McKernel: A Library for Approximate Kernel Expansions in Log-linear Time

J. D. Curtó[*,1,2,3,4], **I. C. Zarza**[*,1,2,3,4], **F. Yang**[2,6],
A. Smola[2,5,6], **F. De La Torre**[2], **C. W. Ngo**[3], **and L. Van Gool**[4]
[1]The Chinese University of Hong Kong [2]Carnegie Mellon [3]City University of Hong Kong
[4]Eidgenössische Technische Hochschule Zürich [5]Amazon [6]Google
{curto,zarza}.2@my.cityu.edu.hk, fengyang@google.com, smola@amazon.com,
ftorre@cs.cmu.edu, cwngo@cs.cityu.edu.hk, vangool@vision.ee.ethz.ch
[*]Both authors contributed equally

Abstract

Kernel Methods Next Generation (KMNG) introduces a framework to use kernel approximates in the mini-batch setting with SGD Optimizer as an alternative to Deep Learning. We provide McKernel[1], a C++ library for KMNG ML Large-scale. It contains a CPU optimized implementation of the Fastfood algorithm in [1], that allows the computation of approximated kernel expansions in log-linear time. The algorithm requires to compute the product of Walsh Hadamard Transform (WHT) matrices. A cache friendly SIMD Fast Walsh Hadamard Transform (FWHT) that achieves compelling speed and outperforms current state-of-the-art methods has been developed. McKernel allows to obtain non-linear classification combining Fastfood and a linear classifier.

1 Introduction

Kernel methods offer state-of-the-art estimation performance. They provide function classes that are flexible and easy to control in terms of regularization properties. However, the use of kernels in large-scale machine learning problems has been beset with difficulty. This is because using kernel expansions in large datasets is too expensive in terms of computation and storage. In order to solve this problem, [1] proposed an approximation algorithm, called Fastfood, based on Random Kitchen Sinks by [2], that speeds up the computation of a large range of kernel functions, allowing us to use them in big data. Recent works on the topic build on Fastfood to propose state-of-the-art deep learning embeddings, [3] and [4].

At its heart, McKernel requires scalar multiplications, a permutation, access to trigonometric functions, and two Walsh Hadamard Transforms (WHT) for implementation. The key computational bottleneck here is the WHT. We provide a fast, cache friendly SIMD oriented implementation that outperforms state-of-the-art codes such as Spiral [5]. To allow for very compact distribution of models, we use hash functions and a Pseudorandom Permutation Generator for portability. In this way, for each feature dimension, we only need one floating point number. In summary, our implementation serves as a drop-in feature generator for linear methods where attributes are generated on-the-fly, such as for regression, classification, or two-sample tests. This obviates the need for explicit kernel computations, particularly on large amounts of data.

[1]McKernel, https://www.github.com/curto2/mckernel/

Preprint. Work in progress.

Outline: We begin with a brief operational description of the feature construction McKernel in Section 2. This is followed by a discussion of the computational issues for a SIMD implementation in Section 3. The concepts governing the API are described in Section 4. KMNG is introduced in Section 5 and illustrated in Section 6. Section 7 gives a brief overall discussion.

2 McKernel

Kernel methods work by defining a kernel function $k(x, x')$ on a domain $\mathcal{X}$. We can write k as inner product between feature maps, as follows

$$k(x, x') = \langle \phi(x), \phi(x') \rangle \tag{1}$$

for some suitably chosen ϕ. Random Kitchen Sinks [2] approximate this feature mapping ϕ by a Fourier expansion in the case of radial basis function kernels, i.e. whenever $k(x, x') = \kappa(x - x')$. This is possible since the Fourier transform diagonalizes the corresponding integral operator [6]. This leads to

$$k(x, x') = \int \exp i \langle w, x \rangle \exp -i \langle w, x \rangle d\rho(w) \tag{2}$$

for some L_2 measurable function $\rho(\omega) \geq 0$ that is given by the Fourier transform of κ. Random Kitchen Sinks exploit this by replacing the integral by sampling $\omega_j \sim \rho(\omega) / \|\rho\|_1$. This allows for finite dimensional expansions but it is costly due to the large number of inner products required. Fastfood resolves this for rotationally invariant κ by providing a fast approximation of the matrix $W := [\omega_1, \dots \omega_n]$.

This is best seen for the RBF Gaussian kernel where $k(x, x') = \exp\left(-\frac{1}{2\sigma^2} \|x - x'\|^2\right)$. Since Fourier transforms of Gaussians are Gaussians, albeit with inverse covariance, it follows that $\rho(\omega) \propto \exp\left(-\frac{\sigma^2}{2} \|\omega\|^2\right)$ and that W contains i.i.d Gaussian random variables. It is this matrix that Fastfood approximates via

$$V := SHG\Pi HB \tag{3}$$

Here S, G and B are diagonal matrices, Π is a random permutation matrix and H is the Hadamard matrix. Whenever the number of rows in W exceeds the dimensionality of the data, we can simply generate multiple instances of V, drawn i.i.d, until the required number of dimensions is obtained.

Diagonal Binary Matrix B This is a matrix with entries $B_{ii} \in \{\pm 1\}$, drawn from the uniform distribution. To avoid memory footprint, we simply use Murmurhash[2] as hash function and extract bits from $h(i, j)$ with $j \in \{0, \dots N\}$.

Walsh Hadamard Matrix H This matrix is iteratively composed of $H_{2n} = \begin{bmatrix} H_n & H_n \\ -H_n & H_n \end{bmatrix}$. It is fixed and matrix-vector products are carried out efficiently in $O(n \log n)$ time using the Fast Walsh Hadamard Transform (FWHT). We will discuss implementation details for a fast variant below.

Permutation Matrix Π We generate a random permutation using the Fisher-Yates shuffle. That is, given a list $L = \{1, \dots n\}$ we generate permutations recursively as follows: pick a random element from L. Use this as the image of n and move n to the position where the element was removed. The algorithm runs in linear time and its coefficients can be stored in $O(n)$ space. Moreover, to obtain a deterministic mapping, replace the random number generator with calls to the hash function.

Gaussian Scaling G This is a diagonal matrix with i.i.d Gaussian entries. We generate the random variates using the Box-Muller transform [7] while substituting the random number generator by calls to the hash function to allow us to recompute the values at any time without the need to store random numbers.

Diagonal Scaling S This is a random scaling operator whose behavior depends on the type of kernel chosen, such as the RBF Matern Kernel, the RBF Gaussian Kernel or any other radial spectral distribution, [8].

[2] https://code.google.com/p/smhasher/

3 FWHT Implementation

A key part of the library is a FWHT efficient implementation. In particular, McKernel offers considerable improvement over the Spiral[3] library, due to automatic code generation, the use of SIMD intrinsics (SSE2 using 128 bit registers) and loop unrolling. This decreases the memory overhead.

McKernel proceeds with vectorized sums and subtractions iteratively for the first $\frac{n}{2^k}$ input vector positions (where n is the length of the input vector and k the iteration starting from 1), computing the intermediate operations of the Cooley-Tukey algorithm till a small Hadamard routine that fits in cache. Then the algorithm continues in the same way but starting from the smallest length and doubling on each iteration the input dimension until the whole FWHT is done in-place.

For instance, on an intel i5-4200 CPU @ 1.60GHz laptop the performance obtained can be observed in Figure 1.

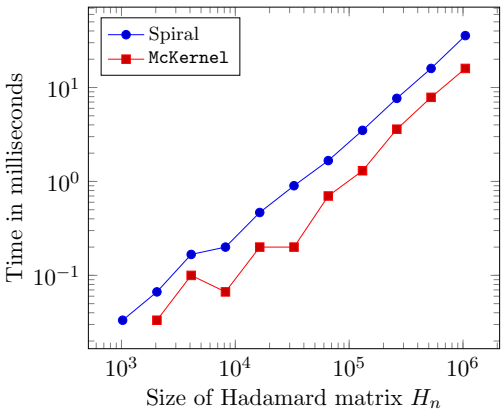

Figure 1: **FWHT Comparison**. McKernel (red) outperforms Spiral (blue) across the range of arguments.

Our code outperforms Spiral by a factor of 2 consistently throughout the range of arguments. Furthermore, Spiral needs to precompute trees and by default can only perform the computation up to matrix size $n = 2^{20}$. On the other hand, our implementation works for any size since it computes the high-level partitioning dynamically.

4 API Description

The API follows the factory design pattern. That is, while the McKernel object is fairly generic in terms of feature computation, we have a factory that acts as a means of instantiating the parameters according to pre-specified sets of parameters for e.g. a RBF Gaussian Kernel or a RBF Matern Kernel. The so-chosen parameters are deterministic, given by the values of a hash function. The advantage of this approach is that there is no need to save the coefficients generated for McKernel when deploying the functions. That said, we also provide a constructor that consumes B, Π, G and S to allow for arbitrary kernels.

4.1 Customizing McKernel

For instance, to generate each S_{ii} entry for RBF Matern Kernel we draw t iid samples from the n-dimensional unit ball S_d, add them and compute its Euclidean norm. To draw efficiently samples from S_d we use the algorithm provided below.

[3] http://www.spiral.net

Let $\mathbf{X} = (X_1, \ldots, X_n)$ be a vector of iid random variables drawn from $\mathcal{N}(0, 1)$, and let $||\mathbf{X}||$ be the Euclidean norm of $\mathbf{X}$, then $\mathbf{Y} = (\frac{X_1}{||\mathbf{X}||}, \ldots, \frac{X_n}{||\mathbf{X}||})$ is uniformly distributed over the n-sphere, where it is obvious to see that $\mathbf{Y}$ is the projection of $\mathbf{X}$ onto the surface of the n-dimensional sphere. To draw uniform random variables in the n-ball, we multiply $\mathbf{Y}$ by $U^{1/n}$ where $U \sim U(0, 1)$. This can be proved as follows: Let $\mathbf{Z} = (Z_1, \ldots, Z_n)$ be a random vector uniformly distributed in the unit ball. Then, the radius $R = ||\mathbf{Z}||$ satisfies $\mathbb{P}(R \leq r) = r^n$. Now, by the inverse transform method we get $R = U^{1/n}$.Therefore to sample uniformly from the n-ball the following algorithm is used: $\mathbf{Z} = rU^{1/n}\frac{\mathbf{X}}{||\mathbf{X}||}$.

5 KMNG

We introduce Kernel Methods Next Generation (KMNG) as an alternative to Deep Learning, where we argue that current Neural Network techniques are surrogates to very large kernel expansions, where optimization is done in a huge parameter space where the majority of learned weights are trivial to the actual problem statement. We introduce here the idea that current developments in very deep neural networks can be achieved by KMNG while drastically reducing the number of parameters learned. We build on the work in [2] and [1] to expand its scope to mini-batch training with SGD Optimizer, we name this approach KMNG ML Large-scale. Our concern is to demonstrate that we are able to get the same gains obtained in Deep Learning by the use of McKernel and a Linear Classifier but with a giant kernel expansion. We believe that current neural network research is just over-optimizing parameters that are indeed not informative for the problem to solve. The same way that a high-school student cannot grasp the whole picture of a specific subject at once, until he is exposed to formal mathematical ideas, Deep Learning researchers have focused in solving trivial problems that are, in fact, not relevant for the tackled optimization problem and overseeing that are just optimizing parameters of a very large kernel expansion. In the following section we build on these ideas to propose some very simple examples to illustrate the essence of the problem and the solution proposed.

6 Empirical Analysis and Experiments

KMNG generalizes the use of McKernel in the mini-batch setting with SGD Optimizer, drastically reducing the number of parameters that need to be learned to achieve comparable Deep Learning state-of-the-art results. Figure 2 and 3 show RBF Matern (MRBF), softmax$(W\tilde{x} + b)$ where $\tilde{x} =$ mckernel(x), performance compared to logistic regression, softmax$(Wx + b)$, in the full-batch and mini-batch setting, respectively. A fixed PRNG seed is used to obtain deterministic reproducible behavior. In full-batch mode, the number of samples for train and test is rounded to the nearest power of 2 due to algorithm constraint.

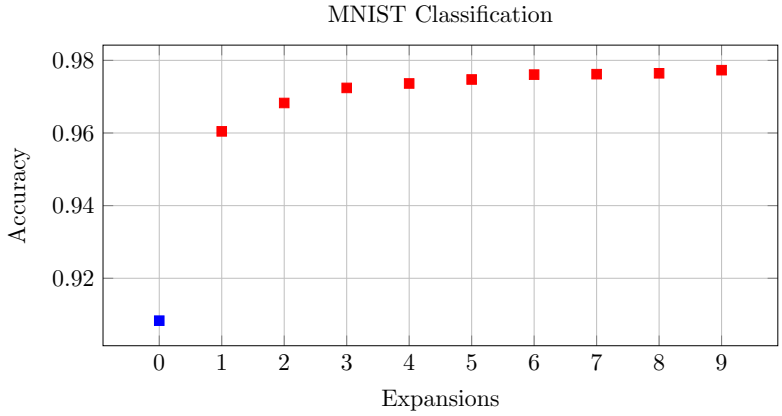

Figure 2: **MNIST Classification.** Logistic Regression (blue) and MRBF (red) with increasing number of Kernel Expansions. 32768 samples of training data and 8192 samples of testing data are used in the learning process. MRBF hyper-parameters, $\sigma = 1.0$, $t = 40$. PNRG seed 1398239763, learning rate 0.001 and batch size 10. LR learning rate 0.01. Number of epochs 20.

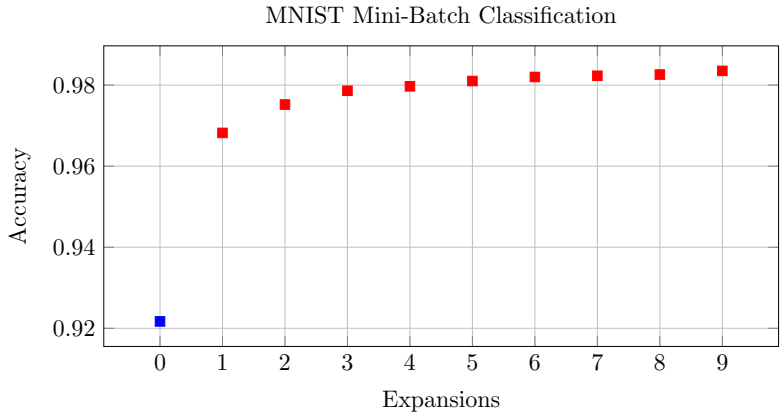

Figure 3: **MNIST Mini-Batch Classification.** Logistic Regression (LR) (blue) and MRBF (red) with increasing number of Kernel Expansions. 60000 samples of training data and 10000 samples of testing data are used in the learning process. MRBF hyper-parameters, $\sigma = 1.0$, $t = 40$. PNRG seed 1398239763, learning rate 0.001 and batch size 10. LR learning rate 0.01. Number of epochs 20.

7 Discussion

In this manuscript we provide the framework for Kernel Methods Next Generation (KMNG) and illustrate with two examples that achieves comparable state-of-the-art Neural Networks performance, proposing a new way to understand Deep Learning, just as a giant kernel expansion where optimization is only performed over the parameters that are actually relevant to the problem at-hand.

References

[1] Le, Q., Sarlós, T., Smola, A.: Fastfood - approximating kernel expansions in loglinear time. ICML (2013) 1, 4

[2] Rahimi, A., Recht, B.: Random features for large-scale kernel machines. NIPS (2007) 1, 2, 4

[3] Yang, Z., Moczulski, M., Denil, M., de Freitas, N., Smola, A., Song, L., Wang, Z.: Deep fried convnets. ICCV (2015) 1

[4] Hong, W., Yuan, J., Bhattacharjee, S.D.: Fried binary embedding for high-dimensional visual features. CVPR (2017) 1

[5] Johnson, J., Püschel, M.: In search of the optimal walsh-hadamard transform. IEEE (2000) 1

[6] Smola, A., Schölkopf, B., Müller, K.R.: General cost functions for support vector regression. Ninth Australian Conference on Neural Networks (1988) 2

[7] Box, G.E.P., Muller, M.E.: A note on the generation of random normal deviates. The Annals of Mathematical Statistics (1958) 2

[8] Yang, Z., Smola, A., Song, L., Wilson, A.G.: À la carte - learning fast kernels. AISTATS (2014) 2