Gaussian processes on graphics cards for near infrared spectroscopy

R. Bouckaert, G. Holmes, B. Pfahringer and D. Fletcher
University of Waikato, Hamilton, New Zealand. E-mail: remco@cs.waikato.ac.nz

Introduction

We are interested in general purpose methods for analysis of NIR data that work well over a wide range of NIR data set sizes. Analysis with Gaussian processes (GP) has proven to be more accurate than the widely used linear regression (LR) models on NIR data after performing PLS on large datasets. In this study, we find that the performance of GPs can be pushed further by properly tuning the two parameters involved in GPs, namely the noise level and the kernel variance. Unfortunately, a limitation of GPs is that training tends to be rather slow on large datasets, for example it takes circa 1 hour for our largest data set in our optimized Java implementation. This makes parameter optimization impracticable for these large datasets, especially when we want to use repeated cross validation to establish these parameters reliably. However, by implementing the GP learning algorithm on a graphics processing unit (GPU) instead of a CPU, we manage to reduce training time for a 7500-sample NIR dataset to 25 seconds, allowing parameter tuning to be performed over a wide range of parameter settings.

Materials and methods

Graphics processing units provide tremendous processing power at low cost. With recently released development tools (CUDA) and accompanying linear algebra libraries (CUBLAS), it becomes feasible to use the computational power of the GPU in general purpose computing tasks. Training time of Gaussian processes scales cubic in the number of training instances, which makes it impractical to perform parameter tuning for large datasets. However, by using the computational power of a GPU, we bring the training time down to a manageable level. To show the effectiveness on a range of data set sizes, we considered 6 real world data sets produced by a NIR machine that outputs 700 values per spectrum. This spectrum is Savitzky-Golay smoothed with a window size of 15 and down-sampled (every 4th wavenumber), resulting in a 171 attribute spectrum. The GP algorithm is implemented using the CUDA 2.2 library on a GTX 285 card with 1 GB memory holding 240 cores and is integrated into the Weka machine learning workbench.

Table 1. All results based on 10 times repeated 10 fold cross validation. Number in brackets is standard deviation.

Data set	Size	Root mean square error		GP train time on GPU (seconds)
		PLS + linear regression*	GP	
Lactic	255	0.469 (0.074)	0.462 (0.084)	<1
Storig	414	2.037 (0.385)	1.557 (0.589)	<1
SS	895	1.316 (0.136)	0.961 (0.128)	<1
OMD	1010	3.152 (0.497)	2.970 (0.633)	1
K	6363	0.366 (0.022)	0.250 (0.029)	15
N	7500	0.212 (0.023)	0.156 (0.028)	25

* selecting best number of PLS components

Results and discussion

Table 1 shows an overview of the RMSE for linear regression on PLS filtered NIR data and GPs on unfiltered NIR data with 171 attributes.

The table shows that the GPs reduce the error compared to standard PLS + linear regression. The differences are statistically significant at 5% for all data sets except Lactic, using the corrected paired t-test. These experiments show that GPs perform well over a wide range of data set sizes and furthermore the training time per dataset, as shown in Table 1, allows us to consider a sufficient number of parameter settings that the GP can be well tuned to the dataset.

GP and NIR

In this article, we treat NIR analysis as a regression problem. We assume that the NIR machine produces a spectrum that (after appropriate filtering and smoothing) can be represented by a vector $\boldsymbol{x}$, and we are interested in determining a quantity of interest y, such as nitrogen content in soil samples. To build a model we gather a set of training n samples $(\boldsymbol{x}_i, y_i)$, $i=[1...n]$ where $\boldsymbol{x}_i$ are the vectors representing the spectrum and y_i the accompanying values of interest. The regression problem consists of predicting a value y^* for a new spectrum $\boldsymbol{x}^*$. The Gaussian processes (GP) technique provides a way to solve a regression problem.[1] and we can apply it to our problem as follows.

The first step is to calculate a kernel matrix, which is an nxn matrix where the entries can be interpreted as the correlation between two spectra. There are a large number of ways to specify the kernel matrix, but in our experience with NIR data, Gaussian kernels perform best. The Gaussian kernel leads to a kernel matrix K with entries

$$k_{ij}=k(\boldsymbol{x}_i,\boldsymbol{x}_j)=\exp(-|\boldsymbol{x}_i-\boldsymbol{x}_j|^2/2\sigma^2) \quad (1)$$

where σ is a parameter that needs to be determined empirically, $|.|$ the vector norm and exp represents the exponential function. Since there are n^2 entries, it requires a quadratic algorithm to calculate the matrix.

Once we have the kernel matrix K the prediction for a spectrum $\boldsymbol{x}^*$ of the GP is easily calculated using

$$y^*=k^{T*}(K+\sigma_n^2 I)^{-1}\mathbf{y} \tag{2}$$

where y* the value to predict, k^{T*} an n-dimensional vector with entries $k(\boldsymbol{x}_i,\boldsymbol{x}^*)$ representing correlation between the current spectrum and the ones in the training set, K the kernel matrix, I the identity matrix and $\boldsymbol{y}$ the vector of values for y in the training set. Furthermore, note that the GP comes with a noise parameter σ_n which needs to be determined empirically. Also, note that there is a matrix inversion involved, which requires an $O(n^3)$ algorithm, and which leads to computational problems for larger problems. Note also that the term $\alpha=(K+\sigma_n^2 I)^{-1}\boldsymbol{y}$ in equation (2) needs to be calculated only once at training time. At prediction time, we just compute $y^*=k^{T*}\alpha$.

In summary, we have two parameters to be determined, namely the Gaussian kernel noise σ and the GP noise σ_n. Ideally, these parameters are obtained from the training set through repeated cross validation. However, training a single GP on the largest NIR dataset, which contains 7500 entries, takes one and a half hours, using an optimized version in Java. Searching a grid of 25 entries, five σ for and five for σ_n, and performing 10 times repeated 10 fold cross validation requires the training of 2500 GPs, which would require approximately 156 days if performed consecutively. This is clearly impractical.

GPU & CUDA

Recently, NVIDIA made the Compute Unified Device Architecture (CUDA) library available.[2] This is an extention of the C programming language with GPU specific commands making it possible to access the GPU.

In the following sections implementation details for the two main calculation steps for GPs will be outlined. Firstly, we describe the kernel calculation and secondly the matrix inversion which takes up most of the GPs training time.

Kernel calculation – equation (1)

The essential observation to speed up kernel matrix calculation is that equation (1) is dominated by the calculation of $|\boldsymbol{x}_i-\boldsymbol{x}_j|^2$ which can be written as $|\boldsymbol{x}_i|^2-2\boldsymbol{x}_i.\boldsymbol{x}_j+|\boldsymbol{x}_j|^2$ where $|\boldsymbol{x}_i|$ is the norm of $\boldsymbol{x}_i$ and $\boldsymbol{x}_i.\boldsymbol{x}_j$ the dot product of $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$. The norms of $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ can be pre-calculated, taking a linear algorithm in n, leaving only the dot-products, which are quadratic in n. Since $\boldsymbol{x}_i.\boldsymbol{x}_j=\boldsymbol{x}_j.\boldsymbol{x}_i$ the kernel matrix is symmetric, so we need to calculate only half of the matrix, say the upper triangle, and get the rest for free by just copying results. Furthermore, by examining equation (1) we see that the diagonal equals the identity matrix. Since we are in fact interested in calculating the matrix $K+\sigma_n^2 I$ from equation (2), it suffices to fill the diagonal with entries $1+\sigma_n^2$.

We exploit the fact that the GPU performs operations in blocks of 16 cores by calculating blocks of 16x16 entries in the kernel matrix, where each core calculates entries for a single column. The first step is calculating the dot product. Note that for row i, every one of the 16 cores needs access to vector $\boldsymbol{x}_i$. Since the cores perform the same operation, but with different thread IDs, each of the threads can obtain one of the first 16 entries of $\boldsymbol{x}_i$ in one step, copying them to shared memory so

that each of the cores can access all of the 16 entries. Following this, the 16 entries for the column associated with the core are obtained from device memory. These data are unique for each core, and does not need to be shared. Then the dot-product between the entries is calculated. This process is repeated till all entries of $\boldsymbol{x}_i$ are processed (taking care of boundary conditions, etc.).

Finally, the two pre-calculated norms $|\boldsymbol{x}_i|$ and $|\boldsymbol{x}_j|$ are added and exponentiation is performed to get the result displayed in equation (1). Efficient storage of results takes place by having the 16 cores executing storage of a result at the same time, so that at every cycle a contiguous memory block of 16 results are pushed out to the main device memory, until the complete 16x16 block is stored.

Matrix inversion with 171 attributes and 7500 instances in double precision takes 81.3 seconds on a CPU and 0.38 seconds in the GPU implementation. In practical terms, 81.3 seconds does not sound like a great amount of time. However, when performing grid search over 25 parameter settings using 10 times 10 fold cross validation, the total amount of time spent just doing kernel calculations adds up to about 56 hours as opposed to the 16 minutes it takes in the GPU implementation.

Matrix inversion— Equation (2)

Since the matrix $K+\sigma_n^2 I$ that we want to invert is symmetric, we implemented straight-forward Gaussian elimination, which in sequential implementations is less efficient than Cholesky decomposition, but allows for easy parallelization.

Gaussian elimination consists of three steps: for a square matrix $A=K+\sigma_n^2 I$, we form a matrix $[AI]$ where I is the identity matrix. Then, we perform a forward elimination, which reduces the block occupied by A to a triangular matrix. Next step is to perform back substitution, which converts the first block in an identity matrix. By the observation that $A^{-1}[AI]=[IA^{-1}]$ the block that is occupied by I in the original matrix $[AI]$ is now occupied by A^{-1}, the inverse of $K+\sigma_n^2 I$ that we are after.

Note that during forward elimination, for every column in A, we perform that one sweep over the matrix. This can be done for each cell in the matrix $[AI]$ independent of any other, and the kth sweep only affects row $k+1$ and larger. This makes parallelization of this step very straightforward; it loops k from 1 to $n-1$ and lets every cell in the matrix in rows $k+1$ and larger update itself. After forward elimination, it is necessary to normalize the diagonal to unity, which requires multiplication of each row with one over the current diagonal value. Again, this is an operation that can be performed for each cell completely independent of all others, so it is easy to parallelize. For back substitution, our original implementation performed quite well. At that time, the CUBLAS library[3] for performing standard linear algebra functions for GPU was extended with a function for doing back substitution, which turned out to be slightly faster.

Pivoting was not used in our implementation. As long as the algorithm was run in double precision on the GPU, there were no noticeable differences between the GPU implementation and the Cholesky decomposition CPU implementation. The reason for this is that in the situation of GPs, the matrix $K+\sigma_n^2 I$ is dominated by the diagonal with much smaller entries in the other cells. Therefore, numerical instabilities did not present a problem making pivoting unnecessary.

Empirical results

Experiments were performed using Weka[4] on a number of soil sample datasets. Reference values were obtained through traditional chemistry and matched with their NIR spectra. These spectra

of 700 values were each Savitzky-Golay smoothed, with a window size of 15 and down-sampled (every 4th wavenumber), resulting in a 171 attribute spectrum. We compared performance of GP regression with PLS filtered NIR data on which linear regression (LR) was applied. The results for the PLS+LR method were optimized to use the optimal number of components. The results for GPs were optimized over 25 values for combinations of 5 kernel noise values and 5 GP noise values using 10 times 10 fold cross validation. Table 1 (above) shows an overview of the RMSE for these methods for datasets of various sizes.

Over all, GPs perform significantly better, according to a corrected paired t-test at the 5% significance level, except for the smallest data set. This indicates that it is well worth considering GPs for NIR regression problems.

Also shown in Table 1 is the training time for a single GP for the dataset. Using the GPU reduced training time for the nitrogen dataset with 7500 instances from about one and a half hours in the original Java implementation to 25 seconds. This reduced the parameter search from an estimated 156 days in the sequential implementation to just over 17 hours in the GPU implementation, making it practical to perform such parameter searches.

References

1. C.E. Rasmussen and C.K.I. Williams. *Gaussian Processes for Machine Learning*. The MIT Press, Massachusetts, USA (2006).
2. NVIDIA CUDA Programming Guide. Available from http://www.nvidia.com/object/cuda_develop.html (accessed 28 October 2009).
3. CUDA CUBLAS Library. Available from http://www.nvidia.com/object/cuda_develop.html (accessed 28 October 2009).
4. M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann and I.H. Witten, "The WEKA Data Mining Software: An Update", *SIGKDD Explorations* **11(1),** (2009).