network, and indeed the MDN also, would simply extrapolate the variance to be low, even if there is not much data available. Although it is certainly desirable to know where the model is uncertain of its predictions due to lack of data, the advantage this brings cannot be quantified in our chosen metric, and so the SPGP may be penalized.

There are a few levels where MDNs also exhibit slightly poorer variance estimation of high error cases than neural networks, though the gap in performance is smaller than between the SPGP and the neural network. It seems both MDNs and SPGPs, methods that optimize the maximum likelihood, give similar answers when modeling complicated variance functions. Still, because of the other SPGP specific problems mentioned, MDNs may be better choice than SPGP for modeling the variance of more complicated functions, if there is confidence that the training data thoroughly covers all possible test cases, and it is not feasible to incur extra training time by increasing the number of reduced dimensions in SPGP.

In general the MDN and the SPGP do no worse than neural networks when estimating variance, and in many cases does much better when separating out cases with low RMSE. Barring the few problematic levels discussed before, the variance estimation performance of residual estimation MDNs is also quite similar to that of residual estimation SPGP on both of the ECMWF/Aqua problems.

However, the MDN does not suffer as much from poor water vapor estimation, so it is unnecessary to use a residual estimation MDN here over directly applying MDN. On the other hand, because SPGP is not as good as a neural net when estimating the actual relative water vapor content, it may a good idea to use a residual estimation SPGP on fairly nonlinear problems like water vapor estimation.

4.3.5 HyMAS results, temperature

Recall that the HyMAS dataset consists of both the normal test data, draw from the same overall dataset as the training data, and a completely separate "golden days" test dataset.

Temperature estimation performance

In the test dataset, for the pressure levels studied, the neural network RMSE is on average 9.83 percent better in estimating temperature RMSE than SPGP (see figure 4-24). MDNs on average lag behind by 9.25 percent. Both MDNs and SPGP perform noticeably worse here than in the results of ECMWF/Aqua. This could be due to the HyMAS data being less noisy (the radiances only have simulated instrument noise, and imperfect spatial matching is no longer a concern). Consequently, the temperature could be a more complex function of the inputs since it can depend on high order components of the PCs that were too noisy in the ECMWF/Aqua dataset. The MDNs and the SPGP, which both rely on optimizing maximum likelihood instead of the RMSE, could simply be less adept at modeling more complicated functions. An more likely hypothesis is that the neural network is overfitting to the training set (due to the dataset as a whole being simulated, and thus less noisy than the ECMWF/Aqua data).

This latter explanation is supported by the performance of the methods on the golden days set; the neural network is only better than SPGP by 1.4 percent and better than MDN by 2.4 percent (see figure 4-25). This implies that either the neural network was overfitting to the training dataset, or that there are features in the golden days set that are simply not present in the training set (which is more unlikely given that SPGP and MDN performance did not decrease nearly as much). Regardless, both MDN and SPGP have learned the most broadly applicable features of the training dataset, explaining its better relative performance. It seems the two maximum likelihood optimization methods are more resistant to overfitting.

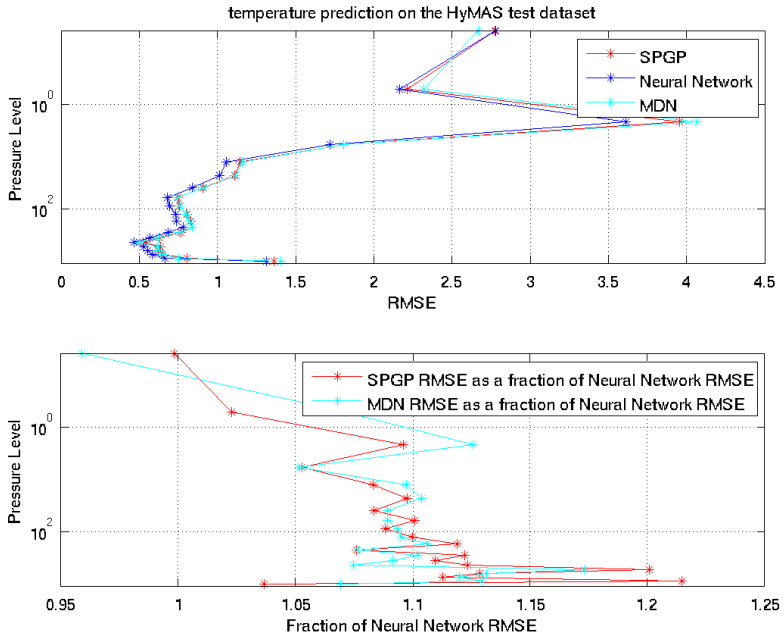


Figure 4-24: This figure compares the performance of various methods on estimating temperature on the HyMAS test dataset. The y-axis represents the pressure level in millibars (surface is at the bottom). The RMSE is in degrees kelvin.

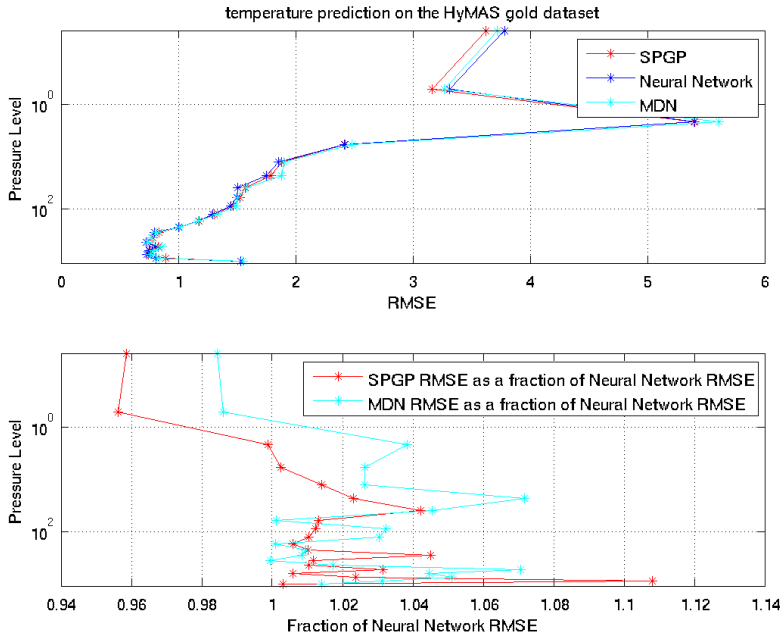


Figure 4-25: This figure compares the performance of various methods on estimating temperature on the HyMAS golden days test dataset (see text for dataset details). The y-axis represents the pressure level in millibars (surface is at the bottom). The RMSE is in degrees kelvin

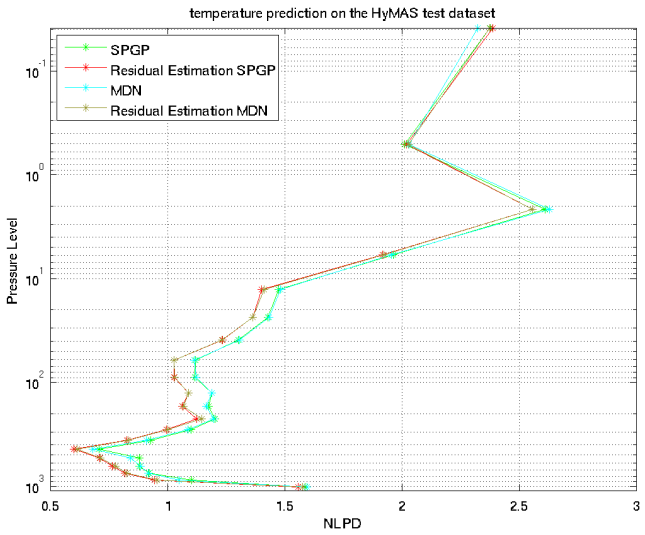


Figure 4-26: This figure shows the NLPD profile of MDNs and SPGPs on the problem of temperature estimation. NLPD is in degrees kelvin, and pressure is in millibars.

Variance estimation performance

For a quick comparison between the methods of MDN and SPGP, see figure 4-26, which shows the NLPD of both methods for all pressure levels on the test set. On that test set, the residual estimation methods are superior, due primarily to the poorer temperature estimation performance of SPGPs and MDNs. Figure 4-27 shows the NLPD of both methods on the golden days test set. There, the SPGP is generally the best method, and both MDNs and SPGPs are superior to the residual estimation methods.

The SPGP and neural network estimation of variance on the test set are for the most part very similar to the results obtained on the ECMWF/Aqua dataset. In all levels, the residual estimation SPGP either does the same, or better, than the neural network doing the same. The residual estimation MDN had roughly the same performance as the residual estimation SPGP (there were a few levels where the MDN was better, and a few where SPGP was better). The residual estimation MDN did have slightly better performance than the residual estimation SPGP on a few levels on the training set, but this did not carry over to the test set, possibly indicating some overfitting.

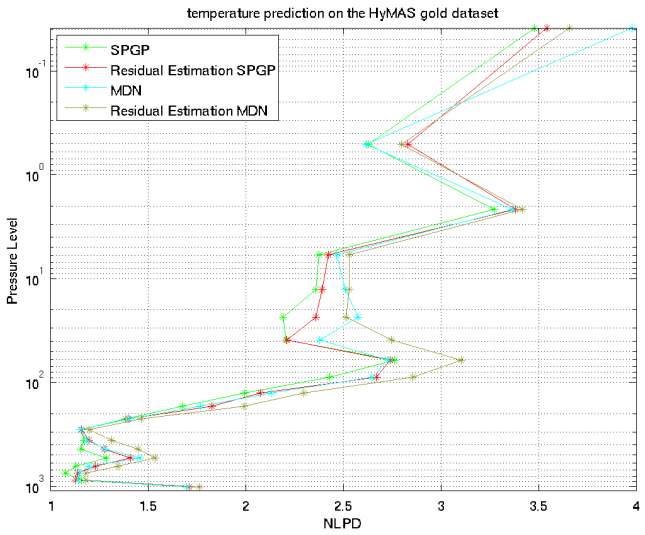


Figure 4-27: This figure shows the NLPD profile of MDNs and SPGPs on the problem of temperature estimation on the golden days set. NLPD is in degrees kelvin, and pressure is in millibars.

As mentioned before, both the MDN and the SPGP estimating temperature directly has much higher RMSE throughout, so that they are not competitive on the test set.

The more interesting cases occurs in the golden days set. For the most part, the neural network does much worse at estimating variance on the golden days set as compared to the neural network. An extreme example occurs at pressure level 56 (see figure 4-28 for the test set, and figure 4-29 for the golden days set). Although all methods underestimate the variance on the golden days dataset, the variance estimation neural network severely underestimates variance, suggesting the variance estimation network was overfitting on the training dataset.

The SPGP directly estimating temperature also becomes much more competitive, as expected from the relative improvement in temperature prediction as compared to the test set. In fact, on some levels such as pressure level 81 (see figure 4-30 for the test set, and figure 4-31 for the golden days set), the SPGP is the best method for estimating variance on the golden days set, although it was the worst performer on the test set. The MDN also shows a large relative improvement compared to its performance on the test set.

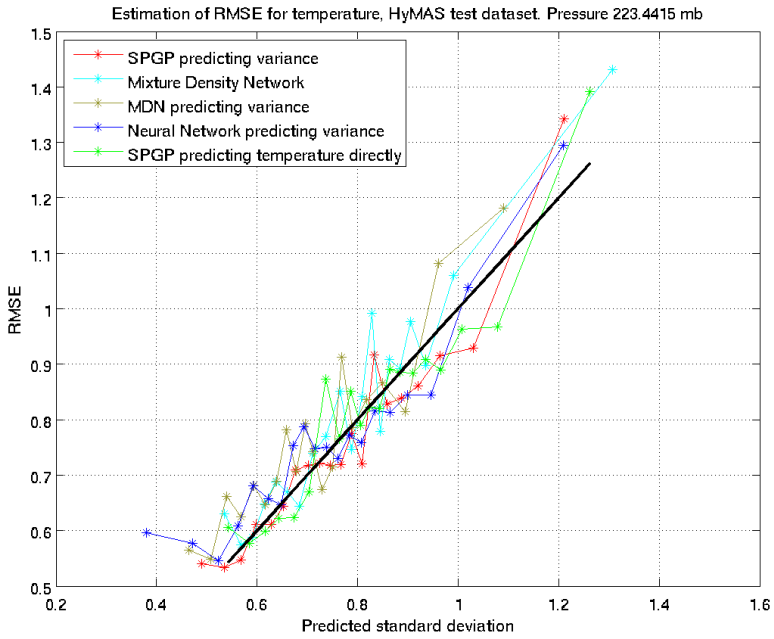


Figure 4-28: This figure compares various methods for estimating variance on the HyMAS test dataset, with respect to temperature at pressure level 223 mb. RMSE is in degrees kelvin.

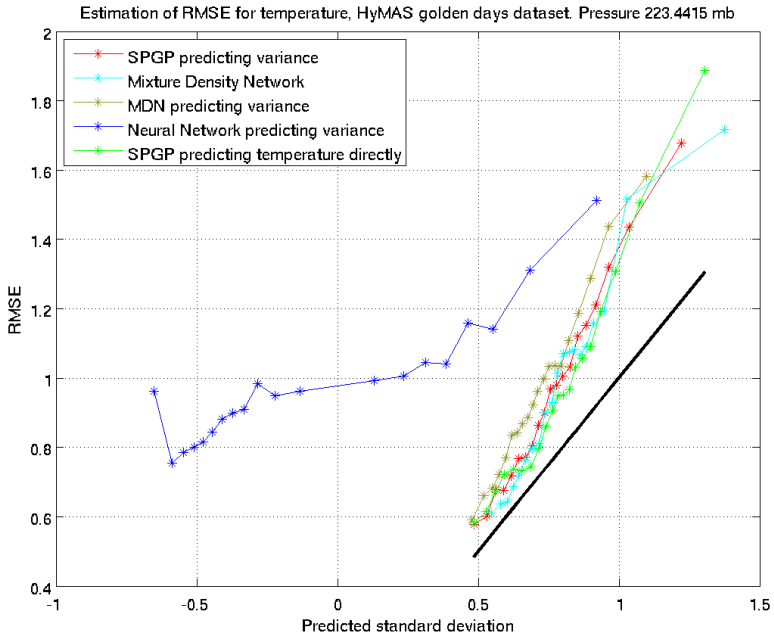


Figure 4-29: This figure compares various methods for estimating variance on the HyMAS golden days test dataset, with respect to temperature at pressure level 223 mb. Compare to figure 4-28. RMSE is in degrees kelvin.

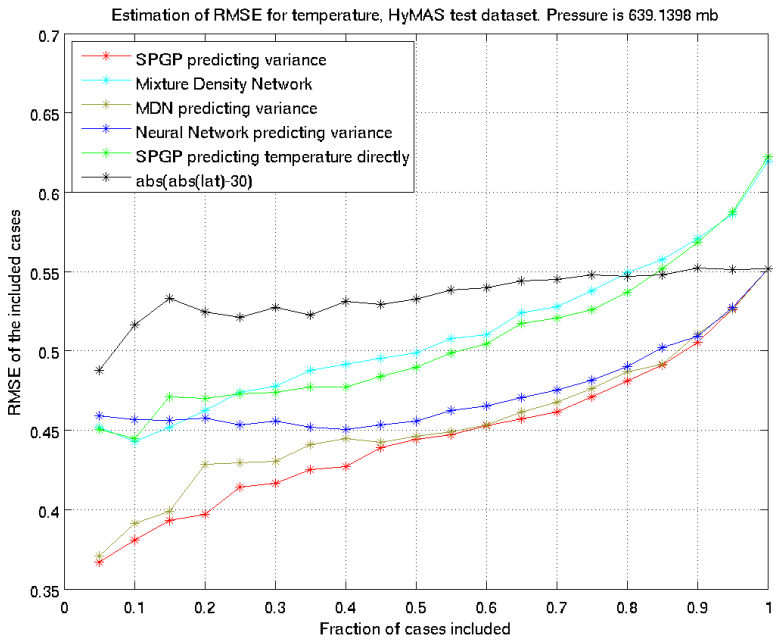


Figure 4-30: This figure compares various methods for estimating variance on the HyMAS test dataset, with respect to temperature at pressure level 639 mb. RMSE is in degrees kelvin. This figure is presented in cumulative RMSE as opposed to RMSE per bin to facilitate comparisons between the various methods (see section 3.3 for an explanation of the two presentation schemes).

One possible explanation for the discrepancy in performance between the test set and the golden days set is that the variance estimation neural network, much like the parameter estimation neural network, has simply overfit to the training data. If this is the case, it would mean that variance estimation neural networks may need much more data than the 30000 training cases provided in order to get a stable estimate of the variance. This can of course be mitigated somewhat by reducing the number of model parameters (hidden nodes), but this strategy is complicated by the fact that there is no obvious sign of overfitting on the test dataset (the training and test performance are similar).

Another possibility, which is closely related, is that the golden days set is fundamentally different in some way. This is somewhat backed up by the fact that a simple function of latitude, which guesses that the error is highest near the poles and the tropics, is quite effective on the golden days set, while being completely ineffective on

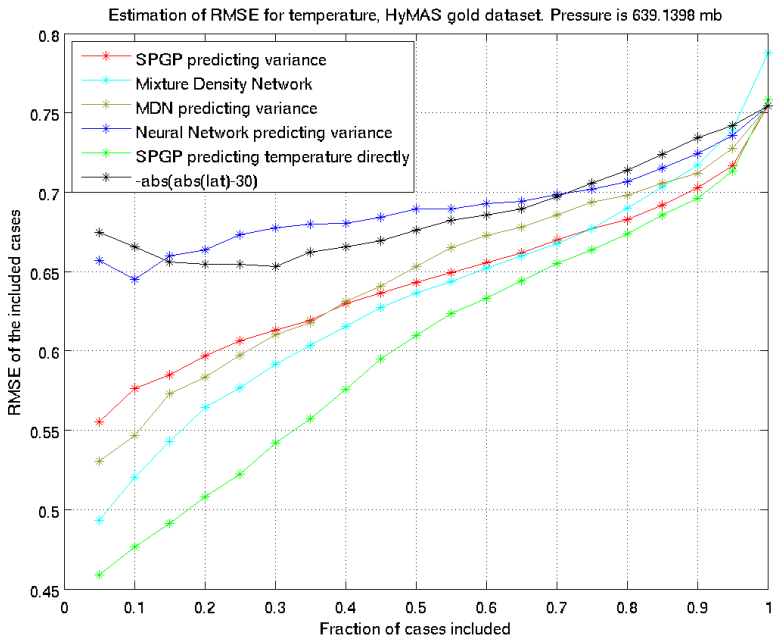


Figure 4-31: This figure compares various methods for estimating variance on the HyMAS golden days test dataset, with respect to temperature at pressure level 639 mb. Compare to figure 4-31. RMSE is in degrees kelvin.

the test set. It is also notable that there are several pressure levels higher up in the atmosphere where no method show skill in estimating variance, suggesting a vastly different variance function as a function of the inputs, at least on those levels (see figure 4-32). Those were also the levels where the neural network performed worse than linear regression at estimating the temperature (see figure 2-7), again suggesting that there are some features, correlated with temperature, which are present in the training dataset but not in the golden days dataset.

Still, whether the poor performance of the variance estimation neural network stems from overfitting or from deficiencies in the training data, both MDNs and SPGPs generalize much better on the golden days set. It is also notable that SPGP achieves the best variance prediction performance by far on some levels in the golden days dataset (such as pressure level 639 mb, shown in figure 4-30), possibly indicating that the extrapolation required on those levels rewards SPGP's ability to account for model uncertainty. It could also be the case that the reduced degrees of freedom due to only having 8 input dimensions prevents overfitting. However, the residual

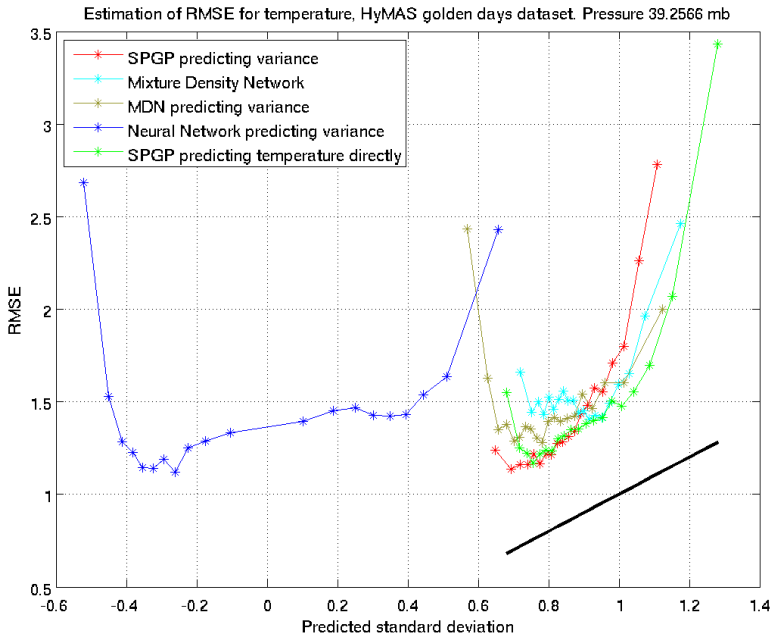


Figure 4-32: The figures compares various methods for estimating variance on the HyMAS golden days test dataset, with respect to temperature on pressure level 39 mb. The RMSE is in degrees kelvin.

estimation MDN and the residual estimation SPGP have similar performance despite the residual estimation SPGP also having only 8 input dimensions, which seemingly discounts overfitting being the main cause of the discrepancy in performance between SPGP and the rest of the methods.

4.3.6 HyMAS results, water vapor

Water vapor estimation performance

The water vapor results mirror those of the temperature. Again the neural network does much better than SPGP at estimating relative water vapor on the test set (shown in figure 4-33), averaging 10.35 percent better RMSE than SPGP, but does only slightly better than SPGP on the golden days dataset (shown in figure 4-34), averaging 4.2 percent improvement in RMSE. Similarly, the MDN lags 7 percent behind the neural network on the test dataset, compared to 5.5 percent worse RMSE on the golden days dataset.

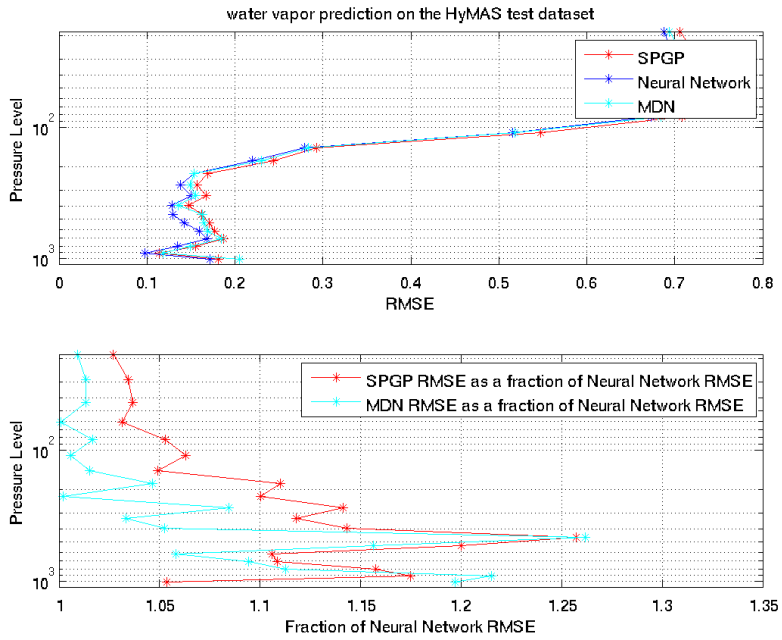


Figure 4-33: These charts compare the performance of the methods in estimating water vapor on the HyMAS test dataset (see text for dataset details). The y-axis represents the pressure level in millibars (surface is at the bottom). RMSE is in normalized mass mixing ratio.

Variance Estimation Performance

For a quick comparison between the methods of MDN and SPGP, see figure 4-35, which shows the NLPD of both methods for all pressure levels on the test set. On the test set, the methods are all competitive, with MDN having a slight edge due to its good water vapor estimation accuracy at a few levels. Despite the poor parameter estimation performance of SPGP and MDNs, their superior variance estimation performance compared to the residual estimation MDNs and SPGPs equalizes their NLPD.

Figure 4-36 shows the NLPD of the methods on the golden days test set. The methods are again all competitive.

Unlike the temperature results, the variance estimation neural network's performance on the golden days dataset is only slightly worse compared to its performance on the test set. On the other hand, even on the test set the variance estimation neural network has problems modeling variance well. This could be due to the water

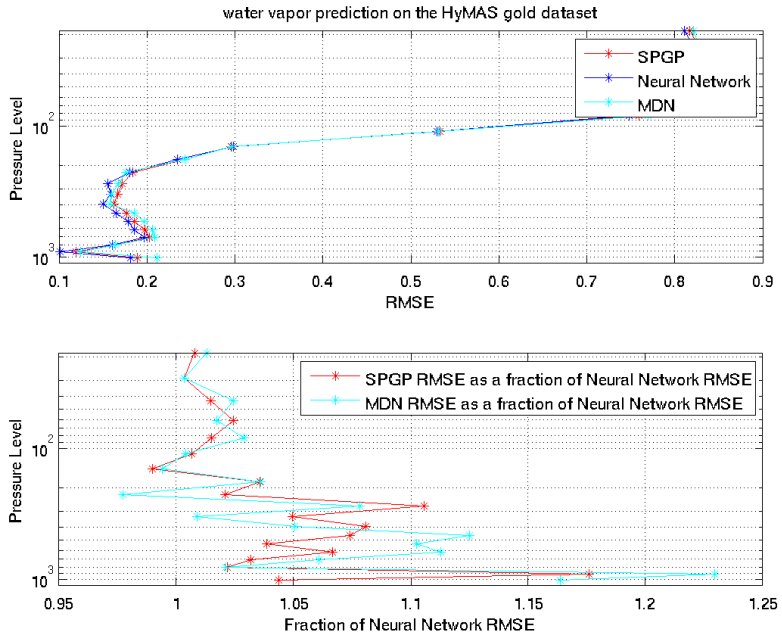


Figure 4-34: These charts compare the performance of an SPGP estimating water vapor to a neural network on the HyMAS golden days test dataset (see text for dataset details). The y-axis represents the pressure level in millibars (surface is at the bottom). RMSE is in normalized mass mixing ratio.

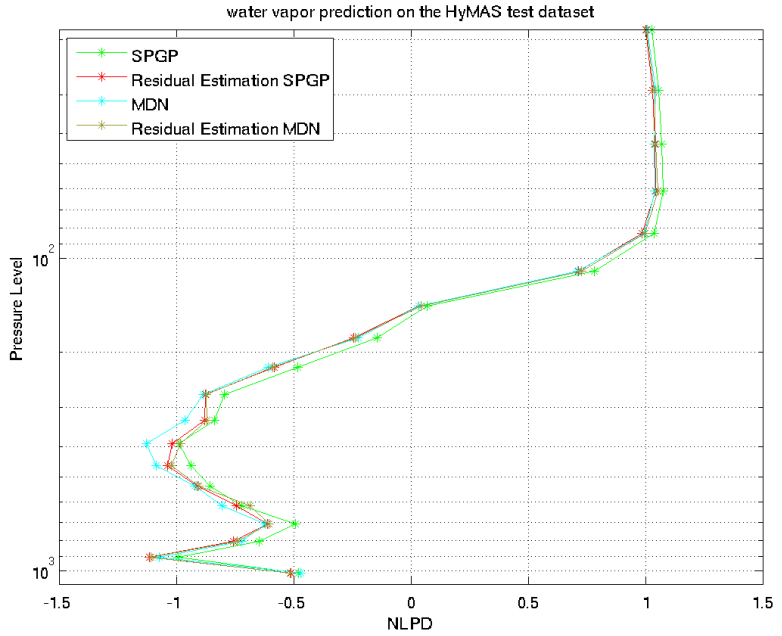


Figure 4-35: This figure shows the NLPD profile of MDNs and SPGPs on the problem of water vapor estimation. RMSE is in normalized mass mixing ratio, and pressure is in millibars.

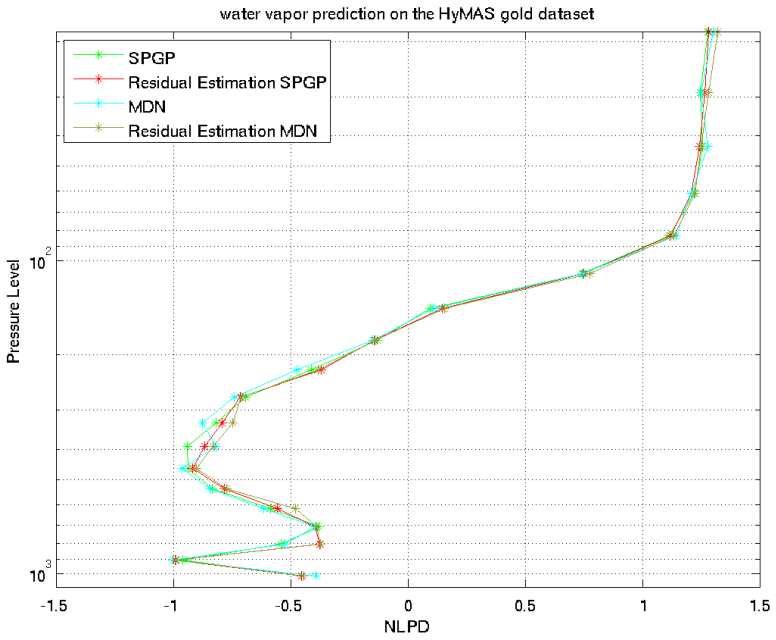


Figure 4-36: This figure shows the NLPD profile of MDNs and SPGPs on the problem of water vapor estimation on the golden days set. RMSE is in normalized mass mixing ratio, and pressure is in millibars.

vapor variances being harder to estimate than the variance of temperature, since the variance estimation neural network does not predict the variances well even on some levels of the test set, such as pressure level 535 mb (see figure 4-37). On that example, the variance estimation neural network (blue), shows little skill in separating out the easiest 60 percent of cases, predicting a similar variance for those cases. The variance estimation neural network's performance is similar on the same level on the golden days test set. (see figure 4-38).

Looking at the actual variance predictions of the neural network, we see that the predicted variance is in fact negative on more than 50 percent of the cases. Since the target residuals are never negative (so the optimal prediction should always be non-negative), this suggests that the variance function predicted by the neural network was complicated and unstable, so that when extrapolating, the variance prediction became negative. Even on the training set, there are many negative variance predictions (since noise was added at every iteration during neural network training, the "training" cases being tested on are not the exact same cases that were used during

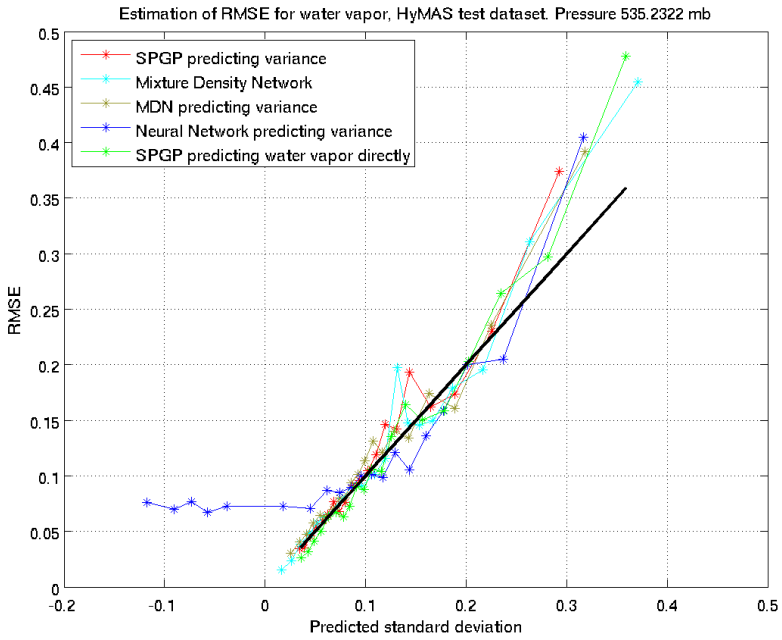


Figure 4-37: This figure compares various methods for estimating variance on the HyMAS test dataset, with respect to water vapor at pressure level 535 mb. The predicted standard deviation as well as the RMSE is in normalized mass mixing ratio.

training).

By contrast, looking at levels where the neural network predicted water vapor variance successfully, we see that the actual variance predicted at those levels was usually positive, suggesting a much more accurate and plausible function.

Still, both the residual estimation MDN and the residual estimation SPGP do not suffer the same problem as the variance estimation neural network, even though they also estimate the residuals of the first neural network. It is not simply overfitting, since the training set performance is close to the test set performance, and because reducing the number of model parameters does not eliminate the problem (see figure 4-40: even with only five hidden nodes, this problem still happens, suggesting that the original 10 hidden node variance estimation neural network was not overfitting).

Instead, it seems that the performance metric being used leads to the variance estimation neural network being prone to undesirable local minima. Although this problem of local minima must occur in all the other datasets as well, it is especially

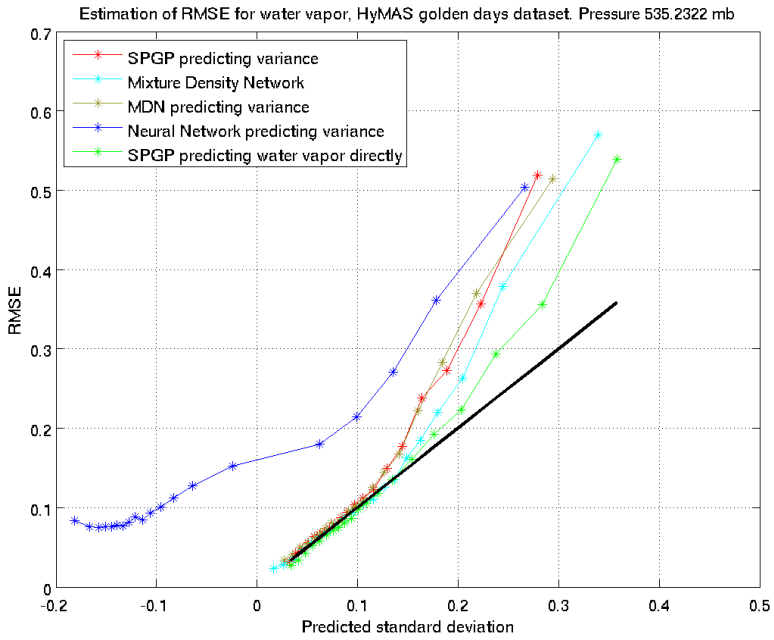


Figure 4-38: This figure compares various methods for estimating variance on the HyMAS "golden days" dataset, with respect to water vapor at pressure level 535 mb. Compare to figure 4-37. The predicted standard deviation as well as the RMSE is in normalized mass mixing ratio.

evident on this particular dataset. To get a sense of how unstable the RMSE metric is, note that the RMSE (on the training set) of the variance estimation neural network at pressure level 535 mb is 0.0952, but the *a priori* standard deviation of the square of the residuals is already only 0.1081. Therefore, predicting a constant variance will be already close to optimal, at least if judged by RMSE. Moreover, if we use the variance predictions of the residual estimation MDN (which is clearly more accurate at that pressure level) as a prediction for the neural network residuals, the RMSE obtained is 0.1008, which is actually higher than the variance estimation neural network. The MDN does not predict the mean squared error of some high residual cases as accurately as the variance estimation neural network, leading to the higher overall RMSE.

Even more telling, examine the variance estimation neural network's performance across multiple trials. Figure 4-41 shows the results of 5 trials of training a variance estimation neural network on that pressure level, 535 mb. Clearly, one trial does

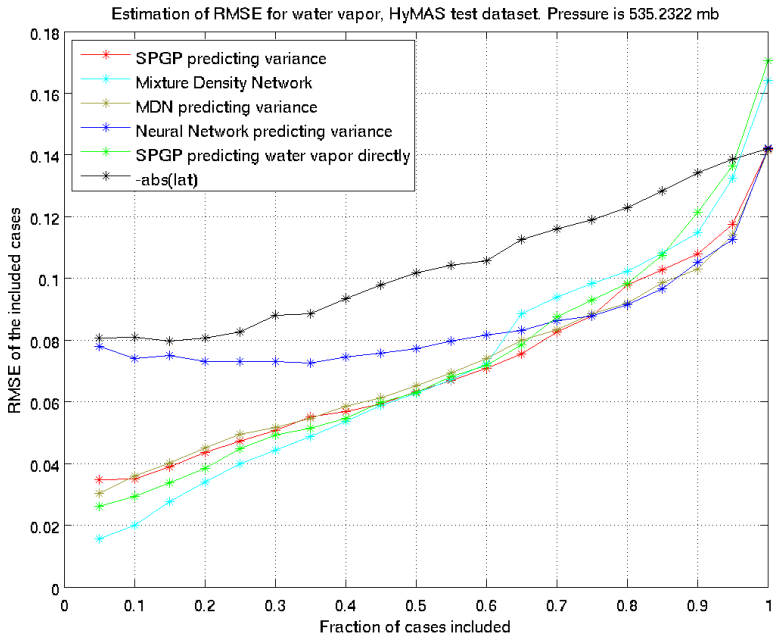


Figure 4-39: This figure compares various methods for estimating variance on the HyMAS test dataset, with respect to water vapor. This is the same problem as depicted in figure 4-37, except the y-axis here is cumulative RMSE in units of normalized mass mixing ratio.

much better at estimating variance than the others. However, the RMSE for that trial is 0.0848, whereas the RMSE for the trial that seemingly performs the worst at variance estimation is actually lower, at 0.0843.

Thus, the RMSE metric is only weakly correlated to the quality of the variance prediction. It is true that, given enough training data, the lowest RMSE should be achieved when the predicted variance is equal to the actual variance of the cases. However, this global minima is unlikely to be achieved, and a local minima that has a lower RMSE could actually be worse at variance prediction than a local minima with a higher RMSE. A potential fix could be to de-emphasize the higher residual cases by changing the targets from the square of the residuals to some other function of the residuals, such as the natural log of the absolute value of the residuals. Unfortunately, then the "variance estimation" neural network would not be predicting the variance.

Another possible objection is that the small residuals (due to the normalization of the dataset) encountered in water vapor estimation may throw off the neural network

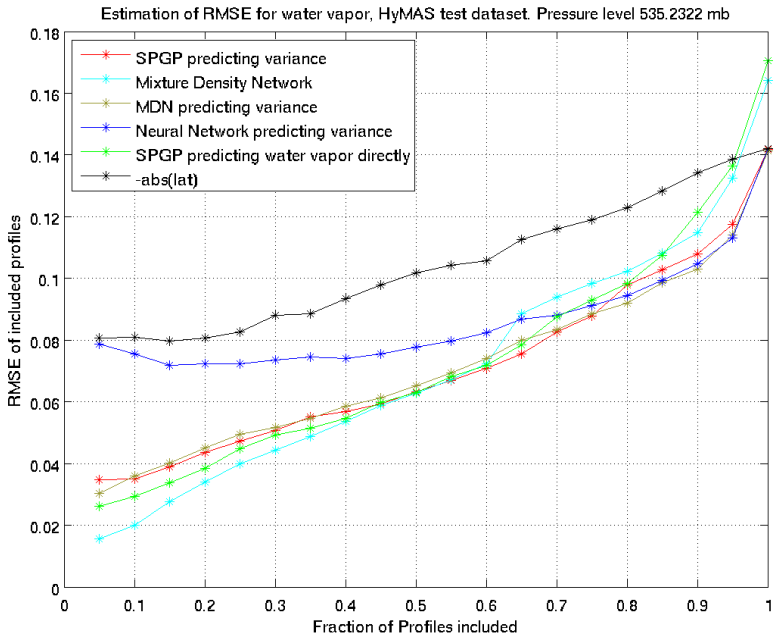


Figure 4-40: The same figure as figure 4-39, except here the variance estimation neural network is using only 5 hidden nodes, as opposed to 10 before.

training somehow by trapping it in undesirable local minima. After all, it is well known that scaling input and output data can lead to different performance. However, it is usually the case in literature that the data is normalized first, as was done here [6] [5]. Moreover, in an experiment where I normalized temperature also, I noticed no consistent difference in performance one way or the other, suggesting that normalization is not the major factor here (see figures 4-42 and 4-43).

4.3.7 Precipitation results

Rain rate retrieval is a nonlinear and fairly hard problem. Competitive algorithms often have multiple stages, based on factors such as terrain type, latitude, and temperature radiances of specific channels [23]. Because I was primarily interested in illustrating the differences between the methods for estimating variance, rather than accurate rain rate retrieval, I did not use any pre or post-processing except for a principal components transform of the inputs. Thus, the performance of the following retrievals can certainly be much improved.

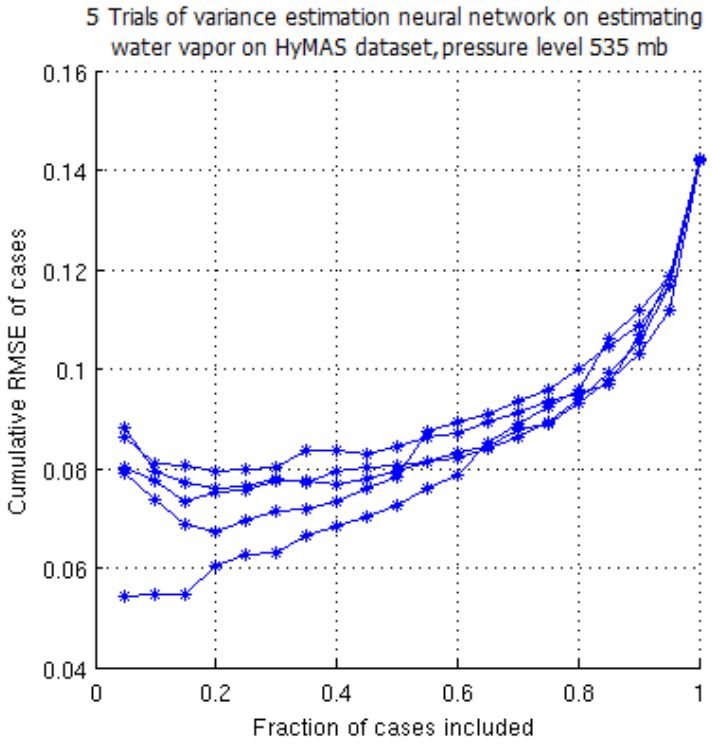


Figure 4-41: This shows the results of 5 trials of training a variance estimation neural network on estimating water vapor on pressure level 535 mb on the HyMAS dataset. RMSE is in units of normalized mass mixing ratio.

The *a priori* standard deviation of the rain rate on the test set was 2.884. However, be aware that the distribution is heavily skewed, with nearly 85 percent of the test cases having a precipitation rate of less than 1 mm/hour, and 28 percent cases having no precipitation at all. See figure 4-44 for a histogram.

Precipitation retrieval performance

The performance of the various methods is shown in table 4.1. Although neural networks perform the best at estimating rainrate, for a majority of cases (those with minimal precipitation) it is actually the MDN that is superior, followed by the SPGP. However, the neural networks have much better performance at estimating the cases with the highest precipitation, followed by the SPGP and then the MDN. Once again, it seems that both the SPGP and the MDN obtains a higher-bias solution than

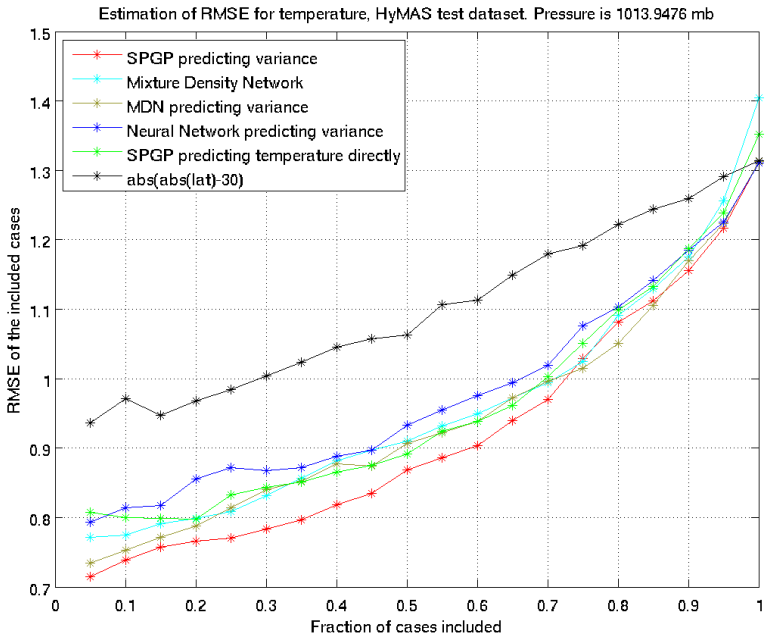


Figure 4-42: This shows the performance of variance estimation neural network (blue) at estimating variance when the temperature is not normalized (the default that I used for all the other figures). See figure 4-43 for the performance when temperature *was* normalized. The RMSE is in degrees kelvin.

MM5 Range (mm/hr)	Number of Cases	Neural Network RMSE	MDN RMSE	SPGP RMSE
[0 0.125)	2875	0.4025	0.0392	0.2914
[0.125 0.25)	212	0.9885	0.1247	1.102
[0.25 0.5)	226	2.004	0.2931	1.165
[0.5 1)	301	1.192	0.6561	1.288
[1 2)	279	1.792	1.377	1.771
[2 4)	195	2.666	2.682	2.746
[4 8)	100	5.037	5.632	4.218
[8 16)	40	5.964	11.99	6.907
[16 32)	17	12.85	22.58	15.64
[32 75)	7	25.29	49.72	37.06
All	4252	**1.911**	**2.87**	**2.229**

Table 4.1: Precipitation retrieval performance of neural networks, MDNs, and SPGPs.

the neural net. The MDN prediction is especially flat–the mean of the *predicted* precipitation is only 0.04, compared to 0.78 for neural networks.

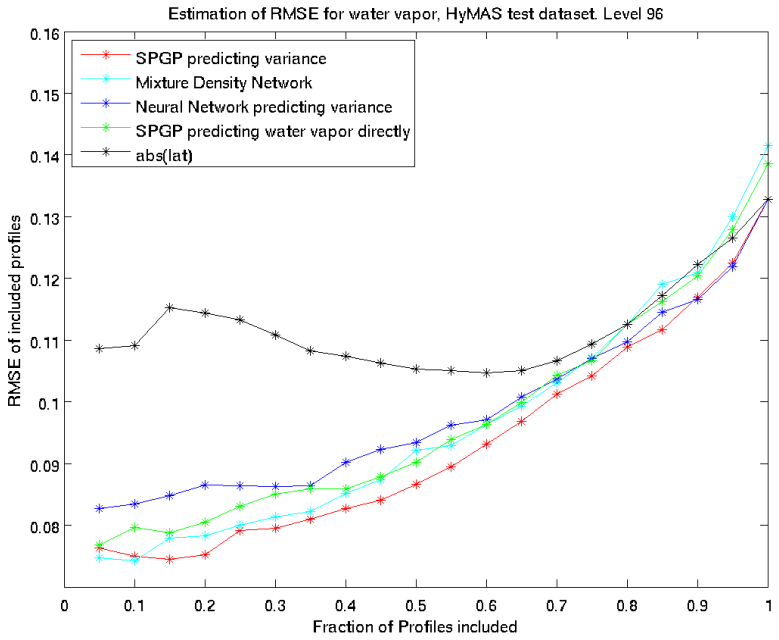


Figure 4-43: This shows the performance of variance estimation neural network (blue) at estimating variance when the temperature *was* normalized. Compared to figure 4-42, there is minimal difference in the performance of the variance estimation neural network. RMSE is in degrees kelvin.

Variance estimation performance

Figure 4-45 shows the variance estimation performance of the various methods on the test set, while figure 4-46 shows the performance on the training set. Both SPGP and MDN are effective at identifying cases with the lowest variance (which quite often also happen to the be cases with the lowest precipitation), much more so than the neural network. This again seems to be a failure of the RMSE metric, discussed under the HyMAS water vapor results. The variance estimation neural network is better at estimating the residuals of a few cases with very high residuals, while being poor at estimating the variance of cases with low residuals simply because the latter do not contribute as much to the RMSE.

Another trend is that the variance predicted by SPGP happens to be more accurate than the MDN at higher values of the predicted variance, possibly because the MDN cannot account for model uncertainty. There are a few cases with very high precipitation that the MDN did not estimate well (thus leading to those cases

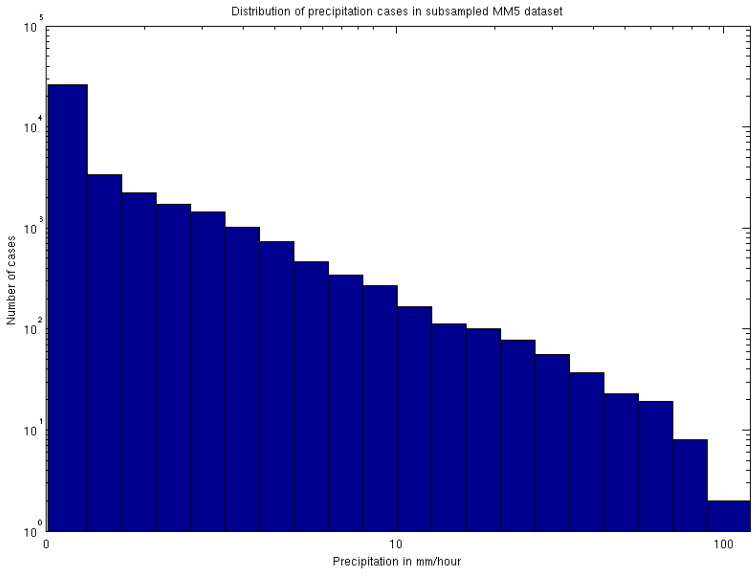


Figure 4-44: This figure shows the distribution of precipitation rates in the entire dataset. The distribution is heavily skewed, with most cases having between 0 and 1 mm/hour of precipitation. Note the logarithmic scale on both the x and the y axes.

having large residuals). Those training cases would then be assigned high variance by the MDN. Unlike SPGP, which would account for the fact that there are only a few training cases with such high residuals (thus relying more on the prior than the data), the MDN would simply predict that any future cases with inputs close to those outliers would have high variance. Indeed, the variance predicted by MDN can be quite high for certain cases, reaching 20000, while the variance predicted by SPGP is more reasonable, topping out at 70 or so.

4.4 Discussion

4.4.1 Accuracy performance

Overall, it is clear that both MDNs and SPGPs are superior to the baseline "variance estimation neural network" for confidence estimation. Not only do MDNs and SPGPs demonstrate superior performance in predicting the difficulty of the cases, but the variances that are predicted are usually very close to the actual variances. This is a

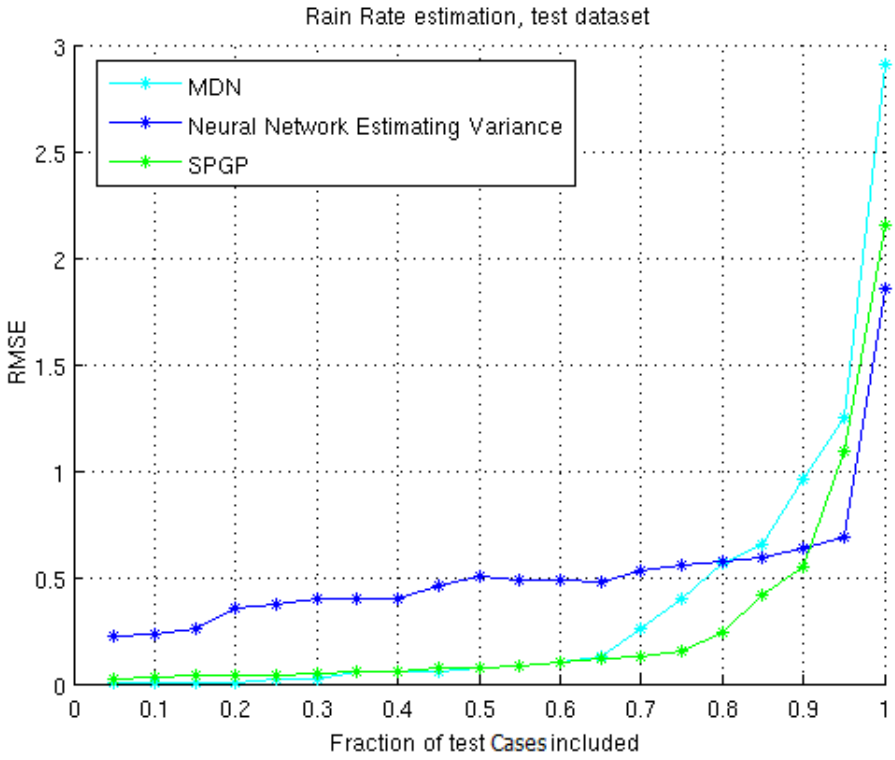


Figure 4-45: Variance estimation performance on the test dataset of various methods on the problem of precipitation retrieval. RMSE is in mm/hour.

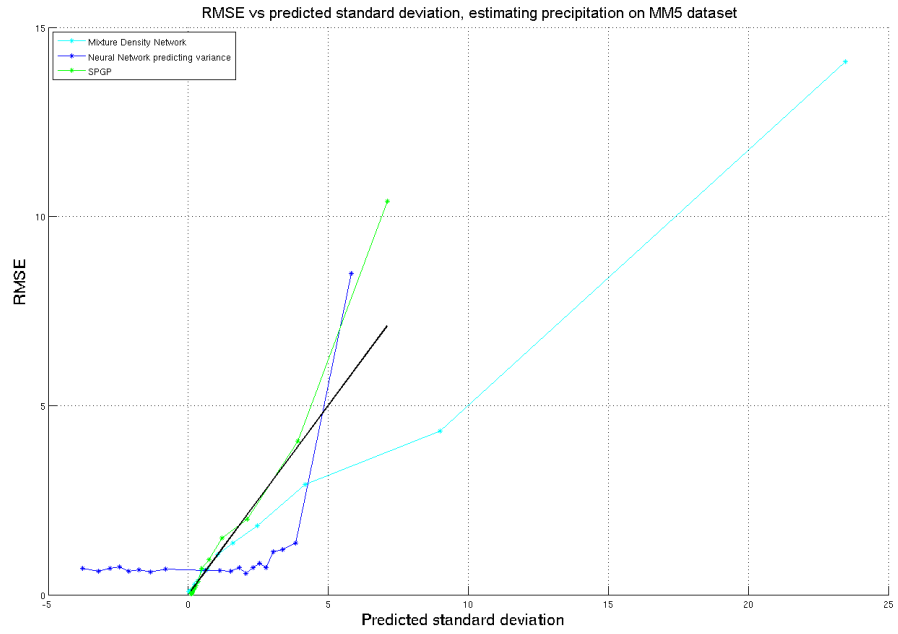


Figure 4-46: Variance estimation performance on the training dataset of various methods on the problem of precipitation retrieval. Compared to figure 4-45, this figure shows the RMSE by bin instead of the cumulative RMSE. RMSE is still in mm/hour.

major weakness of using a neural network to predict variance, since that approach consistently underpredicts the variance.

The variance estimation neural network is also likely to overfit, at least if we are judging it based on its variance estimation performance. This necessitates caution when choosing model parameters.The problem is likely that the metric used to optimize the variance estimation neural network (the RMSE, with the targets being the square of the residuals), only approaches the actual variance as the number of cases increase to infinity. However, it seems that for many problems, the number of available cases is not enough to lead to stable estimates, as evidenced by the problems of the neural network trying to predict variance on the HyMAS "golden days" datasets.

Moreover, with the current metric a variance estimation neural network is prone to getting stuck in local minima, since a large improvement in variance estimation accuracy leads to only small improvements in the metric being optimized. By contrast, the MDN, which optimizes maximum likelihood instead of RMSE, offers much better performance at fitting the variance, even though the network structure is identical to that of the variance estimation neural network. Related to this problem of suboptimal local minima, the variance estimation neural network performance is often inconsistent across multiple trials, whereas both SPGP and MDN were consistently good at variance estimation across multiple trials on the same problems.

In many the remote sensing datasets that were tested, MDNs were roughly comparable to SPGPs in terms of parameter prediction performance (RMSE) and variance prediction performance. Given that the training time and the prediction time of MDNs is shorter than that of SPGP for the hyperparameters chosen, MDNs are a good choice for confidence estimation. However, the extrapolation performance of MDNs remains uncertain and not guaranteed to be accurate, unlike SPGPs, so it is wise to make sure that the training data is complete and thorough. MDNs do not give any indication that the test vector is completely outside the training set (unlike SPGP, which will predict extremely high variance). The precipitation retrieval problem hints at the possible problems of MDNs if such conditions are not met.

Finally, residual estimation MDNs and SPGPs are certainly still useful if another

method (like neural networks) does a much better job at parameter estimation. However, in many of the problems, the weaknesses are also evident. Oftentimes, the neural network whose residuals are being estimated overfits slightly, and the residual estimation methods have no way of detecting this. This impacts the performance negatively, and it is often the case that the residual estimation methods are not as good as "pure" SPGPs and MDNs at predicting variance.

4.4.2 Speed performance

All the methods so far discussed (neural networks, Bayesian neural networks, SPGP, MDNs) scale linearly in training and testing time with the number of training and test cases. If the hyperparameters chosen (the number of hidden nodes and layers for the networks, and the reduced dimensions and number of pseudo-inputs for the SPGP) are assumed to be fixed for now, the only difference between the methods is a constant factor. Table 4.2 summarizes many of the methods' salient features, although many of the later columns in the table relating to variance estimation performance are necessarily problem dependent.

Method	Training Time	Testing Time	Accounts for Heteroscedasticity	Accounts for Model uncertainty	Accurate variance estimates	Correctly determines difficulty of retrieval	Accuracy of Parameter prediction
Variance Estimation Neural Network	1 (baseline, roughly 10 minutes)	1 (baseline, roughly 0.02 seconds)	Yes	No	No	Yes	Best
Mixture Density Networks	0.5	1	Yes	No	Yes	Yes	Good
SPGP	3	1.50 (cached matrix), 30 (recalculate matrix)	Yes	Yes	Yes	Yes	Good
Bayesian Neural Network	1	10 (Hessian precalculated), 3500 (no Hessian)	No	Yes	No	Sometimes	Best

Table 4.2: The testing and training times in this table was derived from application of the methods on the precipitation dataset. Note that the testing time and training time scaling for all methods should be linear in the number of training data and testing data. The last three columns are problem dependent and are necessarily subjective personal judgments on the effectiveness of the methods compared.

Chapter 5

Conclusion

There has been strong interest in statistical retrieval methods, such as neural networks, in the field of remote sensing, due to the advantages such methods have over physics-based inversions in both speed and accuracy. However, for statistical retrievals, the problem of assigning confidence intervals to the retrievals has so far not yet been thoroughly explored, despite its importance in creating more useful retrievals and in gaining greater acceptance for the statistical retrievals.

In this thesis, several variance estimation methods were presented and analyzed on a variety of representative datasets. Bayesian neural networks, first applied by Aires on remote sensing problems, were discovered to be lacking in variance estimation performance due to inability to model heteroscedasticity. Two variance estimation methods that have not yet been used for geophysical parameter retrieval, mixture density networks (MDN) and sparse pseudo-input Gaussian processes (SPGP), were found to be much more accurate at predicting the variance on all the datasets tested. They were also more robust, compared to variance estimation neural networks, when confronted with test data that had features not present in the training set.

MDNs had similar speed in both training and testing time to standard neural networks. SPGPs had about three times longer training time with the hyperparameters used, but had similar testing time to that of a neural network. Overall, MDNs are the best choice for variance estimation if the training time is important and if there is confidence that the training dataset is comprehensive.

However, if training time is not an important factor, SPGPs are theoretically better able to account for the impact of lack of training data in its variance prediction. This advantage over MDNs makes SPGPs the more robust choice.

Both of these methods can also easily be used to just predict the variance, and allow a neural network to estimate the geophysical parameters, if so desired. The variance estimation performance of these residual estimation MDNs and SPGPs can sometimes be slightly worse than simply applying MDNs and SPGPs directly. However, on many problems MDNs and SPGPs had worse accuracy than neural networks in estimating the geophysical parameters, making the residual estimation configuration an attractive option.

5.1 Future Work

There is certainly much more work that can be done. These methods should be applied to datasets that cover land cases as well. Factors lacking in ocean data ,such as the terrain type, may have a large impact on the noise and the variance. On a similar note, different instruments can be tried as well, such as the future Advanced Technology Microwave Sounder (ATMS). It would be interesting to see how these changes in the dataset affects the performance of any of the methods discussed.

Much work can also be done on optimizing the methods themselves. There was only rudimentary work done in optimizing some of the hyperparameters (number of reduced dimension for SPGP, network structure for neural nets and MDNs) via cross-validation. Although empirically modifying the hyperparameters on the methods did not lead to large changes in either RMSE or variance performance, the tradeoff between performance, training/testing time, and the hyperparameter settings could be explored more thoroughly.

Multi-task learning can also be applied by trying to model multiple pressure levels simultaneously, since it seems reasonable that the geophysical parameters (and the variances) are correlated across pressure levels that are close together. Although so far the methods have been trained only one level at a time to simplify things, multi-

task learning might speed up the training process and may even improve the accuracy of the methods.

Finally, the methods themselves can be modified to better suit the problem. Because this thesis was primarily focused on exploring different methods, few variations on any method were considered. Certainly, Gaussian process regression methods in general can be a powerful tool, with SPGPs being only one variation on them. Another metric for MDNs that does not have the multiple local minima that maximum likelihood does could finally allow MDNs with multiple Gaussian outputs (a proper Gaussian mixture model) to be used in practical applications. This could improve the characterization of the output distribution and allow for a more accurate estimate of the variance. It might even be possible to improve variance estimation neural networks by introducing the density of the input data as an additional input, in order to better account for model uncertainty.

Appendix A

Gradients

A.1 Neural Network Gradients

For the following, the activation function of the hidden layer is assumed to be sigmoidal ($f(a) = \frac{1}{1+\exp(-a)}$). Given the n output vectors $\mathbf{y}_i$ and the n output targets $\mathbf{t}_i$, where n is the number of training cases, the error function we wish to optimize is:

$$E = \sum_{i=1}^{n} (\mathbf{y}_i - \mathbf{t}_i)^T (\mathbf{y}_i - \mathbf{t}_i) \tag{A.1}$$

If we rewrite as this as the sum of n error terms E_i, one for each training case, we get:

$$E_i = \sum_{k=1}^{c} (y_i^k - t_i^k)^2 \tag{A.2}$$

Where the y_i^k are one of the c components of the output vector $\mathbf{y}_i$, and t_i^k are similarly defined.

To simplify the expression for the derivatives, define the *errors* δ for each node as:

$$\delta_k = y_i^k - t_i^k \tag{A.3}$$

for the output layer, and

$$\delta_j = f(a_j)(1 - f(a_j)) \sum_{k=1}^{c} w_{kj}\delta_k \tag{A.4}$$

for all nodes in the hidden layer. $f(a_j)$ is the sigmoidal function, and a_j is the input to that particular node.

Then, the derivatives with respect to the weights from the input layer to the hidden layer are:

$$\frac{\partial E_i}{\partial w_{js}} = \delta_j x_i^s \tag{A.5}$$

where x_i^s is the s^{th} component of the input vector $\mathbf{x}_i$.

The derivatives with respect to all other weights are:

$$\frac{\partial E_i}{\partial w_{kj}} = \delta_k f(a_j) \tag{A.6}$$

A more detailed derivation can be found in Bishop's book [5].

A.2 MDN Gradients

Because in my thesis I only used one Gaussian component in the "mixture" model, the gradients presented here are simplified from the ones in Bishop's paper [4].

The metric we optimize (negative log likelihood) is then (refer to section 4.1.2 for notation):

$$E_T = \sum_{(\mathbf{x},t)\in D} E(\mathbf{x},t) \tag{A.7}$$

$$E(\mathbf{x},t) = -\log\left(\frac{1}{\sqrt{2\pi\sigma(\mathbf{x})}}\exp\left(-\frac{||t-\mu(\mathbf{x})||^2}{2\sigma(\mathbf{x})^2}\right)\right) \tag{A.8}$$

where D is the training data consisting of (input, target) pairs $(\mathbf{x}, t)$. Since E_T is a sum of $|T|$ terms $E(\mathbf{x}, t)$ (one for each training case), we only need the derivatives of E. So the derivatives with respect to the outputs z will be $\frac{\partial E}{\partial z^\mu}$ and $\frac{\partial E}{\partial z^\sigma}$.

$$\frac{\partial E}{\partial z^\sigma} = \frac{\partial E}{\partial \sigma}\frac{\partial \sigma}{\partial z^\sigma} = \left(-\frac{||t-\mu||^2}{\sigma^2} - 1\right) \tag{A.9}$$

$$\frac{\partial E}{\partial z^\mu} = \frac{\partial E}{\partial \mu} = \left(-\frac{(\mu - t)}{\sigma^2}\right) \tag{A.10}$$

Combined with the standard neural network derivatives of the outputs with respect to the network weights, we can then use gradient descent algorithms to maximize the likelihood with respect to the network weights.

A.3 SPGP Gradients

For the notation, refer to section 4.2.2. Recall that m are the pseudo-input vectors, and n are the training cases.

For simplicity later on, define

$$\sigma^2\Gamma = \text{diag}(K_n - Q_n) + \sigma^2\mathbf{I} \tag{A.11}$$

Note that Γ is a symmetric matrix.

The negative log likelihood is

$$L = -\log N(\mathbf{0}, Q_n + \sigma^2\Gamma) = \frac{1}{2}\left(\log|Q_n + \sigma^2\Gamma| + \mathbf{t}^T(Q_n + \sigma^2\Gamma)^{-1}\mathbf{t} + n\log 2\pi\right) \tag{A.12}$$

where n is the number of training cases.

Separate L into two terms L_1 and L_2:

$$L = L_1 + L_2 + n\log 2\pi \tag{A.13}$$

$$L_1 = \frac{1}{2}(\log|Q_n + \sigma^2\Gamma|) \tag{A.14}$$

$$L_2 = \frac{1}{2}\mathbf{t}^T(Q_n + \sigma^2\Gamma)^{-1}\mathbf{t} \tag{A.15}$$

Define matrix A as

$$A = \sigma^2 K_m + K_{mn}\Gamma^{-1}K_{nm} \tag{A.16}$$

A.3.1 Derivatives of Hyperparameters in the Kernel

The details of the derivation can be found in Snelson's thesis [21], but the derivative of L_1 and L_2 with respect to a hyperparameter θ is:

$$\frac{\partial L_1}{\partial \theta} = \mathrm{tr}\left(A^{-\frac{1}{2}}\frac{\partial A}{\partial \theta}{A^{-\frac{1}{2}}}^T\right) - \mathrm{tr}\left({K_m}^{-\frac{1}{2}}\frac{\partial K_m}{\partial \theta}{K_m}^{-\frac{1}{2}T}\right) + \mathrm{tr}\left(\Gamma^{-\frac{1}{2}}\frac{\partial \Gamma}{\partial \theta}\Gamma^{-\frac{1}{2}}\right) \quad \text{(A.17)}$$

and

$$\frac{\partial L_2}{\partial \theta} = \frac{1}{\sigma^2}\Bigg[-\frac{1}{2}\mathbf{t}^T\Gamma^{-\frac{1}{2}}\frac{\partial \Gamma}{\partial \theta}\Gamma^{-\frac{1}{2}}\mathbf{t} + (A^{-\frac{1}{2}}K_{mn}\Gamma^{-1}\mathbf{t})^T\Bigg(\frac{1}{2}A^{-\frac{1}{2}}\frac{\partial A}{\partial \theta}{A^{-\frac{1}{2}}}^T(A^{-\frac{1}{2}}K_{mn}\Gamma^{-1}\mathbf{t}) - A^{-\frac{1}{2}}\frac{\partial K_{mn}}{\partial \theta}\Gamma^{-1}\mathbf{t} + A^{-\frac{1}{2}}K_{mn}\Gamma^{-\frac{1}{2}}\frac{\partial \Gamma}{\partial \theta}\Gamma^{-\frac{1}{2}}\mathbf{t}\Bigg)\Bigg] \quad \text{(A.18)}$$

The partial derivatives $\frac{\partial A}{\partial \theta}$ and $\frac{\partial \Gamma}{\partial \theta}$ are defined as:

$$\frac{\partial A}{\partial \theta} = \sigma^2\frac{\partial K_m}{\partial \theta} + 2\,\mathrm{sym}(\frac{\partial K_{mn}}{\partial \theta}\Gamma^{-1}K_{nm}) - K_{mn}\Gamma^{-1}\frac{\partial \Gamma}{\partial \theta}\Gamma^{-1}K_{nm} \quad \text{(A.19)}$$

where the function sym is to make the matrix symmetric, so that $\mathrm{sym}(B) = \frac{B+B^T}{2}$. Finally,

$$\frac{\partial \Gamma}{\partial \theta} = \sigma^{-2}\mathrm{diag}(\frac{\partial K_n}{\partial \theta} - 2\frac{\partial K_{nm}}{\partial \theta}K_m^{-1}K_m n + K_{nm}K_m^{-1}\frac{\partial K_m}{\partial \theta}K_m^{-1}K_{mn}) \quad \text{(A.20)}$$

A.3.2 Noise Derivative

The noise term σ^2 is not present in the kernel, unlike all the other hyperparameters. Thus, the above derivation is not valid since σ^2 was treated as a constant there. Instead the partial derivatives $\frac{\partial L_1}{\partial \sigma^2}$ and $\frac{\partial L_2}{\partial \sigma^2}$ are (again, refer to Snelson's thesis [21] for the full derivation):

$$\frac{\partial L_1}{\partial \sigma^2} = \mathrm{tr}(Q_n + \sigma^2\Gamma)^{-1} \quad \text{(A.21)}$$

$$\frac{\partial L_2}{\partial \sigma^2} = -||(Q_n + \sigma^2\Gamma)^{-1}\mathbf{t}||^2 \quad \text{(A.22)}$$

Appendix B

Matlab Code

This is an incomplete listing of the code that I used, but it covers all of the major functions. I also had several scripts that called the below functions and applied them to different datasets, but those scripts are omitted.

B.1 Neural Networks

The following code depends on Ian Nabney's netlab toolbox, available for download from `http://www1.aston.ac.uk/eas/research/groups/ncrg/resources/netlab/downloads/`

```
function [nnet, nnet_performance, levels_1, levels_2] = nn_retrieval_simple(↩
    pcs_train, pcs_test, pcs_val, prof_train_mr, prof_test_mr, prof_val_mr, ↩
    Num_nodes, Num_trials, noise_matrix, levels_1, levels_2)

% Neural network (single hidden layer) retrieval
%
% This function requires the NETLAB toolbox for MATLAB available for free:
%    http://www.ncrg.aston.ac.uk/netlab/index.php
%    maintained by Ian Nabney (i.t.nabney@aston.ac.uk)
%    For more information, consult Dr. Nabney's textbook:
%    Netlab: Algorithms for Pattern Recognition, ISBN: 1-85233-440-1
%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%% NN Section %%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


if ~exist('levels_1') || ~exist('levels_2')
levels_1 = 1:size(prof_train_mr,1);
levels_2 = 1:size(prof_train_mr,1);
end
%%%%%%%%%%%%%%%%%%%%%% NN Initialization %%%%%%%%%%%%%%%%%%%%%%
MAX_ITER = 200;     % maximum number of training epochs
MAX_DUDS = 10;      % number of consecutive epochs that do not
                    %   reduce the validation error
NUM_VERR_AVG = 100; % number of random noise realizations

% NETLAB options,
options = foptions;
options(1) = 0;        % Set to 1 for verbose display
options(14) = 1;       % maximum number of function evaluations
options(18) = 0.001;  % mu_init
options(19) = 10;      % mu_inc
options(20) = 0.1;     % mu_dec
options(21) = 1e10;  % mu_max
for j = 1:length(levels_1) % loop over profile "level chunks"
   % Initialize the global validation error minimum
   validation_error_best_global = inf;
   output_range = levels_1(j):levels_2(j);
   for trial_num = 1:Num_trials  % loop over trials

      NUM_OUTPUTS = length(output_range);
      fprintf('--- Preparing neural network...\n');
      % Define skeleton network with a single hidden layer
      nnet_ = mlp(size(pcs_train,1),Num_nodes,NUM_OUTPUTS,'linear');
      % Initialize weights and biases using Nguyen-Widrow method
      nnet_ = mlpinit_nw(nnet_, [min(pcs_train,[],2) max(pcs_train,[],2)]);

      % validation error for this particular trial
      validation_error_best = inf;
      clear training_error validation_error testing_error
      keep_looping = 1;
      num_duds = 0;
      options(18) = 0.001;  % Reset mu
      i = 1;

      while (keep_looping)
         fprintf('*** Iteration %d of %d for chunk %d of %d (%d outputs, trial %d ↩
```

```
        of %d).  \n', i, MAX_ITER, j, length(levels_1), NUM_OUTPUTS, trial_num↩
        , Num_trials);
    pcs_train_noisy = pcs_train + noise_matrix * randn(size(pcs_train));

    % Train the NN for one epoch using Levenberg-Marquardt learning
    % method
    [nnet_, options] = netopt(nnet_, options, pcs_train_noisy', prof_train_mr(↩
        output_range,:)', 'lm');

    % Evaluate performance
    validation_error(i) = 0;
    training_error(i) = 0;
    testing_error(i) = 0;
    for k=1:NUM_VERR_AVG
       validation_error(i) = validation_error(i) + mean(mean((prof_val_mr(↩
           output_range,:) - mlpfwd(nnet_, (pcs_val+ noise_matrix * randn(size↩
           (pcs_val)))')').^2));
       training_error(i) = training_error(i) + mean(mean((prof_train_mr(↩
           output_range,:) - mlpfwd(nnet_, (pcs_train+ noise_matrix * randn(↩
           size(pcs_train)))')').^2));
       testing_error(i) = testing_error(i) + mean(mean((prof_test_mr(↩
           output_range,:) - mlpfwd(nnet_, (pcs_test+ noise_matrix * randn(↩
           size(pcs_test)))')').^2));
    end
    validation_error(i) = validation_error(i)/NUM_VERR_AVG;
    training_error(i) = training_error(i)/NUM_VERR_AVG;
    testing_error(i) = testing_error(i)/NUM_VERR_AVG;

    % Check if error has decreased
    if (validation_error(i) < validation_error_best)
       num_duds = 0;
       nnet_best = nnet_;
       validation_error_best = validation_error(i);
       fprintf('--- NEW minimum found!  (Training error = %g, Validation Error↩
            = %g, Testing Error = %g)\n', training_error(i), validation_error(↩
           i), testing_error(i));
    else
       num_duds = num_duds + 1;
    end

    % Has error failed to decrease for MAX_DUDS consecutive times?
    if (num_duds==MAX_DUDS | i==MAX_ITER)
       keep_looping = 0;
       nnet_performance{j}.training_error{trial_num} = training_error;
       nnet_performance{j}.validation_error{trial_num} = validation_error;
```

```
            nnet_performance{j}.testing_error{trial_num} = testing_error;
         else
            i=i+1;
         end

         if validation_error_best < min(validation_error_best_global)
            nnet_best_global = nnet_best;
            fprintf('+++ NEW global minimum found! +++\n');
         end
         validation_error_best_global = [validation_error_best_global ↩
             validation_error_best];
      end
   end
      nnet{j} = nnet_best_global;
   nnet_perfomance{j}.test_residual(output_range,:) = (prof_test_mr(output_range,:)↩
       - mlpfwd(nnet{j},pcs_test'))';
    nnet_perfomance{j}.test_error(output_range) = sqrt(sum((prof_test_mr(↩
        output_range,:) - mlpfwd(nnet{j},pcs_test'))'.^2)/size(prof_test_mr,2));
   nnet_performance{j}.val = validation_error_best_global;
end
```

```
function [nnet, nnet_performance, pcs_train, pcs_test, pcs_val, prof_train_mr, ↩
    prof_test_mr, prof_val_mr] = nn_retrieval_var(tbs, profiles, noise_cov, ↩
    Num_ppcs, Num_nodes, Num_trials, levels_1, levels_2, TRAINING_PROFILES, ↩
    TESTING_PROFILES, VALIDATION_PROFILES, normalize_output)

% Neural network (single hidden layer) retrieval
%
%
% This function requires the NETLAB toolbox for MATLAB available for free:
%    http://www.ncrg.aston.ac.uk/netlab/index.php
%    maintained by Ian Nabney (i.t.nabney@aston.ac.uk)
%    For more information, consult Dr. Nabney's textbook:
%    Netlab: Algorithms for Pattern Recognition, ISBN: 1-85233-440-1
%



NUM_PROFILES = size(profiles,2);
NUM_LEVELS = size(profiles,1);

% Set aside 10 percent of ensemble for validation profiles
NUM_PROFILES = size(profiles,2);
```

```
if nargin < 9
    TESTING_PROFILES = 1:10:NUM_PROFILES;
    VALIDATION_PROFILES = 5:10:NUM_PROFILES;
    TRAINING_PROFILES = 1:NUM_PROFILES;
    TRAINING_PROFILES([VALIDATION_PROFILES TESTING_PROFILES]) = [];
end

% Check to see if we're retrieving water vapor or temperature
if min(min(profiles))<100 && normalize_output
    fprintf('Water vapor detected.  Normalizing...\n');
    TEMPERATURE=0;
else
    TEMPERATURE=1;
end

prof_train = profiles(:, TRAINING_PROFILES);
prof_test = profiles(:, TESTING_PROFILES);
prof_val = profiles(:, VALIDATION_PROFILES);
clear profiles

rad_train = tbs(:, TRAINING_PROFILES);
rad_test = tbs(:, TESTING_PROFILES);
rad_val = tbs(:, VALIDATION_PROFILES);
clear tbs


if nargin < 12
    normalize_output = 1;
end

if normalize_output
    mean_prof = mean(prof_train')';
else
    mean_prof = zeros(size(prof_train,1),1);
end
mean_rad = mean(rad_train')';
rad_train_mr = rad_train - mean_rad * ones(1, length(rad_train));
rad_test_mr = rad_test - mean_rad * ones(1, size(rad_test,2));
rad_val_mr = rad_val - mean_rad * ones(1, size(rad_val,2));
prof_train_mr = prof_train - mean_prof * ones(1, size(prof_train,2));
prof_test_mr = prof_test - mean_prof * ones(1, size(prof_test,2));
prof_val_mr = prof_val - mean_prof * ones(1, size(prof_val,2));

if TEMPERATURE == 1 | ~normalize_output  % Normalize water vapor profile
   std_norm_factor = ones(size(std(prof_train')'));
```

```
else
   std_norm_factor = std(prof_train')';
end

prof_train_mr = diag(1./std_norm_factor) * prof_train_mr;
prof_test_mr = diag(1./std_norm_factor) * prof_test_mr;
prof_val_mr = diag(1./std_norm_factor) * prof_val_mr;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%% PPC Section %%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Crr = rad_train_mr * rad_train_mr' / (length(rad_train)-1);
Cpr = prof_train_mr * rad_train_mr' / (length(rad_train)-1);
Cnn = noise_cov;

if Num_ppcs > 0

    [ppc_evects, ppc_evals] = eigs(Crr, Num_ppcs);
    ppc_evects = real(ppc_evects);  % small imaginary values possible
    clear Crr
    V = ppc_evects(:, 1:Num_ppcs);


    % The following adjustment is needed to make sure that the V's are identical
    % every time. eigs/svd DO NOT return the same answer for successive
    % calls - each column can differ by a scale factor of -1.
    % I'm going to ensure that the first element of each column is always
    % positive so the results will always be consistent.

    scale_factors = ones(size(V(1,:)));
    scale_factors(find(V(1,:)<0)) = -1;
    V = V .* (ones(size(V(:,1))) * scale_factors);
else
    Num_ppcs = size(rad_train,1);
    V = eye(Num_ppcs);
end


pcs_train = V' * rad_train_mr;
clear rad_train
s_pcs_train = std(pcs_train')';
pcs_train = diag(1./s_pcs_train) * pcs_train;

Snn = sqrtm(diag(1./s_pcs_train) * V' * Cnn * V * diag(1./s_pcs_train));
```

```
pcs_test = V' * rad_test_mr;
pcs_val = V' * rad_val_mr;
clear rad_test rad_val
pcs_test = diag(1./s_pcs_train) * pcs_test;
pcs_val = diag(1./s_pcs_train) * pcs_val;

% Initialize output estimate matrix
est_test = zeros(size(prof_test));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%% NN Section %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Assuming 100 levels plus a surface temp

% if NUM_LEVELS < 100
%   error('Check number of profile levels - should be 100 or 101\n');
% end
if nargin<10
    if (TEMPERATURE)
        levels_1 = [1:5:60];  % up to ~22.5 km
        levels_2 = [5:5:60];
    else
        levels_1 = [1:5:60]; % up to ~12.5 km, 1 is top of atmo
        levels_2 = [5:5:60];
    end
end

%%%%%%%%%%%%%%%%%%%%%%% NN Initialization %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
MAX_ITER = 200;     % maximum number of training epochs
MAX_DUDS = 10;      % number of consecutive epochs that do not
%   reduce the validation error
NUM_VERR_AVG = 100; % number of random noise realizations

% NETLAB options,
options = foptions;
options(1) = 0;       % Set to 1 for verbose display
options(14) = 1;      % maximum number of function evaluations
options(18) = 0.001;  % mu_init
options(19) = 10;     % mu_inc
options(20) = 0.1;    % mu_dec
options(21) = 1e10;;  % mu_max

for j = 1:length(levels_1) % loop over profile "level chunks"
    % Initialize the global validation error minimum
```

```
validation_error_best_global = inf;

for trial_num = 1:Num_trials  % loop over trials
    output_range = levels_1(j):levels_2(j);
    NUM_OUTPUTS = length(output_range);
    fprintf('--- Preparing neural network...\n');
    % Define skeleton network with a single hidden layer
    nnet_ = mlp(Num_ppcs,Num_nodes,NUM_OUTPUTS,'linear');
    % Initialize weights and biases using Nguyen-Widrow method
    nnet_ = mlpinit_nw(nnet_, [min(pcs_train,[],2) max(pcs_train,[],2)]);

    % validation error for this particular trial
    validation_error_best = inf;
    clear training_error validation_error testing_error
    keep_looping = 1;
    num_duds = 0;
    options(18) = 0.001;  % Reset mu
    i = 1;

    while (keep_looping)
        fprintf('*** Iteration %d of %d for chunk %d of %d (%d outputs, trial %↩
            d of %d).  \n', i, MAX_ITER, j, length(levels_1), NUM_OUTPUTS, ↩
            trial_num, Num_trials);
        pcs_train_noisy = pcs_train + Snn * randn(size(pcs_train));

        % Train the NN for one epoch using Levenberg-Marquardt learning method

        [nnet_, options] = netopt(nnet_, options, pcs_train_noisy', ↩
            prof_train_mr(output_range,:)', 'lm');

        % Evaluate performance
        validation_error(i) = 0;
        training_error(i) = 0;
        testing_error(i) = 0;
        for k=1:NUM_VERR_AVG
            validation_error(i) = validation_error(i) + mean(mean((prof_val_mr(↩
                output_range,:) - mlpfwd(nnet_, (pcs_val + Snn * randn(size(↩
                pcs_val)))')').^2));
            training_error(i) = training_error(i) + mean(mean((prof_train_mr(↩
                output_range,:) - mlpfwd(nnet_, (pcs_train + Snn * randn(size(↩
                pcs_train)))')').^2));
            testing_error(i) = testing_error(i) + mean(mean((prof_test_mr(↩
                output_range,:) - mlpfwd(nnet_, (pcs_test + Snn * randn(size(↩
                pcs_test)))')').^2));
        end
```

```
            validation_error(i) = validation_error(i)/NUM_VERR_AVG;
            training_error(i) = training_error(i)/NUM_VERR_AVG;
            testing_error(i) = testing_error(i)/NUM_VERR_AVG;

            % Check if error has decreased
            if (validation_error(i) < validation_error_best)
                num_duds = 0;
                nnet_best = nnet_;
                validation_error_best = validation_error(i);
                fprintf('--- NEW minimum found!  (Training error = %g, Validation ↩
                    Error = %g, Testing Error = %g)\n', training_error(i), ↩
                    validation_error(i), testing_error(i));
            else
                num_duds = num_duds + 1;
            end

            % Has error failed to decrease for MAX_DUDS consecutive times?
            if (num_duds==MAX_DUDS | i==MAX_ITER)
                keep_looping = 0;
                nnet_performance{j}.training_error{trial_num} = training_error;
                nnet_performance{j}.validation_error{trial_num} = validation_error;
                nnet_performance{j}.testing_error{trial_num} = testing_error;
            else
                i=i+1;
            end

            if validation_error_best < min(validation_error_best_global)
                nnet_best_global = nnet_best;
                fprintf('+++ NEW global minimum found! +++\n');
            end
            validation_error_best_global = [validation_error_best_global ↩
                validation_error_best];
        end
    end


    nnet{j} = nnet_best_global;
    nnet_performance{j}.val = validation_error_best_global;

    est_test_ = mlpfwd(nnet{j}, (pcs_test + Snn * randn(size(pcs_test)))')';

    est_test_ = diag(std_norm_factor(output_range)) * est_test_;
    nnet_performance{j}.std_norm_factor = std_norm_factor;
```

```
    est_test(output_range,:) = est_test_ + mean_prof(output_range) * ones(1, length(est_test_));

    % Include all the normalization parameters
    nnet{j}.mean_prof = mean_prof;
    nnet{j}.mean_rad = mean_rad;
    nnet{j}.s_pcs_train = s_pcs_train;
    nnet{j}.V = V;
    nnet{j}.Snn = Snn;

end
```

```
function [nnet nnet_variance nnet_train_pred nnet_test_pred nnet_val_pred nnet_var_train_pred nnet_var_test_pred nnet_var_val_pred x_train x_test x_val y_train y_test y_val] = nn_retrieval_with_variance_prediction(x, y, num_ppcs, hn, hn_var, trials, levels1, levels2, levels1_var, levels2_var, TRAINING, TESTING, VALIDATION, noise_cov, normalize)
[nnet,junk,x_train,x_test,x_val,y_train,y_test,y_val] = nn_retrieval_var(x , y, noise_cov, num_ppcs, hn, trials, levels1, levels2, TRAINING, TESTING, VALIDATION,normalize);

%Just in case you want variance estimation neural network stability tests:
%[x_train x_test x_val] = getPCS(x, nnet{1}.V, TRAINING, TESTING , VALIDATION);

[rms nnet_train_pred] = nnet_error_pred(x_train, y_train, nnet, levels1, levels2);
[rms nnet_test_pred] = nnet_error_pred(x_test, y_test, nnet, levels1, levels2);
[rms nnet_val_pred] = nnet_error_pred(x_val, y_val, nnet, levels1, levels2);

for j = 1:length(levels1)
output_range = levels1(j):levels2(j);
nnet_train_residual(output_range,:) = nnet_train_pred(output_range,:) - y_train(output_range,:);
nnet_test_residual(output_range,:) = nnet_test_pred(output_range,:) - y_test(output_range,:);
nnet_val_residual(output_range,:) = nnet_val_pred(output_range,:) - y_val(output_range,:);
end

% Normalize square of residuals
norm_factor = mean(nnet_train_residual.^2)'; %Nx1
Snn = sqrtm(diag(1./nnet{1}.s_pcs_train) * nnet{1}.V' * noise_cov * nnet{1}.V * diag(1./nnet{1}.s_pcs_train));
```

```
[nnet_variance] = nn_retrieval_simple(x_train, x_test, x_val, nnet_train_residual↩
    .^2-repmat(norm_factor,1,size(nnet_train_residual,2)), nnet_test_residual.^2-↩
    repmat(norm_factor,1,size(nnet_test_residual,2)), nnet_val_residual.^2-repmat(↩
    norm_factor,1,size(nnet_val_residual,2)), hn_var, trials, Snn, levels1_var, ↩
    levels2_var);
[rms nnet_var_train_pred] = nnet_error_pred(x_train, nnet_train_residual.^2, ↩
    nnet_variance, levels1_var, levels2_var);
[rms nnet_var_test_pred] = nnet_error_pred(x_test, nnet_test_residual.^2, ↩
    nnet_variance, levels1_var, levels2_var);
[rms nnet_var_val_pred] = nnet_error_pred(x_val, nnet_val_residual.^2, ↩
    nnet_variance, levels1_var, levels2_var);
nnet_var_train_pred = nnet_var_train_pred + repmat(norm_factor,1,size(↩
    nnet_train_residual,2));
nnet_var_test_pred = nnet_var_test_pred + repmat(norm_factor,1,size(↩
    nnet_test_residual,2));
nnet_var_val_pred = nnet_var_val_pred + repmat(norm_factor,1,size(nnet_val_residual↩
    ,2));
```

B.2 Bayesian Neural Networks

The following code depends on Ian Nabney's netlab toolbox, available for download from `http://www1.aston.ac.uk/eas/research/groups/ncrg/resources/netlab/downloads/`

```
function [nnet, nnet_performance, pcs_test, prof_test_mr, mean_prof, levels_1, ↩
    levels_2, V, s_pcs_train] = nn_retrieval_bayes(tbs, profiles, noise_cov, ↩
    Num_ppcs, Num_nodes, Num_trials, levels_1, levels_2, AW1, AB1, AW2, AB2, ↩
    BETAVAL, TRAINING_PROFILES, TESTING_PROFILES, VALIDATION_PROFILES, normalize)

% Neural network (single hidden layer) retrieval
%
% This function requires the NETLAB toolbox for MATLAB available for free:
%    http://www.ncrg.aston.ac.uk/netlab/index.php
%    maintained by Ian Nabney (i.t.nabney@aston.ac.uk)
%    For more information, consult Dr. Nabney's textbook:
%    Netlab: Algorithms for Pattern Recognition, ISBN: 1-85233-440-1
%

NUM_PROFILES = size(profiles,2);
NUM_LEVELS = size(profiles,1);
```

```
% Set aside 10 percent of ensemble for validation profiles
if nargin< 15
NUM_PROFILES = size(profiles,2);
TESTING_PROFILES = 1:10:NUM_PROFILES;
VALIDATION_PROFILES = 5:10:NUM_PROFILES;
TRAINING_PROFILES = 1:NUM_PROFILES;
TRAINING_PROFILES([VALIDATION_PROFILES TESTING_PROFILES]) = [];
end

% Check to see if we're retrieving water vapor or temperature
if min(min(profiles))<100 & normalize
fprintf('Water vapor detected.  Normalizing...\n');
  TEMPERATURE=0;
else
  TEMPERATURE=1;
end

prof_train = profiles(:, TRAINING_PROFILES);
prof_test = profiles(:, TESTING_PROFILES);
prof_val = profiles(:, VALIDATION_PROFILES);
clear profiles

rad_train = tbs(:, TRAINING_PROFILES);
rad_test = tbs(:, TESTING_PROFILES);
rad_val = tbs(:, VALIDATION_PROFILES);
clear tbs

mean_prof = mean(prof_train')';
mean_rad = mean(rad_train')';
rad_train_mr = rad_train - mean_rad * ones(1, length(rad_train));
rad_test_mr = rad_test - mean_rad * ones(1, size(rad_test,2));
rad_val_mr = rad_val - mean_rad * ones(1, size(rad_val,2));
prof_train_mr = prof_train - mean_prof * ones(1, length(prof_train));
prof_test_mr = prof_test - mean_prof * ones(1, length(prof_test));
prof_val_mr = prof_val - mean_prof * ones(1, length(prof_val));

if TEMPERATURE == 0   % Normalize water vapor profile
  std_norm_factor = std(prof_train')';
else
  std_norm_factor = ones(size(std(prof_train')'));
end

prof_train_mr = diag(1./std_norm_factor) * prof_train_mr;
prof_test_mr = diag(1./std_norm_factor) * prof_test_mr;
```

```
prof_val_mr = diag(1./std_norm_factor) * prof_val_mr;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%  PPC Section  %%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Crr = rad_train_mr * rad_train_mr' / (length(rad_train)-1);
Cpr = prof_train_mr * rad_train_mr' / (length(rad_train)-1);
Cnn = noise_cov;




if Num_ppcs > 0

   [ppc_evects, ppc_evals] = eigs(Crr, Num_ppcs);
   ppc_evects = real(ppc_evects);  % small imaginary values possible
   clear Crr
   V = ppc_evects(:, 1:Num_ppcs);

   % The following adjustment is needed to make sure that the V's are identical
   % every time. eigs/svd DO NOT return the same answer for successive
   % calls - each column can differ by a scale factor of -1.
   % I'm going to ensure that the first element of each column is always
   % positive so the results will always be consistent.

   scale_factors = ones(size(V(1,:)));
   scale_factors(find(V(1,:)<0)) = -1;
   V = V .* (ones(size(V(:,1))) * scale_factors);

%     [ppc_evects, ppc_evals] = eigs(Cpr / Crr * Cpr', Num_ppcs);
% [U, S, V] = svd(ppc_evects' * Cpr / Crr);
% clear Crr
% V = V(:, 1:Num_ppcs);
%
%%    [ppc_evects, ppc_evals] = eigs(Crr, Num_ppcs);
%%     ppc_evects = real(ppc_evects);  % small imaginary values possible
%%     clear Crr
%%     V = ppc_evects(:, 1:Num_ppcs);
%
%
%    % The following adjustment is needed to make sure that the V's are identical
%    % every time. eigs/svd DO NOT return the same answer for successive
%    % calls - each column can differ by a scale factor of -1.
%    % I'm going to ensure that the first element of each column is always
```

```
%    % positive so the results will always be consistent.
%
%    scale_factors = ones(size(V(1,:)));
%    scale_factors(find(V(1,:)<0)) = -1;
%    V = V .* (ones(size(V(:,1))) * scale_factors);
else
   Num_ppcs = size(rad_train,1);
   V = eye(Num_ppcs);
end


pcs_train = V' * rad_train_mr;
clear rad_train
s_pcs_train = std(pcs_train')';
pcs_train = diag(1./s_pcs_train) * pcs_train;

Snn = sqrtm(diag(1./s_pcs_train) * V' * Cnn * V * diag(1./s_pcs_train));
pcs_test = V' * rad_test_mr;
pcs_val = V' * rad_val_mr;
clear rad_test rad_val
pcs_test = diag(1./s_pcs_train) * pcs_test;
pcs_val = diag(1./s_pcs_train) * pcs_val;

% Initialize output estimate matrix
est_test = zeros(size(prof_test));

temp = zeros(size(Snn));
temp(1:size(Snn,1), 1:size(Snn,2)) = Snn;
Snn = temp;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%% NN Section %%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Assuming 100 levels plus a surface temp

% if NUM_LEVELS < 100
%   error('Check number of profile levels - should be 100 or 101\n');
% end
if nargin<10
if (TEMPERATURE)
  levels_1 = [1:5:60];  % up to ~22.5 km
  levels_2 = [5:5:60];
else
  levels_1 = [1:5:60]; % up to ~12.5 km, 1 is top of atmo
```

```
  levels_2 = [5:5:60];
end
end
%%%%%%%%%%%%%%%%%%%%%% NN Initialization %%%%%%%%%%%%%%%%%%%%%%
MAX_ITER = 100;      % maximum number of training epochs
MAX_DUDS = 10;       % number of consecutive epochs that do not
                     %   reduce the validation error
NUM_VERR_AVG = 100; % number of random noise realizations

% NETLAB options,
options = foptions;
options(1) = 0;        % Set to 1 for verbose display
options(14) = 25;       % maximum number of function evaluations
options(18) = 0.001;  % mu_init
options(19) = 10;      % mu_inc
options(20) = 0.1;     % mu_dec
options(21) = 1e10;;  % mu_max

for j = 1:length(levels_1) % loop over profile "level chunks"
   % Initialize the global validation error minimum
   validation_error_best_global = inf;

   for trial_num = 1:Num_trials  % loop over trials
      output_range = levels_1(j):levels_2(j);
      NUM_OUTPUTS = length(output_range);
      fprintf('--- Preparing neural network...\n');
      % Define skeleton network with a single hidden layer
      nnet_ = mlp(Num_ppcs,Num_nodes,NUM_OUTPUTS,'linear');
      % Initialize weights and biases using Nguyen-Widrow method
      nnet_ = mlpinit_nw(nnet_, [min(pcs_train,[],2) max(pcs_train,[],2)]);

      % validation error for this particular trial
      validation_error_best = inf;
      clear training_error validation_error testing_error
      keep_looping = 1;
      num_duds = 0;
      options(18) = 0.001;  % Reset mu
      i = 1;

index = mlpprior(Num_ppcs,Num_nodes,NUM_OUTPUTS, AW1, AB1, AW2, AB2);
nnet_.index = index.index;
nnet_.alpha = index.alpha;
nnet_.beta = BETAVAL;

      while (keep_looping)
```

```
        fprintf('*** Iteration %d of %d for chunk %d of %d (%d outputs, trial %d ↩
            of %d).  \n', i, MAX_ITER, j, length(levels_1), NUM_OUTPUTS, trial_num↩
            , Num_trials);
        pcs_train_noisy = pcs_train + Snn * randn(size(pcs_train));

        % Train the NN for one epoch using scaled conjugate gradient learning ↩
            method
        [nnet_, options] = netopt(nnet_, options, pcs_train_noisy', prof_train_mr(↩
            output_range,:)', 'scg');
%revaluate evidence

        % Evaluate performance
        validation_error(i) = 0;
        training_error(i) = 0;
        testing_error(i) = 0;
        for k=1:NUM_VERR_AVG
           validation_error(i) = validation_error(i) + mean(mean((prof_val_mr(↩
               output_range,:) - mlpfwd(nnet_, (pcs_val + Snn * randn(size(pcs_val↩
               )))')').^2));
           training_error(i) = training_error(i) + mean(mean((prof_train_mr(↩
               output_range,:) - mlpfwd(nnet_, (pcs_train + Snn * randn(size(↩
               pcs_train)))')').^2));
           testing_error(i) = testing_error(i) + mean(mean((prof_test_mr(↩
               output_range,:) - mlpfwd(nnet_, (pcs_test + Snn * randn(size(↩
               pcs_test)))')').^2));
        end
        validation_error(i) = validation_error(i)/NUM_VERR_AVG;
        training_error(i) = training_error(i)/NUM_VERR_AVG;
        testing_error(i) = testing_error(i)/NUM_VERR_AVG;

        % Check if error has decreased
        if (validation_error(i) < validation_error_best)
           num_duds = 0;
           nnet_best = nnet_;
           validation_error_best = validation_error(i);
           fprintf('—— NEW minimum found!  (Training error = %g, Validation Error↩
               = %g, Testing Error = %g)\n', training_error(i), validation_error(↩
               i), testing_error(i));

        else
           num_duds = num_duds + 1;
        end

        if mod(i,3)==0
[nnet_, gamma] = evidence(nnet_, pcs_train_noisy', prof_train_mr(output_range,:)↩
```

```
    ', 1);
fprintf(1, '\nRe-estimation cycle');
fprintf(1, '  alpha =  %8.5f\n', nnet_.alpha);
fprintf(1, '  beta  =  %8.5f\n', nnet_.beta);
fprintf(1, '  gamma =  %8.5f\n', gamma);
      end
      % Has error failed to decrease for MAX_DUDS consecutive times?
      if (num_duds==MAX_DUDS | i==MAX_ITER)

         keep_looping = 0;
         nnet_performance{j}.training_error{trial_num} = training_error;
         nnet_performance{j}.validation_error{trial_num} = validation_error;
         nnet_performance{j}.testing_error{trial_num} = testing_error;
      else
         i=i+1;
      end

      if validation_error_best < min(validation_error_best_global)
         nnet_best_global = nnet_best;

         fprintf('+++ NEW global minimum found! +++\n');
      end
      validation_error_best_global = [validation_error_best_global ↩
          validation_error_best];
   end
end


nnet{j} = nnet_best_global;
nnet_performance{j}.val = validation_error_best_global;

est_test_ = mlpfwd(nnet{j}, (pcs_test + Snn * randn(size(pcs_test)))')';

est_test_ = diag(std_norm_factor(output_range)) * est_test_;
nnet{j}.std_norm_factor = std_norm_factor;

est_test(output_range,:) = est_test_ + mean_prof(output_range) * ones(1, length(↩
    est_test_));

% Include all the normalization parameters
nnet{j}.mean_prof = mean_prof;
nnet{j}.mean_rad = mean_rad;
nnet{j}.s_pcs_train = s_pcs_train;
nnet{j}.V = V;
nnet{j}.Snn = Snn;
```

```
end
```

B.3 Mixture Density Networks

The following code depends on Ian Nabney's netlab toolbox, available for download from `http://www1.aston.ac.uk/eas/research/groups/ncrg/resources/netlab/downloads/`

```
function [nnet, nnet_performance, pcs_test, prof_test_mr, mean_prof, levels_1, ↩
    levels_2, V, s_pcs_train] = mdn_retrieval(tbs, profiles, noise_cov, Num_ppcs, ↩
    Num_nodes, Num_mixtures, Num_trials)

% MDN (single hidden layer, single gaussian component) retrieval
%
% This function requires the NETLAB toolbox for MATLAB available for free:
%    http://www.ncrg.aston.ac.uk/netlab/index.php
%    maintained by Ian Nabney (i.t.nabney@aston.ac.uk)
%    For more information, consult Dr. Nabney's textbook:
%    Netlab: Algorithms for Pattern Recognition, ISBN: 1-85233-440-1
%



NUM_PROFILES = size(profiles,2);
NUM_LEVELS = size(profiles,1);

% Set aside 10 percent of ensemble for validation profiles
NUM_PROFILES = size(profiles,2);
TESTING_PROFILES = 1:10:NUM_PROFILES;
VALIDATION_PROFILES = 5:10:NUM_PROFILES;
TRAINING_PROFILES = 1:NUM_PROFILES;
TRAINING_PROFILES([VALIDATION_PROFILES TESTING_PROFILES]) = [];


% Check to see if we're retrieving water vapor or temperature
if min(min(profiles))<100
fprintf('Water vapor detected.  Normalizing...\n');
  TEMPERATURE=0;
else
```

```
  TEMPERATURE=1;
end

prof_train = profiles(:, TRAINING_PROFILES);
prof_test = profiles(:, TESTING_PROFILES);
prof_val = profiles(:, VALIDATION_PROFILES);
clear profiles

rad_train = tbs(:, TRAINING_PROFILES);
rad_test = tbs(:, TESTING_PROFILES);
rad_val = tbs(:, VALIDATION_PROFILES);
clear tbs

mean_prof = mean(prof_train')';
mean_rad = mean(rad_train')';
rad_train_mr = rad_train - mean_rad * ones(1, length(rad_train));
rad_test_mr = rad_test - mean_rad * ones(1, size(rad_test,2));
rad_val_mr = rad_val - mean_rad * ones(1, size(rad_val,2));
prof_train_mr = prof_train - mean_prof * ones(1, length(prof_train));
prof_test_mr = prof_test - mean_prof * ones(1, length(prof_test));
prof_val_mr = prof_val - mean_prof * ones(1, length(prof_val));

if TEMPERATURE == 0   % Normalize water vapor profile
  std_norm_factor = std(prof_train')';
else
  std_norm_factor = ones(size(std(prof_train')'));
end

prof_train_mr = diag(1./std_norm_factor) * prof_train_mr;
prof_test_mr = diag(1./std_norm_factor) * prof_test_mr;
prof_val_mr = diag(1./std_norm_factor) * prof_val_mr;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%  PPC Section  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Crr = rad_train_mr * rad_train_mr' / (length(rad_train)-1);
Cpr = prof_train_mr * rad_train_mr' / (length(rad_train)-1);
Cnn = noise_cov;

if Num_ppcs > 0
  [ppc_evects, ppc_evals] = eigs(Crr, Num_ppcs);
   ppc_evects = real(ppc_evects);  % small imaginary values possible
   clear Crr
   V = ppc_evects(:, 1:Num_ppcs);
```

```
   % The following adjustment is needed to make sure that the V's are identical
   % every time. eigs/svd DO NOT return the same answer for successive
   % calls - each column can differ by a scale factor of -1.
   % I'm going to ensure that the first element of each column is always
   % positive so the results will always be consistent.

   scale_factors = ones(size(V(1,:)));
   scale_factors(find(V(1,:)<0)) = -1;
   V = V .* (ones(size(V(:,1))) * scale_factors);
else
   Num_ppcs = size(rad_train,1);
   V = eye(Num_ppcs);
end


pcs_train = V' * rad_train_mr;
clear rad_train
s_pcs_train = std(pcs_train')';
pcs_train = diag(1./s_pcs_train) * pcs_train;

Snn = sqrtm(diag(1./s_pcs_train) * V' * Cnn * V * diag(1./s_pcs_train));
pcs_test = V' * rad_test_mr;
pcs_val = V' * rad_val_mr;
clear rad_test rad_val
pcs_test = diag(1./s_pcs_train) * pcs_test;
pcs_val = diag(1./s_pcs_train) * pcs_val;

% Initialize output estimate matrix
est_test = zeros(size(prof_test));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%% NN Section %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Assuming 100 levels plus a surface temp

% if NUM_LEVELS < 100
%   error('Check number of profile levels - should be 100 or 101\n');
% end

if (TEMPERATURE)
  levels_1 = [91 81 71 61 51 41 31 21 11 1];  % up to ~22.5 km
  levels_2 = [100 90 80 70 60 50 40 30 20 10];
```

```
else
  levels_1 = fliplr(1:97); % up to ~12.5 km
  levels_2 = fliplr(1:97);
end

%%%%%%%%%%%%%%%%%%%%%% NN Initialization %%%%%%%%%%%%%%%%%%%%%%
MAX_ITER = 300;     % maximum number of training epochs
MAX_DUDS = 12;      % number of consecutive epochs that do not
                    %   reduce the validation error
NUM_VERR_AVG = 20; % number of random noise realizations

% NETLAB options,
options = foptions;
options(1) = -1;       % Set to 1 for verbose display
options(14) = 30;      % maximum number of function evaluations
options(18) = 0.001;  % mu_init
options(19) = 10;     % mu_inc
options(20) = 0.1;    % mu_dec
options(21) = 1e10;;  % mu_max

for j = 1:length(levels_1) % loop over profile "level chunks"
   % Initialize the global validation error minimum
   validation_error_best_global = inf;

   for trial_num = 1:Num_trials  % loop over trials
      output_range = levels_1(j):levels_2(j);
      NUM_OUTPUTS = length(output_range);
      fprintf('--- Preparing neural network...\n');
      % Define skeleton network with a single hidden layer
      nnet_ = mdn(Num_ppcs,Num_nodes,Num_mixtures,NUM_OUTPUTS,'0');
      % Initialize weights and biases using GMM init method (ie. mu = mean,
      % sigma = std. dev)
      alpha = 5;
      init_options = zeros(1, 18);
      init_options(1) = -1; % Suppress all messages
      init_options(14) = 10;  % 10 iterations of K means in gmminit
      nnet_ = mdninit(nnet_, alpha, prof_train_mr(output_range,:)', init_options);

      % validation error for this particular trial
      validation_error_best = inf;
      clear training_error validation_error testing_error
      keep_looping = 1;
      num_duds = 0;
      options(18) = 0.001;  % Reset mu
      i = 1;
```

```
while (keep_looping)
   fprintf('*** Iteration %d of %d for chunk %d of %d (%d outputs, trial %d ↩
       of %d). \n', i, MAX_ITER, j, length(levels_1), NUM_OUTPUTS, trial_num↩
       , Num_trials);
   pcs_train_noisy = pcs_train + Snn * randn(size(pcs_train));

   % Train the NN for one epoch using Scaled Conjugate Gradient learning ↩
       method
   [nnet_, options] = netopt(nnet_, options, pcs_train_noisy', prof_train_mr(↩
       output_range,:)', 'scg');

   % Evaluate performance
   validation_error(i) = 0;
   training_error(i) = 0;
   testing_error(i) = 0;
   for k=1:NUM_VERR_AVG
      validation_error(i) = validation_error(i) + mdnerr(nnet_, (pcs_val + ↩
          Snn * randn(size(pcs_val)))', prof_val_mr(output_range,:)');
      training_error(i) = training_error(i) + mdnerr( nnet_, (pcs_train + Snn↩
           * randn(size(pcs_train)))', prof_train_mr(output_range,:)');
      testing_error(i) = testing_error(i) + mdnerr(nnet_, (pcs_test + Snn * ↩
          randn(size(pcs_test)))', prof_test_mr(output_range,:)');
   end
   validation_error(i) = validation_error(i)/NUM_VERR_AVG;
   training_error(i) = training_error(i)/NUM_VERR_AVG;
   testing_error(i) = testing_error(i)/NUM_VERR_AVG;

   % Check if error has decreased
   if (validation_error(i) < validation_error_best)
      num_duds = 0;
      nnet_best = nnet_;
      validation_error_best = validation_error(i);
      fprintf('--- NEW minimum negative log likelihood found! (Training ↩
          error = %g, Validation Error = %g, Testing Error = %g)\n', ↩
          training_error(i), validation_error(i), testing_error(i));
   else
      num_duds = num_duds + 1;
   end

   % Has error failed to decrease for MAX_DUDS consecutive times?
   if (num_duds==MAX_DUDS | i==MAX_ITER)
      keep_looping = 0;
      nnet_performance{j}.training_error{trial_num} = training_error;
      nnet_performance{j}.validation_error{trial_num} = validation_error;
```

```
            nnet_performance{j}.testing_error{trial_num} = testing_error;
        else
            i=i+1;
        end

        if validation_error_best < min(validation_error_best_global)
            nnet_best_global = nnet_best;
            fprintf('+++ NEW global minimum found! +++\n');
        end
        validation_error_best_global = [validation_error_best_global ↩
            validation_error_best];
    end
  end


  nnet{j} = nnet_best_global;
  nnet_performance{j}.val = validation_error_best_global;
  nnet_performance{j}.std_norm_factor = std_norm_factor;

  % Include all the normalization parameters
  nnet_performance{j}.mean_prof = mean_prof;
  nnet_performance{j}.mean_rad = mean_rad;
  nnet_performance{j}.s_pcs_train = s_pcs_train;
  nnet_performance{j}.V = V;
  nnet_performance{j}.Snn = Snn;

end
```

B.4 SPGP code

The following code depends on Snelson's SPGP implementation and Rasmussen's GPR toolbox, available for download from `http://www.gaussianprocess.org`

```
function [mu_test s2_test mu_train s2_train mu_val s2_val spgp_variables] = ↩
    spgp_dimred(x, y, x_val, y_val, xtest, rd, M, use_early_stopping, numiter, ↩
    noise_cov)
% [mean_test var_test mean_train var_train mean_val var_val ] = spgp_dimred(x_train↩
    y_train x_val
% y_val x_test y_test numReducedDim numPsuedoInputs reset_hyperparmas
% use_early_stopping, number of iterations)
%
```

```
% x are N x dim
% y are N x 1
% spgp_variables is a structure containing rd, M, all_hyperparams,
% projection_matrix, psuedo-inputs, h
%
% This code depends on Snelson's SPGP implementation and Rasmussens's GPR
% toolbox.

spgp_variables.rd = rd;
spgp_variables.M = M;
if nargin < 10 % update if more args
    noise_cov = 0;
end

% maximum number of validation failures
MAX_FAILURES = 4;

y = double(y);
x= double(x);
x_val = double(x_val);
xtest = double(xtest);
y_val = double(y_val);
me_y = mean(y); y0 = y - me_y; % zero mean the data
y_val = y_val - me_y;

[N,dim] = size(x);

    %initialize P sensibly (PCA?)
    % initialize use random orthogonal matrices?
     [junk  P_init] = returnPC(x(:,:)', rd);
%     P_init = rand(dim,rd);
%     P_init = P_init.*repmat(1./sum(P_init,1),dim,1);
    % initialize pseudo-inputs to a random subset of training inputs
    [dum,I] = sort(rand(N,1));
    I = I(1:M);
    xb_init = x(I,:);
    xb_init = xb_init*P_init;

    % initialize hyperparameters sensibly (see spgp_lik for how
    % the hyperparameters are encoded)
    % hyp_init(1,1) = -2*log((quantile(x,.95)-quantile(x,.05))'/2); % log 1/(↩
        lengthscales)^2
    hyp_init(1) = log(var(y0,1)); % log size
    hyp_init(2) = log(quantile(y0,.80) - quantile(y0,.20)); % log noise
```

```
    h = [];
    for i = 1:M
        [dum ind] = closest_points((x*P_init)', xb_init(i,:)', 100);
        h = [h std(y0(ind))];
    end

    %normalize for  good reasons
    h = log(h./mean(h));

    % optimize hyperparameters and pseudo-inputs

    w_init = [reshape(xb_init,M*rd,1);reshape(P_init, rd*dim, 1); h';hyp_init'];

% early stopping

min_iter= min(numiter,50);
numloops = max(1, floor(numiter/min_iter));
w=w_init;
fw_val = inf;
val_failures = 0;
for i = 1:numloops
    disp('bloop')
    ws{i} = w;
    [w,f] = minimize(w,'spgp_lik_dr_ht_eff',-min_iter,y0,x,M,rd);
    if use_early_stopping
        fw = spgp_lik_dr_ht_eff(w,y_val,x_val,M,rd)

        if fw_val > 0
            percentimprovement = 1-fw/fw_val;
        else
            percentimprovement = fw/fw_val-1;
        end

        if percentimprovement<0.0005 % percentage improvement?
            disp('val fail')
            val_failures=val_failures+1;
        else
            fw_val = fw;
            bestw=w;
            val_failures = 0; % reset, we stop when there are consecuitve failures
        end
        if val_failures > MAX_FAILURES
            break;
        end
    end
```

```
end
if use_early_stopping
    w = bestw;
end

% [w,f] = lbfgs(w_init,'spgp_lik',200,10,y0,x,M); % an alternative
% optim = optimset('GradObj','on','Display','iter');
% [w,f] = fminlbfgs(@(dubya)(spgp_lik_dr_ht(dubya, y0, x, M, rd)), w_init,optim); %↩
    an alternative

xb = reshape(w(1:M*rd,1),M,rd);
P = reshape(w(M*rd+1:(M+dim)*rd),dim,rd);
hyp = w((M+dim)*rd+1:end);

spgp_variables.xb = xb;
spgp_variables.hyp = hyp;
spgp_variables.P = P;
spgp_variables.ws = ws;
spgp_variables.bestw = bestw;

% PREDICTION
[mu0,s2] = spgp_pred_dr_ht(y0,x,xb,xtest,hyp,P);
[mu0_val, s2_val] = spgp_pred_dr_ht(y0, x, xb, x_val, hyp, P);
[mu0_train, s2_train] = spgp_pred_dr_ht(y0,x,xb,x,hyp,P);
mu_test = mu0 + me_y; % add the mean back on
% mu = mu.*std_y; % restore std
mu_val = mu0_val + me_y;
% if you want predictive variances to include noise variance add noise:
s2_test = s2 + exp(hyp(end));
s2_train = s2_train + exp(hyp(end));
mu_train  = mu0_train + me_y;

% mu0_train = mu0_train.*std_y;
% s2 = s2.*std_y^2; % restore std, s2 is variance
% s2_train = s2_train .* std_y^2; %restore std, s2 is varaince
```

Bibliography

[1] F. Aires, C. Prigent, and W. B. Rossow. Neural network uncertainty assessment using bayesian statistics: a remote sensing application. *Neural Comput.*, 16(11):2415–2458, 2004.

[2] H. H. Aumann, M. T. Chahine, C. Gautier, M. D. Goldberg, E. Kalnay, L. M. McMillin, H. Revercomb, P. W. Rosenkranz, W. L. Smith, D. H. Staelin, L. L. Strow, and J. Susskind. AIRS/AMSU/HSB on the aqua mission: design, science objectives, data products, and processing systems. *IEEE Transactions on Geoscience and Remote Sensing*, 41:253–264, February 2003.

[3] Y. Bazi and F. Melgani. Gaussian process approach to remote sensing image classification. *Geoscience and Remote Sensing, IEEE Transactions on*, 48(1):186 –197, jan. 2010.

[4] Christopher M. Bishop. Mixture density networks. Technical report, Neural Computing Research Group, Aston University, 1994.

[5] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, November 1995.

[6] William J. Blackwell. A neural-network technique for the retrieval of atmospheric temperature and moisture profiles from high spectral resolution sounding data. *IEEE Transaction on Geoscience and Remote Sensing*, 43(11), November 2005.

[7] William J. Blackwell. *Neural Networks in Atmospheric Remote Sensing*. Artech House, April 2009.

[8] William J. Blackwell, Laura J. Bickmeier, R. Vincent Leslie, Michael L. Pieper, Jenna E. Samra, Chinnawat Surussavadee, and Carolyn A. Upham. Hyperspectral microwave atmospheric sounding. *IEEE Transactions on Geoscience and Remote Sensing*, preprint.

[9] Choongyeun Cho and David Staelin. Cloud clearing of atmospheric infrared sounder hyperspectral infrared radiances using stochastic methods. *Journal of Geophysical Research*, 111, April 2006.

[10] J. Blaisdell P. Rosenkranz Edward T. Olsen, J. Susskind. *AIRS/AMSU/HSB Version 5 Level 2 Quality Control and Error Estimation*. Jet Propulsion Laboratory, March 2010.

[11] J. F. G. De Freitas, M. A. Niranjan, A. H. Gee, and A. Doucet. Sequential monte carlo methods to train neural network models. *Neural Comput.*, 12(4):955–993, 2000.

[12] Bo Hu and Kam-Wah Tsui. Distributed evolutionary monte carlo with applications to bayesian analysis. Technical Report 1112, Department of Statistics, University of Wisconsin-Madison, November 2005.

[13] D. C. Liu and J. Nocedal. On the limited memory bfgs method for large scale optimization. *Math. Program.*, 45(3):503–528, 1989.

[14] David J. C. Mackay. A practical bayesian framework for backpropagation networks. *Neural Computation*, 4:448–472, 1992.

[15] Derrick Nguyen and Bernard Widrow. Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights. Technical report, Information Systems Laboratory, Stanford University, 1990.

[16] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.

[17] O. Reale, J. Susskind, R. Rosenberg, E. Brin, E. Liu, L. P. Riishojgaard, J. Terry, and J. C. Jusem. Improving forecast skill by assimilation of quality-controlled airs temperature retrievals under partially cloudy conditions. *Geophysical Research Letters*, 35:8809–+, April 2008.

[18] P.W. Rosenkranz. Retrieval of temperature and moisture profiles from amsu-a and amsu-b measurements. *Geoscience and Remote Sensing, IEEE Transactions on*, 39(11):2429 –2435, nov 2001.

[19] Fabian H. Sinz, Joaquin Quinonero C, Gokhan H. Bakir, Carl E. Rasmussen, and Matthias O. Franz. Learning depth from stereo. In *In Pattern Recognition, Proc. 26th DAGM Symposium*, pages 245–252. Springer, 2004.

[20] J. Sjberg and L. Ljung. Overtraining, regularization, and searching for minimum in neural networks. In *In Preprint IFAC Symposium on Adaptive Systems in Control and Signal Processing*, pages 669–674, 1992.

[21] Edward Lloyd Snelson. *Flexible and efficient Gaussian process models for machine learning*. Ph.D Thesis. University College London, 2007.

[22] L.L. Strow, S.E. Hannon, S. De Souza-Machado, H.E. Motteler, and D. Tobin. An overview of the airs radiative transfer model. *Geoscience and Remote Sensing, IEEE Transactions on*, 41(2):303 – 313, feb. 2003.

[23] Chinnawat Surussavadee and David H. Staelin. Global precipitation retrievals using the noaa amsu millimeter-wave channels: Comparisons with rain gauges. *Journal of Applied Meteorology and Climatology*, 49(1):124–135, 2010.

[24] J. Susskind, C. D. Barnet, and J. M. Blaisdell. Retrieval of atmospheric and surface parameters from AIRS/AMSU/HSB data in the presence of clouds. *IEEE Transactions on Geoscience and Remote Sensing*, 41:390–409, February 2003.

[25] Hans Henrik Thodberg. Ace of bayes: Application of neural networks with pruning. Technical report, The Danish Meat Research Institute, Maglegaardsvej 2, DK-4000, 1993.

[26] S. Twomey. *Introduction to the mathematics of inversion in remote sensing and indirect measurements / S. Twomey.* Elsevier Scientific Pub. Co : distributors for the U.S., Elsevier/North Holland, Amsterdam ; New York :, 1977.

[27] Christopher Williams and Matthias Seeger. Using the nystrom method to speed up kernel machines. In *Advances in Neural Information Processing Systems 13*, pages 682–688. MIT Press, 2001.