GPU-Accelerated Gaussian Processes for Object Detection

Calum Blair
Institute for Digital Communications
University of Edinburgh
Edinburgh, UK
Email: c.blair@ed.ac.uk

John Thompson
Institute for Digital Communications
University of Edinburgh
Edinburgh, UK

Neil M. Robertson
Institute for Sensors, Signals and Systems
Heriot-Watt University
Edinburgh, UK

***Abstract*—Gaussian Process classification (GPC) allows accurate and reliable detection of objects. The high computational load of squared-error or radial basis function kernels limits the applications that GPC can be used in, as memory requirements and computation time are both limiting factors. We describe our version of accelerated GPC on GPU (Graphics Processing Unit). GPUs have limited memory so any GPC implementation must be memory-efficient as well as computationally efficient. Using a high-performance pedestrian detector as a starting point, we use its packed or block-based feature descriptor and demonstrate a fast matrix multiplication implementation of GPC which is also extremely memory efficient. We demonstrate a speed up of 3.7 times over a multicore, BLAS-optimised CPU implementation. Results show that this is more accurate and reliable than results obtained from a comparable support vector machine algorithm.**

I. Introduction

Gaussian Process Classification (GPC) is an emerging method of performing classification and regression for various machine learning tasks. It has been used for prediction of house prices, detection of various objects in multiple modalities, and behavioural analysis [1]–[3]. It works by approximating the underlying distribution of the variables under observation by using a set of Gaussian distributions.

Recent work has shown that GPCs provide similar levels of classification *accuracy*, while giving predictions which are more *reliable* [4] than other comparable classifiers. See the pedestrian detection example in Figure 1; false positives with high confidence values (green, from another algorithm) are treated less confidently by the GPC (red). In this case, *accuracy* refers to the proportion of samples which are classified correctly [5], while *reliability* is defined as how well a classifier's probabilistic predictions match ground-truth observations. Classification algorithms are more likely to be trusted and used by operators if they behave in a predictable or well-understood manner [6]. Therefore, we argue that ranking classifiers based on reliability – and improving that metric where possible – allows more appropriate deployment of classification and object or anomaly detection algorithms.

The main problem encountered when applying GPCs for large-scale or extremely data-rich machine learning tasks is that the calculations involved are expensive, both in terms of processing time and memory use. General-Purpose Graphics Processing Units (GPUs) have been used for some time now to speed up algorithm processing and evaluation due to the high levels of parallelism available. Other families of machine learning and classification algorithms such as support vector machines (SVMs) [7] and neural networks [8] have been accelerated on GPU, allowing the real-time classification of test images and use in large-scale machine learning tasks.

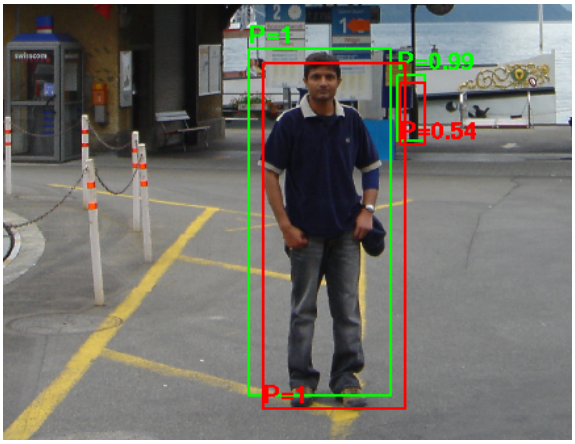

Fig. 1: Existing detector (ACF with Adaboost classifier, green) detects pedestrians and false positives with high confidence. Gaussian Processes classifier (red) assigns low confidence to false positives but maintains high confidence in true positives.

This paper describes our accelerated Gaussian Process classifier implementation focusing on one such problem; reliably detecting pedestrians in images using a state-of-the-art feature extraction pipeline. We then compare this to an accelerated SVM classifier to provide a baseline. Although we concentrate on detection of pedestrians here, this can be extended to any large-scale problem where features are extracted then stored in a memory-efficient manner.

In Section II we describe related work and existing algorithms and implementations. In Section III we describe the algorithm we use in detail, our accelerated implementation of it, and our testing methodology. We present timing, accuracy and reliability results in Section IV and discuss our conclusions and avenues for future work in Section V.

Fig. 2: Feature vector with multiple feature channels (three colour channels, pixel gradient magnitude, and gradient histograms in six angular bins) extracted from an image patch.

II. Related Work

Here we discuss feature extraction, GPCs and implementations.

A. ACF Classifier

The ACF (Aggregate Channel Features) pedestrian detector is one of the top-performing classifiers on current object detection tasks [9]. It is based on one of the most well-known object detectors, histograms of oriented gradients (HOG) [10]. It depends on extracting multiple different feature channels from an image. The reference algorithm by Dollar *et al.* [9] uses ten channels. Starting with a colour image, ACF extracts LUV colour channels, pixel gradient magnitudes, and builds a six-bin histogram of pixel gradient orientations. This approach allows colour and shape features to be extracted. A set of features for an image patch is shown in Figure 2. This is then classified using an Adaboost classifier to give the final detection result. Given the sample resolution involved and the number of channels present, each feature vector has around 5000 entries. Large datasets therefore require large numbers of calculations during training and testing stages. This paper concentrates on acceleration of test-time calculations.

B. Gaussian Process Classification

Here we concentrate on binary classification of samples with labels $y \in \{0, 1\}$. A high-level description of GPCs is given here, then we provide specific details of the approximation algorithm used in Section III. Gaussian Process Classifiers allow probabilistic prediction $p(y = +1|\mathbf{x}_*)$ of a new data sample $\mathbf{x}_*$, after parameter learning on a set of training data $\mathbf{X}$ and matching labels $\mathbf{y}$. This is done in two stages. The first involves defining a latent set of functions $f(x)$. This is assumed to have a Gaussian distribution and can be described by mean $\mu(x)$ and covariance function $k(\mathbf{X}, \mathbf{x}_*)$. This can be a squared-error kernel:

$$k(x_i, x_j) \quad = \quad \sigma \exp\left(-\frac{(\mathbf{x_i} - \mathbf{x_j})^2}{2\ell^2}\right). \tag{1}$$

where σ and ℓ are hyperparameters learned in the training process. A common alternative is the linear kernel:

$$k(x_i, x_j) \quad = \quad \sigma \mathbf{x_i} \cdot \mathbf{x_j}. \tag{2}$$

To train a classifier, the distribution of the variable f_* corresponding to (i.e., which best fits) $\mathbf{x}_*$ must first be estimated:

$$p(f_*|\mathbf{X}, \mathbf{y}, \mathbf{x}_*) = \int p(f_*|\mathbf{X}, \mathbf{x}_*, \mathbf{f}) p(\mathbf{f}|\mathbf{X}, \mathbf{y}) d\mathbf{f}. \tag{3}$$

where $\mathbf{f}$ is the distribution of the latent function over $\mathbf{X}$. The second stage involves 'squashing' f_* with an activation function with output range $[0, 1]$, such as:

$$\sigma(x) = \frac{1}{(1 + \exp(-f(x))}. \tag{4}$$

This can then be used to calculate the final class membership probability:

$$\pi \triangleq p(y = +1|\mathbf{X}, \mathbf{y}, \mathbf{x}_*) = \int \sigma(f_*) p(f_*|\mathbf{X}, \mathbf{y}, \mathbf{x}_*) df_*. \tag{5}$$

The training process is $\mathcal{O}(n^3)$, while testing is $\mathcal{O}(n^2)$.

As a baseline we use the Support Vector Machine (a common classification approach) with a radial basis function (RBF) kernel. The SVM classification equation is given below, with $\alpha, \mathbf{w}$ and b learned during training.

$$f(\mathbf{x}) = \sum_{i=1}^{N} \alpha_i K(\mathbf{x}, \mathbf{w}_i) + b \tag{6}$$

The RBF kernel used is identical to that given in (1).

C. Optimal Computation and Acceleration

Gaussian process regression and classification make extensive use of linear algebra. The standard library used for this is LAPACK (Linear Algebra Package). This relies heavily on BLAS (Basic Linear Algebra Subprograms); this includes vector, matrix and vector-matrix algorithms, such as fast multiplication of two matrices and tools for solution of simultaneous equations[1]. Heavily optimised versions of BLAS and LAPACK have been developed for a wide variety of high-performance processors, where the order of operations and amount of data stored in each level of the cache and memory are tweaked by machine or by hand to ensure optimal performance [11]. Two GPU versions are available, NVIDIA cuBLAS[2] and MAGMA [12].

Software packages such as MATLAB and Numpy rely on BLAS for optimised computation. High-level statements like $C = AB$ are converted to calls to optimised libraries such as:

$$C = \text{sgemm}(A, B)$$

(i.e. single-precision, general matrix-matrix multiply). One of the limitations of using BLAS is that all operations must be reformulated to fit within the available subroutines provided by the library. When evaluated, the exponent term in (1) is expanded to:

$$(\mathbf{x}_i - \mathbf{x}_j)^2 = \mathbf{x}_i^2 + \mathbf{x}_j^2 - 2\mathbf{x}_i\mathbf{x}_j. \tag{7}$$

Three separate calls are then made to BLAS to calculate each right-hand-side expression, resulting in three separate passes over the data. Ideally, we would pass once over each matrix, making all required calculations simultaneously. There is considerable room for improvement here; for extremely large problems (matrices of $> 1000 \times 1000$ in size), memory

[1] http://www.netlib.org/blas/

[2] https://developer.nvidia.com/cuBLAS

accesses as well as computations become a problem; datasets become too big to fit in any cache and must be stored in memory and traversed multiple times for any calculation. This is particularly apparent when working with GPUs, where overall throughput tends to be dominated by memory accesses rather than computational performance. Memory on GPUs is also much more limited than on conventional processors; 2GB or 4GB maximum limits are common. This presents a problem when very large $\mathbf{X}$ matrices are generated by the training process.

The contribution of this paper is to document and make available as code our GPU-accelerated implementation of a GPC inference stage for large classification problems, and present timing, accuracy and reliability results. Although this is coded specifically for the feature extraction stage of the ACF algorithm, it can easily be adapted and used in any algorithm which relies on data extraction from local feature vectors stored in a data-efficient manner.

III. Method

First we describe the algorithm which allows us to approximate the latent function as a Gaussian, then the extraction and storage in memory of ACF descriptors. Finally, we show how the former can be applied to the latter in a computationally and memory efficient manner.

A. GP Inference using Laplacian Approximation

Our goal is to produce an expression for the second term in (5) which we can evaluate analytically or numerically. This requires expressions for the predictive mean $\mathbb{E}[\mathbf{f}_*|\mathbf{X},\mathbf{y},\mathbf{x}_*]$ and predictive variance $\mathbb{V}[\mathbf{f}_*|\mathbf{X},\mathbf{y},\mathbf{x}_*]$. The algorithm here is taken from [13, Ch.3]; see this reference for further details.

We start by defining the relationship between training and test samples:

$$\begin{bmatrix}\mathbf{f}\\ \mathbf{f}_*\end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix}K(\mathbf{X},\mathbf{X}) & K(\mathbf{X},\mathbf{x}_*)\\ K(\mathbf{x}_*,\mathbf{X}) & K(\mathbf{x}_*,\mathbf{x}_*)\end{bmatrix}\right). \tag{8}$$

Where K describes the covariance matrix between training samples $\mathbf{X}$ and test samples $\mathbf{x}_*$, and is calculated using a kernel function $k(x_i, x_j)$ such as (1) or (2).

For notational convenience, we define K_X as $K(\mathbf{X},\mathbf{X})$, K_{X*} as $K(\mathbf{X},\mathbf{x}_*)$ and K_* as $K(\mathbf{x}_*,\mathbf{x}_*)$ throughout. Using [13, Ch.2] and [14, Ch.8,§9.3], we can define a conditional Gaussian on (8) as:

$$\begin{aligned}\mathbf{f}_*|\mathbf{X},\mathbf{x}_*,\mathbf{f} \sim \mathcal{N}(&K(\mathbf{x}_*,\mathbf{X})K_X^{-1}\mathbf{f},\\ &K_* - K(\mathbf{x}_*,\mathbf{X})K_X^{-1},K_{X*})\,.\end{aligned} \tag{9}$$

Evaluating this directly as in (3) is intractable [15], so we use a Laplacian approximation (from [13, Ch.3§4]), which allows the posterior over the training data and labels in (3) to be approximated as a Gaussian:

$$p(\mathbf{f}|\mathbf{X},\mathbf{y}) \approx q(\mathbf{f}|\mathbf{X},\mathbf{y}) = \mathcal{N}(\hat{\mathbf{f}}, A^{-1})\,, \tag{10}$$

where

$$\hat{\mathbf{f}} = \arg\max_{\mathbf{f}} p(\mathbf{f}|\mathbf{X},\mathbf{y})\,, \tag{11}$$

Require: $\mathbf{X}, \mathbf{y}, \mathbf{f}$, kernel function $k(x_i, x_j)$
1: $\hat{\mathbf{f}} \triangleq \mathbb{E}_q[\mathbf{f},\mathbf{X},\mathbf{y}] = \operatorname{argmax}_{\mathbf{f}}\ p(\mathbf{f}|\mathbf{X},\mathbf{y})$ // Using Newton's method
2: $K_X = K(\mathbf{X},\mathbf{X})$
3: $W = -\nabla\nabla\log(p(\mathbf{y}|\hat{\mathbf{f}}))$
4: $L = \text{cholesky}(I + W^{\frac{1}{2}}KW^{\frac{1}{2}})$
5: **return** $W, L, \hat{\mathbf{f}}, K_X$

Fig. 3: Prepare training posterior. This only needs to be done once and can be re-used during testing.

and (where ∇ represents differentiation):

$$A = -\nabla\nabla\log(p(\mathbf{f}|\mathbf{X},\mathbf{y})|_{\mathbf{f}=\hat{\mathbf{f}}}\,. \tag{12}$$

$\hat{\mathbf{f}}$ can thus be found by applying Bayes' rule to the posterior distribution over the training variables, $p(\mathbf{f}|\mathbf{X},\mathbf{y}) = p(\mathbf{y}|\mathbf{f})p(\mathbf{f}|\mathbf{X})/p(\mathbf{y}|\mathbf{X})$. Here, $p(\mathbf{y}|\mathbf{X})$ can be discarded as we wish to maximise $\mathbf{f}$. Taking the log of $p(\mathbf{f}|\mathbf{X},\mathbf{y})$ and differentiating, we obtain the equation for the predictive mean:

$$\mathbb{E}_q[\mathbf{f}_*|\mathbf{X},\mathbf{y},\mathbf{x}_*] = K_{X*}^{\intercal}\nabla\log p(\mathbf{y}|\hat{\mathbf{f}})\,. \tag{13}$$

Similarly, we define predictive variance as:

$$\mathbb{V}_q[\mathbf{f}_*|\mathbf{X},\mathbf{y},\mathbf{x}_*] = K_* - K_{X*}^{\intercal}(K_X + W^{-1})^{-1}K_{X*}\,. \tag{14}$$

With the diagonal matrix $W \triangleq -\nabla\nabla\log(p(\mathbf{y}|\mathbf{f}))$.

Defining the symmetric positive definite matrix $\mathbf{B}$ as $\mathbf{B} = I + W^{\frac{1}{2}}K_XW^{\frac{1}{2}}$ and using a Cholesky decomposition $\mathbf{LL}^{\intercal} = \mathbf{B}$ such that $\mathbf{L} = \text{cholesky}(\mathbf{B})$, (14) can be simplified to:

$$\mathbb{V}_q[\mathbf{f}_*|\mathbf{X},\mathbf{y},\mathbf{x}_*] = K_* - K_{X*}^{\intercal}W^{\frac{1}{2}}(LL^{\intercal})^{-1}W^{\frac{1}{2}}K_{X*} \tag{15}$$

$$\mathbb{V}_q[\mathbf{f}_*|\mathbf{X},\mathbf{y},\mathbf{x}_*] = K_* - \mathbf{v}^{\intercal}\mathbf{v} \tag{16}$$

where:

$$\mathbf{v} = L\backslash(W^{\frac{1}{2}}K_{X*})\,. \tag{17}$$

Finally, this allows the posterior in (5) to be approximated as a Gaussian $q(f_*|\mathbf{X},\mathbf{y},\mathbf{x}_*)$ with mean (13) and variance (16). This can then be used to calculate mean and variance values of new samples, and hence assign them a probabilistic prediction value. These are summarised in Figure 3 (generation of intermediate matrices, given training data) and Figure 4 (test-time prediction). Compute-heavy lines in the prediction stage are marked with ▶; it is these that we must consider accelerating. The usual method of calculating every line in Figure 4, and the one which is best supported by BLAS, is to store training and test data with one column for each sample.

B. Feature Extraction and Sliding-Window Classification

The ACF detector, as with many object detectors, works by the 'sliding window' approach; first a set of features for the whole image is generated, then a classification window is run over all the features representing the source image. All features inside the window are formed into a feature vector and classified. Then the window is moved a short distance (e.g. 8 pixels) in one direction, and this process is repeated. As a single image patch 'block' representing a small

Require: $\mathbf{X}, \mathbf{x}_*, \mathbf{y}, \hat{\mathbf{f}}, W, L$, kernel function $k(x_i, x_j)$
1: $K_{X*} = K(\mathbf{X}, \mathbf{x}_*)$ ►
2: $K_* = K(\mathbf{x}_*, \mathbf{x}_*)$ ►
3: $\mathbb{E}_q[f_*|X, \mathbf{y}, \mathbf{x}_*] = K_{X*}^\intercal \nabla \log(p(\mathbf{y}|\hat{\mathbf{f}}))$ // latent mean
4: $\mathbf{v} = L \backslash (W^{\frac{1}{2}} K_{X*})$ ►
5: $\mathbb{V}_q[f_*|X, \mathbf{y}, \mathbf{x}_*] = K_* - \mathbf{v}^\intercal \mathbf{v}$ // latent variance
6: $\bar{\pi}_* = \int \sigma(z) \mathcal{N}(z|E_q[f_*], \mathbb{V}_q[f_*]) dz$ // prediction
7: **return** $\bar{\pi}$

Fig. 4: Calculate π at test time. Compute-heavy lines marked with ►.

pixel region can belong to many windows simultaneously, the memory-efficient way to do this is to store the features block by block, channel by channel, and iterate through them as they are classified at test time, only forming complete feature vectors as they are presented to the classification algorithm. This is incompatible with the BLAS approach; thus we can either duplicate image blocks many times over to form BLAS-compatible feature vectors, prioritising speed of computation at the expense of memory consumption (and, in practice, slow down calculations as we shuffle test vectors in and out of cache), or prioritise a lower memory footprint, but be unable to use the efficient BLAS routines. Here we combine the benefits of both approaches (low memory consumption and optimised processing).

C. GPU Acceleration

We now consider GPU acceleration of the algorithm in Figure 4. The computationally expensive parts are the calculation of K_{X*} and K_* in Figure 4, lines 1 and 2, and the calculation of $\mathbf{v}$ on line 4. For K_{X*} and K_*, we consider the work documented in [12]. Kurzak *et al.*, through extensive automated parameter space exploration, produced an optimised version of the $C = AB$ matrix multiplication algorithm for a GPU [12]. Given $A = m \times D$ and a $B = D \times n$ matrices, each kernel walked along a vector of dimension D, loaded an optimal number of values from the A and B matrices into fast shared memory, then cross-multiplied and summed to produce a single entry in C.

In our case, the A matrix is arranged as we would expect (as the training matrix has one sample in each row). All samples are separate, as they represent discrete positive or negative training examples with no overlap. The B matrix, however, is packed as described in §III-B. We therefore build a per-block lookup table of B entries, then read the corresponding entry. As B is so densely packed, this greatly reduces the number of reads from slow global memory which must be done, and means the calculation is dominated by reading of A and writing of C (the resulting K_{X*} matrix). As [12] proved, these are already well-arranged. Subsequent kernels take the result of (7) and apply the remaining steps in (1) to get K_{X*} and K_*, which is then transferred to the host to continue with Figure 4. The solution of the division involving the lower triangular matrix L on line 4 requires too much memory to obtain any benefit from performing the calculation on a GPU. In our experiments it proved to be faster to execute this on the CPU; the memory limitations on the GPU meant that the test covariance matrix K_{X*} had to be partitioned into very small batches, because of the large size of K_X. This decreased transfer-to-compute ratio eventually meant that GPU processing of (17) was slower than performing the calculation on CPU.

TABLE I: Time taken to perform matrix multiplication stage as part of classification algorithm in a 640×480 image using baseline (CPU) and GPU-accelerated Gaussian Process Classification and RBF support vector machine classification.

Algorithm	Processor	Implementation	Time(s)	Speedup
GPC	CPU	MATLAB BLAS	10.28	
GPC	GPU	GPGPGPU	2.77	3.7×
SVM	CPU	LIBSVM	66.92	
SVM	GPU	cuSVM [17]	1.74	38.5×

IV. Results

Here, to establish accuracy results, we consider detection performance on the INRIA pedestrians dataset, using the testing methodology in [16]. We compare this to a RBF (radial basis function) SVM classifier accelerated on GPU, as this is an accelerated detector which uses a similar approach [17]; the calculation in (6) is very similar to Figure 4 line 1, but this computation is the only step required for SVM prediction. Figure 5, using a ROC curve, shows the true positive rate (TPR) achieved for a given false positive rate (FPR). Similarly, Figure 6 shows a clearer separation between the two approaches when the rate of false positives per image is plotted against the miss rate. By both measurements, the GPC is more accurate than the SVM version. This is true whether the evaluations are run on CPU or GPU. The reliability diagram in Figure 7 plots the confidence scores of predicted samples against ground truth. Here, a 'well-calibrated' or perfectly reliable detector would lie on the black $y = x$ line (i.e. of all the detections it predicts with 60% confidence, 60% will be evaluated as true. It shows that the GP classifier is more reliable, as it lies closer to the black line representing a 'well-calibrated' classifier.

Timing results are given in Table I. As a baseline we compare this to a MATLAB BLAS GPU-accelerated implementation. The CPU version used an Intel Xeon X5650, with 12 cores at 2.67GHz. The GPU results used a NVIDIA GeForce GTX 680 with 1536 cores and 2GB RAM. This shows a 3.7× speedup compared to an optimised CPU processor. However, every prediction also requires (17) to be applied. In contrast, the cuSVM implementation in Table I performs classification in a single step and runs faster. This is partly because the SVM requires fewer support vectors to be multiplied (3000 as opposed to all 14000 training vectors used by the GPC), and partly because the CPU SVM implementation is less optimised.

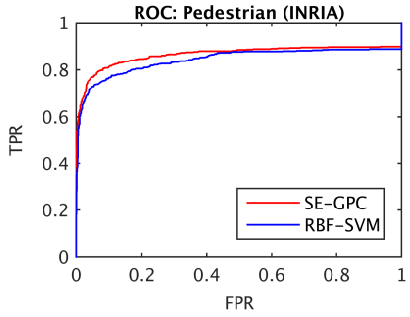

Fig. 5: GPC and SVM Receiver Operating Characteristic curve.

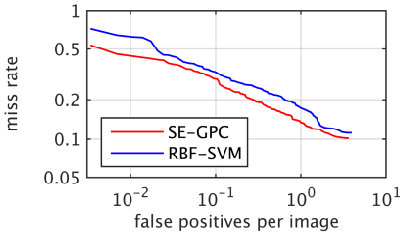

Fig. 6: GPC and SVM accuracy expressed as Detection Error Tradeoff curve. The GPC is always more accurate.

A. Discussion

This has speeded up the operation of the GPC method by a factor of 3.7. An alternative to this approach is to look at algorithmic methods for reducing the number of computations required to classify a new sample. Two methods can be used; reducing the quantity of training or test data which must be used in order to classify a sample, or simplifying the relatively expensive functions in the classification stage. Previous work [4] has considered the latter, showing that GPC with a linear kernel (2) performs significantly less well in both accuracy and reliability.

V. Conclusions

This work has described the motivation behind customised GPU kernels for a common mathematical operation: the matrix version of the operation in (7), and their application to Gaussian Process Classification. This method demonstrates a speedup over the BLAS approach, where matrix multiplications operations are heavily optimised for a given processor. Therefore, when complex algorithms cannot be adequately represented by the available BLAS algorithms, a customised approach still offers a measurable benefit. However, as Table I shows, this approach is still slower than a SVM classifier with a similar kernel. Faster SVM predictions must be balanced against the increased accuracy and reliability available with GPCs. However, more efficient approaches are still needed to deliver real time performance. The code described here is available for download in [18].

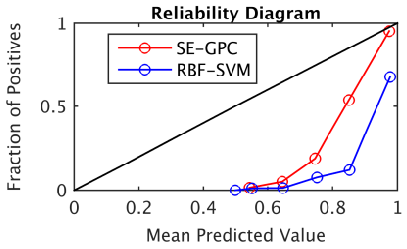

Fig. 7: GPC and SVM Reliability; classifiers closer to black line are more reliable.

Acknowledgement

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC) Grant number EP/J015180/1 and the MOD University Defence Research Collaboration in Signal Processing.

References

[1] R. Urtasun, D. J. Fleet, and P. Fua, "3D people tracking with Gaussian process dynamical models," in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, vol. 1. IEEE, 2006, pp. 238–245.

[2] H. Grimmett, R. Paul, R. Triebel, and I. Posner, "Knowing When We Don't Know: Introspective Classification for Mission-Critical Decision Making," in *Proc. IEEE Int. Conf. Robot. Autom.*, May 2013.

[3] S. Reece, S. Roberts, D. Nicholson, and C. Lloyd, "Determining intent using hard/soft data and Gaussian process classifiers," in *Inf. Fusion, Proc. 14th Int. Conf.*, 2011, pp. 1–8.

[4] C. G. Blair, J. Thompson, and N. M. Robertson, "Introspective Classification for Pedestrian Detection," in *Sens. Signal Process. Def. (SSPD 2014)*, Edinburgh, 2014.

[5] D. J. Hand, *Construction and Assessment of Classification Rules*. Wiley, 1997.

[6] M. T. Dzindolet, S. A. Peterson, R. A. Pomranky, L. G. Pierce, and H. P. Beck, "The role of trust in automation reliance," *Int. J. Hum. Comput. Stud.*, vol. 58, no. 6, pp. 697–718, Jun. 2003.

[7] A. Athanasopoulos, A. Dimou, V. Mezaris, and I. Kompatsiaris, "GPU acceleration for support vector machines," in *Proc. 12th Int. Work. Image Anal. Multimed. Interact. Serv. (WIAMIS 2011)*, no. April, 2011.

[8] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with COTS HPC systems," in *Proc. 30th Int. Conf. Mach. Learn.*, 2013, pp. 1337–1345.

[9] P. Dollár, R. Appel, S. Belongie, and P. Perona, "Fast feature pyramids for object detection," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 36, no. 8, pp. 1532–1545, Aug. 2014.

[10] N. Dalal and B. Triggs, "Histograms of Oriented Gradients for Human Detection," in *Proc. 2005 IEEE Conf. Comput. Vis. Pattern Recognit.* IEEE, 2005.

[11] K. Goto and R. A. van de Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Softw.*, vol. 34, no. 3, pp. 1–25, 2008.

[12] J. Kurzak, S. Tomov, and J. Dongarra, "Autotuning GEMM kernels for the Fermi GPU," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 11, pp. 2045–2057, 2012.

[13] C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning*. University Press Group Limited, 2006.

[14] R. von Mises, *Mathematical Theory of Probability and Statistics*. Academic Press, 1964.

[15] C. Williams and D. Barber, "Bayesian classification with Gaussian processes," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 20, no. 12, pp. 1342–1351, 1998.

[16] P. Dollár, C. Wojek, B. Schiele, and P. Perona, "Pedestrian Detection: An Evaluation of the State of the Art," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 34, no. 4, pp. 743–762, Jul. 2011.

[17] A. Carpenter, "Cusvm: a cuda implementation of support vector classification and regression," 2009.

[18] C. G. Blair, "GPGPGPU example," 2015. [Online]. Available: http://homepages.ed.ac.uk/cblair2/code/gpgpgpu_example.zip