Learning kernels that adapt to GPU

Siyuan Ma and Mikhail Belkin
Department of Computer Science and Engineering
The Ohio State University
{masi, mbelkin}@cse.ohio-state.edu

Abstract

In this work we develop a framework for kernel machines that are efficient, accurate and are adaptive to modern parallel hardware, such as GPU. Our main innovation is in constructing kernel machines that output solutions mathematically equivalent to those obtained using standard kernels, yet capable of fully utilizing the available computing power of a parallel computational resource. Such utilization is key to strong performance as much of the computational resource capability is wasted by the standard iterative methods.

Our approach is based on the idea of interpolation, using the significant empirical evidence that methods achieving near-zero training error show excellent test results. In this work we show how the mathematical and conceptual simplicity of optimization in the interpolation regime can be harnessed to design kernels and automatically choose parameters adaptive to computational resources.

The resulting algorithm, which we call *EigenPro 2.0*, is accurate, principled and very fast. For example, using a single Titan XP GPU, training on ImageNet with 1.3×10^6 data points and 1000 labels takes under an hour, while smaller datasets, such as MNIST, take seconds. As the parameters are chosen analytically, based on the theoretical bounds, little tuning beyond selecting the kernel and kernel parameter is needed, further facilitating the practical use of these methods. See arxiv.org/abs/1806.06144 for the full version of this paper.

1 Introduction

Kernel machines are a powerful class of methods for classification and regression. Given the training data $\{(\boldsymbol{x}_i, y_i), i = 1, \ldots, n\} \in \mathbb{R}^d \times \mathbb{R}$, and a positive definite kernel $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$, kernel machines construct functions of the form $f(\boldsymbol{x}) = \sum_i \alpha_i k(\boldsymbol{x}, \boldsymbol{x}_i)$. These methods are theoretically attractive, show excellent performance on smaller datasets, and are known to be universal learners, i.e., capable of approximating any function from data. However, making kernel machines fast and scalable to large data has been a challenging problem. Recent large scale efforts typically involved significant parallel computational resources, such as multiple (sometimes thousands) AWS vCPU's [TRVR16, ACW16] or super-computer nodes [HAS+14]. Very recently, FALKON [RCR17] and EigenPro [MB17] showed strong classification results on large datasets with much lower computational requirements, a few hours on a single GPU.

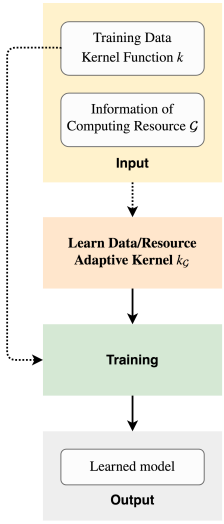
The goal of this paper is to go beyond those algorithms by designing kernel machines that can be trained very quickly on both small and large data, easily scale to millions of data points using standard modern hardware, and consistently show excellent classification performance. We aim to make nearly all aspects of parameter selection automatic, making these methods easy and convenient to use in practice and appropriate for "interactive" exploratory machine learning.

The main problem and our contribution. The main problem addressed in this paper is to minimize the training time for a kernel machine, given access to a parallel computational resource $\mathcal{G}$. Our main contribution is that given a standard kernel, we are able to learn a new data and computational resource dependent kernel to minimize the resource time required for training without changing the

Preprint. Work in progress.

mathematical solution for the original kernel. Our model for a *computational resource* $\mathcal{G}$ is based on a modern graphics processing unit (GPU), a device that allows for very efficient, highly parallel[1] matrix multiplication.

The outline of our approach is shown in the diagram on the right. We now outline the key ingredients.



The interpolation framework. In recent years we have seen that inference methods, notably neural networks, that interpolate or nearly interpolate the training data generalize very well to test data [ZBH+16]. It has been observed in [BMM18] that minimum norm kernel interpolants, i.e., functions of the forms $f(\boldsymbol{x}) = \sum_i \alpha_i k(\boldsymbol{x}, \boldsymbol{x}_i)$, such that $f(\boldsymbol{x}_i) = y_i$, achieve optimal or near optimal generalization performance. While the mathematical foundations of why interpolation produces good test results are not yet fully understood, the simplicity of the framework can be used to accelerate and scale the training of classical kernel methods, while improving their test accuracy. Indeed, constructing these interpolating functions is conceptually and mathematically simple, requiring approximately solving a single system of linear equations with a unique solution, same for both regression and classification. Significant computational savings and, when necessary, regularization [YRC07] are provided by early stopping, i.e., stopping iterations before numerical convergence, once successive iterations fail to improve validation error.

Adaptivity to data and computational resource: choosing optimal batch size and step size for SGD. We will train kernel methods using Stochastic Gradient Descent (SGD), a method which is well-suited to modern GPU's and has shown impressive success in training neural networks. Importantly, in the interpolation framework, dependence of convergence on the batch size and the step size can be derived analytically, allowing for full analysis and automatic parameter selection.

We first note that in the parallel model each iteration of SGD (essentially a matrix multiplication) takes the same time for any mini-batch size up to $m_{\mathcal{G}}^{max}$, defined as the mini-batch size where the parallel capacity of the resource $\mathcal{G}$ is fully utilized. It is shown in [MBB17] that in the interpolation framework convergence per iteration (using optimal step size) improves nearly linearly as a function of the mini-batch size m up to a certain *critical size* $m^*(k)$ and rapidly saturates after that. The quantity $m^*(k)$ is related to the spectrum of the kernel. For kernels used in practice it is typically quite small, less than 10, due to their rapid eigenvalue decay. Yet, depending on the number of data points, features and labels, a modern GPU can handle mini-batches of size 1000 or larger. This disparity presents an opportunity for major improvements in the efficiency of kernel methods. In this paper we show how to construct data and resource adaptive kernel $k_{\mathcal{G}}$, by modifying the spectrum of the kernel by using EigenPro algorithm [MB17]. The resulting iterative method with the new kernel has similar or better convergence per iteration than the original kernel k for small mini-batch size. However its convergence improves linearly to much larger mini-batch sizes, matching $m_{\mathcal{G}}^{max}$, the maximum that can be utilized by the resource $\mathcal{G}$. Importantly, SGD for either kernel converge to the same interpolated solution.

Thus, we aim to modify the kernel by constructing a kernel $k_{\mathcal{G}}$, such that $m^*(k_{\mathcal{G}}) = m_{\mathcal{G}}^{max}$ without changing the optimal (interpolating) solution. This is shown schematically in Figure 1. We see that for small mini-batch size convergence of these two kernels k and $k_{\mathcal{G}}$ is similar. However, values of $m > m^*(k)$ do not help the convergence of the original kernel k, while convergence of $k_{\mathcal{G}}$ keep improving up to $m = m_{\mathcal{G}}^{max}$, where the resource utilization is saturated. For empirical results on real datasets, parallel to the schematic shown above, see Figure 2 in Section 4.

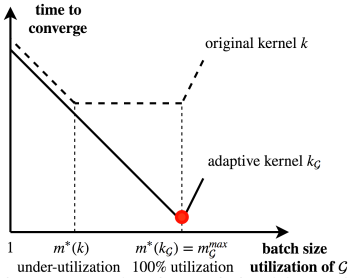


Figure 1: Adaptive/original kernel

We construct and implement these kernels (see github.com/EigenPro/EigenPro2 for the code), and show how to analytically choose parameters, including the batch size and the step size. As a secondary contribution of this work we develop an improved version of EigenPro [MB17] significantly reducing the memory requirements and making the computational overhead over the standard SGD negligible.

[1]For example, there are 3840 CUDA cores in Nvidia GTX Titan Xp (Pascal).

Comparison to related work. In recent years there has been significant progress on scaling and accelerating kernel methods including [TBRS13, HAS+14, LML+14, TRVR16, ACW16, MGL+17]. Most of these methods are able to scale to large data sets by utilizing major computational resources such as supercomputers or multiple (sometimes hundreds or thousands) AWS vCPU's. Two recent methods which allow for high efficiency kernel training with a single CPU or GPU is EigenPro [MB17] (used a as basis for the adaptive kernels in this paper) and FALKON [RCR17]. The method developed in this paper is significantly faster than either of them, while achieving similar or better test set accuracy. Additionally, it is easier to use as much of the parameter selection is done automatically. Mini-batch SGD (used in our algorithm) has been the dominant technique in training deep models. There has been significant empirical evidence [Kri14, YGG17, SKL17] showing that linearly scaling the step size with the mini-batch size up to a certain value leads to improved convergence. This phenomenon has been utilized to scale deep learning in distributed systems by adopting large mini-batch sizes [GDG+17]. The advantage of our setting is that the optimal batch and step sizes can be analyzed and expressed analytically. Moreover, these formulas contain variables which can be explicitly computed and directly used for parameter selection in our algorithms. Going beyond batch size and step size selection, the theoretical interpolation framework allows us to construct new adaptive kernels, such that the mini-batch size required for optimal convergence matches the capacity of the computational resource.

The paper is structured as follows: In Section 3, we present our main algorithm to learn a kernel to fully utilize a given computational resource. We then provide comparisons to state-of-the-art kernel methods on several large datasets in Section 4. We further discuss exploratory machine learning in the context of our method.

2 Setup

We start by briefly discussing the basic setting and kernel methods used in this paper.

Kernel interpolation. We are given n labeled training points $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n) \in \mathbb{R}^d \times \mathbb{R}$. We consider a Reproducing Kernel Hilbert Space (RKHS) $\mathscr{H}$ [Aro50] corresponding to a positive definite kernel function $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$. There is a unique (minimum norm) interpolated solution in $\mathscr{H}$ of the form $f^*(\cdot) = \sum_{i=1}^n \alpha_i^* k(\boldsymbol{x}_i, \cdot)$, where $(\alpha_1^*, \ldots, \alpha_n^*)^T = K^{-1}(y_1, \ldots, y_n)^T$. Here K denotes an $n \times n$ kernel matrix, $K_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$. It is easy to check that $\forall_i f^*(x_i) = y_i$.

Square loss. While the interpolated solution f^* in $\mathscr{H}$ does not depend on any loss function, it is the unique minimizer in $\mathscr{H}$ for the empirical square loss $L(f) \triangleq \frac{1}{n} \sum_{i=1}^n (f(\boldsymbol{x}_i) - y_i)^2$.

Gradient descent. It can be shown that gradient descent iteration for the empirical squared loss in RKHS $\mathscr{H}$ is given by $f \leftarrow f - \eta \cdot \frac{2}{n} \sum_{i=1}^n (f(\boldsymbol{x}_i) - y_i) k(\boldsymbol{x}_i, \cdot)$.

Mini-batch SGD. Instead of calculating the gradient with n training points, each SGD iteration updates the solution f using m subsamples $(\boldsymbol{x}_{t_1}, y_{t_1}), \ldots, (\boldsymbol{x}_{t_m}, y_{t_m})$, $f \leftarrow f - \eta \cdot \frac{2}{m} \{\sum_{i=1}^m (f(\boldsymbol{x}_{t_i}) - y_{t_i}) k(\boldsymbol{x}_{t_i}, \cdot)\}$. It is equivalent to randomized coordinate descent [LL10] for $K\boldsymbol{\alpha} = \boldsymbol{y}$ on m coordinates of $\boldsymbol{\alpha}$, $\alpha_{t_i} \leftarrow \alpha_{t_i} - \eta \cdot \frac{2}{m} \{f(\boldsymbol{x}_{t_i}) - y_{t_i}\}$ for $i = 1, \ldots, m$.

Critical mini-batch size as effective parallelism. Theorem 4 in [MBB17] shows that for mini-batch iteration with kernel k there is a data-dependent batch size $m^*(k)$ such that (a) Convergence per iteration improves linearly with increasing batch size$m$ for $m \leq m^*(k)$ (using optimal constant step size); (b) Training with any batch size $m > m^*(k)$ leads to the same convergence per iteration as training with $m^*(k)$ up to a small constant factor. We can calculate $m^*(k)$ explicitly using kernel matrix K (depending on the data), $m^*(k) = \frac{\beta(K)}{\lambda_1(K)}$ where $\beta(K) \triangleq \max_{i=1,\ldots,n} k(\boldsymbol{x}_i, \boldsymbol{x}_i)$. For any shift invariant kernel k, after normalization, we have $\beta(K) = \max_{i=1}^n k(\boldsymbol{x}_i, \boldsymbol{x}_i) \equiv 1$.

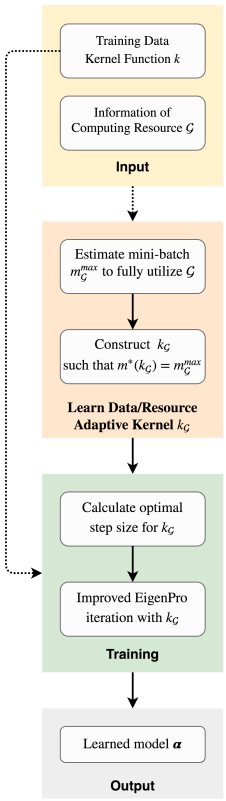
EigenPro iteration [MB17]. To achieve faster convergence, EigenPro iteration performs spectral modification on the kernel operator $\mathscr{K}(f) \triangleq \frac{2}{n} \sum_{i=1}^n \langle k(\boldsymbol{x}_i, \cdot), f \rangle_{\mathscr{H}} k(\boldsymbol{x}_i, \cdot)$ using operator, $\mathscr{P}(f) \triangleq f - \sum_{i=1}^q (1 - \frac{\lambda_q}{\lambda_i}) \langle e_i, f \rangle_{\mathscr{H}} e_i$ where $\lambda_1 \geq \cdots \geq \lambda_n$ are ordered eigenvalues of $\mathscr{K}$ and e_i is its eigenfunction corresponding to λ_i. The iteration uses $\mathscr{P}$ to rescale a (stochastic) gradient in $\mathscr{H}$, $f \leftarrow f - \eta \cdot \mathscr{P} \left\{ \frac{2}{m} \sum_{i=1}^m (f(\boldsymbol{x}_{t_i}) - y_{t_i}) k(\boldsymbol{x}_{t_i}, \cdot) \right\}$.

Abstraction for parallel computational resources. To construct a resource adaptive kernel, we consider the following abstraction for given computational resource $\mathcal{G}$. $\boldsymbol{C}_{\mathcal{G}}$: Parallel capacity of $\mathcal{G}$, i.e., the number of parallel operations that is required to fully utilize the computing capacity of $\mathcal{G}$. $\boldsymbol{S}_{\mathcal{G}}$:

Internal resource memory of $\mathcal{G}$. To fully utilize $\mathcal{G}$, one SGD/EigenPro iteration must execute at least $C_{\mathcal{G}}$ operations using less than $S_{\mathcal{G}}$ memory. In this paper, we primarily adapt kernel to GPU devices. For a GPU $\mathcal{G}$, $S_{\mathcal{G}}$ equals the size of its dedicated memory and $C_{\mathcal{G}}$ is proportional to the number of the computing cores (e.g., 3840 CUDA cores in Titan Xp). Note for computational resources like cluster and supercomputer, we need to take into account additional factors such as network bandwidth.

3 Main Algorithm

Our main algorithm aims to reduce the training time by constructing a data/resource adaptive kernel for any given kernel function k to fully utilize a computational resource $\mathcal{G}$. Its detailed workflow is presented on the right. Specifically, we use the following steps. Step 1: Calculate the resource-dependent mini-batch size $m_{\mathcal{G}}^{max}$ to fully utilize resource $\mathcal{G}$. Step 2: Identify the parameters and construct a new kernel $k_{\mathcal{G}}$ such that $m^*(k_{\mathcal{G}}) = m_{\mathcal{G}}^{max}$. Step 3: Select optimal step size and train using improved EigenPro (see the full paper). Note that due to properties of EigenPro iteration, training with this adaptive kernel converges to the same solution as the original kernel.



EigenPro 2.0

To calculate $m_{\mathcal{G}}^{max}$ for 100% resource utilization, we first estimate the operation parallelism and memory usage of one EigenPro iteration. The improved version of EigenPro iteration (see the full paper) makes computation and memory overhead over the standard SGD negligible. Thus we assume that EigenPro has the same complexity as the standard SGD per iteration.

Cost of one EigenPro iteration with batch size m. We consider training data $(\boldsymbol{x}_i, \boldsymbol{y}_i) \in \mathbb{R}^d \times \mathbb{R}^l, i = 1, \ldots, n$. Here each feature vector $\boldsymbol{x}$ is d dimensional, and each label $\boldsymbol{y}$ is l dimensional. **Computational cost:** It takes $(d + l) \cdot m \cdot n$ operations to perform one SGD iteration on m points. These computations reduce to matrix multiplication and can be done in parallel. **Space usage:** It takes $d \cdot n$ memory to store the training data (as kernel centers) and $l \cdot n$ memory to maintain the model weight. We can now calculate $m_{\mathcal{G}}^{max}$ for the parallel computational resource $\mathcal{G}$ with parameters $C_{\mathcal{G}}, S_{\mathcal{G}}$ and introduced in Section 2.

Step 1: Determining batch size $m_{\mathcal{G}}^{max}$ for 100% resource utilization. We first define two mini-batch notations. $\boldsymbol{m}_{C_{\mathcal{G}}}$: batch size for fully utilizing parallelism in $\mathcal{G}$ such that $(d + l) \cdot m_{C_{\mathcal{G}}} \cdot n \approx C_{\mathcal{G}}$. $\boldsymbol{m}_{S_{\mathcal{G}}}$: batch size for maximum memory usage of $\mathcal{G}$ such that $(d + l + m_{S_{\mathcal{G}}}) \cdot n \approx S_{\mathcal{G}}$. To best utilize $\mathcal{G}$ without exceeding its memory, we set $m_{\mathcal{G}}^{max} = \min\{m_{C_{\mathcal{G}}}, m_{S_{\mathcal{G}}}\}$. Note that in practice, it is more important to fully utilize the memory so that $m_{\mathcal{G}}^{max} \lesssim m_{S_{\mathcal{G}}}$.

Step 2: Learning the kernel $k_{\mathcal{G}}$ given $m_{\mathcal{G}}^{max}$. Next, we show how to construct $k_{\mathcal{G}} = k_{\mathscr{P}_q}$ using EigenPro iteration such that $m^*(k_{\mathcal{G}}) = m_{\mathcal{G}}^{max}$. The corresponding q is defined as $q \triangleq \max\ \{i \in \mathbb{N},\ \text{s.t.}\ m^*(k_{\mathscr{P}_i}) \leq m_{\mathcal{G}}^{max}\}$.

Step 3: Training with adaptive kernel $k_{\mathcal{G}} = k_{\mathscr{P}_q}$. We use the learned kernel $k_{\mathcal{G}}$ with improved EigenPro. Its optimization parameters (batch and step size) are calculated by $m = m_{\mathcal{G}}^{max}, \eta = \frac{m_{\mathcal{G}}^{max}}{\beta(K_{\mathcal{G}})}$

Claim (Acceleration). Using the adaptive kernel $k_{\mathcal{G}}$ decreases the resource time required for training (assuming an idealized model of the GPU and workload) over the original kernel k by a factor of acceleration of $k_{\mathcal{G}}$ over $k = \frac{\beta(K)}{\beta(K_{\mathcal{G}})} \cdot \frac{m_{\mathcal{G}}^{max}}{m^*(k)}$ See the supplementary material for the derivation and a discussion. We note that empirically, $\beta(K_{\mathcal{G}}) \approx \beta(K)$, while $\frac{m_{\mathcal{G}}^{max}}{m^*(k)}$ is between 50 and 500, which is in line with the acceleration observed in practice.

Improved EigenPro Iteration using Nyström Extension. we present an improvement for the EigenPro iteration originally proposed in [MB17]. We significantly reduce the memory overhead of EigenPro over standard SGD and nearly eliminate computational overhead per iteration. The improvement is based on an efficient representation of the EigenPro preconditioner using Nyström extension. See the full paper for the details of the algorithm.

4 Experimental Evaluation

See description for computing resource, datasets, and model in the full paper.

4.1 Comparison to state-of-the-art kernel methods

In the table below, we compare our method to the state-of-the-art kernel methods on several large datasets. For all datasets, our method is significantly faster than other methods while still achieving better or similar results. Moreover, our method uses only a single GPU while many state-of-the-art kernel methods use much less accessible computing resources.

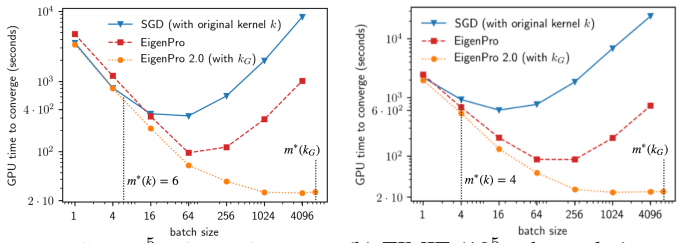
Dataset	Size	EigenPro 2.0 (use 1 GTX Titan Xp)		Results of Other Methods		
		error	GPU time	resource time	error	reference
MNIST	6.7×10^6	0.72%	**19 m**	4.8 h on 1 GTX Titan X	0.70%	EigenPro [MB17]
				1.1 h on 1344 AWS vCPUs	0.72%	PCG [ACW16]
				less than 37.5 hours on 1 Tesla K20m	0.85%	[LML+14]
ImageNet†	1.3×10^6	20.6%	**40 m**	-	19.9%	Inception-ResNet-v2 [SIVA17]
				4 h on 1 Tesla K40c	20.7%	FALKON [RCR17]
TIMIT‡	$1.1 \cdot 10^6$ / $2 \cdot 10^6$	31.7%	24 m (3 epochs)	3.2 h on 1 GTX Titan X	31.7%	EigenPro [MB17]
				1.5 h on 1 Tesla K40c	32.3%	FALKON [RCR17]
				512 IBM Blue Gene/Q cores	33.5%	Ensemble [HAS+14]
		32.1%	**8 m** (1 epoch)	7.5 h on 1024 AWS vCPUs	33.5%	BCD [TRVR16]
				multiple AWS g2.2xlarge instances	32.4%	DNN [MGL+17]
				multiple AWS g2.2xlarge instances	30.9%	SparseKernel [MGL+17] (use learned features)
SUSY	$4 \cdot 10^6$	19.7%	**58 s**	6 m on 1 GTX Titan X	19.8%	EigenPro [MB17]
				4 m on 1 Tesla K40c	19.6%	FALKON [RCR17]
				36 m on IBM POWER8	$\approx 20\%$	Hierarchical [CAS16]

† Our method uses the convolutional features from Inception-ResNet-v2 and Falkon uses the convolutional features from Inception-v4. Both neural network models are presented in [SIVA17] and show nearly identical performance.

‡ There are two sampling rates for TIMIT, which result in two training sets of different sizes.

4.2 Convergence comparison to SGD and EigenPro

In Figure 2, we train three kernel machines with EigenPro 2.0, standard SGD and EigenPro [MB17] for various batch sizes. The step sizes for SGD and EigenPro are tuned for best performance. The step size for EigenPro 2.0 is computed automatically according to Section 3.



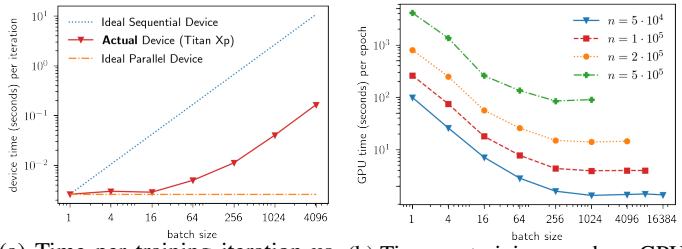
(a) MNIST (10^5 subsamples), stop when target train mse $< 1 \cdot 10^{-4}$ (b) TIMIT (10^5 subsamples), stop when target train mse $< 2 \cdot 10^{-4}$

Figure 2: Time to converge with optimal step sizes

Consistent with the schematic Figure 1 in the introduction, the original kernel k has a critical batch size $m^*(k)$ of size 4 and 6 respectively, which is too small to fully utilize the parallel computing capacity of the GPU device. In contrast, the EigenPro 2.0 kernel k_G has a much larger critical batch size $m^*(k_G) \approx 6500$, which leads to maximum GPU utilization. We see that EigenPro 2.0 significantly outperforms original EigenPro due to better resource utilization and parameter selection, as well as lower overhead (see detailed comparison in the full paper).

4.3 Batch size and GPU utilization

The number of operation required for one iteration of SGD is linear in the batch size. Thus we expect that time required per iteration for a pure sequential machine would scale linearly with batch size. On the other hand an ideal parallel device with no overhead requires the same amount of time to process any mini-batch. In Figure 3a, we show how the training time per iteration for actual GPU depends on the batch size. We see that for small batch sizes time per iteration is nearly constant, like that of an ideal parallel device, and start to increase for larger batches.



(a) Time per training iteration using different batch sizes on actual and ideal devices (TIMIT, $n = 10^5$, $d = 440$)

(b) Time per training epoch on GPU with different sizes of train set (n, which is also the model size) and batch that fit into the GPU memory

Figure 3: Time per iteration / epoch of training using different batch sizes

Note that in addition to time per iteration we need to consider the overhead associated to each iteration. Larger batch sizes incur less overhead per epoch. This phenomenon is known in the systems literature as Amdahl's law [Rod85]. In Figure 3b we show GPU time *per epoch* for different model (training set) size (n). We see consistent speed-ups by increasing mini-batch size across model sizes up to maximum GPU utilization.

4.4 "Interactive" training for exploratory machine learning

Most practical tasks of machine learning require multiple training runs for parameter and feature selection, evaluating appropriateness of data or features to a given task, and various other exploratory purposes. While using hours, days or even months of machine time may be necessary to improve on the state of the art in large-scale certain problems, it is too time-consuming and expensive for most data analysis work. Thus, it is very desirable to train classifiers in close to real time. One of the advantages of our approach is the combination of its speed on small and medium datasets using standard hardware together with the automatic optimization parameter selection. We demonstrate this on several smaller datasets ($10^4 \sim 10^5$ points) using a Titan Xp GPU (see Table 1). We see that in every case training takes no more than 15 seconds, making multiple runs for parameter and feature selection easily feasible. For comparison, we also provide timings for LibSVM, a popular and widely used kernel library [CL11] and ThunderSVM [WSL+18], a fast GPU implementation for LibSVM. We show the results for LibSVM and ThunderSVM using the same kernel with the same parameter. We stopped iteration of our method when the accuracy on test exceeded that of LibSVM, which our method was able to achieve on every dataset. While not intended as a comprehensive evaluation, the benefits of our method for typical data analysis tasks are evident. Fast training along with the "worry-free" optimization create an "interactive/responsive" environment for using kernel methods in machine learning. Furthermore, the choice of kernel (e.g., Laplacian or Gaussian) and its single bandwidth parameter is usually far simpler than the multiple parameters involved in the selection of architecture in neural networks.

Dataset	Size	Dimension	Our method (GPU)	ThunderSVM (GPU)	LibSVM (CPU)
TIMIT	$1 \cdot 10^5$	440	**15 s**	480 s	1.6 h
SVHN	$7 \cdot 10^4$	1024	**13 s**	142 s	3.8 h
MNIST	$6 \cdot 10^4$	784	**6 s**	31 s	9 m
CIFAR-10	$5 \cdot 10^4$	1024	**8 s**	121 s	3.4 h

Table 1: Training time

5 Conclusion and Future Directions

The main contribution of this paper is to develop kernel methods for machine learning capable of minimizing the training time given access to a parallel computational resource. We have developed practical algorithms that are very fast for smaller data and scale easily to several million data points on a modern GPU. It is likely that more effective memory management together with the latest generation of hardware would allow scaling up to 10^7 data points with reasonable training time. Going beyond that to 10^8 or more data points using multi-GPU or other parallel computing setups is the next natural step.

References

[ACW16] H. Avron, K. Clarkson, and D. Woodruff. Faster kernel ridge regression using sketching and preconditioning. *arXiv preprint arXiv:1611.03220*, 2016.

[Aro50] Nachman Aronszajn. Theory of reproducing kernels. *Transactions of the American mathematical society*, 68(3):337–404, 1950.

[BMM18] Mikhail Belkin, Siyuan Ma, and Soumik Mandal. To understand deep learning we need to understand kernel learning. *arXiv preprint arXiv:1802.01396*, 2018.

[CAS16] Jie Chen, Haim Avron, and Vikas Sindhwani. Hierarchically compositional kernels for scalable nonparametric learning. *arXiv preprint arXiv:1608.00860*, 2016.

[CL11] Chih-Chung Chang and Chih-Jen Lin. Libsvm: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2(3):27, 2011.

[GDG+17] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[HAS+14] Po-Sen Huang, Haim Avron, Tara N Sainath, Vikas Sindhwani, and Bhuvana Ramabhadran. Kernel methods match deep neural networks on timit. In *ICASSP*, pages 205–209. IEEE, 2014.

[Kri14] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[LL10] Dennis Leventhal and Adrian S Lewis. Randomized methods for linear constraints: convergence rates and conditioning. *Mathematics of Operations Research*, 35(3):641–654, 2010.

[LML+14] Zhiyun Lu, Avner May, Kuan Liu, Alireza Bagheri Garakani, Dong Guo, Aurélien Bellet, Linxi Fan, Michael Collins, Brian Kingsbury, Michael Picheny, and Fei Sha. How to scale up kernel methods to be as good as deep neural nets. *arXiv preprint arXiv:1411.4000*, 2014.

[MB17] Siyuan Ma and Mikhail Belkin. Diving into the shallows: a computational perspective on large-scale shallow learning. In *Advances in Neural Information Processing Systems*, pages 3781–3790, 2017.

[MBB17] Siyuan Ma, Raef Bassily, and Mikhail Belkin. The power of interpolation: Understanding the effectiveness of sgd in modern over-parametrized learning. *arXiv preprint arXiv:1712.06559*, 2017.

[MGL+17] Avner May, Alireza Bagheri Garakani, Zhiyun Lu, Dong Guo, Kuan Liu, Aurélien Bellet, Linxi Fan, Michael Collins, Daniel Hsu, Brian Kingsbury, et al. Kernel approximation methods for speech recognition. *arXiv preprint arXiv:1701.03577*, 2017.

[RCR17] Alessandro Rudi, Luigi Carratino, and Lorenzo Rosasco. Falkon: An optimal large scale kernel method. In *Advances in Neural Information Processing Systems*, pages 3891–3901, 2017.

[Rod85] David P Rodgers. Improvements in multiprocessor system design. In *SIGARCH*, 1985.

[SIVA17] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI*, volume 4, page 12, 2017.

[SKL17] Samuel L Smith, Pieter-Jan Kindermans, and Quoc V Le. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*, 2017.

[TBRS13] Martin Takác, Avleen Singh Bijral, Peter Richtárik, and Nati Srebro. Mini-batch primal and dual methods for SVMs. In *ICML (3)*, pages 1022–1030, 2013.

[TRVR16] S. Tu, R. Roelofs, S. Venkataraman, and B. Recht. Large scale kernel learning using block coordinate descent. *arXiv preprint arXiv:1602.05310*, 2016.

[WSL+18] Zeyi Wen, Jiashuai Shi, Qinbin Li, Bingsheng He, and Jian Chen. Thundersvm: a fast svm library on gpus and cpus. *The Journal of Machine Learning Research (JMLR)*, 19(1):797–801, 2018.

[YGG17] Yang You, Igor Gitman, and Boris Ginsburg. Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888*, 2017.

[YRC07] Y. Yao, L. Rosasco, and A. Caponnetto. On early stopping in gradient descent learning. *Constructive Approximation*, 26(2):289–315, 2007.

[ZBH+16] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals. Understanding deep learning requires rethinking generalization. *arXiv preprint arXiv:1611.03530*, 2016.